

Algoritmos de Pesquisa em Strings

R. Rossetti, A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, G. Leão
FEUP, MIEIC, CAL

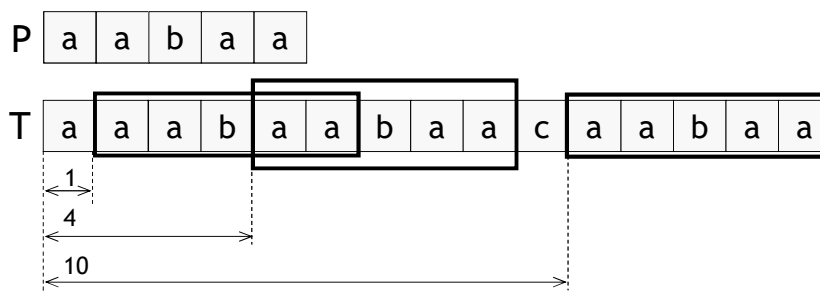
Índice

- Pesquisa exata (*string matching*)
- Pesquisa aproximada (*approximate string matching*)
- Outros problemas de pesquisa

Pesquisa exata (*string matching*)

Problema

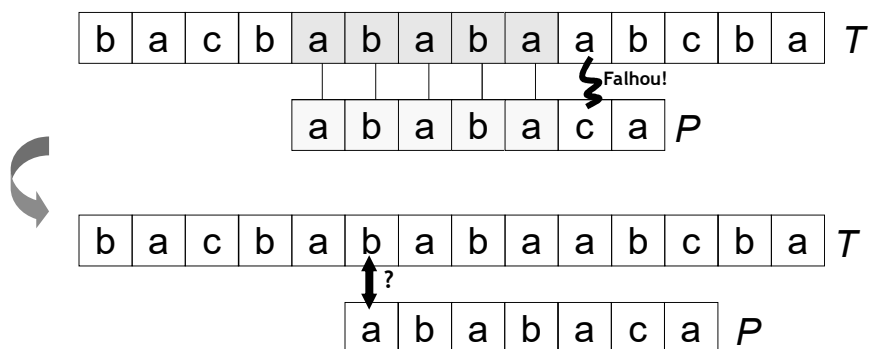
- Encontrar todas as ocorrências de um padrão P num texto T
 - P e T são cadeias de caracteres
 - Ocorrências são definidas pela deslocação em relação ao início do texto
 - Ocorrências podem ser sobrepostas



Algoritmos

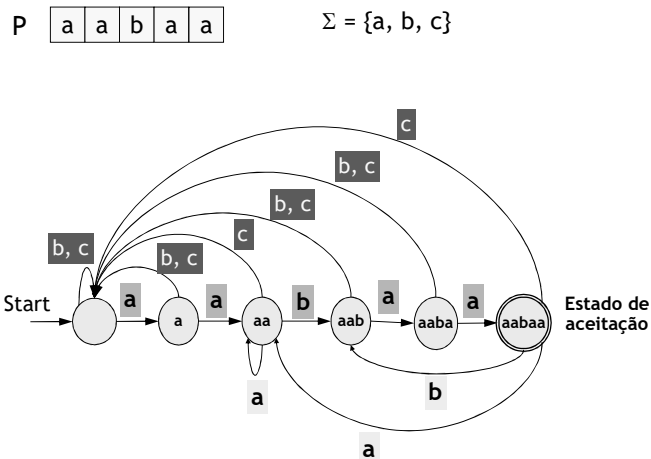
- Algoritmo *naïve*
 - Para cada deslocamento possível, compara desde o início do padrão
 - Ineficiente se o padrão for comprido: $O(|P| \cdot |T|)$
- Algoritmo baseado em autómato finito
 - Pré-processamento: gerar autómato finito correspondente ao padrão
 - Permite depois analisar o texto em tempo linear $O(|T|)$, pois cada carácter só precisa de ser processado uma vez
 - Mas tempo e espaço requerido pelo pré-processamento pode ser elevado: $O(|P| \cdot |\Sigma|)$, em que $|\Sigma|$ é o tamanho do alfabeto
- Algoritmo de Knuth-Morris-Pratt
 - Efetua um pré-processamento do padrão em tempo $O(|P|)$, sem chegar a gerar explicitamente um autómato, seguido de processamento do texto em $O(|T|)$, dando total $O(|T| + |P|)$

Algoritmo *naïve*

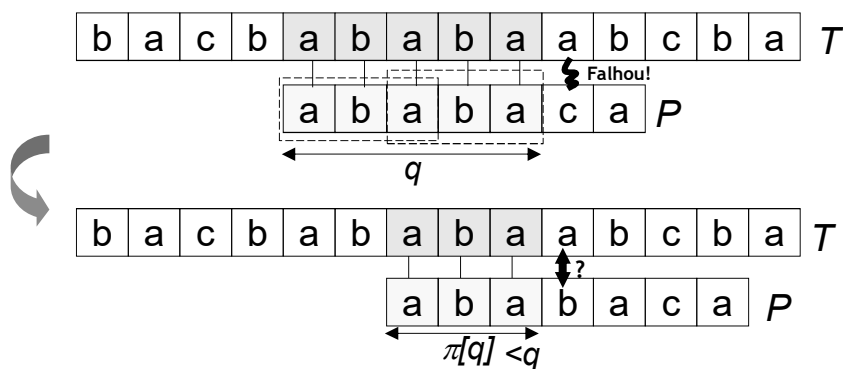


Desloca-se o padrão uma casa para a direita e recomeça-se a comparação do início do padrão! Ineficiente: $O(|P| \cdot |T|)$

Autómato finito correspondente ao padrão



Algoritmo de Knuth-Morris-Pratt



Desloca-se o padrão para a direita de uma forma que permite continuar a comparação na mesma posição do texto! Evita comparações inúteis!

Deslocamento é determinado por uma função $\pi[q]$ calculada numa fase de pré-processamento do padrão!

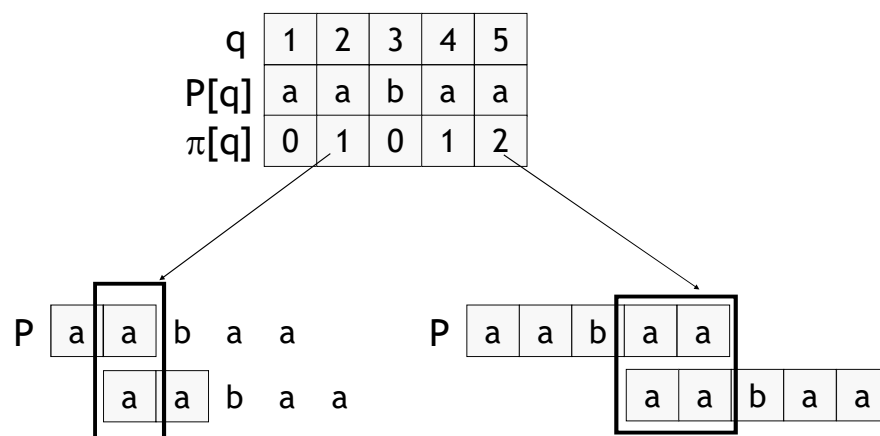
Pré-processamento do padrão

- Compara-se o padrão com deslocações do mesmo, para determinar a **função prefixo**

$$\pi[q] = \max \{k: 0 \leq k < q \text{ e } P[1..k] = P[(q-k+1)..q] \}$$

- $q = 1, \dots, |P|$
- $P[i..j]$ - substring entre índices i e j
- Índices a começar em 1
- $\pi[q]$ é o comprimento do maior prefixo de P que é um sufixo próprio do prefixo de P de comprimento q

Pré-processamento do padrão



Pseudo-código

KMP-MATCHER(T, P)

```

1   $n \leftarrow \text{length}[T]$ 
2   $m \leftarrow \text{length}[P]$ 
3   $\pi \leftarrow \text{COMPUTE-PREFIX-FUNCTION}(P)$ 
4   $q \leftarrow 0$  ▷ Number of characters matched.
5  for  $i \leftarrow 1$  to  $n$  ▷ Scan the text from left to right.
6      do while  $q > 0$  and  $P[q + 1] \neq T[i]$ 
7          do  $q \leftarrow \pi[q]$  ▷ Next character does not match.
8      if  $P[q + 1] = T[i]$ 
9          then  $q \leftarrow q + 1$  ▷ Next character matches.
10     if  $q = m$  ▷ Is all of  $P$  matched?
11         then print "Pattern occurs with shift"  $i - m$ 
12          $q \leftarrow \pi[q]$  ▷ Look for the next match.

```

Pseudo-código

COMPUTE-PREFIX-FUNCTION(P)

```

1   $m \leftarrow \text{length}[P]$ 
2   $\pi[1] \leftarrow 0$ 
3   $k \leftarrow 0$ 
4  for  $q \leftarrow 2$  to  $m$ 
5      do while  $k > 0$  and  $P[k + 1] \neq P[q]$ 
6          do  $k \leftarrow \pi[k]$ 
7      if  $P[k + 1] = P[q]$ 
8          then  $k \leftarrow k + 1$ 
9       $\pi[q] \leftarrow k$ 
10 return  $\pi$ 

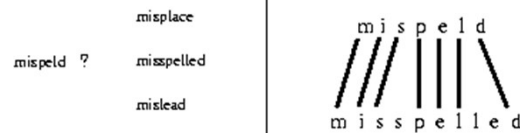
```

* Eficiência do algoritmo Knuth-Morris-Pratt

- KMP-MATCHER (sem incluir COMPUTE-PREFIX-FUNCTION)
 - Eficiência depende do nº de iterações do ciclo “while” interno
 - Dado que $0 \leq \pi[q] < q$, cada vez que a instrução 7 é executada, o valor de q é decrementado de pelo menos 1, sem nunca chegar a ser negativo
 - Dado que o valor de q começa em 0 e só é incrementado no máximo n vezes (+1 de cada vez, na linha 9), o nº máximo de vezes que pode ser decrementado (nas linhas 7 e 12) é também n
 - ⇒ Nº máximo de iterações do ciclo “while” interno (no conjunto de todas as iterações do ciclo “for” externo) é n
 - ⇒ Tempo de execução da rotina é $O(n)$, i.e., $O(|T|)$
- COMPUTE-PREFIX-FUNCTION
 - Seguindo o mesmo raciocínio, tempo de execução é $O(m)$, i.e., $O(|P|)$
- Total: $O(n+m)$, isto é, $O(|T| + |P|)$

Pesquisa aproximada (*approximate string matching*)

Problema



INPUT

OUTPUT

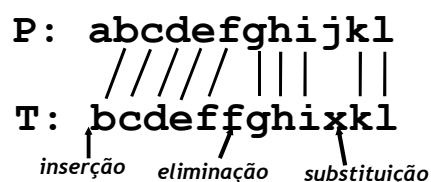
Input description: A text string T and a pattern string P . An edit cost bound k .

Problem description: Can we transform T to P using at most k insertions, deletions, and substitutions?

(Ou: qual é o grau de semelhança entre P e T ?)

Distância de edição entre duas strings

- A *distância de edição* entre P (*pattern string*) e T (*text string*) é o menor número de alterações necessárias para transformar T em P , em que as alterações podem ser:
 - substituir um carácter por outro
 - inserir um carácter
 - eliminar um carácter



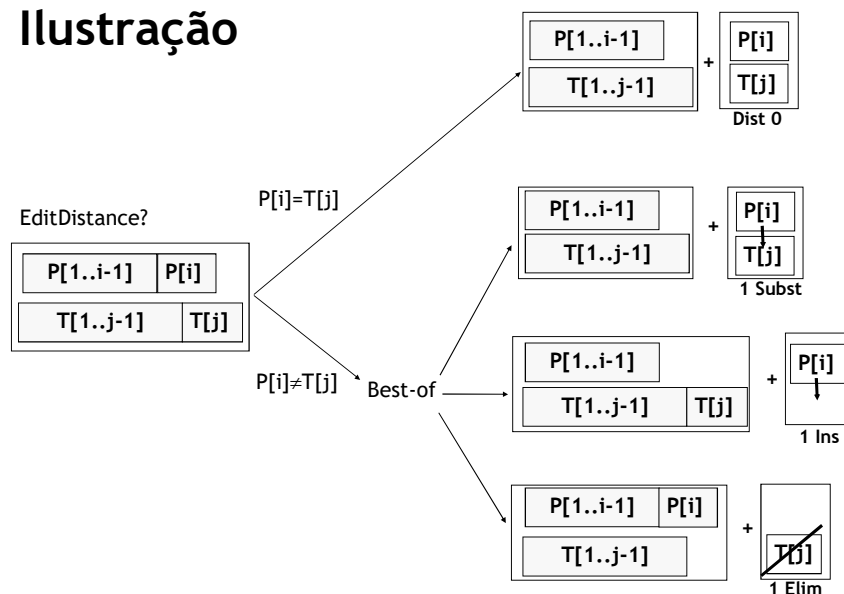
EditDistance(P,T)=3

inserção eliminação substituição

Formulação recursiva

- $D[i, j] = \text{EditDistance}(P[1..i], T[1..j]), 0 \leq i \leq |P|, 0 \leq j \leq |T|$
- Condições fronteira:
 - $D[0, j] = j, D[i, 0] = i$ (porquê?)
- Caso recursivo ($i > 0$ e $j > 0$):
 - Se $P[i] = T[j]$, então $D[i, j] = D[i-1, j-1]$
 - Senão, escolhe-se a operação de edição que sai mais barata; isto é, $D[i, j]$ é o mínimo de:
 - $1 + D[i-1, j-1]$ (substituição de $T[j]$ por $P[i]$)
 - $1 + D[i-1, j]$ (inserção de $P[i]$ a seguir a $T[j]$)
 - $1 + D[i, j-1]$ (eliminação de $T[j]$)

Ilustração



Matriz de programação dinâmica

$D[i,j]$

T

		b	c	d	e	f	f	g	h	i	x	k	l
	0	1	2	3	4	5	6	7	8	9	10	11	12
a	1	1	2	3	4	5	6	7	8	9	10	11	12
b	2	1	2	3	4	5	6	7	8	9	10	11	12
c	3	2	1	2	3	4	5	6	7	8	9	10	11
d	4	3	2	1	2	3	4	5	6	7	8	9	10
e	5	4	3	2	1	2	3	4	5	6	7	8	9
f	6	5	4	3	2	1	2	3	4	5	6	7	8
g	7	6	5	4	3	2	2	2	3	4	5	6	7
h	8	7	6	5	4	3	3	3	2	3	4	5	6
i	9	8	7	6	5	4	4	4	3	2	3	4	5
j	10	9	8	7	6	5	5	5	4	3	3	4	5
k	11	10	9	8	7	6	6	6	5	4	4	3	4
l	12	11	10	9	8	7	7	7	6	5	5	4	3

P

Pseudo-código

Tempo e espaço: $O(|P| \cdot |T|)$

```

EditDistance(P,T) {
    // inicialização
    for i = 0 to |P| do D[i,0] = i
    for j = 0 to |T| do D[0,j] = j

    // recorrência
    for i = 1 to |P| do
        for j = 1 to |T| do
            if P[i] == T[j] then D[i,j] = D[i-1,j-1]
            else D[i,j] = 1 + min(D[i-1,j-1],
                                   D[i-1,j],
                                   D[i,j-1])

    // finalização
    return D[|P|, |T|]
}

```

Optimização de espaço

Espaço: $O(|T|)$

```

EditDistance(P,T) {
  // inicialização
  for j = 0 to |T| do D[j] = j // D[0,j]
  // recorrência
  for i = 1 to |P| do
    old = D[0] // guarda D[i-1,0]
    D[0] = i // inicializa D[i, 0]
    for j = 1 to |T| do
      if P[i] == T[j] then new = old
      else new = 1 + min(old,
        // Ainda tem valor anterior D[i-1,j]
        D[j],
        // Já tem valor da iteração corrente, i.e., D[i, j-1]
        D[j-1])
      old = D[j]
      D[j] = new
  // finalização
  return D[|T|]
}

```

Outros problemas

- Sub-sequência comum mais comprida (*longest common subsequence*)
 - Formada por caracteres não necessariamente consecutivos
 - ABD ? ABCDEF (delete)
- Substring comum mais comprida (*longest common substring*)
 - Formada por caracteres consecutivos
 - ABAB (BAB) BABA (BA) ABBA -> {AB, BA} (tamanho 2)
- Compressão de texto com códigos de Huffman
- Criptografia

Referências e mais informação

- “Introduction to Algorithms”, Second Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press, 2001
 - Fonte consultada para o “matching” exato
- “The Algorithm Design Manual”, Steven S. Skiena, Springer-Verlag, 1998
 - Fonte consultada para o “matching” aproximado
 - Discute como se usa o cálculo da distância de edição para encontrar num texto T a substring que faz o melhor “match” com um padrão de pesquisa P