Algoritmos de Pesquisa em Strings

R. Rossetti, L. Ferreira, H. L. Cardoso, F. Andrade

FEUP, MIEIC, CAL

Índice

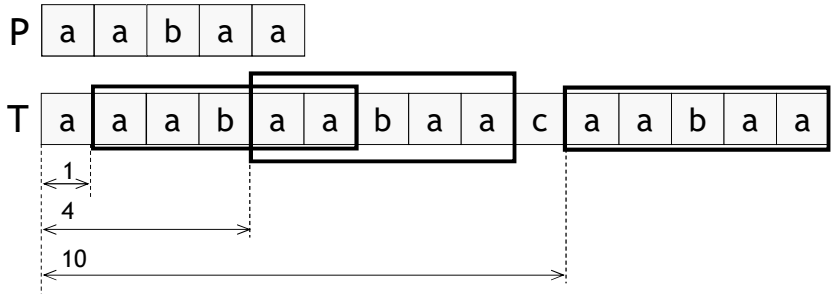
- Pesquisa exata (*string matching*)
- Pesquisa aproximada (*approximate string matching*)
- Outros problemas de pesquisa

Pesquisa exata

(*string matching*)

Problema

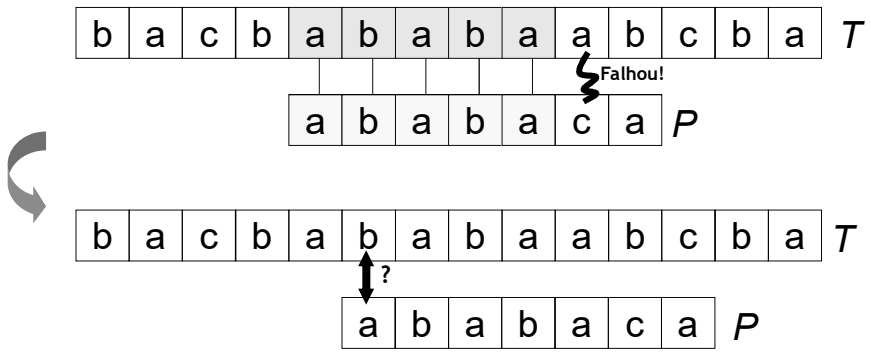
- Encontrar todas as ocorrências de um padrão P num texto T
  - P e T são cadeias de caracteres
  - Ocorrências são definidas pela deslocação em relação ao início do texto
  - Ocorrências podem ser sobrepostas



Algoritmos

- Algoritmo *naïve* → Abordagem caracter-a-caracter
  - Para cada deslocamento possível, compara desde o início do padrão
  - Ineficiente se o padrão for comprido: O(|P|.|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|.|Σ|), em que |Σ| 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 *naive*

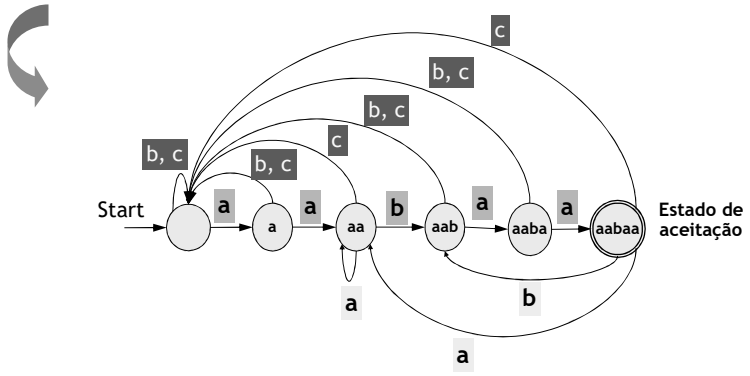


Desloca-se o padrão uma casa para a direita e recomeça-se a comparação do início do padrão! Ineficiente: O(|P| |T|)

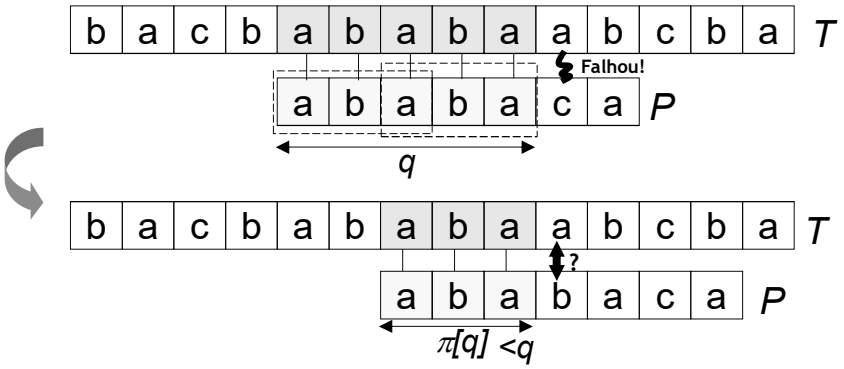
Autómato finito correspondente ao padrão

P | a | a | b | a | a |

Σ = {a, b, c}



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 π[q] calculada numa fase de pré-processamento do padrão!

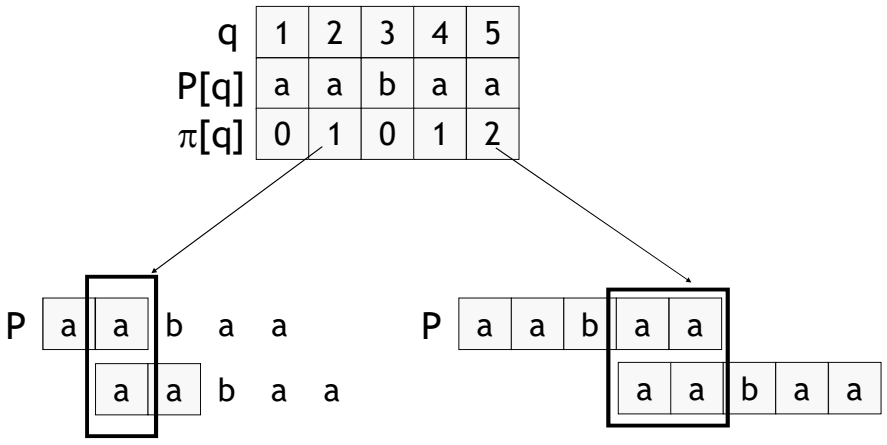
Pré-processamento do padrão

- 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, …, |P|
- P[i…j] – substring entre índices i e j
- Índices a começar em 1
- π[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



MATCH >> i++ (i avança)
MISMATCH >> i = min(0, i - 1) (i recua)
tabela[j] = i (depois de atualizar i)
A cada iteração j incrementado (for loop em j até ao tamanho do pattern [j++])

a	b	a	c	a	b
0	0	1	0	1	2

Pseudo-código

```
KMP-MATCHER(T, P)
 1  n ← length[T]
 2  m ← length[P]
 3  π ← COMPUTE-PREFIX-FUNCTION(P)
 4  q ← 0                              ▷ Number of characters matched.
 5  for i ← 1 to n                     ▷ Scan the text from left to right.
 6      do while q > 0 and P[q + 1] ≠ T[i]
 7           do q ← π[q]               ▷ Next character does not match.
 8         if P[q + 1] = T[i]
 9           then q ← 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 ← π[q]             ▷ Look for the next match.
```


Pseudo-código

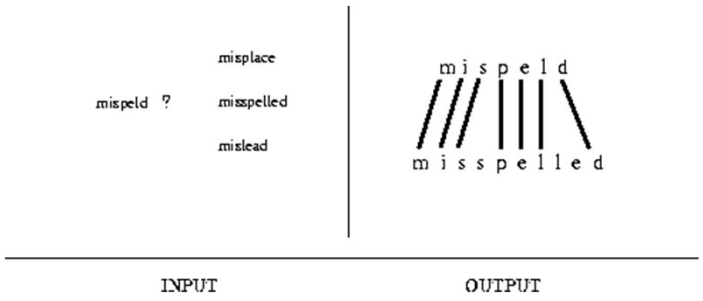
```
COMPUTE-PREFIX-FUNCTION(P)
 1  m ← length[P]
 2  π[1] ← 0
 3  k ← 0
 4  for q ← 2 to m
 5      do while k > 0 and P[k + 1] ≠ P[q]
 6           do k ← π[k]
 7         if P[k + 1] = P[q]
 8           then k ← k + 1
 9         π[q] ← k
10  return π
```

```cpp
int kmpMatcher(std::string text, std::string pattern) {
    int n=text.size();
    int m=pattern.size();
    std::vector<int> prefix = computePrefix(pattern);
    int q=-1;
    int result=0;
    for (int i = 0; i < n; ++i) {
        while (q>=0 && pattern[q+1]!=text[i]){
            q=prefix[q];
        }
        if (pattern[q+1]==text[i]){
            q++;
        }
        if(q==m-1){
            result++;
            q=prefix[q];
        }
    }
    return result;
}

std::vector<int> computePrefix(std::string str){
    int m = str.size();
    std::vector<int> result(m);
    result[0]=-1;
    int k=-1;

    for (int i = 1; i < m; i++) {
        while (k>=0 && str[k+1]!=str[i]){
            k=result[k];
        }
        if (str[k+1]==str[i]){
            k++;
        }
        result[i]=k;
    }

    return result;
}
```

Pesquisa aproximada

(*approximate string matching*)

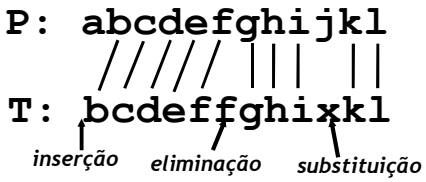
Problema



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

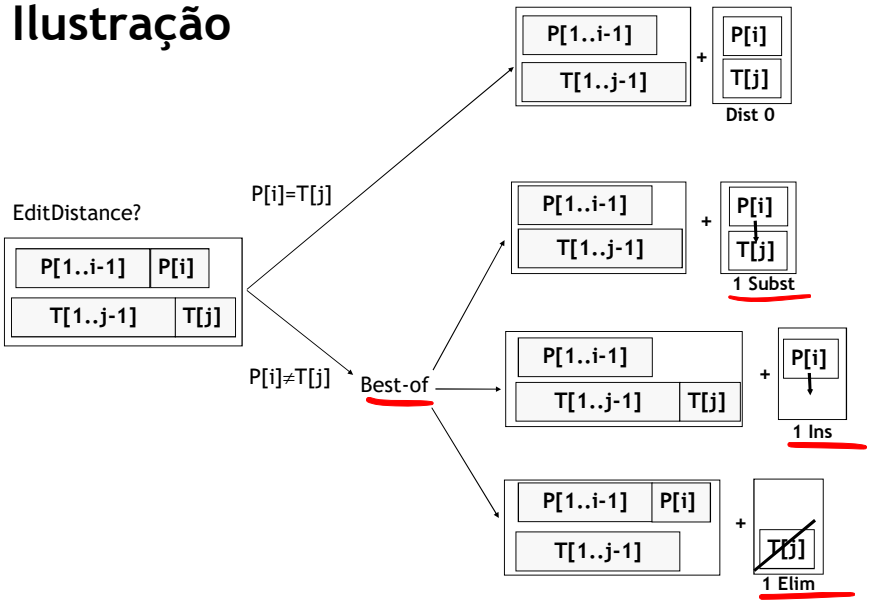
Formulação recursiva

- $D[i,j]$ = *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 $D[i, j] = D[i-1, j-1]$ → Nada a fazer caracteres iguais
  - Senão, escolhe-se a operação de edição que sai mais barata; isto é, $D[i,j]$ é o mínimo de:

O menor para manter o mínimo esforço necessário

- $1 + D[i-1, j-1]$ (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

P

		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

Pseudo-código

Tempo e espaço: O(|P|.|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|]
}
```


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