I.C.M.S.C.

INSTITUTO DE CIÊNCIAS MATEMÁTICAS DE SÃO CARLOS

ALGORITMOS DE BUSCA EM TEXTO

Maria das Graças Volpe Nunes

UNIVERSIDADE DE SÃO PAULO

SÃO CARLOS - SÃO PAULO
BRASIL

ALGORITMOS DE BUSCA EM TEXTO

Maria das Graças Volpe Nunes

Orientadora: *Profa. Dra. Maria Carolina Monard*

Dissertação apresentada ao Instituto de Ciências Matemáticas de São Carlos, da Universidade de São Paulo, para a obtenção do Título de Mestre em Ciências de Computação e Estatística

São Carlos
1984

A meus pais
Guilherme ('In Memorian')
e Ernesta,

Ao Wagner,

Aos meus irmãos,

Dedico este trabalho.

Resumo

Este trabalho tem por objetivo apresentar e analisar os algoritmos que realizam buscas de cadeias de caracteres em textos, bem como identificar, entre eles, os algoritmos apropriados para determinadas circunstâncias sob as quais este procedimento se fizer necessário.

Os métodos de busca em texto podem ser divididos em dois grupos: no primeiro, a busca é feita sobre o texto original; no segundo, uma versão pré-processada do texto original e da cadeia procurada são utilizadas para a busca.

Do primeiro grupo, apresentamos e analisamos o algoritmo Simples, o algoritmo de Knuth, Morris e Pratt, o de Boyer e Moore e o de Aho e Corasick. O segundo grupo é representado pelo Método de Harrison que utiliza assinaturas associadas ao texto e à cadeia procurada para encontrar a primeira ou todas as ocorrências de uma cadeia no texto.

Experiências foram realizadas com os algoritmos citados e a análise dos resultados obtidos é apresentada.

Abstract

In this work we describe and analyse some algorithms for text searching.

Text searching can be divided in two groups: without pre-processing and with pre-processing of the original text as well as the search string.

On the first group we analyse the naive algorithm, the Knuth-Morris-Pratt algorithm, the Boyer-Moore algorithm and the Aho-Corasick pattern matching machines.

On the second group we describe Harrison's method wich uses signature functions. Several signature functions are discussed.

Experimental results for the behavior of the algorithms in different alphabets and in which circunstances the algorithms should be used conclude our work.

Índice

Introdução 01

Capítulo 1

1.0 - Introdução 04

1.1 - Algoritmo Simples 04

1.2 - Algoritmo Menor Frequência 06

1.3 - Algoritmo de Knuth-Morris-Pratt (KMP) 08

1.3.1 - Exemplo 08

1.3.2 - Construção da Tabela prox 12

Capítulo 2

2.1 - Algoritmo de Boyer-Moore (BM) 16

2.1.1 - Exemplo 20

2.1.2 - Construção das Tabelas delta1 e delta2 23

2.1.3 - Algoritmo de Boyer-Moore Simplificado (BMS) 27

Capítulo 3 30

Capítulo 4

4.0 - Introdução 41

4.1 - Algoritmo de Aho-Corasick 41

4.1.1 - Exemplo 42

4.1.2 - Construção das Funções de Transição, de Falha e de Saída 46

4.1.3 - Complexidade dos Algoritmos 53

4.1.4 - Eliminação da Função de Falha 54

Capítulo 5

5.0 - Introdução 60

5.1 - O Método de Harrison 61

5.1.1 - A Escolha dos Parâmetros w e k 65

5.1.2 - A Escolha da Função Hash h 65

5.1.3 - Resultados Experimentais 68

Capítulo 6

Conclusões 78

Bibliografia 83

Apêndice 86

Introdução

O problema de detetar a (não) ocorrência de uma ou várias cadeias de caracteres num texto caracteriza uma parte importante de muitos problemas relacionados a processamento de textos como edição de textos, recuperação de informações, manipulação de símbolos e pesquisa bibliográfica. Desde que o texto pesquisado pode ser extremamente grande - talvez centenas de milhares de caracteres - torna-se essencial utilizar técnicas eficientes para localizar rapidamente a primeira ou todas as ocorrências das cadeias desejadas no texto.

Dividimos os métodos conhecidos em dois grupos que se distinguem principalmente pela maneira de se considerar o texto pesquisado. Num primeiro grupo, colocamos os algoritmos que realizam a busca sobre o texto original. Ainda dentro deste grupo, pode-se diferenciar os algoritmos que fazem a busca comparando os caracteres do texto com os da cadeia, da esquerda para a direita, a partir do primeiro caractere da cadeia, daqueles em que esta pesquisa é feita da direita para a esquerda, a partir do último caractere da cadeia. Entre os primeiramente citados encontram-se: o algoritmo *Simples*, que é o procedimento mais natural para a solução do problema e mostrou-se o menos eficiente na maioria dos casos e o algoritmo de Knuth, Morris e Pratt (KMP) que elimina todo o retrocesso inerente ao algoritmo *Simples*, ou seja, um caractere do texto não é inspecionado mais do que uma vez.

O algoritmo de Boyer e Moore (BM) inverte o sentido das comparações entre cadeia e texto e com isso diminue o número de comparações entre caracteres necessárias.

Um estudo sobre os algoritmos deste primeiro grupo revela a superioridade do algoritmo *BM* sobre os demais além da pequena vantagem do algoritmo *KMP* sobre o algoritmo *Simples*.

Enquanto os algoritmos citados procuram a primeira ou todas as ocorrências de uma única cadeia no texto, o método de Aho e Corasick (*MRC*) realiza simultaneamente esta tarefa para um conjunto de cadeias, baseado numa máquina de estado finito que deteta no texto todas ocorrências de cada cadeia do conjunto.

Num segundo grupo destacamos os algoritmos que realizam pré-processamento do texto original, transformando-o num texto compactado sobre o qual a busca é realizada de uma maneira mais rápida e determinística. O Método de Harrison é representante deste grupo e utiliza o conceito de assinaturas associadas ao texto e à cadeia procurada para compactar e pesquisar o texto. Duas funções distintas que determinam a assinatura adotada são amplamente testadas e os resultados obtidos são apresentados.

Nos quatro primeiros capítulos, apresentamos os algoritmos pertencentes ao primeiro grupo de métodos.

No *Capítulo 1* introduzimos os algoritmos *Simples* e uma versão modificada que utiliza informações sobre caracteres no texto e o algoritmo *KMP*.

Destacamos no *Capítulo 2*, devido a sua importante inovação no sentido das comparações efetuadas, o algoritmo BM e uma versão simplificada, *BMS*.

No *Capítulo 3* fazemos um estudo comparativo dos algoritmos acima citados, através de análises de resultados experimentais. Tomamos como medida de eficiência o número de comparações realizadas entre caracteres do texto e da cadeia. Também ana

lisamos a influência do tamanho do alfabeto sob o qual o texto é escrito, realizando os experimentos sobre textos Binário, Português e Centenário.

A máquina reconhecedora de cadeia de Aho e Corasick é apresentada no *Capítulo 4*.

O *Capítulo 5* é dedicado ao estudo do Método de Harrison, representante do segundo grupo de métodos, onde também são analisados os resultados experimentais obtidos.

Finalmente, no *Capítulo 6*, apresentamos nossas conclusões finais.

Os algoritmos programados em Pascal encontram-se no *Apêndice*.

A notação adotada neste trabalho segue o seguinte critério: *texto* é o vetor de caracteres que representa o texto pesquisado e $n>0$ é seu comprimento; *cad* é a subcadeia que desejamos encontrar e $m>0$ é seu comprimento. Usamos ainda, a notação S[i..j] para representar uma subcadeia constituída pelos caracteres S[i], S[i+1],..., S[j].

Capítulo 1

1.0 - Introdução

Os algoritmos que procuram a primeira ocorrência de uma cadeia num texto não pré-processado, podem ser divididos em dois grupos.

A característica que os distingue é o sentido adotado para a sequência de comparações entre a cadeia e o texto. Em um grupo estão os algoritmos que, a partir do caractere mais à esquerda da cadeia, seguem à direita fazendo comparações com os correspondentes caracteres do texto. Devido a isso, tais algoritmos inspecionam cada caractere do texto (até o ponto onde foi encontrada a cadeia) pelo menos uma vez. O segundo grupo caracteriza-se por adotar um sentido inverso na sequência de comparações, ou seja, estas são feitas da direita para a esquerda.

Neste capítulo, destacamos os algoritmos - *Simples*, *Menorfreq*, *KMP* - pertencentes ao primeiro grupo.

1.1 - Algoritmo Simples

Este algoritmo é, provavelmente, o procedimento mais imediato que nos ocorre ao resolvermos um problema de busca em texto. Inicialmente, alinhamos o caractere mais à esquerda da cadeia - *cad*[1] - com o caractere mais à esquerda do texto - *texto*[1]. Segue-se então, comparando os caracteres alinhados e tão logo ocorra um insucesso, *cad* é deslocada de uma posição à direita, recomeçando o processo a partir da nova posição de *cad*[1].

próxima posição de cad[1]
caso haja insucesso

cad : ___ ↓ ___ ___

cad[1]

texto : ___ ___ ___ ___ ___ ___ ___ ___ ___

texto[1]

⟶ sentido das comparações

Algoritmo 1. Algoritmo ***Simples***

Entrada : *cad*, *texto*, *m*>0, *n*>0.

Saída : a localização do primeiro caractere de *cad* na sua primeira ocorrência em *texto*, se houver sucesso; 0 se houver insucesso.

```
   if  m>n then return(0);
   car  ← cad[1];
   k ← 0;
repeat
   procure em texto[k+1..n-m+1] a
         primeira ocorrência de car;
   if  car  não foi encontrado then return(0);
   k ← posição onde car foi encontrado
until  texto[k..k+m-1]=cad;
return(k);
```

Não é difícil verificar que o pior caso ocorre se, para toda possível posição inicial de *cad* em *texto*, todo, com ex

cessão do último caractere de *cad* coincidir com o correspondente caractere de *texto*; por exemplo, quando $cad=a^{m-1}b$ e $texto=a^{n}$, com $n>>m$. Neste caso, $O(mn)$ comparações são necessárias para determinar que a cadeia não ocorre no texto. Como tal situação é relativamente rara, pode-se dizer que o algoritmo *Simples* é, em média, praticamente linear e pode ser bastante aceitável em algumas situações. Em particular, nenhum método pode superá-lo quando a cadeia possuir apenas um caractere.

1.2 - *Algoritmo Menor Frequência*

Horspool [HORSPOOL-80] observou que, se conhecermos as frequências de ocorrência dos símbolos do alfabeto no texto, podemos melhorar o algoritmo *Simples*, quanto ao número de comparações feitas e, consequentemente, quanto ao tempo de execução.

Suponha que procuramos num texto em Português pela palavra 'EXTRA'; o algoritmo *Simples* localiza, então, ocorrências sucessivas da letra 'E'. Infelizmente, 'E' é uma das letras mais frequentes na língua Portuguesa e o algoritmo encontraria um 'E' em cada 10 caracteres aproximadamente. Assim, haveria muitas comparações entre *cad* e *texto* resultando em insucessos após cada 'E', antes de se obter a correspondência desejada.

Por outro lado, 'X' é uma das letras menos frequentes em Português. Se usarmos o algoritmo para localizar ocorrências sucessivas de 'X' ao invés de 'E', seremos capazes de avançar mais rapidamente a busca ao longo do texto, ou seja, deixamos de fazer muitas comparações que resultariam em insucessos. Assim, se

lecionando o caractere de *cad* com a menor frequência de ocorrência em *texto*, podemos melhorar a velocidade do algoritmo *Simples*. O algoritmo *Menorfreq* a seguir, adota este procedimento.

Algoritmo 2. Algoritmo *Menorfreq*

Entrada: *cad*, *texto*, $m>0$, $n>0$, frequências dos símbolos no texto.

Saída : a localização do primeiro caractere de *cad* na sua primeira ocorrência em *texto*, se houver sucesso; 0 se houver insucesso.

```
if  m>n  then return(0);
encontrar j tal que cad[j] é o caractere de cad com menor fre-
   quência em texto;
car ← cad[j];
k ← j-1;
repeat
   procure em texto[k+1..n-m+j] a primeira ocorrência de car;
   if  car não foi encontrado then return(0);
   k ← posição onde car foi encontrado
until  texto[k-j+1..k+m-j]=cad;
return(k-j+1);
```

As informações sobre frequência utilizadas pelo algoritmo podem ser fornecidas na forma de uma lista de frequências dos possíveis caracteres, de acordo com suas ocorrências no texto.

Quanto maior a cadeia, mais rapidamente esperamos

avançar no texto. Isto ocorre porque em cadeias longas, há maior probabilidade de se encontrar um caractere de baixa frequência no texto.

Além do pior caso dos Algoritmos 1 e 2 - $O(mn)$ - eles possuem uma outra propriedade que os tornam inconvenientes em certas aplicações: eles envolvem retrocessos no texto, ou seja, voltam a comparar um caractere do texto já comparado anteriormente. Isto provoca ineficiência se todo o texto não estiver disponível na memória e operações com "buffers" forem necessárias. O próximo algoritmo não possui esta propriedade e possui um pior caso de apenas $O(m+n)$.

1.3 - *Algoritmo de Knuth-Morris-Pratt* (KMP)

Knuth, Morris e Pratt [KNUTH-77] construiram um algoritmo que pode ser informalmente descrito como: Inicialmente, cadeia e texto estão alinhados como no algoritmo *Simples* e o percurso de comparações é feito à direita. Entretanto, quando ocorre uma não-coincidência de caracteres, a cadeia é deslocada à direita de tal maneira que o processo pode ser reinicializado a partir do ponto de não-coincidência no texto. Assim, nenhum retrocesso é necessário.

1.3.1 - *Exemplo*

Considere a busca da cadeia 'abcabcaca' no texto 'cabccabcabcaabcabcacaa'. Inicialmente alinhamos o primeiro caractere da cadeia com o primeiro do texto e estamos prontos a testar o primeiro caractere do texto:

```
abcabcaca
cabccabcabcaabcabcacaa
↑
```

A flecha indica o caracter atual do texto; desde que 'c' não coincide com 'a', deslocamos a cadeia uma posição à direita e o caractere atual passa a ser o segundo caractere do texto:

```
 abcabcaca
cabccabcabcaabcabcacaa
 ↑
```

Agora houve coincidência, então não movemos a cadeia enquanto os caracteres do texto coincidirem com os respectivos caracteres da cadeia.

```
 abcabcaca
cabccabcabcaabcabcacaa
    ↑
```

Nesse ponto, já verificamos a ocorrência dos 3 primeiros caracteres da cadeia, mas não do quarto, então sabemos que os 4 últimos caracteres do texto são 'abcx' onde $x \neq$ 'a' e não temos que lembrar os caracteres verificados anteriormente, desde que nossa posição na cadeia nos fornece suficiente informação para recriá-los. Neste caso, independente de quem é x (uma vez que não é 'a'), concluímos que a cadeia pode ser deslocada 4 posições à direita, pois uma, duas ou três não levariam a uma coincidência.

Desta maneira, conseguimos uma nova correspondência parcial, agora com uma falha no oitavo caractere da cadeia:

```
     abcabcaca
cabccabcabcaabcabcacaa
            ↑
```

Agora sabemos que os oito últimos caracteres do texto são 'abcabcax', onde $x \neq$ 'c'. A cadeia deve então ser deslocada 3 posições à direita:

```
          abcabcaca
cabccabcabcaabcabcacaa
```

Tentamos o novo caractere da cadeia e novamente este falha, movemos então a cadeia 4 posições mais. Isto produz uma coincidência e continuamos até encontrar uma nova falha ou então descobrir a cadeia toda:

```
          abcabcaca
cabccabcabcaabcabcacaa
```

Vemos que esse processo será eficiente se tivermos uma tabela auxiliar que nos informe exatamente quantas posições devemos mover a cadeia quando detetamos uma falha no seu j-ésimo caractere - *cad*[j].

Seja *prox*[j] a posição do próximo caractere da cadeia que deve ser testado após ocorrer uma falha em *cad*[j]; assim, devemos deslocar *cad*, j-*prox*[j] posições à direita.

A tabela *prox* para a cadeia do exemplo dado seria então:

j=	1	2	3	4	5	6	7	8	9
cad[j]=	a	b	c	a	b	c	a	c	a
prox[j]=	0	1	1	0	1	1	0	5	0

Note que, *prox*[j]=0 indica que devemos deslocar totalmente a cadeia, isto é, o primeiro caractere da cadeia deve ser

comparado com o próximo caractere do texto. Discutiremos a construção de *prox* mais adiante.

Em cada passo do processo, podemos mover ou o ponteiro do texto ou a cadeia à direita, e cada um destes pode mover-se no máximo n vezes; então, no máximo $2n$ passos precisam ser realizados, após a construção da tabela *prox*. Naturalmente a cadeia não se movimenta, o que fazemos é atualizar o ponteiro j.

Algoritmo 3. Algoritmo *KMP*

Entrada : *cad*, *texto*, $m>0$, $n>0$, a função *prox*.

Saída : a localização do primeiro caractere de *cad* na sua primeira ocorrência em *texto*, se houver sucesso; 0 se houver insucesso.

```
j←1;  k←1;
while  k≤n  do
  begin
    while (j>0) and (texto[k] ≠ cad[j]) do
            j←prox[j];
    if  j=m then return(k-m+1)
    else begin
           k←k+1;
           j←j+1
         end
  end;
return(0);
```

Uma versão mais eficiente do algoritmo *KMP* é apre-

sentada em [KNUTH-77].

1.3.2 - *Construção da Tabela prox*

Quando o algoritmo *KMP* executa j←*prox*[j], sabemos que j>0 e que os últimos j caracteres de *texto* até, e inclusive, *texto*[k] são

cad[1]..*cad*[j-1] x

onde x ≠ *cad*[j]. O que queremos é efetuar o menor deslocamento da cadeia tal que esses caracteres possam, possivelmente, corresponder aos da cadeia deslocada; isto é, queremos que *prox*[j] seja o maior i menor que j tal que os últimos i caracteres do texto sejam

cad[1]..*cad*[i-1] x

e *cad*[i] ≠ *cad*[j]- se não existir tal i, fazemos *prox*[j]=0.

A tarefa de calcular *prox*[j] ficaria mais simples se não exigíssemos *cad*[i] ≠ *cad*[j] na definição acima, pois então poderíamos considerar o problema mais simples: Seja *f*[j] o maior i menor que j tal que *cad*[1]..*cad*[i-1]=*cad*[j-i+1]..*cad*[j-1]; desde que esta condição é sempre válida para i=1, sempre temos *f*[j]≥1. Por convenção, fazemos *f*[1]=0.

A cadeia do exemplo anterior possui a seguinte tabela *f*:

j=	1	2	3	4	5	6	7	8	9
cad[j]=	a	b	c	a	b	c	a	c	a
f[j]=	0	1	1	1	2	3	4	5	1

Podemos observar que, se *cad*[j]=*cad*[*f*[j]] então

$f[j+1]=f[j]+1$.

Vamos nos deter ao cálculo de $f[j+1]$ caso a condição acima não seja satisfeita.

Reconhecendo a semelhança deste problema com o nosso propósito original, realizado pelo algoritmo *KMP*, que é encontrar o maior j menor ou igual a k tal que

$$cad[1]..cad[j-1]=texto[k-j+1]..texto[k-1],$$

podemos transferir esta idéia para o problema de encontrar $f[j]$ sendo que agora $texto=cad$.

Vamos, então, calcular $f[j+1]$ assumindo que $f[j]$ e $prox[1],..,prox[j-1]$ já foram calculados:

```
t←f[j];
while(t>0) and (cad[j] ≠ cad[t]) do
            t ← prox[t];
f[j+1] ← t+1;
```

Como exemplo, suponhamos que tivéssemos estabelecido que $f[8]=5$ no caso anterior e vamos calcular $f[9]$. Podemos imaginar duas cópias da cadeia, uma movendo-se à direita em relação à outra. Neste caso, teríamos:

```
    abcabcaca
  abcabcaca
        ↑
```

Desde que $cad[8]\neq$'b', movemos a cadeia superior à direita, sabendo que os caracteres da cópia inferior mais recentemente visitados foram

abcax , com x≠'b'

A tabela *prox* nos diz para movermos 4(j-*prox*[j]=5-1) posições à direita, obtendo:

```
        abcabcaca
abcabcaca
        ↑
```

e novamente não houve correspondência. A próxima mudança faz t=0, portanto *f*[9]=1.

Comparando as definições de *prox*[j] e *f*[j], vemos que, para j>1,

$$prox[j] = \begin{cases} f[j], & \text{se } cad[j] \neq cad[f[j]] \\ prox[f[j]], & \text{se } cad[j] = cad[f[j]] \end{cases}$$

Assim, podemos computar a tabela *prox*, sem acumular os valores *f*[j], através do seguinte algoritmo.

Algoritmo 4. Construção da tabela *prox* para o Algoritmo KMP

Entrada : *cad*, *m*>0

Saída : *prox*, um vetor representando a função *prox* para *cad*.

```
j←1; t←0; prox[1]←0;
while  j<m  do
 begin (* prox[1]..prox[j] são conhecidos - calcular prox[j+1] *)
   while (t>0)  and  (cad[j] ≠ cad[t]) do
          t ← prox[t];
   t ← t+1;  j ← j+1;
   if  cad[j] = cad[t] then
         prox[j] ← prox[t]
   else prox[j] ← t
 end;
```

Algoritmo 4 é $O(m)$ no pior caso pois, se t é inicialmente 0 e é incrementado m-1 vezes de 1, desde que t permanece não-negativo, o comando t←*prox*[t], que decrementa t, pode ser executado no máximo m-1 vezes. Usando esses mesmos argumentos, substituindo t por j, pode-se mostrar que o Algoritmo 3 é $O(n)$ no pior caso. Combinando os Algoritmos 3 e 4, é possível determinar que uma cadeia de comprimento m não ocorre nas primeiras i posições de um texto, em $O(m+i)$ passos.

O algoritmo *KMP* faz, então, uso de duas idéias: pré-computamos movimentos, especificando como mover uma cadeia quando uma não-coincidência ocorre no seu j-ésimo caractere. Essa pré-computação de movimentos é feita eficientemente usando o mesmo princípio, ou seja, movendo a cadeia sobre si mesma.

Apesar de eliminar retrocessos no texto, observamos que o algoritmo *KMP* examina cada caractere do texto, pelo menos uma vez, com exceção daqueles caracteres onde a sequência de comparações é reinicializada.

Capítulo 2

2.1 - *Algoritmo de Boyer-Moore* (BM)

Nos três algoritmos do capítulo anterior - *Simples*, *Menorfreq* e *KMP* - a cadeia é procurada a partir da esquerda para a direita e cada caractere do texto é examinado, pelo menos uma vez. Boyer e Moore [BOYER-77] observaram que mais informações sobre o texto podem ser obtidas, se a busca for feita a partir da direita para a esquerda na cadeia. A informação obtida ao se iniciar pelo final da cadeia permite determinar que, sem ter que examiná-los, alguns caracteres do texto não podem fazer parte de uma ocorrência da cadeia no texto e, portanto, não precisam ser considerados na busca. Dizemos assim, que o algoritmo realiza "saltos" sobre o texto.

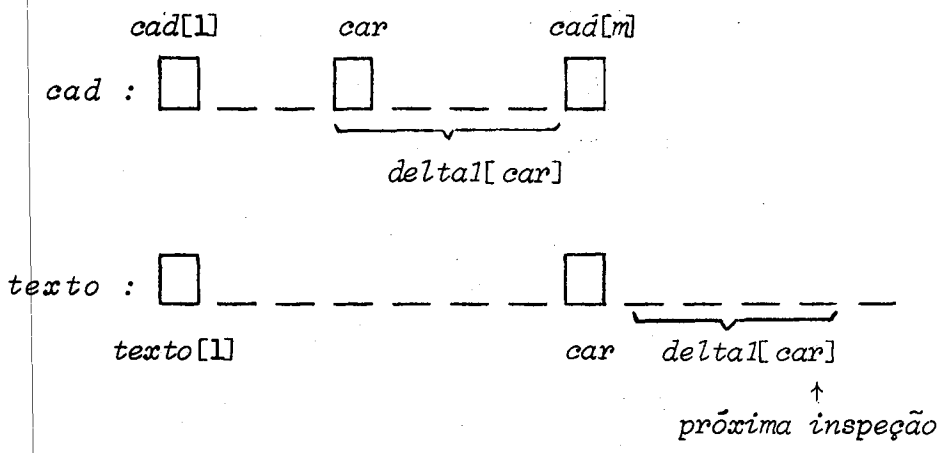
Segue-se a descrição do algoritmo.

Consideremos, novamente, *cad* alinhada com *texto* a partir do início de *texto*. Seja *car* o caractere do texto alinhado com *cad*[*m*], ou seja, *texto*[*m*], podemos então observar que:

Observação 1. Se *car* não ocorre em *cad*, então não precisamos considerar a possibilidade de uma ocorrência de *cad* ter início nas posições 1,2,...,*m* de *texto*. Tal ocorrência requereria que *car* fosse um caractere de *cad*. Neste caso, podemos mover *cad m* posições à direita, ou seja, alinhando *cad*[1] com *texto*[*m*+1].

Observação 2. Mais genericamente, se *car* ≠ *cad*[*m*] e a ocorrência mais à direita de *car* em *cad* se der à *deltal* caracteres de distância de *cad*[*m*], então podemos mover *cad deltal* posições à di-

reita sem inspecionar caracteres. Isto causa o alinhamento de *car* com sua ocorrência mais à direita em *cad* e as comparações podem recomeçar novamente no caractere alinhado com *cad*[*m*].

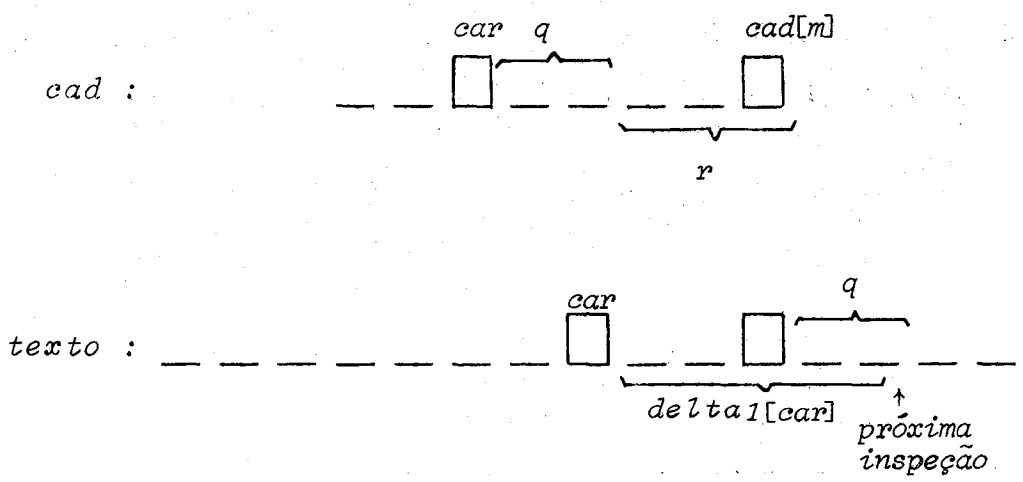


Assim, a menos que *car* corresponda a *cad*[*m*], podemos mover *cad* *delta1* posições à direita, deixando de inspecionar *delta1* caracteres do texto. *Delta1* é uma função do caractere *car* obtido no texto. Se *car* não ocorrer em *cad*, *delta1* é *m*, caso contrário *delta1* é a diferença entre *m* e a posição da ocorrência mais à direita de *car* em *cad*.

Agora, suponhamos que *car* = *cad*[*m*]. Então precisamos determinar se o caractere anterior a *car*, no texto, coincide com *cad*[*m*-1]. Se isso ocorrer, continuamos comparando à esquerda até encontrar toda a cadeia - e então o processo termina - ou encontrar um novo caractere - *car* - que não coincida com o correspondente em *cad* após terem sido inspecionados os últimos *r* caracteres de *cad*. Neste último caso, desejamos mover *cad* o maior número possível de posições à direita.

Observação 3(a). Podemos usar o mesmo procedimento acima - baseado no caractere não-coincidente, *car* e *delta1* - para mover *cad* de

q posições a fim de alinhar as duas ocorrências de *car*. Queremos então, inspecionar o caractere do texto alinhado com *cad[m]*. Assim, desviamos nossa atenção à $q+r$ caracteres à frente em *texto*. A distância q depende da posição de *car* em *cad*. Se a ocorrência mais à direita de *car* em *cad* estiver à direita do ponto de falha - isto é, dentro da região de *cad* já inspecionada - teríamos que mover *cad* para trás a fim de alinhar as duas ocorrências de *car*. Não gostaríamos de fazer isto. Neste caso, dizemos que *delta1* é "inútil" e movemos *cad* uma (q=1) posição à direita, o que nos faz desviar a atenção à $1+r$ posições à direita em *texto*. Se, por outro lado, *car* ocorrer à esquerda do ponto de falha, podemos mover *cad* à direita, q=*delta1[car]*$-r$ posições, para alinhar as duas ocorrências de *car*. Isto desvia nossa atenção para o caractere à direita no texto a uma distância *delta1[car]*$-r+r$=*delta1[car]*.

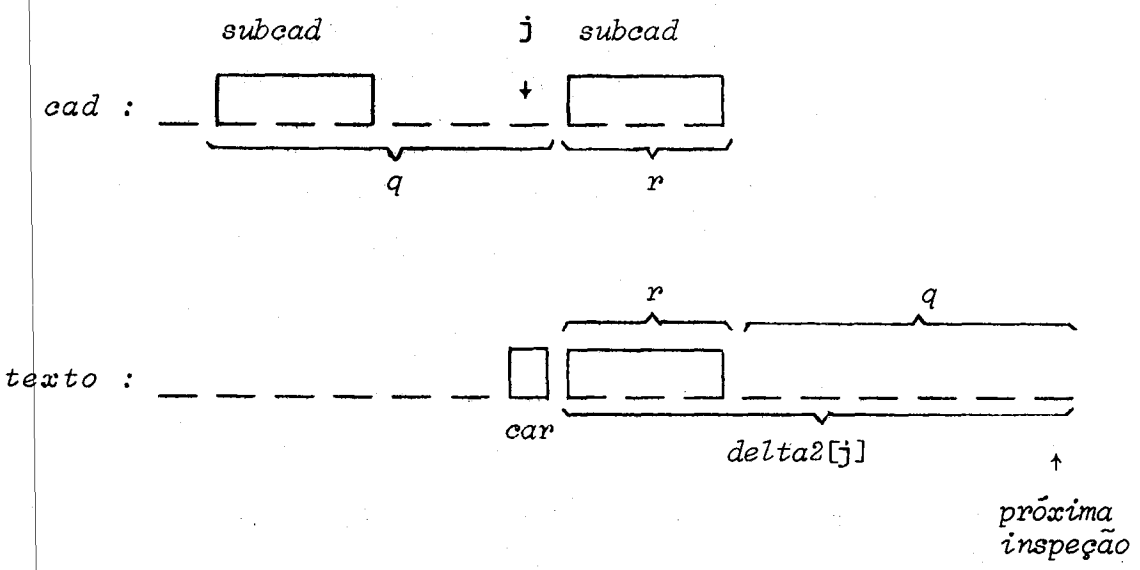


Entretanto, podemos raciocinar sob um outro aspecto.

Observação 3(b). Sabemos que os últimos r caracteres da cadeia coincidem com os próximos r caracteres do texto. Chamamos de *subcad* esta subcadeia de *cad*. Sabemos também que esta ocorrência

de *subcad* em *texto* é precedida por um caractere *car*, que é distinto daquele que precede *subcad* em *cad*. Podemos generalizar o procedimento usado acima e mover *cad* de tal modo que aquela ocorrência de *subcad* em *texto* fique alinhada com a ocorrência mais à direita de *subcad* em *cad* que não seja precedida pelo caractere que precedia sua última ocorrência na cadeia. Chamamos esta repetição de *subcad* em *cad* de "repetição plausível".

Então, de acordo com a *Observação 3(b)*, se observarmos a coincidência dos últimos *r* caracteres de *cad* antes de encontrarmos uma falha, podemos mover *cad* *q* posições, onde *q* é baseado na posição em *cad*, da repetição plausível mais à direita, contendo *r* caracteres. Após esse movimento, queremos inspecionar o caractere do texto alinhado com o último caractere de *cad*. Com isso, deixamos de inspecionar *q+r* caracteres do texto. Chamamos esta distância de *delta2*, que é uma função da posição j em *cad* onde ocorreu a falha; *q* é a distância entre *subcad* e sua repetição plausível mais à direita e é sempre maior ou igual a 1; *r* é exatamente *m*-j.



Assim, no caso em que temos coincidência nos últimos r caracteres de *cad* antes de uma falha, podemos "saltar" $1+r$ ou *delta1*[*car*] ou *delta2*[j] caracteres à frente no texto, de acordo com aquele que nos dá o maior salto. Da definição de *delta2* como $q+r$ onde $q \geq 1$, temos que *delta2* é, no mínimo, igual a $1+r$.

Então podemos nos deter apenas no maior valor entre os dois deltas.

2.1.1 - *Exemplo*

No exemplo a seguir, usamos o símbolo '↑' para indicar o caractere atual do texto que está sendo inspecionado. Quando este ponteiro for movido à direita, ele move consigo a cadeia alinhada ao texto. Quando for movido à esquerda, *cad* permanece fixa em relação ao texto.

```
   cad : SE_FOSSE
 texto : ALGO TERIA A TE__DIZER...._SE_FOSSE VERDADE...
                 ↑
```

Desde que 'R' não ocorre em *cad*, recorremos à *Observação 1* e movemos o ponteiro (e portanto *cad*) $m=8$ posições:

```
   cad :           SE_FOSSE
 texto : ALGO TERIA A TE__DIZER...._SE_FOSSE VERDADE...
                            ↑
```

De acordo com a *Observação 2*, podemos mover o ponteiro *delta1*['_']=5 posições à direita para alinhar os hífens:

```
cad :                    SE_FOSSE
texto : ALGO TERIA A TE__DIZER..._SE_FOSSE VERDADE...
                                 ↑
```

Agora o caractere do *texto* coincide com o de *cad*. Então passamos ao caractere à esquerda:

```
cad :                    SE_FOSSE
texto : ALGO TERIA A TE__DIZER..._SE_FOSSE VERDADE...
                                 ↑
```

De acordo com a *Observação 3(a)*, podemos mover o ponteiro m=8 (*delta1*['Z']) posições, pois 'Z' não ocorre em *cad*. Note que *cad* é movida apenas 7 posições à direita. Já *delta2* permitiria mover o ponteiro apenas 7 (q+r=6+1) posições à direita a fim de alinhar 'E' com sua repetição plausível.

```
cad :                          SE_FOSSE
texto : ALGO TERIA A TE__DIZER..._SE_FOSSE VERDADE...
                                     ↑
```

Novamente houve coincidência entre os dois caracteres. Voltando à esquerda, vemos que o caractere antecedente em *texto* também coincide com seu correspondente em *cad*. Um novo passo à esquerda produz

```
cad :                          SE_FOSSE
texto : ALGO TERIA A TE__DIZER..._SE_FOSSE VERDADE...
                                     ↑
```

Com esta nova falha apelamos à *Observação 3(b)*. O movimento *delta2* é melhor desde que nos permite mover o ponteiro 8 ($q+r=6+2$) posições a fim de alinhar a subcadeia 'SE' com sua repetição plausível no início de *cad*. Note que o movimento *delta1*['_'] levaria o ponteiro 5 posições à direita para alinhar os hífens:

```
cad :                                  SE_FOSSE

texto : ALGO TERIA A TE__DIZER...._SE_FOSSE VERDADE...
                                          ↑
```

Desta vez, descobrimos que todo caractere de *cad* coincide com seu correspondente em *texto*, de maneira que encontramos a primeira ocorrência da cadeia.

Note que fizemos apenas 15 inspeções num trecho de 34 caracteres. Oito delas ocorreram testando toda a cadeia no último passo. As outras sete nos permitiram passar pelos primeiros 26 caracteres de *texto*.

Para apresentar o algoritmo de Boyer-Moore, assumiremos a existência das tabelas *delta1* e *delta2*, cujas construções são discutidas mais adiante. Lembramos que *delta1* possui uma entrada para cada símbolo do alfabeto e *delta2* possui tantas entradas quanto o número de posições de caracteres na cadeia - m e a j-ésima entrada será denotada por *delta2*[j]. As duas tabelas contém inteiros não-negativos.

Algoritmo 5. Algoritmo *BM*

Entrada : *cad*, *texto*, *m*>0, *n*>0, *delta1* e *delta2*

Saída : a localização do primeiro caractere de *cad* na sua primeira ocorrência em *texto*, se houver sucesso; 0 se houver insucesso.

```
k←m;
while  k≤n  do
 begin
   j←m;
   while  (j>0)  and (texto[k]  = cad[j]) do
     begin
        j←j-1;
        k←k-1
     end;
   if  j=0  then return(k+1)
   else k←k + max(delta1[texto[k]], delta2[j])
 end;
return(0);
```

2.1.2 - *Construção das Tabelas delta1* e *delta2*

A tabela *delta1* possui uma entrada para cada caractere, *car*, do alfabeto. A definição de *delta1* é:

delta1[*car*]=*m*, se *car* não ocorre em *cad*; *m*-j caso contrário, onde j é o maior inteiro tal que *cad*[j]=*car*.

O processamento de *delta1* requer, então, um vetor do tamanho do alfabeto. A implementação pode inicializar todas as entradas com *m* e então atualizar aquelas entradas cujos caracte-

res coincidam com os da cadeia, num processo sequencial através da cadeia. Assim, o processamento de *delta1* é linear em *m* mais o tamanho do alfabeto.

A tabela *delta2* tem uma entrada para cada valor do intervalo 1..*m*. Podemos definir *delta2*[j] como

a) a distância que podemos mover *cad* a fim de alinhar a subcadeia formada por seus últimos *m*-j caracteres, com sua repetição plausível mais à direita, mais

b) a distância adicional que precisamos mover o ponteiro do texto de modo a reiniciar o processo no último caractere de *cad*.

Mas, para definir precisamente *delta2*, devemos definir a repetição plausível mais à direita de uma subcadeia terminal de *cad*. Para tal fim, convencionamos:

Seja $ um caractere que não ocorre em *cad* e dizemos que se k<1 então *cad*[k]=$. Dizemos também que duas sequências de caracteres $[c_1..c_p]$ e $[d_1..d_p]$ coincidem, se para todo i de 1 a p, ou $c_i=d_i$ ou c_i=$ ou d_i=$.

Finalmente, definimos a posição da repetição plausível mais à direita da subcadeia terminal que inicia na posição j+1, *rpd*(j), para j de 1 a *m*, como sendo o maior i, menor ou igual a *m*, tal que [*cad*[j+1]..*cad*[*m*]] e [*cad*[i].. *cad*[i+*m*-j-1]] coincidem e ou i≤1 ou *cad*[i-1] ≠ *cad*[j]. Isto é, a posição da repetição plausível mais à direita da subcadeia que começa na posição j+1 é a posição mais à direita onde ela ocorre novamente em *cad* e que não seja precedida pelo caractere *cad*[j] que precede a ocorrência terminal, podendo ou a repetição ou o caractere precedente "ultrapassarem" a extremidade esquerda de *cad*. Desta manei

ra, *rpd*(j) pode ser negativo devido a esse fato.

Assim, a distância que devemos mover *cad* para alinhar a subcadeia encontrada a partir da posição j+1 com sua repetição plausível é j+1-*rpd*(j). A distância que devemos mover o ponteiro do texto, para novamente comparar o último caractere de *cad* é exatamente *r*=*m*-j. Como *delta2*[j] é a soma dos dois termos, *delta2*[j]=*m*+1-*rpd*(j).

Vejamos os seguintes exemplos:

j :	1	2	3	4	5	6	7	8	9
cad :	A	B	C	X	X	X	A	B	C
rpd(j) :	-4	-3	-2	-1	0	1	-1	0	9
delta2[j] :	14	13	12	11	10	9	11	10	1

j :	1	2	3	4	5	6	7	8	9
cad :	A	B	Y	X	C	D	E	Y	X
rpd(j) :	-7	-6	-5	-4	-3	-2	3	0	9
delta2[j] :	17	16	15	14	13	12	7	10	1

O algoritmo que apresentamos a seguir, constrói as tabelas *delta1* e *delta2* em O(*m*+ *q*) passos, onde *q* é o tamanho do alfabeto. A construção da tabela *delta2* foi inicialmente apresentada por Knuth, Morris e Pratt [KNUTH-77] mas deixa de prever alguns casos muito especiais e raros. Devido a esse fato, esta construção foi posteriormente corrigida por Rytter [RYTTER-80] e por Smit [SMIT-82].

Algoritmo 6. Construção das Tabelas *delta1* e *delta2* **para o Algoritmo** *BM*

Entrada : *cad*, *m*>0
Saída : tabelas *delta1* e *delta2*

```
for  cada caractere  c do alfabeto de entrada do
                 delta1[c] ← m;
for  j←1  to  m  do    (* computa delta1 e inicializa delta2 *)
   begin
     delta1[cad[j]] ← m-j;
     delta2[j] ← 2*m-j
   end;
    (* computa delta2 *)
j←m; t←m+1;
while  j>0  do
   begin
     f[j]←t;
     while (t≤m)  and  (cad[j] ≠ cad[t])  do
      begin
         delta2[t] ←  min(delta2[t], m-j);
         t ← f[t]
      end;
     j ← j-1; t ← t-1
   end;
for  i ← 1  to  t  do   delta2[i]← min(delta2[i]; m+t-i);

(* o trecho a seguir foi sugerido por K. Mehlhorn, em comunicação
   a D.E. Knuth, em 1977 [SMIT-82] para garantir valores corretos
   de delta2 em todos os casos *)

tp ← f[t];
while  t≤m  do
   begin
     while  t≤tp  do
       begin
          delta2[t]← min(delta2[t], tp-t+m);
          t ← t+1
       end;
     tp ← f[tp]
   end;
```

Observamos, pela descrição do algoritmo, que ele pode não inspecionar todos os primeiros k+*m*-1 caracteres do texto, antes de encontrar a cadeia na posição k. Devido a isto, dizemos que o algoritmo é geralmente "sublinear", isto é, o valor esperado do número de caracteres inspecionados no texto é c*(k+*m*), onde c<1, e diminui conforme *m* aumenta. Assim, este algoritmo é mais rápido quando busca longas cadeias e mais lento conforme o tamanho do alfabeto diminui, pois aumenta a probabilidade de caracteres coincidentes e os "saltos" serão menores. De fato, quando o alfabeto é grande, precisamos inspecionar apenas *n/m* caracteres em média para concluirmos que a cadeia não ocorre no texto.

Knuth [KNUTH-77], mostrou que o algoritmo *BM* é linear em *n* também no pior caso. Desconsiderando o pré-processamento de *delta1* e *delta2*, ele provou que o algoritmo inspeciona, no máximo, 6k caracteres a fim de descobrir que *cad* não ocorre nos primeiros k caracteres do texto. Portanto, o limite para o pior caso seria 6*n*. Além disso, revelou que a linearidade do algoritmo é inteiramente devida à tabela *delta2* e que a constante 6 deste limite seria provavelmente muito grande. De fato, Guibas e Odlyzko [GUIBAS-77] e Galil [GALIL-79] melhoraram o limite para o pior caso para 4*n* e sugeriram ainda, que o limite real seria 2*n*.

2.1.3 - *Algoritmo de Boyer-Moore Simplificado (BMS)*

A finalidade da tabela *delta2* é otimizar o comportamento do algoritmo no tratamento de cadeias que possuem subcadeias repetitivas (por exemplo 'XABCYYABC') e portanto, evitar um tempo de execução de O(*mn*).

Se considerarmos que cadeias deste tipo não são muito comuns em textos sobre alfabetos razoavelmente grandes, torna-se desvantajoso dispender o considerável esforço necessário para construir a tabela *delta2*.

Mostramos então, uma versão simplificada do algoritmo *BM* a qual chamamos de *BMS*, que faz uso apenas da tabela *delta1*, ignorando *delta2*. Entretanto, um cuidado se faz necessário: agora, ao tomar o valor *delta1* este pode ser "inútil", isto é, a ocorrência do caractere pode se dar à direita do ponto de falha, portanto, devemos tomar o máximo valor entre *delta1* e (*m*-j+1), onde j é a posição onde houve falha e (m-j+1) significa mover a cadeia uma única posição à direita (Lembramos que era o fato de *delta2*[j] ser no mínimo 1+*m*-j que fazia com que não considerássemos este movimento mínimo quando *delta1* fosse "inútil").

Horspool [HORSPOOL-80] sugere que o valor *delta1*[*cad*[*m*]] - zero, pela definição de *delta1* - seja agora tomado de *delta2*[*m*], que nada mais é do que o número de caracteres iguais a *cad*[*m*] que aparecem adjacentes na extremidade direita de *cad* (por ex., para *cad*='ABDD', *delta2*[*m*]=2); assim, no mínimo *delta1*[*cad*[*m*]] será 1. Com isso, evitamos o movimento mínimo quando sabemos que um caractere igual a *cad*[*m*] será comparado na posição onde *cad*[*m*] falhou.

O algoritmo *BMS* é apresentado na página seguinte.

Esperamos que o algoritmo *BMS* comporte-se em média, como o algoritmo *BM* quando o alfabeto for razoavelmente grande, mas seja inferior a este quando o alfabeto for pequeno, pois é neste caso que a tabela *delta2* tem maior importância. Em particular, quando a cadeia é longa, o valor *delta1* é em geral pequeno pois muitos caracteres do alfabeto devem ocorrer na cadeia e ape

Algoritmo 7. Algoritmo *BMS*

Entrada : *cad*, *texto*, *m*>0, *n*>0, *delta1*

Saída : a localização do primeiro caractere de *cad* na sua primeira ocorrência em *texto*, se houver sucesso; 0 se houver insucesso.

```
k←m;
while  k ≤ n  do
  begin
    j ← m;
    while  (j>0)  and  (texto[k] = cad[j]) do
      begin
        j ← j-1;
        k ← k-1
      end;
    if  j=0  then return(k+1)
    else  k ← k + max(delta1[texto[k]], m-j+1)
  end;
return(0);
```

nas as coincidências entre subcadeias terminais causariam "saltos" maiores.

Boyer e Moore provam a correção da versão simplificada do algoritmo *BM* (onde apenas não otimizam o valor *delta1*[*cad*[*m*]]) através do método da asserção indutiva em [BOYER-79], capítulo XVIII.

Capítulo 3

Com o objetivo de verificar o comportamento dos algoritmos *Simples*, *Menorfreq*, *KMP*, e *BMS*, quanto ao número de comparações efetuadas entre caracteres da cadeia procurada e caracteres do texto, realizamos a seguinte experiência:

Tomamos três textos de 20.000 caracteres cada: um texto em Postuguês sobre Programação Estruturada sobre um alfabeto de 64 símbolos (ASCII); um texto sobre um alfabeto Binário (0,1) e um texto sobre um alfabeto Centenário (0..99). Os dois últimos — Binário e Centenário — foram gerados aleatoriamente. A partir de cada texto geramos, aleatoriamente, grupos de cadeias de comprimentos 2,3,..,15, cada um composto de 200 cadeias.

Os algoritmos foram testados sob os três textos, com exceção do algoritmo *Menorfreq* que foi testado apenas com o texto em Português, devido a natureza aleatória dos demais textos. As informações sobre frequências de símbolos foram obtidas a partir do próprio texto.

Para cada cadeia, os algoritmos procuravam todas suas ocorrências no texto em questão. Obteve-se então, o número médio de comparações efetuadas para cada grupo de 200 cadeias.

Medimos o custo de cada busca através do número de referências feitas ao texto, ignorando o pré-processamento para construir as tabelas *prox*(*KMP*), *delta1* e *delta2*(*BM*, *BMS*).

Dividindo o número de referências ao texto pelo número de caracteres passados antes da cadeia ser encontrada (ou o texto ter se esgotado), obtivemos o número de referências ao texto por caractere passado. Essa medida é independente da particular implementação dos algoritmos. Tomamos então, a média das

medidas obtidas em cada grupo de 200 cadeias para todo comprimento de cadeia. Os algoritmos foram programados em Pascal, no computador VAX 11/780+.

As tabelas, a seguir, apresentam para cada alfabeto, o número médio de comparações (em porcentagem) realizadas pelos algoritmos citados, para cada comprimento de cadeia.

comprimento das cadeias	*ALFABETO BINÁRIO - 20.0000 caracteres*			
	número médio de comparações (%)			
	Simples	*KMP*	*BM*	*BMS*
2	121.0	133.6	92.0	98.4
3	139.9	126.2	90.5	111.8
4	158.0	125.5	85.8	122.1
5	172.2	123.6	80.7	136.1
6	183.0	123.8	76.7	146.2
7	189.5	123.7	73.3	149.1
8	193.6	123.5	70.3	157.9
9	195.9	121.5	65.4	153.7
10	197.5	124.4	61.7	153.5
11	198.1	124.6	59.2	154.3
12	198.4	123.6	57.9	158.7
13	198.8	121.6	54.4	159.2
14	198.9	122.2	53.4	157.2
15	198.9	123.3	52.5	162.0

+ *VAX 11/780 User's Guide*

comprimento das cadeias	ALFABETO PORTUGUÊS - 20.000 caracteres				
	número médio de comparações (%)				
	Simples	Menorfreq	KMP	BM	BMS
2	105.7	106.3	106.3	55.6	57.0
3	108.0	105.1	107.2	39.0	40.0
4	107.4	104.2	106.4	30.7	31.6
5	108.1	103.8	106.9	25.3	26.0
6	108.5	102.8	107.2	21.6	22.2
7	108.2	102.8	106.6	19.4	19.9
8	108.4	102.5	107.0	17.5	18.0
9	108.6	102.2	106.9	15.8	16.2
10	108.3	102.2	107.1	14.8	15.2
11	108.2	101.8	106.8	13.9	14.4
12	107.7	101.6	106.3	13.0	13.4
13	108.5	101.6	106.9	12.4	12.8
14	108.1	101.4	106.7	11.6	12.0
15	108.3	101.3	106.8	11.3	11.7

comprimento das cadeias	ALFABETO CENTENÁRIO - 20.000 caracteres			
	número médio de comparações (%)			
	Simples	KMP	BM	BMS
2	100.9	100.9	50.8	51.0
3	100.9	101.0	34.1	34.2
4	100.9	100.9	30.8	25.8
5	100.9	101.0	20.7	20.8
6	100.9	100.9	17.5	17.5
7	100.9	101.0	15.1	15.1
8	100.9	101.0	13.4	13.4
9	100.9	101.0	12.0	12.0
10	100.9	100.9	11.0	11.0
11	100.9	101.0	10.1	10.1
12	100.9	100.9	9.4	9.4
13	100.9	101.0	8.8	8.8
14	100.9	100.9	8.3	8.3
15	100.9	100.9	7.8	7.8

A medida de eficiência adotada indica que tanto melhor é o método, quanto menor é o número de referências feitas ao texto por caractere passado. Lembramos que, com exceção dos algoritmos de Boyer e Moore, todos os demais inspecionam pelo menos uma vez cada caractere do texto.

Para uma análise mais clara, expressamos graficamente os resultados obtidos:

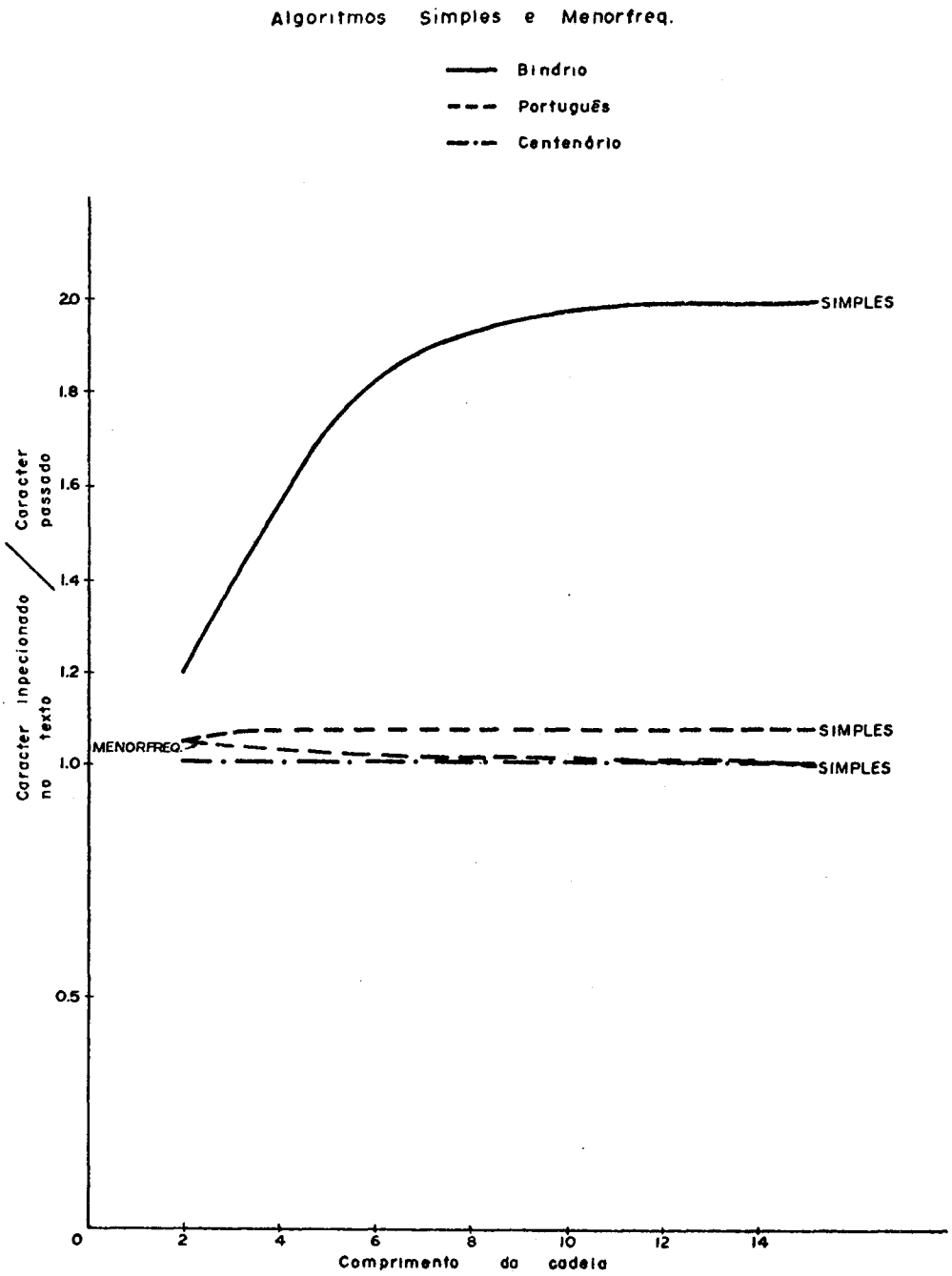


figura 1

A *figura 1* mostra o comportamento do algoritmo *Simples* para os três alfabetos e o algoritmo *Menorfreq* para o alfabeto Português. Quanto a este último, seu comportamento é tanto melhor que o algoritmo *Simples*, quanto maior o comprimento da cadeia procurada. Isto se deve ao fato de que em cadeias mais longas, há maior probabilidade de ocorrerem caracteres pouco frequentes. Por exemplo, para cadeias de comprimento 6, enquanto o algoritmo *Simples* inspeciona, em média, aproximadamente 1.08 caracteres por caractere passado, o algoritmo *Menorfreq* inspeciona aproximadamente 1.02 caracteres.

Quanto ao algoritmo *Simples*, em relação aos demais alfabetos, observa-se que o número de inspeções no texto Centenário por caractere passado, é constante em média, e aproximadamente igual ao número mínimo permitido pelo algoritmo. Contudo, com o texto Binário, este número cresce juntamente com o tamanho da cadeia e, para longas cadeias, o algoritmo inspeciona, em média, aproximadamente 2 caracteres por caractere passado no texto.

Analisando os resultados do algoritmo *KMP* *(figura 2)* sobre os diferentes alfabetos, verificamos que quanto maior o alfabeto, melhor o comportamento do algoritmo. Além disso, notamos que o efeito das modificações introduzidas por este método no algoritmo *Simples*, foi positivo apenas em relação ao alfabeto Binário, sendo que, em média, para cada comparação realizada por *KMP*, o algoritmo *Simples* realiza aproximadamente 1.46 comparações. Por outro lado, o experimento sobre o alfabeto Centenário obteve o melhor desempenho do algoritmo: em média, cada caractere do texto foi inspecionado uma única vez. Observamos ainda a não influência do tamanho da cadeia procurada no número de inspeções realizadas (apenas para cadeias muito curtas - 2,3 caracteres - este

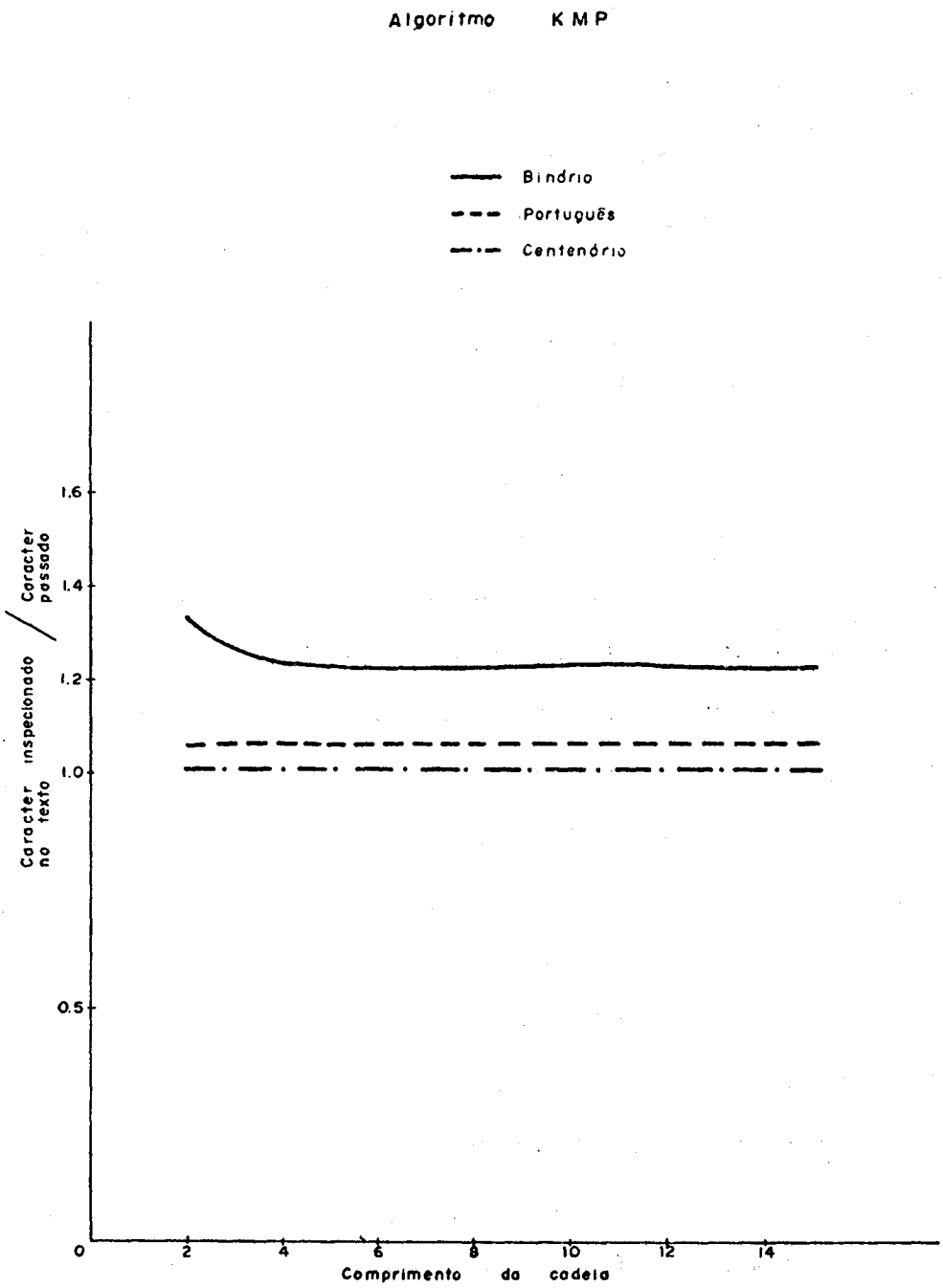


figura 2

número é mais elevado devido à maior frequência dessas cadeias no texto).

Observamos, a partir da *figura 3* que os algoritmos BM e BMS, de Boyer e Moore, possuem um comportamento médio seme-

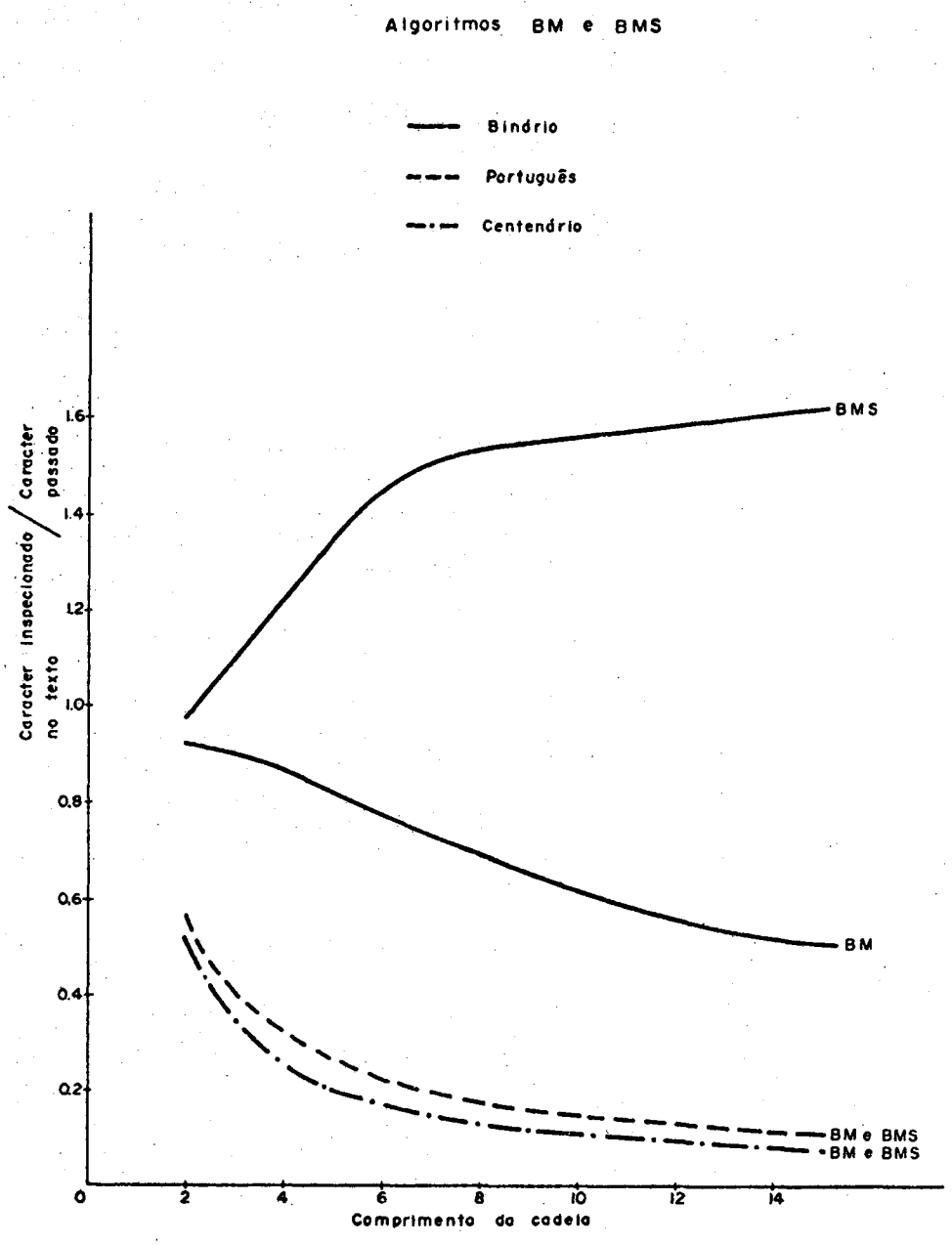


figura 3

lhante quando os alfabetos são de tamanho médio ou grande - Português e Centenário - mas *BM* é muito superior a *BMS* quando usamos um alfabeto pequeno - Binário. Esta conclusão é apenas uma

confirmação de nossas previsões pois, *BMS* distingue-se de *BM* pelo fato de não usar a tabela *delta2* e esta, por sua vez, é dedicada ao tratamento de cadeias que possuem subcadeias repetitivas. Naturalmente, quanto menor o alfabeto e mais longas as cadeias, mais repetitivas serão as cadeias geradas. O fator *BMS/BM* é em média, aproximadamente 1.2 para cadeias curtas (2,3,4 caracteres) e cresce para 2.98 para cadeias longas (13,14,15 caracteres), ou seja, *BMS* realiza cerca de 3 vezes mais comparações que *BM* quando as cadeias são longas.

Outro fato importante a ser notado é a influência do crescimento das cadeias no número de inspeções realizadas no texto. Como já havíamos previsto durante a análise do algoritmo *BM*, quanto maior a cadeia, maior a probabilidade de ocorrer grandes "saltos" sobre o texto. Isto se deve ao fato de que a tabela *delta1* fornece a distância entre a ocorrência mais a direita na cadeia do caractere do texto que causou a falha e o último caractere da cadeia; portanto, função do comprimento da cadeia.

Os gráficos a seguir, mostram, para cada alfabeto, o comportamento dos algoritmos *Simples*, *KMP* e *BM* (deixamos de lado os algoritmos *Menorfreq* e *BMS* por já terem sido comparados com suas respectivas versões *Simples* e *BM*).

As curvas da *figura 4* mostram os comportamentos dos correspondentes algoritmos com relação ao alfabeto Binário.

O preço que se paga por escolher o algoritmo mais eficiente - *BM* - é o pré-processamento das tabelas *delta1* e *delta2*, que são construídas para toda cadeia que se deseja encontrar. Por outro lado, o algoritmo *Simples*, que não exige qualquer informação adicional, pode ser muito ineficiente quanto ao tempo de execução, principalmente no caso de cadeias longas. Para cadeias mui-

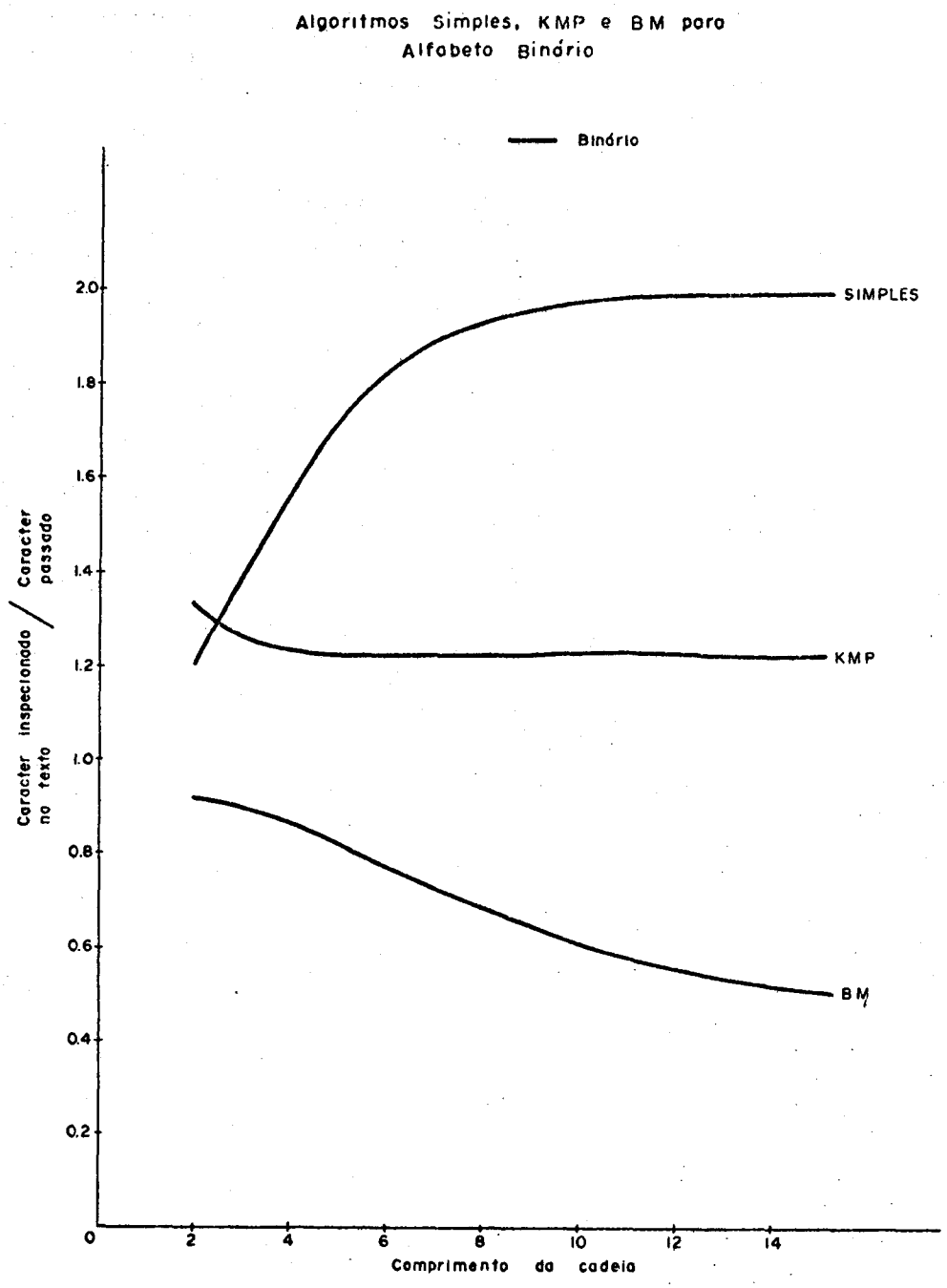


figura 4

to curtas (2 caracteres), este algoritmo inspeciona, em média, 1.2 caracteres por caractere passado, sendo que KMP e BM inspecionam, em média, 1.3 e 0.9 caracteres por caractere passado respectivamente. Tendo em vista a inferioridade, neste caso espe-

cial, de *KMP* em relação a *Simples* e a pouca diferença entre este e *BM*, considerando o trabalho extra de pré-processar as tabelas necessárias, para cadeias de comprimento menor que 3, o algoritmo *Simples* pode ser a melhor escolha.

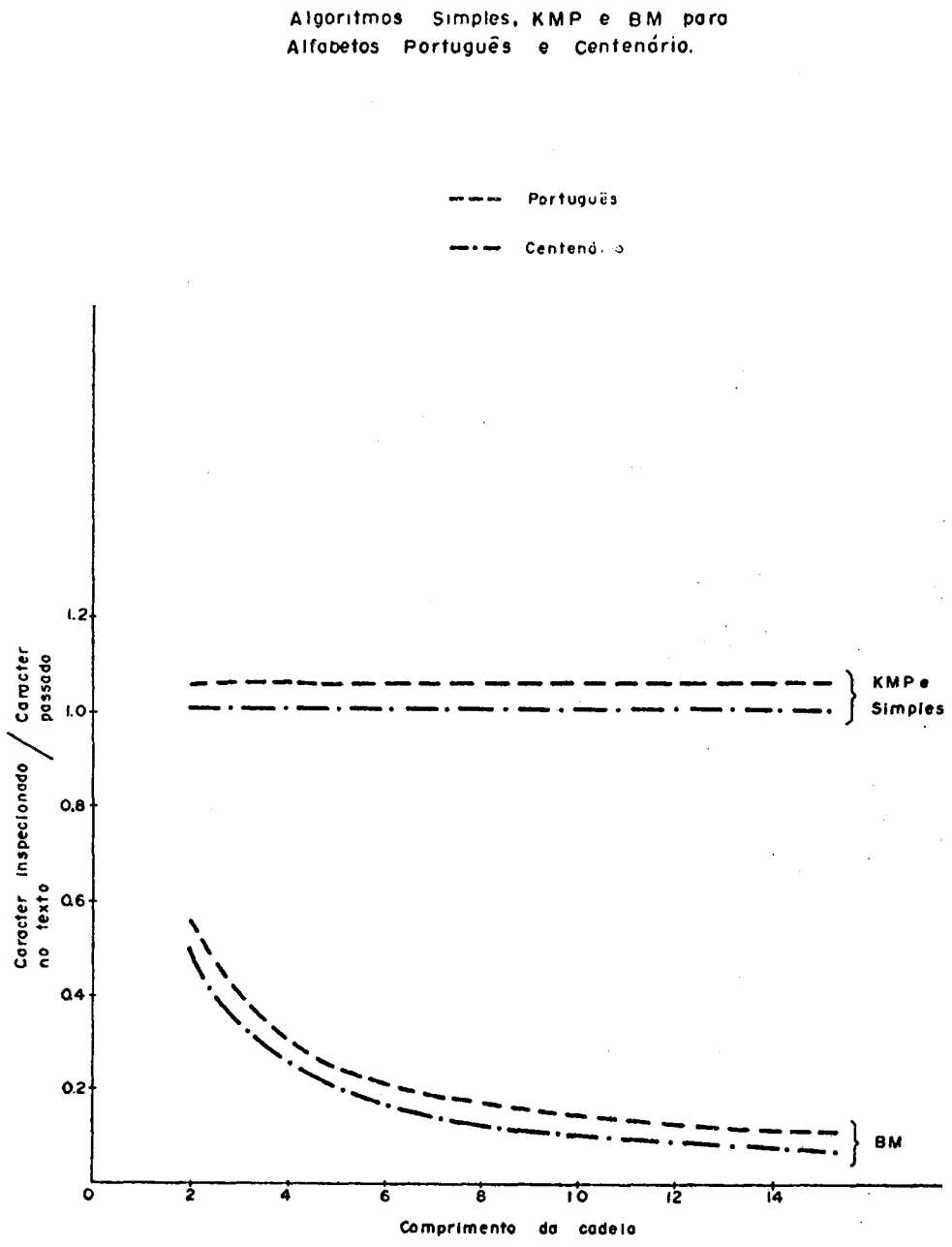


figura 5

A *figura 5* mostra as curvas correspondentes aos al-

goritmos quando os alfabetos Português e Centenário são utilizados.

Ressaltamos novamente, o comportamento semelhante de *KMP* e *Simples* com relação a esses alfabetos. O fator *KMP-Simples/BM* é em média, aproximadamente 4.9 usando-se o alfabeto Português e 5.6 usando-se um alfabeto Centenário.

Sem dúvida, para tais alfabetos, o método mais eficiente é o de Boyer e Moore e neste caso, o algoritmo *BMS* deve ser escolhido por ser mais simples que *BM* e comportar-se de maneira semelhante, em média. Mas, se não for possível utilizar este método devido ao pré-processamento da(s) tabela(s), a escolha deve recair sobre o algoritmo *Simples*, cujo comportamento é, em média, semelhante ao de *KMP* e possui a vantagem de não utilizar qualquer informação extra para a busca.

Capítulo 4

4.0 - Introdução

Os algoritmos apresentados nos capítulos anteriores, em particular os algoritmos *KMP*, *BM* e *BMS*, podem ser adaptados com o objetivo de

(a) encontrar todas as ocorrências de uma cadeia no texto;

(b) encontrar duas ou mais cadeias em sequência, detetando uma ocorrência da primeira seguida pela segunda, etc.;

(c) encontrar duas ou mais cadeias em paralelo, parando tão logo uma delas seja encontrada;

(d) encontrar todas as ocorrências de um conjunto finito de cadeias.

Neste último caso, o pré-processamento das cadeias e a manipulação das tabelas envolvidas tornam inviáveis os algoritmos citados.

O algoritmo que apresentamos neste capítulo, baseado em máquinas de estado finito, realiza a tarefa (d) de uma maneira simples e eficiente. Observamos desde já que este algoritmo é uma generalização do Algoritmo *KMP*.

4.1 - Algoritmo de Aho-Corasick

A máquina reconhecedora de cadeias de Aho-Corasick (*MRC*), como ficou conhecida a técnica apresentada em [AHO-75], é um autômato finito que localiza simultaneamente todas as ocorrências de um conjunto de cadeias num texto.

Esta técnica tem sido usada com sucesso como método

de acesso em sistemas de recuperação de informações, em particular em pesquisas bibliográficas, bem como em vários outros problemas de processamento de texto.

O algoritmo para *MRC* consiste de duas partes principais; um algoritmo para construir uma *MRC* a partir de um conjunto de cadeias e um algoritmo que introduz o texto como entrada para a *MRC*, que acusará, sempre que houver, toda a ocorrência de cadeias do conjunto.

Seja $K = \{y_1, y_2, \ldots y_k\}$ um conjunto finito de cadeias. Nosso problema é localizar e identificar todas as subcadeias do texto que são cadeias em K. Subcadeias podem sobrepor-se umas às outras.

Uma *MRC* para K é um programa que toma como entrada o texto e produz como saída as localizações no texto onde cadeias de K aparecem como subcadeias.

Uma *MRC* consiste de um conjunto de estados onde cada estado é representado por um número. A máquina processa o texto lendo sucessivamente os símbolos do texto, fazendo transições de estados e ocasionalmente emitindo saídas. O comportamento de *MRC* é ditado por 3 funções:

uma função de transição g,

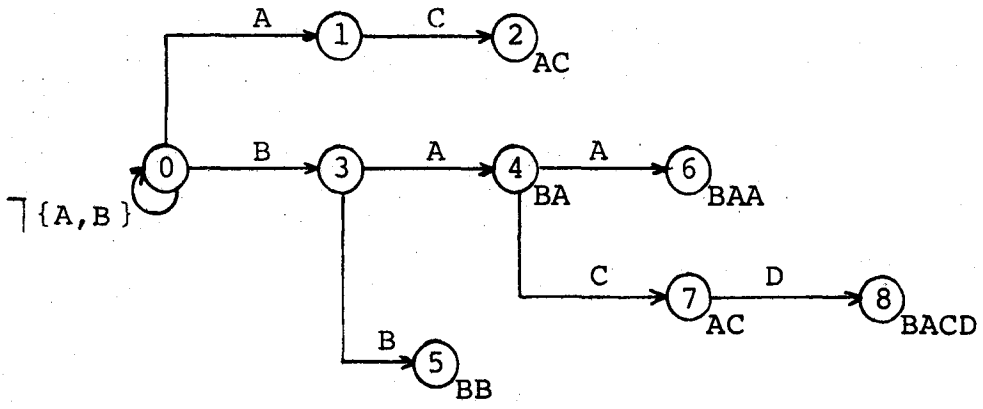
uma função de falha f,

uma função de saída s.

4.1.1 - Exemplo

A seguir, mostramos um exemplo de *MRC* para o conjunto K = {AC, BA, BB, BAA, BACD}.

(*a*) função de transição *g*



(*b*) função de falha *f*

i	1	2	3	4	5	6	7	8
f(i)	0	0	0	1	3	1	2	0

(c) função de saída *s*

i	*s*(i)
2	{AC}
4	{BA}
5	{BB}
6	{BAA}
7	{AC}
8	{BACD}

Um estado (geralmente 0) é designado como estado inicial. No exemplo, os estados são 0, 1, ... ,9. A função *g* mapeia

um par (estado, símbolo) num estado ou numa mensagem de falha. O dígrafo anterior representa a função de transição g. Por exemplo, a aresta rotulada 'A' de 0 a 1 indica que $g(0,A)=1$. A ausência de uma aresta indica falha. Assim $g(4,\sigma)$ = falha, para todo símbolo de entrada σ que não seja 'A' ou 'C'. A função g tem a propriedade de que $g(0,\sigma) \neq$ falha para todo símbolo σ. Veremos que esse fato assegurará que um único símbolo de entrada seja processado pela máquina em cada ciclo da máquina.

A função de falha f mapeia um estado em outro estado e ela é consultada sempre que a função g resulta em falha. Certos estados são designados como estados de saída, que indicam que um subconjunto de cadeias de K foi encontrado. A funçaõ de saída formaliza este conceito associando um conjunto (possivelmente vazio) de cadeias com cada estado.

Seja e o estado corrente da máquina e a o símbolo atual do texto de entrada. Um ciclo de operação de uma *MRC* é definido como:

1. Se $g(e,a)=e'$, a máquina faz uma transição g. Ela passa para o estado e' e o próximo símbolo do texto torna-se o símbolo atual. Ainda mais, se $s(e') \neq \{\}$ então *MRC* emite o conjunto $s(e')$.

2. Se $g(e,a)$ = falha, *MRC* consulta a função de falha f e é feita uma transição de falha. Se $f(e)=e'$, a máquina repete o ciclo com e' como estado e a como símbolo atuais.

No princípio, o estado atual de *MRC* é o estado inicial e o primeiro símbolo do texto é o símbolo atual. A máquina

então processa o texto fazendo um ciclo de operação sobre cada símbolo do texto.

Por exemplo, consideremos o comportamento de *MRC* que usa as funções do exemplo anterior para processar o texto 'CBAAC'.

Sequência de transições de estados:

símbolos	C	B	A	A	C	
estados	0	0	3	4	6	2
					1	

Quando e=4 e o símbolo atual é 'A', desde que $g(4,A)=6$ então *MRC* entra no estado 6, avança ao próximo símbolo de entrada e emite $s(6)$, indicando que encontrou a cadeia 'BAA' no final da posição 4 do texto.

Agora 'C' é o símbolo atual e desde que $g(6,C)$ = falha, *MRC* entra no estado $f(6)=1$. Como $g(1,C)=2$, *MRC* emite o conjunto $s(2)$ = {AC} como saída e termina sua operação pois encontrou o final do texto.

O Algoritmo 8 da página seguinte descreve o comportamento de uma máquina *MRC*.

Neste algoritmo, cada passo do laço **for** representa um ciclo de operação da máquina.

Mais adiante será apresentada uma versão do Algoritmo 8 que dispensa as transições de falha, onde um autômato finito determinístico é usado.

Observamos que o Algoritmo 8 inspeciona cada caractere do texto exatamente uma vez.

Algoritmo 8. *Máquina MRC*

Entrada : *texto*[1..*n*]; *n*>0, uma máquina MRC com funções *g*, *f* e *s* como descritas anteriormente.

Saída : localização do último caractere da cadeia em todas suas ocorrências em *texto*, para toda cadeia de K.

```
e←0;
for i←1 to n do
  begin
    while g(e, texto[i]) = falha do
          e←f(e);
    e←g(e, texto[i]);
    if s(e) ≠ { } then
       begin
         write(i);
         write(s(e))
       end
  end;
```

4.1.2 - *Construção das Funções de Transição, de Falha e de Saída*

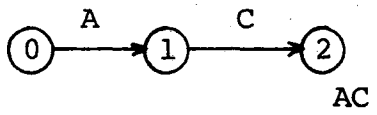
Dizemos que as três funções *g*, *f* e *s* são válidas para um conjunto de cadeias K se com essas funções o Algoritmo 8 indica que a cadeia y termina na posição i do texto se e somente se texto=uyv e o comprimento de uy é i.

Distinguimos duas fases na construção dessas fun-

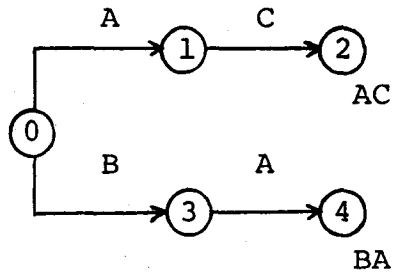
ções. Na primeira, determinamos os estados e a função de transição g. Na segunda, computamos a função de falha f. A construção da função de saída s inicia-se na primeira fase e é completada na segunda fase.

A construção da função g equivale à construção de um dígrafo onde cada vértice corresponde a um estado. Começamos com um único vértice que representa o estado inicial 0 e então, introduzimos cada cadeia y de K no grafo adicionando um caminho direcionado a partir do estado inicial. Novos vértices e arestas são introduzidos tal que haverá, a partir do estado inicial, um caminho que "soletra" a cadeia y. A cadeia y é adicionada à função saída do estado em que o caminho termina. Adicionamos novas arestas ao grafo apenas quando necessário.

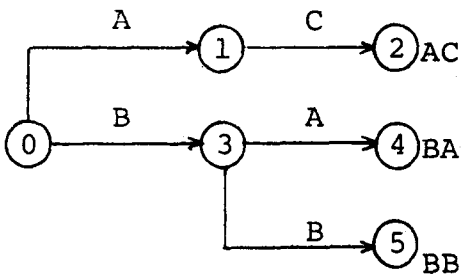
Por exemplo, suponha que {AC, BA, BB, BAA, BACD} é o conjunto de cadeias. Adicionando a primeira cadeia ao grafo, obtemos:



O caminho do estado 0 ao estado 2 "soletra" a cadeia 'AC'; associamos a saída 'AC' ao estado 2. Acrescentando a segunda cadeia 'BA' obtemos o grafo:

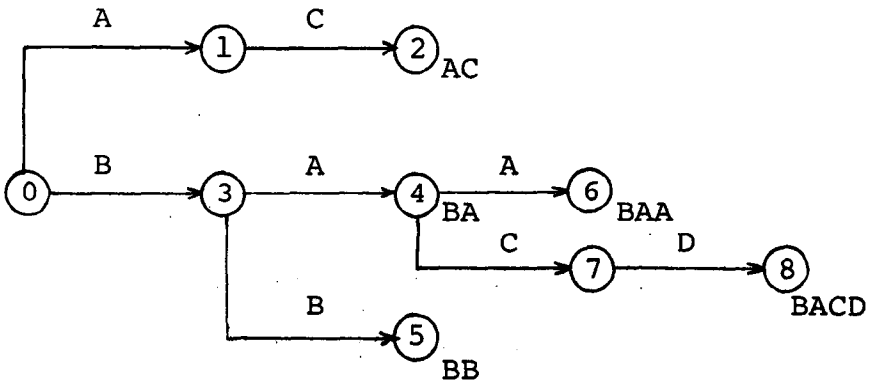


A saída 'BA' é associada ao estado 4. Acrescentando a cadeia 'BB' obtemos o seguinte grafo:



Note que quando acrescentamos 'BB' ao grafo, já há uma aresta rotulada 'B' do estado 0 ao estado 3, então não precisamos adicionar outra aresta 'B' a partir do estado 0. A saída 'BB' é associada ao estado 5.

Acrescentando as cadeias 'BAA' e 'BACD' obtemos:



Associamos a saída 'BAA' ao estado 6 e 'BACD' ao estado 8. Note que foi utilizado o caminho rotulado 'BA' para a construção dos dois novos caminhos.

Até este ponto, o grafo é uma árvore direcionada.

Para completar a construção da função g, adiciona

mos um laço do estado 0 ao estado 0 rotulado por todos os símbolos distintos daqueles que iniciam as cadeias de K. No exemplo, por todos os símbolos distintos de 'A' e 'B'.

Algorítmo 9. Construção da Função de Transição g e parte da Função de Saída s

Entrada : Conjunto de cadeias $K = \{y_1,\ldots,y_k\}$

Saída : Função g e parte da função s

Método : Assumiremos $s(e)=\{\}$ quando e é inicialmente criado, e $g(e,a)$ = falha se a é indefinido ou se $g(e,a)$ ainda não foi definido. O procedimento entra (y) insere no grafo um caminho que "soletra" $y = a_1a_2\ldots a_m$.

```
begin
   novoest ← 0;
   for i←1  to  k  do   entra(y_i);
   for all  a tal que g(0,a) = falha do
              g(0,a) ← 0;
end;
   procedure   entra(a_1a_2...a_m);
     begin
        e←0;   j←1;
     while g(e, a_j) ≠ falha do
        begin
            e ← g(e, a_j);
            j ← j+1
        end;
     for  p ← j  to  m  do
        begin
          novoest ← novoest + 1;
          g(e, a_p) ← novoest;
          e ← novoest
        end;
     s(e) ← {a_1a_2...a_m}
     end;
```

A função de falha f é construída a partir da função de transição g. Definimos a profundidade de um estado e no grafo que representa a função g, como sendo o comprimento do menor caminho desde o estado inicial até e. Assim, no exemplo anterior, o estado inicial 0 tem profundidade 0, os estados 1 e 3 têm profundidade 1, os estados 2, 4 e 5 têm profundidade 3 e assim por diante.

Computaremos a função de falha para todos os estados de profundidade 1 e então para todos de profundidade 2 e assim sucessivamente até que a função seja calculada para todos os estados (exceto para o estado 0 para o qual f não é definida). Fazemos $f(e)=0$ para todos os estados e de profundidade 1. Agora suponha que f foi calculada para todos os estados de profundidade menores que p. A função f para os estados de profundidade p é computada a partir dos valores de f para os estados de profundidades menores que p. Os estados de profundidade p podem ser determinados a partir dos valores válidos (distintos de falha) da função g dos estados de profundidade $p-1$ da seguinte maneira:

Para cada estado r de profundidade $p-1$,

1. Se $g(r,a)$ = falha para todo a, nada é feito.
2. Caso contrário, para cada símbolo a tal que $g(r,a)=e$ faça:
   (a) estado $\leftarrow f(r)$;
   (b) execute o comando estado $\leftarrow f$(estado) zero ou mais vezes até que um valor para estado seja obtido tal que g(estado,a) $\neq$ falha. (Desde que $g(0,a) \neq$ falha para todo a, tal valor sempre será encontrado).

(c) $f(e) \leftarrow g$ (estado, a).

Por exemplo, para computar a função f para o exemplo anterior, faríamos primeiramente $f(1)=f(3)=0$. Então calculamos f para os estados 2, 4 e 5 de profundidade 2. Para calcular $f(2)$ fazemos estado = $f(1)$ = 0 e desde que $g(0,C) \neq$ falha, achamos $f(2)=0$. Para calcular $f(4)$, fazemos estado = $f(3)=0$ e desde que $g(0,A)=1$, achamos $f(4)=1$. Para calcular $f(5)$, fazemos estado= $f(3)=0$ e desde que $g(0,B)=3$, fazemos $f(5)=3$. Continuando desta maneira, obtemos toda a função f.

Durante o cálculo de f também atualizamos a função de saída s. Quando determinamos $f(e)=e'$, intercalamos as saídas do estado e com as saídas do estado e'.

Por exemplo, temos que $f(7)=2$, então intercalamos o conjunto saída do estado 2, {AC}, com o conjunto saída do estado 7, { }, determinando o novo conjunto saída {AC} para o estado 7.

Na página seguinte, apresentamos o Algoritmo 10 que constrói a função de falha f.

A função de falha produzida pelo Algoritmo 10 não é ótima no seguinte sentido. Considere a máquina *MRC* do exemplo dado. Vemos que $g(4,A)=6$ e $g(4,C)=7$. Se *MRC* estiver no estado 4 e o símbolo atual do texto for distinto de 'A' e 'C', *MRC* entraria no estado $f(4)=1$. Desde que *MRC* já determinou que o símbolo atual é distinto de 'A' e 'C', então ela poderia passar diretamente de 4 para 0, pulando uma transição intermediária desnecessária para o estado 1 desde que a única transição existente para este estado prevê 'C' como símbolo atual. A mesma observação pode ser feita para a função de falha calculada para o estado 7.

Para evitar transições de falha desnecessárias pode

Algoritmo 10. Construção da Função de Falha f

Entrada : Funções g e s do Algoritmo 9.

Saída : Função f e função s.

```
fila ← vazia;
for  cada a tal que g(0,a) = e ≠ 0 do
  begin
    fila ← fila U {e};
    f(e) ← 0
  end;
while  fila ≠ vazia do
  begin
    Seja r o próximo estado da fila;
    fila ← fila - {r}
    for  cada a tal que g(r,a) = e  ≠  falha do
       begin
         fila ← fila U {e};
         estado ← f(r);
         while  g(estado,a) = falha do
                estado ← f(estado);
         f(e) ← g(estado,a);
         s(e) ← s(e) U s(f(e))
       end
  end;
```

mos usar f', uma generalização da função *prox* de [KNUTH-77], no lugar de f no Algoritmo 8. Especificamente, definimos $f'(1)=0$. Para $i>1$, $f'(i) = f'(f(i))$ se, para todo símbolo de entrada a,

$g(f(i), a) \neq$ falha implica $g(i, a) \neq$ falha; $f'(i) = f(i)$, caso contrário. Entretanto, mais adiante apresentamos uma versão do Algoritmo 8 que utiliza um autômato finito determinístico onde toda transição de falha é evitada.

4.1.3 - *Complexidade dos Algoritmos*

Veremos a seguir que, usando as funções f, g e s criadas pelos Algoritmos 9 e 10, o número de transições de estados feitas pelo Algoritmo 8 ao processar um texto é independente do número de cadeias do conjunto K. Veremos também que os Algoritmos 9 e 10 podem ser implementados para serem executados em tempo linearmente proporcional à soma dos comprimentos das cadeias de K [AHO-75].

Teorema 1. Usando as funções f, g e s criadas pelos Algoritmos 9 e 10, o Algoritmo 8 faz menos que $2n$ transições de estado ao processar um texto de comprimento n.

Prova: Em cada ciclo de operação, o Algoritmo 8 faz zero ou mais transições de falha seguidas por exatamente uma transição "válida". A partir de um estado e de profundidade p, o Algoritmo 8 nunca pode fazer mais do que p transições de falha em um ciclo de operação. Assim, o número total de transições de falha deve ser pelo menos um a menos que o número total de transições "válidas". Ao processar um texto de comprimento n, o Algoritmo 8 realiza exatamente n transições "válidas". Portanto, o número total de transições de estado é menor que $2n$.

Teorema 2. Algoritmo 9 requer tempo linearmente proporcional à soma dos comprimentos das cadeias do conjunto $K = \{y_1, y_2, \ldots, y_k\}$.

Prova. O tempo gasto pelo Algoritmo 9 é praticamente devido à execução dos k passos necessários para introduzir as cadeias de K. Desde que cada passo i introduz (ou percorre, caso já exista) um caminho, a partir do estado inicial, que "soletra" a cadeia y_i (portanto de comprimento igual ao da cadeia) e têm-se tantos passos quanto o número de cadeias, então o tempo do Algoritmo 9 é proporcional à soma dos comprimentos das cadeias.

Teorema 3. Algoritmo 10 pode ser implementado para executar em tempo proporcional à soma dos comprimentos das cadeias do conjunto $K = \{y_1, y_2, \ldots y_k\}$.

Prova. Usando um argumento similar àquele do Teorema 1, pode-se mostrar que o número total de execuções do comando estado $\leftarrow f$(estado) no Algoritmo 10 é limitado pela soma dos comprimentos das cadeias. Usando listas encadeadas para representar o conjunto saída de um estado, podemos executar o comando $s(e) \leftarrow s(e) \cup s(f(e))$ em tempo constante. Note que $s(e)$ e $s(f(e))$ são disjuntos quando este comando é executado. Assim, o tempo total necessário para implementar o Algoritmo 10 é dominado pela soma dos comprimentos das cadeias.

4.1.4 - Eliminação da Função de Falha

Agora veremos como eliminar todas as transições de falha do Algoritmo 8, construindo um autômato finito determinístico.

Um autômato finito determinístico consiste de um conjunto finito de estados E e uma função de movimento *prox*, tal que para cada estado *e* e símbolo de entrada *a*, *prox*(*e*,*a*) é um estado de E; isto é, o autômato faz exatamente uma transição de estado sobre cada símbolo de entrada.

Usando a função de movimento *prox* de um autômato finito determinístico apropriado no lugar da função de transição *g* do Algoritmo 8, podemos eliminar todas as transições de falha. Isto pode ser feito simplesmente substituindo os dois primeiros comandos do laço **for** do Algoritmo 8 pelo comando único e ← *prox*(e, *texto*[i]). Usando *prox*, o Algoritmo 8 faz exatamente uma transição de estado por caractere de entrada.

Podemos calcular a função de movimento *prox* a partir das funções construídas pelos Algoritmos 9 e 10, usando o algoritmo a seguir.

Algoritmo 11. Construção de um Autômato Finito Determinístico
Entrada : Função *g* do Algoritmo 9 e função *f* do Algoritmo 10.
Saída : Função de movimento *prox*

```
fila ← vazia;
for  cada símbolo a  do
  begin
     prox(0,a) ← g(0,a);
     if g(0,a) ≠ 0 then fila ← fila U {g(0,a)}
  end;
while  fila ≠ vazia  do
  begin
     seja r o próximo estado da fila;
     fila ← fila - {r};
     for  cada símbolo a  do
      if g(r,a) = e ≠ falha  then
         begin
           fila ← fila U {e};
           prox(r,a) ← e
         end
      else prox(r,a) ← prox(f(r), a)
  end;
```

O Algoritmo 11 apenas pré-computa o resultado de toda sequência de possíveis transições de falha. O tempo gasto pelo Algoritmo 11 é linearmente proporcional ao tamanho do conjunto de cadeias, ou seja, ao número de cadeias que se deseja procurar. Na prática, o Algoritmo 11 poderia ser desenvolvido conjuntamente com o Algoritmo 10.

Os valores de *prox* para o autômato finito determinístico para as cadeias {AC, BA, BB, BAA, BACD}, a partir das funções *g* e *f* calculadas pelos Algoritmos 9 e 10 são mostrados a seguir, onde o ponto '.' indica todo caractere diferente dos que aparecem acima dele.

	símbolo texto	próximo estado
estado 0 :	A	1
	B	3
	.	0
estado 1 :	A	1
	B	3
	C	2
	.	0
estado 2 :	A	1
	B	3
	.	0
estado 3 :	A	4
	B	5
	.	0
estado 4 :	A	6
	B	3
	C	7
	.	0
estado 5 :	A	4
	B	5
	.	0

estado 6 :	A	1
	B	3
	C	2
	.	0
estado 7 :	A	1
	B	3
	D	8
	.	0
estado 8 :	A	1
	B	3
	.	0

Computando o tempo dos Algoritmos 9 e 10 juntamente com o Algoritmo 11, concluimos que o método apresentado é $O(n+$soma dos comprimentos das cadeias$)$.

O uso de um autômato finito determinístico no Algoritmo 8 pode reduzir potencialmente o número de transições de estados em cerca de 50%. Mas esta redução quase nunca ocorre na prática uma vez que em aplicações típicas, o Algoritmo 8 gastará a maior parte do tempo no estado 0, a partir do qual não existem transições de falha. Calcular a economia esperada é difícil pois não estão disponíveis definições de conjunto "médio" de cadeias e texto "médio" de entrada.

Aoe, Yamamoto e Shimada [AOE-84] apresentam uma versão modificada da máquina *MRC* onde são otimizados o tempo e o espaço utilizado por ela. Sua versão, *MRC'*, foi comparada com *MRC* num experimento onde foi usado um computador FACOM 230-38+. O programa que constroi a máquina foi escrito em Pascal e o programa de busca foi escrito em FASP, uma linguagem montadora do FACOM 230-38. O texto, constituído de uma bibliografia do livro "Principles

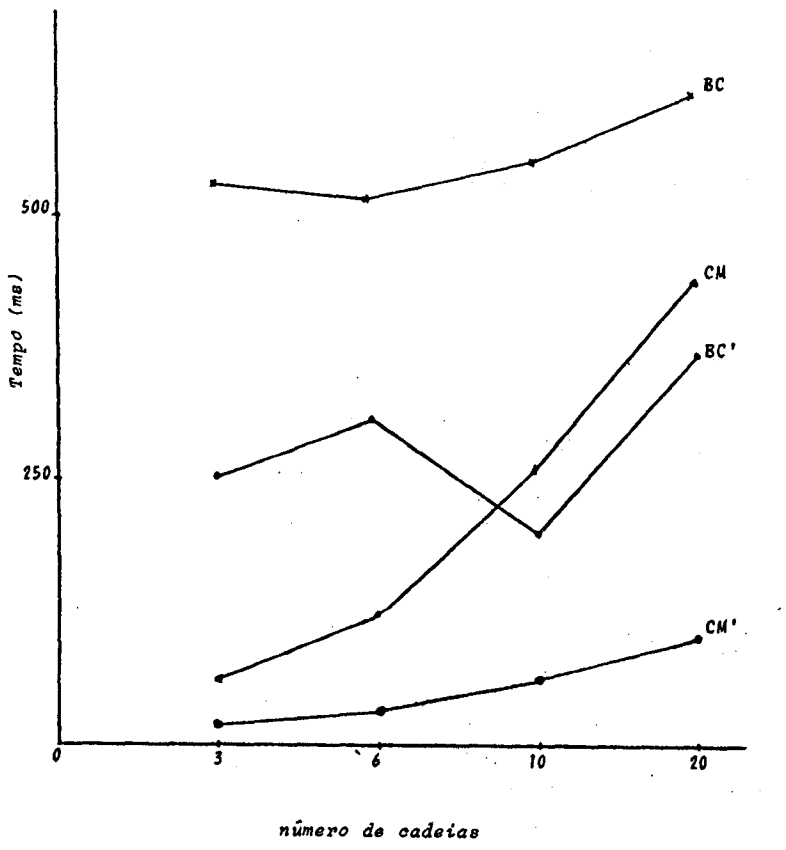
---

+ *FACOM 230-38 OSII/VS User's Manual, Fujitsu Co. Ltd.*

of Compiler Design" [AHO-78], continha 14.647 caracteres.

Os resultados quanto ao tempo de execução são expressos no gráfico a seguir que relaciona tempo (ms) e número de cadeias procuradas. CM(CM') representa o tempo gasto na construção da máquina MRC(MRC') e BC(BC') representa o tempo gasto na busca das cadeias através de MRC(MRC').

Tempos de Execução das máquinas MRC e MRC'



Observou-se então, que CM/CM' aumenta com o número de cadeias mas que BC/BC' é independente do número de cadeias e

este fator é, em média, aproximadamente 1.5.

A máquina *MRC* é também analisada por Arikawa [ARIKAWA-84] sob o ponto de vista de tempo de execução e espaço utilizado. Uma de suas versões otimizadas foi comparada com o algoritmo *BM* [BOYER-77] (o mais rápido para busca de uma única cadeia) adaptado para encontrar todas as ocorrências de um conjunto de cadeias. Sua versão mostrou-se mais rápida que *BM*, em média, para a busca de 6 ou mais cadeias. Em seu experimento, para um texto de 30.000 caracteres, na busca de um conjunto de 456 cadeias, o tempo total da versão otimizada de *MRC* chegou a ser cerca de 40 vezes menor que o tempo gasto pelo algoritmo *BM*, comprovando-se assim, a superioridade do método de Aho-Corasick para a realização de busca simultânea de muitas cadeias.

Capítulo 5

5.0 - Introdução

Considerando os algoritmos apresentados anteriormente, poderíamos melhorar o processo de busca em texto analisando exaustivamente uma parte do texto (por exemplo, uma linha) à procura da cadeia desejada, apenas quando houver uma grande probabilidade de se encontrar a cadeia neste trecho. Assim, veremos como um texto pode ser primeiramente condensado ou codificado (pré processado) de tal maneira que esta representação compacta seja pesquisada em lugar do texto.

Lembramos que todos os algoritmos apresentados anteriormente - *Simples*, *Menorfreq*, *KMP*, *BM*, *BMS* e *MRC* - realizam buscas sobre o texto original.

Um exemplo de pesquisa à uma forma condensada é o método de processamento de arquivos de dados que associa um descritor de w bits a cada registro (ou bloco de registros) do arquivo. Esses descritores podem então, ser combinados para derivar descritores que representem grupos maiores de registros. Na recuperação de informações, ao invés de procurar por cada registro individualmente, ou usar arquivos invertidos, o método pesquisa os descritores a fim de localizar prováveis registros que contenham a informação desejada. Apenas aqueles fortes candidatos são acessados e examinados em sua totalidade.

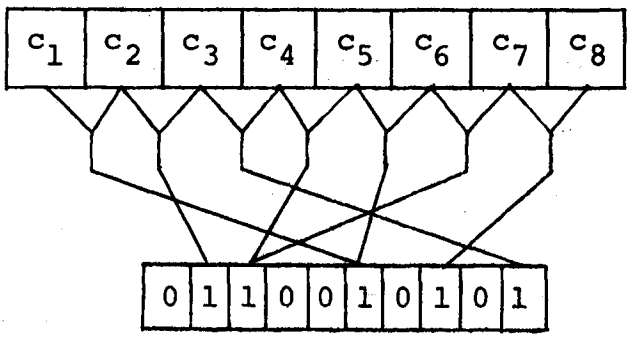
O método que apresentamos a seguir mostra uma maneira de condensar um texto num formato facilmente pesquisado.

5.1 - *O Método de Harrison*

O Método de Harrison [HARRISON-71] associa uma assinatura a cada segmento do texto (por exemplo, uma linha). Ao invés de percorrer cada caractere da linha, apenas a assinatura deve ser examinada. Somente quando o exame da assinatura sugerir uma provável correspondência na linha associada, é que esta linha será examinada caractere por caractere usando um dos algoritmos apresentados nos capítulos anteriores.

A informação contida em uma linha do texto é condensada num código de w bits que passa a ser a assinatura da linha.

O código é obtido aplicando-se uma função hash h a todos grupos consecutivos de k símbolos, produzindo um inteiro entre 0 e w-1. O bit correspondente ao inteiro produzido torna-se então igual a 1. Esta cadeia binária resultante, que contém um bit igual a 1 nas posições que correspondem a certas subcadeias de comprimento k, é chamada k-assinatura da linha do texto.



2-assinatura de $c_1c_2...c_8$ (w=10)

Assim, um conjunto S pode ser representado por uma cadeia binária $b_0b_1...b_{w-1}$ na qual um valor 1 para b_i indica que

S contém pelo menos um elemento do conjunto E_i. Em geral, tal representação não é única, a não ser que cada E_i contenha exatamente um elemento e cada possível elemento esteja contido em algum E_i. Entretanto, ela preserva a propriedade de subconjunto no sentido de que, se o conjunto S_1 é um subconjunto do conjunto S_2, a cadeia binária representando S_2 possuirá 1 em todas as posições onde a cadeia representando S_1 possuir 1.

Construimos a representação binária $b_0b_1...b_{w-1}$ para S da seguinte maneira:

1. Faça todos bits b_i iguais a zero.
2. Para cada subcadeia s de S de comprimento k, compute $i = h(s)$ e faça $b_i=1$.

Como exemplo, se tivéssemos o texto:

Cada palavra constitui-se de 16 bits.

Para k=2, então todos os pares de 2 símbolos seriam argumentos da função hash h, isto é, $h(Ca)$, $h(ad)$, $h(da)$, $h(a\not b)$,...,$h(s.)$, onde $\not b$ é o símbolo para um espaço. Para este exemplo de 37 símbolos, 36 combinações de pares de símbolos seriam codificadas para formar a 2-assinatura para esta linha.

O pré-processamento para constuir as assinaturas do texto é realizado uma única vez usando o procedimento anterior para toda linha do texto que possui então, associada a si, uma cadeia adicional de w bits.

O algoritmo a seguir pré-processa o texto original, construindo a k-assinatura de w bits associada a cada linha. (por exemplo, de 80 caracteres).

Algoritmo 12. Pré-Processamento do Texto

Entrada : *texto*, o vetor de caracteres que contém o texto original.

Saída : *saída* = *k*-assinatura || texto

```
repeat
   assin ← todos-zeros;
   for  i←1 to  80-k+1  do
     begin
       posic ← h(texto[i] ... texto[i+k-1]);
       assin[posic] ← 1
     end;
   saída ← assin || texto
until end-of-file;
```

onde assin é uma cadeia de *w* bits, todos-zeros é uma cadeia de *w* bits com todos iguais a zero e || denota concatenação.

A busca de cadeias é realizada usando as assinaturas do texto. Se, no texto do exemplo anterior, quiséssemos procurar pela palavra *bits*, o primeiro passo seria construir a assinatura da cadeia *bits* da mesma maneira como foi feita para o texto, isto é, $h(bi)$, $h(it)$, $h(ts)$.

O próximo passo é comparar a assinatura da cadeia com a assinatura de cada linha. Se uma assinatura do texto possui 1 em todas as posições que são iguais a 1 na assinatura da cadeia, então a linha correspondente do texto é pesquisada exaustivamente, uma vez que a coincidência desses bits é uma condição necessária, mas não suficiente, para que a cadeia ocorra naquela

linha. Se, por outro lado, a comparação das assinaturas falhar, então é certo que aquela linha não contém a cadeia e, portanto, não precisa ser pesquisada caractere por caractere.

O algoritmo a seguir complementa o Método de Harrison, procurando todas as ocorrências de uma cadeia *cad* num texto pré-processado pelo Algoritmo 12.

Algoritmo 13. Algoritmo de Harrison

Entrada : *cad*, $m>0$, e *texto* pré-processado pelo Algoritmo 12.

Saída : localização de todas ocorrências de *cad* no *texto*.

```
(* construir assinatura da cadeia *)
assin-cad ← todos-zeros;
for i←1 to m-k+1 do
   begin
     posic ← h(cad[i] ... cad[i+k-1]);
     assin-cad[posic] ← 1
   end ;
(* comparar assinatura da cadeia com assinaturas das linhas *)
repeat
   assin-texto ← assinatura de uma linha do texto;
   if (¬assin-texto) and assin-cad = todos-zeros then
      begin
        pesquisar a linha exaustivamente;
        if encontrou then sucesso
      end
until texto-esgotado;
```

5.1.1 - *A Escolha dos Parâmetros w e k*

É conveniente que w, o número de bits da assinatura, seja múltiplo do número de bits de uma palavra de memória, mas isto não é necessário. Observamos que, quanto maior w, mais precisos serão os resultados, no sentido de que menor número de cadeias serão identificadas incorretamente como subcadeias pelo teste de assinaturas. Em nossos experimentos, tomaremos w múltiplo de 8 (64 e 128).

O parâmetro k também pode ser escolhido. Se k=1, nenhuma informação sobre a posição do caractere na linha é incluída na assinatura, de maneira que k deve ser maior ou igual a 2. Se existem n possíveis símbolos nas cadeias, não faz sentido escolher k muito pequeno tal que n^k seja menor que w, desde que há apenas n^k subcadeias distintas de comprimento k, e "sobrarão" bits na assinatura. Por outro lado, se k for muito grande, o número de bits na assinatura pode tornar-se muito pequeno, e de fato será zero se k for maior que o comprimento da cadeia. O máximo conteúdo de informação corresponde a ter cerca de metade dos bits iguais a zero na assinatura. Geralmente escolhe-se k igual a 2 a fim de se procurar cadeias de comprimentos maiores ou iguais a 2.

5.1.2 - *A Escolha da Função Hash h*

A função hash descrita a seguir foi sugerida por Tharp e Tai [THARP-82] e assume k=2, w=64 (8 bytes) e 128 (16 bytes) e linhas com 80 caracteres.

Os símbolos do texto são divididos em classes baseadas na frequência de ocorrência desses símbolos no texto, tal que

as probabilidades de ocorrência são aproximadamente iguais para cada grupo. A Tabela I ilustra a distribuição dos caracteres em oito ou onze classes (a escolha do número de classes será explicada adiante), onde consideramos um texto em Português. O caractere ∅ (branco), devido sua alta probabilidade de ocorrência, é colocado sozinho em uma classe; os demais símbolos são então uniformemente distribuídos entre os outros grupos. Seja T um vetor tal que para todo possível caractere y, T[y] fornece a classe a qual pertence o caractere y.

Tabela Ia. Distribuição dos símbolos para assinatura de 8 bytes

Classe	Símbolos
0	∅ [14.36]++
1	E(11.1)+ F(0.84) 6 : & ' " ? [12.16]
2	A(10.29) G(0.84) J(0.22) W(0.1) X(0.31) 5 ; / * < [12.16]
3	O(8.16) Z(0.35) Q(0.95) B(0.66) Y(0.02) 4 ,) ! > ^ [12.08]
4	S(6.3) N(3.9) V(1.2) 3 (@ [- [12.61]
5	R(5.4) C(3.56) H(0.29) K(0.1) P(2.6) 2 9] ≠ [12.28]
6	I(4.65) D(4.1) U(3.1) 1 8 _ $ [12.18]
7	M(4.35) T(4.24) L(2.4) % + 0 7 \ . = [12.17]

Tabela Ib. Distribuição dos símbolos para assinatura de 16 bytes

Classe	Símbolos
0	∅ [14.36]++
1	E(11.1)+ 1 [11.4]
2	A(10.29) 2 : [10.58]
3	O(8.16) 3 + [8.37]
4	S(6.3) L(2.4) 4 & < [8.82]
5	R(5.4) P(2.6) 5 ' $ [8.06]
6	I(4.65) F(0.84) B(0.66) 6 ")] . [7.64]
7	M(4.35) U(3.1) Z(0.35) 7 ? ! ≠ = [8.02]
8	T(4.24) G(0.84) H(0.29) K(0.1) Q(0.95) 8 ; > - [7.11]
9	D(4.1) J(0.22) V(1.2) W(0.1) X(0.31) 9 / ^ [% , [7.53]
10	N(3.9) C(3.56) Y(0.02) 0 * (@ _ \ [8.11]

+ ( ) porcentagem de ocorrência para cada letra

++[ ] porcentagem total para os símbolos da classe

Para um par de símbolos y_1y_2, a função hash é

$$h(y_1y_2) = \text{número-de-classes} * T[y_1] + T[y_2]$$

O número-de-classes (para k=2) é escolhido como sendo o maior inteiro, n, tal que

$n^2 \leq w$ (onde w é o comprimento da assinatura)

Assim, o número-de-classes para uma assinatura de 64 bits é 8 e a função hash torna-se

$$h(y_1y_2) = 8 * T[y_1] + T[y_2]$$

Esta função mapeia todo par de símbolos num inteiro entre 0 e 63. Da maneira como foram distribuídos os símbolos em classes, todas as oito (0-7) classes têm probabilidades praticamente iguais de serem escolhidas. Portanto, a função h produz números no intervalo 0-63 com igual probabilidade sob o fato de que a ocorrência de um par de símbolos é baseada na distribuição das letras individuais. Neste caso, desde que $8^2 = 64$, todos os 64 bits da assinatura são usados eficientemente.

No caso da assinatura de 16 bytes, temos que $11^2 = 121 \leq w(128)$, logo

$$h(y_1y_2) = 11 * T[y_1] + T[y_2]$$

mapeia todo par de símbolos num inteiro entre 0 e 120 desde que existem onze (0-10) classes de símbolos. Neste caso, então, sete bits da assinatura ficam sem ser usados.

Se tomarmos k=3 e w=64 e 128 obtemos respectivamente

$$h(y_1y_2y_3) = 4^2 * T[y_1] + 4 * T[y_2] + T[y_3]$$

e

$$h(y_1y_2y_3) = 5^2 * T[y_1] + 5 * T[y_2] + T[y_3]$$

com os símbolos distribuídos em 4 e 5 classes de frequência. Lembramos que, neste caso, apenas cadeias de comprimento maior ou igual a 3 devem ser procuradas.

A seguir descrevemos os experimentos realizados e os resultados obtidos.

5.1.3 - *Resultados Experimentais*

Para avaliar o Método de Harrison, nossos experimentos foram realizados sobre um texto em Português cujo conteúdo é sobre Programação Estruturada possuindo 200 linhas com aproximadamente 80 caracteres cada. Todos os programas foram desenvolvidos sob o sistema Apple Pascal+.

Para testar a eficiência da função hash sugerida por Tharp e Tai que utiliza informações sobre frequência de caracteres no texto, como descrita anteriormente, escolhemos uma outra função hash muito simples que não prevê qualquer conhecimento do texto para ser utilizada. Esta função mais simples baseia-se na ordem dos caracteres da subcadeia de comprimento k no conjunto de caracteres adotado pelo compilador Pascal. O valor da função aplicada a k caracteres é então a somatória das ordens dos caracteres módulo o comprimento da assinatura.

+ *Apple Pascal - Operating System Reference Manual by APPLE COMPUTER INC. 1980.*

O Método de Harrison foi aplicado na busca de cadeias no texto, primeiramente utilizando-se a função hash de Tharp e Tai que chamaremos de h_1 e posteriormente a função hash mais simples que chamaremos de h_2. Para ambos os casos, consideramos k = 2 e 3 ou seja, assinaturas construídas a partir da função hash aplicada a subcadeias de 2 e 3 caracteres. Ainda variamos o tamanho da assinatura, w, fazendo w = 64 e 128 bits.

As duas tabelas abaixo mostram as funções utilizadas:

k	w	nc	*função* h_1
2	64	8	$h_1(y_1y_2) = nc*T[y_1] + T[y_2]$
	128	11	
3	64	4	$h_1(y_1y_2y_3) = nc^2*T[y_1] + nc*T[y_2] + T[y_3]$
	128	5	

onde nc é o número de classes de frequências, calculado a partir de w e k, e

k	w	*função* h_2
2	64	$h_2(y_1y_2) = (\mathrm{ord}(y_1) + \mathrm{ord}(y_2)) \bmod w$
	128	
3	64	$h_2(y_1y_2y_3) = (\mathrm{ord}(y_1) + \mathrm{ord}(y_2) + \mathrm{ord}(y_3)) \bmod w$
	128	

onde $\mathrm{ord}(y_1)$ é uma função da linguagem Pascal que fornece a or-

dem do caractere y_1 no conjunto de caracteres adotado pelo compilador Pascal (no caso, o conjunto ASCII).

Para o processo de busca, foram selecionadas aleatoriamente do texto original, cadeias de comprimento 2, 3,.., 15, num total de 50 cadeias de cada um desses comprimentos. No caso k=3, os comprimentos variaram entre 3 e 15.

Para cada grupo de 50 cadeias de cada comprimento, obtivemos o número médio de linhas pesquisadas exaustivamente (ou seja, linhas cujas assinaturas produziram sucesso no teste de assinaturas) e, dentre estas, o número médio de linhas em que a busca resultou em sucesso (ou seja, onde a cadeia foi encontrada).

As Tabelas II e III mostram esses números, em porcentagem, para k=2 e k=3 respectivamente.

Tabela II

k=2	função h_1				função h_2			
	Linhas Pesquisadas (%)		Linhas com Sucesso (%)		Linhas Pesquisadas (%)		Linhas com Sucesso (%)	
Comprimento cadeias	w=64	w=128	w=64	w=128	w=64	w=128	w=64	w=128
2	74.1	57.7	57.2	70.7	79.6	73.6	54.5	58.9
3	58.9	40.8	22.4	30.2	53.3	44.3	18.0	21.7
4	44.9	24.6	11.3	23.1	50.5	41.1	11.5	12.3
5	34.9	15.8	8.5	11.9	39.3	32.5	5.9	5.9
6	28.7	8.7	4.5	14.5	29.8	25.9	6.6	6.1
7	21.7	7.7	5.0	16.3	27.1	19.1	4.0	7.4
8	17.6	4.9	5.5	18.3	21.0	18.0	6.2	4.1
9	13.7	3.9	5,2	22.2	17.7	11.9	3.4	6.5
10	11.1	3.2	4.8	26.7	15.0	9.1	4.6	9.3
11	8.5	2.0	7.0	34.1	10.3	8.7	7.2	7.3
12	6.4	2.0	8.8	28.0	10.2	7.5	6.0	8.2
13	5.4	1.1	9.8	47.3	8.8	5.3	6.9	10.9
14	4.1	1.2	12.6	45.3	7.0	3.2	7.6	16.6
15	3.8	0.8	13.3	62.9	5.7	3.8	9.5	14.7

A Tabela II mostra, por exemplo, que para cadeias de comprimento 8, em média, 17.6% das linhas do texto foram pesqui-

sadas exaustivamente quando h_1 é usada para uma assinatura de 64 bits. Já, 5.5% do total de linhas pesquisadas continham realmente a cadeia procurada; 94.5% das linhas pesquisadas não continham a cadeia. Para a mesma função h_1, com uma assinatura de 128 bits, os resultados foram melhores pois menos linhas foram pesquisadas (4.9%) e, dentre estas, uma porcentagem maior resultou em sucesso (18,3%).

Tabela III

k=3	função h_1				função h_2			
	Linhas Pesquisadas (%)		Linhas com Sucesso (%)		Linhas Pesquisadas (%)		Linhas com Sucesso (%)	
Comprimento Cadeias	w=64	w=128	w=64	w=128	w=64	w=128	w=64	w=128
3	70.5	52.1	17.4	23.5	75.3	60.5	16.3	20.3
4	55.4	30.3	10.0	18.2	48.9	35.0	11.3	15.8
5	45.2	18.0	4.2	10.7	37.6	23.9	5.1	8.0
6	30.2	11.3	4.0	10.8	28.6	14.6	4.2	8.4
7	24.2	7.0	5.6	19.4	21.1	10.9	6.4	12.4
8	20.5	4.2	4.7	22.7	18.6	6.6	5.2	14.5
9	14.0	4.1	8.2	28.1	11.8	5.5	9.7	20.9
10	13.1	2.7	6.7	32.9	8.9	2.9	9.9	30.1
11	9.6	1.8	6.0	31.6	9.0	2.6	6.4	22.1
12	7.5	1.6	8.1	37.5	6.5	2.0	9.5	29.9
13	5.6	1.1	9.1	47.2	4.1	1.2	12.5	40.6
14	4.8	0.9	11.0	55.2	4.0	1.0	12.9	53.0
15	3.6	1.0	15.0	50.9	2.7	0.9	20.3	59.7

Apresentamos a seguir, os gráficos que relacionam, para as funções h_1 e h_2 com assinaturas de 64 e 128 bits, as linhas pesquisadas (%) e as linhas com sucesso (%) com o crescimento do tamanho das cadeias pesquisadas.

Quanto às medidas de eficiência adotadas, podemos adiantar que mais eficiente é o método quanto menor o número de linhas pesquisadas exaustivamente e como o número médio de li-

nhas com sucesso é relativo ao número médio de linhas pesquisadas, o ideal é que este número se aproxime de 100%, ou seja, apenas pesquisaríamos as linhas que contivessem realmente a cadeia.

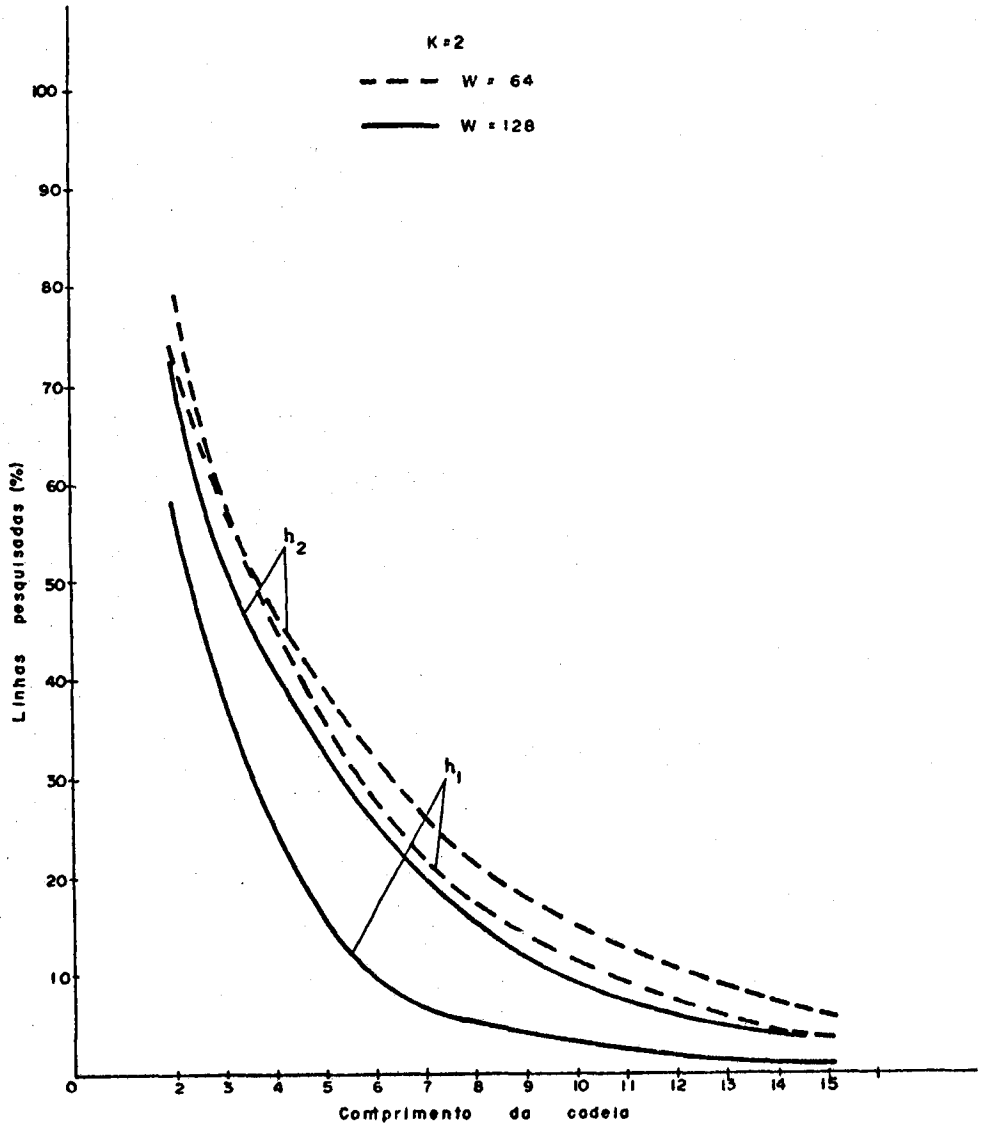


figura 1

Observando a *fig.1* vemos que 3 fatores influenciam o número de linhas pesquisadas: o comprimento da assinatura, w, a função hash aplicada e o comprimento da cadeia procurada. Embora utilize mais memória, a assinatura de 128 bits reduz o número de linhas pes-

quisadas tanto no caso h_1 como h_2. Esta diferença é mais acentuada no caso h_1 e diminui conforme aumenta o tamanho das cadeias e tem seu maior valor para cadeias de comprimento 6 no caso de h_1 e comprimento 3 no caso h_2. Como as medidas obtidas para h_1 são fortemente dependentes do texto e das cadeias, o comportamento das curvas conforme o comprimento das cadeias é o fato mais importante a ser observado. Notamos ainda a pouca diferença entre o comportamento de h_1 e h_2 para $w = 64$, sendo que, ao contrário de h_1, h_2 não necessita de qualquer informação sobre o texto.

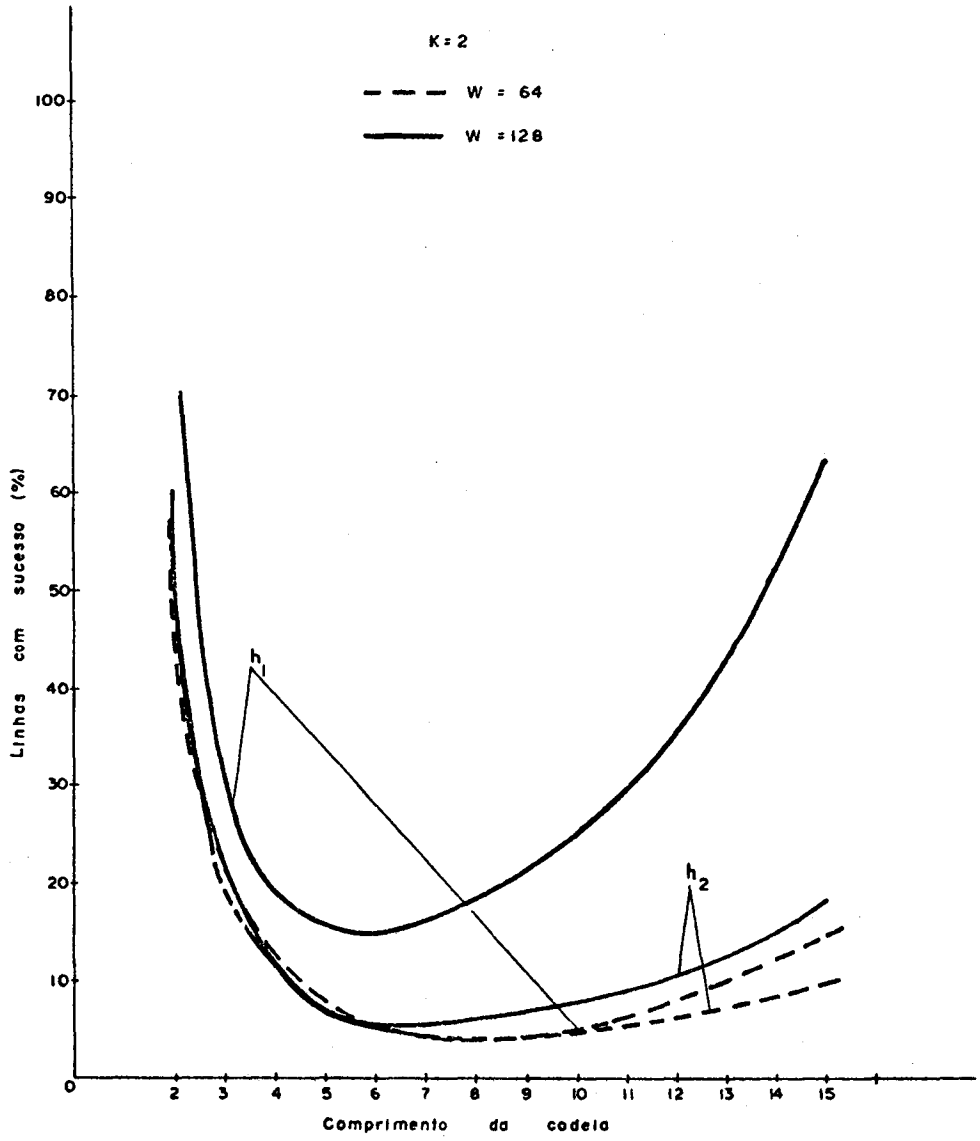


figura 2

O comportamento das curvas da *fig.2* nos indica que para cadeias muito curtas (comprimento 2) grande parte das linhas pesquisadas exaustivamente continham de fato a cadeia procurada. Este fato é bastante aceitável uma vez que cadeias de 2 caracteres tendem a se repetir com razoável frequência no texto. O número de sucessos diminui conforme aumenta o tamanho das cadeias pois o teste de assinaturas tem maior probabilidade de indicar falsas ocorrências. Isto acontece para cadeias de comprimento médio: 6,7,8 e conforme crescem as cadeias, estas tendem a ocorrer mais raramente e neste caso, o teste de assinaturas é mais exato, ocasionando um novo crescimento do número de linhas com sucesso. Lembramos que a *fig.2* não é independente da *fig.1*; de fato, os números são relativos à porcentagem de linhas pesquisadas e devem ser analisados conjuntamente para uma melhor análise. Assim para cadeias de 2 caracteres, assinatura de 128 bits, usando h_1 temos que 57.7% das linhas do texto (≃115) foram pesquisadas exaustivamente e 70.7% destas (≃81) continham a cadeia. Para cadeias mais longas estes números podem ser mais significantes: para a mesma função e assinatura, quando a cadeia possue 15 caracteres, cerca de 0.8% das linhas (≃2) são pesquisadas sendo que 62.9% destas (≃1) contêm a cadeia.

A *fig.2* reafirma a influência do comprimento da assinatura na eficiência do método e a superioridade da função h_1 sobre h_2 no caso $w = 128$ pois para $w = 64$ elas praticamente coincidem para cadeias de até 11 caracteres que, em geral, são as mais procuradas.

A seguir apresentamos os gráficos correspondentes à Tabela III, isto é, quando $k = 3$. Lembramos que neste caso, apenas cadeias contendo mais que 2 caracteres podem ser procuradas.

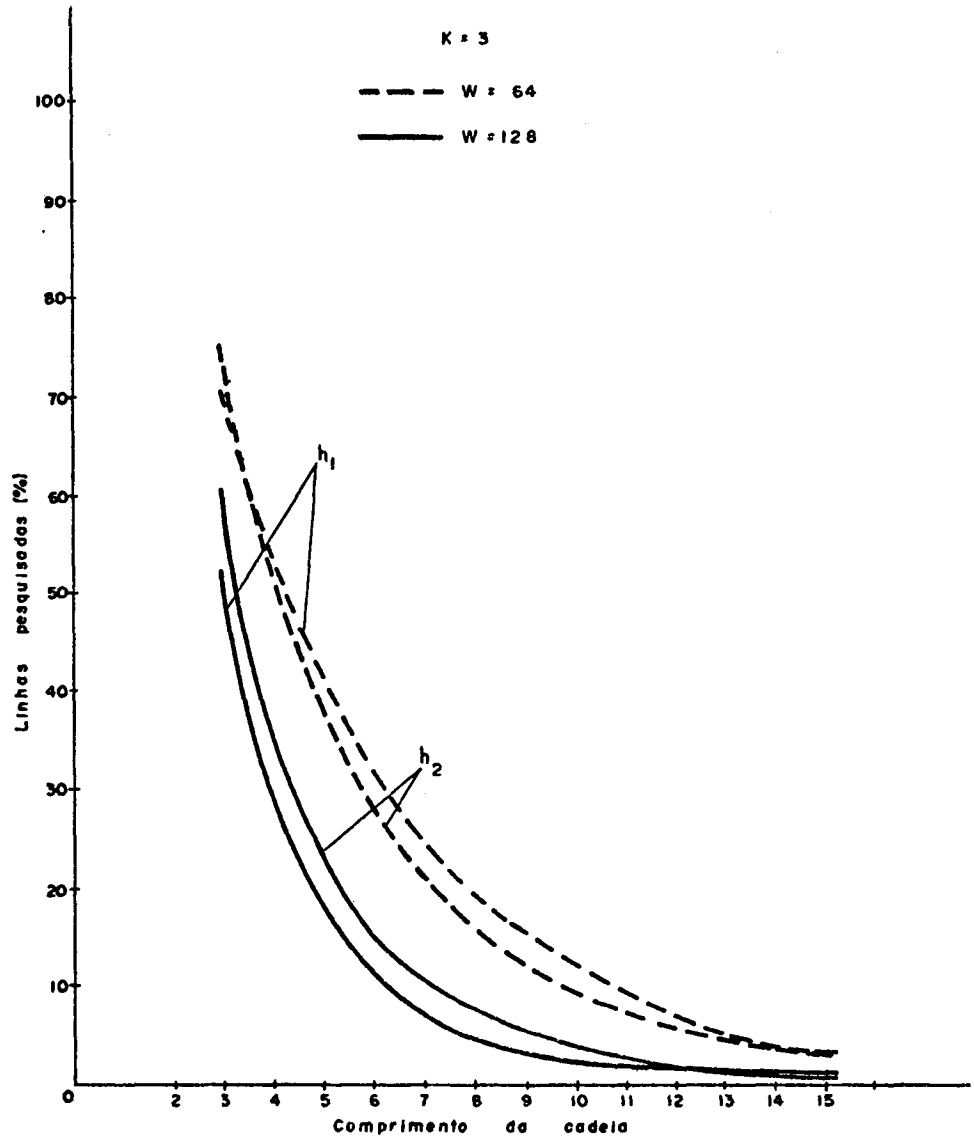


figura 3

Comparando a *fig.3* com a *fig.1*, observamos que o comportamento das curvas é o mesmo para ambos os casos. Ressaltamos, contudo, que devido à grande redução do número de classes de frequência para h_1, quando $k = 3$ (de 8 para 4 quando $w = 64$ e de 11 para 5 quando $w = 128$), aumentou a probabilidade de ocorrência de cada classe e portanto, mais falsas ocorrências da cadeia são detetadas durante o teste de assinaturas. Em particular, para $w = 64$, a função mais simples h_2 seleciona, neste caso,

menor número de linhas para serem pesquisadas exaustivamente em relação a h_1. Além disso, a diferença entre h_1 e h_2 para $w = 128$ é menos acentuada na *fig.3*. Outro fato importante é o comportamento praticamente idêntico de h_1 e h_2, para ambos os comprimentos, quando as cadeias tornam-se longas.

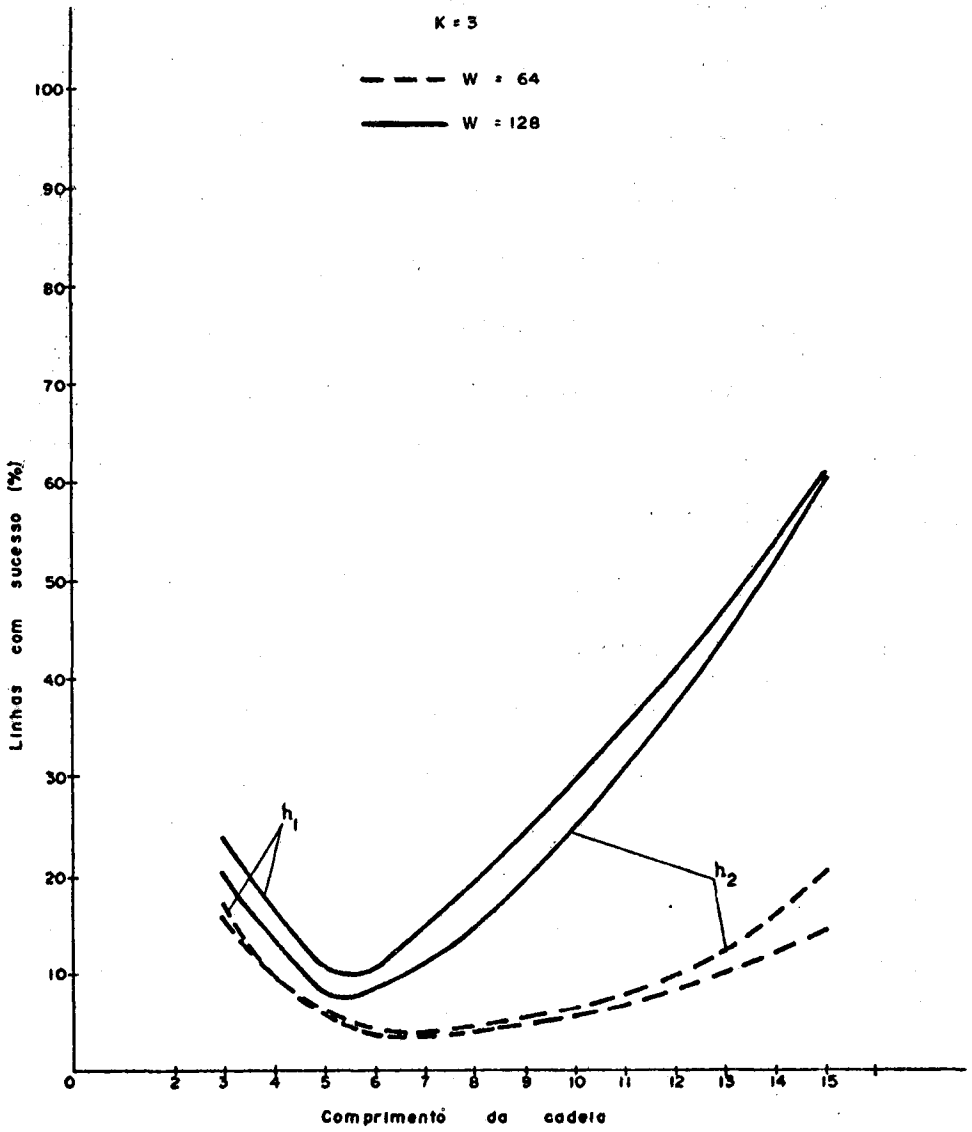


figura 4

Observa-se na *fig.4* uma significante melhoria no comportamento da função h_2, para ambos os comprimentos em relação ao caso $k = 2$. Juntamente com a *fig.3*, notamos que, em rela-

ção a h_1, h_2 para $w = 64$, pesquisou menor número de linhas e dentre estas, uma porcentagem maior continha a cadeia. Além disso, a diferença entre as duas funções, considerando-se o mesmo comprimento de assinatura, não é tão acentuada quanto no caso $k=2$. Novamente, este fato deve-se à redução do número de classes de frequência ocasionando um maior número de linhas pesquisadas.

Se considerarmos a simplicidade da função h_2 e os resultados apresentados nas figuras *3* e *4*, quando quisermos procurar por cadeias com mais de 2 caracteres e não tivermos qualquer informação sobre o texto, podemos optar pelo uso da função h_2, preferencialmente com assinaturas mais longas.

Contudo, quando se dispõe de informações sobre o texto para a construção de classes de frequência para a função h_1 e também de memória suficiente para o armazenamento das assinaturas, espera-se um melhor desempenho do Método de Harrison aplicando-se a função sugerida por Tharp e Tai com assinaturas longas, tanto para $k = 2$ quanto $k = 3$. Na impossibilidade de usar assinaturas longas, a opção pode recair sobre h_2, com $w = 64$, por ser uma função muito simples e comportar-se em média, de uma maneira muito semelhante a h_1 no caso $k = 2$ e superior a esta quando $k = 3$.

Capítulo 6

Conclusões

Nos capítulos anteriores apresentamos os principais métodos para busca de cadeias de caracteres em textos, bem como os resultados experimentais obtidos.

Pretendemos, neste ponto, fornecer uma visão global dos resultados obtidos, no sentido de indicar o método mais adequado para cada um dos casos que julgamos relevantes.

Um fator importante para a escolha do método é o tipo do texto que será pesquisado. Entre as características de um texto que mais afetam a decisão sobre os métodos, citamos: sua classificação quanto à frequência de alterações realizadas (estático/dinâmico); seu tamanho (número de caracteres) e tamanho do alfabeto sobre o qual o texto foi criado. Além disso, informações prévias sobre o texto, como por exemplo, a frequência de cada caractere no texto são, às vezes, fundamentais para uma escolha adequada do método.

A tabela da página seguinte mostra a escolha do algoritmo de acordo com as características do texto e da busca.

Assim, se o texto for pequeno, por exemplo quando usamos editores de texto, o algoritmo *Simples* é o indicado uma vez que não exige qualquer pré-processamento das cadeias procuradas. Caso seja conhecida a frequência de cada caractere no texto, usamos o algoritmo *Menorfreq* que é mais eficiente quanto maior a cadeia procurada. Para textos grandes e alfabetos pequenos, o algoritmo *Simples* pode ser muito lento devido ao grande número de comparações realizadas entre caracteres.

Algoritmos	texto dinâmico				texto estático	
	texto pequeno	texto grande			com informações prévias sobre o texto	sem informações prévias sobre o texto
		com pré-processamento das cadeias	busca simultânea de até 6 cadeias	busca simultânea de mais de 6 cadeias		
Simples	X					
Menorfreq	X (com informações sobre o texto)					
KMP						
BM		X (alfabetos pequenos)	X (adaptado para várias cadeias)			
BMS		X (alfabetos médios e grandes)				
MRC				X		
M.Harrison (h_1)					X	
M.Harrison (h_2)						X

Para textos grandes e dinâmicos, ou seja, textos que frequentemente sofrem alterações, o método proposto por Boyer e Moore [BOYER-77] mostrou-se o mais apropriado, qualquer que seja o tamanho do alfabeto, sendo que quanto maior o alfabeto, mais eficientemente ele se comporta. Este método exige um pré-processamento de cada cadeia procurada e sua versão simplificada, BMS, que usa apenas uma das duas tabelas originais, deve ser utilizada sempre que o tamanho do alfabeto não for muito pequeno. No caso de alfabetos muito pequenos, BM, que pré-processa duas tabelas, *delta1* e *delta2*, deve ser utilizado. Nas duas versões, quanto maior a cadeia procurada, menor é o número de comparações realizadas e consequentemente, mais rápida é a busca.

Se desejamos encontrar, simultaneamente, um conjunto razoavelmente grande de cadeias - acima de 6 cadeias, deve-

mos usar a máquina reconhecedora de cadeias - *MRC* - de Aho e Corasick [AHO-75], que pesquisa todo o texto em tempo linear em seu comprimento mais a soma dos comprimentos das cadeias. Uma versão otimizada do algoritmo *BM*, adaptada para várias buscas simultâneas, pode ser utilizada quando o conjunto de cadeias for pequeno.

Observamos que o algoritmo *KMP* [KNUTH-77], que também pré-processa a cadeia procurada, não é competitivo com o algorítmo de Boyer e Moore, uma vez que só é superior ao algoritmo *Simples* quando o alfabeto for muito pequeno. Mesmo assim, ele requer no mínimo, uma inspeção por caractere do texto, ao contrário de *BM* que faz, em média, menos que uma inspeção por caractere passado.

Se possuirmos um texto grande e razoavelmente estático (por exemplo, em pesquisas bibliográficas), podemos pensar em pré-processá-lo de modo a obter um texto codificado apropriadamente para uma busca mais eficiente. Neste caso, podemos recorrer ao Método de Harrison [HARRISON-71] e transformar nosso texto numa sequência de assinaturas. Desta maneira, apenas uma parte (algumas linhas) do texto será examinada, caractere por caractere a fim de se encontrar a cadeia procurada. Foram exibidos dois tipos de funções-assinatura sobre as quais comentamos a seguir.

Se pudermos construir assinaturas longas (16 bytes) e tivermos informações sobre a natureza do texto a fim de construir as classes de frequências necessárias à função h_1 sugerida por Tharp e Tai [THARP-82], então o Método de Harrison deve ser usado juntamente com tal função, para produzir bons resultados, como mostra a análise no *Capítulo 5*.

Caso não tenhamos informações sobre o texto, uma

função - assinatura mais simples, do tipo h_2, pode ser utilizada, sendo que, para assinaturas menores (8 bytes) seu comportamento é apenas um pouco inferior à h_1. Sob estas mesmas hipóteses, se desejamos encontrar apenas cadeias com mais de 3 caracteres (k=3), o uso de h_2 é preferível ao de h_1, uma vez que o número de classes de frequência diminui, prejudicando o comportamento desta função.

A partir das observações feitas podemos concluir que, se o texto a ser pesquisado for grande o suficiente tal que não seja viável transferí-lo totalmente à memória principal e pesquisá-lo exaustivamente e, além disso, estático de modo a permitir um pré-processamento não oneroso, o Método de Harrison pode ser utilizado para a busca de cadeias. A escolha da função-assinatura deve recair sobre h_1, de Tharp e Tai, se possuirmos as informações necessárias sobre o texto, ou então sobre uma função mais simples, do tipo h_2, caso nenhuma informação seja conhecida.

Se, por outro lado, o texto for pequeno suficiente para que possa ser "lido" totalmente na memória principal e/ou dinâmico de modo a não admitir um alto custo de frequentes pré-processamentos, a escolha deve recair sobre o método de Boyer e Moore.

Pesquisas recentes - [GONNET-83] e [KASHYAP-83] - têm revelado como um problema de interesse, a realização de busca com ruídos em grandes bases de dados sem estrutura (texto). Este tipo de busca é feito de tal maneira que o conjunto de registros (cadeias) obtido como resposta pode conter registros que não satisfazem exatamente a pergunta formulada, cabendo ao usuário fazer um pós-processamento da resposta. Nesse sentido, seria interessante pesquisar funções-assinatura mais adequadas - tal-

vez mais simples - ou mesmo outros tipos de métodos destinados a este tipo de problema.

Bibliografia

[AHO-75] - Aho, A.V., Corasick, M.J. - "Efficient String Matching: An Aid to Bibliographic Search" - Comm. of ACM - vol. 18 - nº 6 - pp. 333-340 - Jun. 75.

[AHO-78] - Aho, A.V., Ullman, J.D. - "Principles of Computer Design" - Addison-Wesley - 1978.

[AOE-84] - Aoe, J., Yamamoto, Y., Shimada, R. - "A Method for Improving String Pattern Matching Machines" - IEEE Transactions on Software Engineering - vol. SE-10, nº 1 - pp. 116-120 - Jan. 84.

[ARIKAWA-84] - Arikawa, S., Shinohara, T. - "A Run-Time Efficient Realization of Aho-Corasick Pattern Matching Machines" - New Generation Computing - nº 2 - pp. 171-186 - 1984.

[BOYER-77] - Boyer, R.S., Moore, J.S. - "A Fast String Searching Algorithm" - Comm. of ACM - vol. 20 - nº 10 - pp. 762-772 - Out. 77.

[BOYER-79] - Boyer, R.S., Moore, J.S. - "A Computational Logic" - Academic Press - 1979.

[DAVIES-82] - Davies, D.J.M. - "String Searching in Text Editors" - Software-Practice and Experience" - vol. 12 - pp. 709-717 - 1982.

[GALIL-79] - Galil, Z. - "On Improving the Worst Case Running Time of the Boyer-Moore String Matching Algorithm" - Comm. of ACM -

vol. 22 - nº 9 - pp. 505-508 - Set. 79.

[GONNET-83] - Gonnet, G.H. - "Unstructured Data Bases or Very Efficient Text Searching" - Technical Report - University of Waterloo - 1983.

[GUIBAS-77] - Guibas, L.J., Odlyzko, A.M. - "A New Proof of the Linearity of the Boyer-Moore String Searching Algorithm" - Proc. 18th Ann. IEEE Symp. Foundations of Computer Science - pp. 189-195 - 1977.

[HARRISON-71] - Harrison, C.M. - "Implementation of the Substring Test by Hashing" - Comm. of ACM - vol. 14 - nº 12 - pp.777-779 - Dez. 71.

[HORSPOOL-80] - Horspool, R.N. - "Practical Fast Searching in Strings" - Software-Practice and Experience - vol. 10 - pp. 501-506 - 1980.

[KASHYAP-83] - Kashyap, R.L., Oommen, B.J. - "The Noisy Substring Matching Problem" - IEEE Transactions on Software Engineering - vol. SE-9, nº 3 - pp. 365-370 - Maio 83.

[KNUTH-77] - Knuth, D.E., Morris, J.H., Pratt, V.R. - "Fast Pattern Matching in Strings" - SIAM J. Comput. - vol. 6 - nº 2 - pp. 323-350 - Jun. 77.

[RYTTER-80] - Rytter, W. - "A Correct Preprocessing Algorithm for Boyer-Moore String-Searching" - SIAM J. Comput. - vol. 9 - nº 3 - pp. 509-512 - Ago. 80.

[SMIT-82] - Smit, G. de V. - "A Comparison of Three String Matching Algorithms" - Software-Practice and Experience - vol. 12 -

pp. 57-66 - 1982.

[THARP-82] - Tharp, A.L., Tai, K. - "The Practicality of Text Signatures for Accelerating String Searching" - Software - Practice and Experience - vol. 12 - pp. 35-44 - 1982.

A P E N D I C E

```
(*
  APRESENTAMOS A SEGUIR,OS ALGORITMOS SIMPLES,MENORFREQ,KMP,BM
E BMS, ESCRITOS COMO PROCEDIMENTOS PASCAL,SENDO QUE TODOS POS-
SUEM ,EM GERAL, A SEGUINTE FORMA:



    PROCEDURE NOME(CAD,M,INICIO,INDICE);


ONDE CAD: A CADEIA DE CARACTERES PROCURADA;

     M  : O COMPRIMENTO DE CAD;

     INICIO: INDICE DO TEXTO ONDE A BUSCA DEVE SER INICIADA E

     INDICE: POSICAO DA PRIMEIRA OCORRENCIA DE CAD EM TEXTO.
             SE CAD NAO FOR ENCONTRADA,INDICE=0.



  OS PROCEDIMENTOS ASSUMEM AS SEGUINTES DEFINICOES DE TIPOS

    TYPE CCAD=0..20; (COMPRIMENTO DAS CADEIAS)

         CTEXTO=0..2002; (COMPRIMENTO DO TEXTO)

         TCAD=PACKED ARRAY[CCAD] OF CHAR;

         TTEXTO=PACKED ARRAY[CTEXTO] OF CHAR;



  ALEM DISSO,TEXTO:TTEXTO,QUE CONTEM O TEXTO A SER PESQUISADO
E N:CTEXTO,SEU COMPRIMENTO,SAO GLOBAIS A TODOS OS PROCEDIMENTOS.

  OS ALGORITMOS SIMPLES,MENORFREQ E KMP SUPOEM,PARA TRATAMENTO
DE CASOS ESPECIAIS, A EXISTENCIA DE 2 CARACTERES QUE NUNCA OCOR-
REM NO TEXTO, NAS POSICOES TEXTO[N+1] E CAD[M+1]; ALEM DISSO,
TEXTO[N+2] = CAD[1] NOS ALGORITMOS SIMPLES E KMP E
TEXTO[N+2] = CARACTERE DE MENOR FREQUENCIA EM CAD, NO ALGORITMO
MENORFREQ.  *)
```

```
PROCEDURE SIMPLES(CAD:TCAD;M:CCAD;INICIO:CTEXTO;
                     VAR INDICE:CTEXTO);


VAR K:CTEXTO;  (*PONTEIRO ATUAL DO TEXTO*)
    J:CCAD;    (*PONTEIRO ATUAL DA CADEIA*)
    FIMLOOP:BOOLEAN; (*PERMITE ABANDONAR O PROCESSO CASO
                        ENCONTRE CAD OU SE ESGOTE O TEXTO*)

    A:CHAR;


BEGIN

 K:=INICIO; J:=1; FIMLOOP:=FALSE;
 A:=CAD[1];
 WHILE NOT FIMLOOP DO
  BEGIN
   WHILE TEXTO[K] <> A DO K:=K+1;
   IF K>N THEN  (*TEXTO ESGOTADO*)
    BEGIN
     INDICE:=0;
     FIMLOOP:=TRUE
    END
   ELSE BEGIN
         K:=K+1; J:=J+1;
         WHILE CAD[J] = TEXTO[K] DO
          BEGIN
           K:=K+1;
           J:=J+1
          END;
         IF J>M THEN  (*CAD ENCONTRADA NA POSICAO K-M DE TEXTO*)
          BEGIN
           INDICE:=K-M;
           FIMLOOP:=TRUE
          END
         ELSE BEGIN
               K:=K-J+2;
               J:=1
              END
        END
  END
END;
```

```
PROCEDURE MENORFREQ(CAD:TCAD;M,POSIC:CCAD;INICIO:CTEXTO;
                    VAR INDICE:CTEXTO);

(* ESTE PROCEDIMENTO ASSUME,COMO VARIAVEL GLOBAL, O VETOR FQ,
 QUE CONTEM A FREQUENCIA DE CADA CARACTERE DO ALFABETO NO
 TEXTO. ALEM DISSO, O PARAMETRO POSIC INDICA A POSICAO EM
 CAD, DO CARACTERE DE MENOR FREQUENCIA DA CADEIA        *)

LABEL 10; (*RETURN*)
VAR K:CTEXTO;
    CAR:CHAR;
FUNCTION IDEM(K,L:CTEXTO):BOOLEAN;

(* IDEM(K,L)=(TEXTO[K]..TEXTO[L]=CAD) *)

VAR I:CTEXTO;
    J:CCAD;
    OK:BOOLEAN;
BEGIN
 I:=K; OK:=TRUE; J:=1;
 WHILE (I<=L) AND OK DO
  IF TEXTO[I] <> CAD[J] THEN OK:=FALSE
  ELSE BEGIN
         I:=I+1;
         J:=J+1
       END;
 IDEM:=OK
END;

BEGIN  (*MENORFREQ*)
 CAR:=CAD[POSIC];
 TEXTO[N+2]:=CAR;
 K:=INICIO+POSIC-2;
 REPEAT
  K:=K+1;
  WHILE TEXTO[K] <> CAR DO K:=K+1;
  IF K>N THEN  (*TEXTO ESGOTADO*)
   BEGIN
    INDICE:=0;
    GOTO 10
   END
 UNTIL IDEM(K-POSIC+1,K+M-POSIC);
 INDICE:=K-POSIC+1;
10:END;
```

```
  PROCEDURE KMP(CAD:TCAD;M:CCAD;INICIO:CTEXTO;VAR INDICE:CTEXTO;
                PRIM:BOOLEAN);

(* PARA ESTE METODO,SUPOE-SE A EXISTENCIA DA VARIAVEL GLOBAL
   PROX:ARRAY[CCAD] OF -1..20 E O PARAMETRO PRIM INDICA SE E'
   A PRIMEIRA VEZ QUE CAD ESTA SENDO PROCURADA; NESTE CASO, A
   TABELA PROX DEVE SER CONSTRUIDA      *)

  LABEL 10,20,30,40;

  VAR K:CTEXTO;
      OK:BOOLEAN;
      T:-1..20;
      J:CCAD;
      A:CHAR;

  BEGIN

   IF PRIM THEN
    BEGIN
     J:=1; T:=0; PROX[1]:=0;
     WHILE J<M DO
      BEGIN
       OK:=TRUE;
       WHILE (T>0) AND OK DO
        IF CAD[J] <> CAD[T] THEN T:=PROX[T]
        ELSE OK:=FALSE;
       T:=T+1; J:=J+1;
       IF CAD[J]=CAD[T] THEN PROX[J]:=PROX[T]
       ELSE PROX[J]:=T
      END
    END;
   J:=1; K:=INICIO; A:=CAD[1];
10: (*J=1*)
  WHILE TEXTO[K] <> A DO K:=K+1;
  IF K>N THEN  (*TEXTO ESGOTADO*)
   BEGIN
    INDICE:=0;
    GOTO 40  (*RETURN*)
   END;
20: (*CARACTERE DE CAD E TEXTO COINCIDEM*)
  J:=J+1;  K:=K+1;
30: (* J>0 *)
  IF TEXTO[K]=CAD[J] THEN GOTO 20;
  J:=PROX[J];
  IF J=1 THEN GOTO 10;
  IF J=0 THEN
   BEGIN
    J:=1;
    K:=K+1;
    GOTO 10
   END;
  IF J>0 THEN GOTO 30;
 (* NESTE PONTO,TEXTO[K-M]..TEXTO[K-1] COINCIDEM COM CAD *)
  INDICE:=K-M;
40:END;
```

```
PROCEDURE BMS(CAD:TCAD;M:CCAD;INICIO:CTEXTO;VAR INDICE:CTEXTO;
              PRIM:BOOLEAN);

(* NESTE CASO, APENAS A TABELA DELTA1 (GLOBAL) E' CONSTRUIDA
   CASO PRIM SEJA TRUE. UM ALFABETO BINARIO E' CONSIDERADO *)

LABEL 10; (**RETURN*)
VAR CAR,ULT:CHAR;
    J:CCAD;
    K:CTEXTO;
    P:'0'..'1';
BEGIN
 IF PRIM THEN
  BEGIN  (* CONSTRUCAO DA TABELA DELTA1 *)
   FOR P:='0' TO '1' DO DELTA1[P]:=M;
   FOR J:=1 TO M-1 DO DELTA1[CAD[J]]:=M-J;
  (* CORRIGIR O VALOR DE DELTA1[CAD[M]] *)
   DELTA1[CAD[M]]:=1;
   J:=M-1;
   WHILE J <> 0 DO
    IF CAD[J]=CAD[M] THEN
     BEGIN
      DELTA1[CAD[M]]:=DELTA1[CAD[M]]+1;
      J:=J-1
     END
    ELSE J:=0
  END;
 ULT:=CAD[M];
 K:=M+INICIO-1;
 WHILE K<=N DO
  BEGIN
   CAR:=TEXTO[K];
   IF CAR = ULT THEN
    BEGIN
     J:=M;
     REPEAT
      J:=J-1;
      IF J=0 THEN BEGIN
                   INDICE:=K;
                   GOTO 10
                  END;
      K:=K-1
     UNTIL TEXTO[K] <> CAD[J];
     CAR:=TEXTO[K];
     IF DELTA1[CAR] < (M-J+1) THEN K:=K+M-J+1
     ELSE K:=K+DELTA1[CAR]
    END
   ELSE K:=K+DELTA1[CAR]
  END;
 INDICE:=0;
10:END;
```

```
PROCEDURE BM(CAD:TCAD;M:CCAD;INICIO:CTEXTO;VAR INDICE:CTEXTO;
             PRIM:BOOLEAN);

(* AS TABELAS DELTA1:ARRAY['0'..'1'] OF 1..20 E
   DELTA2:ARRAY[CCAD] OF  INTEGER, SAO VARIAVEIS GLOBAIS E SAO
   CONSTRUIDAS NESTE PONTO,SE O PARAMETRO PRIM FOR TRUE, OU
   SEJA, SE CAD ESTA SENDO PROCURADA PELA PRIMEIRA VEZ.ESTARE-
   MOS CONSIDERANDO,COMO EXEMPLO,UM ALFABETO BINARIO *)

 LABEL 10; (*RETURN*)
 VAR CAR,ULT:CHAR;
     K:CTEXTO;
     T,J:CCAD;
     P:'0'..'1';
     ACABOU:BOOLEAN;
     F:ARRAY[CCAD] OF INTEGER;
     TP:INTEGER;
 BEGIN
  IF PRIM THEN
   BEGIN
    (* CONSTRUCAO DA TABELA DELTA1 *)
    FOR P:='0' TO '1' DO DELTA1[P]:=M;
    FOR J:=1 TO M DO DELTA1[CAD[J]]:=M-J;
    (* CONSTRUCAO DA TABELA DELTA2 *)
    FOR J:=1 TO M DO DELTA2[J]:=2*M-J;
    J:=M;  T:=M+1;
    WHILE J>0 DO
     BEGIN
      F[J]:=T;
      ACABOU:=FALSE;
      WHILE (T<=M) AND NOT ACABOU DO
       IF CAD[J] <> CAD[T] THEN
        BEGIN
         IF DELTA2[T] > M-J THEN DELTA2[T]:=M-J;
         T:=F[T]
        END
       ELSE ACABOU:=TRUE;
      T:=T-1;  J:=J-1
     END;
    FOR J:=1 TO T DO
     IF DELTA2[J] > M+T-J THEN DELTA2[J]:=M+T-J;
    TP:=F[T];
    WHILE T<=M DO
     BEGIN
      WHILE T<=TP DO
       BEGIN
        IF DELTA2[T] > (TP-T+M) THEN DELTA2[T]:=TP-T+M;
        T:=T+1
       END;
      TP:=F[TP]
     END
    END;
```

```
    ULT:=CAD[M];  K:=M+INICIO-1;
    WHILE K<=N DO
     BEGIN
      CAR:=TEXTO[K];
      IF CAR=ULT THEN
       BEGIN
        J:=M;
        REPEAT
         J:=J-1;
         IF J=0 THEN BEGIN
                      INDICE:=K;
                      GOTO 10
                     END;
         K:=K-1
        UNTIL TEXTO[K] <> CAD[J];
        IF DELTA1[TEXTO[K]] < DELTA2[J] THEN K:=K+DELTA2[J]
        ELSE K:=K+DELTA1[TEXTO[K]]
       END
      ELSE K:=K+DELTA1[CAR]
     END;
    INDICE:=0;
10:END;



(* OS DOIS PROCEDIMENTOS A SEGUIR CONSTITUEM O METODO DE HARRISON.
  A FUNCAO-ASSINATURA UTILIZADA E' DETERMINADA PELO PROCEDIMENTO-
  FUNCAO H(A[I],...,A[I+K-1]) QUE RETORNA A POSICAO QUE DEVE SER
  IGUAL A 1 NA ASSINATURA. K E' O TAMANHO  DAS SUBCADEIAS USADAS
  PARA FORMAR A ASSINATURA. SE H FOR A FUNCAO DE THARP E TAI,UM
  ARRAY QUE INDICA A CLASSE DE FREQUENCIA DE CADA CARACTERE DEVE
  SER UTILIZADO.

   ALEM DAS DEFINICOES DE TIPOS FEITAS NO INICIO DESTE APENDICE,
  CONSIDERAREMOS AS SEGUINTES DECLARACOES:


  CONST W=127; COMPRIMENTO DA ASSINATURA=W+1

        K=2;


  TYPE
        NB=0..W;    NUMERO DE BITS NUMA ASSINATURA

        COMPLINHA=0..82; COMPRIMENTO DA LINHA

        TLINHA=PACKED ARRAY[COMPLINHA] OF CHAR;

        TASSIN=PACKED ARRAY[NB] OF BOOLEAN;

        TARQ=FILE OF RECORD
              ASSIN:TASSIN;
              LIN:TLINHA;
              CL:COMPLINHA
             END;

        NLINHAS=0..200; NUMERO DE LINHAS NO TEXTO
```

O PROCEDIMENTO PREPROCESSA(TEXTIMP,ARQASSIN) REALIZA O PRE-PROCESSAMENTO DO TEXTO QUE SE ENCONTRA NO ARQUIVO TEXTIMP.CADA LINHA DO TEXTO E' LIDA E SUA ASSINATURA E' CONSTRUIDA. O REGISTRO FORMADO POR UMA LINHA,SUA ASSINATURA E SEU COMPRIMENTO E' GRAVADO NO ARQUIVO ARQASSIN.

NA PRATICA, CALCULA-SE A NOT(ASSINATURA) DE CADA LINHA PARA FACILITAR O TESTE DE ASSINATURA DURANTE A BUSCA.*)

```
PROCEDURE PREPROCESSA(VAR TEXTIMP:TEXT;VAR ARQASSIN:TARQ);

VAR LINHA:TLINHA;
    NOTASSIN:TASSIN;
    POSIC,J:NB;
    I:INTEGER;

FUNCTION H(...):NB;
  ....

BEGIN

 RESET(TEXTIMP);
 REWRITE(ARQASSIN,'DISK:ARQ.DATA');
 REPEAT
  (*LER LINHA DO TEXTO*)
  FOR I:=0 TO W DO NOTASSIN[I]:=TRUE;
  I:=0;
  WHILE NOT EOLN(TEXTIMP) DO
   BEGIN
    I:=I+1;
    READ(TEXTIMP,LINHA[I]);
   END;
  READLN(TEXTIMP);
  (*CALCULAR NOT(ASSINATURA DA LINHA)*)
  FOR J:=1 TO I-K+1 DO  (*I E' O COMPRIMENTO REAL DA LINHA*)
   BEGIN
    POSIC:=H(LINHA[J],...,LINHA[J+K-1]);
    NOTASSIN[POSIC]:=FALSE
   END;

   (* GRAVAR LINHA,ASSINATURA E COMPRIMENTO NO
      ARQUIVO ARQASSIN     *)
  WITH ARQASSIN^ DO
   BEGIN
    ASSIN:=NOTASSIN;
    LIN:=LINHA;
    CL:=I;
   END;
  PUT(ARQASSIN)
 UNTIL EOF(TEXTIMP)
END;
```

```
(* O PROCEDIMENTO HARRISON(ARQASSIN,CAD,M,NL,INDICE) PROCURA
 PELA CADEIA CAD,DE COMPRIMENTO M, NO ARQUIVO PRE-PROCESSADO
 ARQASSIN CONSTRUIDO PELO PROCEDIMENTO ANTERIOR. A VARIAVEL NL
 RETORNA O NUMERO DA LINHA EM QUE A CADEIA FOI ENCONTRADA; RE-
 TORNA 0 CASO A CADEIA NAO OCORRA NO TEXTO; INDICE E' A POSICAO
 DA PRIMEIRA OCORRENCIA DE CAD NA LINHA.
  O PROCEDIMENTO BUSCA(LINHA,CL,CAD,M,INDICE) FAZ A BUSCA DE
 CAD EM LINHA,USANDO UM DOS METODOS EXATOS DE BUSCA(SIMPLES,KMP,
 BM,BMS) ONDE CL E' O COMPRIMENTO DA LINHA E INDICE RETORNA 0 SE
 A CADEIA NAO OCORRER NA LINHA; CASO CONTRARIO RETORNA A POSICAO
 DA PRIMEIRA OCORRENCIA DE CAD NA LINHA.  *)



PROCEDURE HARRISON(VAR ARQASSIN:TARQ;CAD:TCAD;M:CCAD;
                    VAR NL:NLINHAS;VAR INDICE:COMPLINHA);



VAR LINHA:TLINHA;
    CL:COMPLINHA;
    ANDASSIN,TUDOZERO,ASSINCAD:TASSIN;
    POSIC:NB;
    J:INTEGER;
    I:NLINHAS;

FUNCTION H(...):NB;
  .....

PROCEDURE BUSCA(LINHA:TLINHA;CL:COMPLINHA;CAD:TCAD;
                  M:CCAD;VAR INDICE:COMPLINHA);
  ....


BEGIN  (*HARRISON*)

(* CALCULAR A ASSINATURA DA CADEIA CAD *)
 FOR J:=0 TO W DO
  BEGIN
   ASSINCAD[J]:=FALSE;
   TUDOZERO[J]:=FALSE
  END;
 FOR J:=1 TO M-1 DO
  BEGIN
   POSIC:=H(CAD[J],...,CAD[J+K-1]);
   ASSINCAD[POSIC]:=TRUE
  END;
 RESET(ARQASSIN);
 I:=0;
 REPEAT
  I:=I+1;
```

```
  (* PARA CADA LINHA I DO ARQUIVO ARQASSIN,
     CALCULAR ANDASSIN=NOTASSIN AND ASSINCAD
     SE ANDASSIN=TUDOZERO ENTAO A LINHA E' PES-
     QUISADA EXAUSTIVAMENTE *)

  FOR J:=0 TO W DO
   ANDASSIN[J]:=ARQASSIN^.ASSIN[J] AND ASSINCAD[J];
  IF ANDASSIN=TUDOZERO THEN
   BEGIN
    BUSCA(ARQASSIN^.LIN,ARQASSIN^.CL,CAD,M,INDICE);
    IF INDICE <> 0 THEN NL:=I
    ELSE GET(ARQASSIN)
   END
 UNTIL (EOF(ARQASSIN) OR (INDICE <> 0));
 IF EOF(ARQASSIN) AND (INDICE = 0) THEN NL:=0
END;
```

Obs: - Os dois últimos procedimentos foram utilizados na experiência realizada durante este trabalho. Numa aplicação real, a estrutura de dados adotada deve ser modificada a fim de dissociar a assinatura de cada linha, dos caracteres que a compõem.