Terceiro Trabalho Prático de Projeto e Análise de Algoritmos - Casamento de Padrão com Busca Aproximada

Kristian Magnani dos Santos

31 de Maio de 2004

Este trabalho e os códigos-fonte dos programas aqui apresentados podem ser encontrados em `http:\\www.dcc.ufmg.br\~kristian`*, sob o link* "Trabalho Prático 3 de PAA".

Conteúdo

1 Introdução

O problema de encontrar um padrão em uma cadeia de caracteres de algum alfabeto Σ é um problema fundamental na Ciência da Computação. Suas aplicações passam por processamento de textos em linguagem natural, códigos, dicionários e chegam até à biologia computacional e ao processamento de imagens. Desta forma, são estudados neste trabalho os principais algoritmos de casamento de padrão com busca exata e busca aproximada.

A implementação proposta encontra-se no Apêndice A, e constitui-se no arquivo `kgrep.c`, que pode ser compilado com o comando `gcc -o kgrep kgrep.c`. Após compilado, a sintaxe geral é:

```
kgrep -[C][c][f][FSBYAD][num] padrao texto
```

A opção `-C` realiza um teste de comparações entre os algoritmos BMH e BMHS, conforme detalhado na seção de testes. A opção `-f` indica que o `padrao` deve ser interpretado como um nome de arquivo onde cada linha é um padrão, ao passo que a opção `-c` ativa o modo de estatisticas, onde apenas a contagem total de ocorrências é impressa (o normal é imprimir a linha onde há a ocorrência). As opções `F, S, B, Y, A, D` indicam qual algoritmo será utilizado, a saber: Força Bruta, Shift-And Exato, BMH, BMHS, Shift-And Aproximado e Programação Dinâmica Aproximado. Quando as opções `A` ou `D` são utilizadas, estas devem ser seguidas por um número indicando a distância de edição máxima permitida para se considerar um eventual ocorrência do padrão no texto.

2 Algoritmos e Estruturas de Dados Utilizados

Nesta seção, são apresentados os princiapis algoritmos utilizados neste trabalho para o casamento de padrões. Estes algoritmos realizam a busca aproximada ou exata, conforme detalhado caso a caso. Todos os algoritmos foram implementados no programa `kgrep`, apresentado anteriormente na introdução deste trabalho. O código-fonte da implementação em C do `kgrep` pode ser encontrado no Apêndice A deste relatório.

A principal estrutura de dados utilizada é o vetor de caracteres, que armazena em memória os arquivos lidos do disco. Este arquivos podem ser tanto arquivos contendo padrões quanto arquivos contendo o texto onde os padrões devem ser procurados. Quando o arquivo é maior que o *buffer* em memória, ele é dividido em partes, e cada parte é carregada separadamente. Esta divisão é feita tomando-se o cuidado de não se quebrar um padrão ao meio, situação onde tal ocorrência não seria detectada se tal cuidado não fosse tomado.

Para a representação das tabelas dos algoritmos de Shift-End Exato e Shif-And Aproximado foram utilizados vetores de inteiros, através da definição do tipo `TabelaShiftAnd`. Já para as tabelas dos algoritmos BMH e BMHS foram utilizados vetores de inteiros através da definição do tipo `TabelaBMHouBMHS`.

A seguir, apresenta-se a explicação do princípio de funcionamento dos algoritmos utilizados na implementação e também a análise de complexidades de tempo e espaço de cada um deles.

2.1 Shift-And Exato

2.1.1 Apresentação do Algoritmo

O algoritmo Shift-And para casamento *Exato* de padrões foi apresentado pela primeira vez por Baeza-Yates e Gonnet, em 1989, e se baseio no conceito de paralelismo de bit. Este paralelismo de bit diz respeito à natureza naturalmente paralela com que operações sobre bits são realizadas em computadores. Este algoritmo é cerca de duas vezes mais rápido que o KMP, além de ser muito mais simples.

Através do paralelismo de bit, diversos valores são agrupados em uma única palavra e todos estes valores podem ser corretamente alterados em uma única operação. Isto reduz o número de operações por um fator que pode ser até mesmo igual ao número de bits por palavra da arquitetura, que hoje em dia em geral é 32 ou 64. Dado um padrão P, o algoritmo mantém um conjunto de todos os prefixos de P que casam com texto já lido. Uma máscara de bits $R = (b_1, b_2, \ldots, b_m)$ para representar este conjunto.

O algoritmo começa colocando o valor 1 na j-ésima posição de R se e somente se $p_1 \ldots p_j$ é um sufixo de $t_1 \ldots t_i$ (nesta notação, cada p_x é um caracter do padrão P, e cada t_x é um caracter do texto T). Esta posição é dita estar ativa, e um casamento acontece sempre que b_m fica ativo.

Quando o próximo caracter é lido, uma nova máscara R' é recalculada a partir de R, da seguinte forma:

$$R' = ((R >> 1)|10^{m-1})\&M[T[i]]$$

onde M é o vetor de máscaras de cada caracter do padrão P, e é construído fazendo igual a 1 a posição de ocorrência do caracter em questão no padrão, caso este caracter ocorra em P. Por exemplo, se $P = \texttt{teste}$, então $M['t'] = 10010$, $M['e'] = 01001$ e $M['s'] = 00100$. A operação $R >> 1$ desloca todos os bits de R para a direita em uma unidade, e 10^{m-1} representa uma seqüência

procedimento Shift-And ($P = p_1 p_2 \ldots p_m$, $T = t_1 t_2 \ldots t_n$)
  {- - *Pré-Processamento* - -}
  para todo c **em** Σ **faça** $M[c] \leftarrow 0^m$;
  para $j \leftarrow 1$ **até** m **faça** $M[p_j] \leftarrow M[p_j] | 0^{j-1} 1 0^{m-j}$;
  {- - *Pesquisa* - -}
  $R \leftarrow 0^m$;
  para $i \leftarrow 1$ **até** n **faça**
    $R \leftarrow ((R >> 1) | 10^{m-1}) \& M[T[i]]$;
  se $(R \& 0^{m-1} 1 \neq 0^m)$ **então**
    imprime("Casamento na posição %d", $i - m + 2$);

Figura 1: O algoritmo Shift-And para casamento Exato de padrões

de bits iniciada por 1 e seguida de $m - 1$ 0's. A operação & é a operação de *and* bit a bit padrão.

O algoritmo é apresentado na Figura 1.

2.1.2 Análise de Complexidade

Para o algoritmo de Shift-And Exato, o custo de tempo é $O(n)$ desde que as operações de manipulação de bits possuam uma implementação eficiente, ou seja, possam ser realizadas em $O(1)$. Para a complexidade de espaço, pode-se considerar que são gastos $O(n)$ para o texto e $O(\Sigma)$ para a montagem da tabela, e portanto o espaço gasto é $O(n + \Sigma)$, segundo este ponto de vista.

2.2 BMH

2.2.1 Apresentação do Algoritmo

O algoritmo BMH é resultado de um simplificação realizada por Horspool em 1980 no algoritmo de casamento exato de padrão proposto por Boyer e Moore (BM) em 1977. A idéia do algoritmo original é pesquisar o padrão no sentido da direita para a esquerda, o que torna o algoritmo mais rápido. A simplificação introduzida por Horspool não prejudica a eficiência do algoritmo, e por isso o BMH deve ser usado no lugar do BM sempre que possível.

Nos algoritmos derivados do BM, o princípio geral é pesquisar o padrão P em uma janela que desliza ao longo do texto T e a cada instante efetuar comparações entre o sufixo da janela e o sufixo de P, da direita para a esquerda. Se a comparação não encontrar desigualdades, então uma ocorrência de P em T foi encontrada, caso contrário um deslocamento é calculado e efetuado, e a busca continua. O deslocamento, no algoritmo original BM, pode ser calculado segundo duas heurísticas, a saber:

```
procedure BMH (var T: TipoTexto; var n: integer;
               var P: TipoPadrao; var m: integer);
var i, j, k: Integer;
    d: array[0..MaxChar] of integer;
begin
  {-- Pre-processamento do padrao --}
  for j := 0 to MaxChar do d[j] := m;
  for j := 1 to m-1 do d[ord(P[j])] := m-j;
  i := m;
  while i <= n do {-- Pesquisa --}
    begin
    k := i;  j := m;
    while (T[k] = P[j]) and (j>0) do begin k := k - 1; j := j - 1; end;
    if j = 0 then writeln(' Casamento na posicao: ', k + 1:3);
    i := i + d[ord(T[i])];
    end;
end;
```

Figura 2: O algoritmo BMH (Boyer-Moore-Horspool).

- **heurística de ocorrência:** alinha a posição do texto que causou a colisão com o primeiro caractere no padrão que casa com ele;
- **heurística de casamento:** ao mover o padrão para a direita, ele casa com o pedaço do texto anteriormente casado.

Enquanto o algoritmo BM realiza os cálculos baseados nas duas heurísticas e então escolhe aquele que propicia o maior deslocamento, o algoritmo BMH simplifica as coisas utilizando o caractere no texto correspondente ao último caractere do padrão. Para pré-computar o padrão, todas as entradas na tabela de deslocamentos $d[]$ são feitos iguais a m. Em seguida, apenas os $m-1$ primeiros caracteres do padrão são usados para obter os outros valores da tabela através da seguinte regra:

$$d[x] = min\{j, \text{ tal que } j = m \mid (1 \leq j < m \ \& \ P[m-j] = x)\}.$$

Assim, para $P = \texttt{teste}$, $d['t'] = 1$, $d['e'] = 3$, $d['s'] = 2$, e todos as outras entradas de d são iguais a $|P|$, neste caso, $m = 5$.

O algoritmo BMH é apresentado na Figura 2, retirado de [1].

2.2.2 Análise de Complexidade

No algoritmo BMH, o deslocamento ocorrência pode ser pré-computado com base apenas no padrão e no alfabeto, e a complexidade de tempo e de espaço nesta fase são ambas iguais a $O(c)$. Além disso, é necessário um espaço $O(n)$ para armazenar o texto em memória. Já durante a pesquisa, a complexidade de tempo possui os seguntes casos:

- pior caso: $O(nm)$
- melhor caso: $O(n/m)$
- caso médio: $O(n/m)$.

Assim, a complexidade de tempo é em média $O(n/m)$, e a de espaço é $O(n+c)$.

2.3 BMHS

2.3.1 Apresentação do Algoritmo

O algoritmo BMHS é resultado de mais uma simplificação no algoritmo BM, desta vez realizada por Sunday em 1990. Esta modificação é uma variante do BMH apresentado anteriormente, onde Sunday propôs endereçar a tabela com o caractere no texto correspondente ao próximo caractere após o último caractere do padrão. Esta proposta difere ligeiramente daquela adotada eno BMH, onde o último caractere é utilizado para o deslocamento.

O pré-cômputo da tabela de deslocamentos é feito da seguinte forma:

- inicialmente, todas as entradas da tabela $d[]$ dão feitas iguais a $m+1$;
- a seguir, os m primeiros caracteres do padrão são utilizados para se corrigir algumas entradas da tabela segundo a expressão:

$$d[x] = min\{j, \text{ tal que } j = m \mid (1 \leq j \leq m \;\&\; P[m+1-j] = x)\}.$$

Para $P =$ `teste`, $d['t'] = 2$, $d['e'] = 1$, $d['s'] = 3$, e todos as outras entradas de d são iguais a $|P|+1$, neste caso, $m+1 = 6$.

O algoritmo BMHS é apresentado na Figura 3, retirado de [1].

2.3.2 Análise de Complexidade

No algoritmo BMHS, o comportamento assintótico é igual ao do algoritmo BMH, mas os deslocamentos são mais longos, podendo até mesmo serem iguais a $m+1$, e saltos relativamentes maiores são alcançados para padrões curtos.

Assim, a complexidade de tempo é em média $O(n/m)$, e a de espaço é $O(n+c)$.

```
procedure BMHS (var T: TipoTexto; var n: integer;
                var P: TipoPadrao; var m: integer);
var i, j, k: Integer;
    d: array[0..MaxChar] of integer;
begin
  {-- Pre-processamento do padrao --}
  for j := 0 to MaxChar do d[j] := m+1;
  for j := 1 to m do d[ord(P[j])] := m + 1 - j;
  i := m;
  while i <= n do {-- Pesquisa --}
    begin
    k := i;  j := m;
    while (T[k] = P[j]) and (j>0) do begin k := k - 1; j := j - 1; end;
    if j = 0 then writeln(' Casamento na posicao: ', k + 1:3);
    i := i + d[ord(T[i+1])];
    end;
end;
```

Figura 3: O algoritmo BMHS (Boyer-Moore-Horspool-Sunday).

2.4 Shift-And Aproximado

2.4.1 Apresentação do Algoritmo

A variação do algoritmo Shift-And que realiza casamento aproximado de cadeias foi proposta por Wu e Manber em 1992. Neste algoritmo, cada linha j do autômato não-determinista é empacotada em uma palavra R_j diferente do computador. Todas as transições do autômato são simuladas a cada caractere lido usando-se operações entre as $k+1$ mácaras de bits. Os novos valores R'_j são obtidos a partir dos valores correntes de R_j, a cada posição i do texto, da seguinte forma:

$$R'_0 = ((R_0 >> 1) \mid 10^{m-1}) \ \& \ M[T[i]]$$

$$R'_j = ((R_j >> 1) \ \& \ M[T[i]]) \mid R_{j-1} \mid (R_{j-1} >> 1) \mid (R'_{j-1} >> 1)$$

Nestas expressões, $M[]$ é a tabela gerada pelo algoritmo Shift-And Exato apresentado anteriormente. Cada R_j é iniciado tal que $R_j = 1^j 0^{m-j}$, e R_0 é equivalente à busca exata, enquanto que cada R_j corresponde aos prefixos encontrados até o momento com distância de edição j.

O algoritmo de Shift-And Aproximado é apresentado na Figura 4.

```
procedure ShiftAndAproximado (var T: TipoTexto; var n: Integer;
                              var P: TipoPadrao; var m, k: Integer);
var
  Masc: array[0..MaxChar - 1] of Integer;
  i, j, Ri, Rant, Rnovo:  Integer;
  R: array[0..NumMaxErros] of Integer;
begin
  { Pre-processamento padrao }
  for i := 0 to MaxChar - 1 do Masc[i] := 0;
  for i := 1 to  m do
    Masc[ord(P[i]) + 127] := Masc[ord(P[i]) + 127] or (1 shl (m - i));
  { Pesquisa }
  R[0] := 0;
  Ri := 1 shl (m - 1);
  for j := 1 to k do R[j] := (1 shl (m -j)) or R[j - 1];
  for i := 0 to n - 1 do
    begin
    Rant := R[0];
    Rnovo := ((Rant shr 1) or Ri) and Masc[ord(T[i + 1]) + 127];
    R[0] := Rnovo;
    for j := 1 to k do
      begin
      Rnovo := ((R[j] shr 1) and Masc[ord(T[i + 1]) + 127]) or
                 Rant or ((Rant or Rnovo) shr 1);
      Rant := R[j];
      R[j] := Rnovo or Ri;
      end;
    if (Rnovo and 1) <> 0
    then writeln( ' Casamento na posicao ', i + 1 );
    end;
end;
```

Figura 4: O algoritmo Shift-And Aproximado.

2.4.2 Análise de Complexidade

Para o pior caso e para o caso médio, a complexidade de espaço do algoritmo Shift-And Aproximado é $O(k\lceil m/w \rceil n)$, onde w é o tamanho em bits de uma palavra do computador em questão. Quando $m \leq w$, esta complexidade de tempo equivale a $O(kn)$. Já a complexidade de espaço para simular os autômatos é $O(km)$, pois são necessários k autômatos de tamanho m.

2.5 Programação Dinâmica Aproximado

2.5.1 Apresentação do Algoritmo

A versão final do algoritmo de casamento aproximado de padrões usando programação dinâmica é atribuído a Sellers, e data de 1980. Segundo [3], este algoritmo não é o mais eficiente, mas é o que possuia maior capacidade de adaptação para funções de distâncias mais complexas.

O algoritmo começa inicializando um vetor $C[]$ de tamanho $m+1$ tal que $C[i] \leftarrow i$, e então processa o texto caractere a caractere. A cada caractere t_j, o vetor $C[]$ é atualizado de acordo com a seguinte fórmula:

$$C'_i = \begin{cases} C_{i-1} & \text{, se } p_i = t_j \\ 1 + min(C_{i-1}, C'_{i-1}, C_i) & \text{, caso contrário} \end{cases}$$

As posições j do texto onde $C[m] \leq k$ são as posições onde estão as ocorrências aproximadas.

O algoritmo de busca aproximada por programação dinâmica é apresentado na Figura 5.

2.5.2 Análise de Complexidade

O algoritmo dinâmico apresentado para o problema da busca aproximada tem complexidade de tempo $O(mn)$ e compexidade de espaço $O(m)$.

3 Resultados Experimentais

A seguir são apresentados os testes realizados com a implementação feita neste trabalho. O código fonte desta implementação pode ser visto no Apêndice A deste trabalho. Todos os gráficos apresentados possuem como eixo das abscissas os valores dos tamanhos das bases de dados `wsj89`, `wsj88_10`, `wsj88_20` e `wsj88`, a saber, 2.8 MB, 10.1 MB, 20.2 MB e 109.4 MB. Para os padrões, foi utilizado o arquivo `wsj89_consultas`.

Para separar o tempo de entrada e saída em cada programa do tempo realmente gasto para fazer a busca, foi usado o comando `time` da shell de comando do Linux. Este comando recebe como argumento a chamada do programa a ser cronometrado, e então executa o programa recebido como

```
procedure DynamicProgrammingAprxSearch(T, P, k)
{
  /* Pre-processamento */
  for (i = 0; i<= m; i++) C[i] = i;
  lact = k + 1;

  /* Busca */
  for (pos = 1; pos <= n; pos++)
  {
    pC = 0; nC = 0;
    for (i = 1; i <= lact; i++)
    {
      if (P[i-1] == T[pos-1]) nC = pC;
      else
      {
        if (pC < nC) nC = pC;
        if (C[i] < nC) nC = C[i];
        nC++;
      }
      pC = C[i]; C[i] = nC;
    }
    while (C[lact] > k) lact--;
    if (lact == m) print("Encontrou uma ocorrência at \%d", pos);
    else lact++;
  }
}
```

Figura 5: O algoritmo de busca aproximada por programação dinâmica.

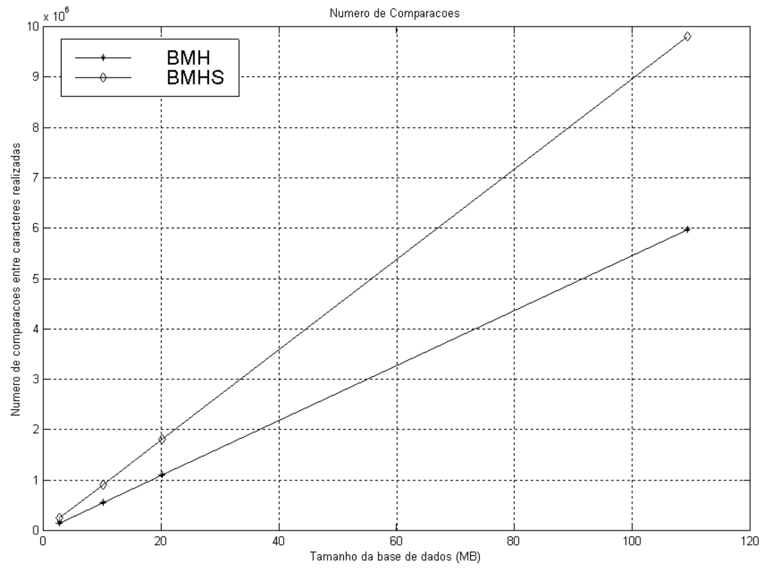

Figura 6: Número de Comparações entre Caracteres para BMH e BMHS

parâmetro, retornando ao final o tempo gasto na execução deste. Esta informação de tempo é separada em três componentes: `real`, `user` e `sys`. A componente utilizada como padrão foi a `user`, que é basicamente, neste caso, o tempo gasto pelo programa `kgrep` excetuando-se as operações de IO.

3.1 Número de Comparações entre Caracteres, para BMH e BMHS

O primeiro teste realizado foi o de número de comparações entre caracteres para os algoritmos BMH e BMHS. Este teste é ativado pela opção `-C` do `kgrep`. O resultado deste teste pode ser visto na Figura 6. O algoritmo BMHS apresentou um número maior de comparações efetuadas, mas este número é compensado na prática por sua heurística de cálculo de saltos no texto, como pode ser visto no teste de tempo de execução a seguir.

3.2 Tempo de Execução, para Shift-And Exato, BMH e BMHS

O próximo teste mede o tempo de execução para os algoritmos que fazem a busca exata. As medições foram feitas conforme explicado no início desta seção. Os resultados dos testes podem ser vistos na Figura 7.

O comportamento dos algoritmos BMH e BMHS foi praticamente igual, como pode ser visto na Figura 7. Já o Shift-And teve um comportamento um pouco inferior. Este resultado pode ser justificado pelo fato de haverem poucas ocorrências dos padrões no texto pesquisado. Como a detecção de uma

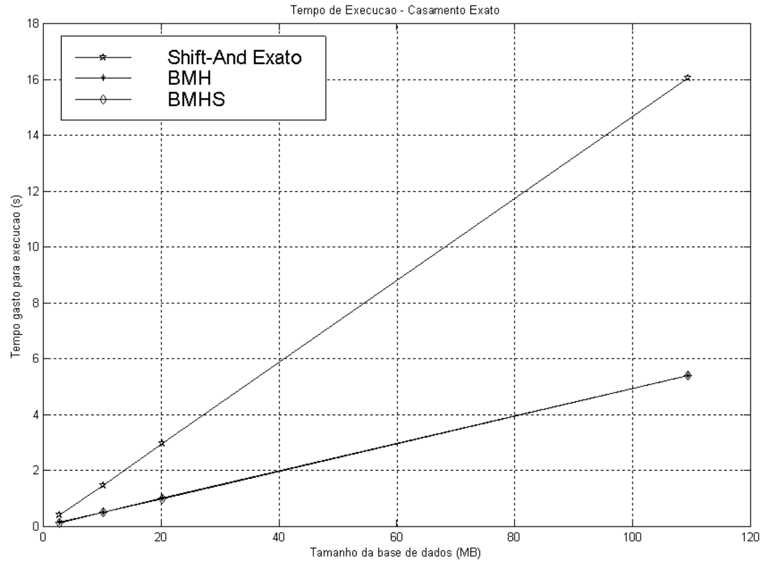

Figura 7: Tempo de Execução para Shift-And Exato, BMH e BMHS

"não-ocorrência" é relativamente rápida para os padrões e textos utilizados, os algoritmos BMH e BMHS realizam grandes saltos, ao passo que o Shift-And realiza uma busca caracter a caractere (embora utilizando paralelismo de bits).

Para ilustrar este fato, foi criado um arquivo chamado `testePatologico.dat`, composto de 40 mil linhas, onde cada linha é uma seqüência de 20 caracteres "a", seguido de um caractere "b". Em seguida foi efetuada uma busca pelo pafrão `aaaaaaaab` neste arquivo, e o resultado é que o algoritmo Shift-And foi duas vezes mais rápido que os algoritmos BMH e BMHS.

3.3 Tempo de Execução, para Shift-And Aproximado e Programação Dinâmica Aproximado

O próximo teste mede o tempo de execução para os algoritmos que fazem a busca aproximada. As medições foram feitas conforme explicado no início desta seção. Os resultados dos testes podem ser vistos nas Figuras 8, 9 e 10, para $k = 1$, $k = 4$ e $k = 8$, respectivamente.

Como pode-se observar, o algoritmo de Shift-And Aproximado foi melhor que o algoritmo de Programação Dinâmica. Este resultado já era esperado segundo a teoria apresentada [3], pois a complexidade de tempo da implementação de programação dinâmica proposta é $O(mn)$, contrastando com a complexidade de tempo de $O(kn)$ para o algoritmo de Shift-And Aproximado. Uma melhoria seria buscar uma implementação do algoritmo de programação dinâmica que apresentasse uma complexidade de tempo melhor.

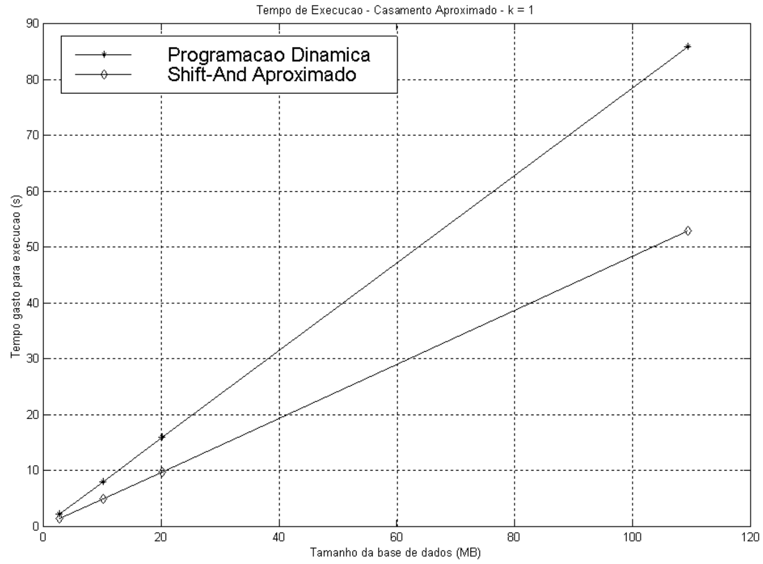

Figura 8: Tempo de Execução para para Shift-And Aproximado e Programação Dinâmica Aproximado - $k = 1$

Alguns pontos de partida para tais implementações são apresentados em [3], página 150, seção 6.2.4.

3.4 Comparação com o GREP e com o AGREP

Quando comparados com o GREP, as implementações de busca exatas mostram um desempenho menor. As implementações do BMH e BMHS são ligeiramente menos rápidas que o GREP, o que mostra que ainda há espaço para alguma pequena otimização de código. Já o Shift-And Exato seria mais adequado em algum caso patológico, como discutido anteriormente. Os resultados podem ser vistos na Figura 11.

A comparação com o AGREP pode ser vista nas Figuras 12, 13 e 14, para $k = 1$, $k = 4$ e $k = 8$, respectivamente. Na comparação com o AGREP, as implementações de Shift-And Aproximado e Programação Dinâmica se mostraram consideravelmente menos eficientes, a princípio. Entretanto, ao se analizar melhor os dados, observa-se que as implementações tendem a se igualar ao AGREP quando k aumenta (k é o número de aproximações permitidas). Além disso, quando a base de dados pesquisada aumenta, o tempo real de execução do AGREP (que é diferente do *user time*, pois considera também o tempo de entrada e saída) cresce bastante, chegando a ser pior que as implementações realizadas. Isto sugere que a forma como os loops de busca do KGREP foram implementados têm influência significativa. No KGREP, o loop de busca mais externo é o que lê do arquivo e carrega

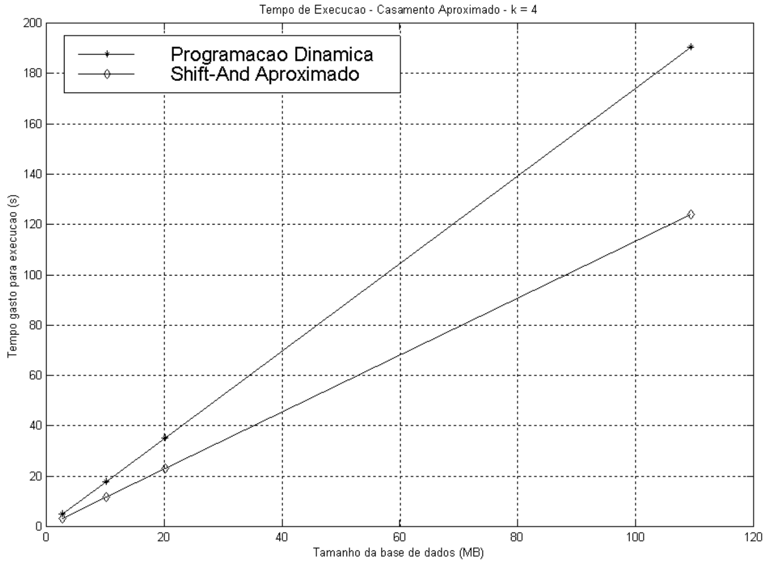

Figura 9: Tempo de Execução para para Shift-And Aproximado e Programação Dinâmica Aproximado - $k = 4$

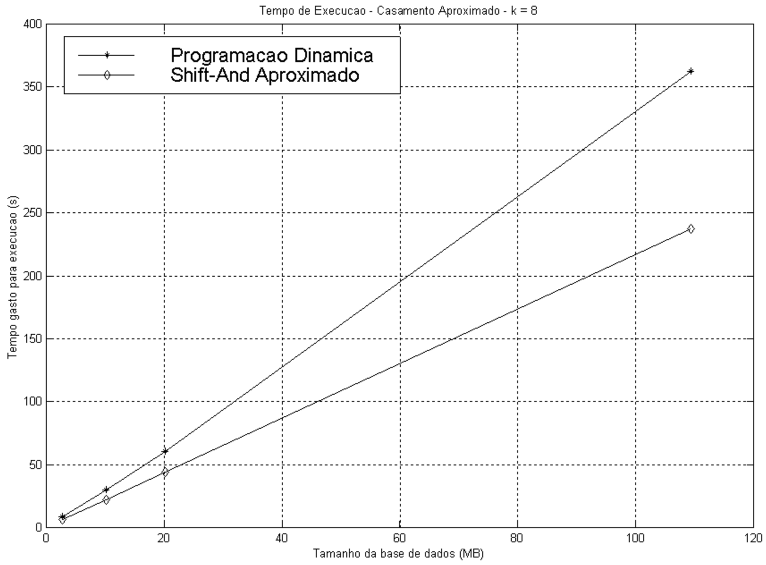

Figura 10: Tempo de Execução para para Shift-And Aproximado e Programação Dinâmica Aproximado - $k = 8$

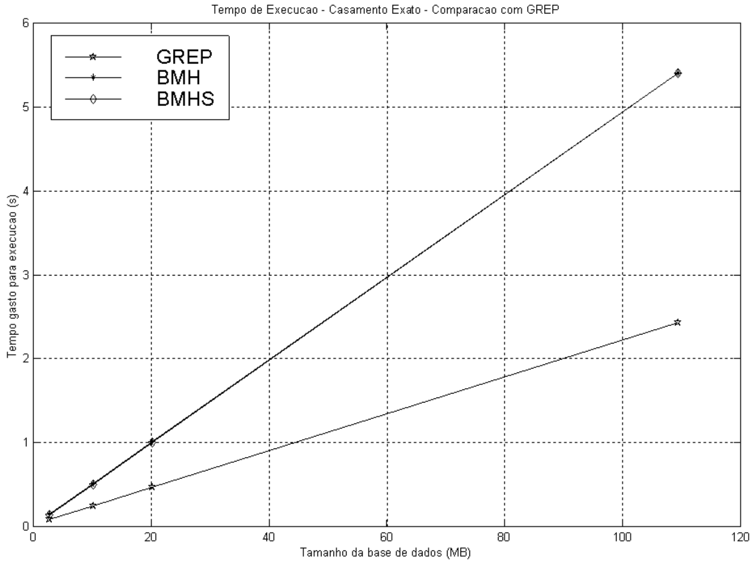

Figura 11: Comparação com GREP - Tempo de Execução para Shift-And Exato, BMH e BMHS

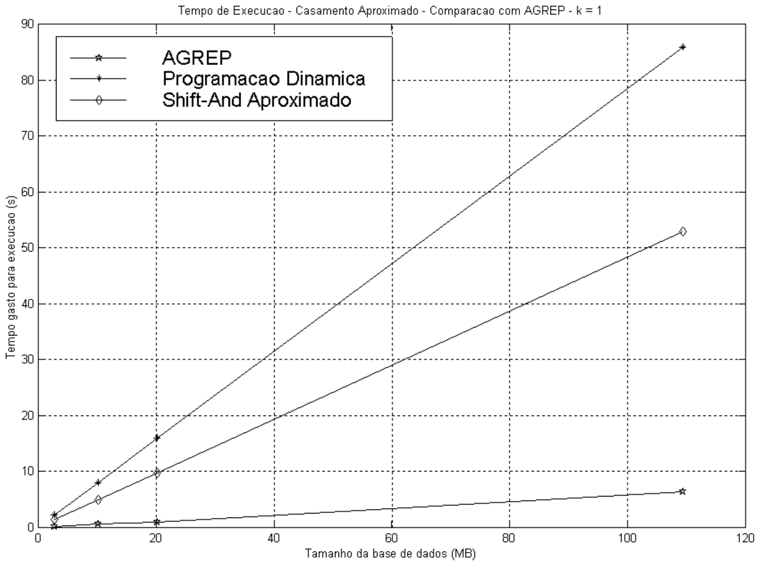

Figura 12: Comparação com AGREP - Tempo de Execução para para Shift-And Aproximado e Programação Dinâmica Aproximado - $k = 1$

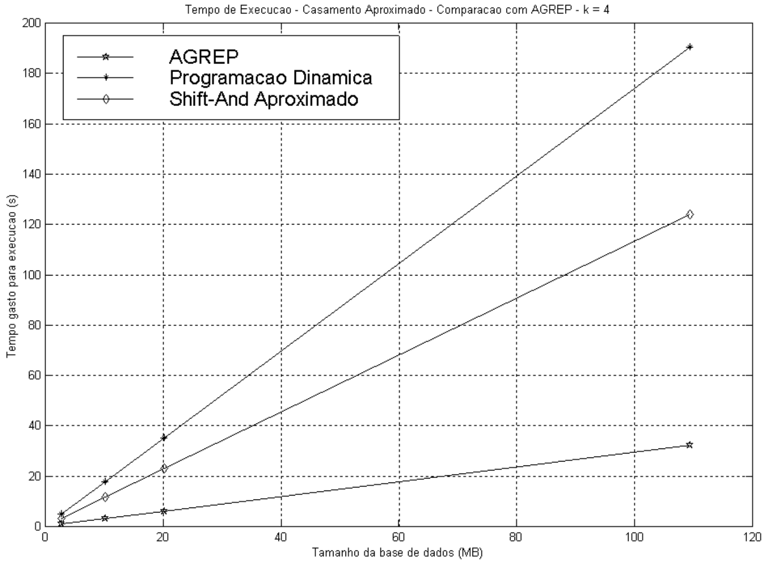

Figura 13: Comparação com AGREP - Tempo de Execução para para Shift-And Aproximado e Programação Dinâmica Aproximado - $k = 4$

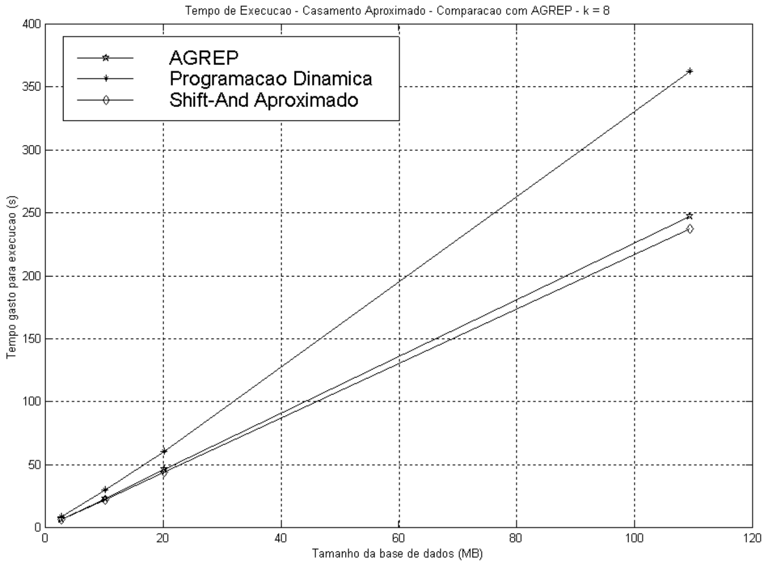

Figura 14: Comparação com AGREP - Tempo de Execução para para Shift-And Aproximado e Programação Dinâmica Aproximado - $k = 8$

na memório, e o mais interno é o que submete os diferentes padrões de busca. Esta decisão de projeto diminui o tempo de IO, mas aumenta o tempo de busca em si, pois um custo fixo associado com a mudança dos padrões é computado várias vezes (por exemplo, montagem de tabelas). Esta decisão se baseia na idéia de que o IO é, em geral, a parte mais demorada do programa. Na forma como o AGREP está sendo executado, a ordem dos loops de busca é exatamente ao contrário, o que favorece a busca em bases de dados menores.

4 Conclusões

Neste trabalho foi possível estudar diversos algoritmos para o problema do casamento de padrões, tanto em sua instância de busca exata quanto de busca aproximada. Além disso, foi possível fazer uma implementação razoável de um programa para busca usando os algoritmos estudados.

Diversos testes foram realizados com o objetivo de confrontar a implementação realizada com programas já estabelecidos. Estes teste também permitiram verificar diferenças entre os algoritmos estudados e implementados.

A implementação realizada forneceu um contato efetivo com a aplicação da teoria estudada na prática, inclusive servindo para ilustrar como algumas dificuldades podem surgir ao tentar implementar em uma linguagem real um algoritmo inicialmente proposto em pseudo-código.

Em suma, este trabalho serviu como um exercício esclarecedor da atividade de projetar e analisar algoritmos.

Referências

[1] Ziviani, N.: *Projeto e Análise de Algoritmos: com implementações em PASCAL e C.* 2a. edição rev. e ampl. São Paulo: Pioneira Thomson Learning, 2004.

[2] Baase, S.: *Computer Algorithms.*

[3] Navarro. Raffinot.: *Flexible Pattern matching in Strings.*

A Código Fonte do KGREP

```
/**
 * Terceiro Trabalho Pratico de PAA - 2004
 * CASAMENTO DE PADRAO COM BUSCA APROXIMADA
 * Prof.: Nivio Ziviani
 *
 * Aluno:     Kristian Magnani dos Santos (kristian@ufmg.br)
 * Matricula: 2004212637
 * Curso:     Mestrado em Ciencia da Computacao
 * Data:      13.05.2004
 *
 * Este arquivo contém:
 * -> Implementacao dos principais algoritmos de ordenacao,
 *    incluindo:
 *      . Forca Bruta
 *      . BMH
 *      . BMHS
 *      . Shift-And Exato
 *      . Shift-And Aproximado
 *      . Programacao Dinamica (Aproximado)
 * -> Funcao principal "main" que faz a analise da entrada e
 *    aciona a busca apropriada segundo as opcoes escolhidas.
 */

/**
 * Inclui funcoes basicas de entrada e saida e de tempo.
 */
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

/**
 * Definicao de constantes.
 */
#define MaxTamTexto     52428800
#define MaxTamPadrao    256
#define Maxchar         256
#define NumMaxErros     10

/**
 * Modos de funcionamento. SHOW imprime linhas onde ha
 * ocorrencia do padrao, ao passo que COUNT imprime
 * estatisticas da busca.
 */
#define MODE_COUNT 1
#define MODE_SHOW  2

/**
```

```
 * Tipo abstrato de dados para texto e padrao.
 */
typedef unsigned char TipoTexto[MaxTamTexto];
typedef unsigned char TipoPadrao[MaxTamPadrao];

/**
 * Tabela para BMH e BMHS.
 */
typedef long TabelaBMHouBMHS[Maxchar + 1];
TabelaBMHouBMHS d;

/**
 * Tabela para Shift-And Exato e Aproximado.
 */
typedef long TabelaShiftAnd[Maxchar];
TabelaShiftAnd Masc;

/**
 * Estruturas usadas para cronometrar tempo gasto nas
 * buscas.
 */
struct timeval t_inicio, t_fim;

/**
 * Algoritmo de busca por forca bruta, baseado na implementacao de
 * [Ziviani2004]. Algumas alteracoes realizadas incluem a passagem de
 * m e n por copia, alteracao de tipo retorno, inclusao de modos de
 * operacao e remocao de efeitos colaterais (impressao em tela direta).
 */
int ForcaBruta(TipoTexto T, long n, TipoPadrao P, long m, short modo)
{
  long i, j, k;
  int count = 0;
  for (i = 1; i <= (n - m + 1); i++)
  {
    k = i;
    j = 1;
    while (T[k-1] == P[j-1] && j <= m)
    {
      j++; k++;
    }
    if (j > m)
      if (modo == MODE_SHOW) return 1; else count++;
  }
  return count;
}

/**
 * Monta a tabela para o algoritmo BMH.
```

```
 */
void MontaTabelaBMH(TipoTexto T, long n, TipoPadrao P, long m, short modo)
{
  int j;
  for (j = 0; j <= Maxchar; j++)
    d[j] = m;
  for (j = 1; j < m; j++)
    d[P[j-1]] = m - j;
}

/**
 * Algoritmo de busca BMH, baseado na implementacao de
 * [Ziviani2004]. Algumas alteracoes realizadas incluem a passagem de
 * m e n por copia, alteracao de tipo retorno, inclusao de modos de
 * operacao e remocao de efeitos colaterais (impressao em tela direta).
 */
int BMH(TipoTexto T, long n, TipoPadrao P, long m, short modo)
{
  long i, j, k;
  int count = 0;
  i = m;
  while (i <= n)    /*-- Pesquisa --*/
  {
    k = i;
    j = m;
    while (T[k-1] == P[j-1] && j > 0)
    {
      k--;
      j--;
    }
    if (j == 0)
      if (modo == MODE_SHOW) return 1; else count++;
    i += d[T[i-1]];
  }
  return count;
}

/**
 * BMH COM NUMERO DE COMPARACOES
 * Algoritmo de busca BMH, baseado na implementacao de
 * [Ziviani2004]. Algumas alteracoes realizadas incluem a passagem de
 * m e n por copia, alteracao de tipo retorno, inclusao de modos de
 * operacao e remocao de efeitos colaterais (impressao em tela direta).
 */
int BMH_COMPARACOES(TipoTexto T, long n, TipoPadrao P, long m)
{
  long i, j, k;
  int comp = 0;
  i = m;
```

```
  while (i <= n)    /*-- Pesquisa --*/
  {
    k = i;
    j = m;
    while (T[k-1] == P[j-1] && j > 0)
    {
      comp++; /* Comparacao entre caracteres */
      k--;
      j--;
    }
    if (j == 0)
      printf(" Casamento na posicao: %3ld\n", k + 1);
    i += d[T[i-1]];
  }
  return comp;
}

/**
 * Monta a tabela para o algoritmo BMHS.
 */
void MontaTabelaBMHS(TipoTexto T, long n, TipoPadrao P, long m, short modo)
{
  int j;
  for (j = 0; j <= Maxchar; j++)
    d[j] = m + 1;
  for (j = 1; j <= m; j++)
    d[P[j-1]] = m - j + 1;
}

/**
 * Algoritmo de busca BMHS, baseado na implementacao de
 * [Ziviani2004]. Algumas alteracoes realizadas incluem a passagem de
 * m e n por copia, alteracao de tipo retorno, inclusao de modos de
 * operacao e remocao de efeitos colaterais (impressao em tela direta).
 */
int BMHS(TipoTexto T, long n, TipoPadrao P, long m, short modo)
{
  long i, j, k;
  int count = 0;
  i = m;
  /*gettimeofday(&t_inicio, NULL);*/
  while (i <= n) /*-- Pesquisa --*/
  {
    k = i;
    j = m;
    while (T[k-1] == P[j-1] && j > 0)
    {
      k--;
      j--;
```

```
    }
    if (j == 0)
      if (modo == MODE_SHOW) return 1; else count++;
    i += d[T[i]];
  }
  return count;
}

/**
 * BMHS COM NUMERO DE COMPARACOES
 * Algoritmo de busca BMHS, baseado na implementacao de
 * [Ziviani2004]. Algumas alteracoes realizadas incluem a passagem de
 * m e n por copia, alteracao de tipo retorno, inclusao de modos de
 * operacao e remocao de efeitos colaterais (impressao em tela direta).
 */
int BMHS_COMPARACOES(TipoTexto T, long n, TipoPadrao P, long m)
{
  long i, j, k;
  int comp = 0;
  i = m;
  while (i <= n) /*-- Pesquisa --*/
  {
    k = i;
    j = m;
    while (T[k-1] == P[j-1] && j > 0)
    {
      comp++; /* Comparacao entre caracteres */
      k--;
      j--;
    }
    if (j == 0)
      printf(" Casamento na posicao: %3ld\n", k + 1);
    i += d[T[i]];
  }
  return comp;
}


/**
 * Monta a tabela para o algoritmo Shift-And Exato.
 */
void MontaTabelaShiftAndExato(TipoTexto T, long n, TipoPadrao P, long m, short modo)
{
  long i;
  for (i = 0; i < Maxchar; i++)
    Masc[i] = 0;
  for (i = 1; i <= m; i++)
  {
    Masc[P[i-1] + 127] |= 1 << (m - i);
```

```
  }
}

/**
 * Algoritmo de busca Shift-And Exato, baseado na implementacao de
 * [Ziviani2004]. Algumas alteracoes realizadas incluem a passagem de
 * m e n por copia, alteracao de tipo retorno, inclusao de modos de
 * operacao e remocao de efeitos colaterais (impressao em tela direta).
 */
int ShiftAndExato(TipoTexto T, long n, TipoPadrao P, long m, short modo)
{
  long i;
  int count = 0;
  long R = 0;
  for (i = 0; i < n; i++)
  {
    R = ((((unsigned long)R) >> 1) |
          (1 << (m - 1))) & Masc[T[i] + 127];
    if ((R & 1) != 0)
      if (modo == MODE_SHOW) return 1; else count++;
  }
  return count;
}

/**
 * Monta a tabela para o algoritmo Shift-And Aproximado.
 */
void MontaTabelaShiftAndAproximado(TipoTexto T, long n, TipoPadrao P,
  long m, short modo)
{
  long i;
  for (i = 0; i < Maxchar; i++)
    Masc[i] = 0;
  for (i = 1; i <= m; i++)
  {
    Masc[P[i-1] + 127] |= 1 << (m - i);
  }
}

/**
 * Algoritmo de busca Shift-And Aproximado, baseado na implementacao de
 * [Ziviani2004]. Algumas alteracoes realizadas incluem a passagem de
 * m e n por copia, alteracao de tipo retorno, inclusao de modos de
 * operacao e remocao de efeitos colaterais (impressao em tela direta).
 */
int ShiftAndAproximado(TipoTexto T, long n, TipoPadrao P, long m, long k, short modo)
{
  long Masc[Maxchar];
  long i, j, Ri, Rant, Rnovo;
```

```
  long R[NumMaxErros + 1];
  int count = 0;
  R[0] = 0;
  Ri = 1 << (m - 1);
  for (j = 1; j <= k; j++)
    R[j] = (1 << (m - j)) | R[j-1];
  for (i = 0; i < n; i++)
  {
    Rant = R[0];
    Rnovo = ((((unsigned long)Rant) >> 1) | Ri) & Masc[T[i] + 127];
    R[0] = Rnovo;
    for (j = 1; j <= k; j++)
  {
      Rnovo = ((((unsigned long)R[j]) >> 1) & Masc[T[i] + 127])
                  | Rant | (((unsigned long)(Rant | Rnovo)) >> 1);
      Rant = R[j];
      R[j] = Rnovo | Ri;
    }
    if ((Rnovo & 1) != 0)
      if (modo == MODE_SHOW) return 1; else count++;
  }
  return count;
}

/**
 * Algoritmo de busca por Programacao Dinamica (Aproximado), baseado na
 * implementacao de
 * [Navarro,Raffinot]. Algumas alteracoes realizadas incluem a passagem de
 * m e n por copia, alteracao de tipo retorno, inclusao de modos de
 * operacao e remocao de efeitos colaterais (impressao em tela direta).
 */
int DynamicProgrammingAprxSearch(TipoTexto T, long n, TipoPadrao P, long m,
  long k, short modo)
{
  int C[MaxTamPadrao + 1];
  int count = 0, i;
  long lact, pos, pC, nC;

  /* Pre-processamento */
  for (i = 0; i<= m; i++)
  {
    C[i] = i;
  }
  lact = k + 1;

  /* Busca */
  for (pos = 1; pos <= n; pos++)
  {
    pC = 0;
```

```
    nC = 0;
    for (i = 1; i <= lact; i++)
    {
      if (P[i-1] == T[pos-1])
      {
        nC = pC;
      }
      else
      {
        if (pC < nC) nC = pC;
        if (C[i] < nC) nC = C[i];
nC++;
      }
      pC = C[i];
      C[i] = nC;
    }
    while (C[lact] > k)
    {
      lact--;
    }
    if (lact == m)
    {
      if (modo == MODE_SHOW) return 1; else count++;
    }
    else
    {
      lact++;
    }
  }

  return count;
}

/**
 * Funcao auxiliar para leitura de linha de arquivo.
 */
int leLinha(FILE *fp, char* buffer)
{
  int i = 0;
  char c;

  c = getc(fp);
  while ((c != '\n') && (c != EOF))
  {
    buffer[i] = c;
    i++;
    c = getc(fp);
  }
  buffer[i] = '\0';
```

```
  /*printf("length = %d\n", i);*/

  return i;
}

/**
 *
 */
int getAproxSize(char* s)
{
  char num[256];
  int i = 0;
  while (s[0] >= '0' && s[0] <= '9')
  {
    num[i++] = s[0];
    s++;
  }
  s[i] = '\0';
  return atoi(num);
}

/**
 * Funcao principal "main" que faz a analise da entrada e
 * aciona a busca apropriada segundo as opcoes escolhidas.
 */
int main(int argc, char **argv)
{
  /* Arquivos (texto e padrao) */
  FILE *fp, *fpattern;

  /* Buffer de entrada (texto) */
  TipoTexto linha;

  /* Opcoes de linha de comando */
  char *opcoes;

  /* Tamanho do texto e do padrao */
  long n, m, k;

  /* Contadores de ocorrencias */
  int ret, totalOcorrencias, i = 0, totalComp1, totalComp2;

  /* struct timeval t_inicio, t_fim; */

  /* Padroes de procura */
  char padroes[MaxTamPadrao][MaxTamPadrao];

  /* Tamanhos dos padores */
  long m_v[MaxTamPadrao];
```

```
  /* Tempo total transcorrido */
  long t_us;

  char* k_src;

  opcoes = NULL;

  /*for (i = 0; i < argc; i++) printf("%s\n", argv[i]);*/
  /*printf("argc = %d\n", argc);*/

  /* Se existem argumentos suficientes... */
  if (argc > 2)
  {
    /* Se existem opcoes ativadas por chaves... */
    if (argc == 4)
    {
      opcoes = argv[argc - 3];

      /* Verifica se padrao vem literalmente ou se deve ser lido de arquivo */
      if (strchr(opcoes, 'f'))
      {
        /* Padroes sao lidos de arquivo, um padrao por linha */
fpattern = fopen(argv[argc - 2], "r");
i = 0;
while (!feof(fpattern))
{
  m_v[i] = leLinha(fpattern, padroes[i]);
  if (m_v[i] > 0)
  {
    i++;
  }
}
      }
      else
      {
        /* Padrao unico eh fornecido na propria linha de comando */
m_v[0] = strlen(argv[argc - 2]);
m_v[1] = 0;
strcpy(padroes[0], argv[argc - 2]);
      }

      /* Chave -C indica modo COMPARACOES (numero
         de comparacoes de caracteres para BMH e BMHS) */
      if (strchr(opcoes, 'C'))
      {
        /* Busca por BMH */
        printf("BMH\n\n");
   totalComp1 = 0;
```

```
t_us = 0;
fp = fopen(argv[argc - 1], "r");
        while (!feof(fp))
        {
          n = leLinha(fp, linha);
          i = 0;
  while (m_v[i] != 0)
          {
    MontaTabelaBMH(linha, n, padroes[i], m_v[i], MODE_COUNT);
    ret = BMH_COMPARACOES(linha, n, padroes[i], m_v[i]);
            totalComp1 += ret;
            i++;
  }
        }
        fclose(fp);

        /* Busca por BMHS */
        printf("BMHS\n\n");
   totalComp2 = 0;
t_us = 0;
fp = fopen(argv[argc - 1], "r");
        while (!feof(fp))
        {
          n = leLinha(fp, linha);
          i = 0;
  while (m_v[i] != 0)
          {
    MontaTabelaBMHS(linha, n, padroes[i], m_v[i], MODE_COUNT);
    ret = BMHS_COMPARACOES(linha, n, padroes[i], m_v[i]);
            totalComp2 += ret;
            i++;
  }
        }
        fclose(fp);

printf("Comparacoes entre caracteres: \n BMH:\t %d \n BMHS:\t %d \n",
       totalComp1, totalComp2);
      }
      /* Chave -c indica modo COUNT (exibe estatisticas) */
      else if (strchr(opcoes, 'c'))
      {
        /* Modo COUNT: mostra numero total de ocorrencias +
   indicadores de performance */
        printf("\n\nModo COUNT\n\n");

        if (strchr(opcoes, 'F'))
        {
          /* Busca por FORCA_BRUTA */
  printf("FORCA_BRUTA\n\n");
```

```
  totalOcorrencias = 0;
  t_us = 0;
  fp = fopen(argv[argc - 1], "r");
          while (!feof(fp))
  {
            n = fread(linha, sizeof(char), MaxTamTexto, fp);
            i = 0;
    while (m_v[i] != 0)
            {
              gettimeofday(&t_inicio, NULL);
              ret = ForcaBruta(linha, n, padroes[i], m_v[i], MODE_COUNT);
              gettimeofday(&t_fim, NULL);
      totalOcorrencias += ret;
              t_us += 1000000 * (t_fim.tv_sec - t_inicio.tv_sec) +
        (t_fim.tv_usec - t_inicio.tv_usec);
      i++;
            }
  }
  printf("OCORRENCIAS: %d\tTEMPO: %d us\n", totalOcorrencias, t_us);
          fclose(fp);
        }
        else if (strchr(opcoes, 'B'))
        {
          /* Busca por BMH */
  printf("BMH\n\n");
  totalOcorrencias = 0;
  t_us = 0;
  fp = fopen(argv[argc - 1], "r");
          while (!feof(fp))
  {
            n = fread(linha, sizeof(char), MaxTamTexto, fp);
            i = 0;
    while (m_v[i] != 0)
            {
      MontaTabelaBMH(linha, n, padroes[i], m_v[i], MODE_COUNT);
              gettimeofday(&t_inicio, NULL);
              ret = BMH(linha, n, padroes[i], m_v[i], MODE_COUNT);
              gettimeofday(&t_fim, NULL);
      totalOcorrencias += ret;
              t_us += 1000000 * (t_fim.tv_sec - t_inicio.tv_sec) +
        (t_fim.tv_usec - t_inicio.tv_usec);
      i++;
            }
  }
  printf("OCORRENCIAS: %d\tTEMPO: %d us\n", totalOcorrencias, t_us);
          fclose(fp);
}
        else if (strchr(opcoes, 'Y'))
        {
```

```
        /* Busca por BMHS */
printf("BMHS\n\n");
totalOcorrencias = 0;
t_us = 0;
fp = fopen(argv[argc - 1], "r");
        while (!feof(fp))
{
          n = fread(linha, sizeof(char), MaxTamTexto, fp);
          i = 0;
  while (m_v[i] != 0)
          {
    MontaTabelaBMHS(linha, n, padroes[i], m_v[i], MODE_COUNT);
            gettimeofday(&t_inicio, NULL);
            ret = BMHS(linha, n, padroes[i], m_v[i], MODE_COUNT);
            gettimeofday(&t_fim, NULL);
    totalOcorrencias += ret;
            t_us += 1000000 * (t_fim.tv_sec - t_inicio.tv_sec) +
      (t_fim.tv_usec - t_inicio.tv_usec);
    i++;
          }
}
printf("OCORRENCIAS: %d\tTEMPO: %d us\n", totalOcorrencias, t_us);
        fclose(fp);
      }
      else if (strchr(opcoes, 'S'))
      {
        /* Busca por SHIFT_AND_EXATO */
printf("SHIFT_AND_EXATO\n\n");
totalOcorrencias = 0;
t_us = 0;
fp = fopen(argv[argc - 1], "r");
        while (!feof(fp))
{
          n = fread(linha, sizeof(char), MaxTamTexto, fp);
          i = 0;
  while (m_v[i] != 0)
          {
    MontaTabelaShiftAndExato(linha, n, padroes[i], m_v[i], MODE_COUNT);
            gettimeofday(&t_inicio, NULL);
            ret = ShiftAndExato(linha, n, padroes[i], m_v[i], MODE_COUNT);
            gettimeofday(&t_fim, NULL);
    totalOcorrencias += ret;
            t_us += 1000000 * (t_fim.tv_sec - t_inicio.tv_sec) +
      (t_fim.tv_usec - t_inicio.tv_usec);
    i++;
          }
}
printf("OCORRENCIAS: %d\tTEMPO: %d us\n", totalOcorrencias, t_us);
        fclose(fp);
```

```
        }
        else if (k_src = (char*)strchr(opcoes, 'A'))
        {
          /* Busca por SHIFT_AND_APROXIMADO */
  k = getAproxSize(++k_src);
  printf("SHIFT_AND_APROXIMADO\n\n");
  totalOcorrencias = 0;
  t_us = 0;
  fp = fopen(argv[argc - 1], "r");
          while (!feof(fp))
  {
            n = fread(linha, sizeof(char), MaxTamTexto, fp);
            i = 0;
    while (m_v[i] != 0)
            {
      MontaTabelaShiftAndAproximado(linha, n, padroes[i], m_v[i], MODE_COUNT);
              gettimeofday(&t_inicio, NULL);
              ret = ShiftAndAproximado(linha, n, padroes[i], m_v[i], k, MODE_COUNT);
              gettimeofday(&t_fim, NULL);
      totalOcorrencias += ret;
              t_us += 1000000 * (t_fim.tv_sec - t_inicio.tv_sec) +
         (t_fim.tv_usec - t_inicio.tv_usec);
      i++;
            }
  }
  printf("OCORRENCIAS: %d\tTEMPO: %d us\n", totalOcorrencias, t_us);
          fclose(fp);
}
        else if (k_src = (char*)strchr(opcoes, 'D'))
        {
          /* Busca por PROOGRAMACA_DINAMICA_APROXIMADO */
  k = getAproxSize(++k_src);
  printf("PROOGRAMACA_DINAMICA_APROXIMADO\n\n");
  totalOcorrencias = 0;
  t_us = 0;
  fp = fopen(argv[argc - 1], "r");
          while (!feof(fp))
  {
            n = fread(linha, sizeof(char), MaxTamTexto, fp);
            i = 0;
    while (m_v[i] != 0)
            {
      gettimeofday(&t_inicio, NULL);
              ret = DynamicProgrammingAprxSearch(linha, n, padroes[i], m_v[i], k,
        MODE_COUNT);
              gettimeofday(&t_fim, NULL);
      totalOcorrencias += ret;
              t_us += 1000000 * (t_fim.tv_sec - t_inicio.tv_sec) +
         (t_fim.tv_usec - t_inicio.tv_usec);
```

```
      i++;
            }
  }
  printf("OCORRENCIAS: %d\tTEMPO: %d us\n", totalOcorrencias, t_us);
          fclose(fp);
}
      }
      else
      {
        /* Modo SHOW: imprime na tela linha com ocorrencia */
        printf("\n\nModo SHOW\n\n");

        if (strchr(opcoes, 'F'))
        {
          /* Busca por FORCA_BRUTA */
          printf("FORCA_BRUTA\n\n");
     totalOcorrencias = 0;
  t_us = 0;
  fp = fopen(argv[argc - 1], "r");
          while (!feof(fp))
          {
            n = leLinha(fp, linha);
            i = 0;
    while (m_v[i] != 0)
            {
      gettimeofday(&t_inicio, NULL);
              ret = ForcaBruta(linha, n, padroes[i], m_v[i], MODE_SHOW);
              gettimeofday(&t_fim, NULL);
              totalOcorrencias += ret;
              t_us += 1000000 * (t_fim.tv_sec - t_inicio.tv_sec) +
         (t_fim.tv_usec - t_inicio.tv_usec);
      if (ret) printf("%s\n", linha);
      i++;
    }
          }
  printf("\nFORCA_BRUTA \t(t = %d us)\n", t_us);
          fclose(fp);
}
        else if (strchr(opcoes, 'B'))
        {
          /* Busca por BMH */
          printf("BMH\n\n");
     totalOcorrencias = 0;
  t_us = 0;
  fp = fopen(argv[argc - 1], "r");
          while (!feof(fp))
          {
            n = leLinha(fp, linha);
            i = 0;
```

```
  while (m_v[i] != 0)
          {
    MontaTabelaBMH(linha, n, padroes[i], m_v[i], MODE_COUNT);
    gettimeofday(&t_inicio, NULL);
            ret = BMH(linha, n, padroes[i], m_v[i], MODE_SHOW);
            gettimeofday(&t_fim, NULL);
            totalOcorrencias += ret;
            t_us += 1000000 * (t_fim.tv_sec - t_inicio.tv_sec) +
      (t_fim.tv_usec - t_inicio.tv_usec);
    if (ret) printf("%s\n", linha);
    i++;
  }
        }
printf("\nBMH \t(t = %d us)\n", t_us);
        fclose(fp);
      }
      else if (strchr(opcoes, 'Y'))
      {
        /* Busca por BMHS */
        printf("BMHS\n\n");
   totalOcorrencias = 0;
t_us = 0;
fp = fopen(argv[argc - 1], "r");
        while (!feof(fp))
        {
          n = leLinha(fp, linha);
          i = 0;
  while (m_v[i] != 0)
          {
    MontaTabelaBMHS(linha, n, padroes[i], m_v[i], MODE_COUNT);
    gettimeofday(&t_inicio, NULL);
            ret = BMHS(linha, n, padroes[i], m_v[i], MODE_SHOW);
            gettimeofday(&t_fim, NULL);
            totalOcorrencias += ret;
            t_us += 1000000 * (t_fim.tv_sec - t_inicio.tv_sec) +
      (t_fim.tv_usec - t_inicio.tv_usec);
    if (ret) printf("%s\n", linha);
    i++;
  }
        }
printf("\nBMHS \t(t = %d us)\n", t_us);
        fclose(fp);
      }
      else if (strchr(opcoes, 'S'))
      {
        /* Busca por SHIFT_AND_EXATO */
        printf("SHIFT_AND_EXATO\n\n");
   totalOcorrencias = 0;
t_us = 0;
```

```
fp = fopen(argv[argc - 1], "r");
        while (!feof(fp))
        {
          n = leLinha(fp, linha);
          i = 0;
  while (m_v[i] != 0)
          {
    MontaTabelaShiftAndExato(linha, n, padroes[i], m_v[i], MODE_COUNT);
    gettimeofday(&t_inicio, NULL);
            ret = ShiftAndExato(linha, n, padroes[i], m_v[i], MODE_SHOW);
            gettimeofday(&t_fim, NULL);
            totalOcorrencias += ret;
            t_us += 1000000 * (t_fim.tv_sec - t_inicio.tv_sec) +
      (t_fim.tv_usec - t_inicio.tv_usec);
    if (ret) printf("%s\n", linha);
    i++;
  }
        }
printf("\nSHIFT_AND_EXATO \t(t = %d us)\n", t_us);
        fclose(fp);
      }
      else if (k_src = (char*)strchr(opcoes, 'A'))
      {
        /* Busca por SHIFT_AND_APROXIMADO */
        printf("SHIFT_AND_APROXIMADO\n\n");
        k = getAproxSize(++k_src);
totalOcorrencias = 0;
t_us = 0;
fp = fopen(argv[argc - 1], "r");
        while (!feof(fp))
        {
          n = leLinha(fp, linha);
          i = 0;
  while (m_v[i] != 0)
          {
    MontaTabelaShiftAndExato(linha, n, padroes[i], m_v[i], MODE_COUNT);
    gettimeofday(&t_inicio, NULL);
            ret = ShiftAndAproximado(linha, n, padroes[i], m_v[i], k, MODE_SHOW);
            gettimeofday(&t_fim, NULL);
            totalOcorrencias += ret;
            t_us += 1000000 * (t_fim.tv_sec - t_inicio.tv_sec) +
      (t_fim.tv_usec - t_inicio.tv_usec);
    if (ret) printf("%s\n", linha);
    i++;
  }
        }
printf("\nSHIFT_AND_APROXIMADO \t(t = %d us)\n", t_us);
        fclose(fp);
      }
```

```
        else if (k_src = (char*)strchr(opcoes, 'D'))
        {
          /* Busca por PROOGRAMACA_DINAMICA_APROXIMADO */
          printf("PROOGRAMACA_DINAMICA_APROXIMADO\n\n");
          k = getAproxSize(++k_src);
  totalOcorrencias = 0;
  t_us = 0;
  fp = fopen(argv[argc - 1], "r");
          while (!feof(fp))
          {
            n = leLinha(fp, linha);
            i = 0;
    while (m_v[i] != 0)
            {
      gettimeofday(&t_inicio, NULL);
              ret = DynamicProgrammingAprxSearch(linha, n, padroes[i], m_v[i], k,
        MODE_SHOW);
              gettimeofday(&t_fim, NULL);
              totalOcorrencias += ret;
              t_us += 1000000 * (t_fim.tv_sec - t_inicio.tv_sec) +
        (t_fim.tv_usec - t_inicio.tv_usec);
      if (ret) printf("%s\n", linha);
      i++;
    }
          }
  printf("\nPROOGRAMACA_DINAMICA_APROXIMADO \t(t = %d us)\n", t_us);
          fclose(fp);        }
      }
    }
    else
    {
      /* Nao foram passadas opcoes. Padrao = COMPARACAO ENTRE BMH E BMHS */
    }

    /*}
    else
    {
      printf("ERRO: Arquivo %s não foi encontrado.\n", argv[argc - 1]);
    }*/
  }
  else
  {
    printf("ERRO: Número insuficiente de argumentos.\n");
  }

  return 0;
}
```