Universidade Federal de Minas Gerais

Departamento de Ciência da Computação

Projeto e Análise de Algoritmos
Casamento de Padrões

Trabalho disponível em:
http://www.dcc.ufmg.br/~fred/paa/tp3

Aluno: Frederico Paiva Quintão

Monitor: Fabiano Cupertino Botelho

Professor: Nivio Ziviani

Belo Horizonte
27 de maio de 2006

Sumário

Lista de Figuras

Lista de Tabelas

Introdução

Neste trabalho, são apresentadas abordagens diferentes para o tratamento do problema de Casamento de Padrões. Na primeira parte do trabalho, são apresentados dois algoritmos para realizar a busca no texto permitindo erros: uma variante do algoritmo de Programação Dinâmica para o problema da Distância de Edição e o algoritmo *Shift-And*. Na segunda parte, utiliza-se o algoritmo *Boyer-Moore-Horspool* para busca em textos comprimidos e não comprimidos.

Para ambas as abordagens, são apresentados resultados computacionais, tanto em termos do número de operações realizadas quanto o tempo de relógio.

Antes de iniciar a apresentação dos algoritmos e dos resultados computacionais, serão apresentadas algumas Decisões de Projeto, principalmente relacionadas a entrada e saída (E/S) de dados. Estas decisões serão utilizadas para guiar o desenvolvimento dos algoritmos.

Uma cópia deste documento pode ser obtida em:
`http://www.dcc.ufmg.br/~fred/paa/tp3`.

Os códigos-fonte desenvolvidos podem ser descarregados através do endereço:
`http://www.dcc.ufmg.br/~fred/paa/tp3/fontes`.

1 Decisões de projeto

1.1 Análise de Entrada e Saída (E/S)

Nesta seção, são apresentadas algumas análises que levarão a decisões importantes do projeto de algoritmos de busca em texto. O objetivo é avaliar diferentes abordagens de entrada e saída (E/S) e decidir quais são as melhores estratégias.

Com o objetivo de avaliar diferentes estratégias, foram desenvolvidos dois programas simples, que fazem uso das diretivas **fgetc** e **fgets**. De acordo com o [2], estas operações trabalham da seguinte maneira:

```
#include <stdio.h>
  char *fgets( char *str, int num, FILE *stream );
```

Esta operação lê até `num-1` caracteres do arquivo `stream` e coloca estes caracteres na *string* `str`. `fgets` pára quando encontra o fim de uma linha.

```
#include <stdio.h>
  int fgetc( FILE *stream );
```

Esta operação retorna o próximo caractere do `stream`, ou `EOF` (*end o file*) se o final do arquivo é atingido ou um erro é encontrado.

Foram então desenvolvidos dois programas simples, que utilizam as diretivas acima:

linha.c: Realiza a leitura do arquivo ASCII linha por linha. Em seguida, cada caractere da linha é visitado e é relizada uma operação simples sobre ele. Dessa maneira, **todos** os caracteres do texto são lidos e processados. A implementação segue na Listagem 1.

Listagem 1: linha.c

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int main (int argc, char ** argv)
{
        FILE *fp;
        char v_Aux;
        char linha[5000];
        int v_Contador=0;
        int v_Tamanho = 0;

        fp = fopen(argv[1], "r");

        while(fgets(linha, 5000, fp))
        {
                v_Tamanho = strlen(linha);
                for (v_Contador = 0; v_Contador < v_Tamanho; v_Contador
                    ++)
                {
                        v_Aux = linha[v_Contador];
                        v_Aux = v_Aux+1;
                }
        }
        return 0;
}
```

caractere.c: Este programa realiza a leitura de um arquivo caractere por caractere. Cada caractere é lido e em seguida é realizada uma operação simples sobre ele. A implementação segue na Listagem 2.

Listagem 2: caractere.c

```
#include<stdio.h>
#include<stdlib.h>


int main (int argc, char ** argv)
{
        FILE *fp;
        char v_Aux;

        fp = fopen(argv[1], "r");

        while(!(feof(fp)))
        {
                v_Aux = fgetc(fp);
                v_Aux = v_Aux + 1;
        }
        return 0;
}
```

Do modo como os programas foram implementados, eles lêem todo o arquivo e fazem uma operação simples sobre todos os seus caracteres, de modo que o resultado final obtido é sempre o mesmo e então pode-se fazer uma comparação justa entre eles.

Foram feitas então medições de tempo com cada programa, sobre os seguintes arquivos: `wsj89` (2.9MB), `wsj88_10` (9.7MB), `wsj88_20` (20MB) e `wsj88` (105MB). Os testes foram feitos em uma máquina Intel Pentium IV, de 2.4GHz e 1GB de memória RAM[1].

A Figura 1 ilustra os resultados obtidos, plotados com os dados das Tabelas 1 e 2.

Arquivo	**Real**	**Usuário**	**Sistema**
wsj89	0.034	0.033	0.001
wsj88_10	0.113	0.102	0.009
wsj88_20	0.219	0.199	0.020
wsj88	1.172	1.082	0.086

Tabela 1: **linha.c**: resultados.

Arquivo	**Real**	**Usuário**	**Sistema**
wsj89	0.17	0.13	0.01
wsj88_10	1.28	0.45	0.039
wsj88_20	1.73	0.91	0.08
wsj88	7.069	4.78	0.083

Tabela 2: **caractere.c**: resultados.

Estes resultados deixam claro que a estratégia de ler linha por linha do arquivo é muito mais interessante. Algumas razões para essa superioridade podem ser:

- Menor número de interrupções gerados no nível do sistema operacional.
- O fato de que uma linha provavelmente tem uma localidade de referência muito maior do que um caractere.
- A possibilidade de operar em memória à medida de que a linha já esteja lida.

Com base nos resultados obtidos, foram tomadas as seguintes decisões de projeto:

- O texto não será colocado todo em memória, tendo em vista que isto pode se tornar inviável caso o arquivo seja muito grande. Além disso, o tempo para colocar o arquivo em memória é alto, caso seja feita a leitura caractere por caractere.
- A leitura do texto será feita linha por linha, pois esta estratégia revelou-se mais interessante. No caso de casamento aproximado, será feito um tratamento especial para a possibilidade de se casar um padrão que começa em uma linha e termina em outra, quando for feita a busca permitindo 1 ou mais erros.

[1] Todos os testes apresentados neste trabalho foram feitas em máquinas com esta configuração.

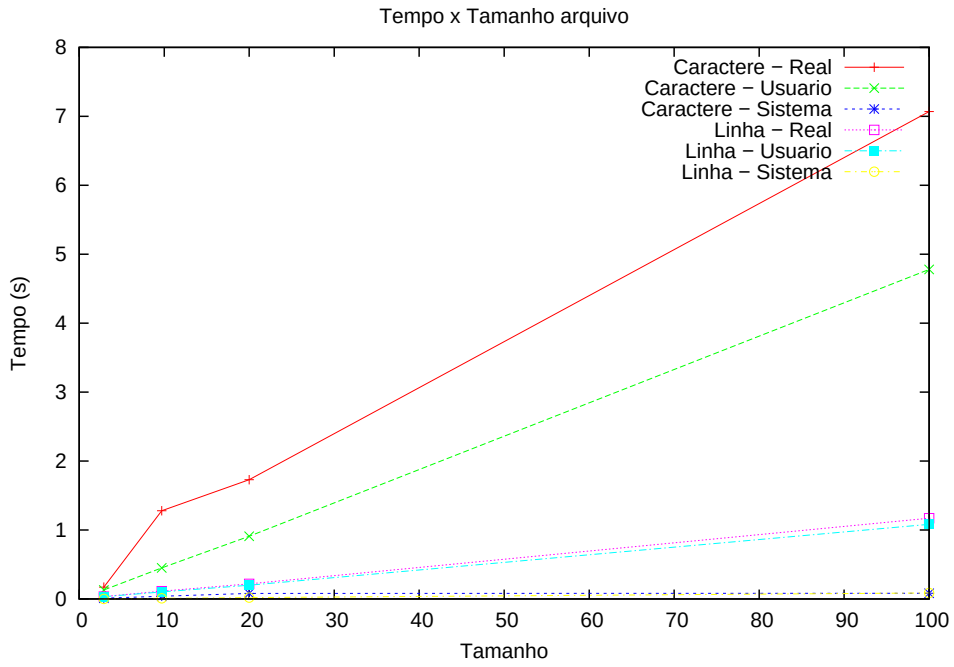


Figura 1: Diferença de tempos entre os sistemas.

1.2 Organização de arquivos e módulos

Durante o desenvolvimento foi levantada a seguinte questão: *Os módulos deverão estar integrados ou devem ser implementados separadamente?* Optou-se pela implementação em separado, pelos seguintes motivos:

- Re-utilização do código fornecido por [4].
- Possibilidade de otimização de código por parte do `gcc`. Dada a menor independência e tamanho do fonte, o `gcc` pode otimizar melhor o código, inclusive através de diretivas `-Oi`, $i \in \{1, 2, 3, \ldots, 9\}$.
- Menor binário gerado e possibilidade de aproveitamento de cada algoritmo individualmente para uso prático.

Foi formada então a estrutura de diretórios apresentada na Figura 2, que pode ser útil para o momento da avaliação do funcionamento dos algoritmos.

Dentro da pasta `comprimido`, encontram-se as seguintes pastas:

1. `lbmh`: contém a primeira implementação do BMH.
2. `bmh`: contém uma implementação alternativa do BMH (ver Seção 3.1).
3. `huffman`: implementação de compressão, descompressão e busca em arquivos comprimidos.

Dentro da pasta `nocomprimido`, encontram-se as seguintes pastas:

1. `ldinamica`: contém a implementação do Algoritmo de Sellers.
2. `lshiftand`: contém a implementação do *Shift-And* aproximado.

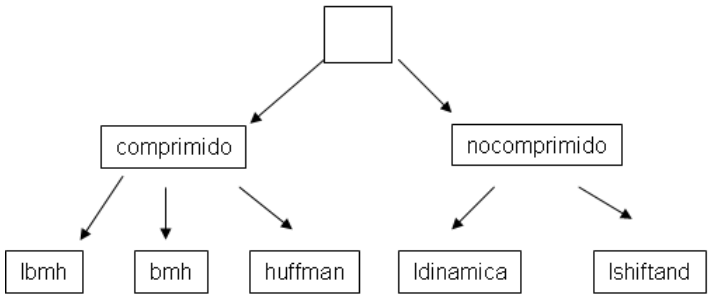


Figura 2: Árvore de diretórios do sistema.

1.3 Interface com o usuário

Outro ponto de discussão é a interface com o usuário, tanto para entrada quanto para saída de dados. Foram utilizadas as diretivas de `getopt` da linguagem C para fornecer entrada de dados (exceto no caso do sistema de busca em texto comprimido). Cada aplicação desenvolvida terá uma seção explicativa sobre a sintaxe de chamada, e portanto ela não é descrita aqui. Por outro lado, com relação à interface de saída, foram tomadas as seguintes decisões:

- Cada linha do texto onde ocorre um padrão é apresentada com o número da linha seguido da linha propriamente dita.
- Caso um padrão ocorra mais de uma vez em uma linha, a linha será impressa somente uma vez. Além disso, aparecerá na frente do número da linha um identificador apresentando quantas vezes o padrão aparece na linha. Por exemplo, considere o seguinte texto de entrada:

  ```
  Meu nome é Frederico. Meu apelido é Fred.
  ```

  Caso seja procurado o padrão `Fred`, o resultado será:

  ```
  2 x [1]: Meu nome é Frederico. Meu apelido é Fred.
  ```

  indicando que o padrão aparece 2 vezes na linha 1.
- Para a realização dos testes, a operação de saída será considerada.

As decisões apresentadas nas três seções acima serão adotadas sempre que possível. Caso haja algum caso que contrarie alguma decisão, ele estará devidamente explicitado no texto.

2 Busca aproximada de padrões

2.1 Busca aproximada usando Programação Dinâmica

O algoritmo apresentado no Trabalho Prático II para o Problema da Distância de Edição pode ser facilmente adaptado para realizar a busca aproximada de padrões no texto, conforme sugerido por Navarro em [1].

Entretanto, do jeito que foi implementado, o algoritmo pode ser muito ineficiente com relação à complexidade de espaço, tendo em vista que ele é $O(|P||T|)$, o que pode se tornar inviável caso o texto seja grande e o padrão também. Por exemplo, para um texto de 100MB e um padrão de tamanho 5, tem-se que 500MB são necessários somente para alocar a matriz em memória. Caso o tamanho do padrão seja aumentado para 20, serão necessários 2GB para armazenar esta mesma matriz. Este valor está próximo do limiar confiável de alocação para sistemas Linux, por exemplo.

Com base nas observações acima, foi utilizado um outro algoritmo, também de Programação Dinâmica, atribuído a Sellers, datado de 1980.

2.2 Algoritmo para Casamento Aproximado de Sellers

Este algoritmo funciona da seguinte maneira: um vetor de custo C, com $|P| + 1$ posições, é inicializado com os seguintes valores: $C_i = i$. Então, o texto é processado caractere por caractere. Para cada caractere y lido do texto, as posições contíguas do vetor são atualizadas de acordo com a seguinte relação:

$$\begin{cases} C'_i = C_{i-1} & \text{se P[i]=y} \\ C_{i,j} = 1 + \min(C_{i-1}, C'_{i-1}, C_i) & \text{caso contrário} \end{cases}$$

O algoritmo de Sellers, em alto nível, é apresentado no Algoritmo 1.

2.3 Complexidade do algoritmo

Com relação ao tempo de processamento, o algoritmo é $O(k|T|)$, onde k é o número de erros permitidos e T é o tamanho do texto. É fácil observar esta complexidade, tendo em vista que o algoritmo possui dois *loops* aninhados, um baseado no tamanho do arquivo (leitura de cada caractere enquanto o arquivo não termina) e outro baseado na variável `lact`, que sempre mantém um tamanho que é $O|P|$.

Com relação ao espaço, o algoritmo foi implementado de modo a consumir a menor quantidade possível de memória. Isto é obtido eliminando-se a necessidade de o texto ficar em memória durante a execução, pois o mesmo não é alocado em uma matriz. Logo, como somente o padrão é armazenado em memória, a complexidade de espaço é $O(|P|)$.

É importante salientar a aplicação das decisoes de projeto no caso presente. O algoritmo foi implementado lendo linha por linha do arquivo. Foram feitos alguns testes utilizando-se esta implementação e uma outra que considera caractere por caractere. Os resultados foram os seguintes:

- O método que utiliza linha por linha aparentemente executa mais rapidamente. Em testes feitos no arquivo `wsj88` considerando o padrão `Columbia`, obteve-se

Algoritmo 1 Dinâmica(p_Padrao, p_NumeroErros, p_ArquivoEntrada)

```
int i, lact, pos, pC, nC, *C, v_TamanhoPadrao;
char y;
v_TamanhoPadrao = tamanho(p_Padrao);
para (i=0; i<=v_TamanhoPadrao; i++) faça
  C[i] = 1;
fim para
lact = p_NumeroErros + 1;
enquanto (!fim(p_ArquivoEntrada)) faça
  linha = proximaLinha(p_ArquivoEntrada);
  para (pos=1; pos <= tamanho(linha); pos++) faça
    y = linha[pos];
    para (i=1; i<=lact; i++) faça
      se p_Padrao[i] == y então
        nC = pC
      senão
        se pC < nC então
          nC = pC;
        fim se
        se C[i] < nC então
          nC = C[i];
        fim se
        nC = nC + 1;
      fim se
      pC = C[i];
      C[i] = nC;
    fim para
    enquanto (C[lact] > p_NumeroErros) faça
      lact = lact - 1;
    fim enquanto
    se lact = v_TamanhoPadrao então
      Reporta casamento em pos;
    senão
      lact = lact + 1;
    fim se
  fim para
fim enquanto
```

que o algoritmo de leitura de caractere por caractere é aproximadamente 1.5 vezes mais lento do que o de leitura por linha.

- O método que utiliza caractere por caractere possui a vantagem de deixar transparente para o desenvolvedor a sobreposição de linhas. Por exemplo, ele é capaz de encontrar padrões que comecem em uma linha e terminem em outra, caso exista a possibilidade de (pelo menos) 1 erro para a busca aproximada. Para o método de linha por linha, é necessário tratar este caso explicitamente. Entretanto, este tratamento é muito simples e não acarreta

prejuízo computacional. Para que ele seja feito, basta remover os caracteres de nova linha ('\n') da linha sendo lida, e manter sempre os valores correntes do vetor C e das variáveis $lact$, pC e nC. Por exemplo, considere que o texto abaixo é lido na entrada:

```
#Makefile TP3 - PAA
#Algoritmo de Programacao Dinamica para Casamento de padroes
```

É possível fazer uma busca com o seguinte padrão: $PAA\#Algoritmo$ (que começa na primeira linha e termina na segunda). O caractere '\n' é ignorado [2]. Basta realizar a busca permitindo um erro que o algoritmo fará o casamento. Esta observação é importante pois será aplicada também no algoritmo *Shift-And*, que será apresentado em seguida.

2.4 Implementação

2.4.1 Estruturas de dados utilizadas

A principal estrutura do algoritmo é o vetor C, que armazena todas as informações sobre transformações possíveis entre P e T. Este conjunto é armazenado sobre a forma de um `int*`.

2.4.2 Descrição dos módulos e métodos

O principal método da implementação é chamado `editDistance`, que recebe como principais parâmetros o arquivo de entrada, o padrão a ser buscado no arquivo e o número k de erros permitidos na busca. O método opera sobre o vetor C definido na Seção anterior, visando buscar o padrão no texto considerando k erros.

2.4.3 Utilização do sistema

Para utilizar o sistema, deve-se chamar o arquivo `dinamica.e` com a seguinte lista de parâmetros:

- `-e`: indica o número de erros aceitáveis na pesquisa aproximada
- `-f`: indica o nome do arquivo no qual se deseja fazer a busca
- `-p`: indica o padrão que se deseja buscar no texto

Por exemplo, a seguinte linha busca pelo padrão `CFLAGS` no arquivo de nome `Makefile` considerando uma pesquisa com 1 erro:

```
./dinamica.e -f Makefile -e 1 -p CFLAGS
```

[2] Acredita-se que esta decisão de projeto não irá diminuir a usabilidade do sistema pois em geral um usuário nunca deverá fazer uma consulta com um padrão que contenha o '\n'

2.5 Busca aproximada utilizando o algoritmo *Shift-And*

Shift-And é um algoritmo para casamento aproximado que utiliza **paralelismo de bit**. Este algoritmo simula um autômato não-determinista e foi apresentado por Wu e Manber em 1992. Trata-se de uma extensão do algoritmo *Shift-And* para casamento exato, proposto por Baeza-Yates e Gonnet em 1989.

O algoritmo simula um autômato não-determinista com k linhas capaz de detectar casamentos aproximados com até $k-1$ erros.

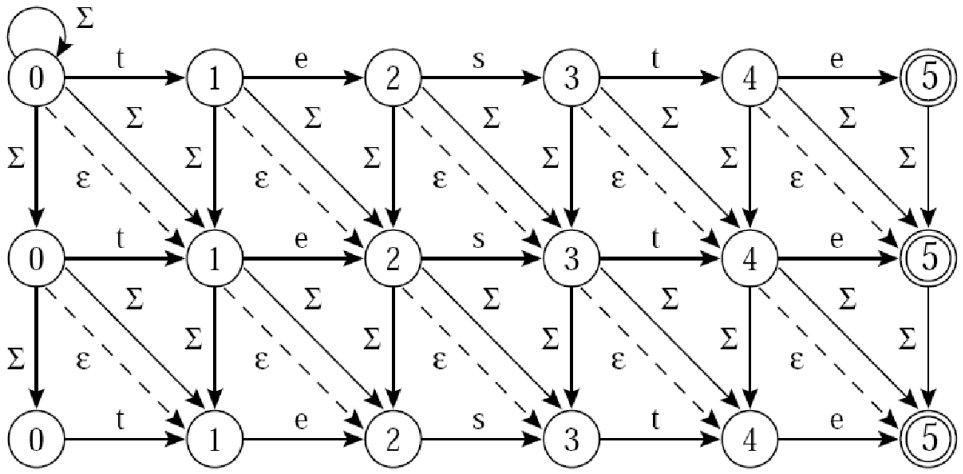


Figura 3: Autômato não-determinista capaz de fazer uma busca com até 2 erros.

0 algoritmo empacota cada linha j, com $0 \leq j \leq k$ do autômato não-determinista em uma palavra R_j diferente do computador. Cada caractere lido implica na simulação de todas as transições do autômato, usando operações entre as $k+1$ máscaras de *bits*. Todas as $k+1$ máscaras de bits têm a mesma estrutura e assim o mesmo *bit* é alinhado com a mesma posição no texto.

Na posição i do texto, os novos valores R'_j são obtidos a partir dos valores correntes de R_j, de acordo com as seguintes relações:

$$\begin{cases} R'_0 = ((R_0 >> 1) \mid 10^{m-1} \ \& \ M[T[i]]) \text{ e} \\ R'_j = ((R_j >> 1) \ \& \ M[T[i]] \mid R_{j-1} \mid (R_{j-1} >> 1) \mid (R'_{j-1} >> 1)) \end{cases}$$

onde M é uma tabela utilizada para armazenar uma máscara de bits $b_1, ...b_{|P|}$ para cada caractere do padrão P. Por exemplo, para o padrão $P =$ `sistema`, é obtida a tabela M apresentada na Tabela 3:

O algoritmo em alto nível é apresentado no Algoritmo 2.

2.5.1 Descrição de módulos, métodos e tipos

O sistema está estruturado em dois arquivos principais, `shiftand.c` e `shiftand.h`. O primeiro arquivo contém as implementações e o segundo contém as definições de tipos e interfaces de funções.

Foram utilizados os tipos sugeridos por Ziviani [4], chamados `TipoPadrao` e `TipoTexto`.

Dentro do módulo `shiftand.c` destacam-se os seguintes métodos:

	1	**2**	**3**	**4**	**5**	**6**	**7**
M[s]	1	0	1	0	0	0	0
M[i]	0	1	0	0	0	0	0
M[t]	0	0	0	1	0	0	0
M[e]	0	0	0	0	1	0	0
M[m]	0	0	0	0	0	1	0
M[a]	0	0	0	0	0	0	1

Tabela 3: Máscaras de bits do Shift-And para a palavra `sistema`.

Algoritmo 2 (Padrão = $p_1p_2 \ldots p_m$, FILE *p_ArquivoEntrada)

para todo (c $\in \Sigma$) **faça**
  M[c] = 0^m;
fim para
para (j=1 **to** m) **faça**
  M[p_j] = M[p_j] | $O^{j-1}10^{m-j}$;
fim para
para (j=0 **to** k) **faça**
  $R_j = 1^j0^{m-j}$;
fim para
enquanto (!fim(p_ArquivoEntrada)) **faça**
  linha = getProximaLinha(p_ArquivoEntrada)
  para (i=0; i$\leq$ tamanho(linha); i++) **faça**
    $R_{ant} = R_O$;
    $R_{novo} = ((R_{ant} >> 1)|10^{m-1}\&M[linha[i]])$;
    $R_0 = R_{novo}$;
    para (j=1 **to** k) **faça**
      $R_{novo} = ((R_j >> 1\&M[linha[i]])|R_{ant}|((R_{ant}|R_{novo}) >> 1)$;
      $R_{ant} = R_j$;
      $R_j = R_{novo}|10^{m-1}$;
    fim para
    se $((R_{novo}\&0^{m-1}) \neq 0^m)$ **então**
      Reporta casamento em *linha*;
    fim se
  fim para
fim enquanto

- `void ShiftAndAproximado(TipoTexto T, long *n, TipoPadrao P, long *m, long *k, long p_NumeroLinhas)`: este método implementar o algoritmo *Shift-And*, conforme apresentado por Ziviani [4].

- `chomp(char *string)`: faz um *chomp* em uma *string*, isto é, remove o '\n' e coloca um '\0' no final.

2.5.2 Utilização do sistema

Para utilizar o sistema, deve-se chamar o arquivo `shiftand.e` com a seguinte lista de parâmetros:

- `-e`: indica o número de erros aceitáveis na pesquisa aproximada
- `-f`: indica o nome do arquivo no qual se deseja fazer a busca
- `-p`: indica o padrão que se deseja buscar no texto

Por exemplo, a seguinte linha busca pelo padrão `main` no arquivo de nome `shiftand.c` considerando uma pesquisa com 2 erros:

```
./shiftand.e -f shiftand.c -e 2 -p main
```

Caso seja necessário buscar um padrão que tenha o caracetere de espaço, o padrão deverá estar entre aspas.

2.6 Resultados Computacionais

Nesta seção, são reportados resultados computacionais obtidos utilizando-se os algoritmos acima, implementados sob a ótica das decisões de projeto da Seção 1. Inicialmente, uma análise de cada algoritmo será apresentada individualmente. Objetiva-se, desta maneira, obter uma visão do comportamento de cada algoritmo em diferentes cenários. Em seguida, serão apresentadas comparações entre os algoritmos, a fim de verificar qual dos dois é superior de acordo com os critérios de número de comparações e tempo de processamento.

2.6.1 Roteiro para os testes

Os algoritmos foram executados sobre os três arquivos da coleção `wsj`: `wsj89`, `wsj88_20` e `wsj88`. A Tabela 4 apresenta os padrões que foram utilizados para as consultas e o tamanho de cada padrão[3].

2.6.2 Resultados do Algoritmo de Programação Dinâmica de Sellers

Os resultados obtidos pelo Algoritmo de Sellers podem ser visualizados através da seguintes tabelas:

- As Tabelas 5, 6, 7 e 8 reportam os resultados obtidos pelo algoritmo de Programação Dinâmica de Sellers para o casamento aproximado dos padrões contidos na Tabela 4 para o arquivo `wsj89` para os diferentes valores de k..
- As Tabelas 9, 10, 11 e 12 reportam os resultados obtidos considerando o arquivo `wsj88_20` e os valores de k.
- Finalmente, as Tabelas 13, 14, 15 e 16 reportam reportam resultados similares, mas para o arquivo `wsj88`.

[3] O caractere de espaço faz parte do padrão e por isso é incluído na contagem.

ID Padrão	Padrão	Tamanho
1	DCC UFMG dollar	15
2	dia branco	10
3	Macunaima administration	24
4	Brazilian coffee	16
5	New York Stock Exchange	23
6	Manacapuru	10
7	Canada Treasury	15
8	Michael Gregory	15
9	price index	11
10	Uberaba	7

Tabela 4: Padrões utilizados para testes e respectivos tamanhos.

As Figuras 4 e 5 apresentam, respectivamente, o número de comparações e os tempos gastos para cada padrão apresentado na tabela 4, de acordo com o **ID do padrão**.

Com relação ao número de comparações, é possível notar que aparentemente não existe relação entre o tamanho do padrão e o número de comparações, pois de fato este parâmetro não impacta o desempenho do algoritmo. A maior diferença do número de comparações é entre o padrão `dia branco` e o padrão `Uberaba`, que chega próxima 80 mil comparações.

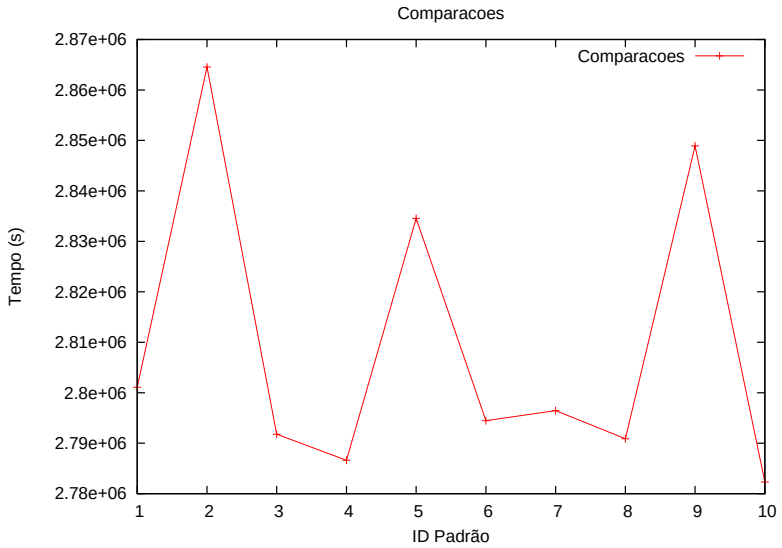


Figura 4: Programação Dinâmica: Comparações realizadas no arquivo **wsj89**, $k = 0$

Com relação ao tempo de processamento, também não há nenhum padrão de variação. Pode-se observar apenas que o tempo de processamento (***User Time*** medido através das diretivas `time` do sistema Linux) permanece na faixa entre 0.103 e 0.114 segundos.

Uma das poucas informações úteis que se pode extrair dos gráficos das Figuras 4 e 5 é o fato de que o algoritmo é altamente dependente do padrão em questão,

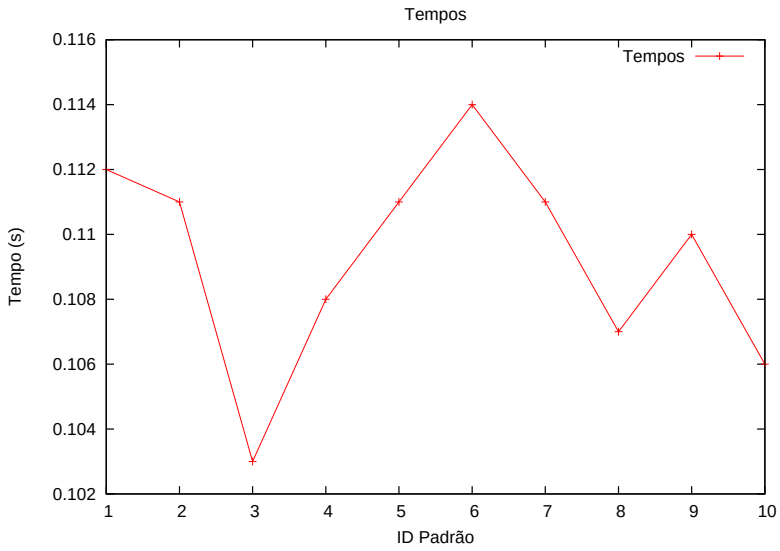


Figura 5: Programação Dinâmica: Tempo gasto para cada padrão. Busca no arquivo **wsj89**, $k = 0$

mas não do tamanho do padrão. É possível observar isso comparando o resultado do par (2,6) ou do trio (1, 7 e 8), apesar de cada conjunto ser de padrões com o mesmo tamanho. Tanto o número de comparações quanto o tempo de execução são bem diferentes entre eles. Isso acontece por que o algoritmo utiliza o artífício de não procurar um casamento com erro maior do que k. Este controle é feito através da variável `lact` do Algoritmo 1, e depende do padrão pois o casamento pode ocorrer em posições diferentes para padrões diferentes.

Como observa-se, resultados apresentados da maneira acima agregam pouquíssima informação. Por este motivo, não serão apresentados os gráficos para os demais valores de k e n. Serão apresentadas agora discussões sobre a variação do valor de k e n sobre o algoritmo. A variação destes parâmetros é interessante pois eles fazem parte da função de custo do algoritmo.

Variações no valor de n

O número de comparações realizadas pelo algoritmo de Sellers, segundo sua função de custo, é diretamente proporcional ao tamanho do arquivo. No presente trabalho, foram apresentados resultados para três arquivos diferentes, conforme apresentado na Seção 2.6.1.

Para montar a estrutura dos testes, foram tomadas as seguintes diretivas, de acordo com os resultados discutidos anteriormente:

- Para cada arquivo, foram escolhidos o menor padrão (`Uberaba`) e o maior padrão (`Macunaima administration`)
- Os dados serão apresentados de acordo com o tamanho dos arquivos.

As Figuras 6, 7, 8 e 9 apresentam o impacto com relação ao número de comparações e tempo de processamento de acordo com a variação do tamanho do arquivo.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	0.038	0.112	0.003	2801084
dia branco	0.195	0.111	0.004	2864543
Macunaima administration	0.213	0.103	0.003	2791780
Brazilian coffee	0.215	0.108	0.003	2786630
New York Stock Exchange	0.439	0.111	0.002	2834554
Manacapuru	0.287	0.114	0.002	2794468
Canada Treasury	0.166	0.111	0.003	2796460
Michael Gregory	0.159	0.107	0.004	2790891
price index	0.202	0.110	0.003	2848901
Uberaba	0.153	0.106	0.001	2782324

Tabela 5: **Programação Dinâmica: Arquivo wsj89,** $k = 0$.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	0.257	0.165	0.002	5612016
dia branco	0.256	0.173	0.003	5879690
Macunaima administration	0.413	0.172	0.001	5751011
Brazilian coffee	0.283	0.172	0.003	5725874
New York Stock Exchange	0.587	0.164	0.004	5870873
Manacapuru	0.388	0.175	0.004	5803084
Canada Treasury	0.221	0.169	0.002	5811393
Michael Gregory	0.225	0.169	0.003	5744281
price index	0.258	0.168	0.006	5834932
Uberaba	0.220	0.169	0.003	5608659

Tabela 6: **Programação Dinâmica: Arquivo wsj89,** $k = 1$.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	0.587	0.204	0.005	8424604
dia branco	0.349	0.215	0.004	9147963
Macunaima administration	0.458	0.210	0.003	8771590
Brazilian coffee	0.321	0.203	0.006	8806785
New York Stock Exchange	0.832	0.218	0.007	9091997
Manacapuru	0.432	0.207	0.004	8902447
Canada Treasury	0.252	0.205	0.004	8923478
Michael Gregory	0.254	0.208	0.001	8768578
price index	0.603	0.214	0.006	8979012
Uberaba	0.605	0.205	0.004	8724065

Tabela 7: **Programação Dinâmica: Arquivo wsj89,** $k = 2$.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	0.341	0.244	0.005	11694911
dia branco	0.803	0.294	0.003	12743293
Macunaima administration	0.507	0.259	0.005	11888707
Brazilian coffee	0.364	0.258	0.003	11921951
New York Stock Exchange	1.022	0.286	0.005	12647783
Manacapuru	0.482	0.254	0.006	12013407
Canada Treasury	0.535	0.266	0.004	12067243
Michael Gregory	0.321	0.259	0.002	11894984
price index	1.210	0.270	0.004	12299073
Uberaba	1.129	0.258	0.009	11939084

Tabela 8: **Programação Dinâmica: Arquivo wsj89,** $k = 3$.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	1.717	0.734	0.088	20065358
dia branco	0.780	0.743	0.016	20644029
Macunaima administration	1.021	0.977	0.015	20081696
Brazilian coffee	0.769	0.721	0.014	20062306
New York Stock Exchange	3.457	0.789	0.022	20422944
Manacapuru	0.843	0.732	0.016	20086370
Canada Treasury	0.808	0.723	0.011	20139532
Michael Gregory	0.800	0.732	0.019	20098150
price index	3.256	0.773	0.018	20535112
Uberaba	0.815	0.715	0.012	20022550

Tabela 9: **Programação Dinâmica: Arquivo wsj88_20,** $k = 0$.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	1.218	1.199	0.010	40260601
dia branco	1.278	1.218	0.012	42421769
Macunaima administration	1.252	1.198	0.013	41421080
Brazilian coffee	1.217	1.160	0.012	41247304
New York Stock Exchange	5.075	1.239	0.031	42307786
Manacapuru	1.339	1.195	0.024	41795314
Canada Treasury	1.302	1.204	0.011	41898173
Michael Gregory	1.278	1.201	0.014	41408518
price index	4.387	1.241	0.018	42076793
Uberaba	1.317	1.189	0.019	40376356

Tabela 10: **Programação Dinâmica: Arquivo wsj88_20,** $k = 1$.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	1.416	1.401	0.010	60447021
dia branco	1.645	1.556	0.013	66085542
Macunaima administration	1.524	1.446	0.018	63225387
Brazilian coffee	1.561	1.440	0.018	63487023
New York Stock Exchange	6.390	1.581	0.031	65528476
Manacapuru	1.605	1.472	0.011	64175681
Canada Treasury	1.609	1.496	0.012	64373127
Michael Gregory	1.561	1.459	0.022	63248323
price index	5.677	1.520	0.028	64799627
Uberaba	2.811	1.452	0.023	62879741

Tabela 11: **Programação Dinâmica: Arquivo wsj88_20,** $k = 2$.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	1.761	1.697	0.018	84066810
dia branco	5.453	2.055	0.023	92169907
Macunaima administration	1.979	1.826	0.015	85737561
Brazilian coffee	2.013	1.824	0.016	85980729
New York Stock Exchange	7.912	2.034	0.030	91284692
Manacapuru	1.960	1.809	0.018	86645957
Canada Treasury	2.567	1.849	0.019	87082783
Michael Gregory	1.959	1.810	0.020	85849679
price index	4.556	1.985	0.020	88846839
Uberaba	9.320	1.792	0.044	86117848

Tabela 12: **Programação Dinâmica: Arquivo wsj88_20,** $k = 3$.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	10.928	3.956	0.491	108276133
dia branco	4.191	3.958	0.076	111418775
Macunaima administration	4.295	3.958	0.086	108371736
Brazilian coffee	4.531	3.927	0.064	108285732
New York Stock Exchange	9.237	4.313	0.159	110364886
Manacapuru	4.292	3.963	0.068	108399474
Canada Treasury	4.489	3.921	0.064	108711134
Michael Gregory	4.273	3.908	0.083	108427025
price index	11.147	4.220	0.108	110799796
Uberaba	4.141	3.880	0.064	108049495

Tabela 13: **Programação Dinâmica: Arquivo wsj88,** $k = 0$.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	6.858	6.407	0.080	217263374
dia branco	6.795	6.581	0.071	228951380
Macunaima administration	6.943	6.470	0.083	223574092
Brazilian coffee	7.368	6.441	0.085	222634568
New York Stock Exchange	17.590	6.591	0.148	228488314
Manacapuru	7.414	6.492	0.064	225621454
Canada Treasury	6.960	6.480	0.094	226205374
Michael Gregory	6.998	6.453	0.093	223420787
price index	23.703	6.632	0.091	227050696
Uberaba	6.773	6.422	0.072	217859696

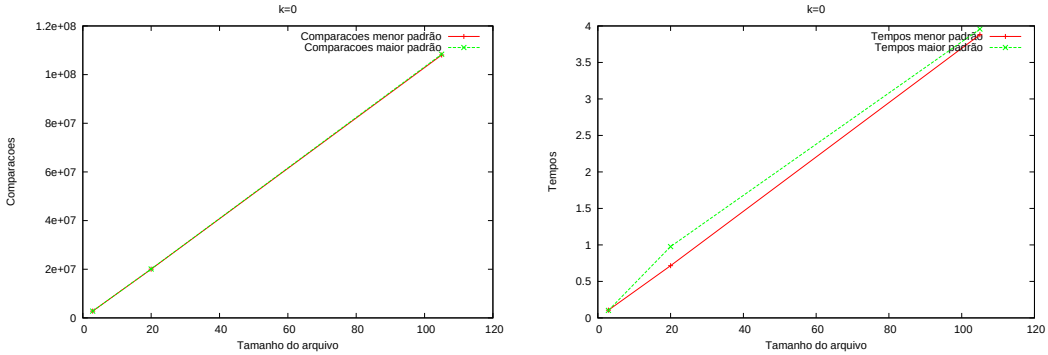
Tabela 14: **Programação Dinâmica: Arquivo wsj88,** $k = 1$.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	8.107	7.603	0.088	326191544
dia branco	9.559	8.369	0.089	356702381
Macunaima administration	8.462	7.850	0.091	341273210
Brazilian coffee	8.500	7.880	0.084	342708359
New York Stock Exchange	15.132	8.562	0.160	353799226
Manacapuru	11.723	7.998	0.087	346444289
Canada Treasury	8.773	7.999	0.074	347539458
Michael Gregory	10.802	7.907	0.103	341262865
price index	23.339	8.216	0.123	349693752
Uberaba	16.147	7.803	0.109	339292500

Tabela 15: **Programação Dinâmica: Arquivo wsj88,** $k = 2$.

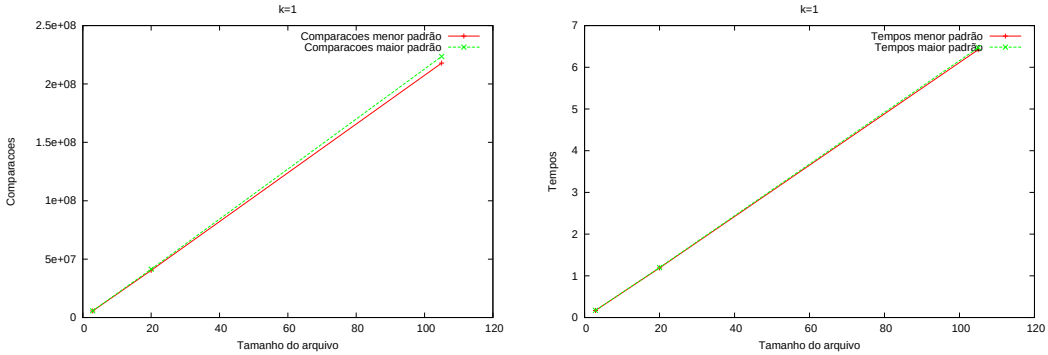
Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	9.526	9.107	0.080	453669108
dia branco	19.024	11.141	0.112	497469213
Macunaima administration	10.607	9.895	0.078	462794935
Brazilian coffee	11.784	9.969	0.101	464156295
New York Stock Exchange	37.741	10.849	0.191	492789764
Manacapuru	12.252	9.888	0.084	467746593
Canada Treasury	15.535	10.729	0.098	470161104
Michael Gregory	10.291	9.799	0.074	463207571
price index	39.849	10.567	0.162	479460570
Uberaba	27.348	10.084	0.242	464751894

Tabela 16: **Programação Dinâmica: Arquivo wsj88,** $k = 3$.



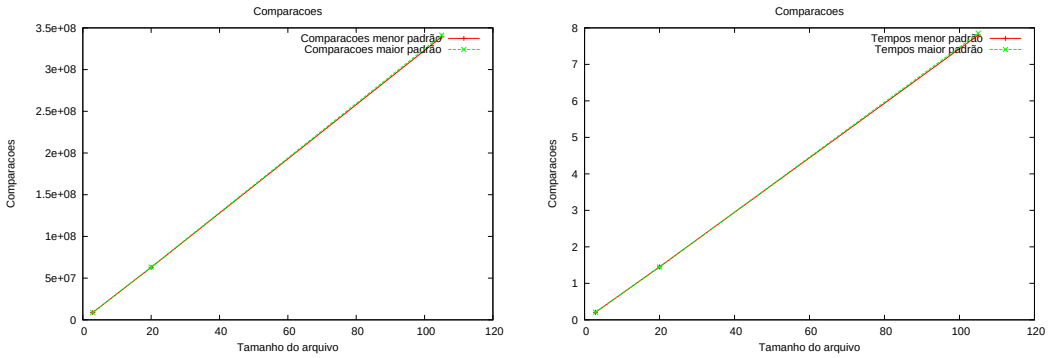
(a) Comparações (b) Tempos

Figura 6: Programação Dinâmica. Variação do tamanho do arquivo para $k = 0$.



(a) Comparações (b) Tempos

Figura 7: Programação Dinâmica. Variação do tamanho do arquivo para $k = 1$.

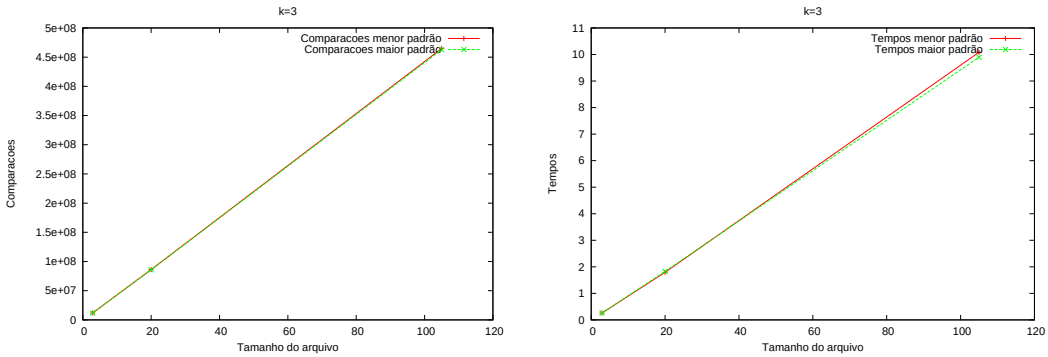


(a) Comparações (b) Tempos

Figura 8: Programação Dinâmica. Variação do tamanho do arquivo para $k = 2$.

As seguintes conclusões podem ser tomadas a partir dos gráficos acima citados e dos resultados das Tabelas:

- O desempenho do algoritmo é de fato diretamente proporcional de acordo com o valor de n. O crescimento do número de comparações e do tempo de processamento é linear com relação ao tamanho do arquivo.
- O algoritmo, como já havia sido discutido acima, é praticamente indiferente ao tamanho do padrão. Apesar de ser dependente de um padrão específico,



(a) Comparações (b) Tempos

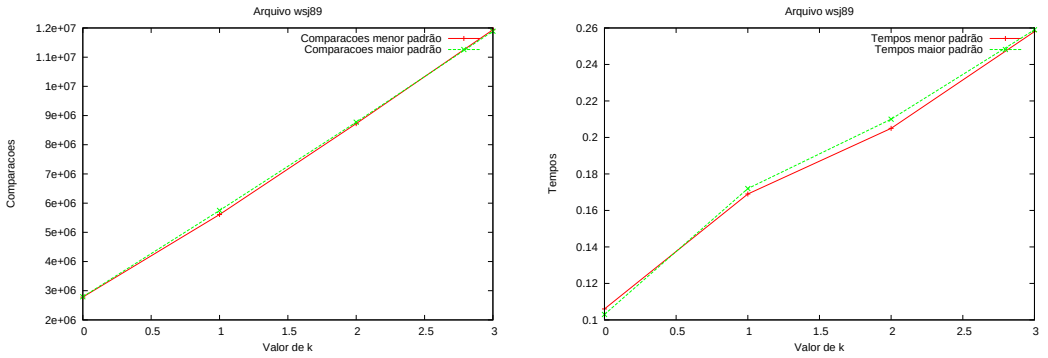
Figura 9: Programação Dinâmica. Variação do tamanho do arquivo para $k = 3$.

o valor do número de comparações e do tempo de processamento é sempre próximo quando se trata o mesmo valor de k.

Variações no valor de k

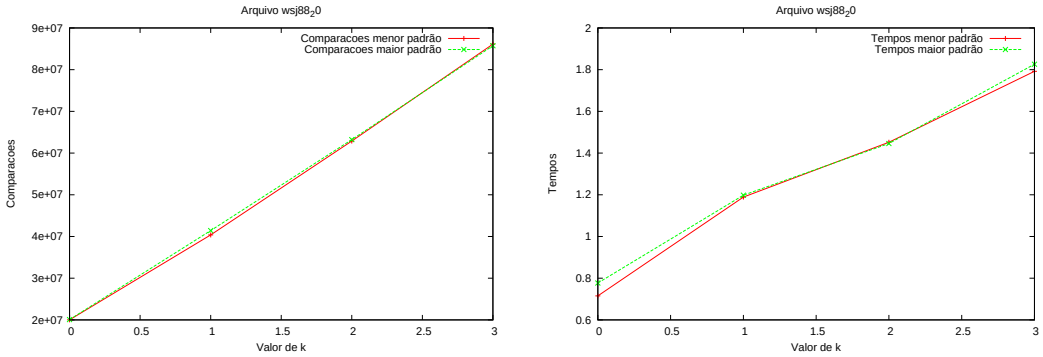
Nesta bateria de testes, o valor de n será mantido fixo e será alterado o valor de k (o número de erros permitidos). Os testes serão feitos da mesma maneira como o anterior, considerando os valores para o maior padrão e também para o menor padrão do conjunto de padrões da Tabela 4.

O resultado obtido segue nas Figuras 10, 11 e 12.



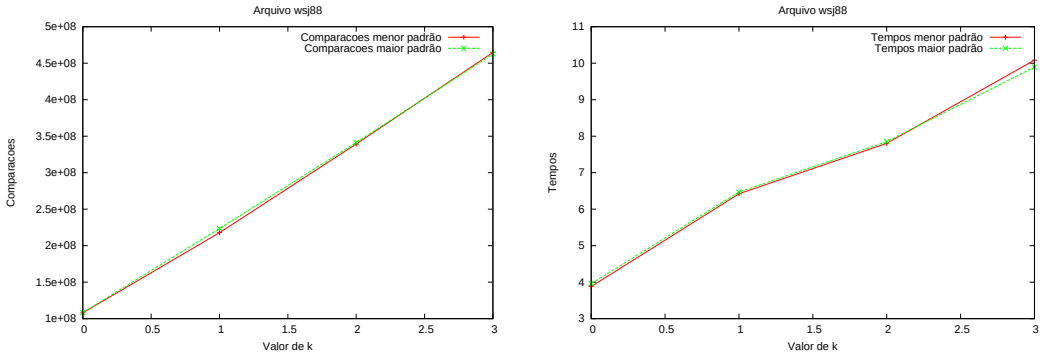
(a) Comparações (b) Tempos

Figura 10: Programação Dinâmica. Variação do valor de k para o arquivo `wsj89`.



(a) Comparações (b) Tempos

Figura 11: Programação Dinâmica. Variação do valor de k para o arquivo `wsj88_20`.



(a) Comparações (b) Tempos

Figura 12: Programação Dinâmica. Variação do valor de k para o arquivo `wsj88`.

Estes novos gráficos permitem as seguintes conclusões:

- O desempenho do algoritmo é praticamente linear com relação ao valor de k, o que já era esperado de acordo com a análise feita na Seção 2.3.
- Foi possível revalidar a observação de que o desempenho do algoritmo é indiferente com relação ao tamanho do padrão.

2.6.3 Resultados do Shift-And

Os resultados obtidos pelo Shift-And podem ser visualizados através da seguintes tabelas:

- As Tabelas 17, 18, 19 e 20 reportam os resultados obtidos pelo algoritmo Shift-And para o casamento aproximado dos padrões contidos na Tabela 4 para o arquivo `wsj89` para os diferentes valores de k.
- As Tabelas 21, 22, 23 e 24 reportam os resultados obtidos considerando o arquivo `wsj88_20` e os valores de k.
- Finalmente, as Tabelas 25, 26, 27 e 28 reportam reportam resultados similares, mas para o arquivo `wsj88`.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	0.995	0.897	0.002	2778109
dia branco	1.167	0.888	0.005	2778109
Macunaima administration	1.137	0.889	0.003	2778109
Brazilian coffee	0.997	0.888	0.005	2778109
New York Stock Exchange	3.981	0.910	0.009	2778109
Manacapuru	1.948	0.899	0.006	2778109
Canada Treasury	1.003	0.888	0.005	2778109
Michael Gregory	1.000	0.893	0.005	2778109
price index	0.998	0.886	0.007	2778109
Uberaba	1.024	0.890	0.010	2778109

Tabela 17: **Shift-And: Arquivo wsj89,** $k = 0$.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	1.049	0.931	0.004	5556218
dia branco	1.037	0.925	0.003	5556218
Macunaima administration	1.178	0.923	0.005	5556218
Brazilian coffee	1.033	0.925	0.004	5556218
New York Stock Exchange	3.944	0.936	0.007	5556218
Manacapuru	1.893	0.927	0.008	5556218
Canada Treasury	1.037	0.924	0.004	5556218
Michael Gregory	1.088	0.916	0.007	5556218
price index	1.033	0.921	0.006	5556218
Uberaba	1.040	0.925	0.003	5556218

Tabela 18: **Shift-And: Arquivo wsj89,** $k = 1$.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	1.095	0.985	0.002	8334327
dia branco	1.094	0.977	0.005	8334327
Macunaima administration	1.235	0.979	0.003	8334327
Brazilian coffee	1.087	0.974	0.006	8334327
New York Stock Exchange	4.250	0.990	0.014	8334327
Manacapuru	2.101	0.980	0.008	8334327
Canada Treasury	1.090	0.972	0.006	8334327
Michael Gregory	1.085	0.976	0.003	8334327
price index	1.392	0.974	0.007	8334327
Uberaba	1.635	0.976	0.007	8334327

Tabela 19: **Shift-And: Arquivo wsj89,** $k = 2$.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	1.141	1.014	0.008	11112436
dia branco	1.655	1.009	0.005	11112436
Macunaima administration	1.255	1.002	0.008	11112436
Brazilian coffee	1.148	1.009	0.003	11112436
New York Stock Exchange	4.243	1.030	0.003	11112436
Manacapuru	2.165	1.012	0.007	11112436
Canada Treasury	1.376	1.005	0.009	11112436
Michael Gregory	1.132	1.011	0.003	11112436
price index	3.654	1.028	0.008	11112436
Uberaba	4.040	1.015	0.005	11112436

Tabela 20: **Shift-And: Arquivo wsj89,** $k = 3$.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	7.222	6.806	0.023	20000341
dia branco	7.058	6.823	0.018	20000341
Macunaima administration	7.053	6.812	0.024	20000341
Brazilian coffee	7.297	6.817	0.018	20000341
New York Stock Exchange	16.640	6.944	0.045	20000341
Manacapuru	7.109	6.807	0.027	20000341
Canada Treasury	7.110	6.817	0.024	20000341
Michael Gregory	7.338	6.829	0.022	20000341
price index	11.557	6.851	0.032	20000341
Uberaba	7.119	6.814	0.020	20000341

Tabela 21: **Shift-And: Arquivo wsj88_20,** $k = 0$.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	8.366	7.085	0.032	40000682
dia branco	7.619	7.117	0.022	40000682
Macunaima administration	7.292	7.044	0.024	40000682
Brazilian coffee	8.386	7.059	0.025	40000682
New York Stock Exchange	9.023	7.205	0.036	40000682
Manacapuru	7.775	7.041	0.027	40000682
Canada Treasury	7.504	7.065	0.026	40000682
Michael Gregory	10.571	7.087	0.033	40000682
price index	12.031	7.120	0.048	40000682
Uberaba	7.371	7.064	0.018	40000682

Tabela 22: **Shift-And: Arquivo wsj88_20,** $k = 1$.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	7.850	7.438	0.027	60001023
dia branco	7.981	7.455	0.015	60001023
Macunaima administration	7.693	7.446	0.026	60001023
Brazilian coffee	9.820	7.454	0.055	60001023
New York Stock Exchange	23.435	7.581	0.058	60001023
Manacapuru	9.222	7.437	0.030	60001023
Canada Treasury	8.064	7.440	0.023	60001023
Michael Gregory	7.972	7.446	0.021	60001023
price index	11.721	7.489	0.053	60001023
Uberaba	11.990	7.576	0.042	60001023

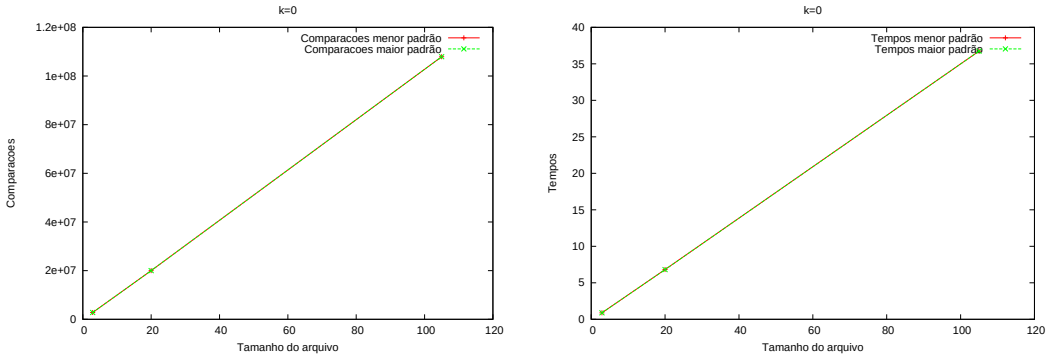
Tabela 23: **Shift-And: Arquivo wsj88_20,** $k = 2$.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	8.256	7.693	0.043	80001364
dia branco	11.336	7.729	0.037	80001364
Macunaima administration	8.027	7.697	0.029	80001364
Brazilian coffee	8.121	7.692	0.024	80001364
New York Stock Exchange	18.081	7.845	0.066	80001364
Manacapuru	9.552	7.702	0.022	80001364
Canada Treasury	8.580	7.702	0.024	80001364
Michael Gregory	8.002	7.692	0.021	80001364
price index	15.316	7.832	0.061	80001364
Uberaba	23.430	7.784	0.081	80001364

Tabela 24: **Shift-And: Arquivo wsj88_20,** $k = 3$.

A discussão será iniciada com a variação do parâmetro n.

Variação no valor de n



(a) Comparações (b) Tempos

Figura 13: Shiftand. Variação do tamanho do arquivo para $k = 0$.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	38.928	36.654	0.141	107926315
dia branco	39.847	36.612	0.164	107926315
Macunaima administration	40.647	36.740	0.160	107926315
Brazilian coffee	38.743	36.611	0.160	107926315
New York Stock Exchange	97.684	37.413	0.258	107926315
Manacapuru	43.381	36.672	0.155	107926315
Canada Treasury	39.558	36.484	0.144	107926315
Michael Gregory	47.393	36.782	0.175	107926315
price index	45.026	36.856	0.193	107926315
Uberaba	40.332	36.744	0.159	107926315

Tabela 25: **Shift-And: Arquivo wsj88,** $k = 0$.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	43.775	37.976	0.170	215852630
dia branco	39.901	37.874	0.149	215852630
Macunaima administration	43.289	38.089	0.193	215852630
Brazilian coffee	42.485	38.039	0.169	215852630
New York Stock Exchange	63.725	39.057	0.319	215852630
Manacapuru	44.455	38.117	0.166	215852630
Canada Treasury	41.401	38.006	0.142	215852630
Michael Gregory	42.179	38.054	0.157	215852630
price index	45.695	38.219	0.173	215852630
Uberaba	44.104	38.090	0.182	215852630

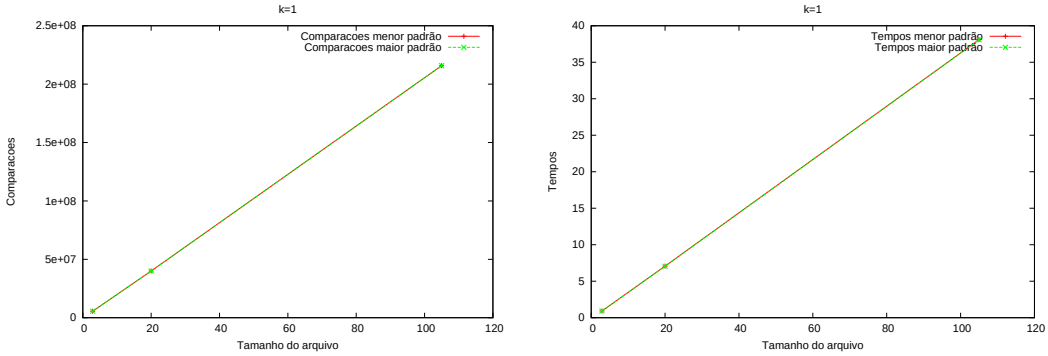
Tabela 26: **Shift-And: Arquivo wsj88,** $k = 1$.

Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	44.923	40.091	0.188	323778945
dia branco	45.699	40.110	0.169	323778945
Macunaima administration	44.768	40.091	0.162	323778945
Brazilian coffee	46.271	40.025	0.165	323778945
New York Stock Exchange	60.278	40.683	0.251	323778945
Manacapuru	42.141	39.788	0.123	323778945
Canada Treasury	51.309	39.917	0.170	323778945
Michael Gregory	50.288	39.863	0.173	323778945
price index	53.265	40.015	0.195	323778945
Uberaba	49.417	39.849	0.177	323778945

Tabela 27: **Shift-And: Arquivo wsj88,** $k = 2$.

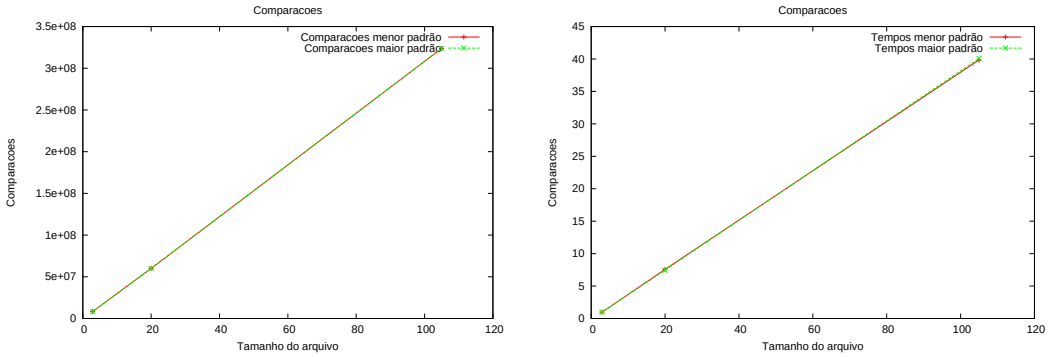
Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	42.939	40.718	1.029	431705260
dia branco	45.671	40.672	0.162	431705260
Macunaima administration	42.191	40.637	0.097	431705260
Brazilian coffee	43.225	40.684	0.138	431705260
New York Stock Exchange	42.558	40.550	0.250	431705260
Manacapuru	41.492	40.576	0.086	431705260
Canada Treasury	43.317	40.723	0.152	431705260
Michael Gregory	41.689	40.638	0.109	431705260
price index	44.598	40.701	0.252	431705260
Uberaba	44.061	40.679	0.431	431705260

Tabela 28: **Shift-And: Arquivo wsj88,** $k = 3$.



(a) Comparações (b) Tempos

Figura 14: Shiftand. Variação do tamanho do arquivo para $k = 1$.



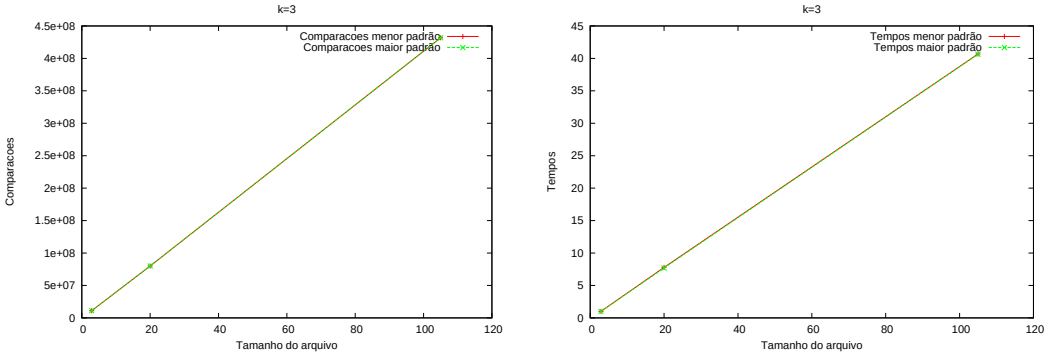
(a) Comparações (b) Tempos

Figura 15: Shiftand. Variação do tamanho do arquivo para $k = 2$.

As Figuras 13, 14, 15 e 16 apresentam o impacto no desempenho do algoritmo quando se aumenta o tamanho do arquivo.

Estes dados revelam o seguinte comportamento para o Shift-And:

- O número de comparações e o tempo de processamento de fato crescem linearmente de acordo com o tamanho do arquivo de entrada



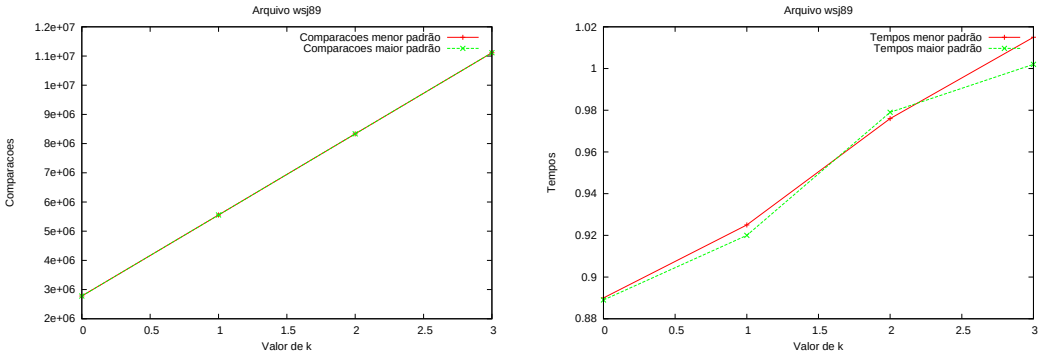
(a) Comparações (b) Tempos

Figura 16: Shiftand. Variação do tamanho do arquivo para $k = 3$.

- O algoritmo é totalmente independente do padrão. De fato, como pode ser observado nas Tabelas, o número de comparações realizadas para o mesmo arquivo e para o mesmo valor de k é constante para todos os padrões. Isso pode ser visto nos gráficos, dado que as curvas dos 2 padrões são sobrepostas.

Variação no valor de k

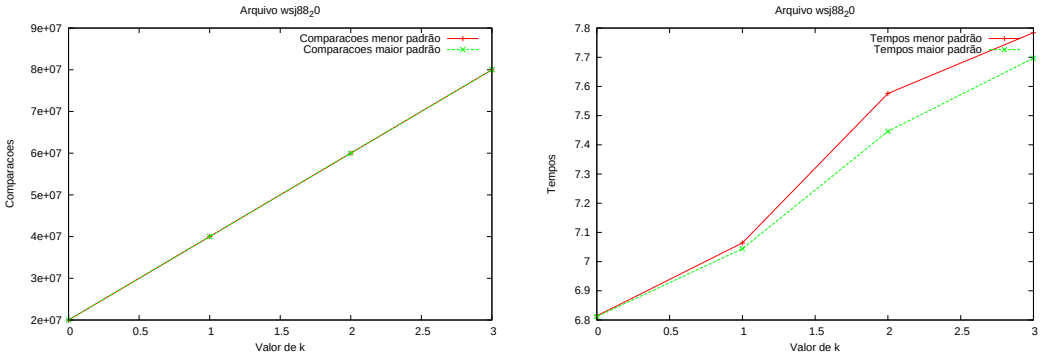
O resultado obtido segue nas Figuras 17, 18 e 19.



(a) Comparações (b) Tempos

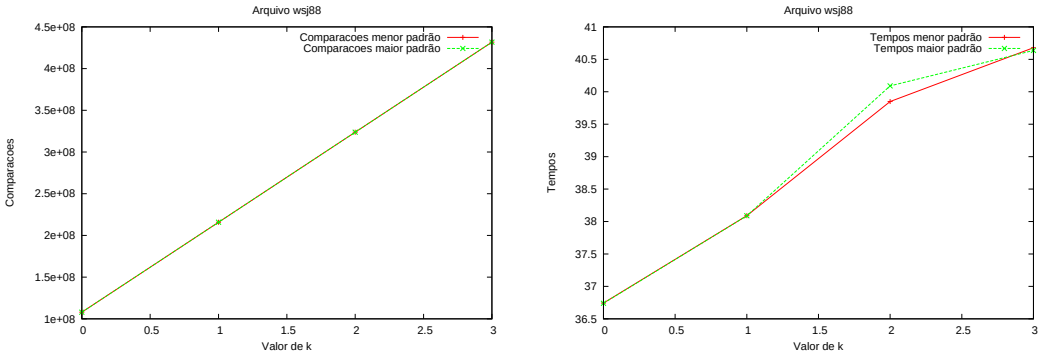
Figura 17: Shiftand. Variação do valor de k para o arquivo `wsj89`.

Novamente, observa-se a independência do algoritmo com relação ao tamanho do padrão para o número de comparações. Existem algumas variações no tempo de processamento que podem ser justificadas por necessidade de E/S.



(a) Comparações (b) Tempos

Figura 18: Shiftand. Variação do valor de k para o arquivo `wsj88_20`.



(a) Comparações (b) Tempos

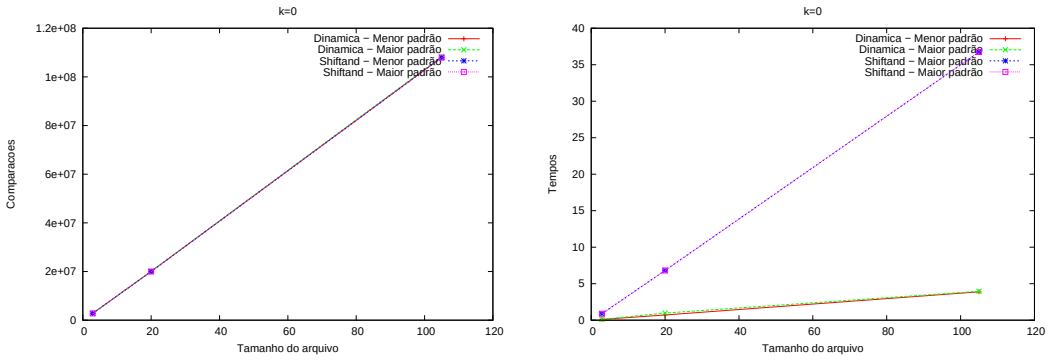
Figura 19: Shiftand. Variação do valor de k para o arquivo `wsj88`.

2.6.4 Comparações Sellers x Shift-And

O objetivo desta Seção é prover uma análise comparativa dos dois algoritmos, com base no arcabouço de informações levantadas nas Seções anteriores. A discussão será iniciada analisando-se a variação no tamanho do arquivo de entrada.

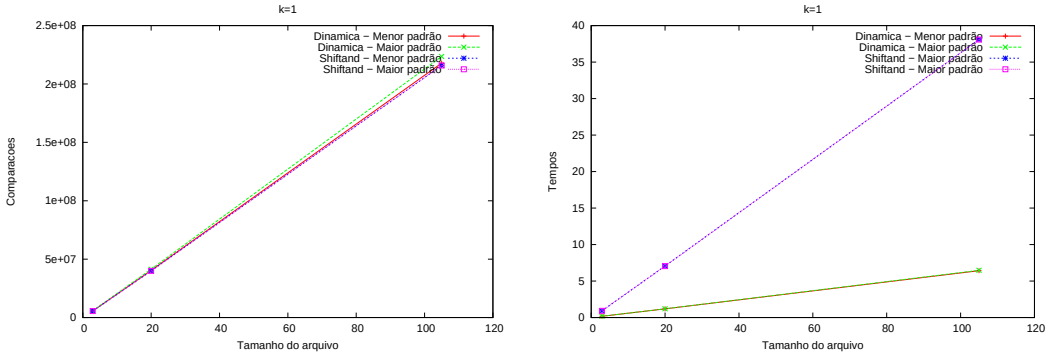
Variação no valor de n

As Figuras 20, 21, 22 e 23 apresentam resultados comparativos para a variação do tamanho do arquivo de entrada mantendo-se o valor de k constante.



(a) Comparações (b) Tempos

Figura 20: Sellers x Shiftand. Variação do tamanho do arquivo para $k = 0$.



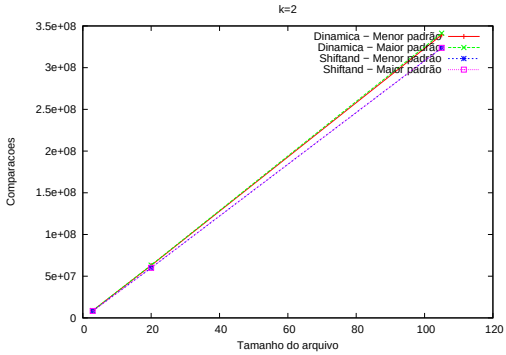
(a) Comparações (b) Tempos

Figura 21: Sellers x Shiftand. Variação do tamanho do arquivo para $k = 1$.

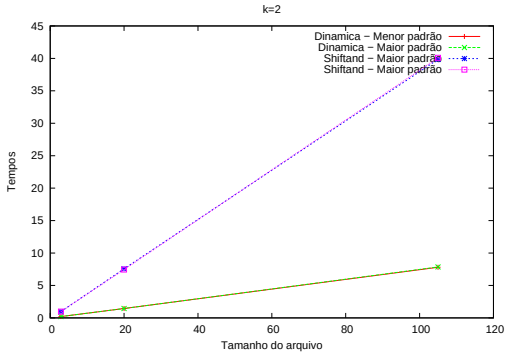
A partir da análise dos gráficos que consideram a variação de n (e considerando as decisões de projeto da Seção 1), pode-se concluir que:

1. O **número de comparações** realizadas pelo algoritmo de Programação Dinâmica de Sellers e pelo Shift-And está na mesma ordem de grandeza. Os valores são muito próximos, como ilustrado pelas Figuras 20(a), 21(a), 22(a) e 23(a). Este resultado era esperado, pois com relação ao número de comparações, ambos os algoritmos são $O(kn)$[4].

[4] Os termos n e $|T|$ serão usadas como sinônimos durante todo este texto.

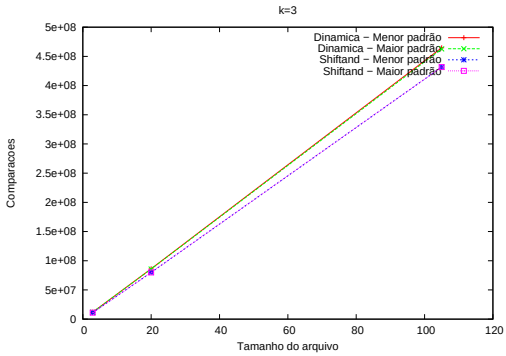


(a) Comparações

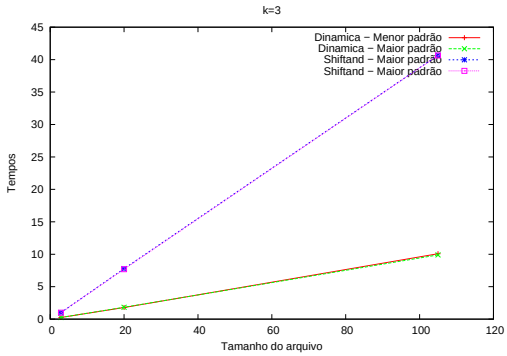


(b) Tempos

Figura 22: Sellers x Shiftand. Variação do tamanho do arquivo para $k = 2$.



(a) Comparações



(b) Tempos

Figura 23: Sellers x Shiftand. Variação do tamanho do arquivo para $k = 3$.

2. Com relação ao **tempo de processamento**, o Shift-And foi bem inferior ao algoritmo de Sellers. Apesar de ambos realizarem praticamente o mesmo número de comparações, provavelmente as iterações do Shift-And são mais caras do que as iterações de Sellers.

Variação no valor de k

As Figuras 24, 25 e 26 apresentam os resultados considerando cada arquivo individualmente e variando-se o valor de k.

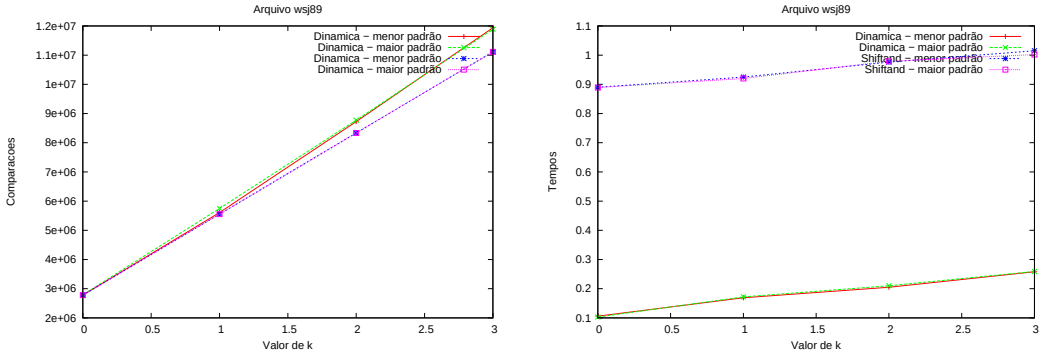
É fácil verificar a superioridade do algoritmo de Sellers quando o tamanho de k aumenta. Em todos os casos, o tempo de processamento é muitas vezes menor. A razão entre os tempos chega a ser 10, em alguns casos.

Estes testes confirmam a superioridade do algoritmo de Sellers sobre o Shift-And.

2.6.5 Parte I – Conclusões gerais

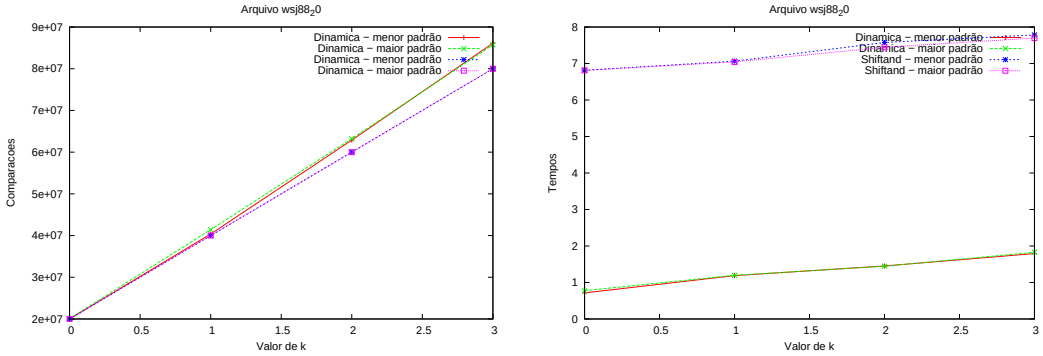
Finalizada a primeira parte do trabalho, obteve-se como conclusões gerais:

- Foi verificada a superioridade do Algoritmo de Programação Dinâmica de Sellers sobre o Shift-And. Provavelmente esta superioridade se deve à simplicidade do algoritmo de Sellers, dado que ambos os algoritmos possuem a mesma



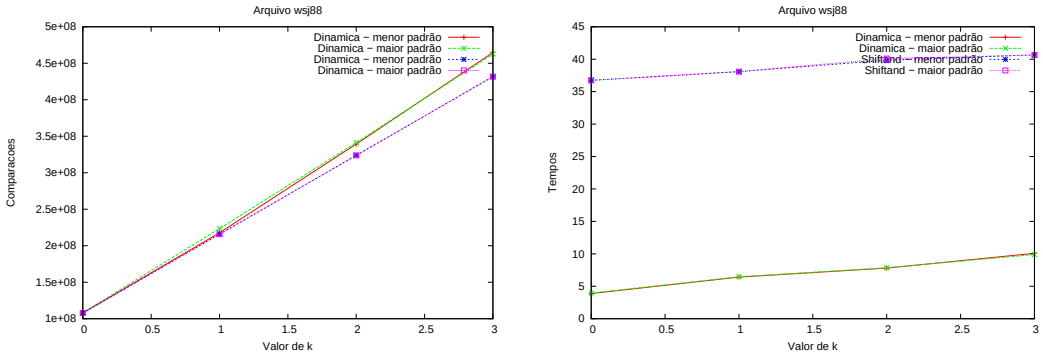
(a) Comparações (b) Tempos

Figura 24: Sellers x Shiftand. Variação do valor de k para o arquivo `wsj89`.



(a) Comparações (b) Tempos

Figura 25: Sellers x Shiftand. Variação do valor de k para o arquivo `wsj88_20`.



(a) Comparações (b) Tempos

Figura 26: Sellers x Shiftand. Variação do valor de k para o arquivo `wsj88`.

função de custo para o número de comparações. Um outro algoritmo de Programação Dinâmica para o mesmo problema, como o citado por Navarro em [1] provavelmente seria pior do que o Shift-And pois o algoritmo itera sobre uma matriz, com custo $O(|P||T|)$.

- Foi possível utilizar a proposta de Baeza-Yates, citada por Ziviani [4]: de que é interessante comparar o tempo de relógio de dois algoritmos quando a função de custo de ambos é da mesma ordem. Os dois algoritmos investigados

possuem função de complexidade igual a $O(kn)$. Entretanto, um dos dois revelou-se muito mais interessante do que outros, em todos os cenários e em praticamente todos os quesitos.

3 Busca exata de padrões

3.1 Revisão das Decisões de Projeto

Neste momento, é importante revisar algumas decisões de projeto pois algumas das premissas adotadas serão alteradas devido a características próprias dos algoritmos. As principais modificações são:

- No algoritmo de Huffman é necessário que todo o texto fique em memória durante o procedimento de busca. Isto ocorre porque não é possível pegar blocos do texto comprimido e passá-los para o BMH. Por exemplo, caso seja determinado que sempre será buscado um bloco de mil unidades (sejam elas bits ou bytes), pode-se cortar um código no meio e gerar inconsistência nos códigos do arquivo binário. Isto exigiria uma alteração nas decisões de projeto apresentadas na Seção 1, para manter um processo de comparação justo entre o BMH que busca no arquivo comprimido e o BMH que busca no texto normal, dado que em um o texto é inicialmente carregado e depois se encontra em memória e no outro o texto é lido por linhas. Por isso, optou-se pela seguinte estratégia:

  - A implementação do **algoritmo de Huffman** será mantida. Carrega-se o arquivo da memória e em seguida opera-se sobre ele todo.
  - Serão **implementados dois algoritmos BMH**: o primeiro deles (que será chamado a partir de agora de BMHα) irá trabalhar seguindo as diretivas de projeto apresentadas na Seção 1. Uma segunda implementação (BMHβ) irá trabalhar da seguinte maneira: irá carregar todo o arquivo para a memória e em seguida irá realizar a busca no arquivo. Desta maneira, pretende-se verificar a implementação mais eficiente, não somente em termos de tempo de processamento, mas também em termos de tempos de sistema e tempo real gastos pela aplicação.

- A interface com o usuário também será alterada. No caso do BMHβ, serão impressas todas as linhas onde o padrão aparece, independentemente de quantas vezes o padrão aparece. Isto é, se o padrão aparece duas vezes em uma linha, então a linha será impressa duas vezes. No caso do BMH aplicado sobre o texto comprimido, será impressa apenas a **posição onde o padrão acontece no texto**.

Na Seção seguinte é apresentado o algoritmo Boyer-Moore-Horspool, para busca exata em texto. Serão apresentadas as duas abordagens: busca em arquivo não-comprimido e busca em arquivo comprimido.

3.2 Busca exata em texto não comprimido usando o BMH

Boyer e Moore apresentaram em 1977 o algoritmo que ficou conhecido como BM. Em 1980, Hoorspool apresentou uma simplificação e o algoritmo passou a ser conhecido como BMH. Em 1990, Sunday apresentou uma outra simplificação e o algoritmo passou a ser conhecido como BMHS.

A idéia central do algoritmo é pesquisar no sentido da direita para a esquerda, o que o torna muito mais rápido. O padrão P desliza em uma janela do texto T. Para cada posição desta janela, o algoritmo pesquisa por um sufixo da janela que casa com um sufixo do padrão P por meio de comparações realizadas da direita para a esquerda. Se há uma igualdade, então houve um casamento. Caso contrário, o algoritmo computa um deslocamento em que o padrão deve ser deslizado para a direita, para iniciar uma nova tentativa de casamento.

Originalmente, no algoritmo BM, existiam duas heurísticas para calcular o deslocamento:

- Heurística da ocorrência: alinha a posição do texto que causou a colisão com o primeiro caractere no padrão que casa com ele.
- Heurística do casamento: ao mover o padrão para a direita, ele casa com o pedaço do texto anteriormente usado.

O algoritmo pode executar as duas heurísticas e decidir em tempo de execução qual será a que provocará o maior salto do padrão dentro do texto. Entretanto, a comparação necessária para realizar esta escolha pode penalizar o tempo de processamento do algoritmo. Ao longo dos anos, foram obtidos melhores resultados de pesquisa considerando apenas a heurística da ocorrência.

Foi nesse contexto que apareceu o trabalho de Hoorspool, que incluiu no algoritmo BM original um procedimento de pré-computação do padrão para computar a tabela de deslocamentos.

Em seguida, houve a simplificação proposta por Sunday em 1990. A principal alteração é relativa ao pré-processamento do padrão e construção da tabela de deslocamentos. De fato, o corpo principal do algoritmo não sofreu qualquer modificação. Por esse motivo, no presente trabalho será utilizado o algoritmo BMH, sem as alterações propostas por Sunday.

O algoritmo 3 apresenta o BMH em alto nível.

3.2.1 Complexidade do BMH

Conforme apresentado por Ziviani [4], o pior caso do algoritmo é $O(|P||T|)$, o melhor caso é $O(|T|/|P|)$ e o caso esperado também é $O(|T|/|P|)$. Com relação ao espaço, a implementação é dependente das decisões de projeto. Caso o texto seja colocado em memória, tem-se $O(|T|)$. Caso sejam lidas linha por linha, o custo é $O(|\Sigma|)$, onde Σ representa o alfabeto utilizado. Este custo é relativo ao armazenamento do vetor de deslocamentos d.

3.2.2 Implementação

Para a implementação do algoritmo, foram adotadas as decisões de projeto apresentadas na Seção 3.1.

No conjunto de códigos-fonte, serão encontradas duas pastas:

- `cd comprimido/lbmh/`: contém a implementação do BMHα que trabalha linha a linha;
- `cd comprimido/bmh/`: contém a implementação do BMHβ, que trabalha com o arquivo todo em memória;

Algoritmo 3 BMH(Padrao P, tamanhoPadrao m, Texto T, tamanhoTexto n)

```
para (j ∈ MaxChar) faça
  d[j] = m;
fim para
para (j=1 to m) faça
  d[ord(P[j])] = m - j;
fim para
i = m;
enquanto (i ≤ n) faça
  k = 1;
  j = m;
  enquanto (T[k] = P[j]) and (j>0) faça
    k = k -1;
    j = j-1;
  fim enquanto
  se (j == 0) então
    Reporta casamento na posicao k+1;
  fim se
  i = i + d[ord(T[i+1])];
fim enquanto
```

3.2.3 Descrição de tipos e módulos

Foram utilizados os tipos `TipoTexto` e `TipoPadrao` apresentados por Ziviani [4].

Para o BMHα, os principais módulos são `bmh.h`, que contém declarações de tipos, constantes de interfaces de métodos e o `bmh.c`, que contém a definição dos tipos e a implementação dos métodos. O método mais importante é o `BMH(FILE *p_ArquivoEntrada, TipoPadrao p_Padrao, int *m)`, que recebe o ponteiro para o arquivo de entrada, o padrão a ser buscado no texto e o tamanho do padrão. No método `main(int argc, char *argv[])` ocorre o recebimento dos dados de entrada utilizando as diretivas `getopt`.

Para o BMHβ, os principais módulos são `bmhs.h` e `bmhs.c` (o nome BMHS **não se refere** à implementação do algoritmo de Boyer-Moore-Hoorspool-Sunday. Este nome foi colocado apenas para diferenciar as duas versões do BMH implementadas). O principal método da versão β é `void BMH(long *n, TipoPadrao P, long *m, int p_NumeroLinhas)`, que recebe o tamanho do texto, o padrão a ser buscado, o tamanho do padrão e o número de linhas do texto. Existe outro método importante, chamado `void processaTextoArquivo(FILE *p_ArquivoEntrada, int *p_NumeroLinhas, int *p_TamanhoArquivo)` que carrega o arquivo apontado por `p_ArquivoEntrada` para a memória e retorna o número de linhas e o número de caracteres, que em um texto ASCII representa o tamanho do arquivo.

3.2.4 Utilização dos sistemas

Para utilizar os sistemas, deve-se prosseguir da seguinte maneira, nas respectivas pastas:

```
./bmh.e -f <arquivo_entrada> -p <padrao>
```

Por exemplo, para buscar o padrão *Columbia* no arquivo `wsj88` localizado na pasta `/textos`, deve-se realizar a seguinte chamada:

```
./bmh.e -f /textos -p Columbia
```

3.3 Busca exata em texto comprimido usando o BMH

Nesta seção são apresentadas discussões a cerca da busca em textos comprimidos. Inicialmente será apresentado um algoritmo de compressão de textos em linguagem natural, devido a David Huffman em 1952.

3.3.1 Compactação de dados com Código de Huffman

Em sistemas de informação, é muito interesse obter a compactação da informação sem perda da mesma e com custo de informação reduzido. Para resolver este problema, pode-se utilizar um código de Representação Binária para cada caractere. A solução ideal é achar um Código de Prefixo Livres [5] para cada representação binária. É isso que o Código de Huffman proporciona. Uma forma simples e eficiente de obter uma representação binária única para cada caracter através de uma árvore, e com uma economia de bits para representação de todos os caracteres.

O Código de Huffman gera uma árvore em que as folhas armazenam os símbolos e a freqüência dos mesmos. A seqüência binária que representa o código de prefixo livre é armazenado pela ordem de arcos da árvore. Cada arco a direita de um nó atribui um "0"ao prefixo e cada arco a esquerda atribui um "1".

O algoritmo opera sobre uma lista de prioridades organizada pela frequência da ocorrência de cada símbolo. A cada iteração, são geradas árvores, que são agrupadas até o momento em que somente uma árvore que armazena o código de prefixos livres com todos os elementos do conjunto é gerado. A cada iteração, as duas menores frequências são somadas e colocadas como raiz de uma árvore sem este nó ter qualquer elemento agregado, apenas a soma das menores freqüências da lista. Finalmente, cada raiz é readicionada na lista de prioridade. O Algoritmo 4 apresenta o algoritmo de construção da árvore de Huffman, geradora dos códigos.

1. Na linha 2 o conjunto de Caracteres é inserido todo numa lista de prioridades Q (excelente opção usar um heap);

2. Da linha 3 a 10 ocorre a formação efetiva da árvore de Huffman, por meio das n-1 iterações o for, onde n é a norma do conjunto C;

   (a) Cria-se um novo nó z para o que vai se tornar uma árvore;

[5] Um código de Prefixo Livre mantém as seguinte propriedades:

- Os códigos não possuem o mesmo número de bits, e sim tamanhos variáveis
- Códigos de menor comprimento são atribuídos a caracteres mais frequentes, e maior comprimento aos de menores frequências
- Nenhum código é também um prefixo de algum outro código, eliminando-se assim a possibilidade de ambiguidade.

Algoritmo 4 Árvore de Huffman(Conjunto C)

```
n = |C|;
Q = C;
para (i=1 to n-1) faça
  criar um novo nó z;
  x = fila.desenfileira();
  y = fila.desenfileira();
  esquerda[z] = x;
  direita[z] = y;
  f(z) = f(x)+f(y)
  fila.insere(z);
fim para
return fila.desenfileira();
```

(b) Nas linhas 5 e 6 as duas menores freqüências da lista de prioridades são retiradas desta e colocadas em uma pequena árvore de três elementos, um pai que armazena a freqüência da soma das duas freqüências, o menor a esquerda e o maior a direita e então a raiz da nova árvore é reinserida na lista de prioridades;

3. Ao fim do algoritmo, depois de n - 1 iterações, a raiz da árvore que contem todos os elementos da lista é retornada para posterior manipulação.

Em [3] é apresentado o seguinte fluxo para compactação:

- Varrer o fluxo de entrada para que sejam mapeadas as freqüências;
- Coloca-se o conjunto de pares caractere-frequência numa fila de prioridade baseada nas frequencias;
- Executar sobre a fila de prioridade o Algoritmo de Huffman (4) para a geração da árvore que produz os códigos de prefixo;
- Com o mapa caractere-código de prefixo e entrada são operados os seguintes passos sobre um fluxo de saída, no arquivo compactado:

  1. Grava-se um valor numérico que representa a quantidade de caracteres do nome do arquivo. Em seguida grava-se em representação normal de tamanho fixo o nome do fluxo original;
  2. Grava-se a quantidade de elementos do mapa de decodificação e depois cada elemento do mapa de decodificação, um mapa de carecteres com as chaves sendo o código de hufman para aquele caractere. Uma estrutura de bytes do seguinte tipo: um caractere em notação normal, um valor numérico que armazena quantos dos seguintes bytes representam o código de huffman do caractere.
  3. Grava-se no fluxo, colocando nos bytes adequadamente, a seqüência de bits que representa o fluxo em código de prefixo;

5. Quando o fluxo é fechado, o arquivo de entrada se encontra compactado.

Ziviani [4] apresenta uma implementação semelhante à apresentada pelo fluxo acima, exceto por alguns pontos de implementação. Além disso, Ziviani estende o conceito do código de Huffman e apresenta o **Huffman orientado a bytes** ou **Byte-Huffman**. Esta versão é ainda dividida em outras duas:

- Huffman pleno: uma sequência de bytes é associada a cada palavra do texto. O grau de cada nó passa de 2 para 256.
- Huffman com marcação: utiliza apenas 7 dos 8 bits de um byte para a codificação, e a árvore passa a ter grau 128. Nesse caso, o oitavo bit é usado para marcar o primeiro byte do código da palavra. Esta versão favorece a busca no texto comprimido, e por isso será utilizada no trabalho prático.

Exemplo de compressão

Para realizar uma compressão, é necessário definir qual é o alfabeto utilizado. Neste trabalho prático, foi utilizado o seguinte alfabeto:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
ÇÈÉÊËÌÍÎÏÑÒÓÔÕÖÙÚÛÜàáâãäçèéêëìíîïñòóôõöùúûü
```

A seguir, serão apresentados os resultados de compressão utilizando o algoritmo de Huffman:

Arquivo	Tamanho original(MB)	Tamanho comprimido(MB)	Razão (%)
`wsj89`	2.7	1.2	44
`wsj88_20`	20	7.4	37
`wsj88`	105	39	37

Tabela 29: **Razão de compressão dos arquivos da coleção** `wsj`.

3.3.2 Busca no arquivo comprimido

Ziviani [4] apresenta uma estratégia para se buscar no arquivo comprimido através do código de Huffman com marcação. A busca é feita pelo algoritmo de BMH, mas uma variante como o BMHS pode ser utilizada também. O esquema básico é o seguinte:

1. **Buscar a palavra no Vocabulário**. O Vocabulário consiste de uma tabela de palavras salvas no arquivo comprimido e contém todas as palavras do texto original. Se a palavra não existe no Vocabulário, então ela não existe no texto comprimido e deve-se interromper a execução. Isto é muito vantajoso, pois caso o Vocabulário seja implementado por uma estrutura do tipo *hash*, então o custo para saber se o Padrão está ou não no texto é $O(1)$.

2. **Obter o código de Huffman do Padrão**. Caso a palavra exista no Vocabulário, então deve-se codificar o padrão.

3. **Buscar a palavra no texto**. Busca-se o padrão codificado no texto comprimido, utilizando-se qualquer algoritmo de casamento. Por ser o algoritmo mais eficiente, sugere-se o uso do BMH e suas variantes.

O fluxo apresentado acima é bem pouco detalhado. Um algoritmo que apresenta o processo de busca é apresentado a seguir no Algoritmo 5.

Para implementar a busca, foi utilizado o código-fonte fornecido por Ziviani [4]. Entretanto, o código de Ziviani só permite buscar padrões simples no arquivo comprimido, isto é, padrões que possuem um espaço entre as palavras não são buscados. Já o Algoritmo 5 permite esse tipo de busca, e por isso o código original foi alterado.

Algoritmo 5 Busca no Arquivo comprimido(FILE *p_ArquivoComprimido, Padrao p_Padrao, Texto)

```
Padrao PadraoABuscar;
para p ∈ p_Padrao faça
  se p ∈ Vocabulário então
    Codigo = Descobre código de p
    concatena(PadraoABuscar, codigo)
  senão
    Retorne sem sucesso /*uma das palavras do padrão não existe no vocabulário, logo não é necessário continuar*/
  fim se
  BMH(Texto, p_PadraoBuscar);
  Retorne Número de ocorrências
fim para
```

3.3.3 Utilização do sistema

Para utilizar o sistema, deve-se fornecer a opção desejada:

- `c`: realiza compressão
- `d`: realiza descompressão
- `p`: realiza pesquisa no texto comprimido.

Exemplos de chamadas:

- Compressão do arquivo `Makefile` para gerar o `Makefile.zip`:

  ```
  ./huffman.e c Makefile Makefile.zip
  ```

- Descompressão do arquivo `tp.zip` para gerar o `tp.new`:

  ```
  ./huffman.e d tp.zip tp.new
  ```

- Busca o padrão *gcc* no arquivo `Makefile.zip`:

  ```
  ./huffman.e p Makefile.zip gcc
  ```

3.4 Resultados Computacionais

A seguir serão apresentados os resultados computacionais dos algoritmos implementados. Os resultados são apresentados indivualmente para cada algoritmo, e em seguida eles serão usados para fazer comparações entre as três diferentes abordagens que foram implementadas nesta segunda parte do Trabalho Prático.

Os testes foram feitos seguindo diretivas semelhantes às da Seção 2.6.1. Serão apresentadas tabelas com os resultados individuais e também gráficos com o tempo e número de comparações para o menor e maior padrão do conjunto de padrões da Tabela 4.

3.4.1 Resultados do BMHα

A Tabela 30 apresenta os resultados obtidos pelo algoritmo BMHα quando aplicado sobre o Arquivo `wsj89`.

Padrão	**Real**	**Usuário**	**Sistema**	**Comparações**
DCC UFMG dollar	0.085	0.079	0.003	236692
dia branco	0.084	0.082	0.003	354548
Macunaima administration	0.087	0.083	0.004	192050
Brazilian coffee	0.086	0.080	0.005	263406
New York Stock Exchange	0.090	0.085	0.003	182135
Manacapuru	0.087	0.080	0.004	309156
Canada Treasury	0.085	0.081	0.004	246425
Michael Gregory	0.086	0.079	0.005	250470
price index	0.084	0.084	0.001	328933
Uberaba	0.086	0.085	0.001	459241

Tabela 30: **BMHα: Arquivo wsj89**.

O pequeno tamanho do arquivo permite que o algoritmo realize sua computação de maneira bastante rápida: todos os tempos de execução são bastante pequenos.

A Tabela 31 apresenta os resultados obtidos pelo algoritmo BMHα quando aplicado sobre o `wsj88_20`. Percebe-se um aumento no tempo de Real de computação, principalmente devido ao aumento no tempo de usuário. O número de comparações também é uma ordem de grandeza superior às comparações do arquivo anterior.

Finalmente, a Tabela 32 apresenta os resultados sobre o maior arquivo da coleção, de 105MB. O tempo de processamento aumenta consideravalmente, e também o número de comparações.

Estes resultados são sintetizados nas Figuras 27(a) e 27(b).

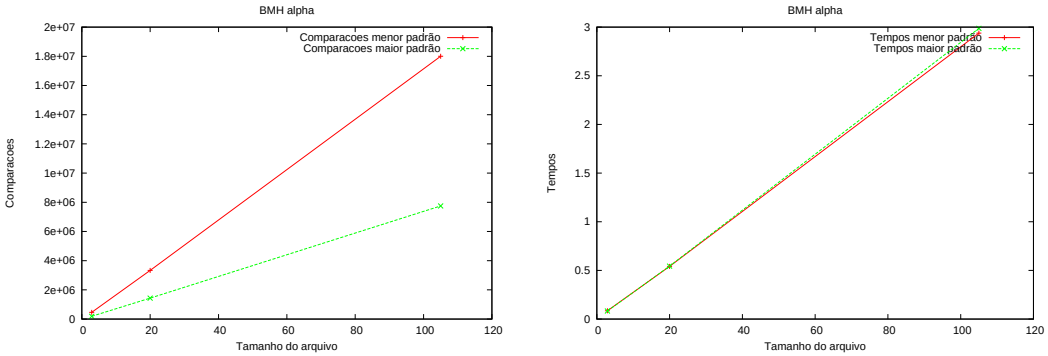
A primeira Figura ilustra que o BMHα caiu no caso esperado, pois quanto maior o padrão, menor o número de comparações. Por outro lado, o algoritmo é bem estável com relação ao tempo de processamento.

Padrão	**Real**	**Usuário**	**Sistema**	**Comparações**
DCC UFMG dollar	0.548	0.528	0.020	1737428
dia branco	0.550	0.533	0.016	2606053
Macunaima administration	0.565	0.544	0.021	1433977
Brazilian coffee	0.767	0.536	0.018	1936241
New York Stock Exchange	0.593	0.548	0.020	1356524
Manacapuru	0.544	0.528	0.015	2251323
Canada Treasury	0.839	0.582	0.009	1816266
Michael Gregory	0.554	0.536	0.016	1837304
price index	0.555	0.540	0.013	2409656
Uberaba	0.818	0.541	0.012	3335124

Tabela 31: **BMHα: Arquivo wsj88_20**.

Padrão	**Real**	**Usuário**	**Sistema**	**Comparações**
DCC UFMG dollar	4.086	2.924	0.062	9385951
dia branco	4.081	2.934	0.057	14058720
Macunaima administration	4.185	2.987	0.050	7749554
Brazilian coffee	4.156	2.980	0.053	10447983
New York Stock Exchange	4.494	3.032	0.109	7339037
Manacapuru	4.029	2.887	0.050	12135517
Canada Treasury	4.079	2.922	0.068	9797333
Michael Gregory	4.061	2.912	0.055	9910739
price index	4.080	2.922	0.062	13005856
Uberaba	4.083	2.938	0.063	18001748

Tabela 32: **BMHα: Arquivo wsj88**.



(a) Comparações (b) Tempos

Figura 27: BMHα. Variação do tamanho do arquivo.

3.4.2 Resultados do BMHβ

As Tabelas 33, 34 e 35 apresentam os resultados obtidos pelo BMH que opera com o texto em memória, de acordo com a variação do tamanho do arquivo de entrada. Foi observada grande variação no tempo de processamento (*User Time*) quando o tamanho do arquivo é aumentado.

O resultado é sintetizado graficamente através das Figuras 28(a) e 28(b).

Padrão	**Real**	**Usuário**	**Sistema**	**Comparações**
DCC UFMG dollar	0.145	0.136	0.007	260750
dia branco	0.141	0.133	0.005	380433
Macunaima administration	0.138	0.125	0.013	222476
Brazilian coffee	0.139	0.132	0.006	290594
New York Stock Exchange	0.151	0.142	0.007	209145
Manacapuru	0.145	0.133	0.006	330642
Canada Treasury	0.138	0.129	0.010	272355
Michael Gregory	0.139	0.133	0.007	275159
price index	0.139	0.131	0.008	355295
Uberaba	0.140	0.129	0.009	481654

Tabela 33: **BMHβ: Arquivo wsj89**.

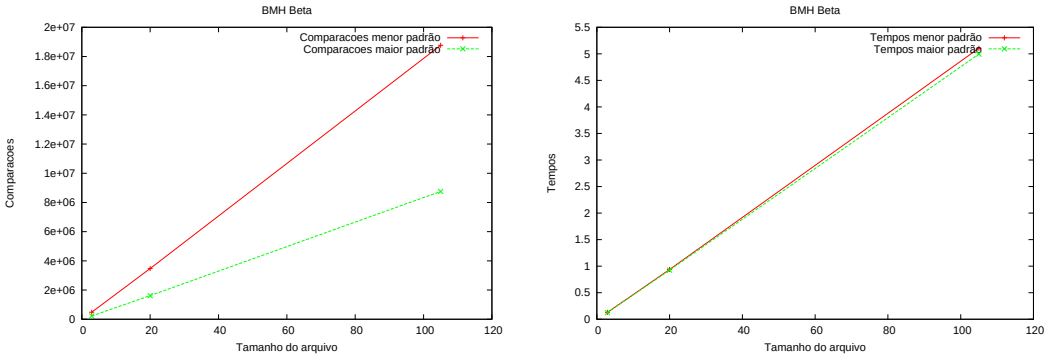
Padrão	**Real**	**Usuário**	**Sistema**	**Comparações**
DCC UFMG dollar	1.726	0.921	0.106	1891602
dia branco	0.983	0.930	0.050	2757510
Macunaima administration	0.973	0.925	0.047	1618916
Brazilian coffee	0.981	0.912	0.065	2123358
New York Stock Exchange	1.436	1.300	0.074	1525476
Manacapuru	0.976	0.943	0.030	2379466
Canada Treasury	0.977	0.938	0.037	1979333
Michael Gregory	0.978	0.938	0.036	2001053
price index	1.022	0.964	0.048	2581155
Uberaba	0.988	0.938	0.046	3473967

Tabela 34: **BMHβ: Arquivo wsj88_20**.

Conclusões semelhantes ao do BMHα podem ser obtidas até o momento. Assintoticamente, ambos os algoritmos possuem o mesmo comportamento. Na Seção de comparações será discutido qual a vantagem de cada um deles de acordo com o cenário.

Padrão	**Real**	**Usuário**	**Sistema**	**Comparações**
DCC UFMG dollar	5.237	4.980	0.244	10223387
dia branco	5.281	5.041	0.230	14886598
Macunaima administration	5.256	4.995	0.248	8753510
Brazilian coffee	5.291	5.051	0.226	11468441
New York Stock Exchange	26.180	17.355	0.470	8254014
Manacapuru	5.533	5.106	0.248	12836780
Canada Treasury	8.490	5.102	0.306	10689181
Michael Gregory	5.509	5.089	0.235	10799573
price index	6.582	6.069	0.264	13944654
Uberaba	5.332	5.105	0.212	18765320

Tabela 35: **BMHβ: Arquivo wsj88**.



(a) Comparações (b) Tempos

Figura 28: BMHβ. Variação do tamanho do arquivo.

3.4.3 Resultados do BMH aplicado ao texto comprimido por Huffman

Finalmente, os resultados do BMH aplicados sobre o texto comprimido com o código de Huffman.

As Tabelas 36, 37 e 38 e apresentam os resultados sobre os arquivos da coleção `wsj` comprimidos, como ilustra a Tabela 29.

Padrão	**Real**	**Usuário**	**Sistema**	**Comparações**
DCC UFMG dollar	0.230	0.190	0.015	173503
dia branco	0.198	0.181	0.017	172396
Macunaima administration	0.194	0.180	0.013	258039
Brazilian coffee	0.197	0.170	0.024	204947
New York Stock Exchange	0.201	0.175	0.022	135919
Manacapuru	0.199	0.181	0.016	509769
Canada Treasury	0.196	0.180	0.015	258380
Michael Gregory	0.196	0.178	0.018	206884
price index	0.195	0.178	0.013	261611
Uberaba	0.197	0.181	0.016	341177

Tabela 36: **Busca no Arquivo wsj89 comprimido**.

Padrão	**Real**	**Usuário**	**Sistema**	**Comparações**
DCC UFMG dollar	0.921	0.870	0.049	0
dia branco	0.953	0.907	0.043	0
Macunaima administration	0.943	0.899	0.045	0
Brazilian coffee	0.957	0.905	0.050	1792369
New York Stock Exchange	1.005	0.935	0.045	946484
Manacapuru	0.936	0.900	0.034	0
Canada Treasury	0.935	0.873	0.060	1821905
Michael Gregory	0.949	0.898	0.048	1796904
price index	0.976	0.918	0.044	1842377
Uberaba	0.940	0.886	0.052	0

Tabela 37: **Busca no Arquivo wsj88_20 comprimido**.

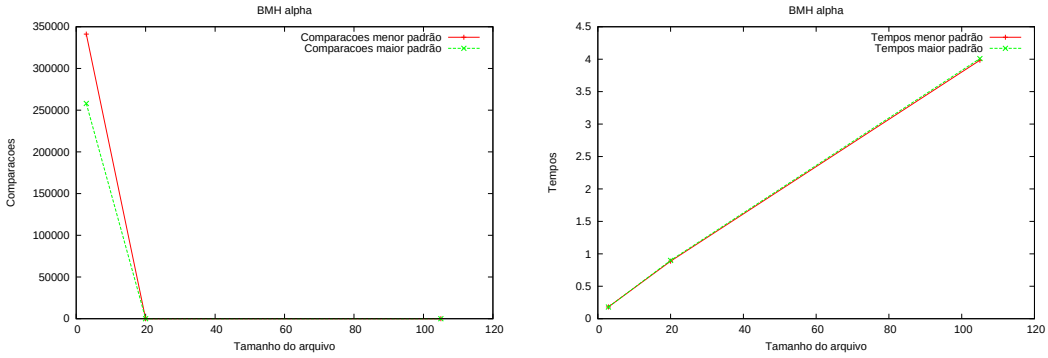
Para o caso dos padrões `Uberaba` (menor padrão) e `Macunaima administration` (maior padrão), obtém-se os gráficos apresentados na Figura 29.

O primeiro gráfico, que relaciona o número de comparações ao tamanho do arquivo possui a forma *estranha* porque, de fato, para o segundo e para o terceiro arquivo da coleção, os padrões procurados não existem no texto. Quando isto ocorre, o padrão não chega sequer a ser buscado pelo BMH, conforme discutido na Seção 3.3.2 e apresentado no Algoritmo 5.

Para que seja observado o comportamento assintótico dos algoritmos, foram escolhidos dois outros padrões: `New York Stock Exchange` e `price index`, pois eles existem nos três arquivos. O resultado é visto na Figura 30.

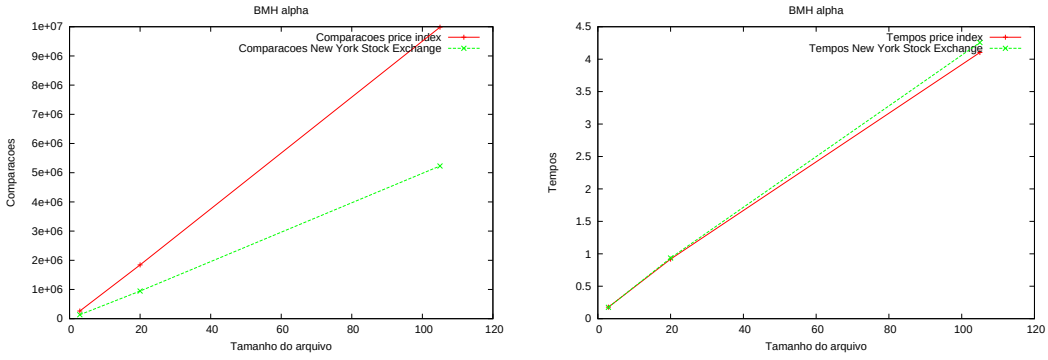
Padrão	Real	Usuário	Sistema	Comparações
DCC UFMG dollar	4.297	4.125	0.160	0
dia branco	4.165	4.008	0.147	0
Macunaima administration	4.175	4.009	0.157	0
Brazilian coffee	4.440	4.274	0.156	9717193
New York Stock Exchange	4.623	4.256	0.164	5230544
Manacapuru	4.173	4.007	0.156	0
Canada Treasury	4.270	4.108	0.149	9883176
Michael Gregory	4.420	4.259	0.149	9728953
price index	4.275	4.103	0.152	9985107
Uberaba	4.143	3.984	0.149	0

Tabela 38: **Busca no Arquivo wsj88 comprimido**.



(a) Comparações (b) Tempos

Figura 29: BMH sobre o arquivo comprimido. Variação do tamanho do arquivo.



(a) Comparações (b) Tempos

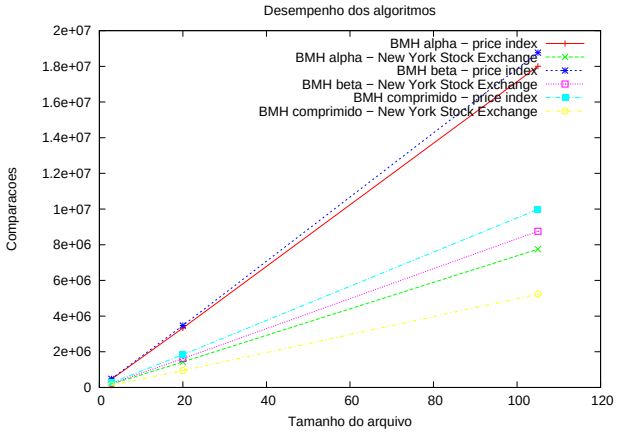
Figura 30: BMH sobre o arquivo comprimido. Variação do tamanho do arquivo considerando novos padrões.

3.5 Resultados computacionais comparativos

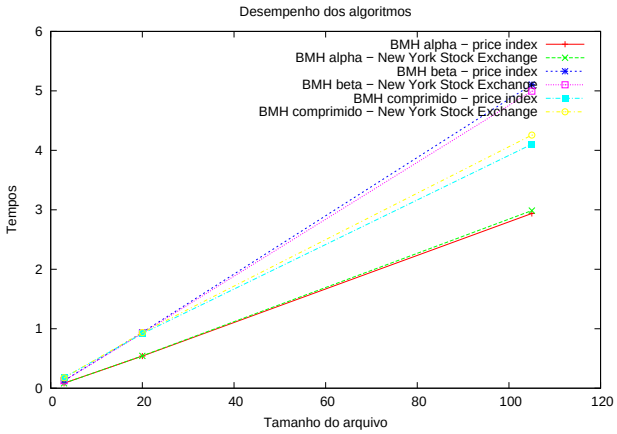
Finalmente, inicia-se o processo comparativo entre as três abordagens implementadas para busca exata.

O primeiro teste a ser realizado será verificar o desempenho do algoritmo para os dois padrões `New York Stock Exchange` e `price index`. Estes padrões foram escolhidos, pois, como esclarecido acima, eles aparecem em todos os arquivos.

As Figuras 31(a) e 31(b) apresentam os resultados computacionais quando se varia o tamanho do arquivo.



(a) Comparações



(b) Tempos

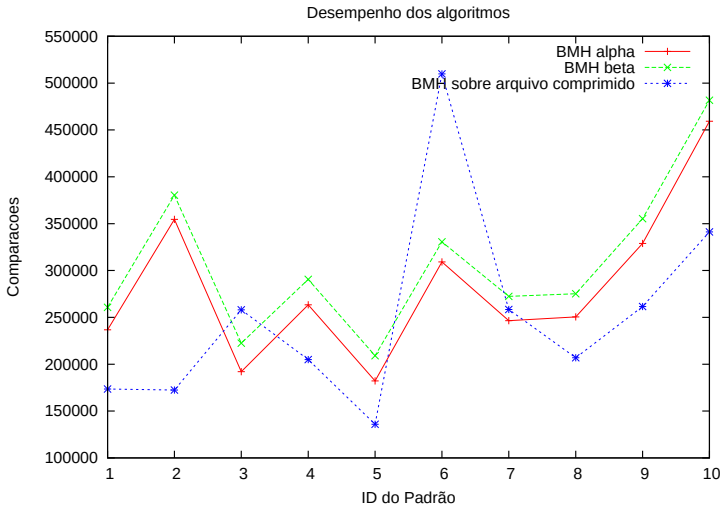
Figura 31: Comparação entre os BMHs desenvolvidos.

Algumas conclusões sobre este primeiro resultado:

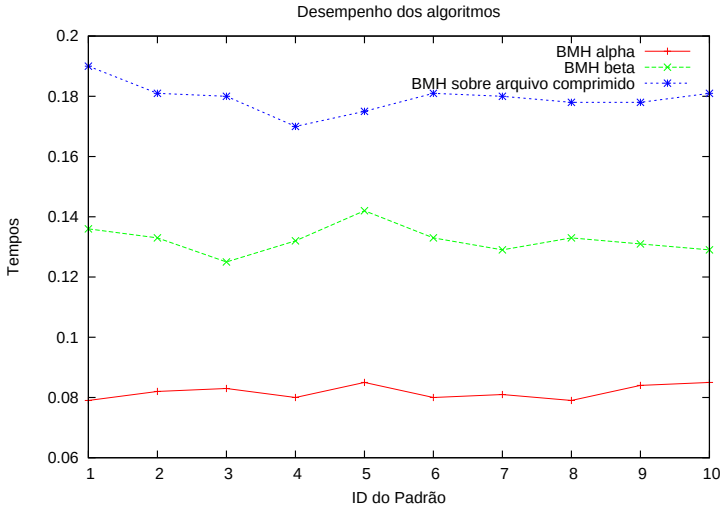
- Quando se comparam padrões específicos, como `price index` para os três algoritmos, o BMH que busca no texto comprimido realiza um número bem menor de comparações. Isto era esperado, pois como o texto está comprimido com códigos menores do que o texto original, o número de comparações deveria ser realmente menor.

As Figuras 32(a), 32(b), 33(a), 33(b), 34(a) e 34(b) apresentam o número de comparações e o tempo gasto para se buscar cada padrão específico em cada um

dos arquivos da coleção. No eixo x de cada gráfico, se encontra o **ID** do padrão, de acordo com a classificação proposta na Tabela 4.



(a) Comparações

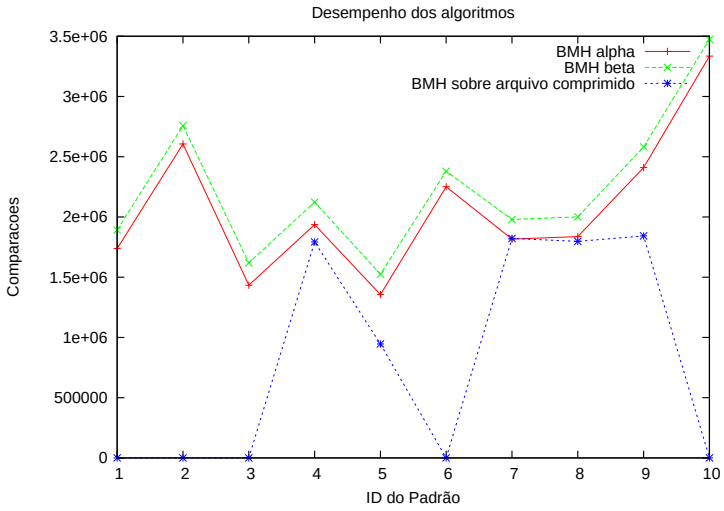


(b) Tempos

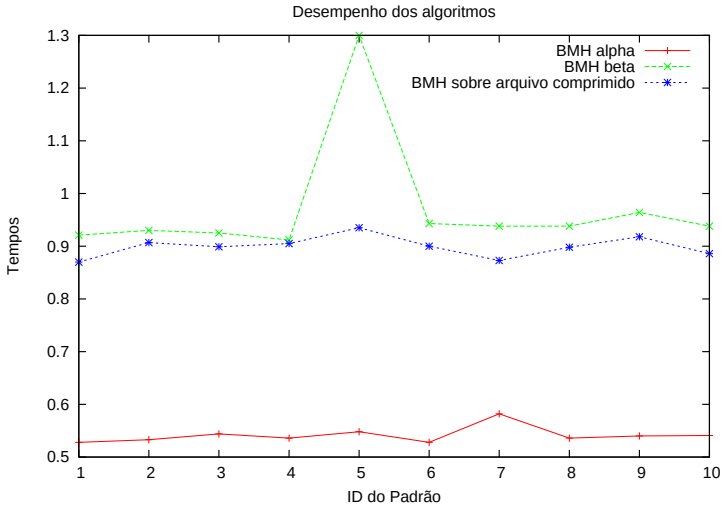
Figura 32: Comparação entre os BMHs desenvolvidos para cada padrão sobre o arquivo `wsj89`.

Fecha-se esta Seção com as seguintes conclusões:

- A busca no texto comprimido realiza um número menor de comparações, como já era esperado.
- Quando se compara o BMHα com a versão BMHβ, vê-se claramente as vantagens das Decisões de Projeto iniciais: a versão α é sempre mais rápida do que a β para se buscar em texto não-comprimido. O algoritmo α é muito rápido e chega mesmo a ser mais rápido do que a versão que busca no texto comprimido, dado o *overhead* associado à busca no texto comprimido: é necessário montar o Vocabulário, fazer busca no Vocabulário, codificar o padrão buscado, etc.



(a) Comparações



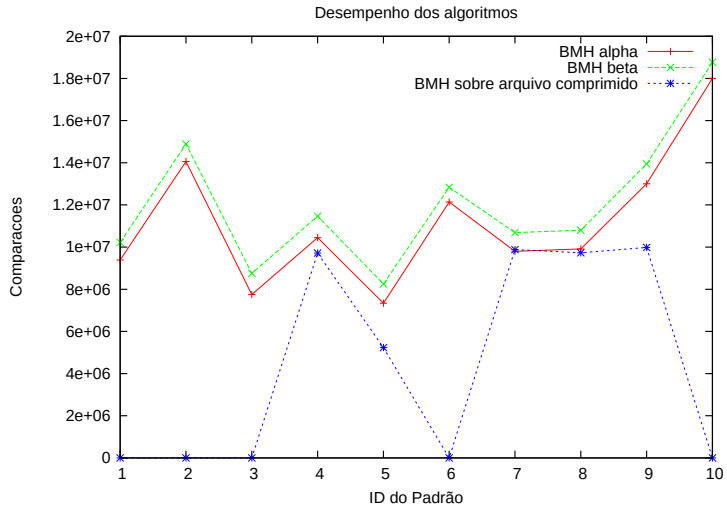
(b) Tempos

Figura 33: Comparação entre os BMHs desenvolvidos para cada padrão sobre o arquivo `wsj88_20`.

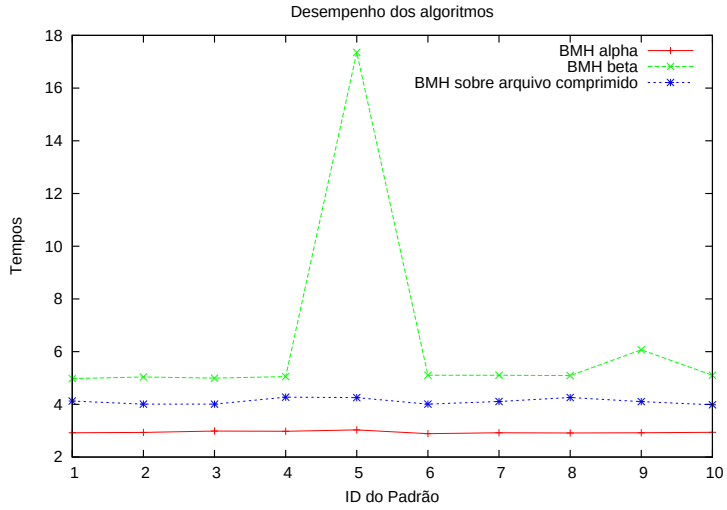
- Ainda com relação às diferentes versões α e β, é interessante observar que o α realiza um número menor de comparações. Isso poderia, à primeira vista, parecer uma incongruência, dado que ambos operam sobre possuem a mesma ordem de complexidade, e esta é determinística. Entretanto, é necessário lembrar que no caso da versão α, os caracteres de nova linha são eliminados das linhas, e isso implica em um número menor de comparações. No caso de arquivos grandes, como o `wsj88` (que contém **milhões de linhas**) esta pequena estratégia causa uma variação considerável.

3.6 Conclusões gerais – Parte II

Nesta segunda parte, foram discutidas as estratégias de busca em texto sem permitir erros nos casamentos. A implementação foi muito produtiva e interessante, principalmente devido aos seguintes fatores:



(a) Comparações



(b) Tempos

Figura 34: Comparação entre os BMHs desenvolvidos para cada padrão sobre o arquivo `wsj88`.

- O conteúdo possui diversas aplicações práticas, de assuntos que vão desde Sistemas Operacionais, passando por desenvolvimento de editores de texto, busca na Web e até Bioinformática.
- Trabalhou-se com arquivos comprimidos. O estudo deste assunto tem-se mostrado interessantíssimo, dado que a cada dia nos vemos mais atolados em uma quantidade cada vez maior de informações.
- Foi interessante notar a importância do planejamento das *Decisões de Projeto*, mesmo sobre um Trabalho Prático. Antes do desenvolvimento começar, foram feitas comparações entre diferentes abordagens, e isso permitiu um desenvolvimento direcionado, que ao final mostrou-se coerente com as decisões tomadas: o BMHα foi bem mais eficiente do que sua versão β, e isso já era esperado de acordo com as decisões da Seção 1.

Gostaria de utilizar este espaço para agradecer aos colegas Gleicy Aparecida Cabral, Daniel Guidoni, Cristiano Gato e Raquel Cabral pelas discussões, sugestões e dúvidas técnicas que foram supridas por eles durante o desenvolvimento deste Trabalho Prático.

4 Listagem dos códigos fonte

Listagem 3: one.h

```
#ifndef _one_h
#define _one_h

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_TAM_LINHA 5000

void procuraLinhaVetorLinhas(long p_Posicao, long p_NumeroLinhas);
void editDistance(char* p_Padrao, char *p_Linha, int p_Erros, int
    p_NumLinha, int *p_NumeroComparacoes, int *p_Ocorrencias);
void processaEntrada (int p_TamEntrada);
int OpenFile(char *nome_arquivo, FILE **fp, char *modo);
int main(int argc, char *argv[]);
#endif
```

Listagem 4: one.c

```
#include "one.h"
#define TRUE 1
#define FALSE 0


/* @chomp: remove a new line identifie <'\n'> from the *string *
 * */
void chomp(char *string)
{
        if (string[strlen(string)-1] == '\n') string[strlen(string)-1]
            = string[strlen(string)];
}

/* @editDistance: implements Seller's algorithm of Dynamic Programming
 * @parameter: p_Padrao: string to be searched
 * @parameter: p_TamanhoTexto: size of the text in number of chars
 * @parameter: p_ArquivoEntrada: input file containing the text
 * @parameter: p_Erros: number of errors allowed for the search
 * @parameter: p_NumLinhas: number of lines in the text
 * */


int i = 0;
int lact = 0;
int pos = 0;
int pC = 0;
int nC = 0;
char y;
int C[200];

void inicia(char *p_Padrao)
{
        int k = 0;
```

```
        int v_Size = strlen(p_Padrao);
        for (k = 0; k<=v_Size; k++)
        {
                C[k] = k;
        }
}

void editDistance(char* p_Padrao, char *p_Linha, int p_Erros, int
    p_NumLinha, int *p_NumeroComparacoes, int *p_Ocorrencias)
{

        int v_TamanhoPadrao = strlen(p_Padrao);
        int p_TamanhoTexto = strlen(p_Linha);
        int v_ImprimeLinha = FALSE;
        int v_NumVezesLinha = 0;

        if (lact == 0)
                lact = p_Erros + 1;

        for (pos=1; pos<=p_TamanhoTexto; pos++)
        {
                pC = 0;
                nC = 0;
                y = p_Linha[pos-1];

                if (y < 0) continue;

                for (i=1; i<=lact; i++)
                {

                        *p_NumeroComparacoes = *p_NumeroComparacoes+1;
                        if (p_Padrao[i-1] == y)
                        {
                                nC = pC;
                        }
                        else
                        {
                                if (pC < nC) nC = pC;
                                if (C[i] < nC) nC = C[i];
                                nC = nC + 1;
                        }
                        pC = C[i];
                        C[i] = nC;
                }
                while ( C[lact] > p_Erros) lact = lact -1 ;
                if (lact == v_TamanhoPadrao)
                {
                        //printf("[%d]: %s\n", p_NumLinha, p_Linha);
                        v_ImprimeLinha = TRUE;
                        v_NumVezesLinha++;
                        *p_Ocorrencias = *p_Ocorrencias + 1;
                }
                else
                {
                        lact = lact + 1;
                }
```

```
            }
            if (v_ImprimeLinha)
            {
                    if (v_NumVezesLinha == 1)
                    {
                            printf("[%d]: %s\n", p_NumLinha,
                               p_Linha);
                    }
                    else
                    {
                            if (v_NumVezesLinha > 1)
                            {
                                    printf("%d x [%d]: %s\n",
                                       v_NumVezesLinha, p_NumLinha,
                                       p_Linha);
                            }
                            else
                            {
                                    printf("Erro. v_EscreverLinha = TRUE
                                       mas n<E3>o tem ocorrencia na linha
                                       .\n");
                                    exit(1);
                            }

                    }
            }

        v_NumVezesLinha = 0;
        v_ImprimeLinha = FALSE;

    }

    /*@processaEntrada: process the values from main.argc.
     *@parameter p_TamEntrada: size of the input. If less than 3, there is
         no enough parameters for the system
     * */
    void processaEntrada (int p_TamEntrada)
    {
            if (p_TamEntrada < 3)
            {
                    printf("Error at call. A chamada padrão deve ser: ./
                       dinamica.e -e #erros permitidos -f arquivo_entrada
                       -p padrao]\n");
                    exit(1);
            }

    }

    /* @OpenFile: open the file with the name *nome_arquivo and returns a
        pointer at **fp
     * @parameter modo: "r"=read, "w"=write ... C default ways for openning
          a file. *
     * */

    int OpenFile(char *nome_arquivo, FILE **fp, char *modo)
```

```
{
        if ((*fp = fopen (nome_arquivo, modo)) == NULL)
        {
                printf("Erro na abertura do arquivo de entrada.\n");
                exit(-1);
        }
        return 0;
}

/*
 * @main: my main
 * @see doc of the system for explanation about input parameters by
     getopt.
 * @author: fred
 * */

int main(int argc, char *argv[])
{

        FILE *v_InputFile;
        char  v_Padrao[100];
        int v_NumErrosPermitidos = 0;
        char v_NomeArquivo[100];
        int v_NumLinha = 0;
        char v_Linha[MAX_TAM_LINHA];
        int v_NumeroComparacoes = 0;
        int i = 0;
        int v_Ocorrencias = 0 ;


        while ( (i = getopt(argc, argv, "e:f:p:")) != -1)
        {
                switch (i)
                {
                        case 'e':
                                v_NumErrosPermitidos = atoi(optarg);
                        break;

                        case 'f':
                                strcpy(v_NomeArquivo, optarg);
                        break;

                        case 'p':
                                strcpy(v_Padrao, optarg);
                        break;

                }
        }

        OpenFile(v_NomeArquivo, &v_InputFile, "r");
        v_NumLinha = 1;

        inicia(v_Padrao);
        while (fgets(v_Linha, MAX_TAM_LINHA, v_InputFile))
        {
                chomp(v_Linha);
```

```
                editDistance(v_Padrao,v_Linha,v_NumErrosPermitidos,
                    v_NumLinha, &v_NumeroComparacoes, &v_Ocorrencias);
                v_NumLinha++;
        }
        printf("\nComparacoes: %d\nOcorrencias: %d\n\n",
            v_NumeroComparacoes, v_Ocorrencias);
        fclose(v_InputFile);
        return 0;
}
```

Listagem 5: shiftand.h

```
#ifndef _shift_and
#define _shift_and

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

typedef char *TipoTexto;
typedef char *TipoPadrao;
void ShiftAndAproximado(FILE *p_Arquivo ,TipoPadrao P, long *m, long *k
    , int *p_Comparacoes, int *p_Ocorrencias);
int OpenFile(char *nome_arquivo, FILE **fp, char *modo);
int main(int argc, char **argv);
void chomp(char *string);
#endif
```

Listagem 6: shiftand.c

```
#include "shiftand.h"

#define Maxchar          256
#define NumMaxErros      10
#define TRUE    1
#define FALSE 0
#define MAX_TAM_LINHA 5000

/* @chomp: remove a new line identifie <'\n'> from the *string
 * */
void chomp(char *string)
{
        if (string[strlen(string)-1] == '\n') string[strlen(string)-1]
            = string[strlen(string)];
}


void ShiftAndAproximado(FILE *p_Arquivo ,TipoPadrao P, long *m, long *k
    , int *p_Comparacoes, int *p_Ocorrencias)
{ long Masc[Maxchar];
  long i, j, Ri, Rant, Rnovo;
  long R[NumMaxErros + 1];
  char linha[MAX_TAM_LINHA];
  int v_NumLinha = 1;
  int v_ocor = 0;
  int v_EscreverLinha = 0;
```

```
25    int v_NumVezesLinha = 0;

      for (i = 0; i < Maxchar; i++)
        Masc[i] = 0;
      for (i = 1; i <= *m; i++)
30      { Masc[P[i-1] + 127] |= 1 << (*m - i); }
      R[0] = 0;
      Ri = 1 << (*m - 1);
      for (j = 1; j <= *k; j++)
        R[j] = (1 << (*m - j)) | R[j-1];
35
      while (fgets(linha,MAX_TAM_LINHA, p_Arquivo))
      {
       chomp(linha);
       for (i = 0; i < strlen(linha); i++)
40      {
          Rant = R[0];
          Rnovo = ((((unsigned long)Rant) >> 1) | Ri) & Masc[linha[i] +
              127];
          R[0] = Rnovo;
          *p_Comparacoes = *p_Comparacoes + 1;
45        for (j = 1; j <= *k; j++)
            { Rnovo = ((((unsigned long)R[j]) >> 1) & Masc[linha[i] + 127])
                      | Rant | (((unsigned long)(Rant | Rnovo)) >> 1);
              Rant = R[j];
              R[j] = Rnovo | Ri;
50            *p_Comparacoes = *p_Comparacoes + 1;
            }
          if ((Rnovo & 1) != 0)
          {
    //      printf("[%d]: %s\n", v_NumLinha, linha);
55          v_EscreverLinha = TRUE;
            v_NumVezesLinha++;
            v_ocor++;
          }
        }
60    if (v_EscreverLinha)
       {
                    if (v_NumVezesLinha == 1)
                    {
                                printf("[%d]: %s\n", v_NumLinha, linha)
                                    ;
65                  }
                    else
                    {
                            if (v_NumVezesLinha > 1)
                            {
70                                  printf("%d x [%d]: %s\n",
                                        v_NumVezesLinha, v_NumLinha, linha)
                                        ;
                            }
                            else
                            {
                                  printf("Erro. v_EscreverLinha = TRUE
                                      mas não tem ocorrencia na linha.\n"
                                      );
75                                exit(1);
```

```
                        }
                    }
                }
                v_NumLinha++;
                v_NumVezesLinha = 0;
                v_EscreverLinha = FALSE;
            }
            *p_Ocorrencias = v_ocor;
        }

        int OpenFile(char *nome_arquivo, FILE **fp, char *modo)
        {
        if ((*fp = fopen (nome_arquivo, modo)) == NULL)
         {
          printf("Erro na abertura do arquivo de entrada.\n");
          exit(-1);
         }
         return 0;
        }

        void processaEntrada(int p_Argc)
        {
                if (p_Argc < 3)
                {
                        printf("Error at call: $->shiftand [errors] [inputfile]
                            [padrao]\n");
                        exit(-1);
                }
        }


        int main(int argc, char **argv)
        {

                TipoPadrao v_Padrao;
                long k = 0;
                FILE *v_ArquivoEntrada;
                char v_NomeArquivo[100];
                char v_StrPadrao[100];
                long v_TamPadrao;
                int i = 0 ;
                int v_NumeroComparacoes = 0;
                int v_Ocorrencias = 0;

                while ( (i = getopt(argc, argv, "e:f:p:")) != -1)
                {
                        switch (i)
                        {
                                case 'e':
                                        k = atoi(optarg);
                                break;

                                case 'f':
                                        strcpy(v_NomeArquivo, optarg);
```

```
                        break;

                        case 'p':
                                strcpy(v_StrPadrao, optarg);
                        break;

                }
        }

        OpenFile(v_NomeArquivo, &v_ArquivoEntrada, "r");
        v_Padrao = (TipoPadrao) v_StrPadrao;
        v_TamPadrao = strlen(v_StrPadrao);
        ShiftAndAproximado(v_ArquivoEntrada, v_Padrao, &v_TamPadrao,&k
            ,&v_NumeroComparacoes, &v_Ocorrencias );

        printf("\nComparacoes: %d\n", v_NumeroComparacoes);
        printf("Ocorrencias: %d\n\n", v_Ocorrencias);

        fclose(v_ArquivoEntrada);
        return 0;
}
```

Listagem 7: bmh.h

```
#ifndef _bmhs_
#define _bmhs_

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define Maxchar          256
#define NumMaxErros      10

#define TRUE 1
#define FALSE 0

#define MAX_TAM_LINHA 5000

typedef char *TipoTexto;
typedef char *TipoPadrao;

TipoTexto m_Texto;

int * v_VetorLinhas;

int OpenFile(char *nome_arquivo, FILE **fp, char *modo);
void processaEntrada(int p_Argc);
int main(int argc, char **argv);
void BMH(FILE *p_ArquivoEntrada, TipoPadrao P, long *m, int *
    p_NumeroComparacoes, int *p_Ocorrencias);
#endif
```

Listagem 8: bmh.c

```
#include "bmh.h"
```

```
/* @chomp: remove a new line identifie <'\n'> from the *string *
 * */
void chomp(char *string)
{
        if (string[strlen(string)-1] == '\n') string[strlen(string)-1]
           = string[strlen(string)];
}



/* @BMH: implements the Boyer-Morre-Hoorspool Algorithm
 * @parameter p_ArquivoEntrada: a pointer for the input file
 * @parameter P: the string to be matched
 * @parameter m: a pointer for the size of P
 * @parameter p_NumeroComparacoes: a pointer that returns the number of
      comparisons
 * */

void BMH(FILE *p_ArquivoEntrada, TipoPadrao P, long *m, int *
   p_NumeroComparacoes, int *p_Ocorrencias)
{
  long i, j, k;
  long d[Maxchar + 1];
     i = *m;
  char linha[MAX_TAM_LINHA];
  int n;
  int v_Linha = 1;
  int v_ImprimeLinha = FALSE;
  int v_NumVezesLinha = 0;


  while (fgets(linha, MAX_TAM_LINHA, p_ArquivoEntrada))
  {
        chomp(linha);

        for (j = 0; j <= Maxchar; j++)
                d[j] = *m;

        for (j = 1; j < *m; j++)
                d[(int)P[j-1]] = *m - j;

        n = strlen(linha);
        i = *m;

        while (i <= n)
        {
                k = i;
                j = *m;
                while (linha[k-1] == P[j-1] && j > 0)
                {
                        k--;
                        j--;
                        *p_NumeroComparacoes = *p_NumeroComparacoes +
                            1;
```

```
55                  }

                    *p_NumeroComparacoes = *p_NumeroComparacoes + 1;

                    if (j == 0)
60                  {
                            v_ImprimeLinha = TRUE;
                            v_NumVezesLinha++;
                            *p_Ocorrencias = *p_Ocorrencias + 1;
                    }
65
                    i += d[(int)linha[i-1]];
            }
            if (v_ImprimeLinha)
            {
70              if (v_NumVezesLinha == 1)
                    {
                                printf("[%d]: %s\n", v_Linha, linha);
                    }
                    else
75                  {
                            if (v_NumVezesLinha > 1)
                            {
                                printf("%d x [%d]: %s\n",
                                    v_NumVezesLinha, v_Linha, linha);
                            }
80                          else
                            {
                                printf("Erro. v_EscreverLinha = TRUE
                                    mas n<E3>o tem ocorrencia na linha
                                    .\n");
                                exit(1);
                            }
85
                    }
            }
            v_Linha++;
            v_NumVezesLinha = 0;
90          v_ImprimeLinha = FALSE;
    }
}


95  int OpenFile(char *nome_arquivo, FILE **fp, char *modo)
    {
    if ((*fp = fopen (nome_arquivo, modo)) == NULL)
     {
      printf("Erro na abertura do arquivo de entrada.\n");
100   exit(-1);
     }
     return 0;
    }

105 void processaEntrada(int p_Argc)
    {
            if (p_Argc < 2)
            {
```

```
                printf("Erro na chamada do programa.\n");
                exit(-1);
        }
}

int main(int argc, char **argv)
{


        TipoPadrao v_Padrao;
        FILE *v_ArquivoEntrada;
        int v_TamanhoArquivo;
        char v_NomeArquivo[100];
        char v_StrPadrao[100];
        long v_TamArquivo;
        long v_TamPadrao;
        int v_NumeroComparacoes = 0;
        int v_Ocorrencias = 0 ;


        int i = 0;


        while ( (i = getopt(argc, argv, "f:p:")) != -1)
        {
                switch (i)
                {
                        case 'f':
                                strcpy(v_NomeArquivo, optarg);
                        break;

                        case 'p':
                                strcpy(v_StrPadrao, optarg);
                        break;

                }
        }

        OpenFile(v_NomeArquivo, &v_ArquivoEntrada, "r");

        v_Padrao = (TipoPadrao)v_StrPadrao;

        v_TamArquivo = (long)v_TamanhoArquivo;
        v_TamPadrao = (long)strlen(v_StrPadrao);

        BMH(v_ArquivoEntrada, v_Padrao, &v_TamPadrao, &
            v_NumeroComparacoes, &v_Ocorrencias);
        printf("\nComparacoes: %d\nOcorrencias: %d\n\n",
            v_NumeroComparacoes, v_Ocorrencias);
        fclose(v_ArquivoEntrada);

        return 0;
}
```

Listagem 9: huffman.c

```
#include <stdio.h>
```

```
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define Vazio           "!!!!!!!!!!\0"
#define Retirado        "**********\0"


#define M               3000000
#define N               50   /* Tamanho da chave */
#define BaseNum         128  /* Base numerica que o algoritmo trabalha
    */
#define MaxAlfabeto     255
/* Constante utilizada em ExtraiProximaPalavra */
#define MaxTamVetoresBO  10
#define TRUE  1
#define FALSE 0

typedef int Apontador;

typedef char TipoChave[N];
typedef int TipoPesos[N + 1];

typedef struct TipoItem {
  TipoChave Chave;
  int Freq, Ordem;
} TipoItem;

typedef int Indice;

typedef TipoItem *TipoDicionario;
typedef short TipoAlfabeto[MaxAlfabeto + 1];

typedef struct TipoBaseOffset {
  int Base, Offset;
} TipoBaseOffset;

typedef TipoBaseOffset TipoVetoresBO[MaxTamVetoresBO + 1];
typedef char TipoPalavra[256];

typedef TipoPalavra *TipoVetorPalavra;

void GeraPesos(TipoPesos p)
{ int i;
 for (i = 0; i < N; i++)
    p[i] = 1 + (unsigned int )rand();
}

Indice h(TipoChave Chave, TipoPesos p)
{ int i; unsigned int Soma = 0;
  for (i = 0; i < strlen(Chave); i++)
    Soma += (unsigned int)Chave[i] * p[i];
  return (Soma % M);
}

void Inicializa(TipoDicionario T)
{ int i;
```

```
  for (i = 0; i < M; i++)
  {
    memcpy(T[i].Chave, Vazio, N + 1);
    T[i].Freq = 0;
  }

}

Apontador Pesquisa(TipoChave Ch, TipoPesos p, TipoDicionario T)
{ unsigned int  i = 0;
  unsigned int  Inicial;

  Inicial = h(Ch, p);
  while ( strcmp (T[(Inicial + i) % M].Chave,Vazio) != 0 &&
          strcmp ( T[(Inicial + i) % M].Chave, Ch) != 0 && i < M)
    i++;

  if (strcmp ( T[(Inicial + i) % M].Chave, Ch) == 0)
  return ((Inicial + i) % M);
  else
    return M;    /* Pesquisa sem sucesso */
}

void Insere(TipoItem x, TipoPesos p, TipoDicionario T)
{ unsigned int i = 0;
  unsigned int Inicial;
  if (Pesquisa(x.Chave, p, T) < M)
  { printf("Elemento ja esta presente\n");
    return;
  }
  Inicial = h(x.Chave, p);
  while ( strcmp ( T[(Inicial + i) % M].Chave,Vazio) != 0 &&
          strcmp ( T[(Inicial + i) % M].Chave, Retirado) != 0 && i < M)
     i++;
  if (i < M)
  { strcpy (T[(Inicial + i) % M].Chave, x.Chave);
    T[(Inicial + i) % M].Freq = x.Freq;
    T[(Inicial + i) % M].Ordem = x.Ordem;
  }
  else
    printf(" Tabela cheia\n");
}

void Retira(TipoChave Ch, TipoPesos p, TipoDicionario T)
{ Indice i;
  i = Pesquisa(Ch, p, T);
  if (i < M)
  memcpy(T[i].Chave, Retirado, N);
  else
    printf("Registro nao esta presente\n");
}

void imprime(TipoDicionario tabela)
{ int  i, j;
  for (i = 0; i <= M; i++)
  { printf("%d  ", i);
    for (j = 0; j < N ; j++)
```

```
115         putchar(tabela[i].Chave[j]);
        printf(" -- %4d -- %4d\n", tabela[i].Freq, tabela[i].Ordem);
      }
    }

120 /* Inicio dos procedimentos do Extrator */
    void DefineAlfabeto(TipoAlfabeto Alfabeto, FILE *ArqAlf)
    {  /* Os Simbolos devem estar juntos em uma linha no arquivo */
      char Simbolos[MaxAlfabeto + 1];
      int i;
125   char *Temp;
      for (i = 0; i <= MaxAlfabeto; i++)
        Alfabeto[i] = FALSE;
      fgets(Simbolos, MaxAlfabeto + 1, ArqAlf);
      Temp = strchr(Simbolos, '\n');
130   if (Temp != NULL)
        *Temp = 0;
      for (i = 0; i <= strlen(Simbolos) - 1; i++)
        Alfabeto[Simbolos[i] + 127] = TRUE;
      Alfabeto[0] = FALSE;    /* caractere de codigo zero: separador */
135 }


    void ExtraiProximaPalavra (TipoPalavra Result, int *Indice, char *Linha
        ,
                               FILE *ArqTxt, TipoAlfabeto Alfabeto)
140 {
      short FimPalavra = FALSE, Aux = FALSE;
      Result[0] = '\0';
      if (*Indice > strlen(Linha)) {
        if (fgets(Linha, MaxAlfabeto + 1,ArqTxt)){
145       /* Coloca um delimitador em Linha */
          sprintf(Linha + strlen(Linha), "%c", (char)0);
          *Indice = 1;
        } else {sprintf(Linha, "%c", (char)0); FimPalavra = TRUE;}
      }
150   while (*Indice <= strlen(Linha) && !FimPalavra) {
        if (Alfabeto[Linha[*Indice - 1] + 127]) {
          sprintf(Result + strlen(Result), "%c", Linha[*Indice - 1]);
          Aux = TRUE;
        } else {
155       if (Aux) {
            if (Linha[*Indice - 1] != (char)0)
              (*Indice)--;
          } else{
            sprintf(Result + strlen(Result), "%c", Linha[*Indice - 1]);
160       }FimPalavra = TRUE;
        }
        (*Indice)++;
      }
    }
165 char * Trim(char *str) {
      int i, j, len;
      char * strtmp = malloc(sizeof(char[strlen(str)]));
      strcpy(strtmp, str);
      len = strlen(strtmp);
170   i = 0;
```

```
    while((i < len) && ((strtmp[i] == ' ') || (strtmp[i] == '\t') ||
          /*(strtmp[i] == '\n') //*/ (strtmp[i] == '\r'))) {
      i++;
    }
    j = len - 1;
    while((j >= 0) && ((strtmp[j] == ' ') || (strtmp[j] == '\t') ||
      /*(strtmp[i] == '\n') //*/ (strtmp[i] == '\r'))) {
      j--;
    }
    if (j >= 0) str[j + 1] = '\0';
    if (i <= j) {
      strcpy(strtmp, strtmp + i);
    } else {
      strcpy(strtmp, "");
    }
    return strtmp;
  }

  /* Procedimentos da Compressao e Descompressao */
  void PrimeiraEtapa(FILE *ArqTxt, TipoAlfabeto Alfabeto, int *Indice,
                     TipoPalavra Palavra, char *Linha, TipoDicionario
                        Vocabulario,
                     TipoPesos p)
  {
    TipoItem Elemento;
    int i;
    do {
      ExtraiProximaPalavra(Palavra, Indice, Linha, ArqTxt, Alfabeto);
      memcpy(Elemento.Chave, Palavra, sizeof(TipoChave));
      Elemento.Freq = 1;
      if (*Palavra != '\0') {
        i = Pesquisa(Elemento.Chave, p, Vocabulario);
        if (i < M)
          Vocabulario[i].Freq++;
        else
          Insere(Elemento, p, Vocabulario);
        do {
          ExtraiProximaPalavra(Palavra, Indice, Linha, ArqTxt, Alfabeto);
          memcpy(Elemento.Chave, Palavra, sizeof(TipoChave));
          /* O primeiro espaco depois da palavra nao e codificado */
          if (strcmp(Trim(Palavra), "") && (*Trim(Palavra)) != (char)0) {
            i = Pesquisa(Elemento.Chave, p, Vocabulario);
            if (i < M)
              Vocabulario[i].Freq++;
            else
              Insere(Elemento, p, Vocabulario);
          }
        } while (strcmp(Palavra, ""));
      }
    } while (Palavra[0] != '\0');
  }
  void CalculaCompCodigo(TipoDicionario A, int n)
  {
    int u = 0;    /* Nodos internos usados */
    int h = 0;    /* Altura da arvore */
    int NoInt;    /* Numero de nodos internos */
    int Prox, Raiz, Folha;
```

```
    int Disp = 1;
    int x, Resto;
    if (n > BaseNum - 1) {
230   Resto = 1 + ((n - BaseNum) % (BaseNum - 1));
      if (Resto < 2) Resto = BaseNum;
    } else Resto = n - 1;
    NoInt = 1 + ((n - Resto) / (BaseNum - 1));
    for (x = n - 1; x >= (n - Resto) + 1; x--)
235   A[n].Freq += A[x].Freq;
    /* Primeira Fase */
    Raiz = n;
    Folha = n - Resto;
    for (Prox = n - 1; Prox >= (n - NoInt) + 1; Prox--) {
240   /* Procura Posicao */
      if (Folha < 1 || Raiz > Prox && A[Raiz].Freq <= A[Folha].Freq)
      {  /* No interno */
        A[Prox].Freq = A[Raiz].Freq;
        A[Raiz].Freq = Prox;
245     Raiz--;
      } else { /* No-folha */
        A[Prox].Freq = A[Folha].Freq;
        Folha--;
      }
250   /* Atualiza Frequencias */
      for (x = 1; x <= BaseNum - 1; x++) {
        if (Folha < 1 || Raiz > Prox && A[Raiz].Freq <= A[Folha].Freq)
        {  /* No interno */
          A[Prox].Freq += A[Raiz].Freq;
255       A[Raiz].Freq = Prox;
          Raiz--;
        } else { /* No-folha */
          A[Prox].Freq += A[Folha].Freq;
          Folha--;
260     }
      }
    }
    /* Segunda Fase */
    A[Raiz].Freq = 0;
265 for (Prox = Raiz + 1; Prox <= n; Prox++) {
      A[Prox].Freq = A[A[Prox].Freq].Freq + 1;
    }
    /* Terceira Fase */
    Prox = 1;
270 while (Disp > 0) {
      while (Raiz <= n && A[Raiz].Freq == h) {
        u++;
        Raiz++;
      }
275   while (Disp > u) {
        A[Prox].Freq = h;
        Prox++;
        Disp--;
        if (Prox > n) {
280       u = 0;
          break;
        }
      }
```

```
      Disp = BaseNum * u;
      h++;
      u = 0;
    }
  }

void Particao(Indice Esq, Indice Dir, Indice *i, Indice *j,
    TipoDicionario A)

{ TipoItem x, w;
  *i = Esq; *j = Dir;
  x = A[(*i + *j) / 2];    /* obtem o pivo x */
  do
    { while (x.Freq < A[*i].Freq) (*i)++;
      while (x.Freq > A[*j].Freq) (*j)--;
      if (*i <= *j)
      { w = A[*i]; A[*i] = A[*j]; A[*j] = w;
        (*i)++; (*j)--;
      }
    } while (*i <= *j);
}

void Ordena(Indice Esq, Indice Dir, TipoDicionario A)
{ Indice i, j;
  Particao(Esq, Dir, &i, &j, A);
  if (Esq < j) Ordena(Esq, j, A);
  if (i < Dir) Ordena(i, Dir, A);
}
void QuickSort(TipoDicionario A, Indice *n)
{ Ordena(1, *n, A);
}

Indice OrdenaPorFrequencia(TipoDicionario Vocabulario)
{
  Indice i;
  Indice n = 1;
  TipoItem Item;

  Item = Vocabulario[1];
  for (i = 0; i <= M - 1; i++) {
    if (strcmp(Vocabulario[i].Chave, Vazio)) {
      if (i != 1) {
        Vocabulario[n] = Vocabulario[i];
        n++;
      }
    }
  }
  if (strcmp(Item.Chave, Vazio))
    Vocabulario[n] = Item;
  else
    n--;
  QuickSort(Vocabulario, &n);
  return n;
}


/* Procedimento para gravar um numero inteiro em um arquivo de bytes */
```

```
void GravaNumInt(FILE *ArqComprimido, int Num)
{
  fwrite(&Num, sizeof(int), 1, ArqComprimido);
}


/* Procedimento para ler um numero inteiro de um arquivo de bytes */
int LeNumInt(FILE *ArqComprimido)
{
  int Num;
  fread(&Num, sizeof(int), 1, ArqComprimido);
  return Num;
}


int ConstroiVetores(TipoVetoresBO VetoresBaseOffset,
  TipoDicionario Vocabulario, int n, FILE *ArqComprimido)
{
  int Wcs[MaxTamVetoresBO + 1];
  int i, MaxCompCod;

  MaxCompCod = Vocabulario[n].Freq;
  for (i = 1; i <= MaxCompCod; i++)
  {
    Wcs[i] = 0; VetoresBaseOffset[i].Offset = 0;
  }
  for (i = 1; i <= n; i++) {
    Wcs[Vocabulario[i].Freq]++;
    VetoresBaseOffset[Vocabulario[i].Freq].Offset = i - Wcs[Vocabulario
        [i].Freq] + 1;
  }
  VetoresBaseOffset[1].Base = 0;
  for (i = 2; i <= MaxCompCod; i++) {
    VetoresBaseOffset[i].Base = BaseNum * (VetoresBaseOffset[i-1].Base
        + Wcs[i-1]);
    if (VetoresBaseOffset[i].Offset == 0)
      VetoresBaseOffset[i].Offset = VetoresBaseOffset[i-1].Offset;
  }
  /* Salvando as tabelas em disco */
  GravaNumInt(ArqComprimido, MaxCompCod);
  for (i = 1; i <= MaxCompCod; i++) {
    GravaNumInt(ArqComprimido, VetoresBaseOffset[i].Base);
    GravaNumInt(ArqComprimido, VetoresBaseOffset[i].Offset);
  }
  return MaxCompCod;
}


int SegundaEtapa(TipoDicionario Vocabulario,
                 TipoVetoresBO VetoresBaseOffset, TipoPesos p,
                 FILE *ArqComprimido)
{
  int Result, i, j, NumNodosFolhas, PosArq;
  TipoItem Elemento;
  char Ch;
  TipoPalavra Palavra;
```

```
395     NumNodosFolhas = OrdenaPorFrequencia(Vocabulario);
        CalculaCompCodigo(Vocabulario, NumNodosFolhas);
        Result = ConstroiVetores(VetoresBaseOffset, Vocabulario,
            NumNodosFolhas,
                                 ArqComprimido);
        /* Grava Vocabulario */
400     GravaNumInt(ArqComprimido, NumNodosFolhas);
        PosArq = ftell(ArqComprimido);
        for (i = 1; i <= NumNodosFolhas; i++) {
          j = 1;
          while (Vocabulario[i].Chave[j-1] != (char)0) {
405         fwrite(&Vocabulario[i].Chave[j-1], sizeof(char), 1, ArqComprimido
                );
            j++;
          }
          Ch = (char)0;
          fwrite(&Ch, sizeof(char), 1, ArqComprimido);
410     }
        /* Le e reconstroi a condicao de hash no vetor que contem o
            vocabulario */
        fseek(ArqComprimido, PosArq, SEEK_SET);
        Inicializa(Vocabulario);
        for (i = 1; i <= NumNodosFolhas; i++) {
415       *Palavra = '\0';
          do {
            fread(&Ch, sizeof(char), 1, ArqComprimido);
            if (Ch != (char)0)
              sprintf(Palavra + strlen(Palavra), "%c", Ch);
420       } while (Ch != (char)0);
          memcpy(Elemento.Chave, Palavra, sizeof(TipoChave));
          Elemento.Ordem = i;
          j = Pesquisa(Elemento.Chave, p, Vocabulario);
          if (j >= M)
425         Insere(Elemento, p, Vocabulario);
        }
        return Result;
      }

430
      void Escreve(FILE *ArqComprimido, int *Codigo, int *c)
      {
        unsigned char Saida[MaxTamVetoresBO + 1];
        int i = 1;
435     int cTmp;
        int LogBase2 = (int)round(log(BaseNum) / log(2.0));
        int Mask = (int)pow(2, LogBase2) -1;
        cTmp = *c;
        Saida[i] = ((unsigned)(*Codigo)) >> ((*c - 1) * LogBase2);
440     if (LogBase2 == 7) Saida[i] = Saida[i] | 0x80;
        i++;
        (*c)--;
        while (*c > 0) {
          Saida[i] = (((unsigned)(*Codigo)) >> ((*c - 1) * LogBase2)) & Mask;
445       i++;
          (*c)--;
        }
        for (i = 1; i <= cTmp; i++)
```

```
    fwrite(&Saida[i], sizeof(unsigned char), 1, ArqComprimido);
}


int Codifica(TipoVetoresBO VetoresBaseOffset, int Ordem,
             int *c, int MaxCompCod)
{
  *c = 1;
  while (Ordem >= VetoresBaseOffset[*c + 1].Offset && *c + 1 <=
      MaxCompCod)
    (*c)++;
  return (Ordem - VetoresBaseOffset[*c].Offset + VetoresBaseOffset[*c].
      Base);
}


void TerceiraEtapa(FILE *ArqTxt, TipoAlfabeto Alfabeto, int *Indice,
  TipoPalavra Palavra, char *Linha, TipoDicionario Vocabulario,
      TipoPesos p,
  TipoVetoresBO VetoresBaseOffset, FILE *ArqComprimido, int MaxCompCod)
{
  Apontador Pos;
  TipoChave Chave;
  int Codigo, c;
  do {
    ExtraiProximaPalavra(Palavra, Indice, Linha, ArqTxt, Alfabeto);
    memcpy(Chave, Palavra, sizeof(TipoChave));
    if (*Palavra != '\0') {
      Pos = Pesquisa(Chave, p, Vocabulario);
      Codigo = Codifica(VetoresBaseOffset, Vocabulario[Pos].Ordem, &c,
                        MaxCompCod);
      Escreve(ArqComprimido, &Codigo, &c);
      do {
        ExtraiProximaPalavra(Palavra, Indice, Linha, ArqTxt, Alfabeto);
        /* O primeiro espaco depois da palavra nao e codificado */
        if (strcmp(Trim(Palavra), "") && (*Trim(Palavra)) != (char)0) {
          memcpy(Chave, Palavra, sizeof(TipoChave));
          Pos = Pesquisa(Chave, p, Vocabulario);
          Codigo = Codifica(VetoresBaseOffset, Vocabulario[Pos].Ordem,
              &c,
                            MaxCompCod);
          Escreve(ArqComprimido, &Codigo, &c);
        }
      } while (strcmp(Palavra, ""));
    }
  } while (*Palavra != '\0');
}


void Compressao(FILE *ArqTxt, FILE *ArqAlf, FILE *ArqComprimido)
{
  TipoAlfabeto Alfabeto;
  TipoPalavra Palavra, Linha;
  int Ind = 1;
  int MaxCompCod;
  TipoDicionario Vocabulario;
  TipoPesos p;
```

```
    TipoVetoresBO VetoresBaseOffset;

    Vocabulario = malloc(M*sizeof(TipoItem));

    /* Inicializacao do Alfabeto */
    DefineAlfabeto(Alfabeto, ArqAlf);    /* Le o alfabeto definido em
        arquivo */
    *Linha = '\0';
    /*Inicializacao do Vocabulario */
    Inicializa(Vocabulario);
    GeraPesos(p);
    /* Inicio da Compressao */
    PrimeiraEtapa(ArqTxt, Alfabeto, &Ind, Palavra, Linha, Vocabulario, p)
        ;
    MaxCompCod = SegundaEtapa(Vocabulario, VetoresBaseOffset, p,
                              ArqComprimido);
    fseek(ArqTxt, 0, SEEK_SET); /* Coloca o cursor de leitura no inicio
        do arquivo*/
    Ind = 1;
    *Linha = '\0';
    TerceiraEtapa(ArqTxt, Alfabeto, &Ind, Palavra, Linha, Vocabulario, p,
                  VetoresBaseOffset, ArqComprimido, MaxCompCod);
}


/*Procedimentos da Descompressao*/
int LeVetores(FILE *ArqComprimido,
              TipoBaseOffset *VetoresBaseOffset)
{
  int MaxCompCod, i;
  MaxCompCod = LeNumInt(ArqComprimido);
  for (i = 1; i <= MaxCompCod; i++) {
    VetoresBaseOffset[i].Base = LeNumInt(ArqComprimido);
    VetoresBaseOffset[i].Offset = LeNumInt(ArqComprimido);
  }
  return MaxCompCod;
}


int LeVocabulario(FILE *ArqComprimido, TipoVetorPalavra Vocabulario)
{
  int NumNodosFolhas, i;
  TipoPalavra Palavra;
  char Ch;
  NumNodosFolhas = LeNumInt(ArqComprimido);
  for (i = 1; i <= NumNodosFolhas; i++) {
    *Palavra = '\0';
    do {
      fread(&Ch, sizeof(unsigned char), 1, ArqComprimido);
      if (Ch != (char)0)    /* As palavras estao separadas pelo caratere
          0 */
        sprintf(Palavra + strlen(Palavra), "%c", Ch);
    } while (Ch != (char)0);
    strcpy(Vocabulario[i], Palavra);
  }
  return NumNodosFolhas;
}
```

```
555
int Decodifica(TipoVetoresBO VetoresBaseOffset,
               FILE *ArqComprimido, int MaxCompCod)
{
  int c = 1;
560  int Codigo = 0, CodigoTmp = 0;
  int LogBase2 = (int)round(log(BaseNum) / log(2.0));
  fread(&Codigo, sizeof(unsigned char), 1, ArqComprimido);
  if (LogBase2 == 7) Codigo -= 128;    /* remove o bit de marcacao */
  while ((c + 1 <= MaxCompCod) &&
565         ((Codigo << LogBase2) >= VetoresBaseOffset[c+1].Base)) {
    fread(&CodigoTmp, sizeof(unsigned char), 1, ArqComprimido);
    Codigo = (Codigo << LogBase2) | CodigoTmp;
    c++;
  }
570  return (Codigo - VetoresBaseOffset[c].Base + VetoresBaseOffset[c].
      Offset);
}


void Descompressao(FILE *ArqComprimido, FILE *ArqTxt, FILE *ArqAlf)
575 {
  TipoAlfabeto Alfabeto;
  int Ind, MaxCompCod;
  TipoVetorPalavra Vocabulario;
  TipoVetoresBO VetoresBaseOffset;
580  TipoPalavra PalavraAnt;

 Vocabulario = malloc(M*sizeof(TipoPalavra));

  DefineAlfabeto(Alfabeto, ArqAlf);    /* Le o alfabeto definido em
      arquivo */
585  MaxCompCod = LeVetores(ArqComprimido, VetoresBaseOffset);
  LeVocabulario(ArqComprimido, Vocabulario);
  Ind = Decodifica(VetoresBaseOffset, ArqComprimido, MaxCompCod);
  fputs(Vocabulario[Ind], ArqTxt);
  while (!feof(ArqComprimido)) {
590    Ind = Decodifica(VetoresBaseOffset, ArqComprimido, MaxCompCod);
    if (Ind > 0) {
      if (Alfabeto[Vocabulario[Ind][0] + 127] && PalavraAnt[0] != '\n')
        putc(' ', ArqTxt);
      strcpy(PalavraAnt, Vocabulario[Ind]);
595      fputs(Vocabulario[Ind], ArqTxt);
    }
  }
}

600
/*Procedimentos para fazer busca*/

#define MaxTamTexto      100000000
#define MaxTamPadrao     1000
605 #define MaxChar          256

typedef char *TipoTexto;
typedef char TipoPadrao[MaxTamPadrao + 1];
```

```
610
    void BMH(TipoTexto T, int n, TipoPadrao P, int m)
    {
      int i, j, k;
      int d[MaxChar + 1];
615   int cont = 0;
      int comp = 0;

      //if (!( P == NULL || strlen(P) == 0 || P == "" ))
     // {
620


      /*-- Pre-processamento do padrao --*/
      for (j = 0; j <= MaxChar; j++) d[j] = m;
      for (j = 1; j <= m - 1; j++) d[P[j] + 128] = m - j;
625   i = m;
      while (i <= n) {/*-- Pesquisa --*/
        k = i;
        j = m;
        while (T[k] == P[j] && j > 0) {
630       k--;
          j--;
          comp = comp + 1;
        }
        comp = comp + 1;
635     if (j == 0)
        {
          printf("Casamento na posicao: %3d\n", k + 1);
          cont++;
        }
640     i += d[T[i] + 128];
      }
      //}
      printf("Comparacoes %d\n", comp);
      printf("Ocorrencias %d\n", cont);
645 }



    void Atribui(TipoPadrao P, int Codigo, int c)
650 {
      int i = 1/*, cTmp = c*/;
      P[i] = (char)((Codigo >> ((c - 1) * 7)) | 0x80);
      i++;
      c--;
655   while (c > 0) {
        P[i] = (char)((Codigo >> ((c - 1) * 7)) & 127);

        i++;
        c--;
660   }
    }

    void imprimePadrao(TipoPadrao Padrao)
    {
665         int i=1;
            for (i = 1 ; i <= strlen(Padrao); i++)
```

```
        {
                printf("%c", Padrao[i]);
        }
}

void Busca(FILE *ArqComprimido, FILE *ArqAlf, char *p_Padrao)
{
  TipoAlfabeto Alfabeto;
  int Ind, Codigo, MaxCompCod;
  TipoVetorPalavra Vocabulario;
  TipoVetoresBO VetoresBaseOffset;
  //TipoPalavra p;
  int c, Ord, NumNodosFolhas;
  TipoTexto T;
  TipoPadrao Padrao;
  TipoPadrao v_PadraoBuscar;
  long n = 1;
  T = malloc(MaxTamTexto*sizeof(char));
  char delims[] = " ";
  char *result;
  int i = 0;
  int j = 1;

  strcpy(v_PadraoBuscar, "");

  Vocabulario = malloc((M+1)*sizeof(TipoPalavra));

  DefineAlfabeto(Alfabeto, ArqAlf);
  MaxCompCod = LeVetores(ArqComprimido, VetoresBaseOffset);

  NumNodosFolhas = LeVocabulario(ArqComprimido, Vocabulario);
  while (fread(&T[n], sizeof(char), 1, ArqComprimido)) n++;


  result = strtok( p_Padrao, delims );


  while( result != NULL )
  {

        if (result != NULL)
        {
                // fred
                Ord = -1;
                printf("Result %s\n", result);
                for (Ind = 1; Ind <= NumNodosFolhas; Ind++)
                     if (!strcmp(Vocabulario[Ind], result)) Ord = Ind;

                Codigo = Codifica(VetoresBaseOffset, Ord, &c,
                   MaxCompCod);
                Atribui(Padrao, Codigo, c);

                if (Ord == -1)
                {
                        printf("%s nao esta no Vocabulario:\
                           nComparacoes 0\n0correncias 0\n", result);
                        return;
```

```
                }

                for (i = 1; i<=c; i++)
                {
                        v_PadraoBuscar[j] = Padrao[i];
                        j++;
                }

         }//end do IF

        result = strtok( NULL, delims );

    }

    BMH(T, n, v_PadraoBuscar,j-1);

}

void trataEntrada(int p_ArgumentosSistema)
{
        if (p_ArgumentosSistema < 4)
        {
                printf("Erro na chamada do programa. Consulte
                   documentacao.\n");
                exit(0);
        }
}


int main(int argc, char *argv[])
{
        FILE *ArqTxt = NULL, *ArqAlf = NULL;
        FILE *ArqComprimido = NULL;
        //TipoPalavra NomeArqTxt, NomeArqAlf, NomeArqComp, Opcao;
        char NomeArqTxt[100], NomeArqAlf[100], NomeArqComp[100], Opcao
            [10];

        strcpy(Opcao, argv[1]);

        strcpy(NomeArqAlf, "alfabeto2.txt");
        ArqAlf = fopen(NomeArqAlf, "r");
        trataEntrada(argc);

        if (Opcao[0] == 'c')
        {
                        // Compressao
                        printf("Compressão de <%s> em <%s>.\n", argv
                            [2], argv[3]);
                        strcpy(NomeArqTxt, argv[2]);
                        strcpy(NomeArqComp, argv[3]);
                        ArqTxt = fopen(NomeArqTxt, "r");
                        ArqComprimido = fopen(NomeArqComp, "w+b");
                        Compressao(ArqTxt, ArqAlf, ArqComprimido);
                        fclose(ArqTxt);
                        ArqTxt = NULL;
                        fclose(ArqComprimido);
                        ArqComprimido = NULL;
```

```
        }
        else if (Opcao[0] == 'd')
        {

                        printf("Descompressão de <%s> em <%s>.\n", argv
                            [2], argv[3]);
                        strcpy(NomeArqComp, argv[2]);
                        strcpy(NomeArqTxt, argv[3]);
                        ArqTxt = fopen(NomeArqTxt, "w");
                        ArqComprimido = fopen(NomeArqComp, "r+b");
                        Descompressao(ArqComprimido, ArqTxt, ArqAlf);
                        fclose(ArqTxt);
                        ArqTxt = NULL;
                        fclose(ArqComprimido);
                        ArqComprimido = NULL;
        }
        else if (Opcao[0] == 'p')
        {

              ArqComprimido = fopen(argv[2], "r+b");
              if (ArqComprimido == NULL)
              {
                      printf("Arquivo comprimido nao encontrado.\n");
                      return 0;;
              }

              Busca(ArqComprimido, ArqAlf, argv[3]);
              //fclose(ArqComprimido);
              ArqComprimido = NULL;
        }
   return 0;
}
/* End. */
```

Referências

[1] Gonzalo Navarro. A Guided Tour to Approximate String Matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[2] CPP Reference. Cpp Reference Web Site. In *Web site: http://www.cppreference.com*, 2006.

[3] WikiPedia. Huffman coding. In *WikiPedia, A Enciclopédia Livre*, 2006.

[4] Nívio Ziviani. *Projeto de Algoritmos Com Implementações em Pascal e C*. Pioneira Thomson Learning, 2004.