Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

PROJETO E ANÁLISE DE ALGORITMOS

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

Código Fonte: disponível em:
http://www.dcc.ufmg.br/~lcrocha/paa/tp3/src

Leonardo Chaves Dutra da Rocha

Professor - Nivio Ziviani

Belo Horizonte
4 de junho de 2006

Resumo

Neste trabalho apresentamos uma avaliação de alguns dos principais algoritmos de recuperação de padrões em arquivos constituídos de documentos comprimidos e não comprimidos. Os algoritmos utilizados nessa avaliação foram um algoritmo de Programação Dinâmica, o algoritmo de Shift-And para casamento aproximado e casamento exato, o algoritmo de casamento exato de Boyer-Moore-Horspool e uma versão melhorada desse último, o Boyer-Moore-Horspool-Sunday. Na primeira parte do trabalho apresentamos um descrição de funcionamento de cada um desses algoritmos além da análise de complexidade dos mesmos. Na segunda parte do trabalho apresentamos as avaliação desses algoritmos divididos em duas etapas. Na primeira etapa apresentamos as avaliações dos algoritmos de casamento aproximado e na segunda parte os de casamento exato em arquivos comprimidos e não comprimidos. As avaliações foram feitas com base no número de comparações de cada que um desses algoritmos realiza, além do tempo de execução do mesmos.

Sumário

1 Introdução

Uma cadeia corresponde a uma seqüência de elementos denominados caracteres. Os caracteres são escolhidos de um conjunto denominado alfabeto. Por exemplo, em uma cadeia de bits o alfabeto é {0,1}. A pesquisa em cadeia de caracteres é um componente importante em diversos problemas computacionais, tais como edição de texto, recuperação de informação e estudo de seqüências de DNA em biologia computacional. O problema de casamento de padrão (do inglês *pattern matching*) consiste em encontrar todas as ocorrências de um dado padrão $P = p_1p_2p_3...p_m$ em um texto $T = t_1t_2t_3...t_n$, onde tando P como T são seqüências de caracteres de um alfabeto. O texto é um arranjo T[1..n] de tamanho n e o padrão é um arranjo P[1..m] de tamanho $m \leq n$. Os elementos P e T são escolhidos de um alfabeto finito $\sum$ de tamanho c. Por exemplo, pode-se ter $\sum$ = {0,1} ou $\sum$ = a,b,...,z. Dadas duas cadeias P (padrão) de comprimento |P| = m e T (texto) de comprimento |T| = n, em que $n \gg m$, deseja-se saber as ocorrências de P em T. No caso de programas editores de texto, o usuário pode estar interessado em buscar todas as ocorrências de um padrão (uma palavra particular) no texto que está sendo editado.

Existem diversos algoritmos para solucionar esse problema, destacando-se como os mais importantes os algoritmos *Knuth-Morris-Pratt* e o *Boyer-Moore*, desenvolvidos em 1977. Esses algoritmos foram desenvolvidos em 1977 e a partir dessa data vários estudos e novas propostas surgirão e estão surgindo.

Existem basicamente duas classes de algoritmos de casamento de padrão: os de casamento exato, apresentado na seção 2 e os de casamento aproximado, apresentados na seção 3. O objetivo deste trabalho é realizar experimentos em um conjunto de programas para recuperar ocorrências de padrões em arquivos de documentos, utilizando algoritmos lineares de busca seqüencial. Esses experimentos são divididos de duas maneiras:

- **Busca Aproximada em Arquivos Não Comprimidos:** Nessa parte do trabalho deverão ser utilizados os seguintes algoritmos:
  - Algoritmo Programação Dinâmica
  - Algoritmo Shift-And

  Deverão ser realizados experimentos que compare o desempenho dos dois algoritmos para tamanhos de erros que variam entre 0 e 3. Os resultados desses experimentos são apresentados nas seções 6.6.3 e 6.6.4.

- **Busca Exata em Arquivos Comprimidos:** Nessa parte do trabalho deverão ser utilizados os seguintes algoritmos:
  - Boyer-Moore-Horspool: Para arquivos comprimidos e não comprimidos

  Deverão ser realizados experimentos que compare o desempenho do BMH para arquivos comprimidos e não comprimidos para casamento exato. Os resultados desses experimentos são apresentados nas seções 6.6.1 e 6.6.2.

2 Casamento Exato

O casamento exato de um padrão em um texto se dá quando todos os caracteres do padrão são encontrados no texto exatamente na ordem em que os mesmos ocorrem no padrão. Vamos tomar como exemplo o seguinte texto: *Existe um grande clube na cidade, que mora dentro do meu coração.* Uma busca exata pela palavra *grande* poderia ser feita e um casamento ocorreria a partir da posição 11 do texto original. No entanto, se no texto a palavra *grande* estivesse sido escrita errôneamente, não teríamos casamento algum.

Nas seções a seguir, apresentamos os três algoritmos para casamento exato de padrão que foram utilizados para a realização desse trabalho: Boyer-Moore-Horspool na seção 2.1, Boyer-Moore-Horspool Sunday na seção 2.1, e Shift-And Exato na seção 2.3

2.1 Boyer-Moore-Horspool

Esse é um algoritmo clássico de casamento de padrões, apresentado originalmente por Boyer e Moore em 1977 [4]. A idéia principal dos algoritmos da classe Boyer-Moore é pesquisar o padrão no sentido da direita para a esquerda, o que torna o algoritmo muito rápido, como veremos a seguir. Horspool [8] apresentou uma simplificação importante e tão eficiente quanto o algoritmo original, que tornou-se conhecida como Boyer-Moore-Horspool. Considerando sua extrema simplicidade de implementação, bem como sua comprovada eficiência, o algoritmo BMH deve ser o escolhido em aplicações de uso geral que necessitam realizar casamento exato de cadeias.

O enfoque dos algoritmos da classe BM e BMH consiste em pesquisar o padrão P em uma janela que desliza ao longo do texto T. Para cada posição desta janela, o algoritmo faz uma pesquisa por um sufixo da janela que casa com um sufixo de P por meio de comparações realizadas no sentido da direita para a esquerda. Se não ocorrer uma desigualdade, então uma ocorrência de P em T foi encontrada, caso contrário o algoritmo calcula um deslocamento em que o padrão deve ser deslizado para a direita antes que uma nova tentativa de casamento se inicie. O algoritmo BM original propõe duas heurísticas para calcular o deslocamento:

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

O funcionamento da heurística de ocorrência pode ser visualizado abaixo:

```
1 2 3 4 5 6 7 8 9 0 1 2
c a c b a c
a a b c a c c a c b a c
  c a c b a c
      c a c b a c
        c a c b a c
            c a c b a c (Casamento!!)
```

O funcionamento da heurística de casamento pode ser visualizado abaixo:

```
1 2 3 4 5 6 7 8 9 0 1 2
c a c b a c
a a b c a c c a c b a c
      c a c b a c
            c a c b a c (Casamento!!)
```

O algoritmo BM decide qual das duas heurísticas deve usar escolhendo a que provoca o maior deslocamento do padrão. Entretanto, esta escolha implica realizar uma comparação entre dois inteiros para cada caractere lido do texto, penalizando o desempenho do algoritmo com relação ao tempo de processamento. Assim, como os melhores resultados são os que consideram apenas a heurística ocorrência, a heurística de casamento já não é mais utilizada na versão Boyer-Moore-Horspool.

No algoritmo Boyer-Moore-Horspool(BMH) está a mais importante simplificação. Horspool observou que qualquer caractere já lido do texto a partir do último deslocamento pode ser usado para endereçar a tabela de deslocamentos. Baseado neste fato, Horspool propôs endereçar a tabela com o caractere no texto correspondente ao último caractere do padrão.

Para pré-computar o padrão, o valor inicial de todas as entradas na tabela de deslocamentos é feito igual a m. A seguir, apenas os m-1 primeiros caracteres do padrão são usados para obter os outros valores da tabela. Formalmente:

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

Para o padrão P = teste, os valores de d são $d[t] = min(1, 4)$, $d[e] = mim(3)$, $d[s] = min(2)$, ou seja, d[t] = 1, d[e] = 3, d[s] = 2. Os demais valores são iguais ao valor de |P|, nesse caso m = 5. Isso é feito uma vez que os padrões possuem caracteres repetidos, no exemplo, o t ocorre na posição 1 e 4 e o caractere e ocorre nas posições 2 e 5.

Nos experimentos realizados nesse trabalho utilizamos a implementação feita em C que está disponibilizada em [23] e a listagem completa do mesmo se encontra no Apêndice C.

2.1.1 Análise de complexidade

A seguir, apresentamos o apenas o algoritmo de Boyer-Moore-Horspool para a análise de complexidade do mesmo.

```
void BMH(char *T, int *n, char *P, int *m, int linha){
  long i, j, k;
  long d[Maxchar + 1];
  for (j = 0; j <= Maxchar; j++)
    d[j] = *m;
  for (j = 1; j < *m; j++)
    d[P[j-1]] = *m - j;
  i = *m;
  while (i <= *n){    /*-- Pesquisa --*/
     k = i;
     j = *m;
     while (T[k-1] == P[j-1] && j > 0) {
       k--; j--;
       comparacoes ++;
     }
     if (j == 0)
     printf("linha(%d)\n",linha);
     i += d[T[i-1]];
  }
}
```

O pré-processamento do padrão ocorre nas duas primeiras linhas do código acima. Nesse pré-processamento é calculado a tabela de deslocamentos *d*. A pesquisa é realizada através de dois loops aninhados. O loop mais externo varia entre m(tamanho do padrão) e n(tamanho do texto) com incrementos de $d[ord(T[i+1])]$ equivalente ao endereço na tabela de deslocamento do caractere que está na i+1-ésima posição no texto, a qual corresponde à posição do último caractere de P.

O deslocamento da ocorrência pode ser pré-computado com base apenas no padrão e no alfabeto, e a complexidade de tempo e de espaço para essa fase é $O(c)$, onde c é o tamanho do alfabeto. Para a fase de pesquisa, o pior caso do algoritmo é $O(nm)$, caso os deslocamentos ocorram apenas de 1 em um. No entanto o melhor caso é $O(n/m)$, onde os deslocamentos são maiores, do tamanho do padrão. O caso esperado também $O(n/m)$, se c não é pequeno e m não é muito grande.

2.1.2 Instrumentação do Código

O algoritmo de Boyer-Moore-Horspool apresentado acima foi instrumentado para que o número de comparações pudesse ser computado. Essa instrumentação pode ser observada na linha de código *comparacoes ++;*. A variável *comparacoes* é uma variável global do programa, inicializada com 0. A cada comparação de caracteres entre o padrão e o texto, a mesma é incrementada. Ao final da consulta essa variável é impressa em um arquivo de saída.

2.2 Boyer-Moore-Horspool-Sunday

O algoritmo de Boyer-Moore-Horspool-Sunday (BMHS) foi proposto por Sunday em 1990 [18]. Trata-se de uma variante do BMH que trás mais uma simplificação. Sunday propôs endereçar a tabela com o caractere no texto correspondente ao próximo caractere após o último caractere do padrão, em vez de deslocar o padrão usando o último caractere como no algoritmo BMH. Para pré-computar o padrão, o valor inicial de todas as entradas na tabela de deslocamentos é feito igual a $m + 1$. A seguir, os m primeiros caracteres do padrão são usados para obter os outros valores da tabela. Formalmente:

$$d[x] = min\{j\ tal\ que\ j = m | (1 \leq j \leq m\ \&\ P[m + 1 - j] = x)\}$$

Assim como no BMH, vamos exemplificar a construção da tabela de deslocamento para o padrão *teste*. Neste caso temos que os valores de d são $d[t] = (5, 2)$, $d[e] = min(4, 1)$, $d[s] = min(3)$, ou seja, $d[t] = 2$, $d[e] = 1$, $d[s] = 3$ e todos os outros valores são iguais ao valor de $|\mathrm{P}| + 1$.

Nos experimentos realizados nesse trabalho utilizamos a implementação feita em C que está disponibilizada em [23] e a listagem completa do mesmo se encontra no Apêndice D.

2.2.1 Análise de complexidade

A seguir, apresentamos o apenas o algoritmo de Boyer-Moore-Horspool-Sunday para a análise de complexidade do mesmo.

```
void BMHS(char *T, int *n, char *P, int *m, int linha){
  long i, j, k;
  long d[Maxchar + 1];
  for (j = 0; j <= Maxchar; j++)
    d[j] = *m + 1;
  for (j = 1; j <= *m; j++)
    d[P[j-1]] = *m - j + 1;
  i = *m;
  while (i <= *n){ /*-- Pesquisa --*/
      k = i;
      j = *m;
      while (T[k-1] == P[j-1] && j > 0){
        k--; j--;
        comparacoes ++;
      }
      if (j == 0)
      printf("linha(%d)\n",linha);
      i += d[T[i]];
    }
}
```

O pré-processamento do padrão para obter a tabela de deslocamento d ocorre nas duas primeiras linhas do código. A fase de pesquisa é constituída por um anel

em que i varia de m até n, com incrementos $d[ord(T[i+1])]$, o que equivale ao endereço na tabela d do caractere que está na i+1-ésima posição no texto, a qual corresponde à posição do último caractere de P.

Observando o código, temos que o comportamento assintótico do algoritmo BMHS é idêntico ao algoritmo BMH. Portanto a complexidade de tempo e de espaço para a fase de pré-processamento é $O(c)$. Para a fase de pesquisa, o pior caso continua $O(nm)$ como no BMH, o melhor caso é $O(n/m)$ e o caso esperado é $O(n/m)$, se c não é pequeno e m não é muito grande. A diferença no entanto é que os deslocamentos são mais longos (podendo ser iguais a $m+1$), levando a saltos relativamente maiores para padrões curtos. Por exemplo, para um padrão de tamanho $m = 1$, o deslocamento é igual a $2m$ quando não há casamento.

2.2.2 Instrumentação do Código

O algoritmo de Boyer-Moore-Horspool-Sunday apresentado acima foi instrumentado para que o número de comparações pudesse ser computado. Essa instrumentação pode ser observada na linha de código *comparacoes ++;*. A variável *comparacoes* é uma variável global do programa, inicializada com 0. A cada comparação de caracteres entre o padrão e o texto, a mesma é incrementada. Ao final da consulta essa variável é impressa em um arquivo de saída.

2.3 Shift-And exato

O algoritmo Shift-And foi proposto por Baeza-Yates e Gonnet em 1989 [2]. Este algoritmo usa o conceito de paralelismo de bit, uma técnica que tira proveito do paralelismo intrínseco das operações sobre bits dentro de uma palavra de computador. Neste caso, é possível empacotar muitos valores em uma única palavra e atualizar todos eles em uma única operação. Pelo fato de tirar proveito do paralelismo de bit, o número de operações que um algoritmo realiza pode ser reduzido por um fator de até w, onde w é o número de bits da palavra do computador. Considerando que nas arquiteturas atuais w é 32 ou 64, o ganho na prática pode ser muito grande.

O algoritmo Shift-And utiliza um autômato para representar o padrão a ser procurado. Esse autômato representa uma transição de estados, que ocorre quando um determinado caractere do padrão é encontrado no texto. Um autômato a que representa um padrão p de tamanho m possui $m+1$ estados. A transição de um estado a_i para a_{i+1} ocorre somente se o caractere p_i do padrão for igual ao caractere atual do texto. Quando o estado a_{m+1} é alcançado indica que uma ocorrência do padrão foi encontrada no texto.

O algoritmo mantém um conjunto de todos os prefixos de P que casam com o texto já lido e utiliza o paralelismo de bit para atualizar o conjunto de cada caractere lido no texto. Este conjunto é representado por uma máscara de bits $R = (b_1,\ b_2, ..., b_m)$. O algoritmo Shif-And pode ser visto como um autômato não determinista que pesquisa um padrão no texto. Vejamos um exemplo desse tipo de autômato na Figura 1

O algoritmo Shift-and funciona da seguinte forma: O valor 1 é colocado na j-ésima posição de $R = (b_1,\ b_2, ..., b_m)$ se e somente se $p_1...p_j$ é um sufixo de $t_1...t_i$, em que i corresponde à posição corrente no texto. A j-ésima posição de R é dita estar ativa. Um casamento é relatado sempre que b_m fica ativo. Na leitura do

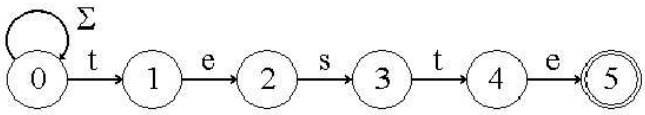


Figura 1: Autômato Não-Determinista - Reconhece Prefixos de P=teste

próximo caractere t_{i+1}, o novo valor do conjunto R é calculado, o qual chamaremos de R'. A posição $j+1$ no conjunto R' ficará ativa se e somente se a posição j estava ativa em R. Assim, em uma linguagem de programação onde as operações *and*, *or*, deslocamento à direita e complemento são realizadas com eficiência, com o uso do paralelismo de bit é possível computar o novo conjunto com custo $O(1)$. Além disso, se o tamanho do padrão m for menor do que a palavra w do computador, então o conjunto R pode ser implementado em um arranjo de bits que cabe em um registrador do computador.

O primeiro passo do algoritmo é a construção de uma tabela M para armazenar uma máscara de bits $b_1, ..., b_m$ para cada caractere. A Tabela 1 apresenta as máscaras de bits para os caracteres presentes em $P = teste$. A máscara em $M[t]$ é 10010, pois o caractere t aparece nas posições 1 e 4 de P.

Tabela 1: Máscara relativa a P = teste

	1	2	3	4	5
M[t]	1	0	0	1	0
M[e]	0	1	0	0	1
M[s]	0	0	1	0	0

Na fase de pesquisa de P em T, o valor do conjunto é inicializado como $R = 0^m$. Para cada novo caractere t_{i+1} lido do texto, o valor do conjunto R' é atualizado de acordo com a seguinte fórmula:

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

A operação $\gg$ desloca todas as posições para a direita no passo $i+1$ para marcar quais posições de P eram sufixos no passo i. A cadeia vazia ϵ também é marcada como um sufixo por meio da operação *or* entre o conjunto obtido após a operação $\gg$ e 10^{m-1}. Essa operação permite que um casamento possa iniciar na posição corrente do texto. Do conjunto obtido até o momento são mantidas apenas as posições em que t_{i+1} casa com p_{j+1} o que alcançado por meio da operação *and* desse conjunto de posições com o conjunto $M[t_{i+1}]$ de posições de t_{i+1} em P. A Tabela 2 mostra o funcionamento do algoritmo Shift-And para pesquisar o padrão P = teste no texto T = os testes testam estes alunos.

O algoritmo parece complexo, no entanto pode ser facilmente implementado utilizando a linguagem C. Temos que a execução do algoritmo simula a execução de um autômato não-determinístico como o da Figura 1. O autômato e seus estados podem ser representados como uma seqüência de bits e, somente com operações *and*

Tabela 2: Exemplo de funcionamento do algoritmo Shift-And

Texto	$(R \gg 1) \vert 10^{m-1}$	R'
o	1 0 0 0 0	0 0 0 0 0
s	1 0 0 0 0	0 0 0 0 0
	1 0 0 0 0	0 0 0 0 0
t	1 0 0 0 0	1 0 0 0 0
e	1 1 0 0 0	0 1 0 0 0
s	1 0 1 0 0	0 0 1 0 0
t	1 0 0 1 0	1 0 0 1 0
e	1 1 0 0 1	0 1 0 0 1
s	1 0 1 0 0	0 0 0 0 0
	1 0 0 0 0	0 0 0 0 0

($\wedge$ lógico aplicado bit a bit), é possível realizar o caminhamento neste autômato. O algoritmo mantém um registrador, representado por uma seqüência de m bits, onde m é o tamanho do padrão, que indica o estado atual do autômato. Cada bit r_i deste registrador define se o estado a_{i+1} do autômato está ativo (1) ou inativo (0).

No autômato, o caminhamento é realizado utilizando as máscaras pré-definidas para cada caractere c do alfabeto. Através das máscaras é possível saber que estados do autômato serão ativados quando o caractere c for encontrado no texto. À medida que os caracteres do texto vão sendo alcançados, as operações *and* (bit a bit) são realizadas entre o registrador e a máscara do caractere atual do texto. Se um bit 1 alcança o último bit do registrador, então uma ocorrência do padrão foi encontrada no texto.

A construção da máscara M_c dos caracteres do alfabeto é feita da seguinte forma: As máscaras de todos os caracteres do alfabeto são iniciadas como uma seqüência de 0 e a partir daí, para cada ocorrência de um caractere c na posição p_i do padrão o i-ésimo bit de sua máscara M_c deve receber o valor 1.

Assim, resumidamente temos que o algoritmo que simula o autômato realiza as seguintes operações:

1. Lê um caractere c do texto;
2. Realiza um "shift" para a direita em r preenchendo o bit mais a esquerda com 1;
3. Realiza uma operação *and* bit a bit entre r e M_c;
4. Se a o último bit do registrador foi alcançado, uma ocorrência do padrão foi encontrada. Caso contrário, o algoritmo segue lendo os outros caracteres do texto e realizando estas operações até que o texto se esgote.

Nos experimentos realizados nesse trabalho utilizamos a implementação feita em C que está disponibilizada em [23] e a listagem completa do mesmo se encontra no Apêndice E.

2.3.1 Análise de complexidade

A seguir, apresentamos o apenas o algoritmo de Shift-And Exato para a análise de complexidade do mesmo.

```
void ShiftAndExato(char *T, int *n, char *P, int *m, int linha) {
  long Masc[Maxchar];
  long i;
  long R = 0;
  for (i = 0; i < Maxchar; i++)
    Masc[i] = 0;
  for (i = 1; i <= *m; i++) {
    Masc[P[i-1] + 127] |= 1 << (*m - i);
  }
  for (i = 0; i < *n; i++) {
      comparacoes ++;
      R = ((((unsigned long)R) >> 1) |
          (1 << (*m - 1))) & Masc[T[i] + 127];
      if ((R & 1) != 0)
      printf("linha(%d)\n",linha);
  }
}
```

O pré-processamento do padrão para obter a tabela de máscara M ocorre na etapa do código. A fase de pesquisa é constituída por um anel em que i varia de 1 até n, com incrementos unitários, no qual cada caractere é analisado e o registrador é alterado para a análise do próximo caractere. Dessa forma, temos que o custo do algoritmo Shift-And é $O(n)$ uma vez que as operações podem ser realizadas em $O(1)$ e levando-se em consideração que o padrão cabe em umas poucas palavras do computador.

2.3.2 Instrumentação do Código

O algoritmo de Shift-And Exato apresentado acima foi instrumentado para que o número de comparações pudesse ser computado. Essa instrumentação pode ser observada na linha de código *comparacoes ++;*. A variável *comparacoes* é uma variável global do programa, inicializada com 0. A cada atualização do registrador, ou seja, a cada caractere lido do texto, a mesma é incrementada. Ao final da consulta essa variável é impressa em um arquivo de saída.

3 Casamento Aproximado

O casamento aproximado de um padrão em um texto se dá quando todos os caracteres do padrão são encontrados no texto exatamente na ordem em que os mesmos ocorrem no padrão, salvo alguns erros que podem ser permitidos de acordo com a escolha. Vamos tomar como exemplo o seguinte texto: *Existe um grande clube na cidade, que mora dentro do meu coração.* Uma busca exata pela palavra *grandes* poderia ser feita e um casamento ocorreria a partir da posição 11 do texto original, permitindo um erro relacionado com o caractere s "a menos" no texto.

O número de operações de inserção, de substituição e remoção de caracteres necessário para transformar uma cadeia M em outra cadeia N é conhecido na literatura como distância de edição [12]. Assim, a distância de edição entre duas cadeias M e N, $ed(M, N)$ é o menor número de operações necessárias para transformar M em N. Por exemplo, $ed(\texttt{Matranda}, \texttt{Saturadas}) = 4$. Quando o número de erros permitidos é 0, temos o problema de casamento exato de padrões.

Na seções a seguir, apresentamos os dois algoritmos para casamento aproximado de padrão que foram utilizados para a realização desse trabalho: Programação Dinâmica na seção 3.1 e Shift-And aproximado na seção 3.2

3.1 Programação Dinâmica

Primeiramente apresentamos um algoritmo que utiliza programação dinâmica para resolver o problema de distância entre strings [15]. Esse problema consiste do seguinte: Sejam dadas duas cadeias de caracteres, X = $x_1x_2x_3x_4...x_m$ e Y=$y_1y_2y_3y_4...y_n$. O número de operações de retirada, inserção e substituição necessárias para transformar X em Y é conhecido como distância de edição, em inglês *edit distance*, ou distância de Levenshtein que advém do cientista russo Vladimir Levenshtein, que considerou esta distância já em 1965. Assim a distância de edição $ed(X, Y)$ corresponde ao número K de operações necessárias para converter X em Y [1, 12].

Vamos tomar como exemplo, se $X = matranda$ e $Y = saturadas$, então $ed(X, Y) = 4$. Nesse caso, a seqüência de operações que devem ser feitas são: (i) substitui x_1 por y_1(s), (ii) insere y_4(u) após x_3, (iii) retira x_6(n) e (iv)insere y_9(s) após x_8.

A técnica de programação dinâmica consiste em resolver incrementalmente problemas menores até se atingir uma solução ótima global para as cadeias que vão sendo separadas. A Figura 2 mostra a matriz de programação dinâmica, estrutura sobre a qual o algoritmo trabalha.

A solução aqui descrita é baseada na apresentada em [11, 12], e Primeiramente cria-se uma matriz de tamanho (M+1)x(N+1), onde M é o tamanho da cadeia de caracteres 1 e N é o tamanho da cadeia de caracteres 2, conforme Figura **??**. A matriz é percorrida pelo algoritmo do topo à esquerda, em direção à base à direita. Cada posição (i, j) da matriz contém a distância de edição entre a cadeia M_0, i e a cadeia N_0, j. O cálculo da distância pode ser formulado da seguinte maneira:

$$
\begin{aligned}
&M_{i,0} \leftarrow i, M_{0,j} \leftarrow j \\
&M_{i,j} \leftarrow \begin{cases} M_{i-1,j-1} & \text{se } x_i = y_j, \\ 1 + min(M_{i-1,j-1}, M_{i-1,j}, M_{i,j-1}) & \text{caso contrário} \end{cases}
\end{aligned} \tag{1}
$$

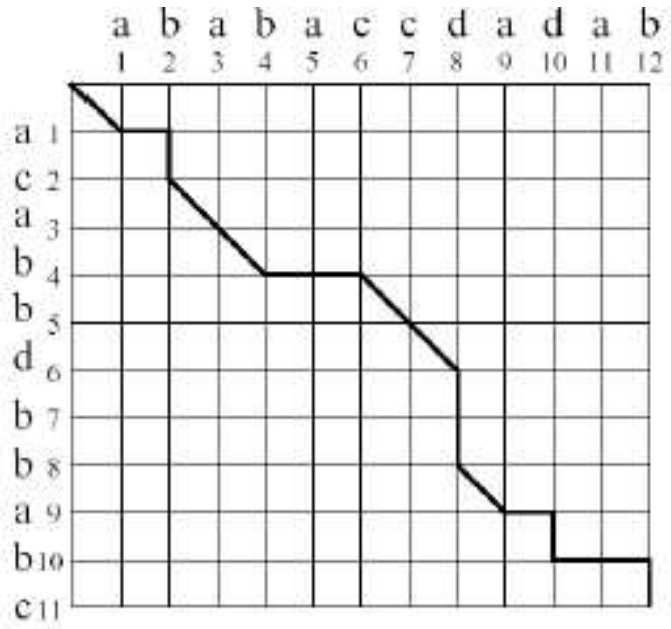


Figura 2: Matriz de programação dinâmica

Transformando essa formulação em um algoritmo, temos a seguinte operações que devem ser executadas nessa ordem:

1. Compara os caracteres M_i e N_j;
2. Se os dois forem iguais, então custo é igual a 0;
3. Se os dois forem diferentes, então custo é igual a 1;
4. Verifica-se então qual das seguintes células vizinhas da célula em questão é menor: $((M_(i-1), N_j) + 1,\ (M_j, N_(j-1)) + 1,\ (M_(i-1), N_(j-1)) + custo)$
5. Assim o custo da célula em questão é o menor deles.

Na Figura 2 podemos ver que o caminho demarcado representa o conjunto mínimo de operações que transforma a cadeia que rotula as linhas na cadeia que rotula as colunas. O caminho de menor custo entre o topo à esquerda e a base à direita da matriz representa o conjunto de operações que transformam M em N. Uma linha vertical representa a inserção de um caractere em N e uma linha horizontal representa a inserção a remoção de um caractere em N. As substituições ocorrem nas linhas diagonais que percorrem caracteres diferentes. No Apêndice A disponibilizamos uma breve prova da corretude do uso desse algoritmo para o problema de distância de edição.

Procurar um padrão P em um texto T é similar a computar a distância entre strings com $x = p$ e $y = T$. A única diferença é que deve-se permitir que uma ocorrência comece em qualquer posição do texto. Isto pode ser feito de maneira bem simples através de $M_{0,j} = 0$ *para todos* $j \in 0...n$ [14, 17]. Essa alteração permite que o padrão ocorra em qualquer posição do texto. Feita essa alteração, basta executar o algoritmo original de distância de edição. Finalizada a execução,

na última linha da matriz temos os valores de distância de edição para cada posição no texto em relação ao padrão procurado, assim basta realizar um procura nessa linha pelas posições no texto onde a distância de edição foi menor que o tamanho máximo de erros permitidos.

Vamos tomar como exemplo a procura do padrão *survey* no texto *surgery*. Primeiramente $M_{0,j} = 0$ *para todos* $j \in 0...n$ e em seguida, executamos o algoritmo. O resultado essa execução pode ser vista na Figura 3.

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	**2**	**2**	**2**

Figura 3: Exemplo de Funcionamento

Analisando a última linha da matriz, temos por exemplo que o texto na posição 5 casa com o padrão com um erro de dois caracteres, ou ainda, que o texto na posição 4 casa com o padrão com um erro de dois caracteres. Ou seja, basta percorrer a última linha a procura de distâncias inferiores ao número de erros permitidos, inclusive erro 0.

Esse proposta de algoritmo possui uma complexidade de tempo de $O(MxN)$, onde M é o tamanho do padrão e N o tamanho do texto. Da mesma forma, temos que a complexidade de espaço também é $O(MxN)$. A solução implementada nesse trabalho utiliza uma abordagem mais econômica em relação a complexidade de espaço.

Nossa implementação se baseia em uma solução apresentada por [14] em que se pode trabalhar somente com $O(m)$ em espaço. A principal observação é que para calcular $M_{*,j}$ precisa-se somente dos valores de $M_{*,j-1}$. Logo, em vez de construir toda a matriz M, processa-se os caracteres de T um a um e mantém-se uma única coluna de M, que é atualizada depois da leitura de cada nova posição do texto j para manter invariante $C_i = M_{i,j}$. Assim, nosso algoritmo inicializa sua coluna $C_{0...m}$ com os valores $C_i \leftarrow i$ e processa o texto caractere por caractere. A cada novo caractere do texto t_j, seu vetor coluna é atualizado para $C'_{0..m}$. A fórmula de atualização da coluna é dada por:

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

e as posições do texto onde $C_m \leq k$ são ditas como posições finais de ocorrências. Observe que se $C = M_{*,j-1}$ é a coluna antiga e $C' = M_{*,j}$ é a coluna nova, então C_{i-1} corresponde a $M_{i-1,j-1}$, C'_{i-1} a $M_{i-1,j}$ e C_i a $M_{i,j-1}$ na Equação 1.

A implementação utilizada nos experimentos foi feita em C e a listagem completa do mesmo se encontra no Apêndice F.

3.1.1 Análise de Complexidade

A seguir, apresentamos o apenas o a parte de avaliação de casamento de string da solução utilizando programação dinâmica para a análise de complexidade do mesmo.

```
void dinamico(char *s, char *t,int tam_str1,int tam_str2,
              int *matriz,int erros,int linha){
    int i, posicao_texto;
    int posicao_acima_nova;
    int posicao_local_nova;
    /* Inicializa a matriz*/
    for(i=0;i<=tam_str1;i++){
        matriz[i] = i;
    }
    /*Inicia-se então a soluçao do problema comparando cada caracter
    de uma string com cada caracter da outra string, no entanto
    aproveitando as soluçoes anteriores*/
    // Para cada caracter do texto
    for(posicao_texto=0; posicao_texto<tam_str2; posicao_texto++){
        posicao_acima_nova = 0;
        for(i=1;i<=tam_str1;i++){ //Atualiza o Vetor de Casamento
            comparacoes ++;
            if (s[i]==t[posicao_texto]){
                posicao_local_nova = matriz[i-1];
            }else{
                posicao_local_nova = 1 + minimo(matriz[i-1],
                                                matriz[i],
                                                posicao_acima_nova);
            }
            matriz[i-1] = posicao_acima_nova;
            posicao_acima_nova = posicao_local_nova;
        }
        matriz[tam_str1] = posicao_acima_nova;
        // verifica casamento
        if (matriz[tam_str1] <=erros){
            printf("linha(%d)\n",linha);
        }
    }
}
```

Observando o código acima, temos dois loops aninhados, um que caminha pelo texto e outra que caminha pelo padrão, assim temos a complexidade de tempo do algoritmo em $O(MN)$. Além disso, podemos observar também que apenas uma coluna da matriz é utilizada, além de duas variáveis auxiliares, ou seja, a complexidade de espaço é de apenas $O(M)$.

3.1.2 Instrumentação do Código

O algoritmo de Programação Dinâmica apresentado acima foi instrumentado para que o número de comparações pudesse ser computado. Essa instrumentação pode ser observada na linha de código *comparacoes ++;*. A variável *comparacoes* é uma variável global do programa, inicializada com 0. A cada comparação de caracteres entre o padrão e o texto, a mesma é incrementada. Ao final da consulta essa variável é impressa em um arquivo de saída.

3.2 Shift-And Aproximado

Nesta seção apresentamos o algoritmo Shift-And para casamento aproximado. Esse algoritmo, que foi proposto por Wu e Manber (1992), é uma extensão do algoritmo Shift And [22] apresentado na seção 2.3. O algoritmo Shift-And para casamento aproximado de cadeias também simula um autômato não determinista para guiar o processo de busca. A diferença entre ele e o exato está na possibilidade de se obter casamentos com erros. Assim como no algoritmo de programação dinâmica, esse algoritmo aceita os seguintes tipos de erros durante o processamento:

- Inserção: possibilita que o padrão ocorra com caracteres a mais em qualquer posição da ocorrência.
- Remoção: permite que o padrão ocorra com caracteres a menos em qualquer posição da ocorrência.
- Substituição: possibilita que o padrão ocorra com caracteres diferentes em qualquer posição da ocorrência.

Isso faz com que novos estados e transições sejam adicionados e novos registradores devam ser criados, um para cada quantidade de erros. Na Figura 4 apresentamos um exemplo de autômato para casamento aproximado com até 2 erros de inserção, substituição e remoção.

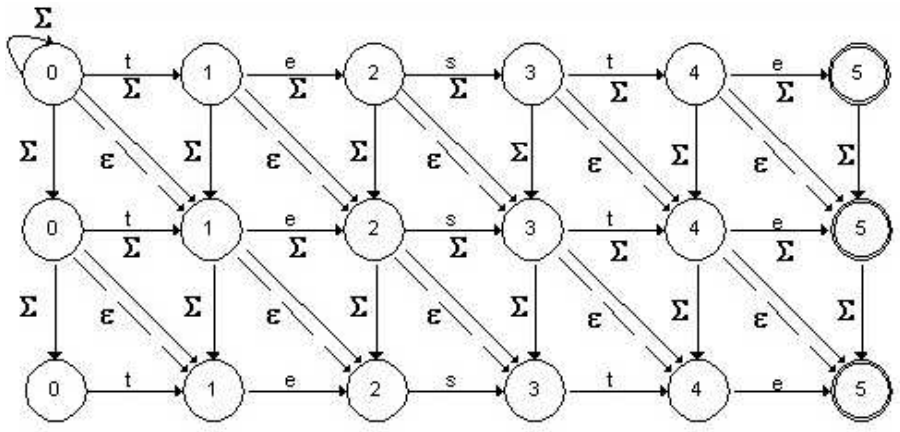


Figura 4: Autômato para Casamento Aproximado com até 2 Erros de Inserção, Substituição ou Retirada

O algoritmo Shift-And empacota cada linha j $(0 < j \leq k)$ do autômato não-determinista em uma palavra R_j diferente do computador. A cada novo caractere lido do texto, todas as transições do autômato são simuladas 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 do texto. Na posição i do texto, os novos valores $R'_j, 0 < j \leq k$, são obtidos a partir dos valores correntes R_j:

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

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

A pesquisa inicia em $R_j = 1^j 0_{m-j}$. R_0 equivale ao algoritmo Shift-And para casamento exato, e as outras linhas R_j recebem 1s também de linhas anteriores. Observando a Figura 4 a fórmula de R' está expressa nas arestas horizontais, indicando o casamento de um caractere. As arestas verticais indicam inserção(R_{j-1}), as arestas diagonais cheias indicam substituição$((R_{j-1} \gg 1))$ e as arestas diagonais tracejadas indicam retirada$((R'_{j-1} \gg 1))$.

A Tabela 3 apresenta o funcionamento do algoritmo Shift-And-Aproximado para pesquisar o padrão $P = teste$ no texto $T = ostestestestam$, permitindo um erro de inserção, um de retirada e um de substituição. Existe uma ocorrência exata na leitura do oitavo caractere e cinco ocorrências, permitindo um erro nas posições 7, 9, 12, 14 e 15.

Tabela 3: Exemplo de funcionamento do algoritmo Shift-And-Aproximado

Texto	$(R_0 \gg 1)\|10^{m-1}$	R'_0	$(R_1 \gg 1)\|10^{m-1}$	R'_1
o	1 0 0 0 0	0 0 0 0 0	1 1 0 0 0	0 0 0 0 0
s	1 0 0 0 0	0 0 0 0 0	1 0 0 0 0	0 0 0 0 0
	1 0 0 0 0	0 0 0 0 0	1 0 0 0 0	0 0 0 0 0
t	1 0 0 0 0	1 0 0 0 0	1 0 0 0 0	1 1 0 0 0
e	1 1 0 0 0	0 1 0 0 0	1 1 1 0 0	1 1 1 0 0
s	1 0 1 0 0	0 0 1 0 0	1 1 1 1 0	0 1 1 1 0
t	1 0 0 1 0	0 0 0 1 0	1 0 1 1 1	1 1 1 1 **1**
e	1 1 0 0 1	0 0 0 0 **1**	1 1 1 1 1	1 1 1 1 **1**
s	1 0 1 0 0	0 0 1 0 0	1 1 1 1 1	0 1 1 1 **1**
	1 0 0 0 0	0 0 0 0 0	1 0 1 1 1	0 0 1 1 0
t	1 0 0 0 0	1 0 0 0 0	1 0 0 1 1	1 1 0 1 0
e	1 1 0 0 0	0 1 0 0 0	1 1 1 0 1	1 1 1 0 **1**
s	1 0 1 0 0	0 0 1 0 0	1 1 1 1 0	0 1 1 1 0
t	1 0 0 1 0	0 0 0 1 0	1 0 1 1 1	1 1 1 1 **1**
a	1 1 0 0 1	0 0 0 0 0	1 1 1 1 1	1 1 0 1 **1**
m	1 0 0 0 0	0 0 0 0 0	1 1 1 0 1	0 0 0 0 0

Nos experimentos realizados nesse trabalho utilizamos a implementação feita em C que está disponibilizada em [23] e a listagem completa do mesmo se encontra no Apêndice G.

3.2.1 Análise de complexidade

A seguir, apresentamos o apenas o algoritmo de Shift-And-Aproximado para a análise de complexidade do mesmo.

```
void ShiftAndAproximado(char *T, int *n, char *P, int *m,
                        int k,int linha) {
  long Masc[Maxchar];
  long i, j, Ri, Rant, Rnovo;
  long R[NumMaxErros + 1];
  for (i = 0; i < Maxchar; i++)
    Masc[i] = 0;
  for (i = 1; i <= *m; i++)
    { 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];
  for (i = 0; i < *n; i++)
    { comparacoes += k+1;
      Rant = R[0];
      Rnovo = ((((unsigned long)Rant) >> 1) | Ri) & Masc[T[i] + 127];
      R[0] = Rnovo;
      for (j = 1; j <= k; j++)
        { Rnovo = ((((unsigned long)R[j]) >> 1) & Masc[T[i] + 127])
                  | Rant | (((unsigned long)(Rant | Rnovo)) >> 1);
          Rant = R[j];
          R[j] = Rnovo | Ri;
        }
      if ((Rnovo & 1) != 0)
      printf("linha(%d)\n",linha);
    }
}
```

O custo da simulação do autômato é $O(\lceil m/w \rceil n$ no pior caso onde w é o tamanho em bits de uma palavra do computador em questão, e no caso médio, o que equivale a $O(kn)$ para padrões típicos na pesquisa em textos (isto é, $m \leq w$) [23]. Já a complexidade de espaço para simular os autômatos é $O(Km)$, onde k é o número de erros permitidos, uma vez que são necessários k autômatos com o tamanho do padrão.

3.2.2 Instrumentação do Código

O algoritmo de Shift-And Aproximado apresentado acima foi instrumentado para que o número de comparações pudesse ser computado. Essa instrumentação pode ser observada na linha de código *comparacoes += k+1;*. A variável *comparacoes* é uma variável global do programa, inicializada com 0. A cada atualização dos registradores, ou seja, a cada caracter lido do texto, a mesma é incrementada 1(sem

erro) mais o número de erros permitidos. Ao final da consulta essa variável é impressa em um arquivo de saída.

4 Compressão

Neste trabalho apresentamos uma análise comparativa entre algoritmos de casamento de padrões em textos comprimidos e não comprimidos. Assim, antes de apresentarmos os resultados propriamente ditos, realizaremos nessa seção uma breve discussão a respeito de compressão, sua importância, além de apresentamos o algoritmo utilizado na compressão dos textos de teste.

Com o crescimento da Web, o que temos visto é uma explosão de informação textual disponíveis *on-line* como bibliotecas digitais, sistemas de automação de escritórios, banco de dados de documentos, etc. Somente na Web, temos hoje em torno de bilhões de paginas estáticas disponíveis, onde cada bilhão ocupa aproximadamente 10 terabytes de texto corrido. Assim, nesse cenário, temos que técnicas de compressão são extremamente eficientes, e com estudos recentes relacionados a busca em textos comprimidos, temos que tais técnicas se tornam ainda mais atrativas no contexto de recuperação de informação como máquinas de busca por exemplo.

Compressão de texto consiste em maneiras de representar o texto original em menos espaço. Na verdade, o que se usa é uma substituição de símbolos por outros que possam ser representados por um número menor de bytes ou bits. O texto comprimido ocupa menos espaço de disco e conseqüentemente leva-se menos tempo para ser lido do disco(tempos de leitura de disco são grandes pontos de conteção), ou para ser transmitido pela rede. Temos que ao longo de 20 anos o tempo de acesso a discos magnéticos tem se mantido praticamente constante enquanto que o a velocidade de processamento aumentou aproximadamente 2 mil vezes. Dessa forma temos que o tempo de leitura de disco de arquivos muitos grandes é maior que o *overhead* de compressão e descompressão.

Existem diversos algoritmos de compressão de textos, no entanto alguns aspectos devem ser considerados antes que os mesmos sejam escolhidos. São eles:

- Velocidade de compressão e descompressão
- Possibilidade de realizar casamento de padrões diretamente no texto comprimido.
- Permitir o acesso a qualquer parte do texto comprimido e iniciar uma descompressão a partir desse ponto.

Um dos métodos de codificação mais conhecidos e utilizados e o de Huffman [9] [19] [20] codificação de Huffman na wikipedia. A idéia desse método é atribuir códigos mais curtos aos símbolos com freqüência alta. Um código único de tamanho variável a cada símbolo diferente do texto. Existem dois tipos de implementações para o Huffman: as mais tradicionais que consideram como símbolos os caracteres e uma mais recente que consideram as palavras como símbolos.

Em nosso caso utilizamos a implementação baseada em palavras um vez que essa implementação permite o acesso randômico a palavras dentro do texto e a tabela de símbolos do codificador é exatamente o vocabulário do texto. Além disso temos

que esse método de Huffman baseados em palavras permite acessar diretamente qualquer parte do texto comprimido sem a necessidade de descomprimir. Um texto em linguagem natural é constituído de palavras e de separadores. Separadores são caracteres que aparecem entre palavras: espaço, vírgula, ponto, ponto e vírgula, interrogação, e assim por diante. Uma forma eficiente de lidar com palavras e separadores é representar o espaço simples de forma implícita no texto comprimido. Nesse modelo, se uma palavra é seguida de um espaço, então, somente a palavra é codificada. Senão, a palavra e o separador são codificados separadamente. No momento da decodificação, supõe-se que um espaço simples segue cada palavra, a não ser que o próximo símbolo corresponda a um separador.

O algoritmo de Huffman primeiramente faz uma passada sobre o arquivo pra que a freqüência das palavras seja contabilizada. Em seguida é feita a codificação de cada uma das palavras extraídas do texto de acordo com a freqüência das mesmas, onde as mais freqüentes são codificadas com tamanho menor e as menos freqüentes recebem um código maior. Em seguida, a compressão é realizada propriamente dita, onde as palavras são substituídas pelos códigos gerados para as mesmas.

O algoritmo de Huffman é uma abordagem guloso que constrói uma árvore de codificação partindo-se de baixo para cima. No início há um conjunto de n folhas representado as palavras do vocabulário e suas respectivas freqüências. A cada iteração, as duas árvores com menores freqüências são combinadas em uma única árvore e a soma de suas freqüências é associada a nó raiz. Ao final das *n-1* iterações, obtém-se a árvore de codificação, na qual os códigos associados a cada palavra são representados pela seqüência dos rótulos das arestas que levam da raiz à folha que a representa. Na Figura 5 apresentamos um exemplo de funcionamento do algoritmo Huffman. *P1, P2, P3, P4, P5, P6, P7, P8 e P9* são palavras com freqüência 10 15 25 25 25 25 35 40 50 respectivamente.

O método de Huffman produz a árvore de codificação que minimiza o comprimento do arquivo comprimido. Existem diversas árvores que produzem a mesma compressão. Entretanto, a escolha preferencial para a maioria das aplicações é a árvore canônica [16]. Uma árvore de Huffman é canônica quando a altura da sub-árvore à direita de qualquer nó nunca é menor que a altura da sub-árvore à esquerda. Um prova mostrando que a árvore de Huffman é ótima foi retirada de [13], traduzida por Sérgio Haruo Nakanishi e apresentada no Apêndice B.

Este tipo de representação de código facilita a forma de visualização, sugerindo métodos de decodificação e codificação bastante simples:

- **Codificação**: a árvore é percorrida emitindo bits ao longo de suas arestas.
- **Decodificação**: os bits de entrada são usados para selecionar as arestas.

No entanto, essa abordagem de representação é ineficiente pois ocupa muito espaço e demanda muito tempo de execução. Em [10, 3, 23] são apresentadas formas mais elegantes e mais eficientes do algoritmo de Huffman. Nesses casos, os comprimentos dos códigos são calculados no lugar dos códigos propriamente ditos, onde a compressão atingida é a mesma, independentemente dos códigos utilizados. Após o cálculo dos comprimentos, há uma forma elegante de codificação e decodificação.

Uma proposta ainda mais eficiente é apresenta por Moura, Navarro, Ziviani e Baeza-Yates em [6], Neste trabalho encontramos uma codificação de Huffman

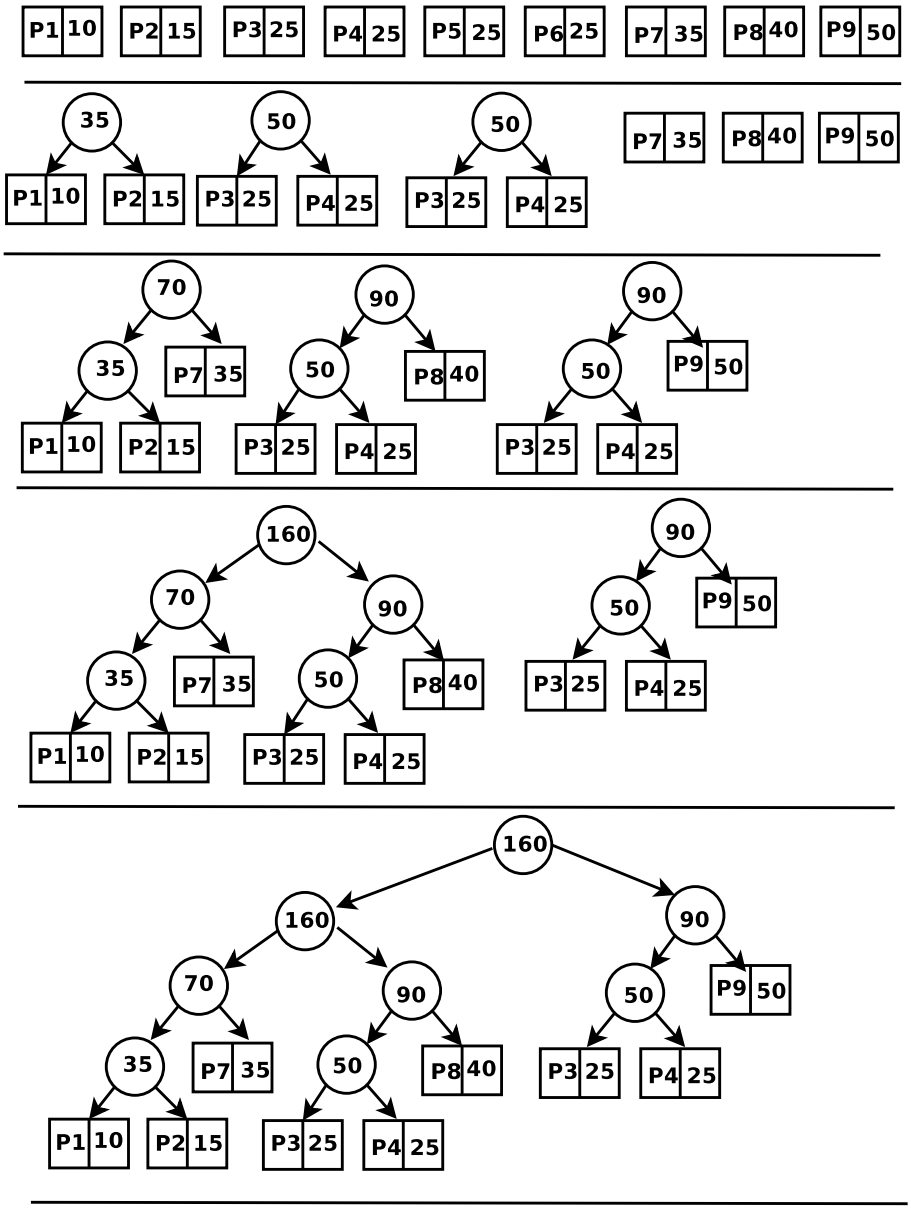


Figura 5: Exemplo de Funcionamento Huffman

orientada a bytes denominada *Huffman pleno*, no qual a atribuição de códigos foi modificada de tal forma que uma seqüência de bytes é associada a cada palavra do texto. Conseqüentemente o grau de cada nó passa de 2 para $256(2^8)$.

Os autores ainda apresentam uma possibilidade alternativa de codificação com marcação para que o início de cada palavra seja identificado chamado de *Huffman com Marcação*. Nesse caso utiliza-se apenas 7 dos 8 bits de cada byte para codificação e o oitavo bit de cada byte indica se é ou não o início de uma palavra, ou seja, o oitavo bit é marcado para identificar o primeiro byte da codificação da palavra. Essa implementação ajuda bastante em pesquisa em arquivos comprimidos [5, 24].

Essa foi a implementação utilizada para realizar a compressão de nossos arquivos de testes disponível em [23]. No próprio código existe um algoritmo de Boyer-Moore-Horspool implementado que realiza casamento de padrão no texto comprimido.

Nesse código disponibilizado foram encontrados alguns erros que foram corrigidos para o funcionamento correto do algoritmo. Esse erros são listados abaixo:

- Estouro da Pilha: Todos os vetores são "mallocados" estaticamente e são passados entres as funções por valor, ou seja, a cada passagem é feita uma cópia desses vetores. No entanto, como para arquivos grandes esses vetores precisam ser maiores(como o vocabulário) temos um estouro de memória. Assim, passamos todos os vetores grandes para apontadores, assim apenas endereços são passados e não temos estouro de memória.

- Palavra inexistente: Temos que, ao procuramos por uma palavra inexistente em um arquivo, o código procura na tabela de vocabulários e mesmo se não se encontra o programa continua a procura. Isso acarreta erros ao programa. A alteração que feita foi continuar a procura somente se encontrar a palavra no vocabulário.

- Alfabeto: O alfabeto disponível está incompleto, assim algumas letras(como as Maiúsculas) não são inseridas no vocabulário o que provoca não encontrar uma palavra quando na verdade a mesma está lá. Dessa forma, incluímos todos os caracteres ASCII que não são separadores no arquivo de alfabeto

- Identificador de Linha: Foi criada uma uma função que coloca em um vetor todas as posições onde ocorrem quebra de linha para que uma busca, ao encontrar o padrão de consulta, pudesse fazer uma busca nesse vetor a procura da linha onde ocorre o mesmo, comparando onde ocorreu o padrão com as posições onde ocorrem as quebras de linhas.

- Mini-Parser: Criamos uma função de parser simplificada para que fosse possível realizar consultas do tipo "casa amarela da vovo".

- Função TRIM: A função Trim do Huffman executa "malloc"sempre que é chamada, e esta área nunca é liberada.

Assim a implementação utilizada nos experimentos foi a disponibilizada em [23] feita em C com as modificações apresentadas acima. A listagem completa do mesmo se encontra no Apêndice H.

5 Outras Aplicações Avaliadas

Além dos programas implementados descritos nas seções anteriores, nesse trabalha utilizamos também outros dois aplicativos de casamento de padrão em arquivos bastante populares na comunidade linux: Grep e Agrep. Esses aplicativos foram considerados na análise comparativa entre os algoritmos implementados, tornando assim a comparação mais realista, uma vez que ambos aplicativos já são bastante utilizados e bastante rápidos. Dessa forma, segue abaixo uma breve descrição de ambos aplicativos.

5.1 Grep

Grep é um utilitário de linha de comando originalmente desenvolvido para sistemas operacionais baseados em Unix. O nome é originado de um comando do editor de texto Unix chamado *ed* que toma a forma de *g/re/p* e que significa: "procure globalmente por linhas que casem com a expressão regular *re* e imprima as mesmas". Isto descreve o comportamento padrão do comando Grep. O Grep toma uma expressão regular pela linha de comando, lê o texto da entrada padrão ou de uma lista de arquivos e imprime as linhas em que ocorreram o casamento com a expressão regular. Essa expressão regular pode ser simplesmente um padrão qualquer. Um exemplo de como utilizar tal aplicativo pode ser visto abaixo:

```
grep <padrao> <lista_arquivos_para_procura>
```

O Grep permite apenas casamento exato de padrões. Para o casamento aproximado, existe o Agrep descrito na próxima seção.

5.2 AGrep

O Agrep é um aplicativo desenvolvido no Departamento de Ciência da Computação da Universidade do Arizona em Junho de 1991 por Sun Wu and Udi Manber [21]. O Agrep é um utilitário de linha de comando originalmente desenvolvido para sistemas operacionais baseados em Unix e que faz procuras em arquivos por um padrão qualquer ou até mesmo expressões regulares. A diferença entre o Grep e o Agrep é que o Agrep permite erros no casamento como inserção de caracteres, substituição de caracteres ou remoção de caracteres, ou seja, casamento aproximado.

A versão utilizada nesse trabalho corresponde a versão 2.04, publicada em março de 1992. Esta versão inclui os seguintes algoritmos:

- **Boyer-Moore**: para o casamento exato de padrões simples, o AGrep utiliza uma variação simplificada do algoritmo de Boyer-Moore.
- **Mgrep**: Algoritmo para casamento de um conjunto de padrões e que possui um tempo médio sub-linear.
- **Amonkey**: Algoritmo para casamento de padrão muito parecido com Boyer-Moore.

- **Mmonkey**: Algoritmo para textos e padrões ASCII que combina o algoritmo uma técnica de partição com Mgrep. Possui a mesma complexidade do Amonkey, no entanto executa bem mais rápido que o mesmo.
- **Bitap**: Algoritmo mais geral do Agrep que suporta muitas extensões como casamento exato, expressões regulares aproximadas, casamento de múltiplos padrões, mistura de casamento exato e aproximado, custos não uniformes, etc.

```
agrep -<numero_erros> <padrao> <lista_arquivos_para_procura>
```

Vamos tomar por exemplo que *Massechusets* casa com *Massachusetts* com dois erros (um substituição e uma inserção). Assim, utilizando o Agrep para procurar *Massechusets* em um arquivo que contenha *Massachusetts*, se o número de erros permitidos for de pelo menos 2, haverá um casamento, caso contrário não.

6 Experimentos

Nesta seção detalhamos como foram feitos os experimentos. Assim primeiramente apresentamos na seção 6.1 como os cada um dos algoritmos deve ser executado. Posteriormente, na seção 6.2 apresentamos alguns testes realizados que comprovam o funcionamento dos mesmo. Na seção 6.3 apresentamos quais os arquivos utilizados nos experimentos, assim como as consultas realizadas. Mais adiante, na seção 6.5, apresentamos o ambiente de execução e como os experimentos foram realizados, na seção 6.4 apresentamos como foram feitas as compressões dos arquivos de teste e, finalmente, na seção 6.6 apresentamos os resultados experimentais obtidos.

6.1 Utilização dos Programas

Em http://www.dcc.ufmg.br/~lcrocha/paa/tp3/src podemos encontrar o código fonte de todas as implementações realizadas nesse trabalho, separados de acordo com o problema e a solução dos mesmos.

Todos os programas foram construídos de forma padronizada e todos possuem um Makefile. Para compilar os programas, basta executar o seguinte comando:

```
make
```

Para executar o programa de Boyer-Moore Horspool, basta entrar no diretório referente ao mesmo e, após executar o Makefile, basta executar a seguinte linha de comando:

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

O programa Boyer-Moore Horspool Sunday se encontra no mesmo arquivo do programa Boyer-Moore Horspool original como uma subrotina. Após executar o Makefile, basta executar a seguinte linha de comando:

```
./bmh.e -i <arquivo_entrada> -p <padrao> -s
```

Para executar o programa de Shit-And para casamento exato, basta entrar no diretório referente ao mesmo e, após executar o Makefile, basta executar a seguinte linha de comando:

```
./shift_and.e -i <arquivo_entrada> -p <padrao> -k 0
```

Para executar o programa de Shit-And para casamento aproximado, basta entrar no diretório referente ao mesmo e, após executar o Makefile, basta executar a seguinte linha de comando:

```
./shift_and.e -i <arquivo_entrada> -p <padrao> -k <numero_erros>
```

Para executar o programa de Programação Dinâmica para casamento aproximado, basta entrar no diretório referente ao mesmo e, após executar o Makefile, basta executar a seguinte linha de comando:

```
./compara_string.e -i <arquivo_entrada> -p <padrao> -k 0
```

Para comprimir arquivos utilizando o programa de compressão huffman, basta entrar no diretório referente ao mesmo e, após executar o Makefile, basta executar a seguinte linha de comando:

```
./huffman.e -i <arquivo_entrada> -o <arquivo_saida_comprimido> -c
```

O programa de Boyer-Moore Horspool que faz busca exata em arquivos comprimidos se encontra dentro do próprio código de Huffman como uma sub-rotina. Dessa forma temos que para realizar uma consulta em um arquivo comprimido, após executar o Makefile, basta executar a seguinte linha de comando:

```
./huffman.e -i <arquivo_entrada_comprimido> -p <padrao>
```

6.2 Validação das Implementações

Nesta seção apresentamos alguns testes realizados com os algoritmos implementados comprovando assim o funcionamento dos mesmos. Para esses testes foram utilizadas as consultas sugeridas pela especificação do trabalho, listadas na Tabela 4.

Tabela 4: Consultas para Testes

Padrão
DCC UFMG dollar
dia branco
Macunaima administration
Brazilian coffee
Canada Treasury
Michael Gregory
price index
Uberaba

As consultas *New York Stock Exchange* e *Manacapuru* não foram consideradas pois o resultados das mesmas eram muito grandes para serem colocados nessa documentação. No entanto, para essa consultas os algoritmos também funcionaram.

1. **DCC UFMG dollar**: Segue abaixo o resultados das execuções dos algoritmos para essa consulta no arquivo *wsj89*. No caso do algoritmo de BMH em arquivos comprimidos, como a compressão é feita por palavra e sua freqüência, temos que o resultado foi diferente pois nesse caso *dollar* não casa com *dollars*. Assim, para o caso de busca em arquivos comprimidos foram feitas das pesquisas, uma usando *dollar* e outra usando *dollars*.

```
./compara_string.e -i wsj89 -p "DCC UFMG dollar" -k 0
linha(436)
linha(444)
linha(445)
linha(4530)
linha(5277)
linha(16311)
linha(17919)
linha(25894)
linha(28030)
linha(29519)
linha(29852)
linha(30869)
linha(34128)
linha(36765)
linha(38656)
linha(41583)
linha(41645)
```

```
./shift_and.e -i wsj89 -p "DCC UFMG dollar" -k 0
linha(436)
linha(444)
linha(445)
linha(4530)
linha(5277)
linha(16311)
linha(17919)
linha(25894)
linha(28030)
linha(29519)
linha(29852)
linha(30869)
linha(34128)
linha(36765)
linha(38656)
linha(41583)
linha(41645)
```

```
./bmh.e -i wsj89 -p "DCC UFMG dollar" -k 0
linha(436)
linha(444)
linha(445)
linha(4530)
linha(5277)
linha(16311)
linha(17919)
```

```
linha(25894)
linha(28030)
linha(29519)
linha(29852)
linha(30869)
linha(34128)
linha(36765)
linha(38656)
linha(41583)
linha(41645)
```

```
./bmh.e -s -i wsj89 -p "DCC UFMG dollar" -k 0
linha(436)
linha(444)
linha(445)
linha(4530)
linha(5277)
linha(16311)
linha(17919)
linha(25894)
linha(28030)
linha(29519)
linha(29852)
linha(30869)
linha(34128)
linha(36765)
linha(38656)
linha(41583)
linha(41645)
```

```
./huffman.e -i wsj89.huff -p "DCC UFMG dollar"
linha(436)
linha(444)
linha(445)
```

```
./huffman.e -i wsj89.huff -p "DCC UFMG dollars"
linha(4530)
linha(5277)
linha(16311)
linha(25894)
linha(28030)
linha(29519)
linha(29852)
linha(30869)
linha(34128)
linha(36765)
linha(38656)
```

```
linha(41583)
linha(41645)
```

2. **dia branco**: Segue abaixo o resultados das execuções dos algoritmos para essa consulta no arquivo *wsj89*.

```
./compara_string.e -i wsj89 -p "dia branco" -k 0
linha(43267)
linha(43275)
```

```
./shift_and.e -i wsj89 -p "dia branco" -k 0
linha(43267)
linha(43275)
```

```
./bmh.e -i wsj89 -p "dia branco" -k 0
linha(43267)
linha(43275)
```

```
./bmh.e -s -i wsj89 -p "dia branco" -k 0
linha(43267)
linha(43275)
```

```
./huffman.e -i wsj89.huff -p "dia branco"
linha(43267)
linha(43275)
```

3. **Macunaima administration**: Segue abaixo o resultados das execuções dos algoritmos para essa consulta no arquivo *wsj89*.

```
./compara_string.e -i wsj89 -p "Macunaima administration" -k 0
linha(19334)
linha(23835)
linha(24034)
linha(27251)
```

```
./shift_and.e -i wsj89 -p "Macunaima administration" -k 0
linha(19334)
linha(23835)
linha(24034)
linha(27251)
```

```
./bmh.e -i wsj89 -p "Macunaima administration" -k 0
linha(19334)
linha(23835)
linha(24034)
linha(27251)
```

```
./bmh.e -s -i wsj89 -p "Macunaima administration" -k 0
linha(19334)
linha(23835)
linha(24034)
linha(27251)
```

```
./huffman.e -i wsj89.huff -p "Macunaima administration"
linha(19334)
linha(23835)
linha(24034)
linha(27251)
```

4. **Brazilian coffee**: Segue abaixo o resultados das execuções dos algoritmos para essa consulta no arquivo *wsj89*.

```
./compara_string.e -i wsj89 -p "Brazilian coffee" -k 0
linha(28852)
```

```
./shift_and.e -i wsj89 -p "Brazilian coffee" -k 0
linha(28852)
```

```
./bmh.e -i wsj89 -p "Brazilian coffee" -k 0
linha(28852)
```

```
./bmh.e -s -i wsj89 -p "Brazilian coffee" -k 0
linha(28852)
```

```
./huffman.e -i wsj89.huff -p "Brazilian coffee"
linha(28852)
```

5. **Canada Treasury**: Segue abaixo o resultados das execuções dos algoritmos para essa consulta no arquivo *wsj89*.

```
./compara_string.e -i wsj89 -p "Canada Treasury" -k 0
linha(424)
```

```
./shift_and.e -i wsj89 -p "Canada Treasury" -k 0
linha(424)
```

```
./bmh.e -i wsj89 -p "Canada Treasury" -k 0
linha(424)
```

```
./bmh.e -s -i wsj89 -p "Canada Treasury" -k 0
linha(424)
```

```
./huffman.e -i wsj89.huff -p "Canada Treasury"
linha(424)
```

6. **Michael Gregory**: Segue abaixo o resultados das execuções dos algoritmos para essa consulta no arquivo *wsj89*.

```
./compara_string.e -i wsj89 -p "Michael Gregory" -k 0
linha(439)
```

```
./shift_and.e -i wsj89 -p "Michael Gregory" -k 0
linha(439)
```

```
./bmh.e -i wsj89 -p "Michael Gregory" -k 0
linha(439)
```

```
./bmh.e -s -i wsj89 -p "Michael Gregory" -k 0
linha(439)
```

```
./huffman.e -i wsj89.huff -p "Michael Gregory"
linha(439)
```

7. **price index**: Segue abaixo o resultados das execuções dos algoritmos para essa consulta no arquivo *wsj89*.

```
./compara_string.e -i wsj89 -p "price index" -k 0
linha(447)
linha(33256)
```

```
./shift_and.e -i wsj89 -p "price index" -k 0
linha(447)
linha(33256)
```

```
./bmh.e -i wsj89 -p "price index" -k 0
linha(447)
linha(33256)
```

```
./bmh.e -s -i wsj89 -p "price index" -k 0
linha(447)
linha(33256)
```

```
./huffman.e -i wsj89.huff -p "price index"
linha(447)
linha(33256)
```

8. **Uberaba**: Segue abaixo o resultados das execuções dos algoritmos para essa consulta no arquivo *wsj89*.

```
./compara_string.e -i wsj89 -p "Uberaba" -k 0
linha(565)
linha(41736)
```

```
./shift_and.e -i wsj89 -p "Uberaba" -k 0
linha(565)
linha(41736)
```

```
./bmh.e -i wsj89 -p "Uberaba" -k 0
linha(565)
linha(41736)
```

```
./bmh.e -s -i wsj89 -p "Uberaba" -k 0
linha(565)
linha(41736)
```

```
./huffman.e -i wsj89.huff -p "Uberaba"
linha(565)
linha(41736)
```

6.3 Arquivos e Consultas de Testes

Na execução de nossos experimentos, utilizamos os arquivos de documentos da coleção TREC, disponível em */pkg/texto2/wsj* na rede do DCC, com as seguintes características:

Tabela 5: Arquivos da Coleção TREC

Coleção	Tamanho(Mb)
wsj88	105
wsj88_10	9.7
wsj88_20	20
wsj89	2.7

Para exemplificar, um registro presente nestas coleções tem o seguinte formato:

```
<DOC>
<DOCNO> WSJ890802-0125 </DOCNO>
<DD> = 890802 </DD>
<AN> 890802-0125. </AN>
<HL> Inside Track:
@  NCNB Director Sold Big Holding in July
@  Worth $7.4 Million More 4 Weeks Later
@  ----
@  By Alexandra Peers and John R. Dorfman
@  Staff Reporters of The Wall Street Journal </HL>
<DD> 08/02/89 </DD>
<SO> WALL STREET JOURNAL (J) </SO>
<CO> NCB BPCO TRN EGGS LABOR </CO>
<IN> STOCK MARKET, OFFERINGS (STK)
SECURITIES INDUSTRY (SCR) </IN>
<TEXT>
   Even insiders make mistakes.
Sometimes, big, fat, $7 million-dollar mistakes. ...
</TEXT>
</DOC>
```

Para a realização dos experimentos utilizamos as consultas de acordo com a Tabela 6 onde o tamanho dos padrões procurados variaram entre 5 e 50. Esses padrões foram retirados de forma aleatória dos arquivos que foram utilizados para a realização dos experimentos. Através dessas consultas e dado que o tamanho dos arquivos também é diferente, foi possível verificar como cada um dos algoritmos se comportam a medida que em que o tamanho do padrão aumenta e a medida que o tamanho dos arquivos aumentam.

Tabela 6: Consultas Realizadas

Padrão	Tamanho(caracteres)
cheer	5
Manacapuru	10
Canada Treasury	15
In a move that would	20
move that would represent	25
In a move that would represent	30
in this city may become the first in the	40
court will vote on the code after a comment period	50

6.4 Compressão dos Arquivos de Teste

Nessa seção apresentamos como foram feitas as compressões dos arquivos de teste que serão utilizados consultas em arquivos comprimidos. A compressão foi feita

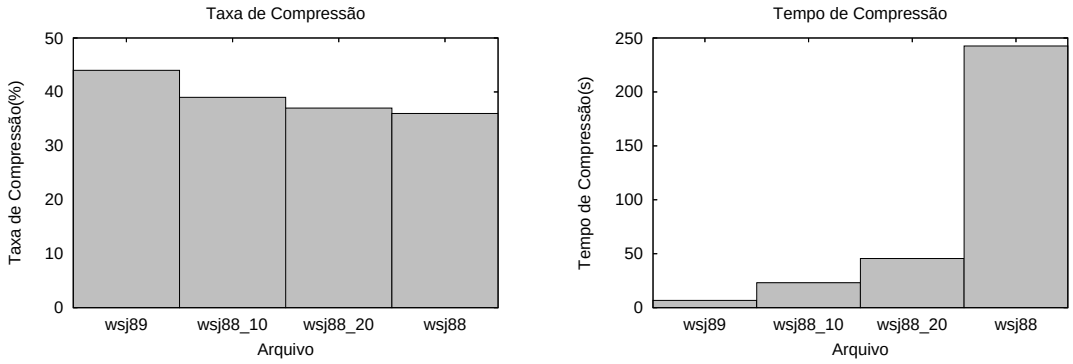
utilizando o *Huffman com Marcação*, conforme apresentado na seção 4.

Primeiramente avaliamos a taxa de compressão para cada um dos arquivos. O resultado dessa análise pode ser visualizada através da Tabela 7. Observando a

Arquivo	**Tam_Descomp(Mb)**	**Tam_Comp(Mb)**	**Taxa_Comp**
wsj89	2,7	1,2	44%
wsj88_10	9,7	3,8	39%
wsj88_20	20	7,4	37%
wsj88	105	38	36%

Tabela 7: Taxa de Compressão

Tabela 7 temos que, para a maioria dos arquivos a taxa de compressão foi na casa dos 30% com exceção do arquivo wsj89, cuja a taxa de compressão foi de 44% superando bastante a expectativa de 25% de compressão [23]. Na média, a taxa de compressão foi de 39%. O gráfico da Figura 6(a) também ilustra essa análise.



(a) Taxa de Compressão

(b) Tempo de Compressão

Figura 6: Análise da Compressão

Além da análise da taxa de compressão foi feita uma análise do tempo para compressão de cada um dos arquivos. Nesse caso, para cada arquivo foram feitas 5 compressões contabilizando o tempo para a realização de cada uma delas. Essa contabilização foi feita através do utilitário *time*, somando-se os tempos de usuário (*user time*) e de sistema (*system time*). Na Tabela 8 apresentamos média dos tempos medidos para compressão de cada um dos arquivos.

Arquivo	**Tam_Descomp(Mb)**	**Tempo Compressão(s)**
wsj89	2,7	6.742974
wsj88_10	9,7	23.132483
wsj88_20	20	45.580070
wsj88	105	242.574123

Tabela 8: Tempo de Compressão

Observando a Tabela 8 temos que existe uma razão linear entre o tempo de compressão e o tamanho dos arquivos para os arquivos avaliados. Para o maior arquivo temos que foi gasto um total aproximado de 242 segundos para a compactação. Apesar do tempo de compactação ser alto, para uma implementação bem feita de um algoritmo de casamento esse tempo é compensado uma vez que o tempo de procura nesses arquivos é bem melhor, além da economia de espaço que essa compressão proporciona, ainda mais quando falamos em terabytes de dados. O gráfico da Figura 6(b) também ilustra essa análise.

6.5 Ambiente de Execução e Detalhes do Experimentos

O ambiente de execução utilizado foi uma máquina com processador Intel Pentium4 3GHz, com uma cache de tamanho 1024 KB, 1GB de memória Ram e com um HD SATA.

Os tempos de execução foram medidos através do utilitário *time*, somando-se os tempos de usuário (*user time*) e de sistema (*system time*). Para garantir que o tempo apresentado represente a realidade, foi calculado o tempo médio de 5 execuções. Para os experimentos relacionados com casamento exato, para cada arquivo, cada uma das consultas era realizada 5 vezes para cada programa analisado. O tempo de execução, assim como o número de consultas realizadas, foi obtido através da média das 5 execuções. Para os algoritmos de casamento aproximado, para cada erro permitido e para cada arquivo, cada uma das consultas eram realizadas 5 vezes. Novamente, o tempo de execução, assim como o número de consultas realizadas, foi obtido através da média das 5 execuções.

Temos que, a forma como é feito cada um dos programas utilizados, o acesso a dispositivos de entrada/saída influencia bastante no desempenho de algoritmos de casamento de cadeias de caracteres. Por exemplo, na primeira execução, como os arquivos ainda não se encontram em cache, o tempo de execução pode ser maior, nas demais execuções, como o arquivo já se encontra em cache, o tempo de execução pode ser menor. Além disso, acesso a disco é bastante lento e isso reduz o desempenho dos algoritmos.

Um providência que poderia ser tomada para minimizar esse efeito dos dispositivos de entrada/saida no desempenho dos algoritmos é carregar todo o arquivo a ser avaliado na memória primária. Como a máquina utilizada possui 1GB de memória primária e o maior texto a ser pesquisado possui 109 Mb, isto não seria problema. No entanto, com essa providência, se realizássemos uma pesquisa por uma palavra que estivesse em uma quebra de linha, a mesma seria encontrada caso permitíssemos um tamanho de erro maior que 1. No entanto, de acordo com o funcionamento do *Agrep*, esse tipo de casamento não ocorre, pois o mesmo trabalha linha a linha. Assim, para manter a compatibilidade de interface e funcionamento com o programa *Agrep*, conforme solicitado na especificação do trabalho, optamos pela leitura do arquivo linha a linha, e as pesquisas também são feitas linha a linha. Isso na certa prejudica o desempenho dos algoritmos, no entanto o prejuízo é igual para todos, não interferindo assim na análise comparativa entre eles.

O único programa que faz o carregamento de todo o arquivo em memória é busca em arquivos comprimidos. Nesse caso, dentro do código de huffman encontra-se o código de busca(BMH), e o huffman disponibilizado já carregava todo o arquivo

em memória, assim optamos por não alterar isso, uma vez que outros problemas já haviam sido encontrados e essa alteração poderia acarretar outros problemas. No entanto, como o mesmo é utilizado apenas para casamento exato de padrões, no caso já mencionado no qual uma palavra é quebrada em duas linhas, como haverá um caractere de quebra linha, a mesma não será encontrada, mantendo assim a coerência com o programa *Agrep*.

Em http://www.dcc.ufmg.br/~lcrocha/paa/tp3/resultados podemos encontrar todos os resultados dos experimentos executados.

6.6 Resultados Experimentais

A seguir apresentamos os resultados empíricos dos experimentos realizados para avaliar o desempenho dos algoritmos, usando tempo de relógio e número de comparações realizadas. Realizamos experimentos para todos os algoritmos implementados, além do Grep e do Agrep.

Para uma melhor apresentação dos resultados, dividimos essa seção em quatro subseções. Na seção 6.6.2 apresentamos os resultados obtidos de tempo de execução para os experimentos envolvendo casamento aproximado. Na seção 6.6.1 apresentamos os resultados obtidos de número de comparações para os experimentos envolvendo casamento aproximado. Na seção 6.6.4 apresentamos os resultados obtidos de tempo de execução para os experimentos envolvendo casamento exato e, finalmente, na seção 6.6.3 apresentamos os resultados obtidos de número de comparações para os experimentos envolvendo casamento aproximado.

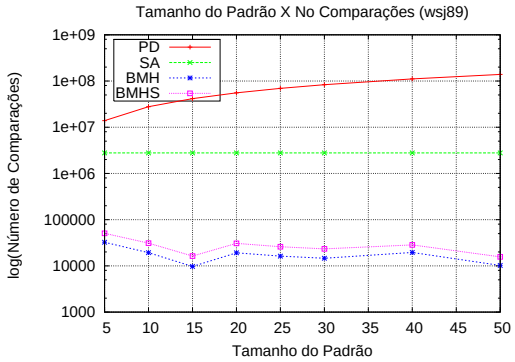
6.6.1 Casamento Aproximado - Número de Comparações

Foram gerados padrões para consultas variando o tamanho das mesmas entre 5 e 50 caracteres aplicando as mesmas para todos os algoritmos implementados que permitiam casamento aproximado com tamanhos de erro igual a 0 em arquivos não comprimidos, ou seja, Boyer-Moore-Horspool(BMH), Boyer-Moore-Horspool-Sunday(BMHS), Shift-And(SA) e Programação Dinâmica(PD), isso para todos os arquivos de teste. O resultado desses experimentos pode ser observado nos gráficos da Figura 7.

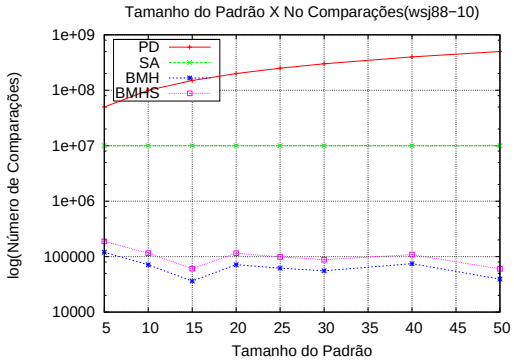
Para o algoritmo de Programação Dinâmica, para todos os arquivos avaliados podemos observar uma relação linear dentre o tamanho do padrão e o número de comparações, o que é coerente com a análise de complexidade realizada anteriormente, uma vez que cada caractere do padrão é comparado com cada caractere do texto. Já para o algoritmo Shit-And, o mesmo é indiferente ao tamanho do padrão, o que também é coerente com a análise apresentada anteriormente além de [23, 2] que fala que o Shift-And depende apenas do tamanho do texto.

Nesses mesmos gráficos, analisando os algoritmos de Boyer-Moore-Horspool e Boyer-Moore-Horspool-Sunday, temos que a medida que o tamanho do padrão aumenta, é possível observar que o número de comparações tendem a diminuir, o que também é coerente com a análise feita anteriormente além de [23, 4, 8] uma vez que os saltos de comparações tendem a serem maiores. Além disso, podemos observar que o número de comparações realizadas pelo Boyer-Moore-Horspool é um pouco menor que o número de comparações do Boyer-Moore-Horspool-Sunday uma vez que

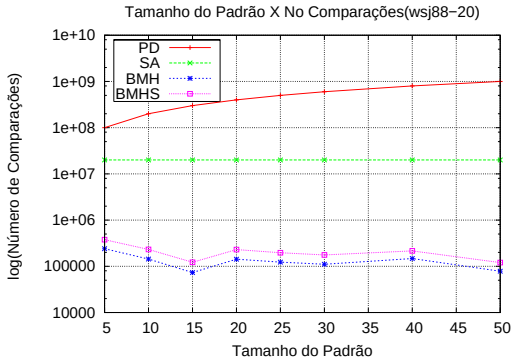
esse último é melhor para padrões curtos pois os saltos tendem a ser maiores($m+1$) e os padrões analisados tem tamanho mínimo 5.



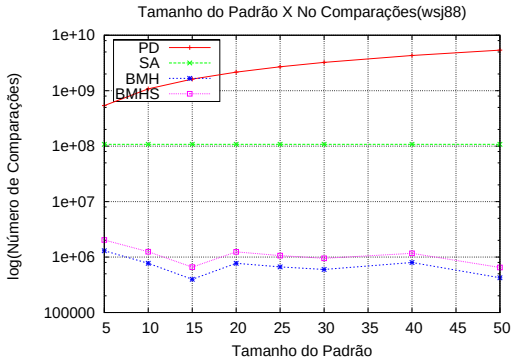
(a) Arquivo wsj89



(b) Arquivo wsj88_10



(c) Arquivo wsj88_20



(d) Arquivo wsj88

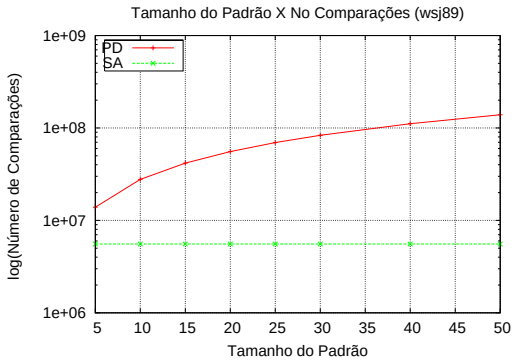
Figura 7: Número de Comparações em Relação ao Tamanho do Padrão - K = 0

Para todos os algoritmos, quanto maior o tamanho do arquivo a ser consultado maior também é o número de comparações, sendo mais uma vez coerente com a literatura pois todos esses algoritmos dependem do tamanho do texto.

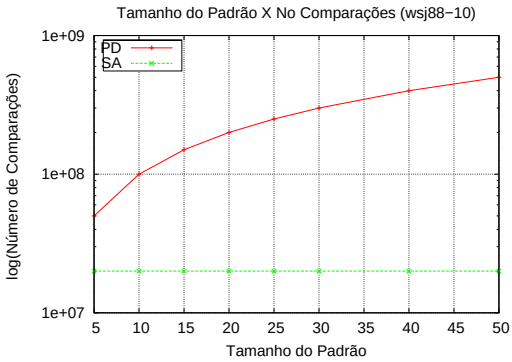
Da mesma forma descrita anteriormente, foram gerados padrões para consultas variando o tamanho das mesmas entre 5 e 50 caracteres aplicando as mesmas para todos os algoritmos implementados que permitiam casamento aproximado com tamanhos de erro igual a 1 em arquivos não comprimidos, ou seja, Shift-And-Aproximado(SAA) e Programação Dinâmica(PD), isso para todos os arquivos de teste. O resultado desses experimentos pode ser observado nos gráficos da Figura 8.

Para o algoritmo de Programação Dinâmica, para todos os arquivos avaliados podemos observar uma relação linear dentre o tamanho do padrão e o número de comparações, o que é coerente com a análise de complexidade realizada anteriormente, uma vez que cada caractere do padrão é comparado com cada caractere do texto. Já para o algoritmo Shit-And-Aproximado, o mesmo é indiferente ao tamanho do padrão, o que também é coerente com a análise apresentada anteriormente além de [23, 22] que fala que o Shift-And-Aproximado depende apenas do tamanho do texto e número de erros permitidos.

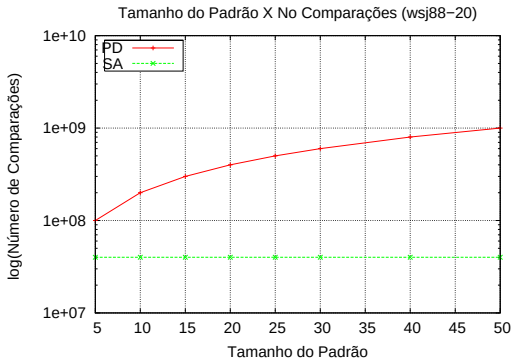
Para ambos os algoritmos, quanto maior o tamanho do arquivo a ser consultado maior também é o número de comparações, sendo mais uma vez coerente com a literatura pois ambos algoritmos dependem do tamanho do texto.



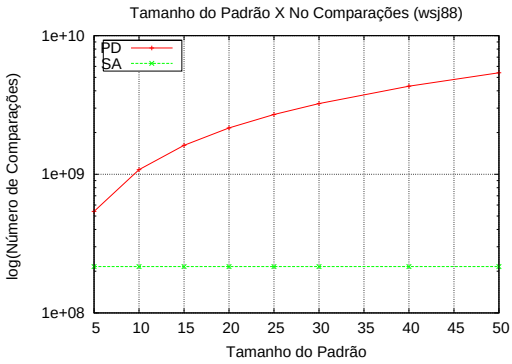
(a) Arquivo wsj89



(b) Arquivo wsj88_10



(c) Arquivo wsj88_20



(d) Arquivo wsj88

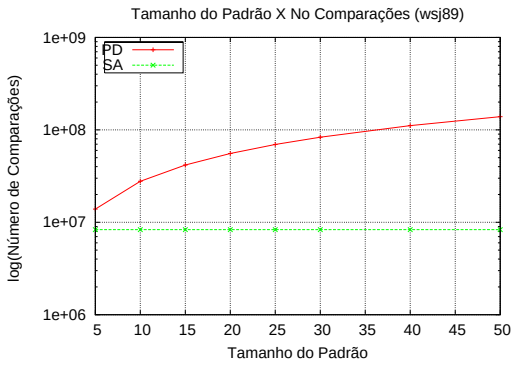
Figura 8: Número de Comparações em Relação ao Tamanho do Padrão - K = 1

Da mesma forma descrita anteriormente, foram gerados padrões para consultas variando o tamanho das mesmas entre 5 e 50 caracteres aplicando as mesmas para todos os algoritmos implementados que permitiam casamento aproximado com tamanhos de erro igual a 2 em arquivos não comprimidos, ou seja, Shift-And-Aproximado(SAA) e Programação Dinâmica(PD), isso para todos os arquivos de teste. O resultado desses experimentos pode ser observado nos gráficos da Figura 9.

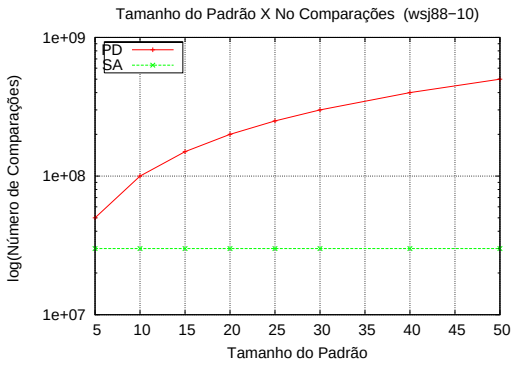
Para o algoritmo de Programação Dinâmica, para todos os arquivos avaliados podemos observar uma relação linear dentre o tamanho do padrão e o número de comparações, o que é coerente com a análise de complexidade realizada anteriormente, uma vez que cada caractere do padrão é comparado com cada caractere do texto. Já para o algoritmo Shit-And-Aproximado, o mesmo é indiferente ao tamanho do padrão, o que também é coerente com a análise apresentada anteriormente além de [23, 22] que fala que o Shift-And-Aproximado depende apenas do tamanho do texto e número de erros permitidos.

Para ambos os algoritmos, quanto maior o tamanho do arquivo a ser consultado maior também é o número de comparações, sendo mais uma vez coerente com a

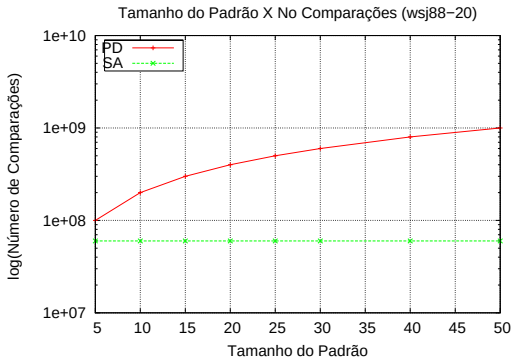
literatura pois ambos algoritmos dependem do tamanho do texto.



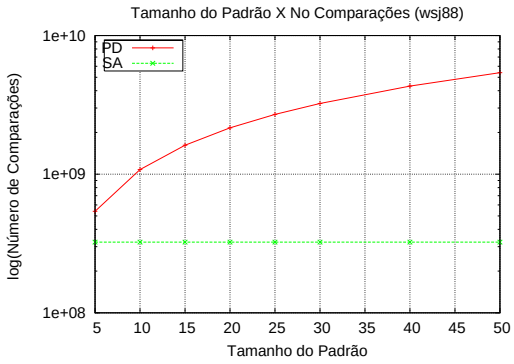
(a) Arquivo wsj89



(b) Arquivo wsj88_10



(c) Arquivo wsj88_20



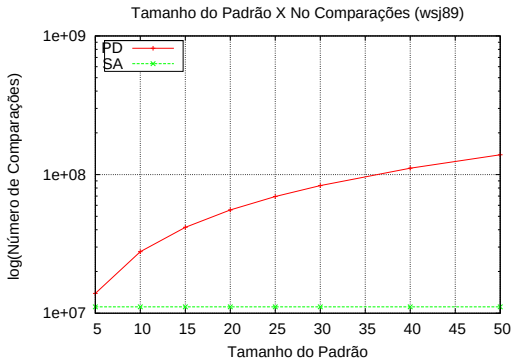
(d) Arquivo wsj88

Figura 9: Número de Comparações em Relação ao Tamanho do Padrão - K = 2

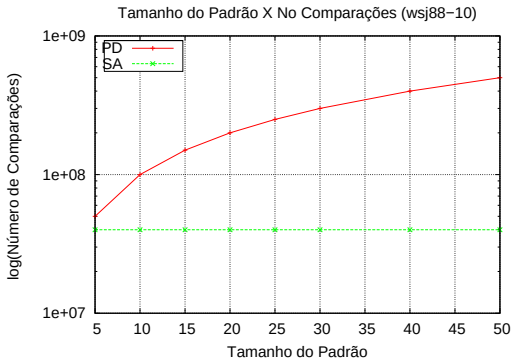
Da mesma forma descrita anteriormente, foram gerados padrões para consultas variando o tamanho das mesmas entre 5 e 50 caracteres aplicando as mesmas para todos os algoritmos implementados que permitiam casamento aproximado com tamanhos de erro igual a 3 em arquivos não comprimidos, ou seja, Shift-And-Aproximado(SAA) e Programação Dinâmica(PD), isso para todos os arquivos de teste. O resultado desses experimentos pode ser observado nos gráficos da Figura 10.

Para o algoritmo de Programação Dinâmica, para todos os arquivos avaliados podemos observar uma relação linear dentre o tamanho do padrão e o número de comparações, o que é coerente com a análise de complexidade realizada anteriormente, uma vez que cada caractere do padrão é comparado com cada caractere do texto. Já para o algoritmo Shit-And-Aproximado, o mesmo é indiferente ao tamanho do padrão, o que também é coerente com a análise apresentada anteriormente além de [23, 22] que fala que o Shift-And-Aproximado depende apenas do tamanho do texto e número de erros permitidos.

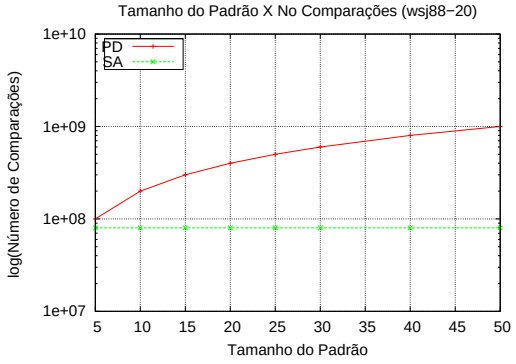
Para ambos os algoritmos, quanto maior o tamanho do arquivo a ser consultado maior também é o número de comparações, sendo mais uma vez coerente com a literatura pois ambos algoritmos dependem do tamanho do texto.



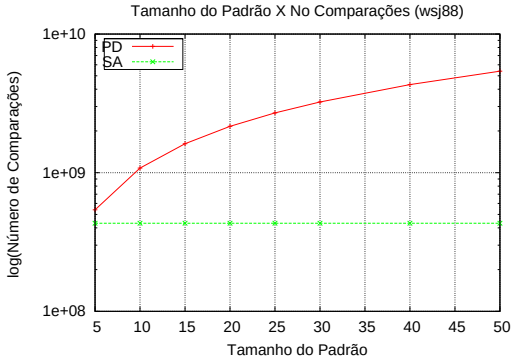
(a) Arquivo wsj89



(b) Arquivo wsj88_10



(c) Arquivo wsj88_20



(d) Arquivo wsj88

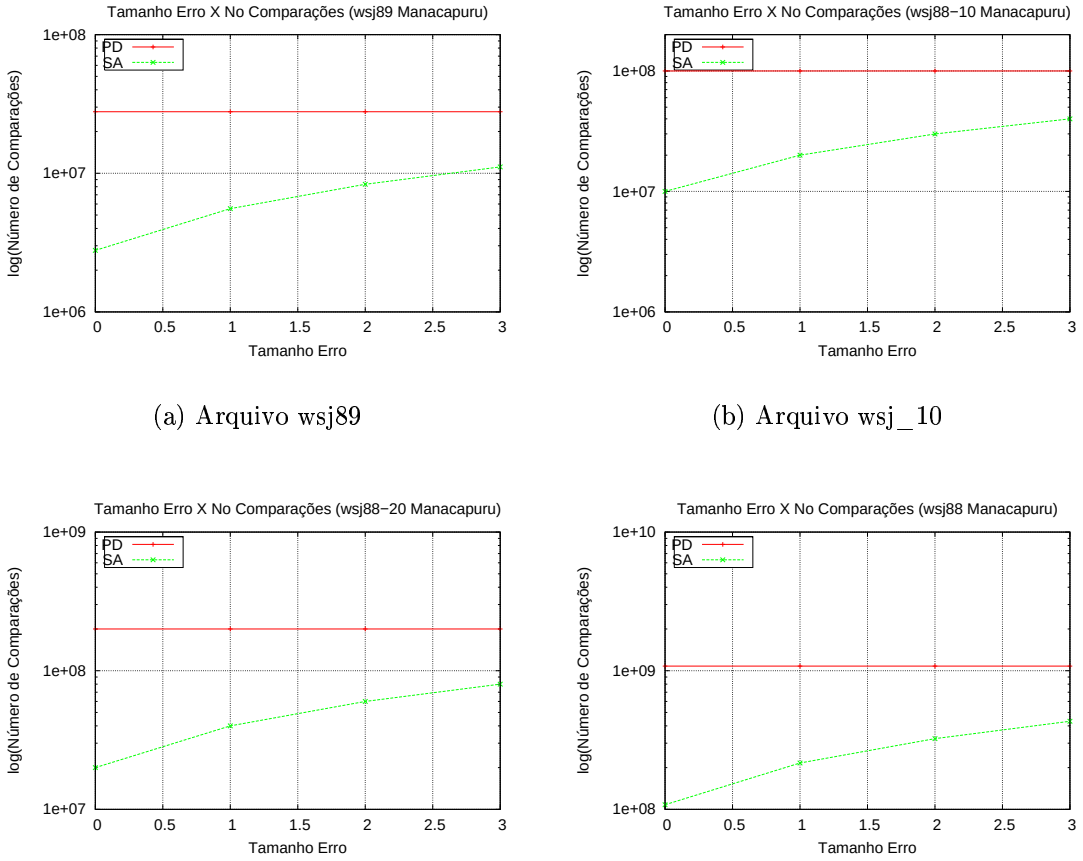
Figura 10: Número de Comparações em Relação ao Tamanho do Padrão - K = 3

Para avaliar o quanto o tamanho do erro permitido influencia no número de comparações realizadas, apresentamos na Figura 11 gráficos que relacionam o número de erros permitidos e o número de comparações realizadas para cada um dos arquivos de teste. Nesse caso, apresentamos apenas os dados referentes a consultas de uma padrão de 10 caracteres.

Podemos observar pelos gráficos que o algoritmo de Programação Dinâmica é indiferente ao tamanho de erros permitidos, o que é coerente com o funcionamento do mesmo apresentado anteriormente. Esse algoritmo realiza todas as comparações independentemente do número de erros permitidos e posteriormente ele avalia em quais posições ocorreram casamento com o número de erros permitidos.

Já o algoritmo de Shift-And-Aproximado possui uma relação linear entre o número de erros permitidos no casamento e o número de comparações realizadas. Esta constatação é coerente com a análise apresentada anteriormente além de [23, 22] que fala que o Shift-And-Aproximado depende do tamanho do texto e número de erros permitidos.

Para ambos os algoritmos, quanto maior o tamanho do arquivo a ser consultado maior também é o número de comparações, sendo mais uma vez coerente com a literatura e com os resultados anteriores, pois ambos algoritmos dependem do tamanho do texto.



(a) Arquivo wsj89

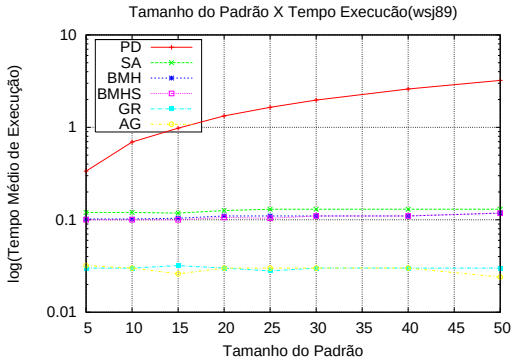
(b) Arquivo wsj_10

(c) Arquivo wsj_20

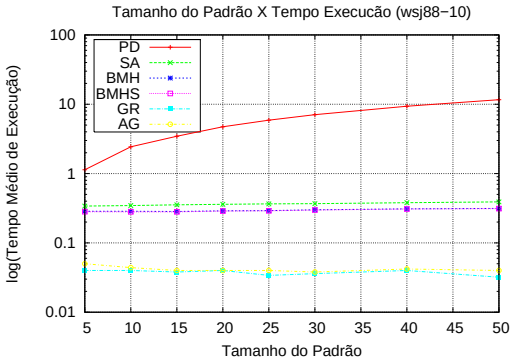
(d) Arquivo wsj88

Figura 11: Número de Comparações em Relação ao Número de Erros Permitidos

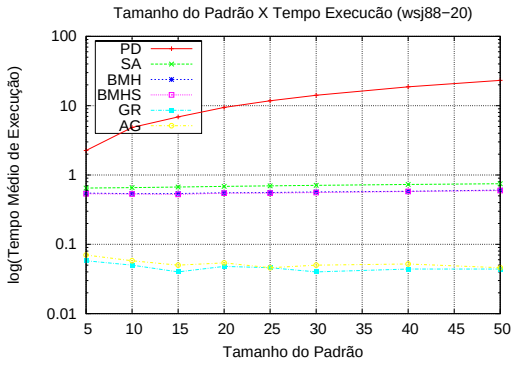
6.6.2 Casamento Aproximado - Tempo



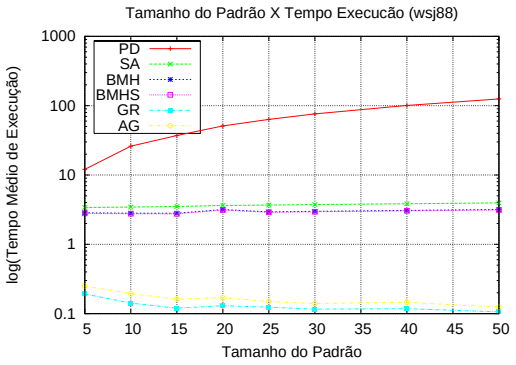
(a) Arquivo wsj89



(b) Arquivo wsj_10



(c) Arquivo wsj_20



(d) Arquivo wsj88

Figura 12: Tempo de Execução em Relação ao Tamanho do Padrão - K = 0

Foram gerados padrões para consultas variando o tamanho das mesmas entre 5 e 50 caracteres aplicando as mesmas para todos os algoritmos implementados que permitiam casamento aproximado com tamanhos de erro igual a 0 em arquivos não comprimidos, ou seja, Boyer-Moore-Horspool(BMH), Boyer-Moore-Horspool-Sunday(BMHS), Shift-And(SA), Programação Dinâmica(PD), Grep(GR) e Agrep(AG), isso para todos os arquivos de teste. O resultado relacionados aos tempos de execução desses experimentos pode ser observado nos gráficos da Figura 12.

Para o algoritmo de Programação Dinâmica, para todos os arquivos avaliados podemos observar uma relação linear dentre o tamanho do padrão e o tempo de execução, o que é coerente com a análise de complexidade realizada anteriormente, uma vez que cada caractere do padrão é comparado com cada caractere do texto. Além disso temos que o tempo de execução desse algoritmo é bem superior aos demais, principalmente se levarmos em consideração que os tempos de execução estão em escala logarítmica.

Já para o algoritmo Shit-And, o tempo o mesmo é indiferente ao tamanho do padrão, o que também é coerente com a análise apresentada anteriormente além de [23, 2] que fala que o Shift-And depende apenas do tamanho do texto. Os tempos

de execução desse algoritmo estão bem abaixo do tempos de execução do algoritmo de Programação Dinâmica e um pouco acima dos tempos de execução dos demais algoritmos.

Nesses mesmos gráficos, analisando os algoritmos de Boyer-Moore-Horspool e Boyer-Moore-Horspool-Sunday, temos que a medida que o tamanho do padrão aumenta, temos a impressão que o tempo de execução é constante, ao invés de diminuir como esperado (saltos maiores). Isso pode ser facilmente explicado. Como já foi dito anteriormente, os casamentos de padrões são feitas linha a linha do arquivo de entrada para esses algoritmos. Assim, já era de nosso conhecimento que o tempo de leitura/escrita em disco influenciaria um pouco em todos os resultados que envolvessem tempo de execução, isso para todos os algoritmos implementados, o que de fato aconteceu. No entanto, para os demais algoritmos essa influência foi pouco significativa uma vez que os tempos totais de execução eram muito maiores que esse tempo de acesso a disco. No entanto, os tempo de execução desses algoritmos são muito baixo se comparado com os demais, o que torna significativa a influência do tempo de acesso a disco nesse caso. Não só esse tempo de acesso, mas qualquer outra carga diferente no sistema, por menor que a mesma fosse. Por exemplo, 0.1 segundo em 10 segundos é muito pouco(0.1%), mas 0.1 segundo em 1 segundo é muita coisa(10%).

Se arquivos maiores para consultas fossem utilizados nos testes, esse tempo influenciaria menos pois o tempo de execução desses algoritmos seria maior. Além disso a escala logarítmica torna quase que imperceptível observar que o tempo de execução diminui(o que realmente ocorre). Apesar de prevermos que isso aconteceria, esse foi o preço pago para que o funcionamento dos algoritmos fosse coerente com o funcionamento dos aplicativos Grep e Agrep que não aceitam casamento de padrões quando o mesmo ocorre quebrado em linhas diferentes. Dessa forma, mais uma vez é possível observar uma coerência entre os resultados obtidos empiricamente com os apresentados na literatura [23, 4, 8].

Por fim podemos observar nos gráficos as curvas de tempo de execução para os aplicativos Grep e Agrep. O tempos de execução desses dois aplicativos são muito bons e é bem provável que apesar dos mesmos não admitirem casamento de palavras com quebra de linha, todo o arquivo deve ser carregado em memória. É possível perceber ainda nessas curvas que o tempo de execução tende a diminuir a medida que o tamanho do padrão aumenta, o tempo de execução tende a diminuir. Isso é coerente com [21] onde os autores apresentam que o Agrep utiliza um versão simplificada do algoritmo de Boyer-Moore para realizar casamento de padrões simples, como é em nosso caso.

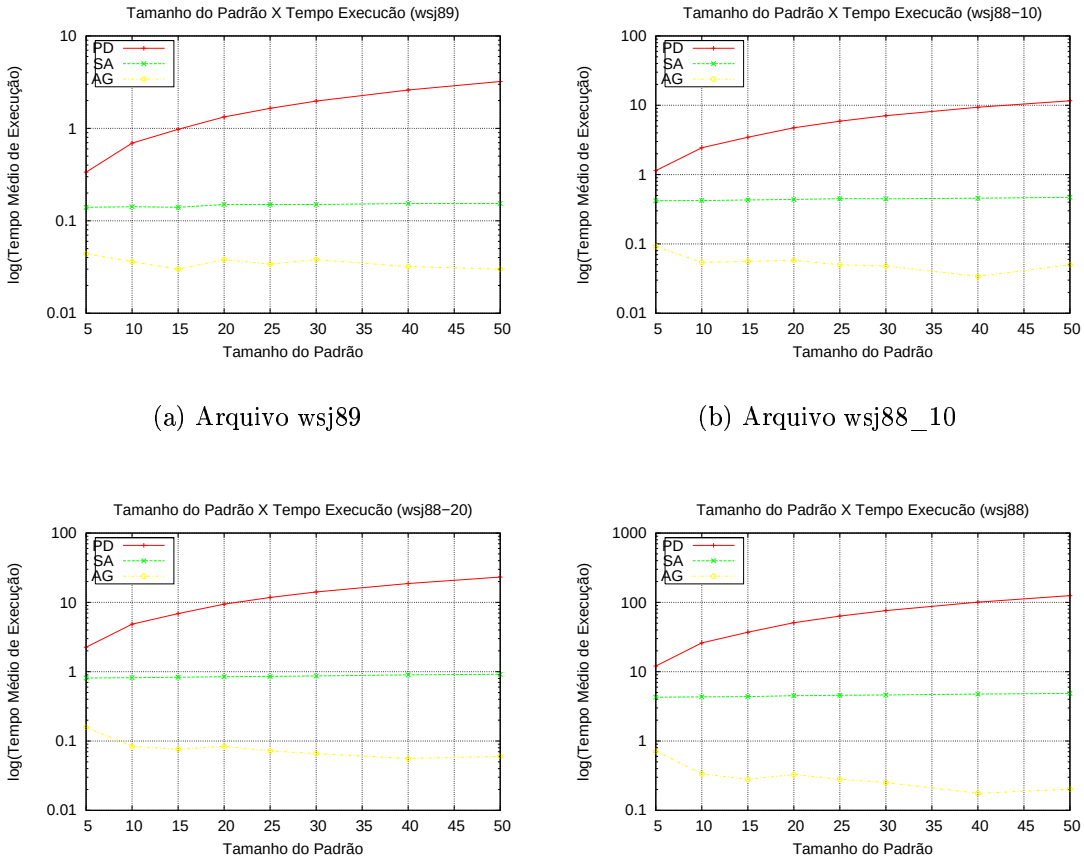
Para todos os algoritmos, quanto maior o tamanho do arquivo a ser consultado maior também é o número de comparações, sendo mais uma vez coerente com a literatura pois todos esses algoritmos dependem do tamanho do texto.

Da mesma forma descrita anteriormente, foram gerados padrões para consultas variando o tamanho das mesmas entre 5 e 50 caracteres aplicando as mesmas para todos os algoritmos implementados que permitiam casamento aproximado com tamanhos de erro igual a 1 em arquivos não comprimidos, ou seja, Shift-And-Aproximado(SAA), Programação Dinâmica(PD) e o aplicativo Agrep(AG), isso para todos os arquivos de teste. O resultado relacionados ao tempo de execução desses experimentos pode ser observado nos gráficos da Figura 13.

Para o algoritmo de Programação Dinâmica, para todos os arquivos avaliados podemos observar uma relação linear dentre o tamanho do padrão e o tempo de execução, o que é coerente com a análise de complexidade realizada anteriormente, uma vez que cada caractere do padrão é comparado com cada caractere do texto. Já para o algoritmo Shit-And-Aproximado, o mesmo é indiferente ao tamanho do padrão, o que também é coerente com a análise apresentada anteriormente além de [23, 22] que fala que o Shift-And-Aproximado depende apenas do tamanho do texto e número de erros permitidos.

Por fim podemos observar nos gráficos as curvas de tempo de execução para o aplicativo Agrep. O tempos de execução desse aplicativo é muito bom. É possível perceber na curva de tempo desse aplicativo que o tempo de execução tende a diminuir a medida que o tamanho do padrão aumenta. Isso é coerente com [21] onde os autores apresentam que o Agrep utiliza um versão simplificada do algoritmo de Boyer-Moore para realizar casamento de padrões simples, como é em nosso caso.

Para todos os algoritmos, quanto maior o tamanho do arquivo a ser consultado maior também é o tempo de execução, sendo mais uma vez coerente com a literatura pois ambos algoritmos dependem do tamanho do texto.



(a) Arquivo wsj89

(b) Arquivo wsj88_10

(c) Arquivo wsj88_20

(d) Arquivo wsj88

Figura 13: Tempo de Execução em Relação ao Tamanho do Padrão - K = 1

Da mesma forma descrita anteriormente, foram gerados padrões para consultas variando o tamanho das mesmas entre 5 e 50 caracteres aplicando as mesmas para todos os algoritmos implementados que permitiam casamento aproximado com tamanhos de erro igual a 2 em arquivos não comprimidos, ou seja, Shift-And-Aproximado(SAA), Programação Dinâmica(PD) e o aplicativo Agrep(AG), isso para todos os arquivos de teste. O resultado relacionados ao tempo de execução desses experimentos pode ser observado nos gráficos da Figura 14.

Para o algoritmo de Programação Dinâmica, para todos os arquivos avaliados podemos observar uma relação linear dentre o tamanho do padrão e o tempo de execução, o que é coerente com a análise de complexidade realizada anteriormente, uma vez que cada caractere do padrão é comparado com cada caractere do texto. Já para o algoritmo Shit-And-Aproximado, o mesmo é indiferente ao tamanho do padrão, o que também é coerente com a análise apresentada anteriormente além de [23, 22] que fala que o Shift-And-Aproximado depende apenas do tamanho do texto e número de erros permitidos.

Por fim podemos observar nos gráficos as curvas de tempo de execução para o aplicativo Agrep. O tempos de execução desse aplicativo é muito bom. É possível perceber na curva de tempo desse aplicativo que o tempo de execução tende a diminuir a medida que o tamanho do padrão aumenta. Isso é coerente com [21] onde os autores apresentam que o Agrep utiliza um versão simplificada do algoritmo de Boyer-Moore para realizar casamento de padrões simples, como é em nosso caso.

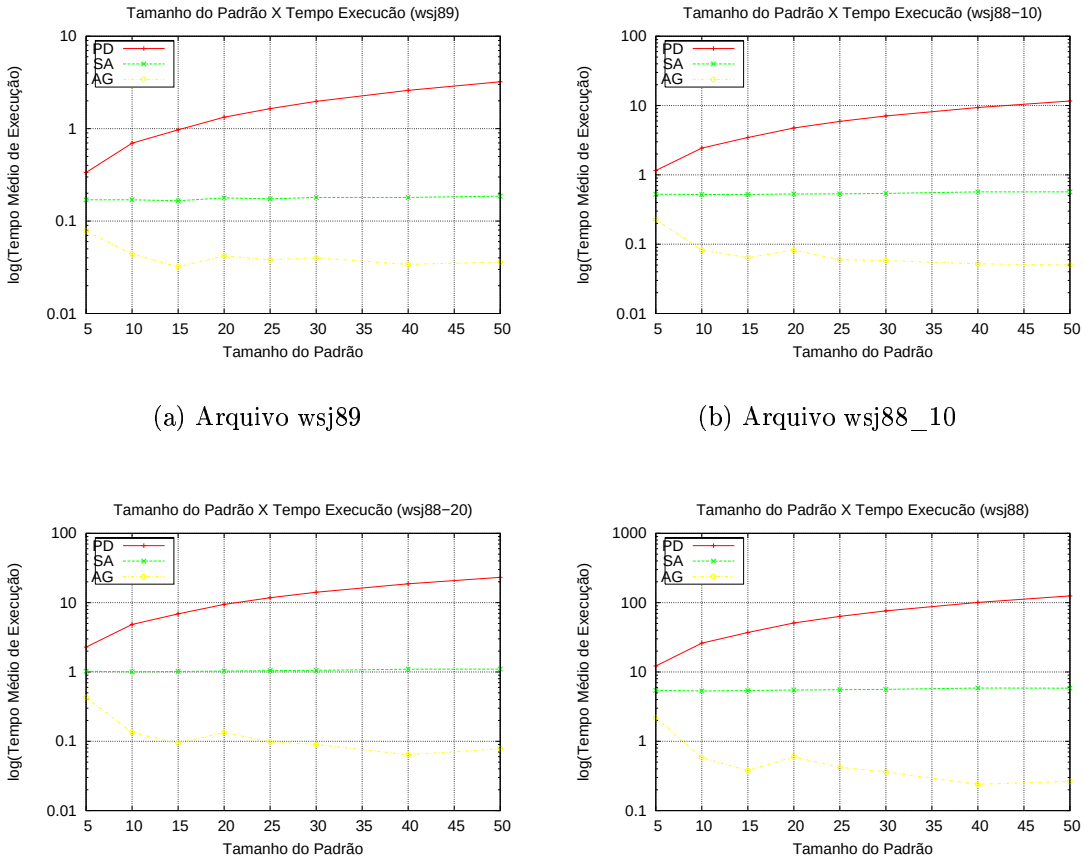
Para todos os algoritmos, quanto maior o tamanho do arquivo a ser consultado maior também é o tempo de execução, sendo mais uma vez coerente com a literatura pois ambos algoritmos dependem do tamanho do texto.

Da mesma forma descrita anteriormente, foram gerados padrões para consultas variando o tamanho das mesmas entre 5 e 50 caracteres aplicando as mesmas para todos os algoritmos implementados que permitiam casamento aproximado com tamanhos de erro igual a 3 em arquivos não comprimidos, ou seja, Shift-And-Aproximado(SAA), Programação Dinâmica(PD) e o aplicativo Agrep(AG), isso para todos os arquivos de teste. O resultado relacionados ao tempo de execução desses experimentos pode ser observado nos gráficos da Figura 15.

Para o algoritmo de Programação Dinâmica, para todos os arquivos avaliados podemos observar uma relação linear dentre o tamanho do padrão e o tempo de execução, o que é coerente com a análise de complexidade realizada anteriormente, uma vez que cada caractere do padrão é comparado com cada caractere do texto. Já para o algoritmo Shit-And-Aproximado, o mesmo é indiferente ao tamanho do padrão, o que também é coerente com a análise apresentada anteriormente além de [23, 22] que fala que o Shift-And-Aproximado depende apenas do tamanho do texto e número de erros permitidos.

Por fim podemos observar nos gráficos as curvas de tempo de execução para o aplicativo Agrep. O tempos de execução desse aplicativo é muito bom. É possível perceber na curva de tempo desse aplicativo que o tempo de execução tende a diminuir a medida que o tamanho do padrão aumenta. Isso é coerente com [21] onde os autores apresentam que o Agrep utiliza um versão simplificada do algoritmo de Boyer-Moore para realizar casamento de padrões simples, como é em nosso caso.

Para todos os algoritmos, quanto maior o tamanho do arquivo a ser consultado maior também é o temppo de execução, sendo mais uma vez coerente com a literatura



(a) Arquivo wsj89

(b) Arquivo wsj88_10

(c) Arquivo wsj88_20

(d) Arquivo wsj88

Figura 14: Tempo de Execução em Relação ao Tamanho do Padrão - K = 2

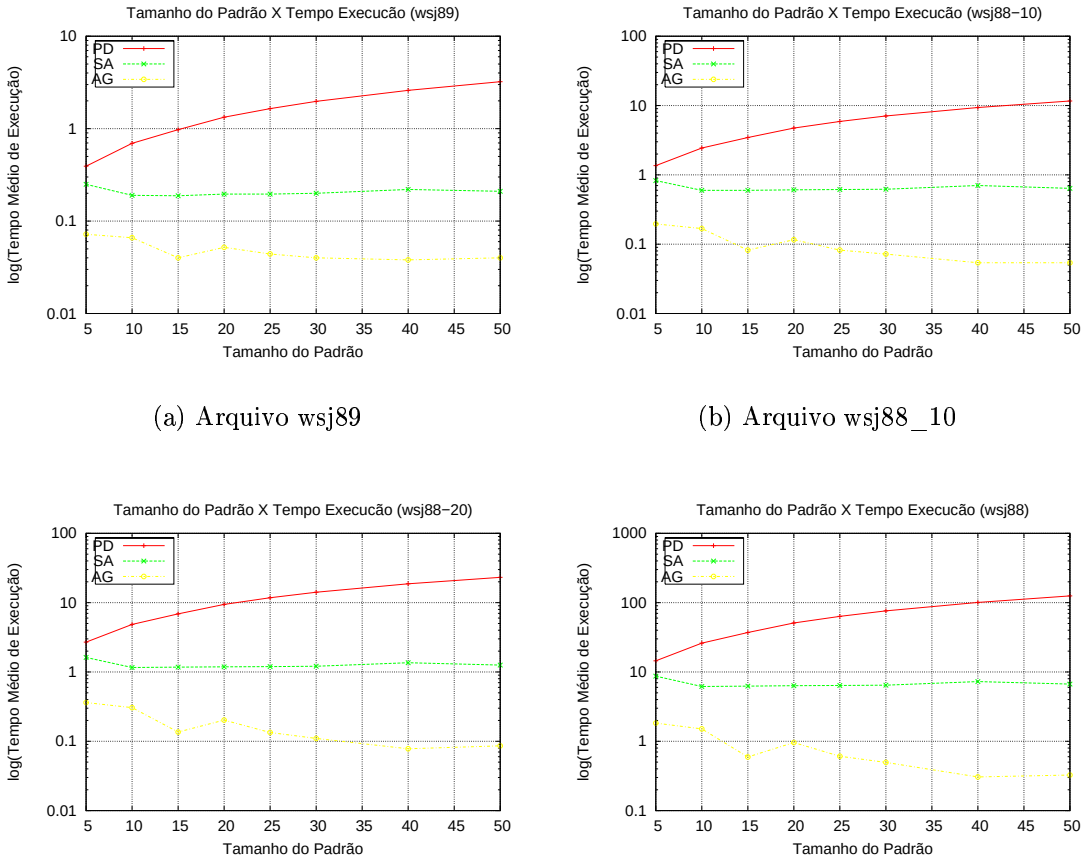
pois ambos algoritmos dependem do tamanho do texto.

Para avaliar o quanto o tamanho do erro permitido influencia no tempo de execução dos algoritmos, apresentamos na Figura 16 gráficos que relacionam o número de erros permitidos e o tempo de execução para cada um dos arquivos de teste. Nesse caso, apresentamos apenas os dados referentes a consultas de uma padrão de 10 caracteres.

Podemos observar pelos gráficos que o algoritmo de Programação Dinâmica é indiferente ao tamanho de erros permitidos, o que é coerente com o funcionamento do mesmo apresentado anteriormente. Esse algoritmo realiza todas as comparações independentemente do número de erros permitidos e posteriormente ele avalia em quais posições ocorreram casamento com o número de erros permitidos, ou seja, independe do número de erros.

Já o algoritmo de Shift-And-Aproximado possui uma relação linear entre o número de erros permitidos no casamento e o tempo de execução. Esta constatação é coerente com a análise apresentada anteriormente além de [23, 22] que fala que o Shift-And-Aproximado depende do tamanho do texto e número de erros permitidos.

Nos gráficos anteriores mostramos que a medida que o tamanho do padrão aumentava o tempo de execução do aplicativo Agrep diminuía. Nos graficos apresentados na Figura 16 temos o tempo de execução do Agrep aumenta a medida que o



(a) Arquivo wsj89

(b) Arquivo wsj88_10

(c) Arquivo wsj88_20

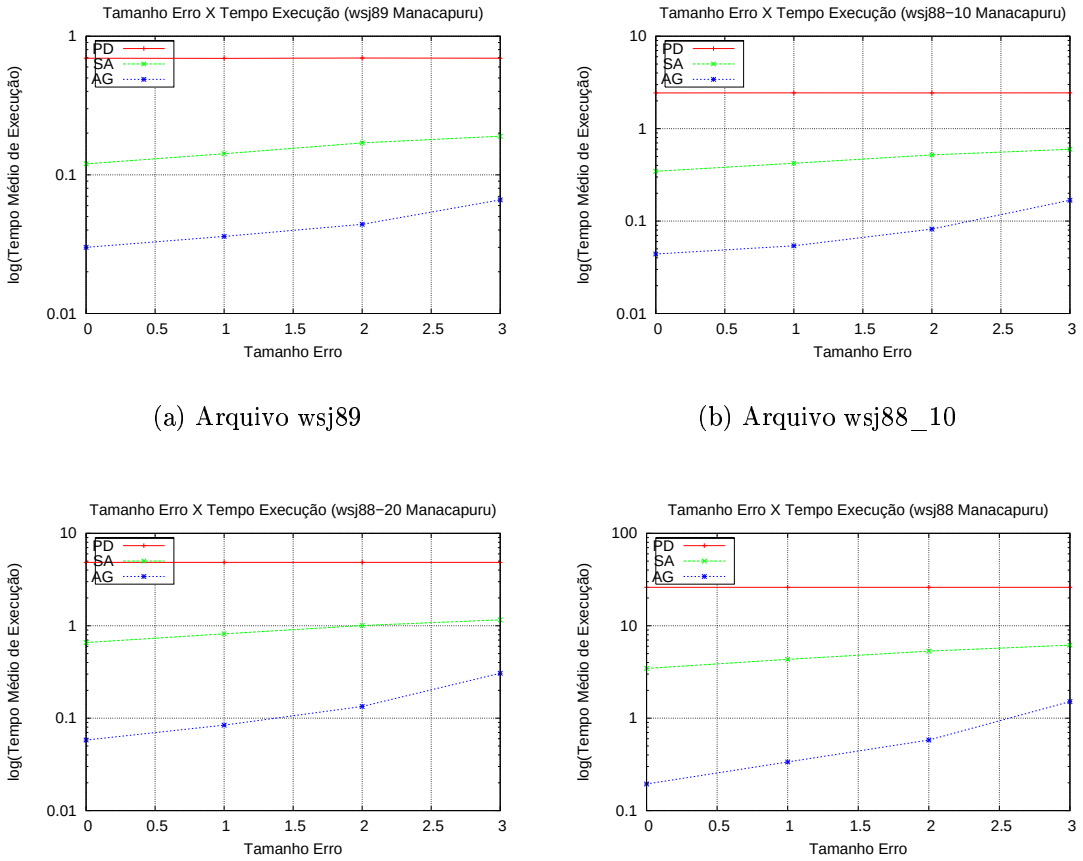
(d) Arquivo wsj88

Figura 15: Tempo de Execução em Relação ao Tamanho do Padrão - K = 3

o número de erros permitidos aumenta. Isso acontece uma vez que o Agrep possui uma implementação conjugada de Shift-And-Aproximado e Boyer-Moore. Os deslocamentos no texto ocorrem segundo o algoritmo de Boyer-Moore, por isso o tempo de execução do Agrep diminui a medida que aumentamos o padrão. Entretanto, a comparação entre o padrão e cada sub-cadeia do texto é feita utilizando o Shift-And-Aproximado para erros sejam permitidos. Como já mencionado, o tempo de execução do Shift-And-Aproximado aumenta com o número de erros, logo o tempo do Agrep também aumenta.

6.6.3 Casamento Exato - Número de Comparações

Foram gerados padrões para consultas variando o tamanho das mesmas entre 5 e 50 caracteres aplicando as mesmas para todos os algoritmos implementados que permitiam casamento exato em arquivos não-comprimidos e comprimidos. Para arquivos não-comprimidos utilizamos os algoritmos de Boyer-Moore-Horspool(BMH), Boyer-Moore-Horspool-Sunday(BMHS), Shift-And(SA) e Programação Dinâmica(PD). Para a busca em arquivos comprimidos utilizamos Boyer-Moore-Horspool(BMHCOMP). Isso para todos os arquivos de teste. O objetivo desses experimentos é realizar uma análise comparativa entre algoritmos de busca exata em arquivos comprimidos



(a) Arquivo wsj89

(b) Arquivo wsj88_10

(c) Arquivo wsj88_20

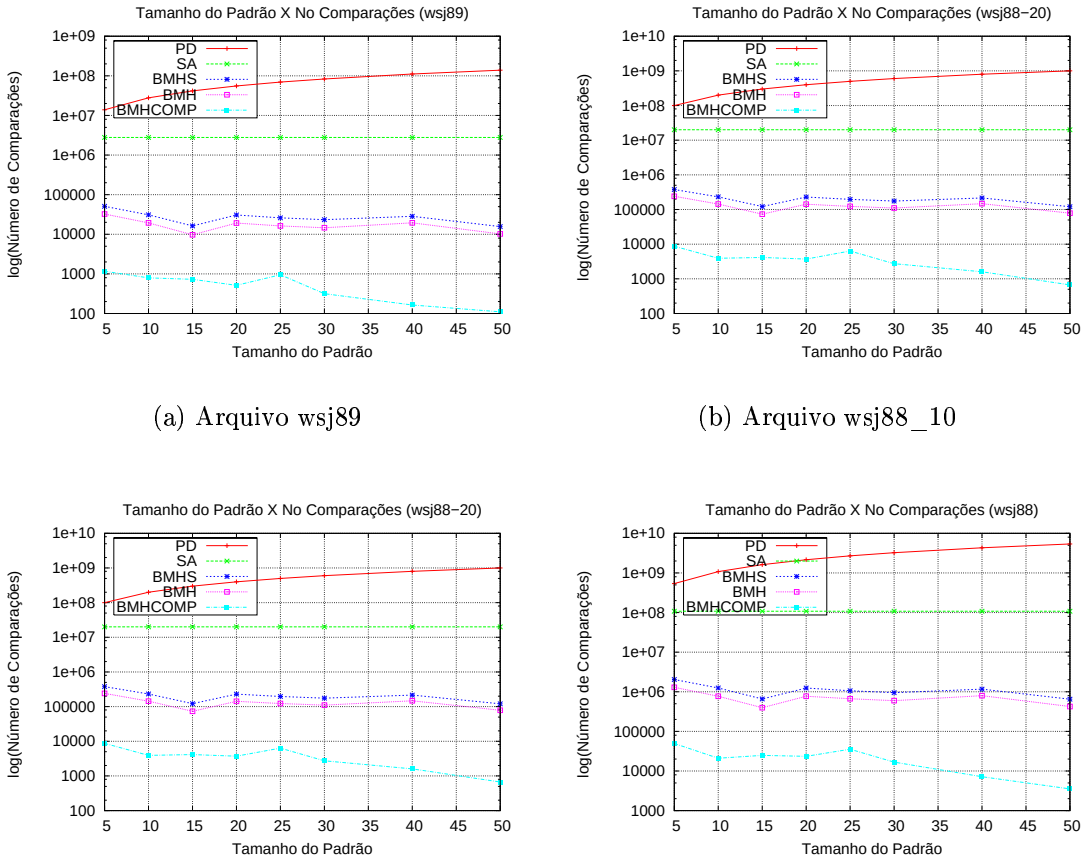
(d) Arquivo wsj88

Figura 16: Tempo de Execução em Relação ao Número de Erros Permitidos

e não-comprimidos. O resultado desses experimentos relacionados ao número de comparações realizados pode ser observado nos gráficos da Figura 17.

Para o algoritmo de Programação Dinâmica, para todos os arquivos avaliados podemos observar uma relação linear dentre o tamanho do padrão e o número de comparações, o que é coerente com a análise de complexidade realizada anteriormente, uma vez que cada caractere do padrão é comparado com cada caractere do texto. Já para o algoritmo Shit-And, o mesmo é indiferente ao tamanho do padrão, o que também é coerente com a análise apresentada anteriormente além de [23, 2] que fala que o Shift-And depende apenas do tamanho do texto.

Nesses mesmos gráficos, analisando os algoritmos de Boyer-Moore-Horspool, Boyer-Moore-Horspool em arquivos comprimidos e Boyer-Moore-Horspool-Sunday, temos que a medida que o tamanho do padrão aumenta, é possível observar que o número de comparações tendem a diminuir, o que também é coerente com a análise feita anteriormente além de [23, 4, 8] uma vez que os saltos de comparações tendem a serem maiores. Além disso, podemos observar que o número de comparações realizadas pelo Boyer-Moore-Horspool é um pouco menor que o número de comparações do Boyer-Moore-Horspool-Sunday uma vez que esse último é melhor para padrões curtos pois os saltos tendem a ser maiores$(m + 1)$ e os padrões analisados tem tamanho mínimo 5. O menor número de comparações acontece para



(a) Arquivo wsj89

(b) Arquivo wsj88_10

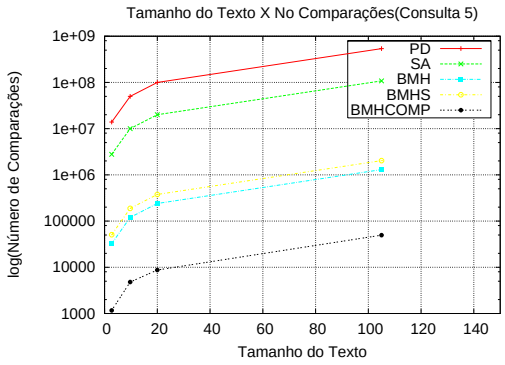
(c) Arquivo wsj88_20

(d) Arquivo wsj88

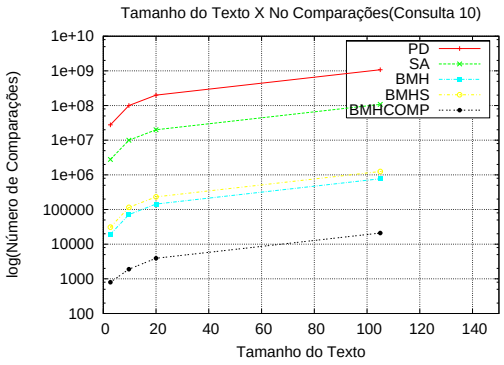
Figura 17: Número de Comparações em Relação ao Tamanho do Padrão - Casamento Exato

o algoritmo de Boyer-Moore-Horspool em arquivos comprimidos, o que é bastante coerente com a literatura [5, 6, 24]. É isso que faz com que, mesmo com o *overhead* para se comprimir e descomprimir, busca em arquivos comprimidos se mostre tão atraente.

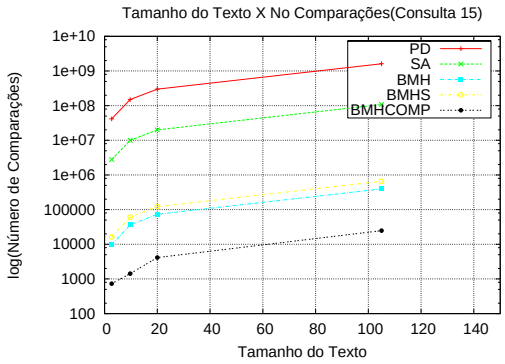
Ainda com base nos experimentos descritos acima, apresentamos nos gráficos da Figura 18 uma análise da relação entre o número de comparações e o tamanho do arquivo a ser pesquisado utilizando consultas de 5 10 15 e 20 caracteres. Por essa Figura é possível perceber a relação linear entre o número de comparações de cada um dos algoritmos implementados e o tamanho do texto, sendo novamente bastante coerente com que é apresentado na literatura [5, 6, 24, 4, 8, 2]. Além disso, podemos ver que o menor número de comparações acontece para o algoritmo de Boyer-Moore-Horspool em arquivos comprimidos, seguido pelos algoritmos Boyer-Moore-Horspool, Boyer-Moore-Horspool-Sunday, Shift-And-Exato e Programação Dinâmica.



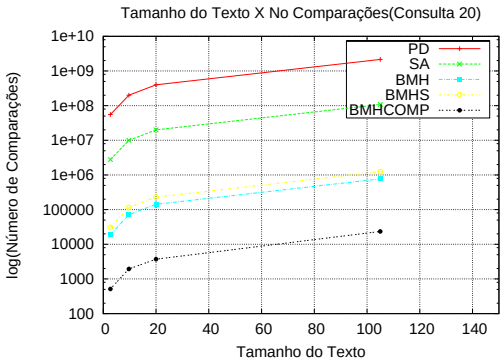
(a) Consulta 5 caracteres



(b) Consulta 10 caracteres



(c) Consulta 15 caracteres



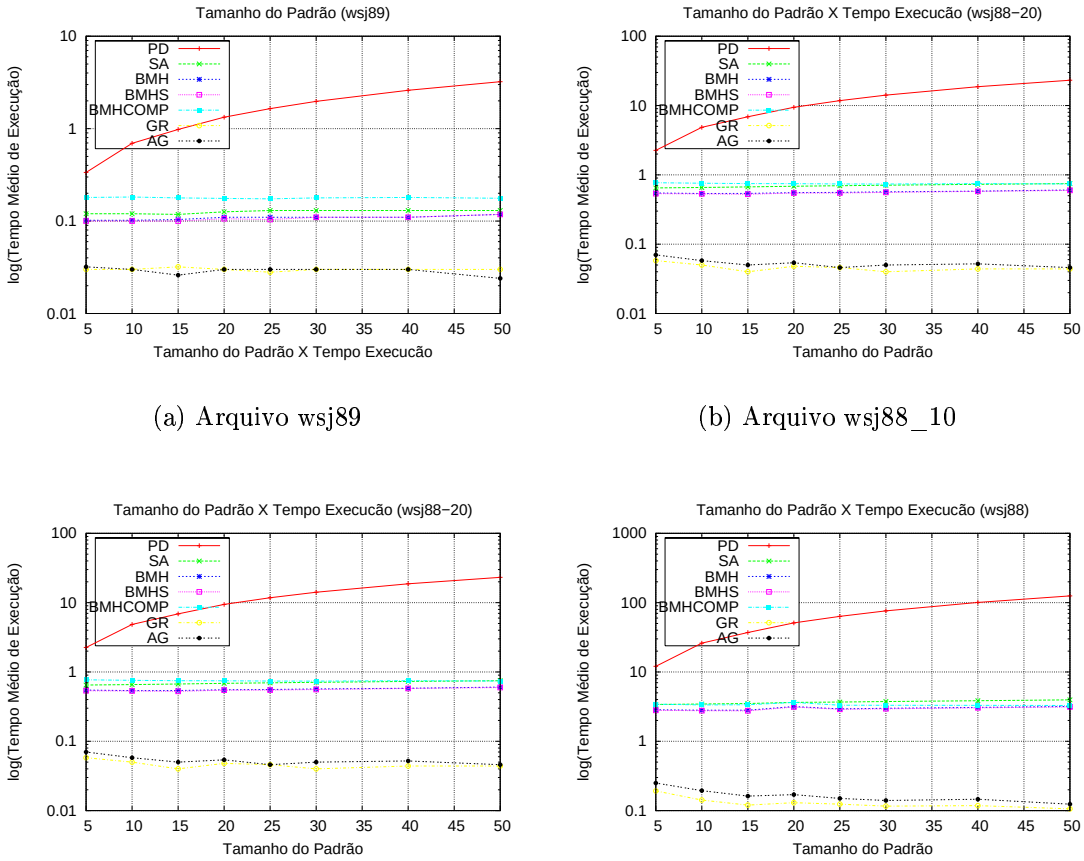
(d) Consulta 20 caracteres

Figura 18: Número de Comparações em Relação ao Tamanho dos Textos

6.6.4 Casamento Exato - Tempo

Foram gerados padrões para consultas variando o tamanho das mesmas entre 5 e 50 caracteres aplicando as mesmas para todos os algoritmos implementados que permitiam casamento exato em arquivos não-comprimidos e comprimidos. Para arquivos não-comprimidos utilizamos os algoritmos de Boyer-Moore-Horspool(BMH), Boyer-Moore-Horspool-Sunday(BMHS), Shift-And(SA) e Programação Dinâmica(PD). Para a busca em arquivos comprimidos utilizamos Boyer-Moore-Horspool(BMHCOMP). Isso para todos os arquivos de teste. O objetivo desses experimentos é realizar uma análise comparativa entre algoritmos de busca exata em arquivos comprimidos e não-comprimidos. O resultado desses experimentos relacionados ao tempo de execução realizados pode ser observado nos gráficos da Figura 19.

Para o algoritmo de Programação Dinâmica, para todos os arquivos avaliados podemos observar uma relação linear dentre o tamanho do padrão e o tempo de execução, o que é coerente com a análise de complexidade realizada anteriormente, uma vez que cada caractere do padrão é comparado com cada caractere do texto. Além disso temos que o tempo de execução desse algoritmo é bem superior aos demais, principalmente se levarmos em consideração que os tempos de execução estão em escala logarítmica.



(a) Arquivo wsj89

(b) Arquivo wsj88_10

(c) Arquivo wsj88_20

(d) Arquivo wsj88

Figura 19: Tempo de Execução em Relação ao Tamanho do Padrão - Casamento Exato

Já para o algoritmo Shit-And, o tempo o mesmo é indiferente ao tamanho do padrão, o que também é coerente com a análise apresentada anteriormente além de [23, 2] que fala que o Shift-And depende apenas do tamanho do texto. Os tempos de execução desse algoritmo estão bem abaixo do tempos de execução do algoritmo de Programação Dinâmica e um pouco acima dos tempos de execução dos demais algoritmos.

Nesses mesmos gráficos,mais uma vez, analisando os algoritmos de Boyer-Moore-Horspool e Boyer-Moore-Horspool-Sunday, temos que a medida que o tamanho do padrão aumenta, temos a impressão que o tempo de execução é constante, ao invés de diminuir como esperado (saltos maiores). Isso pode ser facilmente explicado. Como já foi dito anteriormente, os casamentos de padrões são feitas linha a linha do arquivo de entrada para esses algoritmos. Assim, já era de nosso conhecimento que o tempo de leitura/escrita em disco influenciaria um pouco em todos os resultados que envolvessem tempo de execução, isso para todos os algoritmos implementados, o que de fato aconteceu. No entanto, para os demais algoritmos essa influência foi pouco significativa uma vez que os tempos totais de execução eram muito maiores que esse tempo de acesso a disco. No entanto, os tempo de execução desses algoritmos são muito baixo se comparado com os demais, o que torna significativa a influência do

tempo de acesso a disco nesse caso. Não só esse tempo de acesso, mas qualquer outra carga diferente no sistema, por menor que a mesma fosse. Por exemplo, 0.1 segundo em 10 segundos é muito pouco(0.1%), mas 0.1 segundo em 1 segundo é muita coisa(10%).

Se arquivos maiores para consultas fossem utilizados nos testes, esse tempo influenciaria menos pois o tempo de execução desses algoritmos seria maior. Além disso a escala logarítmica torna quase que imperceptível observar que o tempo de execução diminui(o que realmente ocorre). Apesar de prevermos que isso aconteceria, esse foi o preço pago para que o funcionamento dos algoritmos fosse coerente com o funcionamento dos aplicativos Grep e Agrep que não aceitam casamento de padrões quando o mesmo ocorre quebrado em linhas diferentes. Dessa forma, mais uma vez é possível observar uma coerência entre os resultados obtidos empiricamente com os apresentados na literatura [23, 4, 8].

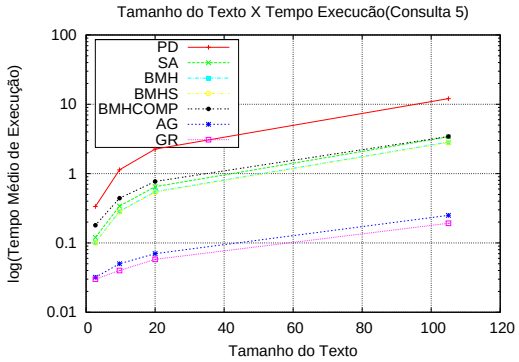
Apesar de realizar um número de comparações bem inferior aos demais algoritmos(como exceção do Grep e Agrep que não puderam ser comparados), o algoritmo de Boyer-Moore-Horspool para arquivos comprimidos obteve um tempo de execução muito alto, sendo superado apenas pelo algoritmo de Programação Dinâmica. Esse resultado, apesar de a primeira vista parecer incoerente, era um resultado esperado. Isso porque a implementação da busca em arquivos comprimidos utilizada(disponibilizada por [23]) é extremamente ineficiente. Nessa implementação, toda vez que uma consulta é feita, primeiramente o padrão a ser pesquisado é procurado no vocabulário para obter-se a codificação do mesmo e esse vocabulário é implementado de forma bastante ineficiente através de um vetor o qual é percorrido até que a padrão em questão seja encontrado, ou seja, tem uma complexidade de $O(n^{-2})$(se considerarmos que o número de palavras distintas em um texto é n^2). Depois dessa busca no vocabulário é que é feita a busca pelo padrão no texto propriamente dito. Esse caso ainda pode ser pior se consideramos que, em caso do padrão ser composto por mais de uma palavra (como é o caso da maioria das consultas realizadas nesses experimentos), essa busca deve ser feita para cada uma dessas palavras contidas no padrão, tornando o algoritmo ainda menos eficiente. Para tornar esse algoritmo competitivo com as demais implementações que se mostraram eficientes, além das alterações já citadas em seções anteriores, o vetor que contém o vocabulário deve ser transformado em uma *hash* e o parser deve ser melhor implementado.

Por fim podemos observar nos gráficos as curvas de tempo de execução para os aplicativos Grep e Agrep. O tempos de execução desses dois aplicativos são muito bons e é bem provável que apesar dos mesmos não admitirem casamento de palavras com quebra de linha, todo o arquivo deve ser carregado em memória. É possível perceber ainda nessas curvas que o tempo de execução tende a diminuir a medida que o tamanho do padrão aumenta, o tempo de execução tende a diminuir. Isso é coerente com [21] onde os autores apresentam que o Agrep utiliza um versão simplificada do algoritmo de Boyer-Moore para realizar casamento de padrões simples, como é em nosso caso.

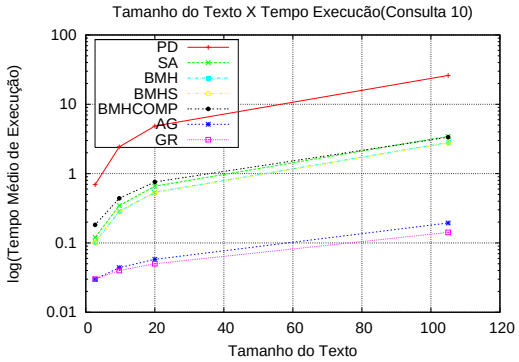
Para todos os algoritmos, quanto maior o tamanho do arquivo a ser consultado maior também é o número de comparações, sendo mais uma vez coerente com a literatura pois todos esses algoritmos dependem do tamanho do texto.

Ainda com base nos experimentos descritos acima, apresentamos nos gráficos da Figura 20 uma análise da relação entre o tempo de execução e o tamanho do

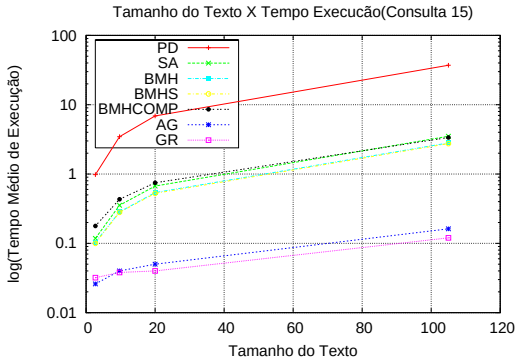
arquivo a ser pesquisado utilizando consultas de 5 10 15 e 20 caracteres. Por essa Figura é possível perceber a relação linear entre o tempo de execução de cada um dos algoritmos implementados e o tamanho do texto, sendo novamente bastante coerente com que é apresentado na literatura [5, 6, 24, 4, 8, 2]. Além disso, podemos ver que o menor tempo de execução acontece para os aplicativo Grep e Agrep, seguidos pelos algoritmos Boyer-Moore-Horspool, Boyer-Moore-Horspool-Sunday, Shift-And-Exato, Boyer-Moore-Horspool para arquivos comprimidos e Programação Dinâmica, resultados estes já discutidos anteriormente.



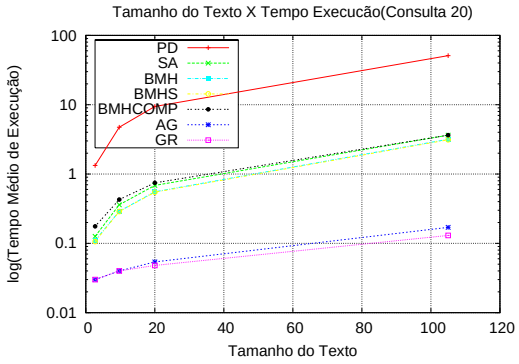
(a) Consulta 5 caracteres



(b) Consulta 10 caracteres



(c) Consulta 15 caracteres



(d) Consulta 20 caracteres

Figura 20: Tempo de Execução em Relação ao Tamanho dos Textos

7 Conclusão

Neste trabalho apresentamos uma avaliação de alguns dos principais algoritmos de recuperação de padrões em arquivos constituídos de documentos comprimidos e não comprimidos. Os algoritmos utilizados nessa avaliação foram um algoritmo de Programação Dinâmica, o algoritmo de Shift-And para casamento aproximado e casamento exato, o algoritmo de casamento exato de Boyer-Moore-Horspool para arquivos comprimidos e não-comprimidos e uma versão melhorada desse último, o Boyer-Moore-Horspool-Sunday. De todos os programas avaliados, o Grep e o Agrep foram o que se mostraram mais eficientes em relação ao tempo de execução, isso para todos os casos. Dos algoritmos implementados, temos que em termos de número de comparações realizadas, o algoritmo de Boyer-Moore-Horspool para arquivos comprimidos foi o que se mostrou mais eficiente, seguido dos algoritmos Boyer-Moore-Horspool, Boyer-Moore-Horspool-Sunday, Shift-And e Programação Dinâmica. No entanto, em nossos experimentos constatamos um ineficiência muito grande na implementação do Boyer-Moore-Horspool para arquivos comprimidos devido a forma com que os padrões são obtidos do vocabulário. Além disso foi possível observar o quanto o tempo de acesso a disco pode influenciar nesse tipo de aplicação.

8 Agradecimentos

Agradeço ao grupo de estudos composto pelos alunos Alex Borges, Elisa Tuler, Guilherme Trielli, Leonardo Chaves, Marcelo Maia, Renata Braga, Bruno Coutinho, entre outros pela ajuda e discussões sobre os resultados e decisões de implementação que contribuíram positivamente para um melhor entendimento de todo trabalho.

Referências

[1] A. Apostolico. String editing and longest common subsequences. HANDBOOK OF FORMAL LANGUAGES, 1997.

[2] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. In *Proceedings of the 12th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 168–175. ACM Press, 1989.

[3] Timothy C. Bell, Alistair Moffat, Ian H. Witten, and Justin Zobel. The mg retrieval system: compressing for space and speed. *Communications of the ACM*, 38(4):41–42, 1995.

[4] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[5] Edleno Silva de Moura, Gonzalo Navarro, Nivio Ziviani, and Ricardo Baeza-Yates. Fast searching on compressed text allowing errors. In *SIGIR '98: Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 298–306, New York, NY, USA, 1998. ACM Press.

[6] Edleno Silva de Moura, Gonzalo Navarro, Nivio Ziviani, and Ricardo Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS)*, 18(2):113–139, 2000.

[7] Dan Gusfield. *Algorithms on Strings,Trees and Sequences.* Cambridge University Press, 1997.

[8] R.N. Horspool. Practical fast searching in strings. In *Soft.-Prac. and Exp.*, volume 10, 1980.

[9] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40:1098–1101, September 1952.

[10] Jyrki Katajainen, Alistair Moffat, and Andrew Turpin. A fast and space - economical algorithm for length - limited coding. In *ISAAC '95: Proceedings of the 6th International Symposium on Algorithms and Computation*, pages 12–21, London, UK, 1995. Springer-Verlag.

[11] Vladimir Levenshtein. Edit distance, 2006. http://pt.wikipedia.org/wiki/Dist

[12] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Technical Report 8, 1966.

[13] Sérgio Haruo Nakanishi. Proof of huffman, 2006. http://www.ime.usp.br/ leo/mac323/03-1/material/huffman/huffman-demonstracao1.html.

[14] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, 2003.

[15] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[16] Eugene S. Schwartz and Bruce Kallick. Generating a canonical prefix encoding. *Communications of the ACM*, 7(3):166–169, 1964.

[17] P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *Journal of Algorithms*, 1(1):359–373, 1980.

[18] D.M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, 1990.

[19] Wikipedia. Huffman coding — wikipedia, the free encyclopedia, 2006. [Online; accessed 20-May-2006].

[20] Wikipedia. Huffman coding — wikipedia, the free encyclopedia, 2006. [Online; accessed 20-May-2006].

[21] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proceedings USENIX Winter 1992 Technical Conference*, pages 153–162, San Francisco, CA, 1992.

[22] Sun Wu and Udi Manber. Fast text searching: allowing errors. *Commun. ACM*, 35(10):83–91, 1992.

[23] Nivio Ziviani. *Projeto de Algoritmos com Implementações PASCAL e C*. Pioneira Thomson Learning, 2004.

[24] Nivio Ziviani, Edleno Silva de Moura, Gonzalo Navarro, and Ricardo Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, 2000.

A Apêndice A - Prova de Correção

Apresentamos aqui a prova de correção da transformação do segmento inicial M(1...i) em N(1...j) usando o mínimo de operações c(i,j).custo. Essa prova segue a apresentada em [11, 12].

1. É inicialmente verdadeira na linha e colunas 0 porque M[1..i] pode ser transformado num string vazio N[1..0] simplesmente apagando todos os i caracteres. Do mesmo modo, podemos transformar N[1..0] em M[1..j] simplesmente adicionando todos os caracteres j.

2. O mínimo é tomado em três distâncias, sendo em qualquer das quais possível que:

   (a) Se podemos transformar M[1..i] em N[1..j-1] em k operações, então nós podemos simplesmente adicionar N[j] depois para obter N[1..j] em k+1 operações.

   (b) Se podemos transformar M[1..i-1] em N[1..j] em k operações, então nós podemos fazer as mesmas operações em M[1..i] e depois remover o M[i] original ao fim de k+1 operações.

   (c) Se podemos transformar M[1..i-1] em N[1..j-1] em k operações, então podemos fazer o mesmo com M[1..i] e depois fazer uma substituição de N[j] pelas M[i] originais no final, se necessário, requerendo k+custo operações.

3. As operações requeridas para transformar M[1..n] em N[1..m] é o número necessário para transformar todos os M em todos os N, e logo C[n,m] contém o nosso resultado desejado.

Esta prova não confirma que o número colocado em c[i,j] seja de fato o mínimo; isso é mais difícil de provar e exige um argumento *Reductio ad absurdum* no qual assumimos que C[i,j] é menor do que o mínimo dos três, e usamos isto para mostrar que um dos três não é mínimo. Em [7] existe uma prova detalhada dessa solução, a qual apresentaremos a seguir de forma bastante sucinta.

Lema 1. *O valor de* $C(i,j)$ *deve ser* $C(i-1,j)+1$, $C(i,j-1)+1$ *ou* $C(i-1,j-1)+custo(i,j)$ *e não há outra possibilidade.*

Demonstração. Considere um corretor de edição para a transformação de $M[1..i]$ a $N[1..j]$ por meio do menor número de operações de edição e mantenha atenção no último símbolo. Esse último símbolo pode ser I (inserção), D (deleção), S (substituição) ou N (quando ocorre um casamento, nada a fazer). Se esse último símbolo for um I, então a última operação é a inserção do caractere $N(j)$ no final da primeira string de transformação M. Assim, o corretor antes de I deve especificar o número mínimo de operações para transformar $M[1..i]$ em $N[1..j-1]$. Se não acontecer essa especificação, então a transformação de $M[1..i]$ em $N[1..j]$ será maior que o número de operações mínimo. Por definição, até a penúltima transformação foram necessárias $C(i,j-1)$ operações de edição. Isso faz com que, se o último símbolo do corretor for um I, então $C(i,j) = C(i,j-1)+1$. Da mesma forma, se o último símbolo do corretor foi um D, significa que a última operação foi uma

remoção de $M(i)$ e os símbolos do corretor que estão à esquerda de D devem especificar o número de operações mínimo para que possamos transformar $M[1..i-1]$ em $N[1..j]$. Assim, por definição, até a penúltima transformação foram necessárias $C(i-1,j)$ operações de edição, ou seja, se o último símbolo do corretor for D, então $C(i,j) = C(i-1,j)+1$. Mas se o último símbolo do corretor for um S, temos então que a última operação de edição deve substituir $M(i)$ por $N(j)$, e os símbolos à esquerda de S especificam assim, o número mínimo de operações de edição que são necessárias para $M[1..i-1]$ se transforme em $N[1..j-1]$, ou seja, $C(i,j) = C(i-1,j-1)+1$. E por fim, se o último símbolo do corretor for um N, temos que $M(i) = N(j)$ e $C(i,j) = C(i-1,j)$. Utilizando a variável de custo $custo(i,j)$, a qual é 0 se $M(i) = N(j)$ e 1 se $M(i) = N(j)$. Assim temos que se o último símbolo do corretor for um N ou um S, então $C(i,j) = C(i-1,j-1) + custo(i,j)$. Dessa forma cobrimos todos os casos do lema. □

Lema 2. *Temos que* $C(i,j) \leq min[C(i-1,j)+1, C(i,j-1)+1, C(i-1,j-1)+custo(i,j)]$.

Demonstração. Primeiramente temos que $M[1..i]$ pode ser transformada em $N[1..j]$ com $C(i,j-1)+1$ operações. No entanto, é necessário apenas transformar $M[1..i]$ em $N[1..j-1]$ com o número mínimo de operações para que possamos utilizar mais uma operação para inserir o caractere $N(j)$ no final. O número de operações que faz essa transformação de M em N é exatamente $C(i-1,j)+1$ operações. É possível também transformar $M[1..i-1]$ em $N[1..j]$ com o menor número de operações para em seguida removermos o caractere $M(i)$. Assim o número de operações desta transformação é igual a $C(i-1,j)+1$. Por último podemos fazer a transformação com $C(i-1,j-1)+custo(i,j)$ operações, utilizando o mesmo argumento descrito acima. □

Assim, a partir desses dois lemas, é possível prova o seguinte teorema:

Teorema 1. *Se i e j são ambos positivos, então* $C(i,j) = min[C(i-1,j)+1, C(i,j-1)+1, C(i-1,j-1)+custo(i,j)]$

Demonstração. O Lema 1 diz que $C(i,j)$ precisa ser igual a um dos valores $C(i-1,j)+1, C(i,j-1)+1, C(i-1,j-1)+custo(i,j)$. Já o lema 2 diz que $C(i,j)$ deve ser menor ou igual ao menor a um desses valores. Assim, $C(i,j)$ precisa ser igual ao menor valor dentre os três valores possíveis. □

Ressaltamos que essa é uma descrição sucinta da prova e que maiores detalhes para a mesma pode ser encontrada em [7].

B Apêndice B - Árvore Huffman Ótima

```
DEFINIÇÕES
profundidade(ni)-o tamanho do caminho do nó ni até a raiz da árvore.
peso(ni)        - se ni for uma folha, o peso de ni é a freqüência
com que aparece a informação (letra) da folha ni, caso não seja uma
folha o peso de ni é a soma dos pesos dos seus filhos.

WPL(T) = somatória de (peso(ni) * profundidade(ni)), para cada FOLHA
ni de uma árvore binária T (weighted path length)
WPL(T) = função custo dado em aula.

Árvore binária completa (ou cheia) - árvore binária onde todos seus
nós tem 2 filhos, exceto as folhas que não tem filho.

LEMA
Seja qualquer árvore binária completa (ou cheia) com seus pesos
associados às suas folhas. Suponha que n1 e n2 sejam as folhas com
os menores pesos. Então podemos construir uma árvore T´ a partir de
T tal que:
1. T´ possue as mesmas folhas que T exceto por n1 e n2, e T´ tem uma
nova folha n3.
2. O peso de n3 é a soma de n1 e n2. O peso das outras folhas não
mudam.
3. WPL(T´) <= WPL(T) - peso(n3).
4. WPL(T´) = WPL(T) - peso(n3), se n1 e n2 são irmãos em T.

Agora pra provar que Huffman é ótimo:

Seja T a árvore codificadora construída pelo algoritmo de Huffman
a partir de um conjunto de nós (folhas).

Se X é qualquer outra árvore codificadora com os mesmos nós,
então  WPL(T) <= WPL(X).

A prova é feita por indução sobre o número de folhas de T:

Caso base: T tem apenas duas folhas. A prova é trivial.
Caso geral:
 Seja n1 e n2 os dois primeiros nós escolhidos pelo algoritmo de
 Huffman na construção de T. Aplique o lema em T e X para produzir
 T´ e X´. Como T foi construída pelo algoritmo de Huffman, n1 e n2
 são  necessariamente irmãos em T.
 Então:
   (1) WPL(T´) = WPL(T) - peso(n1) - peso(n2).
 O Lema também garante que:
   (2) WPL(X´) <= WPL(X) - peso(n1) - peso(n2).
```

```
Usando a indução, podemos afirmar que:
  (3) WPL(T´) <= WPL(X´)
Pois:
  . T´ e X´ tem um nó a menos que T. (Lembrando indução: supondo
    que para n elementos é verdadeiro, provar que para  n+1
    elementos é verdadeiro.
    T tem n+1 elementos.)
  . T´ é equivalente à árvore construída pelo algoritmo de Huffman
  usando as Folhas de T´.
  . T´ e X´ tem o mesmo número de folhas.
Essas três inequações (1,2 e 3)combinadas nos permite concluir que:

  WPL(T) <= WPL(X)

O que completa nossa prova.
```