Aplicação e análise de algoritmos para casamento de cadeia de caracteres

Matheus Henrique, Antônio Carlos

Dezembro, 2022

1 Introdução

O casamento de cadeia de caracteres influenciou muitas pesquisas na ciência da computação e exerce papel importante em vários problemas no mundo real [1]. Segundo a ONU existem cerca de 5,3 bilhões de usuários de internet no mundo [2], realizando trocas de e-mail, *tweets* (publicações de até 280 caracteres no microblog twitter.com), publicações em blogs e criando bibliotecas digitais.

Com essa explosão de informação textual disponível on-line, após a chegada da web 2.0, o estudo de mecanismos para processamento desse tipo de dado se tornou ainda mais necessário, pois tais mecanismos são tarefa base para diversas aplicações, algumas delas descritas a seguir.

- Análise de sentimento: consiste em extrair conteúdo subjetivo expresso em dados textuais [3]. Utilizada amplamente durante eventos como eleições presidenciais, *realitys shows* e lançamentos de séries, para entender o sentimento do público em relação ao evento.
- Detecção de intrusão de sistemas: pacotes de dados que contenham palavras-chave relacionadas à intrusão são encontrados com a aplicação de estratégias de correspondência de strings.
- Bioinformática e sequenciamento de DNA: Cada uma das bases do DNA é composta por uma letra, logo, uma sequência de DNA é representada por um texto.
- Detecção de plágio: comparar textos e detectar as semelhanças entre eles. Com base nessas semelhanças é possível identificar se é um trabalho original ou não.

No entanto, muito antes do uso nessas aplicações, algoritmos para casamento de padrões já eram alvo de estudo, dentre eles: Distância de Levenshtein, Shift-And e o BMH. Apresentaremos a distância de Levenshtein e o Shift-And para casamento aproximado de caracteres. O BMH será apresentado para casamento exato aplicado em arquivos comprimido e não comprimido. Será utilizada a compressão de Huffman orientada a bytes, descrita por Ziviani[4]. Na modelagem serão feitas as definições para o casamento aproximado, casamento exato, texto, padrão e outras.

2 Formato de entrada e saída

A entrada é feita com arquivos .txt. O arquivo texto.txt deve conter o texto a ser pesquisado. O arquivo padrão.txt contém 1 ou mais padrões, com cada padrão em uma linha, que serão pesquisados no texto. Para o algoritmo de compressão de Huffman também é necessário utilizar um arquivo alfabeto.txt com todos os símbolos do alfabeto que compõe o texto a comprimido, sem separadores e em uma única linha.

A saída de cada algoritmo será escrita no arquivo "nomedoalgoritmo.out"(Ex.: BMH.out). A saída conterá cada padrão procurado seguido de uma lista de ocorrências do padrão. Ex.:

Texto: Estou animado para mineração de dados no próximo semestre, será um semestre desafiador.
Padrão: semestre
dados
Saída: semestre 50 68
dados 33
Instruções de uso estão detalhadas no arquivo READ.ME.

3 Modelagem

Uma cadeia é uma sequência de elementos denominados caracteres. O problema do casamento de cadeia de caracteres (por vezes, também utilizaremos o termo casamento de padrão) vem de uma ideia bem simples.

Os elementos que formam a cadeia pertencem a um alfabeto finito Σ.

Ex.: $\Sigma = \{a, b, c, ..., z\}$.

Em um vetor $T[1..n]$ que representa o texto e tem tamanho n, queremos encontrar todas as ocorrências do vetor $P[1..m]$, que representa o padrão, e tem tamanho m, tal que $m \leq n$. De forma resumida queremos encontrar as ocorrências de P em T.

O casamento aproximado de cadeias, é o problema de encontrar P em T com um número limitado de operações (erros) k.

A quantidade k de operações é conhecida na literatura como distância de edição, ou distância de Levenshtein, falaremos mais dela nas próximas secções.

Os tipos de operações são inserção, substituição ou retirada. A operação de inserção refere-se a quando um elemento pode ser inserido no padrão e ainda sim realizar o casamento, na de retirada um elemento pode ser retirado do padrão e na de substituição um elemento pode ser substituído.

Ex.: Para $P = \{mineração\}$ um casamento com inserção seria "mineiração"(inserção da letra i), um casamento com substituição seria "maneração"(substituição da letra i pela letra a) e um casamento com remoção seria "mineação"(remoção da letra r).

Só faz sentido tratar um casamento aproximado para valores de que k que respeitem $0 < k < m$, pois no caso de $k = m$ toda subcadeia de comprimento m pode ser convertida em P. Para $k = 0$ temos um casamento exato.

O casamento exato é simples como o nome, ocorre sempre que há uma ocorrência em T exatamente igual P, ou seja $k = 0$.

4 Estrutura de dados e arquivos teste

O texto do livro O senhor dos anéis - o retorno do rei (utilizaremos a sigla TLOTR), contém 169.133 palavras e 988.483 caracteres. Seu vocabulário é composto por 14.876 palavras únicas.

O texto da Constituição Federal (utlizaremos a sigla CF) Brasileira de 1988 possui, retirado o sumário, 225.013 palavras e 1.346.903 caracteres. Seu vocabulário é composto de 10.149 palavras únicas.

O texto do relatório final da CPI da Covid (utilizaremos a sigla CPI, o arquivo está disponível no site do senado federal) contém 298.174 palavras e 1.917.792 de caracteres. Seu vocabulário é composto por 23.925 palavras únicas.

Esses 3 textos serão nossos arquivos testes para realização dos experimentos.

Com base nessas informações montamos a estrutura de dados para os algoritmos.

```
const
    MaxTamTexto = 2000000;
    MaxTamPadrao = 50;
    MaxChar = 256;
type
    TipoTexto = array[1..MaxTamTexto] of char;
    TipoPadrao = array[1..MaxTamPadrao] of char;
```

O alfabeto Σ utilizado são os caracteres da tabela ASCII. Serão utilizados 10 palavras padrões, com 10 tamanhos diferentes.

5 Algoritmos para casamento aproximado

5.1 Distância de Levenshtein

A distância de Levenshtein ou distância de edição refere-se ao menor número de operações para transformar uma cadeia de caracteres x em uma cadeia de caracteres y, definida então por $ed(x, y)$.

Ex.: $ed(infeliz, felizes) = 4$, pois são duas operações de remoção no prefixo *in* e 2 operações de inserção no sufixo *es*.

Portanto, para o problema de casamento aproximado de cadeias utilizando a distância de Levenshtein, queremos encontrar as ocorrências de P em T, que satisfaçam $ed(P, T') \leq k$ tal que T' é uma instância de T.

O cálculo da distância de Levenshtein pode ser realizado utilizando o paradigma da programação dinâmica.

Um pré-requisito para aplicar a programação dinâmica, para resolver um problema, é que a solução do problema completo possa ser obtida a partir das soluções dos seus subproblemas.

De forma um tanto simples e intuitiva podemos considerar as *strings* (cadeias de caracteres) a (padrão) e b (*string* do texto) até o último caractere:

Se os últimos caracteres de ambas as strings forem iguais, a distância de edição será igual à distância de edição das mesmas duas strings, até o penúltimo caractere. No entanto, se o último caractere for diferente, a distância de edição é igual ao custo mínimo de inserção, exclusão ou substituição até o último caractere da *string* a. Vamos formalizar essa ideia.

$$lev_{a,b}(i,j) = \begin{cases} max(i,j), & \text{se } min(i,j) = 0 \\ min \begin{cases} lev_{a,b}(i-1,j) + 1 \\ lev_{a,b}(i,j-1) + 1 \\ lev_{a,b}(i-1,j-1) + 1 \end{cases} & \text{caso contrário} \end{cases}$$

Considere as *strings* $a = \{sitting\}$ e $b = \{kitten\}$. Considere também $i = 1$ e $j = 1$ como posições do primeiro caractere de cada *string*, variando até m e n respectivamente.

Montamos uma matriz mxn que represente as *strings* e suas posições. Começamos a preencher a matriz da esquerda para a direita e de cima para baixo, aplicando a fórmula para casa posição (i, j).

Com o preenchimento completo, teremos na posição (m, n) a distância de edição necessária para transformar a em b.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 2 & 3 & 4 & 5 \\ 4 & 3 & 2 & 1 & 2 & 3 \\ 5 & 4 & 3 & 2 & 2 & 3 \\ 6 & 5 & 4 & 3 & 3 & 2 \\ 7 & 6 & 5 & 4 & 4 & 3 \end{bmatrix}$$

Matriz para transformação de a em b. $ed(a, b) = 3$

A distância de edição não resolve o problema do casamento aproximado por si só. É necessário que ela satisfaça a desigualdade $ed(a, b) \leq k$. Apresentamos a seguir um algoritmo que utiliza a distância de edição para resolver o problema do casamento aproximado. A ideia é bem simples, comparar P com uma palavra de T, salvando as posições em que casamentos aconteceram.

Pseudocódigo para distância de Levenshtein:

```
function LevenshteinDistance(TipoPadrao s, TipoTexto t):
  for i = 1 to m: d[i, 0] = i;
  for j = 1 to n: d[0, j] = j;

  for j = 1 to n:
    for i = 1 to m:
      if s[i] = t[j]: Cost = 0;
      else: Cost = 1;
```

```
      d[i, j] = min(d[i-1, j] + 1, d[i, j-1] + 1, d[i-1, j-1] + Cost);

  return d[m, n]
endfunction


function ApproximateMatching(TipoPadrao s, TipoTexto t)
    int distance;

    while(t != EOF)
        distance = LevenshteinDistance(s, palavra extraida do texto);

    if distance <= k return 1;
    else return 0;
endfunction
```

Essa é uma versão iterativa e portanto mais eficiente da distância de Levenshtein recursiva discutida anteriormente. Ela funciona para maioria dos casos em que é necessário casar cadeias e tem fácil implementação, o que é uma boa vantagem. No entanto, além de lenta, existem casos particulares em que ela não alcança o resultado esperado, como no exemplo a seguir.

Considere $P = \{\text{teste}\}$, $T = \{\text{os testes testam estes alunos}\}$ e $k = 1$; Repare que existe um casamento com 1 erro de inserção a partir da posição 7 do texto, permitindo o padrão {tes te}, que a distância de edição não reconheceria. Para esse tipo de caso, o algoritmo Shift-And para casamento aproximado é ideal e inteligente.

5.2 Shift-And aproximado

O Shift-And é um algoritmo baseado em um automato finito não determinista que utiliza o conceito de paralelismo de *bit* para realizar casamentos. A ideia é pré-processar o padrão criando máscaras de *bits* para cada elemento. A posição em que determinado caractere do alfabeto ocorre no padrão recebe valor 1. Para o padrão $P = \{\text{dados}\}$ temos $M[d] = 10100$, $M[a] = 01000$, $M[o] = 00010$ e $M[s] = 00001$. Repare que o caractere d aparece nas posições 1 e 3 do padrão. M é uma tabela que armazena as máscaras.

Em seguida, o algoritmo mantém um conjunto R de todos os prefixos de P que casam com caracteres já lidos do texto, atualizando cada conjunto com o paralelismo de bit caractere por caractere. No inicio, $R = 0^m$ (m é o tamanho do padrão e 0^m é a notação, definida por Navarro e Raffinot [5], para 0 repetido m vezes, $01^3 = 0111$ por exemplo). Para cada novo caractere lido no texto o valor R' é atualizado seguindo a fórmula:

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

A operação $>>$ (move os *bits* para a esquerda e entra com zeros a direita) garante que a posição $j + 1$ de R' só ficará ativa (conceito que vem da representação do algoritmo como um automato não determinista) caso a posição j estivesse ativa em R. A operação $|$ *or* permite que um casamento comece em qualquer posição do texto.

A ideia é sempre que um elemento do texto case com um elemento do padrão essa posição receba 1, caso contrário 0. Um casamento acontece quando o bit mais a direita de R' é ativado.

O algoritmo descrito acima é algoritmo Shift-And para casamento exato proposto por Baeza-Yates e Gonnet em 1989. O algoritmo Shift-And para casamento aproximado foi proposto em seguida por Wu e Manbem em 1992 e utiliza conceitos do casamento exato.

No casamento aproximado, R e R' do casamento exato passam a ser identificados por R_0 e R'_0, sua fórmula se mantém. Adicionamos um novo novo valor R_j para cada j tal que $0 < j \leq k$. Dessa forma além do parelelismo do casamento exato, realizamos outras k operações para o casamento aproximado. A cada novo caractere lido do texto todas as operações entre as $k + 1$ máscaras de bits são realizadas simultaneamente.

Para um casamento aproximado com k = 1 temos:

$$R'_0 = ((R_0 >> 1)|10^{m-1})\&M[T[i]] \text{ e}$$
$$R'_1 = ((R_1 >> 1)\&M[T[i]])|R_0|(R_0 >> 1)|(R'_0 >> 1)|10^{m-1}$$

M é a tabela que contém as máscaras de *bits*, R_0 indica um erro de inserção, $R_0 >> 1$ indica um erro de substituição e $R'_0 >> 1$ indica um erro de remoção.

Pseudocódigo para Shift-And:

```
function Shift-And-Aproximado (TipoPadrao p, TipoTexto t);
    { Pre processamento }
    foreach c in alphabet do M[c] := 0(m times);
    for j = 1 to m do M[p[j]] = M[p[j]] | 0(j-1 times)10(m - j times);

    { Pesquisa }
    for j = 0 to k do R[j] = 1(j times)0(m-j times);
    for i = 1 to n do
        Rant = R0;
        Rnovo := ((Rant >> 1) | 10(m1 times) & M[T[i]];
        R0 := Rnovo;
    for j := 1 to k do
        Rnovo := ((Rj >> 1 & M[T[i]]) | Rant | ((Rant | Rnovo) >> 1);
        Rant := Rj ;
        Rj := Rnovo;
        if Rnovo & 0(m1 times)1 is different 0(m times) then 'Casamento na posicao i-m';
endfunction
```

O algoritmo Shift-And resolve a falha da distância de edição apresentada anteriormente. Os melhores algoritmos para casamento aproximado de cadeias utilizam paralelismo de *bit*.

6 Algoritmos para casamento exato

6.1 BMH

O algoritmo Boyer-Moore-Horspool tem implementação muito simples e eficiência comprovada, por esses motivos ele é o algoritmo ideal para aplicações de uso geral para problema do casamento exato.

Sua eficiência vem da técnica de deslocar P ao longo de T de forma a evitar comparações desnecessárias. O algoritmo faz comparações da direita para a esquerda, procurando um sufixo em T que também seja sufixo em P. Se ele encontra esse sufixo, segue comparando até realizar o casamento, caso contrário ele sabe exatamente quantos caracteres pode deslocar, através da sua tabela de deslocamento.

O pré-processamento do padrão é computar a tabela de deslocamento. O algoritmo atribui m para todos os elementos pertencentes a Σ. Feito isso, para os $m-1$ primeiros caracteres do padrão, é atribuído:

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

De forma prática, dado um $P[1..m]$, para cada caractere na posição P_j, tal que $1 \leq j < m$, atribui-se $m-j$ no seu valor de deslocamento. Para os caracteres que pertencem a Σ mas não pertencem a P, atribui-se m. Ex.: $P = \{dados\}$ tem tabela de deslocamento:

Tabela 1: Deslocamento de $P = \{\text{dados}\}$, $m = 5$

i	$d[x]$	valor
1	d[d]	2
2	d[a]	3
3	d[o]	1
#	d[#]	5

O símbolo # representa qualquer outro valor pertencente a Σ.

Realizado o pré-processamento, o algoritmo começa a realizar os casamentos, iniciando as comparações sempre do caractere mais a direita do padrão para o caractere mais a esquerda. Se há casamento, o caractere anterior ao casamento é comparado, se não há, verifica-se o caractere do texto em que ocorreu o conflito, consulta a tabela de deslocamento e desloca, iniciando uma nova comparação no ponto em que parou.

Ex.: $P = \{\text{dados}\}$, $T = \{\text{quero minerar dados}\}$.

```
quero minerar dados
dados

-compara "s" com "o"
-conflito no caractere "o", d[o] = 5

quero minerar dados
     dados

-compara "s" com "e"
-conflito no caractere "e", d[e] = 5

quero minerar dados
              dados

-compara "s" com "s"
-compara "o" com "o"
-compara "d" com "d"
-compara "a" com "a"
-compara "d" com "d"
-casamento realizado
```

Pseudocódigo para BMH:

```
function BMH(TipoTexto T, int n, TipoPadrao P, int m);
    int i , j , k;
    int d[MaxChar];

    {Pre processamento do padrao}
    for j = 0 to MaxChar do d[j] = m;
    for j = 1 to m - 1 do d[P[j]) ] = m - j;
    i = m;

    {Pesquisa}
    while i <= n do
        k := i ; j := m;

        while (T[k] = P[j] & j > 0) do
            k--;
            j--;

        if j = 0 then 'Casamento na posicao: ' , k+1);
        i += d[T[i]];
        end while;
    end while;
endfunction
```

6.2 BMH em arquivo comprimido

6.2.1 Compressão

Menos espaço para armazenamento e pesquisa rápida e eficiente, a técnica de compressão é um caso raro de vencer-vencer. É preciso realizar a compressão para lidar com o grande volume de informação textual que relatamos na secção introdutória. Um dos métodos de compressão mais conhecidos é o método de Huffman.

O método de Huffman baseado em palavras é o método mais eficaz para textos em linguagem natural. Ele considera palavras distintas em um texto como símbolos e atribui frequências a eles de acordo com a ocorrência de cada palavra. Em seguida, comprime o texto substituindo as palavras por códigos.

A codificação é realizada em uma abordagem gulosa que constrói uma árvore binária partindo das folhas até a raiz. A construção da árvore é feita combinando as menores frequências de um texto em uma nova sub-árvore, a cada iteração, e atribuindo a soma de suas frequências ao nó raiz da sub-árvore formada. Ao final das combinações, obtém-se uma árvore binária com um código em bits para cada palavra.

Moura, Navarro, Ziviani e Baeza-Yates alteraram a proposta binária do método de Huffman original para uma associação de *bytes* a cada palavra do texto. Utilizando 7 dos 8 *bits* de cada *byte* para codificação, eles criaram uma árvore com grau 128 em que o oitavo *bit* é utilizado para marcar o primeiro *byte* do código da palavra. Esse é o Huffman com marcação implementado nesse trabalho.

A vantagem desse código é que com ele é possível realizar busca de padrões no texto comprimido como qualquer busca em um texto não comprimido. É necessário apenas comprimir o padrão e realizar a busca, como veremos a seguir.

6.2.2 Busca

A busca em um arquivo comprimido segue passos bem simples. Inicialmente é necessário realizar uma busca de P no vocabulário do texto comprimido. Caso P não exista, a palavra não existe no texto, caso contrário a busca segue. Em seguida, o código de Huffman por marcação é obtido para P e esse código é pesquisado no arquivo comprimido utlizando o BMH. Pseudocódigo para BMH em arquivo comprimido:

```
function BMH(TipoTexto T, int n, TipoPadrao P, int m);
    int i , j , k;
    int d[MaxChar];

    {Pre processamento do padrao}
    for j = 0 to MaxChar do d[j] = m;
    for j = 1 to m - 1 do d[P[j] + 128) ] = m - j;
    i = m;

    while i <= n do {Pesquisa}
        k := i ; j := m;

        while (T[k] = P[j] & j > 0) do
            k--; j--;

        if j = 0 then 'Casamento na posicao: ' , k+1);
        i += d[T[i] + 128];
        end while;
    end while;
endfunction
```

A única mudança em relação ao BMH original está na base numérica do algoritmo adicionando o grau 128 da árvore de bytes.

Pseudocódigo para busca:

```
function Busca(ArqComprimido)
    Texto = open(ArqComprimido);
    if(Padrao exists in Vocabulario) then
        Code = Encode(padrao);
        Atribui(Padrao, Code);
        BMH(Texto, n, Padrao, m);
    else 'Padrao nao existe no texto';
endfunction;
```

O procedimento Atribui preenche P com os *bytes* de Code.

7 Análise de complexidade

7.1 Complexidade de tempo

Tabela 2: Complexidade de tempo

Algoritmo	Caso médio	Pior caso
Distância de edição	$O(m * n)$	$O(m * n)$
Shift-And aproximado	$O(k * n)$	$O(k * \lceil m/w \rceil * n)$
Boyer–Moore–Horspool	$O(n/m)$	$O(n * m)$

7.2 Complexidade de espaço

Tabela 3: Complexidade de espaço

Algoritmo	Complexidade
Distância de edição	$O(m * n)$
Shift-And aproximado	$O(k * m)$
Boyer–Moore–Horspool	$O(c)$

7.3 Discussão

O número de operações que o Shit-And realiza pode ser reduzido em um fator de até w em que w é o número de bits da palavra no computador.

É possível perceber a superioridade do algoritmo Shift-And comparada a distância de edição, de fato, enquanto o a distância de edição realiza comparações simples $m * n$ vezes, palavra por palavra do texto, o autômato não determinista permite que mais de um estado esteja ativo em um determinado instante, isto é, o Shift-And consegue realizar casamentos simultâneos em diversas posições do texto. Essa propriedade torna o algoritmo ideal para tarefas que necessitem de um casamento aproximado.

Além da superioridade de tempo, o Shift-And também leva vantagem em espaço, sua complexidade de espaço é $O(k * m)$ enquanto a distância de edição percorre o padrão por todo o texto com complexidade $O(n * m)$.

O sucesso do casamento exato utilizando BMH vem da sua complexidade para o caso médio, um resultado excelente pois executa em tempo sublinear. Sua complexidade de espaço é dada durante o pré-processamento do padrão onde c é o tamanho do alfabeto, a complexidade de tempo nessa estapa também é $O(c)$. A seguir, discutiremos a eficiência de tempo e espaço para esse algoritmo, aplicado em

arquivos comprimidos e não comprimidos, além do número de comparações para todos os algoritmos discutidos.

8 Análise dos resultados experimentais

As métricas usadas nos experimentos realizados foram o tempo de relógio, com o auxílio da função $gettimeofday$, e número de comparações. Computamos as comparações utilizando uma variável do tipo inteiro chamada *comparacoes*, inicializada com 0. A cada comparação entre texto e padrão ela é incrementada em 1 unidade, ao final do programa o número de comparações é impresso na tela. Para o algoritmo Shift-And é necessário computar as comparações para erros, então o cálculo para esse algoritmo é realizado com $comparacoes += k + 1$.

Os testes foram realizados em uma máquina com processador AMD Ryzen 5 2500U, memória RAM de 12GB e SSD de 128GB.

8.1 Compressão dos arquivos

Apresentamos os resultados da compressão dos arquivos de teste que serão utilizados para realizar buscas. A compressão foi feita utilizando o Huffman por marcação. Avaliamos a razão de compressão e o tempo para comprimir.

Tabela 4: Compressão de arquivos de texto

Arquivo	Original	Comprimido	Taxa
CF	1.440.714 bytes	535.263 bytes	37,15%
CPI	2.012.182 bytes	974.799 bytes	48,44%
TLOTR	1.030.528 bytes	528.219 bytes	51,25%

Os resultados das razões de compressão foram surpreendentes considerando a expectativa de 25% proposta por Ziviani (2000). Em especial, para o menor arquivo teste, TLOTR apresentou uma taxa maior que 50%. A média de compressão foi de 45,61%.

Contabilizamos também o tempo de relógio para a compressão. Realizamos uma rodada de 10 execuções e computamos a média.

Tabela 5: Tempo de compressão

Arquivo	Tempo
CF	3,518 segundos
CPI	4,994 segundos
TLOTR	2,962 segundos

Conforme o esperado, os tempos de execução respeitaram a hierarquia dos arquivos quanto ao número palavras únicas. A primeira vista podem parecer tempos grandes considerando a capacidade computacional que possuímos hoje, no entanto, a economia de espaço que vimos anteriormente e a velocidade de busca que veremos a seguir, compensam esse tempo de compressão.

8.2 Análise do casamento aproximado

Variamos n e k de tal forma que $5 \leq n \leq 20$ e $0 \leq k \leq 3$. Utilizamos o arquivo CPI para realizar os testes. Os padrões utilizados foram: minas (5), cavalgando (10), esporadicamente (15) e incompreensivelmente (20). Os resultados estão a seguir.

Podemos observar nos testes (Figura 1) que a distância de edição tem uma relação de crescimento linear com o tamanho do padrão. No entanto, o número de erros não afeta em nada suas comparações

e tem impacto pouco significativo no tempo de execução, os tempos se mantém por volta de 1 e 2 segundos.

O que observamos no algoritmo Shift-And é que o tamanho do padrão não interfere em nada nas suas comparações ou no seu tempo de execução, o número de comparações é o mesmo e tempo não sofre alterações significativas. No entanto, suas comparações aumentam quando o número de erros aumenta.

Ambos os resultados são apoiados pela complexidade dos algoritmos. A distância de edição tem complexidade $O(m * n)$ tal que, como demonstrado, m altera as métricas do algoritmo, mas k não afeta em nada. O Shift-And tem complexidade $O(k * n)$ tal que m não altera suas métricas, porém k sim.

No comparativo entre os algoritmos, é fácil perceber a ampla eficiência do Shift-And em relacao a distância de edição. Os melhores resultados para a distância de edição ainda estão longe da eficiência do Shift-And.

Optamos por não utilizar os arquivos menores que o CPI nos testes documentados, no entanto, ao analisar a complexidade dos algoritmos é possível perceber o tamanho de n altera as métricas de ambos.

k = 0					
Distância de edição			Shift-And		
Tamanho de P	Comparações	Tempo	Tamanho de P	Comparações	Tempo
5	11.570.538	1,785234 sec	5	2.012.182	0,033562 sec
10	31.405.746	2,280084 sec	10	2.012.182	0,304744 sec
15	59.505.624	2,418381 sec	15	2.012.182	0,0311333 sec
20	95.870.172	2,992949 sec	20	2.012.182	0,324690 sec
k = 1					
Distância de edição			Shift-And		
Tamanho de P	Comparações	Tempo	Tamanho de P	Comparações	Tempo
5	11.570.538	2,151154 sec	5	4.024.364	0,035629 sec
10	31.405.746	2,819691 sec	10	4.024.364	0,031922 sec
15	59.505.624	3,116365 sec	15	4.024.364	0,030772 sec
20	95.870.172	3,302756 sec	20	4.024.364	0,037914 sec
k = 2					
Distância de edição			Shift-And		
Tamanho de P	Comparações	Tempo	Tamanho de P	Comparações	Tempo
5	11.570.538	1,97136 sec	5	6.036.546	0,058601 sec
10	31.405.746	2,170531 sec	10	6.036.546	0,042752 sec
15	59.505.624	2,595575 sec	15	6.036.546	0,037102 sec
20	95.870.172	3,013340 sec	20	6.036.546	0,033781 sec
k = 3					
Distância de edição			Shift-And		
Tamanho de P	Comparações	Tempo (s)	Tamanho de P	Comparações	Tempo
5	11.570.538	1,937738 sec	5	8.048.728	0,068843 sec
10	31.405.746	1,965333 sec	10	8.048.728	0,056013 sec
15	59.505.624	2,589589 sec	15	8.048.728	0,605300 sec
20	95.870.172	2,724885 sec	20	8.048.728	0,058718 sec

Figura 1: Testes experimentais para os algoritmos de busca aproximada.

8.3 Análise do casamento exato

Para a análise do casamento exato, variamos o valor de n e m nos testes. Utilizamos os mesmos tamanhos para n da secção anterior e para m utilizamos os 3 arquivos, TLOTR, CF e CPI (ordem de tamanho). As buscas foram realizadas no arquivo original e no arquivo comprimido. Os resultados dos testes estão na figura 2.

Para a pesquisa em arquivo não comprimido, conforme o esperado, na medida em que o tamanho do padrão cresce, as comparações tendem a diminuir. Isso se dá pois quanto maior o padrão maior o deslocamento possível a ser realizado. O tempo de execução acompanha essa relação, diminuindo enquanto o padrão aumenta. Quanto ao texto, podemos perceber que o tempo de execução e o número de comparações aumentam para textos maiores.

Esse resultado condiz com a complexidade $O(n/m)$ do algoritmo.

Para a pesquisa em arquivo comprimido, o resultado obtido apresentou inconsistências. Com o aumento do padrão, o tempo de execução teve um ligeiro aumento em todos os arquivos teste. Também houve aumento ou igualdade com os tempos de busca para arquivos não comprimidos.

O número de comparações também foi inconsistente, aumentando em alguns casos junto ao aumento do padrão. No entanto, houve grande diminuição no número de comparações em relação ao arquivo não comprimido, o que era esperado.

Estes resultados vão contra o que é estabelecido na literatura, acreditamos que é devido a falhas na implementação do código.

Apesar da inconsistência nos tempos de execução, os mesmos continuam excelentes como o esperado. Quando comparado com o Shift-And para $k = 0$, é possível notar que a utilização do BMH é superior tanto em tempo de execução quanto em número de comparações, seja com arquivo comprimido ou não. Por isso ele deve ser amplamente utilizado para casamentos exatos.

BMH					
TLOTR original			TLOTR comprimido		
Tamanho de P	Comparações	Tempo	Tamanho de P	Comparações	Tempo
5	20.644	0.004452 sec	5	383	0.002393 sec
10	17.672	0.002243 sec	10	510	0.004838 sec
15	15.781	0.001990 sec	15	354	0.006530 sec
20	11.101	0.001389 sec	20	460	0.008225 sec
CF original			CF comprimido		
Tamanho de P	Comparações	Tempo	Tamanho de P	Comparações	Tempo
5	22.521	0.006527 sec	5	334	0.002310 sec
10	19.729	0.003486 sec	10	353	0.004567 sec
15	19.028	0.003075 sec	15	339	0.006922 sec
20	13.428	0.002210 sec	20	435	0.009210 sec
CPI original			CPI comprimido		
Tamanho de P	Comparações	Tempo	Tamanho de P	Comparações	Tempo
5	34.165	0.009479 sec	5	642	0.001970 sec
10	30.155	0.004992 sec	10	573	0.003750 sec
15	28.844	0.004316 sec	15	617	0.005606 sec
20	20.676	0.003057 sec	20	690	0.007440 sec

Figura 2: Testes experimentais para os algoritmos de busca exata.

9 Conclusão

Os algoritmos apresentados nesse trabalho são base para o estudo de processamento de caracteres. Existem melhorias que podem ser realizadas na distância de edição e no método de compressão.

Para que seja possível começar uma casamento em qualquer posição do texto no algoritmo implementado para a distância de edição, é necessário alterar $M_{0j} = 0$ para todo j pertencente ao texto. As correções nos algoritmos de compressão demandam maior estudo do código fonte da nossa parte.

Os resultados obtidos de acordo com a literatura mostram que para casamentos exatos o BMH deve ser utilizado em casos gerais e para casamentos aproximados o Shift-And deve ser usado em casos gerais.

Caso seja importante a facilidade de implementação, o BMH é superior nesse quesito.

O BMH em arquivo comprimido é o mais eficiente tanto em número de comparações quanto em tempo de execução.

10 Referencias

[1] http://www.ijecs.in/index.php/ijecs/article/view/651/580

[2] https://news.un.org/pt/

[3] 3 CHAVES, Rodrigo; SÁ, Giovanni; VIEIRA, Ramon; MOURÃO, Fernando; ROCHA, Leonardo. SACI: Sentiment Analysis by Collective Inspection on Social Media Content. In: CONCURSO

DE TRABALHOS DE INICIAÇÃO CIENTÍFICA DA SBC (CTIC-SBC), 33. , 2014, Brasília. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2014 . p. 51-60

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

[5] G. Navarro and M. Raffinot. Flexible Pattern Matching in String. Cambrige University Press, 2003.