UNIVERSIDADE DO ESTADO DO AMAZONAS
ESCOLA SUPERIOR DE TECNOLOGIA

Jonatas Travessa Souza de Barros
Lucas Mendes Sonoda
Natanael da Mota Figueira
Paulo André Carneiro Fernandes

TRABALHO SOBRE OS ALGORITMOS DE RECONHECIMENTO DE PADRÕES DE CADEIA: HORSPOOL, QUICK SEARCH, TUNED BOYER-MOORE E ZHU-TAKAOKA

Manaus
2021

Jonatas Travessa Souza de Barros
Lucas Mendes Sonoda
Natanael da Mota Figueira
Paulo André Carneiro Fernandes

TRABALHO SOBRE OS ALGORITMOS DE RECONHECIMENTO DE PADRÕES DE CADEIA: HORSPOOL, QUICK SEARCH, TUNED BOYER-MOORE E ZHU-TAKAOKA

Trabalho apresentado como requisito parcial para obtenção de aprovação na disciplina Algoritmos e Estruturas de Dados II, no Curso de Bacharelado em Sistemas de Informação, na Universidade do Estado do Amazonas.

Prof. Sérgio Cleger Tamayo

Manaus
2021

Sumário

Introdução

Este trabalho tem por objetivo demonstrar alguns dos algoritmos que são capazes de procurar um padrão em um texto de forma eficiente, ou seja, programas que são capazes na prática, de verificar se uma determinada sequência de caracteres está presente em um texto maior. Em particular, os algoritmos analisados neste relatório são: Horspool, Quick Search, Tuned Boyer-Moore e Zhu-takaoka, os quais são modificações do algoritmo de Boyer-Moore.

Horspool

Descrição e Exemplo Gráfico

Boyer-Moore-Horspool é um algoritmo para achar substrings em strings. O algoritmo compara cada carácter da substring para achar uma palavra ou os mesmos caracteres na string. Quando um carácter não encaixa, a busca pula para a próxima posição no padrão indicada pelo valor da Bad Match Table.

O algoritmo Boyer-Moore-Horspool é feito para otimizar as buscas de coincidências em strings e tem muitos usos, alguns são: Barras de busca, auto corretores, big data.

A Bad Match Table indica o tamanho do salto que deve ser dado da posição atual para a próxima, a Fórmula para a Bad Match Table é dada por: *Valor = tamanho da substring — index de cada letra na substring — 1*

Exemplo de uma Bad Match Table:

0 1 2 3
a b c d

Letter	a	b	c	d	*
Value	3	2	1	4	4

Value (a) = 4 - 0 - 1 = 3

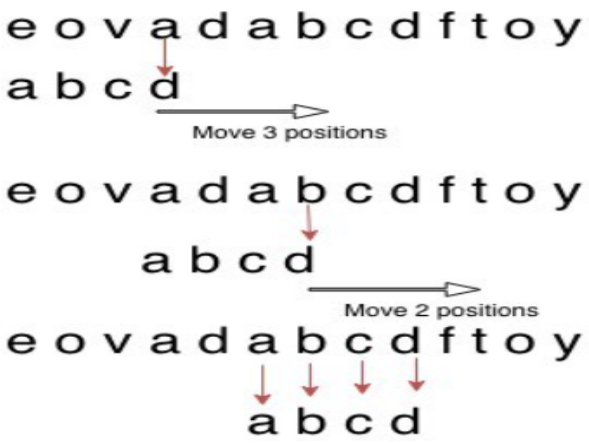
Value (b) = 4 - 1 - 1 = 3

Value (c) = 4 - 2 - 1 = 3

Value (d) = 4

O próximo passo é comparar a substring com uma string, nessa etapa, começamos a comparar pelo final da substring, no caso acima o "d". Se a letra bater com a letra da string, deve-se comparar a letra anterior na substring. Se a letra não bater, deve-se checar o valor na Bad Match Table da letra na string, caso a letra não esteja na substring, é pulado o tamanho da substring, nesse exemplo 4.

Esse processo se repete até achar a substring ou a string ser percorrida por completo.



Código comentado em C++

```cpp
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <iostream>

using namespace std;

const unsigned char* boyermoore_horspool_memmem(const unsigned char* haystack,
    size_t hlen,
    const unsigned char* needle,
    size_t nlen)
{
    size_t scan = 0;
    size_t bad_char_skip[UCHAR_MAX + 1]; // = 256

    /* Check para garantir que existe uma string, uma subtring e que a substring é maior que a string */
    if (nlen <= 0 || !haystack || !needle)
        return NULL;

    /* Pré-processamento */
    /* Inicializa a bad character table */
    /* Quando um caracter que não esta na needle é
     * encontrada, podemos pular todo o tamanho da
     * needle com segurança
     */
    for (scan = 0; scan <= UCHAR_MAX; scan++) bad_char_skip[scan] = nlen;

    /* Ultimo elemento da substring */
    size_t last = nlen - 1;

    /* Preenchendo a bad character table com os valores retirados da substring */
    for (scan = 0; scan < last; scan++) bad_char_skip[needle[scan]] = last - scan;


    /* Matching */

    /* Procurando na string enquanto a substring ainda pode ser contida nela */
    while (hlen >= nlen)
    {
        /* Scan do final da substring */
        for (scan = last; haystack[scan] == needle[scan]; scan--)
        {
            /* Se o primeiro byte bater, foi encontrado */
            if (scan == 0)
            {
                return haystack;
            }
        }

        /* Senão, precisamos pular alguns bytes e comlar novamente */
        hlen -= bad_char_skip[haystack[last]];
        haystack += bad_char_skip[haystack[last]];
    }

    return NULL;
}

int main()
{
    unsigned char needle[] = "capivara";
    unsigned char haystack[] = "isso e um texto capivara para testes";

    const unsigned char* result = boyermoore_horspool_memmem(haystack, 36, needle, 8);
    std::cout << result << std::endl;
}
```

Complexidade

Sendo m o tamanho da string e n o tamanho da substring.

- Pior caso: O(m*n)
- Caso médio: O(n)
- Melhor caso: O(n/m)

Vantagens e Desvantagens

Vantagens:

- Simplificação do algoritmo de Boyers-Moore.
- Fácil de implementar.
- Veloz em strings grandes

Desvantagens:

- Não é aplicável em alfabetos menores (relativo ao tamanho do padrão a ser encontrado).
- Para buscas em strings binárias o algoritmo KMP é mais recomendado.

Quick Search

Descrição e Exemplo Gráfico

O Quick Search é uma versão simplificada do algoritmo de Boyer-Moore que utiliza o chamado bad character shift table, que nada mais é do que uma tabela que armazena o pulo de caracteres quando ocorre um mismatch (o caracter do texto e do padrão que estamos comparando são diferentes). Dessa forma, quando ocorre o mismatch, vários passos de comparação de caracteres são pulados melhorando a eficiência do algoritmo.

Quando se tem um texto muito grande e com bastantes caracteres diferentes e um padrão relativamente menor do que o texto, o Quick Search pode ser uma alternativa eficiente para encontrar todas as ocorrências do padrão no texto.

Diferentemente do Boyer-Moore, o Quick Search itera da esquerda para a direita enquanto há correspondência entre os caracteres. Quando uma desigualdade é encontrada o Quick Search desloca a padrão para a direita pulando um determinado número de posições determinada pelo valor de um caractere na tabela do bad character shift. Dessa forma, o Quick Search consegue encontrar mais de uma vez o padrão procurado e quando o valor do deslocamento é maior do que a última posição do texto ele encerra sua execução.

A tabela do bad character shift é feita durante a fase de pré-processamento, ela é construída da seguinte forma: sendo x um caracter qualquer pertencente ao padrão, atualizamos na tabela seu valor de deslocamento como sendo o tamanho do padrão menos a sua posição mais à direita no padrão, quando o caracter x não pertence ao padrão, seu valor de deslocamento é simplesmente o tamanho do padrão mais um.

A seguir, tem-se uma explicação gráfica sobre o algoritmo.

Suponha que o texto que iremos considerar seja o descrito abaixo.

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G

O padrão a ser procurado segue a seguir

G	C	A	G	A	G	A	G

Primeiramente precisamos construir a tabela de deslocamento.

Caracter	A	C	G	T
Deslocamento				

Na tabela de deslocamento temos todos os caracteres presentes no texto. O cálculo para cada caractere presente no padrão é dado pela fórmula: v(caractere) = (tamanho do padrão) - (última posição do caractere no padrão). Logo, para o A, o valor do deslocamento é v(A) = 8 - 6 = 2.

Caracter	A	C	G	T
Deslocamento	2			

Para o C, temos v(C) = 8 - 1 = 7.

Caracter	A	C	G	T
Deslocamento	2	7		

Para o G, temos v(G) = 8 - 7 = 1.

Caracter	A	C	G	T
Deslocamento	2	7	1	

E, finalmente para o T, o cálculo não é o mesmo, pois T não está presente no padrão, nesse caso o cálculo é dado por v(caracter) = tamanho do padrão + 1. Logo, para o T, v(T) = 8 + 1 = 9.

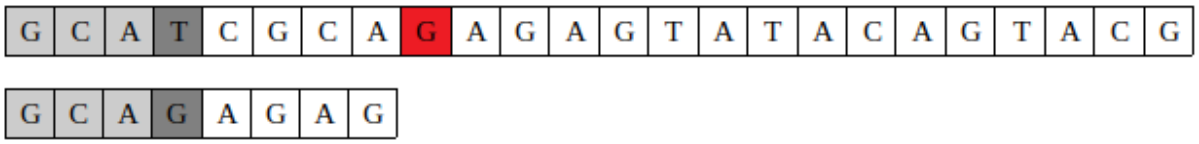
Caracter	A	C	G	T
Deslocamento	2	7	1	9

Agora com o pré-processamento completo podemos seguir para a busca do padrão. Primeiramente, o padrão é alinhado à esquerda junto com o texto de busca. A seguir, caso haja igualdade, o algoritmo vai iterando sobre o padrão e o texto da esquerda para a direita até encontrar uma desigualdade.

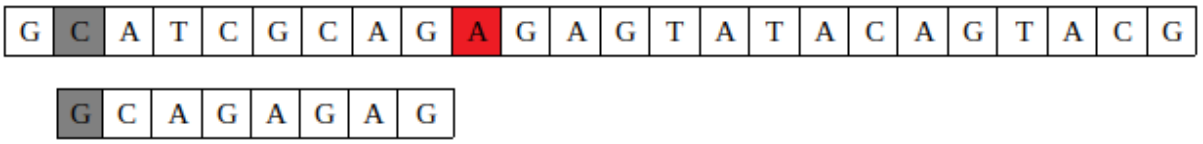
G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G

G	C	A	G	A	G	A	G

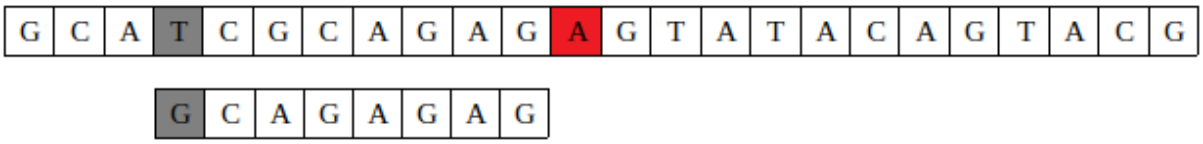
Nesse caso o algoritmo não encontrou o padrão ainda, agora ele vai ver qual é a o caractere do texto da primeira posição à direita da posição final do padrão e verificar qual é o deslocamento desse caractere na tabela.



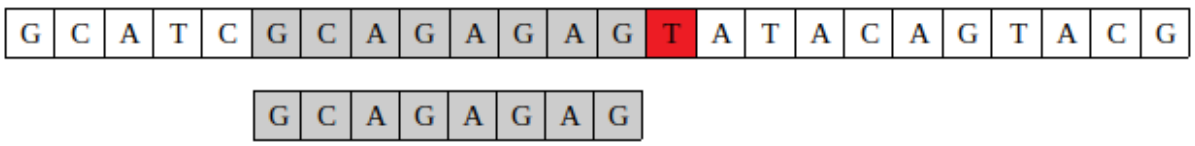
Esse caractere é o G que na tabela de deslocamento possui o valor 1, logo o padrão é deslocado em 1 posição à direita.



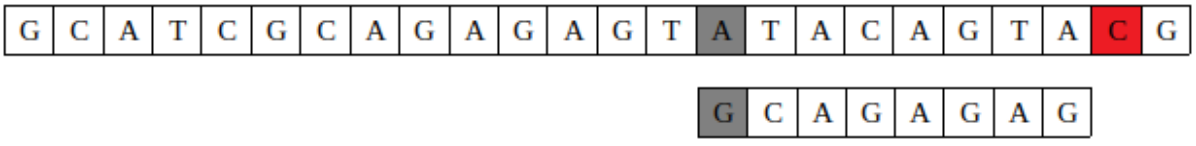
Como demonstrado no passo acima, o padrão foi deslocado para a direita em 1 posição. A seguir o Quick Search, tenta novamente casar os caracteres começando da esquerda para a direita e assim que encontra a primeira de desigualdade, a qual neste caso foi entre o C e o G na primeira verificação, o algoritmo faz o mesmo procedimento que anteriormente e, consequentemente, o caractere que comanda o deslocamento dessa vez é o A que tem valor 2 na tabela. Repare que a cada passo no qual a letra que determina o deslocamento está no padrão, a última posição do padrão é sempre casada com o caractere do texto a cada deslocamento.



Mais uma vez o padrão será deslocado em 2 posições.



O padrão foi encontrado, mas o texto a ser analisado ainda não acabou, isso demonstra que o Quick Sort pode encontrar mais de uma vez o padrão no texto, portanto, o próximo deslocamento é ditado pelo valor de T na tabela.



Como T não fazia parte do padrão, o algoritmo dá um salto do tamanho do padrão + 1, evitando assim, muitas comparações desnecessárias que seriam feitas no força bruta, por exemplo. Agora, como o valor de C na tabela é 7, esse deslocamento é maior do que falta para terminar o texto, nesse momento o algoritmo encerra sua execução.

Apesar do texto do passo a passo acima ter 24 caracteres, com o Quick Search, foram necessárias 15 operações de comparação no total, ou seja, aproximadamente 37,5% menos operações do que seria necessário com o força bruta.

Código Comentado em C++

```
#include <iostream>
#include <string.h>
#include <map>

using namespace std;

/* Tabela do bad shift character */
map <char, int> bad_shift_table;

/* Um iterador para o mapa da tabela do bad shift character */
map <char, int> ::iterator itr;

/* Retorna a última posição do caracter no padrão */
int get_last_char_position_in_pattern(char *pattern, char ch){

    for (int i = strlen(pattern) - 1; i >= 0; i--){
        if (pattern[i] == ch)
            return i;
    }

}

/* Verifica se o caracter do texto está presente no padrão */
int is_in_the_pattern(char *pattern, char ch){

    int it_is = 0;
    for (int i = 0; i < strlen(pattern); i++){
        if (pattern[i] == ch){
            it_is = 1;
            break;
        }

    }

    return it_is;
}

/* Pré-processamento */
void pre_processing(char *pattern, char *text){

    /* Constrói a tabela calculando para cada caracter seu deslocamento */
    for (int i = 0; i < strlen(text); i++){

        /* Se o caracter do texto está no padrão */
        if (is_in_the_pattern(pattern, text[i]) == 1){

            int shift = strlen(pattern) - get_last_char_position_in_pattern(pattern, text[i]);
            bad_shift_table.insert(pair<char, int>(text[i], shift));
        }

        /* Se o caracter do texto não está no padrão */
        else{

            int shift = strlen(pattern) + 1;
            bad_shift_table.insert(pair<char, int>(text[i], shift));
        }
    }
}
```

```
/* Busca */
void search(char *pattern, char *text){

    int pattern_index = 0;
    int char_matched = 0;
    int n_matched = 0;
    int pos_begin_match = 0;

    for (int text_index = 0; text_index < strlen(text); text_index++){

        /* Correspondência de caracter */
        if (text[text_index] == pattern[pattern_index]){

            char_matched++;
            pattern_index++;

            /* Se o número de caracter correspondentes em sequência é igual ao tamnho do padrão,
             * temos uma ocorrência do padrão
             */
            if (char_matched == strlen(pattern)){

                cout << "Pattern found!\n";
                cout << pattern << "\n";
                cout << "Between positions " << text_index + 1 - strlen(pattern) + 1
                     << " and " << text_index + 1 << "\n";

                n_matched++;
            }
        }

        /* Não correspondência de caracter: aplicar o deslocamento correto */
        else {

            itr = bad_shift_table.find(text[text_index + (strlen(pattern) - pattern_index)]);

            /* Se não é um caracter do texto, isso significa o término do texto */
            if (itr->second == 0)
                break;

            pos_begin_match += itr->second;
            text_index = pos_begin_match - 1;
            char_matched = 0;
            pattern_index = 0;
        }
    }

    /* Se nenhuma ocorrência foi encontrada */
    if (n_matched == 0)
        cout << "Pattern not found\n";
}

int main()
{

    char text[] = "GCATCGCAGAGAGTATACAGTACG";

    char pattern[] = "GCAGAGAG";

    /* Fase de pré-processamento */
    pre_processing(pattern, text);

    /* Fase de busca */
    search(pattern, text);

    return 0;
}
```

Complexidade

A fase de pré-processamento tem complexidade de O(m + σ) de tempo, onde m é o tamanho do padrão e σ é o tamanho do alfabeto do padrão. Já a fase de busca do padrão tem complexidade de pior caso de O(m*n), dá pra perceber que se n for igual ou muito próximo de m e o texto e o padrão forem muito parecidos, essa complexidade acaba se tornando quadrática. No entanto, a complexidade de melhor caso é O(n/m) e a complexidade média é de O(m+n). Portanto, na prática, ele é muito eficiente para textos com grandes alfabetos.

Quanto à complexidade de espaço, o algoritmo tem comportamento linear e precisa apenas do espaço extra para armazenar a tabela de deslocamento e portanto, essa complexidade é de O(σ), onde σ é o tamanho do alfabeto do padrão.

Vantagens e Desvantagens

Vantagens:

- Fácil de implementar;
- Na prática, ele é muito rápido para textos com grandes alfabetos e padrões curtos;
- Na grande maioria do casos, mais eficiente que o força bruta;
- Quando ele encontra um caracter que não está presente no padrão, ele salta a quantidade de posições do padrão no texto, evitando muitas comparações desnecessárias;
- Captura todas as ocorrências do padrão no texto.

Desvantagens:

- Para um texto do tipo CDDDDDDDDDDDDD e o padrão a ser procurado for algo como DD, o Quick Search acaba sendo quase tão lento quanto o força bruta, pois os deslocamentos num caso como esse, seriam quase sempre iguais a 1;
- Tem uma tendência de não funcionar bem em alfabetos pequenos como o DNA;
- Quanto mais ocorrências do padrão no texto, menos eficiente ele é.

Resolução da Questão 2651 do URI

Os algoritmos Horspool e Quick Search, com as devidas adaptações, foram utilizados para resolver a questão 2651 do URI. Em resumo, a questão simplesmente pedia para verificar se um determinado padrão estava presente em uma string fornecida como entrada para o programa, em particular, a entrada poderia ter até 10^5 caracteres.

Como resultado, os algoritmos não somente acertaram todos os casos apresentados pela questão, assim como, foram capazes de respeitar o limite de tempo imposto para a aceitação da resolução.

Resolução com o Horspool

No algoritmo em si não foi necessário fazer nenhuma alteração, apenas no tratamento da entrada e colocar uma condição para caso a palavra fosse encontrada. Segue o código abaixo.

```
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <algorithm>
#include<cstring>
#include <iostream>

using namespace std;

const unsigned char* boyermoore_horspool_memmem(const unsigned char* haystack,
                                                size_t hlen,
                                                const unsigned char* needle,
                                                size_t nlen)
{
    size_t scan = 0;
    size_t bad_char_skip[UCHAR_MAX + 1]; // = 256

    if (nlen <= 0 || !haystack || !needle)
        return NULL;

    for (scan = 0; scan <= UCHAR_MAX; scan++) bad_char_skip[scan] = nlen;

    size_t last = nlen - 1;

    for (scan = 0; scan < last; scan++) bad_char_skip[needle[scan]] = last - scan;

    while (hlen >= nlen)
    {

        for (scan = last; haystack[scan] == needle[scan]; scan--)
        {

            if (scan == 0)
            {
                    return haystack;
            }
        }

        hlen     -= bad_char_skip[haystack[last]];
        haystack += bad_char_skip[haystack[last]];
    }

    return NULL;
}

int main()
{
    unsigned char needle[] = "zelda";
    string input;
    cin >> input;

    for_each(input.begin(), input.end(), [](char& c) {
        c = ::tolower(c);
        });

    unsigned char haystack[input.length()+1];
    strcpy((char *)haystack, input.c_str());

    int  needle_length = 5;
    int  haystack_length = input.length()+1;

    const unsigned char* result = boyermoore_horspool_memmem( haystack, haystack_length, needle, needle_length);

    if (result == NULL)
    {
        std::cout << "Link Tranquilo" << std::endl;
    }
    else
    {
        std::cout << "Link Bolado" << std::endl;
    }
}
```

Resolução com o Quick Search

A principal adaptação necessária foi adotar um tipo de retorno para a função de busca, pois quando a string procurada era encontrada, se via necessário retornar esse valor para o main, onde estava a lógica de entrada e saída do problema. Portanto foram necessárias pequenas mudanças apenas na fase de busca, assim como, criar uma função main para lidar com a entrada e saída de dados. Outra pequena alteração foi adicionar uma função com o mesmo retorno adotado para a função de busca, essa função foi nomeada de Quick Search e basicamente seu trabalho era chamar as funções de pré-processamento e busca. Segue-se, em seguida,o código com suas partes alteradas.

```
#include <iostream>
#include <string.h>
#include <map>
#include <algorithm>
#include <cctype>

using namespace std;

map <char, int> bad_shift_table;
map <char, int> ::iterator itr;

int get_last_char_position_in_pattern(string pattern, char ch){

    for (int i = pattern.size() - 1; i >= 0; i--){
        if (pattern[i] == ch)
            return i;
    }

}

int is_in_the_pattern(string pattern, char ch){

    int it_is = 0;
    for (int i = 0; i < pattern.size(); i++){
        if (pattern[i] == ch){
            it_is = 1;
            break;
        }

    }

    return it_is;
}

void pre_processing(string pattern, string text){


    for (int i = 0; i < text.size(); i++){

        if (is_in_the_pattern(pattern, text[i]) == 1){
            int shift = pattern.size() - get_last_char_position_in_pattern(pattern, text[i]);
            bad_shift_table.insert(pair<char, int>(text[i], shift));
        }
        else{
            int shift = pattern.size() + 1;
            bad_shift_table.insert(pair<char, int>(text[i], shift));
        }
    }
}
```

```
/* Retorna 1 se o padrão foi encontrado, 0 caso contrário */
int search(string pattern, string text){

    int pattern_index = 0;
    int char_matched = 0;
    int n_matched = 0;
    int pos_begin_match = 0;

    for (int text_index = 0; text_index < text.size(); text_index++){

        if (text[text_index] == pattern[pattern_index]){

            char_matched++;
            pattern_index++;

            /* Padrão encontrado, não há necessidade de continuar */
            if (char_matched == pattern.size()){
                return 1;
                }
        }
        else {
            itr = bad_shift_table.find(text[text_index + (pattern.size() - pattern_index)]);

            if (itr->second == 0)
                break;

            pos_begin_match += itr->second;
            text_index = pos_begin_match - 1;
            char_matched = 0;
            pattern_index = 0;
        }
    }
    return 0;
}

int quick_search(string pattern, string tetx){

    /* A fase de pré-processamento permaneceu igual */
    pre_processing(pattern, tetx);
    return search(pattern, tetx);

}

int main(){

    string pattern = "zelda";

    string input;

    cin >> input;

    /* Converte a entrada para lower_case */
    transform(input.begin(), input.end(), input.begin(),
        [](unsigned char c){ return std::tolower(c); });

    if (quick_search(pattern, input))
        cout << "Link Bolado\n";
    else
        cout << "Link Tranquilo\n";

    return 0;
}
```

Tuned Boyer-Moore

Descrição e Exemplo Gráfico

O Tuned Boyer-Moore é uma versão simplificada e mais rápida do algoritmo Boyer-Moore, onde assim como o Boyer-Moore ele busca uma substring menor dentro de uma string maior. Ele também pode ser visto como uma implementação eficiente do Horspool.

Assim como o Boyer-Moore, faz o pré-processamento criando a Bad Match Table, para definir os saltos que são usados durante a busca, após encontrar paridade na comparação do último caractere da substring, a comparação de caracter por caracter pode ser feita em qualquer ordem.

O Tuned Boyer-Moore é uma implementação de uma versão simplificada do algoritmo Boyer-Moore que é muito rápido na prática. A parte mais cara de um algoritmo de correspondência de strings é verificar se o caractere do padrão corresponde ao caractere da janela. Para evitar fazer essa parte com muita frequência, é possível desenrolar vários turnos antes de realmente comparar os caracteres. O algoritmo usou a função de deslocamento do bad-character para encontrar x [m-1] em y e continuar se deslocando até encontrá-lo, fazendo três deslocamentos seguidos às cegas. Isso é necessário para salvar o valor de bmBc [x[m-1]] em uma mudança de variável e, em seguida, definir bmBc[x[m-1]] para 0. Isso exigia também adicionar m ocorrências de x [m-1] no final de y. Quando x [m-1] é encontrado o m -1, outros caracteres da janela são verificados e um desvio de comprimento de deslocamento é aplicada.

A seguir tem-se um exemplo gráfico do seu funcionamento.

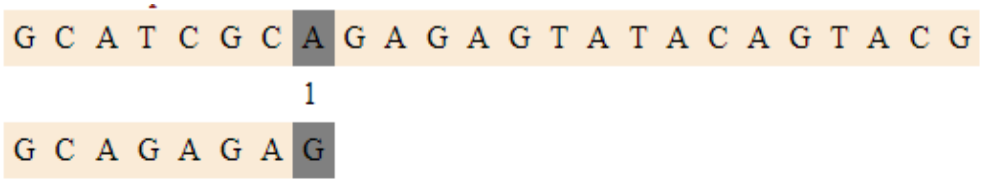
Começamos com a fase pré-processamento, onde construímos a tabela bmBc. Tal tabela para o exemplo mostrado em seguida está disposta abaixo.

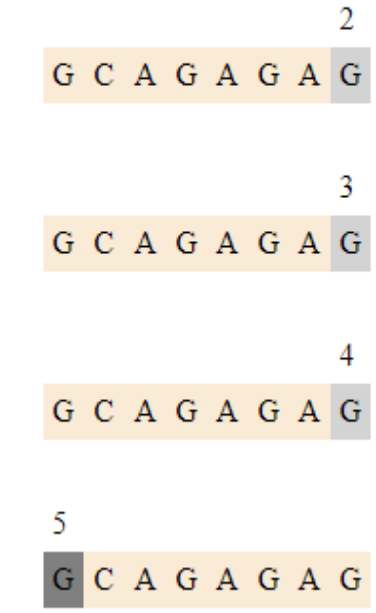
a	A	C	G	T
$bmBc[a]$	1	6	0	8

$shift = 2$

Após a construção da tabela, vem a fase da procura pelo padrão.

1º tentativa

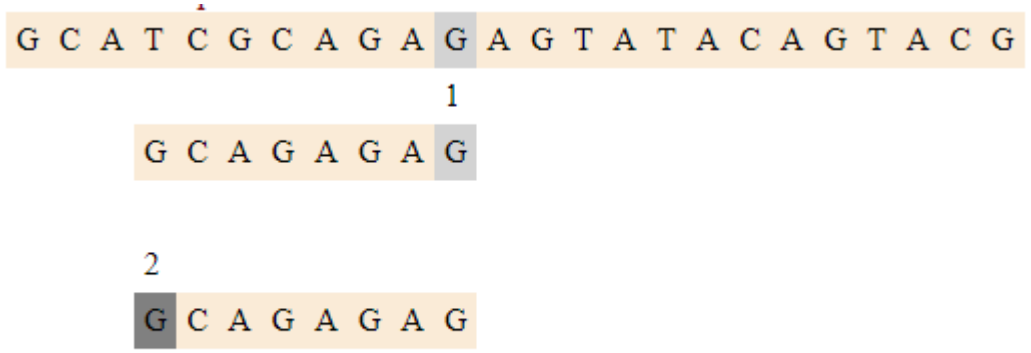




Shift by: 2 (*shift*)

No passo 1: não houve paridade entre substring e string, sendo o caractere A na string sendo analisada, é feito um salto de 1. Nos passos 2, 3 e 4: fica na comparação que encontra paridade entre substring e string, passando para verificação da esquerda para a direita. No passo 5: começando a verificação da esquerda para a direita, não foi encontrado paridade e nesse caso por padrão vai ser feito salto de 2.

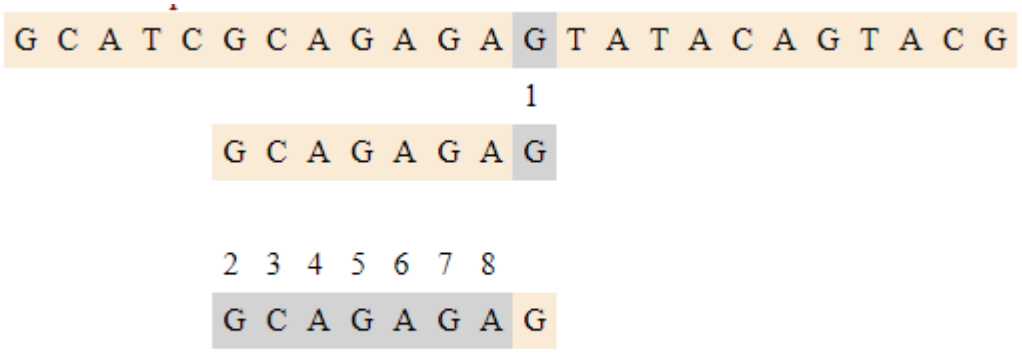
2º Tentativa



Shift by: 2 (*shift*)

No passo 1: foi encontrado paridade no final da substring, assim liberando para a comparação desde o primeiro caractere. No passo 2: é encontrado disparidade, dessa forma sendo necessário o salto de 2.

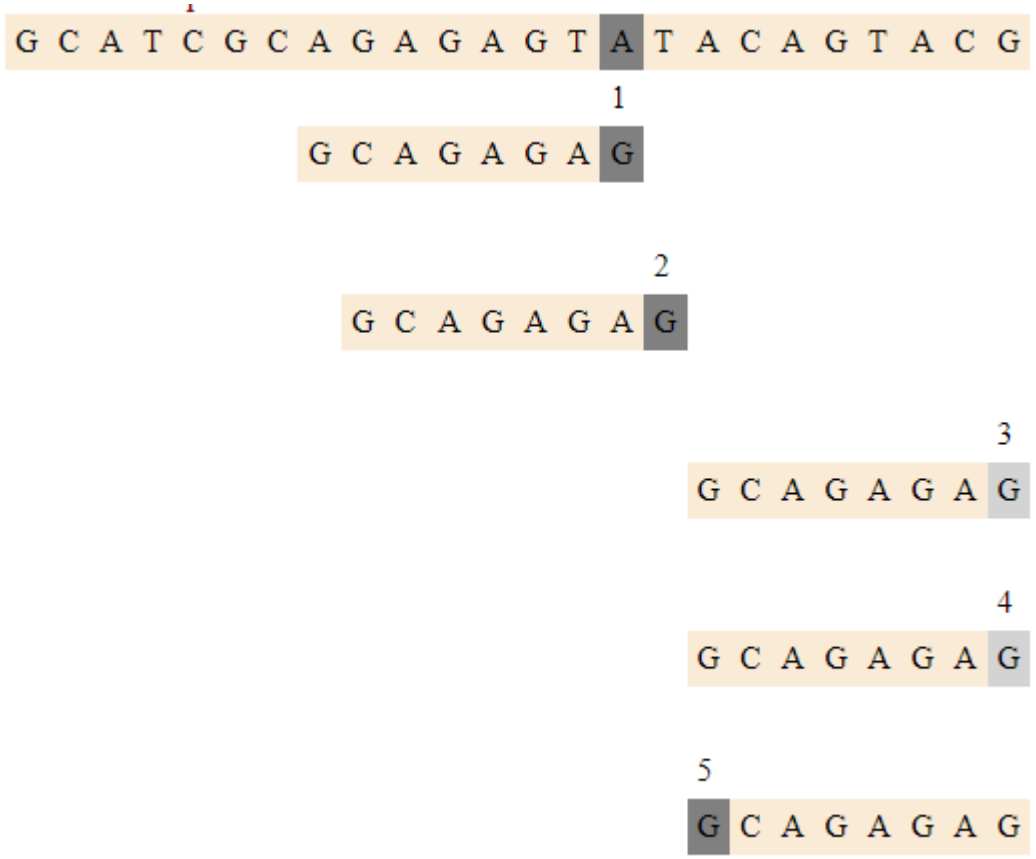
3º Tentativa



Shift by: 2 (*shift*)

No passo 1: é encontrado paridade e pode começar a verificação caractere a caractere. Nos passos 2 a 8: é encontrado paridade em cada um, de tal forma que é encontrado uma ocorrência da substring na string. Como não terminou a string, é feito mais um salto de 2 em busca de mais ocorrências.

4º Tentativa



Shift by: 2 (*shift*)

No passo 1: é encontrado disparidade e por ser a letra A em questão, é feito salto de 1. No passo 2: além da disparidade, é a letra T sendo comparada, ela não possui na substring, então o salto é de 8 (o tamanho da substring). Nos passos 3 e 4: é encontrado paridade, então parte para a comparação do primeiro caractere. No passo 5: é encontrado disparidade e iria ter salto, mas devido já ter chego ao final da string, a procura é finalizada. Ao final de tudo foram feitos apenas 10 passos para procurar a substring na string, demonstrando quanto o algoritmo é rápido.

Código Comentado em C

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

/* função de pré processamento */
void preBmBc(char *x, int m, int bmBc[])
{
    int i;

    /* preenchendo tabela alfabeto */
    for (i = 0; i < 256; ++i)
        bmBc[i] = m;

    /* configurando tabela com caracteres da palavra */
    for (i = 0; i < m - 1; ++i)
        bmBc[x[i]] = m - i - 1;
}

/* função do Tuned Boyer-Moore */
void TUNEDBM(char *x, int m, char *y, int n)
{
    int j, k, shift, bmBc[256];

    /* pré processamento */
    preBmBc(x, m, bmBc);
    shift = bmBc[x[m - 1]];
    bmBc[x[m - 1]] = 0;
    memset(y + n, x[m - 1], m);

    /* Procurando palavra no texto */
    j = 0;
    while (j < n)
    {
        k = bmBc[y[j + m -1]];
        while (k !=  0)
        {
            /* interações iniciais */
            j += k; k = bmBc[y[j + m -1]];
            j += k; k = bmBc[y[j + m -1]];
            j += k; k = bmBc[y[j + m -1]];
        }

        /* Encontrando ocorrencia da palavra no texto */
        if (memcmp(x, y + j, m - 1) == 0 && j < n);
            printf("Ocorrencia na pos: %d\n", j);

        /* shift */
        j += shift;
   }
}
```

Complexidade

Sendo m = tamanho da string e n = tamanho da substring.

- Fase de pré-processamento em tempo O (m + σ) e complexidade de espaço O (σ), σ é o número de alfabetos no padrão;
- Fase de busca na complexidade de tempo O (m*n).

Vantagens e Desvantagens

Vantagens:

- Simplificação do algoritmo Boyer-Moore;
- Fácil de implementar;
- Rápido na prática.

Desvantagens:

- Não é aplicável em alfabetos menores (relativo ao tamanho do padrão a ser encontrado).

Zhu-Takaoka

Descrição e Exemplo Gráfico

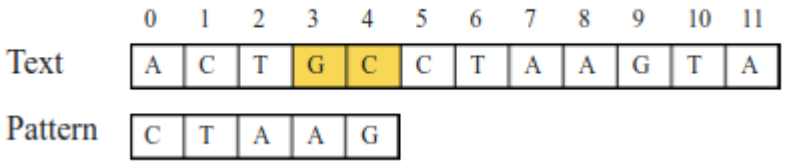
Zhu-Takaoka é um algoritmo de solução de problemas de correspondência de string que é uma variante do algoritmo de Boyer-Moore. O algoritmo apenas melhora a regra do caractere errado do algoritmo de Boyer-Moore.

Zhu e Takaoka modificaram o algoritmo de Boyer-Moore. Eles substituíram pela regra da substring dupla. A regra do sufixo correto continua sendo usada. Eles desenvolveram um algoritmo que performa a operação de shift considerando o shift do caractere errado para dois caracteres de texto consecutivos.

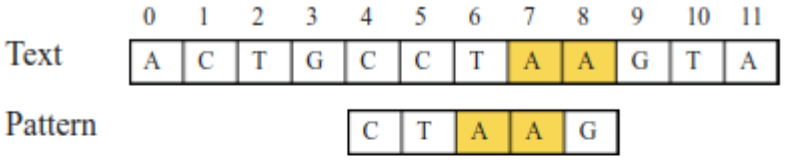
Durante a fase de busca, as comparações são performadas da direita para a esquerda e quando a string alvo é posicionada no fator de texto y[j .. j+m-1] e ocorre um mismatch entre x[m-k] and y[j+m-k] enquanto que no caso de x[m-k+1 .. m-1]=y[j+m-k+1 .. j+m-1] o shift é performado com o princípio do shift do caractere errado para os caracteres de texto y[j+m-2] e y[j+m-1]. A tabela de shift do sufixo correto é também usada para computar os shifts.

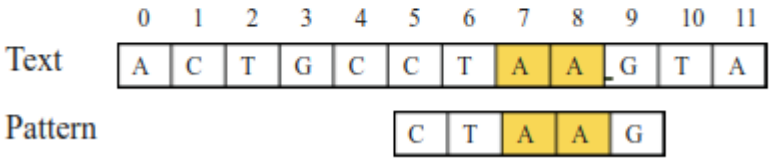
A seguir temos um exemplo gráfico ilustrando seu funcionamento, utilizando a regra da substring dupla.

Considere o texto=ACTGCTAAGTA e o padrão=CTAAG.



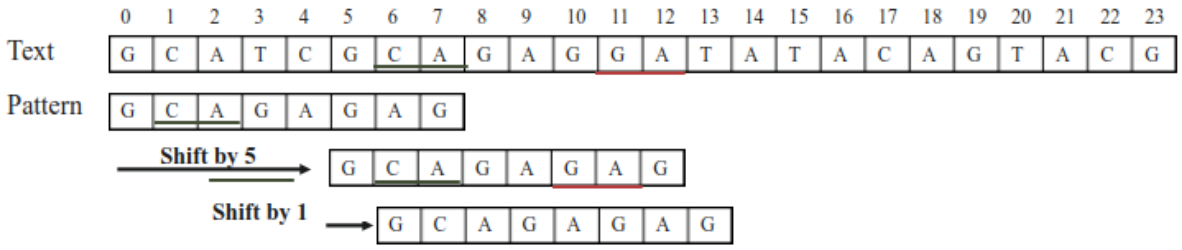
Não há GC em P.





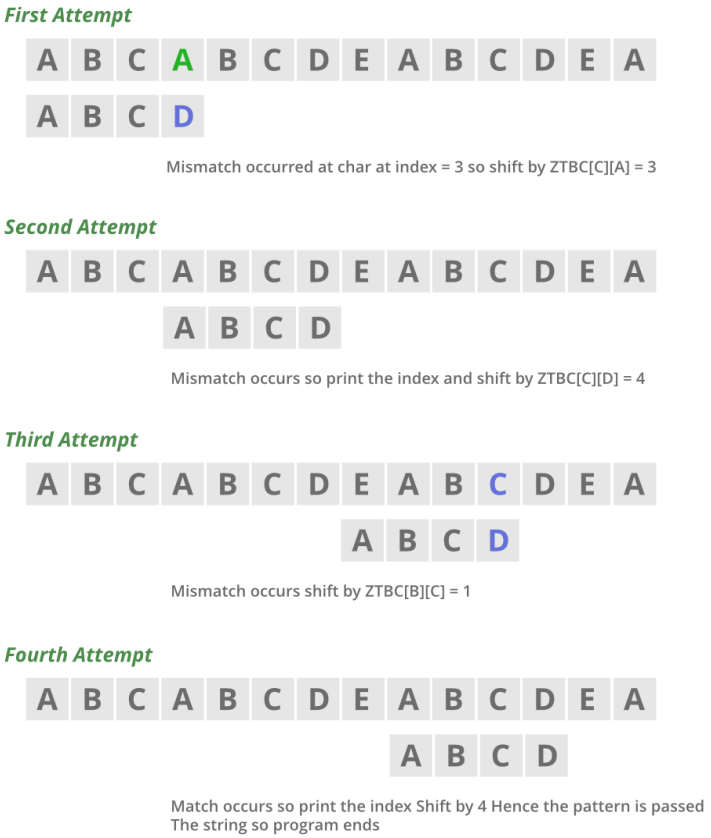
Como saber se uma substring dupla específica aparece em P ou não?

Sempre que não ocorrer uma correspondência ou ocorrer uma correspondência completa, selecionamos a última substring dupla em T e procuramos pela localização mais à direita dessa substring dupla em P, se existente. Isso é feito construindo uma tabela ztBc. Exemplo:



ztBc	A	C	G	*
A	8	8	2	8
C	5	8	7	8
G	1	6	7	8
*	8	8	7	8

T(CA)=5 significa que CA aparece em 5 localizações da extremidade direita. Assim, podemos realizar o shift por 5. T(GA)=1 significa que GA aparece em 1 localização da extremidade direita. Se GA é a substring dupla a ser correspondida, realizamos o shift por 1.



Código Comentado em C++

```cpp
#include <bits/stdc++.h>
using namespace std;

string str = "ABCABCDEABCDEA";
string pattern = "ABCD";

/* Tamanhos da sting e do padrão */
int stringlen = 14;
int patternlen = 4;

/* Matriz para armazenar a tabela ZTBC */
/* Como o alfabeto do padrão tem um total de 26 caracteres,
   logo é necessário uma matriz de 26x26 posições */
int ZTBC[26][26];

/* Pré-processamento e cálculo da tabela ZTBC */
void ZTBCCalculation(){
    int i, j;
    for (i = 0; i < 26; ++i)
        for (j = 0; j < 26; ++j)
                ZTBC[i][j] = patternlen;

    for (i = 0; i < 26; ++i)
        ZTBC[i][pattern[0] - 'A'] = patternlen - 1;

    for (i = 1; i < patternlen - 1; ++i)
        ZTBC[pattern[i - 1] - 'A'][pattern[i] - 'A'] = patternlen - 1 - i;
}

int main(){
    int i, j;
    ZTBCCalculation();
    j = 0;

    while (j <= stringlen - patternlen) {

        i = patternlen - 1;
        while (i >= 0 && pattern[i] == str[i + j])
                --i;
                /* Padrão encontrado */
                if (i < 0) {

                        cout << "Pattern Found at " << j + 1 << endl;
                        j += patternlen;
                }
                /* Padrão não encontrado */
                else
                        j += ZTBC[str[j + patternlen - 2] - 'A'][str[j + patternlen - 1] - 'A'];
                }

    return 0;
}
```

Complexidade

- Fase de pré-processamento possui complexidade de O(m+σ^2) em questão de tempo e espaço;
- Fase de busca possui complexidade de O(m × n);
- O pior caso da fase de busca é quadrático.

Vantagens e Desvantagens

Vantagens:

- Melhora a regra do caractere errado de Boyer-Moore, substituindo-a pela regra da substring dupla;
- O algoritmo não é complexo de se usar;
- Funciona bem em determinadas situações.

Desvantagens:

- Diferença de complexidade de tempo e espaço não muito significativa em relação ao algoritmo de Boyer-Moore e outras variantes;
- Pouca documentação disponível acerca do algoritmo.

Resolução do problema 2049 do URI

Os algoritmos Tuned Boyer-Moore e Zhu-Takaoka, com as devidas adaptações, foram utilizados para resolver a questão 2049 do URI. Em resumo, a questão simplesmente pedia para verificar se um determinado padrão estava presente em uma string fornecida como entrada para o programa. Nessa questão, em particular, as strings eram formadas por dígitos e cada caso de teste poderia ter até $3*10^5$ dígitos por string.

Esses algoritmos são capazes de lidar de forma eficiente com procuras em texto grande número de caracteres, e por isso, são algoritmos recomendados para a resolução desse problema do URI.

Resolução com o Tuned Boyer-Moore

A seguir tem-se o código do Tuned Boyer-Moore com as adequações necessárias para a resolução da questão.

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

void preBmBc(char *x, int m, int bmBc[])
{
    int i;

    for (i = 0; i < 256; ++i)
        bmBc[i] = m;

    for (i = 0; i < m - 1; ++i)
        bmBc[x[i]] = m - i - 1;
}
```

```c
/* função do Tuned Boyer-Moore */
int TUNEDBM(char *x, char *y)
{
    int j, k, shift, bmBc[256];

    /* Pegando tamanho da palavra e do texto coletados */
    int m = strlen(x);
    int n = strlen(y);

    preBmBc(x, m, bmBc);
    shift = bmBc[x[m - 1]];
    bmBc[x[m - 1]] = 0;
    memset(y + n, x[m - 1], m);

    j = 0;
    while (j < n)
    {
        k = bmBc[y[j + m -1]];
        while (k !=  0)
        {
            j += k; k = bmBc[y[j + m -1]];
            j += k; k = bmBc[y[j + m -1]];
            j += k; k = bmBc[y[j + m -1]];
        }

        if (memcmp(x, y + j, m - 1) == 0 && j < n) return 1;

        j += shift;
   }
   return 0;
}

int main()
{
    /* Definindo variaveis pra etrada */
    char txt[300000]; // texto
    char pat[7]; // palavra
    int count = 1;

    /* Pegando entradas e processando */
    while (1)
    {
        /* Pegando palavra a ser verificada no texto */
        scanf("%s", pat);

        /* Caso de parada da entrada */
        if(strcmp(pat,"0") == 0)
        {
            break;
        }
        scanf("%s", txt);

        /* Impressão do resultado */
        printf("Instancia %d\n", count);
        if (TUNEDBM(pat, txt))
        {
            printf("verdadeira\n");
        }
        else
        {
            printf("falsa\n");
        }
        count++;
    }
    return 0;
}
```

Resolução com o Zhu-Takaoka

A seguir tem-se o código do Zhu-Takaoka com as adequações necessárias para a resolução da questão.

```
#include <bits/stdc++.h>
using namespace std;

int ZTBC[10][10];

void ZTBCCalculation(int stringlen, int patternlen, string pattern){
    int i, j;
    for (i = 0; i < 10; ++i)
        for (j = 0; j < 10; ++j)
                ZTBC[i][j] = patternlen;

    for (i = 0; i < 10; ++i)
        ZTBC[i][pattern[0] - '0'] = patternlen - 1;

    for (i = 1; i < patternlen - 1; ++i)
        ZTBC[pattern[i - 1] - '0'][pattern[i] - '0'] = patternlen - 1 - i;
}

int main(){

    string pattern;
    string str;

    int h = 1;
    while (cin >> pattern && pattern != "0"){

        cin >> str;
        int i, j;
        int stringlen = str.size();
        int patternlen = pattern.size();

        ZTBCCalculation(stringlen, patternlen, pattern);

        j = 0;
        bool check = false;
        while (j <= stringlen - patternlen) {
                i = patternlen - 1;
                while (i >= 0 && pattern[i] == str[i + j])
                    --i;

                if (i < 0) {

                    cout << "Instancia " << h << endl;
                    cout << "verdadeira" << endl;

                    j += patternlen;
                    check = true;
                }

                else
                    j += ZTBC[str[j + patternlen - 2] - '0'][str[j + patternlen - 1] - '0'];
        }

        if (check == false){
            cout << "Instancia " << h << endl;
            cout << "falsa" << endl;
        }
        h++;
    }
    return 0;
}
```

Referências

QUICK SEARCH ALGORITHM. **Igm.univ-mlv**. Disponível em: <https://www-igm.univ-mlv.fr/~lecroq/string/node19.html>. Acesso em 10 de julho de 2021.

BHOJASIA, Manish. Quick Search Algorithm Multiple Choice Questions and Answers (MCQs). **Sanfoundry**. Disponível em: <https://www.sanfoundry.com/quick-search-algorithm-multiple-choice-questions-answers-mcqs/>. Acesso em 10 de julho de 2021.

LEE, R, C, T. CHENG, C, W. Quick Search Algorithm. 28 slides. Disponível em: <https://slideplayer.com/slide/5139244/>. Acesso em 10 de julho de 2021.

THE BOYER-MOORE FAST STRING SEARCHING ALGORITHM. **The University of Texas at Austin.** Disponível em: <https://www.cs.utexas.edu/users/moore/best-ideas/string-searching/>. Acesso em 11 de julho de 2021.

KUMAR, Pancaj. Boyer-Moore Algorithm. **Hackerearth.** Disponível em: <https://www.hackerearth.com/practice/notes/boyer-moore-algorithm/>. Acesso em 12 de julho de 2021.

HORSPOOL ALGORITHM. **Igm.univ-mlv**. Disponível em: <https://www-igm.univ-mlv.fr/~lecroq/string/node18.html>. Acesso em 13 de julho de 2021.

SALDANA, Francisco. The Boyer-Moore Horspool Algorithm. **Medium.** Disponível em: <https://medium.com/@fsaldana/the-boyer-moore-horspool-algorithm-51b785afde67>. Acesso em 13 de julho de 2021.

ZHU-TAKAOKA ALGORITHM**. Igm.univ-mlv**. Disponível em: <https://www-igm.univ-mlv.fr/~lecroq/string/node20.html>. Acesso em 14 de julho de 2021.

LEE, R, C, T. TANG, S, W. The Zhu-Takaoka Algorithm. 20 slides. Disponível em: <https://slideplayer.com/slide/5197206/>. Acesso em 14 de julho de 2021.

LEE, R, C, T. CHENG, C, W. Tuned Boyer Moore Algorithm. 25 slides. Disponível em: <https://slideplayer.com/slide/1577533/>. Acesso em 14 de julho de 2021.

Tuned Boyer-Moore algorithm. Disponível em: <https://www-igm.univ-mlv.fr/~lecroq/string/tunedbm.html>. Acesso em 14 de julho de 2021.

CHOUDHARY, H, K. Java Program to Implement Zhu Takaoka String Matching Algorithm. Disponível em: <https://www.geeksforgeeks.org/java-program-to-implement-zhu-takaoka-string-matching-algorithm/>. Acesso em 15 de julho de 2021.