Algoritmos de Reconhecimento de Padrões

- Horspool
- Quick Search
- Tuned Boyer-Moore
- Zhu-Takaoka

Equipe

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

Roteiro

- Horspool
- Quick Search
- Tuned Boyer-Moore
- Zhu-Takaoka

Horspool

Descrição

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.

Exemplo Gráfico

Primeiro é necessário criar a Bad Match Table, ela vai ditar o salto que será dado caso a letra da substring não seja igual a da string.

Valor = tamanho da substring — index de cada letra na substring — 1

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

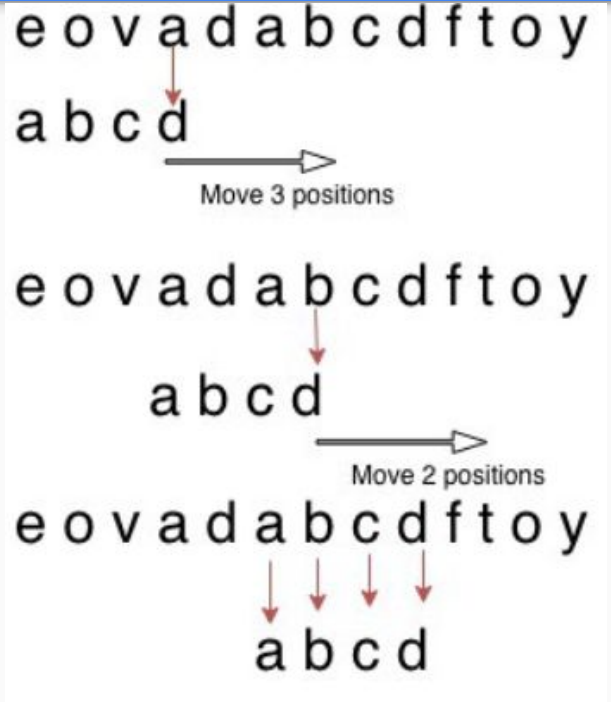
Exemplo Gráfico

Agora será feita a comparação da string com a substring, começando pela última letra da substring.

- 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 que a substring seja achada ou a string ser percorrida por completo

Exemplo Gráfico



Código em C++

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

/* Returns a pointer to the first occurrence of "needle"
 * within "haystack", or NULL if not found. Works like
 * memmem().
 */

 using namespace std;
/* Returns a pointer to the first occurrence of "needle"
 * within "haystack", or NULL if not found. Works like
 * memmem().
 */

/* Note: In this example needle is a Cstring. The ending
 * 0x00 will be cut off, so you could call this example with
 * boyermoore_horspool_memmem(haystack, hlen, "abc", sizeof("abc"))
 */
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;

    /* ---- Preprocess ---- */
    /* 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
       O calculo é feito com base nessa equação: Valor = tamanho da substring - index da letra na
substring - 1,
       nesse caso temos Valor = nLen - 1 - index.
    */
    for (scan = 0; scan < last; scan++) bad_char_skip[needle[scan]] = last - scan;
```

Código em C++

```
    /* ---- Do the 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.
        *  Nota-se que pulamos um valor baseado no ultimo byte da substring
        *  não importa onde não bateu. Então se a substring for "abcd"
        *  pulamos baseado no "d" e esse valor será 4.
        */
        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 = 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.

Questão 2651 do URI

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

/* Returns a pointer to the first occurrence of "needle"
 * within "haystack", or NULL if not found. Works like
 * memmem().
 */

 using namespace std;
/* Returns a pointer to the first occurrence of "needle"
 * within "haystack", or NULL if not found. Works like
 * memmem().
 */

/* Note: In this example needle is a Cstring. The ending
 * 0x00 will be cut off, so you could call this example with
 * boyermoore_horspool_memmem(haystack, hlen, "abc", sizeof("abc"))
 */
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;

    /* ---- Preprocess ---- */
    /* 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
       O calculo é feito com base nessa equação: Valor = tamanho da substring - index da letra na
substring - 1,
       nesse caso temos Valor = nLen - 1 - index.
    */
    for (scan = 0; scan < last; scan++) bad_char_skip[needle[scan]] = last - scan;
```

Questão 2651 do URI

```
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;
    }
}
```

Quick Search

Descrição

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.

Diferentemente do Boyer-Moore, o Quick Search itera da esquerda para a direita enquanto há correspondência entre os caracteres.

Descrição

Quando uma desigualdade é encontrada (mismatch), o Quick Search desloca a padrão para a direita pulando um determinado número de posições do texto determinada pelo valor de um caractere na tabela do bad character shift.

Ele encerra sua execução quando o valor do deslocamento é maior do que a última posição do texto.

Descrição

O Quick Search é um algoritmo de duas fases: pré-processamento e busca.

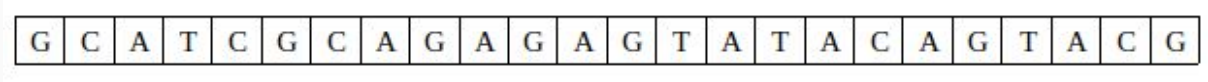
Na fase de pré-processamento é construída a tabela de deslocamentos (bad character shif table).

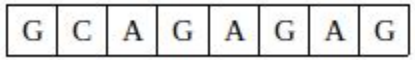
Na fase de busca, ocorre a procura das ocorrências do padrão no texto.

O Quick Search é capaz de encontrar todas as ocorrências do padrão.

Exemplo Gráfico

Para compreendermos seu funcionamento, consideremos o seguinte texto e o padrão que será procurado.





Exemplo Gráfico: Pré-processamento

O primeiro passo é a construção da tabela de deslocamentos.

Percorremos o texto e, para cada caractere único do texto haverá um deslocamento associado a ele.

Caracter	A	C	G	T
Deslocamento				

Exemplo Gráfico: Pré-processamento

Há dois cálculos para o deslocamento.

Se o caractere está presente no padrão seu deslocamento é dado por:
v(caracter) = (tamanho do padrão) - (última posição do caractere no padrão)

Se o caractere não está presente no padrão seu deslocamento é dado por:
v(caracter) = (tamanho do padrão) + 1

Exemplo Gráfico: Pré-processamento

Para o caracter A, temos v(A) = 8 - 6 = 2

Caracter	A	C	G	T
Deslocamento	2			

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

Caracter	A	C	G	T
Deslocamento	2	7		

Exemplo Gráfico: Pré-processamento

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

Caracter	A	C	G	T
Deslocamento	2	7	1	

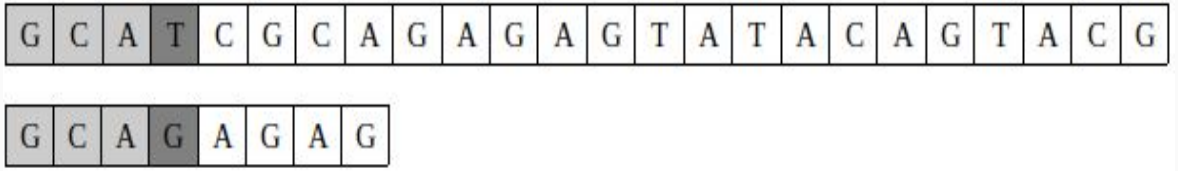
Para o caracter T (não pertence ao padrão), temos v(T) = 8 +1 = 9

Caracter	A	C	G	T
Deslocamento	2	7	1	9

Exemplo Gráfico: Busca

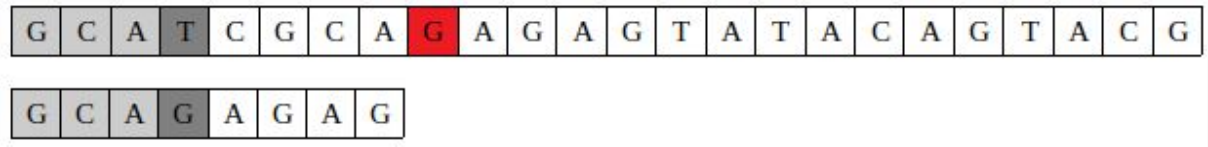
Após a construção da tabela de deslocamento, prosseguimos para a busca em si.

O Quick Search começa da esquerda do texto verificando da esquerda para a direita caracter por caracter.



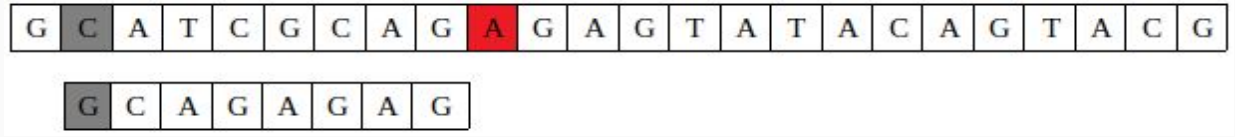
Exemplo Gráfico: Busca

Quando ocorre um mismatch, nesse caso entre o T e G, o Quick Search verifica qual é o caracter do texto da posição correspondente a última posição do padrão + 1, nesse caso, o caracter G em vermelho. É esse caracter que determina o deslocamento.



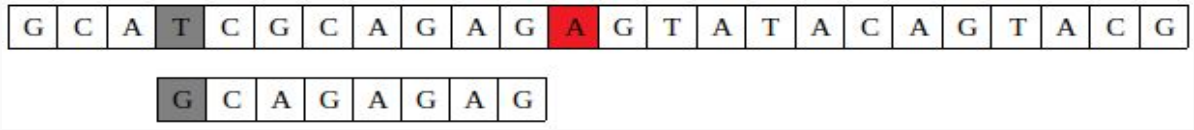
Exemplo Gráfico: Busca

Na tabela G, tem valor 1, portanto a próxima posição para iterar verificando os caracteres é deslocada em 1 para a direita em relação a primeira posição anterior. Aqui acontece um mismatch logo na primeira comparação e dessa vez, é o caracter A quem decide o deslocamento, que nesse caso é de 2.



Exemplo Gráfico: Busca

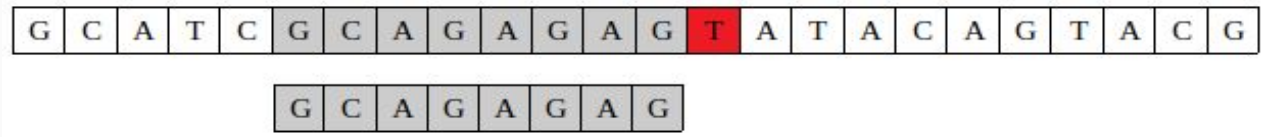
Deslocado duas posições à direita, o mesmo do ocorrido no passo anterior volta a acontecer. Portanto, mais um deslocamento.



Exemplo Gráfico: Busca

Começando a iterar a partir da posição calculada pelo deslocamento anterior, o padrão foi encontrado!

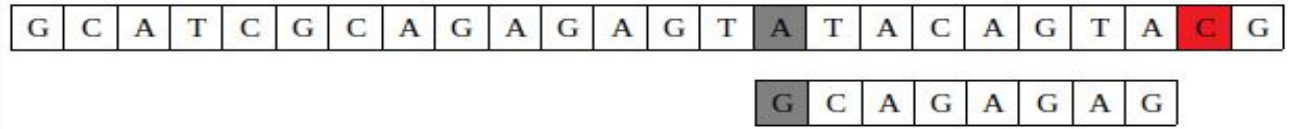
Agora, o primeiro caracter após o padrão é o da vez para ditar o próximo deslocamento, nesse caso o T.



Exemplo Gráfico: Busca

T não fazia parte do padrão, então tivemos um deslocamento do tamanho do padrão + 1, deslocamentos assim são os grandes responsáveis pela eficiência do algoritmo.

Com o próximo deslocamento seria de 7 posições e não há ao menos 7 posições sobrando no texto, o algoritmo pode ser encerrado.



Código Comentado (C++)

Funções Auxiliares

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

using namespace std;

/* Bad shift character table */
map <char, int> bad_shift_table;

/* An iterator for the map of the bad shift character table */
map <char, int> ::iterator itr;

/* Returns last position of the char in the pattern */
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;
    }

}

/* Verify if a text character is present in the pattern */
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;
}
```

Código Comentado (C++)

Pré - Processamento

```
/* Prepocessing */
void pre_processing(char *pattern, char *text){

    /* Construct the table by calculating for each text character its shift */
    for (int i = 0; i < strlen(text); i++){

        /* If the text character is in the pattern */
        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));
        }

        /* If the text character is not in the pattern */
        else{

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

Código Comentado (C++)

Busca

```
/* Search the pattern */
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++){

        /* Single character match */
        if (text[text_index] == pattern[pattern_index]){

            char_matched++;
            pattern_index++;

            /* If the number of matching characters sequence equal to pattern size */
            /* We have a pattern match */
            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++;
            }

        }
        /* Appy the correct shift each time we have a single character mismatch */
        else {

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

            /* If the character does not correspond to one in the table */
            /* we are no longer in the text */
            if (itr->second == 0)
                break;

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

    /* If the pattern was not found */
    if (n_matched == 0)
        cout << "Pattern not found\n";
}
```

Complexidade

- De tempo:
  - Pior caso O(m*n): ocorre quando padrão e texto são extremamente similares tanto em posição dos caracteres quanto em tamanho.
  - Caso médio O(m + n): o que é observado na prática.
  - Melhor caso O(n/m): só existe uma ocorrência do padrão, todos os demais caracteres não fazem parte do padrão e n é muito maior que m.
- De espaço:
  - O(σ): o espaço necessário para armazenar a tabela de deslocamento, σ é a quantidade de caracteres do texto (alfabeto).

Há também a complexidade da fase de pré-processamento: O(m + σ).

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.

Onde Usar

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 as ocorrências do padrão no texto.

Quando não há muitas ocorrências do padrão no texto.

Quando o texto e o padrão não compartilharem de muitas similaridades.

Aplicação do Algoritmo a um Problema do URI

O problema escolhido para a aplicação do Quick Search foi o problema 2651 do URI. É um problema de basicamente encontrar um padrão em textos grandes, caso o padrão fosse encontrado uma saída específica tinha que ser mostrada, em caso contrário, outra saída específica tinha era mostrada.

Adequações do Algoritmo para a Questão

```
/* Return 1 if the pattern was found, 0 otherwise */
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++;

            /* Pattern found, no need to continue search */
            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;
}
```

Adequações do Algoritmo para a Questão

```
int quick_search(string pattern, string tetx){

    /* Pre-processing stayed the same */
    pre_processing(pattern, tetx);

    return search(pattern, tetx);

}

int main(){

    string pattern = "zelda";

    string input;

    cin >> input;

    /* Convert input to 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

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.

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.

Exemplo Gráfico

Fase de Pré Processamento

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

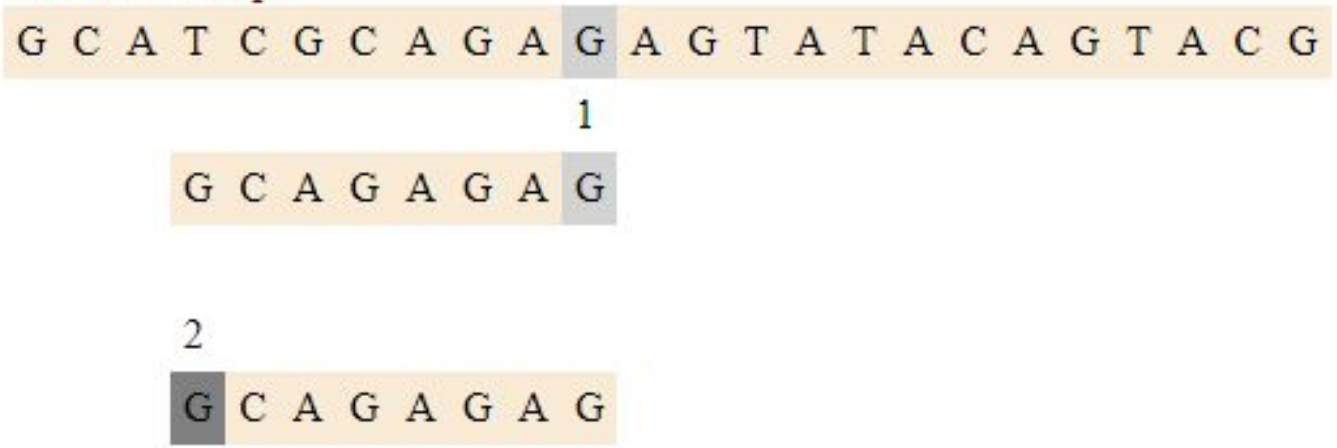
$shift = 2$

bmBc table used by Tuned Boyer-Moore algorithm.

First attempt

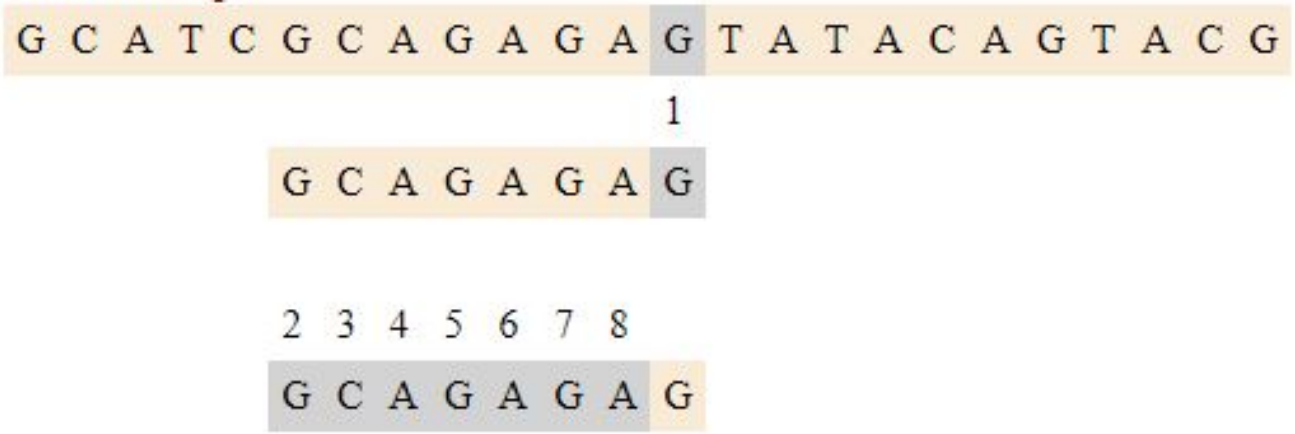
Shift by: 2 (*shift*)

Second attempt



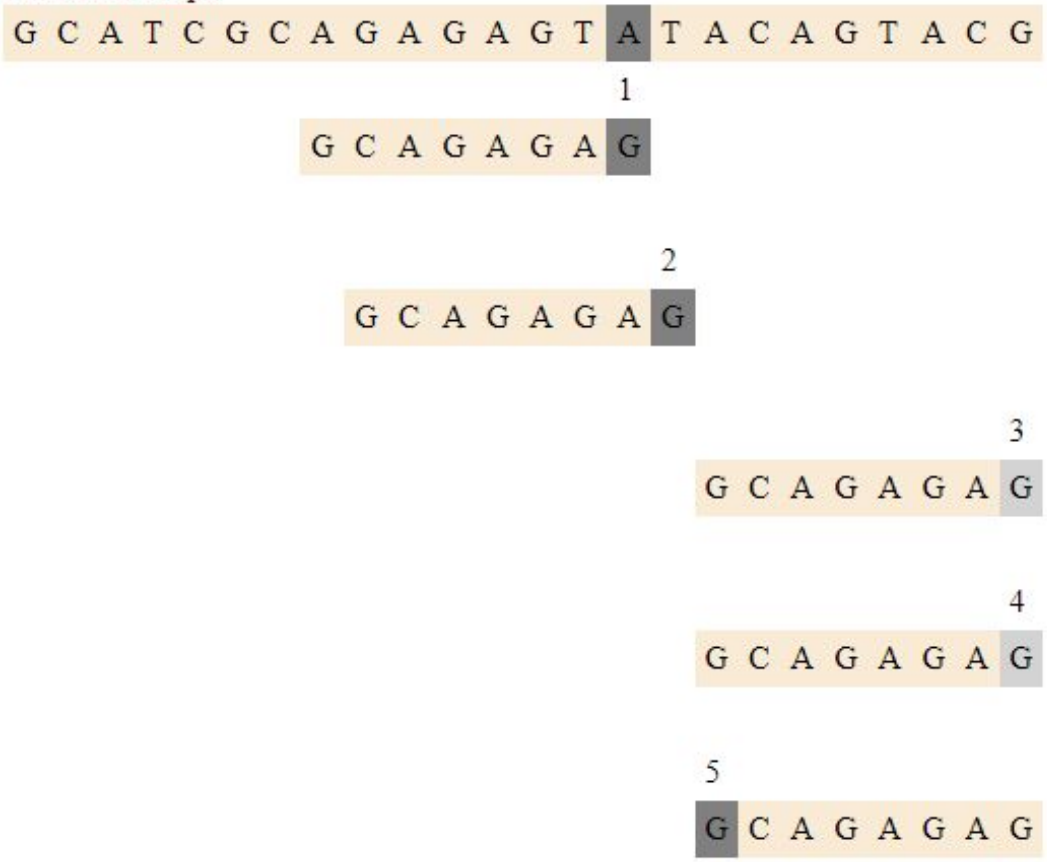
Shift by: 2 (*shift*)

Third attempt



Shift by: 2 (*shift*)

Fourth attempt



Resolução do Problema 2049 do Uri

```
// 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;
}
```

Resolução do Problema 2049 do Uri

```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);

    // 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)
        {
            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) return 1;


        j += shift; /* shift */
    }
    return 0;
```

Resolução do Problema 2049 do Uri

```c
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;
}
```

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).

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).

Zhu-Takaoka

Descrição

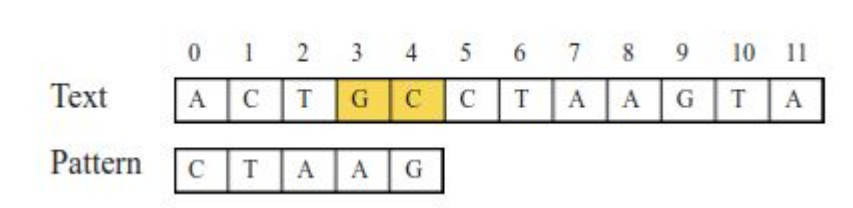
Zhu-Takaoka é um algoritmo de solução de problemas de correspondência de string que é uma variante do algoritmo de Boyer-Moore. O algoritmo apenas melhora a regra do caractere errado do algoritmo de Boyer-Moore.

Zhu e Takaoka modificaram o algoritmo de Boyer-Moore. Eles substituíram pela regra da substring dupla. A regra do sufixo correto continua sendo usada.

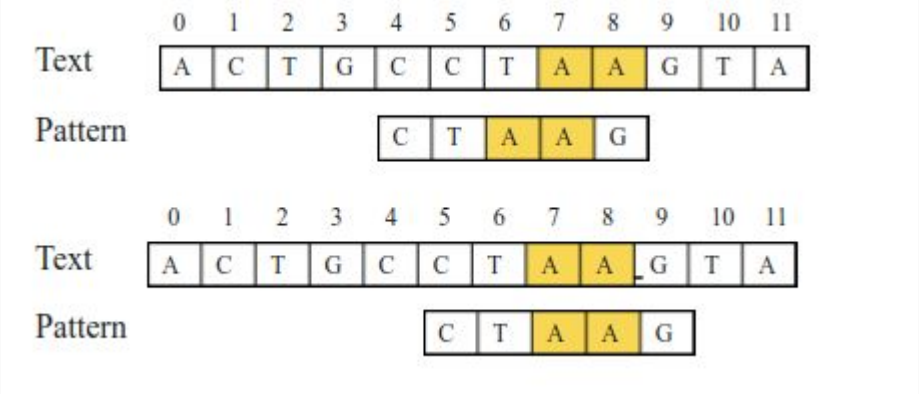
Exemplo Gráfico

Regra da substring dupla

- Considere o texto=ACTGCTAAGTA e o padrão=CTAAG.



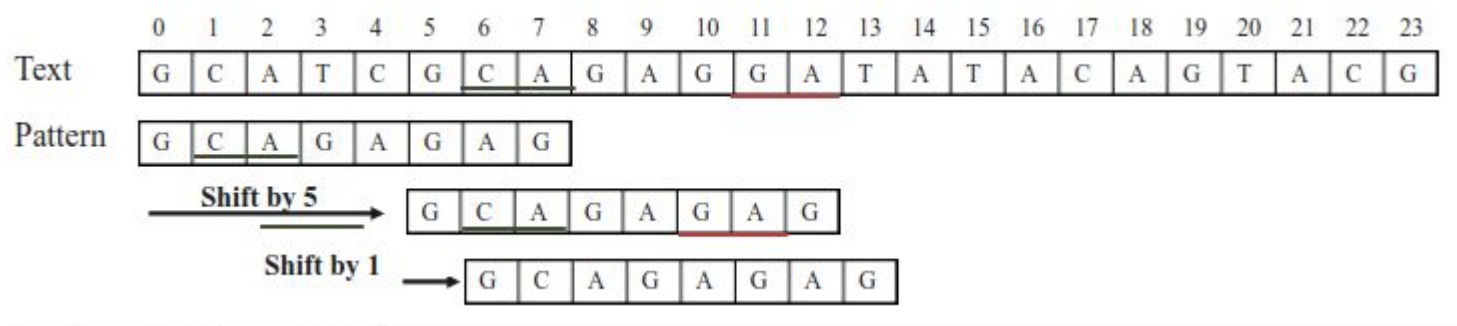
Não há GC em P.



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

Exemplo Gráfico

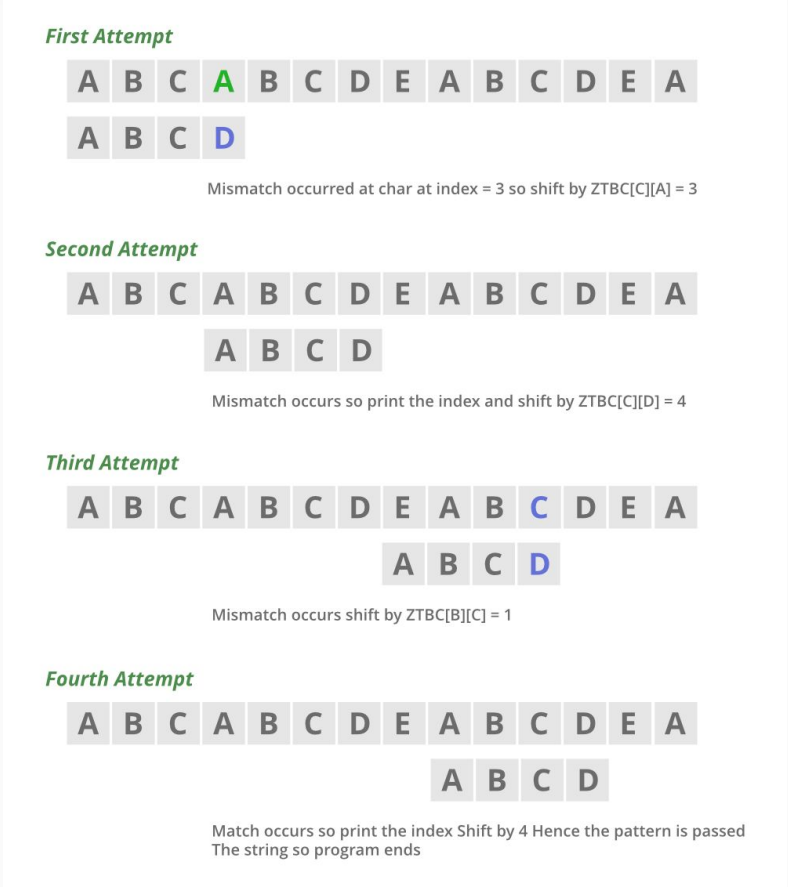
- Sempre que não ocorrer uma correspondência ou ocorrer uma correspondência completa, selecionamos a última substring dupla em T e procuramos pela localização mais à direita dessa substring dupla em P, se existente. Isso é feito construindo uma tabela ztBc.
- Exemplo:



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.

Exemplo Gráfico



Código em C++

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

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

int stringlen = 14;
int patternlen = 4;

int ZTBC[26][26];

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;
}
```

Código em C++

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

    while (j <= stringlen - patternlen) {

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

                // Pattern detected
                cout << "Pattern Found at " << j + 1 << endl;
                j += patternlen;
            }

            // Not detected
            else
                j += ZTBC[str[j + patternlen - 2]
                        - 'A']
                        [str[j + patternlen - 1]
                        - 'A'];
        }

    return 0;
}
```

Resolução do Problema 2049 do Uri

Zhu-Takaoka

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

/* Testes do uri: …

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;
}
```

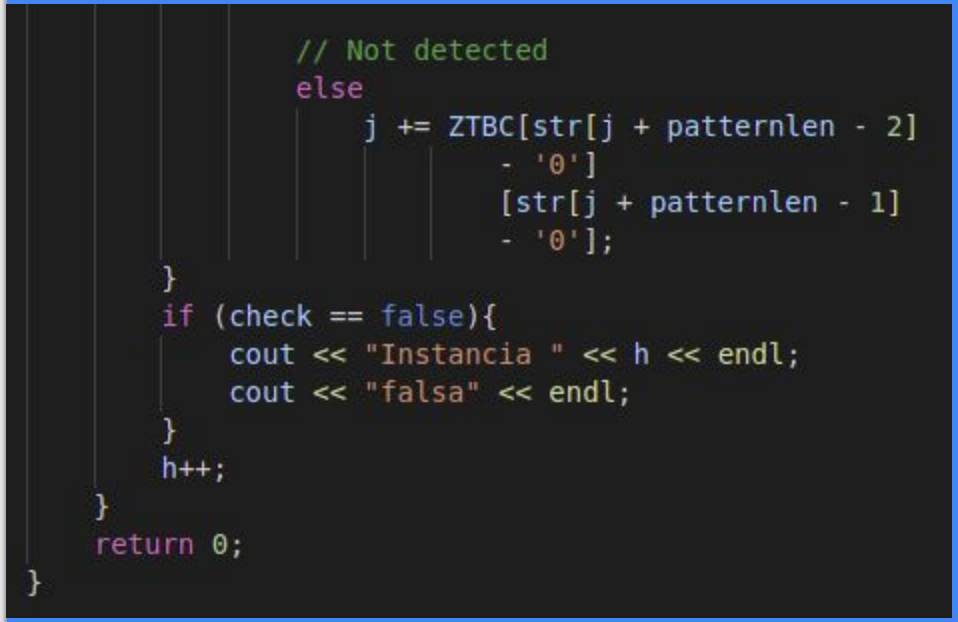
Resolução do Problema 2049 do Uri

Zhu-Takaoka

```
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) {
                    // Pattern detected
                    cout << "Instancia " << h << endl;
                    cout << "verdadeira" << endl;
                    /* cout << "Pattern Found at " << j + 1 << endl; */
                    j += patternlen;
                    check = true;
                }
```

Resolução do Problema 2049 do Uri

Zhu-Takaoka



```
                // Not detected
                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;
}
```

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.

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 Ansewers (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.

Referências

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.

Referências

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.

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.