Universidade Federal de São João del Rei
Departamento de Ciência da Computação
Algoritmo e Estrutura de Dados III

Aplicação e análise entre algoritmos de casamento de caracteres exatos.

Autores: Daniel Marques e Rian Wagner.

Universidade Federal de São João del Rei
Departamento de Ciência da Computação
Algoritmo e Estrutura de Dados III

1 Introdução:

Com o surgimento da internet e das páginas na web, o processamento de caracteres é fundamental para que tenhamos facilidades como os buscadores de páginas como o Google, por exemplo. Segundo a Internet Live Stats, existem mais de 1,88 bilhões de páginas webs [1]. Isso só é possível graças aos processos de manipulação de caracteres.

Uma parte fundamental do processamento de caracteres é a busca por casamentos exatos. Um casamento exato ocorre quando uma determinada sequência de caracteres é encontrada em uma outra sequência de caracteres, exatamente da mesma maneira. Isso significa que os caracteres devem estar em uma ordem específica e não pode haver diferenças entre eles. Os casamentos exatos são comumente utilizados em várias aplicações, como motores de busca, sistemas de autenticação, processamento de textos e análise de dados.

Existem várias aplicações reais em que o casamento de caracteres desempenha um papel importante. Aqui estão algumas delas:

- Motores de busca: Os motores de busca, como o Google, utilizam algoritmos de casamento de caracteres para encontrar resultados relevantes com base nas consultas dos usuários. Eles comparam as palavras-chave pesquisadas com o conteúdo indexado da web para retornar os resultados mais relevantes.
- Sistemas de autenticação: Os sistemas de autenticação, como senhas e reconhecimento biométrico, podem usar o casamento exato para verificar se uma sequência de caracteres inserida pelo usuário corresponde à sequência armazenada previamente. Isso ajuda a garantir a segurança e a proteção dos dados.
- Processamento de linguagem natural: O casamento de caracteres é usado em várias tarefas de processamento de linguagem natural, como extração de informações, análise de sentimentos e chatbots. É utilizado para identificar palavras-chave, expressões ou estruturas gramaticais em texto para realizar análises e fornecer respostas relevantes.

O trabalho proposto pede que sejam implementadas 3 estratégias para resolver problemas envolvendo casamento exato de caráter. As estratégias escolhidas foram, a implementação de um algoritmo de força bruta, o algoritmo de Knuth-Morris-Pratt (KMP) e o algoritmo de Boyer-Moore-Horspool (BMH).

2 Formato de entrada e saída:

A entrada do programa será lida através de um arquivo de texto (*entrada.txt*), sucedido pelo número da estratégia desejada. O arquivo deverá conter o número de casos a serem testados (T) e cada caso teste sucessivamente. Os casos testes deverão conter duas sequências de caracteres, separados por um espaço. As sequências de caracteres são a sequência que descreve a habilidade (Padrão) e a sequência encontrada na pedra

(texto), respectivamente. As duas sequências poderão ser compostas de caracteres de a à z desde que sejam minúsculas. Para a sequência de habilidades (padrão) é possível ter até 100 caracteres, já para a sequência da pedra é possível ter até 10000 caracteres. É importante ressaltar que a pedra é cíclica, logo a sequência pode não estar na ordem esperada.

Exemplo de entrada:

```
4
ava av
patapon npatapatapatapo
isitfriday ohnoitisnt
haskell lleksah
```

***Retirado das especificações do trabalho

A saída deverá ser dada por meio de um arquivo de texto e deverá indicar a posição onde ocorreu o casamento das sequências (S [posição onde ocorreu o casamento], caso não ocorra, deverá sinalizar ao usuário (N).

Exemplo de saída:

```
S1
S10
N
S7
```

***Retirado das especificações do trabalho

3 Modelagem e principais funções :

O casamento de caracteres ou de cadeia de caracteres parte de um pressuposto bem simples. Para que exista casamento exato, como é pedido nas especificações do trabalho, o caractere de determinada posição do texto deve ser correspondente ao caractere da mesma posição do padrão, esse processo deve ser validado para todos ou caracteres próximos e anteriores ao da posição em questão.

Em geral, os algoritmos de casamento exatos são usados para buscar ocorrências exatas de um padrão em um texto. Eles buscam encontrar todas as posições no texto onde o padrão ocorre, sem permitir variações ou erros, diferentemente dos algoritmos de casamento aproximado.

No problema proposto, todas as abordagens são direcionadas para um objetivo em comum que é o de verificar se a cadeia de caracteres que define uma habilidade (padrão), está presente na cadeia de caracteres definida na pedra (texto). No entanto, existem duas possibilidades que exigem que os códigos dos algoritmos já estudados fossem adaptados para o trabalho em questão. O fato de haver a possibilidade das sequências na pedra estarem ao contrário e as pedras serem cíclicas, assim como suas sequências de caracteres, fez com que os algoritmos fosse adaptados para solucionar o problema considerando esses casos

3.1 Modelagem do algoritmo de força bruta:

O algoritmo de busca de força bruta é uma abordagem simples e direta para encontrar uma sequência em uma string. Ele segue um método básico de verificação, comparando a sequência desejada com cada possível posição da string até encontrar uma correspondência ou percorrer todas as posições possíveis.

A ideia por trás do algoritmo é percorrer a string em busca da sequência desejada, começando da posição inicial e verificando caractere por caractere. Se todos os caracteres da sequência corresponderem aos caracteres correspondentes na string a partir dessa posição, consideramos a sequência encontrada.

O algoritmo funciona da seguinte maneira: dado uma sequência a ser encontrada e uma sequência na qual faremos a busca, iniciamos na posição 0 da sequência. Em seguida, percorremos um laço de repetição que varia de 0 até o comprimento do padrão menos o comprimento da sequência da pedra. Isso nos dá todas as posições possíveis onde a sequência pode começar na string.

Dentro de uma estrutura de repetição, comparamos cada caractere da sequência com o caractere correspondente no padrão, começando pela posição atual. Se todos os caracteres coincidirem, significa que encontramos a sequência e retornamos à posição atual. Caso contrário, avançamos para a próxima posição na sequência de texto e repetimos o processo de comparação. Continuamos assim até encontrar uma correspondência ou percorrer todas as posições possíveis no texto sem sucesso.

Se chegarmos ao final do loop sem encontrar a sequência, concluímos que ela não está presente na string. Nesse caso, retornamos um valor indicando que a sequência não foi encontrada.

Para realizarmos as comparações entre o padrão e o texto, utilizamos os recursos da biblioteca `string.h`, as funções: `strcpy`, que copia uma string; `strcat`, que concatenar uma string e `strlen`, que retorna o tamanho da string, foram indispensáveis para a realização do algoritmo.

Embora o algoritmo de força bruta seja simples e fácil de entender, sua eficiência é limitada, especialmente para strings grandes. Isso ocorre porque ele precisa percorrer todas as posições possíveis na string várias vezes. Em casos onde a busca precisa ser rápida, existem algoritmos mais eficientes.

3.1.1 Principais funções:

1ª) Função forçabruta(): Função única que implementa o algoritmo. Nessa função, todas as informações a respeito das sequências de texto e padrão, passadas por parâmetro, são obtidas e as posições dos caracteres são comparadas.

```
void forca_bruta(char *padrao, char *pedra, FILE *saida) {
    int tam_padrao = strlen(padrao);
    int tam_pedra = strlen(pedra);
    for (int i = 0; i < tam_pedra; i++) {
        int casamento = 1;
```

```
        for (int j = 0; j < tam_padrao; j++) {
            if (pedra[(i + j) % tam_pedra] != padrao[j]) {
                casamento = 0;
                break;
            }
        }
        if (casamento == 0) {
            casamento = 1;
        for (int j = 0; j < tam_padrao; j++) {
                if (pedra[(i - j + tam_pedra) % tam_pedra] != padrao[j]) {
                    casamento = 0;
                    break;
                }
            }
        }
        if (casamento == 1) {
            fprintf(saida, "S %d\n", i + 1);
            return;
        }
    }
    fprintf(saida, "N\n");
}
```

3.2 Modelagem do algoritmo de Knuth-Morris-Pratt (KMP):

O algoritmo Knuth-Morris-Pratt (KMP) é um algoritmo eficiente para busca de padrões em sequência de caracteres. Ele utiliza uma abordagem baseada em pré-processamento da sequência de busca para evitar comparações desnecessárias durante a busca.

A ideia central do algoritmo KMP é construir uma tabela de salto (também conhecida como tabela de falhas ou tabela de borda) que armazena informações sobre os padrões existentes na sequência de busca [3]. Essa tabela é utilizada para determinar os deslocamentos apropriados durante a busca, eliminando a necessidade de retroceder a posição do padrão sempre que uma correspondência falhar.

O algoritmo KMP se desenvolve dentro de dois passos muito importantes. Primeiro é a construção da tabela de salto que armazena o comprimento do maior prefixo próprio que é também um sufixo para cada posição do padrão. Ela é construída de forma iterativa, comparando o padrão atual com seus prefixos anteriores e atualizando o comprimento do prefixo na tabela de salto. O processo continua até que a tabela esteja completa, fornecendo os deslocamentos corretos para cada posição do padrão. Por conseguinte, a busca utilizando a tabela de salto, o texto comparada com o padrão a partir da posição inicial. Se os caracteres correspondentes são encontrados, ambos avançam para a próxima posição. Caso contrário, quando ocorre uma não correspondência.

Em vez de retroceder completamente o padrão, o algoritmo KMP utiliza as informações da tabela de salto para realizar avanços significativos. Isso evita repetições desnecessárias na busca e melhora a eficiência do algoritmo [4]. É totalmente possível implementar o algoritmo sem utilizar os recursos da tabela de salto, mas ele diminui sua eficiência.

O processo de comparação e deslocamento continua até que uma correspondência completa do padrão na sequência de busca seja encontrada ou até que a busca percorra toda a sequência sem sucesso.

Para implementar o algoritmo KMP, também utilizamos as funções e manipulação de strings da biblioteca `string.h`.

Para tratar os casos de sequências cíclicas ou ao contrário, utilizamos uma função auxiliar que verifica essa condição.

3.2.1 Principais funções:

1ª) Função tabela_salto: Constrói a tabela de saltos para armazenar os deslocamentos dentro da sequência de texto

```
int *tabela_salto(char *padrao) {
    int tam = strlen(padrao);
    int *tabSalto = (int *)malloc(sizeof(int) * tam);
    int i = 1;
    int j = 0;
    tabSalto[0] = 0;
    while (i < tam) {
        if (padrao[i] == padrao[j]) {
            tabSalto[i] = j + 1;
            i++;
            j++;
        }
        else {
            if (j != 0) {
                j = tabSalto[j - 1];
            }
            else {
                tabSalto[i] = 0;
                i++;
             }
        }
    }
    return tabSalto;
}
```

2ª) Função verificar_padrao_kmp(): Função auxiliar para verificar se a sequência de texto se encaixa nos casos especiais de sequências cíclicas ou contrárias

```
int verificar_padrao_kmp(char *padrao, char *pedra) {
    int tam_pedra = strlen(pedra);
    char *pedra_redonda = (char *)malloc(sizeof(char) * (2 * tam_pedra + 1));
    strcpy(pedra_redonda, pedra);
    strcat(pedra_redonda, pedra);
    int posicao = kmp(pedra_redonda, padrao);
    free(pedra_redonda);
    if (posicao != -1){
        return posicao;
    }
    char *pedra_invertida = (char *)malloc(sizeof(char) * (tam_pedra + 1));
```

```
    for (int i = 0; i < tam_pedra; i++) {
        pedra_invertida[i] = pedra[tam_pedra - i - 1];
    }
    pedra_invertida[tam_pedra] = '\0';
    posicao = kmp(pedra_invertida, padrao);
    free(pedra_invertida);
    if (posicao != -1){
        return tam_pedra - posicao - 1;
    }

    return posicao;
}
```

3ª) Função kmp(): Função que implementa o algoritmo KMP utilizando a tabela de saltos.

```
int kmp(char *texto, char *padrao) {
    int tam_texto = strlen(texto);
    int tam_padrao = strlen(padrao);
    int *tabSalto = tabela_salto(padrao);
    int i = 0;
    int j = 0;
    while (i < tam_texto) {
        if (texto[i] == padrao[j]) {
            i++;
            j++;
        }
        if (j == tam_padrao) {
            free(tabSalto);
            return i - j;
        }
        if (i < tam_texto && texto[i] != padrao[j]) {
            if (j != 0) {
                j = tabSalto[j - 1];
            }
            else {
                i++;
            }
        }
    }
    free(tabSalto);
    return -1;
}
```

3.3 Modelagem algoritmo Boyer-Moore-Horspool (BMH):

O algoritmo BMH (Boyer-Moore-Horspool) é uma abordagem eficiente para busca de padrões em um texto e para a realização do trabalho proposto [4]. Ele utiliza uma estratégia de salto baseada em dois passos: a construção da tabela de salto e a comparação reversa. Assim como o KMP implementado na solução do problema, o BMH também utiliza uma tabela de salto, mas as implementações são diferentes devido a comparação reversa.

A tabela de salto é uma estrutura essencial no algoritmo BMH, diferentemente do KMP. Ela é construída analisando-se o padrão da direita para a esquerda e determina os

deslocamentos a serem realizados em caso de não correspondência entre caracteres do padrão e do texto. Essa tabela permite fazer grandes saltos no texto durante a busca, aproveitando-se das informações pré-calculadas. Isso torna o algoritmo eficiente em termos de tempo de execução, especialmente em casos onde o padrão ocorre com pouca frequência no texto [5].

Durante a busca, o algoritmo BMH compara o padrão com o texto da direita para a esquerda. Se houver uma correspondência completa do padrão na posição atual do texto, o algoritmo retorna essa posição como resultado da busca. Caso contrário, ele utiliza a tabela de salto para determinar o deslocamento a ser aplicado no texto. Esse deslocamento é calculado com base no caractere do texto que não corresponde ao caractere correspondente no padrão.

Em resumo, o algoritmo BMH utiliza a tabela de salto para realizar deslocamentos eficientes durante a busca de padrões em um texto. Ele aproveita informações pré-calculadas para evitar comparações desnecessárias, o que resulta em um desempenho superior em comparação com abordagens ingênuas, como a busca de força bruta. O BMH é amplamente utilizado em diversas aplicações de busca de padrões devido à sua eficiência e rapidez.

3.3.1 Principais funções:

1ª) Função tabelaSalto(): Constrói a tabela de saltos do algoritmo BMH e preenche com os valores dos respectivos saltos para cada caractere do padrão

```
int* tabelaSalto(char* padrao, int tam_padrao) {
    int* tabela = (int*)malloc(sizeof(int) * 256);
    for (int i = 0; i < 256; i++) {
        tabela[i] = tam_padrao;
    }
    for (int i = 0; i < tam_padrao - 1; i++) {
        tabela[padrao[i]] = tam_padrao - i - 1;
    }
    return tabela;
}
```

2ª) Função verifica_max(): Função auxiliar que verifica o deslocamento máximo

```
int verifica_max(int a, int b) {
    if (a > b) {
        return a;
    }
    return b;
}
```

3ª Função verifica_padrao_bmh(): Função auxiliar para tratar os casos de sequências cíclicas ou ao contrário.

```
int verificar_padrao_bmh(char* padrao, char* pedra) {
    int tam_padrao = strlen(padrao);
    int tam_pedra = strlen(pedra);
    int posicao = -1;
    posicao = bmh(pedra, tam_pedra, padrao, tam_padrao);
    if (posicao != -1) {
        return posicao;
    }
    char* pedra_dupla = (char*)malloc(sizeof(char) * (2 * tam_pedra + 1));
    strcpy(pedra_dupla, pedra);
    strcat(pedra_dupla, pedra);
    posicao = bmh(pedra_dupla, 2 * tam_pedra, padrao, tam_padrao);
    if (posicao != -1) {
        int pos_ciclica = (posicao % tam_pedra);
        free(pedra_dupla);
        return pos_ciclica;
    }
    char* padrao_inverso = (char*)malloc(sizeof(char) * (tam_padrao + 1));
    for (int i = 0; i < tam_padrao; i++) {
        padrao_inverso[i] = padrao[tam_padrao - 1 - i];
    }
    padrao_inverso[tam_padrao] = '\0';
    posicao = bmh(pedra, tam_pedra, padrao_inverso, tam_padrao);
    if (posicao != -1) {
        int pos_inversa = tam_pedra - posicao - 1;
        free(padrao_inverso);
        return pos_inversa;
    }
    free(padrao_inverso);
    return -1;
}
```

4ª Função bmh(): Função que implementa o algoritmo BMH:

```
int bmh(char* texto, int tam_texto, char* padrao, int tam_padrao) {
    int* pTabelaSalto = tabelaSalto(padrao, tam_padrao);
    int shift = 0;
    int i = tam_padrao- 1;
    int j;
    while (i < tam_texto) {
        j = tam_padrao - 1;
        while (j >= 0 && texto[i] == padrao[j]) {
            i--;
            j--;
        }
        if (j < 0) {
            free(pTabelaSalto);
            return i + 1;
        }
        shift = verifica_max(pTabelaSalto[texto[i]], tam_padrao - j);
        i += shift;
    }
    free(pTabelaSalto);
    return -1;
}
```

4 Análise das complexidades:

Universidade Federal de São João del Rei
Departamento de Ciência da Computação
Algoritmo e Estrutura de Dados III

Para determinar a complexidade dos algoritmos, é necessário analisar quanto tempo leva para executar o pior caso em relação ao tamanho da entrada.

Algoritmo de Força Bruta:

O tempo necessário para buscar o padrão em uma sequência é dado pela multiplicação do número de posições na sequência pelo número de posições no padrão. A complexidade de tempo é O(n * m), onde n é o tamanho da sequência e “m” é o tamanho do padrão. Portanto, a complexidade total do algoritmo de Força Bruta é O(n * m), onde “n” é o tamanho da sequência e “m” é o tamanho do padrão.

Algoritmo KMP (Knuth-Morris-Pratt):

O algoritmo KMP tem uma complexidade de tempo linear tanto para construir a tabela de salto quanto para realizar a busca do padrão na sequência. A construção da tabela de salto leva tempo proporcional ao tamanho do padrão (O(m)), enquanto a busca leva tempo proporcional ao tamanho da sequência (O(n)). Portanto, a complexidade total do algoritmo KMP é O(m + n), onde “m” é o tamanho do padrão e “n” é o tamanho da sequência. O desempenho do algoritmo é influenciado pelo tamanho do padrão e da sequência, sendo mais eficiente para padrões e sequências menores

Algoritmo BMH (Boyer-Moore-Horspool):

A complexidade de tempo do algoritmo Boyer-Moore-Horspool (BMH) é geralmente O(n * m), onde n é o tamanho da sequência e m é o tamanho do padrão. Isso ocorre devido ao tempo necessário para construir a tabela de salto (O(m)) e a busca usando essa tabela (O(n * m)). No entanto, na prática, o BMH pode ser eficiente em muitos casos, especialmente quando há desigualdades frequentes entre os caracteres do padrão e da sequência.

Em termos de complexidade de tempo, o algoritmo KMP é o mais eficiente, seguido pelo BMH. O algoritmo de Força Bruta é o menos eficiente, especialmente para sequências grandes. No entanto, é importante considerar outros fatores, como o tamanho do padrão, o tipo de dados e o contexto específico do problema ao escolher o algoritmo mais adequado.

5 Resultados:

Com exceção dos 3 exemplos já dados pelo professor, as demais entradas foram feitas pelos integrantes do grupo com o intuito de gerar tratamentos de erros diversos.

Ambiente de compilação e execução : “WSL 2 Ubuntu 22.04 LTS”
Compilador : MinGW gcc
Processador : “Ryzen 5 5600H” e “Intel(R) Core(TM) i5-7200U ” .

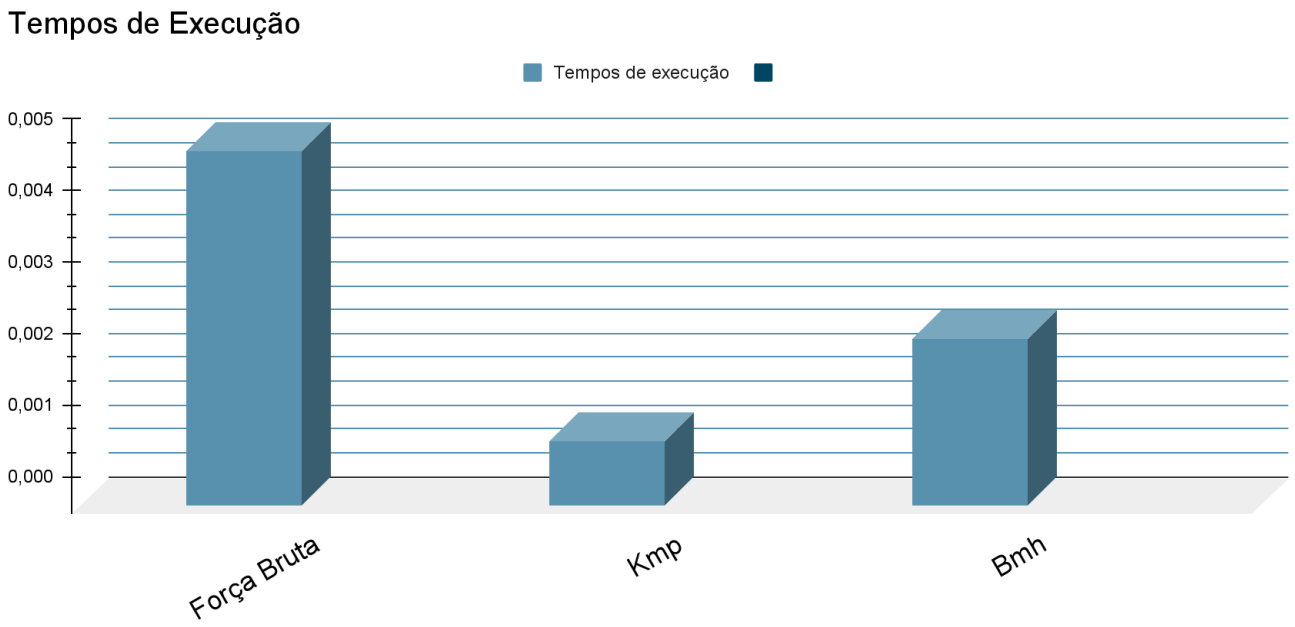
Padrão	Pedra	Validação do Casamento	Posição do Casamento
ava	av	S	1
patapon	npatapatapatapo	S	10
isitfriday	ohnoitisnt	N	
haskell	lleksah	S	7
love	esthelovaisintheairlov	S	19
malloc	locmal	S	4
dani	inad	S	4
nida	dani	S	3

Criamos algumas entradas tentando visar alguns diferentes comportamentos da sequência da pedra tendo em vista que por ser cíclica ela tinha algumas nuances. Então utilizamos sequências invertidas de diferentes formas e forçamos casamentos envolvendo a adjacência do último caractere com o primeiro. Além dessas 8 entradas, foi criado um caso de teste com muitos caracteres para simular um certo ‘delay’ no intuito da melhor avaliação dos tempos de execução.

5.1 Tabela dos tempos de execução:

Algoritmos	Tempo de Execução em segundos.
Força Bruta	0,004944s
Knuth-Morris-Pratt (KMP)	0,000904s
Boyer-Moore-Horspool (BMH)	0,002318s

5.1.2 Análise dos tempos de execução:



Este é um gráfico de colunas que mostra uma comparação entre as medições de tempo, podemos perceber que a coluna do algoritmo KMP é a menor dentre os algoritmos de casamento de caracteres, ele se comportou aproximadamente 81,7% mais rápido que o algoritmo de força bruta, e 61,0% mais rápido que o algoritmo BMH. Isso é devido a sua tabela de salto ou tabela LPS (Longest Proper Prefix which is also Suffix), essa tabela permite que o algoritmo "pule" comparações já realizadas quando ocorre uma não correspondência entre caracteres.

O BMH também possui algumas otimizações que reduzem o número de comparações, ele utiliza de informações sobre o padrão para realizar saltos maiores durante a busca, com base na ocorrência de não correspondências.

6 Conclusão:

O problema de busca de padrões em sequências é uma tarefa comum em ciência da computação e possui várias abordagens algorítmicas. O algoritmo KMP (Knuth-Morris-Pratt) e o algoritmo BMH (Boyer-Moore-Horspool) são duas soluções amplamente utilizadas para lidar com esse problema.

O algoritmo KMP se destaca pela sua eficiência e capacidade de evitar comparações desnecessárias durante a busca, graças à utilização da tabela de salto. Ele oferece uma complexidade de tempo linear, o que o torna uma opção atraente para casos em que o padrão e a sequência podem ser grandes.

Por outro lado, o algoritmo BMH é conhecido por sua simplicidade e eficiência em cenários onde há desigualdades frequentes entre os caracteres do padrão e da sequência. Embora sua complexidade de tempo seja geralmente considerada O(n * m), o BMH pode ser uma escolha viável em situações específicas.

Em resumo, a escolha entre o algoritmo KMP e o algoritmo BMH depende das características do problema em mãos, como o tamanho do padrão, a natureza dos caracteres envolvidos e as restrições de desempenho. Ambos os algoritmos oferecem soluções eficientes e amplamente utilizadas para o desafio de busca de padrões em sequências.

7 Bibliotecas

```c
#include <stdio.h> //Biblioteca padrão de entrada e saída.

#include <stdlib.h> //Funções de alocação de memória.

#include <string.h> //Funções para manipulação de strings.

#include <sys/resource.h> //Funções de medição de tempo.
```

Referências:

[1] INTERNET LIVE STATS. [Internet Live Stats]. Disponível em: https://www.internetlivestats.com/. Acesso em: 05/06.
[2] SAAD, Daniel. Casamento de padrões. Disponível em: https://danielsaad.com/TEA-BCC-IFB/notas-de-aula/strings/assets/casamento-de-padroes.pd. Acesso em: 05/06.
[3] Wikipedia. Knuth-Morris-Pratt algorithm. Disponível em: https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm. Acesso em: 09/06.
[4] Robert Sedgewick e Kevin Wayne (Departamento de Ciência da Computação da Universidade de Princeton) Título: Algorithms, Part I Disponível em: https://algs4.cs.princeton.edu/home/ Acesso em: 12/06.
[5] Boyer, R. S.; Moore, J. S. "A Fast String Searching Algorithm". Communications of the ACM, v. 20, n. 10, p. 762-772, 1977. Disponível em: https://dl.acm.org/doi/10.1145/359842.359859. Acesso em: 12/06.