Ciência da Computação

Algoritmos e Estruturas de Dados III

Documentação Trabalho Prático 3 - Strings Mágicas

Davi dos Reis de Jesus, Gabriel de Paula Meira

2023.1

1 Introdução

Com o objetivo de desvendar o mistério das pedras mágicas do antigo reino de Xulambis, uma equipe de pesquisadores empenha-se em comprovar a teoria de que, quando uma sequência específica de símbolos é inscrita em uma pedra, pode conferir poderes àquele que a possui. Dessa forma, a busca concentra-se em identificar essas sequências, que se encontram dispostas de forma circular na superfície das pedras.

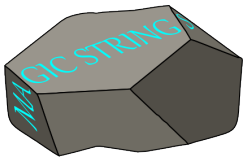

Figura 1: Ilustração lúdica do problema

O desafio a ser resolvido é tido como um problema de casamento de cadeia de caracteres. É possível obter a solução determinando a sequência que fornece os poderes como o padrão, e os símbolos inscritos na pedra como o texto.

A resolução do problema proposto possui uma relevância significativa para a área da computação, tendo em vista que o processamento de caracteres é uma área de extrema importância na atualidade, pois desempenha um papel fundamental em diversas tarefas tais como processamento de linguagem natural, análise de texto, reconhecimento de padrões, algoritmos de busca, entre outras aplicações.

Com o crescente volume de informações disponíveis na web, métodos eficientes de acesso, busca e disseminação desses dados são cada vez mais necessários. Encontrar um conteúdo específico entre essa vasta quantidade de dados pode tornar-se uma tarefa desafiadora.

Diante dessas considerações, a presente documentação tem como objetivo desenvolver e analisar diferentes abordagens para solucionar o desafio. Ao todo foram implementadas quatro estratégias. Além do método de força bruta, estão algoritmos amplamente conhecidos para resolução de problemas envolvendo casamento de cadeia de caracteres, são eles: KMP (Knuth-Morris-Pratt), BMH (Boyer-Moore-Horspool) e Shift-And.

2 Arquitetando a solução

É necessário, antes de iniciar a programação, planejar o funcionamento do programa em cada uma das etapas, a fim de garantir a qualidade do software do início ao fim do projeto, facilitando o processo de depuração e manutenção. O programa foi dividido em quatro etapas principais ilustradas pelo fluxograma da Figura 2.

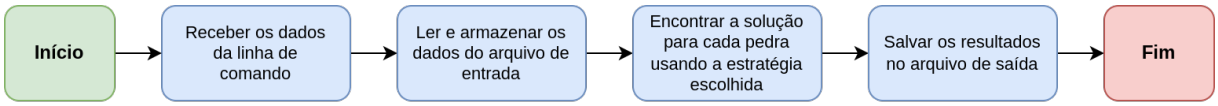

Figura 2: Fluxograma de funcionamento do programa

Essa divisão proporciona uma visão clara e estruturada do processo, no qual cada etapa representa um conjunto de tarefas e funcionalidades específicas a serem implementadas permitindo uma abordagem mais organizada e eficiente para o desenvolvimento.

2.1 Receber dados da linha de comando

Como mencionado anteriormente, o programa deve permitir que o usuário escolha a estratégia a ser utilizada para encontrar as soluções de todas as pedras contidas no arquivo de entrada. Esta operação requer o uso dos argumentos da linha de comando como demonstrado abaixo.

```
./tp3 {arquivoEntrada} {estrategia}
```

Para receber esses dados é necessário utilizar os parâmetros **argc** e **argv**, recebidos pela função **main()**, verificando se as entradas são válidas para o funcionamento correto do programa.

2.2 Ler pedras do arquivo de entrada

Após obter o endereço do arquivo de entrada, é efetuada a leitura completa do mesmo utilizando a função **fscanf()**, pertencente à biblioteca **stdlib.h**.

É definido que a primeira linha contém a quantidade total de casos de teste. O programa deve alocar o espaço necessário para armazenar todos as pedras, cada qual com suas respectivas cadeias de caracteres.

Cada linha do arquivo de entrada contém um padrão de até 10^2 caracteres e um texto de até 10^4, possibilitando que os vetores do tipo *char* possam ter um tamanho fixo. Outro ponto importante a se destacar é a padronização do alfabeto, que contém apenas letras minúsculas de a-z, reduzindo o uso de memória em algumas estratégias.

2.3 Encontrar a solução para cada pedra

As quatro estratégias devem possibilitar o casamento de caracteres com padrões invertidos e/ou cíclicos, modificando funcionalidades específicas conforme a estrutura lógica de cada um.

Visando melhorar a manutenibilidade, adotou-se uma padronização para todas as implementações, que possuem um cabeçalho idêntico ao exemplo abaixo variando apenas o nome da função. São recebidos, por parâmetro, duas cadeias de caracteres (padrão e texto) e uma variável de estado binário para controlar o retorno invertido ou normal. Em caso de busca mal-sucedida o índice de retorno deve ser -1.

```
int function(char* substring, char* string, int reverse);
```

Essa decisão permitiu o uso de um ponteiro para a função designada, o que facilitou o uso da estratégia escolhida pelo usuário evitando repetições desnecessárias de trechos iguais de código. O funcionamento dos algoritmos é descrito com detalhes na seção 4 da documentação.

Para monitorar o tempo decorrido para encontrar cada solução foi utilizada função **gettimeofday()** da biblioteca **time.h**, os resultados obtidos dos testes estão localizados na seção 5.

2.4 Salvar os resultados no arquivo de saída

A solução encontrada para cada pedra é salva em um arquivo de saída padronizado com o mesmo nome do arquivo de entrada seguido da extensão ".out". A função **fprintf**, também da biblioteca **stdio.h**, foi utilizada para a escrita dos dados.

Após a conclusão do programa, todas as estruturas de dados dinamicamente alocadas são liberadas para que não ocorra vazamentos de memória indesejados.

3 Estruturas de dados

Visando garantir uma organização eficiente e uma maior escalabilidade do software, foram criadas estruturas de dados destinadas unicamente para armazenar os dados de cada pedra. Isso permite que as informações possam ser facilmente acessadas e manipuladas durante toda a execução do programa.

```
#define MAX_CHAR_HABILITY 100
#define MAX_CHAR_DESCRIPTION 10000

typedef struct {
    char hability[MAX_CHAR_HABILITY + 1];
    char description[MAX_CHAR_DESCRIPTION + 1];
} Stone;

typedef struct {
    Stone* data;
    int length;
} StoneArray;
```

Em conjunto com a implementação dos tipos abstratos de dados, foram desenvolvidos métodos para a criação (alocação de memória) e exclusão (liberação de memória) dos dados de todas as pedras do vetor.

```
StoneArray createStoneArray(int length);
void freeStoneArray(StoneArray* array);
```

Um diferencial da problemática proposta é a necessidade de que as cadeias de caracteres sejam circulares, ou seja, o último caractere não finaliza a cadeia, que se inicia novamente a partir do primeiro. Esta propriedade possibilita a existência de padrões que não estejam em uma sequência contínua dentro do texto.

A medida adotada para resolver a adversidade foi adaptar os algoritmos para a verificação de caracteres em índices maiores que o tamanho do texto, transformando-os em índices que recomeçam a partir do final. Esse processo utiliza o operador de resto de divisão “%”, assim como exemplifica a Figura 3.

```
char texto[9] = "CIRCULAR\0";
int i = 13;
char letra = texto[i % strlen(texto)];
```

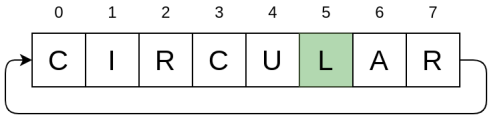

Figura 3: Exemplo de leitura circular em uma cadeia de caracteres

Para lidar com padrões invertidos, todas as pedras são testadas duas vezes: com o padrão normal e invertido. Com as duas soluções obtidas, é retornada a menor, garantindo a precisão dos resultados ao final de cada busca realizada. A diferença existente entre as formas de execução torna necessário, ao inverter o padrão, retornar o índice final da cadeia em vez do inicial.

4 Estratégias de resolução

Conforme visto em seções anteriores, foram elaboradas quatro estratégias distintas para encontrar as soluções de cada pedra de entrada: força bruta, KMP, BMH, Shift-And. Cada algoritmo possui suas peculiaridades e adaptações realizadas para encontrar padrões de até 10^2 caracteres em disposição que pode estar invertida e/ou circular em textos de até 10^4 caracteres.

As análises matemáticas realizadas na documentação para a complexidade dos algoritmos utilizam m como o tamanho do padrão e n como o tamanho do texto.

4.1 Força bruta

A estratégia mais simples é testar todas as combinações possíveis para formar o padrão desejado, letra por letra, assim como demonstra a Figura 4. Toda vez que um caractere do texto corresponde ao do padrão, é testada a posição seguinte, sucessivamente até que o padrão seja encontrado ou o texto chegue ao fim, em caso de falha a próxima posição do texto é analisada.

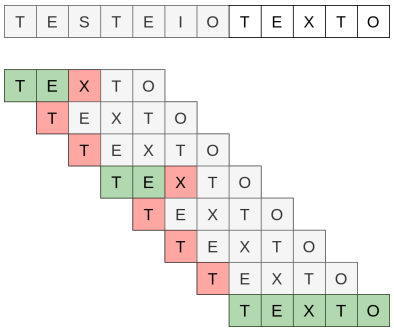

Figura 4: Demonstração do algoritmo de força bruta

O problema dessa estratégia é a grande quantidade de verificações realizadas, podendo, no pior caso, verificar o padrão inteiro para cada caractere do texto, o que atribui ao algoritmo a complexidade $O(mn)$, tornando-o ineficiente para casos em quando tanto o padrão quanto o texto são extensos.

Para possibilitar o casamento de cadeias circulares, basta garantir que a posição a ser analisada no texto não ultrapasse a quantidade de caracteres existentes, voltando sempre ao início. A Figura 5 demonstra como esse processo é realizado usando o operador de resto de divisão mencionado na seção 3.

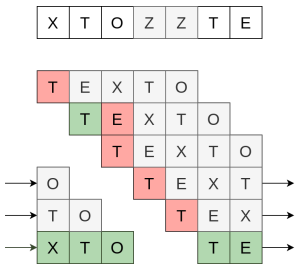

Figura 5: Demonstração do algoritmo de força bruta em cadeias circulares

Essa alteração pode adicionar à estratégia $(m-1) \times n$ comparações, o que não altera a complexidade final. O Algoritmo 1 descreve a implementação de todo o processo descrito.

Algorithm 1 Força bruta para casamento de padrão em texto circular

```
function ForcaBruta(padrao, texto, inverso)
    for i ← 0, tamanho(texto) − 1 do
        for j ← 0, tamanho(padrao) − 1 do
            if padrao[i] ≠ texto[(j + 1) mod n] then
                break
            end if
        end for
        if j = tamanho(padrao) then
            if inverso then
                return (i + j − 1) mod n
            end if
            return i
        end if
    end for
    return −1
end function
```

4.2 KMP (Knuth-Morris-Pratt)

Conhecido por ser o primeiro a resolver o problema do casamento de caracteres em tempo linear, o algoritmo KMP possui a característica de pré-processar o padrão, construindo uma "tabela de prefixos". Essa tabela armazena informações sobre os prefixos mais longos que estão contidos dentro do próprio padrão, dessa forma, a complexidade do procedimento é $O(m)$. A Figura 6 demonstra a tabela da palavra "TEXTO".

T	E	X	T	O
-1	-1	-1	0	-1

Figura 6: Demonstração do pré-processamento do algoritmo KMP

O tamanho do padrão influencia diretamente no uso de memória para armazenar a tabela, sendo um ponto de atenção para casos onde há restrição de memória e padrões extensos.

Os prefixos gerados são utilizados para otimizar a busca do padrão no texto, de forma que se uma correspondência parcial falha em uma posição específica do padrão, a tabela de prefixos é usada para determinar a próxima posição a ser comparada, evitando repetir verificações desnecessárias, conforme ilustrado na Figura 7.

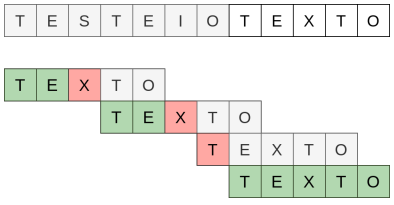

Figura 7: Demonstração do algoritmo KMP

É importante ressaltar a forma em que nenhum caractere do texto é analisado mais de uma vez, comprovando a afirmação de que o algoritmo é linear, ou seja, $O(n)$ em sua essência.

A modificação feita para possibilitar casamentos circulares é permitir que a verificação do padrão continue ao fim do texto somente se existe um casamento em formação, o que dá uma chance de completar o casamento caso este exista, assim como demonstra a Figura 8.

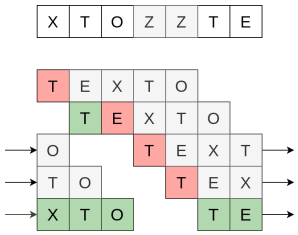

Figura 8: Demonstração do algoritmo KMP em cadeias circulares

Essa alteração adiciona mais comparações ao algoritmo, já que além de comparar com todos os caracteres do texto uma vez, é possível que seja necessário comparar mais uma vez os $m-1$ primeiros caracteres, utilizando o operador "%".

A complexidade final pode ser definida então como $O(n+m)$, não perdendo sua natureza linear. Diferentemente do algoritmo de força bruta, a linearidade da estratégia proporciona uma melhor forma de lidar com padrões e textos extensos. A seção 5 contém a bateria de testes de tempo de execução que comprovam a afirmação. O pseudocódigo representado pelo algoritmo 2 mostra como o KMP foi implementado.

Algorithm 2 KMP para casamento de padrão em texto circular

```
function KMP(padrao, texto, inverso)
    prefixos ← preprocessamento(padrao)
    j ← −1
    for i ← 0, tamanho(texto) − 1 do
        while j > −1 and padrao[j + 1] ≠ texto[i] do
            j = prefixos[j]
        end while
        if padrao[j + 1] = texto[i] then
            j = j + 1
            if i ≥ tamanho(texto) and j < m − 1 then
                while padrao[j + 1] == texto[(i + 1) mod n] do
                    i = i + 1
                    j = j + 1
                end while
            end if
        end if
        if j = tamanho(padrao) − 1 then
            if inverso then
                return i mod n
            end if
            return (i − j) mod n
        end if
    end for
    return −1
end function
```

4.3 BMH (Boyer-Moore-Horspool)

O algoritmo BMH é uma melhoria proposta por Horspool para o algoritmo BM (Boyer-Moore) original. Além de simplificar a implementação do algoritmo, sua complexidade média garante uma eficiência positiva, tornando-o uma escolha popular para encontrar casamentos de padrões.

A etapa do pré-processamento consiste na criação de uma tabela de deslocamentos referente ao alfabeto usado. Todos os elementos são inicializados com um deslocamento igual à quantidade de caracteres do padrão. Feito isso, para todos os $m - 1$ primeiros caracteres do padrão, o deslocamento de c é atualizado para o valor $m - i$.

Considerando que o tamanho do alfabeto seja constante, a complexidade dessa etapa é dada por $O(m)$, tendo em vista que todo o padrão é analisado. A Figura 9 retrata o pré-processamento realizado para o padrão "TEXTO".

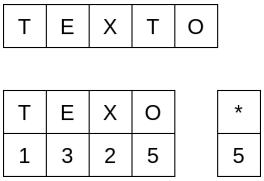

Figura 9: Demonstração do pré-processamento do algoritmo BMH

Diferente dos demais algoritmos desenvolvidos, o BMH possui a característica de iniciar a comparação partindo do fim do padrão, realizando os saltos definidos na tabela de deslocamentos toda vez que o caractere analisado não corresponde com o respectivo do texto, assim como mostra a Figura 10.

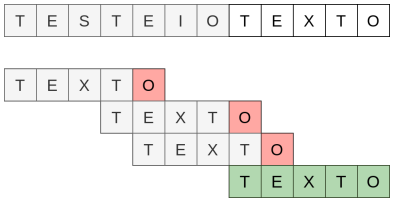

Figura 10: Demonstração do algoritmo BMH

Essa estratégia inevitavelmente possui um pior caso de execução $O(mn)$, se assemelhando ao algoritmo de força bruta. Apesar desse fato, a grande vantagem de sua implementação é o caso médio ser tido como $O(n/m)$ devido aos saltos proporcionados pela tabela evitarem uma quantidade considerável de comparações em grande parte das situações.

A adaptação desenvolvida para atribuir a capacidade de verificar padrões circulares necessita que o processo de busca permitida até $m - 1$ execuções após o final do texto para encontrar uma possível formação.

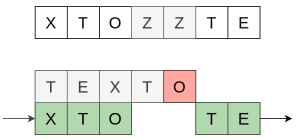

Figura 11: Demonstração do algoritmo BMH em cadeias circulares

Essa alteração adiciona no máximo $m \times n$ passos, não modificando a complexidade da estratégia que se manteve $O(mn)$. A Figura 11 demonstra um caso de casamento circular solucionado pelo Algoritmo 3.

Algorithm 3 BMH para casamento de padrão em texto circular

```
function BMH(padrao, texto, inverso)
    prefixos ← preprocessamento(padrao)
    i ← tamanho(padrao)
    tentativas ← 0
    while i ≤ tamanho(texto) or tentativas ≤ tamanho(padrao) do
        k ← i
        j ← tamanho(padrao)
        while j > 0 and k > 0 and texto[(k − 1) mod tamanho(texto)] ≠ padrao[j − 1]
do
            k = k − 1
            j = j − 1
        end while
        if j == 0 then
            if inverso then
                return i mod n
            end if
            return (i − j) mod n
        end if
        i = i + prefixos[texto[(i − 1) mod n]]
        if i > n then
            tentativas = tentativas + 1
        end if
    end while
    return −1
end function
```

4.4 Shift-And

O Shift-And é um algoritmo amplamente utilizado para a tarefa de encontrar casamento de cadeias de caracteres por sua eficiência e fácil implementação, que envolve o uso de bits e operações lógicas para obter a solução.

Partindo do princípio de pré-processar o padrão a ser procurado a fim de melhorar o desempenho, a estratégia cria máscaras de bits de tamanho m para todos os elementos do alfabeto, inicializando todos com 0 e atribuindo 1 para cada ocorrência do caractere no padrão. A Figura 12 mostra a tabela de máscaras para a palavra "TEXTO".

	T	E	X	T	O
T	1	0	0	1	0
E	0	1	0	0	0
X	0	0	1	0	0
O	0	0	0	0	1
*	0	0	0	0	0

Figura 12: Demonstração do pré-processamento do algoritmo Shift-And

O texto de entrada é então analisado caractere por caractere. Para cada um é adicionado um novo bit 1 no início da representação binária, enquanto o bit do final é removido, isso explica o nome “Shift”, que se refere à operação realizada. Em seguida, a sequência de bits é submetida a uma operação AND com a máscara do padrão. A sequência de instruções se repete até que último bit seja 1 ou o texto se encerre, assim como mostra a Figura 13.

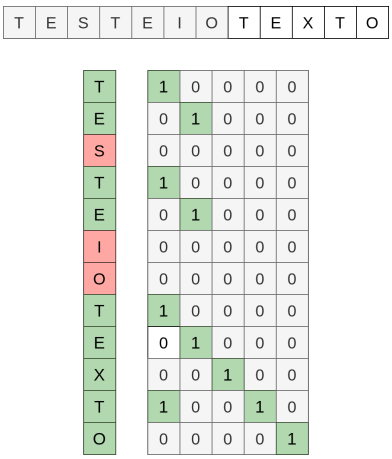

Figura 13: Demonstração do algoritmo Shift-and

Assim como o KMP, o Shift-And verifica cada caractere do texto apenas uma vez, o que define sua complexidade como $O(n)$.

Também semelhante ao KMP, sua adaptação para identificação de padrões circulares exige que sejam realizadas até $m - 1$ execuções após o término do texto, alterando a complexidade para $O(m + n)$. A Figura 14 exemplifica uma ocorrência do caso descrito.

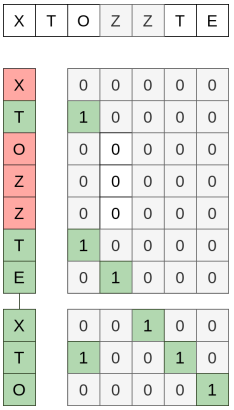

Figura 14: Demonstração do algoritmo Shift-And em cadeias circulares

Os tipos numéricos inteiros comumente utilizados na programação em C não permitem que o algoritmo seja implementado com eficiência máxima, uma vez que o limite de tamanho do padrão definido para o problema é 100, o que exige a mesma quantidade de bits de armazenamento, sendo necessário a utilização de estruturas como matrizes de dimensões adequadas ou até mesmo dividir os cálculos para que um tipo de dado menor consiga realizar as operações.

Uma forma encontrada para contornar o problema foi a utilização do **__int128**, tipo de dado específico do compilador **GCC** que possibilita a utilização de números inteiros de 128 bits, valor suficiente para garantir a fluidez da estratégia implementada no Algoritmo 4.

Algorithm 4 Shift-And para casamento de padrão em texto circular

```
function ShiftAnd(padrao, texto, inverso)
    mascaras ← preprocessamento(padrao)
    r ← 0
    tentativas ← 0
    for i ← 0, tamanho(texto) − 1 do
        repeat
            r ← ((r ≫ 1)|(1 ≪ (m − 1)))&mascaras[texto[i mod n]]
            if i + 1 ≥ tamanho(texto) then
                i = i + 1, tentativas = tentativas + 1
            end if
        until i < tamanho(texto) or tentativas ≥ m or r perder a sequência
        if (r&1) ≠ 0 then
            if i ≥ tamanho(texto) then
                i = i − 1
            end if
            if inverso then
                return i mod n
            end if
            return (i − tamanho(padrao) + 1) mod n
        end if
    end for
    return −1
end function
```

5 Testes de execução

Os testes estiveram presentes em todas as etapas do desenvolvimento do software, desde o funcionamento das funções mais elementares até o desenvolvimento de estratégias para melhorar o tempo de execução do programa, garantindo uma aplicação performática e de fácil manutenção.

Realizar apenas testes randômicos para casamento de cadeias de caracteres é uma tarefa inviável, tendo em vista que a probabilidade de um padrão se encaixar no texto é consideravelmente baixa e a verificação da veracidade dos resultados leva muito tempo e esforço.

Dessa forma, a grande maioria dos testes de funcionamento foram planejados para atingir situações específicas que fariam com que os algoritmos não encontrassem a solução correta. Essa metodologia adotada possibilitou um processo de depuração mais eficiente que garantiu a sanidade dos quatro algoritmos desenvolvidos para solucionar o problema.

5.1 Monitoramento do tempo de execução no pior caso

Foram realizados testes para avaliar de forma qualitativa os algoritmos desenvolvidos considerando sua eficiência nos tempos de execução para diferentes tamanhos de padrão e texto no pior caso de execução, quando é necessário executar o máximo de operações possíveis.

Um exemplo de pior caso da força bruta é o padrão $a^{m-1}b$ com o texto a^n. Por outro lado, o pior caso do algoritmo BMH pode ser um padrão ba^{m-1} com o texto a^n. Os algoritmos KMP e Shift-And, por serem lineares, não possuem casos especiais.

Para uniformizar os testes, foram utilizados textos 10 vezes maiores que seus respectivos padrões. O gráfico da Figura 15 traz os resultados obtidos para todas as estratégias.

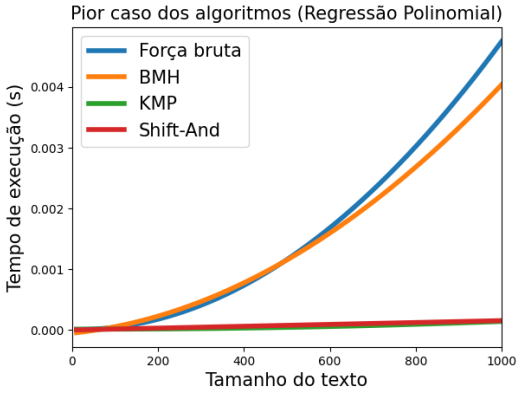

Figura 15: Gráfico do teste de execução do pior caso dos algoritmos

Tomando por base a informação de que $n = 10m$, a análise do gráfico permite constatar que os algoritmos força bruta e BMH possuem uma complexidade quadrática para solucionar o problema, se distanciando dos algoritmos KMP e Shift-And em que os tempos se mantiveram lineares e menores durante a bateria de testes.

5.2 Processamento paralelo usando Pthreads

As buscas pelos casamentos de cadeias caracteres em cada pedra do arquivo de entrada não possuem relação entre si. Portanto, visando reduzir o tempo necessário para encontrar as soluções, é possível processar todas as instruções paralelamente utilizando *Pthreads* (POSIX Threads).

Para cada pedra de entrada é criada uma *thread* que insere a solução encontrada na posição definida do vetor de resultados, evitando conflitos de acesso à memória compartilhada. O gráfico da Figura 16 mostra a diferença proporcionada pelo uso das *Pthreads* conforme o número de testes aumenta na busca utilizando o pior caso do algoritmo de força bruta como parâmetro.

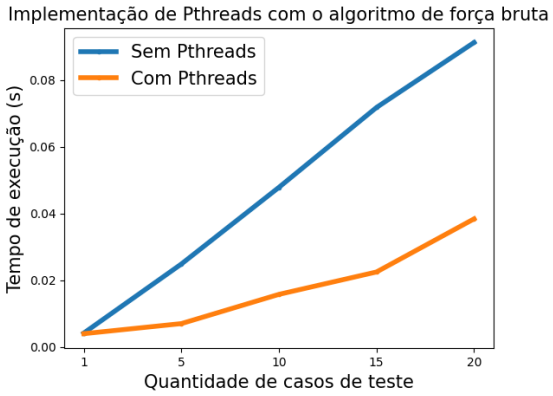

Figura 16: Gráfico do teste de eficiência da implementação de *Pthreads*

O resultado é positivo, pois o tempo necessário para concluir o processo com a implementação das *Pthreads* nunca é superior ao tempo sem ela. No entanto, é importante destacar que a criação de um número excessivo de *threads*, além da capacidade do processador, ocasiona uma execução concorrente, o que leva a uma perda considerável de desempenho. O ideal para resolver a situação é utilizar uma fila na qual uma quantidade fixa de threads é encarregada de solucionar todas as pedras sem conflitos.

6 Conclusão

Com os resultados obtidos através dos testes realizados com os algoritmos desenvolvidos é possível concluir aspectos positivos e negativos sobre cada um. Iniciando pelo método de força bruta, se trata da abordagem mais simples para o casamento de cadeia de caracteres. Embora seja de fácil implementação, possui uma complexidade de tempo $O(mn)$. Isso significa que pode ser ineficiente em casos de texto e padrão extensos.

Por outro lado, o algoritmo KMP (Knuth-Morris-Pratt) é conhecido por sua eficiência no pior caso, tendo em vista que o pré-processamento realizado o atribui a complexidade de tempo linear $O(n + m)$, dependente do tamanho do padrão devido à verificação circular. No entanto, a implementação do KMP pode ser mais complexa, além de requerer espaço adicional para armazenar a tabela de prefixos de cada caractere do padrão.

O BMH (Boyer-Moore-Horspool), assim como o KMP, também utiliza a estratégia de saltos com base em um pre-processamento do padrão, porém com uma implementação relativamente simples. No caso médio, o BMH pode ter uma complexidade de tempo próxima a $O(n/m)$, melhor que o KMP. No entanto, em situações adversas, como as testadas propositalmente na seção 5.1, pode ter a complexidade $O(mn)$, tornando-o inadequado para circunstâncias onde variações grandes de tempo são intoleráveis.

Por fim, o Shift-And, que se destaca por sua eficiência e baixo consumo de memória, utiliza operações lógicas e máscaras de bits para comparar o padrão com o texto. Sua complexidade de tempo linear $O(n+m)$, associada à simplicidade de implementação, o torna um algoritmo excelente para a resolução do problema. No entanto, deve-se ter em mente que em algumas ocasiões o uso de estruturas de dados de bits pode ser problemático e causar perda de desempenho.

Observa-se, portanto, que não há um algoritmo definitivamente melhor ou pior em todos os casos. Cada estratégia possui suas vantagens e desvantagens, e o desempenho de cada uma pode variar dependendo das características específicas do sistema e do conjunto de dados. Isso envolve avaliar o tamanho do texto e do padrão, a dificuldade de implementação, as restrições de memória e o desempenho desejado.

Em conclusão, como dito anteriormente, o processamento de cadeias de caracteres desempenha um papel fundamental em várias tarefas essenciais da tecnologia atual. O avanço nessa área possibilita aprimoramentos em diversas aplicações, impulsionando a eficiência e a precisão dos sistemas computacionais modernos.

Referências

[1] Thomas H. Cormen. *Algoritmos: Teoria e prática*. LTC, 2012.

[2] Jayme L. Szwarcfiter and Lilian Markenzon. *Estruturas de Dados e Seus Algoritmos 3ª edição*, volume 53. GEN, 2015.

[3] Nivio Ziviani. *Projeto de algoritmos: com implementações em Pascal e C*. Cengage Learning Edições Ltda., São Paulo, 3. edição revista e ampliada edition, 2011.