UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS MIRANDA CAPRIS
VÍCTOR CONSTÂNCIO DE AZEVEDO

ALGORITMOS DE BUSCA DE STRING, UMA BREVE ANÁLISE E COMPARAÇÃO

RIO DE JANEIRO
2024

LUCAS MIRANDA CAPRIS
VÍCTOR CONSTÂNCIO DE AZEVEDO

ALGORITMOS DE BUSCA DE STRING, UMA BREVE ANÁLISE E COMPARAÇÃO

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Vinícius Gusmão Pereira de Sá

RIO DE JANEIRO
2024

Capris, Lucas Miranda
C253a Algoritmos de busca de string, uma breve análise e comparação / Lucas Miranda Capris e Víctor Constâncio de Azevedo. – 2024.

59 f.

Orientador: Vinícius Gusmão Pereira de Sá.

Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação)- Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2024.

1. Busca de strings. 2. Algoritmos. 3. Complexidade. 4. Visualização. I. Azevedo, Víctor Constâncio de. II. Sá, Vinícius Gusmão Pereira de (Orient.). III. Universidade Federal do Rio de Janeiro, Instituto de Computação. IV. Título.

LUCAS MIRANDA CAPRIS
VÍCTOR CONSTÂNCIO DE AZEVEDO

ALGORITMOS DE BUSCA DE STRING, UMA BREVE ANÁLISE E COMPARAÇÃO

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 13 de Dezembro de 2024

BANCA EXAMINADORA:

gov.br Documento assinado digitalmente
VINICIUS GUSMAO PEREIRA DE SA
Data: 20/12/2024 04:28:19-0300
Verifique em https://validar.iti.gov.br

Vinícius Gusmão Pereira de Sá
D.Sc (UFRJ)

gov.br Documento assinado digitalmente
CLAUDSON FERREIRA BORNSTEIN
Data: 19/12/2024 17:22:10-0300
Verifique em https://validar.iti.gov.br

Claudson Ferreira Bornstein
PhD (UFRJ)

gov.br Documento assinado digitalmente
MARIA HELENA CAUTIERO HORTA JARDIM
Data: 20/12/2024 06:33:44-0300
Verifique em https://validar.iti.gov.br

Maria Helena Cautiero Horta Jardim
D.Sc. (UFRJ)

AGRADECIMENTOS

Agradecemos profundamente as nossas famílias, Roberto Carvalho de Azevedo e Joana d'Arc Constâncio de Azevedo, Antônio Heitor Figueira Capris, Helena Olivia Miranda Capris, e Gustavo Miranda Capris, e companheiras, Julia Deroci Lopes e Aline Atthie de Souza Pinto, pelo suporte e conselhos ao longo da nossa jornada acadêmica.

Aos nossos professores do instituto de computação por fornecerem a base necessária para que pudéssemos escrever este artigo, em especial, ao nosso orientador Vinícius Gusmão Pereira de Sá por toda a paciência e ajuda durante esta jornada.

RESUMO

Os algoritmos de busca de strings são usados diariamente em diversas ferramentas, desde o navegador web usado em quase qualquer dispositivo moderno, até os ambientes de programação usados por profissionais e estudantes da área. Temos como objetivo nesse artigo explicar e ilustrar as diferenças de funcionalidade, implementação e performance de 4 dos algoritmos mais usados da categoria: Boyer–Moore(–Horspool), Rabin–Karp, Aho–Corasick e Knuth–Morris–Pratt. Realizamos uma implementação no Google Collab como forma prática de visualização da performance. Ela nos permitiu observar os resultados que foram usados como base para determinar as vantagens e desvantagens específicas de cada algoritmo a fim de identificar os casos em que cada um é superior.

Palavras-chave: busca de strings; algoritmos; complexidade; visualização.

ABSTRACT

String search algorithms are used daily in a variety of tools, from the web browser used in almost any modern device to the programming environments used by professionals and students in the field. The aim of this article is to explain and illustrate the differences in functionality, implementation and performance of 4 of the most widely used algorithms in this category: Boyer-Moore(-Horspool), Rabin-Karp, Aho-Corasick and Knuth-Morris-Pratt. We implemented it in Google Collab as a practical way of visualizing the performance. It allowed us to observe the results which were used as a basis for determining the specific advantages and disadvantages of each algorithm in order to identify the cases in which each one is superior.

Keywords: string search; algorithms; complexity; visualization.

SUMÁRIO

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

Os algoritmos de busca de strings desempenham um papel crucial na ciência da computação e na engenharia de software, sendo amplamente utilizados para localizar padrões específicos dentro de sequências de caracteres em diversos contextos. Essas ferramentas são essenciais em tarefas que vão desde a simples busca por palavras em documentos até aplicações mais complexas, como a análise de sequências genômicas, onde a identificação precisa de padrões em cadeias de DNA é fundamental (GUSFIELD, 1997).

O problema de busca de strings, que envolve encontrar uma substring específica dentro de um texto maior, é abordado por uma variedade de algoritmos, cada um com características próprias que os tornam mais ou menos adequados para diferentes tipos de aplicações. Desde métodos mais intuitivos, como o de força bruta, até abordagens mais sofisticadas, como os algoritmos Boyer-Moore e Knuth-Morris-Pratt, cada um oferece soluções que variam em termos de eficiência e complexidade (CHARRAS; LECROQ, 2004).

Esses algoritmos desempenham um papel vital em várias aplicações modernas, sendo fundamentais em diversas áreas tecnológicas. Por exemplo, o Google utiliza técnicas avançadas de busca de strings, onde o algoritmo Boyer-Moore-Horspool é uma das bases para a otimização de pesquisas no navegador Google Chrome. A documentação do algoritmo de busca utilizado pelo navegador, disponível no repositório do Chromium, fornece uma visão detalhada sobre sua implementação e otimizações (GOOGLE, 2024).

Outro exemplo de aplicação é encontrado em sistemas de detecção de intrusões de rede (NID), que utiliza o algoritmo Aho-Corasick para a busca eficiente de múltiplos padrões simultaneamente (AHMED; ABU-RUB; KHATTAB, 2021). Este algoritmo é particularmente útil para detectar assinaturas de ataques em pacotes de rede, garantindo que todas as possíveis ocorrências de padrões maliciosos sejam identificadas de maneira rápida e eficaz.

No campo da bioinformática, o algoritmo Knuth-Morris-Pratt (KMP) tem sido amplamente utilizado em ferramentas que comparam sequências de DNA para identificar padrões genéticos específicos. No artigo de Gancheva e Stoev (GANCHEVA; STOEV, 2024), o desempenho do KMP é comparado ao método CAT proposto, demonstrando sua eficácia na busca por correspondências exatas de sequências. Este algoritmo é altamente eficiente na busca por padrões em grandes sequências genéticas, evitando a repetição desnecessária de comparações e acelerando o processo de alinhamento. A aplicação do KMP é especialmente relevante para o casamento de genes em estudos de variabilidade genética e na identificação de mutações.

Os algoritmos de busca de strings continuam a ser essenciais para o desenvolvimento

de ferramentas e sistemas em várias áreas, conforme destacado em revisões recentes que discutem as direções futuras e os desafios no uso desses algoritmos (HAKAK et al., 2019). Sua adaptabilidade e eficiência garantem seu papel contínuo em aplicações tecnológicas modernas.

1.2 MOTIVAÇÃO

Com o crescente volume de dados disponível na internet e em sistemas corporativos, a eficiência na execução de tarefas computacionais simples, como a busca de strings, tornou-se uma prioridade para empresas que buscam otimizar a experiência do usuário. A busca por maior eficiência não se limita ao setor empresarial, mas também se estende ao uso pessoal, onde usuários demandam respostas rápidas e precisas ao realizarem pesquisas em grandes volumes de dados.

A proposta deste trabalho é realizar um estudo detalhado da performance de alguns dos algoritmos de busca de strings mais populares, analisando suas vantagens e desvantagens em diferentes cenários. Espera-se que os resultados obtidos possam servir de guia para a escolha do algoritmo mais adequado a cada necessidade, seja ela corporativa ou pessoal.

É importante ressaltar que cada algoritmo possui benefícios e malefícios distintos, de maneira que, não há um que se sobressaia em todos os casos. Devido a isso é de grande utilidade entender os diferentes casos em que cada um se destaca, a fim de evitar que novos testes sejam realizados a cada nova aplicação que o usuário deseje realizar.

1.3 OBJETIVO

O objetivo deste trabalho é analisar em profundidade quatro dos principais algoritmos de busca de strings: Boyer–Moore–Horspool, Aho-Corasick, Rabin-Karp e Knuth-Morris-Pratt (CHARRAS; LECROQ, 2004). Consideramos também um algoritmo ingênuo, que realiza uma implementação direta comparando o padrão com o texto checando letra por letra e avançando apenas um espaço em caso de falha em algum ponto, com a finalidade de comparar um algoritmo mais simples com os mais robustos propostos por estudiosos do tema.

É importante ressaltar que o algoritmo de Rabin-Karp tem seu desempenho fortemente atrelado à função de dispersão (*hash function*) utilizada. Devido a isso, realizamos duas implementações do mesmo utilizando funções de hashing diferentes para estudar melhor o impacto causado pela função escolhida.

A análise será estruturada em três critérios principais: funcionalidade, implementação e performance, com o objetivo de oferecer uma visão comparativa que destaque os pontos fortes e fracos de cada algoritmo em diferentes contextos.

Funcionalidade Esta categoria refere-se à forma como cada algoritmo opera do ponto de vista teórico, incluindo a maneira como estruturam suas etapas para alcançar o objetivo de busca de strings.

Implementação Este critério avalia as dificuldades associadas à implementação de cada algoritmo, considerando a facilidade de inserção e modificação do código. A legibilidade e manutenção do código são aspectos críticos no desenvolvimento moderno de software e, portanto, também serão levados em consideração.

Performance A análise de performance focará na eficiência dos algoritmos em termos de tempo de execução e uso de memória. A intenção não é apenas identificar o algoritmo mais rápido, mas entender as compensações entre tempo e espaço, e como essas variáveis influenciam na escolha do algoritmo para diferentes aplicações.

Além da análise teórica e prática, desenvolvemos um código interativo com o intuito de facilitar a compreensão dos algoritmos. Esta plataforma permitirá que os usuários executem e comparem os algoritmos em tempo real, testando diferentes parâmetros e visualizando os resultados de forma dinâmica.

2 ALGORITMOS DE BUSCA DE STRINGS

Neste capítulo, realizaremos uma análise detalhada de quatro dos algoritmos de busca de strings que serão abordados: Boyer-Moore-Horspool, Rabin-Karp, Knuth-Morris-Pratt e Aho-Corasick (HORSPOOL, 1980)(AHO; CORASICK, 1975)(KARP; RABIN, 1987) (KNUTH; MORRIS; PRATT, 1977). O objetivo é fornecer uma compreensão abrangente de cada um desses algoritmos, abordando seu contexto histórico, fundamentos teóricos e detalhes de implementação. Essa abordagem nos permitirá estabelecer uma base sólida para as comparações subsequentes entre os algoritmos, destacando suas principais características e suas aplicabilidades em diferentes cenários.

2.1 VARIANTES DE BUSCA DE STRINGS

É relevante ressaltar que existem diferentes tipos de busca de string. Esses tipos variam dependendo da informação que se deseja obter, podendo ser divididas em busca de padrão simples, busca de padrão múltiplo, busca online e busca aproximada.

Na busca simples, desejamos encontrar apenas um padrão dentro de um texto, ou seja, buscamos apenas uma palavra ou uma frase. Os algoritmos de Boyer-Moore-Horspool e Rabin-Karp são comumente utilizados nesse tipo de busca.

Já na busca por padrões múltiplos, procuramos vários padrões ao mesmo tempo em um texto. Um algoritmo bem conhecido por realizar essa busca de forma eficiente é o de Aho-Corasick.

Na busca online, o texto é processado de forma dinâmica à medida que os dados vão sendo recebidos, ou seja, o texto não é previamente conhecido e o algoritmo deve responder imediatamente ao recebimento de novos dados, como ocorre com alguns sistemas em tempo real. Existe uma variação de algoritmo do Knuth-Morris-Pratt chamada de Morris-Pratt que é utilizada para esse tipo de busca.

Por fim, na busca aproximada, a busca encontra pedaços de textos que são similares mas não exatamente iguais ao padrão desejado, levando em consideração possíveis erros de digitação que podem ocorrer.

Neste artigo iremos ter como foco a busca por padrão simples, dado que é um tipo de busca que é comum a todos os algoritmos que serão abordados mais a fundo. Como o algoritmo de Aho-Corasick é utilizado principalmente para a busca por padrões múltiplos também iremos considerá-la em nossos testes.

2.2.1 Boyer–Moore–Horspool

Contexto histórico: O algoritmo Boyer-Moore foi desenvolvido por Robert S. Boyer e J. Strother Moore em 1977. É amplamente reconhecido por ter sido um dos primeiros algoritmos de busca de strings a utilizar heurísticas para otimizar o processo de busca. Sua inovação principal foi o uso de informações do padrão (substring que está querendo encontrar no texto) para reduzir o número de comparações necessárias, o que representou um avanço significativo em relação aos algoritmos anteriores (BOYER; MOORE, 1977).

Em 1980, Nigel Horspool propôs uma variação simplificada deste algoritmo, conhecida como Boyer-Moore-Horspool. A versão de Horspool manteve a essência da heurística do caractere ruim, mas eliminou a complexidade da heurística do bom sufixo, resultando em uma implementação mais simples e ainda bem eficiente para muitos casos práticos (HORSPOOL, 1980).

Pouco tempo depois, foi publicado um artigo que sugeria uma melhoria na implementação do algoritmo de Booyer-Moore-Horspool, gerando um aumento de velocidade de até 25% (RAITA, 1992). Com o passar do tempo, novas propostas foram feitas de melhoria ao algoritmo, usando, por exemplo, o uso de paralelismo computacional para conseguir um aumento significativo de desempenho na execução do algoritmo (JEONG et al., 2014).

Até hoje o algoritmo de Booyer-Moore-Horspool ainda é amplamente utilizado. Um dos exemplos desse uso é na linguagem de programação Java através do método "String.indexOf" que utiliza Booyer-Moore-Horspool para encontrar a primeira ocorrência de uma substring em uma string.

Base teórica: O algoritmo se baseia na heurística do caractere ruim, que funciona da seguinte forma:

O algoritmo utiliza a informação do padrão que está sendo procurado para otimizar seu funcionamento. Ao encontrar um caractere diferente do padrão, o algoritmo buscará a última ocorrência desse caractere dentro do padrão para determinar o tamanho do salto que poderá realizar. Caso não encontre esse caractere dentro do padrão, ele saltará até todo o padrão ultrapassar o caracter.

Essa heurística permite que o algoritmo faça menos comparações do que os métodos tradicionais de busca, especialmente em padrões grandes e textos longos (HORSPOOL, 1980).

Implementação: A implementação do Boyer-Moore envolve dois passos principais:

1. Pré-processamento: Construção das tabelas de heurística que armazenam a última ocorrência de cada caractere e as informações sobre o padrão.
2. Busca: Utiliza as tabelas de heurística para saltar sobre o texto e encontrar o padrão com eficiência.

A complexidade do algoritmo é $\mathcal{O}(n * m)$ no pior caso e $\mathcal{O}(n/m)$ no melhor caso, onde n é o comprimento do texto e m é o comprimento do padrão (HORSPOOL, 1980).

Pseudocódigo:

Algorithm 1 Algoritmo Boyer-Moore-Horspool

Data: Padrão $P[0 \dots m-1]$, Texto $T[0 \dots n-1]$
Result: Índices de ocorrência de P em T
tabelaDeslocamento $\leftarrow$ construirTabelaDeslocamento(P, m)
  $i \leftarrow 0$ `// Índice para` T
while $i \leq n - m$ **do**
  $j \leftarrow m - 1$ `// Índice para` P
  while $j \geq 0$ ***e*** $P[j] = T[i + j]$ **do**
    $j \leftarrow j - 1$
  end
  if $j < 0$ **then**
    imprimir "Padrão encontrado na posição "i
    $i \leftarrow i +$ tabelaDeslocamento$[T[i + m]]$
  end
  else
    $i \leftarrow i +$ tabelaDeslocamento$[T[i + j]]$
  end
end
retornar

2.2.2 Rabin-Karp

Contexto histórico: O algoritmo Rabin-Karp foi desenvolvido por Michael O. Rabin e Richard M. Karp em 1987. Foi um dos primeiros a usar técnicas de hashing para buscar padrões, e sua abordagem inovadora permitiu a busca de múltiplos padrões simultaneamente (KARP; RABIN, 1987).

Para ser mais específico, Rabin-Karp utiliza string hashing que é uma técnica usada para converter uma string em um valor numérico fixo, chamado hash, que representa a string de maneira compacta. Esse valor é gerado por uma função de hash, que mapeia a string original para um número.

O ganho de desempenho de Rabin-Karp evolui paralelamente à descoberta e uso de novas funções de hashing eficientes. Atualmente, o uso de multihashing (uso de mais de uma função hash) e funções polinomiais com bases e módulos bem específicos são

usados para reduzir significativamente a ocorrência de colisões, sequências diferentes que geram o mesmo valor de hashing, gerando um ganho considerável de eficiência.

Também vale ressaltar que Rabin-Karp se destaca pelo baixo uso de memória. Muitos algoritmos de busca de string utilizam tabelas ou árvores para referência como dados pré-processados, entretanto, Rabin-Karp precisa apenas definir a função de dispersão que vai utilizar e o valor inicial da substring que se deseja encontrar.

Base teórica: Primeiro, definimos a função de dispersão que será utilizada. Depois, é calculado o valor de hash do padrão que desejamos encontrar. Após isso, calculamos o valor de hash da substring dentro do texto e comparamos com o valor de hash do padrão. Se os valores hash coincidirem, uma comparação adicional é feita para verificar a correspondência, uma vez que substrings diferentes podem gerar o mesmo valor hash, que é comumente chamado de colisão.

É importante citar que devemos fazer uso de *rolling hash*, que é uma técnica que pode ser utilizada em diversos hashes, para podermos ganhar eficiência considerável no algoritmo. Trata-se de uma técnica de hashing que permite calcular o hash de todas as substrings de tamanho fixo de uma string de forma eficiente. Em vez de recalcular o hash do zero para cada substring, ele atualiza o valor do hash da substring anterior, tornando o processo muito mais rápido. Por exemplo, se temos uma substring A e queremos calcular o hash de uma nova substring B a partir de A, basta subtrair do hash de A o valor do caractere que foi removido e somar o valor do novo caractere inserido. No contexto do Rabin-Karp, isso significa remover o valor do caractere mais à esquerda da substring anterior e adicionar o valor do caractere que entra na nova substring.

Implementação: O Rabin-Karp é implementado em duas fases principais:

1. Pré-processamento: Calcula o valor hash do padrão e inicializa o valor hash para as substrings do texto.
2. Busca: Desliza a janela sobre o texto, atualiza o hash da substring atual e compara com o hash do padrão.

Sua complexidade no melhor caso é $\mathcal{O}(m + n)$ para a busca de múltiplos padrões, embora a eficiência possa variar bastante dependendo da função de hash utilizada. Sua complexidade no pior caso é $\mathcal{O}(m * n)$ (KARP; RABIN, 1987).

Pseudocódigo:

Algorithm 2 Algoritmo Rabin-Karp

Data: Padrão $P[0 \dots m-1]$, Texto $T[0 \dots n-1]$, Base b, Módulo q
Result: Índices de ocorrência de P em T
$h_P \leftarrow \text{calcularHash}(P, b, q)$
$h_T \leftarrow \text{calcularHash}(T[0 \dots m-1], b, q)$

for $i \leftarrow 0$ **to** $n - m$ **do**
  if $h_P = h_T$ **then**
    if $T[i \dots i+m-1] = P[0 \dots m-1]$ **then**
      imprimir "Padrão encontrado na posição" i
    end
  end
  if $i < n - m$ **then**
    $h_T \leftarrow \text{rollingHash}(T[i+1 \dots i+m], h_T, T[i], T[i+m], b, q)$
  end
end
retornar

2.2.3 Knuth-Morris-Pratt (KMP)

Contexto histórico: O algoritmo Knuth-Morris-Pratt foi desenvolvido por Donald E. Knuth, Vaughan R. Pratt e James H. Morris em 1977. Ele é conhecido por introduzir a técnica de pré-processamento do padrão para evitar comparações desnecessárias, representando uma importante inovação na busca de strings (KNUTH; MORRIS; PRATT, 1977).

Algumas melhorias foram sugeridas para o KMP ao longo do tempo. Colussi propôs alguns passos extras no algoritmo para garantir um ganho de eficiência (COLUSSI, 1991). Em 2010, Xian-Feng propôs um novo algoritmo utilizando o KMP em conjunto com o algoritmo de Boyer-Moore (XIAN-FENG; YU-BAO; LU, 2010).

Base teórica: O KMP utiliza uma tabela de falhas para otimizar a busca, onde é realizado um pré-processamento do padrão para construir uma tabela que indica as posições de falhas, permitindo ao algoritmo pular comparações já realizadas. Esse pré-processamento é feito analisando os prefixos e sufixos do padrão, comparando esses grupos e gerando uma tabela com os valores dos índices de partes repetidas, o que corresponde as posições de falhas. Essa técnica reduz a necessidade de retroceder no texto, tornando a busca mais eficiente (KNUTH; MORRIS; PRATT, 1977).

Implementação: A implementação do KMP é dividida em:

1. Pré-processamento: Geração da tabela de falhas baseada no padrão.
2. Busca: Utiliza a tabela de falhas para evitar comparações redundantes enquanto percorre o texto.

A complexidade é $\mathcal{O}(n+m)$, onde n é o comprimento do texto e m é o comprimento do padrão, no pior caso, o que o torna eficiente em termos de tempo. O algoritmo precisa de apenas $\mathcal{O}(m)$ locais de memória interna se o texto for lido de um arquivo externo, e apenas $\mathcal{O}(\log m)$ unidades de tempo transcorrem entre entradas consecutivas de um único caractere. Importante ressaltar que todas as constantes de proporcionalidade implícitas nessas fórmulas "O" são independentes do tamanho do alfabeto. (KNUTH; MORRIS; PRATT, 1977).

Pseudocódigo:

Algorithm 3 Algoritmo KMP

Data: Padrão $P[0 \ldots m-1]$, Texto $T[0 \ldots n-1]$
Result: Índices de ocorrência de P em T

```
lps ← calcularLPSArray(P, m)
    i ← 0 // Índice para T
j ← 0 // Índice para P
while i < n do
    if P[j] = T[i] then
        i ← i + 1
        j ← j + 1
    end
    if j = m then
        imprimir "Padrão encontrado na posição 'i − j'."
        j ← lps[j − 1]
    end
    else if i < n e P[j] ≠ T[i] then
        if j ≠ 0 then
            j ← lps[j − 1]
        end
        else
            i ← i + 1
        end
    end
end
retornar
```

2.2.4 Aho-Corasick

Contexto histórico: O algoritmo Aho-Corasick foi desenvolvido por Alfred V. Aho e Margaret J. Corasick em 1975. É conhecido por sua capacidade de buscar múltiplos padrões simultaneamente, utilizando uma estrutura de árvore de sufixos para realizar a busca de maneira eficiente (AHO; CORASICK, 1975). Como o Aho-Corasick utiliza árvores, as melhorias que elas recebem também podem ser aplicadas ao algoritmo, aumentando sua eficiência ao longo do tempo. O uso de paralelismo também vem se mostrando muito eficaz para ele, aumentando significativamente seu desem-

penho (ARUDCHUTHA; NISHANTHY; RAGEL, 2013)(THAMBAWITA; RAGEL; ELKADUWE, 2016).

Base teórica: O Aho-Corasick constrói uma máquina de estado finito usando uma árvore como estrutura de dados para armazenar os sufixos em cada ramo, assim como os estados em cada nó. As arestas são as transições entre os estados da máquina, que são geradas a partir dos caracteres dos textos e dos padrões desejados. Isso permite que o algoritmo encontre todas as ocorrências de vários padrões em um único percurso do texto (AHO; CORASICK, 1975).

Implementação: O Aho-Corasick é implementado em duas etapas:

1. Construção da Árvore de Sufixos: Criação da estrutura de dados baseada nos padrões a serem encontrados.
2. Busca: Percorre o texto uma vez, utilizando a máquina de estados para identificar as ocorrências dos padrões.

A complexidade é $\mathcal{O}(n + m + z)$, onde n é o comprimento do texto, m é o total de caracteres dos padrões, e z é o número de ocorrências encontradas, no pior caso, tornando-o muito eficiente na busca de múltiplos padrões (AHO; CORASICK, 1975).

Pseudocódigo:

Algorithm 4 Algoritmo Aho-Corasick

Data: Conjunto de padrões $\{P_1, P_2, \ldots, P_k\}$, Texto $T[0 \ldots n-1]$
Result: Todos os padrões encontrados no texto T

```
trie ← construirTrie({P_1, P_2, ..., P_k})
    falha ← construirLinksFalha(trie)
    saída ← construirLinksSaída(trie)
    estado ← 0 // Estado inicial na Trie
for i ← 0 to n − 1 do
    while estado ≠ 0 e próximoEstado(estado, T[i]) = nulo do
        estado ← falha[estado]
    end
    estado ← próximoEstado(estado, T[i])
    if estado = nulo then
        estado ← 0
    end
    if saída[estado] ≠ ∅ then
        foreach padrão ∈ saída[estado] do
            imprimir "Padrão encontrado na posição 'i − len(padrão) + 1'."
        end
    end
end
retornar
```

3 CONTRIBUIÇÃO E RESULTADOS ESPERADOS

Neste capítulo, iremos apresentar o código desenvolvido para realizar nossos testes, debatendo as escolhas de implementação, sua estrutura e suas funcionalidades. Também iremos comentar algumas aplicações práticas que podem ser realizadas com seu uso.

3.1 ORGANIZAÇÃO DO CÓDIGO

3.1.1 Estrutura Geral do Código

No código realizado, dividimos nossa estrutura em três partes principais determinadas pelas células do colab. Nas primeiras células, temos a implementação dos diferentes algoritmos utilizados. Na célula seguinte, temos o código que gera o espaço onde podemos inserir o texto em que faremos a busca e o padrão que desejamos encontrar. Este pedaço do código também é responsável por mostrar as medidas de desempenho e informar os locais em que o texto corresponde ao padrão. Na célula final, temos o código que gera os gráficos baseados no desempenho que os algoritmos apresentaram para podermos analisar melhor os resultados obtidos.

Abaixo, podemos ver os códigos utilizados na íntegra e a interface gerada:

- Código dos Algoritmos
  - Algoritmo ingênuo

```
def naive_alg(pattern, text):
    m = len(pattern)
    n = len(text)
    positions = []

    for i in range(n - m + 1):
        match = True
        for j in range(m):
            if text[i + j] != pattern[j]:
                match = False
                break
        if match:
            positions.append(i)

    return positions
```

  - Algoritmo de Rabin-Karp Básico

```
def rabin_karp_basic(pattern, text, q=101):
    d = 256
    m = len(pattern)
    n = len(text)
```

```
    p = 0
    t = 0
    h = 1
    positions = []

    for i in range(m - 1):
        h = (h * d) % q

    for i in range(m):
        p = (d * p + ord(pattern[i])) % q
        t = (d * t + ord(text[i])) % q

    for i in range(n - m + 1):
        if p == t:
            if text[i:i + m] == pattern:
                positions.append(i)

        if i < n - m:
            t = (d * (t - ord(text[i]) * h) + ord(text[i + m])) % q

    return positions
```

– Algoritmo de Rabin-Karp Otimizado

```
def rabin_karp_opt(pattern, text):
    m = len(pattern)
    n = len(text)
    p = 31
    m_prime = 10**9 + 9
    positions = []

    h_pattern = 0
    current_hash = 0
    for i in range(m):
        h_pattern = (h_pattern * p + ord(pattern[i])) % m_prime
        current_hash = (current_hash * p + ord(text[i])) % m_prime

    p_m = pow(p, m - 1, m_prime)

    if current_hash == h_pattern and text[:m] == pattern:
        positions.append(0)

    for i in range(1, n - m + 1):
        current_hash = (current_hash - ord(text[i - 1]) * p_m) % m_prime
        current_hash = (current_hash * p + ord(text[i + m - 1])) % m_prime

        if current_hash == h_pattern and text[i:i + m] == pattern:
            positions.append(i)

    return positions
```

– Algotimo de Boyer-Moore-Horspool

```python
def boyer_moore_horspool(pattern, text):
    m = len(pattern)
    n = len(text)
    skip = {char: m - i - 1 for i, char in enumerate(pattern[:-1])}
    positions = []
    k = m - 1

    while k < n:
        j = m - 1
        i = k

        while j >= 0 and text[i] == pattern[j]:
            j -= 1
            i -= 1

        if j == -1:
            positions.append(i + 1)

        k += skip.get(text[k], m)

    return positions
```

– Algoritmo de Aho-Corasick

```python
class AhoCorasick:
    def __init__(self, keywords):
        self.transitions = {}
        self.failures = {}
        self.outputs = {}
        self.build_automaton(keywords)

    def build_automaton(self, keywords):
        new_state = 0

        for keyword in keywords:
            state = 0
            for char in keyword:
                if state not in self.transitions:
                    self.transitions[state] = {}
                if char not in self.transitions[state]:
                    new_state += 1
                    self.transitions[state][char] = new_state
                state = self.transitions[state][char]
            if state not in self.outputs:
                self.outputs[state] = []
            self.outputs[state].append(keyword)

        queue = []
        for char in self.transitions[0]:
            state = self.transitions[0][char]
            self.failures[state] = 0
            queue.append(state)

        while queue:
            state = queue.pop(0)

            for char, next_state in self.transitions.get(state, {}).items():
```

```
                queue.append(next_state)
                fail_state = self.failures[state]

                while fail_state is not None and char not in
                ↪  self.transitions.get(fail_state, {}):
                    fail_state = self.failures.get(fail_state)

                if fail_state is not None and char in self.transitions[fail_state]:
                    self.failures[next_state] = self.transitions[fail_state][char]
                else:
                    self.failures[next_state] = 0

                if next_state not in self.outputs:
                    self.outputs[next_state] = []
                self.outputs[next_state].extend(self.outputs.get(self.failures[next_state],
                ↪  []))

    def search(self, text):
        state = 0
        results = []

        for i in range(len(text)):
            while state is not None and text[i] not in self.transitions.get(state, {}):
                state = self.failures.get(state)

            if state is None:
                state = 0
                continue

            state = self.transitions[state].get(text[i], 0)

            for pattern in self.outputs.get(state, []):
                results.append((i - len(pattern) + 1, pattern))

        return results
```

– Algoritmo de Knuth-Morris-Pratt

```
def knuth_morris_pratt(pattern, text):
    def compute_lps_array(pattern):
        length = 0
        lps = [0] * len(pattern)
        i = 1

        while i < len(pattern):
            if pattern[i] == pattern[length]:
                length += 1
                lps[i] = length
                i += 1
            else:
                if length != 0:
                    length = lps[length - 1]
                else:
                    lps[i] = 0
                    i += 1
        return lps

```

```
    m = len(pattern)
    n = len(text)
    lps = compute_lps_array(pattern)
    positions = []
    i = 0
    j = 0

    while i < n:
        if pattern[j] == text[i]:
            i += 1
            j += 1

        if j == m:
            positions.append(i - j)
            j = lps[j - 1]
        elif i < n and pattern[j] != text[i]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1

    return positions
```

- Código da interface com testes de desempenho

```
import ipywidgets as widgets
from IPython.display import display, clear_output
import time

# Área de texto para exibir o texto carregado ou permitir inserção manual
text_input = widgets.Textarea(
    placeholder="O texto carregado será exibido aqui ou você pode inserir manualmente...",
    layout=widgets.Layout(width="500px", height="150px")
)

# Botão para carregar o arquivo .txt em memória
upload_widget = widgets.FileUpload(
    description="Carregar Arquivo .txt",
    accept=".txt",
    multiple=False,
    layout=widgets.Layout(width="248px")
)

# Botão para exibir o texto na caixa de texto
exibir_texto_btn = widgets.Button(
    description="Exibir Texto em Caixa",
    layout=widgets.Layout(width="248px")
)

# Feedback para o carregamento do arquivo
feedback_label = widgets.Label(value="")

# Área de texto para inserir o padrão com suporte a quebra de linha
pattern_input = widgets.Textarea(
    placeholder="Digite o(s) padrão(ões) para busca. Use ';;' para separar múltiplos padrões.",
    layout=widgets.Layout(width="500px", height="100px")
)
```

```
33
34  # Botão para iniciar a busca
35  botao = widgets.Button(description="Testar", layout=widgets.Layout(width="500px"))
36
37  # Área de saída para exibir os resultados
38  output_area = widgets.Output()
39
40  # Variável global para armazenar o texto carregado
41  texto_em_memoria = None
42
43  # Variáveis globais para armazenar os tempos médios dos algoritmos
44  ingenuo_times = []
45  rabin_karp_basic_times = []
46  rabin_karp_opt_times = []
47  boyer_moore_horspool_times = []
48  aho_corasick_times = []
49  knuth_morris_pratt_times = []
50
51  # Função para carregar o conteúdo do arquivo em memória
52  def carregar_arquivo(change):
53      global texto_em_memoria
54      feedback_label.value = "Carregando..."
55      if upload_widget.value:
56          upload = next(iter(upload_widget.value.values()))
57          texto_em_memoria = upload['content'].decode('utf-8')
58          text_input.value = texto_em_memoria  # Atualiza a área de texto com o conteúdo do arquivo
59          feedback_label.value = "Arquivo carregado em memória com sucesso!"
60
61  # Observador que carrega o arquivo ao ser selecionado
62  upload_widget.observe(carregar_arquivo, names='value')
63
64  # Função para exibir o texto carregado em memória na caixa de texto
65  def exibir_texto_em_caixa(btn):
66      global texto_em_memoria
67      if texto_em_memoria is not None:
68          text_input.value = texto_em_memoria
69          feedback_label.value = "Texto exibido na caixa com sucesso!"
70      else:
71          text_input.value = "Nenhum texto foi carregado em memória."
72          feedback_label.value = "Nenhum texto para exibir."
73
74  exibir_texto_btn.on_click(exibir_texto_em_caixa)
75
76  # Função para medir o tempo médio de execução e retornar o resultado do algoritmo
77  def medir_tempo_execucao(func, *args):
78      execution_times = []
79      num_executions = 10
80      result = None
81
82      for _ in range(num_executions):
83          start_time = time.time()
84          result = func(*args)
85          end_time = time.time()
86          execution_times.append(end_time - start_time)
87
88      return result, sum(execution_times) / num_executions
89
90  # Função principal para executar todos os algoritmos e armazenar os tempos médios
```

```
91  def ao_clicar(botao):
92      global texto_em_memoria, ingenuo_times, rabin_karp_basic_times, rabin_karp_opt_times
93      global boyer_moore_horspool_times, aho_corasick_times, knuth_morris_pratt_times
94
95      # Limpa as listas para evitar dados duplicados
96      ingenuo_times.clear()
97      rabin_karp_basic_times.clear()
98      rabin_karp_opt_times.clear()
99      boyer_moore_horspool_times.clear()
100     aho_corasick_times.clear()
101     knuth_morris_pratt_times.clear()
102
103     # Substitui todos os tipos de quebra de linha por um espaço no padrão e no texto para garantir
        ↪ a leitura de múltiplas linhas em uma única busca
104     raw_pattern = pattern_input.value.replace('\r\n', ' ').replace('\r', ' ').replace('\n', '
        ↪ ').strip()
105     text = texto_em_memoria or text_input.value.strip()  # Usa o texto carregado ou o texto
        ↪ digitado pelo usuário
106
107     # Garantia de uso correto
108     with output_area:
109         clear_output()
110         if not raw_pattern:
111             print("Por favor, insira pelo menos um padrão para busca.")
112             return
113
114         if not text:
115             print("Por favor, insira ou carregue um texto.")
116             return
117
118         # Detecta se há múltiplos padrões usando o delimitador ";;"
119         if ';;' in raw_pattern:
120             patterns = [p.strip() for p in raw_pattern.split(";;")]
121             print("\nExecutando busca com múltiplos padrões...\n")
122         else:
123             patterns = [raw_pattern]
124             print("\nExecutando busca...\n")
125
126         # Algoritmo Ingênuo
127         print("Resultados para Algoritmo Ingênuo:")
128         for pattern in patterns:
129             result_ingenuo, tempo_ingenuo = medir_tempo_execucao(naive_alg, pattern, text)
130             ingenuo_times.append(tempo_ingenuo)
131             print(f"Padrão '{pattern}': Encontrado nas posições {result_ingenuo}.")
132             print(f"Tempo médio: {tempo_ingenuo:.6f} segundos\n")
133
134         # Rabin-Karp Básico
135         print("\nResultados para Rabin-Karp (Básico):")
136         for pattern in patterns:
137             result_rabin_karp_basic, tempo_rabin_karp_basic =
                ↪ medir_tempo_execucao(rabin_karp_basic, pattern, text)
138             rabin_karp_basic_times.append(tempo_rabin_karp_basic)
139             print(f"Padrão '{pattern}': Encontrado nas posições {result_rabin_karp_basic}.")
140             print(f"Tempo médio: {tempo_rabin_karp_basic:.6f} segundos\n")
141
142         # Rabin-Karp Otimizado
143         print("\nResultados para Rabin-Karp (Otimizado):")
144         for pattern in patterns:
```

```
            result_rabin_karp_opt, tempo_rabin_karp_opt = medir_tempo_execucao(rabin_karp_opt,
            ↪  pattern, text)
            rabin_karp_opt_times.append(tempo_rabin_karp_opt)
            print(f"Padrão '{pattern}': Encontrado nas posições {result_rabin_karp_opt}.")
            print(f"Tempo médio: {tempo_rabin_karp_opt:.6f} segundos\n")

        # Boyer-Moore-Horspool
        print("\nResultados para Boyer-Moore-Horspool:")
        for pattern in patterns:
            result_bmh, tempo_bmh = medir_tempo_execucao(boyer_moore_horspool, pattern, text)
            boyer_moore_horspool_times.append(tempo_bmh)
            print(f"Padrão '{pattern}': Encontrado nas posições {result_bmh}.")
            print(f"Tempo médio: {tempo_bmh:.6f} segundos\n")

        # Aho-Corasick
        print("\nResultados para Aho-Corasick:")
        automaton = AhoCorasick(patterns)
        result_aho_corasick, tempo_aho_corasick = medir_tempo_execucao(automaton.search, text)
        aho_corasick_times.append(tempo_aho_corasick)
        print(f"Padrões encontrados: {result_aho_corasick}.")
        print(f"Tempo médio: {tempo_aho_corasick:.6f} segundos\n")

        # Knuth-Morris-Pratt (KMP)
        print("\nResultados para Knuth-Morris-Pratt (KMP):")
        for pattern in patterns:
            result_kmp, tempo_kmp = medir_tempo_execucao(knuth_morris_pratt, pattern, text)
            knuth_morris_pratt_times.append(tempo_kmp)
            print(f"Padrão '{pattern}': Encontrado nas posições {result_kmp}.")
            print(f"Tempo médio: {tempo_kmp:.6f} segundos\n")

# Associando a função ao botão
botao.on_click(ao_clicar)

# Organizando e exibindo os widgets
botoes_layout = widgets.HBox([upload_widget, exibir_texto_btn])
layout = widgets.VBox([
    text_input,
    pattern_input,
    botoes_layout,
    botao,
    feedback_label,
    output_area
])

display(layout)
```

- Interface

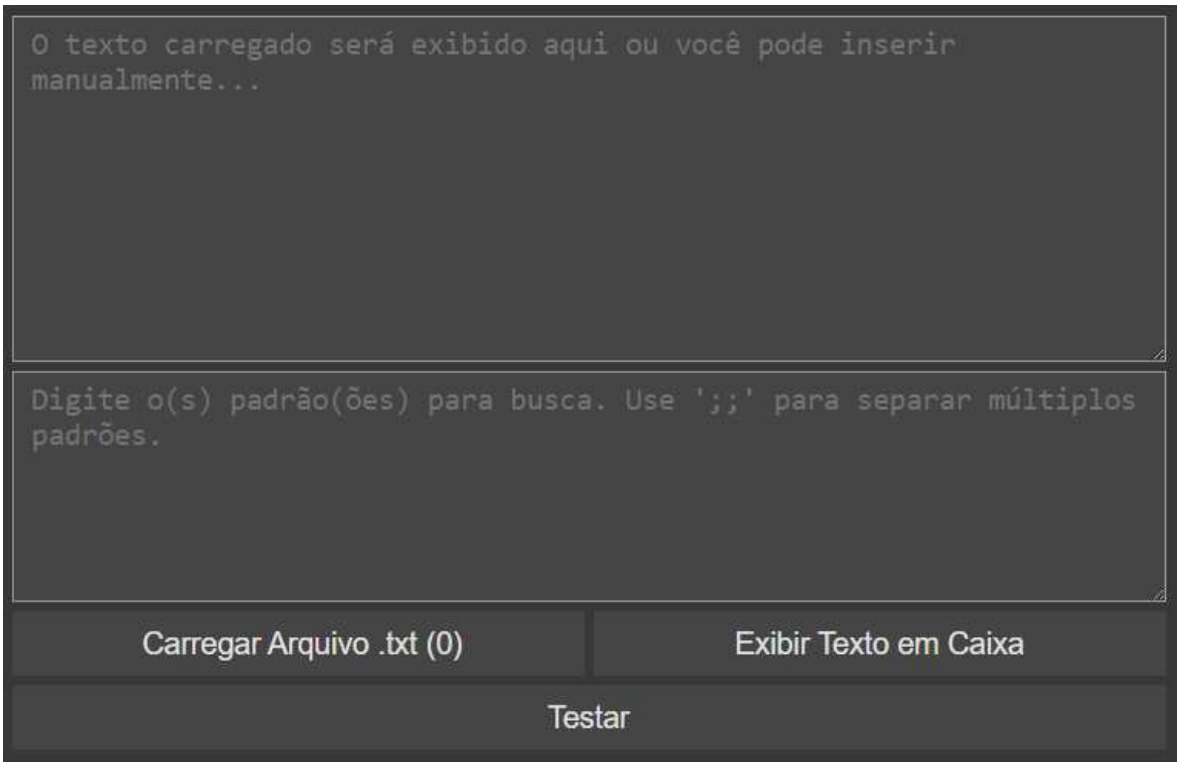


Figura 1 – Interface

- Código para geração de gráficos dinâmicos

```
import matplotlib.pyplot as plt
import numpy as np

# Dados de exemplo para cada algoritmo
algoritmos = ["Ingênuo", "Rabin-Karp (Básico)", "Rabin-Karp (Otimizado)", "Boyer-Moore-Horspool", "Aho-Corasick", "Knuth-Morris-Pratt (KMP)"]
dados = [ingenuo_times, rabin_karp_basic_times, rabin_karp_opt_times, boyer_moore_horspool_times, aho_corasick_times, knuth_morris_pratt_times]

# Verifica a quantidade de entradas para determinar a visualização adequada
num_testes = len(dados[0])  # Considera que todas as listas têm o mesmo comprimento

# Gráfico 1: Gráfico de barras para média de tempos
if all(len(dado) == 1 for dado in dados):
    medias = [np.mean(dado) for dado in dados]
    plt.figure(figsize=(12, 6))
    plt.bar(algoritmos, medias, color='lightblue')
    plt.xlabel("Algoritmos")
    plt.ylabel("Tempo Médio (s)")
    plt.title("Tempo Médio por Algoritmo")
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()

# Gráfico 2: Gráfico de linhas para variação de desempenho por teste (se múltiplos dados estiverem presentes)
if num_testes > 1:
    plt.figure(figsize=(12, 8))
```

```
    for i, dado in enumerate(dados):
        plt.plot(range(len(dado)), dado, marker='o', label=algoritmos[i])
    plt.xlabel("Número de Testes")
    plt.ylabel("Tempo (s)")
    plt.title("Variação de Desempenho por Algoritmo")
    plt.legend()
    plt.tight_layout()
    plt.show()

# Gráfico 3: Boxplot para visualizar a distribuição dos tempos de execução (se múltiplos dados estiverem presentes)
if num_testes > 1:
    plt.figure(figsize=(12, 8))
    plt.boxplot(dados, labels=algoritmos)
    plt.xlabel("Algoritmos")
    plt.ylabel("Tempo (s)")
    plt.title("Distribuição dos Tempos de Execução por Algoritmo")
    plt.tight_layout()
    plt.show()

# Gráfico 4: Gráfico de pizza para distribuição do tempo total por algoritmo
tempos_totais = [sum(dado) for dado in dados]
plt.figure(figsize=(8, 8))
plt.pie(tempos_totais, labels=algoritmos, autopct='%1.1f%%', startangle=140)
plt.title("Distribuição de Tempo Total por Algoritmo")
plt.show()
```

3.1.2 Escolhas de Implementação

Falando sobre as decisões de implementação do código, a ordem anterior foi realizada por alguns fatores. Acreditamos que organizando as células na ordem citada anteriormente, permitimos uma maior leitura e compreensão por parte dos usuários que vão utilizar este código, uma vez que, primeiro são colocados os algoritmos utilizados em células diferentes para melhor visualização de cada algoritmo utilizado sem poluição com outros aspectos do código não relacionados, depois, é disponibilizada a parte que apresenta dados de desempenho e permite configurações na consulta realizada, e por fim, temos a célula que contém o código referente aos aspectos gráficos. Essa organização também permite que sejam realizadas modificações nos parâmetros dos algoritmos mais facilmente, uma vez que cada algoritmo está separado em uma célula diferente para que seja facilmente identificado, analisado e, se o usuário preferir, modificado.

Vale ressaltar que para o algoritmo de Rabin-Karp temos duas variantes do código, um que utiliza um hashing comum, e um que utiliza um hashing mais elaborado. Isso é feito para fins de comparação, uma vez que este algoritmo depende fortemente da escolha do hashing utilizado. Com o hashing mais elaborado, utilizamos o hashing proposto pelos próprios autores do algoritmo e, com o hashing mais comum, utilizamos um hashing polinomial simples. Para o hashing mais elaborado também realizamos uma análise prévia

que adiciona um custo inicial no algoritmo, mas deve economizar tempo de execução durante a busca.

Também implementamos um algoritmo ingênuo para termos uma análise do desempenho de um algoritmo que não utiliza nenhuma técnica da literatura para funcionar. Ele representa uma abordagem mais intuitiva para resolver o problema de busca por padrões. Esperamos que, com isso, possamos ver melhor qual é o ganho por aplicar uma solução algorítmica mais elaborada para o problema citado.

O algoritmo Aho-Corasick usa uma classe Python devido à sua necessidade de um autômato com links de falha, permitindo que o estado interno persista e seja acessível durante a construção e execução para múltiplos padrões. Esse encapsulamento melhora a modularidade e reutilização, especialmente em buscas complexas, ao passo que os outros algoritmos processam os padrões de maneira mais direta e não dependem de estruturas de dados complexas que precisem persistir entre execuções.

O código utilizado pode ser acessado no Colab [1].

3.2 FUNCIONALIDADES

O código permite que analisemos a implementação de diferentes algoritmos de busca de string, bem divididos pela utilização de funções e das células do colab.

Também conseguimos obter dados de desempenho sobre os algoritmos, com o uso de uma interface simples. Nela, podemos inserir um pedaço de texto manualmente ou carregar um arquivo de extensão “txt” apertando o botão “Carregar Arquivo .txt” e selecionando o arquivo desejado. Esse arquivo é diretamente carregado em memória, sem exibir o texto na caixa apropriada a fim de minimizar o tempo de espera do usuário para a execução dos testes, caso seja desejado. Esse mesmo botão realiza uma contagem de quantos arquivos foram carregados pelo tempo de execução do programa, o que confirma a seleção de um novo arquivo, algo importante dado o longo tempo de carregamento para textos de grande tamanho. O botão “Exibir Texto em Caixa” traz o conteúdo armazenado em memória pelo botão descrito anteriormente para a caixa de texto, para caso o usuário deseje visualizar ou modificar o texto carregado. Apertando o botão “Testar” podemos realizar a busca pelo padrão no texto carregado, independentemente de ele estar sendo exibido ou não.

Existem recursos de avisos de falha, onde o usuário é notificado que é necessário ter um texto e um padrão com caracteres preenchidos para que o programa funcione corretamente. Outro aviso de falha decorre da tentativa de exibir um texto sem que haja um salvo em memória. Utilizando o separador “;;”, podemos realizar mais de uma busca simultaneamente, que será feita de uma vez só pelo algoritmo de Aho-Corasick que suporta

[1] https://colab.research.google.com/drive/1Gndn_OQKDMNMxVKi6qkcbNd7_TWT2aaD?usp=sharing

buscas múltiplas, ou, em sequência pelos outros algoritmos. Após a busca ser solicitada, teremos a informação das posições em que cada padrão desejado foi encontrado no texto original e da demora que cada algoritmo levou, em média.

Por fim, na última célula, o programa gera gráficos baseados nos dados obtidos anteriormente, onde conseguimos ver com mais detalhes as informações sobre o desempenho dos algoritmos para o texto e padrão (ou padrões) informados.

3.3 APLICAÇÕES

Como o código fornecido é capaz de fornecer gráficos e informações que ajudem a observar o desempenho de diferentes algoritmos de busca de string, esperamos que ele possa ser utilizado por estudantes de computação que tenham interesse no tema, seja para observar a diferença de performance em casos mais simples, ou, para realizar cadeias de teste mais robustas que auxiliem em seus próprios estudos científicos.

Esperamos também que o código possa ajudar alunos de computação e programadores da área na escolha de um algoritmo de busca que se encaixe melhor com a aplicação desenvolvida ou utilizada por eles, uma vez que, o desempenho dos algoritmos está fortemente ligado ao tipo de texto que se deseja procurar.

4 RESULTADOS COMPUTACIONAIS

Este capítulo englobará os resultados computacionais, determinando a metodologia que será seguida durante nossa análise de desempenho dos algoritmos, os testes que serão realizados e, por fim, utilizaremos recursos gráficos para melhor visualização dos resultados obtidos a fim de interpretá-los os mesmos com mais clareza.

4.1 METODOLOGIA

4.1.1 Considerações Iniciais

Durante o processo de testes, utilizamos como texto base o livro (SHAKESPEARE, 2024) disponível na plataforma Gutenberg, por ser um livro de domínio público bem extenso. Este será usado pois acreditamos que em um grande texto poderemos analisar melhor o desempenho de cada algoritmo. Para uma das buscas achamos interessante utilizar um texto diferente, similar às cadeias de DNA grandes[1]. Isto é feito em parte por ser uma das aplicações mais comuns para algoritmos de busca de string, e também porque para este teste em específico o outro texto utilizado não seria apropriado.

O principal critério considerado durante a realização dos testes será o tempo de execução dos algoritmos em diferentes cenários, a fim de obtermos uma análise justa que leva em consideração as vantagens e desvantagens inerentes a cada algoritmo. Também levaremos em consideração o uso de memória de cada algoritmo.

Todos os testes foram executados dez vezes, e o tempo obtido foi uma média dessas execuções, a fim de reduzir a variância dos testes, obtendo resultados mais confiáveis. Optamos por realizar um número relativamente baixo de repetições pois os algoritmos de estudo são determinísticos, tendo menor taxa de variabilidade. Devido a isso as repetições são principalmente para evitar pequenas variações de desempenho devido ao uso de hardware.

Em um dos casos de teste realizamos uma busca por padrões múltiplos. Como apenas o algoritmo de Aho-Corasick busca esses padrões simultaneamente, para os outros algoritmos realizamos buscas sucessivas e calculamos o somatório do tempo gasto em cada uma delas como tempo médio total.

Por fim, em um único caso de teste utilizamos um específico formado por uma sequência de nove letras “a” seguidas por uma letra “b”, repetindo esse padrão de dez letras cinquenta mil vezes a fim de gerar um texto bem grande. Este caso de teste com esse texto específico foi utilizado com a finalidiade de atacar o ponto fraco do algoritmo de Boyer-Moore-Horspool, que tem o menor tempo médio, a fim de ressaltar que todos os algoritmos possuem casos em que não são a melhor opção.

[1] https://github.com/Lucas-Capris/TCC/blob/main/DNA.txt

4.1.2 Ferramentas de Análise

Para realizar a análise, primeiro, implementamos os algoritmos desejados utilizando a linguagem de programação Python como vimos no capítulo anterior. Usamos o ambiente virtual 'Google Collab' para realizar os testes por ser um ambiente que possui hardware unificado, a fim de reduzir possíveis diferenças de desempenho por performance individual das máquinas. Os testes também serão executados localmente com o uso do profiler para podermos realizar uma medida do uso de memória de cada algoritmo.

4.1.3 Casos de Teste

Ao considerarmos os diferentes aspectos dos algoritmos e as possíveis buscas realizadas optamos por realizar os seguintes casos de teste:

- Uma busca por uma expressão comum que ocorre múltiplas vezes no decorrer da obra escolhida. Este teste é usado para representar uma busca comumente realizada por um padrão esperado em uma escala mais elevada, dado que o padrão será encontrado diversas vezes em um texto extenso.

- Uma busca por um padrão com repetição de caracteres. Este teste é usado para valorizar os algoritmos que se beneficiam de repetição de caracteres no padrão. Vale ressaltar que a repetição por caracteres é bem comum em busca de padrões em sequências de DNA que é uma aplicação comum de algoritmos de busca de string. É neste teste que será utilizado um texto diferente, pois, não há padrões com uma quantidade considerável de repetições nas obras de Shakespeare utilizadas para os outros testes.

- Uma busca de múltiplos padrões simultaneamente. Este teste é usado em especial para ressaltar o benefício do algoritmo de Aho-Corasick por suprir as necessidades de um usuário que deseje realizar tal busca.

- Uma busca que possua um erro proposital no final do padrão. Este teste é usado para visualizarmos o caso de falha. A disparidade ao final serve para destacar a qualidade de otimização dos algoritmos para esses casos em particular.

- Uma busca que possua um erro proposital no final de um padrão grande. Este teste é uma extensão do anterior, mas tentando extrapolar o caso de erro para vermos o desempenho dos algoritmos no pior caso.

- Uma busca por padrões de tamanhos grandes e variados. Este teste é usado para exemplificar a capacidade de cada algoritmo de fazer eficientemente a correspondência de cada caractere de forma extensa.

- Uma busca utilizando a letra “a” como padrão em um texto com uma sequência de nove letras “a” seguido de um “b”. Como mencionamos na metodologia, esse caso de teste tem como finalidade testar um caso onde o algoritmo de Boyer-Moore-Horspool não é eficiente.

Acreditamos que com esses casos de teste, poderemos obter uma boa noção comparativa a respeito do desempenho de cada algoritmo estudado. Vale ressaltar que serão realizadas cinco instâncias para cada um desses casos de teste, resultando em um total de trinta testes por algoritmo.

4.2 TESTES

4.2.1 Busca por padrão comum que ocorre múltiplas vezes

- Input: Time
  - **Algoritmo Ingênuo**
    Tempo médio: 2.032654 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 2.720915 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 3.063333 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.456764 segundos
  - **Aho-Corasick**
    Tempo médio: 1.881719 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 1.843414 segundos
- Input: People
  - **Algoritmo Ingênuo**
    Tempo médio: 1.910308 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 2.699640 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 3.749236 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.325348 segundos

  - **Aho-Corasick**
    Tempo médio: 1.859922 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 1.742824 segundos

- Input: Words
  - **Algoritmo Ingênuo**
    Tempo médio: 2.037430 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 2.665241 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 2.931842 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.500983 segundos
  - **Aho-Corasick**
    Tempo médio: 1.714148 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 1.831605 segundos

- Input: Good morrow
  - **Algoritmo Ingênuo**
    Tempo médio: 1.912037 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 2.683270 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 3.936182 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.188048 segundos
  - **Aho-Corasick**
    Tempo médio: 1.815642 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 1.703515 segundos

- Input: Fare thee well
  - **Algoritmo Ingênuo**
    Tempo médio: 1.909448 segundos

  - **Rabin-Karp (Básico)**
    Tempo médio: 2.691269 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 3.954490 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.179242 segundos
  - **Aho-Corasick**
    Tempo médio: 1.766011 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 1.716693 segundos

4.2.2 Busca por um padrão com repetição de caracteres

- Input:

  TTGAACTTCCCTAACAACATCGCAACATTCACCATACATAGACCTATACGTG
  CTAACAGCTCGACCGCGGTCCTGTATATCTGATGAGACTAGTGGCTGCTAC
  TACTGATTC

  - **Algoritmo Ingênuo**
    Tempo médio: 3.925724 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 5.066310 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 7.300014 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 1.174347 segundos
  - **Aho-Corasick**
    Tempo médio: 4.383061 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 3.374970 segundos

- Input:

  GATATACTTGACGAGCTTAGAACGCGTACGATCGCTAGACCGACTACTGGT
  CAAGTATAGATCAATCCACTTTCACT

  - **Algoritmo Ingênuo**
    Tempo médio: 3.914112 segundos

  - **Rabin-Karp (Básico)**
    Tempo médio: 5.203755 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 7.499078 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.603962 segundos
  - **Aho-Corasick**
    Tempo médio: 4.618170 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 3.368875 segundos

- Input:

  TAGGTAGTACAAACCGAAGGGGTCATCGGTCGGGAGCATTGTCCAAAGTT
  CACTGCTACCGCCAAATCGGACACATGCTTATCCAGATACTCCTC

  - **Algoritmo Ingênuo**
    Tempo médio: 3.910626 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 5.128565 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 7.232424 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.941758 segundos
  - **Aho-Corasick**
    Tempo médio: 4.619378 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 3.420189 segundos

- Input:

  GATCGTTTTATTCGTCTTTTGCTTCAGAGCACCGGTGGTTAGGTAATACC
  TTGCAAAT

  - **Algoritmo Ingênuo**
    Tempo médio: 3.917033 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 5.071007 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 7.278666 segundos

  - **Boyer-Moore-Horspool**
    Tempo médio: 1.037130 segundos
  - **Aho-Corasick**
    Tempo médio: 4.585502 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 3.448294 segundos

- Input:

  CCTTCTATAGATGACGAGCTAACTAGCGCCAGAAGGACCAACATGTTGTA
  TCTCATGA

  - **Algoritmo Ingênuo**
    Tempo médio: 3.913287 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 5.074132 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 7.500692 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 1.599930 segundos
  - **Aho-Corasick**
    Tempo médio: 4.370668 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 3.426242 segundos

4.2.3 Busca de múltiplos padrões simultaneamente

- Input: creatures;;desire;;noble

  - **Algoritmo Ingênuo**
    Tempo médio: 5.891490 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 5.203755 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 7.300014 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 1.102325 segundos
  - **Aho-Corasick**
    Tempo médio: 2.105383 segundos

  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 5.290115 segundos

- Input: stones;;indeed;;melancholy
  - **Algoritmo Ingênuo**
    Tempo médio: 5.797496 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 5.128565 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 7.499078 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 1.001150 segundos
  - **Aho-Corasick**
    Tempo médio: 1.997538 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 5.374847 segundos

- Input: change;;which;;come
  - **Algoritmo Ingênuo**
    Tempo médio: 5.743672 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 5.071007 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 7.232424 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 1.283580 segundos
  - **Aho-Corasick**
    Tempo médio: 1.991951 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 5.266623 segundos

- Input: what;;moralize;;tears
  - **Algoritmo Ingênuo**
    Tempo médio: 5.968776 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 5.074132 segundos

  - **Rabin-Karp (Otimizado)**
    Tempo médio: 7.278666 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 1.188079 segundos
  - **Aho-Corasick**
    Tempo médio: 2.090210 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 5.321887 segundos

- Input: yes;;for his;;swearing
  - **Algoritmo Ingênuo**
    Tempo médio: 5.925920 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 8.36594 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 7.500692 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 1.314930 segundos
  - **Aho-Corasick**
    Tempo médio: 2.067509 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 5.301885 segundos

4.2.4 Busca que possua um erro proposital no final do padrão

- Input: good captaim
  - **Algoritmo Ingênuo**
    Tempo médio: 2.010428 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 2.725665 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 4.050615 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.183548 segundos
  - **Aho-Corasick**
    Tempo médio: 1.912295 segundos

  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 1.706105 segundos

- Input: cowart
  - **Algoritmo Ingênuo**
    Tempo médio: 1.891034 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 2.717468 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 3.761700 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.449741 segundos
  - **Aho-Corasick**
    Tempo médio: 1.908616 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 1.830703 segundos

- Input: soothsayen
  - **Algoritmo Ingênuo**
    Tempo médio: 2.020059 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 2.707475 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 3.957451 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.216951 segundos
  - **Aho-Corasick**
    Tempo médio: 1.893132 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 1.734160 segundos

- Input: foreseee
  - **Algoritmo Ingênuo**
    Tempo médio: 1.880325 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 2.698281 segundos

  - **Rabin-Karp (Otimizado)**
    Tempo médio: 4.019773 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.255220 segundos
  - **Aho-Corasick**
    Tempo médio: 1.779095 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 1.775492 segundos

- Input: you shall paind
  - **Algoritmo Ingênuo**
    Tempo médio: 2.015657 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 2.739267 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 3.964177 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.164467 segundos
  - **Aho-Corasick**
    Tempo médio: 1.771641 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 1.776415 segundos

4.2.5 Busca que possua um erro proposital no final de um padrão grande

- Input:

  O that you were your self, but love you are
  No longer yours, than you yourself here live,
  Against this coming end you should prepare,
  And your sweet semblance to some other give.
  So should that beauty which you hold in lease
  Find no determination, then you were
  Yourself again after yourself's decease,
  When your sweet issue your sweet form should bear.
  Who lets so fair a house fall to decay,
  Which husbandry in honour might uphold,
  Against the stormy gusts of winter's day

And barren rage of death's eternal cold?
O none but unthrifts, dear my love you know,
You had a father, let your son say.

  - **Algoritmo Ingênuo**
    Tempo médio: 2.234598 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 2.746625 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 3.949764 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.033721 segundos
  - **Aho-Corasick**
    Tempo médio: 1.701286 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 1.831453 segundos

- Input:

  Lord of my love, to whom in vassalage
  Thy merit hath my duty strongly knit;
  To thee I send this written embassage
  To witness duty, not to show my wit.
  Duty so great, which wit so poor as mine
  May make seem bare, in wanting words to show it;
  But that I hope some good conceit of thine
  In thy soul's thought (all naked) will bestow it:
  Till whatsoever star that guides my moving,
  Points on me graciously with fair aspect,
  And puts apparel on my tattered loving,
  To show me worthy of thy sweet respect,
  Then may I dare to boast how I do love thee,
  Till then, not show my head where thou mayst prove mee.

  - **Algoritmo Ingênuo**
    Tempo médio: 2.233137 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 2.742948 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 3.923881 segundos

  - **Boyer-Moore-Horspool**
    Tempo médio: 0.031454 segundos
  - **Aho-Corasick**
    Tempo médio: 1.817478 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 1.690370 segundos

- Input:

  Those parts of thee that the world's eye doth view,
  Want nothing that the thought of hearts can mend:
  All tongues, the voice of souls, give thee that due,
  Uttering bare truth, even so as foes commend.
  Thy outward thus with outward praise is crowned,
  But those same tongues that give thee so thine own,
  In other accents do this praise confound
  By seeing farther than the eye hath shown.
  They look into the beauty of thy mind,
  And that in guess they measure by thy deeds,
  Then churls their thoughts (although their eyes were kind)
  To thy fair flower add the rank smell of weeds:
  But why thy odour matcheth not thy show,
  The soil is this, that thou dost common grown.

  - **Algoritmo Ingênuo**
    Tempo médio: 2.177131 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 2.867393 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 3.927678 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.027534 segundos
  - **Aho-Corasick**
    Tempo médio: 1.696159 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 1.831396 segundos

- Input:

LE BEAU.
The eldest of the three wrestled with Charles, the Duke's wrestler, which Charles in a moment threw him and broke three of his ribs, that there is little hope of life in him. So he served the second, and so the third. Yonder they lie, the poor old man their father making such pitiful dole over them that all the beholders take his part with weeping.

ROSALIND.
Alas?

  - **Algoritmo Ingênuo**
    Tempo médio: 2.098454 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 2.681253 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 3.930965 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.043849 segundos
  - **Aho-Corasick**
    Tempo médio: 1.750213 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 1.745105 segundos

- Input:

  MESSENGER.
  The English are embattl'd, you French peers.

  CONSTABLE.
  To horse, you gallant princes! straight to horse!
  Do but behold yon poor and starved band,
  And your fair show shall suck away their souls,
  Leaving them but the shales and husks of men.
  There is not work enough for all our hands;
  Scarce blood enough in all their sickly veins
  To give each naked curtle-axe a stain,
  That our French gallants shall today draw out,
  And sheathe for lack of sport. Let us but blow on them,

The vapour of our valour will o'erturn them.
'Tis positive 'gainst all exceptions, lords,
That our superfluous lackeys and our peasants,
Who in unnecessary action swarm
About our squares of battle, were enough
To purge this field of such a hilding foe,
Though we upon this mountain's basis by
Took stand for idle speculation,
But that our honours must not. What's to say?
A very little little let us do,
And all is done. Then let the trumpets sound
The tucket sonance and the note to mount;
For our approach shall so much dare the field
That England shall crouch down in fear and yield.

Enter Grandpré...

- **Algoritmo Ingênuo**
  Tempo médio: 2.059547 segundos
- **Rabin-Karp (Básico)**
  Tempo médio: 2.728640 segundos
- **Rabin-Karp (Otimizado)**
  Tempo médio: 3.900829 segundos
- **Boyer-Moore-Horspool**
  Tempo médio: 0.023606 segundos
- **Aho-Corasick**
  Tempo médio: 1.679323 segundos
- **Knuth-Morris-Pratt (KMP)**
  Tempo médio: 1.801729 segundos

4.2.6 Busca por padrões de tamanhos grandes e variados

- Input:

For shame deny that thou bear'st love to any
Who for thyself art so unprovident.
Grant if thou wilt, thou art beloved of many,
But that thou none lov'st is most evident:
For thou art so possessed with murd'rous hate,

That 'gainst thyself thou stick'st not to conspire,
Seeking that beauteous roof to ruinate
Which to repair should be thy chief desire:
O change thy thought, that I may change my mind,
Shall hate be fairer lodged than gentle love?
Be as thy presence is gracious and kind,
Or to thyself at least kind-hearted prove,
Make thee another self for love of me,
That beauty still may live in thine or thee

  - **Algoritmo Ingênuo**
    Tempo médio: 2.197377 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 2.592659 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 3.815594 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.033256 segundos
  - **Aho-Corasick**
    Tempo médio: 1.700590 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 1.699828 segundos

- Input:

  Nay, that's past praying for. I have peppered two of them. Two I am sure I have paid, two rogues in buckram suits. I tell thee what, Hal, if I tell thee a lie, spit in my face, call me horse. Thou knowest my old ward. Here I lay, and thus I bore my point. Four rogues in buckram let drive at me

  - **Algoritmo Ingênuo**
    Tempo médio: 2.217563 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 2.572389 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 3.822582 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.051639 segundos

  - **Aho-Corasick**
    Tempo médio: 1.668391 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 1.660644 segundos

- Input:

  Octavius, I have seen more days than you;
  And though we lay these honours on this man,
  To ease ourselves of divers sland'rous loads,
  He shall but bear them as the ass bears gold,
  To groan and sweat under the business,
  Either led or driven, as we point the way;
  And having brought our treasure where we will,
  Then take we down his load, and turn him off,
  Like to the empty ass, to shake his ears,
  And graze in commons

  - **Algoritmo Ingênuo**
    Tempo médio: 2.166730 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 2.594960 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 3.776715 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.046389 segundos
  - **Aho-Corasick**
    Tempo médio: 1.817851 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 1.661090 segundos

- Input:

  Let me look back upon thee. O thou wall
  That girdles in those wolves, dive in the earth
  And fence not Athens! Matrons, turn incontinent!
  Obedience fail in children! Slaves and fools,
  Pluck the grave wrinkled senate from the bench
  And minister in their steads! To general filths
  Convert, o' th' instant, green virginity,

Do't in your parents' eyes! Bankrupts, hold fast;
Rather than render back, out with your knives
And cut your trusters' throats! Bound servants, steal!
Large-handed robbers your grave masters are,
And pill by law. Maid, to thy master's bed,
Thy mistress is o' th' brothel. Son of sixteen,
Pluck the lined crutch from thy old limping sire,
With it beat out his brains! Piety and fear,
Religion to the gods, peace, justice, truth,
Domestic awe, night-rest and neighbourhood,
Instruction, manners, mysteries and trades,
Degrees, observances, customs and laws,
Decline to your confounding contraries,
And let confusion live! Plagues incident to men,
Your potent and infectious fevers heap
On Athens, ripe for stroke! Thou cold sciatica,
Cripple our senators, that their limbs may halt
As lamely as their manners! Lust and liberty,
Creep in the minds and marrows of our youth,
That 'gainst the stream of virtue they may strive
And drown themselves in riot! Itches, blains,
Sow all th' Athenian bosoms, and their crop
Be general leprosy! Breath infect breath,
That their society, as their friendship, may
Be merely poison! Nothing I'll bear from thee
But nakedness, thou detestable town!
Take thou that too, with multiplying bans!
Timon will to the woods, where he shall find
Th' unkindest beast more kinder than mankind.
The gods confound—hear me, you good gods all!—
Th' Athenians both within and out that wall,
And grant, as Timon grows, his hate may grow
To the whole race of mankind, high and low!
Amen

- **Algoritmo Ingênuo**
  Tempo médio: 2.172350 segundos
- **Rabin-Karp (Básico)**
  Tempo médio: 2.586614 segundos

  - **Rabin-Karp (Otimizado)**
    Tempo médio: 3.800054 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.020972 segundos
  - **Aho-Corasick**
    Tempo médio: 1.828570 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 1.674535 segundos

- Input:

  Now they are clapper-clawing one another; I'll go look on. That dissembling abominable varlet, Diomed, has got that same scurvy doting foolish young knave's sleeve of Troy there in his helm. I would fain see them meet, that that same young Trojan ass that loves the whore there might send that Greekish whoremasterly villain with the sleeve back to the dissembling luxurious drab of a sleeve-less errand. O' the other side, the policy of those crafty swearing rascals that stale old mouse-eaten dry cheese, Nestor, and that same dog-fox, Ulysses, is not prov'd worth a blackberry. They set me up, in policy, that mongrel cur, Ajax, against that dog of as bad a kind, Achilles; and now is the cur, Ajax prouder than the cur Achilles, and will not arm today; whereupon the Grecians begin to proclaim barbarism, and policy grows into an ill opinion

  - **Algoritmo Ingênuo**
    Tempo médio: 2.214377 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 2.718859 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 3.810526 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.033416 segundos
  - **Aho-Corasick**
    Tempo médio: 1.834440 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 1.668925 segundos

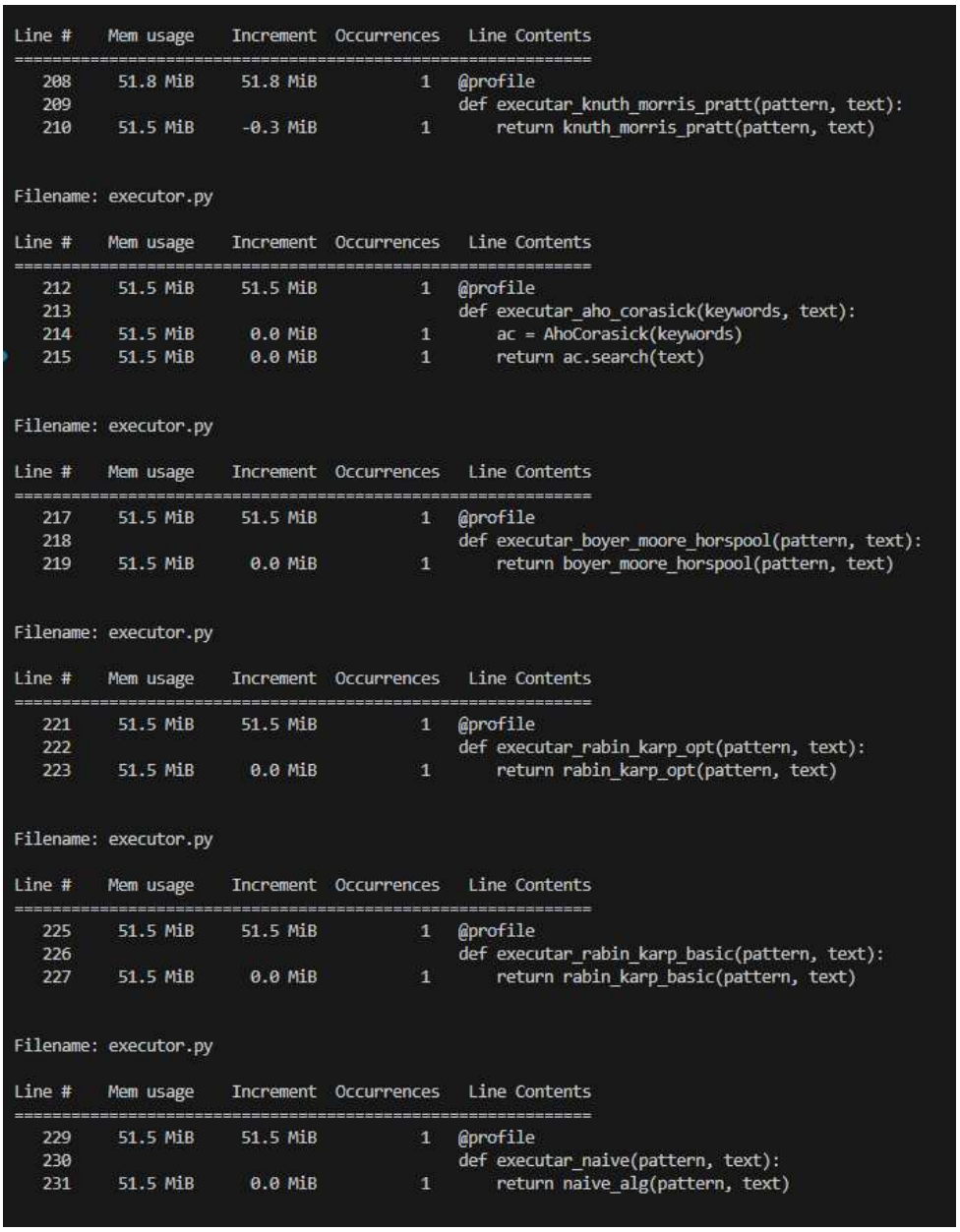
4.2.7 Busca utilizando a letra “a” como padrão em um texto com uma sequência de nove letras “a” seguido de um “b”

- input: a

  - **Algoritmo Ingênuo**
    Tempo médio: 0.152622 segundos
  - **Rabin-Karp (Básico)**
    Tempo médio: 0.363817 segundos
  - **Rabin-Karp (Otimizado)**
    Tempo médio: 0.244247 segundos
  - **Boyer-Moore-Horspool**
    Tempo médio: 0.184541 segundos
  - **Aho-Corasick**
    Tempo médio: 0.361677 segundos
  - **Knuth-Morris-Pratt (KMP)**
    Tempo médio: 0.125870 segundos

4.2.8 Teste de Uso de Memória Utilizando o Profiler

Durante a execução dos algoritmos usando o profiler, obtemos praticamente o mesmo uso de memória para todos os casos testados, portanto, chegamos a conclusão que o uso de memória de todos os algoritmos é similar, portanto, decidimos desconsiderar este fator na análise comparativa. Abaixo, podemos ver a saída com o uso de memória de uma execução do profiler:



```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
   208     51.8 MiB     51.8 MiB           1   @profile
   209                                         def executar_knuth_morris_pratt(pattern, text):
   210     51.5 MiB     -0.3 MiB           1       return knuth_morris_pratt(pattern, text)


Filename: executor.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
   212     51.5 MiB     51.5 MiB           1   @profile
   213                                         def executar_aho_corasick(keywords, text):
   214     51.5 MiB      0.0 MiB           1       ac = AhoCorasick(keywords)
   215     51.5 MiB      0.0 MiB           1       return ac.search(text)


Filename: executor.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
   217     51.5 MiB     51.5 MiB           1   @profile
   218                                         def executar_boyer_moore_horspool(pattern, text):
   219     51.5 MiB      0.0 MiB           1       return boyer_moore_horspool(pattern, text)


Filename: executor.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
   221     51.5 MiB     51.5 MiB           1   @profile
   222                                         def executar_rabin_karp_opt(pattern, text):
   223     51.5 MiB      0.0 MiB           1       return rabin_karp_opt(pattern, text)


Filename: executor.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
   225     51.5 MiB     51.5 MiB           1   @profile
   226                                         def executar_rabin_karp_basic(pattern, text):
   227     51.5 MiB      0.0 MiB           1       return rabin_karp_basic(pattern, text)


Filename: executor.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
   229     51.5 MiB     51.5 MiB           1   @profile
   230                                         def executar_naive(pattern, text):
   231     51.5 MiB      0.0 MiB           1       return naive_alg(pattern, text)
```

Figura 2 – Profiler

4.3 RESULTADOS

4.3.1 Resultado dos testes

Com os dados obtidos dos testes, geramos diferentes gráficos a fim de ajudar a visualizar e interpretar os resultados. É importante destacar que os gráficos apresentados foram gerados por meio de um script separado do utilizado no programa principal, com o objetivo de abranger em conjunto os diferentes testes realizados, contribuindo assim para a análise da performance geral de todos os algoritmos. Abaixo, podemos visualizar os gráficos obtidos:

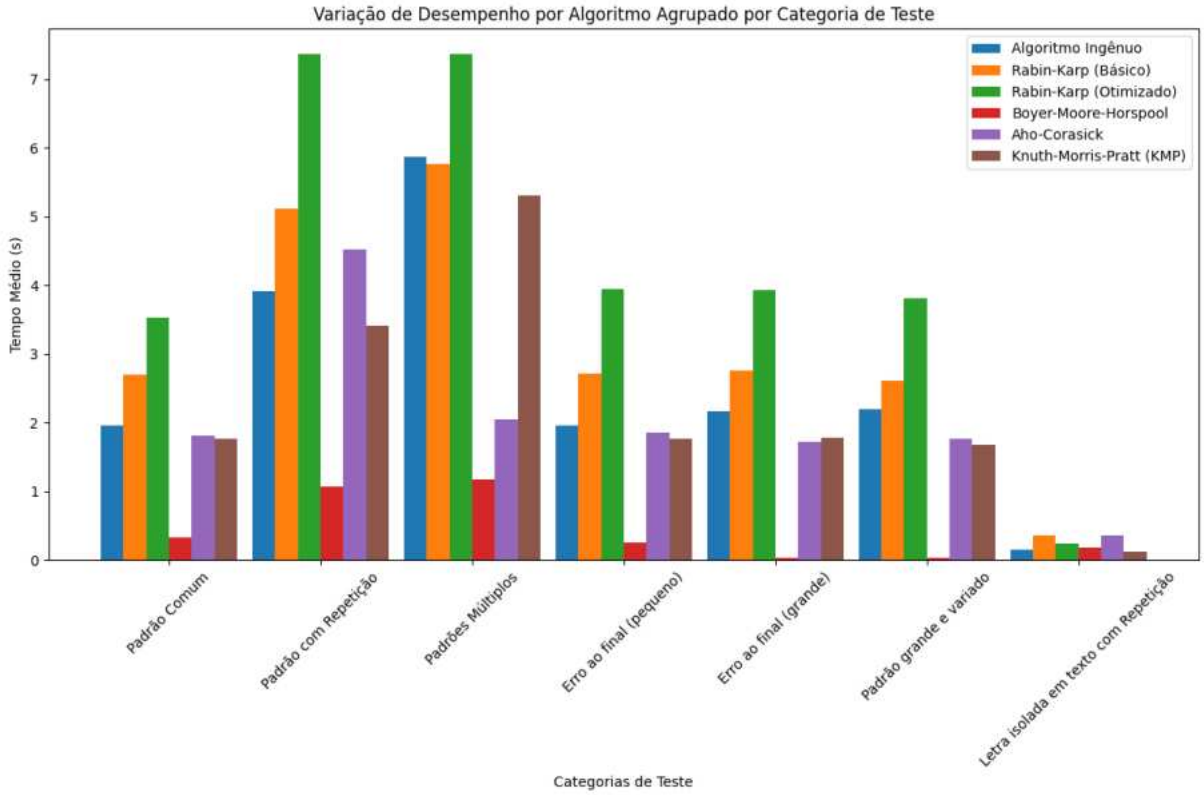


Figura 3 – Gráfico de Barras Verticais para Comparação de Tempo por Caso de Teste

O gráfico acima permite que vejamos o desempenho de cada algoritmo por caso de teste. Podemos observar que em geral, Boyer-Moore-Horspool apresenta melhor desempenho que os demais, com exceção do último caso. Observamos também que o desempenho de Aho-Corasick e Knuth-Morris-Pratt é muito similar, variando levemente dependendo do caso de teste observado. O algoritmo ingênuo também obteve um desempenho razoavelmente bom, enquanto os algoritmos de Rabin-Karp, em especial o que utilizava um hashing mais complexo, não tiveram um desempenho tão bom. Por fim, podemos ver a grande melhora no desempenho de Aho-Corasick na busca por padrões múltiplos em relação aos outros algoritmos, com exceção de Boyer-Moore-Horspool que conseguiu ter um tempo de execução menor mesmo fazendo as buscas de maneira seqencial, entretanto, vale ressaltar que buscamos apenas três padrões simultaneamente, portanto, ao adicionar cada vez mais padrões, o desempenho de Aho-Corasick será cada vez melhor em relação aos outros algoritmos.

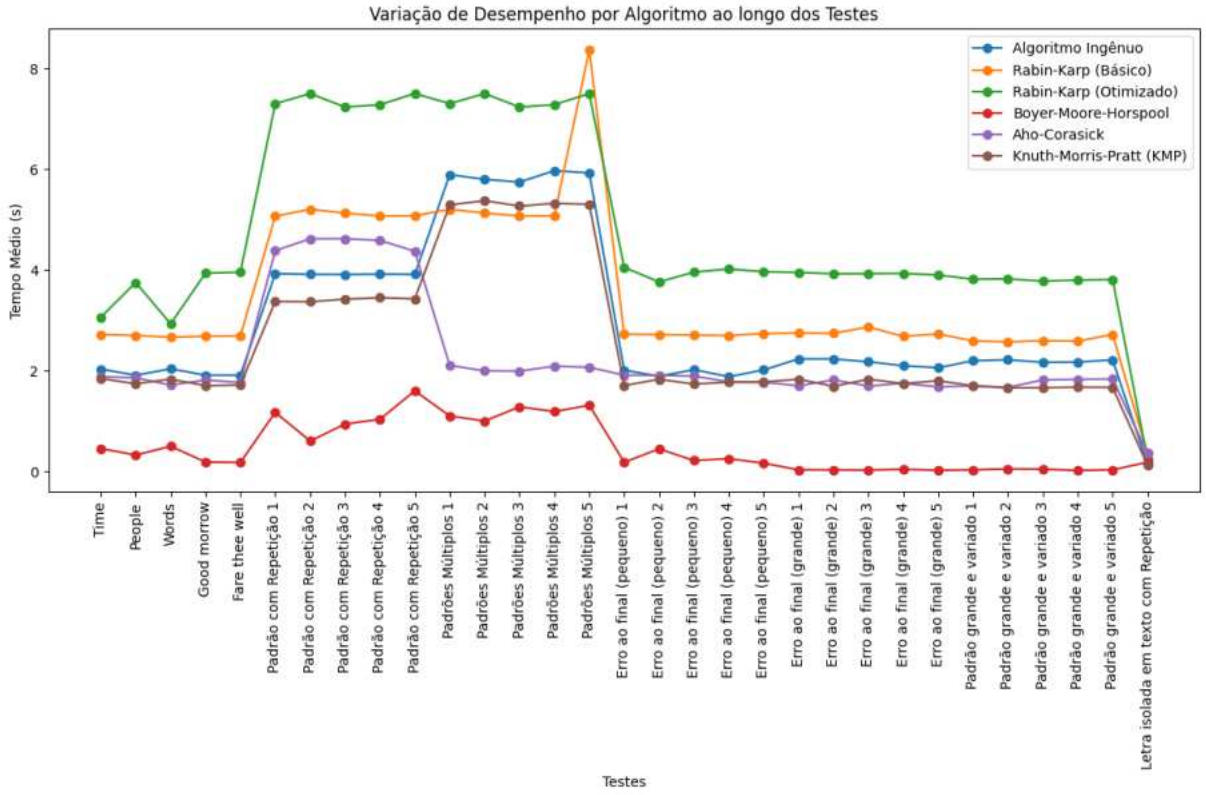


Figura 4 – Gráfico de Linhas para Observar a Variação de Desempenho por Teste

O gráfico acima é similar ao anterior, porém, nele vemos o comportamento dos algoritmos para cada instância de cada caso de teste, permitindo que análise mais profunda. Podemos ver que no geral, os algoritmos não oscilaram tanto por instância uma vez que elas apresentam conceitos similares. Outro ponto também a ser considerado, é que grande parte da oscilação dos algoritmos ocorreu quando mudamos o texto fornecido, no segundo e no último caso de teste, e, no terceiro caso de teste que realiza buscas múltiplas, aumentando o tempo de execução para todos os algoritmos com exceção de Aho-Corasick que consegue realizar essas buscas simultaneamente. Uma exceção, entretato, é Boyer-Moore-Horspool, cujo desempenho oscila consideravelmente por instância, o que faz sentido considerando que na técnica utilizada por esse algoritmo o padrão utilizado é de grande importância. O algoritmo de Rabin-Karp Otimizado também teve uma oscilação moderada no primeiro caso de teste. Por fim, podemos observar que o algoritmo de Rabin-Karp básico teve um ponto destoante na última instâcia do terceiro caso de teste, que pode ter ocorrido por ter procurado palavras com uma quantidade grande de colisões, dado o hashing simplificado que ele utiliza.

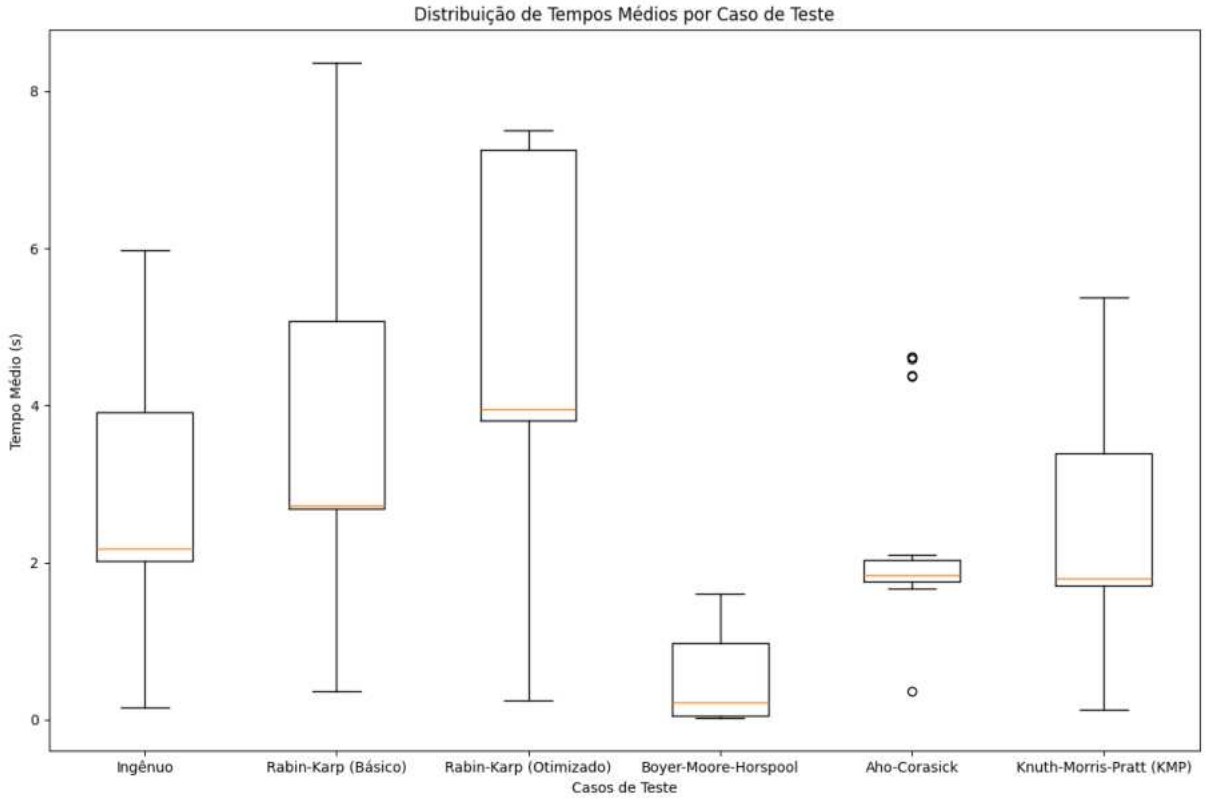


Figura 5 – Boxplot para Visualizar a Distribuição dos Tempos de Execução

No gráfico acima, podemos observar melhor a distribuição dos tempos de cada algoritmo observando bem a variação de seus dados. Vemos que o algoritmo ingênuo possui uma mediana que se concentra na parte de baixo da caixa, indicando que a maior parte de seus resultados está na parte inferior, entretanto, a taxa de variação dele é relativamente alta, o que faz sentido dado que ele não utiliza nenhuma técnica específica, fazendo com que seu tempo de execução possa variar muito dependendo do texto e do padrão utilizados. O Rabin-Karp básico obteve um desempenho similar porém um pouco inferior ao do algoritmo ingênuo, sua taxa de variação acabou sendo maior, provavelmente devido ao seu desempenho fortemente ligado a quantidade de colisões que ocorrem durante o texto. O algoritmo de Rabin-Karp otimizado teve tempo médio similar ao de Rabin-Karp básico, mas com tempo médio superior porém menos variação, o que faz sentido dado que o hashing mais complexo que ele utiliza gera menos colisões em média, tornando o tempo de execução do algoritmo mais estável. Boyer-Moore apresentou o menor tempo médio de execução, com uma taxa de variação também relativamente pequena, o que faz sentido dado que seu tempo médio realmente é menor que o de outros algoritmos, e, mesmo com sua variação dependendo do padrão utilizado, o tempo realizado em suas buscas ainda foi bem baixo no geral. O algoritmo de Aho-Corasick também obteve um bom desempenho, sendo inferior apenas ao algoritmo de Boyer-Moore-Horspool, porém, com uma taxa de variação ainda menor, entretanto, ele obteve alguns outliers durante sua execução. Isso

ocorreu provavelmente por seus seus valores serem tão próximos, que fez com que os tempos diferenciados dos testes de busca múltipla, que melhoraram seu desempenho, e dos testes que mudaram o texto utilizado como o segundo e o último fossem vistos como outliers. Por fim, o algoritmo de Knuth-Morris-Pratt também obteve um bom desempenho em média, porém com uma taxa de variação razoável, similar ao algoritmo ingênuo, entretanto, com uma mediana e tempos médios menores indicando que, em média, o tempo de execução desse algoritmo foi mais baixo.

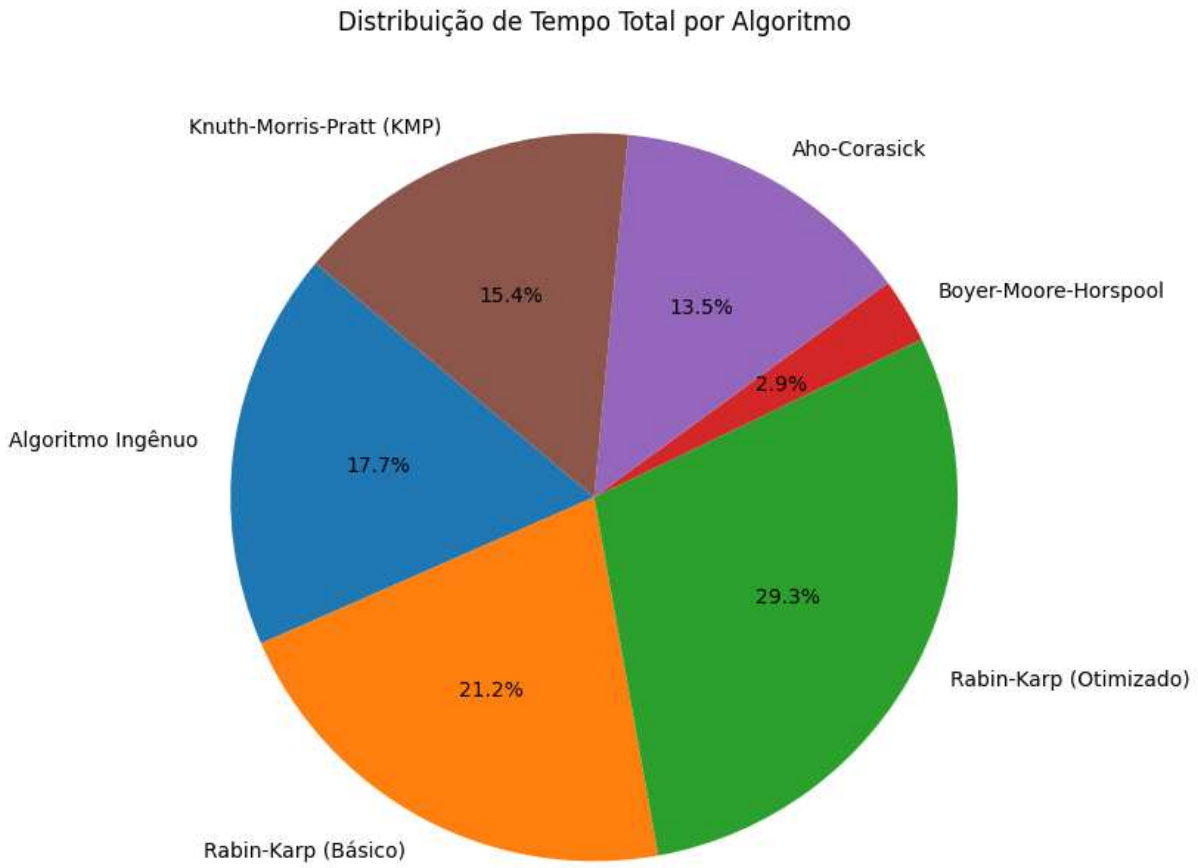


Figura 6 – Gráfico de Pizza para Tempo Total por Algoritmo

O último gráfico é um pouco mais simples, mas serve para dar uma visão geral do desempenho de cada algoritmo. Utilizamos um gráfico de pizza que mostra o tempo de execução total de cada algoritmo considerando todos os casos de teste. Podemos ver que o algoritmo com melhor desempenho em geral foi o de Boyer-Moore-Horspool que ocupou apenas 3% do total de tempo de gasto. Depois, tivemos os algoritmos de Aho-Corasick e Knuth-Morris-Pratt com desempenhos bem similares, porém com uma leve vantagem para o de Aho-Corasick que pode ser devido a vantagem nos testes de busca múltipla, o que exalta o quão similar o desempenho desses algoiritmos se mostrou para

buscas simples em geral. Em sequência, temos o algoritmo ingênuo que mesmo utilizando uma abordagem simples para o problema de busca de strings ainda consegue ter um desempenho semelhante ao de algoritmos mais robustos nos casos de teste utilizados. Por fim, o Rabin-Karp, tanto básico quanto otimizado, obtiveram o pior desempenho em geral. Um fato curioso foi que o Rabin-Karp otimizado acabou possuindo um desempenho abaixo do básico, que pode se dar devido ao custoso cálculo de seus hashes para evitar colisões, que provavelmente gerou um custo que não compensou o ganho com a menor quantidade de colisões esperadas pela sua robustez.

4.3.2 Análise final

Com todos os dados obtidos anteriormente, podemos inferir os resultados de cada algoritmo, obtendo uma conclusão geral sobre cada um deles. Uso de memória será desconsiderado como fator, pois como mencionamos anteriormente, o uso de memória de todos os algoritmos acabou sendo bem similar.

- Algoritmo ingênuo: O algoritmo ingênuo acabou conseguindo um resultado melhor que o esperado, principalmente levando em conta a simplicidade de sua implementação. Seu tempo em média só foi superior ao dos algoritmos de Rabin-Karp. Além disso, ele possui uma grande taxa de variação, fazendo com que sua performance seja relativamente inconsistente. Ainda assim, dada sua simplicidade, o tempo de execução do mesmo acabou sendo razoável, evidenciando que a menos que um texto seja muito grande, ou, que se deseje realmente otimizar o processo de busca, o uso de um algoritmo mais simples não é tão punitivo.

- Rabin-Karp: O algoritmo de Rabin-Karp, em suas variações, possui uma implementação mais simples, que pode ser um fator positivo para casos aonde isto é preferível, entretanto, o desempenho de ambas acabou ficando abaixo do esperado. O Rabin-Karp básico teve um tempo médio levemente mais baixo do que o otimizado, configurando um melhor desempenho, entretanto, teve uma das maiores taxas de variação dentre os algoritmos, sendo menos consistente. A versão otimizada, entretanto, teve uma taxa de variação bem menor que sua versão básica, ainda que tenha tido o pior desempenho geral. Isto provavelmente ocorreu pois o menor número de colisões acabou não tendo ganho suficiente em comparação com seu cálculo mais custoso de hashes, o que ressalta a importância de equilibrar o ganho esperado com o custo de implementação do hash utilizado. Ainda assim, o Rabin-Karp otimizado teve uma menor taxa de variação em relação a sua versão básica, sendo preferível a mesma no geral em casos onde consistência tem mais importância que velocidade.

- Boyer-Moore-Horspool: O algoritmo de Boyer-Moore-Horspool acabou apresentando um desempenho excelente, o que ganha ainda mais força quando consideramos sua simples implementação. Isso está dentro do esperado, dado que é o algoritmo que possui o melhor desempenho médio, evidenciando a força que ele possui. Ainda assim, vale ressaltar que há casos aonde o algoritmo acaba possuindo um desempenho inferior, ressaltando a importância de uma análise mais localizada para a aplicação em que se deseja utilizar os algoritmos. Em especial, Boyer-Moore perde bastante eficiência em textos cujo alfabeto é menor e em buscas por padrões com frequente repetição de caracteres, algo bem comum por exemplo em alfabetos binários, no qual o algoritmo de Knuth-Morris-Pratt apresenta um desempenho melhor.

- Aho-Corasick: O algoritmo de Aho-Corasick teve um bom desempenho em geral, sendo similar ao de Knuth-Morris-Pratt, entretanto, devido a sua dificuldade de implementação pode não ser preferível em usos mais simples. Ainda assim, por ser o único algoritmo capaz de realizar buscas múltiplas, Aho-Corasick possui um nicho de aplicação claro, e, mesmo que possua um código mais complexo, é essencial em casos onde este tipo de busca seja realizada.

- Knuth-Morris-Pratt: O algoritmo de Knuth-Morris-Pratt também teve um bom desempenho em geral. Se comparado ao algoritmo de Aho-Corasick que obteve um desempenho similar, sua implementação mais simples pode tornar ele mais interessante para buscas únicas. Ainda assim, sua implementação não é mais simples que a dos outros algoritmos citados, então, isso também deve ser pesado.

Chegamos a conclusão que, em geral, Boyer-Moore-Horspool acabou possuindo um melhor resultado, ainda mais se levarmos em conta que sua implementação é relativamente simples e seu tempo médio de execução é bem baixo. Ainda assim, os algoritmos utilizados podem variar em desempenho dependendo do texto realizado e dos padrões de busca, como conseguimos ver um caso aonde Boyer-Moore-Horspool acabou sendo inferior, por isso, o conhecimento sobre os pontos fortes e fracos de cada algoritmo e a análise de usabilidade desejada para eles é de extrema importância na hora de decidir qual será utilizado.

5 CONCLUSÃO

Ao longo do trabalho, foram implementados, testados e comparados diferentes algoritmos de busca de strings, como o Ingênuo, Rabin-Karp (em suas variações básica e otimizada), Boyer-Moore-Horspool, Aho-Corasick e Knuth-Morris-Pratt. Cada um deles foi examinado sob os aspectos de performance, consistência, e complexidade de implementação. Esses pontos foram fundamentais para avaliar como cada algoritmo se comporta diante de diferentes casos de teste, abrangendo desde buscas simples até padrões com erros propositais e buscas múltiplas.

Com os resultados dos testes de forma detalhada, conseguimos identificar as forças e fraquezas de cada algoritmo. Por exemplo, a implementação básica do Rabin-Karp, embora simples, mostrou-se mais eficiente do que a versão otimizada em alguns cenários devido ao impacto das colisões de hash, enquanto a versão otimizada destacou-se pela consistência em situações mais complexas. O Boyer-Moore-Horspool, por sua vez, manteve uma performance destacada no geral, mas ligeiramente inconsistente, e o Aho-Corasick mostrou-se extremamente eficiente em buscas múltiplas. Tais análises reforçaram a importância de considerar as características específicas de cada problema ao selecionar o algoritmo adequado.

O processo de desenvolvimento deste trabalho no ambiente colaborativo do Google Colab foi uma experiência valiosa, oferecendo facilidade de uso, integração com bibliotecas, e um ambiente versátil para a execução de testes. No entanto, enfrentamos limitações ao utilizar ferramentas como o profiler para medição detalhada de desempenho, que precisou ser rodado localmente devido a restrições do Colab. Esse aspecto destacou a importância de ajustar as ferramentas de análise ao ambiente de execução, algo que demandou adaptação e aprendizado ao longo do processo.

Refletir sobre a eficiência e as limitações dos algoritmos, comparar as soluções e pensar criticamente sobre a escolha dos métodos exigiu aprofundamento teórico e pragmatismo. A análise inicial dos algoritmos demonstrou que, apesar de muitos deles serem estudados há décadas, sua importância permanece atual, especialmente em aplicações de grande escala e em contextos como buscas de texto em grandes volumes de dados. Os algoritmos de busca de string, portanto, seguem como ferramentas essenciais na computação, servindo de base para sistemas que exigem alta eficiência e precisão.

Acreditamos que este trabalho proporcionou um panorama abrangente e detalhado sobre os algoritmos de busca de strings, além de evidenciar a necessidade de desenvolvimento contínuo e adaptação de métodos para atender às exigências de novos contextos computacionais. A experiência adquirida reforça o papel desses algoritmos não apenas como soluções técnicas, mas como elementos fundamentais no estudo da eficiência computacional e das aplicações práticas em diversas áreas.

REFERÊNCIAS

AHMED, G. F.; ABU-RUB, F. A.; KHATTAB, S. M. Application of the aho-corasick algorithm to create a network intrusion detection system. **IEEE Access**, v. 9, p. 23650–23660, 2021. Accessed: 2024-08-30. Disponível em: https://ieeexplore.ieee.org/abstract/document/9351435.

AHO, A.; CORASICK, M. Efficient string matching: An aid to bibliographic search. **Commun. ACM**, v. 18, p. 333–340, 06 1975.

ARUDCHUTHA, S.; NISHANTHY, T.; RAGEL, R. G. String matching with multicore cpus: Performing better with the aho-corasick algorithm. In: **2013 IEEE 8th International Conference on Industrial and Information Systems**. [S.l.: s.n.], 2013. p. 231–236.

BOYER, R. S.; MOORE, J. S. A fast string searching algorithm. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 20, n. 10, p. 762–772, oct 1977. ISSN 0001-0782. Disponível em: https://doi.org/10.1145/359842.359859.

CHARRAS, C.; LECROQ, T. **Handbook of Exact String Matching Algorithms**. [S.l.]: King's College Publications, 2004. ISBN 0954300645.

COLUSSI, L. Correctness and efficiency of pattern matching algorithms. **Information and Computation**, v. 95, n. 2, p. 225–251, 1991. ISSN 0890-5401. Disponível em: https://www.sciencedirect.com/science/article/pii/0890540191900465.

GANCHEVA, V.; STOEV, H. Optimization and performance analysis of cat method for dna sequence similarity searching and alignment. **Genes**, v. 15, n. 3, p. 341, 2024. Accessed: 2024-08-30. Disponível em: https://www.mdpi.com/2073-4425/15/3/341.

GOOGLE. **String Search Algorithm in Google Chrome**. 2024. https://source.chromium.org/chromium/chromium/src/+/main:v8/src/strings/string-search.h. Accessed: 2024-08-30.

GUSFIELD, D. **Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology**. [S.l.]: Cambridge University Press, 1997.

HAKAK, S. I. et al. Exact string matching algorithms: Survey, issues, and future research directions. **IEEE Access**, IEEE, v. 7, p. 69613–69639, apr 2019.

HORSPOOL, R. N. Practical fast searching in strings. **Software: Practice and Experience**, v. 10, 1980. Disponível em: https://api.semanticscholar.org/CorpusID:6618295.

JEONG, Y. et al. High performance parallelization of boyer-moore algorithm on many-core accelerators. In: **2014 International Conference on Cloud and Autonomic Computing**. [S.l.: s.n.], 2014. p. 265–272.

KARP, R. M.; RABIN, M. O. Efficient randomized pattern-matching algorithms. **IBM J. Res. Dev.**, v. 31, p. 249–260, 1987. Disponível em: https://api.semanticscholar.org/CorpusID:5734450.

KNUTH, D. E.; MORRIS, J. H.; PRATT, V. R. Fast pattern matching in strings. **SIAM J. Comput.**, v. 6, p. 323–350, 1977. Disponível em: https://api.semanticscholar.org/CorpusID:11697579.

RAITA, T. Turning the boyer-moore-horspool string searching algorithm. **Softw. Pract. Exper.**, John Wiley & Sons, Inc., USA, v. 22, n. 10, p. 879–884, oct 1992. ISSN 0038-0644. Disponível em: https://doi.org/10.1002/spe.4380221006.

SHAKESPEARE, W. **The Complete Works of William Shakespeare**. Project Gutenberg, 2024. Acesso em: 29 outubro 2024. Disponível em: https://www.gutenberg.org/cache/epub/100/pg100.txt.

THAMBAWITA, V.; RAGEL, R.; ELKADUWE, D. An optimized parallel failure-less aho-corasick algorithm for dna sequence matching. In: . [S.l.: s.n.], 2016. p. 1–6.

XIAN-FENG, H.; YU-BAO, Y.; LU, X. Hybrid pattern-matching algorithm based on bm-kmp algorithm. In: **2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)**. [S.l.: s.n.], 2010. v. 5, p. V5–310–V5–313.