Análise dos Algoritmos de Rabin-Karp e Knuth-Morris-Pratt

Cássio J. Dhein, Lorenzo W. Martins, Rafael S. Rosa

[1]Escola Politécnica — PUCRS
Av. Ipiranga, 6681 – Partenon
Porto Alegre, RS – 90619-900
{c.jones,l.windmoller,rafael.schaker}@edu.pucrs.br

Abstract. *This work presents a detailed analysis of two efficient pattern matching algorithms: Rabin-Karp and Knuth-Morris-Pratt (KMP). The Rabin-Karp algorithm uses a hash function known as "rolling hash"or "recursive hash"to locate patterns within a text. We explore the proposed hash function in Rabin-Karp, explaining its functionality and analyzing its complexity. Furthermore, we implement the Rabin-Karp algorithm using this hash function and conduct experimental tests to compare its performance. Additionally, we discuss the Knuth-Morris-Pratt algorithm, which employs a pattern preprocessing technique to achieve efficiency. We describe the standard preprocessing of KMP and explain how it is utilized in the algorithm. We provide an illustrative example of KMP and analyze its complexity. Finally, we implement the KMP algorithm and perform experimental tests, comparing the results with those of Rabin-Karp.*

Resumo. *Este trabalho apresenta uma análise detalhada de dois algoritmos eficientes de correspondência de padrões: Rabin-Karp e Knuth-Morris-Pratt (KMP). O algoritmo de Rabin-Karp utiliza uma função de hash conhecida como "rolling hash"ou "recursive hash"para localizar padrões dentro de um texto. Exploramos a função de hash proposta por Rabin-Karp, explicando seu funcionamento e analisando sua complexidade. Em seguida, implementamos o algoritmo de Rabin-Karp utilizando essa função e realizamos testes experimentais para comparar seu desempenho. Além disso, abordamos o algoritmo de Knuth-Morris-Pratt, que utiliza um processo de pré-processamento do padrão sendo procurado para alcançar eficiência. Descrevemos o pré-processamento padrão do KMP, explicando como ele é utilizado no algoritmo. Apresentamos um exemplo ilustrativo do KMP e analisamos sua complexidade. Por fim, implementamos o algoritmo KMP e conduzimos testes experimentais, comparando seus resultados com os do Rabin-Karp.*

1. Introdução

A correspondência de padrões[1] é um problema comum na área da ciência da computação, que consiste em encontrar um padrão específico dentro de um texto. Essa tarefa é fundamental em diversas aplicações, como processamento de texto, busca em bancos de dados e análise de dados.

Neste trabalho, iremos explorar dois algoritmos eficientes de correspondência de padrões: Rabin-Karp e Knuth-Morris-Pratt (KMP). Esses algoritmos fornecem soluções únicas para o problema, cada um com suas características distintas.

[1]O problema de encontrar um padrão específico dentro de um texto.

O algoritmo de Rabin-Karp utiliza uma função de hash especial chamada ”rolling hash”[2] para localizar padrões no texto.

Por sua vez, o algoritmo KMP emprega um processo de pré-processamento do padrão a ser procurado[3].

O objetivo deste trabalho é realizar uma análise detalhada desses algoritmos, explorando suas funcionalidades, complexidades e desempenho. Faremos implementações experimentais dos algoritmos e realizaremos testes comparativos para avaliar seu desempenho em diferentes cenários.

Na seção seguinte, abordaremos o algoritmo de Rabin-Karp, explicando sua função de hash proposta e fornecendo um exemplo ilustrativo. Em seguida, faremos uma análise de sua complexidade. Na sequência, implementaremos o algoritmo de Rabin-Karp e conduziremos testes para comparar seus resultados com os obtidos em aula.

Posteriormente, nos concentraremos no algoritmo KMP, explicando seu processo de pré-processamento do padrão e fornecendo um exemplo detalhado. Realizaremos também uma análise de complexidade para entender o desempenho do algoritmo em diferentes situações. Em seguida, implementaremos o algoritmo KMP e faremos testes comparativos com os resultados obtidos no algoritmo de Rabin-Karp.

Ao final, apresentaremos as conclusões do trabalho, destacando os principais resultados encontrados e as lições aprendidas durante o processo de pesquisa e implementação.

2. Algoritmo de Rabin-Karp

O algoritmo de Rabin-Karp é uma técnica de busca de strings (ou padrões) em um texto, projetada para ser eficiente ao lidar com múltiplos padrões simultaneamente. A essência do algoritmo é a utilização de valores hash para comparações eficientes de substrings. No entanto, em vez de calcular o hash para cada substring do zero, o algoritmo de Rabin-Karp emprega uma abordagem inteligente conhecida como ”rolling hash”ou ”recursive hash”. Esta técnica permite calcular o hash da próxima substring com base no hash da substring atual, economizando uma quantidade considerável de tempo. Esta seção irá explorar a fundo o conceito de rolling hash, oferecerá um exemplo de sua implementação e análise de complexidade, e apresentará um código de implementação do algoritmo Rabin-Karp, junto com os resultados dos testes.

2.1. Função de Hash no Algoritmo de Rabin-Karp

A função de hash utilizada no algoritmo de Rabin-Karp é conhecida como ”rolling hash”ou ”recursive hash”. Este tipo de função de hash tem a propriedade de que, dado o hash de uma janela de texto, é possível calcular o hash da próxima janela deslocada em uma posição com uma quantidade constante de operações, independentemente do tamanho da janela.

A implementação comum do rolling hash envolve o uso de uma base (geralmente um número primo grande) e um módulo (também um número primo grande). O hash para

[2]Função de hash que permite calcular o hash de uma janela deslizante do texto com base no hash anterior.

[3]Processo que cria uma tabela de informações que indica os deslocamentos a serem feitos em caso de não correspondência entre o padrão e o texto.

uma janela de texto é calculado somando-se o valor ASCII de cada caractere, multiplicado pela base elevada à potência correspondente à sua posição na janela, e então tomando o módulo do resultado. Para calcular o hash da próxima janela, subtraímos o valor do caractere que está saindo da janela, multiplicamos o resultado pela base e adicionamos o valor do novo caractere que entra na janela. Tudo isso é feito modulo o número primo escolhido.

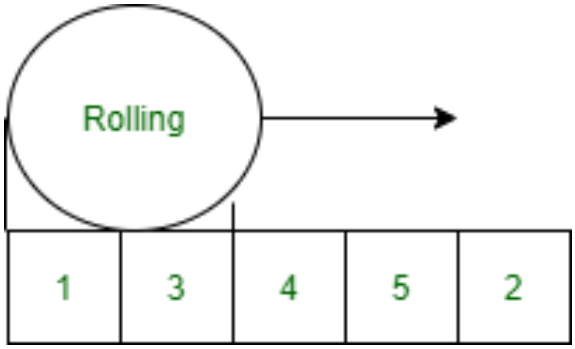


Figura 1. Ilustração do algoritmo Rabin-Karp utilizando hash rotativo.

O uso da função de hash rolling no algoritmo de Rabin-Karp permite que ele execute em tempo linear em relação ao tamanho do texto e do padrão, em vez de tempo quadrático como seria com uma função de hash simples[4].

Algorithm 1 Pseudocódigo para o algoritmo de Rabin-Karp

```
1:  procedure RABINKARP(text, pattern)
2:      n ← LENGTH(text)
3:      m ← LENGTH(pattern)
4:      patternHash ← HASH(pattern)
5:      textHash ← HASH(text[1..m])
6:      for i ← 1 to n − m + 1 do
7:          if textHash = patternHash and text[i..i + m − 1] = pattern then
8:              PRINT("Pattern found at index "i − 1)
9:          end if
10:         if i < n − m + 1 then
11:             textHash ← REHASH(text, i, i + m, textHash)
12:         end if
13:     end for
14: end procedure
```

Este pseudocódigo, portanto, representa a implementação do algoritmo de Rabin-Karp para correspondência de padrões em texto. Ele usa um valor de hash para comparar rapidamente o padrão com substrings do texto, e atualiza esse valor de hash de uma maneira eficiente à medida que desloca a janela ao longo do texto. Segue o passo-a-passo do seuj funcionamento:

[4]Isso é verdadeiro para o caso médio, onde as colisões de hash são raras. No pior caso, quando há muitas colisões de hash, o algoritmo de Rabin-Karp ainda pode degradar para tempo quadrático. No entanto, na prática, isso raramente é um problema se escolhermos uma boa função de hash.

1. **Procedimento RabinKarp** (*text, pattern*): Esta linha define o início do procedimento ou função chamada *RabinKarp* que recebe dois parâmetros: *text* e *pattern*. *text* é a string onde queremos procurar *pattern*.
2. **Calcula o comprimento do texto:** $n \leftarrow$ LENGTH(*text*). Esta linha calcula o comprimento do texto e armazena o resultado na variável *n*. A função *length* está sendo chamada aqui.
3. **Calcula o comprimento do padrão:** $m \leftarrow$ LENGTH(*pattern*). De maneira semelhante, esta linha calcula o comprimento do padrão e armazena o resultado na variável *m*.
4. **Calcula o hash do padrão:** $patternHash \leftarrow$ HASH(*pattern*). Aqui, a função *hash* é chamada com *pattern* como argumento. O resultado, que é o valor de hash do padrão, é armazenado na variável *patternHash*.
5. **Calcula o hash da primeira substring do texto:** $textHash \leftarrow$ HASH(*text[1..m]*). Esta linha calcula o valor de hash da primeira substring do texto que tem o mesmo comprimento que o padrão. O valor de hash resultante é armazenado na variável *textHash*.
6. **Loop principal:** O bloco **Para** $i \leftarrow 1$ **até** $n - m + 1$ contém o loop principal do algoritmo. Ele percorre o texto deslocando uma janela de comprimento *m* da esquerda para a direita.
   (a) **Verifica se a substring atual é igual ao padrão: Se** $textHash = patternHash$ **e** *text[i..i+m-1] = pattern*, o algoritmo verifica se o valor de hash da substring atual é igual ao valor de hash do padrão e se a substring atual é igual ao padrão. Se ambas as condições forem verdadeiras, imprime a posição inicial da substring atual.
   (b) **Atualiza o hash da próxima substring: Se** $i < n - m + 1$, o algoritmo verifica se o fim do texto ainda não foi atingido. Se ainda houver texto restante a ser examinado, o algoritmo atualiza o valor de hash da substring atual usando a função *rehash*.
7. **Fim do procedimento**: **Fim do Procedimento**.

2.2. Exemplo e Análise de Complexidade

A partir dos exemplos de teste fornecidos em aula, foi implementado um algoritmo. Nesse algoritmo, é definido um padrão de busca (comparação) e uma string de entrada (entrada). O algoritmo busca a ocorrência do padrão dentro da string de entrada.

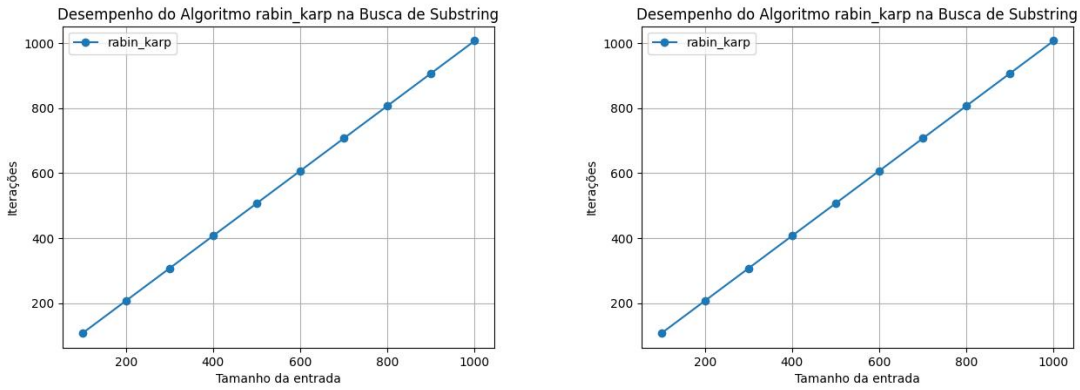
O algoritmo utiliza uma técnica de hash para comparar o padrão com substrings da entrada de forma eficiente. Ele calcula o valor do hash tanto para o padrão quanto para as substrings da entrada. Se os valores dos hashes forem iguais, ele realiza uma verificação mais detalhada para garantir que não seja uma colisão de hash. Se for uma correspondência perfeita, ou seja, se todas as letras do padrão coincidirem com as letras da substring, o algoritmo retorna a posição onde a ocorrência foi encontrada.

Caso não seja uma correspondência perfeita, o algoritmo desliza a janela de comparação para a direita e recalcula o valor do hash para a nova substring. Esse deslizamento é realizado até percorrer todas as possíveis posições da entrada.

O número de iterações é contado para monitorar o desempenho do algoritmo e avaliar a eficiência em diferentes cenários.

Se nenhuma ocorrência do padrão for encontrada, o algoritmo retorna -1 para indicar que não houve correspondência.

Para a aferição da complexidade foram usados os exemplos solicitados na descrição do trabalho (exemplos usados em aula), mais uma simulação para Strings maiores que 500.000, usando essas Strings como Inputs e contando o número de iterações foi possível a geração de gráficos que provam a complexidade linear.



(a) Complexidade para String alvo "ADF" **(b) Complexidade para Strings $>$ 500.000**

Figura 2. Complexidades Rabin-Karp

Pelos gráficos é possível analisar a complexidade linear $O(n)$, ou seja, significando que o algoritmo teria uma complexidade linear em relação ao tamanho da string de entrada, independentemente do tamanho do padrão. Isso implica que o algoritmo tem um desempenho muito eficiente, especialmente para strings de entrada grandes.

No entanto, é importante observar que a complexidade do algoritmo Rabin-Karp não é linear no pior caso, mas sim no caso médio. Em certos casos, como quando há muitas colisões de hash, a complexidade pode se tornar quadrática, resultando em $O(n * m)$ onde n é o tamanho da string de entrada e m é o tamanho do padrão.

2.3. Implementação e Testes

```
    def rabin_karp(self, entrada, comparacao):
        d = 256
        q = 13
        m = comparacao.tam
        n = entrada.tam
        h = pow(d, m - 1) % q
        p = 0
        t = 0
        self.iteracoes = 0

        if m > n:
            return -1

        for i in range(m):
            p = (d * p + ord(comparacao.valor[i])) % q
```

```
            t = (d * t + ord(entrada.valor[i])) % q
            self.iteracoes += 1

        for s in range(n - m + 1):
            if p == t:
                match = True
                for i in range(m):
                    self.iteracoes += 1
                    if entrada.valor[s + i] != comparacao.
                       valor[i]:
                        match = False
                        break
                if match:
                    return s
            if s < n - m:
                t = (t - h * ord(entrada.valor[s])) % q
                t = (t * d + ord(entrada.valor[s + m])) % q
                t = (t + q) % q
            self.iteracoes += 1

        return -1

if __name__ == "__main__":
    entradaA = String('ABCDCBDCBDACBDABDCBADF')
    entradaB = String('A' * 500000 + 'B' * 500000)
    comparacao = String('ADF')
```

3. Algoritmo de Knuth-Morris-Pratt (KMP)

O algoritmo de Knuth-Morris-Pratt (KMP) é um algoritmo clássico para o problema de busca de string, ou seja, encontrar todas as ocorrências de uma string ”padrão”P dentro de uma string maior T. O KMP realiza isso de forma eficiente, garantindo que a string maior T seja analisada apenas uma vez, sem retrocessos.

3.1. Pre-processamento no Algoritmo de KMP

O algoritmo *KMP*[5] usa um conceito chamado ”pré-processamento”da string padrão *P*. Esta fase de pré-processamento cria uma tabela auxiliar, comumente chamada de *LPS*[6], que é usada para decidir o próximo passo do algoritmo quando uma comparação falha. Funciona da seguinte forma:

1. **Criação da Tabela LPS:** Inicialmente, cria-se um array LPS de tamanho igual ao comprimento da string padrão *P*[7]. O *LPS[i]*[8] guarda o comprimento do maior prefixo próprio da substring *P[0..i]* que também é sufixo. Note que a definição de prefixo próprio exclui a string completa.

[5]O algoritmo de Knuth-Morris-Pratt (KMP) é um algoritmo eficiente para busca de padrões em uma string. Ele utiliza a tabela LPS para evitar retrocessos durante a busca.

[6]A tabela LPS (Longest Proper Prefix which is also suffix) é uma tabela auxiliar utilizada pelo algoritmo KMP. Ela armazena o comprimento do maior prefixo próprio de uma substring que também é um sufixo.

[7]Prefixo próprio é qualquer substring obtida removendo o último caractere da string original.

[8]O *LPS[i]* guarda o comprimento do maior prefixo próprio da substring *P[0..i]* que também é sufixo.

2. **Preenchimento da Tabela LPS:** O primeiro valor da tabela *LPS*, *LPS[0]*, é sempre 0, pois uma string de um único caractere não tem prefixo ou sufixo próprio[9]. Então, começa-se a preencher o restante da tabela. Para preencher *LPS[i]*, considera-se o comprimento do prefixo próprio que é também sufixo da substring *P[0..i-1]*. Este comprimento é representado por *len*[10].

Se *P[i] == P[len]*, então encontramos um novo prefixo próprio que é sufixo para *P[0..i]* e aumentamos *len* por um. Se *P[i] != P[len]*, então devemos diminuir *len* até encontrarmos um valor que satisfaça a igualdade ou até que *len* seja 0. Se *len* se torna 0, então não existe prefixo próprio que é sufixo e *LPS[i]* é 0.

O pré-processamento permite que o algoritmo *KMP* evite o retrocesso na string *T*[11]. Quando uma comparação falha, o algoritmo usa a tabela *LPS* para decidir o número de caracteres em *P* que podem ser ignorados antes de começar a próxima comparação. Isso é feito movendo a janela de correspondência na string *T* para a direita sem retroceder na string *T*. Dessa forma, a tabela *LPS* serve como um guia para a janela de correspondência, permitindo que o algoritmo *KMP* opere de maneira muito eficiente.

Algorithm 2 Pseudocódigo para o pré-processamento no algoritmo de KMP

```
1:  procedure COMPUTELPSARRAY(pattern, m, LPS)
2:      length ← 0
3:      LPS[0] ← 0
4:      i ← 1
5:      while i < m do
6:          if pattern[i] = pattern[length] then
7:              length ← length + 1
8:              LPS[i] ← length
9:              i ← i + 1
10:         else
11:             if length ≠ 0 then
12:                 length ← LPS[length − 1]
13:             else
14:                 LPS[i] ← 0
15:                 i ← i + 1
16:             end if
17:         end if
18:     end while
19: end procedure
```

O propósito deste pré-processamento é calcular um vetor chamado LPS (*Longest Proper Prefix which is also suffix*), que será utilizado durante a busca da string alvo. Segue a explicação detalhada:

[9]Comparação de caracteres ocorre durante a busca do padrão, onde são comparados caracteres da string padrão (*P*) e da string de busca (*T*) para identificar ocorrências do padrão na string de busca.

[10]A *len* representa o comprimento do prefixo próprio que é também sufixo da substring *P[0..i-1]*.

[11]Retrocesso ocorre quando, durante a busca, é necessário voltar para trás na string de busca para continuar a comparação com o padrão. O algoritmo KMP evita retrocessos usando a tabela LPS.

1. A função `ComputeLPSArray` é chamada com três argumentos: `pattern` (a string alvo que estamos buscando), `m` (o comprimento da string alvo) e `LPS` (o vetor que estamos calculando).
2. Inicializamos `length` (que rastreia o comprimento do prefixo/sufixo mais longo encontrado até agora) como 0 e `LPS[0]` também como 0, porque não há prefixo/sufixo mais longo para uma string de comprimento 1.
3. Inicializamos `i` como 1 e então entramos em um loop que continua enquanto `i` for menor que `m` (o comprimento da string alvo).
4. Dentro do loop, verificamos se o caractere na posição `i` de `pattern` é igual ao caractere na posição `length`. Se for, incrementamos `length`, atribuímos esse valor a `LPS[i]` e incrementamos `i`.
5. Se o caractere na posição `i` de `pattern` não é igual ao caractere na posição `length` e `length` é diferente de 0, fazemos `length` ser igual a `LPS[length-1]`. Isso efetivamente "retrocede"na string alvo para procurar um prefixo/sufixo mais curto que seja compatível com o caractere atual.
6. Se o caractere na posição `i` de `pattern` não é igual ao caractere na posição `length` e `length` é 0 (ou seja, não encontramos um prefixo/sufixo compatível), definimos `LPS[i]` como 0 e incrementamos `i`.
7. Ao final deste processo, o vetor `LPS` contém, para cada índice `i`, o comprimento do prefixo próprio mais longo que também é sufixo para a sub-string de `pattern` de comprimento `i + 1`. Isso será usado na fase de busca do algoritmo KMP para evitar o *backtracking* desnecessário na string de entrada.

3.2. Exemplo e Análise de Complexidade

O Algoritmo Knuth-Morris-Pratt (KMP) é usado para encontrar a primeira ocorrência de uma "palavra"(ou string) $comparacao$ em um "texto"(ou string) $entrada$.

1. As variáveis M e N são inicializadas para os tamanhos das strings de $comparacao$ e $entrada$, respectivamente.
2. A lista lps (Longest Proper Prefix which is also suffix) é inicializada com todos os elementos como 0. Esta lista será usada para armazenar o comprimento do prefixo próprio mais longo que também é sufixo para cada sub-string de $comparacao$.
3. As variáveis i e j são inicializadas como 1 e 0, respectivamente, e serão usadas para preencher a lista lps.
4. O primeiro loop while é usado para preencher a lista lps:
    - Se o caractere na posição i de $comparacao$ for igual ao caractere na posição j, o valor j é incrementado e atribuído ao elemento i de lps, e então i é incrementado.
    - Se eles não forem iguais e j não for 0, j é atualizado para o valor de lps na posição $j - 1$.
    - Se eles não forem iguais e j for 0, o elemento i de lps é definido como 0 e i é incrementado.
5. As variáveis i e j são redefinidas para 0 para a próxima fase do algoritmo.
6. O segundo loop while é usado para realizar a busca da string $comparacao$ na string $entrada$:
    - Se o caractere na posição j de $comparacao$ for igual ao caractere na posição i de $entrada$, i e j são incrementados.

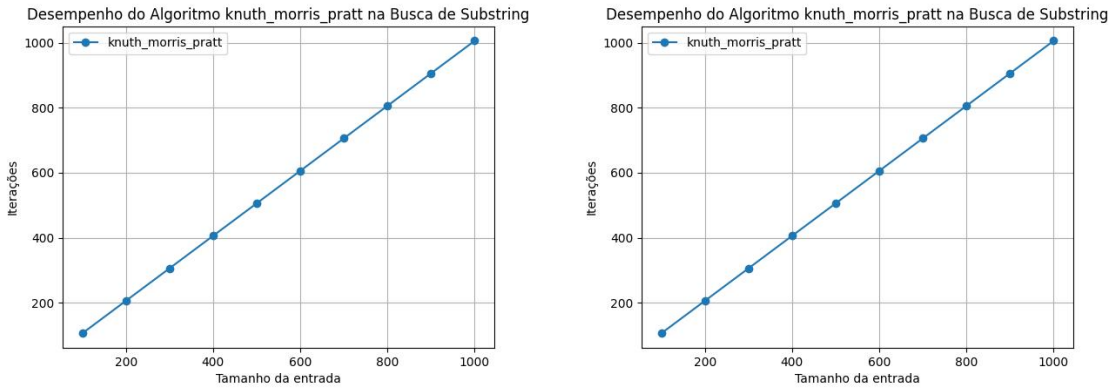
- Se j for igual a M (o que significa que a string $comparacao$ foi encontrada na string $entrada$), j é atualizado para o valor de lps na posição $j - 1$ e a função retorna $i - j$ (a posição inicial da string $comparacao$ na string $entrada$).
- Se i for menor que N e o caractere na posição j de $comparacao$ for diferente do caractere na posição i de $entrada$, se j não for 0, j é atualizado para o valor de lps na posição $j - 1$. Caso contrário, i é incrementado.

7. Se a string $comparacao$ não for encontrada na string $entrada$, a função retorna -1.

O algoritmo KMP é eficiente porque evita o backtracking na string $entrada$, o que torna o tempo de execução linear em relaçãoao tamanho da $entrada$ e da $comparacao$.

O número de iterações é contabilizado para monitorar o desempenho do algoritmo e avaliar sua eficiência em diferentes cenários. Se nenhuma ocorrência do padrão for encontrada, o algoritmo retorna -1 para indicar que não houve correspondência.

Para aferição da complexidade, foram utilizados os exemplos solicitados na descrição do trabalho (exemplos usados em aula), além de uma simulação para strings maiores que 500.000. Usando essas strings como inputs e contando o número de iterações, foi possível a geração de gráficos que comprovam a complexidade linear do algoritmo.



(a) Complexidade para String alvo "ADF" **(b) Complexidade para Strings $>$ 500.000**

Figura 3. Complexidades Knuth Morris Pratt

O algoritmo Knuth-Morris-Pratt (KMP) possui complexidade de tempo $O(n)$, onde n é o tamanho da string de entrada. A razão para isso é que o algoritmo processa cada caractere da string de entrada exatamente uma vez. É importante notar que a complexidade $O(n)$ do KMP se refere ao tempo de busca após a tabela de falha ter sido construída. A construção da tabela de falha em si também tem uma complexidade de tempo de $O(m)$, onde m é o tamanho do padrão. No entanto, uma vez que a tabela de falha é construída, pode ser reutilizada para várias buscas na mesma string de entrada, o que torna o KMP muito eficiente para tarefas de busca de padrões repetidos.

3.3. Implementação e Testes

```
1         def knuth_morris_pratt(self, entrada, comparacao):
2         self.iteracoes = 0
```

```
        M = comparacao.tam
        N = entrada.tam
        lps = [0] * M
        j = 0
        i = 1

        while i < M:
            self.iteracoes += 1
            if comparacao.valor[i] == comparacao.valor[j]:
                j += 1
                lps[i] = j
                i += 1
            elif j != 0:
                j = lps[j - 1]
            else:
                lps[i] = 0
                i += 1

        i = 0
        j = 0

        while i < N:
            self.iteracoes += 1
            if comparacao.valor[j] == entrada.valor[i]:
                i += 1
                j += 1

            if j == M:
                j = lps[j - 1]
                return i - j

            elif i < N and comparacao.valor[j] != entrada.
               valor[i]:
                if j != 0:
                    j = lps[j - 1]
                else:
                    i += 1

        return -1
if __name__ == "__main__":
    entradaA = String('ABCDCBDCBDACBDABDCBADF')
    entradaB = String('A' * 500000 + 'B' * 500000)
    comparacao = String('ADF')
```

4. Conclusão

Ambos os algoritmos Rabin-Karp e Knuth-Morris-Pratt (KMP) apresentaram uma complexidade temporal de O(n) nos testes realizados, tornando-os eficientes para a busca de padrões em strings. No entanto, cada um desses algoritmos tem características distintas

que podem torná-los mais adequados para certos tipos de aplicações.

O algoritmo Rabin-Karp, que é baseado em hash, é particularmente eficaz quando você precisa encontrar todas as ocorrências de muitos padrões na mesma string de entrada. Isso ocorre porque a complexidade temporal permanece linear mesmo quando o número de padrões aumenta. No entanto, o desempenho do Rabin-Karp pode ser afetado negativamente por colisões de hash, onde duas strings diferentes produzem o mesmo valor de hash.

Por outro lado, o algoritmo KMP é eficiente quando você precisa encontrar várias ocorrências do mesmo padrão em uma string de entrada. Ele alcança essa eficiência construindo uma tabela de falhas (ou LPS) que permite que o algoritmo pule segmentos da string de entrada que já sabe que não irão corresponder ao padrão.

Em suma, embora ambos os algoritmos tenham complexidade temporal linear em relação ao tamanho da string de entrada, a escolha entre Rabin-Karp e KMP deve ser guiada pelo tipo específico de problema de busca de padrão que você está tentando resolver. Além disso, fatores como a natureza dos padrões e da string de entrada, bem como as características de desempenho do ambiente de execução, também devem ser levados em consideração ao escolher o algoritmo mais adequado.

5. Apêndice

Logo abaixo está a implementação completa de ambos os algoritmos e támbem a geração dos gráficos:

```
import matplotlib.pyplot as plt


class String:
    def __init__(self, valor):
        self.valor = valor
        self.tam = len(valor)


class StringSearchAlgorithms:
    def __init__(self):
        self.iteracoes = 0

    def rabin_karp(self, entrada, comparacao):
        d = 256
        q = 13
        m = comparacao.tam
        n = entrada.tam
        h = pow(d, m - 1) % q
        p = 0
        t = 0
        self.iteracoes = 0

        if m > n:
            return -1
```

```
26
27          for i in range(m):
28              p = (d * p + ord(comparacao.valor[i])) % q
29              t = (d * t + ord(entrada.valor[i])) % q
30              self.iteracoes += 1
31
32          for s in range(n - m + 1):
33              if p == t:
34                  match = True
35                  for i in range(m):
36                      self.iteracoes += 1
37                      if entrada.valor[s + i] != comparacao.
                            valor[i]:
38                          match = False
39                          break
40                  if match:
41                      return s
42              if s < n - m:
43                  t = (t - h * ord(entrada.valor[s])) % q
44                  t = (t * d + ord(entrada.valor[s + m])) % q
45                  t = (t + q) % q
46              self.iteracoes += 1
47
48          return -1
49
50      def knuth_morris_pratt(self, entrada, comparacao):
51          self.iteracoes = 0
52          M = comparacao.tam
53          N = entrada.tam
54          lps = [0] * M
55          j = 0
56          i = 1
57
58          while i < M:
59              self.iteracoes += 1
60              if comparacao.valor[i] == comparacao.valor[j]:
61                  j += 1
62                  lps[i] = j
63                  i += 1
64              elif j != 0:
65                  j = lps[j - 1]
66              else:
67                  lps[i] = 0
68                  i += 1
69
70          i = 0
71          j = 0
72
73          while i < N:
```

```
74              self.iteracoes += 1
75              if comparacao.valor[j] == entrada.valor[i]:
76                  i += 1
77                  j += 1
78
79              if j == M:
80                  j = lps[j - 1]
81                  return i - j
82
83              elif i < N and comparacao.valor[j] != entrada.
                    valor[i]:
84                  if j != 0:
85                      j = lps[j - 1]
86                  else:
87                      i += 1
88
89          return -1
90
91
92  class GraphGenerator:
93      def __init__(self, algorithm, directory):
94          self.algorithm = algorithm
95          self.directory = directory
96
97      def generate_graph(self, entrada, comparacao):
98          sizes = list(range(100, 1001, 100))
99          counts = []
100
101         for size in sizes:
102             entrada.valor = 'a' * size + 'test'
103             entrada.tam = len(entrada.valor)
104
105             algorithm_instance = StringSearchAlgorithms()
106             algorithm_method = getattr(algorithm_instance,
                    self.algorithm)
107
108             algorithm_method(entrada, comparacao)
109             counts.append(algorithm_instance.iteracoes)
110
111         plt.plot(sizes, counts, marker='o', label=self.
                algorithm)
112         plt.xlabel('Tamanho da entrada')
113         plt.ylabel('Iterations')
114         plt.title('Desempenho do Algoritmo {} na Busca de
                Substring'.format(self.algorithm))
115         plt.grid(True)
116         plt.legend()
117         plt.savefig(self.directory)
118         plt.show()
```

```
119
120
121 if __name__ == "__main__":
122     entradaA = String('ABCDCBDCBDACBDABDCBADF')
123     entradaB = String('A' * 500000 + 'B' * 500000)
124     comparacao = String('ADF')
125
126     algorithms = StringSearchAlgorithms()
127
128     graph_generator_rabin_karp1 = GraphGenerator('rabin_karp
        ', 'Rabin_Karp_ADF.jpg')
129     graph_generator_rabin_karp1.generate_graph(entradaA,
        comparacao)
130
131     graph_generator_rabin_karp2 = GraphGenerator('rabin_karp
        ', 'Rabin_Karp_500000.jpg')
132     graph_generator_rabin_karp2.generate_graph(entradaB,
        comparacao)
133
134     graph_generator_kmp1 = GraphGenerator('
        knuth_morris_pratt', 'Knuth_ADF.jpg')
135     graph_generator_kmp1.generate_graph(entradaA, comparacao
        )
136
137     graph_generator_kmp2 = GraphGenerator('
        knuth_morris_pratt', 'Knuth_500000.jpg')
138     graph_generator_kmp2.generate_graph(entradaB, comparacao
        )
```

Referências

Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2022). *Introduction to Algorithms, fourth edition*. MIT Press.

Kleinberg, J. and Tardos, E. (2013). *Algorithm Design: Pearson New International Edition*. Pearson Education.