Universidade Federal de São João del-Rei

Ciência da computação

Projeto e Análise de Algoritmos III

Documentação Trabalho Prático 3

2024

Vitor Ferreira França

1-Introdução

Na ciência da computação, o problema do casamento de padrões emerge como um dos desafios mais intrigantes e fundamentais. Imagine-se como um jovem aprendiz de magia, embarcando em uma jornada épica para desvendar os segredos de uma substring perdida, escondida nas profundezas de palavras encantadas. Este conto místico serve como metáfora para a complexidade e a sutileza envolvidas na busca eficiente de padrões dentro de cadeias de caracteres. Nesta documentação, exploraremos as diversas técnicas de casamento de padrões, analisando suas eficiências. A missão é clara: encontrar uma substring dentro de uma string maior, considerando múltiplos intervalos de busca e utilizando algoritmos aprendidos em sala de aula. Através de uma análise detalhada de complexidade e desempenho, avaliaremos as escolhas realizadas.Este documento oferece uma visão técnica e objetiva sobre os métodos utilizados na busca pela substring, fundamentando-se em conceitos sólidos e estratégias bem definidas.

2- Construção e funcionamento para resolução do problema.

2.1- Chegada dos dados pelo teminal

A execução do algoritmo é guiada pelos argumentos fornecidos na linha de comando, onde o primeiro argumento especifica o algoritmo de busca de padrão a ser utilizado, como kmp ou forca (“Força Bruta”). O segundo argumento define o arquivo de entrada, de onde são lidas as strings e as consultas de intervalo para verificação da substring. A função getline() é utilizada para capturar a string principal e o padrão, enquanto as consultas são armazenadas em um array dinâmico. A saída, que indica se a substring está presente em cada intervalo consultado, é escrita em um arquivo denominado saida.txt.

Exemplo: “./tp3 kmp teste.txt” ou “./tp3 forca teste.txt”

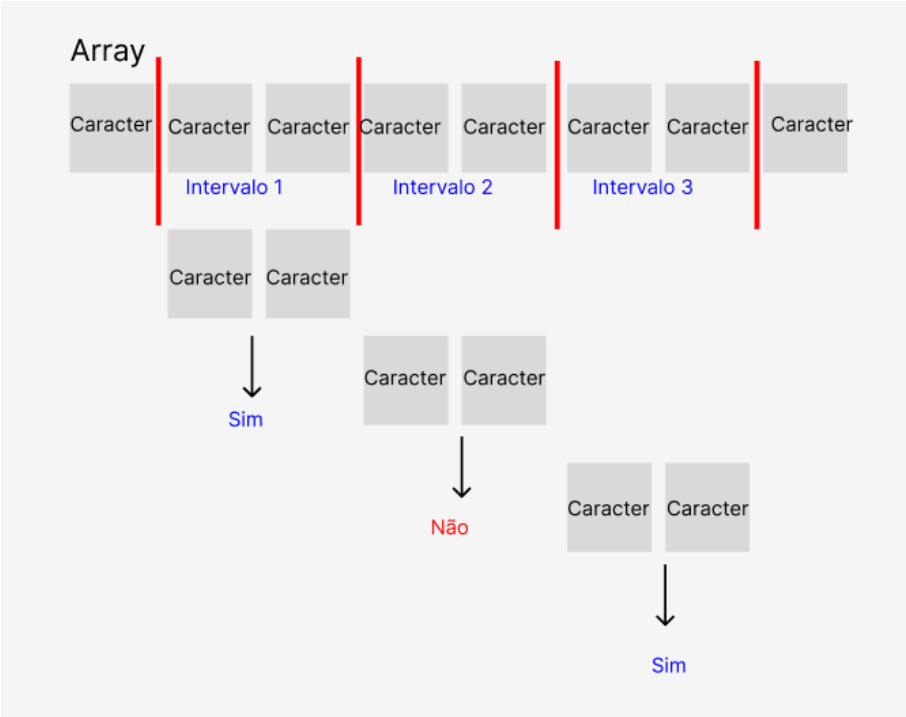
2.2- Variaveis de entrada

inicialmente, o programa segue uma sequência bem definida. Primeiramente, o programa recebe o arquivo de entrada, que contém a string principal, a substring a ser buscada, e os intervalos de consulta. Em seguida, é acionada uma função para calcular o tempo de execução do programa, utilizando a biblioteca sys/time.h e as funções getrusage e gettimeofday. Posteriormente, são lidos os valores das variáveis do arquivo de entrada. A primeira linha do arquivo contém a string principal a ser pesquisada. A segunda linha contém a substring que deve ser buscada. A terceira linha contém um inteiro k, que representa o número de consultas. Em seguida, o programa processa k linhas que descrevem os intervalos de consulta, onde cada linha consiste em dois

inteiros a e b, representando, respectivamente, o início e o fim do intervalo na string principal. O algoritmo de busca é então aplicado para verificar a presença da substring dentro dos intervalos especificados. Dependendo do algoritmo escolhido (KMP ou força bruta), o programa realiza a busca e imprime "sim" ou "nao" para cada consulta. A execução é finalizada com a gravação do tempo total de execução.

2.3- A procura do padrão

É preciso encontrar uma substring específica dentro de uma string maior, mas apenas em certos intervalos dessa string. Para cada intervalo fornecido, é preciso verifica se a substring aparece nele e responde com "sim" ou "não".Para resolver o problema, o primeiro passo é considerar cada intervalo fornecido e extrair a parte da string principal que corresponde a esse intervalo específico. Essa extração permite que você isole o trecho da string onde a busca será realizada.Em seguida, dentro dessa parte extraída, você precisa verificar se a substring que está sendo procurada está presente. Esse processo envolve comparar a substring com possíveis posições na parte isolada da string para determinar se ela aparece exatamente como é.Finalmente, se a substring for encontrada dentro desse trecho, você deve retornar "sim" como resposta. Caso contrário, se a substring não estiver presente na parte extraída, a resposta deve ser "não". Este processo é repetido para cada intervalo fornecido, garantindo que todas as partes relevantes da string principal sejam verificadas.



3-Algoritmos usados.

3.1- Algoritmo KMP (Knuth-Morris-Pratt)

3.1.2-O algoritmo KMP opera em duas fases principais:

Pré-processamento do Padrão:

O KMP inicia com a construção de uma tabela de prefixos, também conhecida como tabela de falhas ou "lps" (longest prefix suffix). Essa tabela é crucial para otimizar o processo de busca. A tabela de prefixos armazena o comprimento do maior prefixo do padrão que também é um sufixo. Com isso, o algoritmo determina até onde deve retroceder na busca após uma falha, evitando a reexaminação dos caracteres já comparados.

Busca no Texto:

Após o pré-processamento, o KMP utiliza a tabela de prefixos para percorrer o texto e localizar o padrão. Quando ocorre uma discrepância (ou falha), o algoritmo emprega a tabela de prefixos para determinar o próximo ponto de comparação no padrão, evitando reiniciar a busca do início. Em vez de recomeçar a comparação a partir do início do padrão a cada falha, o algoritmo ajusta o padrão com base nas informações da tabela de prefixos, reduzindo o número total de comparações realizadas.

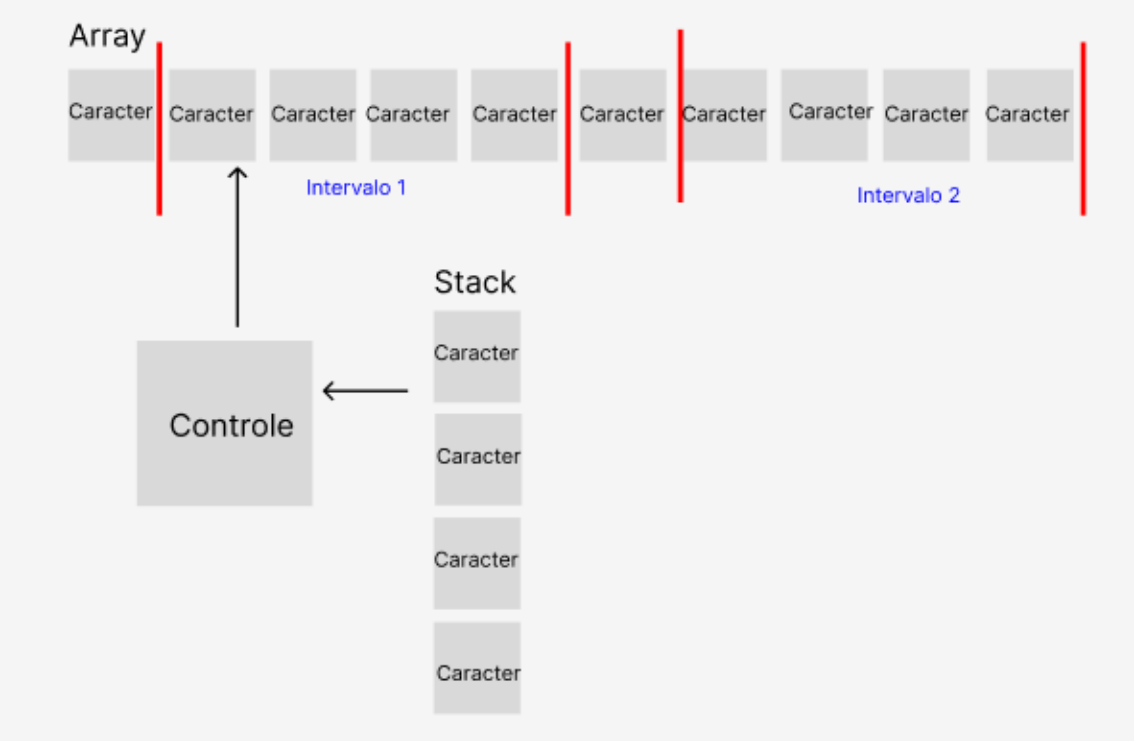
Pseudocodigo:

```
Função calcular_array_lps(padrão, tamanho_padrão, array_lps):
  comprimento_prefixo <- 0
  array_lps[0] <- 0
  índice <- 1

  ENQUANTO índice < tamanho_padrão FAÇA:
    SE padrão[índice] == padrão[comprimento_prefixo] ENTÃO:
      comprimento_prefixo <- comprimento_prefixo + 1
      array_lps[índice] <- comprimento_prefixo
```

```
        índice <- índice + 1
      SENÃO:
        SE comprimento_prefixo != 0 ENTÃO:
          comprimento_prefixo <- array_lps[comprimento_prefixo - 1]
        SENÃO:
          array_lps[índice] <- 0
          índice <- índice + 1

Função busca_kmp(texto, padrão, início, fim):
  tamanho_texto <- fim - início + 1
  tamanho_padrão <- comprimento(padrão)
  array_lps <- criar_array(tamanho_padrão)
  índice_texto <- início
  índice_padrão <- 0

  Calcular_array_lps(padrão, tamanho_padrão, array_lps)

  ENQUANTO índice_texto <= fim FAÇA:
    SE padrão[índice_padrão] == texto[índice_texto] ENTÃO:
      índice_padrão <- índice_padrão + 1
      índice_texto <- índice_texto + 1

    SE índice_padrão == tamanho_padrão ENTÃO:
      Liberar(array_lps)
      RETORNAR 1 // Padrão encontrado

    SENÃO SE índice_texto <= fim E padrão[índice_padrão] != texto[índice_texto]
ENTÃO:
```

```
        SE índice_padrão != 0 ENTÃO:
            índice_padrão <- array_lps[índice_padrão - 1]
        SENÃO:
            índice_texto <- índice_texto + 1

    Liberar(array_lps)
    RETORNAR 0 // Padrão não encontrado
```

3.1.3- Representação



3.1.4-Analise de complexidade

Para examinar a complexidade do algoritmo KMP, é fundamental considerar as duas fases principais do processo: a construção da tabela de prefixos e a busca no texto.

Construção da Tabela de Prefixos:

O algoritmo KMP inicia com a construção da tabela de prefixos (ou array LPS). O tempo necessário para preencher essa tabela é proporcional ao comprimento do padrão. Em cada passo, a tabela é atualizada com base nas comparações entre caracteres do padrão e os valores já calculados. Como a construção da tabela de prefixos exige apenas uma passagem linear pelo padrão, sua complexidade é O(m), onde m é o comprimento do padrão.

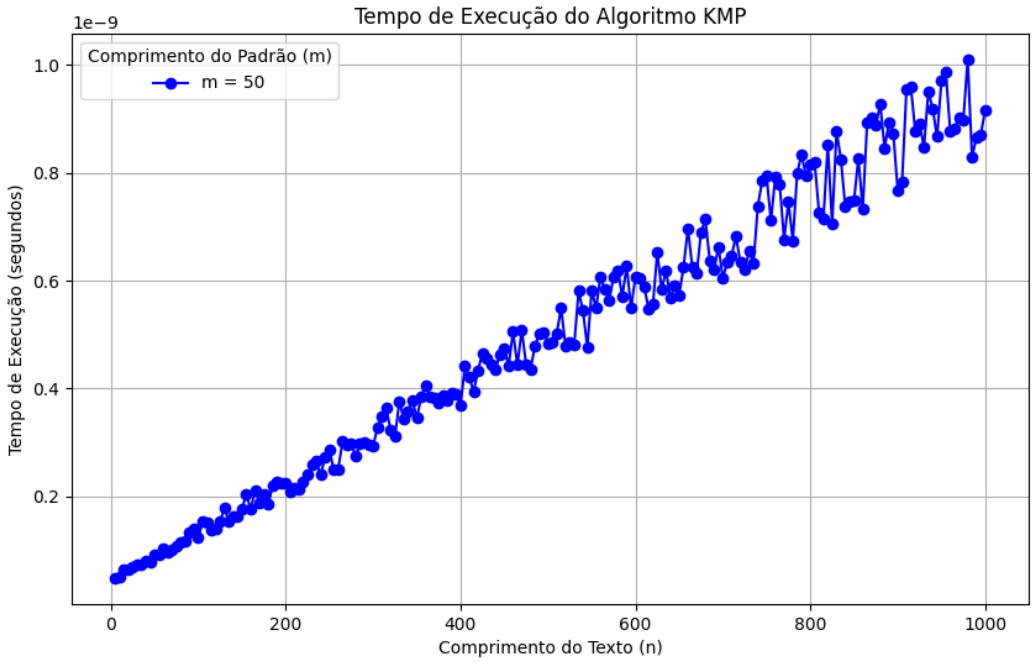
Busca no Texto:

Após a construção da tabela de prefixos, o algoritmo KMP realiza a busca do padrão no texto. A busca é eficiente, pois utiliza a tabela de prefixos para pular comparações desnecessárias. O texto é percorrido uma única vez, e as comparações do padrão são feitas de forma otimizada com base na tabela de prefixos. Portanto, o tempo necessário para a busca no texto é proporcional ao comprimento do texto e ao comprimento do padrão. A complexidade dessa fase é O(n), onde n é o comprimento do texto.

Combinando ambas as fases, a complexidade total do algoritmo KMP é O(m + n). Isso reflete o tempo gasto na construção da tabela de prefixos e na busca do padrão no texto.

Portanto, a complexidade total do algoritmo KMP é O(m + n), tornando-o muito eficiente.

3.1.5-Analise de tempos



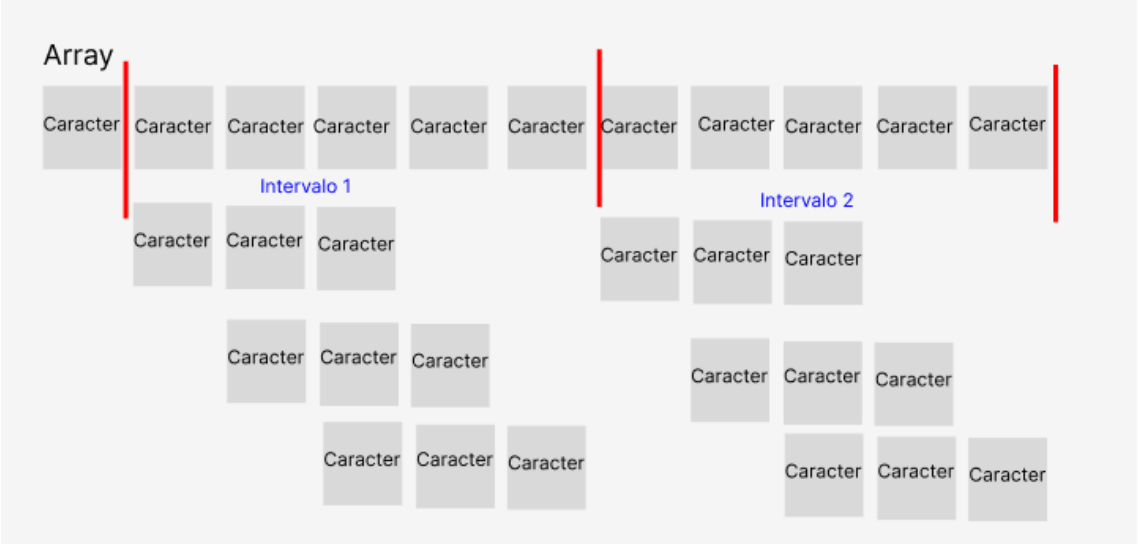
3.2- Força Bruta

3.2.2-O algoritmo de força bruta, é um método simples para entendimento para encontrar uma substring (padrão) dentro de uma string (texto). Ele opera em uma única fase, realizando uma comparação direta entre o padrão e substrings do texto. Percorrendo toda string.

Pseudocodigo:

```
Função busca_força_bruta(texto, padrão):
  tamanho_texto <- comprimento(texto)
  tamanho_padrão <- comprimento(padrão)

  PARA índice_texto de 0 até (tamanho_texto - tamanho_padrão) FAÇA:
    corresponder <- Verdadeiro

    PARA índice_padrão de 0 até (tamanho_padrão - 1) FAÇA:
      SE texto[indice_texto + índice_padrão] != padrão[índice_padrão] ENTÃO:
        corresponder <- Falso
        QUEBRAR // Interrompe a comparação atual

    SE corresponder == Verdadeiro ENTÃO:
      RETORNAR índice_texto // Padrão encontrado na posição índice_texto

  RETORNAR -1 // Padrão não encontrado
```

3.2.3-Representação



3.2.4-Analise de complexidade

O algoritmo de força bruta funciona comparando diretamente o padrão com todas as possíveis substrings do texto.

Iteração sobre o Texto:

O algoritmo percorre cada posição inicial possível no texto onde o padrão poderia começar. Se o texto tem comprimento n e o padrão tem comprimento m, existem n - m + 1 posições iniciais possíveis no texto.

Portanto, o loop externo que percorre o texto tem uma complexidade de O(n - m + 1), que é aproximadamente O(n) no pior caso.

Comparação de Substrings:

Para cada posição inicial no texto, o algoritmo compara o padrão com a substring correspondente caractere a caractere.No pior caso, em que todos os caracteres coincidem até o último, o número de comparações para cada posição inicial é m.Portanto, o loop interno que faz a comparação do padrão tem uma complexidade de O(m).

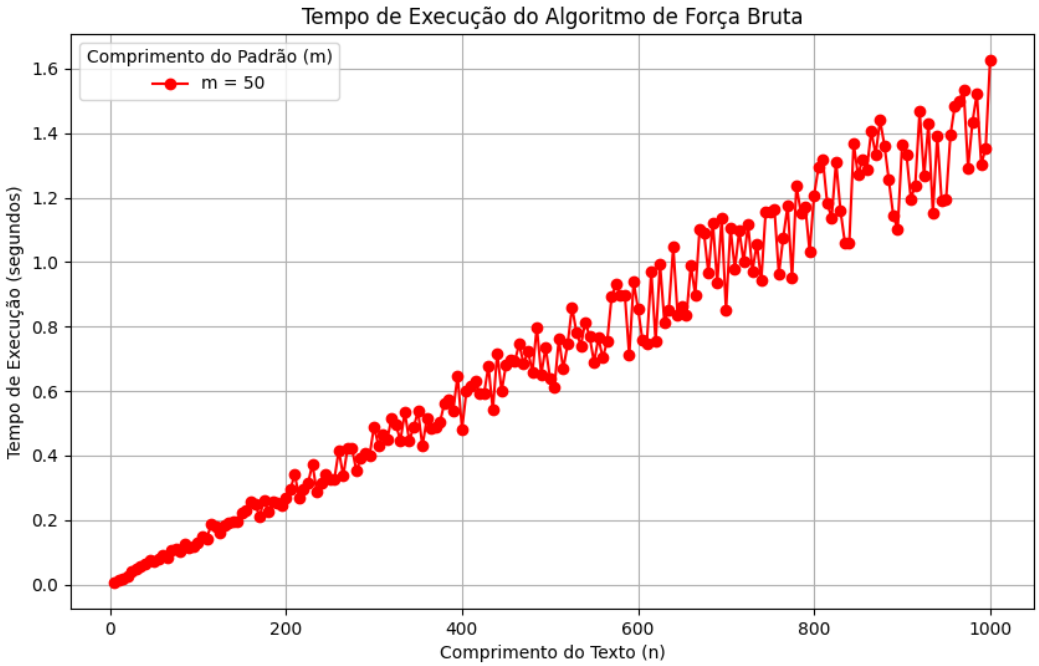
Complexidade Total:

Como o algoritmo realiza a comparação para cada posição inicial no texto, a complexidade total do algoritmo de força bruta é o produto das complexidades dos loops externo e interno.Isso resulta em uma complexidade de tempo de O((n - m + 1) * m), que simplifica para O(n * m) no pior caso.

Melhor caso: O melhor caso ocorre quando a primeira letra do padrão não coincide com a substring do texto repetidamente. Neste caso, o algoritmo detecta rapidamente que o padrão não corresponde, resultando em um tempo de execução menor, aproximadamente O(n) se houver uma comparação inicial rápida.

Pior caso: O pior caso ocorre quando o padrão e a substring correspondente do texto têm um longo prefixo comum. Por exemplo, ao procurar o padrão "AAA" dentro do texto "AAAAA", cada comparação falha apenas no final, exigindo um número máximo de comparações para cada posição inicial. Neste caso, o tempo de execução é O(n * m).

3.1.5-Análise de tempos



4- Conclusão

Durante o desenvolvimento deste trabalho, foram exploradas várias abordagens e interpretações para resolver o mesmo problema, evidenciando a diversidade de soluções e aspectos envolvidos. Inicialmente, a abordagem de força bruta foi considerada, na qual todas as possíveis correspondências entre o texto e o padrão são testadas. No entanto, devido à sua ineficiência em termos de tempo e ao alto consumo de recursos computacionais, essa abordagem foi substituída por métodos mais eficientes.Entre as soluções mais eficazes, destacam-se os algoritmos de Knuth-Morris-Pratt (KMP) e Boyer-Moore, entre outros. Para a segunda fase do desenvolvimento, optou-se pela implementação do algoritmo de Knuth-Morris-Pratt, devido à sua simplicidade e clareza. O KMP demonstrou ser significativamente superior à abordagem de força bruta em termos de tempo de execução, uma vez que utiliza um sistema de autômatos para otimizar a busca e reduzir o poder de processamento necessário. Essa escolha resultou em uma solução mais eficiente e prática para o problema proposto. A

busca por um algoritmo que equilibre facilidade de implementação e revisão com a eficiência na resolução de problemas deve ser uma prioridade para qualquer desenvolvedor qualificado. Essa abordagem não só impacta o custo computacional e o tempo de resposta, mas também contribui significativamente para a qualidade do produto final. Ao adotar práticas que otimizem tanto o desempenho quanto a simplicidade, um desenvolvedor não apenas melhora a solução oferecida ao mercado, mas também avança o campo da ciência da computação como um todo.

4- Referencias

- **Monteiro, Bruno.** *Slides sobre o Algoritmo Knuth-Morris-Pratt (KMP)*. Disponível em: https://homepages.dcc.ufmg.br/~monteirobruno/Slides_KMP.pdf. Acesso em: [10/08/2024].
- **Menotti, D.** *Processamento de Cadeias de Caracteres*. Disponível em: http://www.decom.ufop.br/menotti/paa101/slides/aula-ProcCadCarac.pdf. Acesso em: [09/08/2024].
- **Leonardo Chaves Dutra da Rocha.** Slide Processamento de Cadeias de Caracteres**.** Acesso em: [08/08/2024].