Universidade Federal
de São João del-Rei

PROJETO E ANÁLISE DE ALGORITMOS

Trabalho Prático III - A Busca Mágica pela Substring Perdida

Diego Augusto Lucas Costa

Agosto de 2024

1 Introdução

Em um reino encantado, uma antiga profecia revelou a existência de uma poderosa substring, oculta nas profundezas das palavras místicas, capaz de conceder imenso poder a quem a possuísse. Temendo as consequências de tal poder em mãos erradas, os sábios do reino convocaram um jovem aprendiz de magia para a missão de encontrar a substring perdida. Com o auxílio de um pergaminho mágico, que contém uma string misteriosa e a substring a ser buscada, ele deve explorar palavras encantadas em diferentes intervalos e desvendar os segredos ocultos que podem mudar o destino do reino.

O desafio de encontrar uma substring específica em meio a uma sequência de caracteres é central para muitos problemas de busca em computação, desde a análise de grandes volumes de texto até a identificação de padrões genéticos. Assim como no reino encantado onde se procura uma poderosa sequência perdida, no mundo real, algoritmos de casamento exato de caracteres são fundamentais para localizar informações críticas em dados extensos, garantindo que padrões específicos sejam encontrados com precisão e eficiência.

O objetivo deste trabalho é auxiliar o jovem aprendiz a encontrar a substring perdida dentro de uma série de palavras encantadas. Para isso, são exploradas duas abordagens distintas: uma implementação por força bruta e outra utilizando o algoritmo Boyer-Moore-Horspool (BMH). Ambos os métodos são estudados e comparados em termos de eficiência e complexidade, considerando o problema proposto em diferentes contextos. A análise dos resultados visa identificar qual estratégia oferece melhor desempenho na solução desse desafio em cenários variados.

2 Desenvolvimento do Projeto

Para o desenvolvimento do projeto, foram implementadas duas estratégias distintas para encontrar o casamento de caracteres. Ambas as soluções solucionam o problema proposto, mas obtêm resultados mais eficientes em diferentes cenários, onde cada uma pode se adequar melhor. Uma das soluções foi implementada utilizando o algoritmo de força bruta, cuja escolha se baseou na facilidade de implementação, já que é uma solução mais trivial. No entanto, diante da necessidade de um algoritmo mais otimizado que evitasse comparações desnecessárias, foi escolhido o Boyer-Moore-Horspool como uma alternativa mais eficiente.

2.1 Queries

Uma das requisições do problema é a implementação de várias pesquisas utilizando o mesmo texto e padrão, sendo essas pesquisas denominadas `queries`. Para implementar as queries, optou-se por utilizar um mecanismo simples e que evita cópias do texto original, afinal, precisamos lidar com textos de tamanhos na casa das centenas de megabytes.

A ideia principal é substituir o caractere que se encontra no índice `b` pelo caractere `'\0'` para indicar o fim da string. Depois, cria-se um novo ponteiro a partir do ponteiro para o texto de busca, cujo valor é $a - 1$ caracteres para frente. Assim, obtém-se uma substring do texto original, no intervalo `[a,b]`.

Após a realização da busca, é necessário desfazer a modificação no texto. Por isso, é necessário guardar o valor que se encontrava na posição `b` do texto antes da substituição. E então, basta desfazer a troca utilizando o valor anterior. Como criamos uma cópia do ponteiro original, não é necessário subtrair o valor $a - 1$ do texto.

2.2 Solução Força Bruta

Algorithm 1 Algoritmo de Força Bruta para Casamento de Caracteres

```
function BruteStrStr(Texto, Padrao)
    n ← length(Texto)
    m ← length(Padrao)
    for i ← 0 to n − m do
        k ← i
        j ← 0
        while j < m and Texto[k] = Padrao[j] do
            j ← j + 1
            k ← k + 1
        end while
        if j = m then
            return true
        end if
    end for
    return false
end function
```

O algoritmo de força bruta (também chamado de Naive) para casamento de caracteres, implementado na função `brute_strstr`, foi baseado no pseudocódigo 1. No pseudocódigo apresentado, o texto `Texto` de comprimento `n` e o padrão `Padrao` de comprimento `m` são comparados para encontrar uma correspondência exata. As variáveis auxiliares `i`, `j` e `k` são usadas para iterar através do texto e do padrão. O laço `for` percorre o texto `Texto`, onde `i` varia de `0` até `n - m`, garantindo que o padrão seja comparado em todas as posições possíveis dentro do texto. A variável `k` é utilizada para avançar através do texto durante cada tentativa de correspondência.

Dentro do laço `while`, realiza-se a comparação entre os caracteres do texto e do padrão. Se os caracteres correspondentes forem iguais, `j` e `k` são incrementados para verificar os próximos caracteres. A condição final do `while` verifica se `j` alcançou o valor `m`, indicando que todo o padrão foi comparado com sucesso. Se essa condição for satisfeita, a função retorna `true`, indicando que o padrão foi encontrado. Caso contrário, o controle retorna ao laço `for`, onde `i` é incrementado para tentar a correspondência

a partir da próxima posição no texto. O processo continua até que o padrão seja encontrado ou todas as posições possíveis no texto tenham sido verificadas.

2.3 Solução com Boyer-Moore-Horspool

Algorithm 2 Boyer-Moore-Horspool para Casamento de Padrões

```
procedure BMH_Match(Texto, Padrao)
    Tabela ← Preprocessamento(Padrao)
    Deslocamento ← 0
    Tamanho_Texto ← Length(Texto)
    Tamanho_Padrao ← Length(Padrao)
    while Tamanho_Texto - Deslocamento ≥ Tamanho_Padrao do
        if Comparar(Texto[Deslocamento], Padrao, Tamanho_Padrao) then
            return Deslocamento
        end if
        Deslocamento ← Deslocamento + Tabela[Texto[Deslocamento + Tamanho_Padrao -1]]
    end while
    return -1
end procedure
function Preprocessamento(Padrao)
    Tamanho_Padrao ← Length(Padrao)
    Tabela[0..255] ← Tamanho_Padrao
    for i ← 0 to Tamanho_Padrao - 2 do
        Tabela[Padrao[i]] ← Tamanho_Padrao - 1 - i
    end for
    return Tabela
end function
function Comparar(SubTexto, Padrao, Tamanho_Padrao)
    for i ← Tamanho_Padrao - 1 downto 0 do
        if SubTexto[i] ≠ Padrao[i] then
            return Falso
        end if
    end for
    return Verdadeiro
end function
```

O pseudocódigo 2 foi desenvolvido com base na implementação do algoritmo Boyer-Moore-Horspool segue uma estrutura clara e eficiente para encontrar uma substring dentro de um texto maior. O processo é dividido em três etapas principais: a criação da tabela de deslocamento, a comparação do padrão com o texto e o avanço pelo texto usando as informações obtidas.

Primeiramente, o algoritmo inicia criando uma tabela de deslocamento, onde cada entrada da tabela indica quantos caracteres podem ser pulados se uma determinada letra do texto não corresponder ao final do padrão. Esse pré-processamento é

fundamental para o funcionamento eficiente do algoritmo, pois permite que grandes porções do texto sejam ignoradas quando uma correspondência parcial é encontrada, mas falha no final do padrão.

Em seguida, o algoritmo percorre o texto, utilizando uma variável para indicar a posição atual. A cada iteração, o padrão é comparado com a subsequência do texto correspondente, começando pelo final do padrão. Essa abordagem é diferente de outros algoritmos de busca que comparam do início ao fim, e é o que permite ao Boyer-Moore-Horspool otimizar o processo de busca. Se o padrão for encontrado, o algoritmo retorna a posição inicial onde a correspondência ocorreu. Caso contrário, utiliza-se a tabela de deslocamento para determinar quantos caracteres podem ser pulados, avançando diretamente para a próxima posição relevante no texto.

Por fim, a função de comparação verifica se todos os caracteres do padrão correspondem à subsequência atual do texto. Se todos coincidirem, uma correspondência é confirmada; caso contrário, o algoritmo continua a busca a partir da posição determinada pela tabela de deslocamento. Esse método é especialmente eficiente em textos grandes e padrões mais longos, onde o ganho em desempenho é significativo em relação a métodos mais simples como o de força bruta.

3 Análises de complexidades

3.1 Força Bruta

A complexidade do algoritmo de força bruta para casamento de caracteres, implementado na função `brute_strstr`, é $O(n * m)$, onde `n` é o comprimento da string `haystack` e `m` é o comprimento da substring `needle`. A complexidade do caso esperado do algoritmo de força bruta é influenciada pela probabilidade de coincidência do padrão em diferentes partes do texto. Isso demonstra que o desempenho do algoritmo de força bruta pode variar significativamente com base nas características do texto e do padrão, mas em muitos cenários comuns, ele ainda pode ser consideravelmente eficiente. No pior caso, o algoritmo compara cada caractere de `needle` com todas as possíveis posições de início em `haystack`, resultando em $(n - m + 1)$ iterações do loop externo. Em cada iteração, pode ser necessário comparar até `m` caracteres, levando à complexidade total de $O(n * m)$. Esse comportamento é especialmente evidente em casos onde `needle` e `haystack` possuem muitos caracteres em comum, o que força comparações repetitivas. Embora esse método seja direto, ele se torna ineficiente para textos longos ou padrões extensos. Para resolver essa limitação, o algoritmo Boyer-Moore-Horspool é uma alternativa mais eficiente, pois ele utiliza informações obtidas durante o processo de comparação para pular partes da string `haystack`, resultando em um desempenho significativamente melhor em muitos casos práticos.

3.2 Boyer-Moore-Horspool

No pior caso, a complexidade do algoritmo Boyer-Moore-Horspool é $O(n * m)$, onde `n` é o comprimento do texto `haystack` e `m` é o comprimento do padrão `needle`. Esse caso extremo ocorre quando o texto e o padrão possuem muitos caracteres em comum, mas o padrão nunca é completamente encontrado até a última posição do texto. O algoritmo, nesse caso, pode ser forçado a verificar quase todas as posições do texto, resultando em um comportamento semelhante ao do algoritmo de força bruta. Um exemplo clássico do pior caso ocorre quando o texto consiste em repetições de um único caractere e o padrão é composto por esse mesmo caractere, seguido por um caractere diferente no final. Nessa situação, o algoritmo realiza muitos deslocamentos pequenos, resultando em um grande número de comparações.

No caso esperado, a complexidade média do algoritmo é $O(n/m)$, desde que `c` (o tamanho do alfabeto) não seja pequeno e `m` (o comprimento do padrão) não seja muito grande, o que é significativamente mais eficiente do que a abordagem de força bruta. Isso ocorre porque o algoritmo geralmente consegue pular várias posições no texto, utilizando a tabela de deslocamento de maneira eficaz. Esse comportamento é típico quando o padrão é relativamente longo e as ocorrências do último caractere do padrão são raras no texto. Em tal cenário, o algoritmo pode pular grandes porções do texto, evitando comparações desnecessárias, o que melhora significativamente o desempenho. Exemplos desse caso incluem textos onde os caracteres são distribuídos de forma aleatória e o padrão é distinto o suficiente para permitir grandes saltos durante a busca.

4 Testes e resultados

Os tempos de usuário e sistema são duas diferentes formas de medir o tempo de execução. O tempo de sistema mede quanto tempo o sistema operacional executou tarefas para o programa, como ler arquivos, alocar memória, etc. Enquanto o tempo de usuário mede a quantidade de tempo que o programa utilizou a CPU. Utilizamos as bibliotecas `getrusage.h` e `gettimeofday.h` para avaliar o tempo de execução completo do nosso algoritmo, pois oferecem uma forma de consultar os valores dos tempos de usuário e de sistema e do tempo de relógio. Isso representa uma vantagem significativa em comparação com a biblioteca `gettimeofday.h`, que só registra o tempo físico decorrido. Contudo, a soma dos valores de usuário e sistema resulta aproximadamente no valor retornado pela biblioteca `gettimeofday.h`, sendo mais precisa que a outra para tempos pequenos, devido à imprecisões do tipo de dado `double`.

Os testes a seguir foram sistematicamente executados numa máquina Arch Linux com processador AMD Ryzen 7 5700U, sendo o código compilado utilizando o comando `make`. Além disso, cada entrada foi resolvida 10 vezes para obter uma média mais precisa do tempo de execução. As entradas foram geradas em dois contextos distintos: um simulando o pior caso e outro gerado de forma aleatória para simular os casos esperados, com o incremento da largura de `haystack` de cada teste sendo de 10K e com a `needle` fixa com n igual 5.

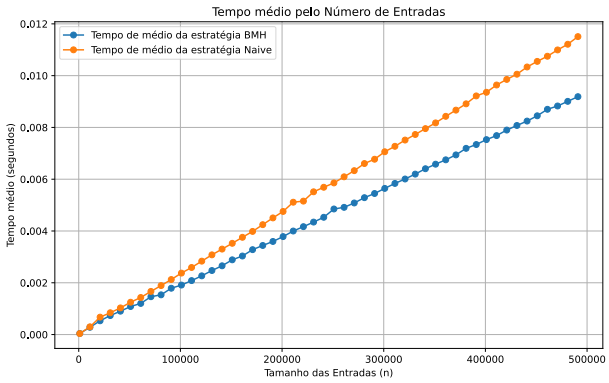

Figura 1: Tempos médios de execução simulando piores casos

Os testes realizados corroboram os resultados discutidos na seção de análise de complexidade, mostrando que, em cenários de pior caso, o comportamento do algoritmo Boyer-Moore-Horspool pode se assemelhar ao do algoritmo de força bruta. Na Figura 1 ilustra o desempenho de ambos os algoritmos, destacando que, em situações adversas, o Boyer-Moore-Horspool pode apresentar uma performance próxima à do método de força bruta.

Embora o Boyer-Moore-Horspool seja geralmente mais eficiente, no pior caso, quando o texto e o padrão têm muitos caracteres em comum e o padrão não é encontrado até as últimas posições do texto, ambos os algoritmos podem ter um desempenho semelhante. Esse comportamento é resultado da necessidade de comparar várias posições do texto, o que pode levar a uma performance comparável em casos específicos.

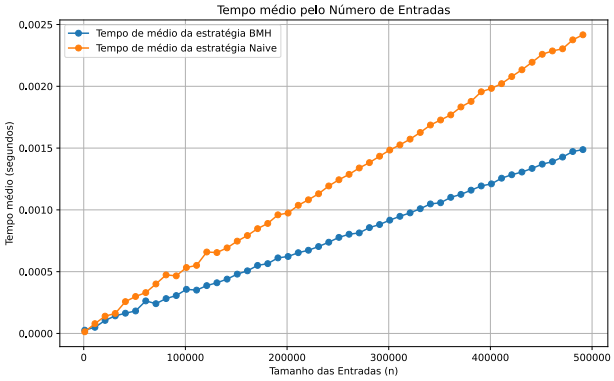

Figura 2: Tempos médios de execução simulando casos esperados

Os resultados dos casos esperados, como ilustrado na Figura 2, mostram que o algoritmo de força bruta, apesar de suas limitações, apresentou uma eficiência melhor em comparação ao seu desempenho no pior caso, conforme esperado. Embora tenha mostrado uma leve melhora, o algoritmo Boyer-Moore-Horspool (BMH) se destacou de maneira significativa.

A vantagem do BMH se deve aos saltos proporcionados pela tabela de deslocamento, que minimizam as comparações repetitivas típicas do algoritmo de força bruta. Esse comportamento permite que o BMH evite revisitar posições já comparadas, resultando em um desempenho superior, como evidenciado pelos resultados obtidos.

5 Conclusão

Em resumo, foram utilizadas duas soluções para o problema apresentado. Apesar de apresentarem implementação similar, o algoritmo Boyer-Moore-Horspool ainda apresenta performance notavelmente superior quando comparado ao algoritmo de Força Bruta. Isso se deve ao fato de que, ao realizar o pré-processamento, é possível obter as informações necessárias para descartar comparações sem sentido, por meio da tabela de deslocamento. Além disso, é possível observar que mesmo pequenas otimizações têm um efeito perceptível no tempo de execução do software, e, considerando que o problema de encontrar substrings dentro de texto é algo extremamente comum no nosso dia-a-dia, é essencial tornar esse processo o mais eficiente possível.

Apesar do algoritmo BMH apresentar resultados superiores na maioria dos casos testados, é importante lembrar que existem inúmeros outros algoritmos de casamento de caracteres. Em situações onde as limitações dos algoritmos estudados se tornam críticas, pode ser mais vantajoso explorar alternativas que ofereçam melhor desempenho para as características particulares do problema em questão.

Por fim, com as duas estratégias desenvolvidas, o jovem aprendiz de magia agora possui as ferramentas necessárias para desvendar os mistérios das palavras encantadas. Seja usando a abordagem simples e direta da força bruta, ou aproveitando a eficiência do Boyer-Moore-Horspool, ele está pronto para enfrentar os desafios que o aguardam em sua jornada. Assim, o destino do reino repousa sobre suas escolhas e a sabedoria em aplicar cada método na busca pela poderosa substring perdida.

Bibliografia

[1] Charles Ornelas Almeida, Fabiano Cupertino Botelho e Nivio Ziviani. *Processamento de Cadeias de Caracteres*. Acesso em 15-08-2024. URL: `http://www.decom.ufop.br/menotti/paa101/slides/aula-ProcCadCarac.pdf`.