Documentação - Trabalho Prático 3 de Algoritmos e Estruturas de Dados III

Arthur Vieira Silva e Felipe Augusto Moreira Chaves

20 de Junho de 2023

Sumário

1 Contextualização

Neste trabalho prático, o objetivo é determinar se há ou não casamento exato de um determinado padrão em um texto. Nesse sentido, temos que o padrão é representado por uma pedra mágica que contém uma sequência de símbolos coloridos e, de acordo com essa sequência, um poder distinto poderia surgir. Além disso, temos também um dicionário que relaciona cada conjunto de símbolos a um poder. Com isso, o objetivo é determinar se uma pedra possui poder ou não utilizando casamento exato de cadeias.

Dessa maneira, para cada caso de teste, temos duas cadeias de caracteres separadas por um espaço: a primeira é a sequência de habilidade (padrão) e a segunda a descrição da pedra (texto). Porém, temos uma condição fundamental de que a pedra é esférica, isto é, o último caractere é adjacente ao primeiro. A seguir, introduziremos os algoritmos utilizados, listaremos as suas rotinas, faremos uma análise de complexidade dessas rotinas, além de uma análise dos resultados obtidos e, para finalizar faremos uma breve conclusão a respeito do trabalho realizado.

2 Introdução

Após analisarmos os diferentes algoritmos para casamento de padrão, decidimos implementar três algorimos que nos ajudassem tanto na resolução do problema quanto na comparação para diferentes entradas. Sendo assim, a primeira estratégia adotada foi um **Algoritmo Força Bruta,** na segunda estratégia escolhemos o **Algoritmo Boyer-Moore-Horspool** e por fim, implementamos o **Algoritmo Shift-And.**

2.1 Algoritmo Força Bruta

Inicialmente, pensamos em solucionar o problema proposto por meio do algoritmo mais simples, isto é, utilizamos o **Algoritmo Força Bruta** para casamento de cadeias. Nessa implementação, a ideia principal é tentar realizar o casamento de qualquer subcadeia no texto de comprimento m com o padrão. Para ilustrarmos o funcionamento do algoritmo, considere o padrão $P = \{\text{arara}\}$ de comprimento $m = 5$ e o texto $T = \{\text{asararas}\}$ de comprimento $n = 8$, o algoritmo força bruta para esse exemplo funciona da seguinte

forma:

a*sararas*
a*rara* Casamento na primeira posição

as*araras*
ar*ara* Não houve casamento!

*a***s***araras*
a*rara* Não houve casamento!

*as***a***raras*
a*rara* Casamento na terceira posição

*as***ar***aras*
ar*ara* Casamento na quarta posição

*as***ara***ras*
ara*ra* Casamento na quinta posição

*as***arar***as*
arar*a* Casamento na sexta posição

*as***arara***s*
arara Casamento na sétima posição

Logo, ocorreu casamento exato do padrão na terceira posição do texto.

2.2 Boyer-Moore-Horspool

Na segunda estratégia, adotamos o **Algoritmo Boyer-Moore-Horspool**, um clássico algoritmo para casamento exato de cadeias. Neste caso, a ideia, ao contrário do **Força Bruta**, é pesquisar o padrão no sentido da direita para a esquerda, tornando o algoritmo muito mais rápido e eficiente. Desse modo, se houver uma desigualdade do sufixo do texto com o do padrão, calcula-se um deslocamento de todo o padrão para a direita do texto. Se essa desigualdade não ocorreu, temos um casamento exato.

Para isso, endereçamos uma tabela de deslocamento com o caractere no texto correspondente ao último caractere do padrão. No pré-processamento do padrão, o valor de todas as posições da tabela de deslocamento é m. Em

seguida, computamos o valor dos caracteres que estão presentes no padrão da seguinte maneira:

$d[x] = min\{j \text{ tal que } j = m \mid (1 \leq j < m \ \& \ P[m-j] = x)\}$

Por exemplo, considere $P = \{\text{teste}\}$ e $T = \{\text{ostestes}\}$. Logo, a tabela de deslocamento será:

$d[t] = 1$, $d[e] = 3$ e $d[s] = 2$ Os outros valores são $m = 5$.

12345678
*test***e** Colisão no caractere s : deslocamento de 2 posições.
*oste***s***tes*

12345678
  teste Casamento exato na terceira posição!
*os***teste***s*

Note que esse algoritmo é muito mais eficiente do que o **Força Bruta** justamente por não "deslocar"o padrão em apenas uma unidade quando há uma colisão.

2.3 Shift-And

Por fim, na terceira estratégia, adotamos o algoritmo **Shift-And**, que utiliza paralelismo de *bits* para verificar se há casamento de padrão. Inicialmente, contruímos uma tabela M que armazena uma máscara de *bits* $b_1, ..., b_m$ para cada caractere. Além disso, o algorimo mantém o conjunto de todos os prefixos do padrão que casam com o texto já lido e atualiza esse conjunto constantemente, utilizando paralelismo de *bits.* Tal conjunto é representado por R e a princípio, $R = 0^m$ (0 repetido m vezes). Após a leitura de cada caractere do texto, $R`$ é atualizado da seguinte forma:

$$R` = ((R >> 1) | 10^{m-1}) \ \& \ M[T[i]].$$

Considere o padrão $P = \{\text{arara}\}$ e o texto $T = \{\text{asararas}\}$. Abaixo temos a máscara de bits relativa ao padrão:

$M[a] = 10101$ e $M[r] = 01010$.

A tabela a seguir mostra o funcionamento do **Shift-And** para esse exemplo:

Texto	$(R >> 1)\|10^{m-1}$	R'
a	10000	**1**0000
s	11000	00000
a	10000	**1**0000
r	11000	0**1**000
a	10100	**10**100
r	11010	0**10**10
a	10101	**101**01
s	11010	00000

Tabela 1: Passo a passo do algoritmo Shift-And

Note que além de informar que ocorreu um casamento exato na terceira posição, também é possível verificar mais de um casamento por vez, utilizando, justamente, o paralelismo de *bits*.

3 Listagem e Análise de Complexidade das Rotinas

A seguir, listaremos todas as rotinas utilizadas no processo de contrução deste trabalho prático, além disso, faremos uma análise de complexidade de cada rotina individualmente, considerando o **número de comparações** realizadas como sendo as operações mais significativas.

- **TextoInvertido:** Esta função foi utilizada apenas para inverter o texto lido e, dessa maneira, realizar uma nova busca sobre o texto invertido. Com isso, nenhuma comparação é realizada nessa função.
- **ForcaBruta:** Função utilizada na primeira estratégia descrita anteriormente. Primeiramente, iremos verificar se o tamanho do texto é maior ou igual ao do padrão. Se for, temos um laço que será realizado até atingirmos o tamanho do texto ou ocorrer um casamento exato.

Dentro desse laço, há um outro *loop* que verifica se em cada posição do texto houve um casamento com o padrão. Se tiver um casamento, incrementamos ambas as posições, caso contrário, incrementamos apenas o índice do texto. Como o texto é circular, ainda há uma outra condição que se atingirmos a última posição do texto e tiver ocorrido um casamento até então, retornamos para a primeira posição do texto e incrementamos o índice do padrão. Por fim, se o índice que caminha sobre o padrão atingir o seu tamanho m, significa que ocorreu um casamento.

Se o tamanho do padrão for maior que o do texto, a única diferença será que o índice que caminha sobre o padrão irá somente até o tamanho do texto, tratamos o caso do texto ser circular da mesma forma com uma única exceção de que verificamos se o índice que caminha sobre o texto atingiu o tamanho máximo n.

A primeira comparação para verificar se o tamanho do texto ou do padrão é maior é realizada apenas uma vez, isto é, $O(1)$. Como o texto é circular, temos que o pior caso do algoritmo é $f(n) = (m \times n)$, como por exemplo em $P = \{\text{aaab}\}$ e $T = \{\text{aaaaaaaaab}\}$, isto é, $O(m \times n)$. O caso médio é $f(n) = \frac{c}{c-1}(1 - \frac{1}{c^m})(n - m + 1) + O(1)$ e o melhor caso é simplesmente se encontrarmos um casamento na primeira posição, ou seja, $O(m)$.

Se o tamanho do padrão for maior, a única alteração é que o melhor caso é $O(n)$.

- **PreProcessamento:** Utilizamos esta função para realizar o pré-processamento do padrão no algoritmo **Boyer-Moore-Horspool.** Ela receberá uma tabela d do mesmo tamanho do texto e, a princípio, todas as suas posições recebem o valor de m. Em seguida, temos um laço mais externo que vai iterar sobre cada caractere do texto e outro mais interno que verifica se um determinado caractere do texto está presente no padrão. Se sim, $d[i]$ receberá um outro valor como especificado na seção 2.2. Como fazemos essa verificação para todos os caracteres do texto, temos que a ordem de complexidade dessa função é $O(n \times m)$, no melhor caso, caso médio e pior caso.

- **BMH:** Este é procedimento é o algoritmo **Boyer-Moore-Horspool** citado previamente, nele alocamos uma tabela de deslocamento d de

tamanho n e chamamos a função PreProcessamento. Em seguida, verificamos se o tamanho do texto é maior que o do padrão, isto é, $O(1)$, se for, teremos um laço que será realizado até que um índice $i = m$ não seja igual à n. Dentro desse *loop,* temos dois índices $j = m$ e $k = i$ que caminham sobre o padrão e o texto, respectivamente. Com isso, temos outro laço mais interno que será realizado enquanto houver casamento e $j > 0$, quando o laço parar de ser executado temos que, se $j = 0$ então, houve casamento, caso contrário $i = i + d[k - 1]$. Logo, temos que o pior caso do BMH é $O(n \times m)$, o melhor caso é $O(\frac{n}{m})$ e o caso médio também é $O(\frac{n}{m})$, se o tamanho do alfabeto não for pequeno e m não for grande demais.

Como o texto é circular, se i for maior que n, dentro de outro *loop,* iremos verificar se a parte do padrão ainda alinhada com o texto casa. Se não ocorreu casamento, deslocamos o padrão em uma unidade e o processo é repetido. Se houver casamento, verificamos agora se o ínicio do texto casa com o restante do padrão.

Se o tamanho do padrão for maior que o do texto, a principal diferença é que os índices iniciam em n ($k = n$ e $j = i$), o laço mais externo será executado até m e tratamos o caso do texto ser circular se $i > m$. Logo, a única diferença é que o melhor e o pior caso são $O(\frac{m}{n})$, o caso esperado se o alfabeto não for pequeno e n não for grande demais.

- **Shift-And:** Este procedimento foi utilizado para implementar o algoritmo **Shift-And** para casamento exato de cadeias. Nele, inicialmente, criamos uma tabela M que armazena uma máscara de *bits* $b_1, .., b_m$ para cada caractere. Em seguida, todas as posições dessa tabela são inicializadas com o valor 0 e verificamos se o tamanho do texto é maior ou igual que o do padrão. Se for, realizamos um pré-processamento do padrão no qual criamos uma máscara de *bits* para cada um dos seus caracteres como exemplificado na seção 2.3. O valor de R também é inicializado como 0 e, assim, realizamos uma pesquisa sobre o texto na qual, para cada novo caractere lido, o valor do conjunto R' é atualizado como mostrado anteriormente e, a cada atualização de R', verificamos se houve ou não casamento exato.

  Se o tamanho do padrão for maior que o do texto, a diferença é que o pré-processamento irá até n e a pesquisa até m, ou seja, até o tamanho máximo do padrão. Dessa maneira, a complexidade desta função é

$O(m)$ ou $O(n)$, dependendo do tamanho do texto e do padrão.

- **SalvarOutput:** Procedimento utilizado apenas para escrever no arquivo de saída os dados armazenados em um *array* $resultados[t]$. O arquivo de saída terá o mesmo nome do de entrada com a extensão *out* e teremos um laço que será executado t vezes no qual, se o valor que estiver no *array* for -1, escrevemos N na saída, caso contrário, escrevemos S e o valor. Logo, a função é $O(t)$ para o melhor caso, pior caso e caso médio.

- **LerArquivo:** Para finalizar, este procedimento foi utilizado para realizar a leitura do arquivo de entrada fornecido. Primeiramente, faremos a leitura do número de casos de teste t, com isso, alocamos uma variável de tamanho t que armazenará o resultado em cada caso. Como não há como saber nem o tamanho do padrão nem o do texto, ambas as *strings* começam com tamanho 1. Dessa maneira, lemos cada caractere da entrada até o fim do arquivo, enquanto não atingirmos o espaço em branco, os caracteres lidos até o momento são do padrão e o seu tamanho é incrementado a todo momento. Quando encontrarmos o espaço em branco, leremos o texto. Após isso, verificamos qual foi a estratégia selecionada e chamamos as suas respectivas funções duas vezes, uma para o texto normal e outra para ele invertido. Em seguida, restauramos as *strings* e os seus tamanhos e repetimos o processo e, por fim, chamamos a função **SalvarOutput.** Como realizamos apenas a leitura dos dados e isso depende do número de casos de testes, temos que esta função é $O(t)$.

Portanto, temos que a complexidade da primeira estratégia é $O(m \times n)$, para o pior caso e $O(m)$ ou $O(n)$ (dependendo dos tamanhos do padrão e do texto) no melhor caso e caso médio. Na segunda estratégia, o pior caso é $O(m \times n)$ e o melhor caso e caso médio é $O(\frac{n}{m})$ ou $O(\frac{m}{n})$ (dependendo também do valor de m e n. E, para finalizar, a complexidade da terceira estratégia é $O(m)$ ou $O(n)$ para o melhor caso, pior caso e caso médio

4 Análise dos Resultados Obtidos

Após vários testes, selecionamos três exemplos que consideramos mais relevantes para as nossas análises tanto do tempo de sistema quanto do tempo

de usuário, e sua relação com os tempos do relógio. Para isso, utilizamos as funções **getrusage** e **gettimeofday**.

O tempo de usuário representa o tempo de CPU gasto no código no modo usuário (fora do kernel) dentro do processo, é apenas o tempo real de CPU usado durante a execução do programa. O tempo de sistema é a quantidade de tempo da CPU gasto durante o período que o programa executa no modo kernel. Já o tempo de relógio é todo o tempo decorrido, isto é, do início ao fim da chamada, inclui o tempo utilizado em outros processos e o tempo que a execução fica parada esperando terminar operações de entrada/saída. Com isso, o primeiro teste possui 10 casos de teste, o segundo 50 e o último 100, a tabela a seguir mostra os dados obtidos durante as execuções para cada teste:

Dados obtidos com as funções getrusage e gettimeofday no teste 1			
Variação do tempo	Estratégia 1	Estratégia 2	Estratégia 3
Sistema	0s - 0,000817s	0s - 0,000744s	0s - 0,000409s
Usuário	0s - 0,000684s	0s - 0,000577s	0s - 0,000385s
Relógio	0,000564s - 0,000816s	0,000575s - 0,000742s	0,000382s - 0,000406s

Dados obtidos com as funções getrusage e gettimeofday no teste 2			
Variação do tempo	Estratégia 1	Estratégia 2	Estratégia 3
Sistema	0s - 0,001006s	0s - 0,000959s	0s - 0,000880s
Usuário	0s - 0,001588s	0s - 0,001145s	0s - 0,001022s
Relógio	0,000538s - 0,001710s	0,000502s - 0,001105s	0,000485s - 0,000992s

Dados obtidos com as funções getrusage e gettimeofday no teste 3			
Variação do tempo	Estratégia 1	Estratégia 2	Estratégia 3
Sistema	0s - 0,003277s	0s - 0,001910s	0s - 0,001621s
Usuário	0s - 0,002895s	0s - 0,002419s	0s - 0,002205s
Relógio	0,001477s - 0,003273s	0,001358s - 0,003178s	0,001295s - 0,002902s

Dessa maneira, nota-se que o **Algoritmo Força Bruta**, apesar de possuir uma implementação mais simples, é relativamente menos eficiente do que os outros dois algoritmos utilizados, sendo mais adequado utilizá-lo para tamanhos de entradas menores. Sendo assim, quando o tamanho do texto for muito grande, podendo levar ao pior caso do algoritmo, não é recomendado utilizá-lo. Por outro lado, o uso do **BMH** é bem produtivo, sendo mais eficiente do que o **Força Bruta** na grande maioria dos casos e, em certas situações, o desempenho do **Shift-And** foi um pouco superior. Já o **Shift-And**, é bem mais eficiente do que o **Força Bruta**, justamente por passar sobre o texto apenas uma vez ($O(n)$) e possui um desempenho superior ao

BMH quando o padrão for pequeno.

5 Conclusão

Portanto, ao realizarmos este trabalho prático, conseguimos implementar as duas primeiras estratégias sem maiores problemas, o que tivemos maior dificuldade foi em tratar o texto como sendo circular. Já na terceira implementação, não tivemos entrave em implementar o algoritmo **Shift-And** para textos que não são circulares. Contudo, não conseguimos tratar o caso do texto circular utilizando esse algoritmo, ocasionando alguns resultados incorretos para alguns testes.

6 Referências

ZIVIANI, N. **Projeto de Algoritmos: com implementações em PASCAL e C.** 3 ed. São Paulo: Cengage Learning, 2013.