Universidade Federal de São João del Rei
Departamento de Ciência da Computação
Curso de Ciência da Computação

Trabalho Prático 3: implementação de algoritmos de casamento de padrão em string circulares

Rodrigo José Zonzin Esteves

1 Introdução

1.1 Apresentação do problema

Seja as seguintes *strings* T = ABCSDEFABC e P = BCA. O objetivo do nosso trabalho é implementar algoritmos de casamento de padrão (como *Shift-And, BMH e etc*) para encontrarmos a posição de ocorrência do primeiro casamento do padrão P em T.

Existe, no entanto, uma peculiaridade a ser tratada: a *string* T é circular. A figura a seguir apresenta a circularidade da *string* tomando como analogia uma fita.

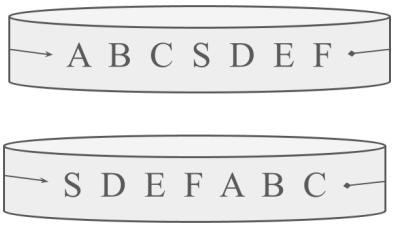


Figura 1: A circularidade da *string* representada por uma fita

1.2 Relevância do tema

A recuperação de informação é um campo que lida com o acesso eficiente a grandes quantidades de dados não estruturados, como texto, imagens e vídeos. Neste sentido, os algoritmos de casamento de padrão são os mais elementares para essa área de estudo e são usados como pilares para a construção de soluções mais complexas.

O presente trabalho busca introduzir a aplicação desses conceitos em um cenário não trivial. Implementar tais algoritmos é uma tarefa relativamente simples, já que o tema é fartamente explorado pela comunidade acadêmica e por fóruns online de ensino de programação. Dessa forma, a não linearidade do texto é o principal desafio enfrentado para a execução correta desse trabalho e a forma como lidamos com esse problema é discutida na seção 2.

2 Modelagem do problema

Sejam as *strings* T_i textos sobre as quais devemos procurar os respectivos padrões p_i.

$$\begin{array}{ll} T_1 = av & p_1 = ava \\ T_2 = npatapatapatapo & p_2 = patapon \\ T_3 = lleksah & p_3 = haskell \end{array}$$

Não faz sentido procurarmos a ocorrência de um padrão que é maior do que o próprio texto. Porém, a circularidade da nossa *string* garante que podemos dar infinitas voltas ao seu redor.

Temos portanto a seguinte situação:

$$\begin{array}{ll} T'_1 = avavav... & p_1 = ava \\ T'_2 = npatapatapatapatapatapatapo... & p_2 = patapon \\ T'_3 = lleksahlleksahlleksah... & p_3 = haskell \end{array}$$

Note que isso soluciona o problema para T'_1 e T'_2. Caso o padrão seja, por exemplo, muitas vezes maior do que o texto, precisaríamos dobrar mais do que apenas uma vez.

Considere o primeiro caso. Se tivéssemos $p'_1 = avavav$, precisaríamos de concatenar o texto em si próprio por pelo menos 3 vezes.

Note também que essa abordagem não soluciona o problema de T_3. Para resolvermos esse problema, adotamos a estratégia elucidada na figura a seguir, onde percorremos o texto nos dois sentidos: $direita - esquerda$ e $esquerda - direita$.

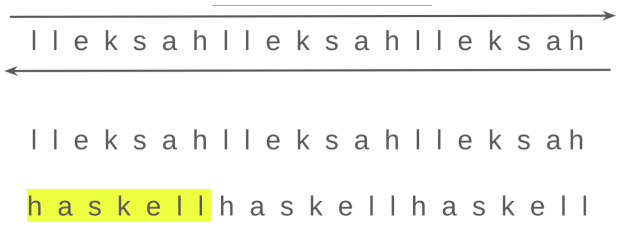


Figura 2: Texto T_3 dobrado e ao revertido

Note que a posição de casamento na string dobrada e revertida é a posição zero, mas a resposta esperada é dada em termos da string original. Para resolvermos esse problema, estabelecemos um mapa de domínio nas posições da string tratada e imagem na string original.

Podemos derivar o termo geral da sequência observando o padrão.

$$\begin{array}{c} 0 \mapsto 20 = 20 - 0 \\ 1 \mapsto 19 = 20 - 1 \\ 2 \mapsto 18 = 20 - 2 \\ ... \\ 20 \mapsto 0 = 20 - 20 \end{array}$$

Temos, portanto, $\delta(n) = 20 - n$. De forma, mais geral, temos $\delta(n) = |T'_i| - n$. [1]. Note, no entanto, que essa função mapeia valores maiores do que o tamanho da string original. Se tirarmos o módulo de $\delta(n)$ pelo tamanho da string original, toda a sequência será mapeada para os valores corretos. Temos, enfim, a seguinte função:

$$\tau(n) = \delta(n) \ mod |T_i|$$
$$\tau(n) = (|T'_i| - n) \ mod |T_i|$$

[1] Notação: O módulo $|T|$ de uma string é o seu tamanho

Retornando ao exemplo do padrão *haskell*, o valor mapeado pelo algoritmo de casamento de padrão sob a string tratada (dobrada e revertida) T_3' seria a posição zero. Tomando $\tau(0)$ obtemos $\tau(0)(20-0)\ mod(7) = 6$, que corresponde, de facto, ao caracter h na string original T_3 .

Caso o algoritmo tivesse retornado a posição 8 (a letra A), teríamos $\tau(8) = (20-8)\ mod(7) = 5$, que é, de fato, a posição correta no texto original *lleksah*.

3 Tratamento da execução dos algoritmos

Com a modelagem exposta anteriormente temos duas preocupações: o casamento de padrão pode ocorrer tanto em um sentido quanto no outro. Dessas duas possibilidades surge a necessidade de executarmos, para cada entrada, duas execuções do algoritmo de casamento de padrões.

Devemos também tratar os resultados diferentes retornados para as execuções nos sentidos opostos. Por exemplo: caso o algoritmo tenha sucesso no sentido $EsqDir$, ele retornará um inteiro contendo a posição do primeiro casamento. No entanto, ele pode não encontrar casamento no sentido $DirEsq$, resultando em um inteiro -1. [2].

Dessa maneira, elencamos as seguintes possibilidades de retornos do algoritmo para os sentidos EsqDir e DirEsq.

1. -1 e -1.

   Não houve casamento nem da esquerda para a direita, nem da direita para a esquerda. Não há casamento e devemos registrar.

2. -1 e *resposta_valida*.

   Não houve casamento em um dos sentidos mas houve no outro. Devemos registrar a ocorrência do casamento no sentido em questão e a posicao válida.

3. $resposta\_valida_1$ e $resposta\_valida_2$.

   Houve casamento de padrão em ambos os sentidos e nossa modelagem assegura que $resposta\_valida_1 = resposta\_valida_2$. Trata-se de um palíndromo.

4 Estrutura geral do código

De maneira geral, temos o seguinte fluxo de execução do código:

```
T = read_inteiro()
strings[T]
sequencias[T]

for i = 0 to T do
  sequencias[i] = read_string()
  strings[i] = read_string()
end

strings = dobra_strings(strings)
strings_revertidas[T] = reverte_strings(strings)

forca_bruta(strings, strings_revertidas, sequqencias)
KNP(strings, strings_revertidas, sequqencias)
BMH(strings, strings_revertidas, sequqencias)

libera(strings)
libera(strings_revertidas)
libera(sequqencias)
```

[2]Só haverá casamento idêntico em ambos os sentidos quando o texto for um palíndromo

Note que não abstraímos por completo o gerenciamento de memória, pois ele é um aspecto extremamente importante.

5 Análise de Complexidade

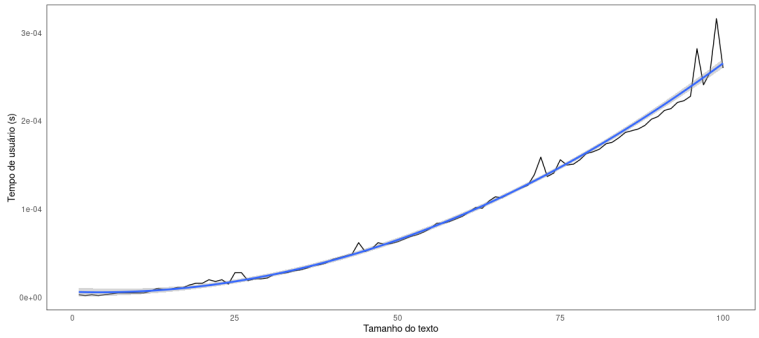
A tabela a seguir apresenta a complexidade para as principais funções utilizadas no código. Como nosso interesse é na complexidade de tempo, tomamos as **comparações** como a **variável de interesse**.

Função	Complexidade de tempo	Justificativa
aloca_pedra(n)	$O(n)$	É preciso iterar sobre todos os n elementos, alocando-lhes uma string de 1000 *chars*.
aloca_sequencia(n)	$O(n)$	Trabalha de forma idêntica à função *aloca_pedra(n)*, exceto pelo tamanho dos blocos alocados: apenas o suficiente para 100 *chars*
muda_extensao(string)	$O(1)$ ou $O(n)$´	Essa função recebe uma string com a extensão *".txt"* e retorna a mesma string, mas com a extensão *".out"*. Não há for no corpo da função, mas ela trabalha com a função *strrchr*, que itera sobre todos os n elementos da string no pior caso.
dobra_e_concatena_pedra(strings)	$O(1)$ ou $O(n)^3$	Essa função é responsável por receber uma *string*, copia-la para um bloco recém alocado de tamanho 2000 bytes e concatenar a mesma string ao final de si. A nível de usuário, essa função tem custo constante $O(1)$, mas sua implementação envolve iterar sobre si através de dois fors aninhados. (Linux Foundation, 2023)
dobra_sequencia(string, n)	$O(n)$	A função inicia alocando uma pedra de n elementos. Como vimos, esse procedimento tem custo $O(n)$. Em seguida, ele itera sobre todos as n strings, usando a função anterior. Com o uso da propriedade multiplicativa, podemos considerar o *for* e a execução da função *dobra_e_concatena_pedra()* como um único código de complecidade $O(O(n) \cdot O(n)) = O(n^2)$. E por meio da propriedade aditiva, representamos a primeira alocação seguida imediatamente por um bloco de código quadrático, como $O(O(n)+O(n^2)) = max(O(n), O(n^2)) = O(n^2)$
reverte_pedra(strings, n)	$O(n)$	A função *strlen*() da biblioteca *string.h* tem complexidade $O(n)$. Além dela, há ainda a iteração sobre todos os n elementos da função, que permuta, par a par, do primeiro ao útlimo elemento. Temos portanto, complexidade $O(n)$.
forca_bruta(string)	$O(n \cdot m)$	Ziviani (2004)
bmh(string)	$O(n \cdot m)$	GeeksforGeeks (2023)
kmp(string)	$O(n)$	Cormen et al. (2009)

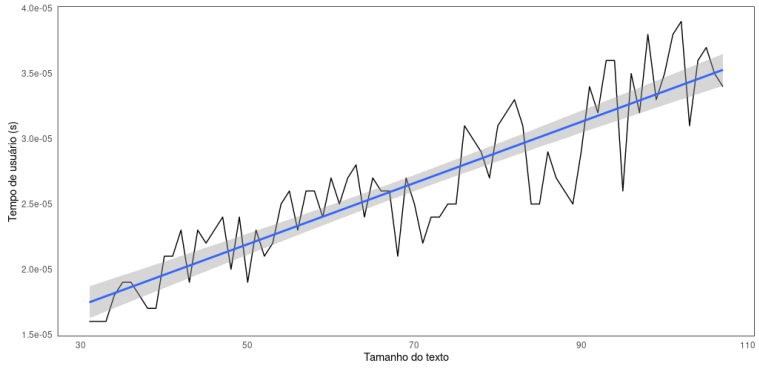
5.1 Análise empírica dos dados

A figura a seguir apresenta os tempos (em segundos) obtidos para os testes realizados. Para criarmos o nosso *dataset* de teste, usamos a biblioteca *RandomWords* da linguagem Python.

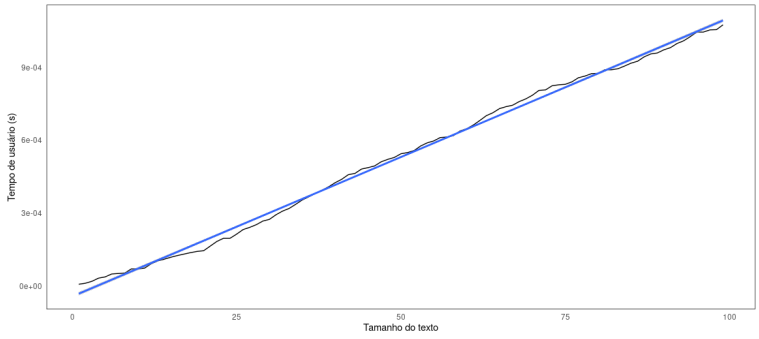
Definimos um padrão único para todos os testes ("**padrao2022**") e $n = 100$ palavras de teste. Para cada uma das n linhas/palavras de teste, concatenamos termos aleatórios e, aleatoriamente, um padrão entre as posições da porção final do teste, *i.e.*, entre as posições $\lceil |T_i|/2 \rceil$ e $|T_i|$.



(a) Força Bruta



(b) KMP



(c) BMH Bruta

Figura 3: Análise do tempo de execução

Como sabemos da literatura, o algoritmo de casamento de padrão por força bruta apresenta complexidade $O(m \cdot n)$, onde $n = \textit{tamanho do texto}$ e $m = \textit{tamanho do padrão}$. Se $n \approxeq m$, temos um comportamento quadrático Ziviani (2004). Nosso algoritmo esboça esse desempenho, ainda q de maneira pouco abrupta.

Algoritmo	Média (s)	Variância
Força Bruta	8.998e-05	6.290e-09
KMP	3.248e-05	1.481e-02
BMH	4.551e-04	2.479e-10

Ademais, os algoritmos BMH e KMP esboçam um comportamento linear. Esse comportamento não está em consonância com a descrição teórica do BMH, que deveria ter complexidade $O(n \cdot m) = O(n^2)$, para $n \approxeq m$. Provavelmente isso se justifique pelo fato de $n >> m$ para os testes gerados. Além disso, também podemos elencar a hipótese de termos um número relativamente baixo de exemplos: $n = 100$. Tal tamanho de testes pode não ser suficiente para permitir uma visualização mais apurada de seu comportamento assintótico.

O algoritmo KMP apresenta variância acentuada, o que pode ser percebido também na figura 3(b). Esse resultado provém, possivelmente, do acesso à tabela de prefixos.

O algoritmo BMH executa, majoritariamente, apenas operações aritméticas e lógicas, em uma tabela estaticamente armazenada.

6 Funcionalidade

A imagem abaixo mostra a saída do terminal após a execução do caso de teste descrito na documentação apresentada em sala.

```
zonzin@rodrigo:~/Documentos/Faculdade/AEDSIII/tp3/primeira_versao$ ./tp3 teste.txt 1
Temppo usuario: 0.000000        Tempo sistema: 0.000010
Temppo usuario: 0.000000        Tempo sistema: 0.000010
Temppo usuario: 0.000000        Tempo sistema: 0.000007
Temppo usuario: 0.000000        Tempo sistema: 0.000007
zonzin@rodrigo:~/Documentos/Faculdade/AEDSIII/tp3/primeira_versao$
```

Figura 4: Saída no terminal

A imagem a seguir apresenta o arquivo gerado com o mesmo nome e a extensão ".out".

```
zonzin@rodrigo:~/Documentos/Faculdade/AEDSIII/tp3/primeira_versao$ cat teste.out
S 1
S 10
N
S 7
```

Figura 5: Respostas corretas

7 Conclusão

Ao analisar os resultados obtidos a partir das funções de complexidade identificadas, constatou-se que eles são consistentes com a implementação dos algoritmos neste trabalho, porém não coincidem com os registros documentados em outras fontes. No entanto, foi comprovado que os algoritmos teoricamente descritos como mais eficientes em relação aos demais realmente se demonstraram melhores. Esse fato fica evidenciado quando comparamos o KMP e a abordagem por força bruta, onde tivemos uma eficiência bem melhor daquele em relação a este.

Apesar de o algoritmo BMH ter apresentado um comportamento diferente do esperado, consideramos que nossa investigação inicial fornece elementos suficientes para um questionamento mais aguçado, que poderá ser feito em trabalhos posteriores. Citamos como limitação do trabalho, a necessidade de usar o algoritmo duas vezes: em um sentido e no outro. Apesar de ainda termos uma complexidade $O(2n) = O(n)$ e a plena funcionalidade da solução, temos um desempenho inferior a outros tipos de abordagens (por lista circular, por exemplo).

Por fim, uma vez que os resultados obtidos confirmaram a eficiência dos algoritmos teoricamente descritos como superiores, sentimo-nos seguros para dizer que a análise empírica do desempenho de algoritmos pode ser uma ferramenta extremamente poderosa. Em particular, podemos usá-la quando uma abordagem analítica não é possível, como no caso do Shellsort, que tem uma excessiva complexidade matemática. Dessa maneira, podemos usar ferramentais estatísticos e computacionais para preencher, ao menos parcialmente, lacunas de importantes áreas da Ciência.

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Algorithms*. The MIT Press, Cambridge, MA, 3rd edition.

GeeksforGeeks (2023). *Boyer-Moore Algorithm for Pattern Searching*.

Linux Foundation (2023). *The linux man-pages project*.

Ziviani, N. (2004). *Projetos de Algoritmos*. Cengage Learning, São Paulo, Brasil.