

Gustavo Viegas - 3026
Heitor Passeado - 3055

Trabalho Prático 4 - Busca de padrão em String, casamento perfeito

Documentação de Trabalho Prático - TP3

Universidade Federal de Viçosa - Campus Florestal

Projeto e Análise de Algoritmos

Ciência da Computação

Florestal

2 de Dezembro de 2018

Lista de ilustrações

Figura 1 – Menu inicial	6
Figura 2 – Texto carregado com sucesso	7
Figura 3 – Ao pesquisar a palavra "rato"no texto carregado, o texto é exibido com todas as ocorrências de "rato"em destaque	7
Figura 4 – Função openFile	8
Figura 5 – Código para exibir o texto em destaque	9

Sumário

1	Introdução	4
1.1	Decisões	4
2	Ambiente	5
3	Instruções gerais	6
4	Desenvolvimento	8
4.1	Util	8
4.1.1	Boolean	8
4.1.2	Logger	8
4.1.3	FileReader	8
4.1.4	Benchmark	9
4.1.5	Interface	9
4.2	Casamento perfeito	9
4.2.1	Força bruta	9
4.2.2	Rabin-Karp	10
5	Análise de tempo de execução	11
6	Conclusão	13
	Referências	14

1 Introdução

Em sala de aula foi estudado diversas formas de como se buscar padrões em um texto, como força bruta, Boyer-Moore, shift-and etc. Também foi discutido formas de se buscar por casamento aproximado, levando em consideração a distância de edição(substituir, inserir ou remover uma letra). O objetivo desta prática é implementar dois desses métodos para casamento perfeito e gerar gráficos e analisar o tempo de execução comparando-os.

1.1 Decisões

Um dos impasses da prática era como se utilizar o "**define**" para determinar o modo que o programa irá executar. A solução foi definir na main que *MODO_ANALISE* era igual a 1 e *MODO_NORMAL* era igual a 0, então se fosse passado um argumento na execução do programa, make analysis mais especificamente, a o programa seria executado no modo análise.

Os algoritmos escolhidos foram força bruta, visto a simplicidade da implementação e a grande diferença do tempo de execução e Rabin karp que consiste na utilização de tabela hash.

Os textos utilizados vieram do gerador de lero-lero.

2 Ambiente

Para o desenvolvimento do trabalho foi utilizada a IDE **Atom** e o sistema operacional **macOS High Sierra**.

A compilação do código C foi feita pelo compilador **Apple LLVM 9.0**. Nenhuma flag ou configuração customizada foi alterada para o processo de *build* do programa.

Para a execução do executável gerado pelo *Xcode* foi utilizado o software **iTerm 2** que substitui o terminal nativo do sistema operacional. Foi utilizado um terminal externo ao da IDE para corretamente mostrar os caracteres coloridos que são utilizados no trabalho, além de ajudar a visualizar as matrizes impressas.

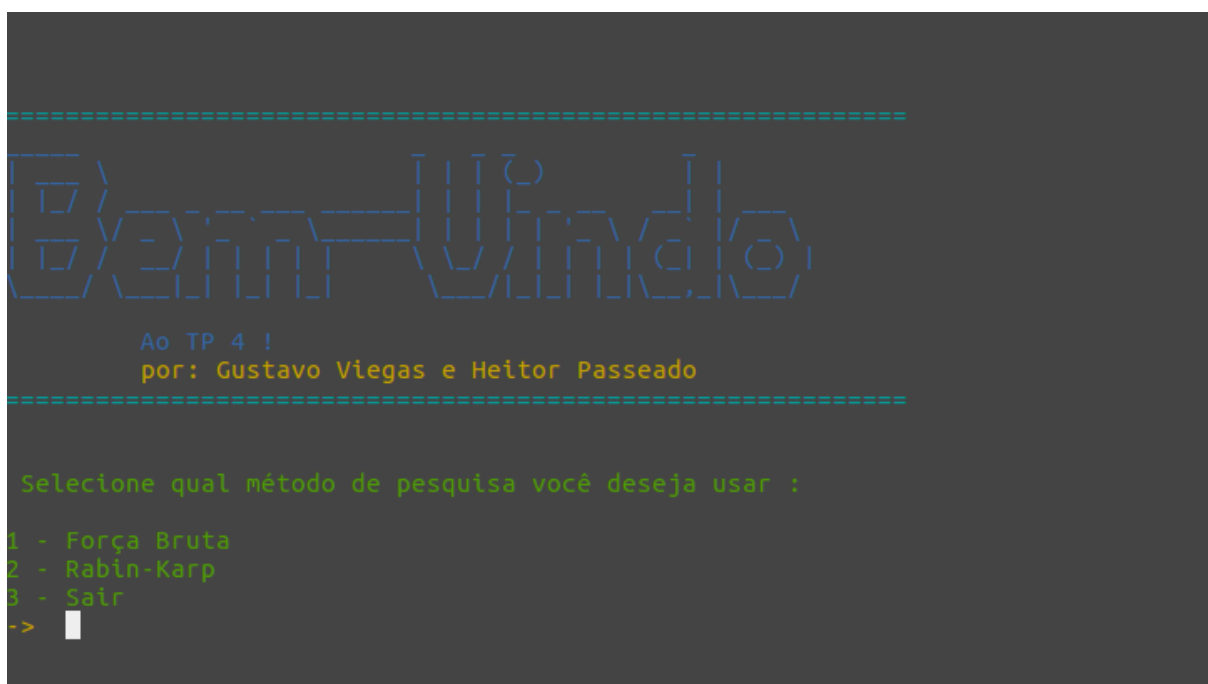
As especificações da máquina utilizada são:

MacBook Pro (15-inch, Late 2011)
Processador: 2,4 GHz Intel Core i7
Memória: 16 GB 1333 MHz DDR3
Disco 120 GB SSD

3 Instruções gerais

Para realizar o build/executar o programa, basta utilizar o Makefile que está contido na raiz do projeto.

Ao abrir o terminal no diretório do programa o usuário deve digitar **make** para fazer o *build* do programa, e então ele pode digitar **make run** para executar o programa na forma padrão ou **make analysis** para executá-lo em modo análise, ao confirmar o comando o terminal exibirá:



```
=====
Ben-Vendo
Ao TP 4 !
por: Gustavo Viegas e Heitor Passeado
=====

Selecione qual método de pesquisa você deseja usar :

1 - Força Bruta
2 - Rabin-Karp
3 - Sair
-> 
```

Figura 1 – Menu inicial

Para realizar sua escolha, o usuário deve digitar o número correspondente e então apertar **enter** e o menu para aquele método de pesquisa será exibido, a primeira coisa que ele pede é para carregar um arquivo contendo o texto. Após carregá-lo, o programa mostrará uma tela com 3 opções: buscar uma palavra no arquivo carregado, carregar outro arquivo ou voltar para o menu principal.

```
[MOD0 FORÇA BRUTA][CAMINHO] (Exemplo: resources/textoBase.txt)
Insira o caminho do arquivo de leitura:
-> resources/textoBase.txt
Texto carregado!

Este eh um texto exemplo, irei buscar palavras para casamento perfeito.
O rato roeu a roupa roubada do rei de roma.
O rato
roeu
a roupa
roubada
do rei de
roma

Pressione enter para continuar...
```

Figura 2 – Texto carregado com sucesso

```
=====
1 - Buscar palavra
2 - Carregar outro arquivo
3 - Voltar ao menu
-> 1
Digite a palavra :
-> rato

=====
Texto com as ocorrências em destaque:
=====
Este eh um texto exemplo, irei buscar palavras para casamento perfeito.
O rato roeu a roupa roubada do rei de roma.
O rato
roeu
a roupa
roubada
do rei de
roma

Pressione enter para continuar...
```

Figura 3 – Ao pesquisar a palavra "rato" no texto carregado, o texto é exibido com todas as ocorrências de "rato" em destaque

4 Desenvolvimento

4.1 Util

Para fazer o programa, foi criado diversos módulos auxiliares como:

4.1.1 Boolean

Define um tipo booleano para deixar o código mais legível quando se envolve operações booleanas.

4.1.2 Logger

Biblioteca que declara um enumerador com cores disponíveis e reimplementa a função *printf* passando uma cor com argumento. Há também uma função pra auxiliar a imprimir uma linha em todo o programa.

4.1.3 FileReader

Contem as seguintes funções:

1. **promptFilePath**: Pede o caminho de um arquivo e o lê do teclado, recebe como parâmetro uma string que armazenará o caminho lido
2. **openFile**: Código auto explicativo.

```
void openFile(FILE** file, char *filePath) {  
    *file = fopen(filePath, "r");  
  
    if (*file == NULL) {  
        cprintf(RED, "O arquivo %s não existe ou não pode ser lido corretamente.\n", filePath);  
        cprintf(BLUE, "Confira o caminho inserido e digite outro.\n\n", filePath);  
        promptFilePath(filePath);  
        return openFile(file, filePath);  
    }  
}
```

Figura 4 – Função openFile

3. **readLine** Recebe o arquivo e uma string, le a linha com **fgets()** e armazena na string.

Além de funções criadas para esse trabalho que consiste em ler um texto de um arquivo e armazenar em um vetor e a de exibir as palavras em destaque.

4.1.4 Benchmark

Módulo que encapsula as chamadas de contagem e verificação do *clock* para calcular o tempo de execução de um método. O módulo pode operar em diversas métricas: Segundos, Milisegundos e Microssegundos.

4.1.5 Interface

Esse módulo foi criado com o objetivo de manipular as escolhas do usuário e redirecioná-lo para os menus dos problemas. contém a única função que é chamada na main.

4.2 Casamento perfeito

Para exibir as palavras encontradas no texto foi criada uma função que recebe um vetor contendo os índices das ocorrências, a quantidade de ocorrências, o tamanho da ocorrência e o texto em si. Para cada posição do texto verifica-se se é um índice que começa a palavra, caso positivo até completar o tamanho das ocorrências os caracteres são exibidos de outra cor.

```
void printOcorrences(int *foundWordArray, int numberOfOccurrences, int searchSize, char *text) {
    printLine();
    int textSize = strlen(text);
    for(int i = 0; i < textSize; i++) {
        for (int j = 0; j < numberOfOccurrences; j++) {
            if (foundWordArray[j] == i) {
                for (int k = i; k < searchSize + i; k++) {
                    cprintf(BLUE, "%c", text[k]);
                }

                i += searchSize;
            }
        }
        printf("%c", text[i]);
    }
}
```

Figura 5 – Código para exibir o texto em destaque

4.2.1 Força bruta

O algoritmo segue padrões bem simples para o casamento perfeito, para cada letra do texto ele percorre todo o tamanho do padrão buscado verificando se está "batendo" letra

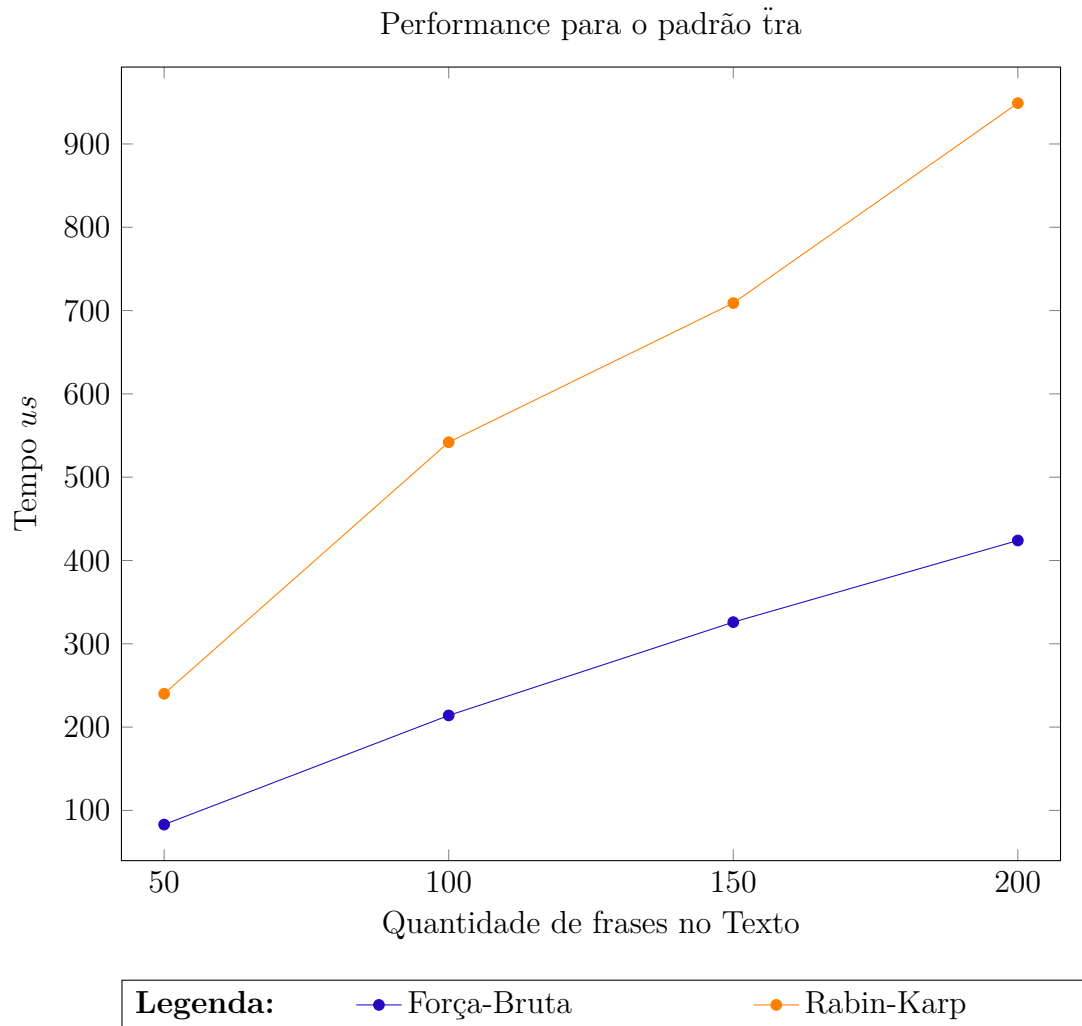
por letra, caso o tamanho do padrão seja alcançado, significa que a palavra inteira correspondeu.

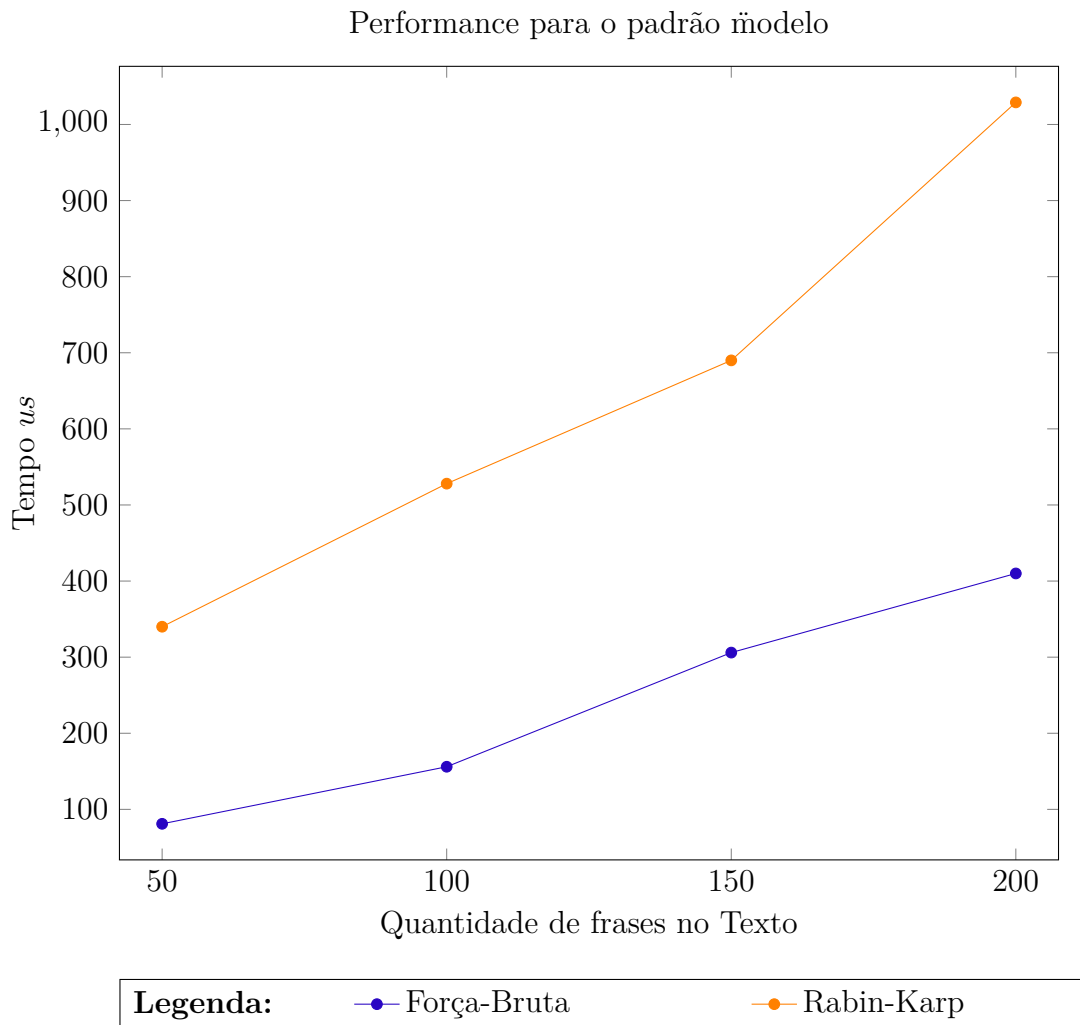
4.2.2 Rabin-Karp

Algoritmo bem parecido com o força-bruta, percorrendo o texto caracter por caracter ao encontrar um padrão de matching da janela de padrão com o uso de uma tabela hash.

5 Análise de tempo de execução

Para a análise, foi utilizado um gerador de lero lero [UFSM \(2018\)](#). Foi gerado 4 arquivos, com 50, 100, 150 e 200 frases cada. Utilizamos estes arquivos para comparar a quantidade de tempo gasto pra uma mesma ocorrência (padrão "tra").





Pra diferentes padrões, que não tiveram gráfico plotados aqui, vemos o mesmo padrão. Um crescimento bem parecido dos dois algoritmos, com a força-bruta sendo levemente mais eficiente pra os dados testados.

A explicação desse fenômeno se dá porquê o Rabin-Karp percorre o texto um por um, tal como na força bruta, mas possui muito mais cálculos pra eficientemente colocar tudo em uma tabela hash. O Rabin-Karp é mais eficiente em memória e outros aspectos que não conseguimos evidenciar só com as ferramentas que possuímos. O Rabin-Karp é um algoritmo mais robusto e tem uma ordem de complexidade levemente menor - $O(m+n)$ no caso médio, mas na prática é difícil ver um ganho tão significativo. A parte boa, é que teremos uma hash no fim das contas que pode ser utilizada como um pré-processamento do arquivo, que pode ser lido diversas vezes e já ter seu valor resgatado, como numa Trie, diferente do força-bruta.

6 Conclusão

O conceito de processamento de cadeia de caracteres foi um dos pontos chave dessa prática, ele foi amplamente pensado e discutido durante a execução da mesma.

O trabalho se mostrou útil para estudar tais algoritmos e evidenciar sua complexidade e estratégias estudadas em sala de aula além de fortalecer nossas habilidades de programadores e conhecimento em algoritmos e estruturas de dados.

Referências

UFSM. *Fabuloso Gerador de Lero Lero*. 2018. Disponível em: <[http://www.cafw-ufsm.br/~bruno/disciplinas/desenvolvimento_web/material/lerolero.html](http://www.cafw.ufsm.br/~bruno/disciplinas/desenvolvimento_web/material/lerolero.html)>. Acesso em: 1.12.2018. Citado na página 11.

- *https : //www.ntg.nl/doc/biemesderfer/ltxcrib.pdf* para realização da documentação;