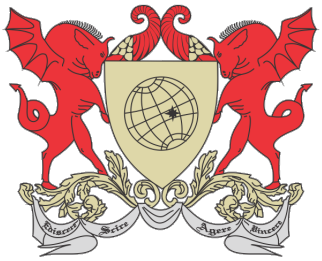

UNIVERSIDADE FEDERAL DE VIÇOSA
CAMPUS FLORESTAL

Trabalho Prático 3
Projeto e Análise de Algoritmos

Artur Papa - 3886
Guilherme Fernandes - 3398

Trabalho Prático apresentado à disciplina de CCF 330 - Projeto e Análise de Algoritmos do curso de Ciência da Computação da Universidade Federal de Viçosa.

Florestal
Dezembro de 2022

CCF 330 - Projeto e Análise de Algoritmos

Trabalho Prático 3

Artur Papa - 3886
Guilherme Fernandes - 3398

10 de Dezembro de 2022

Contents

1 Introdução

Podemos dizer que é um fato que todas as pessoas já se perguntaram como nosso cérebro interpreta a luz para que possamos identificar cores, formatos, ou mesmo quem é sua mãe, sua tia ou aquela pessoa no espelho? Desta forma, para entender isso precisamos entender o conceito de ***similaridade***.

Para determinar a similaridade entre dois objetos são necessárias informações numéricas sobre as características destes objetos e, uma dessas informações pode ser o uso de vetores e o ângulo entre os mesmos, sendo que este uso se chama ***similaridade por cosseno***. Por exemplo, imagine que um vetor sai da origem do sistema de coordenadas e termina no ponto X. Este vetor é usado para localizar o ponto no nosso espaço de características. A ***similaridade*** entre dois pontos é calculada da seguinte maneira:

$$Similaridade = cos(\theta) = \frac{\langle X, A \rangle}{||X||.||A||}$$

Figura 1: Fórmula da similaridade

Dito isso, um dos usos deste conceito seria verificar a ***similaridade*** entre sequências de DNA, podemos fazer a comparação entre a sequência genética de um leão e uma onça por exemplo. Assim, é válido dizer que o objetivo principal do trabalho é analisar a ***similaridade*** entre sequências de DNA de um humano, um cachorro e um chimpanzé.

2 Desenvolvimento

Nesse contexto, tem-se que para calcular a similaridade entre as sequências de DNA usadas foi necessário fazer uso de um algoritmo de padrão de busca, assim, ficou definido pela dupla que o algoritmo a ser utilizado seria o algoritmo de ***Boyer Moore***.

2.1 Algoritmo de padrão de busca

Nesse intuito, o algoritmo de ***Boyer Moore*** implementado é uma combinação de duas heurísticas, sendo elas a **Bad Character Heuristic** e a **Good Suffix Heuristic**.

A ideia do **Bad Character Heuristic** é simples. O carácter do texto que não corresponde ao carácter atual do padrão é chamado de **Bad Character**. Após a incompatibilidade, mudamos o padrão até que a incompatibilidade se torne uma correspondência ou o padrão P move-se além do carácter incompatível. No primeiro caso, procuraremos a posição da última ocorrência do carácter incompatível no padrão e, se o carácter incompatível existir no padrão, deslocaremos o padrão para que fique alinhado ao carácter incompatível no texto T. No segundo caso, procuraremos a posição da última ocorrência do carácter incompatível no padrão e, se o carácter não existir, mudaremos o padrão após o carácter incompatível.[1]

Assim, a implementação do algoritmo ficou da seguinte forma:

```
void badCharHeuristic(char *str, int size, int badChar[NO_CHARS])
{
    int i;
    for (i = 0; i < NO_CHARS; i++)
    {
        badChar[i] = -1;
    }
    for (i = 0; i < size; i++)
    {
        badChar[str[i]] = i;
    }
}
```

Figura 1: Heurística Bad Character

Após ser implementada essa heurística foi feita a construção do algoritmo de busca para saber em que posição o padrão buscado ocorre, dessa maneira bastou modificar o código para que pudesse ser feita a contagem de quantas vezes a busca foi encontrada e armazenar no vetor para ser feita a similaridade depois.

```
while (s <= (n - m))
{
    int j = m - 1;

    while (j >= 0 && pat[j] == txt[s+j])
        j--;
    if (j < 0)
    {
        s += (s+m < n) ? m - badChar[txt[s+m]] : 1;
        count++;
    }
    else
        s += max(1, (j - badChar[txt[s+j]]));
}
```

Figura 2: Função para pesquisa de padrão

2.2 Definições

Vale citar que foram feitas algumas definições para que pudesse melhorar a qualidade do código do projeto.

```
#define NO_CARTESIAN 4
#define NO_CHARS 256
#define BENCHMARK 10
#define LINE 16
#define COL 2
```

Figura 3: Constantes utilizadas no projeto

A constante **NO_CARTESIAN** serve para definir a quantidade de pares que serão selecionadas do nosso produto cartesiano, nesse caso foi feito a definição de um produto cartesiano de tamanho 2, com uma quantidade de 4 elementos escolhidos aleatoriamente para cada iteração feita no momento da simulação. A constante **NO_CHARS** é utilizada na criação do algoritmo de busca de padrão. **BENCHMARK** define a quantidade de vezes que será feita a simulação, apesar de estar 10 na imagem, a simulação foi definida como 100 no projeto final. **LINE** e

COL são para a criação da matriz com os produtos cartesianos gerados e que são escolhidos aleatoriamente durante a execução.

2.3 Estrutura do projeto

Para a estrutura do projeto, foram criadas sete pastas, assim temos a seguinte arquitetura:

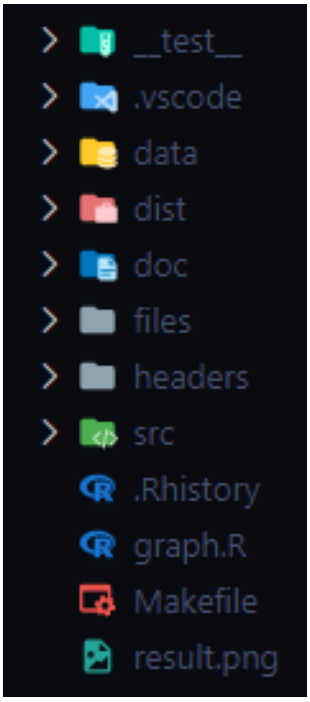

Figura 4: Arquitetura do projeto

Dessa maneira, vale citar que a pasta **data** possui os arquivos de texto com as sequências de DNA que serão utilizadas no programa principal, na pasta **dist** é onde se encontra as saídas dos programas principais, como por exemplo a *main*, após o programa ser compilado, na pasta **doc** encontram-se os documentos do projeto, no diretório **headers** encontram-se os cabeçalhos do projeto e para a pasta **__test__** temos arquivos que foram usados apenas para testar determinadas funções, na pasta **src** é aonde se localiza os códigos fonte do projeto e os demais arquivos com suas funções. Por fim temos a pasta **files** em que ficam armazenados os arquivos **.csv** para gerar o gráfico de desempenho.

2.4 Execução

Para a execução do projeto foi feita a escolha de adicionar um arquivo **Makefile**. Assim, para compilar e executar o programa basta realizar no terminal o comando **make**, para compilar os arquivos e *make run* para rodar o programa. Caso seja executado o programa várias vezes, é recomendável que seja executado o comando **make clean**, para limpar as compilações, e depois executar os comandos para compilar e rodar o programa.

2.5 Simulação

Para a execução da simulação primeiro são feitas as declarações de todas as variáveis, incluindo as do arquivo, após isso é inicializado a matriz com os produtos cartesianos e os vetores em que serão armazenadas as sequências, além de ler os arquivos. Por conseguinte, é feito um loop com a quantidade determinada pela constante **BENCHMARK** - definida com o valor de 100 - como dito anteriormente, dentro desse while a cada iteração são armazenadas as sequências de cada

arquivo, selecionadas aleatoriamente a cada iteração, e também é armazenado o padrão que será analisado para aquela rodagem, depois armazenamos os valores da similaridade em um vetor e, após tendo feito todo o processo de simulação, calculamos a média.

```
while(j < BENCHMARK){
    pattern = selectCartesian(cartesian);
    randomSequence = (rand() % (220000 - 4 + 1)) + 4;
    aux1 = 0;
    aux2 = 1;
    for (int i = 0; i < randomSequence; i++)
    {
        fscanf(fptr, "%c", &firstSequence[i]);
        fscanf(fptr2, "%c", &secSequence[i]);
    }

    for (i = 0; i < NO_CARTESIAN; i++)
    {
        A[i] = search(firstSequence, pattern[aux1], pattern[aux2]);
        B[i] = search(secSequence, pattern[aux1], pattern[aux2]);
        aux1 += 2;
        aux2 += 2;
    }
    printPattern(pattern);
    similarityValues[j] = similarity(A, B);
    similarityAverage += similarityValues[j];
    j++;
}
```

Figura 5: Simulação

```
dist/main
Digite o nome do primeiro arquivo: human.txt
./data/human.txt
Digite o nome do segundo arquivo: dog.txt
./data/dog.txt
Padrão de sequência a ser buscado: TGACGAGG
Padrão de sequência a ser buscado: CTTGATCA
Padrão de sequência a ser buscado: TTTAATGC
Padrão de sequência a ser buscado: CCTACGCT
Padrão de sequência a ser buscado: GCACGACA
Padrão de sequência a ser buscado: TTAGGGTG
Padrão de sequência a ser buscado: ATCTGTAC
Padrão de sequência a ser buscado: ACTCCGGG
Padrão de sequência a ser buscado: CTACGGCG
Padrão de sequência a ser buscado: CACTTGAC
Média da similaridade após 10 execuções: 0.995932
```

Figura 6: Execução com BENCHMARK com valor 10

3 Extra

3.1 Resultados em forma gráfica

O algoritmo foi utilizado com diferentes valores de N para fazer uma análise gráfica. Foi criada uma função que realiza o algoritmo com diversos valores de N e salva em CSV, assim facilitando a leitura e utilização dos dados para análise.

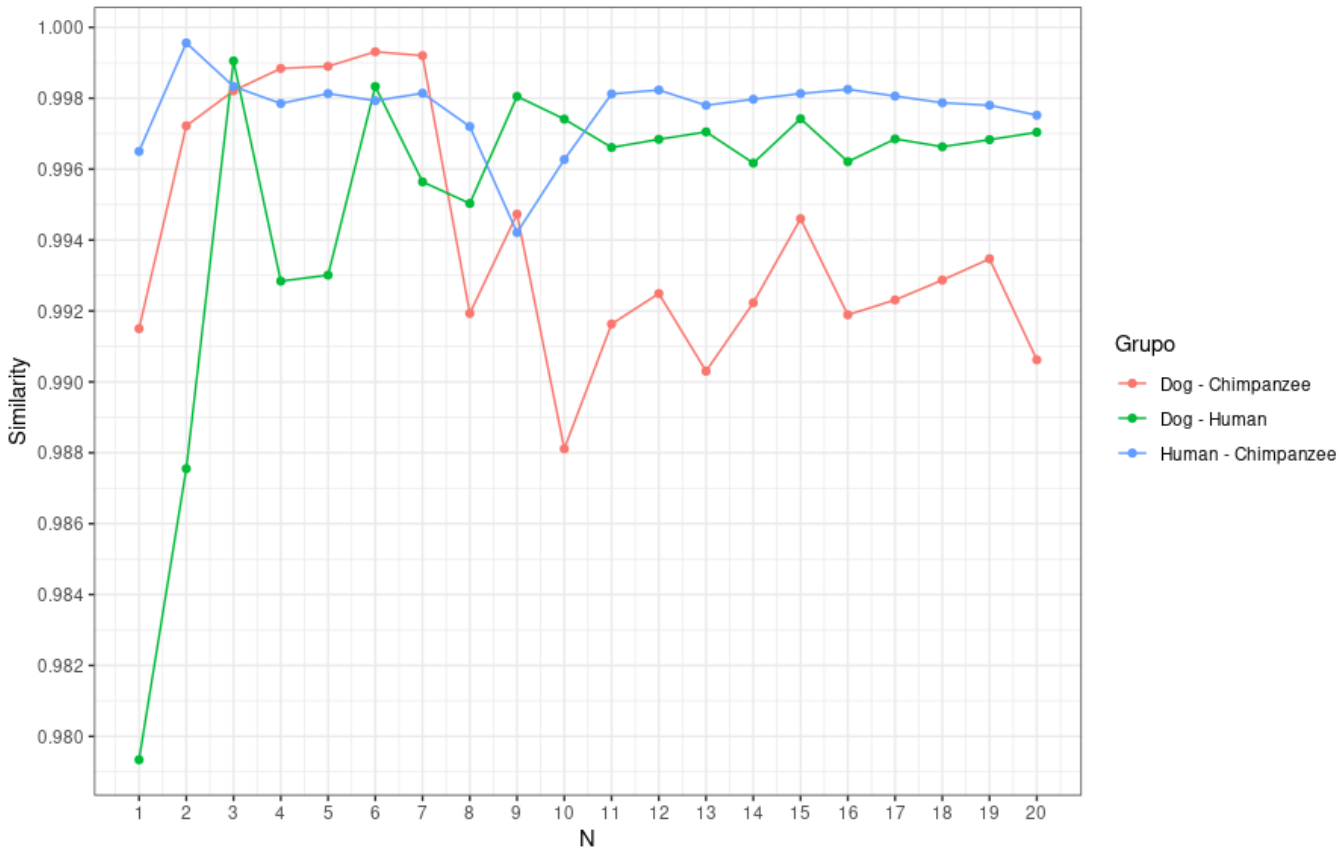

Figura 7: Gráfico de linha da Similaridade x N

Como é possível visualizar no gráfico a Similaridade ficou acima de 0.970 para todas as comparações, sendo assim, não houve diferença para diferentes valores de N.

4 Conclusão

Posto isso, conclui-se que o trabalho em questão foi desenvolvido conforme o esperado, atingindo as especificações requeridas na descrição do mesmo em implementar um programa que utiliza ***busca de padrões*** para determinar a similaridade entre duas sequências de DNA.

Por conseguinte pode-se ressaltar que o material apresentado na disciplina foi de suma importância para o desenvolvimento do projeto, haja vista que as explicações dadas pelo professor ajudou bastante a entender as etapas a serem executadas e na construção dos códigos, assim como as monitorias dadas pelo monitor da disciplina.

Em adição, é válido dizer que apesar das dificuldades na implementação do código, a dupla em questão foi capaz de superar e corrigir quaisquer erros no desenvolvimento do algoritmo. Por fim, verificou-se a assertiva para o objetivo do projeto em implementar um programa que utiliza ***busca de padrões*** para determinar a similaridade entre duas sequências de DNA, além de gerar os gráficos de desmpenho.

References

[1] Boyer-moore. https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/. (Accessed on 12/12/2022).