

Universidade Federal De Viçosa (UFV) - Campus Florestal
Disciplina: Projeto e Análise de Algoritmos
Professor(a): Daniel Mendes Barbosa (CCF330)

Trabalho prático 3

Pair Similarity

Autores:

Miguel Antônio Ribeiro e Silva - 4680

Mateus Henrique Vieira Figueiredo - 4707

Alan Gabriel Martins Silva - 4663

Sumário

1 - Introdução	3
2 - Metodologia	4
3 - Desenvolvimento	5
3.1 - file.c e file.h	5
3.2 - similarity.c e similarity.h	6
3.3 -patternMatching.c e patternMatching.h	9
3.4 - main.c	16
4 - Resultados	18
5 - Conclusão	21
6 - Referências	21

1 - Introdução

O trabalho prático tem como principal problema a elaboração de um programa que calcule a similaridade do DNA entre pares de humano, cachorro e chimpanzé.

Um laboratório, localizado em Wuhan, na China, coletou diversas amostras de DNA de humanos, chimpanzés e cachorros e solicitou que cientistas (conhecidos popularmente como alunos de PAA) da Universidade Federal de Viçosa, determinasse a similaridade entre pares de: (humano-cachorro), (humano-chimpanzé), (cachorro-chimpanzé).

Para isso é necessário calcular a similaridade através da fórmula da similaridade por cossenos, porém, precisa-se saber padrões na sequência de uma fita de DNA, que serão determinadas por algoritmos, como: Boyer-Moore, Shift-And e Knuth-Morris-Pratt.

Após isso, simulações computacionais serão geradas conforme demanda do laboratório.

2 - Metodologia

Após a formação do grupo, destacamos as principais tarefas a serem cumpridas para a realização do trabalho prático. Como:

- Implementação da entrada por arquivos de texto.
- Desenvolvimento de funções simples.
- Pesquisas sobre os algoritmos a serem usados.
- Implementação da fórmula dos cossenos e algoritmos.
- Criação das funções necessárias para o funcionamento do cálculo da similaridade entre pares.
- Simulações com todos os algoritmos.
- Interatividade com o usuário.
- Funções extras, testes e gráficos.
- Makefile e documentação.

O código fonte foi desenvolvido em **C**[1] e versionado no **GitHub**[2], visando que seria a melhor forma de compartilhamento do mesmo entre os integrantes do grupo. Para uma melhor organização e visualização do projeto, este foi dividido em subpastas.

- /src** - implementação dos arquivos **.c** e **.h**.
- /files** - arquivos **.txt** com os DNA a ser analisado.

Para compilar e executar o projeto, é necessário ter [gcc](#) e [make](#) instalado em sua máquina.

Comandos:

-make

Caso não funcione, tente:

- gcc -o main src/main.c src/similarity.c src/similarity.h src/file.c src/file.h src/patternMatching.c src/patternMatching.h -lm

./main

3 - Desenvolvimento

Diversas funções foram criadas, organizadas na pasta /src dentre diversos arquivos fontes e cabeçalho, todas estão devidamente comentadas e referenciadas.

3.1 - file.c e file.h

Funções (**figura 01**) necessárias para a leitura de arquivos de texto.

```
bool readFileIntoArray(int n, char vector[n], char *filename);
```

Figura 01 - file.h.

- ***bool readFileIntoArray(int n, char vector[n], char *filename);***
 - ***função:*** ler os arquivos determinados pela especificação e inseri-los em um vetor.
 - ***parâmetros:***
 - ***n:*** quantidade de registros no arquivo.
 - ***vector[n]:*** vetor com o tamanho de registros.
 - ***filename:*** nome do arquivo.
 - ***retorno:*** se a função foi realizada com sucesso.
 - ***detalhamento:*** todos os registros dos arquivos humano.txt, chimpanze.txt e cachorro.txt, são inseridos, um em cada vetor. Serão posteriormente usados para o cálculo da similaridade.

3.2 - similarity.c e similarity.h

Nesses arquivos, as funções (**figura 02**) necessárias para a elaboração da similaridade foram implementadas.

```
double calculateSimilarity(const int *vectorA, const int *vectorB, int n);  
  
void initializeCartesianProductMatrix(int qtyOfCombinations, int size, char matrix[qtyOfCombinations][size]);  
  
int getVectorSizeForCartesianProduct(int n);
```

Figura 02 - similarity.h.

- ***double calculateSimilarity(const int *vectorA, const int *vectorB, int n);***
 - **função:** calcular a similaridade entre dois vetores usando a fórmula de similaridade por cossenos.
 - **parâmetros:**
 - **vectorA:** ponteiro para o primeiro vetor.
 - **vectorB:** ponteiro para o segundo vetor.
 - **n:** tamanho do vetor
 - **retorno:** valor da similaridade.
 - **detalhamento:** usa a fórmula descrita na especificação deste trabalho.
- ***void initializeCartesianProductMatrix(int qtyOfCombinations, int size, char matrix[qtyOfCombinations][size]);***
 - **função:** define a matriz do produto cartesiano com todas as combinações possíveis do tamanho dado.
 - **parâmetros:**
 - **qtyOfCombinations:** quantidade de combinações
 - **size:** tamanho das combinações
 - **matrix:** matriz a ser preenchida.
 - **retorno:** não possui
 - **detalhamento:** para um dado conjunto de tamanho n, haverá n^k sequências possíveis de comprimento k. A ideia é começar a partir de uma string de saída vazia (nós a chamamos de prefixo no código a seguir). Um por um, adicione todos os caracteres ao prefixo (**figura 03**). Para cada caractere adicionado, imprima todas as strings possíveis com o prefixo atual chamando recursivamente igual a k-1. Disponível em [3].

```
// Store all possible strings of length k in array that can be formed from a set of n character using DNA array
int n = (int) strlen(DNA);
int array[size - 1];

// Initialize array with first k character of DNA
for (int i = 0; i < size - 1; i++)
{
    array[i] = 0;
}

// One by one print all sequences
for (int i = 0; i < qtyOfCombinations; i++)
{
    matrix[i][size - 1] = '\0';

    // Print current combination
    for (int j = 0; j < size - 1; j++)
    {
        matrix[i][j] = DNA[array[j]];
    }

    // Find the rightmost character which is not DNA.length - 1 and increment its value
    int next = size - 2;

    while (next >= 0 && (array[next] + 1 >= n))
    {
        next--;
    }
}
```

Figura 03 - parte da implementação, comentada, da função descrita acima.

- ***void getVectorSizeForCartesianProduct(int n);***
 - ***função:*** obtém o tamanho do vetor de produto cartesiano.
 - ***parâmetros:***
 - ***n:*** tamanho variável
 - ***retorno:*** o tamanho do vetor do produto cartesiano.

3.3 - patternMatching.c e patternMatching.h

Nestes arquivos (**figura 04**), os algoritmos necessários para verificar a correspondência de padrões foram implementados.

```
int getNumberOfPatternMatchingBoyerMooreAlgorithm(char *text, char *pattern);

int getNumberOfPatternMatchingShiftAndAlgorithm(char *text, char *pattern);

int getNumberOfPatternMatchingKnuthMorrisPrattAlgorithm(char *text, char *pattern);

int getNumberOfPatternMatching(char *text, char *pattern, int (*algorithm)(char *, char *));
```

Figura 4 - patternMatching.h

- ***int getNumberOfPatternMatching(char *text, char *pattern, int (*algorithm)(char *, char *));***
 - ***função:*** obtém o número de correspondência de padrões usando um algoritmo passado.
 - ***parâmetros:***
 - ***text:*** ponteiro para um texto.
 - ***pattern:*** ponteiro para um padrão
 - ***algorithm:*** algoritmo a ser usado
 - ***retorno:*** número de correspondências.
 - ***detalhamento:*** função geral para o cálculo.

- ***int getNumberOfPatternMatchingBoyerMooreAlgorithm(char *text, char *pattern);***
 - ***função:*** obtém o número de correspondência de padrões usando o algoritmo Boyer-Moore.
 - ***parâmetros:***
 - ***text:*** ponteiro para um texto.
 - ***pattern:*** ponteiro para um padrão.
 - ***retorno:*** número de correspondências.
 - ***detalhamento:*** ver [4].

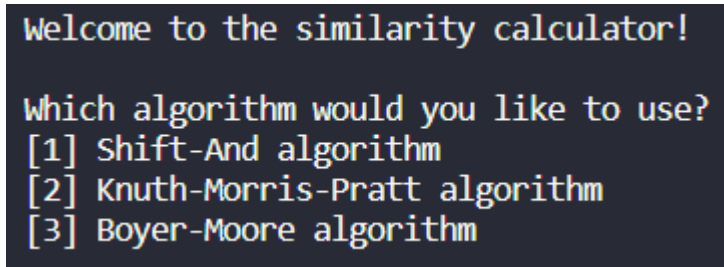
- ***int getNumberOfPatternMatchingKnuthMorrisPrattAlgorithm(char *text, char *pattern);***
 - ***função:*** obtém o número de correspondência de padrões usando o algoritmo Knuth-Morris-Pratt.
 - ***parâmetros:***
 - ***text:*** ponteiro para um texto.
 - ***pattern:*** ponteiro para um padrão.
 - ***retorno:*** número de correspondências.
 - ***detalhamento:*** ver [5].

- ***int getNumberOfPatternShiftAndAlgorithm(char *text, char *pattern);***
 - ***função:*** obtém o número de correspondência de padrões usando o algoritmo Shift-And.
 - ***parâmetros:***
 - ***text:*** ponteiro para um texto.
 - ***pattern:*** ponteiro para um padrão.
 - ***retorno:*** número de correspondências.
 - ***detalhamento:*** ver [6].

3.4 - main.c

Programa principal do projeto.

Primeiramente, os **arquivos são lidos** e inseridos em vetores estaticamente alocados e um menu (**figura 05**) será impresso na tela e o usuário deverá escolher uma das opções.



```
Welcome to the similarity calculator!

Which algorithm would you like to use?
[1] Shift-And algorithm
[2] Knuth-Morris-Pratt algorithm
[3] Boyer-Moore algorithm
```

Figura 05 - Menu do programa

1 - Shift-And algorithm.

Selecionando essa opção, o programa faz o cálculo da similaridade e simulações usando o algoritmo Shift-And.

2 - Knuth-Morris-Pratt algorithm

Selecionando essa opção, o programa faz o cálculo da similaridade e simulações usando o algoritmo Knuth-Morris-Pratt.

3 - Boyer-Moore algorithm

Selecionando essa opção, o programa faz o cálculo da similaridade e simulações usando o algoritmo Boyer-Moore.

Após isso, o **produto cartesiano é calculado**, por padrão, o número de caracteres é 2 e o produto cartesiano, obviamente, 16 possibilidades. Depois, o **número de elementos do conjunto é escolhido aleatoriamente**, entre 1 e o número de possibilidades.

Por fim, os cálculos são feitos e a similaridade é impressa no terminal (**figura 06**)

```
1
Number of characters: 2
Quantity of combinations: 16
Number of elements to choose: 4

Similarity between human and chimp: 0.985854
Similarity between human and dog: 0.964731
Similarity between chimp and dog: 0.974024
```

Figura 6 - similaridade usando Shift-And com o número de elementos igual a 4.

Antes de encerrar, o usuário pode escolher fazer uma simulação. Escolhendo pelo menu, a simulação funciona da seguinte forma:

É executado o passo anterior X vezes e é calculado a média da soma de cada valor de similaridade. **Exemplo:** se o usuário escolheu o algoritmo Boyer-Moore e o número de elementos 4. **A simulação executa 1000 vezes (padrão do programa)** para cada par de DNA. Ao fim, a similaridade média é calculada juntamente com o tempo gasto e ambos são impressos no terminal (**figura 07**).

```
Do you want to run the simulation? (y/n)
y

Average similarity between human and chimp: 0.965090
Average similarity between human and dog: 0.931108
Average similarity between chimp and dog: 0.976841

Simulation size: 1000
Time spent: 0.354001 seconds
```

Figura 7 - simulação da figura 07.

5 - Resultados

A seguir, algumas simulações foram feitas e registradas em tabelas:

Simulação 1
<ul style="list-style-type: none">• algoritmo: Shift-And• caracteres por padrão: 2• elementos por conjunto: 9• tamanho da simulação: 1000 vezes• tempo de execução: 0.933837 seconds
<p><i>Average similarity between human and chimp: 0.961716</i> <i>Average similarity between human and dog: 0.927330</i> <i>Average similarity between chimp and dog: 0.972822</i></p>

Simulação 2
<ul style="list-style-type: none">• algoritmo: Shift-And• caracteres por padrão: 4• elementos por conjunto: 246• tamanho da simulação: 1000 vezes• tempo de execução: 29.917605 seconds
<p><i>Average similarity between human and chimp: 0.811644</i> <i>Average similarity between human and dog: 0.705439</i> <i>Average similarity between chimp and dog: 0.869128</i></p>

Simulação 3
<ul style="list-style-type: none">• algoritmo: Knuth-Morris-Pratt• caracteres por padrão: 2• elementos por conjunto: 9• tamanho da simulação: 5000 vezes• tempo de execução: 7.958911 seconds
<p><i>Average similarity between human and chimp: 0.961520</i> <i>Average similarity between human and dog: 0.926065</i> <i>Average similarity between chimp and dog: 0.973021</i></p>

Simulação 4

- **algoritmo:** Knuth-Morris-Pratt
- **caracteres por padrão:** 3
- **elementos por conjunto:** 22
- **tamanho da simulação:** 5000 vezes
- **tempo de execução:** 20.359843 seconds

Average similarity between human and chimp: 0.911821

Average similarity between human and dog: 0.845600

Average similarity between chimp and dog: 0.944700

Simulação 5

- **algoritmo:** Boyer-Moore
- **caracteres por padrão:** 2
- **elementos por conjunto:** 13
- **tamanho da simulação:** 5000 vezes
- **tempo de execução:** 14.616054 seconds

Average similarity between human and chimp: 0.960711

Average similarity between human and dog: 0.925805

Average similarity between chimp and dog: 0.972426

Simulação 6

- **algoritmo:** Boyer-Moore
- **caracteres por padrão:** 3
- **elementos por conjunto:** 5
- **tamanho da simulação:** 1000 vezes
- **tempo de execução:** 0.790773 seconds

Average similarity between human and chimp: 0.926662

Average similarity between human and dog: 0.867404

Average similarity between chimp and dog: 0.952148

Observações:

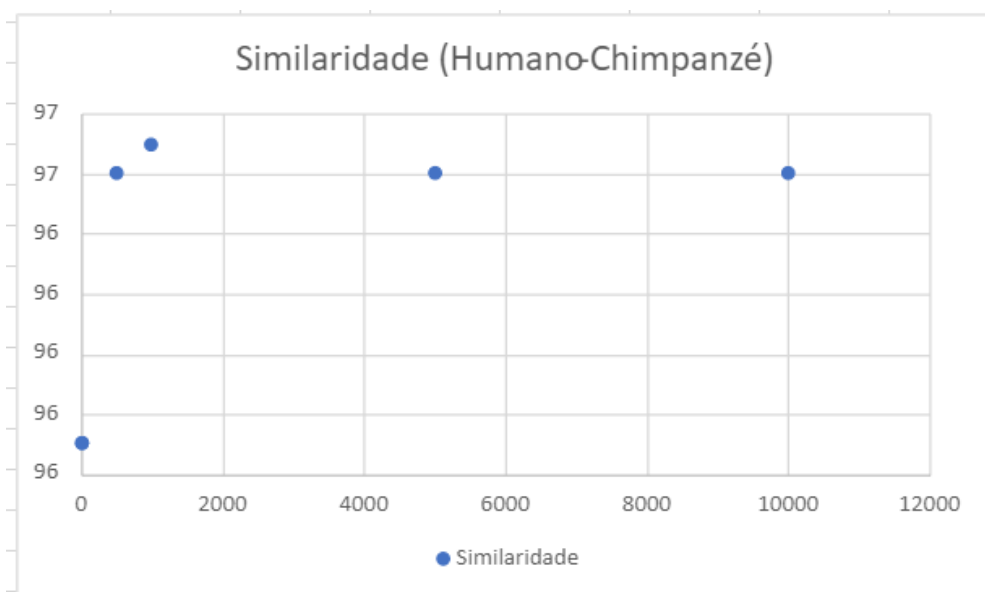
- Foram testados centenas de vezes, para caracteres acima de 4 é inviável, pois o tempo gasto para calcular as semelhanças cresce exponencialmente, devido ao produto cartesiano.
- O tempo medido é apenas na execução da simulação.
- Não houve muitas discrepâncias nos resultados.

Outros testes foram realizados e gráficos foram gerados pelo **Excel**:

- **Shift - And**
- **Configurações:** padrão: 2 | conjunto: 4

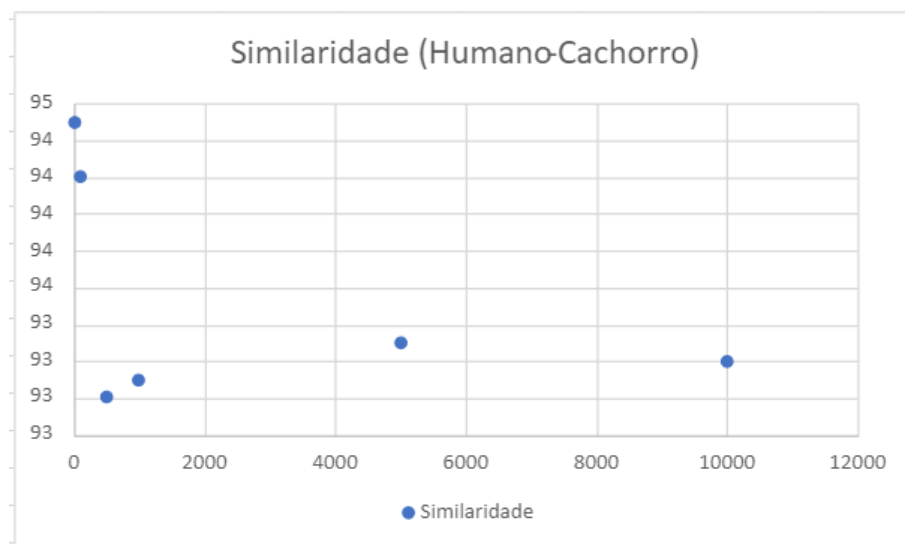
HUMANO - CHIMPANZÉ

Entrada	Similaridade
100	96
500	96,6
1000	96,7
5000	96,6
10000	96,6
10	95,7



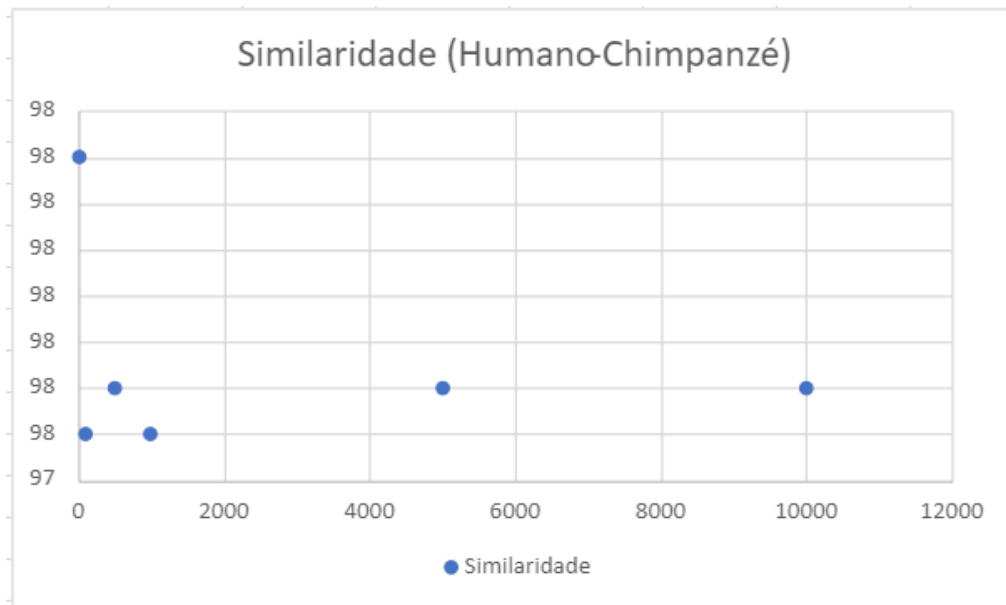
HUMANO - CACHORRO

Entrada	Similaridade
10	95
100	94,2
500	93
1000	93,1
5000	93,3
10000	93,2



CACHORRO - CHIMPANZÉ

Entrada	Similaridade
10	98
100	97,5
500	97,6
1000	97,5
5000	97,6
10000	97,6



Observações:

- Foram testados com os outros algoritmos, além do Shift-And, mas os resultados foram muito semelhantes, obviamente.
- **Verifica-se que, as melhores entradas para simulação são de 1000 para cima, fazer simulações com números baixos não faz sentido pois varia muito a média de cálculo. Simulações acima de 10000 não alteram muito o resultado.**
- Foram testadas com outras configurações de entrada, número de elementos no padrão e quantidade no conjunto e verificou-se também que para simulações >1000 os resultados ficam mais concisos e próximos da realidade.

Por fim, qual o melhor algoritmo?

Configurações:

padrão - 2

conjunto - 4

simulação - 10000 (dez mil)

SHIFT - AND: 2,4244394 seconds

KNUTH-MORRIS-PRATT: 4,0888188 seconds

BOYER-MOORE: 5,5043333 seconds

Configurações:

padrão - 2

conjunto - 8

simulação - 10000 (dez mil)

SHIFT - AND: 7.088935 seconds

KNUTH-MORRIS-PRATT: 8.937544 seconds

BOYER-MOORE: 11.898892 seconds

Configurações:

padrão - 2

conjunto - 16

simulação - 10000 (dez mil)

SHIFT - AND: 10.952765 seconds

KNUTH-MORRIS-PRATT: 21.958317 seconds

BOYER-MOORE: 28.483999 seconds

Configurações:

padrão - 3

conjunto - 6

simulação - 10000 (dez mil)

SHIFT - AND: 2.864154 seconds

KNUTH-MORRIS-PRATT: 6.394344 seconds

BOYER-MOORE: 6.647568 seconds

SHIFT - AND: 10.952765 seconds

KNUTH-MORRIS-PRATT: 21.958317 seconds

BOYER-MOORE: 28.483999 seconds

Configurações:

padrão - 3

conjunto - 20

simulação - 10000 (dez mil)

SHIFT - AND: 11.015149 seconds

KNUTH-MORRIS-PRATT: 21.543509 second

BOYER-MOORE: 22.673978 seconds

Ao final constata-se que o algoritmo que desempenhou melhor foi o SHIFT - AND.

5 - Conclusão

Após o fim do trabalho prático podemos apontar certas dificuldades e facilidades encaradas pelo grupo durante a implementação desse projeto.

Notamos uma certa facilidade na hora de entender como o problema deveria ser resolvido.

Tivemos dificuldades de implementar algumas funções relacionadas aos algoritmos.

Os resultados obtidos, foram úteis na compreensão de como um algoritmo de casamento de padrões funciona e como simular essas combinações.

Por fim é visível o proveito e as lições aprendidas durante esse período de desenvolvimento do trabalho prático.

6 - Referências

[1] [C \(programming language\) - Wikipedia](#)

[2] <https://github.com/Mateus-Henr/PairSimilarity>

[3] [Print all possible strings of length k that can be formed from a set of n characters - GeeksforGeeks](#)

[4] [Boyer Moore Algorithm for Pattern Searching - GeeksforGeeks](#)

[5] [Shift-AND algorithm for exact pattern matching](#)

[6] [KMP Algorithm for Pattern Searching - GeeksforGeeks](#)

