



Universidade Federal do Espírito Santo

CENTRO UNIVERSITÁRIO NORTE DO ESPÍRITO SANTO – CEUNES

DEPARTAMENTO DE COMPUTAÇÃO e ELETRÔNICA – DCE

Engenharia da Computação

Disciplina: ESTRUTURA DE DADOS II

LUCAS RANIERE NEVES SILVA

MATHEUS COUTO MARCARINI

PAULO ROBERTO DE JESUS GONÇALVES

RELATÓRIO DO TRABALHO PRÁTICO

SÃO MATEUS-ES

03 DE DEZEMBRO DE 2023

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

Lucas Raniere Neves Silva

Matheus Couto Marcarini

Paulo Roberto de Jesus Gonçalves

TRABALHO PRÁTICO

Trabalho valendo 20% da nota para o
Componente Curricular: Estrutura De
Dados II, ministrada pela professora
orientadora Luciana Lee.

São Mateus, ES

Novembro, 2023

Sumário

1. Introdução	04
2. Objetivos	05
3. Metodologia	06
1. Main.c	
2. Funções de suporte	
3. Funções auxiliares	
4. Utils.c	
5. Funções Principais	
6. Utils.h	
7. Makefile	
4. Resultados	20
5. Conclusão	23
6. Bibliografia	24
7. Apêndices	25

1 Introdução

Este trabalho consiste na implementação de um algoritmo para a busca da correspondência de cadeias de caracteres de uma ocorrência de um determinado vírus em uma espécie de animais. A espécie de animais é investigada através do DNA do hospedeiro.

A construção de uma estrutura capaz de identificar corretamente as posições de ocorrência na busca de um padrão é realizada usando a estratégia de Correspondência de Strings por meio do algoritmo de Knuth-Morris-Pratt. Este algoritmo possui um tempo de pré-processamento $\Theta(m)$, com m sendo o comprimento do padrão, e um tempo de emparelhamento $\Theta(n)$, com n sendo o comprimento do texto. O algoritmo é referenciado como “muito mais inteligente” pelo autor Thomas H. Cormen, por ser assintoticamente melhor do que outros algoritmos de correspondência de cadeias com a mesma finalidade.

Além disso, todos os dados das estruturas implementadas deverão estar armazenados em arquivos ".txt" por meio de uma abordagem de Sistema de Arquivos em C, considerada “extremamente poderosa e flexível” pelo autor Herbert Schildt.

A implementação foi feita na linguagem C e a forma de compilar é feita a partir de um arquivo Makefile.

2 Objetivos

O objetivo deste trabalho é criar um programa que facilite a análise do comportamento do vírus nos genes dos hospedeiros, proporcionando maior eficiência. Ao receber arquivos com sequências de DNA de animais e variantes do vírus, o programa visa verificar o número de ocorrências de padrões específicos, facilitando o estudo do comportamento das variantes. A justificativa para esse trabalho reside na necessidade de implementar um sistema com um algoritmo de resposta linear e eficiente, contribuindo para que o Dr. Manhattan tenha maior controle sobre os dados coletados na floresta.

3 Metodologia

O grupo adotou uma abordagem combinada para desenvolver a metodologia do trabalho. Inicialmente, definimos horários em reuniões dedicadas à elaboração do projeto. Em seguida, nos dedicamos à compreensão do desafio de implementar um algoritmo para buscar padrões em sequências de DNA, e por fim através de pesquisas aplicadas ao contexto conseguimos implementar uma solução eficiente que solucionasse a problemática. Esta implementação será descrita pelos trechos a seguir:

3.1 Arquivo Main.c

Essa é a função principal do programa, a qual é responsável pelo gerenciamento de toda a estrutura de dados. Nela será feita a abertura de arquivos, a inicialização das variáveis, a chamada das funções para realizar leitura de padrões de vírus e busca das sequências de DNA, e também o fechamento de arquivos que foram usados.

Arquivos-fonte

```
#include "utils.h"
```

Esta diretiva instrui o compilador para a inclusão do arquivo utils.h criado para o projeto.

Funções de suporte

```
FILE *arquivoVirus = fopen("PadroesVirus.txt", "r");  
FILE *arquivoDNAs = fopen("BaseDadosDNA.txt", "r");
```

A função **fopen** é utilizada para abrir os arquivo-textos PadroesVirus.txt e BaseDadosDNA.txt, permitindo leitura. Além de abrir para uso ela também associa um arquivo as streams declaradas.

```
if (arquivoVirus == NULL || arquivoDNAs == NULL) {  
    perror("Erro ao abrir o arquivo");  
    return 1;  
}
```

É feita uma verificação para certificar-se que não houve erro algum durante a abertura dos arquivos. Caso haja é retornado o valor 1 ao terminal durante a execução.

```
//Para cada vírus procura a ocorrência
while(lerPadraoVirus(arquivoVirus, padrao, nomeVirus) != FIM_DE_ARQUIVO_VIRUS) {
```

O laço while é usado para ocorrer repetição do bloco de comandos enquanto a condição de fim de arquivo do vírus for falsa, assim garantindo que continue a buscar a ocorrência enquanto houver caracteres a serem combinados pelo “arquivoVirus”.

```
//Copia o nome do padrão que esta lendo atualmente
pos++;
pos = copiaNome(buffer, nomeDna, pos, arquivoDNAs, &tamanho);
//o tamanho é usado para o offset da busca então deve ser o tamanho do buffer desde o inicio
//do padrão
tamanho=tamanho-pos;
calcularLPS(padrao, strlen(padrao), lps);
printf("%s", nomeVirus);
```

Após ler o padrão de vírus e o nome do primeiro DNA no arquivo calcula o vetor de prefixos e exibe o nome do vírus para que a exibição das ocorrências apareçam após isso.

A variável “pos” recebe os parâmetros da copiaNome passando alguns parâmetros, incluindo um buffer , um “nomeDna”, a “pos”, e o “arquivoDNAs”, e um ponteiro para a variável tamanho. A função copiaNome copia parte do conteúdo do buffer para o nome, e então atualiza a variável pos, enquanto a função calcularLPS constrói a tabela LPS com os dados enviados do padrão dos vírus e o tamanho, assim para ser utilizada no combinador kKMP, e por fim o array do maior prefixo que também é sufixo, e imprime as ocorrências dos vírus na tela.

```

while (strcmp(nomeDna, "EOF") != 0) {

    res = buscarKMP(buffer, lps, padrao, &pos, offset, &ocorrencias, posicoes, &auxPosicoes);

    if (res == LEITURA_PADRAO_COMPLETA) {

        //tira o \n do nome
        if (ocorrencias != 0) {
            printf("[%s] no. de ocorrencias: %d posicoes: ", nomeDna, ocorrencias);
            for (int i = 0; i < auxPosicoes; i++) {
                printf("%d ", posicoes[i]);
            }
            printf("\n");
        }

        auxPosicoes = 0;
        ocorrencias = 0;

        pos++;

        pos = copiaNome(buffer, nomeDna, pos, arquivoDNAs, &tamanho);

        offset = 0;
        tamanho = tamanho-pos;

    } else if (res == LEITURA_PADRAO_INCOMPLETA) {

        int deslocamento = (strlen(padrao))*-1 +1 ;

        offset += deslocamento + tamanho;

        fseek(arquivoDNAs, deslocamento, SEEK_CUR);

        tamanho = lerProximos(buffer, arquivoDNAs);

        pos = 0;

    }

}

```

Função **lerProximos** que eventualmente serve para ler um bloco de dados do arquivo de DNA.

Também foi criado a variável “res” que basicamente armazena o resultado da busca efetuada pelo algoritmo KMP, este por sua vez é usado para buscar o padrão de vírus no bloco de dados “buffer” sendo representado na implementação como a função **buscaKMP** que será abordada posteriormente neste relatório.

Quando “res” se encontra no estado de leitura completa isso quer dizer que o KMP encontrou o caractere ‘>’, e este bloco é responsável por efetuar a exposição das ocorrências na tela do terminal. A função **strlen** é usada para devolver o comprimento da string terminada por um nulo, para esta situação em especial será

feito a remoção do caractere “\n” que indica uma quebra de linha no final do nome do DNA.

O comando de seleção if verifica se há ocorrências do padrão no bloco de dados, logo o valor será diferente de zero se houver, neste caso deve ser impresso as informações associadas a essas ocorrências, como nome do DNA, número de ocorrências e posições.

Ao atribuir zero as variáveis “auxPosicoes” e “ocorrencias” é realizado a reinicialização das variáveis relacionadas a posições e ocorrências no programa.

A próxima função fgets será para ler a string do próximo nome do DNA para o próximo bloco de dados até que um caractere de nova linha seja lido ou que tamanho(35-1) caracteres tenham sido lidos.

Após isto temos atualização da variável “offset” para a posição inicial que era zero. E tamanho recebendo (**tamanho – pos**) simbolizando a quantidade de caracteres que ainda precisa ser considerada na busca pelo padrão de vírus no restante do bloco

Por fim o último trecho indica que caso “res” esteja no estado de leitura do padrão incompleta que ocorre quando o algoritmo KMP termina a leitura sem encontrar o caractere '>', indicando que a leitura do padrão é incompleta, então teremos os seguintes procedimentos:

Criação de variável “deslocamento” para receber o valor com base no comprimento do padrão do vírus. Faz uso da função strlen para retornar o comprimento da string armazenada na variável “padrão”, invertendo o sinal do comprimento do padrão e depois adicionando +1 ao resultado para compensar o deslocamento anterior, uma vez que a meta é retroceder no texto.

Depois é feita a atualização da variável “offset” para refletir o deslocamento total necessário para retomar a busca na próxima leitura. Nesta etapa “tamanho” representa o número de caracteres lidos no último bloco de dados.

É preciso utilizar a função fseek para mover o ponteiro do arquivo de acordo o deslocamento para trás, permitindo a leitura do próximo bloco começando no início do padrão.

Ao fim é chamada a função **lerProximos** empregada na leitura do próximo bloco de dados do arquivo de DNA e “tamanho” armazenará a quantidade de caracteres lidos. E com isto concluído é reiniciado a variável “pos” para zero, para que ela possa ser

usada como posição durante uma nova busca subsequente. Então o laço while continua até que todo o arquivo de DNA seja processado, quando acaba então sai do loop de repetição.

```
fseek(arquivoDNAs, 0, SEEK_SET);  
tamanho = lerProximos(buffer, arquivoDNAs);  
pos = 0;  
printf("\n");
```

Depois é realizado o deslocamento do indicador de posição de arquivo para o início do arquivoDNAs, também é feito tamanho receber o resultado da leitura do próximo bloco de dados através da função **lerProximos** e reiniciado a variável “pos” para outra busca subsequente, além da impressão de uma linha em branco para separar as informações impressas anteriormente das informações da próxima iteração.

```
fclose(arquivoDNAs);  
fclose(arquivoVirus);
```

A função **fclose** é utilizada para liberar o descritor do arquivo DNAs para que ele possa ser reutilizado com outro arquivo, também é feito a liberação do descrito do arquivo Virus para ser reutilizado com outro arquivo, logo são fechados por não serem mais necessários no programa.

3.4 Utils.c

Nesta seção, serão analisados os códigos das funções principais que são responsáveis por realizar as tarefas necessárias para a correspondência de strings dos arquivos.

```
#define TAMANHO_BUFFER 50000 //O tamanho desse buffer deve ser maior que o maior padrão de vírus
```

Usando o **#define** podemos definir o identificador (nome da macro) como TAMANHO_BUFFER, e o valor de 50000 como o tamanho máximo do buffer reservado para a leitura do arquivo do DNA por vez, caso ele seja maior o programa deverá realizar mais leituras.

3.5 Funções Principais

```

void calcularLPS(char *padrao, int m, int lps[]) {
    int len = 0; // Comprimento do prefixo atual
    int i = 1;

    lps[0] = 0; // O primeiro caractere sempre tem LPS igual a zero


    while (i < m) {
        if (padrao[i] == padrao[len]) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}

```

A função “**void calcularLPS(char *padrao, int m, int lps[]){**” é a função prefixo do programa, ela encapsula o conhecimento sobre o modo como o padrão se compara com os deslocamentos dele próprio. Esta função possui 3 parâmetros, sendo o “padrão” passado como uma chamada por referência, “m” o tamanho do comprimento do padrão e “lps[]” o comprimento do prefixo mais longo do Padrão que é um sufixo próprio do Padrão.

A partir destas variáveis deve ser gerado uma variável “len” para receber o comprimento do prefixo atual, e inicializado o “lps” na primeira posição valendo zero. A variável “i” é uma contadora e se iniciará em 1 para realizar as verificações adequadas no padrão.

Nesta implementação temos que o calculo da função prefixo é realizado dentro do laço while de repetição que tem como condição; Caso o contador “i” seja menor do que o tamanho do padrão “m”, ou seja há mais de 1 elemento no array, logo então executará os seguintes blocos de comando;

- O padrão na posição de “i” é equivalente ao padrão na posição “len”, a princípio é o mesmo que  padrão[1] = padrao[0]? Caso o bloco de seleção seja verdadeiro isso quer dizer que o primeiro caractere e o segundo

caractere do padrão são idênticos, pois possuem o mesmo valor. Com isso é possível entrar no bloco de comandos do **if**, e lá temos a incrementação da variável “len”, a atribuição do valor novo de “len” ao vetor “lps[na posição ”i”], e por fim o contador “i” também é incrementado.

- Caso contrário, o bloco de comandos do corpo do **else** é executado.

E isto significa que o “padrão[i]” é diferente do “padrão[len]”, neste caso é feito um novo comando de seleção para verificar se “len” está no início, se a expressão for falsa então quer dizer que len é diferente de zero, assim len armazenará a informação do lps[len-1] ou seja, o valor do comprimento do prefixo atual deve ser repassados para ele, pois não teve match entre aquelas duas posições, onde len estava em algum lugar da tabela, mas o padrão parou de se repetir. Caso o comando de seleção contenha uma expressão verdadeira então quer dizer que “len” estará no início, logo a situação é diferente, deverá ser atribuído ao “lps[na posição corrente de “i”]” o valor de zero, pois isto quer dizer que ocorreu um descasamento entre as duas posições verificadas, e a variável “len” era 0, então o valor do “lps[i]” não pode ser diferente de 0, afinal não teve match com a primeira string da tabela naquela posição de “i”. e conclui-se com “i” incrementando ao fim desta situação.

Após todas essas verificações o laço retorna a condição inicial. Então o while é executado até que a condição de “i” ser do mesmo tamanho do padrão “m” ocorra, e quando isto ocorre a condição estabelecida passa a ser falsa e se encerra a função.

```
int buscarKMP(char *texto, int lps[], char *padrao, int *pos, int offsetPos, int *ocorrencias, int *posicoes, int *auxPosicoes) {
    int m = strlen(padrao);
    int n = strlen(texto);

    int posinicial = *pos;
    int i = *pos; // Índice para o texto
    int j = 0; // Índice para o padrão
    int deslocamento = 0;
    while (i < n) {
        if (texto[i] == '\n') {
            i++;
        }
        if (texto[i] == '>') {
            //Retorna a posicao em que acabou um padrão para que a próxima leitura
            //do arquivo parta dessa posicao
            *pos = i;
            return LEITURA_PADRAO_COMPLETA;
        }

        if (padrao[j] == texto[i]) {
            i++;
            j++;
        }
    }
}
```

```

    if (j == m) {
        (*ocorrencias)+=1;
        deslocamento = i-posinicial;
        posicoes[*auxPosicoes] = deslocamento - j + offsetPos - (deslocamento - j + offsetPos)/69;
        (*auxPosicoes)+=1;
        j = lps[j - 1];
    } else if (i < n && padrao[j] != texto[i] && texto[i] != '\n') {
        if (j != 0) {
            j = lps[j - 1];
        } else {
            i++;
        }
    }
}

*pos = i;
return LEITURA_PADRAO_INCOMPLETA;
}

```

A função “**int buscarKMP(char *texto, char *padrao, int *pos, int offsetPos, int *ocorrencias, int *posicoes, int *auxPosicoes)**” é o algoritmo de emparelhamento de Knuth-Morris-Pratt adaptado para a situação especificada do trabalho. Este algoritmo trabalha de modo eficiente, pois recebe as informações da função prefixo, assim realizando um tempo de emparelhamento de $\Theta(n)$. O algoritmo KMP é utilizado para encontrar todas as ocorrências de um padrão em um texto, mantendo um registro das posições onde os padrões são encontrados. A ideia básica é que uma vez que conhecemos o padrão P a ser buscado, quando um “descasamento” é identificado, pode não ser necessário retroceder a verificação desde o início de P.

A função começará recebendo 7 parâmetros que serão utilizados na implementação, ela inicializa algumas variáveis auxiliares, como m que está a receber o tamanho do padrão, e n que esta a receber o tamanho do texto, por meio da **strlen**. Também criar um array “lps[m]” que armazenará o maior prefixo-sufixo do padrão. Além disto cria uma variável “posInicial” para receber a “pos” que é passada como argumento na main. Após estas inserções, é realizado o calculo do LPS do padrão, e termina atribuindo a “pos” a uma variável contadora “i” e j recebendo zero, assim como o deslocamento a ser efetuado.

Após isto temos a entrada em um laço **while** que garante que será percorrido o texto enquanto a posição “i” for menor do que o comprimento do texto “n”, esta etapa é o

casamento das cadeias, e as seguintes expressões deveram ser conferidas para realizar a combinação correta de acordo com suas condições:

- Dentro dos blocos de comandos é verificado a ocorrência do tratamento de caracteres especiais, onde o comando de seleção **if** verifica se o caractere atual for um '\n', ou seja uma quebra de linha. Caso isto ocorra deve avançar o contador "i" para o próximo.
- Logo a seguir é verificado através de outro comando de seleção **if** se o caractere atual é um '>', porque isto indica o final de um padrão, e neste caso o bloco de comandos fará o ponteiro "pos" receber o valor do contador "i", para atualizar a posição para o próximo caractere e retornar este estado como Leitura_Padrao_Completa, pois foi verificado o DNA de um animal por completo.
- Posteriormente temos a **comparação** da posição "j" do padrão com a posição "i" do padrão, neste comando de seleção temos que se a informação for verdadeira então temos que os caracteres do padrão e do texto são iguais, avançando os índices.
- Após isto, é verificado outro comando de seleção que cuja a expressão aborda a sentença de j ser equivalente a m, caso seja verdadeiro isto, significa que houve uma ocorrência do padrão no texto, portanto j alcançou o comprimento do padrão, assim o padrão foi encontrado, assim o bloco de comandos atualiza o contador de ocorrências em +1, calcula o deslocamento e armazena a posição no array "posicoes", e, por fim, atualiza o índice "j" usando o vetor LPS para salvar o último maior prefixo-sufixo da última ocorrência para o caso de encontrar um padrão se iniciando no durante outro.
- Uma estrutura de controle de fluxo **else if** é usada a fim de avaliar o deslocamento, para isso a seguinte expressão deve se confirmar: contador "i" ainda não ter alcançado o final do texto, o padrão na posição "j" ser diferente do texto na posição "i" e que o texto na posição "i" não seja uma quebra de linha, uma vez que elas devem ser tratadas como caracteres especiais e não pode ser usada como parâmetro para deslocamento. Assim o bloco de comando desta estrutura fará uma nova verificação:
 - Caso o contador "j" seja diferente de zero, deve ser avançado no texto o j,

de forma que ele receberá o array do maior prefixo-sufixo na posição “j – 1”, com isto ele receberá a última informação deste comprimento registrada.

- Caso contrário, o valor de i incrementa em um.

Por fim ao finalizar a condição desse laço while, teremos feito o contador “i” percorrer por todo o texto do arquivo passado, e assim concluímos a função atualização a posição “pos” para o valor do contador de “i” e retornando o estado de LEITURA_PADRAO_INCOMPLETA, para indicar que a leitura do padrão ainda não foi concluída.

```
int lerPadraoVirus(FILE *arquivoVirus, char *padrao, char *nomeVirus) {
    fseek(arquivoVirus, 1, SEEK_CUR);
    fgets(nomeVirus, sizeof(char) * 35, arquivoVirus);
    if (strcmp(nomeVirus, "EOF") == 0) {
        return FIM_DE_ARQUIVO_VIRUS;
    }
    fgets(padrao, sizeof(char) * 75, arquivoVirus);
    padrao[strlen(padrao)-1] = '\0';
    return LEITURA_VIRUS_SUCESSO;
}
```

A função “**int lerPadraoVirus(FILE *arquivoVirus, char *padrao, char *nomeVirus){**” é responsável por ler informações sobre um vírus de um arquivo.

Utiliza a função **fseek** para mover o ponteiro de posição do “*arquivoVirus” para a frente em 1 byte a partir da posição atual. Isso pode ser feito para pular algum tipo de delimitador ou caractere de controle no início do registro de vírus. a função **fgets** é utilizada para ler até 35 caracteres do arquivoVirus e armazená-los na string “nomeVirus”. Essa operação parece ser destinada à leitura do nome do vírus. A seguir temos um comando de seleção que por meio da função **strcmp** verifica se o final do arquivo foi alcançado, basicamente observando se encontrou o EOF no “nomeVirus”, quando isto ocorre a função retorna o estado FIM_DE_ARQUIVO_VIRUS. Em uma outra linha é chamado a fgets para lê as informações do arquivoVirus que deve conter até 70 caracteres e faz o armazenamento na string padrão. Por fim é removido o ‘\n’ do final da string “padrao” e adicionado o caractere nulo no lugar de modo a garantir que a string termine corretamente. Ao final do bloco de construção é praticado o retorno do estado de LEITURA_VIRUS_SUCESSO para indicar houve sucesso na leitura do vírus.

```

int copiaNome(char *buffer, char *nomeDest, int pos, FILE *arquivoDna, int *tamanho) {
    int contadorDest = 0;

    if (buffer[pos] == '\0') {
        *tamanho = lerProximos(buffer, arquivoDna);
        pos = 0;
    }

    while (buffer[pos] != '\n') {
        nomeDest[contadorDest] = buffer[pos];
        pos++;
        contadorDest++;
        nomeDest[contadorDest] = '\0';
        if (buffer[pos] == '\0') {
            if (strcmp(nomeDest, "EOF") == 0) {
                return pos;
            }
            *tamanho = lerProximos(buffer, arquivoDna);
            pos = 0;
        }
    }
    nomeDest[contadorDest] = '\0';
    pos++;
    return pos;
}

```

A função “**int CopiaNome(char *buffer, char *nomeDest, int pos, FILE *arquivoDna, int *tamanho)**” que recebe 5 parâmetros, é responsável por realizar a cópia dos caracteres de uma String *buffer para uma String *nomeDest até encontrar a quebra da linha.

Funcionamento: a função começa inicializando a variável “contadorDest” do tipo inteiro com o valor zero, como ela foi declarada dentro da função será uma variável local que atuará no rastreamento da posição atual durante a cópia dos caracteres.

A seguir temos o comando de seleção if verificando se a posição que foi passada como parâmetro da String “buffer” é um caractere nulo, representando que o final da string chegou, assim adentra no bloco de comandos que chama a função **lerProximos** para ler mais dados para a string buffer a partir do “arquivoDna” passado, e fechando o bloco com a atualização da variável “pos” para zero.

No caso onde não houve identificação do final do “buffer”, então deve ser feito um laço while que verificará a condição de teste da posição “pos” passada como parâmetro no “buffer” para descobrir se é diferente de um ‘\n’, ou seja diferente de uma nova linha significa que a condição é verdadeira, nesta situação então é processado dentro do loop um bloco de comandos que fará a cópia dos caracteres

da string “buffer” para a string “nomeDest” até encontrar um caractere de nova linha, se por acaso achar um ‘\0’ durante o laço, isso quer dizer que um caractere nulo foi detectado no comando de seleção, então é feita uma nova verificação por meio da strcmp, caso a string “nomeDest” seja equivalente a “EOF” então retornará 0, isso quer dizer que o final da string conquistado, então é solicitada a função **lerProximos** para ler mais dados.

No momento em que encontrou a quebra de linha ocorre a saída do laço, e logo após é atribuído o caractere nulo ao final de “nomeDest” novamente, de modo a garantir que a string resultante seja acabada de maneira correta. Por fim, incrementa mais uma vez a posição de “pos” para apontar para o próximo caractere em “buffer” e retorna esta posição atual para que a função possa ser chamada a partir deste ponto.

```
int lerProximos(char *buffer, FILE *arquivo) {  
    int lido = fread(buffer, sizeof(char), TAMANHO_BUFFER-1, arquivo);  
  
    buffer[lido] = '\0';  
    return lido;  
}
```

A função “**int lerProximos(char *buffer, FILE *arquivo){**” é responsável por ler os próximos bytes de um arquivo e armazená-los no array de caracteres “buffer”, retornando o número de elementos lidos, dentro da função. **LerProximos** recebe os parâmetros “buffer” do tipo char e “*arquivo” do tipo FILE, e começa criando uma variável, inteira, que armazenará o resultado da chamada da **fread**. Esta função permite a leitura de blocos de qualquer tipo de dado, devolvendo o número de itens lidos. Para isso precisa passar um ponteiro para uma região da memória que receberá os dados do arquivo, descrito por “*buffer”, o número de bytes específicos que devem ser lidos (neste caso é o tamanho de bytes de um char que é 1 byte), o argumento “TAMANHO_BUFFER-1” que determina quantos itens serão lidos, e, por fim, um ponteiro para uma stream aberta anteriormente, em outros termos o arquivo. Após isto é feita a atribuição do caractere nulo ao final do buffer na posição “lido” que se refere ao tamanho total do arquivo, deve ser feito isto para garantir que o array de caracteres seja tratado como uma string válida em C uma vez que as strings são terminadas com caractere nulo, e ao fim a função se encerra devolvendo o valor armazenado na variável “lido”.

3.6 Arquivo Utils.h

Esse código é um cabeçalho (header) em linguagem C para um conjunto de funções relacionadas a detecção de vírus em um texto, utilizando o algoritmo Knuth-Morris-Pratt (KMP) para a busca de padrões.

Arquivos de cabeçalho

```
#include <string.h>
#include <stdio.h>
```

A diretiva `#include` esta a instruir o compilador a ler e compilar o arquivo de cabeçalho padrão: `string.h` e `stdio.h` como rotinas de arquivos em disco da biblioteca. O arquivo de cabeçalho `string.h` permite que a implementação tenha suporte para funções de strings, e o arquivo de cabeçalho `stdio.h` permite que a implementação possa suporta E/S com arquivos.

Todas as funções que estão relacionadas a estes arquivos de cabeçalho estarão listadas no Apêndice A deste relatório.

Pré-processamento

```
#ifndef _UTILS_H_
#define _UTILS_H_
```

Através de um método de compilação condicional verifica se a macro `UTILS_H` está atualmente indefinida, nesse caso a partir da próxima linha estaremos a definindo com um bloco de código **#define**, com esta prática garantimos que o conteúdo do cabeçalho só será incluído no caso da `UTILS_H` estar indefinida evitando problemas com inclusões múltiplas em um mesmo código.

```
#endif
```

Uma diretiva de compilação condicional usada para marcar o final do bloco `#ifdef _UTILS_H_` criado no início do cabeçalho.

Enumeração

```
enum {
    LEITURA_PADRAO_COMPLETA,
    LEITURA_PADRAO_INCOMPLETA,
    FIM_DE_ARQUIVO_VIRUS,
    LEITURA_VIRUS_SUCESSO
};
```

Neste trecho usamos uma extensão da linguagem C, que é o **enum** para definir um

conjunto de constantes enumeradas inteiras que especifica todos os estados possíveis que a leitura do padrão vírus pode apresentar. Onde os valores associados a estas constantes têm o papel descrito na própria lista de enumeração, e todas as constantes são declaradas variáveis do tipo enum.

Protótipo de Funções

```
int lerPadraoVirus(FILE *arquivoVirus, char *padrao, char *nomeVirus);
```

Função que serve para ler o padrão de vírus a partir de um arquivo de vírus, armazenando o padrão e o nome do vírus. Retorna um indicador de sucesso ou fim de arquivo.

```
int buscarKMP(char *texto, int *lps, char *padrao, int offsetPos, int *ocorrencias, int *posicoes, int *auxPosicoes);
```

Função que realiza a busca de padrões em um texto usando o algoritmo KMP. Retorna um indicador de leitura do padrão: completa ou incompleta.

Onde os parâmetros são o texto onde a busca ocorrerá, o vetor LPS que estará previamente calculado para o padrão, também o padrão a ser buscado no texto, a posição de deslocamento no texto, um ponteiro para retornar o número de ocorrências do padrão encontrado, um vetor para guardar as posições onde estes padrões foram encontrados, e, por fim, um auxiliar de posições que contém o número de posições encontradas, todos nesta ordem de entrada respectivamente.

```
void calcularLPS(char *padrao, int m, int lsp[]);
```

Função que Calcula o vetor LPS (Maior sufixo que também é prefixo) para um padrão. Em vez de retornar valores ela atualiza diretamente o vetor LPS passado como argumento. Ao final da execução, o vetor LPS conterá informações sobre o comprimento do maior prefixo próprio que também é um sufixo próprio para cada posição no padrão.

```
int lerProximos(char *buffer, FILE *arquivo);
```

Função que lê caracteres de um arquivo até encontrar o caractere '>', ou até atingir o tamanho máximo definido para o buffer. Retorna o número total de caracteres lidos

```
int copiaNome(char *buffer, char *nomeDest, int pos, FILE *arquivoDna, int *tamanho);
```

Função que efetua a cópia de caracteres do buffer para o "nomeDest" até encontrar uma quebra de linha, indicando o final do nome. Se o final do buffer for alcançado

antes disso, a função lê mais caracteres do arquivo para o buffer e continua a cópia. Essa função é usada para extrair nomes do arquivo de DNA durante a execução do algoritmo principal.

3.7 Makefile

```
CC = gcc
CFLAGS = -Wall -Wextra -std=c99

# Arquivos fonte e objeto
SRCS = main.c utils.c
OBJS = $(SRCS:.c=.o)

# Nome do executável
TARGET = main

.PHONY: all clean run

all: $(TARGET)

$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) $(OBJS) -o $@

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    $(RM) $(TARGET) $(OBJS)

run: $(TARGET)
    ./$(TARGET)

debug:
    make clean && make all && make run
```

O Makefile é empregado como programa utilitário que automatiza o processo de recompilação para um grande programa que é composto de diversos arquivos como é o caso desta situação. O make compilará os módulos necessários e criar um programa executável. Este arquivo Makefile foi projetado de forma que as dependências sejam hierárquicas, assim todas as dependências subordinadas e as principais logo verificarão se foi bem-sucedido a execução.

4 Resultados

Para diferentes entradas verificamos os seguintes padrões de saída:

```

variante Q87.2
[0989487.1 paca] no. de ocorrencias: 1, posicoes: 82
[K992392.3 araponga] no. de ocorrencias: 2, posicoes: 17 60

```

```

variante K12.3
[L034233.4 tatu bola] no. de ocorrencias: 1, posicoes: 29

```

```

variante L10.2
[K992392.3 araponga] no. de ocorrencias: 1, posicoes: 38

```

```

variante C9.3

```

```

variante Z90.2
[0989487.1 onca pintada] no. de ocorrencias: 1, posicoes: 582
[L034233.4 lobo guara] no. de ocorrencias: 1, posicoes: 240
[B741807.1 jaguatirica] no. de ocorrencias: 1, posicoes: 61
[H103183.2 arara] no. de ocorrencias: 1, posicoes: 377
[P123456.4 papagaio cubano] no. de ocorrencias: 3, posicoes: 6 82 85

```

```

variante A9.2
[0989487.1 onca pintada] no. de ocorrencias: 1, posicoes: 7

```

```

variante H20.1
[0989487.1 onca pintada] no. de ocorrencias: 1, posicoes: 20

```

```

variante B43.2
[Y487143.9 capivara] no. de ocorrencias: 1, posicoes: 59
[P123456.4 papagaio cubano] no. de ocorrencias: 1, posicoes: 0

```

```

variante V5.1
[M944455.7 canario] no. de ocorrencias: 1, posicoes: 77

```

```

variante B10.2
[W879647.8 cachorro-do-mato] no. de ocorrencias: 1, posicoes: 92

```

```

variante L13.2
[R531273.1 preguica] no. de ocorrencias: 11, posicoes: 100 104 125 129 167 171 175 179 183 187 191
[A141445.2 jacare] no. de ocorrencias: 6, posicoes: 96 100 119 123 127 131
[M944455.7 canario] no. de ocorrencias: 6, posicoes: 163 167 194 261 309 313

```

```

variante P2.3
[R531273.1 preguica] no. de ocorrencias: 14, posicoes: 58 62 85 101 105 126 130 168 172 176 180 184 188 192
[A141445.2 jacare] no. de ocorrencias: 7, posicoes: 97 101 120 124 128 132 203
[M944455.7 canario] no. de ocorrencias: 9, posicoes: 160 164 168 172 195 262 306 310 314
[W879647.8 cachorro-do-mato] no. de ocorrencias: 1, posicoes: 23

```

Obtivemos resultados satisfatórios relacionados ao projeto, de tal maneira que testamos entradas variadas para verificar as condições em que era possível aplicar o programa, assim as seguintes características foram pressupostas para funcionamento correto:

- Seja BaseDadosDNA.txt o arquivo de texto com um tamanho N e PadroesVirus.txt o arquivo do padrão com tamanho M, a ser pesquisado em BaseDadosDNA.txt. Temos que é necessário $M \leq N$. Em nossa implementação o PadroesVirus.txt pode ser maior que a BaseDadosDNA.txt porém os dois deverão ser menores do que o valor do Buffer.
- Os elementos da BaseDadosDNA.txt e de PadroesVirus.txt são caracteres pertencentes ao alfabeto finito Σ da sequência de DNA dados por: A (adenina), T (Timina), C(citosina), e G(guanina).
- As entradas de DNA devem possuir no máximo 70 caracteres por linha.
- Por convenção, limitamos o buffer para o valor de 50000, o que pode ser traduzido em 50000 bytes, isto quer dizer que é possível ler 50 mil bytes por vez, uma vez que cada caractere do tipo char equivale a 1 byte.

OBS:SE O TAMANHO DO BUFFER PRECISAR SER MODIFICADO, SÓ PRECISA MODIFICAR A DEFINIÇÃO DE TAMANHO_BUFFER, LOGO ALTERANDO O VALOR NA MACRO "#define".

```
#define TAMANHO_BUFFER 50000 //O tamanho desse buffer deve ser maior que o maior padrão de virus
```

Deste modo como um arquivo-texto comum espera-se ter o tamanho em disco em KB, logo ao realizar uma conversão temos que esta implementação poderá lidar com dados da BaseDadosDNA.txt até quase 50KB por vez em execução de leitura, caso o arquivo seja maior ele fará quantas leituras forem necessárias, basicamente seguindo a regra, $\text{TamanhoArquivo}/50\text{KB} = x$, então seria realizada x leituras. A conversão de bytes em Kilobytes é dada conforme a ilustração a seguir:

1 kilobyte (KB) = 1024 bytes

Então, para converter 50 mil bytes para kilobytes, você divide a quantidade de bytes por 1024:

$50,000 \text{ bytes} \div 1024 \text{ bytes/KB} \approx 48.828 \text{ KB}$

Ao fim desta etapa obtivemos resultados conforme o esperado, e com muita exatidão, tendo apresentado consistência no tempo de execução uma vez que o algoritmo KMP possui um emparelhamento de cadeias de tempo linear do tamanho do texto BaseDadosDNA.txt, e possui um tempo de pré-processamento linear do tamanho da entrada PadraoVirus.txt.

5 Conclusão

Ao fim deste projeto concluímos que o problema de encontrar todas as ocorrências de um padrão em um texto é algo desafiador, e que as soluções eficientes melhoram muito o nível de resposta para programas de edição de texto.

Não atoa que correspondência de strings é aplicado na busca de padrões específicos em sequências de DNA, uma vez que isto é a base para o descobrimento de muitos problemas genéticos do mundo.

Pudemos perceber, no decorrer deste trabalho, que foi extremamente necessário entender todo o conceito do algoritmo de Knuth-Morris-Pratt, ensinado em sala, para resolver o trabalho. Ao decorrer também avaliamos muitas revisões do conteúdo base de arquivos em C, que é lembrado no decorrer da disciplina, e a partir do uso destas ferramentas e a pesquisa feita nos livros que foi de grande ajuda conseguimos concluir este projeto. Tal trabalho possibilitou a nossa melhora nas práticas de programação, noção de problemas reais, entendimento dos códigos, além de melhorar a nossa capacidade de pesquisa e leitura de conteúdo voltado a

área da computação.

6 Bibliografia

SCHILDT, Herbert. **C: COMPLETO E TOTAL** 3ª edição revista e atualizada. São Paulo: Pearson Makron Books, 1997.

CORMEN, Thomas H. **INTRODUCTION TO ALGORITHMS** Third edition. Cambridge, Massachusetts London, England: The MIT Press, 2009.

Link correspondente aos slides da aula síncrona referente ao conteúdo:

https://drive.google.com/file/d/1rB5h4wCi-R6Ohx9SLyupl9j_7GSF5iWW/view?usp=sharing

7 Apêndices

Apêndice A:

Biblioteca stdio.h

Função: fopen, fclose, fread, fseek, printf, fgets, perror

Biblioteca string.h

Função: strcmp, strlen,

Apêndice B:

- Makefile:
- CC = gcc: Define o compilador a ser usado como gcc
- CFLAGS = -Wall -Wextra -std=c99: Define as opções de compilação, incluindo ativação de avisos (-Wall e -Wextra) e especifica que o código segue o padrão C99 (-std=c99).
- SRCS = main.c utils.c: Lista dos arquivos fonte.
- OBJS = \$(SRCS:.c=.o): Lista dos arquivos objeto, derivados dos arquivos fonte. Substitui a extensão .c por .o.
- TARGET = main: Nome do executável que será gerado.
- PHONY: all clean run: Indica que all, clean, e run não são arquivos reais, mas sim comandos.
- all: \$(TARGET): Indica que o alvo padrão é a construção do executável \$(TARGET).
- \$(TARGET): \$(OBJS): Define a regra para construir o executável \$(TARGET) a partir dos objetos \$(OBJS).
- \$(CC) \$(CFLAGS) \$(OBJS) -o \$@: Comando de compilação, onde \$@ representa o alvo (\$(TARGET)).
- %.o: %.c: Define a regra para compilar objetos a partir de arquivos-fonte.
- \$(CC) \$(CFLAGS) -c \$< -o \$@: Comando de compilação de objetos, onde \$< representa o arquivo-fonte e \$@ representa o alvo.
- clean:: Define a regra para limpar os arquivos gerados durante a compilação.
- \$(RM) \$(TARGET) \$(OBJS): Comando para remover o executável e os arquivos objeto.
- run: \$(TARGET): Define a regra para executar o programa.
- ./\$(TARGET): Comando para executar o programa.
- debug:: Define a regra para compilar, limpar e executar o programa para fins de depuração.
- make clean && make all && make run: Comando para limpar, compilar e executar o programa em sequência.

