SERVIÇO PÚBLICO FEDERAL · MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE VIÇOSA · UFV
CAMPUS FLORESTAL
PROFESSOR: DANIEL MENDES BARBOSA
SEMESTRE: 2025-2

Documentação Trabalho Prático 3

Projeto e Análise de Algoritmos

Júlio César De Souza Oliveira [5903]
Érick Vinícius Issa Silva [5936]
Heitor Porto Jardim de Oliveira [5895]
Helom Felipe Marques Alves [5892]

Florestal - MG
2025

1. Introdução

O trabalho tem como objetivo desenvolver um algoritmo para realizar a criptoanálise de 12 arquivos criptografados. Dentro do contexto fictício, o algoritmo servirá para decifrar os textos presentes em uma caverna, contendo 12 profecias que ajudarão a encontrar os 12 heróis predestinados com sangue dourado.

O código envolve algoritmos de casamento de caracteres exato e aproximado, utilizando respectivamente algoritmos de BMH e Shift-and aproximado, testes de chaves de acordo com análise de frequência das letras, além de leitura e escrita de arquivos.

2.Algoritmos Projetados e Implementação

Após a compilação e execução, o programa funciona de forma iterativa a partir das escolhas do usuário. Posteriormente, o arquivo de texto a ser processado terá seu conteúdo criptografado e armazenado e então será exibido um menu de operações a serem realizadas sobre ele.

```
PS C:\Users\julio\Documents\ufv\PAA\TP3---PAA> ./main.exe
Deseja ler o texto do grupo (1) ou inserir manualmente outro texto?(2): 1
--------------------MENU--------------------
1 - Apresentar o estado atual da criptografia
2 - Fazer um chute baseado na analise de frequencia no texto encriptografado
3 - Realizar casamento exato de caracteres no texto encriptografado
4 - Realizar casamento aproximado de caracteres no texto parcialmente decifrado
5 - Alterar chave de criptografia
6 - Exportar resultado e encerrar o programa
Escolha uma opcao acima:
```

Figura 1 - Menu.

O sistema deve ser capaz de realizar uma criptografia por substituição, para tanto foi criado a função “cifraDeDeslocamento” que criptografa um texto utilizando a cifra de César com um deslocamento aleatório. Além disso, a função não criptografa pontuações e outros símbolos. Segue abaixo a implementação dessa função de criptografia.

```
void cifraDeDeslocamento(char *texto) {
    srand(time(NULL));
```

```
          int chave = gerarNmrAleatorio(1,25);
          for (int i = 0; texto[i] != '\0'; i++) {
              char c = texto[i];
              if(texto[i] == '.' || texto[i] == 'n' || texto[i] == '"'
|| texto[i] == ' '){
                  continue;
              }
              if (c >= 'a' && c <= 'z') {
                  texto[i] = ((c - 'a' + chave) % 26) + 'a';
              }
              else if (c >= 'A' && c <= 'Z'){
                  texto[i] = ((c - 'A' + chave) % 26) + 'A';
              }     }

      }
```

Figura 2 - Função de criptografia usando uma cifra de deslocamento.

Caso o usuário execute a operação “1” será exibido o texto criptografado e logo depois as chaves, a normal e a parcial. Logo em seguida segue o texto parcialmente decifrado, onde as letras vermelhas indicam as letras que ainda estão criptografadas e os caracteres verdes identificam aqueles foram descriptografados. Segue abaixo um exemplo de execução.

```
Escolha uma opcao acima: 1

=== Texto criptografado ===
J KZDOJ YZ BZJMDJ Z J XJMKJ YJ YMVBVJ XVHDICVIOZ NPKJMOVH V OZMMV YZNKZYVXVYV, NPKJMOVIYJ HDG VIJN YZ VBJIDV.

J YVI CZIB DIJHDIVYJ, CZMYZDMJ YZ XMDNJ LPZ BPVMYV V XCVHV YV OZMMV, QJXZ YZQZ ADMHVM J HPIYJ ZH NPV LPZYV, Z BPDVM OJYV V QDYV YV OZMMV
KVMV PH IJQJ GVM VGZH.

"QJXZ ZNXVQVMV NPV KMJKMDV NZKPGOPMV Z LPZDHVMV IV AJMIVGCV YV MZWZGDVJ."

=== Chave ===
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
              O

=== Texto parcialmente decifrado ===
J KZDJJ YZ BZJMDJ Z J XJMKJ YJ YMVBVJ XVHDICVIJZ NPKJMJVH V JZMMV YZNKZYVXVYV, NPKJMJVIYJ HDG VIJN YZ VBJIDV.

J YVI CZIB DIJHDIVYJ, CZMYZDMJ YZ XMDNJ LPZ BPVMYV V XCVHV YV JZMMV, QJXZ YZQZ ADMHVM J HPIYJ ZH NPV LPZYV, Z BPDVM JJYV V QDYV YV JZMMV
KVMV PH IJQJ GVM VGZH.

"QJXZ ZNXVQVMV NPV KMJKMDV NZKPGJPMV Z LPZDHVMV IV AJMIVGCV YV MZWZGDVJ."

--------------------MENU--------------------
1 - Apresentar o estado atual da criptografia
2 - Fazer um chute baseado na analise de frequencia no texto encriptografado
3 - Realizar casamento exato de caracteres no texto encriptografado
4 - Realizar casamento aproximado de caracteres no texto parcialmente decifrado
5 - Alterar chave de criptografia
6 - Exportar resultado e encerrar o programa
Escolha uma opcao acima:
```

Figura 3 - Resultado ao executar a opção 1 do menu.

2.1. Estruturas de Dados

A principal estrutura de dados implementada no projeto está no arquivo “TAD_criptografia.h”, nele foi criado um tipo “criptografia” que é composto de um atributo do tipo chave, que armazena a chave normal e a cifra a ser descoberta, e outros três atributos relativo às diferentes versões do texto, seja ele claro, criptografado e parcialmente criptografado.

Além dos atributos, destaca-se aqui algumas das principais operações possíveis, dentre elas temos a inicialização desse novo tipo com intuito de alocar memória suficiente e armazenar o texto. Além disso, há operações de impressão, tanto das chaves quanto das versões do texto armazenado, e uma função que permite exportar os resultados. Vale ressaltar que a medida que o programa é executado e a chave vai sendo preenchida, o texto parcial é capaz de ser gerado pelas operações. Ao imprimir esse texto parcial, letras que não foram alteradas, ou seja ainda encriptografadas, serão exibidas em vermelho. Enquanto letras que foram substituídas a partir da cifra atual, será exibido em verde.

Esse tipo abstrato de dados será utilizado e manipulado por todo o programa, visto que nele se encontram as chaves e os textos. O arquivo .h dessa estrutura pode ser observado na figura abaixo.

```
#ifndef TAD_CRIPTOGRAFIA_H
#define TAG_CRIPTOGRAFIA_H

typedef struct{
    char normal[26];
    char cifra[26];
}chave;

typedef struct{
    chave Chaves;
    char * claro;
    char * criptografado;
    char * parcial;
}criptografia;

void inicializaChaves(criptografia * cripto, char * texto);
void imprimeClaro(criptografia * cripto);
```

```
void imprimeParcial(criptografia * cripto);
void imprimeCriptografado(criptografia * cripto);
void imprimeChaves(criptografia * cripto);
void alterarChave(criptografia *cripto, char original, char
encriptada);
void exportarResultado(criptografia *cripto);
#endif
```

Figura 4 - “TAD_criptografia”

2.2. Análise de Frequência

A análise de frequência possuí quatro opções sendo elas respectivamente, 1: chute com base apenas na frequência do texto do grupo em relação ao alfabeto, 2:chute com base frequência dos 12 textos disponibilizados em relação ao alfabeto, 3: essa opção é um extra no qual ela analisa a letra com maior frequência e procura deslocar todo o resto do alfabeto em relação a essa letra, 4: essa opção mostra as tabelas de frequência do texto do grupo, os 12 textos e o do alfabeto.

```
            {
            case 1:
                chutaCifraTexto(colecao, &teste);
                break;
            case 2:
                chutaCifra12Textos(colecao, &teste);
                break;
            case 3:
                cravaMapeamento(colecao, &teste);
                break;
            case 4:
                imprimeColecao(colecao);
            default:
                break;
            }
```

Figura 5 - Principais funções chamadas para cada operação

A estrutura de armazenamento consiste em um TAD coleção que possui 3

ListasFrequencia[] de tamanho 27 como definido em tamVetor. O tamanho 27 foi escolhido devido a quantidade de letras existentes no alfabeto romano, com um espaço extra. Cada indice de ListaFrequencia possuí o caractere analisado no texto, a quantidade de vezes que ele aparece no texto e sua frequência relativa.

```c
#ifndef TAD_FREQUENCIA_H
#define TAG_FREQUENCIA_H

#include "../include/TAD_Criptografia.h"

#define tamVetor 27

typedef struct{
    char caractere;
    double qntd;
    double frequencia;
}ListaFrequencia;

typedef struct{
    ListaFrequencia listaTexto[tamVetor];
    ListaFrequencia lista12textos[tamVetor];
    ListaFrequencia listaAlfabeto[tamVetor];
}ColecaoFrequencias;
```

Figura 6 - TAD_Frequencia implementado

A frequência é calculada pela função criaColecaoFrequencia() que recebe uma variável ColecaoFrequencias. Ao chamar essa função, são lidos os textos presentes na pasta input(importante ressaltar que é necessário todos os arquivos em input para funcionamento do código em suas devidas pastas). Esse cálculo é dado da seguinte forma, o programa lê uma char* processada anteriormente pelo TAD_entrada.c, a partir disso é contabilizada cada letra, aumentando a quantidade de ocorrências a cada vez que ela aparece repetida, a função calculaFrequencia() conta a quantidade de letras totais desse char* passado e faz a frequência relativa. Esse processo é feito similarmente para as 3 ListaFrequencia em coleção com seus respectivos arquivos de texto. Por fim é executado um Quicksort decrescente para ordenar com base no valor decimal da frequência.

```
void executaFrequencia(ListaFrequencia lista[], char *texto){
    criaListaFrequencia(lista);
    contabilizaLetra(lista, texto);
    calculaFrequencia(lista, texto);
    QuickSort(lista,tamVetor);
}
```

Figura 7 - função executaFrequencia()

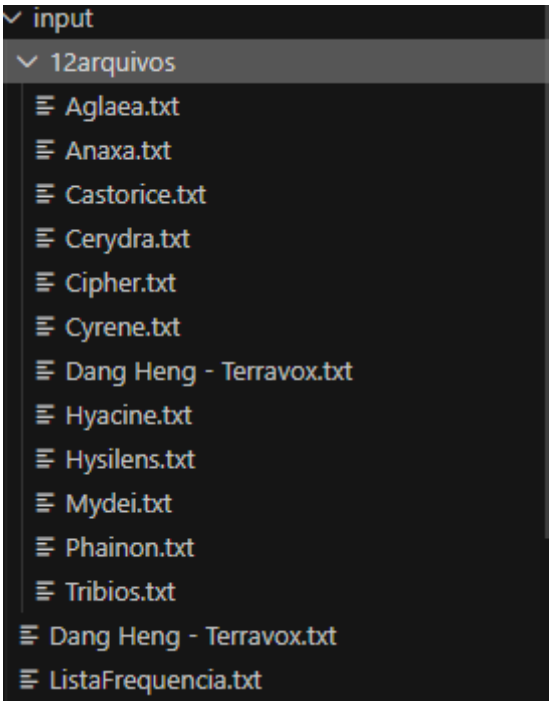
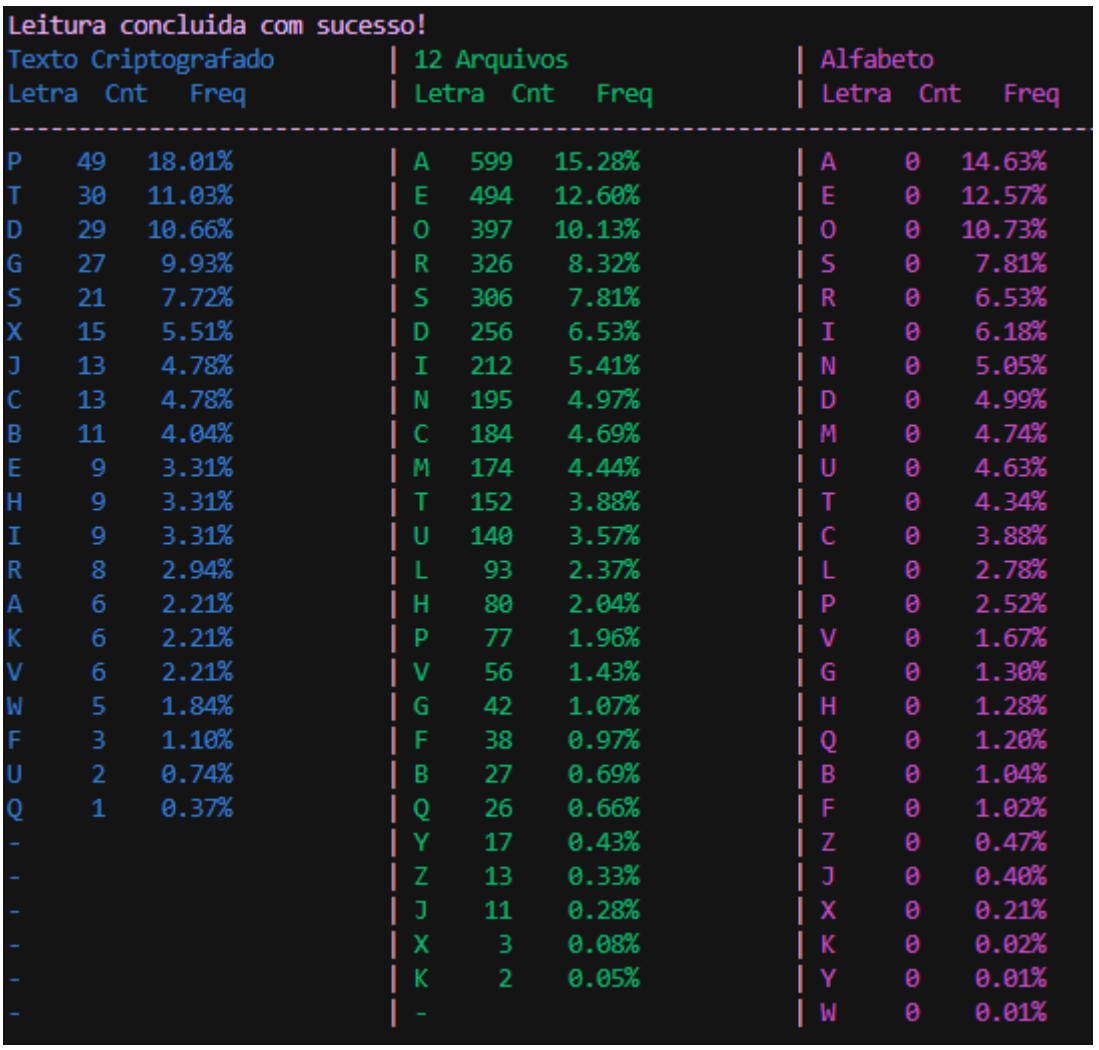


Figura 8 - Estrutura de como deve ser posicionados e nomeados os arquivos de texto

O arquivo txt ListaFrequencia.txt possui as frequências do alfabeto, como extra, caso seja preciso a tabela de frequência de outra letra, se deve posicionar as letras da seguinte forma para a leitura ser efetuada com sucesso:

a 0.1463
b 0.0104
c 0.0388
d 0.0499
e 0.1257

A sequência é: letra frequência relativa.



```
Leitura concluida com sucesso!
Texto Criptografado        | 12 Arquivos              | Alfabeto
Letra  Cnt   Freq          | Letra  Cnt   Freq        | Letra  Cnt   Freq
-------------------------------------------------------------------------
P    49   18.01%           | A   599   15.28%         | A     0   14.63%
T    30   11.03%           | E   494   12.60%         | E     0   12.57%
D    29   10.66%           | O   397   10.13%         | O     0   10.73%
G    27    9.93%           | R   326    8.32%         | S     0    7.81%
S    21    7.72%           | S   306    7.81%         | R     0    6.53%
X    15    5.51%           | D   256    6.53%         | I     0    6.18%
J    13    4.78%           | I   212    5.41%         | N     0    5.05%
C    13    4.78%           | N   195    4.97%         | D     0    4.99%
B    11    4.04%           | C   184    4.69%         | M     0    4.74%
E     9    3.31%           | M   174    4.44%         | U     0    4.63%
H     9    3.31%           | T   152    3.88%         | T     0    4.34%
I     9    3.31%           | U   140    3.57%         | C     0    3.88%
R     8    2.94%           | L    93    2.37%         | L     0    2.78%
A     6    2.21%           | H    80    2.04%         | P     0    2.52%
K     6    2.21%           | P    77    1.96%         | V     0    1.67%
V     6    2.21%           | V    56    1.43%         | G     0    1.30%
W     5    1.84%           | G    42    1.07%         | H     0    1.28%
F     3    1.10%           | F    38    0.97%         | Q     0    1.20%
U     2    0.74%           | B    27    0.69%         | B     0    1.04%
Q     1    0.37%           | Q    26    0.66%         | F     0    1.02%
-                          | Y    17    0.43%         | Z     0    0.47%
-                          | Z    13    0.33%         | J     0    0.40%
-                          | J    11    0.28%         | X     0    0.21%
-                          | X     3    0.08%         | K     0    0.02%
-                          | K     2    0.05%         | Y     0    0.01%
-                          | -                        | W     0    0.01%
```

Figura 9 - Tabelas calculadas lado a lado

Dado calculadas todas as tabelas de frequência é realizado os chutes pelos seguintes métodos: chutaCifraTexto(), chutaCifra12textos() e cravaMapeamento(). As funções chutaCifraTexto() e chutaCifra12textos() tem lógica similar, no qual é selecionando cada índice em ordem crescente das posições da lista, a frequência desse índice é pesquisada pela função buscaBinariaAproximada(), que retorna o índice que é utilizado em colecao.listaAlfabeto[indice].caractere para achar o caractere que possui a frequência mais parecida com o caractere cifrado. Por fim, o método alteraChave() muda isso na estrutura criptografia.

```c
void chutaCifraTexto(ColecaoFrequencias colecao, criptografia *cripto){
    for(int i = 0; i < 27; i++){
        if(colecao.listaTexto[i].caractere == '*')
            continue;

        double freq = colecao.listaTexto[i].frequencia;
        int indice = buscaBinariaAproximada(colecao.listaAlfabeto, tamVetor, freq);

        //printf("%c -> %c\n", colecao.listaTexto[i].caractere, colecao.listaAlfabeto[indice].caractere);
        //substituir funcao troca
        alterarChave(cripto, colecao.listaAlfabeto[indice].caractere, colecao.listaTexto[i].caractere);
        colecao.listaAlfabeto[indice].caractere = '*';
    }
    printf("\n");
}
```

Figura 10 - método chutaCifraTexto()

O método cravaMapeamento() adota uma abordagem determinística baseada no deslocamento padrão de cifras de rotação (como a Cifra de César). Inicialmente, percorre-se a colecao.listaTexto para isolar o caractere que possui a maior frequência absoluta (idxMax). Em seguida, a frequência desse caractere é submetida à função buscaBinariaAproximada() na colecao.listaAlfabeto, identificando qual letra do alfabeto padrão corresponde a essa frequência. A partir da diferença entre o caractere cifrado e o caractere padrão encontrado, calcula-se um valor de deslocamento. Por fim, esse deslocamento é aplicado matematicamente a todos os caracteres da coleção para deduzir suas versões decodificadas, e o mapeamento resultante é registrado na estrutura através da função alterarChave().

*RESSALTA-SE: Essa função é um extra, para não atrapalhar o fluxo de execução principal, ao selecionar a opção 2 no menu, não selecionar após isso a opção (3 - Chute feito com base na letra que tem maior frequência, e após isso deslocamento das outras).

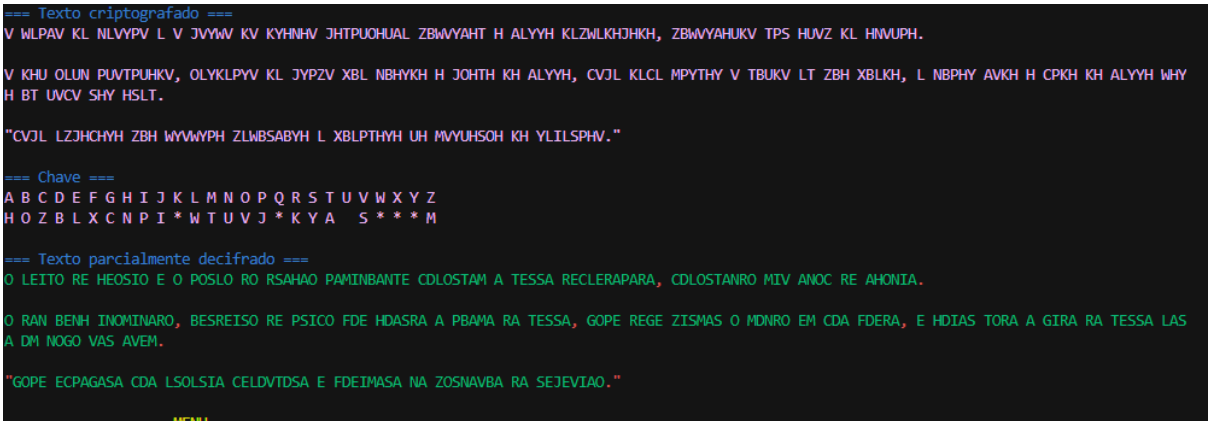


Figura 11 - Mapeamento feito com base na frequência do texto do grupo

2.3. Busca Exata (Casamento de Padrões)

Para realizar o casamento exato de padrões no texto criptografado, foi utilizado uma adaptação do algoritmo BMH visto em sala. Seu código foi removido de um site (Referência 1), que disponibilizou uma implementação em Java. A partir dele foi feita uma tradução para C, utilizando o Chat GPT, e posteriormente foi modificado para que, além de retornar onde o casamento ocorreu, também informasse a quantidade dessas ocorrências e a frequência em relação ao total de letras. Segue abaixo a função “casamento_exato” localizada no arquivo “src/processador_cripto.c”.

```c
void casamento_exato(char *texto, char *padrao,float * frequencia, int *contador) {
    int n = strlen(texto);
    int m = strlen(padrao);
    int lastOcc[128];

    computeLastOcc(padrao, lastOcc);

    *contador = 0;

    int i0 = 0;

    while (i0 <= n - m) {
        int j = m - 1;

        while (j >= 0 && padrao[j] == texto[i0 + j]) {
```

```
            j--;
        }

        if (j < 0) {
            // casamento exato
            printf("Ocorrencia encontrada na posicao %d\n", i0);
            (*contador)++;


            i0++;
        } else {
            unsigned char c = texto[i0 + m - 1];
            int shift = (m - 1) - lastOcc[c];
            if (shift <= 0) shift = 1;
            i0 += shift;
        }
    }
    float aux = (*contador) ;
    *frequencia = ((aux) / (n/m)) * 100;
}
```

Figura 12 - Função que calcula o casamento exato.

2.4. Busca Aproximada (Shift-And Aproximado)

Para a busca aproximada, utilizou-se o algoritmo do shift-and aproximado, apenas com a operação de substituição. Ao escolher a opção 4 no menu, o usuário escolhe qual a quantidade máxima de operações. Logo após, ele tem a opção de visualizar ou não os detalhes do shift-and. No caso afirmativo, são mostradas as tabelas de R e R', além da máscara de bits utilizada no processo. Após os detalhes, é mostrada a saída padrão da opção 4, isto é, a informação de quantas ocorrências foram encontradas para cada k (máximo de erros), e para cada ocorrência, é impresso no terminal o texto inteiro em branco, com a palavra da ocorrência colorida, sendo as letras que casam pintadas de verde, e as letras que não casam com a cor vermelha. Segue exemplo de saída para o padrão "CARO", com distância de edição 2 nas figuras abaixo:

```
--------------------MENU--------------------
1 - Apresentar o estado atual da criptografia
2 - Fazer um chute baseado na analise de frequencia no texto encriptografado
3 - Realizar casamento exato de caracteres no texto encriptografado
4 - Realizar casamento aproximado de caracteres no texto parcialmente decifrado
5 - Alterar chave de criptografia
6 - Exportar resultado e encerrar o programa
Escolha uma opcao acima: 4
Qual o padrao utilizado?
> CARO
Qual o limite de operacoes?
>2
Gostaria de visualizar, alem das ocorrencias, as tabelas do shift-and? entradas aceitas:(s/n)
s

K       | k = 0 :1000 0000 || k = 1 :1000 1000 || k = 2 :1000 1000 ||
        | k = 0 :1000 0000 || k = 1 :1100 1000 || k = 2 :1100 1100 ||
L       | k = 0 :1000 0000 || k = 1 :1100 1000 || k = 2 :1110 1100 ||
A       | k = 0 :1000 0000 || k = 1 :1100 1100 || k = 2 :1110 1100 ||
E       | k = 0 :1000 0000 || k = 1 :1110 1000 || k = 2 :1110 1110 ||
P       | k = 0 :1000 0000 || k = 1 :1100 1000 || k = 2 :1111 1100 ||
K       | k = 0 :1000 0000 || k = 1 :1100 1000 || k = 2 :1110 1100 ||
        | k = 0 :1000 0000 || k = 1 :1100 1000 || k = 2 :1110 1100 ||
Z       | k = 0 :1000 0000 || k = 1 :1100 1000 || k = 2 :1110 1100 ||
A       | k = 0 :1000 0000 || k = 1 :1100 1100 || k = 2 :1110 1100 ||
        | k = 0 :1000 0000 || k = 1 :1110 1000 || k = 2 :1110 1110 ||
C       | k = 0 :1000 1000 || k = 1 :1100 1000 || k = 2 :1111 1100 ||
A       | k = 0 :1100 0100 || k = 1 :1100 1100 || k = 2 :1110 1100 ||
K       | k = 0 :1010 0000 || k = 1 :1110 1010 || k = 2 :1110 1110 ||
N       | k = 0 :1000 0000 || k = 1 :1101 1000 || k = 2 :1111 1101 ||
```

Figura 13 - Exemplo de entrada para casamento aproximado.

```
K       | k = 0 :1000 0000 || k = 1 :1100 1000 || k = 2 :1110 1100 ||
.       | k = 0 :1000 0000 || k = 1 :1100 1000 || k = 2 :1110 1100 ||
"       | k = 0 :1000 0000 || k = 1 :1100 1000 || k = 2 :1110 1100 ||


C 1 0 0 0
A 0 1 0 0
R 0 0 1 0
O 0 0 0 1


Ocorrencias com 0 operacoes:  0
Ocorrencias com 1 operacoes:  0
Ocorrencias com 2 operacoes:  3


Ocorrencias para 0 operacoes:


Ocorrencias para 2 operacoes:
Ocorrencia 1:
K LAEPK ZA CAKNEK A K YKNLK ZK ZNWCWK YWIEJDWJPA OQLKNPWI W PANNW ZAOLAZWYWZW, OQLKNPWJZK IEH WJKO ZA WCKJEW.

K ZWJ DAJC EJKIEJWZK, DANZAENK ZA YNEOK MQA CQWNZW W YDWIW ZW PANNW, RKYA ZARA BENIWN K IQJZK AI OQW MQAZW, A CQEWN PKZW W REZW ZW PANNW LWNW QI JKRK HWN

"RKYA AOYWRWNW OQW LNKLNEW OALQHPQNW A MQAEIWNW JW BKNJWHDW ZW NAXAHEWK."
Ocorrencia 2:
K LAEPK ZA CAKNEK A K YKNLK ZK ZNWCWK YWIEJDWJPA OQLKNPWI W PANNW ZAOLAZWYWZW, OQLKNPWJZK IEH WJKO ZA WCKJEW.

K ZWJ DAJC EJKIEJWZK, DANZAENK ZA YNEOK MQA CQWNZW W YDWIW ZW PANNW, RKYA ZARA BENIWN K IQJZK AI OQW MQAZW, A CQEWN PKZW W REZW ZW PANNW LWNW QI JKRK HWN

"RKYA AOYWRWNW OQW LNKLNEW OALQHPQNW A MQAEIWNW JW BKNJWHDW ZW NAXAHEWK."
Ocorrencia 3:
K LAEPK ZA CAKNEK A K YKNLK ZK ZNWCWK YWIEJDWJPA OQLKNPWI W PANNW ZAOLAZWYWZW, OQLKNPWJZK IEH WJKO ZA WCKJEW.

K ZWJ DAJC EJKIEJWZK, DANZAENK ZA YNEOK MQA CQWNZW W YDWIW ZW PANNW, RKYA ZARA BENIWN K IQJZK AI OQW MQAZW, A CQEWN PKZW W REZW ZW PANNW LWNW QI JKRK HWN

"RKYA AOYWRWNW OQW LNKLNEW OALQHPQNW A MQAEIWNW JW BKNJWHDW ZW NAXAHEWK."


--------------------MENU--------------------
```

Figura 14 - Exemplo de saída para casamento aproximado.

A posição de cada ocorrência é armazenada em um vetor de listas encadeadas, que em cada linha guarda as ocorrências de uma certa distância de edição k. Abaixo a imagem de como as variáveis importantes estão sendo declaradas.

```
void casamentoAproximado(int maxOperacoes, char texto[], char padrao[]){
    //declarações
    int tamanhoPadrao = strlen(padrao);
    int tamanhoTexto = strlen(texto);
    int tamanhoM = 0;
    int infos = -1;
    char M[tamanhoPadrao][tamanhoPadrao+1];
    Ocorrencias ocorrencias[maxOperacoes+1];
    int R[maxOperacoes+1][tamanhoPadrao];
    int Rlinha[maxOperacoes+1][tamanhoPadrao];
```

Figura 15 - Declarações de variáveis.

Para fazer o shift dos bits de R', foi necessário criar uma função própria para isso, uma vez que os dados são armazenados em um vetor de inteiros. Essa função foi implementada da seguinte forma:

```
void shift(int vetor[], int tamanho){
    for(int i = tamanho - 1;  i>0; i--){
        vetor[i] = vetor[i-1];
    }
    vetor[0] = 0;
}
```

Figura 16 - Função shift.

2.5. Alterar Chave de criptografia

A operação 5 permite que o usuário atualize manualmente a chave de criptografia durante o processo de criptoanálise. A partir de inferências obtidas nas demais operações (como análise de frequência, casamento exato ou aproximado), o usuário pode informar um par de caracteres representando a letra original e a letra para a qual ela foi mapeada no texto criptografado.

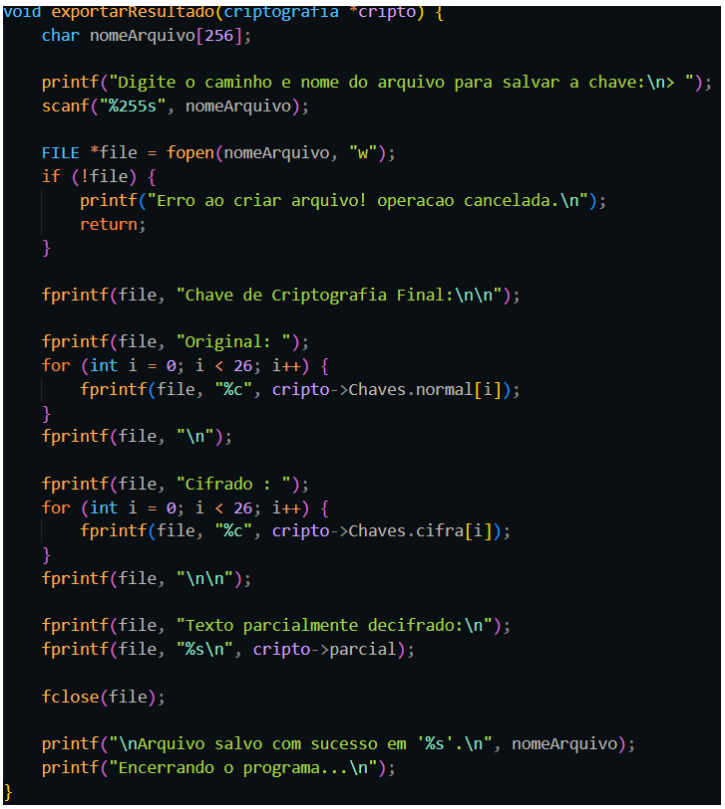
O programa, então, registra essa associação na estrutura de chave, substituindo todas as ocorrências da letra criptografada pela correspondente letra original.

```c
void alterarChave(criptografia *cripto, char original, char encriptada) {

    original = toupper(original);
    encriptada = toupper(encriptada);
    if(original< 65 || original > 90 || encriptada < 65 || encriptada > 90){
        printf("Caracter invalido! Operacao cancelada.\n");
        return;
    }
    int idx = original - 'A';
    cripto->Chaves.cifra[idx] = encriptada;

    printf("Registrado: %c -> %c\n\n", original, encriptada);

    int n = strlen(cripto->criptografado);

    for (int i = 0; i < n; i++) {
        if (cripto->criptografado[i] == encriptada) {
            cripto->parcial[i] = original;
        }
    }
}
```

Figura 17 - Função alterarChave.

2.6. Exportar Resultados

A operação 6 encerra o processo de criptoanálise. Ao escolhê-la, o programa solicita ao usuário um caminho de arquivo para armazenar o resultado final. Em seguida, gera um arquivo de texto contendo a chave de criptografia construída ao longo da execução e o estado final do texto parcialmente decifrado.



```c
void exportarResultado(criptografia *cripto) {
    char nomeArquivo[256];

    printf("Digite o caminho e nome do arquivo para salvar a chave:\n> ");
    scanf("%255s", nomeArquivo);

    FILE *file = fopen(nomeArquivo, "w");
    if (!file) {
        printf("Erro ao criar arquivo! operacao cancelada.\n");
        return;
    }

    fprintf(file, "Chave de Criptografia Final:\n\n");

    fprintf(file, "Original: ");
    for (int i = 0; i < 26; i++) {
        fprintf(file, "%c", cripto->Chaves.normal[i]);
    }
    fprintf(file, "\n");

    fprintf(file, "Cifrado : ");
    for (int i = 0; i < 26; i++) {
        fprintf(file, "%c", cripto->Chaves.cifra[i]);
    }
    fprintf(file, "\n\n");

    fprintf(file, "Texto parcialmente decifrado:\n");
    fprintf(file, "%s\n", cripto->parcial);

    fclose(file);

    printf("\nArquivo salvo com sucesso em '%s'.\n", nomeArquivo);
    printf("Encerrando o programa...\n");
}
```

Figura 18 - Função exportarResultado.

2.7. Leitura de arquivo

A manipulação da entrada de dados é centralizada no módulo responsável pela leitura e pré-processamento dos textos. A função lerEntrada() gerencia a aquisição do conteúdo, operando em dois modos distintos determinados por uma flag: leitura direta de um arquivo padrão pré-definido ou solicitação interativa do caminho do arquivo ao usuário. Para garantir a alocação eficiente de memória, o algoritmo utiliza fseek e ftell para determinar o tamanho exato do buffer necessário antes da leitura dos dados.

Após a carga do texto cru, a normalização é executada pela função removeAcentoseMaiusculas(). Este procedimento itera sobre a cadeia de caracteres, identificando bytes específicos da codificação UTF-8 (como o prefixo 0xC3) para substituir vogais acentuadas e cedilhas por seus equivalentes ASCII, convertendo simultaneamente todos os caracteres para caixa alta, conforme exigido para a criptoanálise. Adicionalmente, para atender ao requisito de análise conjunta dos textos, a função folder_to_string() percorre um diretório especificado, filtrando arquivos com extensão .txt e concatenando seus conteúdos em uma única estrutura dinâmica, que é realocada (realloc) progressivamente conforme novos arquivos são processados.

3. Compilação e Execução

Para compilar o projeto, foi criado um arquivo “*Makefile”* compatível tanto com *Windows* quanto com *Linux*. No caso do *Linux*, o processo é feito diretamente pelo terminal. Basta acessar a pasta do projeto e executar o comando “*make*” para compilar todos os arquivos fonte localizados em nas pasta “\src”, gerando os objetos na pasta “obj/” e o executável “*main*”. Após a compilação, o programa pode ser executado com “./main”. Para remover arquivos temporários e recompilar tudo do zero, utilizam-se os comandos “*make clean*” e “*make rebuild*”, respectivamente.

No Windows, o processo é equivalente, utilizando o “mingw32-make” disponível com o compilador MinGW. É necessário que o “MinGW” esteja instalado e configurado na variável de ambiente “*PATH*”, permitindo o uso dos comandos “gcc” e “mingw32-make” no terminal. Dentro do diretório do projeto, deve-se executar o comando “mingw32-make” para gerar os arquivos objeto e o executável “main.exe”. A execução é feita digitando “./main.exe” no terminal. Assim como no Linux, é possível limpar os arquivos compilados com “mingw32-make clean” e recompilar o projeto com”mingw32-make rebuild”. Dessa forma, o mesmo “*Makefile*” permite o uso multiplataforma sem necessidade de alterações no código-fonte.

. Inicialmente o usuário poderá optar por utilizar o texto do grupo, já pré-definido pela especificação, ou informar o caminho do arquivo em específico de interesse. Caso opte pela primeira opção, nada mais deverá ser feito e basta continuar a execução, por outro lado, se ocorrer a segunda alternativa, o arquivo deve estar armazenado dentro da pasta “/*input*” e deve ser digitado o seu caminho, como por exemplo “input/arquivo.txt”.

4. Resultados finais

A execução do programa de criptoanálise culminou na obtenção integral do texto claro e na identificação da chave de substituição utilizada.

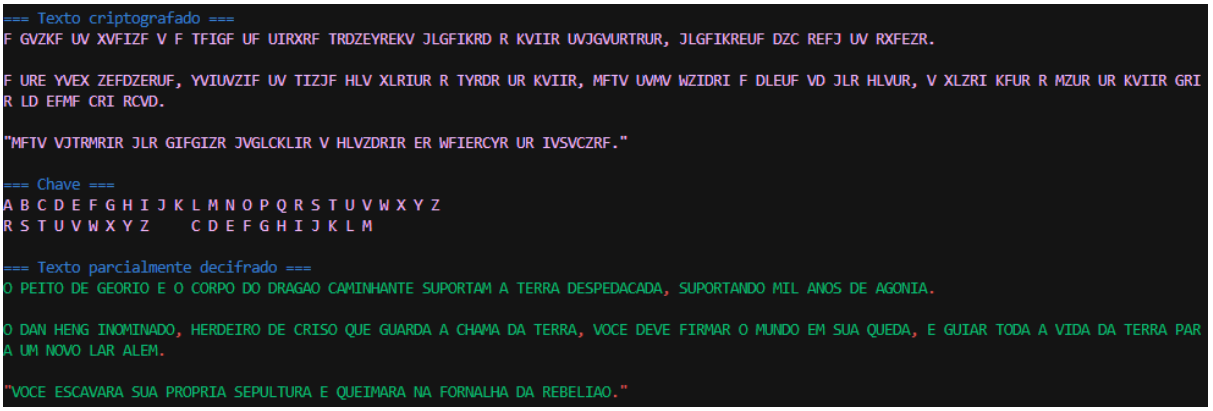


Figura X - Entrada criptografa e resolvida após operações

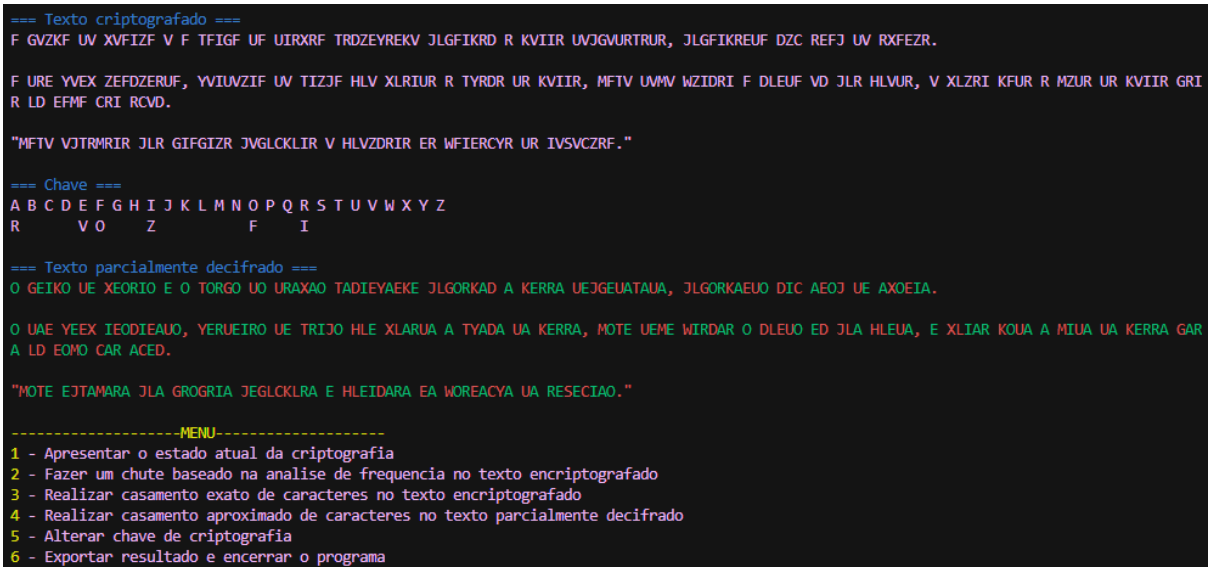


Figura X - execução, troca de chave parcial

5.Github

Segue o link do repositório Github:
https://github.com/kamizane/TP3---PAA

5. Conclusão

O desenvolvimento do sistema de criptoanálise consolidou a aplicação prática de algoritmos de manipulação de strings e estruturas de dados. O fluxo de execução é controlado por um loop interativo que integra as funções de análise

estatística e busca textual. A recuperação da informação original é viabilizada pela combinação da análise de frequência, que fornece a base inicial para o mapeamento, com os algoritmos de casamento de padrões. Especificamente, a implementação do algoritmo *Shift-And* aproximado demonstrou-se essencial para a identificação de palavras no texto parcialmente decifrado, permitindo inferir caracteres faltantes através da tolerância de erros por substituição.

A manipulação da chave de criptografia ocorre de forma dinâmica através da função associada à opção de alteração de chave, atualizando as estruturas de dados que armazenam o mapeamento entre o texto cifrado e o texto claro. A integridade da solução é verificada visualmente pelo usuário a cada iteração e, por fim, o processo é concluído com a exportação da chave encontrada para um arquivo externo, validando a eficácia das técnicas algorítmicas empregadas na decifragem das profecias.

6. Referências

1. Algoritimo BMH em java - Disponível em: https://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Text/Matching-Boyer-Moore2.html>. Acessado pela última vez em: 28/11/2025