Cifras de Transposição e Substituição

Lucas Gabriel de Oliveira Lima - 231003406

21 de abril de 2025

Resumo

Neste trabalho, serão apresentados conceitos de algoritmos de cifras de substituição e transposição. Com substituição, será trabalhado a encriptação e decriptação por Ceasar Cipher, e também ataques ao Shift Cipher de modo cipher-text only, tanto por força bruta quanto por distribuição de frequência. Com transposição, será trabalhado a encriptação e decriptação por Columnar Permutation Cipher, e também ataques a essa cifra de modo cipher-text only, tanto por força bruta quanto por distribuição de frequência. Além disso, será mostrado as partes principais do código-fonte da implementação desses algoritmos, utilizando a linguagem **C++**.

1 Introdução

Criptografia é uma forma de proteger dados, tentando modificar esses dados de forma que apenas o emissor e o receptor consigam saber o que ele de fato significa.

Nesse sentido, uma das formas de encriptar dados é a criptografia simétirica. Nela, as chaves de criptografia são privadas, ou seja, a chave que o emissor utiliza para criptografar o texto é a mesma para que no processo de decriptação o texto cifrado seja revertido para o texto original. Um ponto positivo dessa estratégia, se comparado à criptografia assimétrica, é que o algoritmo de criptografia tende a ser mais rápido. No entanto, caso alguma das parte deixe escapar a chave, a entidade que a obter teria mais facilidade para realizar a decriptação do dado.

Com isso, será trabalharemos estratégias de cifrar dados com criptografia simétrica, dando foco a duas formas de cifras principais: por **substituição** e por **transposição**.

Além disso, para cada um desse tipo de cifras, serão realizados ataques simulando o papel de alguém que estária no meio do caminho entre emissor e receptor, tentando quebrar essas cifras contendo apenas o texto cifrado (cipher-text only), ou seja, o atacante não tem conhecimento da chave privada.

Diante disso, utilizaremos duas abordagens principais de ataque para cada uma das diferentes formas de cifra: por **força bruta** e por **distribuição de frequência**. Por força bruta, tentamos todas as possíveis chaves e passamos por todas as diferentes combinações possíveis para o texto cifrado. Já por distribuição de frequência, partimos tendo em nosso conhecimento a frequência esperada de cada letra e cada digrama (duas letras juntas) no idioma que estamos trabalhando, e com isso calculamos uma pontuação com base na frequêcia das letras do texto que momentaneamente decriptamos, e com base nisso assumimos que o texto com melhor pontuação é o de maior probabilidade de corresponder ao texto original.

Para a quebra das cifras, foi-se utilizado os valores de frequência de letras e digramas no dicionário inglês. Como base, utilizamos o artigo *English Letter Frequency Counts: Mayzner Revisited or ETAOIN SRHLDCU* [May12]. No código-fonte, é possível encontrar esses valores no arquivo "Utils.hpp".

1.1 Instruções

O código-fonte do projeto pode ser encontrado no seguinte GitHub https://github.com/lucasdbr05 (link clicável direto para o repositório). Nele, há um arquivo README.md, no qual se encontra os comandos de compilação e execução do programa. A implementação dos requisitos pedidos no trabalho foi feito utilizando C++.

2 Cifra por substituição

Consiste em um método de encriptação onde cada caractere do texto original é substituido por um caractere diferente seguindo um padrão pré estabelecido, e de acordo com uma chave k. De inicio, cada letra do nosso alfabeto é mapeado a um respectivo número seguindo a ordem alfabética (A=0, B=1, C=2, ...).

Nesse processo utilizamos conceitos de aritmética modular. Algoritmos de **Shift Cipher**, dada uma chave k predefinada, texto original S, e texto resultante da encriptação C, cada caractere de C será dado pela seguinte fórmula:

$$C[i] = (S[i] + k) \mod 26$$

Tendo o texto cifrado C encripitado, para decriptografá-lo, basta aplicar a seguinte fórmula para cada caractere de C para obter o respectivo de S—:

$$S[i] = (C[i] - k + 26) \mod 26$$

Nessas fórmulas, (mod 26) corresponde ao resto positivo de um número inteiro na divisão por 26.

A seguir está a minha implementação da **Shift Cipher**:

```
#pragma once
#include <string>
using namespace std;

class ShiftCipher {
    private:
        int k;

        bool is_upper_case(char c) { return ('A' <= c && c <= 'Z'); }
        bool is_lower_case(char c) { return ('a' <= c && c <= 'z'); }

        char encrypt_char(char c) {
            if(is_upper_case(c)) {
                c = 'A' + (c - 'A' + k)%26;
            } else if(is_lower_case(c)) {
                c = 'a' + (c - 'a' + k)%26;
            }

            return c;
        }

        char decrypt_char(char c) {
            if(is_upper_case(c)) {
                c = 'A' + (c - 'A' - k + 26)%26;
            } else if(is_lower_case(c)) {
                c = 'a' + (c - 'a' - k + 26)%26;
            }
            return c;
        }


    public:
        ShiftCipher() {}

        ShiftCipher(int k) {
            this->k = k;
        }
        string encrypt(string message) {
            for(char &c: message) {
                c = encrypt_char(c);
            }
            return message;
        }
```

```
44
45          string decrypt(string message) {
46              for(char &c: message) {
47                  c =  decrypt_char(c);
48              }
49              return message;
50          }
51
52  };
```

Um caso especial da cifra de substituição se trata da **Ceasar Cipher**, que segue as mesmas especifições descritas acima, com a chave $k = 3$.

2.1 Análise de Complexidade

Nos métodos de encriptar (encrypt) e decriptar (decrypt) do código acima, podemos ver, que em ambos, se itera pelo texto, que é uma operação *O(n)*, e que nessas iterações é aplicada uma operação no caractere, que tem custo *O(1)*. Com isso, vemos que a complexidade de encriptar e decriptar por Shift Cipher tem complexidade *O(n)*.

2.2 Quebra por força bruta

Para uma cifra de substituição, temos 26 diferentes chaves possíveis, pois todas elas pertencem a Z_{26}.

Com isso, basta iterarmos por todas as chaves e aplicar a função de decriptar utilizando k no texto cifrado para obter um novo texto cifrado. Com isso, realizariamos 26 vezes uma operação de decriptar, o que já discutimos ser *O(n)*, o que resultaria em uma complexidade de *O(26*n)*. A implementação da quebra por força bruta pode ser encontrada a seguir.

```
1
2           vector<string> brute_force(string cipher_text) {
3               vector<string> res;
4               for(int k=0; k<26; k++) {
5                   shift_cipher = ShiftCipher(k);
6                   res.push_back(
7                       shift_cipher.decrypt(cipher_text)
8                   );
9               }
10              return res;
11          }
```

2.3 Quebra por distribuição de frequência

Para trabalharmos com distribuição de frequência, inicialmente verificamos a frequência das letras no texto cifrado, e então analisamos isso comparado à frequencia de letras esperada no nosso idioma.

Nessa analise, comparamos a letra mais frequente em nosso texto e, vamos comparando com as letras mais frequentes do idioma para, calcular o valor de k que seria esperado caso a letra mais frequente no texto cifrado correspondesse ao i-ésimo mais frequente no idioma. Então fazemos essa decripatação e calculamos o score do texto momentaneamente decifrado. Então, calculamos a pontuação para esses textos e consideramos o texto de melhor pontuação como o de maior probabilidade de corresponder ao texto original.

Esse processo é realizado comparando às 5 letras mais frequentes no idioma trabalhado, que corresponde $\lfloor\sqrt{26}\rfloor$, que foi o que retornou resultados mais satisfatórios nos testes realizados. Com isso, temos uma complexidade de *O(5*n)*.

A implementação da quebra por distribuição de frequência pode ser encontrada a seguir.

```
1           string frequency_distribution(string cipher_text) {
2               double best_score = utils.INF;
3               int best_k = -1;
4               string plain_text;
```

```
5
6           map<char, double> frequency = get_letters_frequency_in_text(
                cipher_text);
7           vector<pair<double, char>> sorted_frequency =
                get_letters_frequency_in_descending_order(frequency);
8
9           for(int i=0; i*i<26; i++) {
10              int k = calculate_key(sorted_frequency[0].second, i);
11              shift_cipher = ShiftCipher(k);
12              string current_plain_text = shift_cipher.decrypt(cipher_text);
13              double current_score = calculate_score(current_plain_text);
14              if(current_score < best_score) {
15                  best_score = current_score;
16                  plain_text = current_plain_text;
17                  best_k = k;
18              }
19          }
20          return plain_text;
21      }
```

Para realizar o cálculo da pontuação utilizamos a seguinte fórmula: $FE =$ Frequência esperada de uma letra no idioma (percentual) $FP =$ Frequência percebida de uma letra no texto (percentual) $L =$ Conjunto com as 26 letras do alfabeto

$$score = \sum_{l \in L} |FE[l] - FP[l]|$$

E consideramos um score melhor quanto menor o seu valor. A seguir está a implementação da função de cálculo de score:

```
1       double calculate_score(string& text) {
2           map<char, double> letters_frequency =
                get_letters_frequency_in_text(text);
3           double score = 0;
4
5           for(int i=0; i<26; i++) {
6               score += abs(letters_frequency['A'+i] - utils.
                    letter_percent_occurrence[i]);
7           }
8           return score;
9       }
```

2.4 Conclusões

A cifra por substituição se apresenta como um estratégia muito simples de criptografia, se aplicada isoladamente. Ela apresenta algumas fraquezas, como ser fácilmente quebrável pelo método de força bruta.

Além disso, observa-se que em textos grandes, geralmente compensa mais aplicar a análise por distribuição de frequência, pois por ser probabilistico, quanto maior o espaço-amostral (tamanho do texto) ele tende a ter mais chances de acertar. Enquanto para textos menores, a força bruta tende a valer mais a pena, pois o tempo para executar o algoritmo tende a ser razoávelmente bom e combre os casos de erro utilizando distribuição de frequência.

3 Cifra de Tranposição

Consiste é um método de encriptação que permuta as posições dos caracteres sem modificar os valores dos caracteres. Um exemplo de cifra por transposição é a **Columnar Permutation Cipher**.

Dado um texto S de tamanho n, e uma chave k, que é uma string sem caracteres repetidos, de tamanho x, a **Columnar Permutation Cipher** consiste em escrever esse S nas linhas de uma

matriz de $\lceil n/x \rceil$ linhas por x colunas, e então permutamos as colunas dessa matriz seguindo a ordem lexicográfica dos caracteres de k. Após essa permutação, por fim, itera-se ordenadamente por cada coluna adicionando no texto cifrado os caracteres daquela coluna.

Por exemplo, para uma entrada S = "EOFLUMINENSEFUTEBOLCLUBE", e com uma chave k = "FLUZAO", primeiramente é montado a seguinte matriz:
EOFLUM
INENSE
FUTEBO
LCLUBE

E então, seguindo a ordem de k (2, 3, 5, 6, 1, 4), teriamos a seguinte permutação de colunas:
UEOMFL
SINEEN
BFUOTE
BLCELU

E então, pegando os caracteres por coluna, teriamos como texto cifrado:
"USBBEIFLONUCMEOEFETLLNEU".

Para decriptar, basta fazer o processo de escrever S na matriz, mas escrevê-lo pelas colunas da matriz ordenadamente, e então aplicar a permutação nas colunas de acordo com k, e por fim iterar ordenadamente palas linhas da matriz adicionando cada caractere no que seria o texto original.

Para textos S tais que n não é divisível por x, realizamos um *padding*, adicionando caracteres à direita de S para tornar seu tamanho divisível por x.

A seguir está a minha implementação da **Columnar Permutation Cipher**:

```
#pragma once
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

class ColumnarPermutationCipher {
    private:
        string key;
        int key_size;
        vector<vector<char>> t_matrix;

        void permute_columns() {
            vector<vector<char>> new_matrix(key_size);
            vector<pair<char, int>> letter_by_column(key_size);
            for(int i=0; i<key_size; i++) {
                letter_by_column[i] = {key[i], i};
            }
            sort(letter_by_column.begin(), letter_by_column.end());

            for(int i=0; i<key_size; i++){
                new_matrix[i] = t_matrix[letter_by_column[i].second];
            }
            t_matrix = new_matrix;

            return;
        }
        void invert_permute_columns() {
            vector<vector<char>> new_matrix(key_size);
            vector<pair<char, int>> letter_by_column(key_size);
            for(int i=0; i<key_size; i++) {
                letter_by_column[i] = {key[i], i};
            }
```

```
36              sort(letter_by_column.begin(), letter_by_column.end());
37
38              for(int i=0; i<key_size; i++){
39                  new_matrix[letter_by_column[i].second] = t_matrix[i];
40              }
41              t_matrix = new_matrix;
42
43              return;
44          }
45
46          void padding(string& s){
47              int padding_size = (key_size - s.size()%key_size) % key_size;
48              while(padding_size--) s += ' ';
49          }
50
51          string build_cipher_text(int rows) {
52              string text = "";
53              for(int i=0; i<key_size; i++){
54                  for(int j=0; j<rows; j++){
55                      text+= t_matrix[i][j];
56                  }
57              }
58              return text;
59          }
60
61          string build_plain_text(int rows) {
62              string text = "";
63              for(int j=0; j<rows; j++){
64                  for(int i=0; i<key_size; i++){
65                      text+= t_matrix[i][j];
66                  }
67              }
68              return text;
69          }
70
71      public:
72          ColumnarPermutationCipher(){}
73
74          ColumnarPermutationCipher(string key) {
75              this->key = key;
76              this->key_size = key.size();
77          }
78
79          string encrypt(string plain_text) {
80              int rows = plain_text.size()/key_size + (plain_text.size()%
                    key_size > 0);
81              t_matrix = vector(key_size, vector<char>(rows, ' '));
82              padding(plain_text);
83
84              for(int i=0; i<rows; i++){
85                  for(int j=0; j<key_size; j++){
86                      int current_char = i*key_size + j;
87
88                      t_matrix[j][i] = plain_text[current_char];
89                  }
90              }
91
92              permute_columns();
93              return build_cipher_text(rows);
94          }
95
96          string decrypt(string cipher_text) {
```

```
            int rows = cipher_text.size()/key_size + (cipher_text.size()%
                key_size > 0);
            t_matrix = vector(key_size, vector<char>(rows, ' '));
            padding(cipher_text);

            int current_char =0;
            for(int j=0; j<key_size; j++){
                for(int i=0; i<rows; i++){
                    t_matrix[j][i] = cipher_text[current_char];
                    current_char++;
                }
            }

            invert_permute_columns();
            return build_plain_text(rows);
        }
};
```

Nesse projeto, para as etapas de encriptação e decriptação, utilizamos uma chave $k =$ "FLUZAO".

3.1 Análise de Complexidade

Para um texto S de tamanho n e uma chave k de tamanho x, no processo de encriptação, basta iterarmos pelos caracteres de S, modo que a matriz será montado seguindo a seguinte fórmula, que tem custo *O(n)*:

$$matrix[\lfloor i/x \rfloor][i\%x] = S[i]$$

Após isso, é realizado uma reodenação de colunas. Na implementação, temos que essa reordenação tem complexidade *O(x*log(x))*, no entanto, como a chave x tem um tamanho relativamente pequeno ($2 \leq x \leq 8$) comparado a n, consideraremos esse valor despezível. Portanto a encriptação tem complexidade *O(n)*.

Para decriptar, são realizadas operações simiralares às de encriptar, mas em uma ordem difrente, o que não modifica a complexidade. Logo, o processo de decriptar também têm complexidade *O(n)*.

3.2 Quebra por força bruta

Para realizar uma quebra por força bruta, sabemos que a chave pode variar k pode ter um tamanho x tal que $2 \leq x \leq 8$. Foi-se utilizado esses limites, porque para chaves tal que $x8$, o tempo de execução do algoritmo e uso de memória passa a se tornar muito grande.

Portanto, inicialmente iteramos por todos os possíveis valores de x, e para uma chave desse tamanho (com letras distintas), realizamos todas as possíveis permutações cujos caracteres gerem uma ordenação distinta para as colunas no processo de decriptar.

Com isso, temos que essa geração de chaves distintas tem complexidade *O(x!)*. Como cara cada chave realiza uma decriptação, que já discutimos ser *O(n)*, temos que a complexidade total desse processo é *O(n*(x!))*.

A implementação da quebra por força bruta pode ser encontrada a seguir.

```
        vector<string> brute_force(string cipher_text) {
            vector<string> res;
            int n = cipher_text.size();
            int max_cols = 8;

            for(int key_size=2; key_size<= min(max_cols, 26); key_size++) {
                string key = utils.alphabet.substr(0, key_size);

                do {
                    permutation_cipher = ColumnarPermutationCipher(key);

                    res.push_back(
                        permutation_cipher.decrypt(cipher_text)
                    );
```

```
                } while(
                    next_permutation(key.begin(), key.end())
                );
            }
            return res;
        }
```

3.3 Quebra por distribuição de frequência

Para trabalharmos com distribuição de frequência, inicialmente verificamos a frequência das digramas no texto cifrado, e então analisamos isso comparado à frequencia de digramas esperada no nosso idioma. Para essa cifra, foi escolhido trabalhar com os digramas do idioma, pois eles traziam um resultado mais satisfatório que trabalhando com as letras apenas.

Com isso, também foi necessário passar por todas as chaves possíveis para uma string de tamanho x, mas a distribuição de frequência foi util para obter a saída que mais provavelmente corresponde ao texto original após realizar o cálculo da pontuação. Com isso, a quebra por distribuição de frequência também tem complexidade $O(n * (x!))$.

A implementação da quebra por distribuição de frequência pode ser encontrada a seguir.

```
        string frequency_distribution(string cipher_text) {
            double best_score = utils.INF;
            string best_key;
            string plain_text;

            int n = cipher_text.size();
            int max_cols = 8;

            for(int key_size=2; key_size<=min(max_cols, 26); key_size++) {
                string key = utils.alphabet.substr(0, key_size);

                do {
                    permutation_cipher = ColumnarPermutationCipher(key);
                    string current_plain_text = permutation_cipher.decrypt(
                        cipher_text);

                    double current_score = calculate_score(current_plain_text)
                        ;

                    if(current_score < best_score) {
                        best_score = current_score;
                        plain_text = current_plain_text;
                        best_key = key;
                    }
                } while(
                    next_permutation(key.begin(), key.end())
                );

            }
            return plain_text;
        }
```

Para realizar o cálculo da pontuação utilizamos a seguinte fórmula:
FE = Frequência esperada de um digrama no idioma (percentual)
FP = Frequência percebida de um digrama no texto (percentual)
B = Conjunto com os 676 digramas no alfabeto

$$score = \sum_{bi \in B} |FE[bi] - FP[bi]|$$

E consideramos um score melhor quanto menor o seu valor. A seguir está a implementação dessa função de cálculo de score:

```
1         double calculate_score(string& text) {
2             map<string, double> bigrams_frequency =
                  get_bigrams_frequency_in_text(text);
3
4             double score = 0;
5
6             for(int i=0; i<26; i++){
7                 for(int j=0; j<26; j++){
8                     string bigram; bigram += 'A' + i; bigram+= 'A'+j;
9                     score += abs(bigrams_frequency[bigram] - utils.
                          bigrams_percent_occurrence[bigram[0]-'A'][bigram[1]-'A
                          ']);
10                }
11            }
12
13            return score;
14        }
```

3.4 Conclusões

Ademais, vale também para a cifra por transposição que em textos grandes, geralmente compensa mais aplicar a análise por distribuição de frequência, pois da mesma forma é probabilistico. Enquanto para textos menores geralmente vale mais a pena utilizar a quebra por força bruta.

Referências

[May12] Mark Mayzner. English letter frequency counts: Mayzner revisited or etaoin srhldcu. 2012.