Trabalho 1 Segurança Computacional

1st Eduardo Ferreira Marques Cavalcante
Departamento de ciência da computação
Universidade de Brasília
Brasília, Brasil
202006368@aluno.unb.br

2nd Pedro Rodrigues Diógenes Macedo
Departamento de ciência da computação)
Universidade de Brasília
Brasília, Brasil
211042739@aluno.unb.br

I. Introdução

A cifra de Vigenère, uma das técnicas de criptografia clássica mais conhecidas, tem sido objeto de estudo contínuo devido à sua resiliência aos métodos tradicionais de criptoanálise. Neste trabalho, exploramos a aplicação dessa cifra em mensagens tanto em inglês quanto em português, oferecendo a capacidade de cifrar e decifrar mensagens de acordo com uma chave fornecida pelo usuário. Além disso, abordaremos um ataque à cifra de Vigenère, primeiramente determinando o tamanho da chave (senha) e, posteriormente, a recuperação completa da chave para decifrar a mensagem fornecida pelo usuário. Nossos métodos de criptoanálise se baseiam em análise de frequência, uma abordagem que se revela fundamental na quebra dessa cifra clássica, proporcionando uma compreensão aprofundada de seus mecanismos e uma contribuição valiosa para o campo da segurança da informação. Este estudo visa aprimorar nossa compreensão das vulnerabilidades da cifra de Vigenère e oferecer insights para aprimorar a segurança das comunicações criptografadas.

O trabalho foi realizado utilizando a linguagem de programação Python com as bibliotecas padrões como *string* e *regex*. A implementação pode ser encontrada em [1]

II. Metodologia

A. Codificação e decodificação

Como dito anteriormente, a primeira parte do trabalho se caracteriza pela cifração e decifração de uma mensagem, em português ou inglês, dada uma chave do usuário .

A palavra chave é repetida várias vezes ao longo da mensagem para gerar uma palavra chave tão grande quanto a mensagem, por exemplo, se a mensagem for "HELLO"e a chave for "KEY", a chave repetida será "KEYKE". As letras da mensagem são deslocadas de acordo a letra correspondente na palavra chave utilizando cifras de César [4]. Assim, com o alfabeto de tamanho 26, e deslocamento máximo de 25, podemos representar a codificação da seguinte forma: E_i = Letra cifrada, P_i = Letra original, K_i = Letra da palavra-chave, D_i = Letra decifrada [2]

$$E_i = (P_i + K_i) \mod 26 \tag{1}$$

Para a parte de decodificação, a qual é a inversa da codificação, em que devemos retirar o deslocamento, podemos representar da seguinte forma:

$$D_i = (E_i - K_i) \mod 26 \tag{2}$$

Essa representação em código:

```python
# Cifrar a message de acordo com a password_key
def vigenere_crypt(message: str, password_key: str):
    message = message.upper()

    resultado = ""


    k = check_key(message, password_key)
    i = 0

    for letra in message:
        if letra in ALFABETO:

            resultado += ALFABETO[((ord(letra) + ord(k[i])) % 26)]
            i += 1
        else:
            resultado += letra


    return resultado

# Decifrar a message de acordo com a password_key
def vigenere_decrypt(message: str, password_key: str):

    message = message.upper()
    resultado = ""

    k = check_key(message, password_key)
    i = 0

    for letra in message:
        if letra in ALFABETO:
            resultado += ALFABETO[(ord(letra) - ord(k[i]) % 26 + 26) % 26]
            i += 1
        else:
            resultado += letra

    return resultado
```

Figura 1. Cifração em cima, decifração em baixo.

$check_key$ é a função que repete as letras da chave para caber o tamanho completo da mensagem.

B. Quebrando a cifra de Vigenère

Na segunda parte desse trabalho o objetivo era quebra uma cifra tendo somente ela como informação. Para isso, utilizamos a seguinte estratégia que é resumida em dois passos.

O primeiro consiste em encontrar o tamanho da chave que foi utilizada para cifrar a mensagem. Com esse intuito, percorremos a cifra procurando por trigramas repetidos e calculamos a distância entre elas [3]. Com essa distância temos alguns tamanhos de chaves possíveis que são todos os divisores da distância calculada (no caso do nosso programa, encontramos chaves de tamanho de no mínimo 2 e no máximo 20). Assim, incrementamos o fator de tamanho de acordo com esses divisores encontrados. Após calcular todas as distâncias entre trigramas repetidos e seus respectivos divisores, a função $keyLenght$ imprime os três tamanhos mais prováveis da chave ter, ou seja, os tamanhos com os maiores fatores.

Como essa técnica não é muito exata para determinar qual é o tamanho real da chave, optamos por deixar o usuário decidir qual chave ele irá utilizar, dados os fatores calculados anteriormente.

A implementação da função $keyLenght$ está a seguir:

```python
#mostra os três tamanhos mais possíveis de serem a chave
def key_length(cipher: str):

    newCipher = ajeita_cifra(cipher)
    maiorFator = (-1,-1)
    trigramas = {}
    tamanhos_possiveis ={2:0, 3:0, 4:0, 5:0, 6:0, 7:0, 8:0, 9:0,
                         10:0, 11:0, 12:0, 13:0, 14:0, 15:0, 16:0,
                         17:0, 18:0, 19:0, 20:0}

    i = 0
    j = 3

    while j <= len(newCipher):

        if newCipher[i:j] in trigramas.keys():
            espaco = i - trigramas[newCipher[i:j]]

            t_possiveis = dividores_ate_20(espaco)

            for a in t_possiveis:
                tamanhos_possiveis[a] += 1
                #if 5 <= a <= 8 and tamanhos_possiveis[a] > maiorFator[1]:
                #    maiorFator = (a, tamanhos_possiveis[a])

        trigramas[newCipher[i:j]] = i

        i += 1
        j += 1

    tamanhos_possiveis = dict(sorted(tamanhos_possiveis.items(), key=lambda item: item[1], reverse=True))

    print("Os tamanhos mais prováveis encontrados foram:")

    i = 1
    for tamanho in tamanhos_possiveis.keys():
        print("Tamanho:", tamanho, "--", "Fator:", tamanhos_possiveis[tamanho])
        if i == 3:
            break
        i += 1
```

Figura 2. Calculo dos fatores dos tamanhos possíveis de chave.

A função $ajeitaCifra$ retorna a mesma cifra, porém sem os símbolos que não são utilizados para cifrar, já que eles não são relevantes para encontrar o tamanho da chave.

A função $divisoresAte20$ recebe um inteiro e retorna uma lista com os seus divisores de 2 até 20.

Agora, tendo o tamanho da chave, vamos para o segundo e último passo para encontrar de fato a chave. Para isso, utilizaremos as probabilidades de cada letra aparecer em uma palavra, frase ou texto dependendo da língua a qual a mensagem está. No caso da nossa implementação, conseguimos trabalhar com a língua inglesa e a portuguesa do Brasil. Para utilizar essas probabilidades, precisamos calcular as probabilidades a cada letra da chave e comparar com as probabilidades da língua. Ou seja, caso o tamanho da chave seja 3 e queremos encontrar a primeira letra da chave, pegamos as letras nas posições 1, 4, 7, ... até o fim da cifra, e calculamos as probabilidades das letras encontradas. Como nessas posições foram feitas o mesmo deslocamento, basta agora "alinhar"as probabilidades encontradas com as probabilidades da língua e, assim, encontramos o quanto foi deslocado, o que corresponde a letra da chave. E fazemos isso para cada letra da chave.

Caso o usuário tenha escolhido um tamanho de chave errado e apresentar uma decifração errada da cifra, o usuário pode escolher novamente o tamanho da chave quantas vezes quiser até achar que foi o suficiente. Para cada chave encontrada, o programa decifra a mensagem e a mostra na tela.

A função que $foundKey$ recebe a cifra, o tamanho da chave e a língua a qual será utilizada, e retora a chave encontrada. Já a função $foundLetter$ recebe a cifra, o índice da letra da chave que será encontrada, o tamanho da chave e a língua, e retorna a letra da chave daquele respectivo índice.

Essas funções estão descritas na seguinte implementação:

```python
def found_key(cipher: str, key_len: int, language: str):

    newCipher = ajeita_cifra(cipher)

    key = ""

    for i in range(key_len):

        key += found_letter(newCipher, i, key_len, language)

    return key
```

Figura 3. Função $foundKey$.

```python
#encontra a letra de um determinado indice da chave
def found_letter(cipher: str, indice: int, key_len: int, language: str):
    eng_probabilities = [8.167, 1.492, 2.782, 4.253, 12.702, 2.228, 2.015,
                         6.094, 6.966, 0.153, 0.772, 4.025, 2.406, 6.749, 7.507,
                         1.929, 0.095, 5.987, 6.327, 9.056, 2.758, 0.978, 2.360,
                         0.150, 1.974, 0.074]
    pt_probabilities = [14.63, 1.04, 3.88, 4.99, 12.57, 1.02, 1.30, 1.28, 6.18,
                         0.40, 0.02, 2.78, 4.74, 5.05, 10.73, 2.52, 1.20, 6.53,
                         7.81, 4.34, 4.63, 1.67, 0.01, 0.21, 0.01, 0.47]

    letter_counts = {}
    letter_frequencies = []
    count_letters = 0
    ajuste = 0
    menor_diferenca = 1e9

    for i in range(indice, len(cipher), key_len):
        letter_counts[cipher[i]] = letter_counts.get(cipher[i], 0) + 1
        count_letters += 1

    for i in ALFABETO:
        letter_frequencies.append((letter_counts.get(i, 0) / count_letters) * 100)

    for i in range(26):
        diferenca = 0
        for j in range(26):
            if language == "EN":
                diferenca += abs(eng_probabilities[j] - letter_frequencies[j])
            else:
                diferenca += abs(pt_probabilities[j] - letter_frequencies[j])
        if diferenca < menor_diferenca:
            menor_diferenca = diferenca
            ajuste = i

        letter_frequencies = shift_lista(letter_frequencies)

    return chr(ord('A') + ajuste)
```

Figura 4. Função $foundLetter$.

A função $shiftLista$ recebe uma lista e retorna a mesma lista, mas com o primeiro elemento sendo o último.

III. Conclusões

Neste trabalho, exploramos conceitos e técnicas de processamento de imagem para segmentação, utilizamos aprendizado de máquina para definir as operações morfológicas necessárias para a separação entre texto e imagem. Acreditamos que os resultados adquiridos foram parecidos com o do artigo. Como foi dito no artigo, tais procedimentos realizados são ideias para um pré-processamento das imagens dos mangás, abrindo espaço para outros algoritmos serem aplicados para a separação completa dos textos e diminuição de ruído.

Referências

[1] E. Ferreira, R. Pedro, “Projeto 1 de Segurança Computacional 2023.2,” GitHub, Oct. 02, 2023. https://github.com/EduardoFMC/SC-Vigenere

[2] GeeksforGeeks, “Vigenère Cipher - GeeksforGeeks,” GeeksforGeeks, Oct. 07, 2016. https://www.geeksforgeeks.org/vigenere-cipher/

[3] B. Veitch, “Cryptography - Breaking the Vigenere Cipher,” YouTube. Mar. 18, 2014. Available: https://www.youtube.com/watch?v=P4z3jAOzT9I&abchannel=BrianVeitch

[4] “Vigenere CIpher,” www.youtube.com. https://www.youtube.com/watch?v=SkJcmCaHqS0&ab_channel=Udacity .