Criptografia clássica

Trabalho 1, Grupo 3, Modelo 6

Gustavo Zambonin, Ranieri Schroeder Althoff
Introdução à Criptoanálise (UFSC-INE5451)

Questão 1

O processo utilizado para obter as frequências com relação à língua inglesa é listado abaixo:

- Lembrando: os cálculos realizados mostram que, desde que exista uma boa quantidade de textos (um livro já é o suficiente para construção do vocabulário diversificado), as frequências sempre respeitarão um intervalo intrínseco. Os valores descritos representam apenas uma aproximação das frequências da língua inglesa moderna. Comparações podem ser feitas através de dados presentes aqui, assim como resultados de outros arquivos analisados por este código.
- Obteve-se uma edição completa do livro *Les Misérables*, traduzida para inglês, disponível gratuitamente através do *Project Gutenberg*.
- O código utilizado para processar o arquivo é comentado logo abaixo.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from pprint import pprint
from sys import argv

allowed_letters = list(map(chr, range(97, 123)))
digraph_index = {}
letter_index = {l: 0 for l in allowed_letters}


def process_text(text):
    for word in text.lower().split():
        if len(word) > 1:
            digraphs = tuple(word[x:x + 2] for x in range(len(word) - 1))
            for dig in digraphs:
                if dig[0] in allowed_letters and dig[1] in allowed_letters:
                    letter_index[dig[0]] += 1
                    digraph_index[dig] = digraph_index[dig] + 1 \
                        if dig in digraph_index else 1

        if word[-1] in allowed_letters:
            letter_index[word[-1]] += 1


def process_data():
    total = sum(digraph_index.values())

    for let in letter_index:
        letter_index[let] /= total

    for dig in digraph_index:
        digraph_index[dig] /= total

    return sorted(letter_index.items(), key=lambda o: o[1], reverse=True), \
        sorted(digraph_index.items(), key=lambda o: o[1], reverse=True)[:10]


if __name__ == '__main__':

    for arg in argv[1:]:
        with open(arg) as raw:
            for line in raw:
                process_text(line)

    if len(argv) > 1:
        pprint(process_data())
```

{gustavo.zambonin,ranieri.althoff}@grad.ufsc.br src/

- Linha 13: garante que todas as letras serão convertidas para suas versões minúsculas na lista de palavras.
- Linha 15: cria todos os digramas possíveis para uma palavra de tamanho válido.
- Linha 18: adiciona na contagem de letras a primeira letra de cada digrama, desconsiderando repetições.
- Linhas 19–20: trata de adicionar um digrama novo ou incrementar a contagem de um existente.
- Linhas 22–23: a última letra de cada palavra é finalmente tratada, visto que esta nunca será a primeira letra de um digrama, e portanto, nunca seria contada.
- Linhas 35–36: funções *inline* que ordenam as estruturas de dados por suas chaves numéricas.
- Linha 41: habilita a entrada de múltiplos arquivos de texto para análise conjunta.

1. A frequência relativa do alfabeto A-Z em inglês segue abaixo:

e	$\approx 12.900494344206656\%$	**d**	$\approx 4.045562489940431\%$	**b**	$\approx 1.5355173283924984\%$
t	$\approx 9.228622978258158\%$	**l**	$\approx 3.9625337394346226\%$	**y**	$\approx 1.3292832976368709\%$
a	$\approx 8.495812955201934\%$	**u**	$\approx 2.7658412228484253\%$	**v**	$\approx 1.0720053151572956\%$
o	$\approx 7.534992892953012\%$	**c**	$\approx 2.7387961613301724\%$	**k**	$\approx 0.5378550590525007 5\%$
i	$\approx 7.176120980751902\%$	**m**	$\approx 2.401041625930986\%$	**j**	$\approx 0.2396595862393739\%$
h	$\approx 7.16191923611903\%$	**w**	$\approx 2.2774658653863966\%$	**x**	$\approx 0.15218507219631834\%$
n	$\approx 6.63501392800086\%$	**f**	$\approx 2.264293232683442\%$	**q**	$\approx 0.10402263387614184\%$
s	$\approx 6.014994572463682\%$	**g**	$\approx 1.8497669473123507\%$	**z**	$\approx 0.07668942101751178\%$
r	$\approx 5.7699836041887324\%$	**p**	$\approx 1.7295255094206964\%$		

2. A frequência relativa de digramas na língua inglesa é apresentada abaixo, filtrando apenas os 10 resultados mais comuns. O conjunto total não será mostrado pois, além de ser demasiadamente extenso, não existe, neste arquivo, uma representação de todas as palavras presentes no vocabulário moderno da língua inglesa, utilizadas de modo contextualmente válido, impossibilitando a construção de todos os digramas válidos.

th	$\approx 4.276658660240072\%$
he	$\approx 4.0442594323565895\%$
in	$\approx 2.2615934381157863\%$
er	$\approx 2.111202678574571\%$
an	$\approx 1.9470851796970966\%$
re	$\approx 1.8199745678214054\%$
ed	$\approx 1.4616211932810376\%$
on	$\approx 1.4461346155683348\%$
ha	$\approx 1.4253685227263014\%$
at	$\approx 1.386953765023363\%$

Questão 2

- Seja $y = DES(c, K)$ a função que codifica o texto c com a chave K. c, por sua vez, será dividido em duas metades: L_i, a parte esquerda, e R_i, a parte direita, para cada rodada i. Portanto, existe uma função $DES(L_iR_i, K)$ que caracteriza uma rodada da rede de Feistel.
- Hipótese: $DES(L_iR_i, K)$ é complemento de $DES(\overline{L_iR_i}, \overline{K})$ e vice-versa.
- Em $DES(L_iR_i, K)$:
  - $L_{i+1} = R_i$
  - $R_{i+1} = L_i \oplus f(R_i, K_i)$
- Em $DES(\overline{L_iR_i}, \overline{K})$:
  - $\overline{L_{i+1}} = \overline{R_i}$
  - $\overline{R_{i+1}} = \overline{L_i} \oplus f(\overline{R_i}, \overline{K_i})$
  - A função $f(R_i, K_i)$ combina os bits de R_i e K_i utilizando o operador lógico $\oplus$. Como a operação de $\oplus$ é comutativa e associativa, $f(R_i, K_i) = f(\overline{R_i}, \overline{K_i})$. (É importante lembrar que a expansão de bits em R_i não afeta seu complemento pois apenas duplica bits e troca-os de lugar.)
  - Portanto, segue que $\overline{R_{i+1}} = \overline{L_i} \oplus f(R_i, K_i)$.
  - Porém, deseja-se obter $\overline{R_{i+1}} = \overline{L_i \oplus f(R_i, K_i)}$ a partir dos resultados acima. Isso pode ser mostrado através da tabela verdade entre os termos:

$\overline{L_i}$	$f(R_i, K_i)$	$\overline{L_i \oplus f(R_i, K_i)}$	$\overline{L_i} \oplus f(R_i, K_i)$
0	0	1	1
0	1	0	0
1	0	0	0
1	1	1	1

- Todas as operações que compõem as funções $DES(L_iR_i, K)$ e $DES(\overline{L_iR_i}, \overline{K})$ foram provadas complementos mútuos de acordo com as regras impostas, portanto a hipótese é verdadeira.
- Por fim, por indução fraca, prova-se que o complemento cifrado funciona em todas as rodadas do DES, de número arbitrário, usando o procedimento acima.

□

Questão 3

A máquina Enigma apresentada tem as seguintes características:

- *Steckerbrett* sem cabos conectados;
- *Ringstellung* com a configuração **AAA**;
- O texto cifrado é datado de maio de 1942.

Destas informações, é possível inferir os seguintes aspectos para início da criptoanálise:

- Não é possível aplicar um método que utilize um ataque de texto em claro conhecido (Bomba de Turing).
- Não é possível aplicar um método que dependa de mais de um texto cifrado (*females* de Zygalski).
- Não é possível aplicar um método que aproveite da repetição da chave secundária, já que isto não acontecia mais em 1942 (folhas perfuradas de Zygalski, método das características).
- Existem cinco modelos de rotor (*walzen* I, II, III, IV e V, presentes nas Enigma I e M3), visto que não se foi discutido em profundidade o funcionamento dos rotores de *notch* duplo utilizados pela Marinha alemã.
- Portanto, o número de testes para descobrir a ordem e escolha dos rotores (*walzenlage*) é $\binom{5}{3} = 60$.
- Ademais, deseja-se descobrir também o *grundstellung* (posição em que os rotores foram encaixados inicialmente). Existem 26^3 possibilidades, visto que todos os rotores podem ser encaixados com todas as letras apontando para a janela da carcaça da máquina.
- Logo, o número de possibilidades é igual a $60 \times 26^3 = \mathbf{1054560}$.
- O texto cifrado fornecido tem cerca de 400 caracteres. De acordo com [2], existe uma probabilidade muito baixa de que o texto com o maior IC não seja o correto para este número de caracteres. Deste modo, essa estratégia foi escolhida. É válido comentar que não se obteve sucesso em construir um método não-trivial caso existissem múltiplos textos de tamanho pequeno cifrados com diversas chaves.

O algoritmo que simula a Enigma é apresentado abaixo:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from sys import argv

alphabet = list(map(chr, range(65, 91)))
desloc = alphabet[1:] + alphabet[:1]
shift_up = {alphabet[i]: desloc[i] for i in range(len(alphabet))}


class Steckerbrett:
    def __init__(self, *args):
        self.map = {}

        for arg in args:
            if arg[0] in self.map or arg[1] in self.map:
                raise KeyError('Same letter used twice in plugboard')

            self.map[arg[0]] = arg[1]
            self.map[arg[1]] = arg[0]

    def swap(self, letter):
        if letter in self.map:
```

```
            return self.map[letter]
        return letter


class Umkehrwalze:
    def __init__(self, wiring):
        assert isinstance(wiring, str)
        self.wiring = wiring

    def reflect(self, letter):
        return self.wiring[alphabet.index(letter)]


class Walzen:
    def __init__(self, index, grundstellung, ringstellung):
        if index == 'I':
            self.wiring = 'EKMFLGDQVZNTOWYHXUSPAIBRCJ'
            self.notch = 'Q'
        elif index == 'II':
            self.wiring = 'AJDKSIRUXBLHWTMCQGZNPYFVOE'
            self.notch = 'E'
        elif index == 'III':
            self.wiring = 'BDFHJLCPRTXVZNYEIWGAKMUSQO'
            self.notch = 'V'
        elif index == 'IV':
            self.wiring = 'ESOVPZJAYQUIRHXLNFTGKDCMWB'
            self.notch = 'J'
        elif index == 'V':
            self.wiring = 'VZBRGITYUPSDNHLXAWMJQOFECK'
            self.notch = 'Z'

        if isinstance(self.notch, str) and len(self.notch) == 1:
            self.notch = alphabet.index(self.notch)

        if isinstance(ringstellung, str) and len(ringstellung) == 1:
            self.ringstellung = alphabet.index(ringstellung)

        if isinstance(grundstellung, str) and len(grundstellung) == 1:
            self.grundstellung = alphabet.index(grundstellung)

    def wire(self, forward):
        permut = {}
        for i in range(len(self.wiring)):
            permut[alphabet[i]] = self.wiring[i]
        if not forward:
            return {v: k for k, v in permut.items()}
        return permut

    def move(self, char, steps, permutation):
        if steps < 0:
            permutation = {v: k for k, v in permutation.items()}
            steps *= -1
        for i in range(steps):
            char = permutation[char]
        return char


class Enigma:
    def __init__(self, rotors, plugboard=None):
        assert plugboard is None or isinstance(plugboard, Steckerbrett)

        assert isinstance(rotors, tuple)
        for index in range(len(rotors)):
            assert isinstance(rotors[index], Walzen)

        self.plugboard = plugboard
        self.rotors = rotors
        self.reflector = Umkehrwalze(wiring='YRUHQSLDPXNGOKMIEBFZCWVJAT')

    def cipher(self, message):
        assert isinstance(message, str)

        message = message.upper()
        ciphered = ''

        p1 = self.rotors[0].grundstellung
        p2 = self.rotors[1].grundstellung
        p3 = self.rotors[2].grundstellung

```

```
103         n1 = self.rotors[0].notch
104         n2 = self.rotors[1].notch
105 
106         r1 = self.rotors[0].ringstellung
107         r2 = self.rotors[1].ringstellung
108         r3 = self.rotors[2].ringstellung
109 
110         m1 = (n1 - p1) % 26
111         m2 = m1 + 26 * ((n2 - p2 - 1) % 26) + 1
112 
113         i1 = p1 - r1 + 1
114 
115         for j in range(len(message)):
116             letter = message[j]
117 
118             k1 = int((j - m1 + 26) / 26)
119             k2 = int((j - m2 + 650) / 650)
120 
121             i2 = p2 - r2 + k1 + k2
122             i3 = p3 - r3 + k2
123 
124             letter = self.rotors[0].move(letter, i1 + j, shift_up)
125             letter = self.rotors[0].move(letter, 1, self.rotors[0].wire(True))
126             letter = self.rotors[0].move(letter, -i1 - j, shift_up)
127             letter = self.rotors[1].move(letter, i2, shift_up)
128             letter = self.rotors[1].move(letter, 1, self.rotors[1].wire(True))
129             letter = self.rotors[1].move(letter, -i2, shift_up)
130             letter = self.rotors[2].move(letter, i3, shift_up)
131             letter = self.rotors[2].move(letter, 1, self.rotors[2].wire(True))
132             letter = self.rotors[2].move(letter, -i3, shift_up)
133             letter = self.reflector.reflect(letter)
134             letter = self.rotors[2].move(letter, i3, shift_up)
135             letter = self.rotors[2].move(letter, 1, self.rotors[2].wire(False))
136             letter = self.rotors[2].move(letter, -i3, shift_up)
137             letter = self.rotors[1].move(letter, i2, shift_up)
138             letter = self.rotors[1].move(letter, 1, self.rotors[1].wire(False))
139             letter = self.rotors[1].move(letter, -i2, shift_up)
140             letter = self.rotors[0].move(letter, i1 + j, shift_up)
141             letter = self.rotors[0].move(letter, 1, self.rotors[0].wire(False))
142             letter = self.rotors[0].move(letter, -i1 - j, shift_up)
143 
144             ciphered += letter
145 
146         return ciphered
147 
148 
149 if __name__ == '__main__':
150     # INSERT FROM RIGHT TO LEFT LIKE NORMAL ENIGMA
151     rotors = (Walzen(index='I', grundstellung='C', ringstellung='A'),
152               Walzen(index='II', grundstellung='S', ringstellung='A'),
153               Walzen(index='III', grundstellung='F', ringstellung='A'))
154 
155     if len(argv) == 2:
156         print(Enigma(rotors=rotors).cipher(argv[1]))
```

- Linhas 6–8: Definição do deslocamento em relação ao alfabeto na sua ordem normal.
- Linhas 11–25: Classe que define a *plugboard*, não utilizada neste programa, que contém apenas o método de troca de letras.
- Linhas 28–34: Classe que define um refletor e sua permutação simples de acordo com a definição.
- Linhas 37–78: Classe que define um rotor, levando em conta seu *ringstellung* e *grundstellung*, além do seu nome, para que sejam atribuídas suas características intrínsecas. O método *wire* constrói as permutações de cada rotor. O método *move* guia as letras ao passo dos rotores.
- Linhas 81–146: A classe de nível mais alto, que define a Enigma, sua composição e seu propósito, o método *cipher*, que implementa matematicamente a lógica de movimentação de rotores.
- Linhas 148–156: Uma instância da máquina com os rotores certos para a mensagem cifrada (mais abaixo) é criada, que consegue cifrar qualquer texto enviado por linha de comando com estas configurações.

O algoritmo responsável por analisar o IC de cada possível texto é mostrado abaixo:

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3 
```

```
4  from enigma import Enigma, Walzen
5  from itertools import permutations, product
6  from sys import argv
7
8  rotors = ('I', 'II', 'III', 'IV', 'V')
9  _upper = list(map(chr, range(65, 91)))
10 rot_comb = list(permutations(rotors, 3))
11 let_prod = list(product(_upper, repeat=3))
12
13
14 def index_of_coincidence(text):
15     def freq(i):
16         c = text.count(i)
17         return c * (c - 1)
18
19     l = len(text)
20     return sum(map(freq, _upper)) / (l * (l - 1))
21
22
23 def test_rotors(text):
24     for i, j, k in rot_comb:
25         for l, m, n in let_prod:
26             deciphered = Enigma(rotors=(
27                 Walzen(index=i, ringstellung='A', grundstellung=l),
28                 Walzen(index=j, ringstellung='A', grundstellung=m),
29                 Walzen(index=k, ringstellung='A', grundstellung=n)
30             )).cipher(text)
31             yield ('(%s%s%s %s %s %s): %s' % (l, m, n, i, j, k, deciphered),
32                    index_of_coincidence(deciphered))
33
34
35 if __name__ == '__main__':
36
37     if len(argv) == 2:
38         ics = test_rotors(argv[1])
39         filtered_ics = filter(lambda v: 0.05 < v[1] <= 0.07, ics)
40         sorted_ics = sorted(filtered_ics, key=lambda o: o[1], reverse=True)
41
42         with open('result.out', 'w') as fp:
43             for i in sorted_ics:
44                 fp.write(str(i))
```

- Linhas 14–20: calcula $IC = \sum_{i=A}^{Z} \frac{f_i \times (f_i - 1)}{n \times (n-1)}$ tal como descrito por Smart [3, p. 73].
- Linhas 31–32: Armazena o texto em um gerador, que será tratado para os resultados desejados.
- Linha 39: Filtra o conjunto de ICs. Por precaução, a amplitude do intervalo continua sendo alta.
- Linha 40: Ordena os resultados pelo IC, para que o resultado seja facilmente distinguível.
- Linhas 42–44: Armazena o resultado em disco para acesso posterior e pesquisa, se necessário.

O resultado final foi modificado apenas para adição de espaços. O texto original foi adicionado abaixo para comparação de conteúdo, visto que a pontuação e ênfase contextual podem ser ambíguas. Após o texto decifrado, o IC apresenta um valor muito próximo ao IC da língua inglesa (0.068). Por conta da quantidade de caracteres no texto cifrado, nenhuma outra possibilidade de texto ultrapassou a marca de 0.043 em relação ao índice.

```
('(FSC III II I): AS ECONOMICS IS KNOWN AS THE MISERABLE SCIENCE SOFTWARE
ENGINEERING SHOULD BE KNOWN AS THE DOOMED DISCIPLINE DOOMED BECAUSE IT CANNOT
EVEN APPROACH ITS GOAL SINCE ITS GOAL IS SELFCONTRADICTORY SOFTWARE ENGINEERING
OF COURSE PRESENTS ITSELF AS ANOTHER WORTHY CAUSE BUT THAT IS EYEWASH IF YOU
CAREFULLY READ ITS LITERATURE AND ANALYSE WHAT ITS DEVOTEES ACTUALLY DO YOU
WILL DISCOVER THAT SOFTWARE ENGINEERING HAS ACCEPTED AS ITS HOW TO PROGRAM IF
YOU CANNOT EDSGER DIJKSTRA', 0.06592372083555963)
```

As economics is known as "The Miserable Science", software engineering should be known as "The Doomed Discipline", doomed because it cannot even approach its goal since its goal is self-contradictory. Software engineering, of course, presents itself as another worthy cause, but that is eyewash: if you carefully read its literature and analyse what its devotees actually do, you will discover that software engineering has accepted as its charter "How to program if you cannot.". [1]

Finalmente, o *grundstellung* é igual a **F, S, C** e o *walzenlage* é igual a **III, II, I**.

Questão 4

O ataque diferencial da SPN será discutido logo abaixo. O código responsável por cifrar uma mensagem através da SPN será omitido neste documento, mas pode ser encontrado aqui. Este ataque está direcionado para uma certa subchave e *input-xor* definido, para que um paralelo com Stinson [4, p. 94] seja realizado.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from random import sample
from spn import SPN

bits = list(map(lambda x: "{0:04b}".format(x), range(16)))
key = '00111010100101001101011000111111'
permut = {
    1: 1, 2: 5, 3: 9, 4: 13, 5: 2, 6: 6, 7: 10, 8: 14, 9: 3,
    10: 7, 11: 11, 12: 15, 13: 4, 14: 8, 15: 12, 16: 16,
}
sbox = {
    '0000': '1110', '0001': '0100', '0010': '1101', '0011': '0001',
    '0100': '0010', '0101': '1111', '0110': '1011', '0111': '1000',
    '1000': '0011', '1001': '1010', '1010': '0110', '1011': '1100',
    '1100': '0101', '1101': '1001', '1110': '0000', '1111': '0111',
}
_spn = SPN(permut, sbox)


def gen_quadruples(chosen_xor='1011'):
    tuples = []
    for i in bits:
        u = ''
        for j in range(4):
            u += _spn.xor(chosen_xor[j], i[j])
        tuples += [(i, u, _spn.sbox[i], _spn.sbox[u])]

    quadruples = []
    for each in tuples:
        for i in range(16):
            for j in range(4):
                bits1 = bits[i] + each[0] + '1001' + bits[j]
                bits2 = bits[i] + each[1] + '1001' + bits[j]
                quadruples.append((bits1, bits2, _spn.encode(bits1, key),
                                   _spn.encode(bits2, key)))

    return sample(quadruples, 200)


def multi_xor(arg1, arg2):
    return ''.join(_spn.xor(arg1[i], arg2[i]) for i in range(len(arg1)))


def diff_attack(quadruples=gen_quadruples()):
    inverse_sbox = {v: k for k, v in _spn.sbox.items()}

    tentatives = []
    for i in range(16):
        for j in range(16):
            tentatives.append([bits[i], bits[j]])

    count = [0] * len(tentatives)

    for each in quadruples:
        if each[2][0:4] == each[3][0:4] and each[2][8:12] == each[3][8:12]:
            for key in tentatives:
                v2 = multi_xor(key[0], each[2][4:8])
                v4 = multi_xor(key[1], each[2][12:16])
                v2_star = multi_xor(key[0], each[3][4:8])
                v4_star = multi_xor(key[1], each[3][12:16])

                u2 = inverse_sbox[v2]
                u4 = inverse_sbox[v4]
                u2_star = inverse_sbox[v2_star]
                u4_star = inverse_sbox[v4_star]

                u2_line = multi_xor(u2, u2_star)
                u4_line = multi_xor(u4, u4_star)

                if u2_line == '0110' and u4_line == '0110':
                    count[tentatives.index(key)] += 1

```

```
75      return tentatives[count.index(max(count))]
76
77
78  assert diff_attack() == ['0110', '1111']
```

- Linhas 22–39: Cria todas as quádruplas de texto em claro, e cifrado, disponíveis sendo um *input-xor* fixo. Após o procedimento, 200 instâncias aleatórias, sem repetição, são escolhidas e repassadas ao próximo método. Relembrando: a quádrupla consiste de um x e um x^* amarrados por um *input-xor*, e seus resultados y e y^* após serem codificados.
- Linhas 42–43: Declaração de uma operação $\oplus$ *multibit* para praticidade nas iterações do ataque diferencial.
- Linhas 46–75: O método principal consiste da construção de todas as possibilidades de subchave para que a contagem seja realizada, e da lógica de ataque probabilístico. Após o filtro de pares inúteis, os bits v' são manipulados de modo que passam pela permutação inversa da caixa-S e, se forem iguais aos resultados desejados, contribuem para um contador de subchave correta.
- Linha 78: Como este caso é fixo, porém probabilístico, o algoritmo não mostrará sucesso em todas as situações.

Entretanto, com esta implementação, não foi possível concluir que $T = c \times e^{-1}$, onde $e \approx 38$ e $50 < T < 100$. A quantidade observada foi um pouco maior — cerca de 200 quádruplas foram necessárias para obtenção da subchave em 80% das vezes. Com $T = 100$, inferiu-se que em menos de 50% das vezes isso acontece.

Referências

[1] Edsger W. Dijkstra et al. On the cruelty of really teaching computing science. *Communications of the ACM*, 32(12):1398–1404, 1989.

[2] Andrew Regan. Attacking the Enigma with an Index of Coincidence. November 2004. http://cs.rochester.edu/u/brown/Crypto/studprojs/Enigma.html.

[3] Nigel Smart. *Cryptography: An Introduction*. McGraw-Hill College, London, 3rd edition, 2004. http://www.cs.umd.edu/~waa/414-F11/IntroToCrypto.pdf.

[4] Douglas Stinson. *Cryptography: Theory and Practice*. CRC/C&H, 2nd edition, 2002.