Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

«Московский государственный технический университет имени Н.Э. Баумана» (МГТУ им. Н.Э. Баумана)

Факультет информатика и управление (ИУ)

Кафедра Информационные системы и телекоммуникации (ИУ-3)

Теория информационных процессов и систем

3-й курс, 5-й семестр.

Отчет

по лабораторной работе №4

«Анализ корректирующей способности»

Группа ИУ3-51 Б

Выполнил: Магомедов В.О.

Проверила: Руденкова Ю.С.

Москва, 2020

Цель работы:

Анализ корректирующей способности линейных блоковых кодов.

Задание:

1. Подать на вход источника данных из системы, созданной в рамках лабораторной работы №1, текст длиной 10000 символов.
2. Последовательно передать эти данные в кодер канала, помехоустойчивый кодер, канал передачи данных с помехами, помехоустойчивый декодер, декодер канала и приемник.
3. Посчитать количество двоичных символов, переданных через канал связи при передаче текста. Далее посчитать, сколько было передано 0 и сколько было передано 1.
4. Посчитать среднее количество двоичных символов, которое понадобилось для передачи 1 буквы.
5. Посчитать следующие количественные характеристики ошибок:
   a. Количество ошибок возникших в канале.
   b. Количество исправленных ошибок.
   c. Количество неисправленных ошибок.
   d. Количество неверно исправленных ошибок.

Ход работы:

1) Вычисление размера алфавита, количество информационных битов, количество проверочных битов, создание таблицы символ- код (без проверочных символов).

2) Помехоустойчивый кодер – кодирует информацию, внося избыточность. Таким образом символы в пакете становятся связаны между собой некоторыми математическими операциями

3) Помехоустойчивый декодер – декодирует информацию и проверяет, возникли ли ошибки в канале связи, и если находит их, то пытается их исправить.

Работа программы:

Программа использует файл input.txt для считывания исходных данных. Затем кодирует (помехоустойчиво) полученное сообщение и записывает закодированный результат в файл code.txt. Затем в канал связи добавляется шум и происходит декодирование (помехоустойчивое) и его результат сохраняется в файл output.txt. Используется входной файл с текстом, размером в 10000 символов.

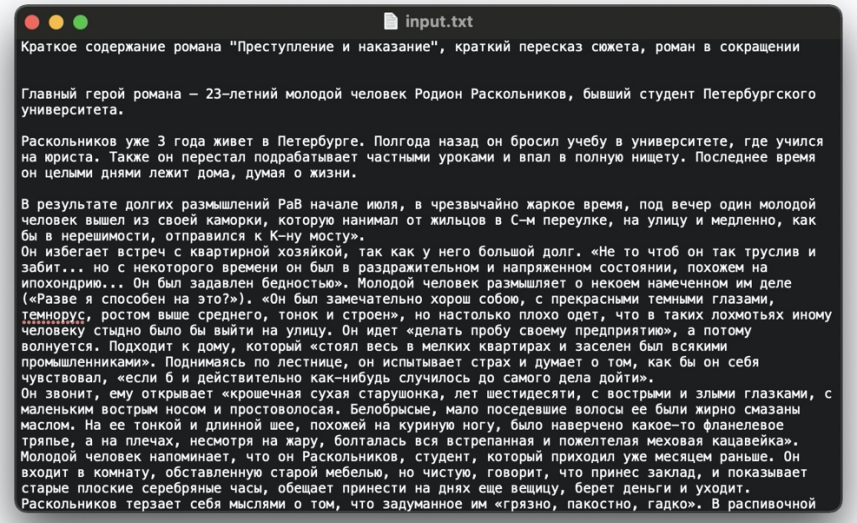

Рис. 1 - Входной файл input.txt

После помехоустойчивого декодирования текст остается вполне читаемым, хоть немного и отличается от исходного.

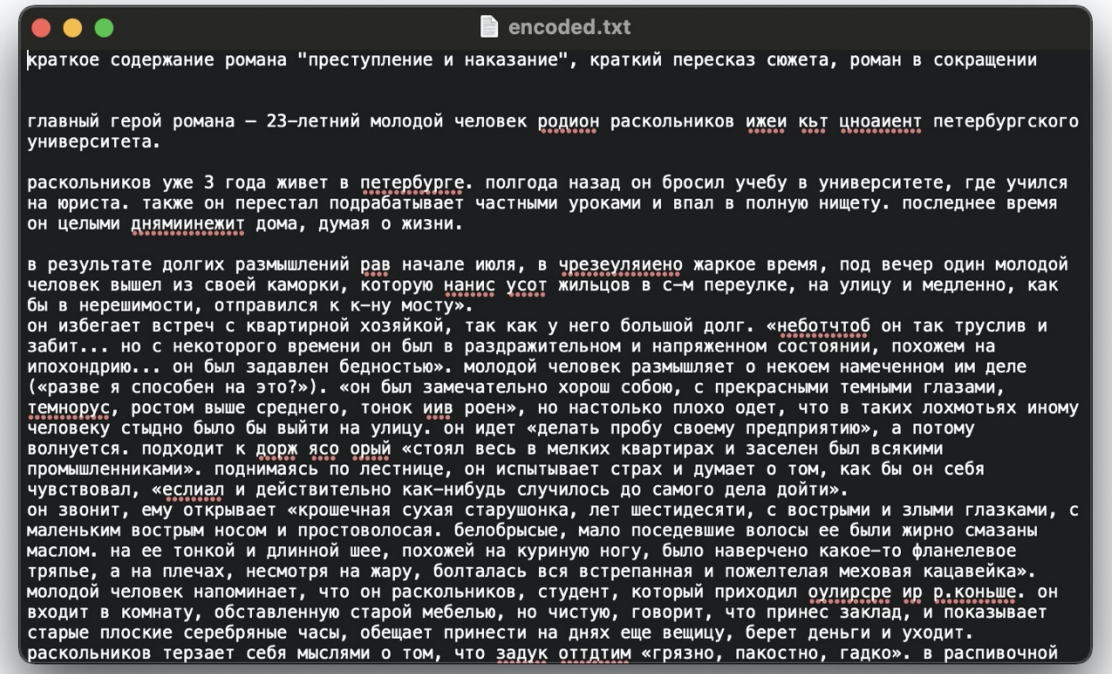

Рис. 2 - Выходной файл encoded.txt

```
[[1. 1. 1. 0. 1. 0. 0.]
 [1. 1. 0. 1. 0. 1. 0.]
 [1. 0. 1. 1. 0. 0. 1.]]
Нужно преобразовать: 88370
Преобразовано: 404204
0`s преобразовано: 187392
1`s преобразовано: 216812
Преобразовано в текст: 4.5739956998981555
Ошибок: 532
Не исправлено:  82868

Process finished with exit code 0
```

Рис. 3 – Результат работы программы

Вывод:

• Самокорректирующий код обладает огромной избыточностью, которая, однако, уменьшается с увеличением входного алфавита, так как увеличивается отношение c_databits/c_checkbits.
•Код позволяет одну ошибку, что недостаточно для коррекции при шуме даже с вероятностью 0.9. Так как количество неисправленных ошибок составляет порядка 30% от всех ошибок
•Время выполнения самокорректирующего кода, естественно, больше.
•Так и объем закодированного файла больше на 40%, при алфавите от 33 до 64 символов.

Листинг:

```
"""created by Vali Magomedov and Vladislav Zhilin"""
import heapq
import numpy as np
import scipy as sc
from collections import Counter, deque
import random

failures = 0



class HeapNode:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __cmp__(self, other):
        if (other == None):
            return -1
        if (not isinstance(other, HeapNode)):
```

```
            return -1
        return self.freq >= other.freq

    def __lt__(self, other):
        return self.freq < other.freq

    def __str__(self):
        return "%s: %s" % (self.char, self.freq)


class HuffmanCoding:
    def __init__(self, use_blocks=False, use_same_freq=False):
        self.heap = []
        self.codes = {}
        self.reverse_mapping = {}
        self.use_blocks = use_blocks
        self.use_same_freq = use_same_freq

    def make_frequency_dict(self, text):
        frequency = {}
        for character in text:
            if self.use_same_freq:
                frequency[character] = 50
            else:
                if not character in frequency:
                    frequency[character] = 0
                frequency[character] += 1

        print(frequency)
        return frequency

    def make_heap(self, frequency):
        for key in frequency:
            node = HeapNode(key, frequency[key])
```

```
            heapq.heappush(self.heap, node)

    def merge_nodes(self):
        while len(self.heap) > 1:
            node1 = heapq.heappop(self.heap)
            node2 = heapq.heappop(self.heap)

            merged = HeapNode(None, node1.freq + node2.freq)
            merged.left = node1
            merged.right = node2

            heapq.heappush(self.heap, merged)

    def make_codes_helper(self, root, current_code):
        if root is None:
            return

        if root.char is not None:
            self.codes[root.char] = current_code
            self.reverse_mapping[current_code] = root.char
            return

        self.make_codes_helper(root.left, current_code + "0")
        self.make_codes_helper(root.right, current_code + "1")

    def make_codes(self):
        root = heapq.heappop(self.heap)
        current_code = ""
        self.make_codes_helper(root, current_code)

    def get_encoded_text(self, text):
        encoded_text = ""
        for character in text:
            encoded_text += self.codes[character]
```

```
        return encoded_text

    def compress(self, data):
        text = data.rstrip()

        if self.use_blocks:
            n = 2
            text = [text[i:i + n] for i in range(0, len(text), n)]

        frequency = self.make_frequency_dict(text)

        self.make_heap(frequency)
        self.merge_nodes()
        self.make_codes()

        encoded_text = self.get_encoded_text(text)
        return encoded_text

    def decompress(self, encoded_text):
        current_code = ""
        decoded_text = ""

        for bit in encoded_text:
            current_code += bit
            if (current_code in self.reverse_mapping):
                character = self.reverse_mapping[current_code]
                decoded_text += character
                current_code = ""

        return decoded_text


class Source:
    data = ""
```

```
    def __init__(self, data):
        self.data = data


class Coder(object):
    def __init__(self, array):
        self.text = array
        self.freq_array = Counter(array)
        self.coded = self.code()
        self.coded_dict = dict(self.coded)
        self.coded_string = ''.join([self.coded_dict[i] for i in self.text])
        with open('coded.txt', 'w', encoding='UTF-8') as file_to_output:
            file_to_output.write(self.coded_string)

    def code(self):
        heap = [[count, [symbol, ""]] for symbol, count in
self.freq_array.items()]
        heapq.heapify(heap)
        while len(heap) > 1:
            lower = heapq.heappop(heap)
            higher = heapq.heappop(heap)
            for pair in lower[1:]:
                pair[1] = '0' + pair[1]
            for pair in higher[1:]:
                pair[1] = '1' + pair[1]
            heapq.heappush(heap, [lower[0] + higher[0]] + lower[1:] +
higher[1:])
        return sorted(heapq.heappop(heap)[1:], key=lambda i: (len(i[-1]), i))


class Transiver(object):
    def __init__(self, G, H, P):

        global failures
```

```
        with open('coded.txt', 'r', encoding='UTF-8') as file_to_read:
            self.data = file_to_read.read()
        self.input_matrix = np.fromstring(' '.join(list(self.data)), sep=' ')
        self.input_array = np.array([self.input_matrix[i - 4:i] for i in
range(4, len(self.input_matrix), 4)])
        self.matrix_to_write = np.concatenate([array.dot(G) for array in
self.input_array], axis=0)
        self.string_to_write = ''.join([str(int(item % 2)) for item in
self.matrix_to_write])

        rand_range = int((1 - P) * len(self.string_to_write))  # шумы
        for i in range(0, len(self.string_to_write), rand_range):
            if self.string_to_write[i] == '0':
                self.string_to_write = self.string_to_write[:i] + '1' +
self.string_to_write[i + 1:]
            if self.string_to_write[i] == '1':
                self.string_to_write = self.string_to_write[:i] + '0' +
self.string_to_write[i + 1:]

        with open('coded.txt', 'w', encoding='UTF-8') as file_to_output:
            file_to_output.write(self.string_to_write)

        with open('coded.txt', 'r', encoding='UTF-8') as file_to_read:  #
поиск ошибок
            self.refactored_data = file_to_read.read()
        self.refactored_input_matrix = np.fromstring('
'.join(list(self.refactored_data)), sep=' ')
        self.refactored_input_array = np.array(
            [self.refactored_input_matrix[i - 7:i] for i in range(7,
len(self.refactored_input_matrix), 7)])
        self.s_array = [array.dot(H.transpose()) for array in
self.refactored_input_array]
        self.s_array = [item % 2 for item in self.s_array]

        for index, value in enumerate(self.s_array):  # исправление
            if 1 in value:
                failures = failures + 1
```

```
                position = int(value[0]) * 4 + int(value[1]) * 2 +
int(value[2]) * 1 - 1
                self.refactored_input_array[index][position] = 1 if
self.refactored_input_array[index][

position] == 0 else \
                    self.refactored_input_array[index][position] == 0
        self.refactored_input_array = [array[:4] for array in
self.refactored_input_array]
        with open('coded.txt', 'w', encoding='UTF-8') as file_to_output:
            file_to_output.write(''.join([str(int(item)) for item in
np.concatenate(self.refactored_input_array)]))

        with open('coded.txt', 'r', encoding='UTF-8') as file_to_read:
            self.data = file_to_read.read()



class Decoder(object):
    def __init__(self, data, config):
        self.data = deque(data)
        self.config = config
        self.decoded_string = ''
        self.decode()

    def decode(self):
        s = ''
        while self.data:
            s = s + self.data.popleft()
            for key, value in self.config.items():
                if s == value:
                    self.decoded_string = self.decoded_string + key
                    s = ''



class Receive(object):
    def __init__(self, data):
```

```
        with open('encoded.txt', 'w', encoding='UTF-8') as file_to_write:
            file_to_write.write(data)


with open('input.txt', 'r') as f:
    data = f.read()

G_matrix = np.array([[1., 0., 0., 0., 1., 1., 1.], [0., 1., 0., 0., 1., 1.,
0.], [0., 0., 1., 0., 1., 0., 1.],
                     [0., 0., 0., 1., 0., 1., 1.]])
P_matrix = np.array([array[len(G_matrix):] for array in G_matrix])
P_matrix_transposed = P_matrix.transpose()
H_matrix = np.hstack((P_matrix_transposed, np.eye(3)))
print(H_matrix)

probability = 0.999
with open('input.txt', 'r', encoding='UTF-8') as file:
    text = file.read().lower()

source = Source(text)
coder = Coder(source.data)
transiver = Transiver(G_matrix, H_matrix, probability)
decoder = Decoder(transiver.data, coder.coded_dict)
receiver = Receive(decoder.decoded_string)

print(f'Нужно преобразовать: {len(text)}')
print(f'Преобразовано: {len(transiver.data)}')
print(f'0`s преобразовано: {transiver.data.count("0")}')
print(f'1`s преобразовано: {transiver.data.count("1")}')
print(f'Преобразовано в текст: {len(transiver.data) / len(text)}')
print(f'Ошибок: {failures}')
not_fixed = 0
with open('input.txt', 'r', encoding='UTF-8') as file:
    text = file.read().lower()
with open('encoded.txt', 'r', encoding='UTF-8') as file:
```

```
    encoded_text = file.read().lower()
for i in range(min(len(text), len(encoded_text))):
    if text[i] != encoded_text[i]:
        not_fixed = not_fixed + 1
print(f'Не исправлено:  {not_fixed}')
```