Федеральное государственное бюджетное образовательное учреждение высшего образования  
«Сибирский государственный университет телекоммуникаций и информатики»  
(СибГУТИ)

Институт информатики и вычислительной техники  
Кафедра прикладной математики и кибернетики

**Практическая работа №5  
 по дисциплине «Теория информации»  
на тему «Кодирование и декодирование кодом Хэмминга»**

Выполнил:  
студент гр.ИП-014

Обухов А.И.

Проверила:  
Старший преподаватель каф. ПМиК  
Дементьева Кристина Игоревна

Новосибирск 2024 г.

**Цель работы:** Изучение кода Хэмминга.

**Язык программирования:** С, С++, С#, Python

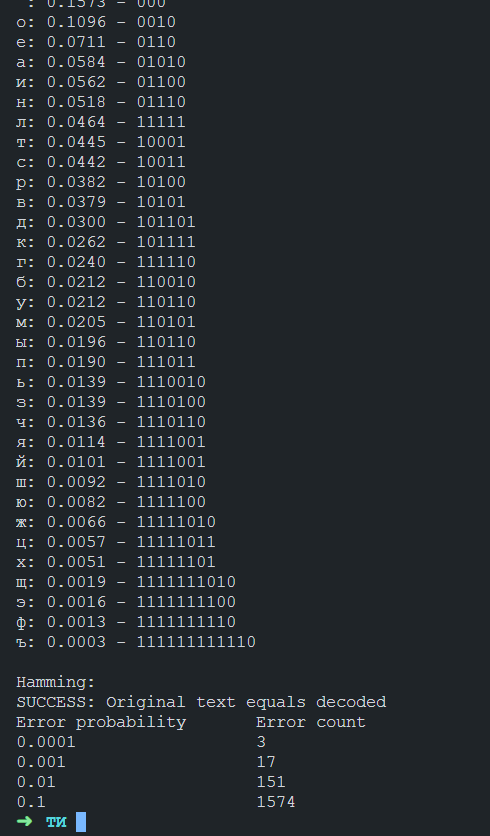
**Результат:** программа, тестовые примеры, отчет.

**Задание:**

1. Для выполнения работы необходим закодированный побуквенным кодом файл из практической работы 2. В таком файле содержатся только 0 и 1.
2. Реализовать кодирование и декодирование файла кодом Хэмминга (длина кода 7 или 15).
3. После кодирования кодом Хэмминга в закодированном файле случайным образом с вероятностью р заменить 0 на 1, 1 на 0 (сделать ошибки при передаче файла в симметричном канале). Декодировать измененный файл. Сравнить исходный и декодированный файлы, подсчитать количество ошибок и заполнить таблицу.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Вероятность ошибки | р = 0.0001 | р = 0.001 | р = 0.01 | р = 0.1 |
| Количество ошибок | 3 | 17 | 151 | 1574 |

**Скриншоты работы программы**



**Анализ результатов работы программы**

1. **Вероятность ошибки и количество ошибок:** Из представленной таблицы следует, что количество ошибок, обнаруженных методом Хэмминга, возрастает с ростом вероятности ошибки. Это объясняется тем, что при повышении вероятности ошибки больше битов данных будет изменено случайным образом, что усложняет процесс декодирования. Следовательно, при увеличении вероятности ошибки увеличивается сложность декодирования, что приводит к уменьшению вероятности успешного обнаружения и исправления ошибок.
2. **Объем текста и количество ошибок:** Также следует учитывать, что объем текста может оказывать влияние на количество ошибок, обнаруживаемых методом Хэмминга. При работе с более объемными данными вероятность наличия множества ошибок также возрастает, что может сказаться на эффективности процесса декодирования..
3. **Структура данных:** Эффективность кодирования и декодирования с применением метода Хэмминга также зависит от структуры данных. Например, данные с определенной структурой или особенностями могут быть более или менее подвержены ошибкам при передаче.
4. **Длина кодового слова:** Длина кодового слова, в данном случае 7, играет также важную роль. Более длинные коды обычно обеспечивают более высокий уровень защиты от ошибок, поскольку позволяют обнаруживать и исправлять больше ошибок. Однако такие коды могут также требовать больше вычислительных ресурсов и быть менее эффективными при ограниченных ресурсах.

**Листинг программы**

import math

import numpy as np

from lab1 import \*

def shannon\_encode(text: str) -> str:

    print("Shannon:")

    def decimal\_converter(num):

        if num == 0.0:

            return 0.0

        while num > 1:

            num /= 10

        return num

    def float\_bin(number: float, places: int):

        whole, dec = str(number).split(".")

        whole = int(whole)

        dec = int(dec)

        res = bin(whole).strip("0b") + "."

        for x in range(places):

            whole, dec = str((decimal\_converter(dec)) \* 2).split(".")

            dec = int(dec)

            res += whole

        return res

    split\_line = list(text[i: i + 1] for i in range(len(text)))

    probabilities = {k: v / len(split\_line) for k, v in Counter(split\_line).items()}

    probabilities = dict(

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

    )

    code\_length = [math.ceil(-math.log2(i)) for i in probabilities.values()]

    cumulative\_probs = [float(0) for \_ in range(len(probabilities))]

    for i in range(1, len(probabilities)):

        cumulative\_probs[i] = cumulative\_probs[i - 1] + list(probabilities.values())[i - 1]

    codes = list()

    for i in range(len(cumulative\_probs)):

        codes.append(float\_bin(cumulative\_probs[i], code\_length[i])[1:])

        print(f"{list(probabilities.keys())[i]}: {list(probabilities.values())[i]:.4f} - {codes[i]}")

    print()

    with open("./output/coded\_shannon.txt", "w") as f:

        for i in text:

            index = list(probabilities.keys()).index(i)

            f.write(codes[index])

    with open("./output/coded\_shannon.txt", "r") as f:

        text = f.readline()

    return text

def calcRedundantBits(m):

    for i in range(m):

        if 2\*\*i >= m + i + 1:

            return i

def hamming\_encode(file\_path: str) -> str:

    def posRedundantBits(data, r):

        j = 0

        k = 1

        m = len(data)

        res = ''

        for i in range(1, m + r+1):

            if(i == 2\*\*j):

                res = res + '0'

                j += 1

            else:

                res = res + data[-1 \* k]

                k += 1

        return res[::-1]

    def calcParityBits(arr, r):

        n = len(arr)

        for i in range(r):

            val = 0

            for j in range(1, n + 1):

                if(j & (2\*\*i) == (2\*\*i)):

                    val = val ^ int(arr[-1 \* j])

            arr = arr[:n-(2\*\*i)] + str(val) + arr[n-(2\*\*i)+1:]

        return arr

    with open(file\_path, 'r') as file:

        data = file.read()

    m = len(data)

    r = calcRedundantBits(m)

    arr = posRedundantBits(data, r)

    arr = calcParityBits(arr, r)

    return arr

def hamming\_decode(encoded\_data: str) -> str:

    def detectError(arr, nr):

        n = len(arr)

        res = 0

        for i in range(nr):

            val = 0

            for j in range(1, n + 1):

                if j & (2\*\*i) == (2\*\*i):

                    val = val ^ int(arr[-1 \* j])

            res = res + val\*(10\*\*i)

        return int(str(res), 2)

    m = len(encoded\_data)

    r = calcRedundantBits(m)

    error\_position = detectError(encoded\_data, r)

    if error\_position != 0:

        error\_index = error\_position - 1

        encoded\_data = encoded\_data[:error\_index] + ('0' if encoded\_data[error\_index] == '1' else '1') + encoded\_data[error\_index+1:]

    decoded\_data = ''

    j = 0

    for i in range(1, len(encoded\_data) + 1):

        if i != 2\*\*j:

            decoded\_data += encoded\_data[-i]

        else:

            j += 1

    return decoded\_data[::-1]

def count\_errors(original\_data: str, decoded\_data: str) -> int:

    errors = sum(bit1 != bit2 for bit1, bit2 in zip(original\_data, decoded\_data))

    return errors

def main():

    print("------ text.txt ------\n")

    input\_text = preprocess\_file('./input/text.txt', 'ru')

    encoded\_text\_shannon = shannon\_encode(input\_text)

    print("Hamming:")

    encoded\_text\_hamming = hamming\_encode('./output/coded\_shannon.txt')

    decoded\_text = hamming\_decode(encoded\_text\_hamming)

    if (encoded\_text\_shannon == decoded\_text):

        print("SUCCESS: Original text equals decoded")

    else:

        print("ERROR: Original text doesn't equal decoded")

    probabilities = [0.0001, 0.001, 0.01, 0.1]

    print("Error probability\tError count")

    for probability in probabilities:

        encoded\_copy = list(encoded\_text\_hamming)

        for i in range(len(encoded\_copy)):

            if random.random() < probability:

                encoded\_copy[i] = '0' if encoded\_copy[i] == '1' else '1'

        corrupted\_data = ''.join(encoded\_copy)

        decoded\_text = hamming\_decode(corrupted\_data)

        errors = count\_errors(encoded\_text\_shannon, decoded\_text)

        print(f"{probability}\t\t\t{errors}")

if \_\_name\_\_ == "\_\_main\_\_":

    main()