МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
“БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ”

Кафедра ИИТ

ОТЧЁТ
По лабораторной работе №1
«Избыточное кодирование данных в информационных системах. Код Хемминга»

Выполнил:
Студент группы ИИ-22
Варицкий М.И.
Проверила:
Хацкевич А.С.

Брест 2024

Цель работы: приобретение практических навыков кодирования/декодирования двоичных данных при использовании кода Хемминга.

Задание.

1. Закрепить теоретические знания по использованию методов помехоустойчивого кодирования для повышения надежности передачи и хранения в памяти компьютера двоичных данных.
2. Разработать приложение для кодирования/декодирования двоичной информации кодом Хемминга с минимальным кодовым расстоянием 3 или 4.
3. Результаты выполнения лабораторной работы оформить в виде отчета с листингом разработанного приложения, методики выполнения экспериментов с использованием приложения и результатов эксперимента.
4. Ответить на контрольные вопросы

Ход работы

Вариант	M[1]	r
2	832	5

1. Составить код Хемминга (классический алгоритм) (M+r, M), допустить ошибку в одном из разрядов и отыскать её по алгоритму.

2. Составить код Хемминга (расширенный алгоритм) (первые 7 битов M + 3 проверочных, первые 7 битов M), допустить 2 или более ошибок в разрядах и отыскать их по алгоритму

Код программы:

Код Хэминга

```
class HammingCode:
    def __init__(self, message, r):
        self.message = message
        self.r = r
        self.n = len(message) + r  # Полная длина закодированного слова
        self.code = [0] * self.n

    def encode(self):
        # Расставляем информационные биты и проверочные биты на позиции, кратные 2^k
        j = 0
        for i in range(1, self.n + 1):
            # Проверочные биты на позициях 1, 2, 4, 8 и т.д.
            if (i & (i - 1)) == 0:
                continue  # Эти позиции оставляем для проверочных битов
            else:
                # Расставляем информационные биты
                if j < len(self.message):  # Проверяем, чтобы не выйти за пределы
индексов
                    self.code[i - 1] = int(self.message[j])
                    j += 1

        # Вычисляем значения проверочных битов
        for i in range(self.r):
            pos = 1 << i  # Позиция проверочного бита (1, 2, 4, 8, ...)
            if pos - 1 >= self.n:  # Проверяем, чтобы не выйти за пределы code
                continue
            sum_bits = 0
            for j in range(1, self.n + 1):
                if j & pos:
                    sum_bits ^= self.code[j - 1]
            self.code[pos - 1] = sum_bits
        print("Сформированный код:", ''.join(map(str, self.code)))
        return self.code

    def introduce_error(self, position):
        # Инвертируем бит на указанной позиции для имитации ошибки
        self.code[position - 1] ^= 1
        print("Код с ошибкой в позиции", position, ":", ''.join(map(str, self.code)))

    def detect_and_correct(self):
        # Обнаружение позиции ошибки
        error_position = 0
        for i in range(self.r):
            pos = 1 << i
            sum_bits = 0
            for j in range(1, self.n + 1):
```

```
                if j & pos:
                    sum_bits ^= self.code[j - 1]
            if sum_bits != 0:
                error_position += pos

        if error_position == 0:
            print("Ошибок не обнаружено.")
        else:
            print("Обнаружена ошибка в позиции:", error_position)
            self.code[error_position - 1] ^= 1
            print("Исправленный код:", ''.join(map(str, self.code)))
        return error_position

    def decode(self):
        # Извлекаем только информационные биты
        decoded_message = []
        for i in range(1, self.n + 1):
            if (i & (i - 1)) != 0:
                decoded_message.append(str(self.code[i - 1]))
        decoded_message = ''.join(decoded_message[:len(self.message)])
        print("Расшифрованное сообщение:", decoded_message)
        return decoded_message


# Пример использования
message = "1101000000"  # Исходное сообщение длиной 10 бит
r = 4  # Количество проверочных битов

hamming = HammingCode(message, r)
encoded_code = hamming.encode()  # Кодирование
hamming.introduce_error(3)  # Допустим ошибку в третьем бите
hamming.detect_and_correct()  # Обнаружение и исправление ошибки
decoded_message = hamming.decode()  # Расшифровка исходного сообщени
```

Расширенный код хэминга

```
class ExtendedHammingCode:
    def __init__(self, message):
        if len(message) != 7:
            raise ValueError("Длина сообщения должна быть ровно 7 бит.")
        self.message = message
        self.n = 12  # Полная длина закодированного слова: 7 информационных битов + 4
проверочных + 1 бит четности
        self.code = [0] * self.n

    def encode(self):
        # Устанавливаем информационные биты в кодовом слове
        j = 0
        for i in range(1, self.n):
            if (i & (i - 1)) != 0:  # Проверяем, что позиция не является степенью 2 (1,
2, 4, 8)
                self.code[i - 1] = int(self.message[j])
                j += 1

        # Вычисляем проверочные биты
        for i in range(4):
            pos = 1 << i  # Позиции проверочных битов: 1, 2, 4, 8
            parity = 0
            for j in range(1, self.n):
                if j & pos:
                    parity ^= self.code[j - 1]
            self.code[pos - 1] = parity

        # Вычисляем общий бит четности
        self.code[-1] = sum(self.code[:-1]) % 2
        print("Сформированный код:", ''.join(map(str, self.code)))
        return self.code

    def introduce_errors(self, positions):
        # Внесение ошибок на указанные позиции
        for pos in positions:
            if 1 <= pos <= self.n:
                self.code[pos - 1] ^= 1
        print("Код с ошибками на позициях", positions, ":", ''.join(map(str,
self.code)))

    def detect_and_correct(self):
        # Проверяем позиции проверочных битов для обнаружения ошибки
        error_position = 0
        for i in range(4):
            pos = 1 << i
            parity = 0
            for j in range(1, self.n):
                if j & pos:
                    parity ^= self.code[j - 1]
            if parity != 0:
                error_position += pos

        # Проверка общего бита четности
        parity_check = sum(self.code) % 2

        # Интерпретация результатов
        if error_position == 0 and parity_check == 0:
            print("Ошибок не обнаружено.")
        elif error_position != 0 and parity_check == 1:
            print("Обнаружена одиночная ошибка в позиции:", error_position)
            self.code[error_position - 1] ^= 1
            print("Исправленный код:", ''.join(map(str, self.code)))
        elif error_position != 0 and parity_check == 0:
            print("Обнаружена двойная ошибка. Исправление невозможно.")
        else:
            print("Обнаружена ошибка в бите четности. Исправление невозможно.")

    def decode(self):
        # Извлекаем только информационные биты
        decoded_message = []
        for i in range(1, self.n):
            if (i & (i - 1)) != 0:  # Информационные биты (позиции не являются степенями
двойки)
                decoded_message.append(str(self.code[i - 1]))
        decoded_message = ''.join(decoded_message)
        print("Расшифрованное сообщение:", decoded_message)
        return decoded_message


# Пример использования
message = "1101001"  # Исходное сообщение длиной 7 бит
hamming = ExtendedHammingCode(message)
encoded_code = hamming.encode()  # Кодирование
hamming.introduce_errors([3,5])  # Внесение одиночной ошибки на позиции 3
```

```
hamming.detect_and_correct()  # Обнаружение и исправление ошибки
decoded_message = hamming.decode()  # Расшифровка исходного сообщения
```

Результат работы:

```
Сформированный код: 10101010000000
Код с ошибкой в позиции 3 : 10001010000000
Обнаружена ошибка в позиции: 3
Исправленный код: 10101010000000
Расшифрованное сообщение: 1101000000
```

```
Сформированный код: 011010110010
Код с ошибками на позициях [3, 5] : 010000110010
Обнаружена двойная ошибка. Исправление невозможно.
Расшифрованное сообщение: 0001001
```

Вывод: приобрёл практические навыки кодирования/декодирования двоичных данных при использовании кода Хемминга.