МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В.Г. Шухова)**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

РГЗ

по дисциплине: Теория информации

Тема: «Моделирование канала передачи данных для источника Хартли с использованием кодера LZW в условиях воздействия помех»

Выполнил: студент группы ПВ-223

Дмитриев Андрей Александрович

Проверил:
Твердохлеб В.В.

Белгород 2024 г.

Задание:

Выбрать сочетание источник-кодер-помехоустойчивость(опционально)-характер помех и построить модель канала.

На приёмной стороне реконструировать данные и отобразить результат, проанализировать.

Ход работы:

Источник. Выбран источник информации Хартли введу того, что эффективность LZW зависит от повторяемости слов.

Кодер. Используется кодер LZW. Семейство алгоритмов LZ предполагает сжатие без потерь путём использования ссылок на повторяющиеся слова, такая особенность позволяет не передавать информацию о алфавите.

Помехоустойчивость. Реализован алгоритм Хэмминга, который применяется для блоков данных, что даёт возможность транслировать сообщение через множественные нарушения битовой последовательности.

Помехи. Смоделированы следующим образом: на последовательности гарантировано инвертируются биты в случайных местах в указанном количестве.

Реализация:

Код написан на языке Kotlin – высокоуровневый язык, написанный для запуска на JVM. Описаны два класса Encoder и Decoder с главными методами encode и decode соответственно.

Функция encode (Листинг 1) производит вызов процедур в порядке:

1. Сжатие LZW для ASCII символов;
2. Приведение массива чисел полученного от алгоритма сжатия к последовательности бит;
3. Обеспечение помехоустойчивости кодом Хэмминга.
4. Формирование метаданных

```
fun encode(controlBits: Number): List<Boolean> {
    if (sourceData.isEmpty()) return emptyList()
    // получение списка чисел
    val numberSequence = lzwCompressAscii(sourceData)
    // выравнивание и получение последовательности бит
    val bitSequence = numbersCompress(numberSequence)
    // вес символа
    val lengthOfElemBitSeq = bitSequence.size / numberSequence.size
    // получение помехоустойчивого кода
    val bitSequenceResistance = genHammingCode(bitSequence, controlBits.toInt())
    // формирование метаданных
    val metadata = genHammingCode(
        numbersCompress(
            listOf(
                controlBits.toInt(),
                lengthOfElemBitSeq,
                16
            )
        ).subList(0, 10)
    )
    // единная последовательность бит метаданные+данныеы
    return metadata.toMutableList().apply { addAll(bitSequenceResistance) }
}
```

Листинг 1 – Функция encode.

LZW алгоритм (Листинг 2) принимает только символы ASCII, поэтому первоначальный словарь формируется из 128 символов. Во время выполнения формируется словарь из повторяющихся отрезков символов. Словарь не ограничен по количеству полей поэтому особо большие сообщения следует кодировать с осторожностью.

```
private fun lzwCompressAscii(data: String): List<Int> {
    val dictionary = mutableMapOf<String, Int>().apply {
        for (i in 0..<128) put(i.toChar().toString(), i.toInt())
    }
    val res = mutableListOf<Int>()
    var buf = ""
    for (c in data) {
        val wc = buf + c
        if (dictionary.containsKey(wc)) {
            buf = wc
        } else {
            res.add(dictionary[buf]!!)
            dictionary[wc] = dictionary.size
            buf = c.toString()
        }
    }

    if (buf.isNotEmpty()) res.add(dictionary[buf]!!)

    return res
}
```

Листинг 2 – Алгоритм LZW.

Генератор Кода Хэмминга (Листинг 3) принимает блоки цельного сообщение и определяет для них необходимое количество контрольных бит. Затем добавляет под них место и производит их вычисление следуя алгоритму Хэмминга. Метаданные также шифруются этим алгоритмом и передаются вместе с сообщением.

Метаданные составляют 10 битовое число под названием metadata, где 5 бит отводится для хранения информации о количестве бит на символ, остальные 5 бит обозначают количество контрольных бит.

```
private fun genHammingCode(bitSequence: List<Boolean>): List<Boolean> {
    val block = mutableListOf<Boolean>().apply { addAll(bitSequence) }

    // add parity bits
    var pbit = 0
    while (1.shl(pbit) - 1 < block.size) {
        block.add(1.shl(pbit) - 1, false)
        pbit++
    }

    // init parity bits
    pbit = 0
    while (1.shl(pbit) - 1 < block.size) {
        var isCount = true
        var isParity = true
        for (i in 1.shl(pbit) - 1..<block.size step 1.shl(pbit)) {
```

```
            for (j in i..<i + 1.shl(pbit)) {
                if (j >= block.size) {
                    break
                }
                if (isCount) {
                    isParity = if (block[j]) !isParity else isParity
                }
            }
            isCount = !isCount
        }
        block[1.shl(pbit) - 1] = !isParity
        pbit++
    }

    return block
}
```

Листинг 3 – Алгоритм Хэмминга.

Для принимающей стороны реализован метод decode (Листинг 4), который совершает обратные действия. Расшифровывает метаданные, которые будут применены в аргументах функций. Далее выполняет корректировку по коду Хэмминга, что может спродуцировать ошибку NotCorrectedByPBitException в случае критических повреждений данных. Полученная последовательность чисел подаётся на вход LZW, получая ответ.

```
fun decode(): String {
    val metadata = decodeHammingCode(encodedData.subList(0, 14))
    val controlBits = metadata.subList(0, 5).toInt()
    val lengthOfElem = metadata.subList(5, 10).toInt()

    val decodedData = encodedData.subList(14, encodedData.size)
    val numSequence = decodeHammingCode(decodedData, controlBits, lengthOfElem)
    val data = decodeLzw(numSequence)

    return data
}

private fun decodeHammingCode(bitSequence: List<Boolean>): List<Boolean> {
    val block = bitSequence.toMutableList()

    // init parity bits
    var pbit = 0
    var posForCorrecting = -1
    while (1.shl(pbit) - 1 < block.size) {
        var isCount = true
        var isParity = true
        for (i in 1.shl(pbit) - 1..<block.size step 1.shl(pbit)) {
            for (j in i..<i + 1.shl(pbit)) {
                if (j >= block.size) {
                    break
                }
                if (isCount) {
```

```
                    isParity = if (block[j]) !isParity else isParity
                }
            }
            isCount = !isCount
        }
        if (!isParity) {
            posForCorrecting =
                (if (posForCorrecting < 0) 0 else posForCorrecting) + 1.shl(pbit)
        }
        pbit++
    }

    if (posForCorrecting - 1 >= block.size) {
        throw NotCorrectedByPBitException()
    } else if (posForCorrecting >= 0) {
        block[posForCorrecting - 1] = !block[posForCorrecting - 1]
    }

    // del parity bits
    pbit = log2(block.size.toDouble()).toInt()
    while (pbit >= 0) {
        block.removeAt(1.shl(pbit) - 1)
        pbit--
    }

    return block
}

private fun decodeLzw(numSequence: List<Int>): String {
    val dictionary = mutableMapOf<Int, String>().apply {
        for (i in 0..<128) put(i, i.toChar().toString())
    }
    var data = numSequence.toList()
    var prevBuf = data[0].toChar().toString()
    data = data.drop(1)
    var res = prevBuf
    for (k in data) {
        var buf = ""
        if (dictionary.containsKey(k)) {
            buf = dictionary[k]!!
        } else if (k == dictionary.size) {
            buf = prevBuf + prevBuf[0]
        }
        res += buf
        try {
            dictionary[dictionary.size] = prevBuf + buf[0]
        } catch (e: StringIndexOutOfBoundsException) {
            throw NotCorrectedByPBitException()
        }
        prevBuf = buf
    }

    return res
}
```

Листинг 4 – Логика декодирования сообщения.

Ниже представлен пример работы консольной программы в режиме кодирования с внесением ошибочных бит (Листинг 5) и процесс декодирования (Листинг 6).

```
add generator of errors (y)es or (n)o
y
Change mode to (e)ncode or (d)ecode
e
Change mode of reading from (f)ile or (s)tring
f
Change mode of writing to (f)ile or (s)tring
f
Enter path to file to encode
C:\Users\dmitr\Projects\2kurs2sem\InformTheor\rgz\src\main\kotlin\src.txt
input num of pbits
4
input num of errors
5
error bits on positions: [35943, 28925, 25356, 6553, 21255]
Enter path to file to write encoded data
C:\Users\dmitr\Projects\2kurs2sem\InformTheor\rgz\src\main\kotlin\enc_data.txt
source: 40628
compressed: 36914
```

Листинг 5 – Пример кодирования.

В процессе кодирование, ошибки внесены в биты на позиции: 35943, 28925, 25356, 6553, 21255 – они на достаточном удалении друг от друга поэтому код Хэмминга корректно выполняет преобразования. Ошибка осуществляется инвертированием случайного в диапазоне бита в сообщении установленное количество раз (Формула 1).

$$bit(x) = !\,(rand(|a - b|)) \quad (1)$$

```
add generator of errors (y)es or (n)o
n
Change mode to (e)ncode or (d)ecode
d
Change mode of reading from (f)ile or (s)tring
f
Change mode of writing to (f)ile or (s)tring
f
Enter path to file to decode
C:\Users\dmitr\Projects\2kurs2sem\InformTheor\rgz\src\main\kotlin\enc_data.txt
Enter path to file to write decoded data
C:\Users\dmitr\Projects\2kurs2sem\InformTheor\rgz\src\main\kotlin\dec_data.txt
```

Листинг 6 – Пример декодирования.

Ниже представлен отрывок расшифрованного текста (Рисунок 1).

```
Abraham begot Isaac, Isaac begot Jacob, and Jacob begot Judah and his brothers.
Judah begot Perez and Zerah by Tamar, Perez begot Hezron, and Hezron begot Ram.
Ram begot Amminadab, Amminadab begot Nahshon, and Nahshon begot Salmon.
Salmon begot Boaz by Rahab, Boaz begot Obed by Ruth, Obed begot Jesse,
and Jesse begot David the king. David the king begot Solomon by her who had been the wife of Uriah.
Solomon begot Rehoboam, Rehoboam begot Abijah, and Abijah begot Asa.
Asa begot Jehoshaphat, Jehoshaphat begot Joram, and Joram begot Uzziah.
Uzziah begot Jotham, Jotham begot Ahaz, and Ahaz begot Hezekiah.
Hezekiah begot Manasseh, Manasseh begot Amon, and Amon begot Josiah.
Josiah begot Jeconiah and his brothers about the time they were carried away to Babylon.
And after they were brought to Babylon, Jeconiah begot Shealtiel, and Shealtiel begot Zerubbabel.
Zerubbabel begot Abiud, Abiud begot Eliakim, and Eliakim begot Azor.
Azor begot Zadok, Zadok begot Achim, and Achim begot Eliud.
Eliud begot Eleazar, Eleazar begot Matthan, and Matthan begot Jacob.
And Jacob begot Joseph the husband of Mary, of whom was born Jesus who is called Christ.
So all the generations from Abraham to David are fourteen generations, from David until the captivity
Now the birth of Jesus Christ was as follows: After His mother Mary was betrothed to Joseph, before th
Then Joseph her husband, being a just man, and not wanting to make her a public example, was minded to
```

Рисунок 1 – Отрывок дешифрованного текста.

В этом примере заданы следующие параметры: текст размером 40628 бит, каждый блок кодировался 4 контрольными битами (т.е. блок содержал не более $2^4 - 4 - 1 = 11$ бит), генерировалось 5 ошибок, которые попали в разные блоки и были исправлены.

Кодированное сообщение составило 36914 бит, что оказалось меньше исходного текста с учётом добавления метаданных и контрольных бит. Это составляет 90.85% от исходного текста.

Ниже представлена модель передачи сигнала (Рисунок 2).

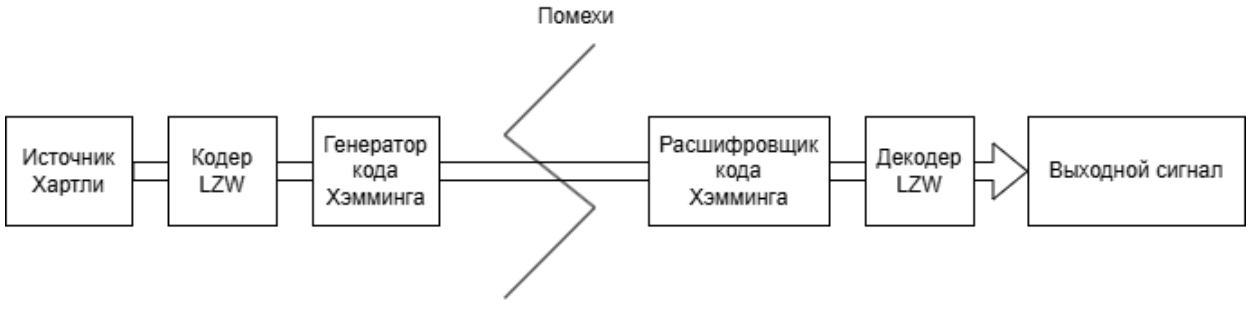

Рисунок 2 – Модель передачи сигнала.

Список литературы:

1. Алгоритм Лемпеля — Зива — Велча — Википедия [Электронный ресурс]. Режим доступа: (wikipedia.org) (Дата обращения 15.05.2024)
2. Теорема Шеннона — Хартли — Википедия [Электронный ресурс]. Режим доступа: wikipedia.org (Дата обращения 14.05.2024)
3. Аддитивная мера информации по Хартли [Электронный ресурс]. Режим доступа: peredacha-informacii.ru (Дата обращения 12.05.2024)

Вывод: в ходе работы были применены методы помехоустойчивого кодирования Хэмминга и сжатия Лемпела-Зива-Велча. Мы рассмотрели пример работы, в котором программа показала положительный результат.

Метод следует модифицировать, например, ограничить словарь и формировать новый; деление данных на пакеты, с возможностью перевызывать их.