МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информационных технологий

Кафедра информационных систем и технологий

Специальность Информационные системы и технологии

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №4 НА ТЕМУ:**

**Избыточное кодирование данных в информационных системах. Код Хемминга**

Выполнил студент 3 курса 1 группы

Халалеенко Андрей Николаевич

Минск 2024

**Задание 1:** На основе информационного сообщения, представленного символами русского/английского алфавитов, служебными символами и цифрами, содержащегося в некотором текстовом файле сформировать информационное сообщение в двоичном виде; длина сообщения в бинарном виде должна быть не менее 16 символов.

**Задание 2:** Для полученного информационного слова построить проверочную матрицу Хемминга.

**Задание 3:** Используя построенную матрицу вычислить избыточные символы (слово Xr).

**Задание 4:** Принять исходное слово со следующим числом ошибок: 0, 1, 2. Позиция ошибки определяется случайным образом.

**Задание 5:** Для полученного слова Yn = Yk, Yr, используя уже известную проверочную матрицу Хемминга, вновь вычислить избыточные символы.

Для выполнения данного задания используются методы, разработанные в предыдущих заданиях.

**Задание 6:** В случае, если анализ синдрома показал, что информационное сообщение было передано с ошибкой (или 2 ошибками), сгенерировать унарный вектор ошибки n и исправить одиночную ошибку, используя формулу (4.7).

Для выполнения заданий был использован следующий код на языке Typescript:

// Определение размера резервных данных

function calculateRedundancyLength(dataLength: number): number {

    const redundancyLength: number = Math.floor(Math.log2(dataLength) + 1.99);

    return redundancyLength;

}

// Генерация проверочной матрицы

function generateParityMatrix(dataLength: number): number[][] {

    const redundancyLength: number = calculateRedundancyLength(dataLength);

    const totalLength: number = dataLength + redundancyLength;

    const rDouble: number = redundancyLength - 1;

    const rPow: number = Math.pow(2, rDouble);

    const matrix: number[][] = new Array(totalLength).fill(0).map(() => new Array(redundancyLength).fill(0));

    const combinations: number[][] = new Array(rPow).fill(0).map(() => new Array(redundancyLength).fill(0));

    for (let i = 0; i < rPow; i++) {

        for (let j = 0; j < redundancyLength; j++) {

            combinations[i][j] = 0;

        }

    }

    for (const segmentLength of Array.from({ length: redundancyLength - 2 }, (\_, i) => i)) {

        if (segmentLength \* redundancyLength > dataLength) break;

        for (const i of Array.from({ length: segmentLength + 2 }, (\_, i) => i)) {

            combinations[segmentLength \* redundancyLength][i] = 1;

        }

        for (const segmentPosition of Array.from({ length: redundancyLength - 1 }, (\_, i) => i + 1)) {

            for (const i of Array.from({ length: redundancyLength - 1 }, (\_, j) => j)) {

                combinations[segmentLength \* redundancyLength + segmentPosition][i + 1] = combinations[segmentLength \* redundancyLength + segmentPosition - 1][i];

            }

            combinations[segmentLength \* redundancyLength + segmentPosition][0] = combinations[segmentLength \* redundancyLength + segmentPosition - 1][redundancyLength - 1];

        }

        if (segmentLength === redundancyLength - 3) {

            for (const i of Array.from({ length: redundancyLength }, (\_, j) => j)) {

                combinations[rPow - 1][i] = 1;

            }

        }

    }

    for (const i of Array.from({ length: dataLength }, (\_, i) => i)) {

        for (const j of Array.from({ length: redundancyLength }, (\_, j) => j)) {

            matrix[i][j] = combinations[i][j];

        }

    }

    for (const i of Array.from({ length: redundancyLength }, (\_, i) => i)) {

        matrix[i + dataLength][i] = 1;

    }

    return matrix;

}

// Коррекция ошибок

function correctErrors(data: number[], parityMatrix: number[][], dataLength: number): number[] {

    const redundancyLength: number = calculateRedundancyLength(dataLength);

    const totalLength: number = dataLength + redundancyLength;

    const originalParityBits: number[] = new Array(redundancyLength).fill(0);

    for (const i of Array.from({ length: totalLength - dataLength }, (\_, i) => i)) {

        originalParityBits[i] = data[i + dataLength];

    }

    data = calculateSyndrome(parityMatrix, data, dataLength);

    for (const i of Array.from({ length: totalLength - dataLength }, (\_, i) => i)) {

        if (originalParityBits[i] === data[i + dataLength]) {

            data[i + dataLength] = 0;

            if (Array.from({ length: totalLength - dataLength }, (\_, k) => k).filter(k => originalParityBits[k] === data[k + dataLength]).length === redundancyLength) {

                for (const k of Array.from({ length: totalLength - dataLength }, (\_, k) => k)) {

                    data[k + dataLength] = originalParityBits[k];

                }

                return data;

            }

        } else {

            data[i + dataLength] = 1;

        }

    }

    for (const i of Array.from({ length: totalLength }, (\_, i) => i)) {

        let count: number = 0;

        for (const j of Array.from({ length: redundancyLength }, (\_, j) => j)) {

            if (parityMatrix[i][j] === data[j + dataLength]) {

                count++;

            }

        }

        if (count === redundancyLength) {

            data[i] = (data[i] + 1) % 2;

        }

    }

    data = calculateSyndrome(parityMatrix, data, dataLength);

    return data;

}

// Расчет синдрома

function calculateSyndrome(parityMatrix: number[][], data: number[], dataLength: number): number[] {

    const redundancyLength: number = calculateRedundancyLength(dataLength);

    const totalLength: number = dataLength + redundancyLength;

    const syndrome: number[] = new Array(redundancyLength).fill(0);

    for (const i of Array.from({ length: redundancyLength }, (\_, i) => i)) {

        let l: number = 0;

        for (const j of Array.from({ length: dataLength }, (\_, j) => j)) {

            if (parityMatrix[j][i] === 1 && data[j] === 1) {

                l++;

            } else {

                syndrome[i] = 0;

            }

        }

        if (l % 2 === 1) {

            syndrome[i] = 1;

        } else {

            syndrome[i] = 0;

        }

    }

    for (const i of Array.from({ length: redundancyLength }, (\_, i) => i)) {

        data[i + dataLength] = syndrome[i];

    }

    return data;

}

// Преобразование строки в массив

function stringToArray(binaryString: string, dataLength: number): number[] {

    const redundancyLength: number = calculateRedundancyLength(dataLength);

    const array: number[] = new Array(binaryString.length + redundancyLength).fill(0);

    for (const i of Array.from({ length: binaryString.length }, (\_, i) => i)) {

        if (binaryString[i] === '1') {

            array[i] = 1;

        } else {

            array[i] = 0;

        }

    }

    return array;

}

// Вывод двумерного массива (матрицы)

function outputArray(array: number[][], dataLength: number): void {

    const totalLength: number = calculateRedundancyLength(dataLength) + dataLength;

    for (const row of array) {

        for (const element of row) {

            process.stdout.write(element.toString());

        }

        process.stdout.write("|"); // Разделитель между строками

        console.log(); // Переход на новую строку

    }

    console.log(); // Дополнительный переход на новую строку для красоты

}

// Основная логика программы

function main(): void {

    const input: string = "010101"; // Пример входных данных

    const dataLength: number = input.length;

    const binaryInput: string = input;

    console.log("Ваша строка в бинарном виде: " + binaryInput);

    const parityMatrix: number[][] = generateParityMatrix(dataLength);

    console.log("\nПроверочная матрица:");

    outputArray(parityMatrix, dataLength);

    const data: number[] = stringToArray(binaryInput, dataLength);

    console.log("\nВходной массив:");

    outputArray([data], dataLength);

    console.log("\nМассив с ошибкой:");

    // Предположим, что в данных есть ошибка, и попробуем ее исправить

    data[3] = (data[3] + 1) % 2;

    outputArray([data], dataLength);

    console.log("\nМассив с вектором ошибки:");

    const correctedData: number[] = correctErrors(data, parityMatrix, dataLength);

    outputArray([correctedData], dataLength);

}

// Вызов основной логики программы

main();

Листинг 1 – задания

Результат работы итогового кода:

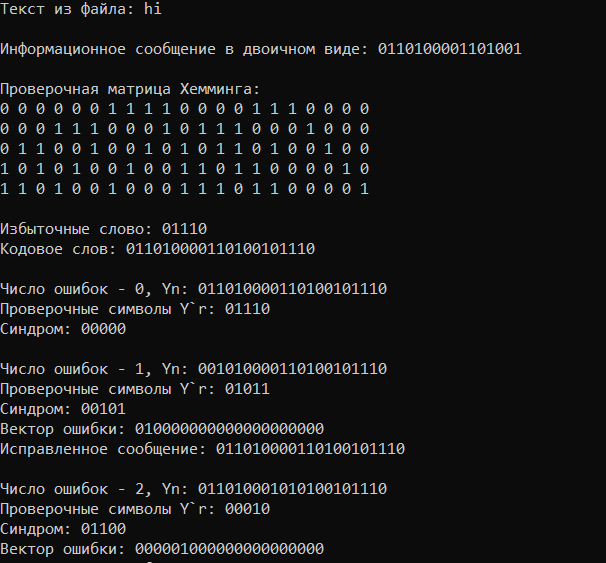


Рисунок 1 ­­­­– Результат работы программы

**Вывод:** в ходе лабораторной работы были приобретены практические навыки кодирования/декодирования двоичных данных при использовании кода Хемминга.