

Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ»

Защита информации и надёжность информационных систем

Студент: Лопатнюк П.В.
ФИТ 3 курс 1 группа
Преподаватель: Нистюк О.А.

Минск 2024

Лабораторная работа № 6

ИЗБЫТОЧНОЕ КОДИРОВАНИЕ ДАННЫХ В ИНФОРМАЦИОННЫХ СИСТЕМАХ. ЦИКЛИЧЕСКИЕ КОДЫ

Цель: приобретение практических навыков кодирования/декодирования двоичных данных при использовании циклических кодов (ЦК).

Задачи:

1. Закрепить теоретические знания по алгебраическому описанию и использованию ЦК для повышения надежности передачи и хранения в памяти компьютера двоичных данных, для контроля интегральности файлов информации.
2. Разработать приложение для кодирования/декодирования двоичной информации циклическим кодом.
3. Результаты выполнения лабораторной работы оформить в виде описания разработанного приложения, методики выполнения экспериментов с использованием приложения и результатов эксперимента.

Теоретические сведения

Циклические коды – это семейство помехоустойчивых кодов, одной из разновидностей которых являются коды Хемминга.

Основные свойства ЦК:

- относятся к классу линейных, систематических;
- сумма по модулю 2 двух разрешенных кодовых комбинаций дает также разрешенную кодовую комбинацию; каждый вектор (кодое слово), получаемый из исходного кодового вектора путем циклической перестановки его символов, также является разрешенным кодовым вектором; к примеру, если кодовое слово имеет следующий вид: 1101100, то разрешенной кодовой комбинацией будет и такая: 0110110;
- при простейшей циклической перестановке символы кодового слова перемещаются слева направо на одну позицию, как в приведенном примере;
- поскольку к числу разрешенных кодовых комбинаций ЦК относится нулевая комбинация 000...00, то минимальное кодовое расстояние d_{\min} для ЦК определяется минимальным весом разрешенной кодовой комбинации;
- циклический код не обнаруживает только такие искаженные помехами кодовые комбинации, которые приводят к появлению на стороне приема других разрешенных комбинаций этого кода;
- в основе описания и использования ЦК лежит полином или многочлен некоторой переменной (обычно X).

Характеризуя ЦК в общем случае, обычно отмечают следующее: ЦК составляют множество многочленов $\{B_j(X)\}$ степени r (r – число проверочных символов в кодовом слове), кратных порождающему (образующему) полиному $G(X)$ степени r , который должен быть делителем

бинома $X^n + 1$, т. е. остаток после деления бинома на $G(X)$ должен быть нулевым.

Порождающими могут быть только такие полиномы, которые являются делителями двучлена (бинома) $X^z + 1$: $(X^z + 1) / G(X) = H(X)$ при нулевом остатке: $R(X) = 0$.

Возможные порождающие полиномы для различных длин k информационного слова, найденные с помощью ЭВМ, сведены в обширные таблицы.

Синдромом ошибки в этих кодах является наличие остатка от деления принятой кодовой комбинации на порождающий полином. Если синдром равен нулю, то считается, что ошибок нет. В противном случае с помощью полученного синдрома можно определить номер разряда принятой кодовой комбинации, в котором произошла ошибка, и исправить ее примерно по той же схеме, которую мы использовали для кода Хемминга. При этом следует обратить внимание на важную деталь: умножение полинома на x приводит к сдвигу членов полинома на один разряд влево, а при умножении на x^r – на r разрядов влево с заменой r младших разрядов полинома на нули. Деление же полинома на x приводит к соответствующему сдвигу членов полинома вправо с уменьшением показателей членов на 1. Такой сдвиг требует дописать справа r проверочных символов к исходной кодовой комбинации $A_i(X)$ после умножения ее на x^r .

Деление полиномов позволяет представить кодовые слова в виде блочного кода, т. е. информационных $X_k (A_i(X))$ и проверочных $X_r (R_i(X))$ символов. Поскольку число последних равно r , то для компактной их записи в младшие разряды кодового слова надо предварительно к кодируемому (информационному) слову $A_i(X)$ справа дописать r нулевых символов.

Основная операция: принятое кодовое слово (Y_n) нужно поделить на порождающий полином, который использовался при кодировании. Если Y_n принадлежит коду, т. е. слово не искажено помехами, то остаток от деления (синдром) будет нулевым. Ненулевой ток свидетельствует о наличии ошибок в принятой кодовой комбинации: $Y_n \neq X_n$.

Декодирование ненулевого синдрома имеет целью определение ошибочного бита в принятом сообщении или, иначе говоря, определение вектора E_n . Наряду с полиномиальным способом задания кода, структуру построения кода можно определить с помощью матричного представления. При этом в ряде случаев проще реализуется построение кодирующих и декодирующих устройств ЦК. Здесь нам следует вернуться к упомянутым матрицам и представлению ЦК с их помощью.

Практическое задание

1. Задание выполняется по указанию преподавателя в соответствии с вариантом из таблицы 1.1, из которого выбирается порождающий полином ЦК, а по значению соответствующего ему значения r – длина k информационного слова X_k . Полагаем, что каждый полином соответствует

коду, обнаруживающему и исправляющему одиночные ошибки в кодовых словах. Определить параметры (n, k)-кода для своего варианта. Основой задания является разработка приложения.

Таблица 1.1

Вариант	Количество избыточных символов кода, r	Полином
8	5	$x^5 + x^4 + x^3 + x^2 + 1$

Задаём параметры для выполнения:

```
int r = 5;
int n = 10;
int k = n - r;

int[] g = { 1, 1, 1, 1, 0, 1 };
```

Листинг 1.1 – Параметры для выполнения.

2. Составить порождающую матрицу (n, k)-кода в соответствии с формулой, трансформировать ее в каноническую форму и далее – в проверочную матрицу канонической формы.

```
static int[,] GenerateGeneratorMatrix(int k, int r, int[] g)
{
    int n = k + r;
    int[,] G = new int[k, n];

    for (int i = 0; i < k; i++)
    {
        for (int j = 0; j <= r; j++)
        {
            G[i, i + j] = g[j];
        }
    }

    return G;
}

static void ToCanonicalForm(int[,] G, int k, int n)
{
    Console.WriteLine("\nПреобразование в каноническую форму:");
    for (int i = 0; i < k; i++)
    {
        if (G[i, i] == 0)
        {
            for (int j = i + 1; j < k; j++)
            {
                if (G[j, i] == 1)
                {
                    Console.WriteLine($"Строка {i} += Строка {j}");
                    for (int t = 0; t < n; t++)
                        G[i, t] ^= G[j, t];
                    break;
                }
            }
        }
    }
}
```

```

    }

    for (int j = 0; j < k; j++)
    {
        if (j != i && G[j, i] == 1)
        {
            Console.WriteLine($"Строка {j} += Строка {i}");
            for (int t = 0; t < n; t++)
                G[j, t] ^= G[i, t];
        }
    }
}
}

```

Листинг 2.1 – Функция для составления порождающей матрицы и приведения к каноническому виду.

```

        static int[,] GenerateHMatrix(int[,] G, int k, int n)
    {
        int r = n - k;
        int[,] H = new int[r, n];

        for (int i = 0; i < r; i++)
        {
            for (int j = 0; j < k; j++)
                H[i, j] = G[j, k + i];

            H[i, k + i] = 1;
        }

        return H;
    }

```

Листинг 2.3 – Функции для составления проверочной матрицы.

```

        int[,] G = GenerateGeneratorMatrix(k, r, g);
        Console.WriteLine("Порождающая матрица G:");
        for (int i = 0; i < k; i++)
        {
            for (int j = 0; j < n; j++)
                Console.Write(G[i, j] + " ");
            Console.WriteLine();
        }

        ToCanonicalForm(G, k, n);

        Console.WriteLine("\nКанонический вид матрицы G:");
        for (int i = 0; i < k; i++)
        {
            for (int j = 0; j < n; j++)
                Console.Write(G[i, j] + " ");
            Console.WriteLine();
        }

        int[,] H = GenerateHMatrix(G, k, n);

        Console.WriteLine("\nПроверочная матрица H:");

```

```

for (int i = 0; i < r; i++)
{
    for (int j = 0; j < n; j++)
        Console.Write(H[i, j] + " ");
    Console.WriteLine();
}

```

Листинг 2.3 – Выполняем составление матриц.

```

Преобразование в каноническую форму:
Количество избыточных символов кода, r=5
Полином
x5 + x3 + x2 + x + 1
Порождающая матрица G:
1 0 0 0 0 0 0 1 0 1
0 1 0 0 0 1 1 1 0 0
0 0 1 0 0 0 1 1 1 0
0 0 0 1 0 0 0 1 1 1
0 0 0 0 1 1 1 1 0 1

Каноническая порождающая матрица G:
1 0 0 0 0 0 0 1 0 1
0 1 0 0 0 1 1 1 0 0
0 0 1 0 0 0 1 1 1 0
0 0 0 1 0 0 0 1 1 1
0 0 0 0 1 1 1 1 0 1

Проверочная матрица H:
0 1 0 0 1 1 0 0 0 0
0 1 1 0 1 0 1 0 0 0
1 1 1 1 1 0 0 1 0 0
0 0 1 1 0 0 0 0 1 0
1 0 0 1 1 0 0 0 0 1

```

Рисунок 2.1 – Результат работы программы

3. Используя порождающую матрицу ЦК, вычислить избыточные символы (слово X_r) кодового слова X_n и сформировать это кодовое слово.

```

static int[] MultiplyVectorByMatrix(int[] vector, int[,] matrix)
{
    int[] result = new int[matrix.GetLength(1)];
    for (int j = 0; j < matrix.GetLength(1); j++)
    {
        int sum = 0;
        for (int i = 0; i < vector.Length; i++)
            sum ^= vector[i] * matrix[i, j];
        result[j] = sum;
    }
    return result;
}

```

Листинг 3.1 – Функция для вычисления избыточных символов

```
Введите информационное слово длины 5: 10110
Кодовое слово Xn: 1011001100
```

Рисунок 3.1 – Результат работы программы

4. Принять кодовое слово Y_n со следующим числом ошибок: 0; 1; 2. Позиция ошибки определяется (генерируется) случайным образом.

```
static int[] IntroduceErrors(int[] codeword, int errorCount, out
List<int> flippedPositions)
{
    Random rand = new Random();
    int[] corrupted = (int[])codeword.Clone();
    flippedPositions = new List<int>();

    while (flippedPositions.Count < errorCount)
    {
        int pos = rand.Next(codeword.Length);
        if (!flippedPositions.Contains(pos))
        {
            corrupted[pos] ^= 1;
            flippedPositions.Add(pos);
        }
    }

    flippedPositions.Sort();
    return corrupted;
}
```

Листинг 4.1 – Функция для генерации ошибок

5. Для полученного слова Y_n вычислить и проанализировать синдром. В случае, если анализ синдрома показал, что информационное сообщение было передано с ошибкой (или 2 ошибками), сгенерировать унарный вектор ошибки $E_n = e_1, e_2, \dots, e_n$ и исправить одиночную ошибку, используя выражение (6.5); проанализировать ситуацию при возникновении ошибки в 2 битах.

```
static int[] CorrectSingleError(int[] syndrome, int[,] H, int[]
received, out int errorPosition)
{
    int r = H.GetLength(0);
    int n = H.GetLength(1);
    errorPosition = -1;

    for (int j = 0; j < n; j++)
    {
        bool match = true;
        for (int i = 0; i < r; i++)
        {
            if (H[i, j] != syndrome[i])
            {
                match = false;
                break;
            }
        }
    }
}
```

```

        }
    }
    if (match)
    {
        errorPosition = j;
        break;
    }
}

int[] corrected = (int[])received.Clone();
if (errorPosition != -1)
    corrected[errorPosition] ^= 1;

return corrected;
}

static void AnalyzeWithErrorCount(int[] codeword, int[,] H, int
errorCount, int n)
{
    Console.WriteLine($"\\n--- Анализ с {errorCount} ошибкой(ами) ---
");

    List<int> errorPositions;
    int[] received = IntroduceErrors(codeword, errorCount, out
errorPositions);

    Console.WriteLine("Yn: " + string.Join("", received));
    if (errorCount > 0)
        Console.WriteLine("Сгенерированные ошибки на позициях: " +
string.Join(", ", errorPositions));
    else
        Console.WriteLine("Ошибка не была сгенерирована");

    int[] syndrome = ComputeSyndrome(H, received);
    Console.WriteLine("Синдром: " + string.Join("", syndrome));

    if (syndrome.All(bit => bit == 0))
    {
        Console.WriteLine("Ошибок не обнаружено.");
    }
    else
    {
        int errorPos;
        int[] corrected = CorrectSingleError(syndrome, H, received,
out errorPos);

        if (errorPos != -1)
        {
            Console.WriteLine($"Обнаружена ошибка в позиции:
{errorPos}");
            Console.WriteLine("En: " +
string.Join("", Enumerable.Range(0, n).Select(i => i
== errorPos ? "1" : "0")));
            Console.WriteLine("Исправленное слово: " +
string.Join("", corrected));
        }
        else
        {

```



```

        Console.WriteLine("Синдром не соответствует одиночной
ошибке – возможно, 2 ошибки.");
        Console.WriteLine("Исправление невозможно.");
    }
}

```

Листинг 5.1 – Функции для обнаружения и исправления ошибок.

```

==== Анализ с 0 ошибкой(ами) ====
Принятое слово Yn: 1011110001
Ошибок нет.
Синдром: 00000
Ошибок не обнаружено.

==== Анализ с 1 ошибкой(ами) ====
Принятое слово Yn: 1011111001
Ошибки на позициях: 6
Синдром: 01000
Обнаружена одиночная ошибка в позиции: 6
Унарный вектор ошибки En: 0000001000
Исправленное слово:      1011110001

==== Анализ с 2 ошибкой(ами) ====
Принятое слово Yn: 1111110011
Ошибки на позициях: 1, 8
Синдром: 11110
Синдром не соответствует одиночной ошибке - возможно, 2 ошибки.
Исправление невозможно.

```

Рисунок 5.1 – Результат работы программы

Вывод

Эта программа демонстрирует основные принципы кодирования и декодирования с использованием линейного блочного циклического кода, где целью является добавление избыточности к информационному сообщению для обнаружения и исправления ошибок, возникающих при передаче по каналу с шумами; ключевыми элементами являются порождающий полином, задающий структуру кода, порождающая матрица G для кодирования информационного слова в кодовое, проверочная матрица H для вычисления синдрома, указывающего на наличие ошибок, и сам процесс кодирования, включающий добавление проверочных символов; при передаче по каналу кодовое слово может быть искажено, а декодирование включает вычисление синдрома, обнаружение ошибок и, в простых случаях, исправление одиночных ошибок путем инвертирования соответствующего бита, при этом программа демонстрирует исправление только одиночных ошибок, а более сложные коды способны исправлять большее количество, но

требуют более сложных алгоритмов; суть программы заключается в демонстрации математических операций, позволяющих создать систему, устойчивую к ошибкам, выбор подходящего кода определяет эффективность обнаружения и исправления ошибок, а также сложность реализации.