|  |  |
| --- | --- |
|  | **Министерство образования и науки Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение высшего образования**  **«Московский государственный технический университет** **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

**Отчёт по домашнему заданию №2 по курсу**

**«Сети и телекоммуникации»**

**Тема работы: "Разработка программных средств определения обнаруживающей и корректирующей способностей кода в линейных протоколах"**

Выполнил: Забурунов Леонид, РТ5-51Б

Проверил:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

20 ноября 2020 г.

ЗАЧТЕНО / НЕ ЗАЧТЕНО\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись)

Оглавление

[1. Цель выполнения работы 3](#_Toc56238202)

[2. Задание по варианту 4](#_Toc56238203)

[2.1 Формулировка задания 4](#_Toc56238204)

[2.2 Таблица значений по варианту 4](#_Toc56238205)

[3. Метод решения 5](#_Toc56238206)

[3.1 Общее описание метода решения 5](#_Toc56238207)

[3.2 Скелет разрабатываемой программы 5](#_Toc56238208)

[3.3 Класс «Program» 6](#_Toc56238209)

[3.4 Класс «Hamming» 7](#_Toc56238210)

[3.5 Класс «ErrorVector» 11](#_Toc56238211)

[4. Результаты работы 16](#_Toc56238212)

[5. Примечания 18](#_Toc56238213)

[6. Используемая литература 19](#_Toc56238214)

# Цель выполнения работы

Основной целью выполнения домашнего задания является изучение способности кода Хэмминга обнаруживать искажения при передаче сигнала путём моделирования канала связи с помощью программных инструментов.

# Задание по варианту

# Формулировка задания

Имеется дискретный канал связи, на вход которого подаётся закодированная последовательность значений информационного вектора. В канале возможно возникновение ошибок любой кратности. Вектор ошибки принимает любые значения от единицы в младшем разряде до полностью единичного вектора. После декодирования алгоритм определяет наличие ошибки и пытается её исправить. Необходимо определить обнаруживающую или корректирующую способность кода.

# Таблица значений по варианту

|  |  |  |  |
| --- | --- | --- | --- |
| Вариант | Инф. вектор | Тип кода | Способность кода |
| 7 | 10101010011 | Х[15; 11] |  |

# Метод решения

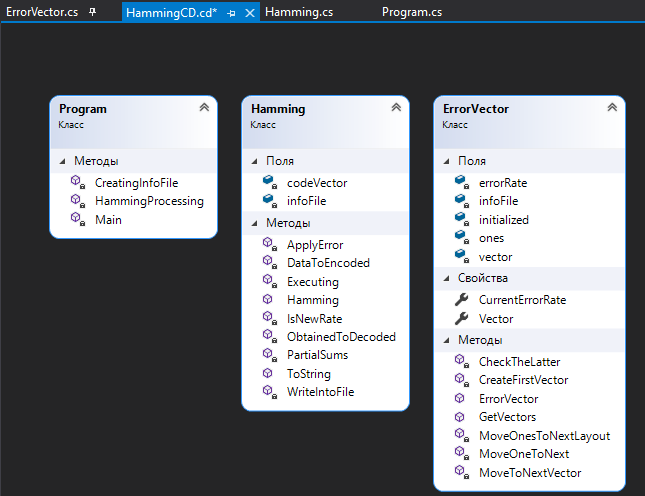
# Общее описание метода решения

Для решения поставленной задачи используется консольное приложение на языке программирования C#. По ходу моделирования всех возможных случаев поведения канала связи с точки зрения формирования искажений будем записывать расчёты программы в файл для проверки результатов.

# Скелет разрабатываемой программы

Помимо основного класса программы Program используются два пользовательских класса: Hamming и ErrorVector. Первый класс отвечает за моделирование сигнала в канале связи, второй выделен в отдельную сущность как генератор всех возможных ошибок.

Диаграмма классов для созданного проекта:



# Класс «Program»

Данный класс является инициатором работы других классов (стартовая и завершающая точка программы); его работа не несёт никаких сложностей и не требует дополнительных пояснений.

Краткое описание используемых методов:

1. *Main* – инициализатор работы модели;
2. *CreatingInfoFile* – создание файла в корне проекта;
3. *HammingProcessing* – запуск модели.

Код класса:

using System;

using System.IO;

namespace NetworksHW {

class Program {

static void Main(string[] args) {

Console.WriteLine("Забурунов Леонид, РТ5-51Б.\nДомашняя работа #2 по курсу \"Сети и телекоммуникации\"" +

"\nИтоги работы программы представлены в файле \"HammingInfo.txt\" в корневой папке проекта.");

HammingProcessing();

Console.ReadKey();

}

static void HammingProcessing() {

FileStream information = CreatingInfoFile("HammingInfo.txt");

FileStream result = CreatingInfoFile("HammingResult.txt");

Hamming ham = new Hamming(information, result);

}

/// <summary>

/// Создаём файл для записи в корне проекта

/// </summary>

/// <param name="filename">Имя файла</param>

/// <returns>Объект для потокового ввода-вывода</returns>

static FileStream CreatingInfoFile(string filename) {

// Начальный путь (в каталоге с приложением); имя файла

string path = filename;

// Получаем абсолютный путь

path = Path.GetFullPath(path);

// Поднимаеся в корень проекта

for (int i = 0; i < 4; i = i + 1) {

path = Path.GetDirectoryName(path);

}

// В итоге получили место и имя для создаваемого файла

return File.Create(Path.Combine(path, filename));

}

}

}

# Класс «Hamming»

В данном классе содержится вся работа по моделированию канала связи и записи итогов работы программы в файл.

Дадим краткое описание всем методам класса:

1. *Hamming* – конструктор класса, инициализирующий все поля и вызывающий следующий метод;
2. *Executing* – основной метод для моделирования канала связи, отсюда вызываются все операции по работе с кодом Хэмминга;
3. *DataToEncoded* – преобразование полученного информационного вектора в код Хэмминга;
4. *ApplyError* – наложение вектора ошибки на закодированный вектор;
5. *ObtainedToDecoded* – поиск ошибки в искажённом векторе;
6. *PartialSums* – вычисление проверочных сумм;
7. *WriteInfoFile* – запись строки (строка – параметр функции) в файл;
8. *IsNewRate* – проверка на получение сигнала о переходе на новое значение кратности ошибки;
9. *ToString* – преобразование кодового массива в строчный вид для занесения в файл

Код класса:

using System;

using System.IO;

using System.Text;

namespace NetworksHW {

class Hamming {

private int[] codeVector;

private FileStream infoFile;

private FileStream resultFile;

public Hamming(FileStream infoFile, FileStream resultFile) {

int[] infCodeVector = new int[11];

Console.Write("Введите информационный вектор: ");

for (int i = infCodeVector.Length - 1; i >= 0; i = i - 1) {

infCodeVector[i] = Convert.ToInt32(Console.Read() - 48);

}

codeVector = new int[15];

this.infoFile = infoFile;

this.resultFile = resultFile;

WriteIntoFile("Получен информационный вектор " + this.ToString(infCodeVector), resultFile);

Executing(infCodeVector);

}

/// <summary>

/// Основной метод для проведения операций с кодом Хэмминга

/// </summary>

/// <param name="data"></param>

private void Executing(int[] data) {

// Преобразуем в кодовый вектор

DataToEncoded(data);

WriteIntoFile("ПОЛУЧЕН КОДОВЫЙ ВЕКТОР" + this.ToString(codeVector), infoFile);

WriteIntoFile("\nПолучен кодовый вектор " + this.ToString(codeVector), resultFile);

ErrorVector error = new ErrorVector(infoFile, resultFile);

// Кол-во итераций на текущей кратности

int iterationCounter = 0;

// Обнаружено на текущей кратности

int detectionCounter = 0;

// Кол-во итераций всего

int totalIterCounter = 0;

// Обнаружено всего

int totalDetCounter = 0;

// Перебираем все вектора ошибки

foreach (int[] err in error.GetVectors()) {

totalIterCounter = totalIterCounter + 1;

// Итератор выбрасывает нулевой вектор при переходе на новую кратность в качестве сигнала

if (IsNewRate(err)) {

// Записываем в файлы промежуточные итоги и обновляем счётчики

if (error.CurrentErrorRate != 1) {

WriteIntoFile("\n\tОБНАРУЖЕНО " + detectionCounter.ToString() + " ОШИБОК.", infoFile);

double detectionRate = (double)detectionCounter / iterationCounter;

WriteIntoFile("\n\tВсего " + iterationCounter + " комбинаций" + "\n\tНайдено " + detectionCounter + " ошибок" + "\n\tОбнаруживающая способность = " + System.Math.Round(detectionRate, 3), resultFile);

}

WriteIntoFile("\nДля кратности " + error.CurrentErrorRate, resultFile);

WriteIntoFile("\nПРОИЗВОДИМ РАСЧЁТ ДЛЯ КРАТНОСТИ " + error.CurrentErrorRate, infoFile);

totalDetCounter = totalDetCounter + detectionCounter;

iterationCounter = 0;

detectionCounter = 0;

} else {

iterationCounter = iterationCounter + 1;

// Моделируем канал связи и записываем в файл итоги работы

WriteIntoFile("\n\tДля вектора ошибки " + this.ToString(err) + ":", infoFile);

// Получаемый искажённый вектор

int[] distortedCodeVector = ApplyError(err);

WriteIntoFile("\n\t\tВектор с наложенной ошибкой: " + this.ToString(distortedCodeVector), infoFile);

// Работаем с искажённым вектором

int syndromeBit;

ObtainedToDecoded(ref distortedCodeVector, out syndromeBit);

if (syndromeBit != 0) {

WriteIntoFile("\n\t\tОбнаружена ошибка в " + syndromeBit.ToString() + " бите." + "\n\t\tИсправленный вектор: " + this.ToString(distortedCodeVector), infoFile);

detectionCounter = detectionCounter + 1;

} else {

WriteIntoFile("\n\t\tОшибка не обнаружена", infoFile);

}

}

}

WriteIntoFile("\nВСЕГО ОБНАРУЖЕНО " + totalDetCounter.ToString() + " ОШИБОК.", infoFile);

}

/// <summary>

/// Преобразуем полученную информацию в кодовый вектор

/// </summary>

/// <param name="data">Исходный вектор</param>

private void DataToEncoded(int[] data) {

// Получив информационный вектор, необходимо заполнить все поля кодового

// Информационные поля...

codeVector[2] = data[0];

codeVector[4] = data[1];

codeVector[5] = data[2];

codeVector[6] = data[3];

codeVector[8] = data[4];

codeVector[9] = data[5];

codeVector[10] = data[6];

codeVector[11] = data[7];

codeVector[12] = data[8];

codeVector[13] = data[9];

codeVector[14] = data[10];

// Проверочные поля...

int[] checkingBits = PartialSums(codeVector, false);

codeVector[0] = checkingBits[0];

codeVector[1] = checkingBits[1];

codeVector[3] = checkingBits[2];

codeVector[7] = checkingBits[3];

}

/// <summary>

/// Наложение на кодовый вектор текущего значения вектора ошибки

/// </summary>

/// <param name="errorVector">Вектор ошибки</param>

/// <returns>Искажённый вектор</returns>

private int[] ApplyError(int[] errorVector) {

int[] distorted = new int[15];

// Не изменяем исходный вектор, чтобы потом проделать ту же операцию на следующих итерациях

codeVector.CopyTo(distorted, 0);

for (int i = 0; i < errorVector.Length; i = i + 1) {

// Если текущий бит ошибки равен 1, то инвертируем бит кода

if (errorVector[i] == 1) {

if (distorted[i] == 1) {

distorted[i] = 0;

} else {

distorted[i] = 1;

}

}

}

return distorted;

}

/// <summary>

/// Вычисление синдрома ошибки и инвертирование бита по найденному порядковому номеру

/// </summary>

/// <param name="vector">Искажённый вектор</param>

/// <param name="bitNumber">Бит, на который указывает синдром ошибки</param>

private void ObtainedToDecoded(ref int[] vector, out int bitNumber) {

// Определяем синдром ошибки

int[] errorSyndrome = PartialSums(vector, true);

// Получаем бит синдрома ошибки

string syndromeStr = "";

foreach (int bit in errorSyndrome) {

syndromeStr = bit.ToString() + syndromeStr;

}

bitNumber = Convert.ToInt32(syndromeStr, 2);

if (bitNumber != 0) {

if (vector[bitNumber - 1] == 1) {

vector[bitNumber - 1] = 0;

} else {

vector[bitNumber - 1] = 1;

}

}

}

/// <summary>

/// Получение проверочных битов с помощью сложения по модулю 2

/// </summary>

/// <param name="vector">Кодовый вектор, из которого подсчитываем контрольные суммы</param>

/// <param name="isDecoding">Данный параметр определяет, будут ли для подсчёта контрольных сумм использоваться проверочные биты</param>

/// <returns>Массив проверочных битов</returns>

private int[] PartialSums(int[] vector, bool isDecoding) {

int[] values = new int[4] { 0, 0, 0, 0 };

// Получаем сумму из информационных битов

values[0] = vector[2] + vector[4] + vector[6] + vector[8]

+ vector[10] + vector[12] + vector[14];

values[1] = vector[2] + vector[5] + vector[6] + vector[9]

+ vector[10] + vector[13] + vector[14];

values[2] = vector[4] + vector[5] + vector[6] + vector[11]

+ vector[12] + vector[13] + vector[14];

values[3] = vector[8] + vector[9] + vector[10] + vector[11]

+ vector[12] + vector[13] + vector[14];

// При подсчёте синдрома ошибки так же добавляем проверочные биты

if (isDecoding) {

values[0] = values[0] + vector[0];

values[1] = values[1] + vector[1];

values[2] = values[2] + vector[3];

values[3] = values[3] + vector[7];

}

// Отрезаем сумму по модулю 2

for (int i = 0; i < values.Length; i = i + 1) {

values[i] = values[i] % 2;

}

return values;

}

/// <summary>

/// Запись текстовой информации в файл

/// </summary>

/// <param name="file">Файл, в который заносим строку</param>

/// <param name="strToWrite">Строка, которую заносим в файл</param>

private void WriteIntoFile(string strToWrite, FileStream file) {

byte[] written = new UTF8Encoding().GetBytes(strToWrite);

file.Write(written, 0, written.Length);

}

/// <summary>

/// Метод определяет, исчерпал ли итератор варианты вектора ошибок данной кратности.

/// Это позволяется из-за того, что в качестве сигнала перед переходом на следующую кратность итератор выбрасывает нулевой вектор

/// </summary>

/// <param name="errorVector">Полученное значение вектора ошибки</param>

/// <returns>Произошёл ли переход на новый показатель кратности ошибки?</returns>

private bool IsNewRate(int[] errorVector) {

for (int i = 0; i < errorVector.Length; i = i + 1) {

if (errorVector[i] != 0) return false;

}

return true;

}

/// <summary>

/// Текстовое представление битового массива для вывода в файл

/// </summary>

/// <param name="intArray">Массив нулей и единиц</param>

/// <returns>Строка, соответствующая полученному массиву</returns>

public string ToString(int[] intArray) {

string s = "";

for (int i = intArray.Length - 1; i >= 0; i = i - 1) {

s = s + intArray[i].ToString();

}

return s;

}

}

}

# Класс «ErrorVector»

В данном классе содержится функционал по генерации вектора ошибки.

Краткое описание методов класса:

1. *ErrorVector* – конструктор, инициализирующий поля экземпляра класса;
2. *GetVectors* – основной элемент данного класса, - итератор, проходящий по всем возможным значениям вектора ошибки. Итератор не использует последовательное преобразование в двоичный вид целых чисел от 1 до 32767 (дабы не нагромождать отчёт, ссылаюсь на файл *HammingInfo.txt*, в котором можно посмотреть порядок обхода векторов ошибки).   
   Несмотря на кажущуюся простоту такого (от 1 до 32767) алгоритма на этапе получения вектора, на программу возлагается дополнительная нагрузка по определению количества единиц в текущем векторе «на лету»; нелинейность обхода создаёт дополнительные сложности, которые исключены в алгоритме, представленном ниже. Выбранный алгоритм тратит больше времени на переход к следующему коду ошибки - при грубой оценке разница в работе составила 20 % (в среднем для полной работы программы результаты составили 60мс и 72мс), - но в рамках учебной задачи это допустимо. Вдобавок, при таком подходе мы избегаем необходимости тратить дополнительные ресурсы на подсчёт числа сочетаний, поскольку за это отвечает простой счётчик;
3. *CheckTheLatter* – проверка, является ли текущий вектор ошибки последним из всех векторов ошибки, возможных для данного значения кратности;
4. *CreateFirstVector* – создание начального вектора ошибки при переходе на новое значение кратности;
5. *MoveToNextVector* – переход к следующему вектору в рамках текущей кратности ошибки. Следующие два метода вызываются из данного;
6. *MoveOneToNext* – сдвиг единички на один шаг влево;
7. *MoveOnesToNextLayout* – перенос первой неупёршейся единицы на один шаг влево и установка всех упёршихся в следующие позиции.

Код класса:

using System.IO;

using System.Collections;

using System.Collections.Generic;

namespace NetworksHW {

class ErrorVector {

// Текущий вектор ошибки

private int[] vector;

// Для проверки существования вектора данной кратности в классе

private bool initialized;

// Текущее значение кратности

private int errorRate;

// Храним индексы единиц

private List<int> ones;

// Файл, в который будем записывать всю информацию (в консоль не помещается :) )

private FileStream infoFile;

// Файл, в который будем записывать результат

private FileStream resultFile;

/// <summary>

/// Текущий вектор ошибки

/// </summary>

public int[] Vector { get { return this.vector; } }

/// <summary>

/// Текущее значение кратности ошибки

/// </summary>

public int CurrentErrorRate { get { return this.errorRate; } }

public ErrorVector(FileStream infoFile, FileStream resultFile) {

vector = new int[15] { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

initialized = false;

errorRate = 0;

ones = new List<int>();

this.infoFile = infoFile;

this.resultFile = resultFile;

}

/// <summary>

/// Итератор, перебирающий все комбинации векторов всех кратностей ошибок от 1 до 15 и выбрасывающий все эти комбинации.

/// </summary>

/// <returns>Текущий вектор</returns>

public IEnumerable GetVectors() {

while (true) {

// Если мы ещё не создали вектор для текущей кратности

if (!initialized) {

// Проверяем границы диапазона значений кратности

if (errorRate == 15) {

// Отправляем сигнал об окончании перебора векторов данной кратности и завершаем работу итератора

yield return new int[15] { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

yield break;

} else {

// Переходим к новой кратности

CreateFirstVector();

// Оповещаем о том, что произошёл переход на новую кратность

yield return new int[15] { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

}

}

// Итерируемся по текущему значению кратности, пока не дойдём до последнего

// (если дошли до последнего, то функция проверки снимает флаг initialized)

yield return vector;

if (!CheckTheLatter()) {

MoveToNextVector();

}

}

}

/// <summary>

/// Проверяем, является ли текущий вектор ошибки последним из возможных для данной кратности

/// </summary>

/// <returns>Является ли?</returns>

private bool CheckTheLatter() {

/\* Последним вектором считается тот,

\* который имеет в n старших разрядах вектора единицы,

\* а во всех остальных - нули (n - кратность ошибки).\*/

int currentIndex = vector.Length - 1;

while (currentIndex >= vector.Length - errorRate) {

if (vector[currentIndex] == 0) {

return false;

}

currentIndex = currentIndex - 1;

}

initialized = false;

ones.Clear();

return true;

}

/// <summary>

/// Создаём начальный вектор для нового значения кратности

/// </summary>

private void CreateFirstVector() {

/\* Первым вектором считается тот,

\* который имеет в n младших разрядах единицы,

\* а во всех остальных - нули (n - кратность ошибки)

\*/

errorRate = errorRate + 1;

initialized = true;

// Заносим начальный вектор

for (int i = 0; i < vector.Length; i = i + 1) {

if (i < errorRate) {

ones.Add(i);

vector[i] = 1;

} else {

vector[i] = 0;

}

}

}

/// <summary>

/// Переходим к следующему вектору данной кратности

/// </summary>

private void MoveToNextVector() {

if (ones[errorRate - 1] != vector.Length - 1) {

// Если доступна, то просто сдвигаем её

MoveOneToNext(errorRate - 1);

} else {

// Если же недоступна, то необходимо перегруппировать единицы в векторе

// Определяем первый доступный для сдвига уровень

int currentLevel = errorRate - 1;

while (currentLevel > 0) {

currentLevel = currentLevel - 1;

if (vector[ones[currentLevel] + 1] == 0 && vector[ones[currentLevel]] == 1) { // Второе условие добавлено исключительно для читаемости кода

break;

}

}

MoveOnesToNextLayout(currentLevel);

}

}

/// <summary>

/// Сдвиг одной единицы на одну позицию влева

/// (с изменением индекса внутри массива единиц)

/// </summary>

/// <param name="oneLevel">Текущее положение сдвигаемой единички</param>

private void MoveOneToNext(int oneLevel) {

vector[ones[oneLevel]] = 0;

vector[ones[oneLevel] + 1] = 1;

ones[oneLevel] = ones[oneLevel] + 1;

}

/// <summary>

/// Установка всех упёршихся в край массива единиц в новое расположение

/// (с изменениями индексов внутри массива единиц)

/// </summary>

/// <param name="oneLevel">

/// Положение первой единицы, доступной для сдвига на 1

/// (остальные будут передвинуты в соседние для этой единицы позиции)

/// </param>

private void MoveOnesToNextLayout(int oneLevel) {

// Сдвинем предыдущую единицу вперёд

vector[ones[oneLevel]] = 0;

vector[ones[oneLevel] + 1] = 1;

ones[oneLevel] = ones[oneLevel] + 1;

// После сдвига пред. единицы начинаем возвращать по цепочке все упёршиеся

int currentIndex = ones[oneLevel] + 1,

currentLevel = oneLevel + 1;

while (currentLevel < errorRate) {

vector[ones[currentLevel]] = 0;

vector[currentIndex] = 1;

ones[currentLevel] = currentIndex;

currentLevel = currentLevel + 1;

currentIndex = currentIndex + 1;

}

}

}

}

# Результаты работы

Содержимое файла, содержащего результат работы программы:

*Получен информационный вектор 10101010011*

*Получен кодовый вектор 101010100011110*

*Для кратности 1*

*Всего 15 комбинаций*

*Найдено 15 ошибок*

*Обнаруживающая способность = 1*

*Для кратности 2*

*Всего 105 комбинаций*

*Найдено 105 ошибок*

*Обнаруживающая способность = 1*

*Для кратности 3*

*Всего 455 комбинаций*

*Найдено 420 ошибок*

*Обнаруживающая способность = 0,923*

*Для кратности 4*

*Всего 1365 комбинаций*

*Найдено 1260 ошибок*

*Обнаруживающая способность = 0,923*

*Для кратности 5*

*Всего 3003 комбинаций*

*Найдено 2835 ошибок*

*Обнаруживающая способность = 0,944*

*Для кратности 6*

*Всего 5005 комбинаций*

*Найдено 4725 ошибок*

*Обнаруживающая способность = 0,944*

*Для кратности 7*

*Всего 6435 комбинаций*

*Найдено 6000 ошибок*

*Обнаруживающая способность = 0,932*

*Для кратности 8*

*Всего 6435 комбинаций*

*Найдено 6000 ошибок*

*Обнаруживающая способность = 0,932*

*Для кратности 9*

*Всего 5005 комбинаций*

*Найдено 4725 ошибок*

*Обнаруживающая способность = 0,944*

*Для кратности 10*

*Всего 3003 комбинаций*

*Найдено 2835 ошибок*

*Обнаруживающая способность = 0,944*

*Для кратности 11*

*Всего 1365 комбинаций*

*Найдено 1260 ошибок*

*Обнаруживающая способность = 0,923*

*Для кратности 12*

*Всего 455 комбинаций*

*Найдено 420 ошибок*

*Обнаруживающая способность = 0,923*

*Для кратности 13*

*Всего 105 комбинаций*

*Найдено 105 ошибок*

*Обнаруживающая способность = 1*

*Для кратности 14*

*Всего 15 комбинаций*

*Найдено 15 ошибок*

*Обнаруживающая способность = 1*

*Для кратности 15*

*Всего 1 комбинаций*

*Найдено 0 ошибок*

*Обнаруживающая способность = 0*

# Примечания

Ссылка на разработанную программу:

https://github.com/Zaburunier/NetworksHW

# Используемая литература

1. Галкин В. А. «Методическое пособие по выполнению домашнего задания по дисциплине «Сети и телекоммуникации»» - М.: 2018
2. Hamming, R.W. (Richard Wesley), «The art of doing science and engineering: learning to learn» - Amsterdam, 1997.
3. «Документация по C#» [Электронный ресурс]: документация корпорации Microsoft к языку программирования C#. URL: https://docs.microsoft.com/ru-ru/dotnet/csharp/