|  |  |  |
| --- | --- | --- |
| Министерство науки и высшего образования  Российской Федерации | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования | | |
| «Новосибирский государственный технический университет» | | |
|  | | |
| Теоретической и прикладной математики | | |
|  | | |
| Лабораторная работа № 3 | | |
| по дисциплине «ОСНОВЫ ТЕОРИИ ИНФОРМАЦИИИ КРИПТОГРАФИИ» | | |
|  | | |
| **ПОМЕХОУСТОЙЧИВОЕ КОДИРОВАНИЕ** | | |
|  | | |
|  | Факультет: | ПМИ |
| Группа: | ПМИ-02 |
| Вариант: | 7 |
| Студент: | Сидоров Даниил, |
|  | Дюков Богдан |
| Преподаватель: | Авдеенко Татьяна Владимировна, |
|  | Сивак Мария Алексеевна. |
|
|  |  |
|  | | |
| Новосибирск | | |
| 2022 | | |

1. **Цель р****а****боты**

Освоить основные алгоритмы помехоустойчивого кодирования.

1. **Задача**

Часть 1

1. Реализовать приложение, кодирующее заданную последовательность символов X = [] по алгоритму из соответствующего варианта лабораторной работы № 2 для случая равновероятного появления символов алфавита с проверкой на четность. Проверка на четность означает, что на выходе из кодера получается сообщение Y = [], при этом биты являются проверочными. Если длина кодового слова четна, то в окончательном виде оно будет иметь вид Y = [] (например, в вариантах № 3, 4, 8, 11, 14); иначе: Y = [] . Приложение должно удовлетворять следующим требованиям:

* подлежащая кодированию последовательность символов задается через входной файл;
* закодированная последовательность сохраняется в файл.

1. Реализовать приложение для декодирования сообщения, закодированного в пункте I.1 задания к этой лабораторной работе.

Приложение должно удовлетворять следующим требованиям:

* подлежащая декодированию последовательность символов задается через входной файл;
* результатом работы программы являются все части сообщения, которые удалось раскодировать (т. е. в них не было ошибок при передаче по каналу связи), а также сообщения об ошибках с номерами позиций, в которых они произошли (если ошибки были);
* раскодированная последовательность сохраняется в один файл, а сообщения об обнаруженных ошибках – в другой либо выводятся на форму (если реализован графический интерфейс).

1. Исследования:

* протестировать разработанные программы, подавая на вход декодера как правильные, так и искаженные кодовые слова; исследовать созданный код на кратность обнаружения и исправления ошибок;
* найти кодовое расстояние кода d0, расстояние Хэмминга, границу Хэмминга, границу Плоткина, границу Варшамова–Гильберта.

Часть 2

1. Реализовать приложение, кодирующее заданную последовательность символов X = [] по алгоритму кодирования Хэмминга. Конкретный вид проверочной или порождающей матрицы задан в варианте. При этом считать, что алфавит источника A = [], и 0 00000, 1 00001, 2 00010, ..., 31 11111 для 1-го варианта.

Приложение должно удовлетворять следующим требованиям:

* подлежащая кодированию последовательность символов задается через входной файл;
* закодированная последовательность сохраняется в файл.

1. Реализовать приложение для декодирования сообщения, закодированного в пункте II.1 задания к этой лабораторной работе.

Приложение должно удовлетворять следующим требованиям:

* подлежащая декодированию последовательность символов задается через входной файл;
* результатом работы программы является раскодированное сообщение, а также сообщения об исправленных ошибках с номерами позиций (если ошибки были);
* раскодированная последовательность сохраняется в один файл, а сообщения об исправленных ошибках – в другой либо выводятся на форму (если реализован графический интерфейс).

1. Исследования:

* протестировать разработанные программы, подавая на вход декодера как правильные, так и искаженные кодовые слова; исследовать созданный код на кратность обнаружения и исправления ошибок;
* найти кодовое расстояние кода d0, расстояние Хэмминга, границу Хэмминга, границу Плоткина, границу Варшамова–Гильберта.

| Вари­ант | Проверочная или порождающая матрица  (n, k) кода Хэмминга |
| --- | --- |
| 7 |  |

**3. Метод решения задачи**

Часть 1

В задании выполняется кодирование с обнаружением ошибок, основанное на проверке на четность. На основании проверочного бита реализуется проверка на ошибку.

Для данных кодов d0 = 2, значит, кратность гарантированно распознанных ошибок = 1. Однако достоинством этого кода является то, что он распознает все сочетания ошибок нечетной кратности (то есть не только = 1, но и 3,5,7, и т. д.).

Мы использовали модифицированный алгоритм, разработанный в лабораторной работе №2.

**Кодирование**

При равновероятном появлении символов алфавита необходима проверка на четность/нечетность длины кодового слова. Для нашего варианта длина кодового слова является нечетной, значит окончательный вид кодового слова будет таким: Y = . Таким образом находим для каждого символа алфавита новое четное кодовое слово. Затем из полученного файла с текстом каждому символу ставим в соответствие кодовое слово. Полученная строка представляет собой закодированную последовательность.

**Декодирование**

Закодированный текст, поданный на вход, мы анализируем на наличие недопустимых символов (в файле должны быть только 1 и 0) и на некорректную запись (общая длина последовательности не кратна длине кодового слова). После этого мы считываем по 4 символа. Первые три – это информационные биты, мы применяем к ним операцию “xor” (сумма по модулю два) и сравниваем с 4 битом, который является проверочным. В случае совпадения значений, мы находим соответствующий символ алфавита и заменяем им данное кодовое слово. В противном случае делаем вывод, что произошла ошибка и выясняем номер позиции, в которой она произошла. Номера позиций, в которых произошла ошибка, записываем в специальное окно приложения. Символ, у которого произошла ошибка мы не записываем в декодированный файл. Полученная строка представляет собой раскодированную строку.

Часть 2

Кодом Хэмминга называется (n, k) -код с проверочной матрицей , у которой r = n−k строк и n столбцов, причем столбцы являются различными ненулевыми последовательностями.

Код Хэмминга, обеспечивающий исправление всех одиночных ошибок, должен иметь минимальное кодовое расстояние = 3. При передаче кода может быть искажён любой символ. Однако может быть и такой случай, когда ни один из символов не искажён.

Мы разработали приложение для реализации алгоритма Хэмминга. Начальные условия даны в условии задачи.

**Кодирование**

Алфавит для нашего случая имеет вид: 0 → 0000, 1 → 0001, 2 → 0010, ..., 15 → 1111.

Из условия нам дана порождающая матрица:

Последовательно умножая вектора алфавита на порождающую матрицу Хэмминга, получаем кодовые слова для каждого из символов. Кодовое слово у нас состоит из 8 символов, четыре из которых информационные (единичная матрица содержит информационные биты), остальные – проверочные. Кодируя поступившую на вход последовательность из файла, заменяем каждый символ получившимся кодовым словом. Полученная строка представляет собой закодированную строку.

**Декодирование**

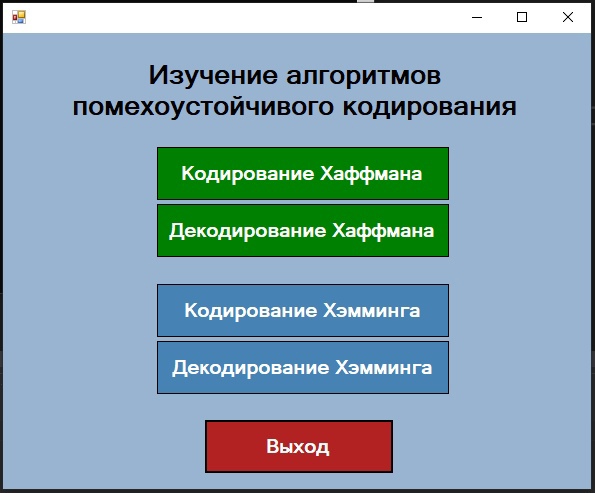
Для декодирования закодированной последовательности, нам нужна проверочная матрица , которую мы можем получить из порождающей матрицы :

То есть проверочная матрица состоит из транспонированной подматрицы, соответствующей проверочным битам порождающей матрицы, к которой добавляют справа единичную матрицу порядка r. Получаем:

Для декодирования мы считываем по 8 символов последовательности (длина нашего кодового слова = 8) и умножаем вектор кодового слова на проверочную матрицу Н, если получаем нулевой вектор (нулевой синдром), то искажений при передаче не произошло, и по кодовому слову мы можем найти соответствующий символ алфавита. Если синдром ненулевой (полученный вектор содержит единицы на некоторых позициях), значит произошла ошибка, мы вычисляем по синдрому позицию этой ошибки в векторе с кодовым словом и инвертируем нужный бит. Полученная строка представляет собой раскодированную строку.

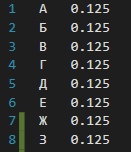
1. **Разработанное программное средство**

Разработанное программное средство представляет собой приложение Windows Forms. Вся лабораторная работа сделана в одном приложении. Главное меню:

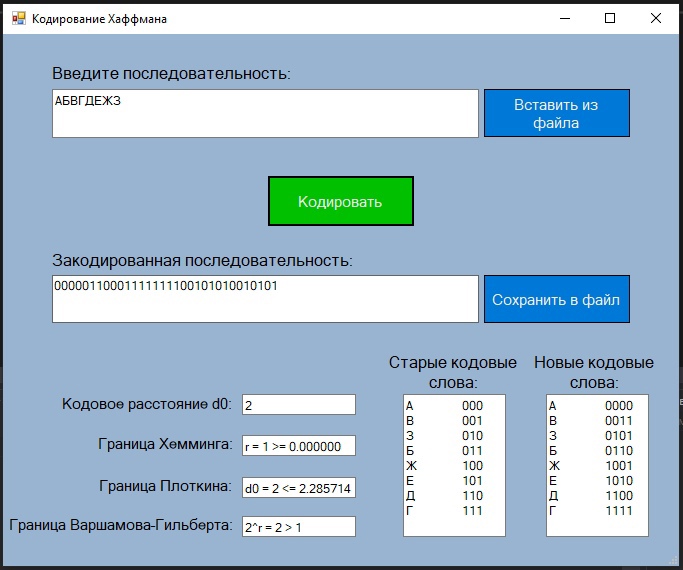


Часть 1

У пользователя есть возможность кодирования и декодирования текста. Файл Alphabet.txt с алфавитом и вероятностями появления каждого символа:

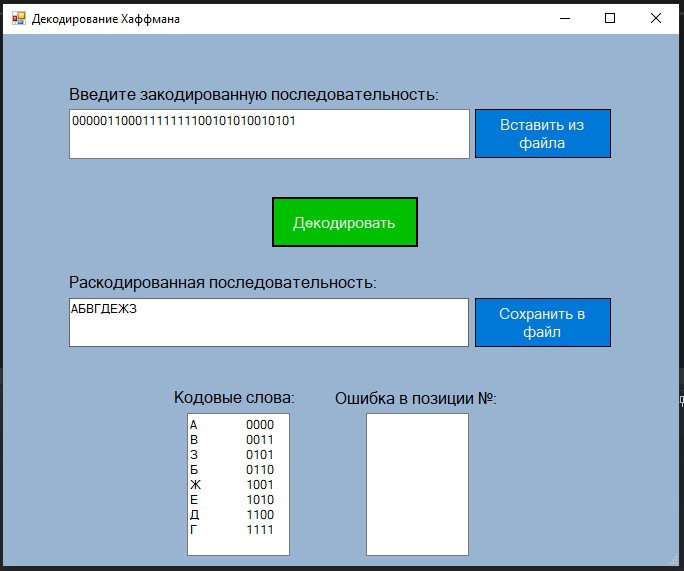


Закодируем последовательность в алфавитном порядке (получена из файла):



Каждому символу входной последовательности было поставлено в соответствие новое кодовое слово, содержащее в себе 3 информационных бита и один проверочный. Сохраним закодированную последовательность в файл.

Теперь раскодируем сохраненную последовательность:

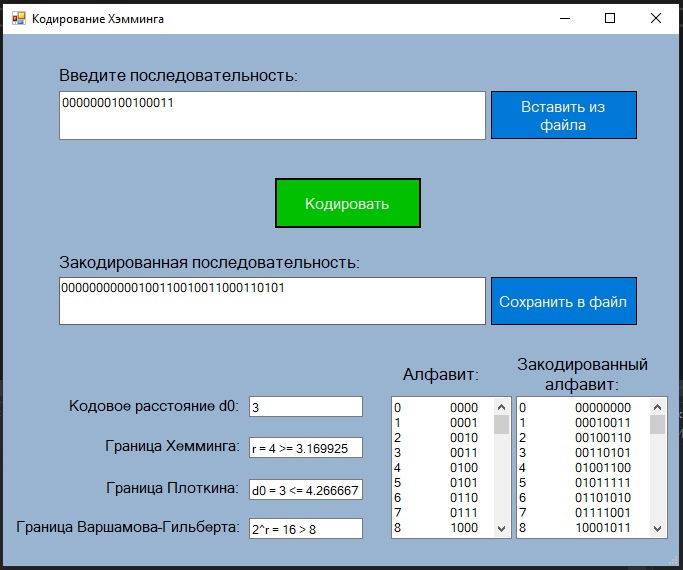


Раскодированную последовательность можно сохранить в файл. Сравнив содержимое текстбокса с исходной последовательностью и текстбокса с раскодированной последовательностью, убедились в корректности работы алгоритма (корректность можно было проверить, посмотрев содержимое файла с исходной последовательностью и файла с раскодированной последовательностью).

Часть 2

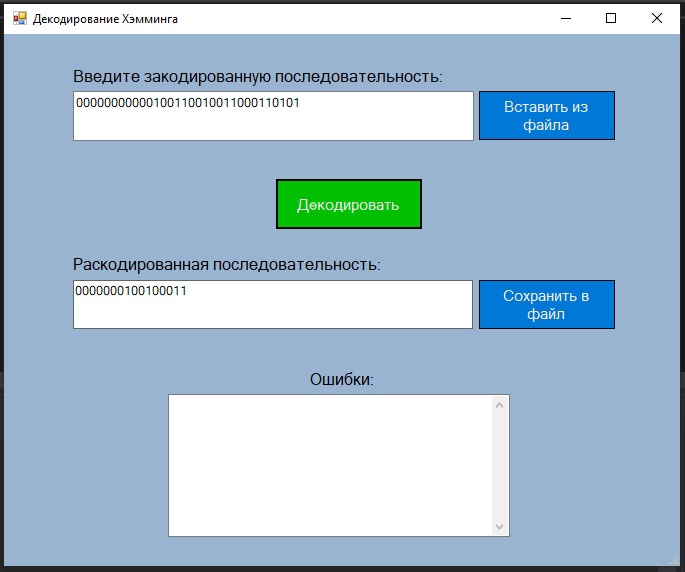
У пользователя есть возможность кодирования и декодирования текста. Следует уточнить, что заданный алфавит вшит программно.

Закодируем последовательность от 0 до 3, полученную из файла:



В отдельных окнах выводим алфавит источника и кодовые слова символов алфавита. Сохраним закодированную последовательность в файл.

Теперь раскодируем сохраненную последовательность:



Раскодированную последовательность можно сохранить в файл. Сравнив содержимое текстбокса с исходной последовательностью и текстбокса с раскодированной последовательностью, убедились в корректности работы алгоритма (корректность можно было проверить, посмотрев содержимое файла с исходной последовательностью и файла с раскодированной последовательностью).

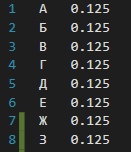
1. **Исследования**

Определение понятий:

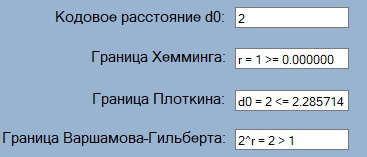
* **Расстояние Хэмминга** (кодовое расстояние между двумя кодовыми словами) d – число позиций, в которых два кодовых двоичных слова отличаются друг от друга.
* **Кодовое расстояние кода d0** – наименьшее расстояние Хэмминга между различными словами кода (наименьшее число различий в битах между двумя кодовыми словами).
* **Граница Хэмминга**: , где n – длина кодового слова, k – количество информационных разрядов, r – количество проверочных разрядов. Это выражение является нижней границей в том смысле, что оно устанавливает то минимальное соотношение корректирующих и информационных разрядов, ниже которого код не может сохранять заданные корректирующие способности.
* **Граница Плоткина**: . Границы Хэмминга и Плоткина являются нижними границами для кодового расстояния при заданных n, k, задающими минимальную избыточность, при которой существует помехоустойчивый код, имеющий минимальное кодовое расстояние и гарантированно исправляющий -кратные ошибки.
* **Граница Варшамова-Гильберта**: является нижней границей для числа проверочных разрядов r в случае кодов большой разрядности, необходимого для обеспечения заданного кодового расстояния d0.

Часть 1

Нам дан алфавит, состоящий из 8 символов с одинаковой частотой встречаемости:

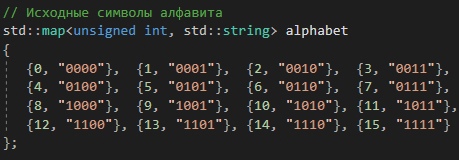


Проведенные исследования:

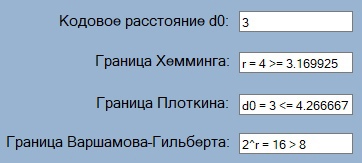


Часть 2

Нам дан алфавит, состоящий из 16 символов:



Проведенные исследования:



1. **Код программы**

Заголовочный файл EncodingAndDecodingFunctions.h:

#pragma once

#ifndef \_EncodingAndDecodingFunctions\_H

#define \_EncodingAndDecodingFunctions\_H

#include <msclr\marshal\_cppstd.h>

#include <iostream>

#include <string>

#include <queue>

#include <unordered\_map>

#include <fstream>

#include <sstream>

#include <unordered\_map>

#include <vector>

// Узел дерева

struct Node

{

std::string symbol;

double chance;

Node\* left;

Node\* right;

};

// Компаратор, который будет использоваться для упорядочивания узлов в очереди

struct comp

{

bool operator()(Node\* l, Node\* r)

{

return l->chance > r->chance;

}

};

// Создание нового узла дерева

Node\* GetNewNode(std::string ch, double freq, Node\* left, Node\* right);

// Определение кодовых слов из дерева и запись их в контейнер huffmanCode по ключу соответствующего символа

void EncodingSymbolsFromTree(Node\* root, std::string encodedText, std::unordered\_map<std::string, std::string>& huffmanCode);

// Получение закодированного текста

std::string GetEncodedText(std::unordered\_map<std::string, std::string> huffmanCode, std::string originalText);

// Расшифровка кодовых слов из дерева и последовательная запись символов в decodingText

void DecodingCodesFromTree(Node\* root, int& index, std::string decodedText, std::string& decodingText);

// Построение Хаффмановского дерева (функция возвращает адрес корня дерева)

Node\* BuildHuffmanTree(std::unordered\_map <std::string, double > symbolAndChance);

// Получение текста (последовательность алфавитных букв или кодов) из файла

std::string GetTextFromFile(std::string fileName);

// Получение символов алфавита и вероятности их появления

std::unordered\_map<std::string, double> GetAlphabet(std::string fileName);

// Запись текста (закодированного или декодированного) в файл

void WriteTextToFile(std::string fileName, std::string text);

// Проверка неравенства Крафта

bool CheckingCraftInequality(std::unordered\_map<std::string, std::string> huffmanCode);

// Получение энтропии

double GetEntrophy(std::unordered\_map<std::string, double> alphabet);

// Получение средней длины кодового слова

double GetAverageLength(std::unordered\_map<std::string, std::string> huffmanCode, std::unordered\_map<std::string, double> alphabet);

// Получение избыточности

double GetRedundancy(double entrophy, double averageLenth);

// Построение новых кодовых слов с проверочным битом

void BuildNewCodes(std::unordered\_map<std::string, std::string>& huffmanCode);

// Декодирование последовательности с проверкой на четность, запись только прошедших

// проверку кодов в переменную и сохранение номеров позиций с ошибочными битами

void NewDecodingCodes(std::string encodingText, std::string& decodingText,

std::vector<int>& positionNumber, std::unordered\_map<std::string, std::string> huffmanCode);

// Получить ключ карты по значению

std::string GetValueExistence(std::unordered\_map<std::string, std::string> huffmanCode, std::string value);

// Получение d0

int GetCodeDistance(std::unordered\_map<std::string, std::string> huffmanCode);

// Получение границы Хемминга

std::string GetHemmingBoundary(int n, int k);

// Число сочетаний

int Combinations(int n, int k);

// Граница Плоткина

std::string GetPlotkinBoundary(int n, int k, int d0);

// Граница Варшамова–Гильберта

std::string GetVarshamovGilbertaBoundary(int n, int k, int d0);

#endif

Файл EncodingAndDecodingFunctions.cpp:

#include "EncodingAndDecodingFunctions.h"

// Создание нового узла дерева

Node\* GetNewNode(std::string ch, double freq, Node\* left, Node\* right)

{

Node\* node = new Node();

node->symbol = ch;

node->chance = freq;

node->left = left;

node->right = right;

return node;

}

// Определение кодовых слов из дерева (запись кодовых слов в контейнер huffmanCode по ключу соответствующего символа)

void EncodingSymbolsFromTree(Node\* root, std::string encodedText, std::unordered\_map<std::string, std::string>& huffmanCode)

{

if (root == nullptr)

{

return;

}

// Дошли до листа, записали кодовое слово

if (!root->left && !root->right)

{

huffmanCode[root->symbol] = encodedText;

}

EncodingSymbolsFromTree(root->left, encodedText + "0", huffmanCode);

EncodingSymbolsFromTree(root->right, encodedText + "1", huffmanCode);

}

// Получение закодированного текста (генерация последовательности из значений, взятых из контейнера по ключам,

// равным символам исходного текста)

std::string GetEncodedText(std::unordered\_map<std::string, std::string> huffmanCode, std::string originalText)

{

std::string encodingText;

std::string firstSymbol;

while(originalText != "")

{

firstSymbol = originalText.substr(0, 1);

encodingText += huffmanCode[firstSymbol];

originalText.erase(0, 1);

}

return encodingText;

}

// Расшифровка кодовых слов из дерева и последовательная запись символов в decodingText

void DecodingCodesFromTree(Node\* root, int& index, std::string encodingText, std::string& decodingText)

{

if (root == nullptr)

{

return;

}

// Дошли до листа, записали символ

if (!root->left && !root->right)

{

decodingText += root->symbol;

return;

}

index++;

if (encodingText[index] == '0')

DecodingCodesFromTree(root->left, index, encodingText, decodingText);

else

DecodingCodesFromTree(root->right, index, encodingText, decodingText);

}

// Построение Хаффмановского дерева (функция возвращает адрес корня дерева)

Node\* BuildHuffmanTree(std::unordered\_map<std::string, double> symbolAndChance)

{

// Приоритетная очередь для хранения активных узлов

std::priority\_queue<Node\*, std::vector<Node\*>, comp> activeNodes;

// Добавление в приоритетную очередь созданных узлов для каждого символа последовательности

for (auto elem : symbolAndChance)

{

activeNodes.push(GetNewNode(elem.first, elem.second, nullptr, nullptr));

}

// Пока в очереди более 1 узла:

// 1) Убираем из очереди пару узлов, содержащих символы с минимальными вероятностями

// 2) Помещаем в очередь новый узел с этими двумя узлами в качестве дочерних и вероятностью, равной сумме вероятностей обоих узлов

while (activeNodes.size() != 1)

{

Node\* left = activeNodes.top();

activeNodes.pop();

Node\* right = activeNodes.top();

activeNodes.pop();

double sum = left->chance + right->chance;

activeNodes.push(GetNewNode("\0", sum, left, right));

}

return activeNodes.top();

}

// Получение закодированного текста из файла

std::string GetTextFromFile(std::string fileName)

{

std::ifstream file;

file.open(fileName);

// Eсли файл не открыт, генерируем исключение

if (!file.is\_open())

{

throw std::invalid\_argument("The file is not open");

}

std::string encodedText;

file >> encodedText;

file.close();

return encodedText;

}

// Получение символов алфавита и вероятности их появления

std::unordered\_map<std::string, double> GetAlphabet(std::string fileName)

{

std::ifstream alphabetFile;

alphabetFile.open(fileName);

// Eсли файл не открыт, генерируем исключение

if (!alphabetFile.is\_open())

{

throw std::invalid\_argument("The file is not open");

}

std::string symbol;

double chance;

std::unordered\_map<std::string, double> alphabet;

while (!alphabetFile.eof())

{

alphabetFile >> symbol;

alphabetFile >> chance;

alphabet[symbol] = chance;

}

alphabetFile.close();

return alphabet;

}

// Запись декодированного текста в файл

void WriteTextToFile(std::string fileName, std::string text)

{

std::ofstream file;

file.open(fileName);

// Eсли файл не открыт, генерируем исключение

if (!file.is\_open())

{

throw std::invalid\_argument("The file is not open");

}

file << text;

file.close();

}

// Проврка неравенства Крафта

bool CheckingCraftInequality(std::unordered\_map<std::string, std::string> huffmanCode)

{

auto k = huffmanCode.size();

double sum = 0.0;

for (auto elem : huffmanCode)

sum += pow(2.0, -1.0 \* elem.second.length());

return sum <= 1;

}

// Получение энтропии

double GetEntrophy(std::unordered\_map<std::string, double> alphabet)

{

double entrophy = 0.0;

for (auto elem : alphabet)

if (elem.second > 0.0)

entrophy += elem.second \* log2(elem.second);

return -1.0 \* entrophy;

}

// Получение средней длины

double GetAverageLength(std::unordered\_map<std::string, std::string> huffmanCode, std::unordered\_map<std::string, double> alphabet)

{

double averageLength = 0.0;

for (auto elem : alphabet)

averageLength += elem.second \* huffmanCode[elem.first].length();

return averageLength;

}

// Получение избыточности

double GetRedundancy(double entrophy, double averageLenth)

{

return averageLenth - entrophy;

}

// Преобразование кодовых слов (т.к. длина кодового слова нечетна, то Y = [y1, y2, y3, y1 XOR y2 XOR y3])

void BuildNewCodes(std::unordered\_map<std::string, std::string>& huffmanCode)

{

for(auto elem: huffmanCode)

{

auto codeWord = elem.second;

codeWord += elem.second[0] ^ elem.second[1] ^ elem.second[2];

huffmanCode[elem.first] = codeWord;

}

}

// Декодирование последовательности с проверкой на четность, запись только прошедших

// проверку кодов в переменную и сохранение номеров позиций с ошибочными битами

void NewDecodingCodes(std::string encodingText, std::string& decodingText,

std::vector<int>& positionNumbers, std::unordered\_map<std::string, std::string> huffmanCode)

{

for(int i = 0; i < encodingText.length(); i += 4)

{

auto value = GetValueExistence(huffmanCode, encodingText.substr(i, 4));

if((encodingText[i] ^ encodingText[i + 1] ^ encodingText[i + 2]) == encodingText[i + 3]

&& value != "")

{

decodingText += value;

}

else

{

positionNumbers.push\_back(i + 4);

}

}

}

// Получить ключ карты по значению

std::string GetValueExistence(std::unordered\_map<std::string, std::string> huffmanCode, std::string value)

{

for(auto elem: huffmanCode)

{

if (elem.second == value)

return elem.first;

}

return "";

}

// Получение d0

int GetCodeDistance(std::unordered\_map<std::string, std::string> huffmanCode)

{

std::vector<std::string> codeWords;

for (auto elem : huffmanCode)

codeWords.push\_back(elem.second);

auto d0 = 4;

auto counter = 0;

for(int i = 0; i < codeWords.size() - 1; i++)

{

for (int j = 0; j < 4; j++)

{

if ((codeWords[i][j] ^ codeWords[i + 1][j]) == 1)

counter++;

}

if (counter < d0)

d0 = counter;

counter = 0;

}

return d0;

}

// Число сочетаний

int Combinations(int n, int k)

{

if (k == 0 || k == n)

return 1;

return Combinations(n - 1, k - 1) \* n / k;

}

// Получение границы Хемминга

std::string GetHemmingBoundary(int n, int k)

{

auto r = n - k;

int amount = 0;

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

{

amount += Combinations(n, i);

}

std::string result = "r = ";

result += std::to\_string(r);

result += " >= ";

result += std::to\_string(log2(amount));

return result;

}

// Граница Плоткина

std::string GetPlotkinBoundary(int n, int k, int d0)

{

std::string result = "d0 = ";

result += std::to\_string(d0);

result += " <= ";

result += std::to\_string(n \* pow(2, k - 1) / (pow(2, k) - 1));

return result;

}

// Граница Варшамова–Гильберта

std::string GetVarshamovGilbertaBoundary(int n, int k, int d0)

{

auto r = n - k;

int amount = 0;

for (int i = 0; i < d0 - 1; i++)

{

amount += Combinations(n - 1, i);

}

std::string result = "2^r = ";

result += std::to\_string((int)pow(2, r));

result += " > ";

result += std::to\_string(amount);

return result;

}

// Записать текст в файл

void WriteTextToFile(std::string fileName, std::string text)

{

    std::ofstream file(fileName);

    file << text;

    file.close();

}

Заголовочный файл Hemming.h:

#pragma once

#ifndef \_Hemming\_H

#define \_Hemming\_H

#include <iostream>

#include <fstream>

#include <vector>

#include <string>

#include <map>

#include <msclr\marshal\_cppstd.h>

#include <unordered\_map>

class Hemming

{

private:

// Исходная порождающая матрица

std::vector<std::string> G

{

"10001011",

"01001100",

"00100110",

"00010011"

};

// Полученная из G проверочная матрица

std::vector<std::string> H

{

"1011",

"1100",

"0110",

"0011",

"1000",

"0100",

"0010",

"0001"

};

// Исходные символы алфавита

std::map<unsigned int, std::string> alphabet

{

{0, "0000"}, {1, "0001"}, {2, "0010"}, {3, "0011"},

{4, "0100"}, {5, "0101"}, {6, "0110"}, {7, "0111"},

{8, "1000"}, {9, "1001"}, {10, "1010"}, {11, "1011"},

{12, "1100"}, {13, "1101"}, {14, "1110"}, {15, "1111"}

};

// Алфавит кодовых слов

std::map<unsigned int, std::string> alphabetEncoded;

// Число сочетаний

int Combinations(int n, int k);

public:

// Получение алфавита

std::map<unsigned int, std::string> GetAlphabet();

// Получение закодированного алфавита

std::map<unsigned int, std::string> GetAlphabetEncoded();

// Получение d0

int GetCodeDistance(std::map<unsigned int, std::string> alphabetEncode);

// Получение границы Хэмминга

std::string GetHemmingBoundary(int n, int k);

// Формирование кодов хэмминга

void GenerateHammingCodes();

// Кодирование последовательности

std::string GetEncodedText(std::string sourceText);

// Декодирование последовательности с сохранением ошибочных кодов,

// преобразованных верных кодов и позиции в коде, где появилась ошибка

std::string GetDecodedText(std::string encodedText, std::vector<std::string>& codeWord,

std::vector<std::string>& newCodeWord, std::vector<int>& position);

};

#endif

Файл Hemming.cpp:

#include "Hemming.h"

// Получение алфавита

std::map<unsigned int, std::string> Hemming::GetAlphabet()

{

return alphabet;

}

// Получение закодированного алфавита

std::map<unsigned int, std::string> Hemming::GetAlphabetEncoded()

{

return alphabetEncoded;

}

// Формирование кодов хэмминга

void Hemming::GenerateHammingCodes()

{

std::string hammingCode;

for (auto elem : alphabet)

{

hammingCode = "00000000";

for (int i = 0; i < elem.second.size(); i++)

{

if (elem.second[i] == '1')

{

std::string help = "";

for (int j = 0; j < 8; j++)

{

help += std::to\_string(hammingCode[j] ^ G[i][j]);

}

hammingCode = help;

}

}

alphabetEncoded[elem.first] = hammingCode;

}

}

// Получение d0

int Hemming::GetCodeDistance(std::map<unsigned int, std::string> alphabetEncode)

{

std::vector<std::string> codeWords;

for (auto elem : alphabetEncode)

codeWords.push\_back(elem.second);

auto d0 = 4;

auto counter = 0;

for (int i = 0; i < codeWords.size() - 1; i++)

{

for (int j = 0; j < 8; j++)

{

if ((codeWords[i][j] ^ codeWords[i + 1][j]) == 1)

counter++;

}

if (counter < d0)

d0 = counter;

counter = 0;

}

return d0;

}

// Число сочетаний

int Hemming::Combinations(int n, int k)

{

if (k == 0 || k == n)

return 1;

return Combinations(n - 1, k - 1) \* n / k;

}

// Получение границы Хэмминга

std::string Hemming::GetHemmingBoundary(int n, int k)

{

auto r = n - k;

size\_t amount = 0;

for (int i = 0; i < 2; i++)

{

amount += Combinations(n, i);

}

std::string result = "r = ";

result += std::to\_string(r);

result += " >= ";

result += std::to\_string(log2(amount));

return result;

}

// Кодирование последовательности

std::string Hemming::GetEncodedText(std::string sourceText)

{

std::string encodedText = "";

for(int i = 0; i < sourceText.length(); i += 4)

{

auto code = sourceText.substr(i, 4);

auto number = (code[0] - '0') \* 8;

number += (code[1] - '0') \* 4;

number += (code[2] - '0') \* 2;

number += (code[3] - '0');

encodedText += alphabetEncoded[number];

}

return encodedText;

}

// Декодирование последовательности с сохранением ошибочных кодов,

// преобразованных верных кодов и позиции в коде, где появилась ошибка

std::string Hemming::GetDecodedText(std::string encodedText, std::vector<std::string>& codeWord,

std::vector<std::string>& newCodeWord, std::vector<int>& position)

{

std::string syndrome = "0000";

std::string result = "";

for(int i = 0; i < encodedText.length(); i += 8)

{

for (int k = i; k < i + 8; k++)

{

if (encodedText[k] == '1')

{

std::string help = "";

for (int j = 0; j < 4; j++)

{

help += std::to\_string(syndrome[j] ^ H[k - i][j]);

}

syndrome = help;

}

}

if (syndrome != "0000")

{

codeWord.push\_back(encodedText.substr(i, 8));

position.push\_back(std::find(H.begin(), H.end(), syndrome) - H.begin());

auto lastCode = codeWord[codeWord.size() - 1];

auto lastPosition = position[position.size() - 1];

auto code = lastCode;

code[lastPosition] = lastCode[lastPosition] == '0' ? '1' : '0';

newCodeWord.push\_back(code);

for (int j = 0; j < alphabetEncoded.size(); j++)

{

if (alphabetEncoded[j] == newCodeWord[newCodeWord.size() - 1])

{

result += alphabet[j];

break;

}

}

}

else

{

auto codeWord = encodedText.substr(i, 8);

for (int j = 0; j < alphabetEncoded.size(); j++)

{

if (alphabetEncoded[j] == codeWord)

{

result += alphabet[j];

break;

}

}

}

syndrome = "0000";

}

return result;

}

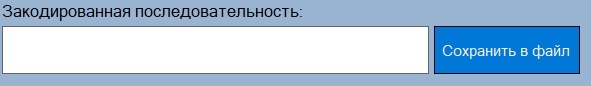
1. **Тесты**

Часть 1

Простейший тест был продемонстрирован ранее в разделе “описание разработанного программного средства”. Рассмотрим ситуации с ошибочным вводом и искажением кодов.

1. В случае, если файл с исходной последовательностью пуст:

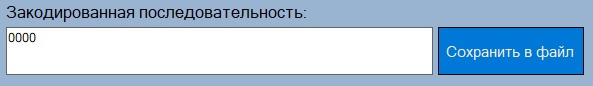


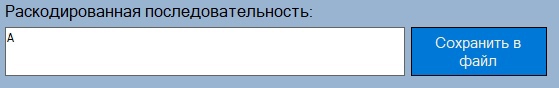


Декодированная последовательность, как и ожидалось, тоже будет пустой.

1. Попробуем ввести символ, не принадлежащий алфавиту:

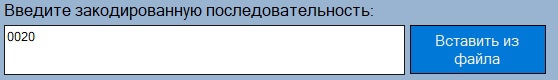




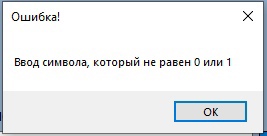


Как мы видим, программа игнорирует “неизвестный” символ и кодирует только известные ей символы.

1. В окне декодирования вводим символ, который не равен 0 или 1:

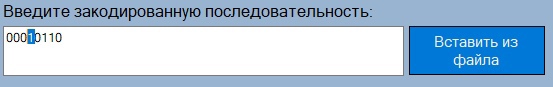


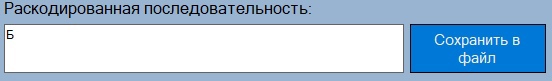
Если попробовать декодировать данную строку, получим:

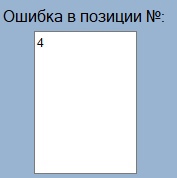


И окно декодирования, как и ожидалось, пустое.

1. Сделаем ошибку в одном слове при декодировании, намеренно заменив проверяющий бит:

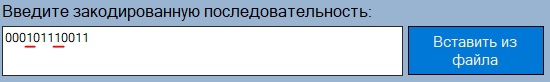


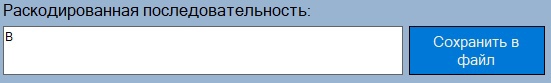


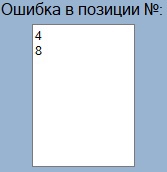


Изначально была последовательность 00000110, которая при декодировании давала АБ. В результате замены одного бита, не был записал символ A, в коде которого была обнаружена ошибка. Позиция ошибки указана в отдельном окне.

1. Сделаем ошибку в двух различных словах при декодировании:

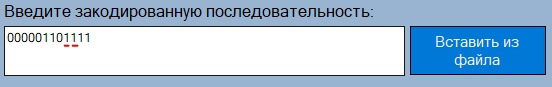


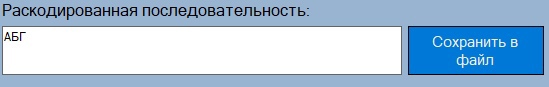




Изначально была последовательность 000001100011, которая при декодировании давала АБВ. В результате замены нескольких битов, не были записаны символы A и Б, в коде которых была обнаружена ошибка. Позиция ошибки указана в отдельном окне.

1. Следует отметить, что возможна ситуация, когда результат совпадет и ошибка не выявится. А также может быть ситуация, когда исказится информационный бит, ошибка выявится, но исправится неверно:





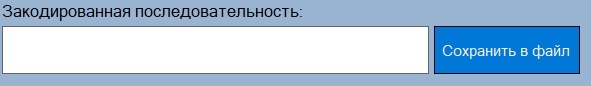


Искажение происходит в последнем кодовом слове. Изначальная кодовая последовательность 000001100011, которая при декодировании давала АБВ, стала давать АБГ. Ошибок обнаружено не было.

Часть 2

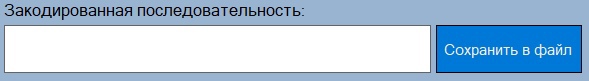
1. В случае, если файл с исходной последовательностью пуст:

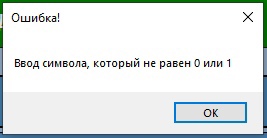




Декодированная последовательность, как и ожидалось, тоже будет пустой.

1. Попробуем ввести символ, не принадлежащий алфавиту:

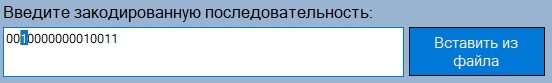
 

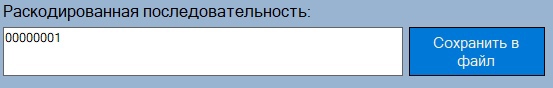


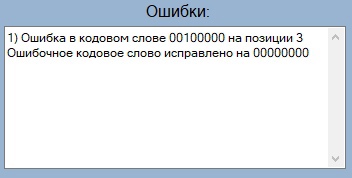
Как мы видим, программа не кодирует последовательность, если в ней присутствуют символы, не принадлежащие алфавиту. Также на экран выводится модальное окно с описанием ошибки.

Такая же ситуация возникнет при декодировании, если в нем будут символы, не принадлежащие алфавиту.

1. Сделаем ошибку в одном кодовом слове при декодировании:

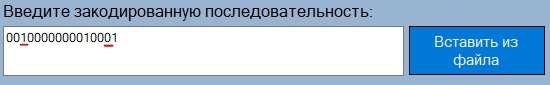


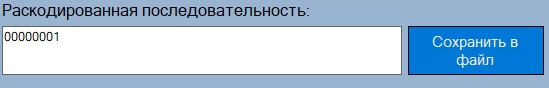


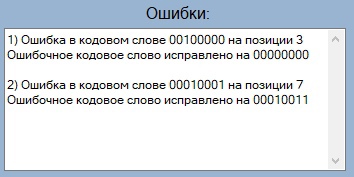


Входная последовательность была следующей: 00000001. Мы намеренно поменяли 3-й бит первого кодового слова и попробовали декодировать. Получили верный ответ, а также номер позиции в кодовом слове, где произошла ошибка.

1. Сделаем ошибку в двух различных словах при декодировании:

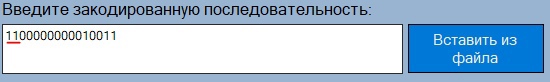


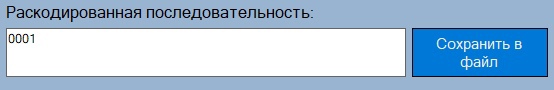


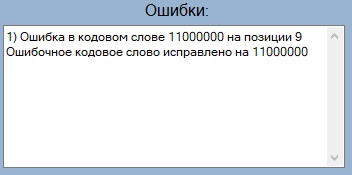


Как и в прошлом тесте, входная последовательность была 00000001. В результате совершения ошибок в двух различных кодовых словах, получили верный ответ и номера позиций, где были совершены ошибки.

1. Стоит отметить, что алгоритм может исправить одиночную ошибку. Если мы сделаем две ошибки в одном кодовом слове, то программа будет стараться исправить один символ в кодовом слове так, чтобы получилось определить символ алфавита:







В результате мы видим неверный результат как раскодированной последовательности, так и номера позиции в кодовом слове.

1. **Вывод**

В ходе проведения лабораторной работы мы освоили основные алгоритмы помехоустойчивого кодирования такие, как коды с обнаружением ошибок, основанные на проверки четности/нечетности длины кодового слова и коды с исправлением ошибок.