МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний аерокосмічний університет ім. М. Є. Жуковського

«Харківський авіаційний інститут»

Кафедра комп’ютерних систем, мереж та кібербезпеки

**Лабораторна робота №5**

СТИСНЕННЯ ТЕКСТОВОЇ ІНФОРМАЦІЇ

«Теорія інформації та кодування»

(назва дисципліни)

ХАІ.503.525СТ2.20О

Виконали: студенти 2 курсу групи № 000

напряму підготовки (спеціальності)

123-«Комп’ютерні системи та мережі»

(шифр і назва напряму підготовки (спеціальності))

студент

(прізвище й ініціали студента)

студент

(прізвище й ініціали студента)

студент

(прізвище й ініціали студента)

студент

(прізвище й ініціали студента)

Прийняв: к.т.н., доцент\_

\_\_\_\_\_\_\_\_\_\_\_\_Колісник М.О.\_\_\_

(посада, науковий ступінь, прізвище й ініціали)

Дата сдачи:

Національна шкала: \_\_\_\_\_\_\_\_\_\_

Кількість балів: \_\_\_\_\_

Оцінка: ECTS \_\_\_\_\_

Харків – 2020

**Цель работы:** Освоить методику сжатия текстовой информации с использованием метода Хаффмана и арифметического кодирования

**Задание:**

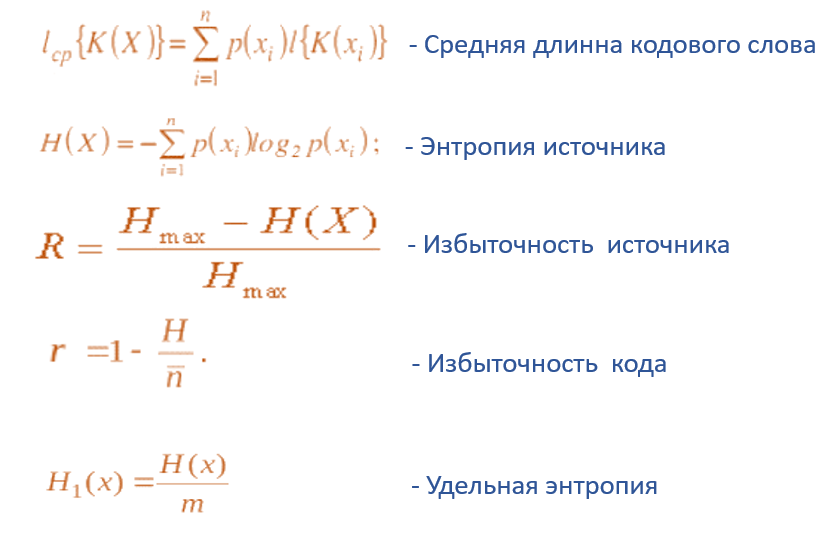
Задание 1.

* Задать произвольным образом вероятности появления символов.
* Заполнить все таблицы.
* Построить на одном графике зависимости Hmax(S) = f(ℓ ) и H1max (S)= f(ℓ ), где ℓ - количество объединяемых в блок исходных символов х1 и х2, (ℓ = 1, 2, 3, 4). На основании данных таблицы 5 построить следующие зависимости: = f(ℓ ) и nc = f(ℓ ) на одном графике; χ и = f(ℓ ) и χк = f(ℓ ) на одном графике;

Задание 2:

Закодировать арифметическим кодом текст на русском языке. Размер текста определен в соответствии с вариантом. Использовать усредненные частоты появления символов.

**Математична постановка задачи:**



**Ход работы**

Команда №3

**Листинг программы 1:**

**Form1.cs:**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace HammingCode

{

public partial class Form1 : Form

{

static ConvertText ct;

public Form1()

{

InitializeComponent();

ct = new ConvertText();

}

private void Button1\_Click(object sender, EventArgs e)

{

Form\_Table\_2 form2 = new Form\_Table\_2(textBox1.Text);

form2.ShowDialog();

}

private void Button2\_Click(object sender, EventArgs e)

{

Form\_table\_3 form3 = new Form\_table\_3();

form3.ShowDialog();

}

private void Button3\_Click(object sender, EventArgs e)

{

Form\_Table\_4 form4 = new Form\_Table\_4();

form4.ShowDialog();

}

private void Button4\_Click(object sender, EventArgs e)

{

Form\_Table\_5 form5 = new Form\_Table\_5();

form5.ShowDialog();

}

private void Button5\_Click(object sender, EventArgs e)

{

Form\_Table\_6 form6 = new Form\_Table\_6();

form6.ShowDialog();

}

}

}

**Form\_Table\_2.cs:**

using System;

using System.Collections;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace HammingCode

{

public partial class Form\_Table\_2 : Form

{

public Form\_Table\_2(string text)

{

InitializeComponent();

SetDGV(text);

}

public void SetDGV(string text)

{

if (text.Length == 0) return;

HuffmanTree hf = new HuffmanTree();

Dictionary<char, double> tmpDict = hf.BuildNodes(text);

ConvertText.table\_2\_Prob = tmpDict;

ConvertText.table\_2\_Prob = ConvertText.NormalizeIndex(ConvertText.table\_2\_Prob);

Dictionary<char, BitArray> codes = hf.BuildHuffmanTree();

dgv.RowCount = codes.Keys.Count;

for(int i =0; i < codes.Keys.Count; i++)

{

dgv[0, i].Value = codes.Keys.ToArray()[i];

BitArray bt = codes.Values.ToArray()[i];

foreach(object o in bt)

{

if((bool)o == true)

{

dgv[3, i].Value += '1'.ToString();

}

else

{

dgv[3, i].Value += '0'.ToString();

}

}

dgv[1, i].Value = "x" + i.ToString();

dgv[2, i].Value = hf.Frequencies[codes.Keys.ToArray()[i]];

dgv[4, i].Value = dgv[3, i].Value.ToString().Length;

}

Dictionary<char, BitArray> newDict = new Dictionary<char, BitArray>();

newDict.Clear();

for (int i = 0; i < codes.Count; i++)

{

newDict.Add(char.Parse(i.ToString()), codes.Values.ToArray()[i]);

}

ConvertText.table\_2\_Code = newDict;

}

}

}

**Form\_Table\_3.cs:**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.Collections;

namespace HammingCode

{

public partial class Form\_table\_3 : Form

{

public Form\_table\_3()

{

InitializeComponent();

SetDGV();

}

public void SetDGV()

{

Dictionary<char, double> newDict = ConvertText.GetDictionaryForTable3(3);

HuffmanTree hf = new HuffmanTree();

Dictionary<char, double> tmpDict = hf.BuildNodes(ConvertText.table\_3\_Prob);

Dictionary<char, BitArray> codes = hf.BuildHuffmanTree();

ConvertText.table\_3\_Code = codes;

dgv.RowCount = codes.Keys.Count;

for (int i = 0; i < codes.Keys.Count; i++)

{

BitArray bt = codes.Values.ToArray()[i];

foreach (object o in bt)

{

if ((bool)o == true)

{

dgv[3, i].Value += '1'.ToString();

}

else

{

dgv[3, i].Value += '0'.ToString();

}

}

dgv[1, i].Value = "x" + i.ToString();

dgv[2, i].Value = hf.Frequencies[codes.Keys.ToArray()[i]];

dgv[4, i].Value = dgv[3, i].Value.ToString().Length;

}

dgv[0, 0].Value = "x1\*x1";

dgv[0, 1].Value = "x1\*x2";

dgv[0, 2].Value = "x2\*x1";

dgv[0, 3].Value = "x2\*x2";

}

}

}

**Form\_Table\_4.cs:**

using System;

using System.Collections;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace HammingCode

{

public partial class Form\_Table\_4 : Form

{

public Form\_Table\_4()

{

InitializeComponent();

SetDGV();

}

public void SetDGV()

{

Dictionary<char, double> newDict = ConvertText.GetDictionaryForTable3(4);

HuffmanTree hf = new HuffmanTree();

Dictionary<char, double> tmpDict = hf.BuildNodes(ConvertText.table\_4\_Prob);

Dictionary<char, BitArray> codes = hf.BuildHuffmanTree();

ConvertText.table\_4\_Code = codes;

dgv.RowCount = codes.Keys.Count;

List<Node> orderedNodes = hf.nodes.OrderBy(node => node.AddName).ToList<Node>();

int i = 0;

foreach (var o in hf.Frequencies)

{

dgv[1, i].Value = "x" + i.ToString();

dgv[2, i].Value = o.Value.ToString();

foreach (var q in codes[o.Key])

{

if ((bool)q == true)

{

dgv[3, i].Value += '1'.ToString();

}

else

{

dgv[3, i].Value += '0'.ToString();

}

}

dgv[4, i].Value = dgv[3, i].Value.ToString().Length;

i++;

}

dgv[0, 0].Value = "x1\*x1\*x1";

dgv[0, 1].Value = "x1\*x1\*x2";

dgv[0, 2].Value = "x1\*x2\*x1";

dgv[0, 3].Value = "x1\*x2\*x2";

dgv[0, 4].Value = "x2\*x1\*x1";

dgv[0, 5].Value = "x2\*x1\*x2";

dgv[0, 6].Value = "x2\*x2\*x1";

dgv[0, 7].Value = "x2\*x2\*x2";

}

}

}

**Form\_Table\_5.cs:**

using System;

using System.Collections;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace HammingCode

{

public partial class Form\_Table\_5 : Form

{

public Form\_Table\_5()

{

InitializeComponent();

SetDGV();

}

public void SetDGV()

{

Dictionary<char, double> newDict = ConvertText.GetDictionaryForTable3(5);

HuffmanTree hf = new HuffmanTree();

Dictionary<char, double> tmpDict = hf.BuildNodes(ConvertText.table\_5\_Prob);

Dictionary<char, BitArray> codes = hf.BuildHuffmanTree();

ConvertText.table\_5\_Code = codes;

dgv.RowCount = codes.Keys.Count;

List<Node> orderedNodes = hf.nodes.OrderBy(node => node.AddName).ToList<Node>();

int i = 0;

foreach (var o in hf.Frequencies)

{

dgv[1, i].Value = "x" + i.ToString();

dgv[2, i].Value = o.Value.ToString();

foreach (var q in codes[o.Key])

{

if ((bool)q == true)

{

dgv[3, i].Value += '1'.ToString();

}

else

{

dgv[3, i].Value += '0'.ToString();

}

}

dgv[4, i].Value = dgv[3, i].Value.ToString().Length;

i++;

}

dgv[0, 0].Value = "x1\*x1\*x1\*x1";

dgv[0, 1].Value = "x1\*x1\*x1\*x2";

dgv[0, 2].Value = "x1\*x1\*x2\*x1";

dgv[0, 3].Value = "x1\*x1\*x2\*x2";

dgv[0, 4].Value = "x1\*x2\*x1\*x1";

dgv[0, 5].Value = "x1\*x2\*x1\*x2";

dgv[0, 6].Value = "x1\*x2\*x2\*x1";

dgv[0, 7].Value = "x1\*x2\*x2\*x2";

dgv[0, 8].Value = "x2\*x1\*x1\*x1";

dgv[0, 9].Value = "x2\*x1\*x1\*x2";

dgv[0, 10].Value = "x2\*x1\*x2\*x1";

dgv[0, 11].Value = "x2\*x1\*x2\*x2";

dgv[0, 12].Value = "x2\*x2\*x1\*x1";

dgv[0, 13].Value = "x2\*x2\*x1\*x2";

dgv[0, 14].Value = "x2\*x2\*x2\*x1";

dgv[0, 15].Value = "x2\*x2\*x2\*x2";

}

}

}

**Form\_Table\_6.cs:**

using System;

using System.Collections;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace HammingCode

{

public partial class Form\_Table\_6 : Form

{

const int COUNT\_GROUPS = 4;

public Form\_Table\_6()

{

InitializeComponent();

SetDGV();

}

public void SetDGV()

{

dgv.RowCount = COUNT\_GROUPS;

SetGenerallyParams();

SetParamsRow\_1();

SetParamsRow\_2();

SetParamsRow\_3();

SetParamsRow\_4();

}

void SetGenerallyParams()

{

List<double[]> listProbab = new List<double[]> { ConvertText.table\_2\_Prob.Values.ToArray(),

ConvertText.table\_3\_Prob.Values.ToArray(),

ConvertText.table\_4\_Prob.Values.ToArray(),

ConvertText.table\_5\_Prob.Values.ToArray() };

dgv[0, 0].Value = (Math.Log(2, 2)).ToString();

dgv[0, 1].Value = (Math.Log(4, 2)).ToString();

dgv[0, 2].Value = (Math.Log(8, 2)).ToString();

dgv[0, 3].Value = (Math.Log(16, 2)).ToString();

for (int i = 0; i< COUNT\_GROUPS; i++)

{

dgv[1, i].Value = (CalcEntropy(listProbab[i])).ToString();

}

}

void SetParamsRow\_1()

{

dgv[5, 0].Value = CalcAvarageLengthPerSymb(ConvertText.table\_2\_Prob, ConvertText.table\_2\_Code);

dgv[4, 0].Value = CalcAvarageLenght(ConvertText.table\_2\_Code);

dgv[3, 0].Value = 1 - (double.Parse(dgv[1, 0].Value.ToString()) / Math.Log(2, 2));

double tmp = double.Parse(dgv[5, 0].Value.ToString());

dgv[7, 0].Value = (tmp - double.Parse(dgv[1, 0].Value.ToString())) /tmp;

dgv[2, 0].Value = ConvertText.table\_2\_Code.Values.ToArray().Min(k => k.Count);

dgv[6, 0].Value = double.Parse(dgv[2, 0].Value.ToString()) / double.Parse(dgv[5, 0].Value.ToString());

}

void SetParamsRow\_2()

{

dgv[5, 1].Value = CalcAvarageLengthPerSymb(ConvertText.table\_3\_Prob, ConvertText.table\_3\_Code);

dgv[4, 1].Value = CalcAvarageLenght(ConvertText.table\_3\_Code);

dgv[3, 1].Value = 1 - (double.Parse(dgv[1, 1].Value.ToString()) / Math.Log(4, 2));

double tmp = double.Parse(dgv[5, 1].Value.ToString());

dgv[7, 1].Value = (tmp - double.Parse(dgv[1, 1].Value.ToString())) / tmp;

dgv[2, 1].Value = ConvertText.table\_3\_Code.Values.ToArray().Min(k => k.Count);

dgv[6, 1].Value = double.Parse(dgv[2, 1].Value.ToString()) / double.Parse(dgv[5, 1].Value.ToString());

}

void SetParamsRow\_3()

{

dgv[5, 2].Value = CalcAvarageLengthPerSymb(ConvertText.table\_4\_Prob, ConvertText.table\_4\_Code);

dgv[4, 2].Value = CalcAvarageLenght(ConvertText.table\_4\_Code);

dgv[3, 2].Value = 1 - (double.Parse(dgv[1, 2].Value.ToString()) / Math.Log(8, 2));

double tmp = double.Parse(dgv[5, 2].Value.ToString());

dgv[7, 2].Value = (tmp - double.Parse(dgv[1, 2].Value.ToString())) / tmp;

dgv[2, 2].Value = ConvertText.table\_4\_Code.Values.ToArray().Min(k => k.Count);

dgv[6, 2].Value = double.Parse(dgv[2, 2].Value.ToString()) / double.Parse(dgv[5, 2].Value.ToString());

}

void SetParamsRow\_4()

{

dgv[5, 3].Value = CalcAvarageLengthPerSymb(ConvertText.table\_5\_Prob, ConvertText.table\_5\_Code);

dgv[4, 3].Value = CalcAvarageLenght(ConvertText.table\_5\_Code);

dgv[3, 3].Value = 1 - (double.Parse(dgv[1, 3].Value.ToString()) / Math.Log(16, 2));

double tmp = double.Parse(dgv[5, 3].Value.ToString());

dgv[7, 3].Value = (tmp - double.Parse(dgv[1, 3].Value.ToString())) / tmp;

dgv[2, 3].Value = ConvertText.table\_5\_Code.Values.ToArray().Min(k => k.Count);

dgv[6, 3].Value = double.Parse(dgv[2, 3].Value.ToString()) / double.Parse(dgv[5, 3].Value.ToString());

}

double CalcEntropy(double[] probabilities)

{

double entropy = 0;

foreach(var a in probabilities)

{

entropy += a \* Math.Log(a, 2);

}

return (0 - entropy);

}

double CalcAvarageLengthPerSymb (Dictionary<char, double> probab, Dictionary<char, BitArray> codes)

{

//string tmp = "CODES dictionary\n";

//foreach(var o in codes)

//{

// tmp += "key:" + o.Key + "|value:" + o.Value + "\n";

//}

//MessageBox.Show(tmp);

double n\_c = 0;

foreach(var pair in probab)

{

//MessageBox.Show("pair.Value:" + pair.Value.ToString() + "|key:" + pair.Key + "\n");

n\_c += pair.Value \* codes[pair.Key].Count;

}

return n\_c;

}

double CalcAvarageLenght (Dictionary<char, BitArray> codes)

{

double n = 0;

foreach(var pair in codes)

{

n += pair.Value.Count;

}

n /= (double)codes.Count;

return n;

}

}

}

**ConvertText.cs:**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace HammingCode

{

class ConvertText

{

public static Dictionary<char, double> table\_2\_Prob = new Dictionary<char, double>(2);

public static Dictionary<char, double> table\_3\_Prob = new Dictionary<char, double>(4);

public static Dictionary<char, double> table\_4\_Prob = new Dictionary<char, double>(8);

public static Dictionary<char, double> table\_5\_Prob = new Dictionary<char, double>(16);

public static Dictionary<char, BitArray> table\_2\_Code = new Dictionary<char, BitArray>(2);

public static Dictionary<char, BitArray> table\_3\_Code = new Dictionary<char, BitArray>(4);

public static Dictionary<char, BitArray> table\_4\_Code = new Dictionary<char, BitArray>(8);

public static Dictionary<char, BitArray> table\_5\_Code = new Dictionary<char, BitArray>(16);

public Dictionary<char, double> GetProbabiltys(string text)

{

if (text.Length == 0) return null;

Dictionary<char, double> charsProbab = new Dictionary<char, double>();

foreach(char c in text)

{

if(!charsProbab.Keys.Contains(c))

{

charsProbab[c] = text.Count(el => el == c) / text.Length;

}

}

return charsProbab;

}

public static Dictionary<char, double> GetDictionaryForTable3(int numberOfForm)

{

int COUNT\_VARS = 0;

object numberTable = null;

int middleTable = 0;

switch (numberOfForm)

{

case 3:

{

numberTable = table\_2\_Prob;

COUNT\_VARS = 4;

middleTable = 2;

}

break;

case 4:

{

numberTable = table\_3\_Prob;

COUNT\_VARS = 8;

middleTable = 4;

} break;

case 5:

{

numberTable = table\_4\_Prob;

COUNT\_VARS = 16;

middleTable = 8;

} break;

}

Dictionary<char, double> newDict = new Dictionary<char, double>(COUNT\_VARS);

newDict.Clear();

for (int i = 0; i < COUNT\_VARS; i++)

{

if (i < COUNT\_VARS / 2)

{

newDict[char.Parse(Convert.ToString(i, 16))] = ((Dictionary<char, double>)numberTable)[char.Parse((i % middleTable).ToString())] \* table\_2\_Prob['0'];

}

else

{

newDict[char.Parse(Convert.ToString(i, 16))] = ((Dictionary<char, double>)numberTable)[char.Parse((i % middleTable).ToString())] \* table\_2\_Prob['1'];

}

}

switch (numberOfForm)

{

case 3: table\_3\_Prob = newDict; break;

case 4: table\_4\_Prob = newDict; break;

case 5: table\_5\_Prob = newDict; break;

}

return newDict;

}

public static Dictionary<char, double> NormalizeIndex(Dictionary<char, double> oldDict)

{

Dictionary<char, double> newDict = new Dictionary<char, double>();

newDict.Clear();

for(int i=0; i< oldDict.Count; i++)

{

newDict.Add(char.Parse(i.ToString()), oldDict.Values.ToArray()[i]);

}

string tm = "NormalizeIndex";

for (int i =0; i < oldDict.Count; i++)

{

tm += "Key:" + oldDict.Keys.ToArray()[i].ToString() + "|Value:" + oldDict.Values.ToArray()[i].ToString()+"\n";

}

MessageBox.Show(tm);

return newDict;

}

}

}

**HuffmanEncodeRealization.cs:**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace HammingCode

{

public class HuffmanTree

{

public int indexAddName = 0;

public List<Node> nodes = new List<Node>();

public Node Root { get; set; }

public Dictionary<char, double> Frequencies = new Dictionary<char, double>();

public Dictionary<char, double> BuildNodes(string text)

{

if (text.Length == 0) return null;

Frequencies.Clear();

foreach (char c in text)

{

if (!Frequencies.Keys.Contains(c))

{

Frequencies.Add(c, text.Count(el => el == c) / (double)text.Length);

}

}

nodes.Clear();

indexAddName = 0;

Frequencies = Frequencies.OrderBy(k => k.Value).ToDictionary(k => k.Key, k => k.Value);

string tmp = "Build Nodes\n";

foreach (KeyValuePair<char, double> c in Frequencies)

{

tmp += c.Key.ToString() + c.Value.ToString();

nodes.Add(new Node { Symbol = c.Key, Frequency = c.Value, AddName = "x"+indexAddName++.ToString() });

}

MessageBox.Show(tmp);

return Frequencies;

}

public Dictionary<char, double> BuildNodes(Dictionary<char, double> inputProbabilities)

{

Frequencies = inputProbabilities;

nodes.Clear();

indexAddName = 0;

foreach(KeyValuePair<char, double> c in Frequencies)

{

nodes.Add(new Node { Symbol = c.Key, Frequency = c.Value, AddName = "x"+indexAddName++.ToString()});

}

return Frequencies;

}

public Dictionary<char, BitArray> BuildHuffmanTree()

{

while (nodes.Count > 1)

{

List<Node> orderedNodes = nodes.OrderBy(node => node.Frequency).ThenByDescending(node => node.AddName).ToList<Node>();

//string tmp = "";

//for(int i =0; i< orderedNodes.Count;i++)

//{

// tmp += $"char:{orderedNodes[i].Symbol}|value:{orderedNodes[i].Frequency}|addName:{orderedNodes[i].AddName}\n";

//}

//MessageBox.Show(tmp);

if (orderedNodes.Count >= 2)

{

// Take first two items

List<Node> taken = orderedNodes.Take(2).ToList<Node>();

// Create a parent node by combining the frequencies

Node parent = new Node()

{

Symbol = '!',

Frequency = taken[0].Frequency + taken[1].Frequency,

Left = taken[0],

Right = taken[1],

AddName = "x" + indexAddName++.ToString()

};

//string tmp1 = $"{taken[0].Symbol} = ${taken[0].Frequency}|||{taken[1].Symbol} = ${taken[1].Frequency}";

//MessageBox.Show(tmp1);

nodes.Remove(taken[0]);

nodes.Remove(taken[1]);

nodes.Add(parent);

}

this.Root = nodes.FirstOrDefault();

}

Dictionary<char, BitArray> huffmanCodesTable = new Dictionary<char, BitArray>();

// Frequencies = Frequencies.OrderBy(i => i.Key).ToDictionary<KeyValuePair<char, double>>(k => k.Key);

string tmp = "Frequinces:\n";

foreach(var a in Frequencies)

{

tmp += "key:" + a.Key.ToString() + "|value:" + a.Value.ToString() + "\n";

}

foreach (char c in Frequencies.Keys)

{

huffmanCodesTable.Add(c, new BitArray(this.Root.Traverse(c, new List<bool>()).ToArray()));

}

return huffmanCodesTable;

}

public BitArray Encode(string source)

{

List<bool> encodedSource = new List<bool>();

for (int i = 0; i < source.Length; i++)

{

List<bool> encodedSymbol = this.Root.Traverse(source[i], new List<bool>());

encodedSource.AddRange(encodedSymbol);

}

BitArray bits = new BitArray(encodedSource.ToArray());

return bits;

}

public string Decode(BitArray bits)

{

Node current = this.Root;

string decoded = "";

foreach (bool bit in bits)

{

if (bit)

{

if (current.Right != null)

{

current = current.Right;

}

}

else

{

if (current.Left != null)

{

current = current.Left;

}

}

if (IsLeaf(current))

{

decoded += current.Symbol;

current = this.Root;

}

}

return decoded;

}

public bool IsLeaf(Node node)

{

return (node.Left == null && node.Right == null);

}

}

public class Node

{

public char Symbol { get; set; }

public double Frequency { get; set; }

public Node Right { get; set; }

public Node Left { get; set; }

public string AddName { get; set; }

public List<bool> Traverse(char symbol, List<bool> data)

{

// Leaf

if (Right == null && Left == null)

{

if (symbol.Equals(this.Symbol))

{

return data;

}

else

{

return null;

}

}

else

{

List<bool> left = null;

List<bool> right = null;

if (Left != null)

{

List<bool> leftPath = new List<bool>();

leftPath.AddRange(data);

leftPath.Add(false);

left = Left.Traverse(symbol, leftPath);

}

if (Right != null)

{

List<bool> rightPath = new List<bool>();

rightPath.AddRange(data);

rightPath.Add(true);

right = Right.Traverse(symbol, rightPath);

}

if (left != null)

{

return left;

}

else

{

return right;

}

}

}

}

}

**Program.cs:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace HammingCode

{

static class Program

{

/// <summary>

/// The main entry point for the application.

/// </summary>

[STAThread]

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Application.Run(new Form1());

}

}

}

**Листинг программы 2:**

**Program.cs:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace Lab\_5\_TIC\_ArithmeticCoding

{

static class Program

{

/// <summary>

/// The main entry point for the application.

/// </summary>

[STAThread]

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Application.Run(new Form1());

}

}

}

**Logic.cs:**

using System;

using System.Collections.Generic;

using System.Linq;

namespace Lab\_5\_TIC\_ArithmeticCoding

{

public class Node

{

public char Symbol { get; set; }

public decimal Low { get; set; }

public decimal High { get; set; }

public decimal Range { get; set; }

public int Frequency { get; set; }

}

public class ArifmCode

{

private List<Node> nodes = new List<Node>();

public Dictionary<char, int> Frequencies = new Dictionary<char, int>();

public void Build(string source)

{

for (int i = 0; i < source.Length; i++)

{

if (!Frequencies.ContainsKey(source[i]))

{

Frequencies.Add(source[i], 0);

}

Frequencies[source[i]]++;

}

foreach (KeyValuePair<char, int> symbol in Frequencies)

{

nodes.Add(new Node() { Symbol = symbol.Key, Frequency = symbol.Value });

}

nodes = nodes.OrderBy(node => node.Frequency).ToList<Node>();

}

public List<Node> GetSymbolsRanges(string source)

{

decimal low = 0.0m;

foreach (Node node in nodes)

{

node.Range = ((decimal)(node.Frequency) / (source.Length));

node.Low = low;

node.High = low + node.Range;

low += node.Range;

}

return nodes;

}

public decimal Encode(string source, out List<Node> resNodes)

{

nodes.Reverse();

List<Node> allNodes = new List<Node>();

for (int i = 0; i < source.Length; i++)

{

for (int j = 0; j < nodes.Count; j++)

{

if (source[i] == nodes[j].Symbol)

{

allNodes.Add(new Node() { Symbol = nodes[j].Symbol, Low = nodes[j].Low, High = nodes[j].High });

}

}

}

for (int i = 1; i < allNodes.Count; i++)

{

for (int j = 0; j < nodes.Count; j++)

{

if (allNodes[i].Symbol == nodes[j].Symbol)

{

allNodes[i].High = allNodes[i - 1].Low + (allNodes[i - 1].High - allNodes[i - 1].Low) \* nodes[j].High;

allNodes[i].Low = allNodes[i - 1].Low + (allNodes[i - 1].High - allNodes[i - 1].Low) \* nodes[j].Low;

}

}

}

resNodes = allNodes;

return Math.Round((allNodes[allNodes.Count - 1].Low + ((allNodes[allNodes.Count - 1].High - allNodes[allNodes.Count - 1].Low)/2)), 8);

}

public string Decode(List<Node> allNodes, int count, decimal inCode)

{

string decode = "";

decimal code = inCode;

int symbolsCount = 0;

while (true)

{

for (int i = 0; i < nodes.Count; i++)

{

if (code >= nodes[i].Low && code < nodes[i].High)

{

decode += nodes[i].Symbol;

code = Math.Round(((code - nodes[i].Low) / (nodes[i].High - nodes[i].Low)), 8);

symbolsCount++;

if (symbolsCount == count)

return decode;

}

}

}

}

}

}

**Form1.cs:**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace Lab\_5\_TIC\_ArithmeticCoding

{

public partial class Form1 : Form

{

const int COUNT\_SUMBOLS\_PER\_SUBSTRING = 5;

public Form1()

{

InitializeComponent();

}

private void BtnEncode\_Click(object sender, EventArgs e)

{

ArifmCode arthCode = new ArifmCode();

List<Node> resNodes;

string[] sliceInTextPer\_5\_Symb;

txtBoxStrPart.Clear();

txtBoxFraction.Clear();

txtBoxBinary.Clear();

arthCode.Build(txtBoxInput.Text);

arthCode.GetSymbolsRanges(txtBoxInput.Text);

sliceInTextPer\_5\_Symb = splitStrOn\_N\_Symbols(txtBoxInput.Text, COUNT\_SUMBOLS\_PER\_SUBSTRING);

foreach (string str in sliceInTextPer\_5\_Symb)

{

txtBoxStrPart.Text += str + "\r\n";

decimal res = arthCode.Encode(str, out resNodes);

txtBoxFraction.Text += (res.ToString() + "\r\n");

txtBoxBinary.Text += Convert.ToString(GetFractionPart(res), 2) + "\r\n";

}

}

private string[] splitStrOn\_N\_Symbols(string inputString, int sizeSubstr)

{

string[] splitedStrings = new string[(int)Math.Ceiling((double)inputString.Length / sizeSubstr)];

if(inputString.Length < sizeSubstr)

{

splitedStrings[0] = inputString;

return splitedStrings;

}

int i, j;

for (i = 0, j = 0; i < inputString.Length;i+=sizeSubstr, j++)

{

splitedStrings[j] = inputString.Substring(i, Math.Min(sizeSubstr, inputString.Length-i));

}

return splitedStrings;

}

private int GetFractionPart(decimal number)

{

string fractionPartStr = (number - Math.Floor(number)).ToString();

fractionPartStr = fractionPartStr.Split(',')[1];

return int.Parse(fractionPartStr);

}

}

}

**Результаты работы программы 1**

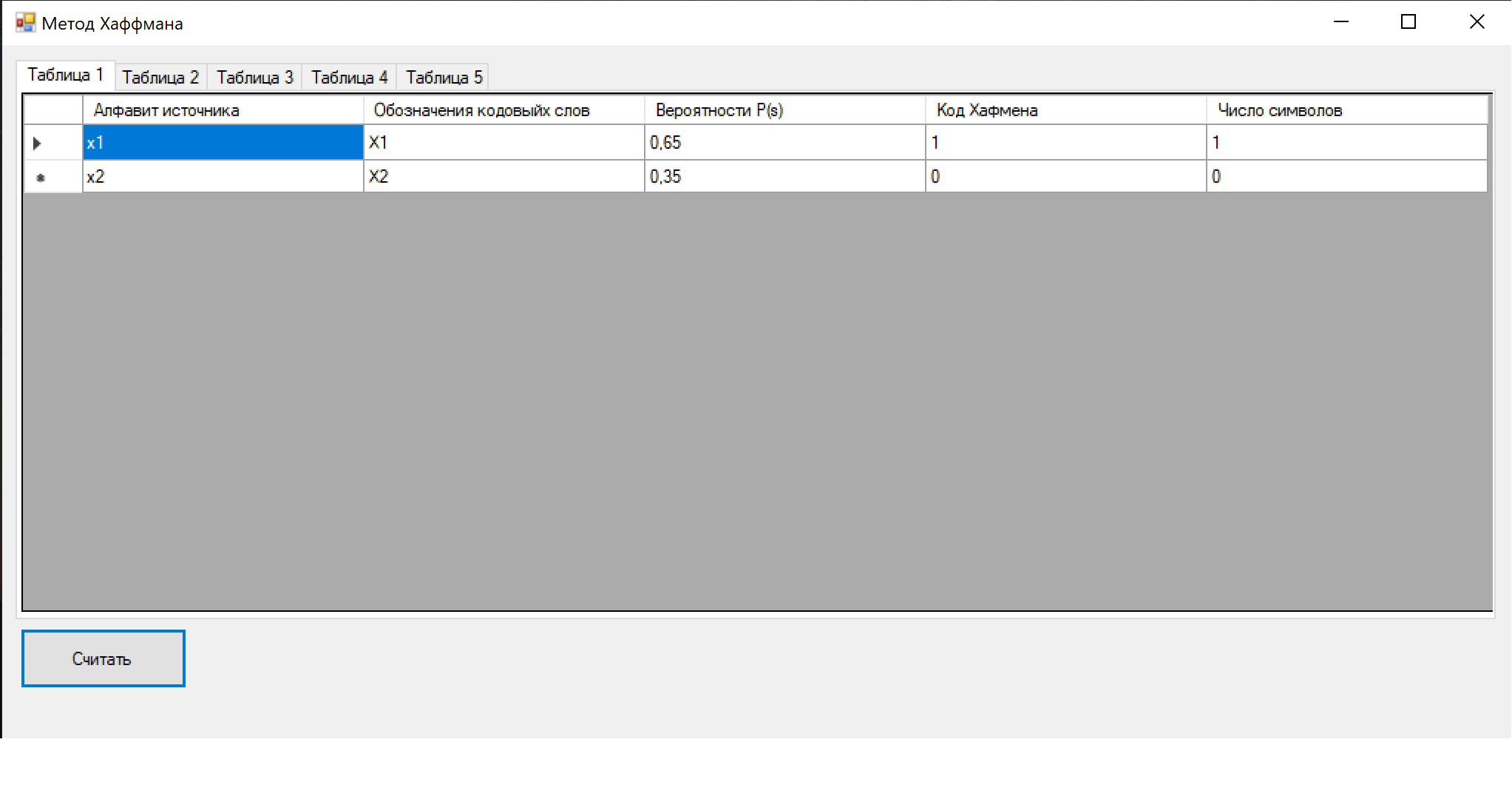
****

Рисунок 1 – Таблица 1

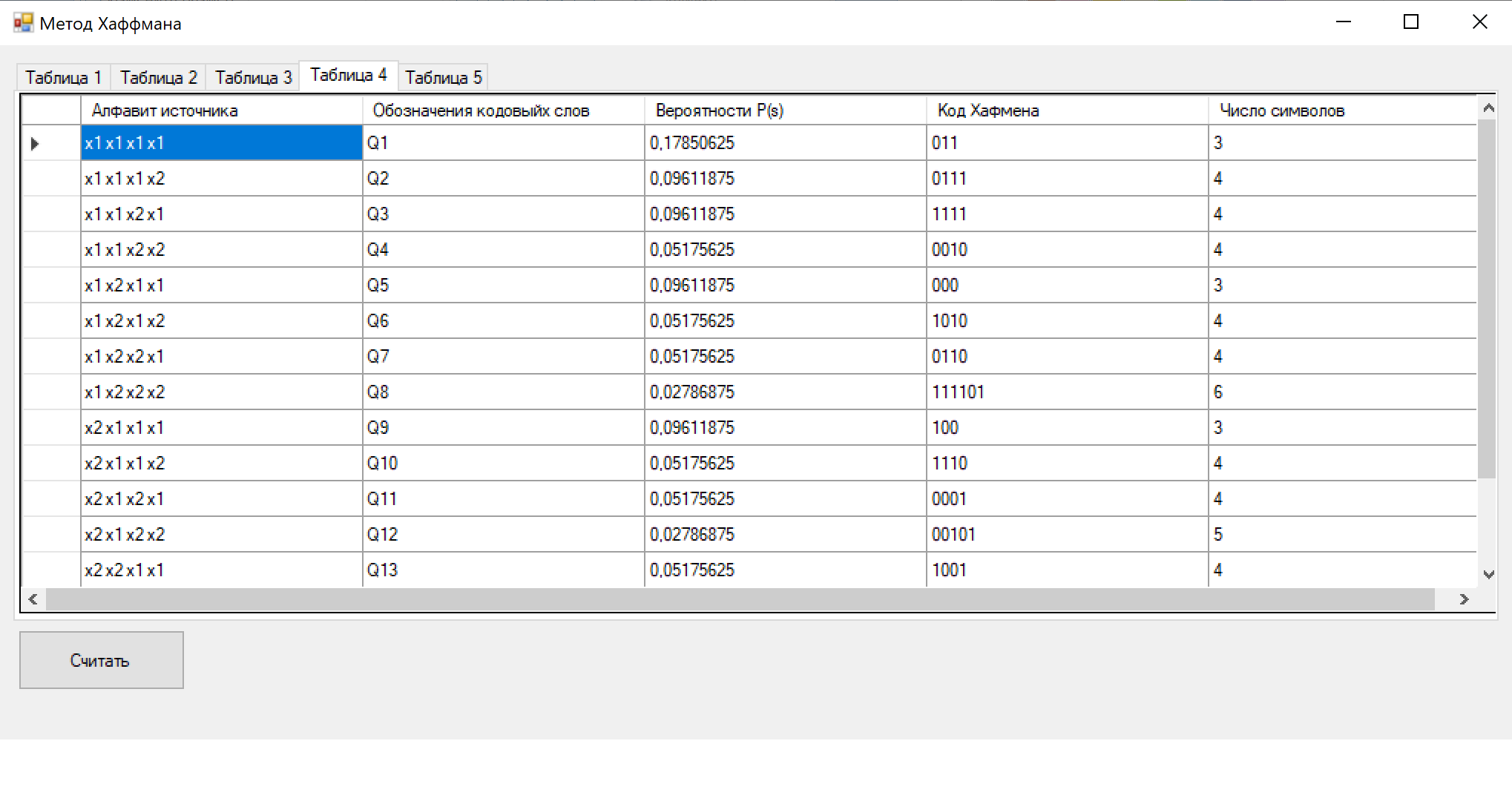


Рисунок 2 – Таблица 4

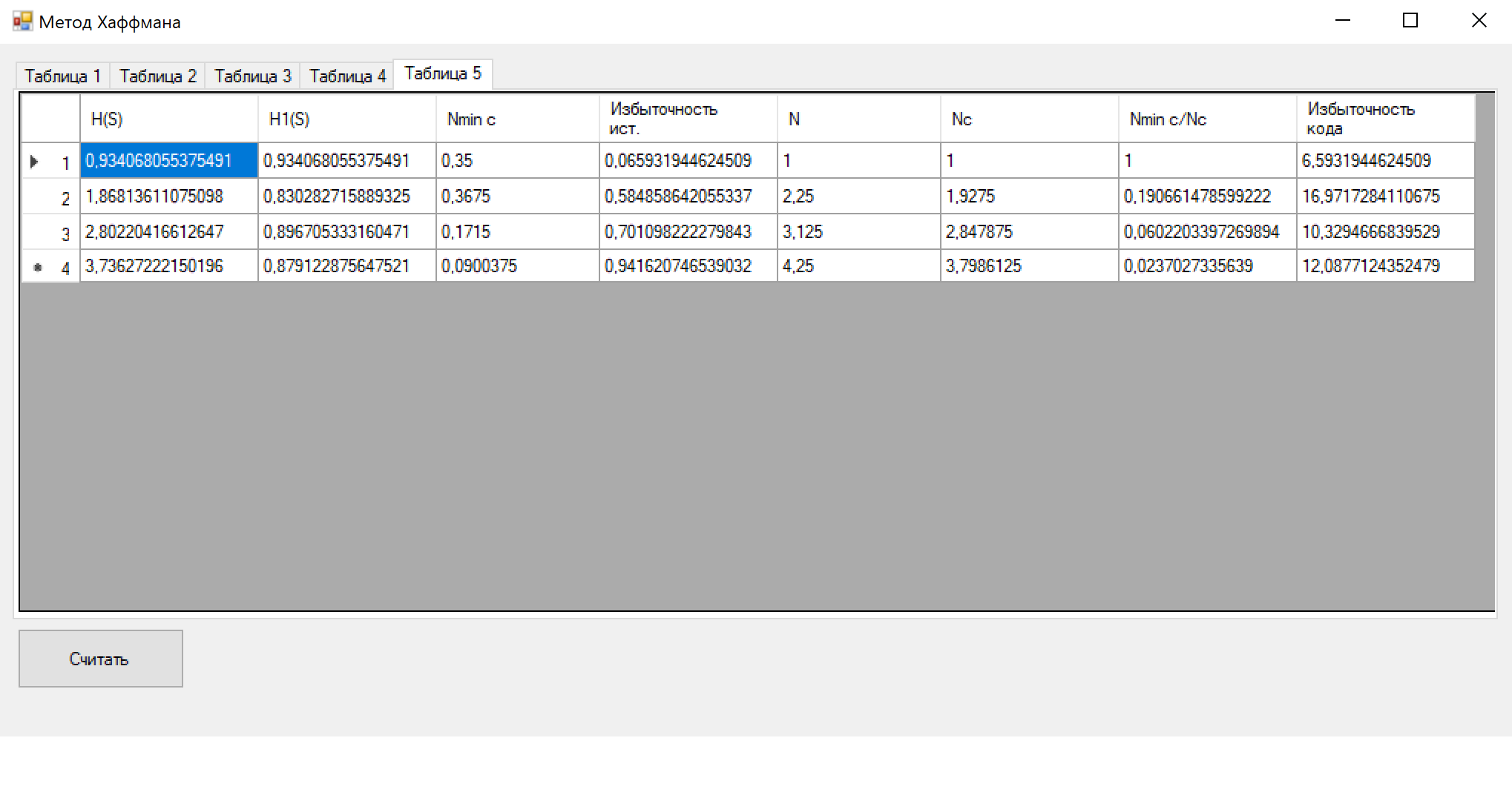


Рисунок 3 – Таблица 5

**Результаты работы программы 1**

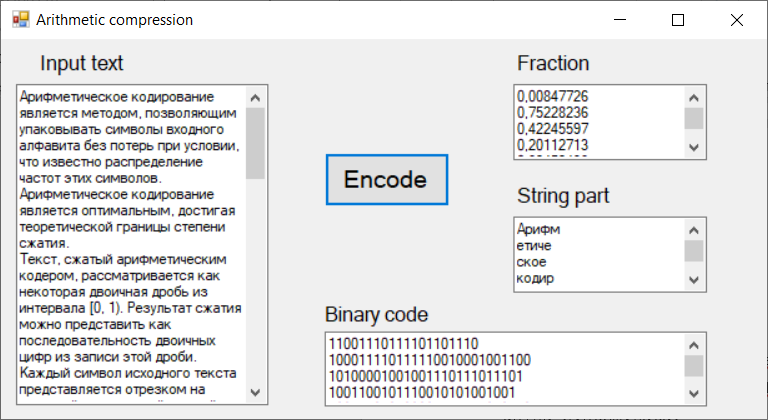


Рисунок 4 – Работа программы 2

**Висновки:**

В ходе выполнения лабораторной работы мы узнали что сжатие Хаффмана– статистический метод сжатия, который уменьшает среднюю длину кодового слова для символов алфавита. Код Хаффмана является примером кода, оптимального в случае, когда все вероятности появления символов в сообщении - целые отрицательные степени двойки. Для заданного распределения частот символов может существовать несколько возможных кодов Хаффмана. Возможно определить 'каноническое' дерево Хаффмана, выбрав одно из возможных деревьев. Может показаться что кодирование Хаффмана лучшее средство для сжатия. Однако это не так. Как было замечено выше, этот метод оптимален только в том случае, когда все символы в сообщении имеют вероятности появления равные целым отрицательным степеням двойки, что в общем случае не так. Метод арифметического кодирования не имеет этого ограничения: он достигает одинакового эффекта, т.к. рассматривает сообщение как единое целое (что для кодирования по Хаффману потребовало бы нумерации каждого из всех возможных сообщений), и таким образом достигает теоретической энтропийной границы эффективности сжатия для любого источника.