

Федеральное государственное бюджетное образовательное учреждение высшего
образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Кафедра прикладной математики и кибернетики

Лабораторная работа № 5
по дисциплине «Теория Информации»

Выполнил:
студент группы ИП-712
Алексеев Степан
Владимирович
ФИО студента

Работу проверил:
доцент кафедры ПМИК Мачикина Е.П.
ФИО преподавателя

Новосибирск 2021 г.

Оглавление

ЗАДАНИЕ	3
Решение	4
Анализ	4
Скриншоты	5
Листинг кода	6

ЗАДАНИЕ

Теория информации

Практическая работа №5

Недвоичное кодирование

Цель работы: Сравнение свойств двоичного и недвоичного кодирования.

Язык программирования: C, C++, C#, Python

Результат: программа, тестовые примеры, отчет.

1. Запрограммировать процедуру недвоичного кодирования текстового файла одним из методов (метод Хаффмана, метод Фано, метод Шеннона, метод Гилберта-Мура), размер кодового алфавита выбирается самостоятельно. Текстовые файлы использовать те же, что и в практических работах №1-3.
2. После кодирования текстового файла вычислить энтропию выходной последовательности, используя частоты отдельных символов, пар символов и тройки символов.
3. Заполнить таблицу и сравнить полученные результаты с результатами из практической работы 4.

Метод кодирования	Название текста	Энтропия выходной посл-ти (частоты пар символов)	Энтропия выходной посл-ти (частоты пар символов)	Энтропия выходной посл-ти (частоты троек символов)

4. Оформить отчет, загрузить отчет и файл с исходным кодом в электронную среду. Отчет обязательно должен содержать заполненную таблицу и анализ полученных результатов.

По желанию в отчет можно включить описание программной реализации.

В отчет не нужно включать содержимое этого файла.

Решение

Строю тернарное дерево Хаффмана(3 символа в алфавите).

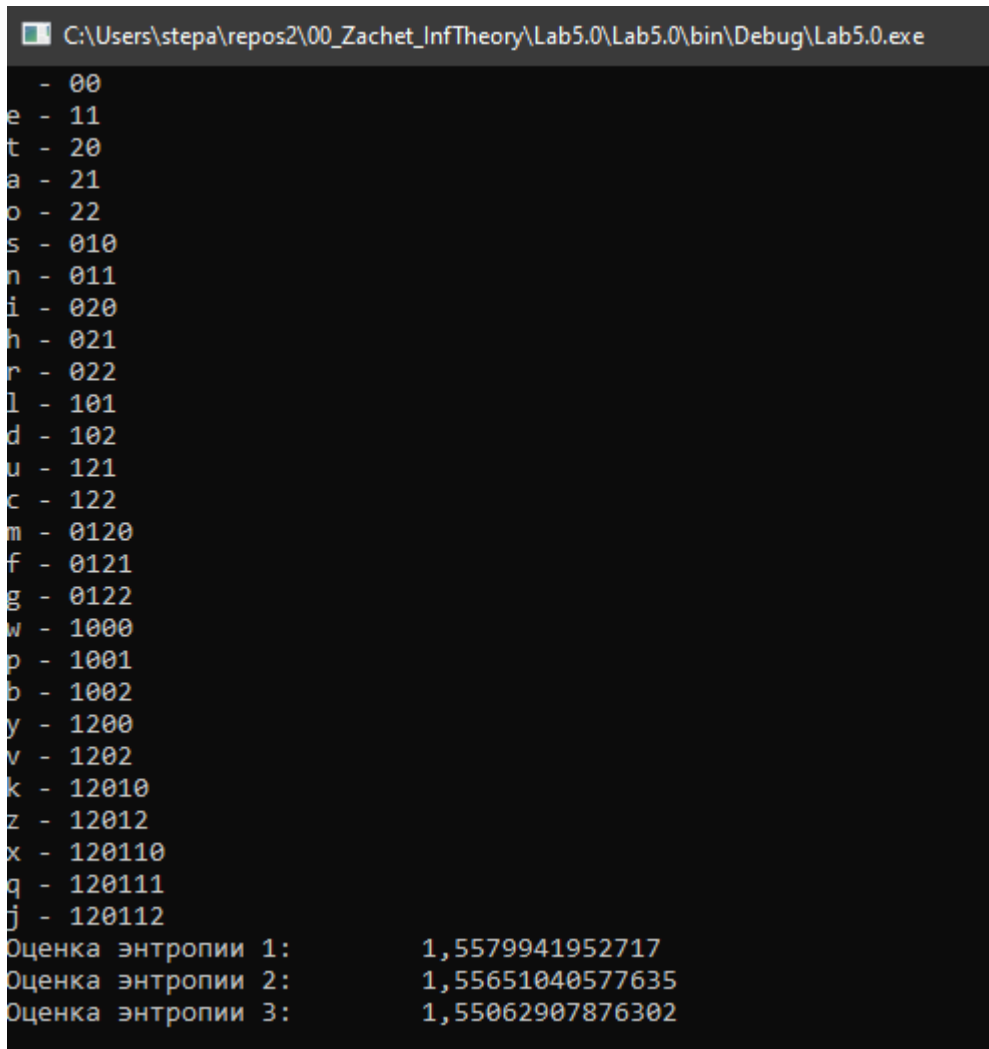
Метод кодирования	Название текста	Энтропия выходной посл-ти (частоты пар символов) (двоичное)	Энтропия выходной посл-ти (частоты пар символов) (недвоичное)	Энтропия выходной послти (частоты троек символов) (недвоичное)
Хаффман	F1.txt(равномер. 3 символа)	0,967475414874	1,58486372265	1,58473124358
Хаффман	Hyperion	0,994933595466	1,55651040577	1,55062907876
Хаффман	Program	0,994817473882	1,56114315677	1,54371112037

Анализ

При кодировании троичным алфавитом энтропия увеличилась в полтора раза.
2,32528514294179

Построил троичное дерево на практическом занятии.

Скриншоты



```
C:\Users\stepa\repos2\00_Zachet_InfTheory\Lab5.0\Lab5.0\bin\Debug\Lab5.0.exe
- 00
e - 11
t - 20
a - 21
o - 22
s - 010
n - 011
i - 020
h - 021
r - 022
l - 101
d - 102
u - 121
c - 122
m - 0120
f - 0121
g - 0122
w - 1000
p - 1001
b - 1002
y - 1200
v - 1202
k - 12010
z - 12012
x - 120110
q - 120111
j - 120112
Оценка энтропии 1:      1,5579941952717
Оценка энтропии 2:      1,55651040577635
Оценка энтропии 3:      1,55062907876302
```

Листинг кода

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Collections;
using System.IO;

namespace Lab5._0
{
    public class Program//making a ternary Huffman coding. Тернарный вроде как
    субоптимальный(по отношению к бинарному), т.к. надо делать тройки.
    {
        static string input;
        static Dictionary<string, double> dicti1 = new Dictionary<string, double>();
        static Dictionary<string, double> dicti2 = new Dictionary<string, double>();
        static Dictionary<string, double> dicti3 = new Dictionary<string, double>();
        static int numberOfChars = 0;
        static int numberOfLettersInABlock = 1;
        static double codeWordAverageLength = 0;
        static void Main(string[] args)
        {
            string path = "C:/Users/stepa/repos2/00_Zachet_InfTheory/Lab5.0/Hyperion.txt";
            using (StreamReader sr = File.OpenText(path))
            {
                input = sr.ReadToEnd();
            }
            // input = "hi hippie";
            HuffmanTree huffmanTree = new HuffmanTree();
            huffmanTree.Build(input);
            huffmanTree.printTree();
            // codeWordAverageLength = huffmanTree.printTreeAndCountAverageLengthOriginal();
            //BitArray encoded = huffmanTree.EncodeOriginal(input);
            List<int> encoded = huffmanTree.Encode(input);
            /* Console.WriteLine("encoded symbols:");
            foreach (var item in encoded)
            {
                Console.Write(item);
            }*/

            string path2 = "C:/Users/stepa/repos2/00_Zachet_InfTheory/Lab5.0/TextConverted.txt";
            using (StreamWriter sw = File.CreateText(path2))
            {
                foreach (int bit in encoded)
                {
                    sw.Write(bit);
                }
            }
        }
    }
}
```

```

    }
}

countProbabilitiesBasedOnRealFrequencyInFile("C:/Users/stepa/repos2/00_Zachet_InfTheory/
Lab5.0/TextConverted.txt", dicti1, numberOfLettersInABlock);
    double first = ShennonFormulaForEntropy(dicti1, numberOfLettersInABlock);
    Console.WriteLine("Оценка энтропии 1:    " + first);

    numberOfLettersInABlock = 2;

countProbabilitiesBasedOnRealFrequencyInFile("C:/Users/stepa/repos2/00_Zachet_InfTheory/
Lab5.0/TextConverted.txt", dicti2, numberOfLettersInABlock);
    Console.WriteLine("Оценка энтропии 2:    " + ShennonFormulaForEntropy(dicti2,
numberOfLettersInABlock));

    numberOfLettersInABlock = 3;

countProbabilitiesBasedOnRealFrequencyInFile("C:/Users/stepa/repos2/00_Zachet_InfTheory/
Lab5.0/TextConverted.txt", dicti3, numberOfLettersInABlock);
    Console.WriteLine("Оценка энтропии 3:    " + ShennonFormulaForEntropy(dicti3,
numberOfLettersInABlock));

    Console.ReadLine();

}
static double ShennonFormulaForEntropy(Dictionary<string, double> dict, int
numberOfLettersInABlock)
{
    //Количество информации, которое мы получаем, достигает максимального
значения, если события равновероятны... Здесь, видимо,
    //сравниваются значения, полученные применением формулы Хартли...
    //Формула Шеннона позволяет высчитать среднее кол-во информации,
передаваемое любым сообщением(блоком символов).
    double sum = 0;
    foreach (var item in dict)
    {
        sum += item.Value * Math.Log(1 / item.Value, 2);
    }
    return sum / numberOfLettersInABlock;
}
static void countProbabilitiesBasedOnRealFrequencyInFile(string path, Dictionary<string,
double> dict, int numberOfLettersInABlock)
{
    string str;
    using (StreamReader sr = File.OpenText(path))
    {
        str = sr.ReadToEnd();
    }
}

```

```

        numberOfChars = str.Length;
        char[] str_chars = str.ToCharArray();
        for (int i = 0; i < numberOfChars - numberOfLettersInABlock; i++)
        {
            string block = str_chars[i].ToString();
            for (int j = 1; j < numberOfLettersInABlock; j++)
            {
                block += str_chars[i + j].ToString();
            }
            if (dict.ContainsKey(block))
            {
                dict[block] += ((double)1 / ((double)numberOfChars));
            }
            else
            {
                dict.Add(block, ((double)1 / ((double)numberOfChars)));
            }
        }
    }
}

public class HuffmanTree
{
    private List<Node> nodes = new List<Node>();
    public Node Root { get; set; }
    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]]++; //Считаем кол-во вхождений каждого символа
        }

        foreach (KeyValuePair<char, int> symbol in Frequencies) //для каждого символа
        алфавита создаём Node
        {
            nodes.Add(new Node() { Symbol = symbol.Key, Frequency = symbol.Value });
        }
        //Для тернарного дерева число листьев д.б. нечётным, чтобы дерево построилось:
        if (nodes.Count % 2 == 0)
            nodes.Add(new Node() { Symbol = '¢', Frequency = 0 });

        while (nodes.Count > 1)
        {

```



```
List<Node> orderedNodes = nodes.OrderBy(node =>
node.Frequency).ToList<Node>();//После каждого создания нового родителя сортирую узлы
по частотам. По возрастанию.
```

//В итоге дерево собирается так как надо(по правилам построения дерева Хаффмана)(новые узлы сортируются по частотам(вероятностям появления символов) и только потом дерево продолжает построение).

```
if (orderedNodes.Count >= 3)
{
    List<Node> taken = orderedNodes.Take(3).ToList<Node>();//берём 3 элемента из
начала и делаем из них List
```

```
// Create a parent node by combining the frequencies:
```

```
Node parent = new Node()
```

```
{
```

```
    Symbol = '*',//У нас 2 или более узлов, соответственно данный узел не будет
листом и его называем звёздочкой.
```

```
    Frequency = taken[0].Frequency + taken[1].Frequency +
taken[2].Frequency,//Складываю частоты. В начале - это наименьшие частоты
```

```
    Left = taken[0],
```

```
    Center = taken[1],
```

```
    Right = taken[2]
```

```
};
```

```
nodes.Remove(taken[0]);
```

```
nodes.Remove(taken[1]);
```

```
nodes.Remove(taken[2]);
```

```
nodes.Add(parent);
```

```
}
```

```
this.Root = nodes.FirstOrDefault();//Корнем дерева назначаю просто первый или
null из nodes
```

```
}
```

```
}
```

```
public void printTree()
```

```
{
```

```
List<Noda> ln = new List<Noda>();
```

```
foreach (var item in Frequencies)
```

```
{
```

```
    List<int> bitarr = Encode(item.Key.ToString());
```

```
    string codeWord = "";
```

```
    foreach (int itemInner in bitarr)
```

```
{
```

```
    if (itemInner == 0)
```

```
        codeWord += "0";
```

```
    else if (itemInner == 1) codeWord += "1";
```

```
    else codeWord += "2";
```

```
}
```

```
    ln.Add(new Noda() { frequency = item.Value, symbol = item.Key, codeInString =
codeWord });
```

```
}
```

```

List<Noda> SortedList = ln.OrderByDescending(o => o.frequency).ToList();
foreach (var item in SortedList)
{
    Console.WriteLine(item.symbol.ToString() + " - " + item.codeInString);
}
}
public class Noda
{
    public int frequency { get; set; }
    public char symbol { get; set; }
    public BitArray code { get; set; }
    public string codeInString { get; set; }
}

public List<int> Encode(string source)
{
    List<int> encodedSource = new List<int>();

    for (int i = 0; i < source.Length; i++)
    {
        List<int> encodedSymbol = this.Root.Traverse(source[i], new List<int>()); //В метод
        Traverse передаю символ для поиска и новый
        //список для хранения кодового слова. Он у меня интовый, т.к. алфавит
        небинарный и требуется больше, чем 2 символа(в отличие от массива BitArray).
        //Возвращается ссылка на этот же список, но уже заполненный естественно.
        encodedSource.AddRange(encodedSymbol);
    }

    List<int> bits = new List<int>();
    return encodedSource;
}

public string Decode(List<int> bits)
{
    Node current = this.Root;
    string decoded = "";

    foreach (int bit in bits)
    {
        if (bit == 2)
        {
            if (current.Right != null)
            {
                current = current.Right;
            }
        }
        else if (bit == 1)
        {

```

```

        if (current.Center != null)
        {
            current = current.Center;
        }
    }
    else
    {
        if (current.Left != null)
        {
            current = current.Left;
        }
    }

    if (IsLeaf(current))
    {
        decoded += current.Symbol;
        current = this.Root;
    }
}

return decoded;
}

public bool IsLeaf(Node node)//(is the last element of a branch)
{
    return (node.Left == null && node.Right == null);
}

}

public class Node
{
    public char Symbol { get; set; }//a symbol of this Node
    public int Frequency { get; set; }
    public Node Right { get; set; }
    public Node Left { get; set; }
    public Node Center { get; set; }

    public List<int> Traverse(char symbol, List<int> data)
    {
        // Leaf
        if (Right == null && Center == null && Left == null)
        {
            if (symbol.Equals(this.Symbol))//Ищем символы, проходя по дереву
            {
                return data;//Если дошли до листа и его символ равен искомому, то
                возвращаем переданный сюда как параметр List<int> data
            }
        }
    }
}

```

```

else
{
    return null;
}
}
else
{
    List<int> left = null;
    List<int> center = null;
    List<int> right = null;

    if (Left != null)
    {
        List<int> leftPath = new List<int>();
        leftPath.AddRange(data);
        leftPath.Add(2); //приписываем циферки к рёбрам(стрелкам на нижние узлы)

        left = Left.Traverse(symbol, leftPath); //В Traverse передаю уже сохранённый
leftPath, а там(в следующем рекурсивном вызове Traverse) создаётся ещё один leftPath и
        //к нему опять же приписывается тот leftPath, который был передан при вызове
Traverse (leftPath.AddRange(data));, а также приписывается соответствующее пути
название ребра
        //(для leftPath это 0, для centerPath это 1 и т.д.). Если дерево построено
правильно, то только один из путей закончится тем, что ссылки на Right, Center, Left будут
null,
        //и symbol.Equals(this.Symbol) выполнится для данного узла, и вернётся
data(список символов, описывающих путь к данному узлу).

        //В итоге все узлы, не содержащие искомый символ, вернут null, в самом конце
этого метода, соответственно, произойдёт отбор и возврат единственного полученного
пути, который будет не null,
        //а то, что вернул лист, содержащий искомый символ.

        if (Center != null)
        {
            List<int> centerPath = new List<int>();
            centerPath.AddRange(data);
            centerPath.Add(1);

            center = Center.Traverse(symbol, centerPath);
        }

        if (Right != null)
        {
            List<int> rightPath = new List<int>();
            rightPath.AddRange(data);
            rightPath.Add(0);

```

```
        right = Right.Traverse(symbol, rightPath);
    }

    if (left != null)
    {
        return left;
    }
    else if (center != null)
    {
        return center;
    }
    else
    {
        return right;
    }
}
}
```