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

Кафедра прикладной математики и кибернетики

Лабораторная работа № 4
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

Практическая работа №4

Оптимальное побуквенное кодирование

Цель работы: Экспериментальное изучение процесса сжатия текстового файла.

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

Результат: программа, тестовые примеры, отчет.

1. Запрограммировать процедуру двоичного кодирования текстового файла. В качестве метода кодирования использовать или метод Шеннона, или метод Фано, или метод Хаффмана. Текстовые файлы использовать те же, что и в практических работах 1, 2, 3.

2. Проверить, что построенный код для каждого файла является префиксным. Вычислить среднюю длину кодового слова и оценить избыточность каждого построенного кода.

3. После кодирования текстового файла вычислить оценки энтропии выходной последовательности, используя частоты отдельных символов, пар символов и троек символов и заполнить таблицу.

Метод кодирования	Название текста	Оценка избыточности кодирования	Оценка энтропии выходной посл-ти (частоты символов)	Оценка энтропии выходной посл-ти (частоты пар символов)	Оценка энтропии выходной посл-ти (частоты троек символов)

Избыточность кодирования определяется как $r = L_{ср} - H$, где H – энтропия текста, $L_{ср}$ – средняя длина кодового слова.

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.

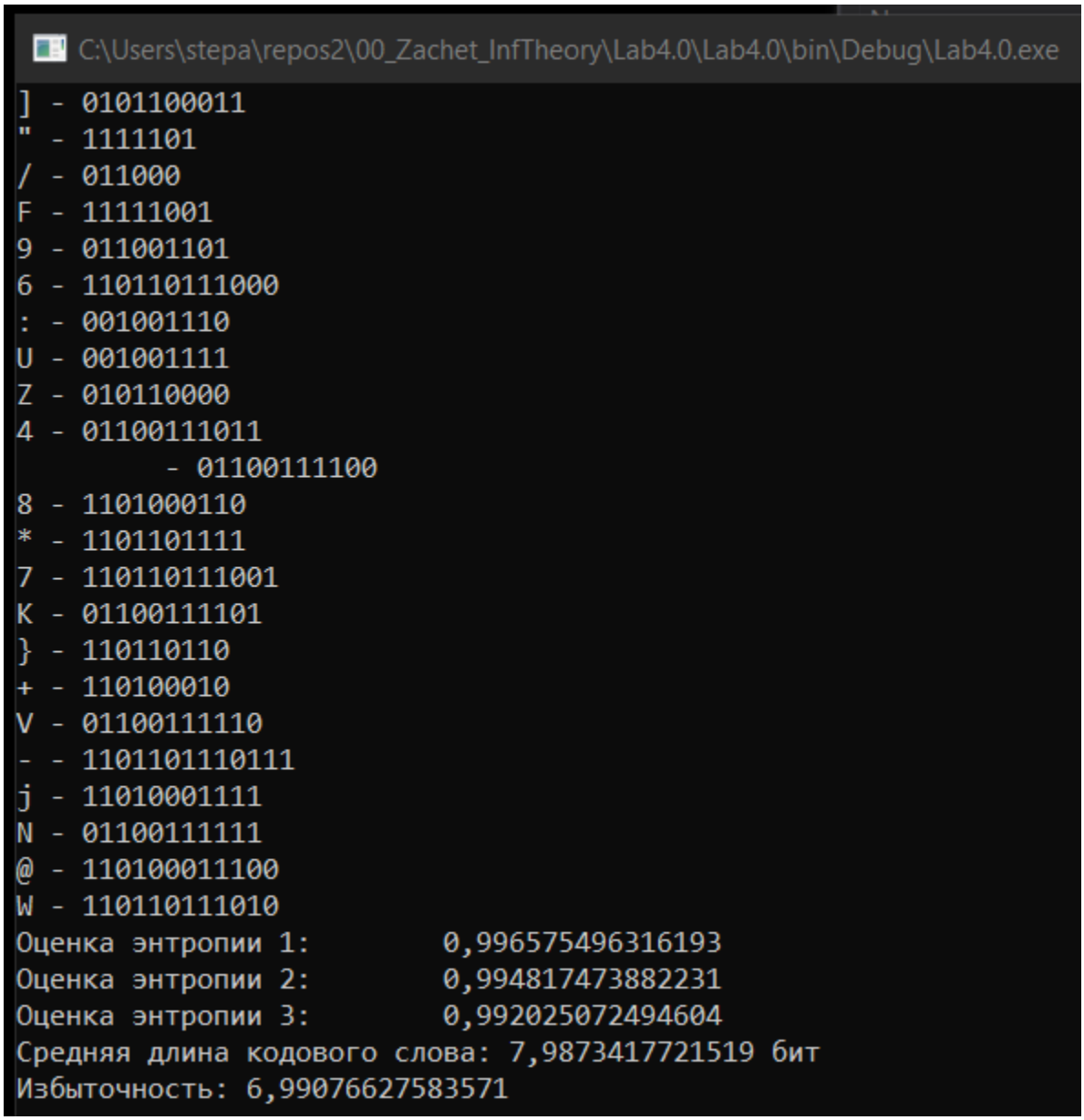
В отчет не нужно включать содержимое этого файла.

Решение

Метод кодирования	Название текста	Оценка избыточности кодирования	Оценка энтропии выходной посл-ти (частоты символов)	Оценка энтропии выходной посл-ти (частоты пар символов)	Оценка энтропии выходной посл-ти (частоты троек символов)
Хаффман	F1.txt(равновер. 3 символа)	0,694970831173	0,971695835492	0,967475414874	0,964623983170
Хаффман	Hyperion.txt	4,74572079807	0,995019942667	0,994933595466	0,99474061251
Хаффман	Program.cs	6,99076627583	0,996575496316	0,994817473882	0,992025072494

Анализ

Скриншоты



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 Lab4._0
{
    public class Program
    {
        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)
        {
            readAFileToString();
            HuffmanTree huffmanTree = new HuffmanTree();
            huffmanTree.Build(input);
            codeWordAverageLength = huffmanTree.printTreeAndCountAverageLength();
            BitArray encoded = huffmanTree.Encode(input);

            string path2 =
"C:/Users/stepa/repos2/00_Zachet_InfTheory/Lab4.0/ProgramConverted.txt";
            using (StreamWriter sw = File.CreateText(path2))
            {
                foreach (bool bit in encoded)
                {
                    sw.Write((bit ? 1 : 0) + "");
                }
            }

countProbabilitiesBasedOnRealFrequencyInFile("C:/Users/stepa/repos2/00_Zachet_InfTheory/L
ab4.0/ProgramConverted.txt", dicti1, numberOfLettersInABlock);
            double first = ShennonFormulaForEnthropy(dicti1, numberOfLettersInABlock);
            Console.WriteLine("Оценка энтропии 1:        " + first);

            numberOfLettersInABlock = 2;

countProbabilitiesBasedOnRealFrequencyInFile("C:/Users/stepa/repos2/00_Zachet_InfTheory/L
ab4.0/ProgramConverted.txt", dicti2, numberOfLettersInABlock);
            Console.WriteLine("Оценка энтропии 2:        " +
ShennonFormulaForEnthropy(dicti2, numberOfLettersInABlock));

            numberOfLettersInABlock = 3;

countProbabilitiesBasedOnRealFrequencyInFile("C:/Users/stepa/repos2/00_Zachet_InfTheory/L
ab4.0/ProgramConverted.txt", dicti3, numberOfLettersInABlock);
            Console.WriteLine("Оценка энтропии 3:        " +
ShennonFormulaForEnthropy(dicti3, numberOfLettersInABlock));

            Console.WriteLine("Средняя длина кодового слова: " + codeWordAverageLength +
" бит");
            Console.WriteLine("Избыточность: " + (codeWordAverageLength - first));

            Console.ReadLine();

        }
        static double ShennonFormulaForEnthropy(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));//   /
(double)numberOfLettersInABlock));
                }
                else
                    dict.Add(block, ((double)1 / ((double)numberOfChars)));// /
(double)numberOfLettersInABlock))) ;
            }
        }
        public static void readAFileToString()
        {
            string path = "C:/Users/stepa/repos2/00_Zachet_InfTheory/Lab4.0/Program.txt";
            using (StreamReader sr = File.OpenText(path))
            {
                input = sr.ReadToEnd();
            }
        }
    }
    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 });
            }
```

```
            while (nodes.Count > 1)
            {
                List<Node> orderedNodes = nodes.OrderBy(node =>
node.Frequency).ToList<Node>();//Сортирую узлы по частотам. По возрастанию.

                if (orderedNodes.Count >= 2)
                {
                    // Take first two items
                    List<Node> taken = orderedNodes.Take(2).ToList<Node>();//берём 2
элемента и делаем из них List

                    // Create a parent node by combining the frequencies
                    Node parent = new Node()                //Дерево строю
                    {
                        Symbol = '*',
                        Frequency = taken[0].Frequency + taken[1].Frequency,
                        Left = taken[0],
                        Right = taken[1]
                    };

                    nodes.Remove(taken[0]);
                    nodes.Remove(taken[1]);
                    nodes.Add(parent);
                }
                this.Root = nodes.FirstOrDefault();
            }
        }

        public double printTreeAndCountAverageLength()
        {
            double L = 0;
            foreach (var item in Frequencies)
            {
                BitArray bitarr = Encode(item.Key.ToString());
                Console.Write(item.Key.ToString() + " - ");
                string codeWord = "";
                foreach (bool itemInner in bitarr)
                {
                    if (itemInner)
                        codeWord += "1";
                    else codeWord += "0";
                }
                Console.WriteLine(codeWord);
                L += codeWord.Length;
            }
            return L / (double)Frequencies.Count;
        }

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

}
public class Node
{
    public char Symbol { get; set; }
    public int Frequency { get; set; }
    public Node Right { get; set; }
    public Node Left { 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;
                }
            }
        }
    }
}
```