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 3](#_Toc66657736)

[Решение 4](#_Toc66657737)

[Анализ 4](#_Toc66657738)

[Скриншоты 5](#_Toc66657739)

[Листинг кода 5](#_Toc66657740)

# ЗАДАНИЕ

Практическая работа №4

Оптимальное побуквенное кодирование

Цель работы: Экспериментальное изучение процесса сжатия текстового файла.

Язык программирования: С, С++, С#, 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.

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

1. 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* = *Lcp* −*H* , где *H* – энтропия текста, *L*cp  – средняя длина кодового слова.

1. 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

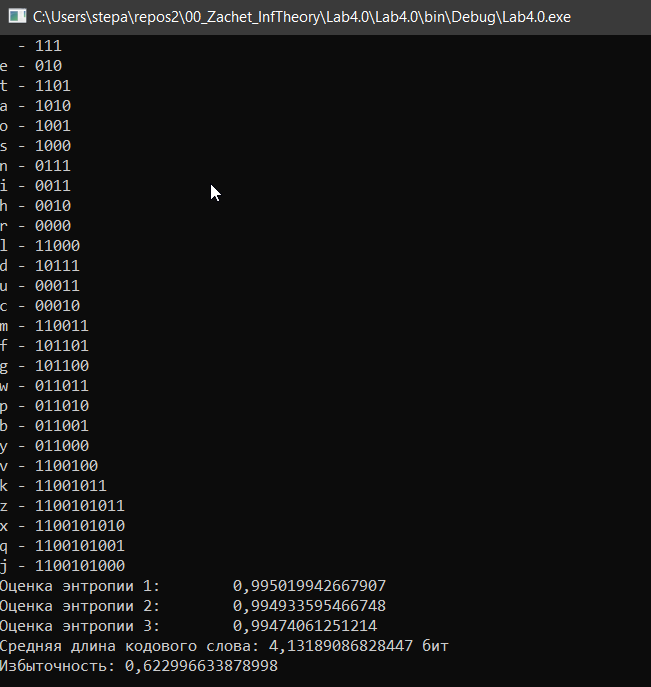
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Метод кодирования | Название текста | Оценка избыточности кодирования | Оценка энтропии выходной посл-ти (частоты символов) | Оценка энтропии выходной посл-ти (частоты пар символов) | Оценка энтропии выходной посл-ти (частоты троек символов) |
| Хаффман | F1.txt(равновер.  3 символа) | 0,336066666666 | 0,971695835492 | 0,967475414874 | 0,964623983170 |
| Хаффман | Hyperion.txt | 0,868109131715 | 0,995019942667 | 0,994933595466 | 0,99474061251 |
| Хаффман | Program.cs | 2,32528514294 | 0,996575496316 | 0,994817473882 | 0,992025072494 |

# Анализ

Избыточность текста Hyperion соответствует известной примерной избыточности европейских языков (примерно 0.7).

Видно, что энтропия стремится к логарифму 2 по основанию 2 (1), т.к. код Хаффмана оптимальный(качественно кодирует с минимальной длиной среднего кодового слова, избыточность становится минимальной).

# 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//codes are read from root to leafs

{

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/Lab4.0/Hyperion.txt";

using (StreamReader sr = File.OpenText(path))

{

input = sr.ReadToEnd();

}

HuffmanTree huffmanTree = new HuffmanTree();

huffmanTree.Build(input);

codeWordAverageLength = huffmanTree.printTreeAndCountAverageLength(input);

BitArray encoded = huffmanTree.Encode(input);

string path2 = "C:/Users/stepa/repos2/00\_Zachet\_InfTheory/Lab4.0/TextConverted.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/Lab4.0/TextConverted.txt", dicti1, numberOfLettersInABlock);

double first = ShennonFormulaForEnthropy(dicti1, numberOfLettersInABlock);

Console.WriteLine("Оценка энтропии 1: " + first);

numberOfLettersInABlock = 2;

countProbabilitiesBasedOnRealFrequencyInFile("C:/Users/stepa/repos2/00\_Zachet\_InfTheory/Lab4.0/TextConverted.txt", dicti2, numberOfLettersInABlock);

Console.WriteLine("Оценка энтропии 2: " + ShennonFormulaForEnthropy(dicti2, numberOfLettersInABlock));

numberOfLettersInABlock = 3;

countProbabilitiesBasedOnRealFrequencyInFile("C:/Users/stepa/repos2/00\_Zachet\_InfTheory/Lab4.0/TextConverted.txt", dicti3, numberOfLettersInABlock);

Console.WriteLine("Оценка энтропии 3: " + ShennonFormulaForEnthropy(dicti3, numberOfLettersInABlock));

double wholeFileEntropy = Math.Log(huffmanTree.Frequencies.Count, 2);

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

Console.WriteLine("Избыточность: " + (wholeFileEntropy - codeWordAverageLength));

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));

}

else

dict.Add(block, ((double)1 / ((double)numberOfChars)));

}

}

}

public class HuffmanTree

{

public 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 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, соотвтетсвенно данный узел не будет листом и его называем звёздочкой.

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

Left = taken[0],

Right = taken[1]

};

nodes.Remove(taken[0]);

nodes.Remove(taken[1]);

nodes.Add(parent);

}

this.Root = nodes.FirstOrDefault();

}

}

public double printTreeAndCountAverageLength(string inp)

{

double L = 0;

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

foreach (var item in Frequencies)

{

BitArray bitarr = Encode(item.Key.ToString());

string codeWord = "";

foreach (bool itemInner in bitarr)

{

if (itemInner)

{

codeWord += "1";

}

else codeWord += "0";

}

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

L += codeWord.Length \* (item.Value / (double)inp.Length);

}

List<Noda> SortedList = ln.OrderByDescending(o => o.frequency).ToList();

foreach (var item in SortedList)

{

Console.WriteLine(item.symbol.ToString() + " - " + item.codeInString);

}

return L;

}

public class Noda

{

public int frequency { get; set; }

public char symbol { get; set; }

public BitArray code { get; set; }

public string codeInString { get; set; }

}

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;

}

}

}

}

}