Министерство образования и науки РФ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Ивановский государственный энергетический университет имени В.И. Ленина»

Кафедра программного обеспечения компьютерных систем

ОТЧЕТ

ПО КУРСОВОЙ РАБОТЕ

по курсу

«Алгоритмы и структуры данных»

Разработка кодера, использующего статический алгоритм Шеннона-Фано

Выполнил: студент группы

3-42 Решетников Н.Р.

Проверил: Пантелеев Е.Р.

Иваново 2023

# Постановка задачи

Написать кодер, использующий статический алгоритм Шеннона-Фано.

# Описание алгоритма

Алгоритм Шеннона — Фано строит модель кодирования в формате так называемого дерева префиксных кодов — ДПК. Это бинарное дерево, листьям которого сопоставлены символы входного потока кодера, а дуги размечены разрядами двоичного кода таким образом, что для любой родительской вершины одна исходящая дуга помечена нулем, а вторая — единицей.

Прежде чем строить дерево, построим таблицу частот символов. Для этого считаем всё содержимое из входящего файла. Соберем список уникальных символов, которые есть в файле, посчитаем повторения и соотнесем каждому символу количество его повторений в файле. Полученный список структур (с полями: символ, частота) отсортируем по убыванию частоты символов. Так получим таблицу частот.

Для заполнения дерева будем рекурсивно разбивать нашу таблицу на две, таким образом, чтобы сумма частот символов в каждой из двух таблиц отличалась минимально. Одна из получившихся таблиц пойдет в левое поддерево, другая в правое. Каждую из таблиц будем разбивать соответственно, пока не дойдем до таблицы в один символ. Запишем информацию о символе и его частоте в лист дерева. По окончании этого этапа мы будем иметь дерево, содержащее в себе все символы входящего потока.

Следующим этапом мы пробежимся по дереву, чтобы соотнести каждому символу в нем свой бинарный код. Из-за способа построения дерева символы с наибольшей частотой будут иметь наименьшую длину кода, что и поможет сжать входящий файл. Шаг по дереву влево примем за 0, а вправо за 1. Пройдясь до символа и запомнив шаги мы получим код этого символа. Получив все коды символов соберем их в такой же список структур (поля: символ, код). Так получим таблицу кодов.

Перепишем входной поток, заменяя символы из него на их коды с помощью таблицы. Коды будут получаться разной длины. Записывать мы их будем подряд в массив байтов, применяя к ним операции побитового сдвига, чтобы не оставить между ними свободного места.

В выходной файл запишем следующее: символ (char) и его частота (int). Так запишем в строку без разделителей все пары символ-частота, затем, последней будет идти пара символ \0 и его код. После этого запишем в строку массив байтов – исходные данные, которые мы закодировали.

# Допущения при разработке быстрого прототипа

* Входящий поток состоит из символов.
* Код символа, полученный в результате процесса сжатия не должен превышать размера одного байта.
* Ввод данных осуществлявляется из текстового файла. Путь к файлу указывается в коде программы.

# Код программы на данном этапе разработки

using System;

using System.Collections.Generic;

using System.Collections.Immutable;

using System.Collections.Specialized;

using System.IO;

using System.Net.Http.Headers;

using System.Text;

using static System.Collections.Specialized.BitVector32;

using static System.Net.Mime.MediaTypeNames;

namespace ShenonFano // Note: actual namespace depends on the project name.

{

public class Program

{

#region Tree

public static bool TREE\_LEFT = true;

public static bool TREE\_RIGHT = false;

public class TreeNode<T>

{

private T \_value;

private TreeNode<T> \_left;

private TreeNode<T> \_right;

private int \_count;

public TreeNode(T value)

{

\_value = value;

\_count = 0;

}

public TreeNode<T> this[bool trig]

{

get

{

if (trig) return \_left;

else return \_right;

}

}

public TreeNode<T> Root { get; private set; }

public TreeNode<T> Left { get => \_left; set { \_left = value; \_left.Count = \_count + 1; } }

public TreeNode<T> Right { get => \_right; set { \_right = value; \_right.Count = \_count + 1; } }

public int Count { get => \_count; private set => \_count = value; }

public T Value { get => \_value; set => \_value = value; }

public virtual TreeNode<T> AddChild(T value, bool trig)

{

var node = new TreeNode<T>(value) { Root = this };

node.Count = \_count + 1;

if (trig)

\_left = node;

else

\_right = node;

return node;

}

public virtual TreeNode<T> AddChild(TreeNode<T> value, bool trig)

{

value.Root = this;

var node = value;

node.Count = \_count + 1;

if (trig)

\_left = node;

else

\_right = node;

return node;

}

public bool RemoveChild(TreeNode<T> node, bool trig)

{

if (trig)

{

if (\_left != null)

{ \_left = null; return true; }

else

return false;

}

else

{

if (\_right != null)

{ \_right = null; return true; }

else

return false;

}

}

public void ActionDo(Action<T> action)

{

action(Value);

if (\_left != null) \_left.ActionDo(action);

if (\_right != null) \_right.ActionDo(action);

}

public void ActionNodeDo(Action<TreeNode<T>> action)

{

action(this);

if (\_left != null) \_left.ActionNodeDo(action);

if (\_right != null) \_right.ActionNodeDo(action);

}

}

#endregion

public class tableNote : IComparable

{

public char symbol;

public int count;

public int zeros;

public tableNote(char s, int c)

{

symbol = s;

count = c;

zeros = 0;

}

public tableNote(char s, int c, int z)

{

symbol = s;

count = c;

zeros = z;

}

public int CompareTo(object obj)

{

if (obj == null)

{

return 1;

}

else

{

tableNote other = obj as tableNote;

return -(this.count - other.count);

}

}

}

static void Main(string[] args)

{

string inputFilePath = "D:\\test.txt";

string inputString = "";

FileStream inputFile = File.OpenRead(inputFilePath);

BinaryReader inputReader = new BinaryReader(inputFile, encoding: Encoding.UTF8);

while(inputReader.PeekChar() != -1)

{

inputString += inputReader.ReadChar();

}

inputString += '\0';

List<tableNote> table = new List<tableNote>();

// построение таблицы частот

foreach (char symb in inputString)

{

bool newsymb = true;

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

{

if (symb == table[i].symbol)

{

tableNote el = new tableNote(symb, table[i].count + 1);

table[i] = el;

newsymb = false;

break;

}

}

if(newsymb)

{

table.Add(new tableNote(symb, 1));

}

}

// сортировка

table.Sort();

// разбиение таблицы

TreeNode<tableNote> tree = new TreeNode<tableNote>(new tableNote('o', 0));

SplitTable(table, tree);

List<tableNote> codeTable = new List<tableNote>(); // здесь поле count отвечает за код символа

MergeCodeTable(codeTable, tree, 0, 0, true);

List<byte> outputWord = new List<byte>();

byte tbyte = 0;

int freebits = 8;

foreach (char symb in inputString)

{

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

{

if (symb == codeTable[i].symbol)

{

int codeLen = 1;

int codeSymb = codeTable[i].count; // код символа

for (int j = 1; j < 8; j++)

{

if (codeTable[i].count >= (int)Math.Pow(2, j))

codeLen++;

else

break;

}

codeLen += codeTable[i].zeros;

NewByte:

if (freebits >= codeLen)

{

codeSymb = codeSymb << (freebits - codeLen);

tbyte |= (byte)codeSymb;

freebits -= codeLen;

}

else if (freebits == 0)

{

outputWord.Add(tbyte);

tbyte = 0;

freebits = 8;

goto NewByte;

}

else

{

int part1Len = freebits; // длина первой части кода

int part2Len = codeLen - part1Len; // длина второй части кода (идет в другой байт)

byte mask1 = 0;

byte mask2 = 0;

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

{

if (j <= part2Len - 1)

{

mask2 += (byte)Math.Pow(2, j);

}

else

{

mask1 += (byte)Math.Pow(2, j);

}

}

byte part1, part2;

part1 = (byte)(codeSymb & mask1);

part1 = (byte)(part1 >> part2Len);

tbyte |= part1;

outputWord.Add(tbyte);

tbyte = 0;

part2 = (byte)(codeSymb & mask2);

part2 = (byte)(part2 << (8 - part2Len));

tbyte |= part2;

freebits = 8 - part2Len;

}

break;

}

}

}

outputWord.Add(tbyte);

byte[] inputWord = outputWord.ToArray();

string filepath = $"D:/note.dat";

using (BinaryWriter writer = new BinaryWriter(File.Open(filepath, FileMode.OpenOrCreate), encoding:Encoding.UTF8))

{

foreach(tableNote tnote in table)

{

writer.Write(tnote.symbol);

writer.Write(tnote.count);

}

writer.Write(inputWord);

writer.Close();

}

}

public static void SplitTable(List<tableNote> table, TreeNode<tableNote> tnode)

{

if (table.Count == 2)

{

tnode.AddChild(table[0], TREE\_LEFT);

tnode.AddChild(table[1], TREE\_RIGHT);

}

else if (table.Count < 2)

{

tnode.Value = table[0];

}

else

{

int sum = 0;

int min = int.MaxValue;

foreach (tableNote t in table)

{

sum += t.count;

}

int sumL = 0;

int sumR = sum;

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

{

sumL += table[i].count;

sumR -= table[i].count;

int diff = Math.Abs(sumL - sumR);

if (diff > min)

{

i--;

List<tableNote> tLeft = new List<tableNote>();

List<tableNote> tRight = new List<tableNote>();

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

{

if (j <= i)

{

tLeft.Add(table[j]);

}

else

{

tRight.Add(table[j]);

}

}

tnode.AddChild(new tableNote('o', 0), TREE\_LEFT);

tnode.AddChild(new tableNote('o', 0), TREE\_RIGHT);

SplitTable(tLeft, tnode.Left);

SplitTable(tRight, tnode.Right);

break;

}

else

{

min = diff;

}

}

}

}

public static void MergeCodeTable(List<tableNote> table, TreeNode<tableNote> tnode, int code, int zerocount, bool zeroflag)

{

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

{

zerocount--;

table.Add(new tableNote(tnode.Value.symbol, code, zerocount));

}

else

{

if (tnode.Left != null)

{

if (zeroflag)

{

zerocount++;

}

MergeCodeTable(table, tnode.Left, code \* 2, zerocount, zeroflag);

}

if(tnode.Right != null)

{

MergeCodeTable(table, tnode.Right, code \* 2 + 1, zerocount, false);

}

}

}

}

}

# Результаты выполнения программы

На рисунке 1 показан входной \*.txt файл. На рисунке 2 показан результирующий файл \*.dat.

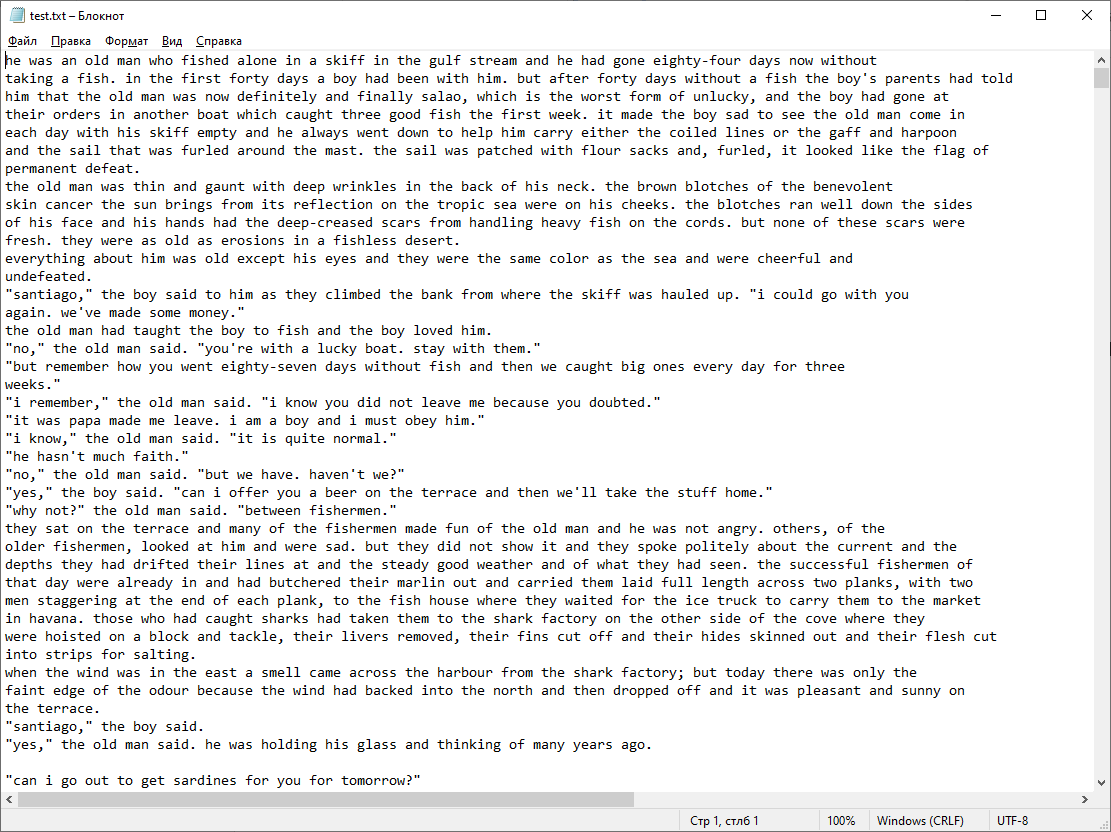


Рисунок 1 – входящий \*.txt файл.

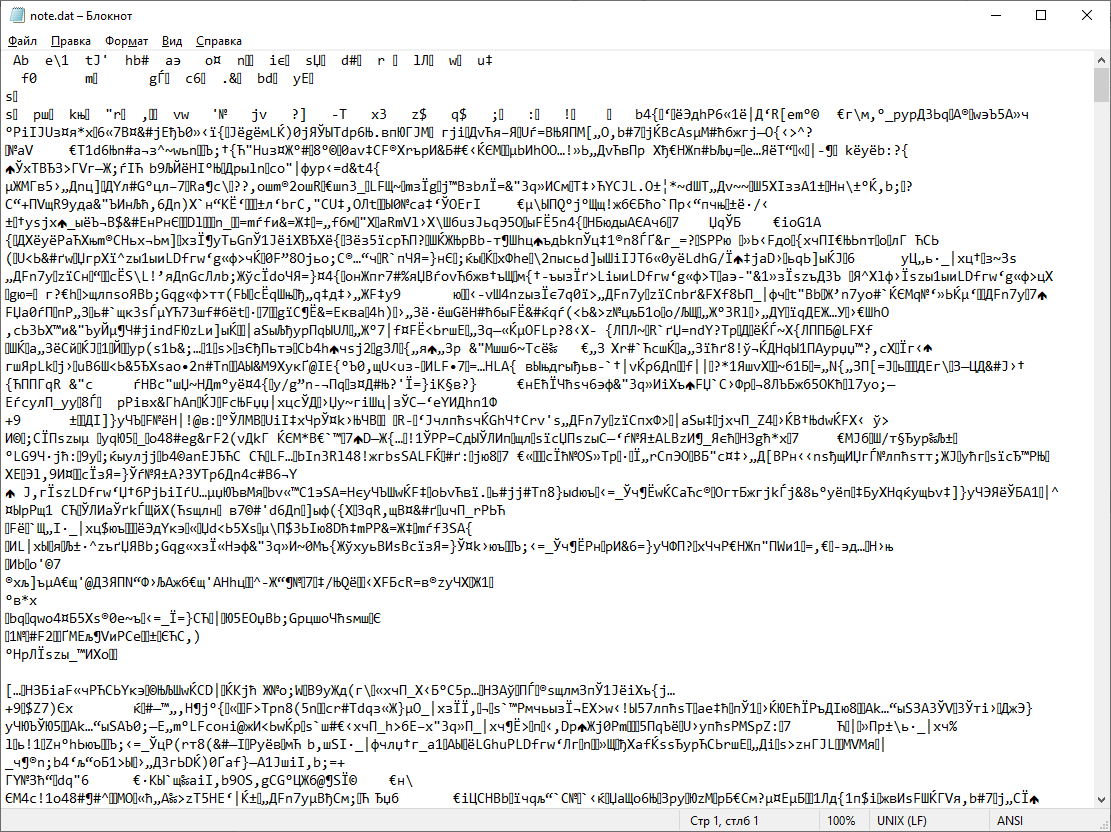


Рисунок 2 – результирующий \*.dat файл.