Министерство образования и науки РФ

ФГБОУ ВПО «Тамбовский государственный технический университет»

Кафедра « »

Лабораторная работа №3

по дисциплине «Теория информации»

«Алгоритм Хаффмана»

Выполнил: студент гр. −31 Д. Ю.

Проверил: Н. В.

Тамбов,

***Цели и задачи.***

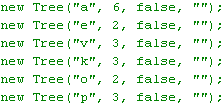
Разработать приложение, реализующее алгоритм Хаффмана.

***Теория.***

Идея, положенная в основу кодировании Хаффмана, основана на частоте появления символа в последовательности. Символ, который встречается в последовательности чаще всего, получает новый очень маленький код, а символ, который встречается реже всего, получает, наоборот, очень длинный код.

***Решение задачи****.*

Для наглядности приведу пример таблицы частот, тоесть это подразумивает то, что этап с подсчетом вероятностей пройден и получена следующая таблица:

Каждая запись может быть прочитана как «символ а встречается 6 раз в исходном сообщении». После необходимо построить дерево Хаффмана, где минимальные по вероятности символы образуют новый «лист», а процедура будет повторяться до тех пор, пока не выйдем на корень. Пример такого дерева для исходной таблицы:

После того, как составили дерево, необходимо пройтись парсером от корня до каждого конечного листа и составить «цепочки» - зависимости символа от его «глубины» в дереве (влево для 0 и вправо для 1):

v = 000

k = 001

a = 01

e = 100

o = 101

p = 11

Осталось соотнести каждому символу исходного сообщения свою цепочку.

***Модули***

За подсчет количества символов/частоты отвечает модуль **private** Dictionary**<**string**,** int**>** ЧастотыБукв**():**

/// <summary>

/// Вернуть список Буква-Частота

/// </summary>

**private** Dictionary**<**string**,** int**>** ЧастотыБукв**()**

**{**

var dict **=** **new** Dictionary**<**string**,** int**>();**

char**[]** arr **=** \_txtCodr**.**ToCharArray**();**

//посчитать каждый символ

**for** **(**int i **=** 0**;** i **<** \_txtCodr**.**Length**;** i**++)**

**{**

var chr **=** \_txtCodr**[**i**];**

int indx **=** 1**;**

**for** **(**int j **=** i**+**1**;** j **<** \_txtCodr**.**Length**;** j**++)**

**{**

**if** **(**chr **==** arr**[**j**])**

**{**

arr**[**j**]** **=** 'ⓤ'**;**//типо занулим это место

indx**++;**//количество повторов символа chr

dict**[**chr**.**ToString**()]** **=** indx**;**

**}**

**else** **if** **(!**dict**.**ContainsKey**(**chr**.**ToString**()))**

dict**.**Add**(**chr**.**ToString**(),** indx**);**

**}**

//для входной строки из одного символа:

**if** **(**i **+** 1 **==** \_txtCodr**.**Length **&** arr**.**Length **==** 1**)**

dict**[**chr**.**ToString**()]** **=** indx**;**

//для окончаний:

**else** **if** **(!**dict**.**ContainsKey**(**chr**.**ToString**()))**

dict**.**Add**(**chr**.**ToString**(),** indx**);**

**}**

**return** dict**;**

**}**

За составление дерева отвечает модуль **private** void Raspred**(**Dictionary**<**string**,** int**>** dic**):**

/// <summary>

/// Составим дерево

/// </summary>

**private** void Raspred**(**Dictionary**<**string**,** int**>** dic**)**

**{**

int len **=** dic**.**Count**;**

Tree**[,]** trees **=** **new** Tree**[**len**,** len**];**

int ik **=** 0**;**

**foreach** **(**KeyValuePair**<**string**,** int**>** keyValuePair **in** dic**)**

**{**

trees**[**0**,** ik**]** **=** **new** Tree**(**keyValuePair**.**Key**,** keyValuePair**.**Value**,** **false,** ""**);**

ik**++;**

**}**

Tree**.**Length **=** len**;**//количество записей trees[,] в строке

//--------------------------------------------------------------

**for** **(**int k **=** 0**;** k **<** len **-** 1**;** k**++)**

**{**

var arr **=** **new** Tree**[**len **-** k**];**

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

**{**

arr**[**i**]** **=** trees**[**k**,** i**];**

**}**

Tree tmin1**,** tmin2**;**

**new** Tree**(**arr**,** **out** tmin1**,** **out** tmin2**);**

var sum **=** tmin1 **+** tmin2**;**

trees**[**k **+** 1**,** 0**]** **=** **new** Tree**(**sum**);**

var index **=** 0**;**

**for** **(**int i **=** 1**;** i **<** len **-** **(**k **+** 1**);** **)**

**{**

**if** **(**trees**[**k**,** index**]** **!=** tmin1 **&&** trees**[**k**,** index**]** **!=** tmin2**)**

**{**

trees**[**k **+** 1**,** i**]** **=** **new** Tree**(**trees**[**k**,** index**]);**

i**++;**

**}**

index**++;**

**}**

**}**

//--------------------------------------------------------------

СимволЦепь **=** **new** List**<**KeyValuePair**<**Tree**,** string**>>();**

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

**{**

var keyPair **=** Parser**(**trees**);**

СимволЦепь**.**Add**(**keyPair**);**

**}**

**}**

За проход по дереву и составление цепочек отвечает модуль **private** static KeyValuePair**<**Tree**,** string**>** Parser**(**Tree**[,]** trees**):**

/// <summary>

/// Проити по дереву и построить цепи 010110

/// </summary>

/// <param name="trees">Дерево</param>

/// <returns>Вернет пару элемент=цепь</returns>

**private** static KeyValuePair**<**Tree**,** string**>** Parser**(**Tree**[,]** trees**)**

**{**

var str **=** ""**;** var s **=** "0"**;**

var tmp **=** trees**[**Tree**.**Length **-** 1**,** 0**];**

var indx **=** Tree**.**Length **-** 2**;**

var flag **=** **false;**

**for** **(**int i **=** 0**;** i **<** Tree**.**Length**;** **)**

**{**

**if** **(**trees**[**indx**,** i**]** **==** **null)**

**{**

tmp**.**Flag **=** **true;**

UpPars**(**trees**,** **out** tmp**,** **out** indx**,** indx **+** 1**,** tmp**);**

flag **=** **true;**

i **=** 0**;**

**if** **(**str**.**Length **==** 1**)** str **=** "0"**;**

**else** str **=** str**.**Remove**(**str**.**Length **-** \_ival**);**

**}**

**if** **(**trees**[**indx**,** i**].**Equals**(**tmp**))**

**{**

//опускаемся на уровень ниже

//когда элементы не изменяются

tmp **=** trees**[**indx**,** i**];**

indx**--;**

**if** **(**indx **<** 0**)**

**{**

**if** **(!**tmp**.**Flag**)**

**{**

tmp**.**Flag **=** **true;**

tmp**.**Napravlenie **=** str**;**

//достигли дна, возвращаем элемент tmp

//и комбинацию переходов 010101

**return** **new** KeyValuePair**<**Tree**,** string**>(**tmp**,** str**);**

**}**//иначе надо подниматься

UpPars**(**trees**,** **out** tmp**,** **out** indx**,** indx**,** tmp**);**

flag **=** **true;**

**}**

i **=** 0**;**

**}**

**else**

**{**

**if** **(**tmp**.**Comparer**(**trees**[**indx**,** i**])** **&&** trees**[**indx**,** i**].**Flag**)**

s **=** "1"**;**

**if** **(**tmp**.**Comparer**(**trees**[**indx**,** i**])** **&&** **!**trees**[**indx**,** i**].**Flag**)**//опускаемся на уровень ниже

**{**

tmp **=** trees**[**indx**,** i**];**

**if** **(**flag**)**

**{**

str **=** str**.**Remove**(**str**.**Length **-** \_ival**)** **+** s**;**

**}**

**else** str **+=** s**;**

s **=** "0"**;**

flag **=** **false;**

indx**--;**

**if** **(**indx **<** 0**)**

**{**

//достигли дна, возвращаем элемент

**if** **(!**tmp**.**Flag**)**

**{**

tmp**.**Flag **=** **true;**

**return** **new** KeyValuePair**<**Tree**,** string**>(**tmp**,** str**);**

**}**

//иначе надо подниматься

UpPars**(**trees**,** **out** tmp**,** **out** indx**,** indx**,** tmp**);**

**}**

i **=** 0**;**

**}**

**else** i**++;**

**}**

**if** **(**i **==** Tree**.**Length**)**

**{**

tmp**.**Flag **=** **true;**

//иначе надо подниматься

UpPars**(**trees**,** **out** tmp**,** **out** indx**,** indx **+** 1**,** tmp**);**

i **=** 0**;**

flag **=** **true;**

**}**

**}**

**return** **new** KeyValuePair**<**Tree**,** string**>();**

**}**

**private** static int \_ival **=** 0**;**

/// <summary>

/// Найти развилку//поднятся

/// </summary>

/// <param name="trees"></param>

/// <param name="tmp"></param>

/// <param name="indx"></param>

**private** static void UpPars**(**Tree**[,]** trees**,** **out** Tree tmp**,** **out** int indx**,** int indxIn**,** Tree tmpIn**)**

**{**

\_ival **=** 0**;**

**if** **(**indxIn **<** 0**)** indxIn **=** 0**;**

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

**{**

**if** **(**trees**[**indxIn**,** i**]** **==** **null)**

**{**

indxIn**++;**

i **=** **-**1**;**

**}**

**if** **(**trees**[**indxIn**,** i**].**Equals**(**tmpIn**))**

**{**

trees**[**indxIn**,** i**].**Flag **=** **true;**

tmpIn **=** trees**[**indxIn**,** i**];**

indxIn**++;**

i **=** **-**1**;**

**}**

**else**

**{**

**if** **(**tmpIn**.**Compare**(**trees**[**indxIn**,** i**]))**

**{**

tmpIn **=** trees**[**indxIn**,** i**];**

\_ival**++;**

**if** **(!**tmpIn**.**Flag**)**

**{**

//tmpIn.Flag = true;

**break;**

**}**

indxIn**++;**

i **=** **-**1**;**

**}**

**}**

**}**

tmp **=** tmpIn**;**

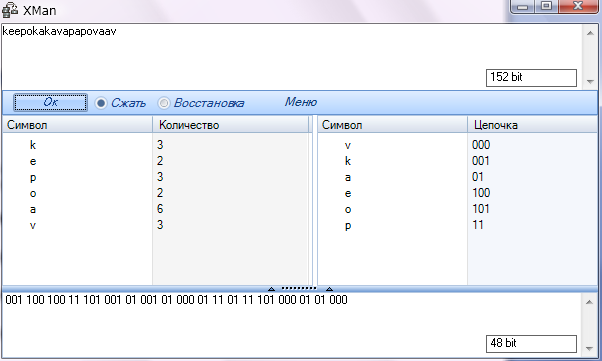
indx **=** indxIn **-** 1**;**

**}**

***Выбор языка программирования***

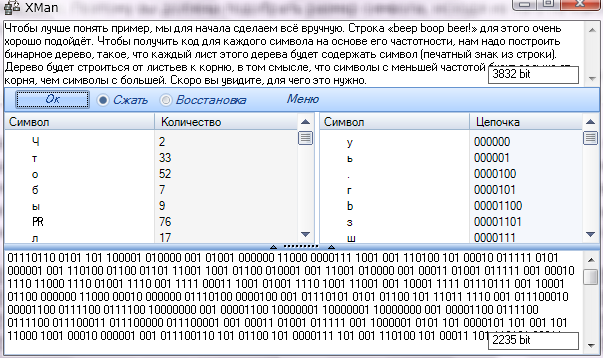
Язык C#, самый легкий и больше всего изученный мною.

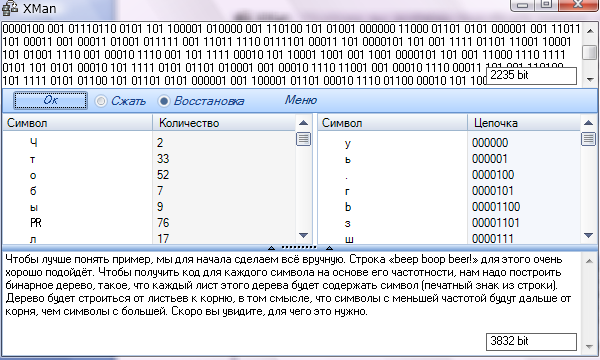
***Тестирование программы***

Рассмотрим пример, который был разобран выше:

Для строки «keepokakavapapovaav», в её текущем виде, на каждый знак тратится по одному байту. Это означает, что вся строка целиком занимает 19\*8 = 152 бит памяти. После кодирования строка займёт 48 бит (на практике, в программе мы выведем на консоль последовательность из 48 нулей и единиц, представляющих собой биты кодированного текста. Чтобы получить из них настоящую строку размером 48 бит, нужно применять битовую арифметику, а это тема отдельной лабораторной).

В заключении что-нибудь подлинее:



И обратный процесс: