**Министерство образования и науки Российской Федерации**

**Федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»**

**(МГТУ им. Н.Э. Баумана)**

Факультет *Информатика и системы управления (ИУ)*

Кафедра *"Информационная безопасность" ИУ-8*

**Отчет по домашнему заданию**

**Алгоритмы и структуры данных**

**Рауткин Владимир Юрьевич**

**Группа ИУ8-53**

1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

**1.1. Постановка задачи.**

Реализовать алгоритм сжатия данных. Для проверки корректности работы алгоритма должна быть реализована функция распаковки.

**1.2. Описание известной задачи**

Передача данных по сети интернет приобретает все большую популярность, магнитные носители, cd, dvd диски сходят на нет. Но появляется одна проблема, скорость передачи данных растет гораздо медленнее, чем размеры передаваемых файлов. Объем современных жестких дисков достаточно большой, но ограниченный, и не всегда его хватает. Удаление временно ненужных файлов и программ не есть выход из проблемы. Решением данных проблем будет сжатие файлов. Нам нужен простой и удобный архиватор.

В этой области существует несколько сильных игроков, таких как Win.rar, PowerArchiver. Но в наше время, каждая из программ имеет мощный функционал, который зачастую не используется простым пользователем, высокую ресурсоемкость, что не всегда положительно сказывается на работе, и достаточно сложную навигацию, в которой нужно разобраться перед использованием. Все архиваторы основаны на трех базовых алгоритмах сжатия данных, а именно алгоритм RLE (Run Length Encoding), алгоритмы KWE (KeyWord Encoding) и алгоритм Хаффмана.

Поскольку пользователь не всегда готов тратить время на передачу больших файлов и деньги на хранение большого количества информации, то возникает проблемный вопрос, как быстро уменьшить размер файлов, с возможность без потери качества вернуть исходный файл, при этом сделать весть процесс простым и понятным?

Объектом данной работы является – сжатие файла.

Предметом исследования данной работы является – алгоритм сжатия.

Целью данной работы – создать программу для сжатия данных, которая учтет выделенные мною недостатки.

Пункты для достижения цели:

1. Анализ предметной области, сравнение алгоритмов.
2. Сравнение существующих разработок, анализ их достоинств и недостатков.
3. Формулирование перечня задач, которые будет решать программа.
4. Проектирование программы.
5. Реализация и тестирование программы.

В данной работе старый алгоритм реализуется в новом свете, в виде архиватора, с использованием C#, dot.net и новыми технологиями. Стоит отметить, что это первый архиватор на C#, который использует алгоритм Хаффмана, как основное ядро.

Главной задачей программы является сжатие информации, что в свою очередь помогает ускорить передачу данных по сети, сохранить место на жестком диске.

При создании программы первым шагом была реализация алгоритм Хаффмана.

## 1.2 Возможные варианты алгоритмов

**Алгоритм RLE (Run Length Encoding).** Основная идея алгоритма RLE состоит в выявления повторяющихся последовательностей данных и замены их более простой структурой, в которой указывается код данных и коэффициент повторения. Несмотря на то, что кодер RLE, как правило, дает очень незначительное сжатие, он может работать очень быстро. А скорость работы декодера RLE вообще близка к скорости простого копирования блока информации. Алгоритм. К положительным сторонам алгоритма, можно отнести то, что он не требует дополнительной памяти при работе, и быстро выполняется. Алгоритм применяется в форматах РСХ, TIFF, ВМР. Интересная особенность группового кодирования в PCX заключается в том, что степень архивации для некоторых изображений может быть существенно повышена всего лишь за счет изменения порядка цветов в палитре изображения.

**Алгоритмы группы KWE(KeyWord Encoding).** В основе алгоритма сжатия по ключевым словам положен принцип кодирования лексических единиц группами байт фиксированной длины. Примером лексической единицы может быть обычное слово. На практике, на роль лексических единиц выбираются повторяющиеся последовательности символов, которые кодируются цепочкой символов (кодом) меньшей длины. Результат кодирования помещается в таблице, образовывая так называемый словарь. Алгоритмы сжатия этой группы наиболее эффективны для текстовых данных больших объемов и малоэффективны для файлов маленьких размеров (за счет необходимости сохранение словаря).

**Алгоритм Хаффмана.** В основе алгоритма Хаффмана лежит идея кодирования битовыми группами. Сначала проводится частотный анализ входной последовательности данных, то есть устанавливается частота вхождения каждого символа, встречающегося в ней. После этого, символы сортируются по уменьшению частоты вхождения.

Основная идея состоит в следующем: чем чаще встречается символ, тем меньшим количеством бит он кодируется. Результат кодирования заносится в словарь, необходимый для декодирования.

Составим небольшую сравнительную таблицу 2.1 и выделим плюсы и минусы:

Табл. 2.1 Сравнение алгоритмов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Название | Выходная структура | Сфера применения | Явные плюсы | Явные минусы |
| RLE | Список | Графические данные | 1. Не требует дополнительной памяти при работе  2. Эффективность не зависит от объема  3. Высокая скорость работы | 1.Ограниченный круг применения.  2. Низкая степень сжатия |
| KWE | Таблица словаря | Текстовые данные | 1. Очень эффективен при работе с текстами  2. Хорошая скорость работы. | 1.Ограниченный круг применения  2. Малоэффективны при работе с маленькими объемами данных |
| Алгоритм Хаффмана | Дерево кодировки | Любые данные | 1.Универсальность  2. Не увеличивает объем при неудаче (не считая таблицы) | 1. Эффективен для большого объема данных.  2. Двух шаговая обработка |

## 2.1 Алгоритм Хаффмана

Поскольку архиватор направлен на широкую аудиторию, то универсальность будет ключевой особенностью при выборе алгоритма для решения поставленной задачи. Алгоритм Хаффмана, является наиболее универсальным и легкореализуемым, а так же не увеличит размер сжатия, не считая таблицы, в случае неудачного сжатия. А значит он и ляжет в основу будущей программы. Поэтому подробнее об алгоритме Хаффмана: Алгоритм Хаффмана — адаптивный жадный алгоритм оптимального префиксного кодирования алфавита с минимальной избыточностью. Был разработан в 1952 году аспирантом Массачусетского технологического института Дэвидом Хаффманом при написании им курсовой работы. В настоящее время используется во многих программах сжатия данных.

Классический алгоритм Хаффмана на входе получает таблицу частот встречаемости символов в сообщении. Далее на основании этой таблицы строится дерево кодирования Хаффмана (Н-дерево) [3].

Рассмотрим этот процесс детально по шагам:

1. Получение входных данных, дальнейшее преобразование в массив.
2. Подсчет встречаемости символов (частоты).
3. Составление таблицы частот.
4. Постарение дерева кодирование.
5. Обход дерева и привязывание битов к символам таким образом, чтобы наиболее встречаемые символы получили минимальный набор бит.
6. Сохранения таблицы и данных в файл.

Визуально работу алгоритма можно изобразить так, рисунок 2.2.1:

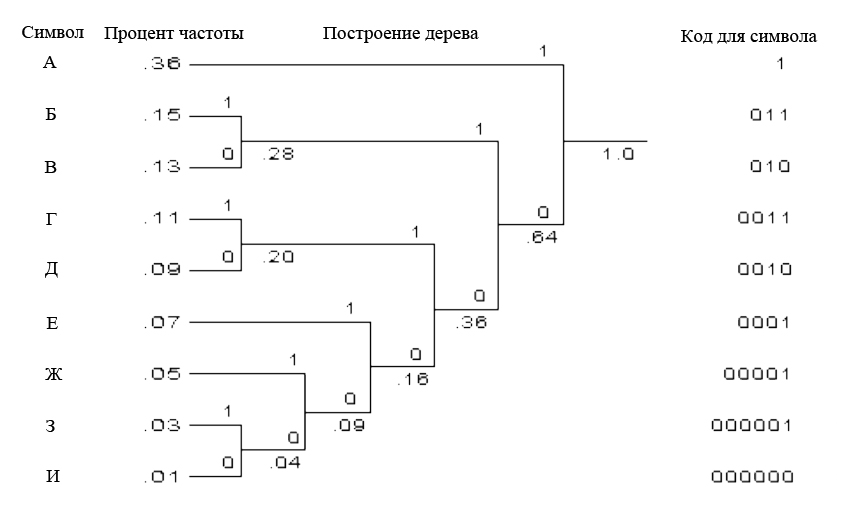


Рис. 1. Изображение алгоритма Хаффмана

В программе весь алгоритм будет представлять из себя три основных блока:

1. Основной блок, получающий, выводящий данные.

В его задачи будет входить получение данных, преобразование их в массив, передача в блок построения таблицы частот, получение данных от блока построения дерева кодирования, вывод данных.

1. Блок построения таблицы частот.

У этого блока ключевой задачей будет построение таблицы частот и передача вышеуказанной в блок построения дерева.

1. Блок построения дерева кодирования.

Основной задачей данного блока будет построение дерева из таблицы частот, присваивание символам байтового обозначения и передача преобразованных данных в основной блок.

**2.2 Создание алгоритмов и написание кода**

Создадим два файла в формате \*.cs, первый будет Tree.cs, класс в котором разместим алгоритм по построению дерева и таблицы, а так же функцию кодирования и декодирования. Второй NodalPoint.cs, в котором опишем класс узловых точек, нужный для построения дерева и создания листов такого класса.

Теперь подробнее о добавленных нами классах.

Tree.cs будет содержать несколько функций:

Build, на вход подаем строку для сжатия, и получаем таблицу и дерево. Для построения таблицы и хранения таблицы мы воспользуемся стандартным классом словарь, который хранит в себе информацию любых заданных типов в виде ключ-значение. Ниже предоставлен отрывок кода с инициализацией и циклом заполнения:

if (key == true)//При новом файле

{

for (int i = 0; i < input.Length; i++)//подсчет символов

{

if (!Frequency.ContainsKey(input[i]))//Проверка на наличие в словаре

{

Frequency.Add(input[i], 0);//Добавление в словарь

}

Frequency[input[i]]++;//Прибавляем показатель частоты встречаемости

}

Далее показан отрывок кода с алгоритмом построения дерева, а именно использование класса orderNodesOfFreq, в котором содержатся отсортированные по возрастанию частоты узловые точки, из которого берем два узла, и из них создаем родительский, в котором заполняем левый и правый лист дерева, и добавляем вместо двух взятых.

while (nodes.Count > 1)//Строим дерево, пока узлов больше 1

{

List<NodalPoints> orderedNodes = nodes.OrderBy(node => node.Frequency).ToList<NodalPoints>();

if (orderedNodes.Count >= 2)//Пока узлов больше 2

{

List<Node> taken = orderedNodes.Take(2).ToList<NodalPoints>();//Взять два пункта и создать узел

NodalPoints parent = new NodalPoints()//Узел родитель

{

Symbol = '\*', Frequency = take[0].Frequency + take[1].Frequency, Left = take[0], Right = take[1]

};

nodes.Remove(take[0]);//Удаляем первый взятый элемент

nodes.Remove(take[1]);//Удаляем второй взятый элемент

nodes.Add(parent);//Добавляем новый элемент, ставший корнем дерева предыдущих двух

}

this.Root = nodes.FirstOrDefault();//Вернем прежний корень

}

Encode, на вход подаем строку для сжатия, и получаем последовательность бит, которую будем хранить в стандартном классе BitArray, в виде True и False.

public BitArray Encode(string source, Form1 frm)

{

List<bool> encodedSource = new List<bool>();//Заготовка для массива бит

();

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

{

List<bool> encodedSymbol = this.Root.ByPassTree(source[i], new List<bool>());

encodedSource.AddRange(encodedSymbol);//Добавляем бит в виде True/False

}

BitArray bits = new BitArray(encodedSource.ToArray());//Преобразования листа в одномерный массив-класс BitaArray

return bits;

}

Decode, на вход подаем последовательность байт и получаем раскодированный текст. Для работы требуется дерево, которое будет воссоздаваться из таблицы.

Node.cs будет содержать четыре поля, имя, частоту, правую и левую ветвь.

Для сохранения таблицы и кодированного файла мы воспользуемся бинарной сериализацией, сохраняя все в выбранный файл. При восстановлении, для предотвращения ошибок закодированный файл будет считываться всего один раз, т.е. считывание таблицы, работа с таблицей, считывание закодированной последовательности, и только после этого, закрытие потока.

FileStream f1 = new FileStream(adress, FileMode.Create);

BinaryFormatter bf = new BinaryFormatter();

bf.Serialize(f1, Frequency); // сохранение объекта в потоке f

f1.Close();

Поскольку программа является архиватором, следующей информацией будет степень сжатия, вес до и после, получение которой написано ниже:

long fileMas = FId.Length;

FileInfo FIp = new FileInfo(outputFile);

long fileMas1 = FIp.Length;

long percent = 100 - fileMas1 \* 100 / fileMas;

**2.3 Тест работы архиватора**

Тестирование на текстовом файле, заполненным рандомной последовательностью 0 и 1.

На вход программе задаются параметры: -zip (архивация)/-unzip (разархивация) /путь/до/файла /путь/для/сохранения.

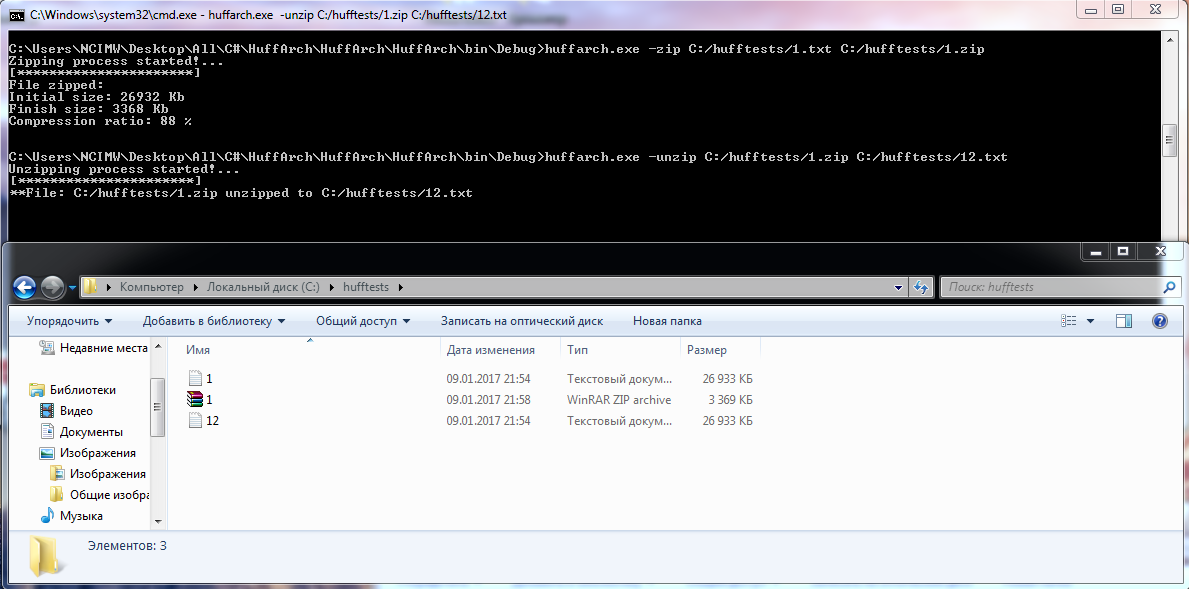


Рис.2 Тест файла размером ~27мб

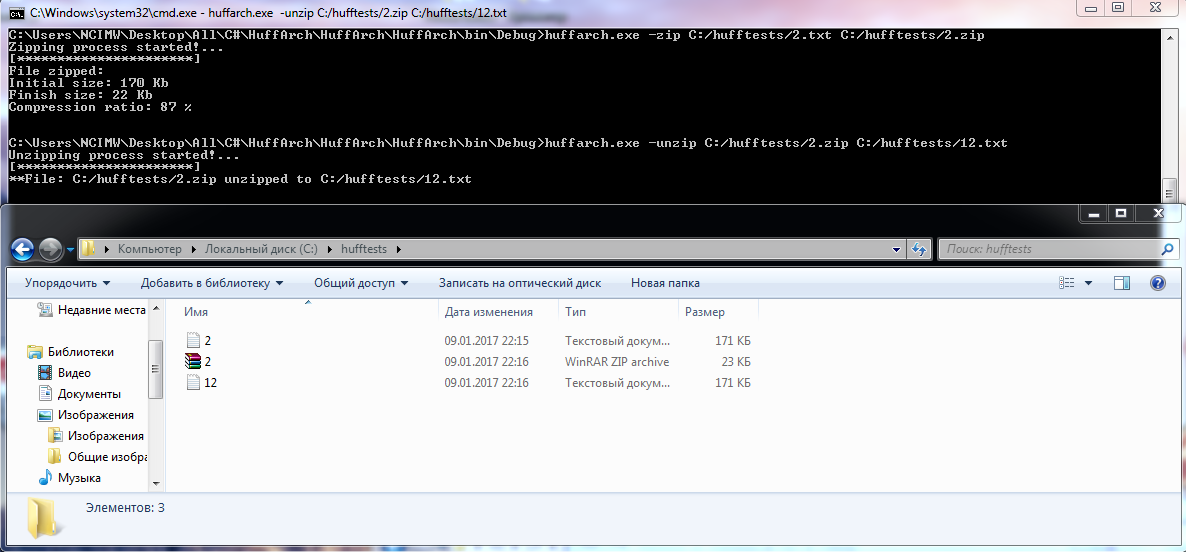


Рис.3 Тест файла размером 171кб

**3.1 Сложность реального алгоритма**

Принимая за n общее количество символов в кодируемом файле:

* Построение/восстановление таблицы и листа с узлами => О(n)
* Кодирование массива символов => O(n)
* Декодирование => O(n)

Итоговая сложность по времени O(n+n+n) = O(n).

Сложность по памяти: принимая объем памяти занимаемой структурой Cell за 1 единицу, в процессе генерации алгоритма расходуется n единиц дополнительной памяти на хранение массива символов входного файла + n единиц на хранение таблицы + константное количество памяти на дополнительные расчетные переменные => О(n).