МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОСИЙСКОЙ ФЕДЕРАЦИИ  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ   
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «ТЮМЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»  
Институт математики и компьютерных наук  
Кафедра программного обеспечения

Курсовая работа

по дисциплине

«Структуры и алгоритмы компьютерной обработке данных»

на тему: «Сравнение алгоритмов сжатия данных без потерь»

**Выполнил**:

Студент группы 174-1

Соловьев А.Н.

**Проверил**:

Ассистент кафедры программного обеспечения Ромазанов А.Р.

Содержание

[Введение 3](#_Toc533444480)  
[1. Постановка задачи 4](#_Toc533444481)  
[2. Теория 5](#_Toc533444482)  
[2.1. Обзор алгоритмов 6](#_Toc533444483)  
[2.1.1. RLE 6](#_Toc533444484)  
[2.1.2. Huffman 6](#_Toc533444485)  
[2.1.3. LZMA 7](#_Toc533444486)  
[2.1.4. Deflate 9](#_Toc533444487)[3. Описание программы 11](#_Toc533444488)[4. Конфигурации компьютера 14](#_Toc533444489)[5. Результаты эксперимента 15](#_Toc533444490)[Заключение 22](#_Toc533444491)[Список литературы 23](#_Toc533444492)[Приложения 24](#_Toc533444493)

Введение

Сжатие информации - проблема, имеющая достаточно давнюю историю, гораздо более давнюю, нежели история развития вычислительной техники, которая (история) обычно шла параллельно с историей развития проблемы кодирования и шифровки информации. Целью процесса сжатия, как правило, есть получение более компактного выходного потока информационных единиц из некоторого изначально некомпактного входного потока при помощи некоторого их преобразования.

1. Постановка задачи

Рассмотреть 4 алгоритма сжатия данных: RLE, LZMA, Deflate, Huffman. Изучить принципы работы данных алгоритмов. Сравнить алгоритмы по эффективности сжатия, а также по времени сжатия и распаковки. Написать программу на языке Python для сжатия файлов различного типа с помощью программ на языке C, а также анализа данных, получаемых при сжатии файлов различного типа. Программа должна сжимать и распаковывать файлы, сохранять результаты анализа сжатия в таблицу формата CSV, а также выводить диаграммы для наглядной демонстрации эффективности уменьшения размера файлов и сравнения времени работы алгоритмов. Сделать выводы, основываясь на полученных данных.

2. Теория

Сжатие данных — алгоритмическое преобразование данных, производимое с целью уменьшения занимаемого ими объёма. Применяется для более рационального использования устройств хранения и передачи данных. Синонимы — упаковка данных, компрессия, сжимающее кодирование, кодирование источника. Обратная процедура называется восстановлением данных (распаковкой, декомпрессией).

Сжатие основано на устранении избыточности, содержащейся в исходных данных. Простейшим примером избыточности является повторение в тексте фрагментов (например, слов естественного или машинного языка). Подобная избыточность обычно устраняется заменой повторяющейся последовательности ссылкой на уже закодированный фрагмент с указанием его длины. Другой вид избыточности связан с тем, что некоторые значения в сжимаемых данных встречаются чаще других. Сокращение объёма данных достигается за счёт замены часто встречающихся данных короткими кодовыми словами, а редких — длинными (энтропийное кодирование). Сжатие данных, не обладающих свойством избыточности (например, случайный сигнал или белый шум, зашифрованные сообщения), принципиально невозможно без потерь.

Сжатие данных без потерь (англ. lossless data compression) — метод сжатия данных (видео, аудио, графики, документов, представленных в цифровом виде), при использовании которого закодированные данные однозначно могут быть восстановлены с точностью до бита, пикселя, вокселя и т.д. При этом оригинальные данные полностью восстанавливаются из сжатого состояния. Этот тип сжатия принципиально отличается от сжатия данных с потерями. Для каждого из типов цифровой информации, как правило, существуют свои оптимальные алгоритмы сжатия без потерь.

Отношение сжатия – Величина, равная отношению размеров файла до и после сжатия, где  **–** размер файла после сжатия, – размер исходного файла.

# **2.1. Обзор алгоритмов**

## 2.1.1. RLE

Кодирование длин серий (run-length encoding, RLE) — алгоритм сжатия данных, заменяющий повторяющиеся символы (серии) на один символ и число его повторов. Серией называется последовательность, состоящая из нескольких одинаковых символов. При кодировании (упаковке, сжатии) строка одинаковых символов, составляющих серию, заменяется строкой, содержащей сам повторяющийся символ и количество его повторов. [1]

Например, строку «АААААAAAFBBBCC», требующую для хранения 14 байт (при условии, что на хранение одного символа отводится байт), можно заменить на «5А1F3B2C», состоящую из 8 байт. Очевидно, что этот алгоритм тем эффективнее, чем длиннее серия повторов.

Основным недостатком этого алгоритма является его крайне низкая эффективность на последовательностях неповторяющихся символов. Например, если рассмотреть последовательность «ABABAC» (6 байт), то после применения алгоритма RLE она превратится в «1A1B1A1B1A1C» (12 байт). Для решения проблемы неповторяющихся символов существуют различные методы.

Самым простым методом является следующая модификация: байт, кодирующий количество повторов, должен хранить информацию не только о количестве повторов, но и об их наличии. Если первый бит равен 1, то следующие 7 бит указывают количество повторов соответствующего символа, а если первый бит равен 0, то следующие 7 бит показывают количество символов, которые надо взять без повтора. Если закодировать «ABABAC» с использованием данной модификации, то получим «-6ABABAC» (7 байт).

## 2.1.2. Huffman

Один из первых алгоритмов эффективного кодирования информации был предложен Д. А. Хаффманом в 1952 году. Идея алгоритма состоит в следующем: зная вероятности символов в сообщении, можно описать процедуру построения кодов переменной длины, состоящих из целого количества битов. Символам с большей вероятностью ставятся в соответствие более короткие коды. Коды Хаффмана обладают свойством префиксности (то есть ни одно кодовое слово не является префиксом другого), что позволяет однозначно их декодировать. [5]

Для получения кодов Хаффмана используют следующий алгоритм:

1. Все символы алфавита представляются в виде свободных узлов, при этом вес узла пропорционален частоте символа в сообщении;
2. Из множества свободных узлов выбираются два узла с минимальным весом и создаётся новый (родительский) узел с весом, равным сумме весов выбранных узлов;
3. Выбранные узлы удаляются из списка свободных, а созданный на их основе родительский узел добавляется в этот список;
4. Шаги 2-3 повторяются до тех пор, пока в списке свободных больше одного узла;
5. На основе построенного дерева каждому символу алфавита присваивается префиксный код;
6. Сообщение кодируется полученными кодами.

Среди недостатков алгоритма Хаффмана значительную часть составляют проблемы, связанные со сложностью реализации. Использование для хранения частот символов вещественных переменных сопряжено с потерей точности, поэтому на практике часто используют целочисленные переменные, но, т.к. вес родительских узлов постоянно растёт, рано или поздно возникает переполнение. Таким образом, несмотря на простоту алгоритма, его корректная реализация до сих пор может вызывать некоторые затруднения, особенно для больших алфавитов. [1]

## 2.1.3. LZMA

LZMA (англ. Lempel-Ziv-Markov chain-Algorithm) — алгоритм сжатия данных, разрабатываемый с 2001 года. Используется в архиваторе 7-Zip для создания сжатых архивов в формате 7z. Алгоритм основан на схеме сжатия данных по словарю, сходной с использованной в LZ77, и обеспечивает высокий коэффициент сжатия (обычно превышающий коэффициент, получаемый при сжатии с использованием bzip2), а также позволяет использовать словари различного размера (до 4 Гб). [5]

Как говорилось выше, алгоритм основывается на методе, принадлежащему Лемпелю и Зиву, семейство алгоритмов, использующих данных метод обычно называется [LZ](http://www.compression.ru/arctest/descript/lzwcomp.htm)\* семейством. [4]

Суть данного метода состоит в следующем: упаковщик постоянно хранит некоторое количество последних обработанных символов в буфере. По мере обработки входного потока вновь поступившие символы попадают в конец буфера, сдвигая предшествующие символы и вытесняя самые старые. Размеры этого буфера, называемого также скользящим словарем (sliding dictionary), варьируются в разных реализациях кодирующих систем. [4]

Затем, после построения хэш-таблиц алгоритм выделяет (путем поиска в словаре) самую длинную начальную подстроку входного потока, совпадающую с одной из подстрок в словаре, и выдает на выход пару (length, distance), где length - длина найденной в словаре подстроки, а distance - расстояние от нее до входной подстроки (то есть фактически индекс подстроки в буфере, вычтенный из его размера). В случае, если такая подстрока не найдена, в выходной поток просто копируется очередной символ входного потока.

В первоначальной версии алгоритма предлагалось использовать простейший поиск по всему словарю. Время сжатия при такой реализации было пропорционально произведению длины входного потока на размер буфера, что совсем непригодно для практического использования. Однако, в дальнейшем, было предложено использовать двоичное дерево и хеширование для быстрого поиска в словаре, что позволило на порядок поднять скорость работы алгоритма.

Таким образом, алгоритм Лемпеля-Зива преобразует один поток исходных символов в два параллельных потока длин и индексов в таблице (length + distance). Очевидно, что эти потоки являются потоками символов с двумя новыми алфавитами, и к ним можно применить один из методов (RLE, кодирование Хаффмэна или арифметическое кодирование).

Так мы приходим к схеме двухступенчатого кодирования - наиболее эффективной из практически используемых в настоящее время. При реализации этого метода необходимо добиться согласованного вывода обоих потоков в один файл. Эта проблема обычно решается путем поочередной записи кодов символов из обоих потоков.

## 2.1.4. Deflate

Deflate — это алгоритм сжатия без потерь, использующий комбинацию алгоритмов LZ77 и Хаффмана. Изначально был описан Филом Кацем для второй версии его архиватора PKZIP, который впоследствии был определён в RFC 1951 (1996 год). [2]

Deflate считается свободным от всех существующих патентов, и пока оставался в силе патент на LZW (он применяется в формате GIF), это привело к использованию Deflate не только в формате ZIP, для которого Кац изначально его спроектировал, но также в компрессоре/декомпрессоре gzip и в PNG-изображениях.

Deflate-поток содержит серии блоков. Перед каждым блоком находится трёхбитовый заголовок:

Один бит: флаг последнего блока.

1: блок последний.

0: блок не последний.

Два бита: метод, с помощью которого были закодированы данные.

00: данные не закодированы (в блоке находятся непосредственно выходные данные).

01: данные закодированы по методу статического Хаффмана.

10: данные закодированы по методу динамического Хаффмана.

11: зарезервированное значение (ошибка).

Каждый блок сжимается с использованием комбинации алгоритма LZ77 и кодирования Хаффмана. Деревья Хаффмана для каждого блока независимы от деревьев для предыдущих или последующих блоков; алгоритм LZ77 может использовать ссылку на дублированную строку, встречающуюся в предыдущем блоке, вплоть до 32K байтах раньше.

Каждый блок состоит из двух частей: пары деревьев кода Хаффмана, которые описывают представление сжатой части данных и саму сжатую часть данных. (Деревья Хаффмана сами сжаты используя кодирование Хаффмана.) Сжатые данные состоят из серии элементов двух типов: буквенные байты (строк, которые не были определяется как дублированный в предыдущих 32K входных байтах), и указатели на дублированные строки, где указатель представлен в виде пара <длина, обратное расстояние>. Представление, используемое в формат "deflate" ограничивает расстояния до 32 Кбайт и длину до 258 байт, но не ограничивает размер блока, за исключением несжимаемых блоков, которые ограничены, как отмечено выше. [2]

**3. Описание программы**

Программа написана на языке Python 3.5 и использует библиотеки NumPy, Pandas, Matplotlib.[3] Полный список используемых модулей и их версий находится в текстовом файле “reqs.txt”.

Для алгоритмов LZMA и Deflate, используются встроенные библиотеки python lzma и gzip соответственно. Данные библиотеки являются интерфейсами для запуска программ, написанных на языке C (liblzma и zlib). Для алгоритмов RLE и Huffman также используются программы, написанные на языке C, исходный код которых находится в директории “C” в папках с названиями “rle” и “huffman” соответственно. Скомпилированные динамические библиотеки, необходимые для запуска из Python данных алгоритмов находятся в корневой папке проекта под названиями “librle.dylib” и “libhuffmun.dylib”.

Сама программа курсовой, находится в файле “analyze.py”.

Описание основных функций:

analyze\_directory(directory, times)

directory – директория для анализа, times – количество запусков сжатия/распаковки для каждого файла.

analyze\_average\_directory(directory, times)

directory – директория для анализа, times – количество запусков сжатия/распаковки для каждого файла.

analyze\_directories\_average(dirs)

dirs – список директорий

Запуск программы из консоли

Для запуска программы необходимо иметь интерпретатор языка python и все необходимые библиотеки (из файла reqs.txt) установленными на компьютере, либо предварительно активировать виртуальную среду в папках venv и venv\_win.

В консоли перейти в директорию с файлом analyze.py и выполнить команду:

python analyze.py <режим> <папка1> <папка2> … <папкаN>

<режим> - “average” или отсутствует. Если режим отсутствует, то проводится анализ для каждого отдельного файла в директории. Если введен режим average, то в результате приводятся данные для всей директории, путем вычисления среднего размера файла до и после сжатия, среднего отношения, процента и времени сжатия.

<папка> - указывает имя директории для которой необходимо провести анализ, если директории не существует, то она пропускается. Директории с файлами необходимо поместить в директорию “FILES” в директории проекта (“Курсовая/FILES”).

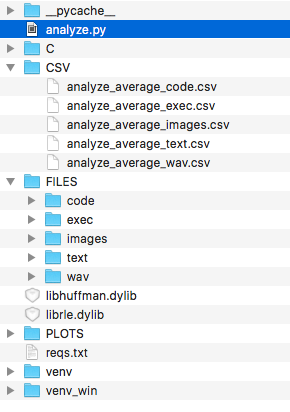


Рисунок 1 - Структура проекта программы

Результат работы программы

В результате запуска программы создадутся таблицы в формате csv в папке “CSV”. Таблицы имеют название – “analyze\_<dir>.csv” или “analyze\_<dir>\_average.csv” в зависимости от режима, где dir – название директории для которой запускалась программа.

А также выводятся диаграммы со сравнением полученных данных.

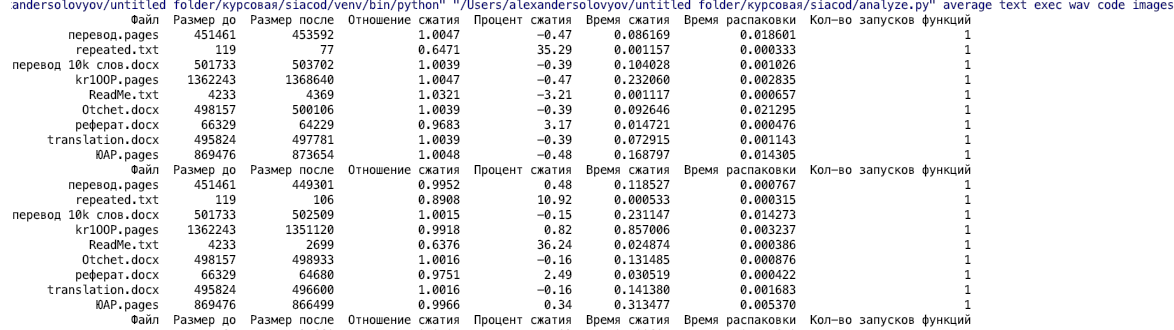
****

Рисунок 2 - Скриншот запуска программы

**4. Конфигурации компьютера**

Компьютер на котором проводился эксперимент:

|  |  |
| --- | --- |
| Процессор | Intel Core I7 6700K на частоте 4.5GHz |
| Память | DDR4 16 GB на частоте 2400 MHz |
| Операционная система | OS X 10.11.6 |
| Разрядность системы | 64 |
| Версии языка и компилятора | Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)  [GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on Darwin |

**5. Результаты эксперимента**

Данные на которых проводилось сравнение приведены в директории “FILES”. В директории “FILES” приведены следующие директории: “code ”, “exec”, “images”, “text”, “wav”.

Описание файлов в данных директориях приведены в таблице ниже:

|  |  |  |
| --- | --- | --- |
| Директория | Описание | Количество файлов |
| text | Разные текстовые файлы в форматах: “txt”, “docx”, “pages” | 9 |
| images | Графические файлы в формате JPEG | 21 |
| exec | Исполняемые файлы windows в формате EXE | 8 |
| code | Исходные файлы программ на Python, C#, Javascript. Форматы: “py”, “cs”, “js”. | 22 |
| wav | Звуковые файлы в формате WAV | 44 |

Данные из CSV файлов, полученных в результате работы программы:

Таблица 1

Данные анализа директории code

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Алгоритм | Размер до | Размер после | Отношение сжатия | Процент сжатия | Время сжатия | Время распаковки | Кол-во запусков функций |
| 0 | RLE | 23533.0 | 20013.0 | 0.886 | 11.42 | 0.0015 | 0.0001 | 10 |
| 1 | Huffman | 23533.0 | 13585.0 | 0.682 | 31.8 | 0.0023 | 0.0002 | 10 |
| 2 | LZMA | 23533.0 | 4574.0 | 0.345 | 65.51 | 0.0171 | 0.0005 | 10 |
| 3 | DEFLATE | 23533.0 | 4965.0 | 0.32 | 67.97 | 0.002 | 0.0002 | 10 |

Таблица 2

Данные анализа директории exec

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Алгоритм | Размер до | Размер после | Отношение сжатия | Процент сжатия | Время сжатия | Время распаковки | Кол-во запусков функций |
| 0 | RLE | 258625.0 | 254276.0 | 0.872 | 12.75 | 0.0153 | 0.001 | 10 |
| 1 | Huffman | 258625.0 | 251045.0 | 0.766 | 23.44 | 0.0253 | 0.0011 | 10 |
| 2 | LZMA | 258625.0 | 217634.0 | 0.516 | 48.42 | 0.0997 | 0.0046 | 10 |
| 3 | DEFLATE | 258625.0 | 221606.0 | 0.54 | 45.99 | 0.0085 | 0.0014 | 10 |

Таблица 3

Данные анализа директории images

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Алгоритм | Размер до | Размер после | Отношение сжатия | Процент сжатия | Время сжатия | Время распаковки | Кол-во запусков функций |
| 0 | RLE | 29103.0 | 28973.0 | 0.999 | 0.11 | 0.0019 | 0.0002 | 10 |
| 1 | Huffman | 29103.0 | 29550.0 | 1.025 | -2.52 | 0.0042 | 0.0003 | 10 |
| 2 | LZMA | 29103.0 | 28043.0 | 0.962 | 3.75 | 0.0146 | 0.0023 | 10 |
| 3 | DEFLATE | 29103.0 | 27966.0 | 0.962 | 3.81 | 0.0007 | 0.0004 | 10 |

Таблица 4

Данные анализа директории text

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Алгоритм | Размер до | Размер после | Отношение сжатия | Процент сжатия | Время сжатия | Время распаковки | Кол-во запусков функций |
| 0 | RLE | 472175.0 | 474017.0 | 0.964 | 3.63 | 0.0282 | 0.0019 | 10 |
| 1 | Huffman | 472175.0 | 470272.0 | 0.944 | 5.65 | 0.0459 | 0.0018 | 10 |
| 2 | LZMA | 472175.0 | 446997.0 | 0.895 | 10.47 | 0.1678 | 0.0265 | 10 |
| 3 | DEFLATE | 472175.0 | 451307.0 | 0.863 | 13.65 | 0.0114 | 0.0033 | 10 |

Таблица 5

Данные анализа директории wav

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Алгоритм | Размер до | Размер после | Отношение сжатия | Процент сжатия | Время сжатия | Время распаковки | Кол-во запусков функций |
| 0 | RLE | 134746.0 | 125506.0 | 0.93 | 6.99 | 0.008 | 0.0006 | 10 |
| 1 | Huffman | 134746.0 | 94945.0 | 0.708 | 29.22 | 0.0127 | 0.0004 | 10 |
| 2 | LZMA | 134746.0 | 64883.0 | 0.471 | 52.89 | 0.0586 | 0.0078 | 10 |
| 3 | DEFLATE | 134746.0 | 82861.0 | 0.603 | 39.73 | 0.0071 | 0.0011 | 10 |

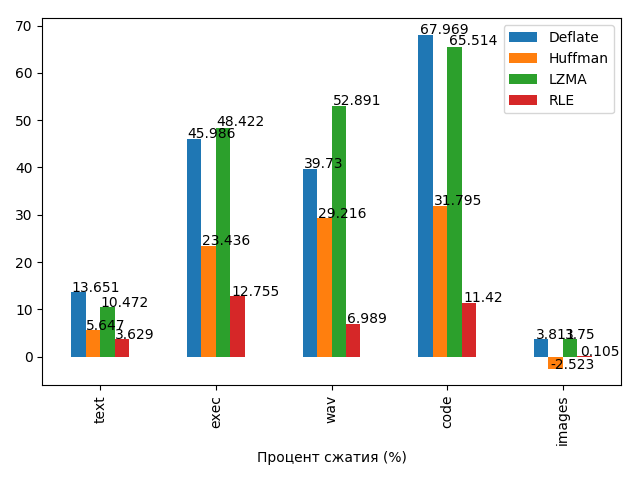
Диаграммы, полученные в результате работы программы:

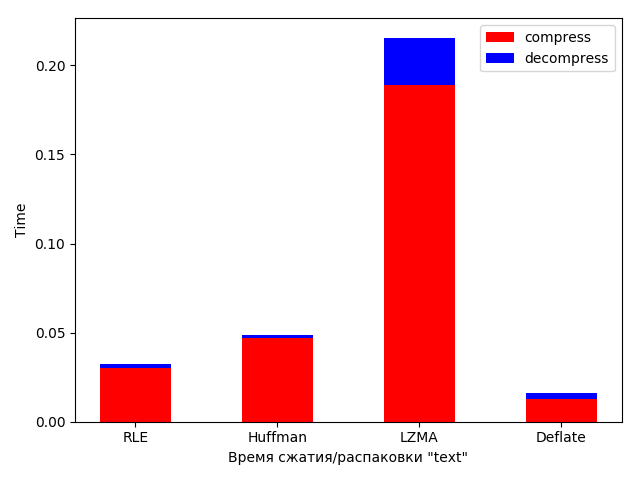
Рисунок 3 - Процент сжатия

Как видно на диаграмме, самый низкий процент сжатия все алгоритмы показали на данных из директории jpeg. Как было описано ранее, директория images содержит файлы в формате JPEG. Файлы данного формата уже являются сжатыми (причем с потерями), поэтому сжатие данных файлов малоэффективно. В случае сжатия по алгоритму Huffman средний размер файла даже увеличился.

Хороший процент сжатия был получен у файлов изначально несжатых форматов: исполняемые файлы, исходные файлы программ, текстовые и звуковые файлы в формате WAV. Самый низкий процент сжатия на представленных данных показал алгоритм RLE.

Алгоритмы LZMA и Deflate в среднем превосходят по проценту сжатия алгоритм сжатия по Хаффману в 1.5-2 раза.

Как видно, современные алгоритмы сжатия, такие как LZMA и Deflate способны сжимать файлы достаточно эффективно, так как они используют комбинацию методов, используемых в алгоритмах RLE и Huffman, а также применяют словарный метод.

Рисунок 4 - Время сжатия/распаковки директории text

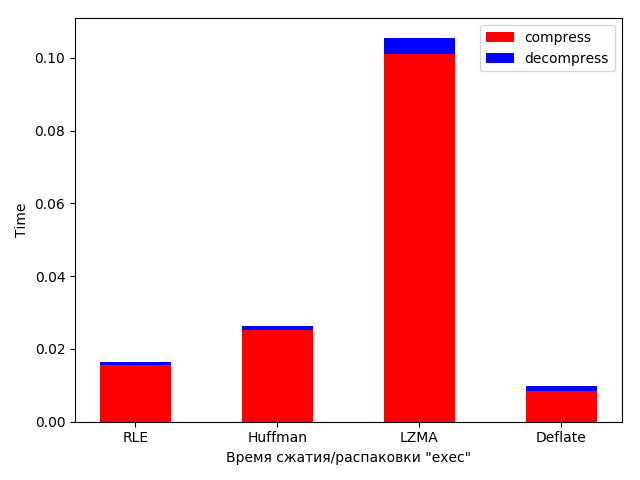


Рисунок 5 - Время сжатия/распаковки директории exec

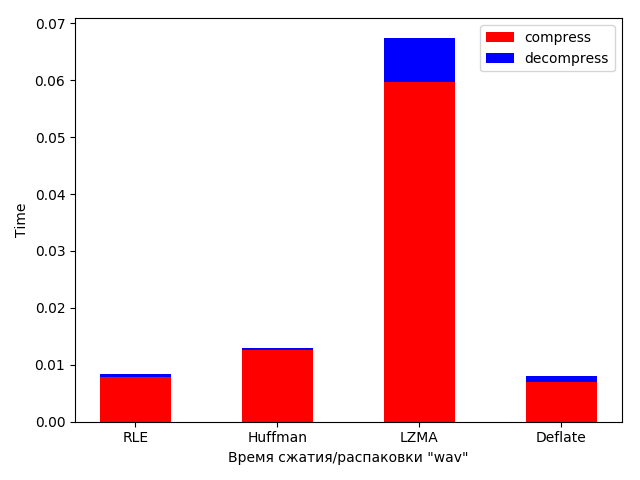


Рисунок 6 - Время сжатия/распаковки директории wav

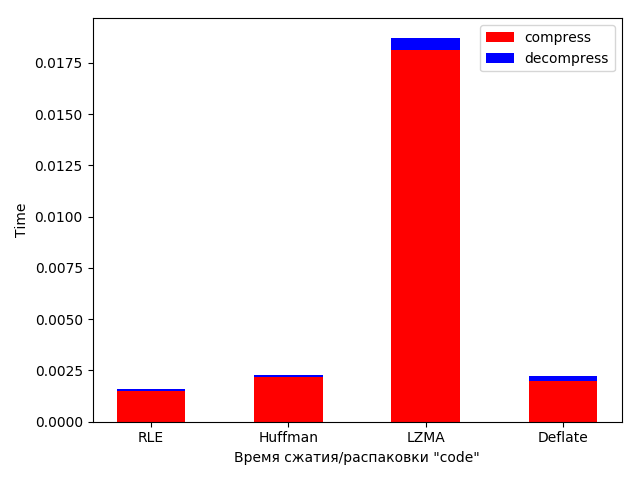


Рисунок 7 - Время сжатия/распаковки директории code

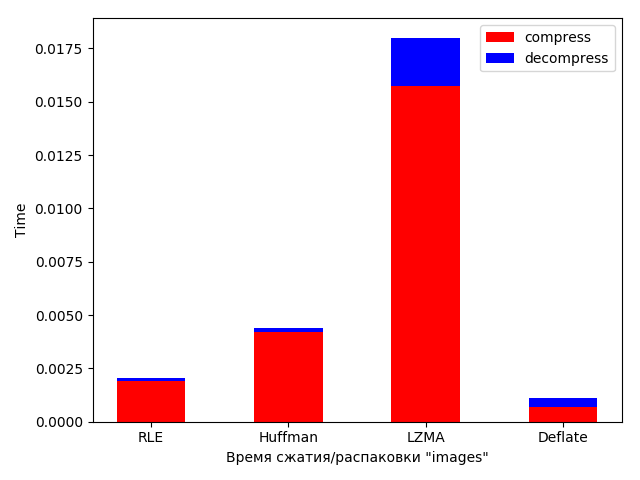


Рисунок 8 - Время сжатия/распаковки директории images

На основании полученных диаграммах можно сделать вывод, что время сжатия превышает время распаковки в разы во всех случаях. Данное заключение вполне логично, так как для сжатия данных используется несколько проходов, в отличии от распаковки (Которая происходит за одну итерацию чтения файла по определенным правилам, либо словарю в начале закодированного файла). Также можно заметить, что время работы алгоритма LZMA превосходит время работы остальных примерно в 4-6 раз, хотя как следует из полученных данных видно, что эффективность алгоритмов LZMA и Deflate находится примерно на одном уровне.

**Заключение**

В результате выполнения данной курсовой работы было изучено 4 алгоритма сжатия данных без потерь: RLE, LZMA, Deflate, Huffman. Было проведено сравнение данных алгоритмов по эффективности сжатия, а также по времени сжатия и распаковки. Также была написана программа для сжатия файлов различного типа с помощью программ на языке C, а также анализа данных, полученных при сжатии файлов различного типа.

**Список литературы**

1. Д. Ватолин, А. Ратушняк, М. Смирнов, В. Юкин. Методы сжатия данных. Устройство архиваторов, сжатие изображений и видео.
2. RFC 1951 — DEFLATE Compressed Data Format Specification version 1.3 (англ.)

[Электронный ресурс. https://tools.ietf.org/html/rfc1951.Дата обращения 19.12.2018]

1. Python data analysis toolkit (англ.) [Электронный ресурс. http://pandas.pydata.org/pandas-docs/stable/. Дата обращения 19.12.2018]
2. Jacob Ziv, Abraham Lempel. A Universal Algorithm for Sequential Data Compression IEEE Transactions on Information Theory, 23(3), pp. 337-343, May 1977. (англ.)
3. History of Lossless Data Compression Algorithms (англ.) [Электронный ресурс. https://ethw.org/History\_of\_Lossless\_Data\_Compression\_Algorithms. Дата обращения 19.12.2018]

# **Приложения**

Приложение 1

Реализация алгоритма RLE на языке C

**#include** <stdio.h>  
**#include** <limits.h>  
**#include** <errno.h>  
  
**int** RleEncodeFile(**char** \*in, **char** \*out)  
{  
 **int** currChar;  
 **int** prevChar;  
 **unsigned char** count;  
 FILE \*inFile=NULL;  
 FILE \*outFile=NULL;  
 **if**(in){inFile = fopen(in, **"rb"**);}  
 **if**(out){ outFile = fopen(out, **"wb"**);}  
 **if** ((NULL == inFile) || (NULL == outFile))  
 {  
 errno = ENOENT;  
 **return** -1;  
 }  
 prevChar = EOF;  
 count = 0;  
 **while** ((currChar = fgetc(inFile)) != EOF)  
 {  
 fputc(currChar, outFile);  
 **if** (currChar == prevChar)  
 {  
 count = 0;  
 **while** ((currChar = fgetc(inFile)) != EOF)  
 {  
 **if** (currChar == prevChar)  
 {  
 count++;  
 **if** (count == UCHAR\_MAX)  
 {  
 fputc(count, outFile);  
 prevChar = EOF;  
 **break**;  
 }  
 }  
 **else** {  
 fputc(count, outFile);  
 fputc(currChar, outFile);  
 prevChar = currChar;  
 **break**;  
 }  
 }  
 }  
 **else** {  
 prevChar = currChar;  
 }  
 **if** (currChar == EOF)  
 {  
 fputc(count, outFile);  
 **break**;  
 }  
 }  
 fclose(outFile);  
 **return** 0;  
}  
  
  
**int** RleDecodeFile(**char** \*in, **char** \*out)  
{  
 **int** currChar;  
 **int** prevChar;  
 **unsigned char** count;  
  
 FILE \*inFile=NULL;  
 FILE \*outFile=NULL;  
 **if**(in)  
 {  
 inFile = fopen(in, **"rb"**);  
 }  
  
 **if**(out)  
 {  
 outFile = fopen(out, **"wb"**);  
 }  
  
 **if** ((NULL == inFile) || (NULL == outFile))  
 {  
 errno = ENOENT;  
 **return** -1;  
 }  
  
 prevChar = EOF;  
  
 **while** ((currChar = fgetc(inFile)) != EOF)  
 {  
 fputc(currChar, outFile);  
 **if** (currChar == prevChar)  
 {  
 count = fgetc(inFile);  
 **while** (count > 0)  
 {  
 fputc(currChar, outFile);  
 count--;  
 }  
 prevChar = EOF;  
 }  
 **else** {  
 prevChar = currChar;  
 }  
 }  
 **return** 0;  
}