**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра систем автоматизированного проектирования**

отчет

**по лабораторной работе №2**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: «Реализация алгоритма JPEG»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3352 |  | Гультяев А.С. |
| Преподаватель |  | Пестерев Д.О. |

Санкт-Петербург

2025

# Введение

**Цель:** реализация алгоритма , а также его составных частей, таких как: AC/DC кодирование, квантование матриц и так далее.

**Задачи:**

1. Реализовать переход цветового пространства из в и обратно.
2. Реализовать downsampling матрицы цветового канала.
3. Осуществить разбиение изображения на блоки размерами , при этом дополнив неполные блоки некоторым значением.
4. Изучить и реализовать прямое и обратное DCT-II 2D для блока размером .
5. Осуществить изменение матрицы квантования в зависимости от уровня сжатия.
6. Зигзаг обход матрицы .
7. Изучить и реализовать разностное DC, переменное кодирование AC и DC коэффициентов.
8. Реализовать RLE кодирование AC коэффициентов.
9. Кодирование разностей DC коэффициентов и последовательностей Run/Size по таблице кодов Хаффмана и упаковки результата в байтовую строку.
10. Собрать итоговый алгоритм кодирования и декодирования .
11. Построить графики зависимости размеров зависимости размера сжатого файла от коэффициента качества сжатия с шагом 2.

# 

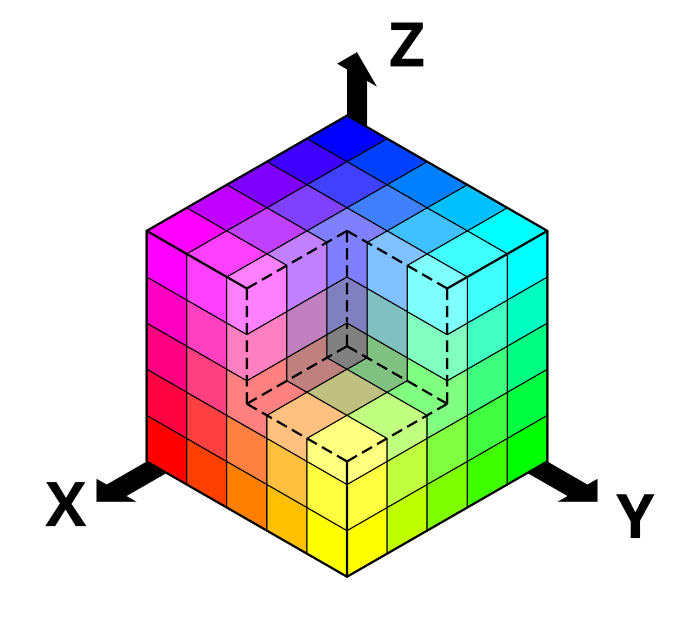
# Теоретическая часть

1. :

– аддитивная цветовая модель, которая описывает способ кодирования цвета для цветовоспроизведения с помощью трёх цветов ( – red (красный), – green (зеленый), – blue (синий)).

В самой простой реализации можно представить в виде трехмерной системы координат, где каждая из каналов/осей () начинается с 0 в общей точки и заканчивается на значении 255 (рис. 1.1). Такое представление выбрано вследствие того, что один байт принимает значение от 0 до 255. В изображениях же каждый из каналов накладываются друг на друга, и значения для каждого отдельного из каналов означают интенсивность цвета, то есть 255 – цвет максимально насыщенный, 0 – цвет прозрачный.

Рисунок 1.1 – Простейшее представление



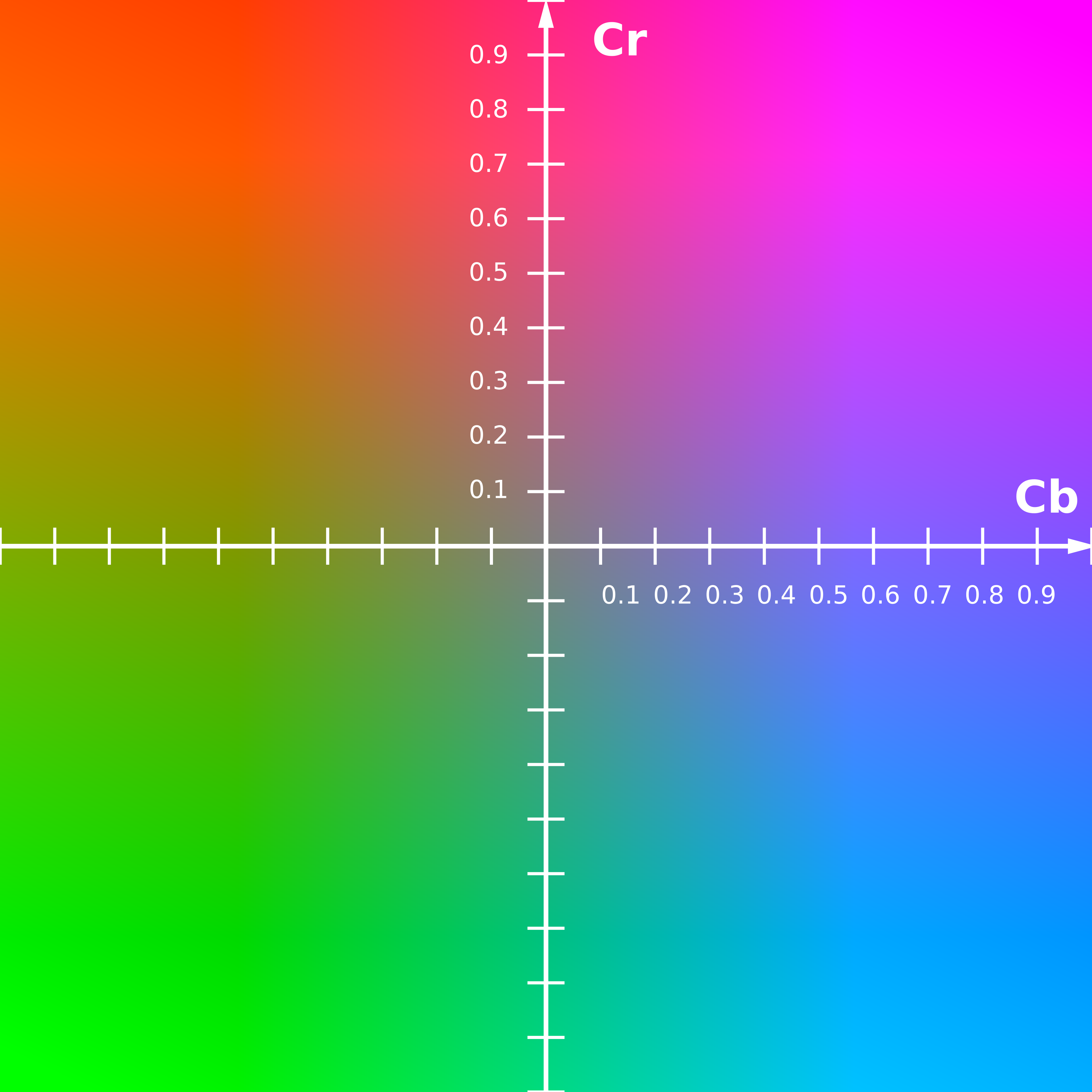
В подавляющем большинстве, будь то компьютеры, телефоны или любая другая техника, в которой используется графическое отображение, используется немного иная запись. Так, например, для более точной передачи используется шестнадцатеричное отображение формата #RRGGBB, где каждое из значений может принимать одно из цифр шестнадцатеричной системы – от 0 до F. В такой реализации первые два значения означают интенсивность для красного, следующие два – интенсивность для зеленого, и последние два – интенсивность для красного цветов.

1. :

– семейство цветовых пространств, которые используются для передачи цветных изображений в компонентном виде и цифровой фотографии. В таком виде – компонента яркости, и – синяя и красная цветоразностные компоненты.

В отличии от , проще представить в виде плоскости так, как это приведено на рисунке 1.2. Однако, на данном рисунке не приведена ось , которая отвечает за яркость.

Рисунок 1.2 – Простейшее представление



Такой формат изображений делается, поскольку яркость воспринимается глазом сильнее, чем другая цветовая информация и .

1. Переходы цветовых пространств:

Для того, что бы перейти из в необходимо использовать следующие заранее используемые формулы, в которых происходит деление на 255 для нормализации данных. Нормализация – процесс приведения данных к общему виду. Сами формулы приведены ниже:

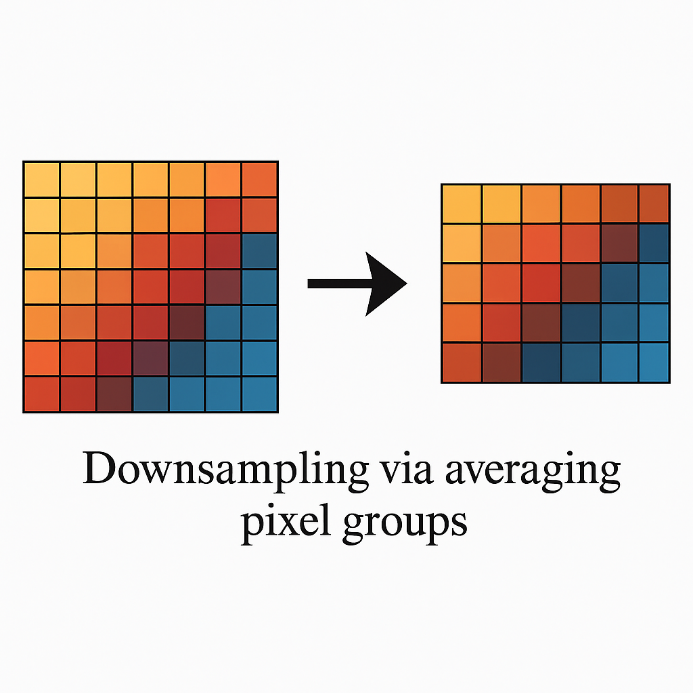
Для перехода из в используются обратные формулы:

Для наглядности, на рисунке 1.3 приведен пример изображения в , а также отдельно разбитые каналы для каждого из цветовых отображений.

1. Downsampling и Upsampling:

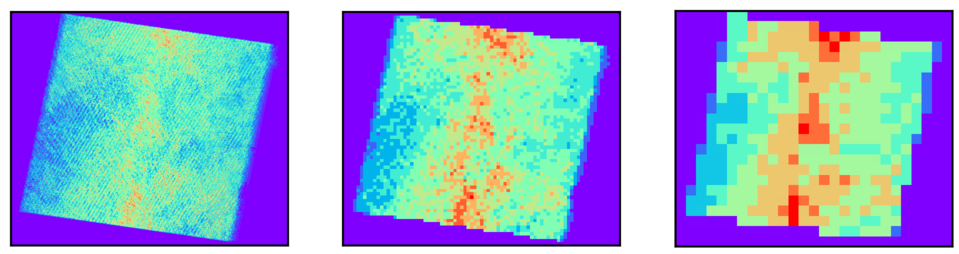
Downsampling – это процесс уменьшения размера изображения в раз. Данное действие происходит путем усреднения групп пикселей. Так, например, для и изображения размерами будет преобразовано так, как это показано на изображении 1.3.

Рисунок 1.3 – Downsampling преобразование



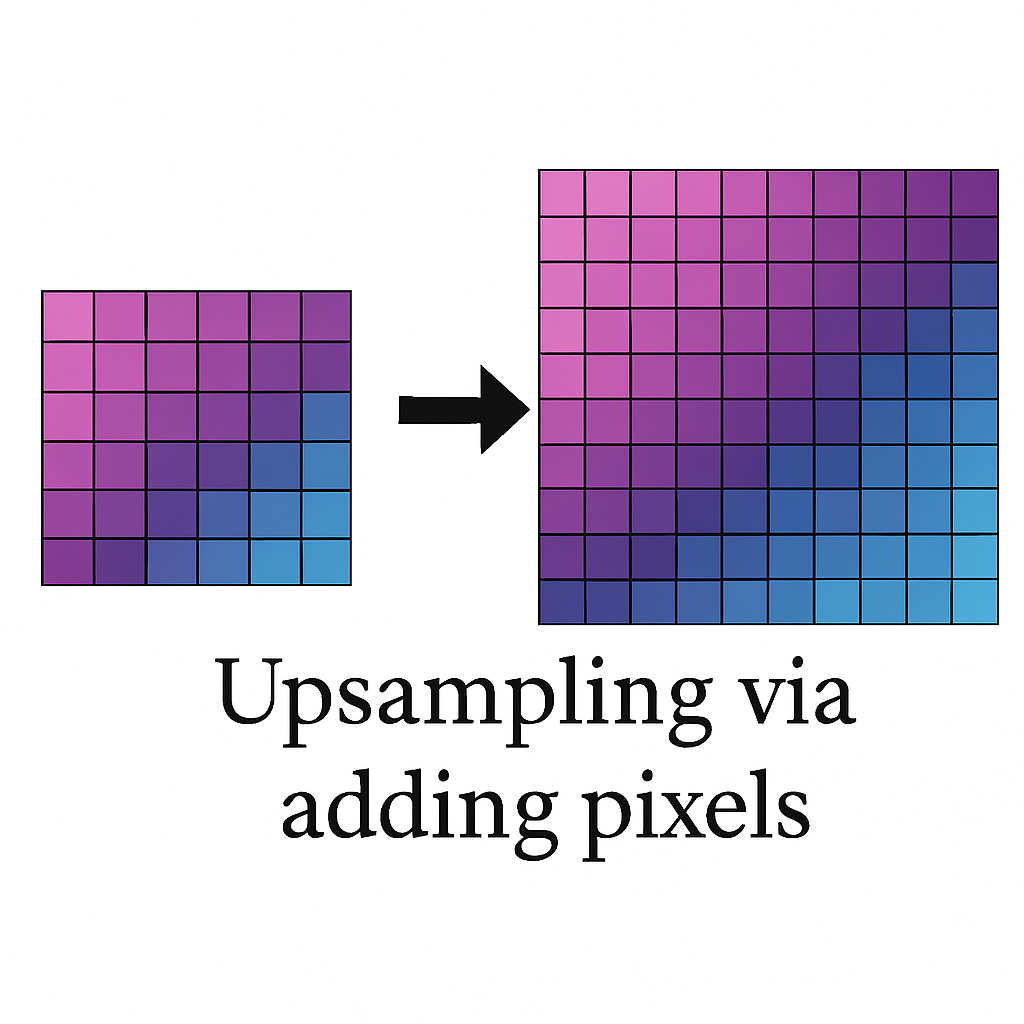
С помощью Downsampling изображение теряет незначительную часть своей информации, но оставляет большую часть понятной. Пример изображения до и после преобразования приведен на рисунке 1.4.

Рисунок 1.4 – Пример downsampling преобразования



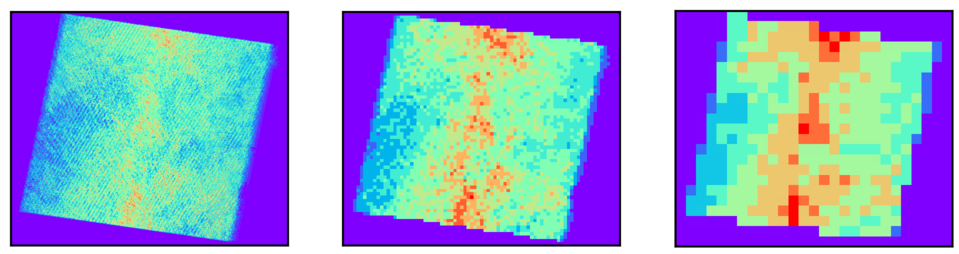
Upsampling – это процесс увеличения размера изображения в раз. Данное действие происходит путем добавления новых пикселей на основе существующих. Таким образом, находится крайний пиксель и расширяется в квадрат на основе этого. Так, например, для и изображения размерами будет преобразовано так, как это показано на изображении 1.5.

Рисунок 1.5 – Upsampling преобразование



С помощью Upsampling изображение восстанавливает часть своей ранее сжатой с помощью downsampling информации, но оставляет большую часть понятной. Пример изображения до и после преобразования приведен на рисунке 1.6.

Рисунок 1.6 – Изображение до и после upsampling преобразования



1. DCT-II 2D и iDCT-II 2D:

DCT-II 2D (далее просто DCT) – это математическое преобразование, которое используется для преобразования сигналов (звука, изображений) между временной, пространственной и частотными областями.

Сам DCT преобразует пиксели изображения в набор частотных коэффициентов. Эти коэффициенты показывают какие шаблоны присутствуют в сигнале.

Таким образом, DCT выделяет важные и неважные компоненты сигнала, так, например, в изображениях основные детали конструкции кодируются большими коэффициентами, а мелкие шумы – малыми. При сжатии малозначимые коэффициенты отбрасываются, что уменьшает размер файла.

iDCT-II 2D (далее просто iDCT) – это математическое преобразование, которое фактически выполняет обратное DCT действие. То есть iDCT восстанавливает исходные данные из частотных коэффициентов.

Формулы для DCT и iDCT приведены ниже:

где:

* значение в позиции ;
* коэффициент DCT в частотной области в позиции ;

1. Матрица квантования:

Матрица квантования – матрица с некоторыми значениями, размер которой совпадает с размером блока . При этом матрица квантования используется для управления степенью сжатия и выполняется сразу же после DCT.

Для существует две основных матрицы квантования, выведенные совершенно нетривиальным образом:

Для квантования блока изображения необходимо поделить каждый коэффициент блока на значение матрицы по тем же индексам. Формула выглядит следующим образом: .

Деквантование же происходит обратным образом, путем перемножения матрицы квантования на значение восстановленных из файла данных . Формально формула выглядит так: .

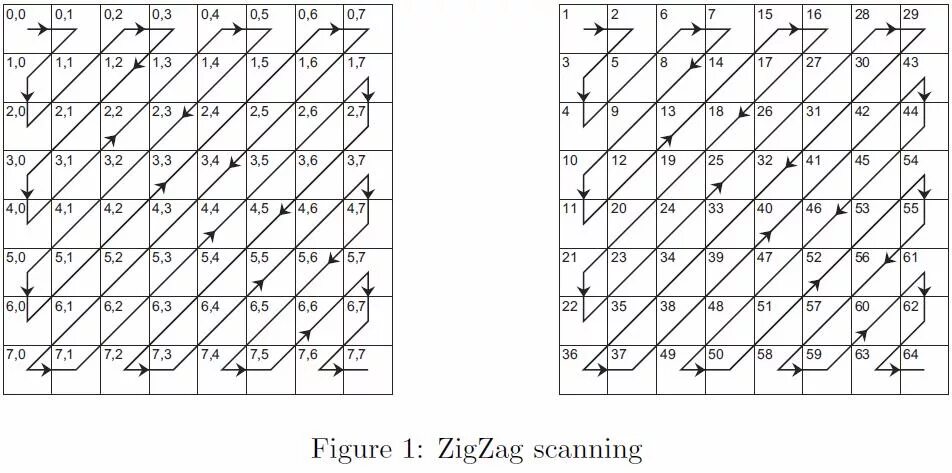
Также, матрицы квантования обладают еще одной интересной характеристикой – увеличением или уменьшением качества изображения. Данный параметр изменяет матрицу квантования по двум случаям:

Далее масштабирование матрицы выполняется по формуле, где нулевые элементы заменяются на единицу:

1. Зигзаг обход:

Зигзаг обход матрицы – это обход матрицы, который начинается с левого верхнего угла, и заканчивается на правом нижнем углу матрицы. Такой подход реализует преобразование матрицы в вектор-строку. Сам процесс прохода методом зигзаг по матрице приведен на рисунке 1.7.

Рисунок 1.7 – Зигзаг обход матрицы



Обратный зигзаг обход выполняется тривиальным образом, необходимо лишь идти в обратном порядке.

1. Разностное DC кодирование

Разностное DC кодирование – это метод сжатия, применяемый только к DC-элементам блоков. Вместо записи абсолютных DC значений, сохраняется разница между текущим DC-коэффициентом и следующим. DC-коэффициент – это первый элемент в матрице.

Сам алгоритм для каждого блока вычисляет разницу между его DC-коэффициентом и DC-коэффициентом предыдущего блока того же канала:

Далее, в сжатый поток записывается не сам DC-коэффициент, а его разность . Первый блок всегда кодируется так, как есть – без изменений.

1. Переменное AC и DC кодирование

AC/DC-кодирование – гибридный метод сжатия для , который комбинирует разностное DC-кодирование и кодирование для AC-коэффициентов. Наименование «переменное» означает, что метод адаптируется в зависимости от того, какой коэффициент подается на вход.

Поскольку разностное DC-кодирование было рассмотрено в прошлом пункте, то далее будет рассмотрено лишь DC и AC-кодирование, а также его входящие, такие как RLE и Huffman кодирование.

DC-коэффициенты после разностного кодирования определяют с помощью Хаффман кодирования по категории разности (см. пункт Huffman кодирование)

AC-коэффициенты же кодируются в два этапа. Первый из них – кодирование с помощью RLE (см. пункт RLE кодирование), а далее применяется Huffman кодирование так же, как и для DC-коэффициентов.

1. RLE кодирование

RLE используется для кодирования последовательностей нулей. Вместо хранения каждого нулевого значения отдельно, RLE записывает пары вида где – количество подряд идущих нулей перед текущем ненулевым коэффициентом, – количество бит, необходимых для кодирования абсолютного ненулевого значения, и вычисляется по следующей формуле:

Пример RLE кодирования приведен на рисунке 1.8.

Рисунок 1.8 – пример RLE кодирования

ТУТ

1. Huffman кодирование

После применения RLE, полученные пары подвергаются кодированию Хаффмана. Частотный словарь этих пар строится для всей последовательности и на его основе генерируется оптимальное дерево Хаффмана. Каждая пара затем заменяется соответствующим кодом Хаффмана.

Следом, для каждого ненулевого коэффициента записывается его остаточная часть – двоичное представление значения с длиной, соответствующей его категории. Отрицательные значения кодируются методом инвертирования всех битов.

1. DC/AC-декодирование

Для декодирования AC коэффициентов с файла читается частотный словарь и на его основе восстанавливается дерево Хаффмана. Далее последовательность последовательности битов разбираются, извлекая пары RLE, где нулевые значения восстанавливаются, а ненулевые декодируются. И в завершении производится восстановление полной последовательности.

DC-декодирование же тривиально, для этого необходимо найти категорию, определить по коду Хаффмана ее значение, а затем прочитать необходимое количество битов, которые содержат закодированную разность. Эта разность восстанавливается с учетом знака. После получения всех разностей происходит поэлементное накопление, а что бы восстановить абсолютные DC-значения для каждого блока используется формула:

где – декодированная разность между текущим и предыдущим DC-коэффициентом.

1. JPEG-кодировщик

JPEG – метод кодирования и декодирования изображений, который основан на DCT, квантовании, AC/DC-кодирования. Последовательность действий следующая:

1. Преобразование цветового пространства

Для начала исходное изображение переводится из одного из пространств () в . Данный этап нужен для того, что бы разделить изображение на 3 канала, описанных в пункте про .

1. Субдискретизация цветности

Каналы и подвергаются субдискретизации (downsampling) с коэффициентом 4:2:0, что уменьшает разрешение цветовой составляющей вдвое по вертикали и горизонтали, сохраняя при этом детализацию яркости.

1. Блочное разбиение

Все каналы () разбиваются на блоки размерами , так как DCT далее будет применятся к каждому отдельному блоку.

1. Дискретное косинусное преобразование (DCT)

К каждому из блоков применяется DCT, которая преобразует значения пикселей в частотное представление, что позволяет сконцентрировать энергию изображения в левом верхнем углу.

1. Квантование

Полученные DCT коэффициенты делятся на соответствующие значения матрицы квантования в зависимости от качества. На данном шаге происходит основная потеря данных из изображения.

1. Зигзаг чтение

Квантованные блоки преобразуются в одномерный вектор-строку в порядке зигзагообразного обхода. Это группирует нулевые коэффициенты, подготавливая данные для RLE.

1. AC/DC-кодирование

После всех этапов, описанные выше, применяется адаптивное AC/DC-кодирование. На данном этапе происходит основное сжатие и уменьшение размера исходного изображения.

В свою очередь, декодирование же происходит в обратном порядке, начиная с AC/DC-кодирования и заканчивая преобразованием из в исходный формат.

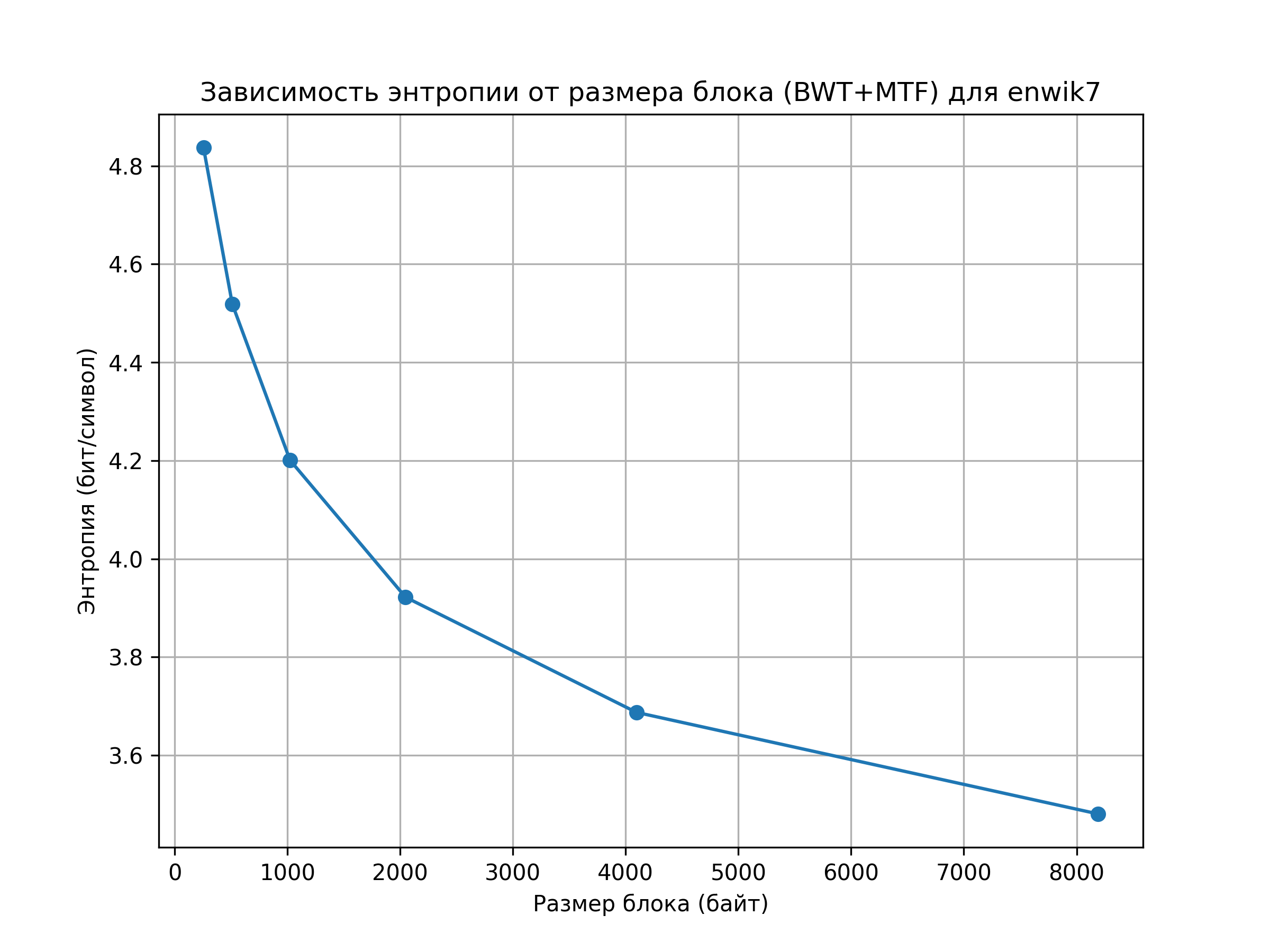
# Практическая часть.

## Зависимость энтропии для BWT и MTF.

Энтропия – это мера неопределенности или случайности данных, которая выражена в битах на символ для текущего примера. Вообще, энтропия определяет среднее количество информации, которая необходима для кодирования одного символа. То есть, энтропия выражает необходимое количество бит для кодирования одного символа.

BWT и MTF уменьшают размер данных, за счет чего энтропия будет уменьшаться. Однако, смотря на принципы работы алгоритмов тяжело сказать, будет ли она уменьшаться в зависимости от размера блока. Скорее всего – да. Ниже рассмотрен график, который был построен на эксперименте с различными размерами блоков.

Изображение 12. График для BWT и MTF.



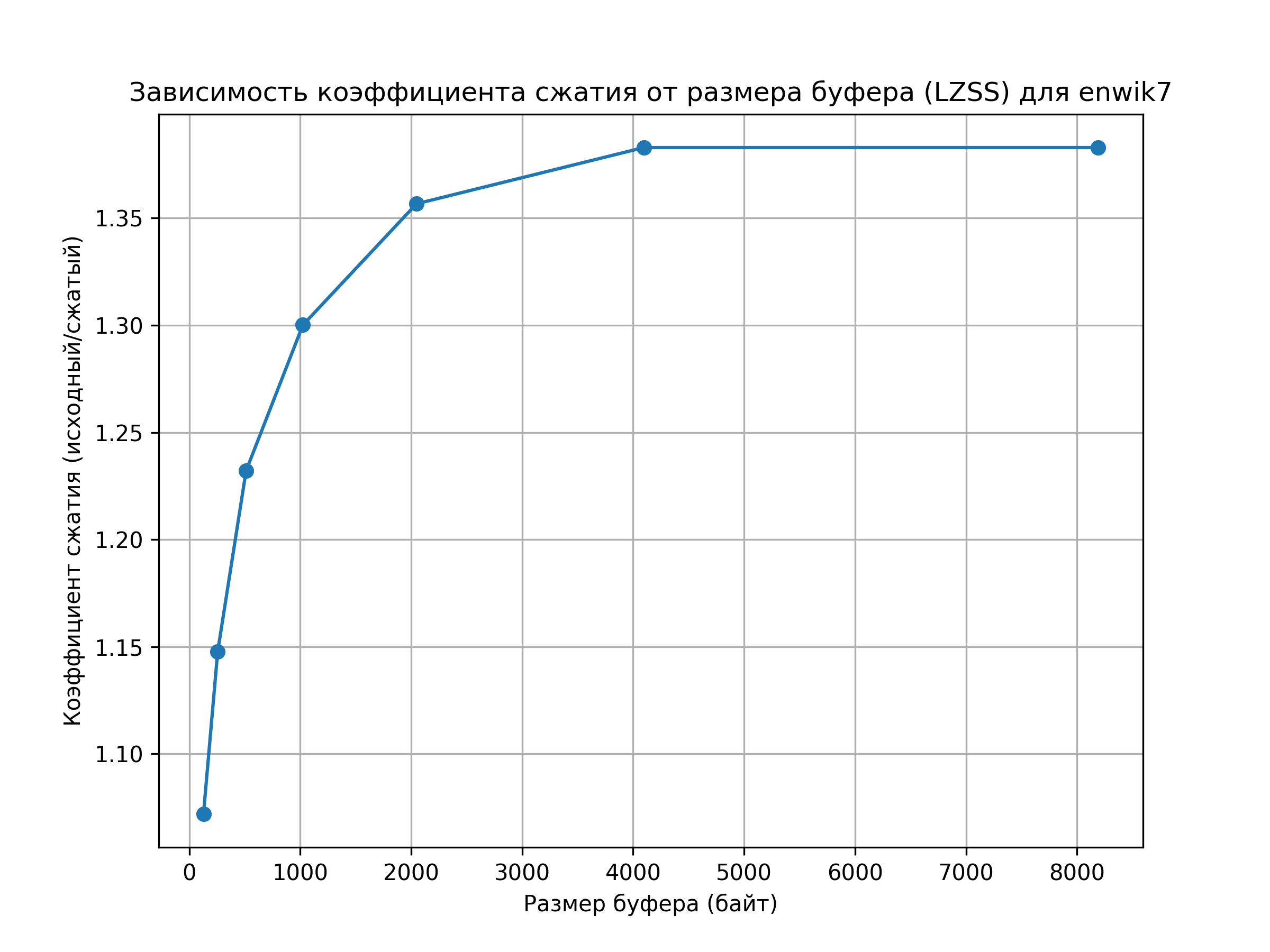
Как видно из изображения, количество бит на символ уменьшается с уменьшением размера блока. Если изначально символ может кодироваться от 8 до 24 бит, то уже с использованием данных преобразованием и размером блока в 256, этот размер уменьшается до 4.9 бит, что округляется в большую сторону – 5 бит на символ.

Однако, с увеличением размера блока увеличивается и время обработки последовательностей. Также увеличиваются и затраты памяти, поскольку требуется больше вычислительной мощи. Поэтому оптимальным значением, ведущим к среднему значению – 2048, поскольку с таким размером блока количество бит на символ в среднем будет равно 4.

## Зависимость коэффициента сжатия от размера буфера при использовании LZSS.

Как было описано в теоретической части – LZSS работает с некоторым буфером, в котором ищет схожие с текущей последовательностью сообщения. Стоит предположить, что в зависимости от увеличения буфера коэффициент сжатия сильно уменьшиться, что хорошо. Такое предположение сделано из-за того, что поскольку размер буфера стал больше, то и вероятность найти схожие последовательности становится выше. Ниже, на изображении 13 приведен экспериментально полученный график.

Изображение 13. График для коэффициента сжатия LZSS.

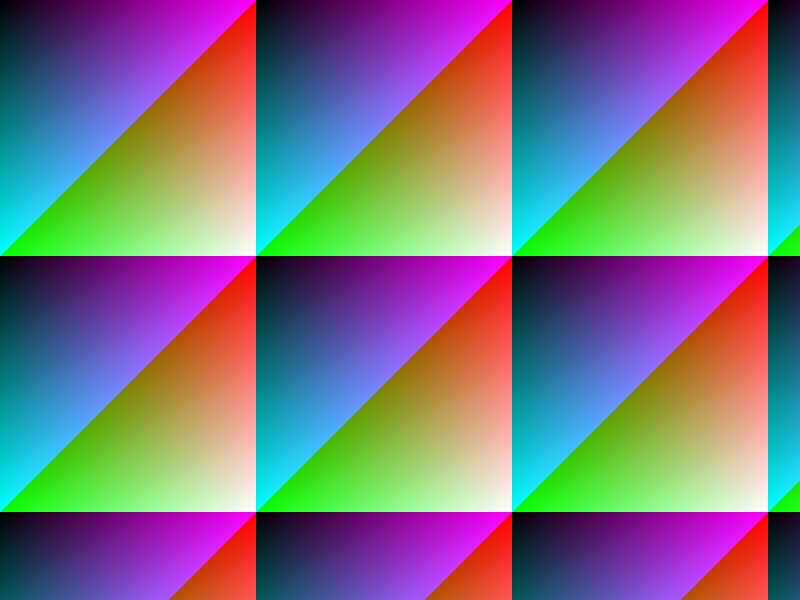


Из данного графика видно, что до 4096 коэффициент сжатия уменьшается, однако, после этого значения данный коэффициент изменяется очень и очень незначительно. Поэтому, оптимальным размером буфера с учетом временных и пространственных затрат – 2048, поскольку при таком размере коэффициент не сильно отличается от буфера, который в два раза больше, при этом, гораздо более значим, нежели при 1024.

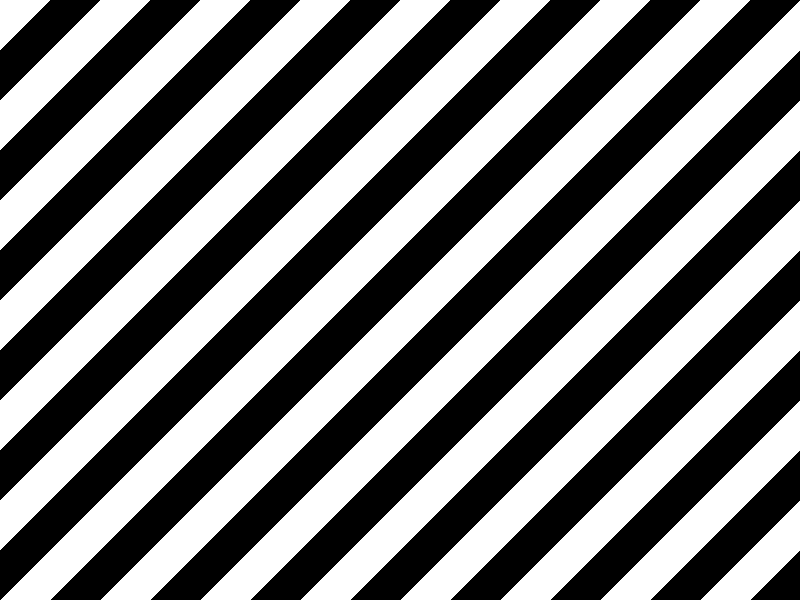
## Итоговые данные.

Коэффициентом сжатия далее будет называться отношение исходного размера файла к преобразованному. Для каждого из ниже приведенных видов информации были проделаны различные компрессоры. На изображениях 14–18 приведены примеры такой информации. В таблицах 1–5 приведены полученная информация о проделанных экспериментах.

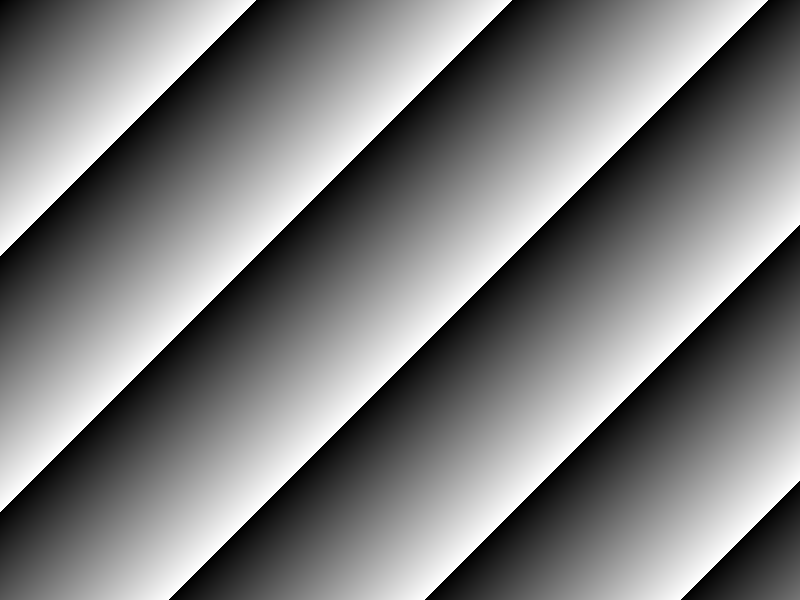
Изображение 14. Пример RGB raw изображения.



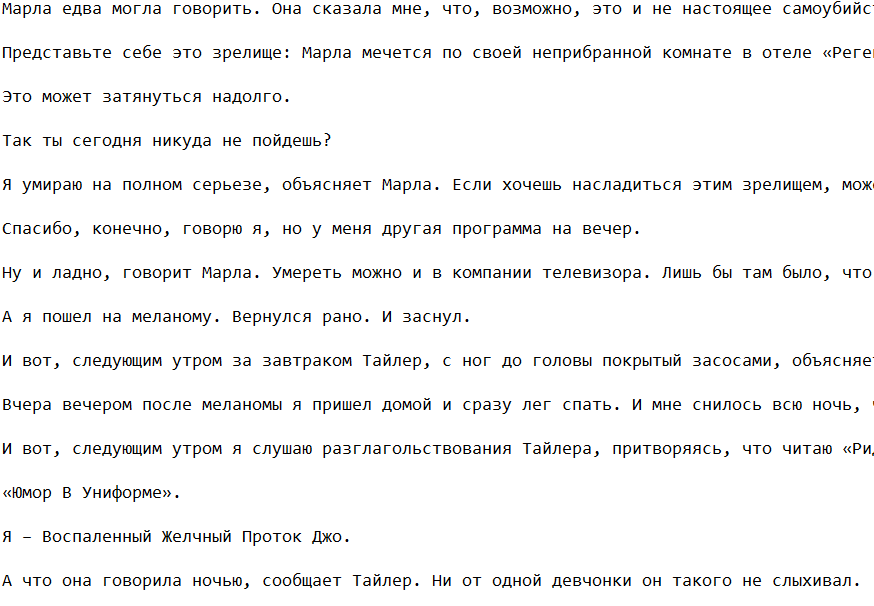
Изображение 15. Пример BW raw изображения.



Изображение 16. Пример GrayScale raw изображения.



Изображение 17. Пример текста с utf-8.



Изображение 18. Пример файла enwik7.

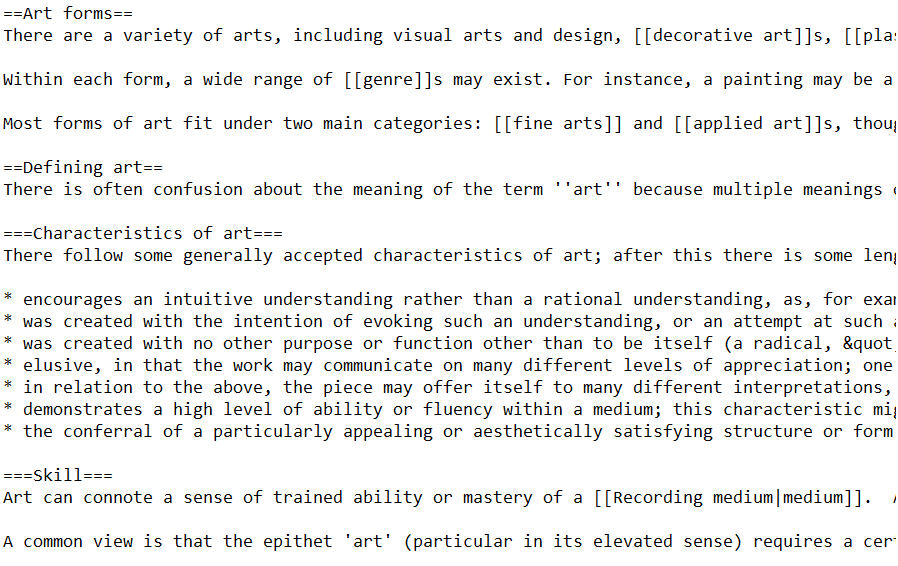


Таблица 1. Обработка RGB raw файла.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Алгоритм | Исходный размер (байты) | Сжатый размер (байты) | Коэффициент сжатия | Время кодирования (сек) | Время декодирования (сек) | Общее время (сек) |
| HA | 1440002 | 2067107 | 0.697 | 0.756 | 1.274 | 2.030 |
| RLE | 1440002 | 2869430 | 0.502 | 0.215 | 0.351 | 0.567 |
| BWT+RLE | 1440002 | 1666468 | 0.864 | 3.724 | 0.694 | 4.418 |
| BWT+MTF+HA | 1440002 | 1304897 | 1.104 | 5.204 | 1.460 | 6.664 |
| BWT+MTF+RLE+HA | 1440002 | 1657333 | 0.869 | 6.483 | 2.733 | 9.216 |
| LZSS | 1440002 | 942482 | 1.528 | 1.106 | 0.230 | 1.336 |
| LZSS+HA | 1440002 | 1335323 | 1.078 | 2.097 | 1.177 | 3.274 |
| LZW | 1440002 | 2262168 | 0.637 | 0.661 | 0.794 | 1.455 |
| LZW+HA | 1440002 | 2816333 | 0.511 | 2.041 | 3.024 | 5.066 |

Таблица 2. Обработка BW raw файла.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Алгоритм | Исходный размер (байты) | Сжатый размер (байты) | Коэффициент сжатия | Время кодирования (сек) | Время декодирования (сек) | Общее время (сек) |
| HA | 480002 | 64365 | 7.458 | 0.069 | 0.050 | 0.118 |
| RLE | 480002 | 20626 | 23.272 | 0.019 | 0.007 | 0.026 |
| BWT+RLE | 480002 | 11788 | 40.720 | 1.232 | 0.135 | 1.367 |
| BWT+MTF+HA | 480002 | 74114 | 6.477 | 1.481 | 0.272 | 1.753 |
| BWT+MTF+RLE+HA | 480002 | 13424 | 35.757 | 1.559 | 0.237 | 1.796 |
| LZSS | 480002 | 8762 | 54.782 | 1.114 | 0.055 | 1.169 |
| LZSS+HA | 480002 | 8816 | 54.447 | 1.102 | 0.061 | 1.164 |
| LZW | 480002 | 55696 | 8.618 | 0.111 | 0.035 | 0.146 |
| LZW+HA | 480002 | 45737 | 10.495 | 0.135 | 0.065 | 0.200 |

Таблица 3. Обработка GrayScale raw файла.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Алгоритм | Исходный размер (байты) | Сжатый размер (байты) | Коэффициент сжатия | Время кодирования (сек) | Время декодирования (сек) | Общее время (сек) |
| HA | 480002 | 782682 | 0.613 | 0.249 | 0.799 | 1.048 |
| RLE | 480002 | 960944 | 0.500 | 0.070 | 0.123 | 0.192 |
| BWT+RLE | 480002 | 128002 | 3.750 | 1.175 | 0.161 | 1.336 |
| BWT+MTF+HA | 480002 | 359823 | 1.334 | 1.527 | 0.415 | 1.942 |
| BWT+MTF+RLE+HA | 480002 | 265275 | 1.809 | 1.471 | 0.435 | 1.905 |
| LZSS | 480002 | 74000 | 6.487 | 0.333 | 0.056 | 0.389 |
| LZSS+HA | 480002 | 117389 | 4.089 | 0.339 | 0.132 | 0.471 |
| LZW | 480002 | 421428 | 1.139 | 0.168 | 0.112 | 0.280 |
| LZW+HA | 480002 | 577304 | 0.831 | 0.385 | 0.471 | 0.856 |

Таблица 4. Обработка utf-8 текста.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Алгоритм | Исходный размер (байты) | Сжатый размер (байты) | Коэффициент сжатия | Время кодирования (сек) | Время декодирования (сек) | Общее время (сек) |
| HA | 534637 | 344172 | 1.553 | 0.129 | 0.252 | 0.381 |
| RLE | 534637 | 1063798 | 0.503 | 0.079 | 0.242 | 0.321 |
| BWT+RLE | 534637 | 440956 | 1.212 | 1.442 | 0.370 | 1.812 |
| BWT+MTF+HA | 534637 | 313881 | 1.703 | 1.683 | 0.503 | 2.186 |
| BWT+MTF+RLE+HA | 534637 | 330669 | 1.617 | 1.725 | 0.579 | 2.303 |
| LZSS | 534637 | 376558 | 1.420 | 1.799 | 0.120 | 1.919 |
| LZSS+HA | 534637 | 464718 | 1.150 | 1.999 | 0.433 | 2.432 |
| LZW | 534637 | 427330 | 1.251 | 0.187 | 0.113 | 0.299 |
| LZW+HA | 534637 | 548458 | 0.975 | 0.402 | 0.466 | 0.868 |

Таблица 5. Обработка файла enwik7.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Алгоритм | Исходный размер (байты) | Сжатый размер (байты) | Коэффициент сжатия | Время кодирования (сек) | Время декодирования (сек) | Общее время (сек) |
| HA | 10000000 | 7909248 | 1.264 | 2.886 | 5.981 | 8.867 |
| RLE | 10000000 | 18859124 | 0.530 | 1.576 | 2.479 | 4.056 |
| BWT+RLE | 10000000 | 11851308 | 0.844 | 29.162 | 6.046 | 35.208 |
| BWT+MTF+HA | 10000000 | 7404740 | 1.350 | 35.158 | 12.145 | 47.303 |
| BWT+MTF+RLE+HA | 10000000 | 9116458 | 1.097 | 33.980 | 12.524 | 46.504 |
| LZSS | 10000000 | 7781514 | 1.285 | 17.652 | 1.706 | 19.358 |
| LZSS+HA | 10000000 | 9740539 | 1.027 | 23.041 | 7.691 | 30.732 |
| LZW | 10000000 | 10126276 | 0.988 | 3.801 | 2.557 | 6.358 |
| LZW+HA | 10000000 | 12753399 | 0.784 | 8.634 | 9.746 | 18.380 |

Также интересным моментом является то, что png, jpg, jpeg являются уже сжатыми форматами, поэтому попытка сжать любой из них приведет к увеличению занимаемой памяти, а не уменьшению. Подтверждение тому данные, которые приведены в таблице 6. Сам же пример такого изображения приведен на изображении 19.

Изображение 19. Пример настоящего jpeg изображения.



Таблица 6. Обработка jpeg файла.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Алгоритм | Исходный размер (байты) | Сжатый размер (байты) | Коэффициент сжатия | Время кодирования (сек) | Время декодирования (сек) | Общее время (сек) |
| HA | 46488 | 74812 | 0.621 | 0.022 | 0.044 | 0.066 |
| RLE | 46488 | 91782 | 0.507 | 0.006 | 0.010 | 0.016 |
| BWT+RLE | 46488 | 91626 | 0.507 | 0.107 | 0.028 | 0.135 |
| BWT+MTF+HA | 46488 | 75551 | 0.615 | 0.162 | 0.071 | 0.233 |
| BWT+MTF+RLE+HA | 46488 | 113764 | 0.409 | 0.202 | 0.098 | 0.300 |
| LZSS | 46488 | 52084 | 0.893 | 0.034 | 0.005 | 0.039 |
| LZSS+HA | 46488 | 81423 | 0.571 | 0.068 | 0.053 | 0.121 |
| LZW | 46488 | 89494 | 0.519 | 0.021 | 0.018 | 0.039 |
| LZW+HA | 46488 | 110207 | 0.422 | 0.059 | 0.079 | 0.138 |

И в конце необходимо проверить все кодировщики, но уже на запускаемом exe файле. Результаты представлены в таблице 7.

Таблица 7. Обработка exe файла.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Алгоритм | Исходный размер (байты) | Сжатый размер (байты) | Коэффициент сжатия | Время кодирования (сек) | Время декодирования (сек) | Общее время (сек) |
| HA | 1096224 | 940123 | 0.858 | 0.3425 | 0.595 | 0.938 |
| RLE | 1096224 | 2051712 | 0.534 | 0.241 | 0.214 | 0.456 |
| BWT+RLE | 1096224 | 1218404 | 0.900 | 2.779 | 0.495 | 3.273 |
| BWT+MTF+HA | 1096224 | 1110656 | 0.987 | 3.257 | 1.150 | 4.407 |
| BWT+MTF+RLE+HA | 1096224 | 1397064 | 0.785 | 3.720 | 1.452 | 5.172 |
| LZSS | 1096224 | 802361 | 1.366 | 2.627 | 0.160 | 2.787 |
| LZSS+HA | 1096224 | 1142071 | 0.960 | 2.686 | 0.809 | 3.494 |
| LZW | 1096224 | 1283488 | 0.854 | 0.385 | 0.275 | 0.659 |
| LZW+HA | 1096224 | 1609095 | 0.681 | 0.930 | 1.177 | 2.107 |

# Код программы

1. encoders\_decoders/blockProcessor.py – файл с реализацией блочной обработки:

import struct

class BlockProcessor:

BLOCK\_HEADER = struct.Struct('>I')

use\_header = True

@classmethod

def split\_blocks(cls, data: bytes, block\_size: int) -> list:

if cls.use\_header:

return [data[i:i + block\_size] for i in range(0, len(data), block\_size)]

else:

return [data]

@classmethod

def add\_block\_header(cls, block: bytes) -> bytes:

if cls.use\_header:

return cls.BLOCK\_HEADER.pack(len(block)) + block

else:

return block

@classmethod

def read\_block(cls, data: bytes, ptr: int) -> tuple:

if cls.use\_header:

if ptr + cls.BLOCK\_HEADER.size > len(data):

return None, ptr

block\_len = cls.BLOCK\_HEADER.unpack\_from(data, ptr)[0]

ptr += cls.BLOCK\_HEADER.size

return data[ptr:ptr + block\_len], ptr + block\_len

else:

return data[ptr:], len(data)

1. encoders\_decoders/bwt.py – файл с реализацией BWT:

from .blockProcessor import \*

class \_Sorting:

def \_\_init\_\_(self):

pass

def sort\_indices(self, rotations):

indices = list(range(len(rotations)))

return self.merge\_sort(rotations, indices)

def merge\_sort(self, rotations, indices):

if len(indices) <= 1:

return indices

mid = len(indices) // 2

left = self.merge\_sort(rotations, indices[:mid])

right = self.merge\_sort(rotations, indices[mid:])

return self.merge(rotations, left, right)

def merge(self, rotations, left, right):

result = []

i = j = 0

while i < len(left) and j < len(right):

# При равных ротациях сравниваем исходные индексы

if (rotations[left[i]] < rotations[right[j]] or

(rotations[left[i]] == rotations[right[j]] and left[i] < right[j])):

result.append(left[i])

i += 1

else:

result.append(right[j])

j += 1

result.extend(left[i:])

result.extend(right[j:])

return result

class BWT:

def \_\_init\_\_(self, block\_size):

self.block\_size = block\_size

def encode(self, data: bytes) -> bytes:

encoded = bytearray()

for block in BlockProcessor.split\_blocks(data, self.block\_size):

if not block:

continue

rotations = [block[i:] + block[:i] for i in range(len(block))]

sa = \_Sorting().sort\_indices(rotations)

last\_col = bytes(rotations[i][-1] for i in sa)

orig\_idx = sa.index(0)

encoded\_block = struct.pack('>II', orig\_idx, len(block)) + last\_col

encoded.extend(BlockProcessor.add\_block\_header(encoded\_block))

return bytes(encoded)

def decode(self, data: bytes) -> bytes:

decoded = bytearray()

ptr = 0

while ptr <= len(data):

block, ptr = BlockProcessor.read\_block(data, ptr)

if not block:

break

if len(block) < 8:

continue

try:

orig\_idx, blen = struct.unpack('>II', block[:8])

except struct.error:

continue

last\_col = block[8:8 + blen]

if len(last\_col) != blen:

continue

# Создаём пары (символ, индекс) и выполняем стабильную сортировку

tuples = [(last\_col[i], i) for i in range(blen)]

sorted\_tuples = sorted(tuples, key=lambda x: (x[0], x[1]))

LF = [t[1] for t in sorted\_tuples]

first\_col = [t[0] for t in sorted\_tuples]

current\_idx = orig\_idx

result = bytearray()

for \_ in range(blen):

result.append(first\_col[current\_idx])

current\_idx = LF[current\_idx]

decoded.extend(result)

return bytes(decoded)

1. encoders\_decoders/huffman.py – файл с реализацией HA:

from .blockProcessor import \*

from collections import defaultdict

from heapq import heappop, heapify, heappush

class Huffman:

def \_\_init\_\_(self, block\_size):

self.block\_size = block\_size

def encode(self, data: bytes) -> bytes:

bp = BlockProcessor()

encoded = bytearray()

for block in bp.split\_blocks(data, self.block\_size):

freq = defaultdict(int)

for b in block:

freq[b] += 1

heap = [[wt, [sym, ""]] for sym, wt in freq.items()]

heapify(heap)

while len(heap) > 1:

lo = heappop(heap)

hi = heappop(heap)

for pair in lo[1:]:

pair[1] = '0' + pair[1]

for pair in hi[1:]:

pair[1] = '1' + pair[1]

heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])

codes = {sym: code for sym, code in heap[0][1:]}

bit\_str = ''.join(codes[b] for b in block)

padding = (8 - (len(bit\_str) % 8) % 8)

bit\_str += '0' \* padding

header = struct.pack('>BH', padding, len(freq))

for sym, count in freq.items():

header += struct.pack('>BI', sym, count)

block\_enc = bytearray()

for i in range(0, len(bit\_str), 8):

block\_enc.append(int(bit\_str[i:i + 8], 2))

encoded.extend(bp.add\_block\_header(header + block\_enc))

return bytes(encoded)

def decode(self, data: bytes) -> bytes:

bp = BlockProcessor()

decoded = bytearray()

ptr = 0

while ptr < len(data):

block, ptr = bp.read\_block(data, ptr)

if not block:

break

padding = block[0]

num\_syms = struct.unpack('>H', block[1:3])[0]

freq = {}

pos = 3

for \_ in range(num\_syms):

sym = block[pos]

count = struct.unpack('>I', block[pos + 1:pos + 5])[0]

freq[sym] = count

pos += 5

heap = [[wt, [sym, ""]] for sym, wt in freq.items()]

heapify(heap)

while len(heap) > 1:

lo = heappop(heap)

hi = heappop(heap)

for pair in lo[1:]:

pair[1] = '0' + pair[1]

for pair in hi[1:]:

pair[1] = '1' + pair[1]

heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])

codes = {sym: code for sym, code in heap[0][1:]}

rev\_codes = {v: k for k, v in codes.items()}

bit\_str = ''.join(f'{b:08b}' for b in block[pos:])

bit\_str = bit\_str[:-padding] if padding else bit\_str

current = ''

for bit in bit\_str:

current += bit

if current in rev\_codes:

decoded.append(rev\_codes[current])

current = ''

return bytes(decoded)

1. encoders\_decoders/lzss.py – файл с реализацией LZSS:

from .blockProcessor import \*

import struct

"""

Update:

1) (12, 12) -> (15, 9), [(offset, len)]

2) флаг исправлен, теперь вместо дополнительного флага в виде байта используется битовый флаг

3) find и вложенный цикл заменен на словарь, который хранит список позиций, где уже была эта последовательность

4) Ограничение совпадения до 511 бит (9 байт)

"""

class LZSS:

def \_\_init\_\_(self, block\_size, window\_size=2048):

self.window\_size = window\_size

self.block\_size = block\_size

def encode(self, data: bytes) -> bytes:

bp = BlockProcessor()

encoded = bytearray()

for block in bp.split\_blocks(data, self.block\_size):

block\_enc = bytearray(struct.pack('>I', self.window\_size))

i = 0

pos\_dict = {}

while i < len(block):

flag = 0

tokens = bytearray()

token\_count = 0

while token\_count < 8 and i < len(block):

window\_start = max(0, i - self.window\_size)

best\_length = 0

best\_offset = 0

if i < len(block) - 1:

key = block[i:i + 2]

if key in pos\_dict:

for candidate in pos\_dict[key]:

if candidate < window\_start:

continue

length = 0

while (i + length < len(block) and

candidate + length < len(block) and

block[candidate + length] == block[i + length]):

length += 1

if length >= min(511, len(block) - i):

break

if length > best\_length:

best\_length = length

best\_offset = i - candidate

if best\_length == min(511, len(block) - i):

break

if best\_length >= 3:

ref = (best\_offset << 9) | best\_length

tokens.extend(ref.to\_bytes(3, 'big'))

flag |= (1 << token\_count)

# Обновляем словарь для всех позиций, покрытых ссылкой

for j in range(best\_length):

if i + j < len(block) - 1:

key = block[i + j:i + j + 2]

pos\_dict.setdefault(key, []).append(i + j)

i += best\_length

else:

# Литерал – записываем один байт

tokens.append(block[i])

if i < len(block) - 1:

key = block[i:i + 2]

pos\_dict.setdefault(key, []).append(i)

i += 1

token\_count += 1

block\_enc.append(flag)

block\_enc.extend(tokens)

encoded.extend(bp.add\_block\_header(block\_enc))

return bytes(encoded)

def decode(self, data: bytes) -> bytes:

bp = BlockProcessor()

decoded = bytearray()

ptr = 0

while ptr < len(data):

block\_enc, ptr = bp.read\_block(data, ptr)

if not block\_enc:

break

\_ = struct.unpack('>I', block\_enc[:4])[0]

i = 4

buf = bytearray()

while i < len(block\_enc):

flag = block\_enc[i]

i += 1

for bit in range(8):

if i >= len(block\_enc):

break

if flag & (1 << bit):

if i + 3 > len(block\_enc):

break

ref = int.from\_bytes(block\_enc[i:i+3], 'big')

i += 3

offset = ref >> 9

length = ref & 0x1FF

if length < 3:

length = 3

start = len(buf) - offset

for j in range(length):

if start + j < 0 or start + j >= len(buf):

break

buf.append(buf[start + j])

else:

buf.append(block\_enc[i])

i += 1

decoded.extend(buf)

return bytes(decoded)

1. encoders\_decoders/lzw.py – файл с реализацией LZW:

from .blockProcessor import \*

class LZW:

def \_\_init\_\_(self, block\_size):

self.block\_size = block\_size

self.reset\_dict()

def reset\_dict(self):

self.dict\_size = 256

self.dictionary = {bytes([i]): i for i in range(256)}

self.rev\_dict = {i: bytes([i]) for i in range(256)}

def encode(self, data: bytes) -> bytes:

bp = BlockProcessor()

encoded = bytearray()

for block in bp.split\_blocks(data, self.block\_size):

self.reset\_dict()

w = b''

block\_enc = bytearray()

for c in block:

wc = w + bytes([c])

if wc in self.dictionary:

w = wc

else:

block\_enc.extend(struct.pack('>H', self.dictionary[w]))

self.dictionary[wc] = self.dict\_size

self.dict\_size += 1

w = bytes([c])

if w:

block\_enc.extend(struct.pack('>H', self.dictionary[w]))

encoded.extend(bp.add\_block\_header(block\_enc))

return bytes(encoded)

def decode(self, data: bytes) -> bytes:

bp = BlockProcessor()

decoded = bytearray()

ptr = 0

while ptr < len(data):

block\_enc, ptr = bp.read\_block(data, ptr)

if not block\_enc:

break

self.reset\_dict()

idx = 0

if len(block\_enc) < 2:

continue # Пропустить неполные данные

# Обработка первого кода

code = struct.unpack('>H', block\_enc[idx:idx+2])[0]

idx += 2

if code not in self.rev\_dict:

continue

prev = self.rev\_dict[code]

decoded.extend(prev)

# Обработка оставшихся кодов

while idx < len(block\_enc):

if idx + 2 > len(block\_enc):

break

code = struct.unpack('>H', block\_enc[idx:idx+2])[0]

idx += 2

if code in self.rev\_dict:

entry = self.rev\_dict[code]

elif code == self.dict\_size:

entry = prev + prev[0:1]

else:

break # Некорректный код

decoded.extend(entry)

self.rev\_dict[self.dict\_size] = prev + entry[0:1]

self.dict\_size += 1

prev = entry

return bytes(decoded)

1. encoders\_decoders/mtf.py – файл с реализацией MTF:

from .blockProcessor import \*

class MTF:

def \_\_init\_\_(self, block\_size):

self.block\_size = block\_size

def encode(self, data: bytes) -> bytes:

bp = BlockProcessor()

encoded = bytearray()

symbols = list(range(256))

for block in bp.split\_blocks(data, self.block\_size):

block\_enc = bytearray()

local\_symbols = symbols.copy()

for b in block:

idx = local\_symbols.index(b)

block\_enc.append(idx)

del local\_symbols[idx]

local\_symbols.insert(0, b)

encoded.extend(bp.add\_block\_header(block\_enc))

return bytes(encoded)

def decode(self, data: bytes) -> bytes:

bp = BlockProcessor()

decoded = bytearray()

ptr = 0

while ptr < len(data):

block\_enc, ptr = bp.read\_block(data, ptr)

if not block\_enc:

break

symbols = list(range(256))

for idx in block\_enc:

b = symbols[idx]

decoded.append(b)

del symbols[idx]

symbols.insert(0, b)

return bytes(decoded)

1. encoders\_decoders/rle.py – файл с реализацией RLE:

from .blockProcessor import \*

class RLE:

def \_\_init\_\_(self, block\_size):

self.block\_size = block\_size

def encode(self, data: bytes) -> bytes:

encoded = bytearray()

for block in BlockProcessor.split\_blocks(data, self.block\_size):

block\_enc = bytearray()

prev = block[0] if block else 0

count = 1

for b in block[1:]:

if b == prev and count < 255:

count += 1

else:

block\_enc.extend([count, prev])

prev = b

count = 1

if count > 0:

block\_enc.extend([count, prev])

encoded.extend(BlockProcessor.add\_block\_header(block\_enc))

return bytes(encoded)

def decode(self, data: bytes) -> bytes:

decoded = bytearray()

ptr = 0

while ptr < len(data):

block\_enc, ptr = BlockProcessor.read\_block(data, ptr)

if not block\_enc:

break

for i in range(0, len(block\_enc), 2):

count, byte = block\_enc[i], block\_enc[i + 1]

decoded.extend([byte] \* count)

return bytes(decoded)

1. supplement/generate.py – файл с реализацией генерацией raw изображений:

import os

import numpy as np

from PIL import Image

import random

def convert\_to\_raw(image\_path, output\_path, image\_type):

img = Image.open(image\_path)

img = img.convert('L' if image\_type == 'gray' else 'RGB')

pixels = np.array(img)

with open(output\_path, 'wb') as f:

if image\_type == 'bw':

f.write(b'BW')

f.write(pixels.tobytes())

elif image\_type == 'gray':

f.write(b'GR')

f.write(pixels.tobytes())

elif image\_type == 'color':

f.write(b'CL')

f.write(pixels.tobytes())

class DataGenerator:

@staticmethod

def create\_directory(path='test\_data'):

if not os.path.exists(path):

os.makedirs(path)

return path

class ImageGenerator:

@staticmethod

def generate\_images(path):

# Генерация всех изображений

ImageGenerator.generate\_bw\_image(path)

ImageGenerator.generate\_gray\_image(path)

ImageGenerator.generate\_color\_image(path)

@staticmethod

def generate\_bw\_image(path, size=(800, 600)):

img = Image.new('1', size, 0)

pixels = np.array(img)

for y in range(size[1]):

for x in range(size[0]):

pixels[y, x] = (x + y) % 100 < 50

img = Image.fromarray(pixels)

img.save(os.path.join(path, 'bw\_image.png'))

@staticmethod

def generate\_gray\_image(path, size=(800, 600)):

img = Image.new('L', size, 128)

pixels = np.array(img)

for y in range(size[1]):

for x in range(size[0]):

pixels[y, x] = (x + y) % 256

img = Image.fromarray(pixels)

img.save(os.path.join(path, 'gray\_image.png'))

@staticmethod

def generate\_color\_image(path, size=(800, 600)):

img = Image.new('RGB', size, (255, 0, 0))

pixels = np.array(img)

for y in range(size[1]):

for x in range(size[0]):

pixels[y, x] = (x % 256, y % 256, (x + y) % 256)

img = Image.fromarray(pixels)

img.save(os.path.join(path, 'color\_image.png'))

class TextGenerator:

@staticmethod

def generate\_realistic\_text(path, size\_mb=5):

filename = os.path.join(path, 'realistic\_text.txt')

chunks = []

# Пример текста с разной структурой

templates = [

"В чащах юга жил бы цитрус? Да, но фальшивый экземпляр! ",

"Съешь же ещё этих мягких французских булок, да выпей чаю. ",

"Широкая электрификация южных губерний даст мощный толчок подъёму сельского хозяйства. ",

"Летают ли коровы над радугой? Это философский вопрос. ",

"Размышления о смысле жизни приводят к неожиданным выводам. "

]

# Рассчитываем необходимое количество данных

target\_size = size\_mb \* 1024 \* 1024

current\_size = 0

with open(filename, 'w', encoding='utf-8') as f:

while current\_size < target\_size:

# Случайно выбираем шаблон и модифицируем его

chunk = random.choice(templates)

chunk = chunk \* random.randint(1, 5)

# Добавляем случайные числа

chunk += ''.join(str(random.randint(0, 9)) \* random.randint(0, 5))

# Добавляем случайные знаки препинания

chunk += random.choice(['!', '?', '.', ',']) \* random.randint(0, 3)

# Добавляем переносы строк

chunk += '\n' \* random.randint(0, 3)

# Записываем и считаем размер

f.write(chunk)

current\_size += len(chunk.encode('utf-8'))

return filename

class RawConverter:

@staticmethod

def convert\_all\_images(path):

# Конвертация всех изображений в RAW

RawConverter.convert\_to\_raw(

os.path.join(path, 'bw\_image.png'),

os.path.join(path, 'bw\_image.raw'),

'bw'

)

RawConverter.convert\_to\_raw(

os.path.join(path, 'gray\_image.png'),

os.path.join(path, 'gray\_image.raw'),

'gray'

)

RawConverter.convert\_to\_raw(

os.path.join(path, 'color\_image.png'),

os.path.join(path, 'color\_image.raw'),

'color'

)

@staticmethod

def convert\_to\_raw(image\_path, output\_path, image\_type):

img = Image.open(image\_path)

if image\_type == 'bw':

img = img.convert('1')

elif image\_type == 'gray':

img = img.convert('L')

elif image\_type == 'color':

img = img.convert('RGB')

pixels = np.array(img)

with open(output\_path, 'wb') as f:

f.write(b'BW' if image\_type == 'bw' else b'GR' if image\_type == 'gray' else b'CL')

f.write(pixels.tobytes())

1. graphs\_and\_analysis/comp\_ration.py – файл для построения графика коэффициента LZSS:

import matplotlib.pyplot as plt

from pathlib import Path

from encoders\_decoders import LZSS

def compress\_lzss(data: bytes, buffer\_size: int) -> bytes:

"""

Сжимает данные с использованием алгоритма LZSS с заданным размером буфера.

Предполагается, что конструктор LZSS принимает размер буфера как параметр.

"""

lzss = LZSS(buffer\_size)

return lzss.encode(data)

def main():

# Путь к файлу enwik7 (при необходимости измените путь)

file\_path = Path("Z:\prog\аисдик\compression\_test\_data\enwik7")

if not file\_path.exists():

print("Файл enwik7 не найден!")

return

with open(file\_path, "rb") as f:

data = f.read()

original\_size = len(data)

buffer\_sizes = [256, 512, 1024, 2048, 4096, 8192]

compression\_ratios = []

for bs in buffer\_sizes:

compressed\_data = compress\_lzss(data, bs)

compressed\_size = len(compressed\_data)

ratio = compressed\_size / original\_size

compression\_ratios.append(ratio)

print(f"Размер буфера: {bs} байт, Коэффициент сжатия: {ratio:.4f}")

plt.figure(figsize=(8, 6))

plt.plot(buffer\_sizes, compression\_ratios, marker="o")

plt.xlabel("Размер буфера (байт)")

plt.ylabel("Коэффициент сжатия (сжатый/исходный)")

plt.title("Зависимость коэффициента сжатия от размера буфера (LZSS) для enwik7")

plt.grid(True)

plt.savefig("compression\_ratio\_vs\_buffer\_size.png", dpi=300)

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

main()

1. graphs\_and\_analysis/graph\_entropy.py – файл для построения энтропии:

import math

import matplotlib.pyplot as plt

from pathlib import Path

from encoders\_decoders import BWT, MTF

def compute\_entropy(data: bytes) -> float:

"""

Вычисляет энтропию (в битах на символ) для заданных данных.

"""

if not data:

return 0.0

freq = {}

for byte in data:

freq[byte] = freq.get(byte, 0) + 1

total = len(data)

entropy = 0.0

for count in freq.values():

p = count / total

entropy -= p \* math.log2(p)

return entropy

def bwt\_mtf\_encode(data: bytes, block\_size: int) -> bytes:

"""

Применяет последовательность преобразований BWT и MTF для данных,

разбитых на блоки указанного размера.

Каждый блок обрабатывается независимо.

"""

result = b""

for i in range(0, len(data), block\_size):

block = data[i:i + block\_size]

bwt = BWT(block\_size)

mtf = MTF(block\_size)

encoded\_block = mtf.encode(bwt.encode(block))

result += encoded\_block

return result

def main():

# Путь к файлу enwik7 (при необходимости измените путь)

file\_path = Path("Z:\prog\аисдик\compression\_test\_data\enwik7")

if not file\_path.exists():

print("Файл enwik7 не найден!")

return

with open(file\_path, "rb") as f:

data = f.read()

block\_sizes = [256, 512, 1024, 2048, 4096, 8192]

entropies = []

for bs in block\_sizes:

encoded\_data = bwt\_mtf\_encode(data, bs)

entropy\_value = compute\_entropy(encoded\_data)

entropies.append(entropy\_value)

print(f"Размер блока: {bs} байт, Энтропия: {entropy\_value:.4f} бит/символ")

plt.figure(figsize=(8, 6))

plt.plot(block\_sizes, entropies, marker="o")

plt.xlabel("Размер блока (байт)")

plt.ylabel("Энтропия (бит/символ)")

plt.title("Зависимость энтропии от размера блока (BWT+MTF) для enwik7")

plt.grid(True)

plt.savefig("entropy\_vs\_block\_size.png", dpi=300)

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

main()

# Заключение

В ходе выполнения лабораторной работы были изучены методы компрессии и декомпрессии данных, а также методы обработки данных, такие как BWT и MTF. Во время проведения экспериментов выяснилось, что наилучшим размером буфера для LZSS является 4096. Так же наилучший размер блока для BWT и MTF 2048. Среди всех алгоритмов сжатия лучше всего себя показали BWT+MTF+HA и LZSS. Оказалось, что jpeg, jpg, png форматы уже являются сжатыми, и попытки их дальнейшего сжатия лишь увеличили размер занимаемого места.

# Ссылка на репозиторий GitHub

<https://github.com/zamnisad/encoders>