

**МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра САПР**

**ОТЧЕТ
по лабораторной работе №1
по дисциплине «Алгоритмы и структуры данных»
Тема: «Алгоритмы сжатия без потерь»**

Студент гр. 2302

Фролов А. Э.

Преподаватель:

Пестерев Д. О.

Санкт-Петербург

2024

1. Подготовка данных.

Требуется подготовить тестовый набор данных:

- enwik7;
- Текст на русском языке, объемом не менее 1Мб;
- Черно-белое изображение;
- Изображение в оттенках серого;
- Цветное изображение.

Дополнительно к этому набору я добавил 3 черно-белых изображения, чтобы получить больше различных показателей энтропии при анализе.

Подготовленные изображения (1280x720). Показаны на Рис. 1.

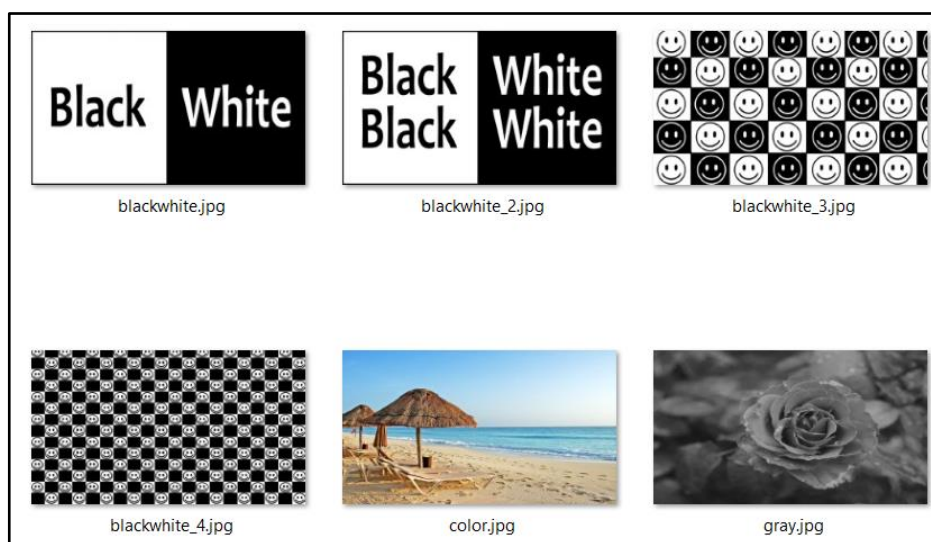


Рис. 1 – Тестовые изображения.

Полученные изображения требуется перевести в собственный raw формат, где один байт задает значение цветности (за исключением возможных мета-данных). Алгоритм следующий: каждому пикселю цветного изображения мы сопоставим 3 байта – значения компонент R, G, B (от 0 до 255), а каждому пикселю изображений в оттенках серого достаточно будет сопоставить один байт. Полученные файлы и их размеры приведены на Рис. 2.







 blackwhite.raw	25.09.2024 1:05	Файл "RAW"	901 КБ
 blackwhite_2.raw	05.11.2024 15:26	Файл "RAW"	901 КБ
 blackwhite_3.raw	05.11.2024 15:27	Файл "RAW"	901 КБ
 blackwhite_4.raw	05.11.2024 15:27	Файл "RAW"	901 КБ
 color.raw	25.09.2024 1:05	Файл "RAW"	2 701 КБ
 gray.raw	05.11.2024 3:44	Файл "RAW"	901 КБ

Рис. 2 – Размеры тестовых изображений в raw формате.

2. Теоретическая часть.

В теоретической части будет описана идея для реализации всех алгоритмов сжатия, а также необходимые метаданные (дополнительные данные, которые нужно хранить для возможности обратного преобразования) для каждого алгоритма.

Для дальнейшего анализа нам потребуется вычислять энтропию тестовых данных (см. формулу (1)).

$$H(x) = - \sum_i P_i * \log_2(P_i) \quad (1)$$

, где P_i – вероятность появления в тексте i -го символа алфавита.

Энтропия – показатель неопределенности информации в тексте. Чем выше энтропия, тем более сложный текст и тем сложнее предугадать следующий символ.

Энтропия измеряется в битах и также представляет собой теоретический предел сжатия данных (энтропийная оценка кодирования Шеннона). Иначе говоря, идеальное сжатие – это сжатие, при котором каждый символ закодированного текста в среднем кодируется количеством бит, равному энтропии исходного текста.

2.1. Алгоритм RLE (Run-Length Encoding).

Run-Length Encoding (кодирование длин серий) – метод сжатия текста, идея которого состоит в замене повторяющихся последовательностей на один символ и число его повторов.

Функция сжатия. Будем идти по тексту, разделяя его на блоки из последовательностей одинаковых символов и блоки последовательностей из разных символов (см. Рис. 3, а). Далее каждый блок повторяющихся символов будет заменяться на положительное число и один символ из блока, а блок из различных символов – на отрицательное число и соответствующую последовательность (см. Рис. 3, б). Таким образом мы сможем корректно декодировать любую закодированную последовательность.

Длина последовательностей в редких случаях будет превышать 127, поэтому для кодирования чисел будет рационально выделить наименьшую ячейку памяти – 1 байт (тип int8). Таким образом, мы сможем обрабатывать последовательности длиной до 127 символов. Сами символы будут кодироваться в том же виде, в котором они были считаны.

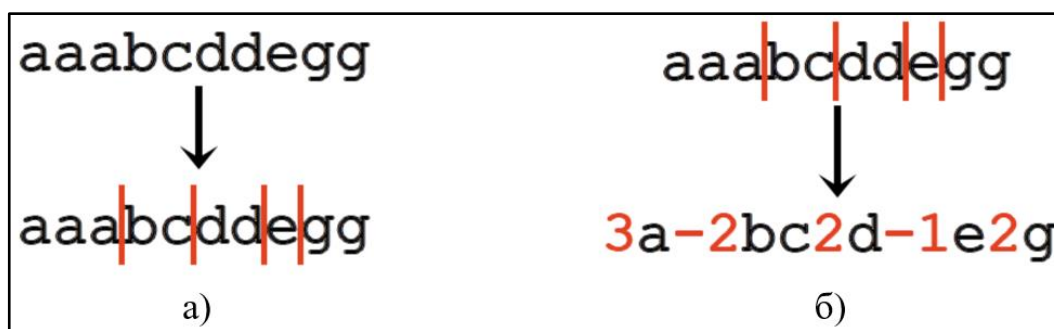


Рис. 3 – Алгоритм функции сжатия RLE.

Обратное преобразование. Будем считывать закодированную последовательность и параллельно собирать исходную строку с учетом вышеперечисленных правил.

Метаданные. Для того, чтобы вовремя завершить считывание при декодировании, в качестве метаданных придется сохранить длину исходной строки.

2.2. Алгоритм MTF (Move-To-Front).

Move-To-Front (Движение к началу) – метод для кодирования данных, разработанный для улучшения энтропийного кодирования.

Кодирование. Перед началом находим алфавит всех символов текста. Далее кодируем каждый символ текста его индексом (порядковым номером) в алфавите. После кодирования каждого символа выполняется сдвиг вправо всех символов алфавита до текущего индекса (крайний правый элемент переместится в начало алфавита).

Пример работы алгоритма для слова «рапата» показан на Рис. 4.

input_str	chars	output_arr	list
p		15	abcdefghijklmnopqrstuvwxyz
a		15 1	pabcdefghijklmnopqrstuvwxyz
n		15 1 14	apbcdefghijklmnopqrstuvwxyz
a		15 1 14 1	napbcdefghijklmnopqrstuvwxyz
m		15 1 14 1 14	anpbcdefghijklmnopqrstuvwxyz
a		15 1 14 1 14	manpbcdefghijklmnopqrstuvwxyz

Рис. 4 – Пример алгоритма функции кодирования MTF.

Декодирование. Берем полученный массив индексов и поочередно считываем элементы, собирая исходную строку из символов алфавита под соответствующими индексами. После обработки каждого индекса выполняется сдвиг алфавита.

Метаданные. Чтобы корректно декодировать текст, в качестве метаданных придется хранить алфавит, а также длину исходной строки, чтобы вовремя завершить считывание.

2.3. Алгоритм BWT (Burrows-Wheeler Transform).

Burrows-Wheeler Transform – это алгоритм, используемый для преобразования исходных данных с целью улучшения энтропийного сжатия.

Кодирование. В стандартной реализации мы сначала строим массив всех циклических сдвигов исходной строки (см. Рис. 5, а). Далее сортируем полученные строки по возрастанию (см. Рис. 5, б) и затем собираем строку из

последних символов отсортированного массива сдвигов (см. , в). Полученная строка и будет закодированным текстом. Дополнительно мы сохраняем индекс исходной строки в отсортированном массиве сдвигов.

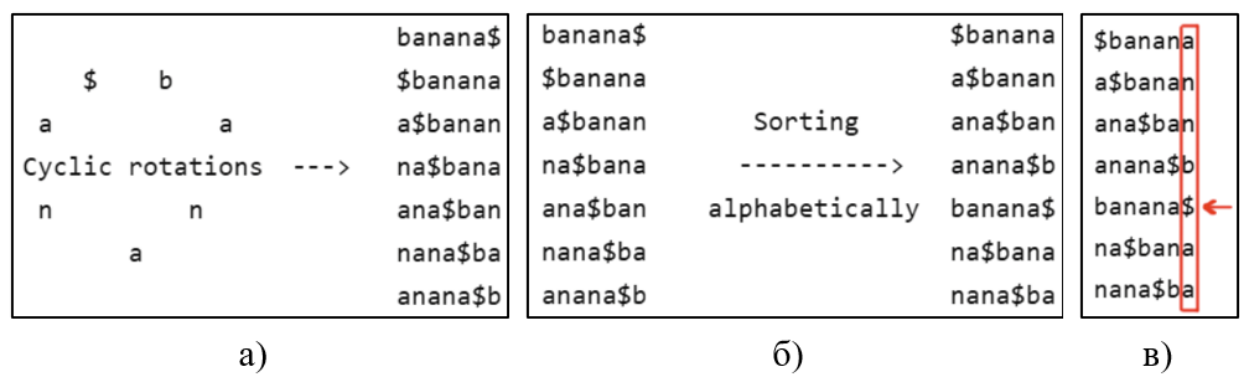


Рис. 5 – Алгоритм наивной реализации BWT.

Очевидно, что большие строки обрабатывать таким алгоритмом неэффективно с точки зрения расхода памяти. Поэтому теперь рассмотрим эффективную реализацию с использованием массива суффиксов.

Суффиксный массив — это лексикографически отсортированный массив всех суффиксов строки. Пример показан на Рис. 6.

Суффикс строки – это подстрока, которая является частью исходной строки, в которой обрезана часть символов слева.

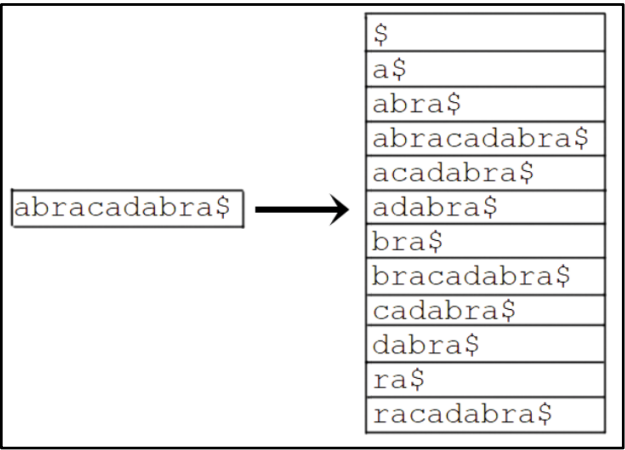


Рис. 6 – Пример суффиксного массива на примере строки «abracadabra».

Процесс получения массива суффиксов из исходной строки можно оптимизировать по потреблению памяти до $O(N)$. В своей программе я использую реализацию с потреблением памяти $\Theta(16N)$.

Итак, оказывается, что если в конец строки добавить лексикографически минимальный символ (например, «\$»), то полученный суффиксный массив будет соответствовать отсортированным перестановкам строки. Более того, оказывается, что для кодирования текста достаточно хранить только индексы суффиксов внутри исходной строки. Итак, имея массив индексов всех суффиксов, эквивалентный отсортированным перестановкам строки, мы можем проходиться по этому массиву и брать символ строки, предшествующий символу под индексом суффиксного массива (т.к. в алгоритме BWT закодированная строка содержит последние символы отсортированных перестановок, а индексы суффиксного массива – наоборот, первые символы).

Обратное преобразование. В случае наивной реализации алгоритма мы записываем закодированную строку в первый столбец матрицы $N \times N$ и сортируем каждую строку матрицы в алфавитном порядке. Далее сдвигаем все столбцы матрицы вправо и добавляем закодированную строку в первый столбец и опять сортируем строки. Повторяем эти шаги, пока матрица полностью не заполнится. Строка под сохраненным индексом будет исходной (см. Рис. 7).

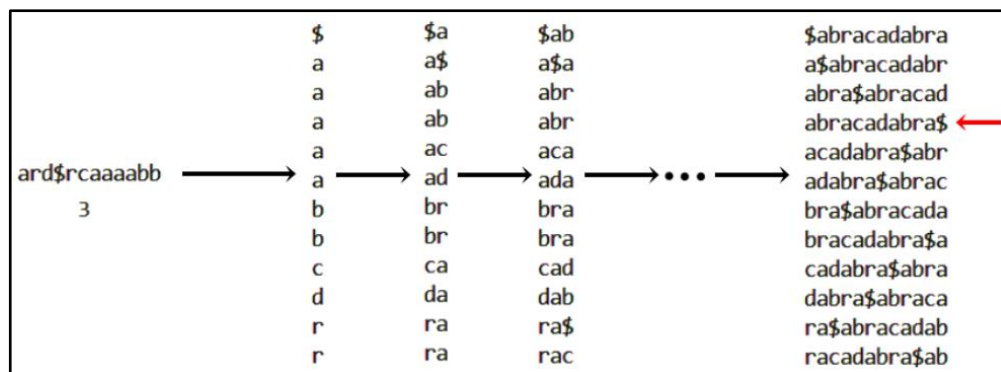


Рис. 7 - Алгоритм обратного преобразования наивной реализации BWT.

Для обратного преобразования в эффективной реализации сортируем закодированную строку лексикографически с отслеживанием индексов относительно несортированной строки. Назовем полученный массив индексов после сортировки массивом перестановок. Теперь посимвольно декодируем

строку. Начиная с сохраненного индекса, находим элемент по этому индексу в массиве перестановок (назовем его индексом №2) и добавляем его в выходную строку. Затем обновляем индекс на индекс №2 и повторяем цикл до полного декодирования (P.S. описание скорее всего непонятное, потому что я и сам не очень понимаю, как работает декодирование. Я просто описал, что происходит в моем коде, который, я когда-то давно написал).

Метаданные. Для реализации обратного преобразования нужно сохранить полученный индекс. Также, чтобы вовремя завершить считывание при декодировании, придется сохранить длину исходной строки.

2.4. Алгоритм НА (Huffman Algorithm).

Алгоритм Хаффмана – это жадный алгоритм оптимального префиксного кодирования алфавита с минимальной избыточностью.

Алгоритм функции сжатия. Берем алфавит текста и строим дерево Хаффмана. Для этого находим два наиболее редких символа из алфавита объединяем их в родительский узел. Родительскому узлу присваиваем вероятность, равную сумме вероятностей дочерних узлов. Повторяем процедуру, пока все символы из алфавита не будут задействованы. Проходимся по полученному дереву и сопоставляем символам бинарные коды (см. Рис. 8). Полученными кодами кодируем весь исходных текст.

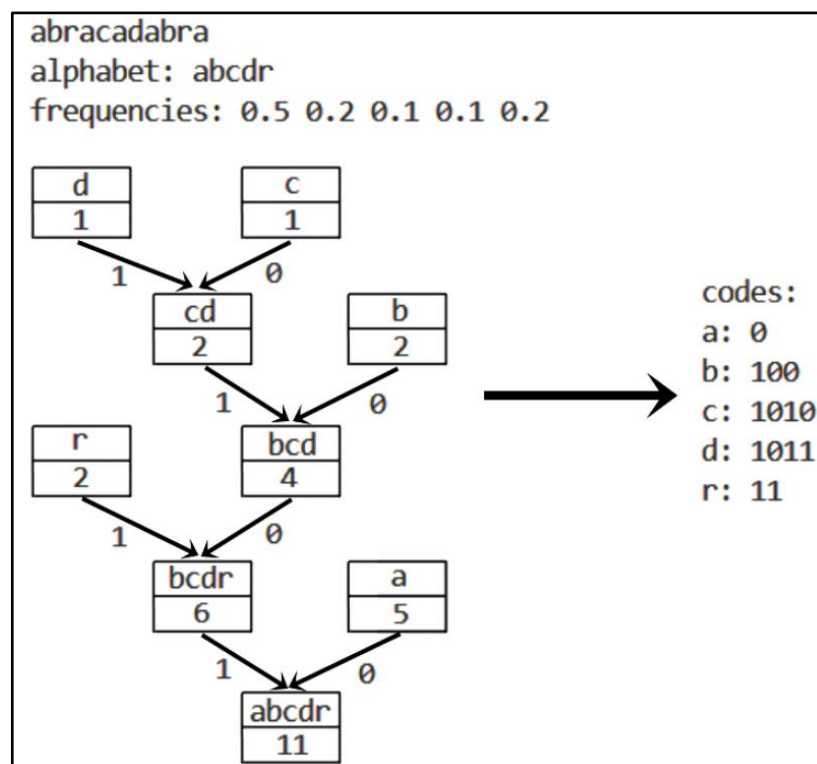


Рис. 8 – Пример сжатия алгоритмом Хаффмана на примере текста «abracadabra».

В данной реализации для корректного декодирования нам бы пришлось хранить коды для каждого символа алфавита, что достаточно неэффективно. Для решения такой проблемы используются канонические коды Хаффмана – коды Хаффмана, удовлетворяющие следующему условию (*).

ни один код не является начало любого другого кода. (*)

Такое условие обеспечивает однозначное декодирование и позволяет хранить в метаданных не сами коды, а только их длины. Итак, теперь опишу, как мы получаем канонические коды. Для начала строим дерево Хаффмана и сохраняем длины полученных кодов. Проходимся по полученным длинам по возрастанию и сопоставляем им еще не использованные коды, также проверяя условие (*).

Декодирование. Считываем длины канонических кодов и заново генерируем коды, удовлетворяющие условию (*). Далее просто проходимся по закодированной строке, пока не найдем соответствие какому-нибудь коду. Добавляем символ, соответствующий этому коду, в итоговую строку и продолжаем считывание.

Метаданные. Для обеспечения декодирования будем хранить алфавит и длины канонических кодов. Также, чтобы вовремя завершить считывание, будем хранить размер исходной строки.

Дополнительно отмечу, что кодирование больших текстов методом Хаффмана требует много памяти, поэтому существуют два подхода – адаптивное кодирование (автоматическое определение блоков текста для наиболее эффективного сжатия) и кодирование небольшими блоками фиксированной длины. В своей программе я использую второй вариант, т.к. он проще с точки зрения реализации, однако для профессионального использования, конечно, следует написать адаптивный алгоритм.

2.5. Алгоритм AC (Arithmetical Coding).

Arithmetical Coding (арифметическое кодирование) – эффективный метод энтропийного кодирования, обеспечивающий почти идеальную степень сжатия с точки зрения энтропийной оценки Шеннона (см. стр. 3).

Алгоритм функции сжатия. Берем алфавит и частоты появления символов. Сортирует частоты в обратном порядке и откладываем на отрезке от 0 до 1 (см. Рис. 9). Затем по каждому символу исходного текста и находим соответствующий интервал на отрезке. Затем разбиваем полученный интервал на аналогичные отрезки и повторяем процедуру. Результатом будет любое число типа double внутри последнего отрезка.

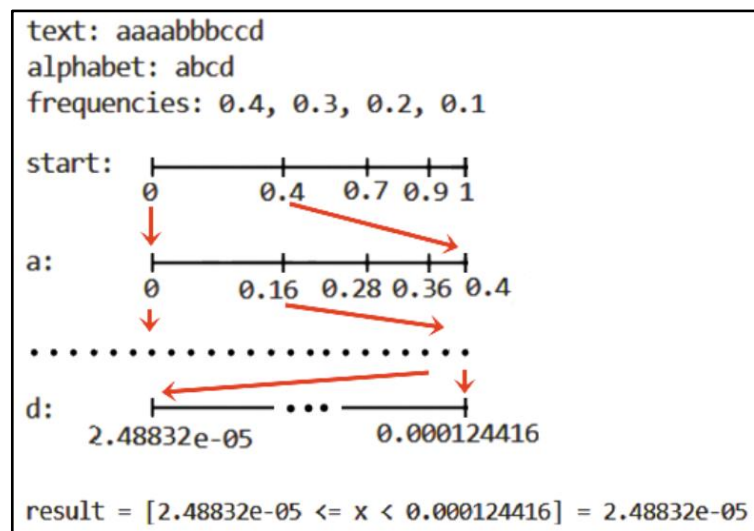


Рис. 9 – Пример алгоритма функции сжатия арифметического кодирования.

Очевидно, что большие текста обрабатывать таким образом не получится, т.к. итоговое число спустя всего пару символов не сможет уместиться в ячейку типа double. Поэтому теперь поговорим о том, как избежать этой проблемы. Способ, который я сейчас опишу, называется «Кодирование с помощью длинного целого». Итак, суть метода в том, что хранить итоговое число мы будем в бинарном виде. Вспомним, что дробное число от 0 до 1 представляется в бинарном виде путем деления пополам начиная с отрезка от 0 до 1 до тех пор, пока мы максимально не приблизимся к требуемому числу (см. Рис. 10). Теперь просто проведем аналогию на арифметическое кодирование. Каждый раз, когда мы двигаемся влево, в итоговое бинарное представление добавляем 0, иначе – 1. Как только мы оказались внутри какого-то интервала – растягиваем его обратно до отрезка от 0 до 1, также растягивая последнюю точку, в которой мы оказались. Таким образом, мы сможем обрабатывать очень большие текста.

	$x = 0.15$
$[0, 1]$	$x < 0.5 \Rightarrow 0$
$[0, 0.5]$	$x < 0.25 \Rightarrow 0$
$[0, 0.25]$	$x > 0.125 \Rightarrow 1$
$[0.125, 0.25]$	$x < 0.1875 \Rightarrow 0$
.....	

Рис. 10 - Пример перевода дробного числа в двоичный вид.

Декодирование. Возьмем исходный алфавит и частоты и воспроизведем отрезок от 0 до 1 с нужными интервалами. Далее просто двигаемся вглубь по этому отрезку согласно закодированному числу, как только окажемся внутри ключевого интервала – фиксируем полученный символ в итоговую строку и продолжаем движение.

Метаданные. Для воспроизведения начального отрезка требуется хранить алфавит и частоты символов. Также, чтобы вовремя завершить считывание, нужно хранить длину исходной строки.

2.6. Алгоритм LZ77.

LZ77 – словарный алгоритм сжатия без потерь, основная идея которого заключается в замене повторяющихся подстрок в исходной тексте.

Алгоритм функции сжатия. Определим размер окна предпросмотра (lookahead buffer) и окна поиска (search buffer). Будем идти по тексту и искать в окне предпросмотра максимальную последовательность, которая есть в окне поиска. Закодируем эту последовательность парой (offset, length), где offset – расстояние от текущего положения до найденной последовательности, а length – длина найденной последовательности. Повторяем цикл до полного прохождения по строке. В случае, если в окне поиска не найдена ни одна последовательность из окна предпросмотра, то просто кодируем текущий

символ парой $(0, 0)$ с, где с – текущий символ. Пример кодирования для строки «aaaabbbccd» приведен на Рис. 11.

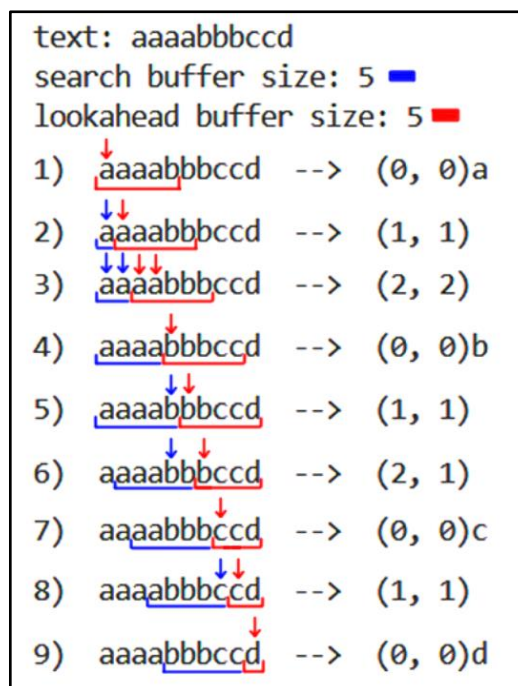


Рис. 11 – Пример алгоритма сжатия LZ77.

Алгоритм обратного преобразования. Пройдемся последовательно по закодированным парам, параллельно собирая исходную строку. В случае обработки пары $(0, 0)$ с будем просто добавлять в итоговую строку символ с. Иначе, выполним указанный сдвиг влево (offset) и считаем строку нужной длины (length) внутри текущей полученной строки. Действуя таким образом, мы корректно декодируем любую строку. Пример обратного преобразования для строки «aaaabbbccd» приведен на Рис. 12.

1)	(0, 0)	a	-->	a
2)	(1, 1)		-->	aa
3)	(2, 2)		-->	aaaa
4)	(0, 0)	b	-->	aaaab
5)	(1, 1)		-->	aaaabb
6)	(2, 1)		-->	aaaabbb
7)	(0, 0)	c	-->	aaaabbbc
8)	(1, 1)		-->	aaaabbbcc
9)	(0, 0)	d	-->	aaaabbbccd

Рис. 12 – Пример обратного преобразования LZ77.

Метаданные. Для того, чтобы вовремя завершить считывания, понадобится хранить длину исходной строки или количество полученных пар. В своей программе я выбрал первый способ.

3. Практическая часть.

В практической части будет исследована эффективность написанных алгоритмов на тестовых данных.

Перед началом зафиксируем показатели энтропии тестовых данных.

Файл	blackwhite	blackwhite_2	blackwhite_3	blackwhite_4	gray	color
Энтропия	1.3811	1.9319	2.9143	3.903	6.975	7.6193
Файл	russian_1mb	enwik7				
Энтропия	4.8155	5.101				

Показатели в графическом виде отображены на Рис. 13.

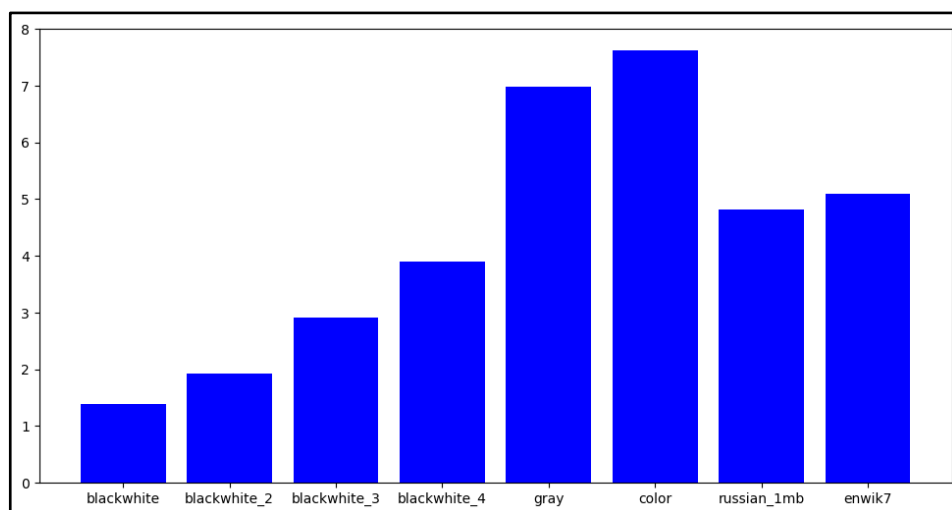


Рис. 13 – Показатели энтропии для тестовых данных.

Таким образом, дополнительные черно-белые изображения позволили получить более плавные показатели энтропии, что позволит более точно проанализировать сжатие.

3.1. Кодирование длин серий (RLE).

Полученные коэффициенты сжатия приведены ниже.

Файл	blackwhite	blackwhite_2	blackwhite_3	blackwhite_4	gray	color
к. сжатия	11.747	5.386	2.23	1.758	1.173	0.99
Файл	russian_1mb	enwik7				
к. сжатия	0.99	0.965				

Зависимость коэффициентов сжатия от энтропии отображена на Рис. 14.

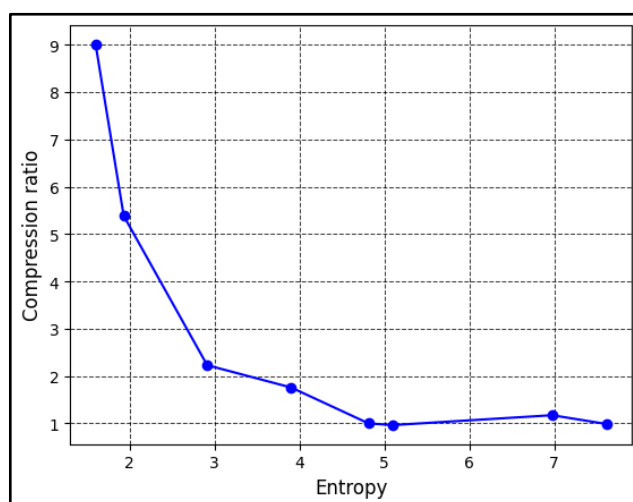


Рис. 14 – Результаты сжатия методом RLE.

Как и ожидалось, чем меньше энтропия текста, тем лучше сжатие (за исключением небольших отклонений). По графику также можно заметить практически экспоненциальную зависимость. Положительные результаты получились у всех изображений, кроме цветного.

Отмечу, что сила сжатия зависит не только от энтропии, поэтому зависимость силы сжатия от энтропии не всегда будет иметь такой вид.

3.2. Алгоритм Хаффмана (НА).

Размер блоков внутри текста, к которым применяется алгоритм, я сделал статичным и равным 10000 символов.

Полученные коэффициенты сжатия приведены ниже.

Файл	blackwhite	blackwhite_2	blackwhite_3	blackwhite_4	gray	color
к. сжатия	4.225	3.667	2.456	1.922	1.205	1.071
Файл	russian_1mb	enwik7				
к. сжатия	2.927	1.551				

Зависимость коэффициентов сжатия от энтропии отображена на Рис. 15.

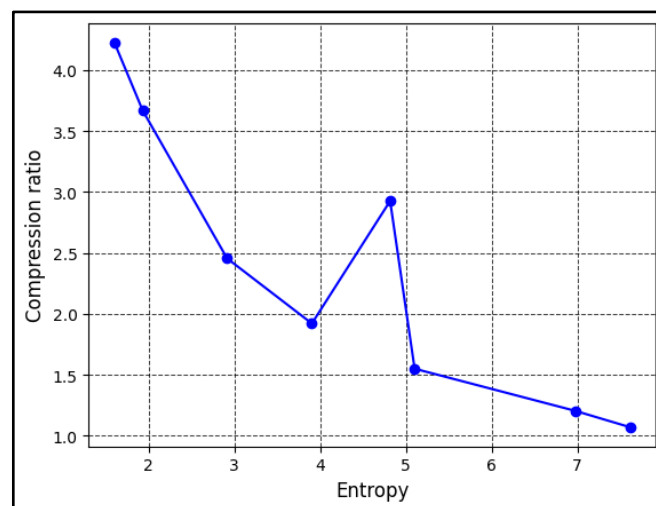


Рис. 15 – Результаты сжатия методом Хаффмана.

Как и ожидалось, чем меньше энтропия текста, тем лучше сжатие (за исключением отклонения у русского текста). В сравнении с методом RLE, тут спад графика более плавный. Положительные результаты получились у всех

тестовых данных. Наиболее сильно были сжаты черно-белые изображения и русский текст.

Отмечу, что сила сжатия зависит не только от энтропии, поэтому зависимость силы сжатия от энтропии не всегда будет иметь такой вид.

3.3. Арифметическое кодирование (АС).

Полученные коэффициенты сжатия приведены ниже.

Файл	blackwhite	blackwhite_2	blackwhite_3	blackwhite_4	gray	color
к. сжатия	3.409	2.852	2.186	1.672	0.997	0.923
Файл	russian_1mb	enwik7				
к. сжатия	2.508	1.331				

Зависимость коэффициентов сжатия от энтропии отображена на Рис. 16.

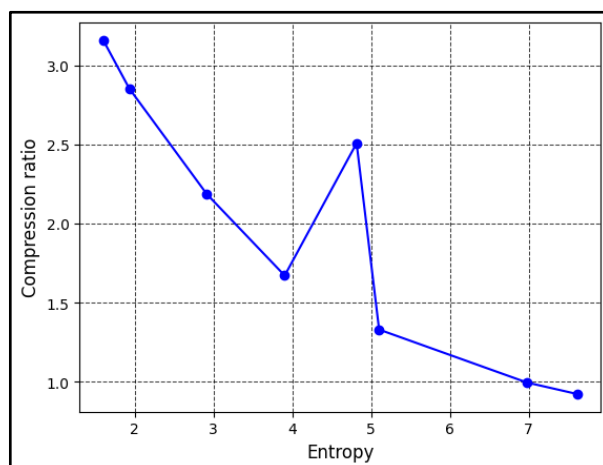


Рис. 16 – Результаты сжатия методом арифметического кодирования.

Как и ожидалось, чем меньше энтропия текста, тем лучше сжатие (за исключением отклонения у русского текста). Положительные результаты получились у всех тестовых данных, кроме цветного и серого изображений. Отметим, что вид графика практически такой же, как тот, что получен методом Хаффмана. В целом, результаты сжатия всех тестовых данных уступают результатам, полученным методом Хаффмана, однако это компенсируется гораздо большей скоростью выполнения.

Отмечу, что сила сжатия зависит не только от энтропии, поэтому зависимость силы сжатия от энтропии не всегда будет иметь такой вид.

3.4. Компрессор BWT+ RLE.

Полученные коэффициенты сжатия приведены ниже.

Файл	blackwhite	blackwhite_2	blackwhite_3	blackwhite_4	gray	color
к. сжатия	10.569	5.549	2.271	2.265	1.078	1.539
Файл	russian_1mb	enwik7				
к. сжатия	1.852	1.726				

Зависимость коэффициентов сжатия от энтропии отображена на Рис. 17.

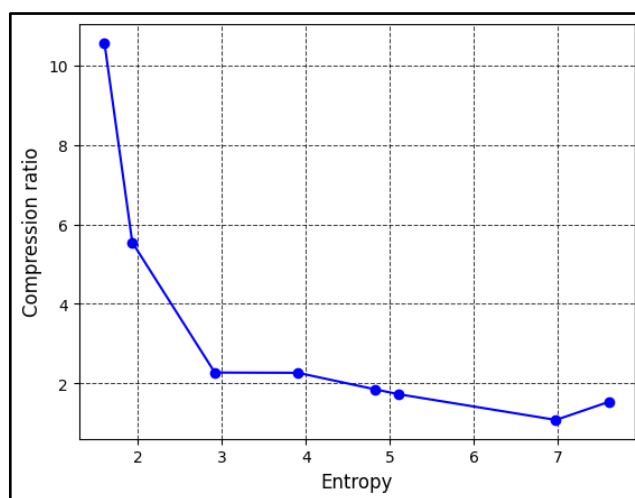


Рис. 17 – Результаты сжатия компрессором BWT+RLE.

Как и ожидалось, чем меньше энтропия текста, тем лучше сжатие (за исключением небольших отклонений). Положительные результаты получились у всех тестовых данных. Вид графика очень близок к тому, что получен методом RLE, однако результаты значительно лучше. Выходит, что метод BWT сильно влияет на степень сжатия.

Отмечу, что сила сжатия зависит не только от энтропии, поэтому зависимость силы сжатия от энтропии не всегда будет иметь такой вид.

3.5. Компрессор BWT+MTF +HA.

Полученные коэффициенты сжатия приведены ниже.

Файл	blackwhite	blackwhite_2	blackwhite_3	blackwhite_4	gray	color
к. сжатия	6.125	5.087	3.185	3.053	2.725	1.741

Файл	russian_1mb	enwik7
к. сжатия	5.632	3.208

Зависимость коэффициентов сжатия от энтропии отображена на Рис. 18.

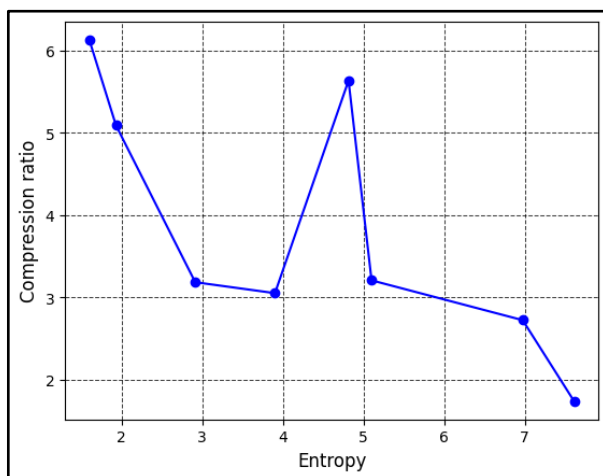


Рис. 18 – Результаты сжатия компрессором BWT+MTF+HA.

Как и ожидалось, чем меньше энтропия текста, тем лучше сжатие (за исключением отклонений у текстов). Положительные результаты получились у всех тестовых данных. Результаты для текстов получились отличные, а коэффициент сжатия русского текста практически соразмерен с коэффициентом сжатия черно-белого изображения. Для сжатия текстов это лучший компрессор из всех написанных.

Отмечу, что сила сжатия зависит не только от энтропии, поэтому зависимость силы сжатия от энтропии не всегда будет иметь такой вид.

3.6. Компрессор BWT+MTF+AC.

Полученные коэффициенты сжатия приведены ниже.

Файл	blackwhite	blackwhite_2	blackwhite_3	blackwhite_4	gray	color
к. сжатия	5.351	4.271	2.668	2.733	2.184	1.488
Файл	russian_1mb	enwik7				
к. сжатия	4.725	2.633				

Зависимость коэффициентов сжатия от энтропии отображена на Рис. 19.

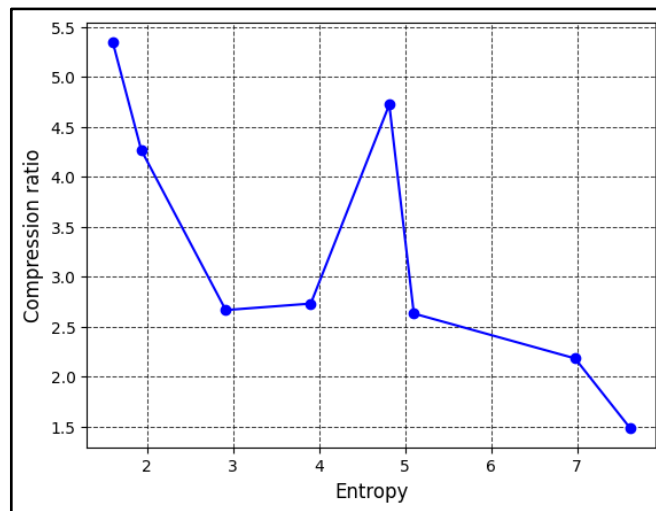


Рис. 19 – Результаты сжатия компрессором BWT+MTF+AC.

Как и ожидалось, чем меньше энтропия текста, тем лучше сжатие (за исключением отклонений у текстов). Положительные результаты получились у всех тестовых данных. Вид графика практически такой же, как у компрессора BWT+MTF+HA, однако результаты хуже. Причина в том, что метод Хаффмана в моей реализации сжимает лучше, чем метод арифметического кодирования.

Отмечу, что сила сжатия зависит не только от энтропии, поэтому зависимость силы сжатия от энтропии не всегда будет иметь такой вид.

3.7. Компрессор BWT+MTF+RLE+HA.

Полученные коэффициенты сжатия приведены ниже.

Файл	blackwhite	blackwhite_2	blackwhite_3	blackwhite_4	gray	color
к. сжатия	15.111	7.95	3.185	2.753	1.944	1.783
Файл	russian_1mb	enwik7				
к. сжатия	4.55	2.715				

Зависимость коэффициентов сжатия от энтропии отображена на Рис. 20.

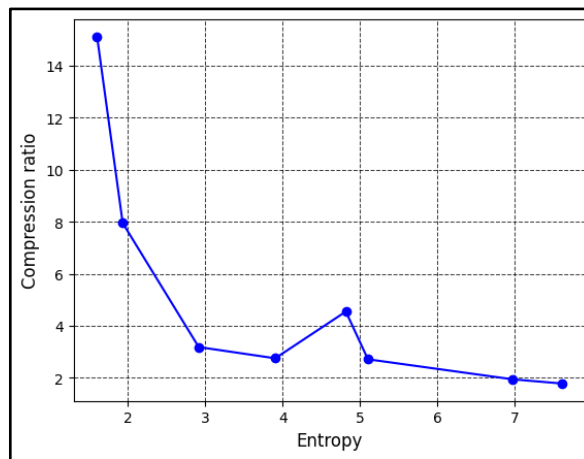


Рис. 20 – Результаты сжатия компрессором BWT+MTF+RLE+HA.

Как и ожидалось, чем меньше энтропия текста, тем лучше сжатие (за исключением отклонений у текстов). Положительные результаты получились у всех тестовых данных. Результаты для текстов получились очень хорошие, а коэффициент сжатия первого черно-белого изображения является наилучшим.

Отмечу, что сила сжатия зависит не только от энтропии, поэтому зависимость силы сжатия от энтропии не всегда будет иметь такой вид.

3.8. Компрессор BWT+MTF+RLE+AC.

Полученные коэффициенты сжатия приведены ниже.

Файл	blackwhite	blackwhite_2	blackwhite_3	blackwhite_4	gray	color
к. сжатия	12.674	6.798	2.736	2.411	1.63	1.598
Файл	russian_1mb	enwik7				
к. сжатия	3.828	2.27				

Зависимость коэффициентов сжатия от энтропии отображена на Рис. 21.

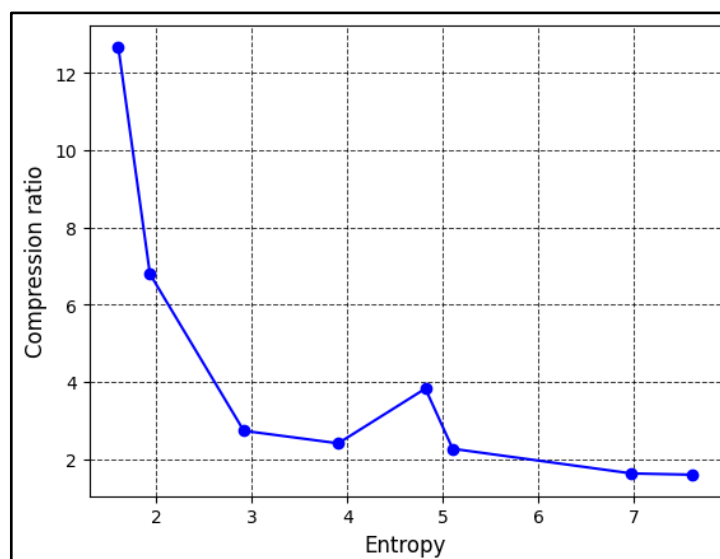


Рис. 21 – Результаты сжатия компрессором BWT+MTF+RLE+AC.

Как и ожидалось, чем меньше энтропия текста, тем лучше сжатие (за исключением отклонения у русского текста). Положительные результаты получились у всех тестовых данных. Вид графика практически такой же, как у компрессора BWT+MTF+RLE+HA, однако результаты хуже. Причина в том, что метод Хаффмана в моей реализации сжимает лучше, чем метод арифметического кодирования.

Отмечу, что сила сжатия зависит не только от энтропии, поэтому зависимость силы сжатия от энтропии не всегда будет иметь такой вид.

3.9. Алгоритм LZ77.

Размер окна поиска в моей реализации я выбрал статичным и равным 32Кб, а размер окна предпросмотра – 128 байт (128 символов).

Полученные коэффициенты сжатия приведены ниже.

Файл	blackwhite	blackwhite_2	blackwhite_3	blackwhite_4	gray	color
к. сжатия	10.475	6.265	2.449	1.909	1.552	1.062
Файл	russian_1mb	enwik7				
к. сжатия	3.57	1.89				

Зависимость коэффициентов сжатия от энтропии отображена на Рис. 22 - Результаты сжатия алгоритмом LZ77..

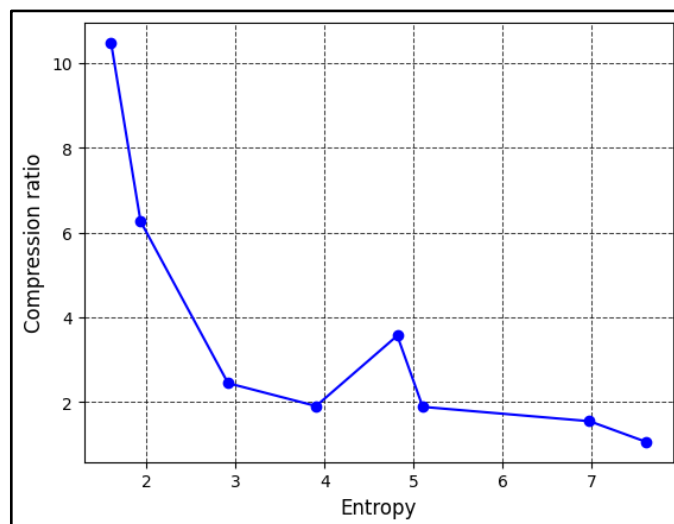


Рис. 22 - Результаты сжатия алгоритмом LZ77.

Хоть данный алгоритм и не зависит от энтропии, но в данном случае зависимость прослеживается. Так произошло потому, что тестовые данные с небольшой энтропией содержат больше одинаковых подстрок, что и нужно алгоритму LZ77 для хорошего сжатия. Положительные результаты получились у всех тестовых данных, кроме цветного изображения. Вероятно, это потому, что в цветном изображении алгоритм чаще всего будет ссылаться на один пиксель, чем на несколько. Более того, offset и length из-за большого размера окна поиска требуют в сумме 3 байта памяти, в то время как один пиксель в цветном изображении требует также 3 байта. Таким образом, при кодировании цветного изображения мы будем чаще не уменьшать размер файла, а лишь увеличивать за счет случаев, когда появляется новый пиксель, который не был замечен в окне поиска.

Отметим, что зависимость на графике напоминает ту, что получилась в алгоритме RLE, за исключением текстов – тут их результаты сжатия гораздо лучше.

3.10. Компрессор LZ77+HA.

Сжатие алгоритм Хаффмана в связке с LZ77 работает непозволительно долго на текстах, из-за чего я не тестировал алгоритм на них. Вероятно, это

как-то связано с энтропией выходного текста после работы LZ77, из-за чего канонические коды получаются большими и требуют большего времени на их поиск.

Файл	blackwhite	blackwhite_2	blackwhite_3	blackwhite_4	gray	color
к. сжатия	12.615	7.063	2.731	2.229	1.747	1.276
Файл	russian_1mb	enwik7				
к. сжатия	-	-				

Зависимость коэффициентов сжатия от энтропии отображена на Рис. 23
 - Результаты сжатия компрессором LZ77+НА..

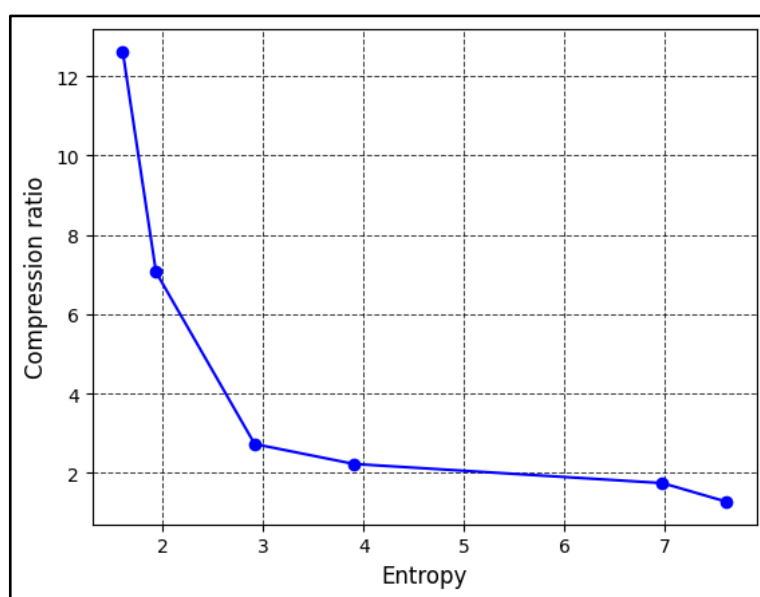


Рис. 23 - Результаты сжатия компрессором LZ77+НА.

Таким образом, компрессор работает немного лучше, чем обычный алгоритм LZ77, однако по какой-то причине алгоритм Хаффмана очень долго обрабатывает последовательность, полученную после алгоритма LZ77 на тестовых данных russian_1mb.txt, enwik7.txt. Причина, к сожалению, не была найдена, но отмечу, что это может быть связано с моей реализацией алгоритма Хаффмана, который использует самое наивное нахождение канонических кодов.

3.11. Обобщенный вывод.

Обобщим результаты в одну таблицу. Результат приведен ниже.

Файл Компр-р	bw	bw_2	bw_3	bw_4	gray	color	russian_1mb	enwik7
RLE	11.747	5.386	2.23	1.758	1.173	0.99	0.99	0.965
HA	4.225	3.667	2.456	1.922	1.205	1.071	2.927	1.551
AC	3.409	2.852	2.186	1.672	0.997	0.923	2.508	1.331
BWT+RLE	10.569	5.549	2.271	2.265	1.078	1.539	1.852	1.726
BWT+MTF+ HA	6.125	5.087	3.185	3.053	2.725	1.741	5.632	3.208
BWT+MTF+ AC	5.351	4.271	2.668	2.733	2.184	1.488	4.725	2.633
BWT+MTF+ RLE+HA	15.111	7.95	3.185	2.753	1.944	1.783	4.55	2.715
BWT+MTF+ RLE+AC	12.674	6.798	2.736	2.411	1.63	1.598	3.828	2.27
LZ77	10.475	6.265	2.449	1.909	1.552	1.062	3.57	1.89
LZ77+HA	12.615	7.063	2.731	2.229	1.747	1.276	-	-

Таким образом, в моей реализации можно выделить следующее:

- Наилучший компрессор для сжатия текста - BWT+MTF+HA;
- Лучшие компрессоры для сжатия изображений - BWT+MTF+HA и BWT+MTF+RLE+HA;
- Лучший алгоритм сжатия (не совокупность алгоритмов) – LZ77.

4. Выводы.

В результате данной лабораторной работы были изучены и реализованы на практике основные алгоритмы сжатия без потерь. Все алгоритмы были проверены на тестовых данных. Все результаты были зафиксированы, основные выводы сделаны.

Благодаря проделанной работе мы убедились, что эффективность написанных алгоритмов сжатия сильно зависит от энтропии. Наилучшими компрессорами в моей реализации оказались BWT+MTF+HA и BWT+MTF+RLE+HA.

Стоит отметить, что написанный алгоритм Хаффмана не подходит для профессионального использования, т.к. не является адаптивным и не эффективен по скорости (сжатие и обратное преобразование на enwik7 занимает около 10 минут на процессоре Intel Pentium Gold 7505). Однако отмечу, что алгоритм использует простейший наивный перебор для нахождения канонических кодов, поэтому он имеет перспективы к сильной оптимизации.

Арифметическое кодирование напротив, выполняется достаточно быстро (сжатие и обратное преобразование на enwik7 занимает уже около 10 секунд) и является лучшим выбором в соотношении силы сжатия и скорости.

5. Ссылка на программу.

<https://github.com/afrlfff/university-student-works/tree/master/algorithms-and-data-structures/lab4-text-compressors>