**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

**Отчёт**

**по лабораторной работе №1**

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

**Тема: сравнение методов сжатия данных**.

|  |  |  |
| --- | --- | --- |
| Студентка гр. 3354 |  | Чикарёва М.Д. |
| Преподаватель |  | Пестерев Д. О. |

**Санкт-Петербург**

**202****5**

**Цель лабораторной работы**: реализовать алгоритмы методов сжатия данных, исследовать зависимость энтропии от размера блоков, на которые разбивается текст, подаваемый на вход BWT+MTF, для enwik7, чтобы сделать вывод об оптимальном размере блока, исследовать зависимость коэффициента сжатия от размера буфера для алгоритма LZ77 и сделать вывод об оптимальном размере буфера, а также собрать на основе алгоритмов компрессоры.

**Представленные в работе алгоритмы методов сжатия** : HA, BWT, MTF, RLE, LZ77, LZ78.

**Представленные в работе компрессоры**: HA, Run-length encoding (RLE), BWT + RLE, BWT + MTF + HA, BWT + MTF + RLE + HA, LZ77, LZ77 + HA, LZ78, LZ78 + HA.

.

# Теоретическая часть

**HA (Huffman Algorithm)**

1. Описание алгоритма.

Кодирование Хаффмана - алгоритм сжатия данных без потерь. Этот метод использует частоту (или вероятность) символов в данных для создания оптимального префиксного кода. Основной идеей является создание дерева, где более частые символы получают более короткие коды, а редкие - более длинные.

**Основные шаги алгоритма:**

1. Подсчет частот: Для каждого символа в исходном сообщении подсчитывается количество его появлений (частота).
2. Создание дерева Хаффмана:

- Создается список узлов, каждый из которых представляет символ и его частоту. Эти узлы становятся листьями будущего дерева Хаффмана.

- Пока в списке больше одного узла:

\* Выбираются два узла с наименьшей частотой.

\* Создается новый родительский узел, частота которого равна сумме частот выбранных дочерних узлов. Выбранные узлы становятся дочерними узлами нового родительского узла.

\* Новый родительский узел добавляется в список узлов, а выбранные дочерние узлы удаляются из списка.

1. Назначение кодов:

- Проходим дерево Хаффмана от корня к каждому листу (символу).

- Каждому левому ребру назначается значение ‘0’, а каждому правому ребру - значение ‘1’.

- Кодовое слово для символа получается путем конкатенации значений ребер на пути от корня к этому символу.

**Пример: abracadabra:**

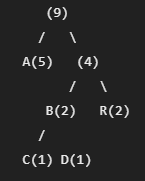
- a: 5

- b: 2;

- c: 1;

- d: 1;

- r: 2.





Из этого:

- A: 0;

- B: 10;

- C: 100;

- D: 101;

- R: 11.

1. Асимптотическая временная и пространственная сложность

**Временная сложность**: O(nlogn), где:

- Подсчет частот: O(n), где n - длина входной строки.

- Создание дерева Хаффмана: O(n log n), где n - количество уникальных символов. Это связано с использованием очереди с приоритетами (например, heap) для выбора узлов с наименьшей частотой.

- Назначение кодов: O(n), где n - количество уникальных символов. По сути, мы обходим дерево.

Пространственная сложность: O(n).

**BWT (Burrows-Wheeler Transform)**

1. Описание алгоритма

Преобразование Барроуза-Уиллера (BWT) - это обратимое преобразование текста, используемое для предварительной обработки данных перед сжатием. Само по себе оно не сжимает данные, но перегруппировывает символы так, чтобы одинаковые символы образовывали длинные последовательности. Это делает данные более пригодными для последующего применения алгоритмов сжатия, таких как Move-to-Front (MTF) и Run-Length Encoding (RLE).

**Основные шаги алгоритма:**

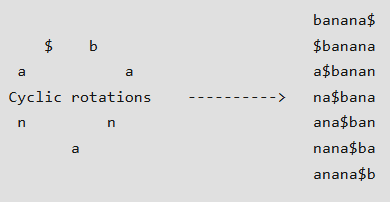
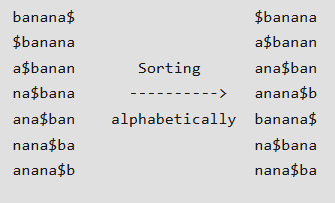
- Подготовка строки: К входной строке добавляется специальный символ окончания (обычно $), чтобы избежать неоднозначности.

- Циклические сдвиги: Создаются все возможные циклические сдвиги (вращения) исходной строки.

- Сортировка: Полученные циклические сдвиги сортируются лексикографически (в алфавитном порядке).

- Выбор последнего столбца: В качестве выходной строки выбирается последний столбец таблицы преобразования и номер строки, совпадающей с исходной.

**Пример:** возьмём «banana$».

BWT(banana$) = a**nn**b$**aa**

1. Асимптотическая временная и пространственная сложность.

**Временная сложность:**

- Циклические сдвиги: O(n2), где n - длина строки.

- Сортировка: O(n2logn), если используется алгоритм сортировки сравнения (например, mergesort, quicksort) для строк.

**Общая временная сложность:** O(n2logn)

**Пространственная сложность:**

- Циклические сдвиги: O(n2) (хранение всех сдвигов)

- BWT: O(n)

**Общая пространственная сложность:**O(n2)

**MTF (Move-to-Front)**

1. Описание алгоритма.

Алгоритм Move-to-Front (MTF) - это простой и эффективный алгоритм кодирования, используемый для сжатия данных, в которых часто встречаются повторяющиеся символы. Он работает, поддерживая список (или массив) символов, который постоянно обновляется. Когда встречается символ, он кодируется индексом его позиции в списке, а затем перемещается в начало списка.

**Основные шаги алгоритма:**

- Инициализация списка: Создается список, содержащий все символы из алфавита.

- Обработка символов: Считывается по одному символу за раз из входной строки. Выводится позиция, в которой этот символ отображается в списке. Символ перемещается в начало списка, данный процесс повторяется до тех пор, пока не будут получены индексы для всех введённых символов.

**Пример.**

Для примера возьмём слово **"evidence"**.

Список изначально содержит буквы из английского алфавита по порядку.

**input\_str chars output\_arr list**

e 4 abcd**e**fghijklmno**p**qrstuvwxyz

v 4 21 e**a**bcdfghijklmnopqrstu**v**wxyz

i 4 21 9 veabcdfgh**i**jklm**n**opqrstwxyz

d 4 21 9 6 iveabc**d**fghjklm**n**opqrstwxyz

e 4 21 9 6 3 div*e*abcfghjklm**n**opqrstwxyz

n 4 21 9 6 3 14 edivabcfghjklm**n**opqrstwxyz

c 4 21 9 6 3 14 7 nedivab**c**fghjklmopqrstwxyz

e 4 21 9 6 3 14 7 cnediveabcfghjklmopqrstwxyz

Input: evidence

Output: 4 21 9 6 3 14 7 2

1. Асимптотическая временная и пространственная сложность.

**Временная сложность:**

- При каждой операции для поиска индекса символа требуется O(m) (где m — количество уникальных символов). В худшем случае, для каждого символа необходимо пройтись по всему списку, чтобы найти его индекс.

**Общая временная сложность для строки длиной n:** O(n·m)

**Пространственная сложность:** O(m), где m - размер алфавита.

**RLE (Run-Length Encoding)**

1. Описание алгоритма.

Алгоритм Run-Length Encoding (RLE) — это простой метод сжатия данных, который эффективен при сжатии последовательностей повторяющихся символов. Основная идея RLE заключается в замене серий одинаковых символов на один символ и количество его повторений. Это особенно полезно для данных, содержащих длинные серии повторяющихся символов.

**Основные шаги алгоритма:**

- Инициализация: Начинаем с пустой строки для хранения результирующего закодированного текста.

- Перебор символов:

\* Считать текущий символ и его количество повторений (порядка).

\* Если следующий символ равен текущему, увеличить счетчик.

\* Если следующий символ отличается, записать текущий символ и его количество в результирующую строку, а затем перейти к следующему символу.

 - Результат: После завершения перебора записать оставшийся символ и его количество в результирующую строку.

**Пример:** пусть есть входная последовательность «AAABBBBCCDAAAAA».

Найдём все повторы (прогоны символов):

\* «AAA» - прогон символа «А», длина - 3.

\* «BBBB» - прогон символа «B», длина - 4.

\* «CC» - прогон символа «C», длина - 2.

\* «D» - прогон символа «D», длина - 1.

\* «AAAAA» - прогон символа «А», длина - 5.

Тогда закодированная последовательность: (A, 3), (B, 4), (C, 2), (D, 1), (A, 5) или 3A4B2C1D5A.

Однако данный алгоритм не выгоден тогда, когда в данных мало прогонов, потому что RLE может увеличить размер данных. Например, “ABCDEF” -> “A1B1C1D1E1F1”.

1. Асимптотическая временная и пространственная сложность.

**Временная сложность:** О(n), где n - длина входной последовательности.

**Пространственная сложность:** В худшем случае (когда нет прогонов) размер закодированных данных может быть 2n (если каждый символ кодируется как пара). В лучшем случае (когда вся последовательность состоит из одного символа) размер закодированных данных будет O(1). В среднем, пространственная сложность зависит от количества и длины прогонов.

**LZ77 (Lempel-Ziv 77)**

1. Описание алгоритма.

LZ77 (Lempel-Ziv 77) - это алгоритм сжатия данных без потерь, который является одним из фундаментальных алгоритмов семейства LZ. Он работает, заменяя повторяющиеся последовательности данных (фразы) ссылками на предыдущие вхождения этих последовательностей в уже обработанных данных. LZ77 использует “скользящее окно”, которое состоит из двух частей: “окна поиска” (search buffer) и “окна просмотра вперед” (look-ahead buffer).

**Основные шаги алгоритма:**

- Инициализация: Устанавливается размер окна поиска (search buffer size) и окна просмотра вперед (look-ahead buffer size). Окно поиска изначально пустое, а окно просмотра вперед содержит начало входных данных.

- Поиск соответствий: Находится самая длинная строка, начинающаяся в начале окна просмотра вперед, которая также встречается в окне поиска.

- Кодирование: Закодировать найденную строку как триплет (сдвиг, длина, следующий символ):

\* Сдвиг (Offset): Расстояние назад от конца окна поиска до начала найденной строки.

\* Длина (Length): Длина найденной строки.

\* Следующий символ (Next Symbol): Символ в окне просмотра вперед, следующий за найденной строкой. Если строка в окне просмотра вперед не найдена в окне поиска, сдвиг и длина равны 0, а следующий символ – это первый символ окна просмотра вперед.

- Сдвиг окна: Сдвинуть скользящее окно вправо на длину найденной строки плюс 1 (на длину строки + следующий символ). Новые символы добавляются в конец окна просмотра вперед из входных данных. Строка, вышедшая из окна поиска, отбрасывается.

Повторять описанные действия, пока не будут обработаны все входные данные.

**Пример:** возьмём строку «abracadabra».

| **Шаг** | **Строка** | **Окно поиска** | **Окно просмотра** | **Совпадение** | **Закодированная посл-ность** | **Примечание** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | abracadabra | [ ] | [abra] | Нет | (0, 0, a) | Буфер пуст |
| 2 | a bracadabra | [a ] | [brac] | Нет | (0, 0, b) | В буфере нет ‘b’ |
| 3 | abr acadabra | [ab ] | [raca] | Нет | (0, 0, r) | В буфере нет ‘r’ |
| 4 | abra cadabra | [abr ] | [acad] | Нет | (0, 0, a) | В буфере нет ‘a’ |
| 5 | abrac adabra | [abra ] | [cada] | Нет | (0, 0, c) | В буфере нет ‘c’ |
| 6 | abraca dabra | [abrac] | [dabr] | Нет | (0, 0, d) | В буфере нет ‘d’ |
| 7 | abracad abra | [bracad] | [abra] | abra (offset 4, len 4) | (4, 4, None) | В буфере найдена последова-  тельность “abra”.  Конец строки |

**Пояснения:**

- Окно поиска (буфер): Содержит последние 5 символов.

- Окно просмотра вперед: Содержит следующие символы, которые мы пытаемся закодировать.

- Совпадение: Самая длинная последовательность в окне просмотра вперед, найденная в окне поиска. Обозначается как “offset, длина”.

- Закодированная последовательность: Триплет (смещение, длина, следующий символ), где:

\* смещение: Расстояние назад от конца окна поиска до начала найденного совпадения.

\* длина: Длина найденного совпадения.

\* следующий символ: Первый символ в окне просмотра вперед, который не входит в совпадение, либо None, если совпадение доходит до конца строки.

**Результат кодирования (список триплетов):**

[(0, 0, a), (0, 0, b), (0, 0, r), (0, 0, a), (0, 0, c), (0, 0, d), (4, 4, None)].

1. Асимптотическая временная и пространственная сложность.

**Временная сложность:** O(n · m · k), где n - длина входной строки, m - размер окна поиска, а k - средняя длина строки, которую мы пытаемся сопоставить. Поиск самой длинной совпадающей строки может занять O(m · k) для каждого символа во входной строке.

**Пространственная сложность:**

- Хранение выходной строки: O(n).

- Дополнительная структура данных для сохранения окна — O(m).

**LZ78 (Lempel-Ziv 78)**

1. Описание алгоритма.

LZ78 (Lempel-Ziv 78) - это алгоритм сжатия данных без потерь, который, как и LZ77, использует идею замены повторяющихся последовательностей данных ссылками. Однако, в отличие от LZ77, LZ78 строит временный словарь или таблицу фраз во время кодирования, вместо использования скользящего окна. Этот словарь содержит уникальные фразы, встречающиеся во входных данных.

**Основные шаги алгоритма:**

- Инициализация: Создается пустой словарь (таблица фраз). Обычно, первый элемент словаря (индекс 0) - это пустая строка (“”).

- Кодирование: Просматривается входная последовательность символ за символом. На каждом шаге определяется самая длинная фраза, начинающаяся с текущей позиции, которая уже есть в словаре. Если фраза найдена, она расширяется следующим символом из входной последовательности. Если же расширенная фраза уже есть в словаре, процесс расширения повторяется.

Если расширенной фразы нет в словаре, то:

\* Добавляется расширенная фраза в словарь с новым индексом.

\* Закодировать текущую позицию как пару (индекс фразы из словаря, следующий символ). Если фраза не была найдена (совпадение с пустой строкой), индекс будет 0.

Начинается кодирование со следующего символа после закодированной фразы.

**Пример:** так же возьмём строку «abracadabra».

| **Шаг** | **Входные данные** | **Текущий**  **символ** | **Словарь (индекс: фраза)** | **Найденная фраза (индекс)** | **Следующий символ** | **Закодиро-ванная**  **пара** | **Вывод** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | abracadabra | (инициа-  лизация) | 0: “” |  |  |  |  |
| 1 | abracadabra | a | 0: “” | 0 | a | (0, a) | a |
| 2 | bracadabra | b | 0: “”, 1: “a” | 0 | b | (0, b) | b |
| 3 | racadabra | r | 0: “”, 1: “a”, 2: “b” | 0 | r | (0, r) | r |
| 4 | acadabra | a | 0: “”, 1: “a”, 2: “b”, 3: “r” | 1 | c | (1, c) | ac |
| 5 | adabra | a | 0: “”, 1: “a”, 2: “b”, 3: “r”, 4: “ac” | 1 | d | (1, d) | ad |
| 6 | abra | a | 0: “”, 1: “a”, 2: “b”, 3: “r”, 4: “ac”, 5: “ad” | 1 | b | (1, b) | ab |
| 7 | ra | r | 0: “”, 1: “a”, 2: “b”, 3: “r”, 4: “ac”, 5: “ad”, 6: “ab” | 3 | a | (3, a) | ra |

**Пояснения:**

- Входные данные: Оставшаяся необработанная часть строки.

- Текущий символ: Символ, с которого начинается поиск соответствия в словаре.

- Словарь (индекс: фраза): Текущее состояние словаря.

- Найденная фраза (индекс): Индекс самой длинной фразы в словаре, которая соответствует началу входных данных. Если соответствия нет (совпадение с пустой строкой), то индекс равен 0.

- Следующий символ: Символ, следующий за найденной фразой во входных данных.

- Закодированная пара: Пара (индекс фразы, следующий символ), которая представляет закодированные данные.

- Вывод: Декодированная строка на данном шаге.

**Словарь после кодирования:**

0: “”

1: “a”

2: “b”

3: “r”

4: “ac”

5: “ad”

6: “ab”

7: “ra”

**Закодированные данные (список пар):**

(0, a), (0, b), (0, r), (1, c), (1, d), (1, b), (3, a)

1. Асимптотическая временная и пространственная сложность.

**Временная сложность:** O(n · m), где n - длина входной строки, а m - средняя длина фразы, которая ищется в словаре. Поиск фразы в словаре и добавление новой фразы может занять O(m) в среднем.

**Пространственная сложность:** O(p), где p - количество уникальных фраз в словаре. В худшем случае, количество фраз может быть близко к длине входной строки, если данные не содержат повторяющихся последовательностей.

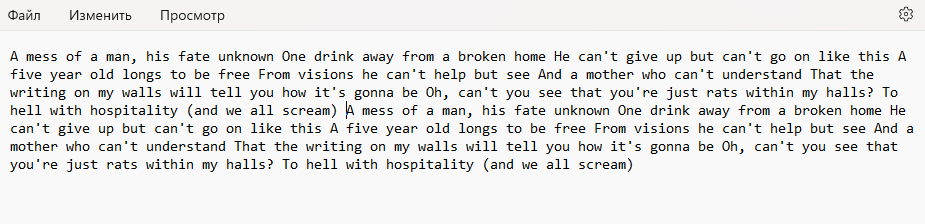
# Практическая часть

**Проверка работы алгоритмов.**

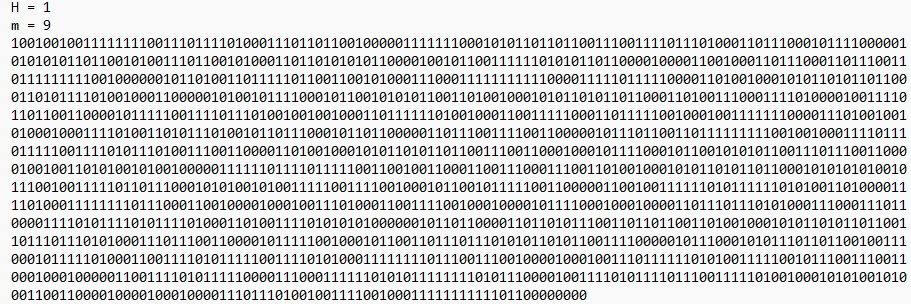
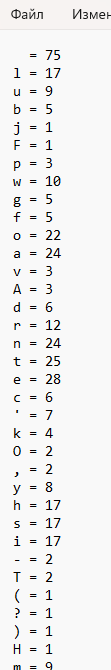
**(примеры взяты простые, чтобы убедиться, что всё правильно)**

1. HA

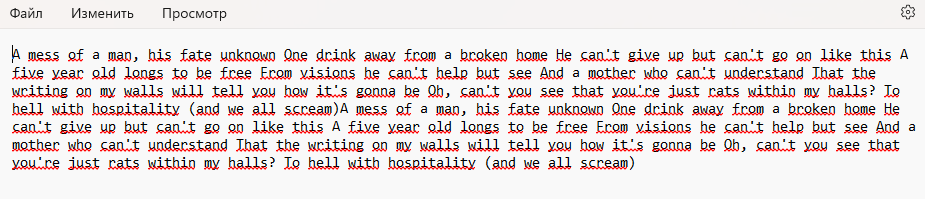
“input.txt”



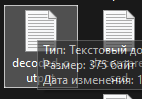
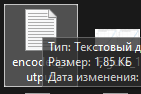
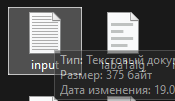
“encoded\_output.txt”



“decoded\_output.txt”



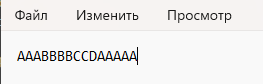
Размеры файлов:



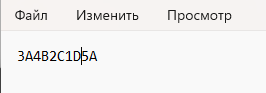
Данный тип сжатия без потери данных, однако сжимаемый файл может стать больше, так как в нём дополнительно хранится «шапка» для декомпрессии.

1. RLE

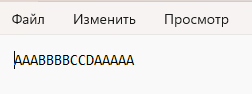
“input.txt”



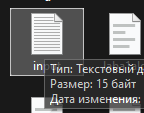
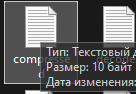
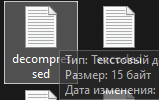
“compressed.txt”



“decompressed.txt”



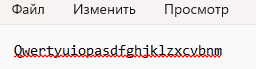
Размеры файлов:

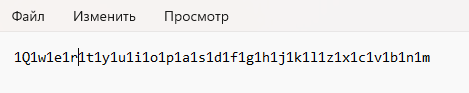
Как мы можем увидеть, файл действительно стал меньше, а также при декодировании не потерял данные и размер такой же, как и до. Однако это удачный случай кодирования, когда у нас есть повторяющиеся элементы. А если их не будет, то файл не меньше станет, а больше.

Пример такого случая:

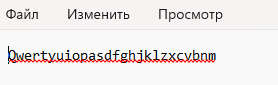
“input.txt”



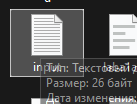
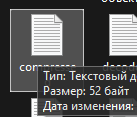
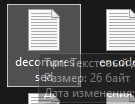
“compressed.txt”



“decompressed.txt”



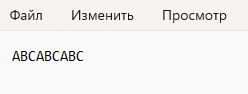
Теперь размеры файлов:

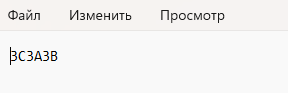
Размер стал в два раза больше, так как и символов стало в два раза больше (каждый просто удвоил свой вес).

1. BWT + RLE

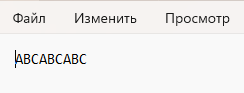
“input.txt”



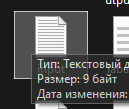
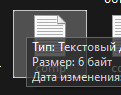
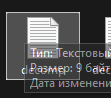
“comp.txt”



“decomp.txt”



Размеры файлов:

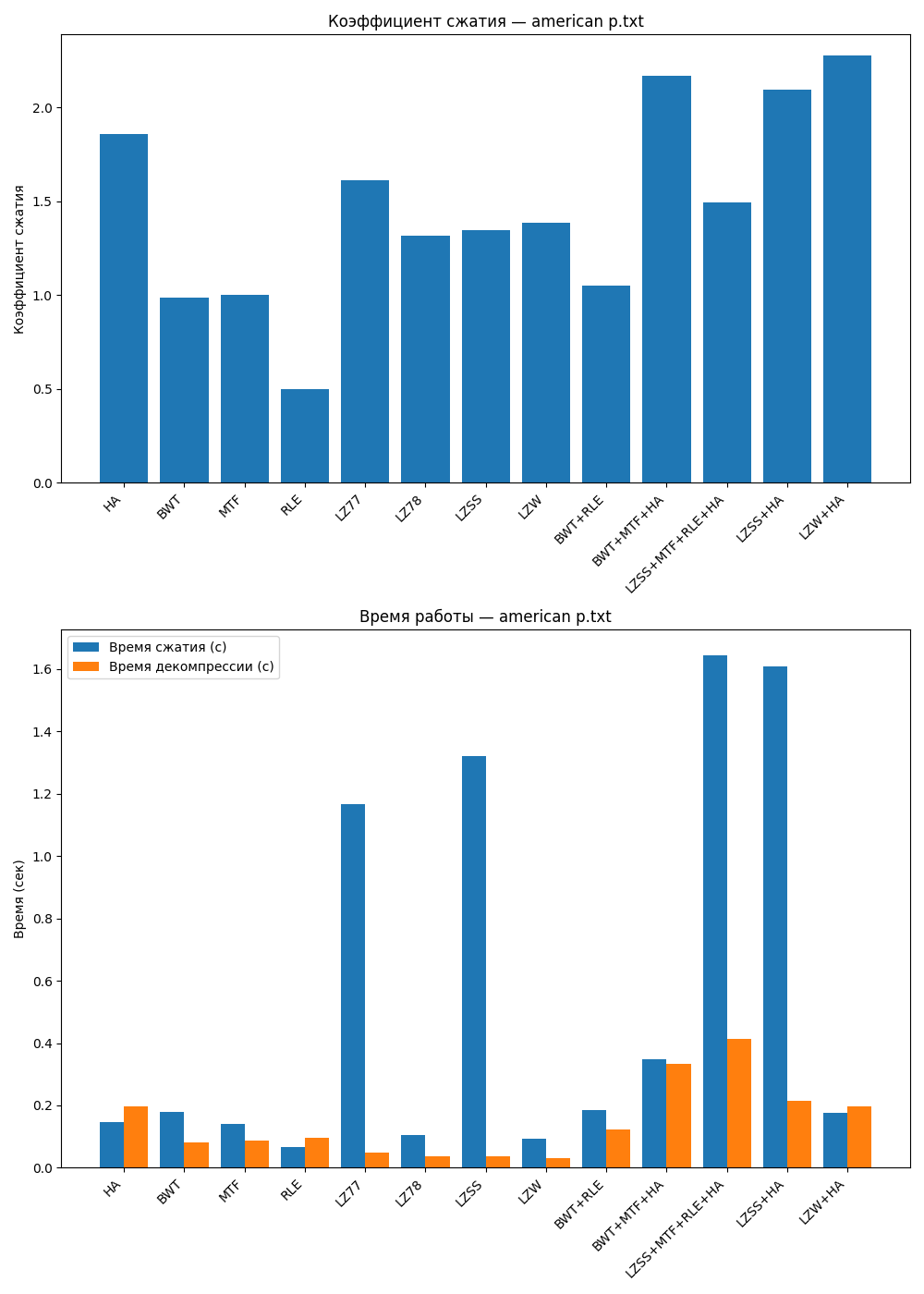
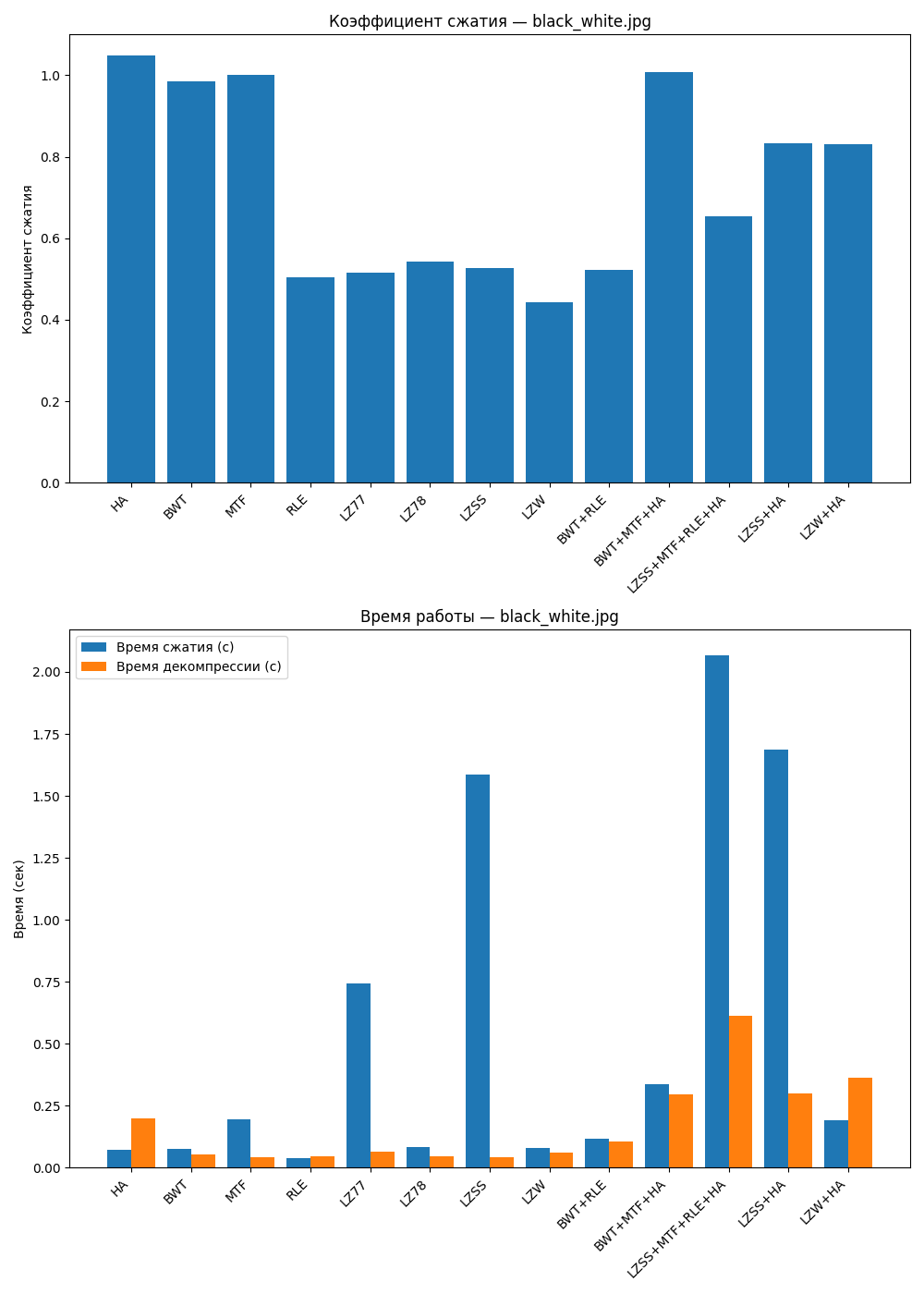
  

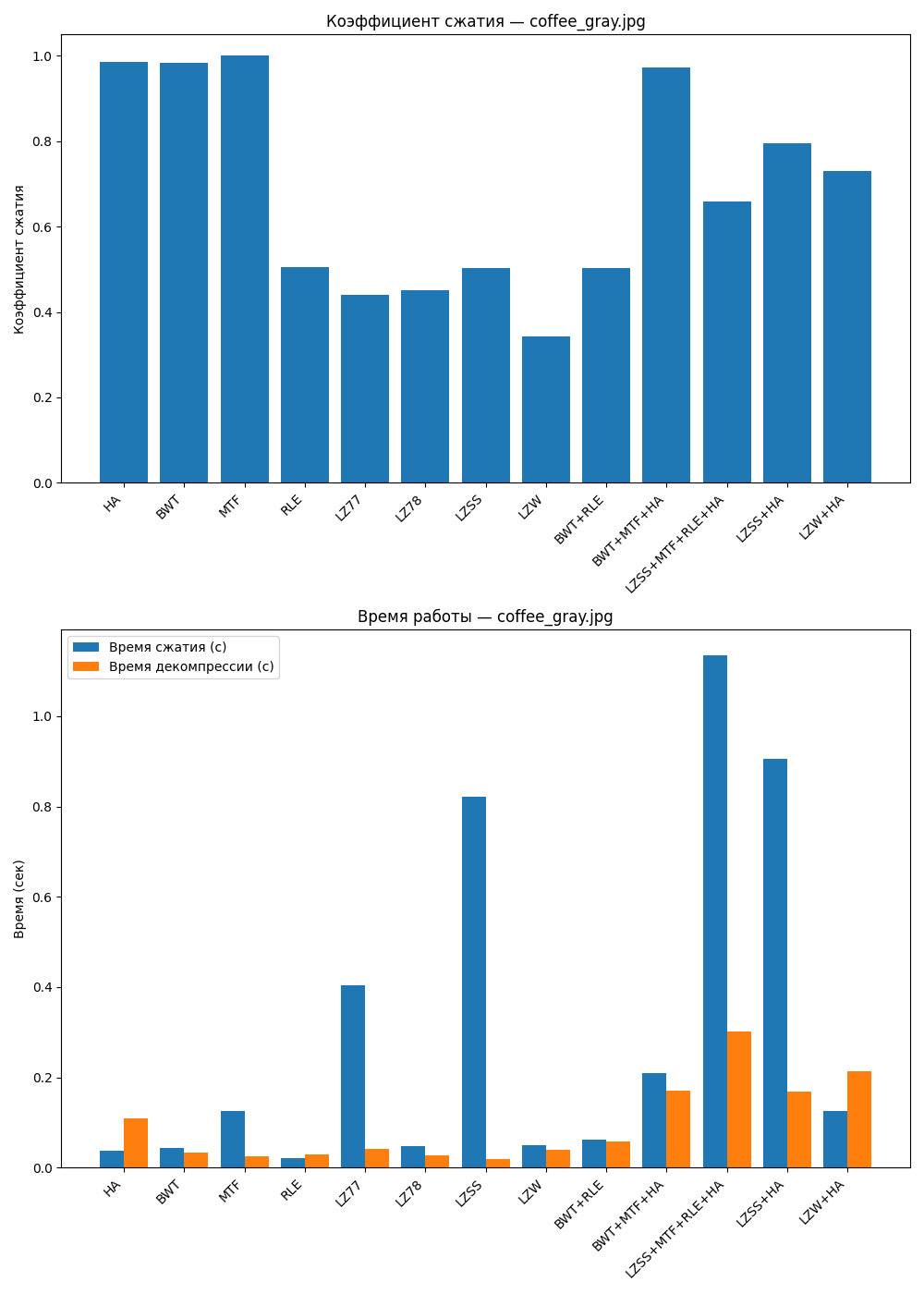
Видно, что файл сжался, а также не произошло никаких потерь данных.

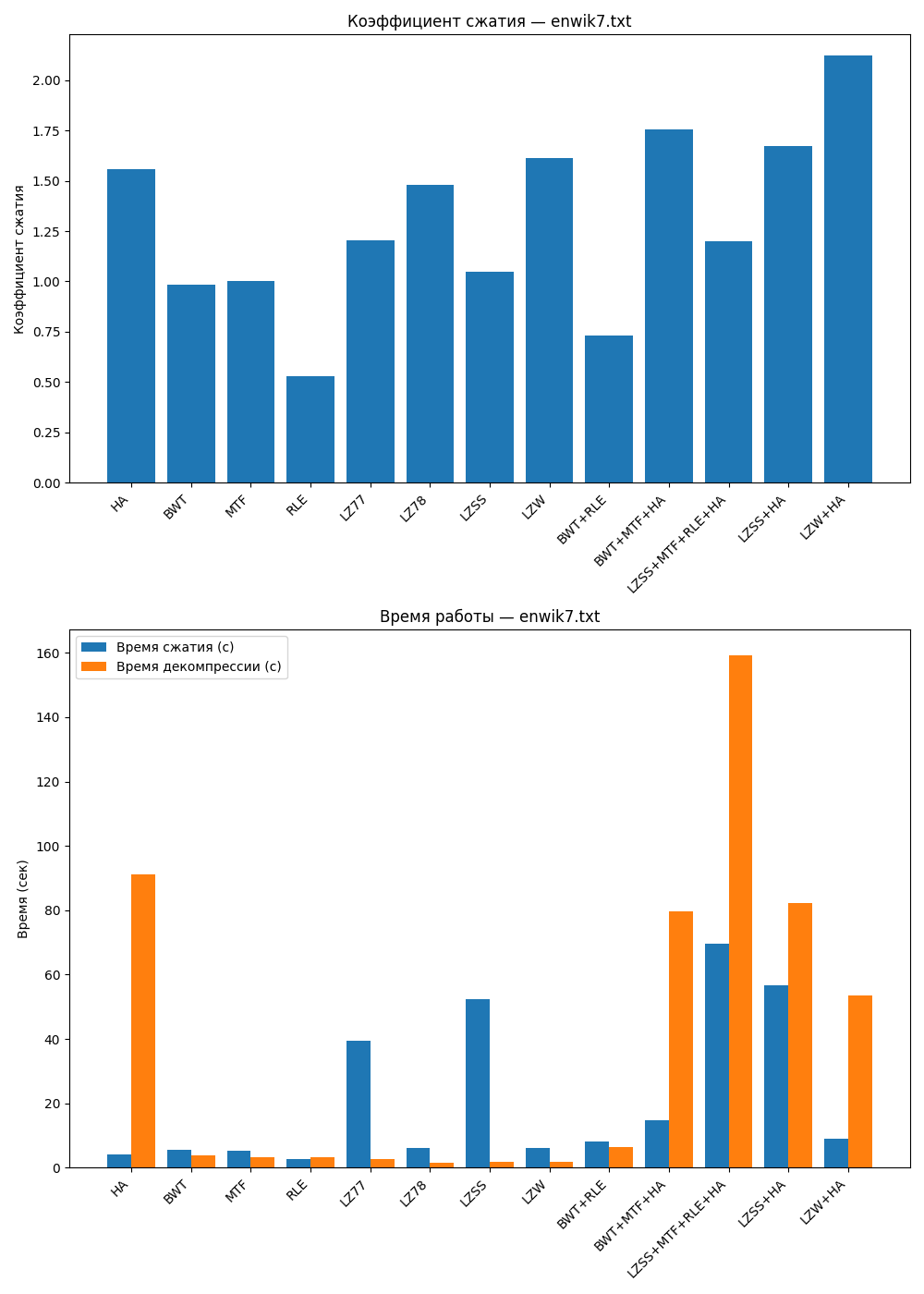
**Таблицы**

1. **Для enwik7.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Компрессор | Размер до  компрессии | Размер после компрессии | Размер после декомпрессии | Коэффициент сжатия |
| HA |  |  |  |  |
| RLE |  |  |  |  |
| BWT + RLE |  |  |  |  |
| BWT + MTF + HA |  |  |  |  |
| BWT + MTF + RLE + HA |  |  |  |  |
| LZ77 |  |  |  |  |
| LZ77 + HA |  |  |  |  |
| LZ78 |  |  |  |  |
| LZ78 + HA |  |  |  |  |







Изображение выглядит как текст, снимок экрана, диаграмма, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.