Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу «Дискретный анализ»

Студент: А. С. Бычков
Преподаватель: Н. К. Макаров
Группа: М8О-301Б

Дата:
Оценка:
Подпись:

Москва, 2025

Курсовой проект "Архиватор"

Задача: Необходимо реализовать два известных метода сжатия данных для сжатия одного файла. Методы сжатия выбираются из следующих групп:

1. Арифметическое кодирование, кодирование по Хаффману
2. $LZ77, LZW, BWT + MTF + RLE$

Формат запуска должен быть аналогичен формату запуска программы *gzip*. Должны быть поддержи- ваться следующие ключи: $-c, -d, -k, -l, -r, -t, -1, -9$. Должно поддерживаться указание символа дефиса в качестве стандартного ввода.

Теоретическая справка

1 BWT

Преобразование Барроуза — Уилера[2] (англ. $Burrows-Wheelertransform$) — алгоритм, используемый для предварительной обработки данных перед сжатием, разработанный для улучшения эффективности последующего кодирования. Преобразование Барроуза — Уилера меняет порядок символов во входной строке таким образом, что повторяющиеся подстроки образуют на выходе идущие подряд последовательности одинаковых символов.

Преобразование выполняется в три этапа:

1. Составляется таблица всех циклических сдвигов входной строки.
2. Производится лексикографическая (в алфавитном порядке) сортировка строк таблицы.
3. В качестве выходной строки выбирается последний столбец таблицы преобразования и номер строки, совпадающей с исходной.

2 MTF

Преобразование MTF[3] (англ. $move-to-front$, движение к началу) — алгоритм кодирования, используемый для предварительной обработки данных (обычно потока байтов) перед сжатием, разработанный для улучшения эффективности последующего кодирования.

Изначально каждое возможное значение байта записывается в список (алфавит), в ячейку с номером, равным значению байта, т.е. $(0, 1, 2, 3, \ldots, 255)$. В процессе обработки данных этот список изменяется. По мере поступления очередного символа на выход подается номер элемента, содержащего его значение. После чего этот символ перемещается в начало списка, смещая остальные элементы вправо.

3 RLE

Алгоритм RLE[4] (англ. $Run-LengthEncoding$) — алгоритм сжатия, заменяющий идущие подряд одинаковые символы парой (повторяющийся символ, количество повторений). Например, строчку *aaababbcbbb* он переводит в $(a, 3)(b, 1)(a, 1)(b, 2)(c, 1)(b, 3)$. Этот алгоритм эффективен для строк, содержащих много цепочек повторяющихся символов, например, результата преобразования Барроуза — Уилера.

4 Хаффман

Коды Хаффмана[1] ($Huffmancodes$) — широко распространенный и очень эффективный метод сжатия данных, который, в зависимости от характеристик этих данных, обычно позволяет сэкономить от 20% до 90% объема. Мы рассматриваем данные, представляющие собой последовательность символов. В жадном алгоритме Хаффмана используется таблица, содержащая частоты появления тех или иных символов. С помощью этой таблицы определяется оптимальное представление каждого символа в виде бинарной строки. Таблица строится таким образом, что мы получаем оптимальный префиксный код для каждого символа.

1 Описание

Из всех этих 4-х алгоритмов я использовал только 3. А именно, я не использовал алгоритм RLE. Причина этому то, что исходя из прошлой работы, где я реализовал $BWT + MTF + RLE$ на алфавите в 26 символов я получал количество пар равно 94% от исходного количества символов. Даже если выделять 1 бит на запись числа (что еще не всегда может получится, т.к. число может быть и больше 2), то никакого сжатия здесь не будет.

В моем архиваторе *azip* есть два уровня сжатия: 1 и 9. На первом уровне сжатия применяется только алгоритм Хаффмана. На последнем - цепочка $BWT- > MTF- > Huffman$.

В обоих случаях у файла есть хедер. Он выглядит следующим образом:

1. checksum crc32 - 4 байта
2. meta info
   (a) Размер сжатого файла - 8 байт
   (b) Размер несжатого файла - 8 байт
   (c) Длина исходного имени файла - 8 байт
   (d) Имя исходного файла.
3. Тип сжатия - 1 байт
4. Сериализованное дерево из алгоритма Хаффмана
5. alignment (то, сколько бит я дописал до полного байта в алгоритме Хаффмана) - 1 байт
6. BWT_index (индекс исходной строки в результате BWT) - 8 байт.

1 BWT

1.1 Кодирование

Для алгоритма BWT сначала строится строится суффиксный массив за $O(n * log_2 n)$. После, имея этот суффиксный массив, очень легко получить результат BWT.

Ассимптотически это не самый лучший вариант, т.к. можно построить суффиксный массив за $O(n)$, через суффиксное дерево. Такой вариант я реализовал, но суффиксное дерево съедает очень много памяти, из-за чего при попытке сжатия файла в 50МБ съедается около 10ГБ ОЗУ и система убивает процесс.

Кроме того, константа в дереве слишком большая и прирост производительности невелик.

1.2 Декодирование

Декодирование сильно легче. Мы сортируем строку, которая к нам пришла и получаем первый столбец всех отсортированных циклических суффиксов. Строка, которая к нам пришла, является последним столбцом этих суффиксов. После этого, можно определить, какой символ первого столбца следует за символом последнего столбца и восстановить исходную строку.

Итоговая сложность - $O(N)$

2 MTF

В алгоритме MTF нужно просто аккуратно для каждого элемента поддерживать количество элементов, меньших чем он. В итоге, при реализации наивного алгоритма сложность была $O(N * |ALPHABET|)$, где $|ALPHABET|$ - количество уникальных символов.

После, я нашел[3] пример того, как можно реализовать за $O(N * log_2|ALPHABET|)$ с использованием Декартова дерева, и реализовал так.

3 Хаффман

3.1 Кодирование

Первым делом делаю подсчет количества каждого символа и сортирую по количеству байты. Далее, с помощью дополнительного массива за $O(N)$ строю дерево, содержащее префиксные коды каждого символа.

После чего, код каждого символа переношу в $std :: array$ и за $O(N)$ составляю итоговый список бит.

В конце надо будет сереализовать дерево и запомнить то, сколько бит дополнили до байта, чтобы однозначно декодировать текст.

3.2 Декодирование

При декодировании я сначала десериализую дерево, после читаю текст по битам. Если я встретил бит 0, то иду к левому ребенку дерева, иначе - к правому. Когда я в листе, то в ответ выписываю байт и возвращаюсь к корню дерева. Итоговая сложность: $O(N)$

2 Тест производительности

Тестировать будем не только комбинации $Huffman$ и $BWT- > MTF- > Huffman$, но и их многократное повторение.

Рассматривать отдельно $BWT + MTF$ нет смысла, т.к. они только преобразовывают текст, не сжимая его.

Кроме того, бесполезно делать $Huffman- > BWT- > MTF$ по аналогичным причинам.

В столбцах «время сжатия» и «время декодировани» величина, равная времени сжатия 1КБ исходного файла.

1 Huffman

Размер файла (байт)	**Тип файла**	**Коэффициент сжатия**	**Время сжатия (мс)**	**Время декодирования (мс)**
44762946	Видео	62.9%	7380	8852
1068158	Изображение	-0.2%	332	577
102400	Случайный текст	-2.1%	55	76

Таблица 1: Замеры для различных файлов при сжатии и декодировании.

2 (Huffman)x2

Размер файла (байт)	**Тип файла**	**Коэффициент сжатия**	**Время сжатия (мс)**	**Время декодирования (мс)**
44762946	Видео	68.2%	12123	15844
1068158	Изображение	-0.4%	669	1155
102400	Случайный текст	-3.9%	88	150

Таблица 2: Замеры для различных файлов при сжатии и декодировании.

3 (Huffman)x3

Размер файла (байт)	**Тип файла**	**Коэффициент сжатия**	**Время сжатия (мс)**	**Время декодирования (мс)**
44762946	Видео	69.3%	16501	22766
1068158	Изображение	-0.6%	1023	1734
102400	Случайный текст	-5.8%	124	202

Таблица 3: Замеры для различных файлов при сжатии и декодировании.

4 BWT + MTF + Huffman

Размер файла (байт)	**Тип файла**	**Коэффициент сжатия**	**Время сжатия (мс)**	**Время декодирования (мс)**
44762946	Видео	64.7%	147228	139341
1068158	Изображение	-0.2%	2362	4133
102400	Случайный текст	-2.1%	151	1155

Таблица 4: Замеры для различных файлов при сжатии и декодировании.

5 (BWT + MTF + Huffman)x2

Размер файла (байт)	**Тип файла**	**Коэффициент сжатия**	**Время сжатия (мс)**	**Время декодирования (мс)**
44762946	Видео	70.6%	248560	191134
1068158	Изображение	-0.4%	5269	8232
102400	Случайный текст	-4.0%	317	1155

Таблица 5: Замеры для различных файлов при сжатии и декодировании.

6 (BWT + MTF + Huffman)x3

Размер файла (байт)	**Тип файла**	**Коэффициент сжатия**	**Время сжатия (мс)**	**Время декодирования (мс)**
44762946	Видео	71.5%	85065	241586
1068158	Изображение	-0.6%	9412	12311
102400	Случайный текст	-5.7%	528	1155

Таблица 6: Замеры для различных файлов при сжатии и декодировании.

3 Выводы

Разработанный мной архиватор работает довольно хорошо (по сжатию).

В случаях, когда мой архиватор может сжать файл исходный *gzip* тоже его сжимает, правда немного лучше.

В случаях же, когда мой архиватор увеличивает файл, *gzip* делает то же самое.

Кроме того, многократное применение алгоритмов не так выгодно, как первое его использование.

Список литературы

[1] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 2-е издание.* — Издательский дом «Вильямс», 2007. Перевод с английского: И. В. Красиков, Н. А. Орехова, В. Н. Романов. — 1296 с. (ISBN 5-8459-0857-4 (рус.))

[2] *Преобразование Барроуза-Уилера. - ИТМО.*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Преобразование_Барроуза-Уилера` (дата обращения: 07.01.2025).

[3] *Преобразование MTF. - ИТМО.*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Преобразование_MTF` (дата обращения: 07.01.2025).

[4] *Алгоритм RLE. - ИТМО.*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_RLE` (дата обращения: 07.01.2025).