САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра Системного Программирования

Дудин Виктор Дмитриевич

Сравнение алгоритмов сжатия коротких текстовых сообщений в задаче информационного поиска

Дипломная работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
ст. преп. Луцив Д. В.

Рецензент:
к. ф.-м. н., Кураленок И. Е.

Санкт-Петербург
2014

SAINT-PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Chair of Software Engineering

Viktor Dudin

Comparison of compression algorithms for short text messages in information retrieval problem

Graduation Thesis

Admitted for defence.
Head of the chair:
professor Andrey Terekhov

Scientific supervisor:
senior lect. Dmitry Luciv

Reviewer:
PhD Igor Kuralenok

Saint-Petersburg
2014

Оглавление

Введение

Предметная область

Интернет представляет собой мировой океан знаний. Объем знаний в нем быстро и неуклонно растет, а процесс сбора данных автоматизируется и упрощается. В связи с этим, поисковым системам приходится иметь дело с огромным количеством информации, которую необходимо хранить и обрабатывать. В таких условиях задача сжатия данных встает особенно остро.

Проблема эффективного использования дискового пространства возникла еще на заре создания электронно-вычислительных систем. С тех пор она была успешно решена множеством различных архиваторов, как общего назначения, так и предназначенных для определенного типа данных.

Отдельное внимание уделялось сжатию текстовых данных. За счет лингвистических особенностей языка, таких как наличие часто встречающихся сочетаний букв, текст удавалось сжимать более компактно, чем другие типы данных. Тем не менее, архивирование текста было успешным только на больших файлах, когда можно было собрать репрезентативную статистику использования комбинаций символов. На коротких файлах этой статистики оказывалось недостаточно, в результате чего архиваторы либо сжимали слабо, либо не сжимали вовсе.

Формулировка проблемы

Современные поисковые системы хранят на своих серверах огромное количество разнообразных текстовых данных. Одними из таких данных являются URL'ы – "адреса" интернет-страниц. URL представляет из себя небольшую строку, в среднем не более 150 символов в длину. В поисковых системах количество хранимых URL'ов измеряется миллиардами. Тем не менее, на серверах URL'ы хранятся в несжатом виде, чтобы обеспечить к ним быстрый доступ. Эффективно сжать их не получается, поскольку если собрать все URL'ы в одном файле и сжать этот файл, то для доступа к какому-либо URL'у придется разархивировать весь файл, что слишком долго.

В связи с этим, необходимо найти существующий архиватор (или придумать свой), способный компактно сжимать URL'ы по отдельности. Одновременно с этим, данный архиватор должен обеспечивать высокую скорость разархивации, чтобы предоставлять быстрый доступ к исходному URL'у.

Стоит отметить, что URL'ы – не единственные короткие текстовые данные, которые хранятся на серверах в несжатом виде. Наличие эффективного алгоритма сжатия коротких текстовых сообщений позволило бы значительно сократить количество используемых серверов и, тем самым, сэкономить затраты на их аренду и обслуживание.

1. Постановка задачи

Данная дипломная работа посвящена проблеме эффективного сжатия коротких текстовых сообщений, обладающих каким-либо общим лингвистическим признаком: одинаковым алфавитом, одинаковыми популярными комбинациями символов, прочее.

Цель данной работы – определить архиватор, который наилучшим образом подходит для сжатия коротких текстовых данных. Критериями качества архиватора являются степень сжатия данных и скорость их распаковки. Необходимо выбрать либо один из существующих архиваторов, либо разработать прототип своего собственного архиватора.

Для реализации поставленных целей были определены следующие задачи:

1. изучить существующие алгоритмы сжатия данных;
2. определить архиваторы, наилучшим образом сжимающие текстовые данные;
3. реализовать прототип своего архиватора, основанного на кодах Хаффмана с расширенным словарем;
4. сравнить существующие архиваторы с собственным архиватором, определить его применимость для сжатия коротких текстовых сообщений.

2. Обзор существующих решений

В ходе работы прежде всего были изучены существующие алгоритмы сжатия данных. Внимание уделялось алгоритмам всех типов, а не только ориентированным на сжатие текста. Без ознакомления с принципами работы этих алгоритмов невозможно понять, как функционируют современные архиваторы.

Стоит отметить, что внимание уделялось исключительно алгоритмам, предоставляющим сжатие данных без потерь. Это означает, что цепочка действий

$$\text{Архивировать} \rightarrow \text{Разархивировать}$$

должна возвращать файл в состояние, полностью идентичное исходному. Сжатие, при котором допускается потеря части исходных данных, в этой работе не рассматривается.

2.1. Обзор алгоритмов

Алгоритмов сжатия данных очень много, но их все можно условно разделить на несколько типов, каждый из которых будет рассмотрен далее. [6]

2.1.1. Run-length encoding (RLE)

Кодирование длин серий — простой алгоритм сжатия данных, который оперирует сериями данных, то есть последовательностями, в которых один и тот же символ встречается несколько раз подряд. При кодировании строка одинаковых символов, составляющих серию, заменяется строкой, которая содержит сам повторяющийся символ и количество его повторов.

Рассмотрим изображение, содержащее простой чёрный текст на сплошном белом фоне. Здесь будет много серий белых пикселей в пустых местах, и много коротких серий чёрных пикселей в тексте. В качестве примера приведена некая произвольная строка изображения в черно-белом варианте. Здесь b представляет чёрный пиксель, а w обозначает белый:

wwwwwwwwwbwwwwwwwwwwwbbbwwwwwwwwwwwwwwwwwbwwwwwww

Если применить простое кодирование длин серий к этой строке, получится следующее:

9w1b10w3b16w1b7w

Таким образом, код представляет исходные 47 символов в виде всего лишь 16.

Очевидно, что такое кодирование эффективно для данных, содержащих большое количество серий, например, для простых графических изображений, таких как иконки и графические рисунки. Однако это кодирование плохо подходит для изображений с плавным переходом тонов, таких как фотографии.

Другой пример применения – звуковые данные, которые имеют длинные последовательные серии байт (такие как низкокачественные звуковые семплы). Они могут быть эффективно сжаты с помощью RLE после того, как к ним будет применено Дельта-кодирование. [15]

2.1.2. Entropy encoding

Энтропийное кодирование — кодирование с помощью усреднения вероятностей появления элементов в закодированной последовательности. При энтропийном кодировании происходит замена символов одинаковой длины на кодовые последовательности. Эти кодовые последовательности выбираются так, чтобы длина каждой последовательности была пропорциональна отрицательному логарифму вероятности исходного символа. Таким образом, чем чаще встречается символ, тем короче его кодовая последовательность. [12][6]

Один из наиболее популярных методов энтропийного кодирование – алгоритм Хаффмана. Данный алгоритм создает коды таким образом, чтобы минимизировать

$$\sum_{symbols} prob \cdot codeLength \tag{1}$$

где $prob$ – вероятность появления данного символа в кодируемом файле, $codeLength$ – длина кодирующей последовательности. [7]

Стоит отметить, что алгоритм Хаффмана строит беспрефиксный код (prefix-free). Это означает, что ни одна кодирующая последовательность не является префиксом другой кодирующей последовательности. За счет этого исходные символы могут быть однозначно декодированы при чтении кодов из потока.

Классический алгоритм Хаффмана имеет ряд существенных недостатков. Прежде всего, для восстановления содержимого сжатого сообщения декодер должен знать таблицу частот, которой пользовался кодер. Следовательно, длина сжатого сообщения увеличивается на длину таблицы частот, что может свести на нет все усилия по сжатию сообщения. Кроме того, необходимость наличия полной частотной статистики перед началом кодирования требует двух проходов по сообщению: одного для построения таблицы частот и кодирующих последовательностей, другого для, собственно, кодирования. В-третьих, для источника с энтропией, не превышающей 1 (например, для двоичного источника), непосредственное применение кода Хаффмана бессмысленно.

Первые две проблемы позволяет решить адаптивное кодирование Хаффмана – модификация алгоритма Хаффмана, которая позволяет не передавать модель сообщения вместе с ним самим и ограничиться одним проходом по сообщению как при кодировании, так и при декодировании. Такой подход существенно сокращает время выполнение алгоритма, однако сжимает хуже, чем классический Хаффман.[14]

Другой популярный метод энтропийного кодирования – арифметическое кодирование. Данный тип кодирования, в отличие от прочих энтропийных методов, не кодирует каждый элемент исходного файла по-отдельности, а заменяет все содержимое целиком. Арифметическое кодирование преобразует содержимое в одно дробное число, лежащее в интервале $[0, 1)$.[6]

2.1.3. Dictionary coders

Сжатие с использованием словаря – это класс алгоритмов сжатия, которые действуют путем поиска совпадений между исходным текстом и набором строк (словарем), который поддерживается алгоритмом. Когда алгоритм находит совпадение, он заменяет строку на ее индекс в словаре.

Некоторые словарные алгоритмы используют статический, неизменяемый словарь. Этот словарь определен еще до начала архивации и не изменяется в процессе кодирования. Этот подход наиболее часто используется, когда сообщение или набор сообщений, подлежащих кодированию, являются фиксированными и большими; например, многие программные пакеты, которые хранят содержимое Библии в сжатом виде, строят статический словарь из согласования текста, а затем используют этот словарь для сжатия стихов.

Более распространенными являются методы, в которых словарь начинается с некоторого заданного состояния и меняется в процессе кодирования, на основе уже закодированных данных. LZ77 и LZ78 – два широкоизвестных представителя этого класса алгоритмов – работают на этом принципе. В LZ77 структура данных, называющаяся "скользящим окном", используется для хранения последних N байт обрабатываемых данных; это окно сохраняет все подстроки, которые появились в последних N байтах, как элементы словаря. При таком подходе вместо одного индекса, идентифицирующего элемент словаря, необходимы два значения: длина заменяемой подстроки и смещение "скользящего окна".

Алгоритмы сжатия семейства LZ являются одними из самых популярных алгоритмов для хранения без потерь. Алгоритм DEFLATE является вариацией LZ, оптимизированной для быстрого разархивирования и высокой степени сжатия. DEFLATE используется в PKZIP, GZIP и PNG. Другой представитель семейства, LZW (Lempel-Ziv-Welch), используется в GIF изображениях. В настоящее время выделяют две модификации семейства[6]:

1. LZX – используется в формате CAB Microsoft
2. LZMA – используется в архиваторе 7-Zip

2.1.4. Prediction by Partial Matching (PPM)

Предсказание по частичному совпадению — адаптивный статистический алгоритм сжатия данных, основанный на контекстном моделировании и предсказании. Модель PPM использует контекст — множество символов в несжатом потоке, предшествующих данному, чтобы предсказывать значение символа на основе статистических данных. Сама модель PPM лишь предсказывает значение символа, непосредственное сжатие осуществляется алгоритмами энтропийного кодирования. Оригинальный алгоритм, опубликованный в 1984, для сжатия использовал арифметическое кодирование.

Длина контекста, который используется при предсказании, обычно сильно ограничена. Эта длина обозначается n и определяет порядок модели PPM, что обозначается как PPM(n). Неограниченные модели также существуют и обозначаются просто PPM*. Если предсказание символа по контексту из n символов не может быть произведено, то происходит попытка предсказать его с помощью $n-1$ символов. Рекурсивный переход к моделям с меньшим порядком происходит, пока предсказание не произойдёт в одной из моделей, либо когда контекст станет нулевой длины ($n=0$). Модели степени 0 и -1 следует описать особо. Модель нулевого порядка эквивалента случаю контекстно-свободного моделирования, когда вероятность символа определяется исключительно из частоты его появления в сжимаемом потоке данных. Подобная модель обычно применяется вместе с кодированием по Хаффману. Модель порядка -1 представляют собой статическую модель, присваивающую вероятности символа определенное фиксированное значение; обычно все символы, которые могут встретиться в сжимаемом потоке данных, при этом считаются равновероятными. Для получения хорошей оценки вероятности символа необходимо учитывать контексты разных длин. PPM представляет собой вариант стратегии перемешивания, когда оценки вероятностей, сделанные на основании контекстов разных длин, объединяются в одну общую вероятность. Полученная оценка кодируется любым энтропийным кодером, обычно это некая разновидность арифметического кодера. На этапе энтропийного кодирования и происходит собственно сжатие.

Большое значение для алгоритма PPM имеет проблема обработки новых символов, ещё не встречавшихся во входном потоке. Эту проблему называют *проблемой нулевой частоты*. Некоторые варианты реализаций PPM полагают счётчик нового символа равным фиксированной величине, например, единице. Другие реализации, как например, PPMd, увеличивают псевдосчётчик нового символа каждый раз, когда, действительно, в потоке появляется новый символ. (Другими словами, PPMd оценивает вероятность появления нового символа как отношение числа уникальных символов к общему числу используемых символов).

Опубликованные исследования алгоритмов семейства PPM появились в середине 1980-х годов. Программные реализации не были популярны до 1990-х годов, потому как модели PPM требуют значительное количество оперативной памяти. Современные реализации PPM являются одними из лучших среди алгоритмов сжатия без потерь для текстов на естественном языке. [6] [8] [1]

Варианты алгоритма PPM на данный момент широко используются, главным образом для компрессии избыточной информации и текстовых данных. Следующие популярные архиваторы используют PPM[10]:

- RAR (версии 3 и выше) — реализация варианта PPMd, PPMII
- 7-Zip — реализация варианта PPMd
- WinZip (версии 10 и выше) — реализация варианта PPMd

2.1.5. Context Mixing (CM)

Смешивание контекстов представляет собой отдельный класс алгоритмов сжатия данных. В таких алгоритмах предсказания следующего символа от двух или более статистических моделей "смешиваются", чтобы получить новое предсказание, которое часто оказывается более точным, чем любое из предсказаний по отдельности. Одним из способов смешивания является усреднение вероятностей, назначенных каждой моделью. Другим способом является *Random Forest* – из всех предсказаний выбирается значение, которое встретилось наибольшее число раз. Объединение моделей является активной сферой исследований в области машинного обучения.[3]

Алгоритм смешивания контекстов используется в ряде архиваторов, демонстрирующих очень высокий коэффициент сжатия текстовых данных. Среди них[9]:

- Архиваторы серии PAQ: PAQ, LPAQ, ZPAQ
- WinRK 3.0.3 (Malcolm Taylor) в режиме максимального сжатия PWCM
- NanoZip (Sami Runsas) в режиме максимального сжатия

2.1.6. Burrows-Wheeler Transform (BWT)

Преобразование Барроуза-Уилера, также известное как блочно-сортирующее сжатие, – это алгоритм, используемый в техниках сжатия данных для преобразования исходных данных. Сам по себе не является алгоритмом сжатия, однако используется в комбинации с другими алгоритмами.

BWT меняет порядок символов во входной строке таким образом, что повторяющиеся подстроки образуют на выходе идущие подряд последовательности одинаковых

символов. Таким образом, сочетание BWT и RLE выполняет задачу сжатия исключением повторяющихся подстрок, то есть задачу, аналогичную алгоритмам LZ.

Кроме того, почти точно повторяющиеся (с незначительными отличиями) подстроки входного текста дают на выходе последовательности одинаковых символов, редко перемежающиеся другими символами. Если после этого выполнить шаг по замене каждого символа расстоянием до его предыдущей встречи (алгоритм move-to-front, MTF), то полученный набор чисел будет иметь крайне удачное статистическое распределение для применения энтропийного сжатия типа Хаффмана или же арифметического. На практике алгоритм сжатия вида

$$\text{BWT} \rightarrow \text{MTF/RLE} \rightarrow \text{Huffman}$$

применённый в архиваторе bzip2, немного превосходит лучшие реализации LZH по качеству сжатия при аналогичной скорости.[11]

2.2. Обзор архиваторов

Несмотря на то, что существует множество различных алгоритмов сжатия, количество различных архиваторов заметно больше. Отчасти потому, что многие алгоритмы сжатия имеют настраиваемые параметры, и каждая комбинация параметров может порождать уникальный архиватор. Отчасти и потому, что большинство современных архиваторов в процессе работы используют несколько алгоритмов, запущенных в определенном порядке. Замена какого-либо из этих алгоритмов, или же смена их порядка применения, также может породить новый архиватор.

Поскольку данная дипломная работа посвящена сжатию коротких текстовых сообщений, дальнейший обзор архиваторов будет ориентирован исключительно на архиваторы, предназначенные для максимального сжатия текстовых данных.

Прежде всего, необходимо выделить из всего множества архиваторов только те, которые компактно сжимают текст. Для решения этой задачи использовались существующие онлайн-проекты, на которых представлены результаты тестирования большого числа архиваторов на различных наборах данных, включая текстовые данные. Среди таких сайтов [2] [5] [4]:

- *Compression Ratings* [1]
- *Maximum Compression* [2]
- *Large Text Compression Benchmark* [3]

[1] http://compressionratings.com/txt2.html
[2] http://www.maximumcompression.com/data/text.php
[3] http://mattmahoney.net/dc/text.html

Изучив содержание этих сайтов и архиваторы, которые тестировались на этих сайтах, становится ясно, что проект *Large Text Compression Benchmark* является самым актуальным из всех вышеперечисленных. Доказательством этому является наличие истории изменений сайта и записи от 12 мая (менее недели назад). Более того, все алгоритмы, представленные на других сайтах, присутствуют и на этом сайте. Учитывая все вышесказанное, дальнейший обзор текстовых архиваторов можно вести, опираясь лишь на сайт *Large Text Compression Benchmark*, поскольку информация, опубликованная на нем, является актуальной и полной.

Изучая результаты тестирования на этом сайте, можно заметить, что некоторые архиваторы присутствуют в таблице по несколько раз. Они отличаются друг от друга либо версией программы, либо параметрами запуска. Тем не менее, значительных отличий у таких архиваторов, скорее всего, нет. Чтобы не исследовать одинаковые алгоритмы по несколько раз, было решено из каждого семейства алгоритмов изучать только топовую модификацию.

Ниже представлены лучшие представители от 10 лучших семейств архиваторов.

Семейство	**Представитель**	**enwik8**	**enwik9**	**Алго**
durilca	durilca'kingsize	16.209%	12.738%	PPM
cmix	cmix v1	16.076%	12.865%	CM
paq	paq8hp12any	16.230%	13.205%	CM
nanozip	nanozip 0.09a	18.723%	15.004%	CM
xwrt	xwrt 3.2	18.679%	15.117%	CM
WinRK	WinRK 3.03	18.612%	15.629%	CM
ppm	ppmonstr J	19.055%	15.701%	PPM
slim	slim 23d	19.077%	15.977%	PPM
bwmonstr	bwmonstr 0.02	20.307%	16.047%	BWT
zcm	zcm 0.60d	19.786%	16.273%	CM

Здесь столбцы *enwik8* и *enwik9* показывают, какой процент от исходного размера файла составляют сжатые enwik8 и enwik9, соответственно. enwik8 (enwik9) – первые 10^8 (10^9) байт от файла enwiki-20060303-pages-articles.xml – "слепка" англоязычной Википедии, сделанного в 2006 году.

Столбец *Алго* обозначает класс алгоритма сжатия, использованного в архиваторе. В случае, если архиватор использует комбинацию алгоритмов, в столбце указывается алгоритм, использованный на последней стадии перед непосредственно кодированием.

Такие показатели, как скорость сжатия/расжатия, указанные на данном сайте, нельзя считать достаточно объективными, поскольку каждый из перечисленных ар-

хиваторов тестировался на отдельной ЭВМ. Ключевые системные характеристики, такие как CPU, RAM и число ядер процессора, на всех ЭВМ были различными.

Как видно из приведенной выше таблицы, наилучший уровень сжатия текстовых данных достигают архиваторы, основанные либо на *смешивании контекстов*, либо на алгоритмах класса *PPM*.

3. Алгоритм Хаффмана с расширенным словарем

Наряду с существующими алгоритмами и архиваторами, в данной работе описан еще один подход к архивированию текстовых данных. Этот подход, названный *алгоритмом Хаффмана с расширенным словарем*, обеспечивает высокую степень сжатия текстовых сообщений (как коротких, так и длинных), обладающих каким-либо общим лингвистическим признаком: одинаковым алфавитом, одинаковыми популярными комбинациями символов, прочее.

3.1. Предпосылки алгоритма

Современные поисковые системы хранят внутри себя множество различной служебной информации, необходимой для предоставления быстрого и качественного поиска. В связи с огромным количеством обрабатываемых данных, структуры данных, необходимые для функционирования системы, также велики, а их производительность напрямую зависит от их размера. Чем больше размер структуры, тем больше времени требуется для обработки запроса. В связи с этим эффективное сжатие структуры данных может не только сэкономить дисковое пространство, но и увеличить производительность как самой структуры, так и поисковой системы в целом.

Примером задачи, решаемой в поисковых системах, является хранение данных по некоторому ключу. В частности, необходимо по URL'ам хранить мета-информацию, не зависящую от поискового запроса. Чтобы предоставлять быстрый доступ к такой информации, необходимо иметь эффективную структуру данных. Одной из возможных структур, подходящих для данной задачи, является *трай (trie)*, также известный как *префиксное дерево*. Пример трая проиллюстрирован на Рис. 1. На нем прямоугольниками изображены вершины трая, треугольниками – листья. Символы, указанные на ребрах трая, обозначают условия перехода из одной вершины в другую. Построив трай по всему множеству URL'ов, можно разместить в его листья метаданные, соответствующие URL'у, заданному путем от корня трая до данного листа.

Однако использование классического трая имеет ряд существенных недостатков. Средняя длина URL'а составляет 30-40 символов. Если в трае переход по каждому ребру будет соответствовать только 1 символу, то трай получится очень высоким, что негативно скажется на его производительности. Помимо этого, трай очень несбалансирован: в случае с URL'ами поддерево, начинающееся с набора символов *http*, будет содержать значительно больше элементов, чем поддерево, начинающееся с символа, например, *z*.

Поскольку URL'ов очень много, построенный на них трай не вмещается в оперативную память, в связи с чем его приходится читать с диска. Каждый проход по ребру трая – это новое рандомное чтение с диска (*seek*), которое занимает много времени. Чем глубже по траю нужно пройти, тем больше seek'ов потребуется сделать.

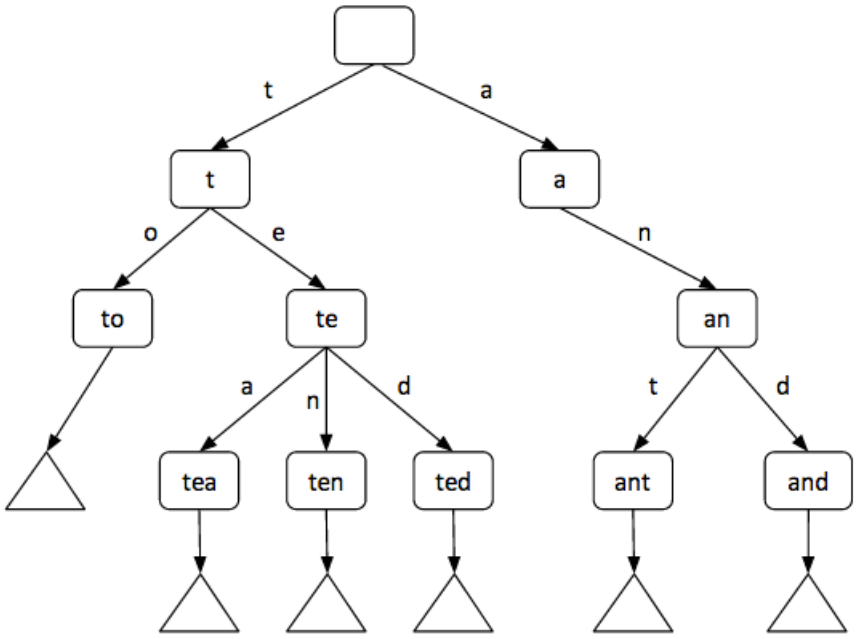

Рис. 1: Трай, построенный по словам *to*, *tea*, *ten*, *ted*, *ant*, *and*

В связи с этим возникает желание сделать так, чтобы для чтения метаданных по URL'у требовалось как можно меньше seek'ов. Для этого необходимо сделать трай более сбалансированным по высоте. Добиться этого можно при помощи объединения пар наиболее зависимых символов в новые элементы словаря (алфавита). Добавляя такие пары в словарь в качестве новых символов, получится расширенный словарь V. Используя V, каждый URL можно будет представить меньшим количеством символов, а значит трай, построенный по этим URL'ам, будет иметь меньшую высоту. Меньшая высота трая гарантирует меньшее количество seek'ов, а значит уменьшит время обработки запросов. Помимо этого, сам трай будет заниматься меньше дискового пространства, а значит сэкономит ресурсы поисковой системы.

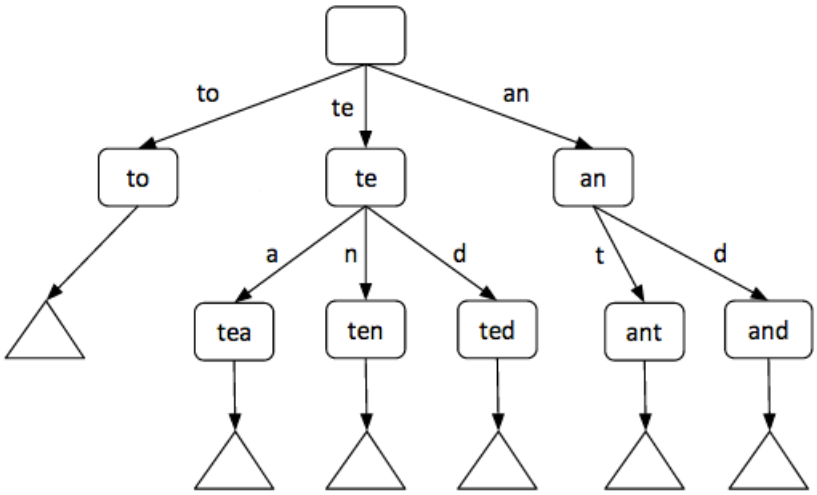

Рис. 2: Трай, использующий дополнительный словарь { *to*, *te*, *an* }

3.2. Описание подхода

Решая задачу с оптимизацией трая, было замечено, что описанный расширенный словарь может быть использован и в других целях. Одним из его дополнительных применений может являться сжатие текстовых данных и, в частности, URL'ов. Сгенерировав расширенный словарь V и переписав каждый URL в элементах V, получим множество тех же самых URL'ов но в более короткой записи. Применив затем какой-либо алгоритм энтропийного кодирования (например, алгоритм Хаффмана), получится еще больше сжать весь массив данных.

Примечательной особенностью данного алгоритма является то, что расширенный словарь V не обязательно составлять отдельно для каждого URL'а. В данном подходе делается предположение, что некоторые популярные комбинации символов (*http*, *www*, *.ru*, *.com*, *wiki*, др.) будут характерны для большого процента от всего множества URL'ов. А значит, построив словарь V один раз на некотором репрезентативном подмножестве URL'ов, его можно будет применять ко всем URL'ам без больших потерь в качестве сжатия.

Более того, обучая словарь V на подмножестве URL'ов, можно не только выбрать популярные комбинации символов, но и собрать статистику их использования при кодировании данного обучающего множества. На основе собранной статистики можно построить таблицу частот, по которой, в свою очередь, определить коды Хаффмана для каждого элемента словаря V. Таким образом, кодируя URL'ы элементами из V, нет необходимости повторно собирать статистику использования этих элементов. Это означает, что для кодирования URL'а необходим всего 1 проход по строке, а не 2, как это делается в обычных энтропийных алгоритмах. Такой прием обеспечивает высокую скорость сжатия данных.

Помимо прочего, раз словарь V используется один для всего множества URL'ов, его не обязательно сохранять вместе с каждым сжатым URL'ом, а достаточно сохранить 1 раз в системе, и в дальнейшем использовать его только для архивирования и разархивирования всех URL'ов.

Главной проблемой большинства алгоритмов сжатия является необходимость сохранения мета-информации, требующейся для разархивирования, вместе с архивируемым файлом. Из-за этого все архиваторы не могут эффективно сжимать короткие текстовые сообщения: размер мета-информации + сжатых данных оказывается равен (а иногда и превосходит) размер исходных данных. Данный алгоритм лишен этой проблемы, поскольку он не сохраняет словарь вместе с каждым файлом. За счет этого он способен эффективно сжимать как длинные, так и короткие текстовые сообщения.

Стоит отметить, что описанный алгоритм подход для сжатия не только URL'ов, но и других массивов текстовых данных, обладающих общими лингвистическими признаками: одинаковый алфавит, одинаковые популярные комбинации символов, др.

3.3. Реализация

3.3.1. Выбор элементов для расширенного словаря

Первый вопрос, который возникает при реализации описанного выше метода – каким образом определять, какие комбинации символов должны войти в расширенный словарь V в качестве отдельного элемента? Каким требованиям должна удовлетворять последовательность символов, чтобы ее добавили в V?

Интуитивные размышления наводят на мысль, что для включения в словарь комбинация символов должна быть:

1. популярной, т.е. ожидается, что данная комбинация встретится большое количество раз, а не останется невостребованным элементом словаря

2. максимально полезной, то есть добавление именно этой комбинации (а не какой-либо другой) принесет наибольшее увеличение коэффициента сжатия. Наглядный пример: пусть комбинации *http://* и *.ru* встречаются одинаковое число раз. Очевидное, что полезнее добавить *http://*, нежели *.ru*, поскольку эта комбинация сжимает большее число символов в 1 символ (при равном количестве появлений комбинаций).

Поскольку исследовать сразу все встречающиеся подстроки слишком сложно (в силу их огромного количества), было принято решение исследовать только пары символов, выбирая из них самую полезную для добавления. После того, как какая-либо пара добавляется в словарь V отдельным символом, начинают дополнительно рассматриваться всевозможные пары, составленные из этого символа и всех других элементов V. Таким образом, если какая-то комбинация символов заслуживает того, чтобы присутствовать в V, она там рано или поздно окажется. В качестве примера можно привести получение комбинации *http*:

$$h + t \rightarrow ht$$

$$ht + t \rightarrow htt$$

$$htt + p \rightarrow http$$

Возможно, пары будут добавляться в другом порядке (сначала *ht*, затем *tp*, затем *http*), но в итоге полезная подстрока все равно окажется в V.

По-прежнему остается открытым вопрос о способе выбора наиболее полезной пары символов. В ходе данной работы было опробовано 2 подхода. Один из них считает наиболее полезной ту пару, у которой наибольший показатель k:

$$k = count \cdot length$$

Здесь *count* означает количество вхождений пары в текст (текст – в символах текущего расширенного словаря), а *length* – длину пары (в символах исходного словаря).

Другой подход заключается в использовании Пуассоновской вероятности. Наилучшей признается пара, у которой количество вхождений (*count*) превосходит Пуассоновское мат. ожидание, а Пуассоновская вероятность является наименьшей. Такой подход обеспечивает выбор пары наиболее зависимых символов, тем самым снижая зависимости между символами текста. Согласно [7], коды Хаффмана являются оптимальными в случае, когда элементы словаря независимы между собой. Таким образом, снижая зависимости между символами, обеспечивается наиболее эффективный словарь для дальнейшего применения алгоритма Хаффмана.

Пуассоновская вероятность вычисляется по формуле

$$\begin{aligned} \lambda &= P(a) \cdot P(b) \cdot n \\ prob &= \frac{\lambda^k}{k!} \cdot e^{-\lambda} \end{aligned}$$

где a и b – первый и второй элементы пары соответственно, $P(a)$ и $P(b)$ – вероятности появления символов a и b в тексте соответственно, n – суммарное количество пар в тексте (текст – в текущем словаре V), k – количество пар вида ab.

На ранних стадиях экспериментов было выявлено, что подход с Пуассоновской вероятностью генерирует более качественный словарь и обеспечивает более высокую степень сжатия. В связи с этим, в дальнейшем использовался только Пуассоновский подход.

3.3.2. Генерация словаря

Определив, как именно следует выбирать комбинации символов для добавления в V, встает вопрос о том, каким образом генерировать весь словарь V. В данной работе использовалось 2 подхода, которые описаны ниже. В обоих случаях в качестве начального состояния V использовался набор всех символов, используемых во всем массиве текстовых данных (а не только в подмножестве, на котором обучается словарь). Иными словами, начальное состояние V – это алфавит, используемый в сжимаемом массиве данных.

Итеративное наполнение словаря

Первый подход можно условно назвать *итеративным наполнением словаря*. На каждой итерации в словарь V добавляется новая пара символов, которая признается самой полезной для добавления. Данный подход подразумевает обучение V на некоторой фиксированной выборке из массива сжимаемых данных. Если выборка является достаточно репрезентативной, то полученный по ней расширенный словарь окажется близким к оптимальному для сжатия всего массива данных.

Ход работы алгоритма можно описать следующим образом:

1. считывание данных, инициализация начального состояния V;
2. создание *pairs* – множества всех пар (из элементов V), присутствующих в тексте; подсчет количества вхождений каждой пары;
3. вычисление Пуассоновской вероятности для каждого элемента *pairs*;
4. на каждой итерации:
   (a) выбор *newPair* – пары с наименьшей Пуассоновской вероятностью;
   (b) добавление *newPair* в словарь V в качестве нового символа *newSymbol*;
   (c) обновление счетчиков у тех пар из *pairs*, которые пересекались одним из своих символов с *newPair*. Если у какой-либо пары счетчик опустился до 0, она удаляется из *pairs*;
   (d) добавление в *pairs* всевозможных пар, составленных из *newSymbol* и других элементов V, и при этом присутствующих в тексте;
   (e) проверка элементов V, из которых составлена *newPair*. Если какой-либо из этих элементов больше не присутствует ни в одной паре из *pairs*, он признается ненужным и удаляется из V. Исключение составляют символы, входившие в начальное состояние V: их удаление некорректно, поскольку они могут больше не использоваться в текущем обучаемом множестве, но быть востребованы во всем массиве данных. Таким образом, их удаление из V приведет к невозможности архивирования исходного набора данных;
   (f) пересчет Пуассоновских вероятностей для всех пар из *pairs*.
5. полученный словарь V является готовым целевым словарем.

Одна из сложностей, возникающих в ходе работы алгоритма - это размер *pairs* и необходимость пересчитывать Пуассоновские вероятности на каждой итерации алгоритма. При сжатии URL'ов размер *pairs* быстро увеличивался до нескольких сотен тысяч, в связи с чем скорость работы алгоритма резко падала. Чтобы оптимизировать данный подход, было принято решение оставлять после каждой итерации только ТОП лучших пар из *pairs* (пар с наименьшей Пуассоновской вероятностью). Такой прием незначительно портил качество получаемого словаря, однако заметно ускорял алгоритм и уменьшал количество требуемой оперативной памяти.

Метод последовательных приближений

Второй подход можно условно назвать *методом последовательных приближений*: на каждой итерации алгоритма расширенный словарь V пополняется новыми, более полезными парами, а самые бесполезные пары исключаются. Данный подход подразумевает обучение с помощью последовательности выборок различного размера. Еще одной особенностью является фиксированный размер V - в конце каждой итерации количество элементов в V не будет превосходить заранее определенного числа *SIZE* (не считая элементы исходного алфавита);

Ход работы алгоритма можно описать следующей последовательностью действий:

1. считывание данных, инициализация начального состояния V;
2. на каждой итерации:
   (a) получение рандомной выборки *dataSample1*;
   (b) кодирование *dataSample1* элементами из V, получение информации об использованных символах и парах символов (*pairs*);
   (c) подсчет Пуассоновской вероятности для элементов *pairs*, добавление наиболее полезных пар в V в качестве новых символов;
   (d) получение рандомной выборки *dataSample2*;
   (e) кодирование *dataSample2* элементами из V, получение информации об использованных символах *symbols*;
   (f) сортировка элементов *symbols* по убыванию показателя $count \cdot length$, где $count$ – количество использований элемента при кодировании *dataSample2*, $length$ – длина элемента в символах исходного алфавита;
   (g) в V остаются первые *SIZE* элементов из *symbols* с наилучшим показателем $count \cdot length$. Помимо этого, в V остаются все элементы начального алфавита.
3. полученный словарь V является готовым целевым словарем с количеством элементов не более *SIZE*.

Кодирование набора данных элементами из V выполняется "жадным" способом: в первую очередь к началу потока символов применяются самые длинные элементы V; затем, если ни один из длинных элементов не подошел, применяются более короткие элементы; если и они не подошли, применяются еще более короткие; и так далее до тех пор, пока не будет найден элемент из V, совпадающий с началом потока. Хотя бы одно совпадение гарантированно будет найдено, поскольку V изначально содержал все элементы алфавита, использованные в исходных данных.

Для наглядности рассмотрим пример "жадного" кодирования строки *"abcdtabc"* элементами {"*abcd*","*abc*","*ab*","*dt*","*a*","*b*","*c*","*d*","*t*"}. Данная строка будет закодирована следующим образом:

$$abcd \rightarrow t \rightarrow abc$$

В первую очередь будет применен символ "*abcd*", а не "*abc*" или "*ab*", поскольку этот символ самый длинный из всех подходящих. Потом будет использован символ "*t*", поскольку он единственный подходит к началу оставшейся строки "*tabc*". Затем будет использован символ "*abc*" (по той же причине, что и "*abcd*" в начале кодирования).

Отдельный вопрос вызывает пункт *2.c*: сколько пар из *pairs* стоит добавлять в V? На этот вопрос нет однозначного ответа. Интуитивно кажется, что количество добавляемых пар будет влиять только на скорость обучения, но не отразится на качестве полученного словаря. В качестве одного из решений можно добавлять в V только те пары, чье количество вхождений превосходит Пуассоновское мат. ожидание.

Еще один открытый вопрос данного подхода касается размера выборок *dataSample1* и *dataSample2*. Имея словарь V и вероятности появления его элементов в прошлой выборке, можно рассчитать размер новой выборки, необходимый для того, чтобы каждый элемент V встретился в новой выборке хотя бы 1 раз. Эти вычисления можно сделать, опираясь на теорему Пуассона [13]. Следуя этим вычисления, размер каждой новой выборки должен быть не меньше подсчитанного минимального размера.

Размер словаря

Главный вопрос, который встает при практическом использовании описанных подходов – сколько итераций необходимо сделать, чтобы получить оптимальный словарь V? (В случае второго подхода: какой размер словаря *SIZE* оптимален?) В какой момент необходимо остановиться и перестать расширять словарь? Стоит помнить, что каждому элементу из V будет сопоставлен некий код Хаффмана. Чем больше размер V, тем больше будет длина кодов Хаффмана, как у редких элементов, так и у популярных. Таким образом, слишком большой размер словаря приведет к ухудшению качества сжатия исходных данных. С другой стороны, слишком маленький размер словаря не сможет вместить все множество популярных комбинаций символов, а значит не обеспечит максимального качества сжатия.

В данной работе этот вопрос остается без ответа. При тестировании размеры целевого словаря выбирались экспериментально, без какого-либо теоретического обоснования. Изучение данного вопроса заслуживает отдельного исследования.

3.3.3. Кодирование

Последним этапом описываемого алгоритма сжатия является создание таблицы частот элементов из V и непосредственно кодирование. В случае с первым подходом к генерации расширенного словаря (*итеративное наполнение словаря*) частоты вхождения элементов в обучающее множество не нужно вычислять отдельно – они вычисляются и поддерживаются в ходе работы алгоритма. В случае со вторым подходом (*метод последовательных приближений*) предлагается взять еще одну выборку и на основе нее вычислить частоты появления элементов V в кодируемом массиве данных.

Получив таблицы частот элементов расширенного словаря V, для каждого элемента вычисляется код Хаффмана, соответствующий его частоте. Полученный словарь V вместе с кодами Хаффмана будет в дальнейшем использоваться для архивирования и разархивирования как всего массива исходных данных (набор URL'ов), так и отдельных его элементов (1 URL).

Теоретически, используя таблицу частот, можно применить любой другой алгоритм энтропийного кодирования для сжатия исходных данных. В частности, арифметическое кодирование могло бы (в теории) достигать лучшей степени сжатия данных. Однако в данной работе проводились тесты только с использованием алгоритма Хаффмана.

Для организации эффективного архивирования строится трай по множеству элементов V. В листьях трая располагаются коды Хаффмана, соответствующие элементу, заданному путем от корня трая до данного листа. Реализуя "жадный" подход при архивировании (аналогичный тому, который используется в генерации словаря *методом последовательных приближений*), данный алгоритм осуществляет сжатие данных всего за 1 проход по данным, в то время как классическим энтропийным алгоритмам для сжатия требуется 2 прохода.

При разархивировании описанный алгоритм действует так же, как и стандартный алгоритм Хаффмана, с той лишь разницей, что таблица кодов Хаффмана и соответствующих им комбинаций символов хранится отдельно от разжимаемого файла.

Стоит отметить, что второй подход к генерации словаря V выглядит более "правильным", поскольку он выбирает комбинации символов, которые лучшим образом подходят для "жадного" кодирования (которое как раз используется при сжатии данных). В свою очередь первый подход выбирает лучшие комбинации независимо от того, где именно в тексте они встречаются: в начале, в середине или в конце. В связи с этим интуитивно ожидается, что второй подход к генерации V окажется более результативным.

4. Сравнение алгоритмов

Кульминацией данной дипломной работы является сравнение различных архиваторов, призванное определить, какой архиватор является наилучшим для сжатия коротких текстовых сообщений.

4.1. Критерии сравнения

Прежде всего, необходимо определить критерии качества, по которым будет проводиться сравнение. В поисковых системах ключевую роль играют два показателя: скорость доступа к данным и пространство на диске, занимаемое данными. В связи с этим были определены два критерия сравнения:

1. скорость разархивирования
2. коэффициент сжатия

Прочие характеристики архиваторов, такие как скорость архивирования или объем используемой оперативной памяти, в данной работе не учитывались.

4.2. Тестовые данные

При проведении тестирования использовался корпус из более чем 750'000 валидных URL'ов, являющихся репрезентативной выборкой URL части .ru сегмента сети Интернет. В дальнейшем данный корпус будет обозначаться как *PoolURLs*.

Для сравнения качества сжатия коротких сообщений было создано несколько URL-выборок различных размеров. Размеры выборок можно условно разделить на 3 типа:

1. *small* - 1 URL среднего размера (30-45 байт)
2. *medium* - 60 URL ($\approx$ 2 Кбайт)
3. *large* - 6000 URL ($\approx$ 200 Кбайт)

Для каждого типа было создано по 3 представителя, набранных случайным образом. Размеры подготовленных тестовых файлов представлены в следующей таблице:

	small	**medium**	**large**
instance 1	41	1989	223816
instance 2	30	2150	227996
instance 3	45	2033	207293

Дополнительно тестирование проводилось на всем имевшемся множестве URL'ов. Использование *PoolURLs* должно было оценить качество работы архиваторов при сжатии больших массивов текстовых данных.

4.3. Реализации Хаффмана с расширенным словарем

Перед тем, как проводить сравнение существующих архиваторов, было проведено тестирование алгоритма, предложенного в данной работе. Несмотря на то, что сжатие файлов осуществляется по строго определенному принципу, подходов к генерации словаря было представлено несколько (два). Словари, сгенерированные различными подходами, могут значительно отличаться, что неизбежно повлечет за собой разное качество сжатия данных.

Итеративное наполнение словаря

Первый подход к генерации словаря базируется на постепенном наполнении словаря новыми комбинациями символов, каждая из которых признается наиболее полезной на какой-либо итерации алгоритма. Как было сказано ранее, этот подход подразумевает обучение словаря на некоторой фиксированной выборке из *PoolURLs*.

В данной работе при тестировании использовались обучающие выборки из 10% и 25% данных из массива *PoolURLs*. При подготовке каждого словаря обучающая выборка набиралась рандомно, фиксированным был лишь ее размер. Также было опробовано различное количество итераций алгоритма: 2000, 10000, 50000 и 100000. Для каждого числа итераций обучение выполнялось по 3 раза, чтобы исключить возможные статистические выбросы.

Ниже представлены результаты сжатия *PoolURLs* с использованием словарей, обученных в различных конфигурациях. Значения в таблицах означают отношение размера сжатого файла к размеру исходного файла, выраженное в процентах.

	2'000	**10'000**	**50'000**	**100'000**
attempt 1	40.87	36.66	32.66	32.44
attempt 2	40.86	36.31	32.49	32.11
attempt 3	41.04	36.30	32.80	32.03

Таблица 1: Словари обучены на 10% данных *PoolURLs*

	2'000	**10'000**	**50'000**	**100'000**
attempt 1	40.86	35.97	31.23	29.63
attempt 2	40.89	36.20	31.44	29.14
attempt 3	41.35	36.24	31.38	29.11

Таблица 2: Словари обучены на 25% данных *PoolURLs*

Из приведенных таблиц можно сделать вывод, что при обучении на 10%-ной выборке увеличение числа итераций с 50'000 до 100'000 не приводит к ощутимому изме-

нению результата. Возникает предположение, что достигнутое качество сжатия (около 32%) является максимальным для данного тестового файла *PoolURLs* и 10%-ной обучающей выборки. В свою очередь, у словарей с 25%-ной выборкой таких проблем не наблюдается. Вероятно, их максимальный порог сжатия можно достичь при еще большем числе итераций.

Также из данных таблиц видно, что размер обучающего множества начинает играть заметную роль только при большом числе итераций. Для словарей, созданных всего за 2'000 итераций, обучающие множества различных размеров дают приблизительно одинаковый результат.

Среднее время генерации словаря в зависимости от размера обучающего множества и числа итераций представлено в Таблице 3.

	2'000	**10'000**	**50'000**	**100'000**
10%	90	364	3058	9469
25%	156	522	4116	12466

Таблица 3: Среднее время обучения словаря, сек

Полученные результаты показывают, что размер обучающего множества ощутимо влияет на скорость обучения. При каждом числе итераций время обучения возрастает примерно на треть при увеличении объема обучающей выборки.

Метод последовательных приближений

Другой подход к генерации словаря, описанный в данном дипломе, работает по принципу улучшения существующего результата: каждая итерация алгоритма пытается улучшить тот словарь, который уже получился. У этого подхода существует несколько настраиваемых параметров, что усложняет тестирование:

- количество итераций
- размер расширенного словаря
- размер выборки для обучения
- размер выборки для построения таблицы частот

На первых стадиях тестирования проводились замеры, насколько число итераций влияет на качество обученного словаря. Тесты проводились на числе итераций 50, 100 и 200. Было замечено, что на 100 итерациях результаты в среднем лучше, чем на 50, однако между 100 и 200 итерациями не было никакой разницы. В связи с этим, все дальнейшие эксперименты проводились исключительно с количеством итераций, равным 100.

Для построения таблицы частот на последнем этапе обучения словаря использовалась 25%-ная выборка из данных *PoolURLs*. Такой размер выборки кажется достаточным для сбора достоверной статистики использования элементов словаря.

При проведении тестирования использовались следующие значения параметров:

- размер расширенного словаря: 2'000, 10'000, 50'000, 10'000
- размер выборки для обучения: 2%, 5%, 10%

Размер выборки для обучения определялся как процент от всего массива данных, то есть от *PoolURLs*. По каждой конфигурации обучение словаря выполнялось по 3 раза, в таблицах приведены усредненные значения. Как и в случае с первым подходом, качество обученных словарей проверялось на всем корпусе данных *PoolURLs*.

	2'000	**10'000**	**50'000**	**100'000**
2%	44.68	37.97	34.84	34.88
5%	46.17	38.61	32.37	32.08
10%	45.95	38.77	31.82	29.94

Таблица 4: Качество сжатия с помощью обученных словарей, % от исходного размера

Полученные результаты оказались достаточно противоречивыми. С одной стороны, как и стоило ожидать, качество сжатия растет с увеличением обучающей выборки. Это хорошо заметно на словарях размером 50'000 и 100'000. Однако на меньших размерах словаря качество сжатия не улучшается, а иногда и ухудшается с ростом обучающего множества. Более того, при размерах словаря в 2'000 и 10'000 среднее отклонение при измерениях составляло от 0.2 до 0.8 процентов, то есть качество полученного словаря было достаточно нестабильным. При больших размерах словаря среднее отклонение не превышало 0.03%.

	2'000	**10'000**	**50'000**	**100'000**
2%	11	16	18	18
5%	27	38	48	52
10%	54	72	94	108

Таблица 5: Время обучения словарей, сек

Из Таблицы 5 видно, что время обучения линейно пропорционально объему обучающей выборки. Это правило сохраняется (с небольшими отклонениями) для каждого размера словаря.

Сравнение методов

Из проведенного тестирования следует, что первый подход генерирует словари качественнее, чем второй. Однако время генерации различается столь сильно, что встает вопрос о применимости первого метода в реальных системах, когда размер обучающей выборки будет исчисляться сотнями мегабайт. Такой подход будет хорош лишь в системах со статической структурой сжимаемых данных, когда нет необходимости часто обучать словарь заново.

Помимо этого, стоит отметить, что второй подход очень плохо генерирует словари небольшого размера. Проведенные эксперименты свидетельствуют, что его словари из 2'000 и 10'000 элементов хуже по качеству и более нестабильны, чем в первом подходе. При генерации небольших словарей использование первого метода кажется более предпочтительным на фоне лучшего качества и сравнительно небольшого времени обучения.

4.4. Выбор архиваторов для сравнения

Следующим этапом в процессе тестирования является сравнение подхода, предложенного в данной работе, с существующими архиваторами. Но прежде чем проводить сравнение, необходимо определить, какие именно архиваторы будут участвовать в тестировании.

Прежде всего, предлагается взять 3 архиватора из тех, которые демонстрируют наибольшую степень сжатия текстовых данных. Опираясь на таблицу, представленную в разделе 2.2, такими архиваторами являются

- durilca'kingsize
- cmix v1
- paq8hp12any

однако архиватор *durilca'kingsize* является закрытым коммерческим продуктом. Его реализацию не удалось найти в Интернете, в связи с чем его тестирование оказалось невозможным. Вместо него был использован следующий по качеству архиватор *nanozip 0.09a*.

Помимо специализированных архиваторов в тестировании принял участие ряд широкоиспользуемых архиваторов. Было любопытно узнать, насколько хорошо они справятся со сжатием коротких текстовых сообщений. Среди выбранных программ:

- gzip (Apple gzip 2)
- bzip2 (ver. 1.0.6)
- 7zip (p7zip ver. 9.20)

Еще один алгоритм, выбранный для тестирования - это аналог предложенного в данной работе алгоритма, только без расширенного словаря. Перед тем, как сжимать данные, этот алгоритм обходит весь корпус *PoolURLs* и строит коды Хаффмана для каждого символа. В дальнейшем вся работа этого архиватора происходит исключительно на основе подготовленных кодов. Этот алгоритм призван продемонстрировать, насколько оправданно создание словаря и присвоение каждому элементу словаря собственного кода Хаффмана.

Последним тестируемым архиватором является алгоритм, описанный в данной работе. Поскольку созданные словари могут сильно варьироваться по качеству (в зависимости от конфигурации генератора), для тестирования был выбран наилучший из созданных словарей.

4.5. Сравнение с существующими архиваторами

Сравнение архиваторов проводилось отдельно на каждом из трех типов тестовых данных *small*, *medium*, *large*, а также на всем массиве URL'ов *PoolURLs*. В каждой из приведенных таблиц *степень сжатия* означает, какой процент от исходного размера файла составляет размер сжатого файла. Столбец *разархив.* означает время, затраченное на разархивирование, подсчитанное в миллисекундах. Для всех алгоритмов замеры проводились на каждом из 3 экземпляров тестового типа данных, после чего показатели усреднялись.

Данные типа *small*. 30-45 байт

Программа	**Степень сжатия**	**Разархив., мс**
cmix	116.80	22287
paq8hp12	159.08	22433
nanozip	297.20	13
gzip	180.39	1
bzip2	197.34	1
7zip	420.52	4
Huffman	75.20	1
Huffman with V	**47.15**	1

Таблица 6: Сравнение алгоритмов на данных типа *small*

Результаты, представленные в Таблице 6, наглядно демонстрируют неспособность тестируемых алгоритмов сжимать короткие текстовые сообщения. Все архиваторы, кроме предложенного в работе, в результате архивирования увеличивали размер ис-

ходного файла, сводя на Нет сам смысл архивирования. Единственным алгоритмом (помимо предложенного), действительно сжавшим файл, оказался *Хаффман без словаря*, однако его качество сжатия заметно уступает *Хаффману со словарем*.

Данные типа *medium*. ≈ 2 Кбайта

Программа	**Степень сжатия**	**Разархив., мс**
cmix	41.09	23600
paq8hp12	41.75	22657
nanozip	53.04	17
gzip	51.53	1
bzip2	52.59	1
7zip	57.30	4
Huffman	64.29	1
Huffman with V	**27.91**	1

Таблица 7: Сравнение алгоритмов на данных типа *medium*

Данные Таблицы 7 свидетельствуют о том, что *Хаффман со словарем* оказывается лучшим и при сжатии более длинных текстовых сообщений. Стоит отметить, что на файлах данного размера все тестируемые алгоритмы смогли уменьшить размер исходного файла.

Данные типа *large*. ≈ 200 Кбайт

Программа	**Степень сжатия**	**Разархив., мс**
cmix	**24.97**	112930
paq8hp12	25.15	33420
nanozip	32.06	50
gzip	37.51	3
bzip2	33.36	14
7zip	33.19	11
Huffman	65.18	27
Huffman with V	28.91	16

Таблица 8: Сравнение алгоритмов на данных типа *large*

Результаты Таблицы 8 показывают, что алгоритмы *cmix* и *paq8hp12* не случайно

считаются одними из лучших в сжатии текстовых данных. На тестируемых файлах типа *large* они смогли "обогнать" алгоритм, описанный в данной работе. Однако их скорость разархивирования столь велика, что оба этих алгоритма оказываются непригодны для использования в высоконагруженных поисковых системах. Поэтому *Хаффман со словарем* можно считать лучшим архиватором и для файлов типа *large*.

Полный массив данных *PoolURLs*. 27.5 Мбайт

Программа	**Степень сжатия**	**Разархив., мс**
cmix	–	–
paq8hp12	**16.83**	1618470
nanozip	21.08	2280
gzip	37.34	129
bzip2	30.47	1097
7zip	23.93	421
Huffman	65.18	2220
Huffman with V	29.11	1212

Таблица 9: Сравнение алгоритмов на всем корпусе *PoolURLs*

Таблица 9 демонстрирует, насколько компактно различные алгоритмы сжимают большие текстовые данные. Как видно, практически все алгоритмы достигают сжатия исходного файла до 30%. Отдельное упоминание необходимо сделать про алгоритм *cmix*. Результаты по данному архиватору не были получены, поскольку спустя час сжатие все еще не было закончено, и программу было решено остановить. Касательно алгоритма, предложенного в данной работе - он оказался сравним с архиватором bzip2, как по качеству сжатия, так и по времени работы.

Выводы

Подводя итог, можно сказать, что алгоритм, описанный в данной работе, способен эффективно сжимать короткие текстовые сообщения, демонстрируя при этом высокую скорость разархивирования. Сравнивая данный алгоритм с *Хаффманом без словаря*, становится очевидным, что использование словаря оправданно и помогает лучше сжимать текстовые данные.

Заключение

В ходе дипломной работы были выполнены следующие задачи:

1. изучены существующие алгоритмы сжатия данных;
2. определены архиваторы, наилучшим образом сжимающие текстовые данные;
3. предложен и реализован прототип своего архиватора, основанного на кодах Хаффмана с расширенным словарем;
4. проведено сравнение существующих архиваторов с собственным архиватором.

В результате проделанной работы было определено, что предложенный алгоритм эффективен для сжатия текстовых сообщений, как длинных, так и очень коротких.

Отдельное внимание стоит уделить генераторам словаря, реализованным в процессе работы. Словарь, полученный с их помощью, имеет широкий набор применений, отличных от обычного архивирования. Этот факт делает полученный результат еще более ценным, а вероятность его внедрения в реальные поисковые системы - еще более высокой.

Список литературы

[1] Cleary John G., Witten Ian H. Data Compression Using Adaptive Coding and Partial String Matching. Transactions on Communications, vol. COM-32, No. 4. — IEEE, 1984.

[2] CompressionRatings.com. Compression Ratings. Text 2 // Compression Ratings. — 2012. — http://compressionratings.com/txt2.html.

[3] Mahoney Matthew V. Adaptive Weighing of Context Models for Lossless Data Compression. — Florida Tech : Technical Report CS-2005-16, 2005.

[4] MattMahoney.net. Large Text Compression Benchmark // Matt Mahoney. — 2014. — http://mattmahoney.net/dc/text.html.

[5] MaximumCompression.com. English Text compression test // Maximum Compression. — 2006. — http://www.maximumcompression.com/data/text.php.

[6] MaximumCompression.com. Lossless data compression theory and algorithms // Maximum Compression. — 2011. — http://www.maximumcompression.com/algoritms.php.

[7] Proof of Optimality of Huffman Codes. — University of Toronto : CSC373, 2009.

[8] Sayood Khalid. Introduction to Data Compression, 2nd. edition. — San Francisco : Morgan Kaufmann, 2000.

[9] Wikipedia. Context mixing // Wikipedia, The Free Encyclopedia. — 2013. — http://en.wikipedia.org/wiki/Context_mixing.

[10] Wikipedia. Алгоритс сжатия PPM // Википедия, свободная энциклопедия. — 2013. — http://ru.wikipedia.org/wiki/Алгоритм_сжатия_PPM.

[11] Wikipedia. Преобразование Барроуза-Уилера // Википедия, свободная энциклопедия. — 2013. — http://ru.wikipedia.org/wiki/Преобразование_Барроуза_Уилера.

[12] Wikipedia. Entropy encoding // Wikipedia, The Free Encyclopedia. — 2014. — http://en.wikipedia.org/wiki/Entropy_encoding.

[13] Wikipedia. Poisson limit theorem // Wikipedia, The Free Encyclopedia. — 2014. — http://en.wikipedia.org/wiki/Poisson_limit_theorem.

[14] Wikipedia. Код Хаффмана // Википедия, свободная энциклопедия. — 2014. — http://ru.wikipedia.org/wiki/Код_Хаффмана.

[15] Wikipedia. Кодирование длин серий // Википедия, свободная энциклопедия. — 2014. — http://ru.wikipedia.org/wiki/Кодирование_длин_серий.