Министерство образования и науки Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

«Анализ и разработка алгоритмов сжатия коротких текстов»

Автор: Минаев Борис Юрьевич _____
Направление подготовки (специальность): 01.04.02 Прикладная математика и информатика
Квалификация: Магистр
Руководитель: Буздалов М.В., канд. техн. наук _____

К защите допустить
Зав. кафедрой Васильев В.Н., докт. техн. наук, проф. _____

«___» _____ 20___ г.

Санкт-Петербург, 2017 г.

Студент Минаев Б.Ю. **Группа** М4239 **Кафедра** компьютерных технологий **Факультет** информационных технологий и программирования

Направленность (профиль), специализация Технологии проектирования и разработки программного обеспечения

Квалификационная работа выполнена с оценкой _____

Дата защиты «___» _____ 20___ г.

Секретарь ГЭК *Павлова О.Н.* Принято: «___» _____ 20___ г.

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

УТВЕРЖДАЮ
Зав. каф. компьютерных технологий
докт. техн. наук, проф.
_____ Васильев В.Н.
«___» _____ 20___ г.

ЗАДАНИЕ
НА МАГИСТЕРСКУЮ ДИССЕРТАЦИЮ

Студент Минаев Б.Ю. **Группа** М4239 **Кафедра** компьютерных технологий **Факультет** информационных технологий и программирования **Руководитель** Буздалов Максим Викторович, канд. техн. наук, доцент кафедры компьютерных технологий Университета ИТМО

1 Наименование темы: Анализ и разработка алгоритмов сжатия коротких текстов

Направление подготовки (специальность): 01.04.02 Прикладная математика и информатика

Направленность (профиль): Технологии проектирования и разработки программного обеспечения

Квалификация: Магистр

2 Срок сдачи студентом законченной работы: «___» _____ 20___ г.

3 Техническое задание и исходные данные к работе.

Требуется проанализировать существующие алгоритмы сжатия текстовых данных. Оценить их применимость к сжатию коротких текстовых сообщений, посылаемых в социальных сетях. Реализовать собственный алгоритм сжатия текстовых сообщений, который можно было бы применить для уменьшения обьемов исползуемой памяти на серверах, которые хранят текстовые сообщения.

4 Содержание магистерской диссертации (перечень подлежащих разработке вопросов)

а) Изучение существующих алгоритмов сжатия текстовых данных
б) Изучение особенностей посылаемых в социальных сетях текстовых сообщений с точки срения их сжатия
в) Разработка нового алгоритма сжатия
г) Выводы

5 Перечень графического материала (с указанием обязательного материала)

Не предусмотрено

6 Исходные материалы и пособия

а) Б.Д.Кудряшов. Основы теории кодирования

7 Календарный план

№№ пп.	Наименование этапов магистерской диссертации	Срок выполнения этапов работы	Отметка о выполнении, подпись руков.
1	Изучение существующих алгоритмов сжатия	12.2015	
2	Изучение особенностей данных	03.2016	
3	Применение существующих алгоритмов, выяснение их достоинст и недостатков	06.2016	
4	Разработка собственного алгоритма, сравнение с аналогами	03.2017	
5	Написание пояснительной записки	05.2017	

8 Дата выдачи задания: «___» _____ 20___ г.

Руководитель _____

Задание принял к исполнению _____ «___» _____ 20___ г.

Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

АННОТАЦИЯ
МАГИСТЕРСКОЙ ДИССЕРТАЦИИ

Студент: Минаев Борис Юрьевич

Наименование темы работы: Анализ и разработка алгоритмов сжатия коротких текстов

Наименование организации, где выполнена работа: Университет ИТМО

ХАРАКТЕРИСТИКА МАГИСТЕРСКОЙ ДИССЕРТАЦИИ

1 Цель исследования: Разработка алгоритма сжатия коротких текстовых сообщений

2 Задачи, решаемые в работе:

а) уменьшение количества памяти, необходимое для хранение текстовых сообщений

б) увеличиние скорости считывания сообщений с диска

3 Число источников, использованных при составлении обзора: _____

4 Полное число источников, использованных в работе: 12

5 В том числе источников по годам

Отечественных			Иностранных		
Последние 5 лет	От 5 до 10 лет	Более 10 лет	Последние 5 лет	От 5 до 10 лет	Более 10 лет

6 Использование информационных ресурсов Internet: _____

7 Использование современных пакетов компьютерных программ и технологий: Для сравнения алгоритмов сжатия они были реализованы на языке программирования Java. Разработанный алгоритм был переписан на язык C для увеличения скорости его работы. Для написания автоматических тестов был использован язык PHP. Для визуализвции данных был использован Python и matplotlib.

8 Краткая характеристика полученных результатов: Был разработан алгоритм сжатия коротких текстовых сообщений, который был применен в ООО «В Контакте». По сравнению с предыдущим используемым алгоритмом размер сжатых сообщений уменьшился на 5-7%. Такое улучшение соответсвует экономии нескольких терабайт оперативной памяти на масштабах ООО «В Контакте»

9 Гранты, полученные при выполнении работы: Грантов получено не было.

10 Наличие публикаций и выступлений на конференциях по теме работы: Публикацей подготовлено не было.

Выпусник: Минаев Б.Ю. _____

Руководитель: Буздалов М.В. _____

«___» _____ 20___ г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В последнее время количество информации во всем мире стремительно увеличивается. Согласно [1] за каждые два года суммарное количество данных в интернете увеличивается в два раза. Аналогичные результаты подтверждаются исследованиями, которые были проведены автором во время работы в компании ВКонтакте. Эта компания владеет одноименным сайтом, на котором пользователи могут обмениваться сообщениями. На рис. 1 показано количество сообщений, отправленных пользователями в течении разных месяцев. Эти данные заставили задуматься о пересмотре архитектуры хранения сообщений, а также улучшении эффективности их обработки.

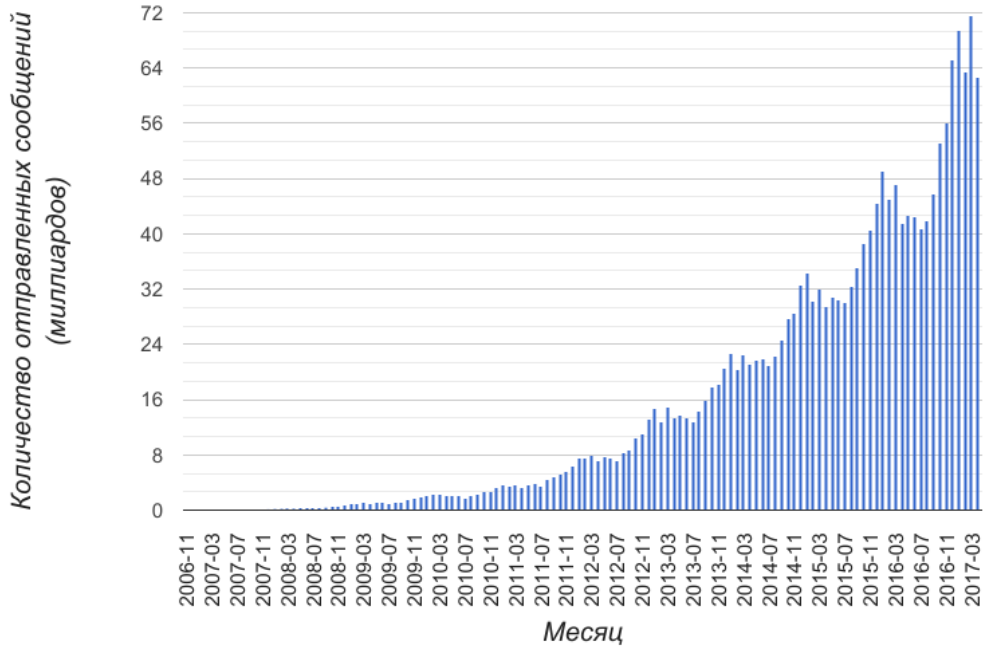

Рисунок 1 – Количество отправляемых сообщений

Конечной целью улучшения способа хранения пользовательских сообщений является уменьшение количества серверов, на которых работает сервис, а также улучшение качества его работы. К последнему в основном относится скорость работы. Интересует как средняя скорость ответа на запросы, так и, например, ответ на вопрос «какая часть запросов выполняется дольше секунды». Забегая вперед, хочется сказать, что

все эти показатели удалось существенно улучшить, уменьшив при этом количество используемых серверов в два раза.

Одним из способов оптимизации хранения сообщений является их сжатие. Оно позволяет уменьшить количество занимаемого места в несколько раз. Однако, сервис обмена сообщениями должен предоставлять пользователю возможность прочитать любое свое сообщение, а также уметь искать по ним, что накладывает некоторые ограничения на алгоритмы, которые можно использовать для сжатия. Существует огромное количество алгоритмов сжатия данных без потерь. История их создания начинается в середине прошлого века [2].

Интересным примером того, как хорошо сейчас умеют сжимать данные является [3]. Участникам соревнования предлагается написать алгоритм сжатия первого миллиарда символов xml-версии английской википедии. На текущий момент лучший алгоритм смог сжать исходные данные в 8.29 раза. Это действительно впечатляющий результат, однако, таких коэффициентов сжатия сложно добиться в реальных задачах. Проблема связана как со спецификой данных (в исходных данных для соревнования много xml-сущностей, которые хорошо сжимаются), так и в ресурсах, которые необходимы алгоритму. Так для сжатия гигабайта Википедии алгоритм-победитель использовал больше недели процессорного времени и 27 дополнительных гигабайт оперативной памяти.

Проблема применения существующих алгоритмов для сжатия сообщений пользователей в том, что необходимо либо сжимать каждое сообщение отдельно, либо уметь разархивировать произвольный кусок данных. В первом случае большинство алгоритмов сжатия теряют свою эффективность из-за маленького размера сообщений. А алгоритмы, которые могут разжимать произвольный кусок данных достаточно эффективно, автору работы не известны.

В данной работе рассмотрены различные способы сжатия текстовых сообщений короткой длины, а также представлен новый способ, основанный на алгоритме Хаффмана.

ГЛАВА 1. ОБЗОР

1.1. Способ сравнения алгоритмов

Перед тем как выбирать лучший алгоритм сжатия необходимо определиться с тем, как их сравнивать. Поскольку выбор метрик сильно зависит от места, где используется алгоритм, рассмотрим архитектуру системы хранения сообщений в социальной сети «ВКонтакте», для которой и разрабатывался алгоритм.

До середины 2016 года сообщения хранились на 1000 серверах, при этом сообщения конкретного пользователя хранились на одном заранее выбранном сервере. У такой архитектуры есть очевидный недостаток — текст сообщения хранится несколько раз. Если это личная переписка, то он хранится два раза, а если это мультичат, то столько раз, сколько людей в чате. Популярность мультичатов стремительно возрастала, и было принято решение переделать архитектуру таким образом, чтобы текст каждого сообщения хранился ровно один раз.

Также во время перехода на другую архитектуру были улучшены алгоритмы кеширования и сжатия сообщений. Последнему и посвящена данная работа. Новая архитектура изображена на рис. 9. Она состоит из двух слоев. Внутренний слой, состоящий из сервисов, называемых chat-engine, хранит тексты сообщений, поисковые индексы и информацию об участниках чатов. Каждый реальный чат хранится ровно в одном из этих сервисов. Внешний слой состоит из сервисов, называемых user-engine, они хранят списки сообщений, которые есть у пользователей, а также кешируют тексты сообщений. Информация о конкретном пользователе хранит ровно один сервис внешнего слоя.

Когда пользователь посылает сообщение, он отправляет его в соответствующий ему user-engine, который, зная в какой чат послано сообщение, отправляет его в нужный chat-engine, который в свою очередь, рассылает данное сообщение всем участникам чата в нужные user-engine. Когда пользователь хочет получить сообщения, он посылает запрос соответствующему ему user-engine, который хранит список всех его сообщений с ссылками на нужные чаты. User-engine запрашивает нужные сообщения из chat-engine (либо берет их из своего кеша) и отдает пользователю.

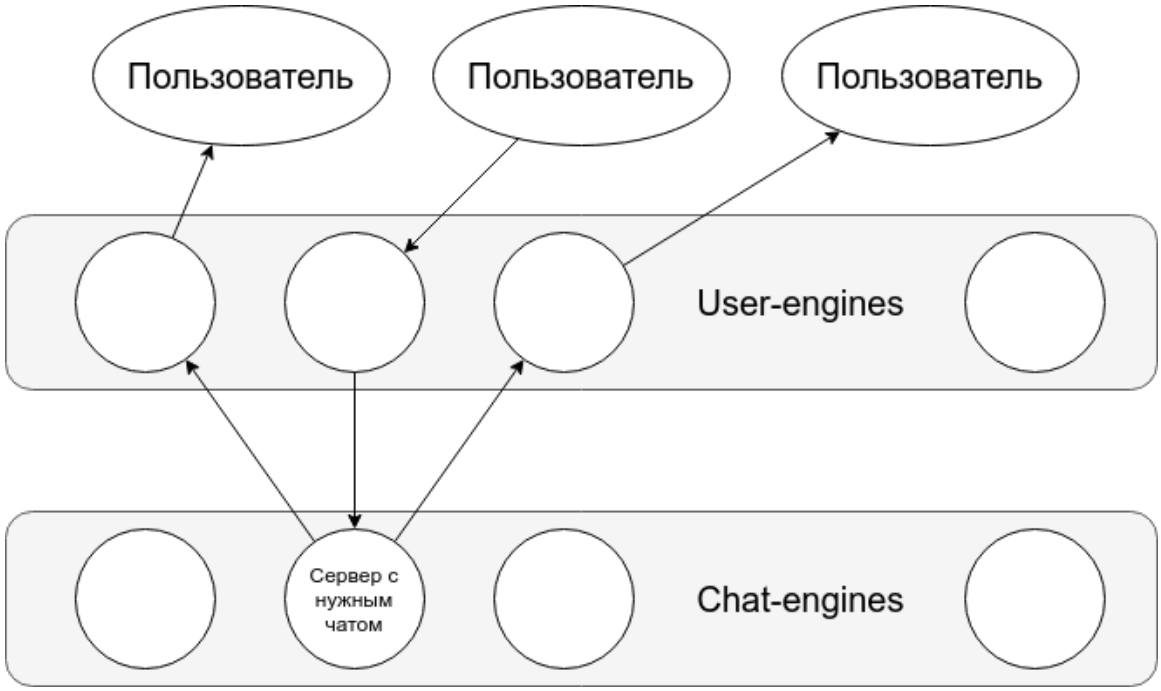

Рисунок 2 – Отправка сообщения во «В Контакте»

Все эти сервисы располагаются на 500 серверах, каждый из которых имеет по 128 гигабайт оперативной памяти. На каждом сервере запущено восемь user-engine и восемь chat-engine (по количеству жестких дисков). Каждому user-engine выделено 8 гигабайт оперативной памяти, а каждому chat-engine 4,5 гигабайта. Нас интересует способ хранения сообщений в chat-engine. На каждый chat-engine приходится в среднем 500 миллионов сообщений суммарным размером 13 гигабайт. Из 4,5 гигабайт памяти, которая выделена chat-engine, он тратит примерно половину на поисковый индекс и хранение метаинформации сообщений. На хранение текстов сообщений остается в среднем 2,5 гигабайта.

Таким образом, если бы удалось сжать сообщения в 5 раз, то все сообщения поместились бы в память и не было бы никаких проблем. Но нам не удалось достичь такой степени сжатия, поэтому сообщения подгружаются с диска по мере необходимости. Если же свободная память заканчивается, то сообщения чата, которые дольше всего не спрашивали, выгружаются из памяти.

Из описанной архитектуры следует, что самую важную роль при выборе алгоритма сжатия играет общий размер сообщений, который может быть сохранен в памяти. При этом следует учитывать, что данная архитектура позволяет сохранить некоторую дополнительную ин-

формацию и использовать ее при сжатии и расжатии данных. Можно считать, что у алгоритма есть в распоряжении примерно 2,5 гигабайта памяти, которую он может использовать под дополнительную информациюю (обозначим ее через $additional\ info$) и хранение сжатых сообщений ($compressed\ size$). Обозначим суммарный размер исходных сообщений как $messages\ size$, тогда $compress\ ratio = \frac{messages\ size}{compressed\ size + additional\ info}$. Как раз $compress\ ratio$ нам и нужно оптимизировать. Чем он больше — тем лучше.

Также важную роль играет скорость, с которой алгоритм может сжимать и расжимает данные. Нельзя допустить, чтобы часто случалась ситуация, когда сообщения отдаются пользователю слишком долго. Это условие довольно сложно описать формально. Во-первых, чем быстрее средняя скорость сжатия/расжатия, тем лучше. Но при этом будет плохо, если будет много случаев, когда сообщения отдаются дольше секунды. Также важно не только время конвертации одного сообщения, но и суммарная нагрузка на процессор, которая создается за счет конвертации данных.

1.2. Существующие алгоритмы сжатия

На данный момент существует огромное множество алгоритмов сжатия данных. Нас интересуют только алгоритмы, которые сжимают данные без потерь. Среди них нас интересуют те, которые ориентированы на сжатие текстовых данных. Далее будут рассмотрены некоторые из них. Более детально можно ознакомиться с ними в [4].

1.2.1. Run-Length encoding

Кодирование длин серий (или RLE) — один из самых простых алгоритмов сжатия данных. Он работает хорошо, если в тексте много подряд идущих одинаковых символов. Идея алгоритма заключается в том, чтобы заменить n подряд идущих символов c на строку nc. Например, строка ABAAACCCCCA будет заменена на 1A1B3A5C1A.

В показаном примере размер строки уменьшился с 11 символов до 10, однако очень просто привести пример на котором размер строки увеличится. Например, строка ABC превратится в 1A1B1C и станет в два раза длинее.

Данный метод редко используют на практике для сжатия текстовых данных, однако во многих современных алгоритмах сжатия текста используется LZ77, который является обощением RLE [5].

1.2.2. LZ77

Идея алгоритма заключается в том, чтобы ссылаться на предыдущее вхождение текста. Более формально — кодировщик хранит последние несколько килобайт закодированных данных, а когда хочет закодировать очередной кусок текста, находит наибольшее вхождение текста, которое он помнит, и выписывает пару чисел, которые обозначают как давно он видел этот текст и какой он длины.

Согласно [6] одним из недостатков данного метода является малая эффективность при кодировании небольшого объема данных. В решаемой нами задаче средняя длина сообщения очень маленькая, поэтому алгоритм скорее всего не даст хороших результатов.

1.2.3. PPM

Предсказание по частичному совпадению (Prediction by Partial Matching) — алгоритм, который предсказыавет вероятность появления очередного символа основываясь на предыдущем опыте. Параметром модели PPM является число n — максимальный размер контекста, который анализируется. Обычно n порядка нескольких символов. Для того, чтобы оценить вероятность появления конкретной очередной буквы рассматриваются все подстроки длины n, которые совпадают с последними n символами. Рассматриваются все символы, которые следуют после найденных подстрок. Если нужного символа нет, то записывается символ-исключение и рассматривается контекст размера $n - 1$ аналогичным образом. Так происходит пока не найдется подстрока, после которой идет нужная буква. Если такой строки найти не удалось, используется контекст степени -1, в котором есть все нужные символы.

К этому алгоритму также применяется следующая оптимизация. Если контекст длины n не подошел, то точно известно, что необходимый символ точно не тот, который мог бы следовать после подстроки длины n. Поэтому вероятности этих символов при рассмотрении контекста размера $n - 1$ можно прировнять нулю.

Заметим, что данный алгоритм только предсказывает вероятность увидеть очередной символ, но не говорит как нужно кодировать данные. Поэтому его нужно использовать вместе с алгоритмом энтропийного кодирования. Например, можно использовать алгоритм Хаффмана или арифметическое кодирование.

1.2.4. Алгоритм Хаффмана

Алгоритм Хаффмана — алгоритм, который был изобретен в середине прошлого века. Идея заключается в следующем. Пусть для каждого символа i известна вероятность того, что он сейчас встретится p_i. Задача состоит в том, чтобы сопоставить каждому символу битовую строку s_i и минимзировать $\sum p_i \cdot |s_i|$. При этом необходимо, чтобы не было двух строк таких, что одна является префиксом другой.

Оказывается, что решить эту задачу можно с помощью следующего жадного алгоритма. Рассмотрим два символа с наименьшими вероятностями. Заменим их на один виртуальный символ, вероятность которого равна сумме вероятностей исходных символов. Будем повторять этот процесс пока не останется один символ. После этого построим двоичное дерево следующим образом. Возьмем единственный символ, он будет корнем дерева. Будем повторять следующую операцию пока возможно. Рассмотрим лист дерева, который соответсвует виртуальному символу. Добавим этой вершине двух детей и скажем, что они соответствуют символам, из которых была собрана текущая виртуальная вершина.

После того как дерево построено, каждому листу в нем соответствует некоторый символ исходного алфавита. Также ему можно сопоставить путь из корня до него. Запишем его как битовую строку, в которой переход к левому ребенку соответсвует нулю, а к правому — единице. Скажем, что исходному символу соответствует полученная битовая строка.

1.2.5. Арифметическое кодирование

Недостатком алгоритма Хаффмана является то, что он не может закодировать символ нецелым числом бит. Этого недостатка лишено арифметическое кодирование. Оно работает следующим образом. Каждому сообщению сопоставляется некоторое вещественное число. Изначально берется отрезок [0, 1] и разбивается на столько подотрезков,

сколько символов алфавита. При этом размеры отрезков должны быть пропорциальны вероятности получения символа.

Перейдем к отрезку, который соответсвует первому символу сообщения и разобъем его опять в таком же соотношении (либо в другом, если вероятности встретить конкретные символы поменялись). Перейдем в отрезок, который соответствует второму символу, и. т. д. После рассмотрения последнего символа выберем вещественное число из отрезка, которое записывается с помощью наименьшего числа бит. Это и будет закодированное сообщение.

Эффективность такого сжатия строго не хуже алгоритма Хаффмана, а в большинстве случаев превосходит его. Однако, эффективная реализация данного алгоритма достаточно затруднительна.

1.2.6. Современные архиваторы

Сейчас существует очень много различных архиваторов, но все они основаны на одних и тех же принципах, которые применяются в алгоритмах, описанных выше. Сейчас стандартом сжатия является Deflate [7], который используется в zip, gzip, zlib и других архиваторах. Сам алгоритм является смесью LZ77 и алгоритма Хаффмана.

Также интересен алгоритм zstd [8], который был разработан в Facebook, и сочетает в себе LZ77 и энтропийное кодирование типа Finite State Entropy.

ГЛАВА 2. РАЗРАБОТКА АЛГОРИТМА

Большинство алгоритмов сжатия обучают свою модель в процессе сжатия текста основываясь на только что полученных данных. Исходные данные, которые рассматриваются в нашей задаче, не позволяют так делать, так как данных одного сообщения зачастую слишком мало, чтобы заметить в них закономерности. Поэтому необходим алгоритм, который может обучиться на полном наборе сообщений, а потом использовать полученные знания для кодирования отдельных сообщений. При этом полученные знания должны быть сохранены достаточно компактно.

2.1. Однобуквенный Хаффман

В качестве основы для нашего решения был взят алгоритм Хаффмана [?], так как он позволяет заранее построить модель на исходных данных, а потом использовать ее для сжатия отдельных сообщений. Идея алгоритма достаточно проста. Пусть у нас есть алфавит из n символов. Посчитаем cnt_i — количество раз, которое символ i встречается в исходном наборе сообщений. Положим $p_i = \frac{cnt_i}{\sum cnt_i}$ — вероятность встретить символ i. Далее каждому символу i ставится в соответствие некоторая битовая строка s_i, такая что $\sum\limits_i |s_i| \cdot p_i$ как можно меньше и $\forall i \neq j : s_i$ не является префиксом s_j. Второе условие позволяет однозначно декодировать закодированное сообщение, а первое говорит о том, что код является оптимальным среди всех префиксных кодов. После того как получены строки s_i можно приступать к кодированию. Для этого алгоритм рассматривает все символы строки слева направо и заменяет символ i на битовую строку s_i.

Чтобы декодировать сообщение алгоритм рассматривает битовую запись полученных данных. Каждый раз выбирается строка s_i, которая является префиксом еще не декодированных данных, на выход записывается символ i, а из исходных данных выкидываются первые $|s_i|$ символов. Заметим, что существует только одна строка s_i, которая может быть взята, так как код является префиксным. Чтобы быстро осуществлять поиск подходящей строки s_i все строки удобно добавить в структуру данных префиксное дерево [9].

Достоинством этого алгоритма является количество данных, которые ему нужно хранить после обучения на исходном наборе сообще-

ний. А именно, единственное, что нужно знать алгоритму, это количество вхождений каждой буквы в сообщениях. Если считать, что размер алфавита 256, а количество вхождений хранится в 32-битном типе данных, то на хранение дополнительной информации потребуется всего $256 \times 4 = 1024$ байт = 1 кбайт. К сожалению, эффективность сжатия у такого алгоритма не очень высокая. В частности на тестовой выборке сообщений был получен коэффициент сжатия 1.37.

2.2. Хаффман по словам

Логичным улучшением этого подхода является изменение алфавита. А именно, в качестве алфавита можно использовать не 256 символов, а, например, слова [4]. Предлагается поддерживать два отдельных словаря. Один для того, что называется словами в обычном понимании, а другой для знаков препинания и всего остального. При этом в сообщениях слова из двух словарей должны строго чередоваться. Также при кодировании необходимо указать из какого словаря взято первое слово. Либо всегда считать, что оно из конкретного словаря, но тогда в него необходимо добавить «пустое» слово и, соответственно, подсчитать его вероятность. Для разделения на слова и знаки пунктуации будем использовать следующий метод. Разделим все символы, которые могут встречаться, на два множества. В первое отнесем большие и маленькие буквы (как латинские, так и кириллические), а во второе все остальные. Словом будем считать наибольший по включению набор соседних символов, которые лежат в одном множестве.

В [4] отмечены следующие недостатки данного метода:

а) Словари достаточно большие. Поэтому они могут не помещаться в память, и для работы с ними нужно использовать эффективные алгоритмы.

б) На входных данных маленького размера практически все слова различны и поэтому алгоритм не успеет обучиться.

в) Необходимо два прохода по данным: один чтобы разбить на слова и подсчитать вероятности, второй чтобы закодировать.

г) Словарь, посчитанный после первого прохода, необходимо передать декодеру, а он может быть очень большим, что сильно ухудшает эффективность сжатия.

д) Слова могут быть сколь угодно длинными, что создает дополнительные проблемы.

Также отмечается, что подобная версия алгоритма Хаффмана дает лучше сжатие чем та, которая сжимает отдельные символы, но работает медленнее.

Заметим, что большинство отмеченных недостатков не применимы к нашей задаче. Во-первых, нам в любом случае необходимо сделать двухпроходный алгоритм, так как алгоритм должен обучиться на большой выборке сообщений. Во-вторых, хоть каждое отдельное сообщение имеет короткую длину, суммарное количество данных очень большое, и поэтому алгоритм сможет посчитать необходимую статистику по встречаемости слов достаточно точно. Что касается больших словарей, то проблема действительно применима к нашей задаче и далее будут предложены пути ее решения.

2.3. Выбор размера словаря

Если действовать строго по алгоритму, который описан в [4], то размер словарей действительно получается очень большой. Поэтому актуальным является вопрос уменьшения его размера. Легко заметить, что можно выбросить из словаря слова, которые встречаются в тексте всего один раз, т. к. при записи их в несжатом виде суммарный размер данных не увеличится. Причем поскольку словарь всегда находится в памяти, а сами сообщения нет, то такое действие окажет точно положительный эффект.

Логичным продолжением данной идеи является выкидывание из словаря слов, которые встрчеаются меньше чем K раз. Но стает вопрос о том, как узнать какое K является оптимальным. Для того, чтобы ответить на этот вопрос нами был создан специальный набор сообщений, на котором проводилось тестирование. Этот набор состоял из реальных сообщений, которые были посланы пользователями. Каждое сообщение попало в этот набор с вероятностью $\frac{1}{200000} = 0.000005$, т.е. набор по размеру составляет 2% от сообщений, которые обрабатвыаются одним chat-engine. Суммарный его размер равен 310 мегабайт и в нем содержится 10 миллионов сообщений. В набор были добавлены только непустые сообщения.

Чтобы узнать правильное значение K был проведен ряд экспериментов. Размер выбраного набора был достаточно маленьким, так как время работы алгоритма достаточно большое, а хотелось провести много экспериментов. Но оказалось, что этого достаточно. Было проведено 5 групп экспериментов. В них было оставлено $5 \cdot 10^5, 10^6, 2{,}5 \cdot 10^6, 5 \cdot 10^6, 10^7$ сообщений. В каждой группе был запущен алгоритм для $K = 1..15$.

В каждом запуске алгоритма нас интересовала степень сжатия данных, а также степень сжатия с учетом словаря. Именно последнюю величину нужно было максимизировать. На рис. 3, 4, 5 показаны результаты некоторых экспериментов. Как можно заметить, результаты экспериментов для разного количества сообщений очень похожи. Оказалось, что для всех проведенных экспериментов наилучшим значеним K было 8. Т. е. оказалось, что оптимально оставлять в словаре слова, которые встретились в сообщениях восемь и больше раз вне зависимости от исходного количества сообщений. Разумеется, данная константа справедлива только при достаточно большом количестве сообщений и только на данных такого же типа как и данные, которые были рассмотрены.

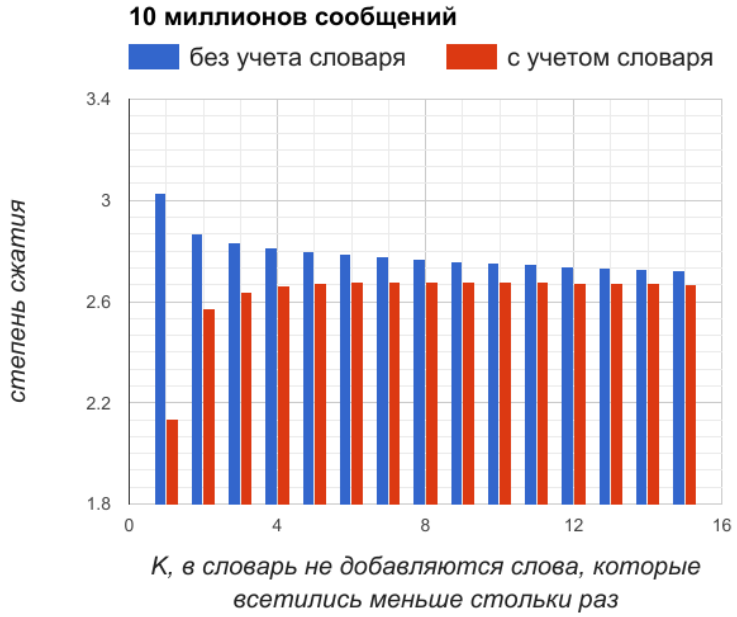

Рисунок 3 – Степень сжатия в зависимости от размера словаря (10 миллионов сообщений)

Также интересной статистикой является размер словаря, который получается при выборе различных K. График зависимости количества слов в словаре от того, начиная с какой частоты их добавлять в словарь,

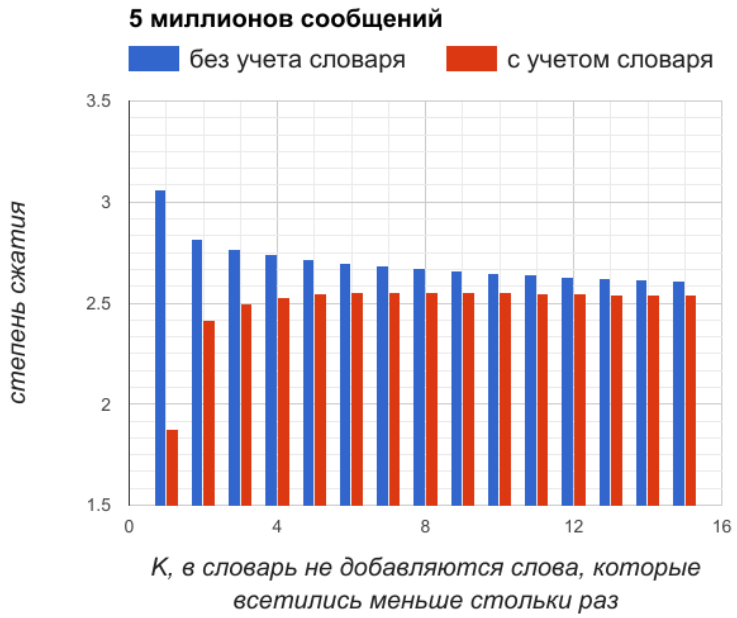

Рисунок 4 – Степень сжатия в зависимости от размера словаря (5 миллионов сообщений)

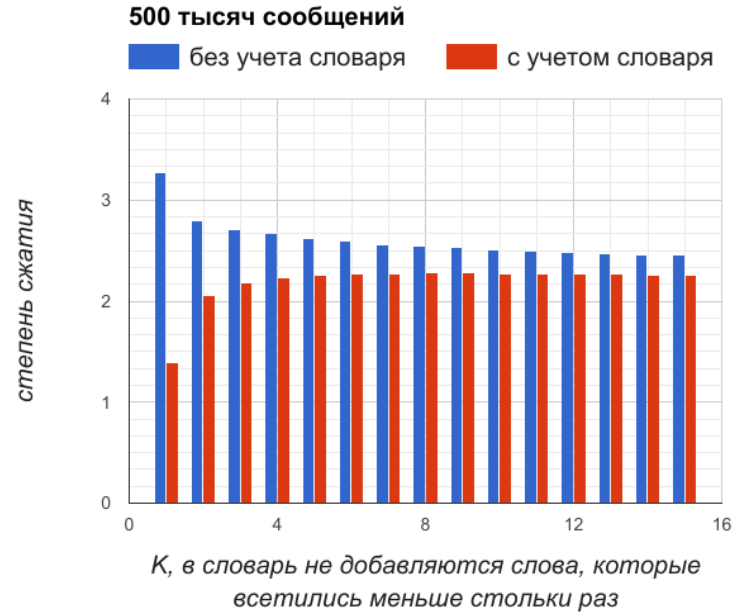

Рисунок 5 – Степень сжатия в зависимости от размера словаря (500 тысяч сообщений)

изображен на рис. 6. Аналогичный график с размером словаря изображен на рис. 7.

2.4. Кодирование слов, которые не попали в словарь

Описание алгоритма Хаффмана, который оперирует словами, встречается в небольшом количестве источников и во всех из них

Рисунок 6 – Количество слов в словаре в зависимости от K

Рисунок 7 – Размер словаря в зависимости от K

предлагается записывать слова, которые не встретились в словаре, как несжатую последовательность байт, предварительно написав символ-исключение. Такой подход разумен в случае, если в словарь добавляются все слова, которые встретились в тексте. Однако в нашем подходе для уменьшения размера словаря большое количество слов исключается из него. Поэтому предлагается сжимать слова не из словаря с помощью какого-нибудь другого алгоритма сжатия.

Однако выбор алгоритма для сжатия таких слов не очень прост. Во-первых, алгоритм не должен использовать дополнительную память или использвать ее мало. Во-вторых, данные для сжатия довольно плохо могут быть сжаты по своей сути. Например, мы знаем, что там нет слов, которые встречаются много раз. В-третьих, мы хотим сжимать отдельные слова, а не связный текст.

Заметим, что обойтись без дополнительной памяти совсем скорее всего не получится, т. к. в отдельном сообщении максимум несколько слов, которые не попали в словарь, и найти какие-то закономерности алгоритм не успеет.

Достаточно хорошо под критерии подходит обычный алгоритм Хаффмана, который работает с символами. Действительно, как было показано ранее, он использует всего один килобайт дополнительной памяти, что очень хорошо нам подходит. Использование этого алгоритма позволило улучшить коэффициет сжатия с 2.04 до 2.38 на тестовых данных.

2.5. Двухсимвольный Хаффман

Нам хотелось еще больше соптимизировать часть сжатия, которая касается слов, которые не попали в словарь. Для этого был придуман следующий алгоритм, который основан на обычном алгоритме Хаффмана.

Большинство алгоритмов сжатия используют контекст, т. е. информацию о том, какие символы были закодированы последними. Предлагается сделать тоже самое для алгоритма Хаффмана. А именно, посчитаем p_{ij} —вероятность того, что после буквы i идет буква j. Пусть размер алфавита равен n, тогда для каждого символа i построим дерево Хаффмана с n листьями на вероятностях p_{ij}. Путь в дереве i до листа j сопоставим строке s_{ij}. Алгоритм кодирования данных показан в листинге 1, а декодирования в листинге 2.

Если при определении, какие слова нужно оставлять в словаре, суммарный размер сообщений почти не играл роли, то на коэффициент сжатия он влияет довольно сильно. Это происходит потому что дополнительная информация, необходимая для работы алгоритма, константного размера ($4 \cdot 256 \cdot 256 = 256$ кб), а значит ее относительный размер

Листинг 1 – Кодирование двухсимвольным Хаффманом

```
function Code(word, s_ij)
    prev ← 0
    res ← empty string
    for i ← 1, len(word) do
        res ← res + s_{prev,word[i]}
        prev ← word[i]
    end for
    return res
end function
```

Листинг 2 – Декодирование двухсимвольным Хаффманом

```
function Decode(encoded, s_ij, n)
    it ← 1
    res ← empty string
    prev ← 0
    while it ⩽ len(encoded) do
        for i ← 1, n do
            len ← |s_{prev,i}|
            if encoded[it..it + len − 1] = s_{prev,i} then
                it ← it + len
                res ← res + i
                prev ← i
                break
            end if
        end for
    end while
    return res
end function
```

уменьшается. На рис. 8 показано сравнение коэффициентов сжатия следующих алгоритмов в зависимости от количества сообщений:

а) Алгоритм Хаффмана по словам.

б) Алгоритм Хаффмана по словам. Слова, которые не попали в словарь, сжимаются односимвольным Хафмманом.

в) Алгоритм Хаффмана по словам. Слова, которые не попали в словарь, сжимаются двухсимвольным Хафмманом.

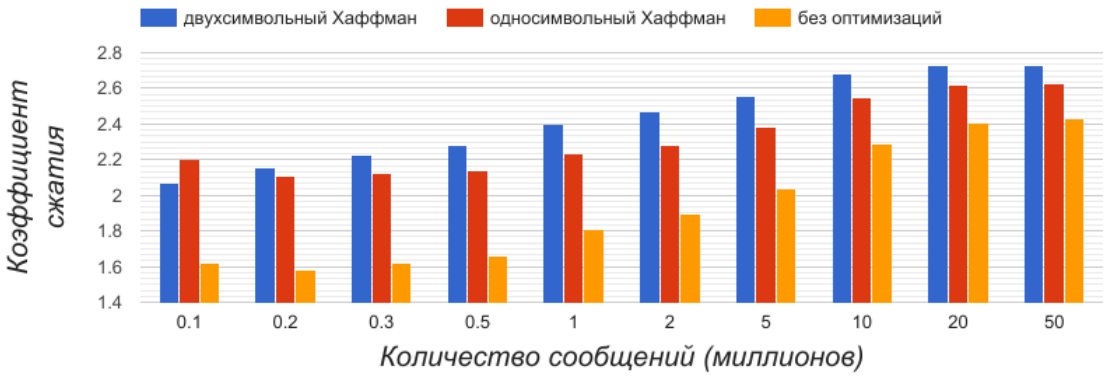

Рисунок 8 – Коэффициент сжатия в зависимости от количества сообщений

2.6. Реализация алгоритма

Теперь остановимся более подробно на реализации разработанного алгоритма. Наш алгоритм должен уметь выполнять следующие функции:

а) train(String s) — обучись на данном сообщении.
б) trainEnd() — обучение закончено. Алгоритм должен построить все необходимые ему структуры.
в) compress(String s) — сжать строку. Алгоритм должен вернуть некоторую последовательность байт.
г) decompress(bytes[] data) — расжать строку. Алгоритм должен расжать переданные ему данные.

Самое главное свойство алгоритма — для любой строки s decompress(compress(s)) = s.

Реализация функций train и compress во многом совпадает. Необходимо разбить исходную строку на составные части, а потом что-то с ними сделать. Реализация фукциии split приведена в листинге 3. Он предполагает, что в коде есть функция $isAlphabetic$, которая отвечат на вопрос о том, как делить все символы на два различных множества. В ней нужно учесть кодировку, в которой хранятся данные, а также язык, на котором посылаются сообщения. Например во «В Контакте» есть много сообщений на украинском языке и необходимо добавить букву ‘i’ в правильное множество.

Таким образом функция train должна выглдять также как и split, а внутри добавить слово в словарь, который соответсвует $needAlphabetic$.

Листинг 3 – Алгоритм разбития сообщения на слова

```
 public void split(String s) {
    boolean needAlphabetic = true;
    for (int i = 0; i < s.length(); ) {
        int j;
        while (j != s.length() && isAlphabetic(s.chatAt(j)) ==
           needAlphabetic) {
            j++;
        }
        // do smth with s[i..j]
        i = j;
        needAlphabetic = !needAlphabetic;
    }
}
```

Функция trainEnd должна выполнить два одинаковых действия с двумя словарями. А именно из него необходимо удалить слова, которые встречаются мало раз. При этом посчитаем вероятности p_{ij} того, что после символа i следует символ j в этих словах. Кроме того посчитаем суммарную вероятность слов, которые были удалены, и добавим слово-исключение с такой вероятностью.

Далее построим дерево Хаффмана для оставшихся слов. Кроме того необходимо построить n дополнительных деревьев Хаффмана, где n — размер алфавита. i-е такое дерево должно соответсвовать символу, который хотим написать после символа i. Также необходимо сделать все вероятности ненулевыми, чтобы иметь возможность записать даже слово, которое раньше не видели.

Также необходимо добавить все слова в хеш-таблицу, в которой по слову можно узнать какое битовое представление ему соответствует.

Теперь рассмотрим реализацию функции compress. Во-первых необходимо аналогично функции split разбить сообщение на слова. Каждое слово необходимо попытаться найти в хеш-таклице, которая соответсвует $needAlphabetic$. Если слово найдено, то запишем его битовую запись и перейдем к следующему слову. В противном случае необходимо написать слово-исключение, а потом закодировать слово в с помощью двухсимвольного Хафммана, который был описан в листинге 1.

Функция decompress чуть более сложна в реализации. Она представлена в листинге 4. Чтобы код не получился слишком громоздким и нечитабельным были сделаны некоторые допущения:

а) Существует класс BitStream, который переводит набор байтов в последовательность битов. Также он умеет спускаться по дереву переданному ему дереву Хаффмана, читая биты, и вернуть полученный лист дерева;

б) Можно пренебречь тем, что конкатенация строк работает в худшем случае за квадрат от суммарной длины. В реальности необходимо использвать StringBuilder;

в) decodeTwoSybolHuffman — функция описанная в листинге 2. Можно считать, что декодирование происходит до тех пор пока не встречен специальный символ окончания.

Листинг 4 – Алгоритм декодирования сообщения

```
public String decompress(bytes[] data) {
    String res = "";
    BitStream bitStream = new BitStream(data);
    boolean needAlphabetic = true;
    while (bitStream.hasMoreData()) {
        Token token = bitStram.readToken(needAlphabetic ? wordsTree
            : notWordsTree);
        if (token.isEscapeWord()) {
            res += decodeTwoSymbolHuffman(bitStream);
        } else {
            res += token.getWord();
        }
        needAlphabetic = !needAlphabetic;
    }
    return res;
}
```

2.7. Оптимизации

На практике при реализации описанного алгоритма были применены некоторые оптимизации, которые позволили увеличить скорость работы алгоритма.

Расссмотрим функцию BitStream.readToken, которая была использована в листинге 4. Ее реализация представлена в листинге 5.

Такая реализация работает очень долго. У нее есть несколько проблем.

Листинг 5 – Спуск по дереву Хаффмана

```
public Token readToken(TreeNode node) {
    while (node.left != null) {
        if (readBit()) {
            node = node.right;
        } else {
            node = node.left;
        }
    }
    return node.getToken();
}
```

Во-первых, на каждый бит сжатых данных происходит условный переход. Причем процессор не может предсказать в какую именно ветку условия нужно пойти, поскольку переходы по ним происходят примерно одинаково часто (за счет того, что это дерево Хаффмана). Это полностью ломает конвеерную обработку запросов процессором, что может ухудшить время работы в несколько раз.

Во-вторых, на каждый бит сжатых данных происходит переход в случайную ячейку памяти. Поскольку размер деревьев в нашем алгоритме может достигать миллиона элементов, такой переход не попадет ни в какой кеш процессора.

Чтобы ускорить время выполнения этой функции была реализована следующая идея. Предподсчитаем, что сделает функция readToken, в зависимости от того, какие будут результаты выполнения следующих 16 вызовов функции readBit для всех возможых 2^{16} вариантов. Существует два варианта:

а) функция прочитает не более 16 бит, дойдет до листа в дереве Хаффмана и вернет его;
б) функция прочитает все 16 бит и остановится в некоторой внутреней вершине дерева.

Также необходимо, чтобы BitStream мог наперед выдать следующие 16 бит сжатых данных. Тогда вместо вызова readToken можно посмотреть в предподсчитанную таблицу. После этого, если нужно, дочитать оставшиеся биты старым способом. Потом информировать BitStream сколько бит действительно было прочитано и вернуть нужный лист дерева.

Такой метод требует дополнительно сотни килобайт памяти, но увеличивает скорость работы в несколько раз.

Аналогично для 256 деревьев двухсимвольного Хаффмана были построены таблицы с предподсчитанными переходами на 8 бит вперед.

Кроме того важно правильно реализовать класс BitStream. В нем необходимо избегать условных переходов везде где только можно.

ГЛАВА 3. СРАВНЕНИЕ С СУЩЕСТВУЮЩИМИ РЕШЕНИЯМИ

Разработанный алгоритм сложно сравнивать с другими из-за специфики исходных данных. Средняя длина сообщений примерно равна 30 символам, а распределение по длинам показано на рис. 9.

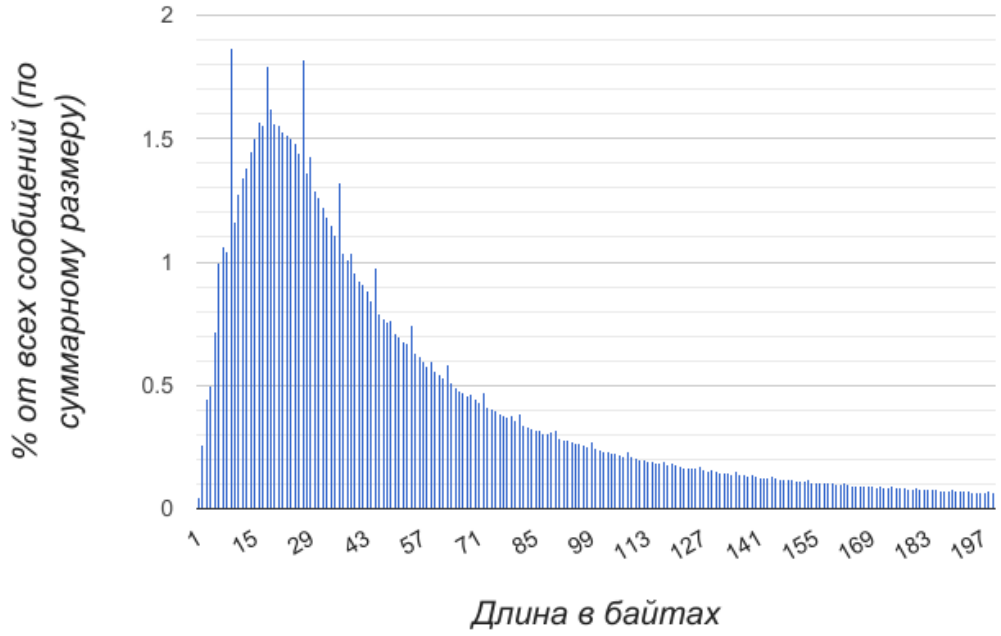

Рисунок 9 – Распределение сообщений по их длинам

Проблема заключается в том, что большинство алгоритмов сжатия расчитаны на сжатие большого объема данных. В случае же когда сообщения имеют длину порядка 30 байт, то такое сообщение, чаще всего, при сжатии стандартными архиваторами, станет еще больше. Конечно, можно сжимать сразу весь набор сообщений, но тогда сравнение будет некорректным, так как такой метод не позволяет получить произвольное сообщение быстро, а значит решение противоречит поставленной задаче.

Поэтому для сравнения можно брать известные алгоритмы и пытаться модифицировать их так, чтобы они стали применимы к исходной задаче, либо использовать алгоритмы, которые дает возможность обучиться на некотором наборе данных, а потом использовать полученный словарь, чтобы сжимать новые данные.

3.1. LZW

Алгоритм Лемпеля-Зива-Велча (LZW) был опубликован в 1984 году [10]. Идея алгоритма заключается в следующем. Создадим таблицу из слов размера 2^k для некоторого k. Изначально в нее поместим только отдельные буквы. Данная таблица будет пополняться по мере кодирования сообщений. Так при кодировании находится наибольшее слово, которое уже есть в словаре, записывается позиция этого слова в таблице, а также в таблицу добавляется слово на один большей длины. Знчение k при этом может динамически увеличиваться.

Этот алгоритм необходимо было несколько адаптировать под рассматриваемую задачу. А именно, поскольку сообщения короткие, предлагается вначале «прогнать» алгоритм через все сообщения, чтобы набрать базу слов, а потом кодировать отдельные сообщения с использованием полученной таблицы. При этом необходимо побороться с тем, что таблица может переполниться. Чтобы такого не произошло будем удалять самые старые слова для освобождения места.

При этом необходимо следить, чтобы не было ситуации, когда некоторое слово есть в таблице, а его префикса нет. Такая ситуация приводит к бесполезной трате оперативной памяти. Чтобы такого не случалось, необходимо при использовании некоторого слова, переместить все его префиксы в начале LRU (двусвязного списка для удаления старых слов).

Кроме того непонятно, какое значение k использовать. Очевидно, что для маленького количества сообщений необходимо использовать k поменьше, а для большого — больше. Чтобы определить оптимальное значение было проведено несколько экспериментов с разными k.

На рис. 10 показано сравнение разработанного алгоритма и алгоритмов LZW с различными значениями k. В качестве набора сообщений для тестов был взят тот же набор, что описывался в главе про разработку алгоритма.

Как видно из рисунка, разработанный алгоритм достаточно сильно выигрывает у LZW вне зависимости от k. Что же касается выбора размера таблицы, то видно, что на наборах с 20 и 50 миллионами сообщений выигрывает таблица размером 2^{22}, на 5 и 10 миллонах лучший размер —

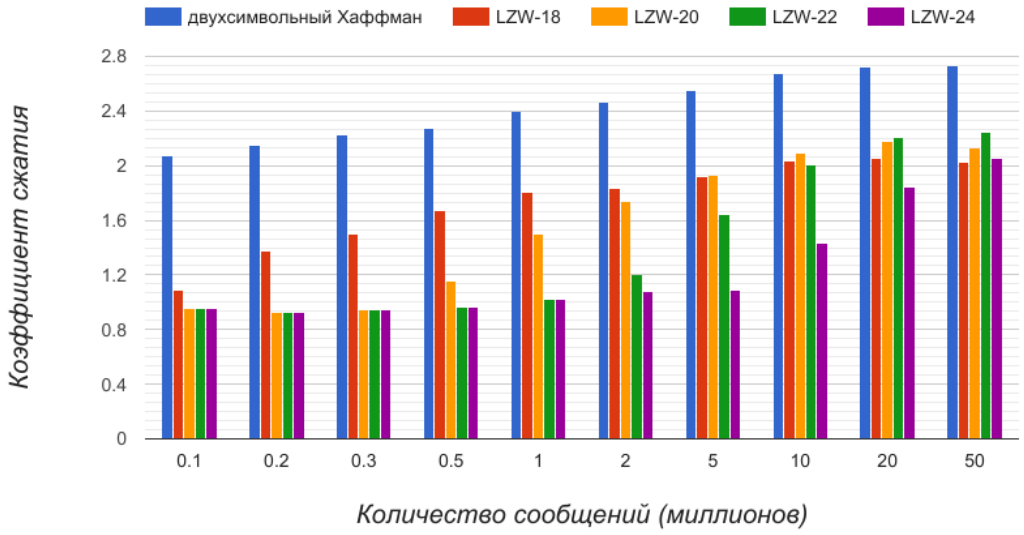

Рисунок 10 – Сравнение разработанного алгоритма с LZW

2^{20}, а на всех остальных тестах лучше всего показал себя алгоритм с маленькой таблицей.

3.2. SMAZ

SMAZ — одна из немногих библиотек, которые расчитаны на сжатие коротких текстов [11]. В частности, в описании проекта упоминается, что большинство алгоритмов не могут сжать сообщения длиной меньше 100 символов, а SMAZ может сжать слово the до одного байта.

Внутри у данного алгоритма фиксированный набор самых часто употребляемых английских слов. Поэтому, если сжимается английский текст, то он может достигнуть коэффициента сжатия порядка 2 (это все еще значительно меньше чем разработанный нами алгоритм), а на русскоязычных текстах SMAZ не может ничего сжать.

3.3. zstd

Довольно интересным является алгоритм zstd, предложенный Facebook [7] Кроме того, что он имеет хороший коэффициент сжатия и скорость, он также имеет возможность обучаться на выборке сообщений.

В описании алгоритма есть целый раздел про дополнительное обучение алгоритма, в котором сказано, что дополнительное обучение по-

могает существенно улучшить степень сжатия JSON размером порядка килобайта.

Проведенные тесты показывают, что алгоритм действительно хорошо сжимает данные размером порядка килобайта после обучения. Однако на сообщениях размером сотни байт, zstd проигрывает разработанному алгоритму.

ЗАКЛЮЧЕНИЕ

Был разработан алгоритм сжатия коротких текстовых сообщений. Он основан на алгоритме Хаффмана. Был детально рассмотрен вопрос о размере получаемого словаря и то, как кодировать слова, которые в него не попали. Разработанный метод был детально описан.

Разработанный алгоритм был применен для сжатия сообщений в ООО «В Контакте», что позволило сократить их размер на 5-7% по сравнению с предыдущей версией сжатия. Такое улучшение в масштабах «В Контакте» соответсвует экономии нескольких терабайт оперативной памяти.

Разработанный алгоритм сложно сравнивать с существующими алгоритмами сжатия, так как они не расчитаны на сжатие коротких сообщений. Все найденные решения проигрывают разработанному алгоритму по степени сжатия.

Разработанный алгоритм, а также все дополнительные материалы, доступны в [12].

Как одно из направлений развития алгоритма можно попробовать использовать арифметическое кодирование вместо алгоритма Хаффмана.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *EMC russia* Мировой объем данных увеличивается более чем в два раза каждые два года, большие объемы данных открывают новые возможности и изменяют роль ИТ: тех. отч. — 2011. — URL: https://russia.emc.com/about/news/press/2011/20110628-01.htm.

2 *Кудряшов Б.* Основы теории кодирования. — БХВ-Петербург, 2016.

3 *Mahoney M.* Large Text Compression Benchmark. — 2017. — URL: http://mattmahoney.net/dc/text.html.

4 *Salomon D.*, *Motta G.* Handbook of Data Compression. — Springer-Verlag London Limited, 2010. — ISBN 978-1-84882-903-9.

5 Run-length encoding. — URL: https://en.wikipedia.org/wiki/Run-length_encoding.

6 LZ77. — URL: https://ru.wikipedia.org/wiki/LZ77.

7 *Collet Y.*, *Turner C.* Smaller and faster data compression with Zstandard. — Авг. 2016. — URL: https://code.facebook.com/posts/1658392934479273/smaller-and-faster-data-compression-with-zstandard/.

8 Zstandard. — URL: https://en.wikipedia.org/wiki/Zstandard.

9 *Кнут Д. Э.* Искусство программирования. Том 3. Сортировка и поиск. — Москва: Вильямс, 2007. — ISBN 5-8459-0082-1.

10 *Victor S. Miller M. N. W.* Variations on a theme by Ziv and Lempel // Combinatorial algorithms on words. — 1985. — С. 131–140.

11 *Sanfilippo S.* SMAZ. — 2012. — URL: https://github.com/antirez/smaz.

12 *Минаев Б. Ю.* MessagesCompress. — 2017. — URL: https://github.com/BorysMinaiev/MessagesCompress.