РОССИЙСКАЯ АКАДЕМИЯ НАУК
СИБИРСКОЕ ОТДЕЛЕНИЕ

ИНСТИТУТ СИСТЕМ ИНФОРМАТИКИ
им. А. П. Ершова

На правах рукописи

Кадач Андрей Викторович

Эффективные алгоритмы неискажающего сжатия текстовой информации

Специальность 05.13.11 — Математическое и программное обеспечение вычислительных машин, комплексов, систем и сетей

ДИССЕРТАЦИЯ

на соискание ученой степени
кандидата физико-математических наук

Научный руководитель:

доктор физико-математических наук,
профессор **И. В. Поттосин**

Новосибирск 1997

Содержание

Часть I

Предисловие

Глава 1

ВВЕДЕНИЕ

1.1. АКТУАЛЬНОСТЬ ТЕМЫ

Наблюдаемое в течение последних 15 лет стремительное увеличение количества используемых вычислительных машин (сопровождаемое неуклонным ростом их возможностей) привело к значительному расширению как круга пользователей ЭВМ, так и сферы применения компьютеров. Одновременное развитие телекоммуникационных систем, рост пропускной способности и общего количества линий связи, появление и развитие глобальных компьютерных сетей, в настоящее время доступных сотням миллионов пользователей, вызвали быстрое увеличение[1] объемов хранимой и передаваемой в информации.

В связи с продолжающимся ростом глобальных сетей и расширением спектра предоставляемых ими услуг при экспоненциальном росте числа пользователей сетей, а также успешно выполняемыми проектами по переводу в электронно-читаемый вид[2] содержимого огромных библиотек (Королевской Библиотеки Великобритании, Библиотеки Конгресса США и др.), выставочных залов и художественных галерей, следует ожидать, что объемы хранимой и передаваемой по системам связи информации будут продолжать увеличиваться с такой же, если не большей, скоростью.

Таким образом, задача хранения и передачи текстовой, графической, звуковой и другой информации в наиболее компактном виде достаточно актуальна.

1.2. ОБЩИЕ СВЕДЕНИЯ

Перед тем как перейти к описанию целей и задач данной работы, необходимо дать краткое представление об используемых терминах, существующих методах сжатия, критериях их оценки и т. д.; математически точные определения терминов и обзор методов сжатия приводятся в главе 2.

1.2.1. КЛАССИФИКАЦИЯ МЕТОДОВ СЖАТИЯ

Методы сжатия делятся на несколько классов:

- **неискажающие** (loseless) методы сжатия гарантируют, что декодированные данные будут в точности совпадать с исходными;

[1] По оценкам автора, каждый год объем хранимой и передаваемой информации увеличивается в 2–4 раза.

[2] Информация, представленная в электронно-читаемом виде, может храниться и обрабатываться при помощи ЭВМ, в отличие от информации, содержащейся в книгах, газетах, фильмах, аналоговых звукозаписях и т. д.

- **искажающие** (lossy) методы сжатия (называемые также методами сжатия *с потерями*) могут искажать исходные данные, например, за счет удаления шумов сигнала, сужения его спектра и т. д.

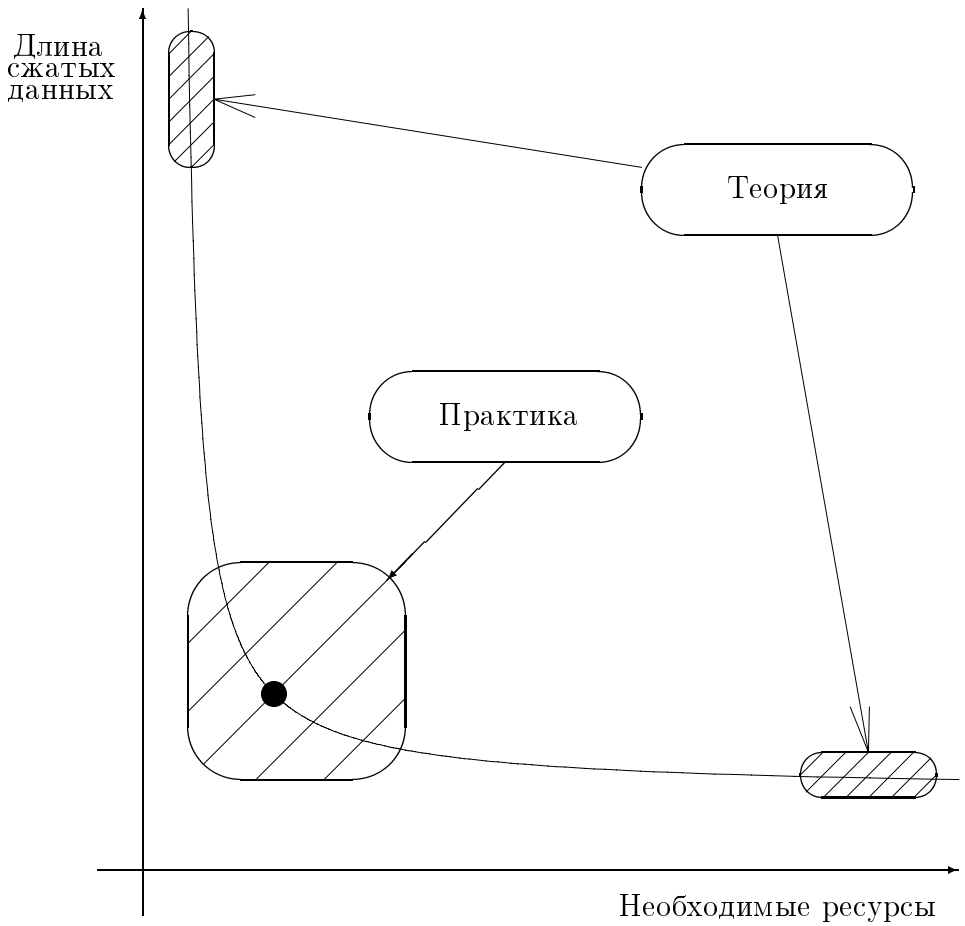

Рисунок 1.1. Соотношение качества сжатия и использованных ресурсов

Кроме того, можно выделить

- методы сжатия **общего назначения** (general-purpose), которые не зависят от физической природы входных данных и, как правило, ориентированы на сжатие текстов, исполняемых программ, объектных модулей и библиотек и т. д., т. е. данных, которые в основном и хранятся в ЭВМ;
- **специальные** (special) методы сжатия, которые ориентированы на сжатие данных известной физической природы, например, звука, изображений и т. д. и за счет знания специфических особенностей сжимаемых данных достигают существенно лучшего качества и/или скорости сжатия, чем при использовании методов общего назначения.

По определению, методы сжатия общего назначения — неискажающие; искажающими могут быть только специальные методы сжатия. Как правило, искажения допустимы только при обработке всевозможных сигналов (звука, изображения, данных с физических датчиков), когда известно, до каким образом и какой степени можно изменить данные без потери их потребительских качеств.

1.2.2. КРИТЕРИИ ОЦЕНКИ МЕТОДОВ СЖАТИЯ

Основными свойствами какого-либо алгоритма сжатия данных являются:

- **качество сжатия**, т. е. отношение длины (в битах) сжатого представления данных к длине исходного представления;

- **скорость кодирования и декодирования**;
- **объем требуемой памяти**.

В области сжатия данных, как это часто случается, действует *закон рычага* (см. рис. 1.1): алгоритмы, использующие больше ресурсов (времени и памяти), обычно достигают лучшего качества сжатия, и наоборот: менее ресурсоемкие алгоритмы по качеству сжатия, как правило, уступают более ресурсоемким.

Таким образом, построение оптимального с практической точки зрения алгоритма сжатия данных представляется достаточно нетривиальной задачей, т. к. необходимо добиться достаточно высокого качества сжатия (не обязательно оптимального с теоретической точки зрения) при небольшом объеме используемых ресурсов.

Понятно, что критерии оценки методов сжатия с практической точки зрения сильно зависят от предполагаемой области применения. Например, при использовании сжатия в системах реального времени необходимо обеспечить высокую скорость кодирования и декодирования; для встроенных систем критический параметр — объем требуемой памяти; для систем долговременного хранения данных — качество сжатия и/или скорость декодирования и т. д.

1.2.3. ПРАКТИЧЕСКИЕ ТРЕБОВАНИЯ

В критериях оценки тех или иных алгоритмов сжатия существуют некоторые различия. Если с теоретической точки зрения наибольший интерес представляет изучение крайних случаев и построение методов сжатия, оптимальных по качеству или скорости (см. [20]), то область практического интереса значительно уже и находится в окрестности точки перегиба графика зависимости длины сжатых данных от объема требуемых ресурсов (см. рис. 1.1): чрезмерно ресурсоемкие методы сжатия, равно как и обладающие невысоким качеством, с прагматической точки зрения наилучшими не являются [38, 39, 125, 166].

Чтобы ясно представлять положение и роль различных методов сжатия, необходимо четко сформулировать требования к практической реализации с учетом существующего положения и тенденций развития техники.

1.2.3.1. ОБЪЕМ ПАМЯТИ

В настоящее время наблюдается устойчивый рост объемов оперативной памяти, вызванный ее [относительной] дешевизной (2–3 доллара США за мегабайт), в связи с чем подавляющее большинство современных компьютеров (включая персональные) имеют не менее 16–32 Мб ОЗУ, а многие встроенные системы — 8–16 Мб; следует ожидать, что в ближайшие 3–5 лет объем доступной оперативной памяти удвоится. Таким образом, *значимость объема оперативной памяти*, требуемой алгоритму сжатия данных, *стремительно снижается*.

Однако это не означает, что объем требуемой памяти не важен. Поскольку одному процессу обычно разрешается использовать не более 50–75% оперативной памяти системы, алгоритмы сжатия не должны использовать более 16–20 Мб.

В ряде случаев ограничения на объем выделяемой процессу памяти очень жесткие (обычно это относится ко встроенным системам передачи данных — факсимильным аппаратам, модемам, сетевым маршрутизаторам, узловым коммуникаторам и т. д.); в таких случаях необходимо ограничиться использованием не более 1–2 Мб. Учитывая широкое распространение таких встроенных систем, разработка алгоритмов сжатия

данных, не требующих больших объемов памяти, представляет большой практический интерес.

1.2.3.2. БЫСТРОДЕЙСТВИЕ

Несмотря на постоянное повышение быстродействия ЭВМ (примерно в 1.5 раза ежегодно), *требования к скорости кодирования и декодирования остаются достаточно жесткими*, т. к. пропускная способность линий связи и объемы внешних носителей информации увеличиваются быстрее, чем растет производительность процессоров: например, с 1993 г. тактовая частота процессоров общего назначения выросла лишь в 3–5 раз (с 40–60 МГц до 150–300 Мгц), тогда как объем жестких дисков — в 50 раз (с 40–100 Мб до 2–5 Гб), а скорость передачи данных по телефонным линиям связи увеличилась почти в 10 раз — с 2400 бод (битов в секунду) в России и 9600 бод в развитых странах до 28–33 Кбод и 56–128 Кбод соответственно.

В настоящее время алгоритмы со скоростью кодирования и декодирования ниже 30 Кб/с не пригодны для практического использования, начиная с 50 Кб/с — приемлемы, а со скоростью в 100–150 Кб/с могут применяться почти всегда. Такие значения объясняются тем, что

- пропускная способность обычных телефонных линий связи (не говоря уже о специальных линиях) — 3–10 Кб/с, поэтому скорость кодирования и декодирования (с учетом коэффициента сжатия) должна быть не ниже 10–30 Кб/с;
- объем внешних накопителей информации (жестких дисков) составляет 2–5 Гб, поэтому при скорости кодирования и декодирования менее 50 Кб/с (180 Мб/ч) время создания резервной копии или восстановления данных превысит 12–24 часов, что совершенно не приемлемо.

1.2.3.3. НАДЕЖНОСТЬ ПРОГРАММ И СЛОЖНОСТЬ АЛГОРИТМОВ

Надежность программных систем и комплексов очень важна и обеспечивается как безошибочностью программирования и дизайна, так и характеристиками использованных алгоритмов.

Если количество ошибок в основном определяется полнотой и качеством тестирования (а также квалификацией и культурой программирования) и мало зависит от воли разработчика, то выбор алгоритмов — вполне управляемый и контролируемый процесс.

Для обеспечения конечного и заранее известного времени сжатия [в наихудшем случае], необходимо, чтобы алгоритм обладал *хорошо детерминированным временем работы* (желательно, мало зависящим от кодируемых данных) и *заранее известным объемом требуемой памяти*. В частности, выполнение этих требований необходимо при разработке встроенных систем, систем реального времени, файловых систем со сжатием данных и других систем с жесткими ограничениями на разделяемые различными процессами ресурсы.

Отметим, что если с теоретической точки зрения *полиномиальные* алгоритмы считаются хорошим решением проблемы (например, многие методы сжатия, описанные в [20], обладают полиномиальной или экспоненциальной сложностью), то на практике приемлемы только алгоритмы с *линейной* или *линейно-логарифмической* временной сложностью, причем крайне желательно, чтобы среднее время работы (на типичных данных) было линейным.

1.2.4. СОВРЕМЕННЫЕ МЕТОДЫ СЖАТИЯ

Без преувеличения можно сказать, что известны тысячи различных методов сжатия данных, однако многие из них заметно уступают другим по всем параметрам и поэтому не представляют интереса. Оставшиеся методы можно разбить на три больших класса.

1.2.4.1. АЛГОРИТМЫ СТАТИСТИЧЕСКОГО МОДЕЛИРОВАНИЯ

Наилучшие по качеству сжатия алгоритмы статистического моделирования [38, 39] источников Маркова семейств PPM (от англ. Prediction by Partial Matching) [39, 65, 66, 140, 191], DMC (от англ. Dynamic Markov Compression) [41, 71, 168, 169], ACB (от англ. Associative Coding by Buyanovski) [1] предсказывают вероятность появления следующего символа на основе анализа частоты появления различных последовательностей символов в ранее закодированной части сообщения.

Эти алгоритмы[3] обладают очень низкой скоростью сжатия[4] (2–20 Кб/с) и требуют большого объема оперативной памяти (8–32 Мб); скорость декодирования практически не отличается от скорости кодирования.

Несмотря на очень хорошие характеристики в смысле качества сжатия, использовать алгоритмы статистического моделирования на практике часто затруднительно или невозможно ввиду невысокой скорости сжатия.

Кроме того, многие предложенные способы реализации методов сжатия статистическим моделированием для получения оценок вероятностей появления символов используют команды умножения и/или деления [1, 39, 140, 168, 169, 191], а иногда и вычисления с плавающей точкой [41, 50, 51, 52, 53, 68]). Часто эти команды не реализованы аппаратно (например, на многих моделях RISC-процессоров, процессорах для встроенных систем и т. д.), что в таких случаях приводит к увеличению времени сжатия на порядок. Так как такие реализации предъявляет очень жесткие требования к аппаратному обеспечению, область их применения ограничена.

Может показаться, что через несколько лет в связи с постоянным ростом быстродействия ЭВМ методы статистического моделирования будут использоваться гораздо шире, однако, по мнению автора, это не так. Пропускная способность линий связи и объем внешних хранителей информации увеличивается такими же и часто опережающими темпами: например, в 1993 г. тактовая частота процессоров общего назначения (Intel 386/486, Motorola MC68040/MC68060, ARM, Sun Sparc, MIPS и др.) равнялась 40–60 МГц, объем жесткого диска — 40–100 Мб, а типичная скорость передачи данных по телефонным линиям связи составляла 2400 бод в России и 9600 бод в развитых странах; к концу 1997 г. тактовая частота процессоров увеличилась в 4–5 раз, объем жестких дисков — в 50 раз, а скорость передачи данных по телефонным линиям связи — в 10–15 раз. С учетом имеющихся тенденций следует ожидать, что требования к производительности алгоритмов сжатия будут оставаться достаточно высокими.

1.2.4.2. АЛГОРИТМЫ СЛОВАРНОГО СЖАТИЯ

Алгоритмы словарного сжатия [38, 126, 166, 201, 202] заменяют подстроки кодируемой последовательности символов ссылками в словарь на идентичные подстроки.

[3] Характеристики этих методов сжатия и других, описанных далее, приводятся в процитированных работах, [38, 39] и в главе 11; см. также регулярно проводимое с 1992 г. сравнение характеристик более 60 различных архиваторов Archiver Comparison Test [92].

[4] Значения производительности приводятся для процессора, исполняющего за секунду 100 млн. команд типа регистр-регистр.

С практической точки зрения [38, 39, 166] наилучшими представляются алгоритмы семейства LZ77 [35, 36, 38, 43, 47, 87, 155, 167, 166, 196, 201] (впервые предложенные Лемпелом и Зивом в 1997 г. [201]), которые заменяют начало непросмотренной части кодируемого сообщения ссылкой на самое длинное вхождение идентичной подстроки в уже закодированной части.

Обычно для ускорения поиска совпадающих подстрок и ограничения объема требуемой памяти область поиска ограничивается W последними символами закодированной части; такая модификация LZ77 называется *LZ77 со скользящим окном* (LZ77 with sliding window).

Алгоритмы семейства LZ77 (см. [38, 39, 65] и главу 11) в 1.3–1.7 раза уступают методам статистического моделирования по качеству сжатия, однако обладают очень высокой скоростью кодирования (50–150 Кб/с) при сравнительно небольшом объеме требуемой памяти (300–800 Кб); за счет ухудшения качества сжатия (уменьшением ширины окна) в 1.5 раза можно добиться скорости в 300–700 Кб/с.

Огромное преимущество алгоритмов семейства LZ77 — чрезвычайно высокая скорость декодирования (1–2 Мб/с при использовании вторичного сжатия по Хаффману, 10–20 Мб/с без такового). Это позволяет применять их в тех случаях, когда декодирование осуществляется гораздо чаще кодирования или скорость декодирования очень важна (например, при хранении данных на CD ROM, в файловых системах со сжатием и т. д.).

Подавляющее большинство[5] современных промышленных систем сжатия данных построено на основе различных вариантов алгоритма LZ77, в течение многих лет заслуженно считавшихся наилучшими по соотношению скорости и качества сжатия.

1.2.4.3. АЛГОРИТМЫ СЖАТИЯ СОРТИРОВКОЙ БЛОКОВ

Алгоритмы сжатия сортировкой блоков семейства BWT/BS, разработанные в 1994 г. Барроузом и Уилером [55, 185], разбивают кодируемую последовательность на блоки из $N \sim 10^6$ символов, переставляют (обратимым образом) символы каждого блока так, что появляется много повторений одного и того же символа, а затем сжимают преобразованные данные каким-либо достаточно простым способом.

По качеству сжатия (см. [55, 84, 185] и главу 11) они приближаются к методам статистического моделирования (уступая им в 1.2–1.3 раза), а по скорости (50–150 Кб/с) — к алгоритмам семейства LZ77, при меньшем по сравнению с методами статистического моделирования объеме требуемой памяти (5–10 Мб); скорость декодирования также достаточно высока (300–500 Кб/с).

Ввиду своей новизны алгоритмы сжатия сортировкой блоков мало изучены, а известные реализации страдают серьезными недостатками (см. [55, 185] и главу 8): в частности, указанная скорость сжатия (50–150 Кб/с) достигается только в среднем (на текстовых данных), а в наихудшем случае уменьшается в десятки тысяч раз, что совершенно неприемлемо при создании надежных систем.

1.2.5. МЕТОДЫ ЭНТРОПИЙНОГО КОДИРОВАНИЯ

Как правило, вышеперечисленные методы сжатия применяются не самостоятельно, а

[5] Из более 150 архиваторов, которые хранятся на ftp://ftp.elf.stuba.sk/pub/pc/pack, 9 реализуют методы статистического моделирования, около 20 — алгоритм LZW [184], 5 — сжатие сортировкой блоков, а остальные (более 75%) реализуют различные варианты алгоритма LZ77.

в сочетании с каким-либо методом *энтропийного*[6] кодирования, заменяющего символы их *кодовыми словами* — строками нулей и единиц — так, что более часто встречающимся символам соответствуют более короткие кодовые слова.

Такие методы кодирования известны с конца 40-х гг. и хорошо изучены. Их можно разбить на два больших класса: префиксные (Хаффмана, Шеннона, Шеннона — Фано) и арифметические.

1.2.5.1. ПРЕФИКСНЫЕ КОДЫ

Префиксные коды называются так потому, что ни одно кодовое слово не является полным началом (т. е. префиксом) никакого другого слова, что гарантирует однозначность декодирования.

Известно много способов построения префиксных кодов: коды Шеннона [163] и Шеннона — Фано [78] почти оптимальны, а код Хаффмана — оптимален среди префиксных кодов.

Так как длина каждого кодового слова выражается целым числом битов, то префиксные коды неэффективны на алфавитах малой мощности (2–8 символов) или при наличии символов с очень большой (более 30–50%) вероятностью появления и по качеству сжатия могут уступать арифметическим.

Применение *блочных кодов*, кодирующих не отдельные символы, а блоки из k символов, позволяет построение кодов, сколь угодно близких по качеству кодирования к арифметическим, однако из-за полиномиальной сложности блочного кодирования по размеру блока [20, 39, 166] и ряда других причин (см. п. 2.5.1.4) блочное кодирование почти никогда не применяется на практике.

Как правило, алгоритмы словарного сжатия и сжатия сортировкой блоков используют для кодирования выхода основного алгоритма сжатия коды Хаффмана (см. ftp://ftp.elf.stuba.sk/pub/pc/pack).

1.2.5.2. АРИФМЕТИЧЕСКИЕ КОДЫ

Арифметические коды не стоят явного соответствия между символами и кодовыми словами; они основаны на других принципах, рассмотрение которых выходит за рамки данной работы. Арифметическому кодированию посвящено много публикаций [4, 23, 24, 27, 46, 68, 79, 80, 81, 94, 103, 104, 122, 141, 147, 153, 156, 170, 195], т. к. его качество лучше, чем у посимвольного префиксного кодирования, и близко к теоретическому минимуму и при малой мощности алфавита, и при очень неравномерном распределении вероятностей появления символов.

С другой стороны, кодирование и декодирование арифметических кодов при достаточно большой мощности кодируемого алфавита ($|\Sigma| \geq 100$) заметно медленнее кодирования и декодирования префиксных кодов, а разница в качестве сжатия обычно незначительна и не превышает 1% [39, 166, 180, 181]; по этим и ряду других причин (см. п. 2.6.4) автор считает, что префиксное кодирование более предпочтительно для практического использования.

Арифметические коды обычно применяются в сочетании с методами статистического моделирования для кодирования символов в соответствии с предсказанными вероятностями.

[6] Следуя [39, 51, 52], вместо термина посимвольное частотное кодирование источников Бернулли [20] используется термин энтропийное кодирование ([zero-order] entropy coding).

1.3. ЦЕЛЬ РАБОТЫ

Цель данной работы заключается в разработке **высокоэффективных методов и алгоритмов неискажающего сжатия данных**, которые одновременно с высокой скоростью обладают и высоким качеством сжатия.

Особое значение придавалось достижению **очень высокой средней скорости** работы алгоритма сжатия на типичных данных (текстах на естественных языках, двоичных образах исполняемых программ, объектных файлов и их библиотек и т. д.), которая имеет наибольшее практическое значение, **при приемлемой скорости сжатия в наихудшем случае**.

Качеству сжатия также уделялось пристальное внимание. В некоторых случаях допускалось ухудшение качества на 0.5–1.5%, если это ускоряло алгоритм в 1.5–2 раза, однако одновременно с этим предлагались более эффективные методы кодирования, компенсирующие такое ухудшение качества сжатия.

1.4. ЗАДАЧИ ИССЛЕДОВАНИЯ

В данной работе рассматривались только методы сжатия, допускающие **эффективную программную реализацию на последовательных ЭВМ** (см. п. 2.1), поскольку в настоящее время параллельные компьютеры встречаются достаточно редко, а область применения аппаратных решений очень узкая.

Основные усилия были направлены на изучение, анализ и дальнейшее развитие **универсальных алгоритмов словарного сжатия семейства LZ77 и сжатия сортировкой блоков**, обладающих хорошим качеством сжатия и очень высокой скоростью кодирования и декодирования [38, 39, 55, 166, 185].

Большой практический интерес представляют **специализированные системы сжатия текстов на естественных языках**, основанные на замене слов их номерами в словаре, т. к. тексты на естественных языках (английском, русском и т. д.) составляют значительную и часто подавляющую часть данных, пересылаемых по линиям связи (письма, сообщения в группы новостей и т. д.) и хранимых на ЭВМ (тексты подсказок, документы и т. д.), а такие методы сжатия [39, 194] обладают очень хорошим качеством сжатия, высокой скоростью кодирования и декодирования и, в отличие от многих других методов, допускают почти произвольный доступ к сжатым данным, что важно для многих систем обработки текстовой информации: систем контекстных подсказок, информационно-поисковых систем, текстовых процессоров и т. д.

Особое внимание уделено **методам построения и декодирования префиксных кодов**, которые используются в сочетании со всеми вышеперечисленными методами сжатия и в ряде случаев существенно влияют на скоростные характеристики систем сжатия; например, для алгоритмов семейства LZ77 60–90% времени декодирования [39] занимает декодирование префиксных кодов. Наиболее подробно рассмотрены методы построения и декодирования кодов Хаффмана, которые оптимальны среди префиксных кодов по качеству сжатия.

Рассмотрим более конкретно основные сложности и проблемы, возникающие при реализации вышеперечисленных методов сжатия.

1.4.1. ПРЕФИКСНЫЕ КОДЫ И КОДЫ ХАФФМАНА

Так как качество кодирования однозначно определяется вероятностной структурой источника сообщений и процесс кодирования сводится к конкатенации коротких битовых

строк, его характеристики не могут быть улучшены.

Однако при большой мощности алфавита **время построения кода Хаффмана** традиционными методами может быть значительным. Его можно заметно улучшить применением других алгоритмов, учитывающих тот факт, что длина кодируемой последовательности и мощность алфавита заведомо ограничены сверху.

Декодирование префиксных кодов может быть ускорено в несколько раз за счет перехода к обработке групп битов.

1.4.2. АЛГОРИТМЫ СЖАТИЯ ТЕКСТОВ ЗАМЕНОЙ СЛОВ

Специальные методы сжатия текстов на естественных языках (русском, английском, немецком и т. д.) [38, 39, 194] обычно заменяют слова их номерами в словаре; несмотря на простоту идеи, существует множество различных вариантов этого метода [39, 143, 148, 149, 193, 194].

Применение подходов, использующих общие закономерности естественных языков [160, 199, 200], позволяет улучить качество и скорость сжатия и избежать серьезных сложностей (см. [110, 111, 123, 173, 174, 175, 176]), связанных с необходимостью кодирования символов очень большого алфавита, в качестве которых выступают слова языка и последовательности разделителей.

1.4.3. АЛГОРИТМЫ СЕМЕЙСТВА LZ77

Усилия автора были направлены на построение **высокоэффективных** алгоритмов **поиска совпадающих подстрок**, определяющих скорость кодирования, и разработку методов **выбора оптимальной последовательности замен подстрок ссылками** и **энтропийного кодирования выхода LZ77**, которыми определяется качество сжатия.

1.4.4. АЛГОРИТМЫ СЖАТИЯ СОРТИРОВКОЙ БЛОКОВ

Алгоритмы сжатия сортировкой, обладающие высоким качеством сжатия и хорошим быстродействием, но требующие достаточно много памяти, были открыты в 1994 г. [55] и поэтому мало изучены. Известные асимптотически оптимальные по скорости **алгоритмы сортировки блоков** (см. [55] и главу 8) обладают низким средним быстродействием и требуют очень много оперативной памяти (30–60 Мб), а применяемые эвристические методы [55, 185] замедляются в тысячи раз при наличии в сжимаемых данных большого количества совпадающих подстрок. Таким образом, **необходимо разработать алгоритмы сортировки, которые требуют мало памяти, обладают высокой средней производительностью и приемлемой скоростью в наихудшем случае**.

Альтернативный способ достижения высокой скорости сжатия заключается в **построении нового класса обратимых преобразований, которые заведомо могут быть реализованы очень эффективно** и также порождают в преобразованной последовательности много повторений одного и того же символа.

Другое направление исследований заключается в разработке **более эффективных** [по сравнению с известными [34, 55, 83, 84, 185 **по качеству сжатия методов кодирования результата преобразования сортировкой блоков**, обладающих высокой производительностью и не оказывающие заметного влияния на общее время сжатия.

1.5. МЕТОДЫ ИССЛЕДОВАНИЯ

В процессе исследований были использованы основные положения теории информации, теории кодирования дискретных источников сообщений, комбинаторики, дискретной математики, теории чисел, методы динамического программирования, математического анализа и теории асимптотических функций.

1.6. НАУЧНАЯ НОВИЗНА РЕЗУЛЬТАТОВ РАБОТЫ

Автором разработаны, исследованы и детально обоснованы как с теоретической, так и с практической точки зрения методы решения описанных задач и алгоритмы их реализации.

Как правило, при разработке алгоритмов существенно использовался тот факт, что длина кодируемой последовательности символов, равно как и мощность алфавита, естественным образом ограничены сверху: например, ввиду ограниченных объемов носителей информации длина кодируемой последовательности заведомо меньше $2^{64} \approx 1.6 \cdot 10^{19}$. Предложенные алгоритмы не обязательно являются асимптотически оптимальными, однако близки к таковым с практической точки зрения, обладая при этом рядом преимуществ.

1. Разработаны и исследованы новые методы построения кодов Хаффмана и декодирования префиксных кодов для последовательностей ограниченной сверху длины.
   - Показано, что предложенные методы построения кода Хаффмана быстрее известных, если длина кодируемого сообщения не очень велика (меньше 2^{32}), а мощность кодируемого алфавита достаточно большая (более 100 символов).
   - Доказано, что разработанные автором алгоритмы декодирования префиксных кодов линейны по количеству закодированных символов (время декодирования известными алгоритмами пропорционально длине закодированной последовательности в битах). Доказано, что почти все известные методы декодирования являются частными случаями предложенных.
2. Разработаны и исследованы новые методы сжатия текстов на естественных языках (русском, английском и т. д.). Доказано, что предложенные методы кодирования почти оптимальны по качеству сжатия.
3. Разработаны и исследованы новые однопроходные алгоритмы управления порядком поиска идентичных подстрок и замены их ссылками (для LZ77-схем). Доказано, что предложенные алгоритмы оптимальны по стоимости кодирования, линейны по количеству неоднозначностей кодирования и минимальны по количеству необходимых поисков подстрок, определяющих скорость сжатия. Доказано, что все используемые в настоящее время эвристические методы управления поиском подстрок и определения порядка кодирования — частные случаи алгоритмов, предложенных автором.
4. Разработана новая методика построения и реализации алгоритмов сжатия сортировкой блоков.
   - Предложены и исследованы новые алгоритмы сортировки блоков, требующие небольшого объема оперативной памяти ($2N$–$3N$ слов). Доказано, что они оптимальны по скорости; среднее время сортировки блока размера N почти линейно и равно $O(N \log\log N)$, а в наихудшем случае не превышает $O(N \log N)$; при таком же объеме требуемой памяти время других известных эффективных в среднем алгоритмов сортировки может достигать $O(N\sqrt{N} \log N)$.

- Разработан и исследован новый класс алгоритмов сжатия *частичной* сортировкой блоков. Доказано, что такие алгоритмы обладают достаточно редким и очень важным качеством: время сжатия или восстановления не зависит от исходных данных и прямо пропорционально их длине, при этом требуется $2N$ слов дополнительной памяти.
- Предложены и исследованы новые методы кодирования выхода преобразования сортировкой блоков (полной или частичной). Доказано, что они оптимальны по скорости, а по качеству сжатия лучше известных. Показано, что качество такого сжатия мало отличается и часто лучше качества наиболее эффективных статистических методов сжатия.

1.7. ПРАКТИЧЕСКАЯ ЦЕННОСТЬ РЕЗУЛЬТАТОВ

1. Разработанные методы построения и декодирования кодов Хаффмана в 3–6 раз быстрее известных алгоритмов.
2. Предложенные алгоритмы сжатия текстов на естественных языках в 5–100 раз быстрее известных при лучшем в 1.5–2 раза качестве сжатия.
3. Разработанные алгоритмы управления поиском и методы реализации поиска подстрок для LZ77-схем со скользящим окном позволили в 2–5 раз увеличить скорость[7] сжатия (до 250–400 Кб/с) и на 3–5% — качество, при таком же и часто меньшем объеме требуемой памяти (300 Кб при ширине окна 64 Кб).
4. Предложенные алгоритмы сжатия сортировкой блоков (полной или частичной) требуют 9–18 Мб оперативной памяти — столько же или меньше, чем необходимо другим известным реализациям алгоритмов сжатия сортировкой блоков или статистического моделирования, при этом:
   - разработанные алгоритмы сжатия полной сортировкой блоков по средней производительности (150 Кб/с) превосходят почти все другие известные реализации, в отличие от них обеспечивая приемлемую скорость сжатия и в наихудшем случае, при лучшем на 15–20% качестве, не уступающем качеству сжатия методов статистического моделирования;
   - предложенные алгоритмы сжатия частичной сортировкой блоков по скорости сжатия (300–400 Кб/с) вдвое опережают алгоритмы сжатия полной сортировкой блоков и часто быстрее алгоритмов семейства LZ77, при этом скорость сжатия не зависит от сжимаемых данных, а качество лишь на 1–5% хуже, чем при полной сортировке, и в 1.3–1.5 раза лучше, чем у LZ77.

Таким образом, разработанные методы сжатия являются наилучшими по всем параметрам как среди требующих очень небольшого объема памяти алгоритмов (словарного сжатия), так и среди более ресурсоемких (сжатия сортировкой блоков и статистическим моделированием).

1.8. РЕАЛИЗАЦИЯ И ВНЕДРЕНИЕ РЕЗУЛЬТАТОВ

Все алгоритмы и методы сжатия, описанные в данной работе, реализованы автором в универсальной библиотеке сжатия данных PRS (около 15 000 строк на языке Си),

[7] Значения скорости сжатия приводятся для процессора Intel Pentium с тактовой частотой 100 МГц. Поскольку все алгоритмы реализованы автором на языке Си (с целью обеспечения переносимости), аккуратной реализацией на языке Ассемблера можно увеличить производительность на 15–20%.

которая перенесена на почти все существующие платформы и операционные системы (Intel 80x86 под MSDOS, Windows 95, Windows NT, OS/2, Linux, SCO, FreeBSD; Hewlett Packard под HPUX; SunSparc под SunOS и Solaris; Silicon Graphics под Irix; DEC Alpha под OSF-1 и Windows NT; IBM PowerPC под AIX и Windows NT).

Библиотека PRS применяется рядом коммерческих компаний (ProPro Group Inc., XDS Ltd. и др.) для создания дистрибутивных версий распространяемых продуктов.

В 1994–1995 г. система сжатия текстов на естественных языках была использована компанией ProPro Group Inc. для хранения текстов подсказок и сопровождающей документации (на русском, английском, немецком, французском и итальянском языках), что позволило сократить объем системы почти на 20%.

В 1992–1993 гг. автором была разработана система PRSEXE сжатия исполняемых файлов для MS DOS, распространяемая компанией AIP (Netherlands) в составе пакета UC2 под названием UCEXE. Согласно регулярно проводимым с 1992 г. сравнительным исследованиям систем сжатия данных ACT [92], этот продукт до сих пор остается одним из лучших и неизменно занимает лидирующее положение по взвешенным показателям.

1.9. АПРОБАЦИЯ РАБОТЫ И ПУБЛИКАЦИИ

Результаты работы были опубликованы автором в местной и центральной печати [6—15] (все публикации — без соавторов).

Работа была апробирована на заседаниях ученого совета Института систем информатики им. А. П. Ершова СО РАН, семинарах кафедры программирования механико-математического факультета Новосибирского государственного университета и семинарах Новосибирского института связи (СибГАТИ).

Результаты работы используются в курсе лекций, читаемых автором для студентов магистратуры кафедр программирования ММФ НГУ и АФТИ ФФ НГУ.

1.10. СТРУКТУРА РАБОТЫ

Диссертационная работа состоит из пяти частей, каждая из которых состоит из одной или нескольких глав.

Первая часть (вводная) состоит из данной главы и главы 2, где определяются используемые термины и аббревиатуры, описываются известных коды, алгоритмы и методы сжатия, которые неоднократно используются или упоминаются в дальнейшем. Предполагается, что изложенного в первой главе материала достаточно для понимания содержимого остальных частей работы (при условии, что читатель обладает достаточной математической подготовкой и знаком со структурами данных и алгоритмами, широко применяемыми в программировании).

Вторая часть работы[8] **(глава 3)** посвящена префиксным кодам. В ней описываются эффективные способы порождения и декодирования префиксных кодов; особое внимание уделено оптимальному по качеству сжатия коду Хаффмана.

Третья часть работы[9], состоящая из четырех глав, описывает методы создания эффективных по скорости, качеству сжатия и объему требуемой памяти словарных алгоритмов сжатия вообще и алгоритмов семейства LZ77 в частности. **Глава 4** посвящена эффективным методы сжатия текстов на естественных (русском, английском и т. д.)

[8] Результаты второй части опубликованы в [6, 7, 13, 14].

[9] Результаты третьей части опубликованы в [8, 9, 10, 15].

языках, допускающим произвольный доступ к сжатым данным, что необходимо при создании систем контекстных подсказок, организации электронных библиотек и т. д. **В главе 5** описываются эффективные методы выбора оптимальной последовательности кодирования для LZ77-схем. **Глава 6** посвящена эффективным методам реализации поиска совпадающих подстрок в LZ77-схемах, скоростью которых в основном и определяется производительность алгоритмов сжатия семейства LZ77. **В главе 7** изложены близкие к оптимальным эффективные методы кодирования выхода LZ77, применение которых позволяет заметно улучшить качество сжатия.

Четвертая часть работы[10], состоящая из трех глав, посвящена разработке методов сжатия сортировкой блоков. **В главе 8** описывается разработанный автором высокоэффективный алгоритм блочной сортировки, являющийся центральной частью алгоритма сжатия и определяющий скорость работы и объем требуемой памяти. **В главе 9** предлагается алгоритм сжатия частичной (а не полной) сортировкой блоков, производительность которого существенно превышает производительность подавляющего числа других известных методов сжатия при незначительно худшем по сравнению с полной сортировкой качестве сжатия. **В главе 10** описываются эффективные методы реализации и повышения качества сжатия выхода алгоритмов блочной сортировки.

Пятая, заключительная, часть работы состоит из двух глав. **В главе 11** приводятся результаты сжатия тестовых данных авторскими программами и созданными другими разработчиками архиваторами, реализующими все наиболее эффективные методы сжатия. **В главе 12** суммируются полученные автором результаты.

Работа заканчивается списком цитируемой литературы, содержащем около двухсот ссылок.

[10] Результаты четвертой части опубликованы в [11, 12].

Глава 2

ОСНОВНЫЕ ТЕРМИНЫ И КОНЦЕПЦИИ

В данной главе даются основные определения используемых терминов и обозначений, описываются концепции и идеи основных алгоритмов; описанные здесь методы, подходы и введенная терминология будут использоваться в дальнейшем.

Автор старался быть предельно аккуратным при описаниях алгоритмов и их идей и всегда указывать ссылки на первоисточники, даже если они не опубликованы официально (в некоторых случаях единственными документами, подтверждающими приоритет, являются WWW-странички, тексты программ или сообщения в группы новостей Internet comp.compression или comp.compression.research; такие документы могут быть найдены в соответствующих электронных депозитариях).

2.1. МОДЕЛЬ ВЫЧИСЛЕНИЙ

В качестве модели вычислительной машины принимается последовательная машина фон Неймана с памятью ограниченного объема с произвольным доступом и косвенной адресацией, представлением целых чисел в двоично-дополнительном коде и обработкой данных блоками по w битов за единицу времени[1] (unit-cost RAM machine [135]); в качестве примера такой вычислительной машины может служить машина MIX, описанная Кнутом [16].

С одной стороны, подробное формальное определение такой машины занимает несколько десятков страниц текста [16, 135] и поэтому не приводится в данной работе; с другой стороны, почти все современные архитектуры ЭВМ достаточно точно соответствуют этой спецификации, поэтому достаточно описать основные особенности таких RAM-машин.

Память машины состоит ограниченного количества w-битовых *слов*, возможно, составленных из b-битовых *байтов*; каждое слово может хранить в двоично-дополнительном коде целое число или в интервале $[-2^{w-1}, 2^{w-1}-1]$ (знаковое целое), или в интервале $[0, 2^w-1]$ (беззнаковое); интерпретация содержимого слова зависит от использованной команды. Объем памяти не превышает $C2^w$ слов, где C — небольшая константа, а все обрабатываемые данные находятся в памяти.

Такая машина, используя в качестве операндов w-битовые слова, способна выполнить над ними за одну единицу времени (такт) одну из следующих арифметических операций: сложения, вычитания, знакового и беззнакового сравнения, побитового И, ИЛИ, исключающего ИЛИ, логического и/или арифметического и/или циклического побитового сдвига. Кроме того, машина реализует команды условных и безусловных переходов, вызова подпрограмм, загрузки в память w-битовых константных

[1] Поскольку русскоязычное определение очень длинно, в дальнейшем будет использоваться англоязычный эквивалент.

значений и извлечения из памяти значений слов по указанному адресу — целому беззанковому числу, — также исполняемые за один такт.

При анализе алгоритмов будут оцениваться *объем памяти* (выраженный в количестве слов и/или байтов), требуемой алгоритму, а также *время* работы работы алгоритма, выраженное в количестве команд RAM-машины, подлежащих исполнению.

RAM-машины бывают безадресными, одно–, двух–, трехадресными, регистровыми, аккомуляторными или стековыми (магазинными) [135]; эти различия непринципиальны с точки зрения временной оценки, поскольку количество исполняемых команд отличается не более чем на постоянный множитель. Унарные операции (смена знака, взятие абсолютного значения, побитового дополнения и др.) и ряд бинарных операций (например, исключающее ИЛИ) реализуются с помощью фиксированного количества бинарных операций, поэтому наличие или отсутствие унарных операций среди реализуемых RAM-машиной не существенно.

RAM-машины с таким набором команд относятся к классу AC_0 [135]: схемотехническая сложность их реализации (выраженная в количестве требуемых однобитовых элементов И, ИЛИ, НЕ) полиномиальна по w.

Если же RAM-машина реализует еще и команды умножения и/или деления, то она относится к классу AC_1, поскольку схемотехническая сложность реализации умножения и деления экспоненциальна по w. Такое отличие AC_0 и AC_1 RAM-машин существенно не только с теоретической, но и с практической точки зрения: в подавляющем большинстве случаев время умножения на процессорах общего назначения в 10–30 раз больше времени исполнения команд класса AC_0, а время деления больше в 40–70 раз; более того, часто команды деления и умножения не реализованы аппаратно (например, на младших моделях RISC-процессоров, процессорах для встроенных систем и т. д.).

В последние годы неоднократно отмечалось [29, 32, 151], что многие ранние работы по анализу алгоритмов (например, [16]) для RAM-машин подразумевали (не указывая явно), что ширина слова $w = O(\log n)$, где n — длина обрабатываемых данных (в словах или байтах), поскольку предполагалось, что индексирование массивов, разыменование указателей и другая адресная арифметика выполняется за фиксированное время. Это условие не выполняется, если w фиксировано, а $n \to \infty$: запись индекса или адреса требует $\log_2 n \to \infty$ битов. Итак, необходимо, чтобы или $w = O(\log n)$, или чтобы RAM-машина была способна выполнять за фиксированное время арифметические операции над целыми числами неограниченной длины; доказано [117], что в последнем случае некоторые нижние временные оценки не верны.

Следуя [29, 32, 151], в данной работе будет предполагаться, что все обрабатываемые данные находятся в оперативной памяти unit-cost RAM-машины с шириной слова w, а их длина n ограничена сверху: $n = O(2^w)$.

2.2. ИДЕАЛЬНЫЙ АЛГОРИТМ СЖАТИЯ

Необходимо сказать, что *не существует идеального неискажающего* алгоритма сжатия, т. е. такого, который сокращает длину *любой* последовательности символов некоторого (например, двоичного) алфавита, гарантируя при этом возможность однозначного декодирования. Любой неискажающий алгоритм сжатия, который способен уменьшать длину некоторых сообщений, обязан *увеличивать* длину других.

А.Н. Колмогоров [19] ввел понятие *сложности* кодирования, ныне носящей его имя: сложность сообщения есть длина программы (с точностью до аддитивной константы) для универсальной машины Тьюринга, результатом работы которой является закоди-

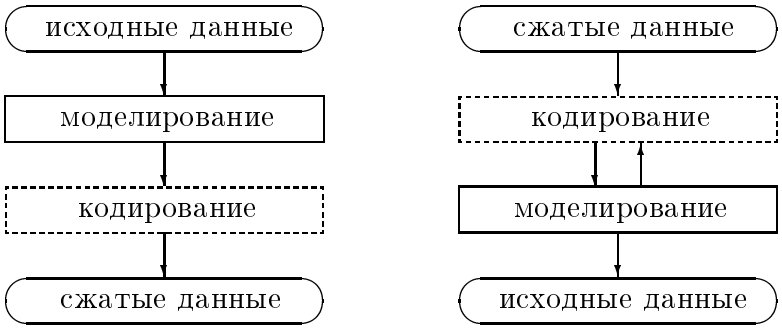

Рисунок 2.1. Моделирование и кодирование

рованное сообщение. К сожалению, колмогоровская сложность не вычислима и представляет чисто теоретический интерес. Более практичный подход — использование в качестве меры объема информации, несомой сообщением, понятия *энтропии*, рассмотренного далее.

2.3. МОДЕЛИРОВАНИЕ И КОДИРОВАНИЕ

В 1981 г. Риссаненом и Лангдоном [153] была предложена интересная концепция разделения процесса сжатия на две независимые стадии: *моделирование* и *кодирование*. На этапе *моделирования* с входными данными производятся некоторые преобразования, уменьшающие их размер или преобразующие данные к более пригодному для сжатия виду. После этого производится *кодирование*, заменяющее более часто используемые данные более короткими кодами (см. рис. 2.1).

Например, если сжимаемые данные представляют собой запись уровней сигнала синусоидальной формы с некоторым временным шагом, то они практически не поддаются сжатию. Однако если заметить, что разность уровней сигнала в соседних точках отличается мало, то можно перейти от кодирования уровней сигнала к кодированию *разностей* уровней и тем самым существенно улучшить качество кодирования. Такой переход и называется *моделированием*, т. к. фактически строится некоторая *модель* поведения входных данных, *предсказывающая* их поведение на основе уже закодированных данных или за счет некоторых заранее известных закономерностей.

Модель не обязана описывать кодируемый процесс, она должна лишь предсказывать поведение данных, и чем точнее предсказания, тем лучше сжатие. Единственное ограничение, налагаемое на модель, заключается в том, что при кодировании можно пользоваться только той информацией, которая будет доступна при декодировании; никаких других ограничений не существует. Более того, модель может изменяться по мере кодирования, приспосабливаясь к кодируемым данным; такие модели называются *адаптивными*.

Вообще говоря, моделирование и кодирование — две достаточно независимых и самостоятельных области теории информации. Как правило, качество сжатия в основном определяется использованной моделью, однако за счет хорошего (малоизбыточного) кодирования результирующее качество сжатия может быть существенно улучшено, поэтому обе эти области представляют большой практический интерес; более того, применение некоторых методов моделирования без эффективного кодирования их результата бесмысленно (яркий пример — описанное выше разностное кодирование).

Следуя [39, 51, 52], в дальнейшем во избежание возможных разночтений стадия кодирования результатов моделирования будет называться **энтропийным кодированием**, в то время как термин **кодирование** будет относиться к только к стадии моделирования или процессу сжатия в целом.

2.4. КОНЕЧНЫЕ ВЕРОЯТНОСТНЫЕ ИСТОЧНИКИ

Допустим, что существует некоторый *конечный*[2] алфавит Σ и некоторый источник S, вырабатывающий последовательности символов (строки) алфавита Σ, называемые *сообщениями*.

Определение 2.1. Источник S называется *бернуллиевским*, если вероятность появления символа α в сообщении в точности равна $p_S(\alpha)$ и не зависит от предыдущих символов. ■

Определение 2.2. Если вероятность появления символа α, порождаемого источником S, зависит только от одной предыдущей буквы, то такой источник называется *источником Маркова первого порядка*. ■

Источник Маркова первого порядка S определяется матрицей $\mathcal{P}_S = p_{ij}$ размерности $|\Sigma| \times |\Sigma|$, где p_{ij} — вероятность появления символа α_j вслед за α_i. Вероятность того, что S породит строку $\alpha_{i_1} \dots \alpha_{i_n}$, равна $q_{i_1} p_{i_1 i_2} \dots p_{i_{n-1} i_n}$, где q такой вектор, что $\mathcal{P}_S q = q$.

Аналогичным образом определяются источники Маркова порядка k, для которых вероятность появления символа зависит от k предыдущих.

В данной работе опущены вопросы, которые приводятся в классических работах по теории информации [20, 163, 162], связанные с построение меры на множестве Σ^* всех строк конечной длины алфавита Σ, введением понятия количества информации, несомой сообщением и др., поскольку они весьма нетривиальны, и их детальное изложение выходит за рамки данной работы.

Определение 2.3. *Кодом* называется отображение[3] $f : \Sigma^* \to \{0,1\}^*$, где A^* — множество всех строк *конечной* длины алфавита A. ■

Определение 2.4. В дальнейшем для краткости алфавит $\{0,1\}$ будем называть *двоичным* или *бинарным*, а символы этого алфавита — *битами*. ■

Определение 2.5. *Кодовым словом* строки $s \in \Sigma^*$ называется соответствующая ей последовательность нулей и единиц $f(s)$, а *длиной кодового слова* $|f(s)|$ — длина строки $f(s)$. ■

Определение 2.6. *Стоимостью кодирования* $\mathcal{C}_S(f)$ кода f на источнике S называется средняя длина кодового слова

$$\mathcal{C}_S(f) = \sum_{s \in \Sigma^*} p_S(s) |f(s)|. \tag{2.1}$$

■

Определение 2.7. *Битовая строка* $a \in \{0,1\}^*$ называется *однозначно декодируемой*, если существует не более одной строки $s \in \Sigma^*$ такой, что $f(s) = a$; *код* f называется *однозначно декодируемым*, если любая битовая строка $a \in \{0,1\}^*$ однозначно декодируема. ■

Задача *однозначно декодируемого*, или *неискажающего*, кодирования дискретных вероятностных источников, впервые рассмотренная Шенноном [163], является одной из центральных в теории информации и заключается в отыскании по источнику S

[2] В ряде работ [20, 163, 162] рассматривались и бесконечные алфавиты, однако в данной работе будут рассматриваться только конечные алфавиты.

[3] Для простоты изложения мы ограничимся изучением только двоичных кодов; все определения, описанные алгоритмы и полученные результаты легко обобщаются на случай недвоичных кодов.

однозначно декодируемого кода f, средняя длина кодового слова которого минимальна или близка к минимальной.

Определение 2.8. *Энтропией источника Бернулли* S (или энтропией нулевого порядка) называется

$$\mathcal{E}(S) = \sum_{\alpha \in \Sigma} p_S(\alpha) \log_2 \frac{1}{p_S(\alpha)}. \tag{2.2}$$

■

Определение 2.9. *Энтропией источника Маркова первого порядка* S называется

$$\mathcal{E}_1(S) = \sum_{i,j=1}^{|\Sigma|} p_{ij} \log_2 \frac{1}{p_{ij}}; \tag{2.3}$$

энтропия источников Маркова порядка k определяется аналогичным образом. ■

Шенноном было показано, что для источников Маркова[4] средняя длина кодового слова однозначно декодируемого кода не может быть меньше *энтропии* источника.

Определение 2.10. *Избыточностью* кода f для источника S называется разность между стоимостью кодирования и энтропией:

$$\mathcal{R}_S(f) = \mathcal{C}_S(f) - \mathcal{E}_S. \tag{2.4}$$

■

Известно, что код f однозначно декодируем тогда и только тогда, когда для него выполняется неравенство Крафта — Макмиллиана[5] [121, 128]:

$$\sum_{s \in \Sigma^*} 2^{-|f(s)|} \leq 1. \tag{2.5}$$

Определение 2.11. *Коды* f_1 и f_2 называются *эквивалентными*, если длины всех кодовых слов совпадают: $|f_1(s)| = |f_2(s)| \ \forall s \in \Sigma^*$. ■

Важность источников Маркова объясняется тем [20, 39, 166], что многие типичные данные, подлежащие сжатию, — например, тексты на естественных языках, — хорошо описываются марковскими моделями. Однако, как отмечалось в [39, 166], в ряде случаев построение точное построение и описание источников невозможно — как, например, для текстов на естественных языках: не существует источника [39], порождающего все когда-либо написанные книги, сказанные речи, а также все книги, которые будут написаны, речи, которые будут сказаны и т. д. Поскольку в для описания таких источников используются приближенные модели, построенные для них методы кодирования не всегда являются действительно наилучшими. Ситуация осложняется тем, что в ряде случаев асимптотическая оптимальность какого-либо метода сжатия не означает, что такой метод будет наилучшим при сжатии реальных данных, поскольку их длина всегда конечна, а малые по сравнению со стремящейся к бесконечности длиной кодируемых данных слагаемые могут оказаться значительными по сравнению с длиной реальных данных (см. [39, 166]).

[4] Шенноном рассматривался гораздо более общий случай, нежели описанный в данной работе; для наших целей достаточно рассмотрения источников Маркова.

[5] В 1949 г. это соотношение было доказано Крафтом для префиксных кодов, а в 1956 г. Макмиллиан показал, что оно выполняется для любых однозначно декодируемых кодов

2.5. КОДИРОВАНИЕ ИСТОЧНИКОВ БЕРНУЛЛИ С ИЗВЕСТНОЙ ВЕРОЯТНОСТНОЙ СТРУКТУРОЙ

2.5.1. ПРЕФИКСНЫЕ КОДЫ

Для источника Бернулли вероятность появления строки есть произведение вероятностей появления составляющих ее символов, поэтому кодирование строки можно заменить последовательным кодированием составляющих ее символов; такое кодирование называется *посимвольным*.

Если иного не оговорено явно, в дальнейшем под энтропией будет подразумеваться энтропия нулевого порядка, т. е. источника Бернулли.

В дальнейшем предполагается, что f — посимвольный код для *фиксированного* источника Бернулли S, т. е. известны вероятности $p_S(\alpha)$ появления символов конечного алфавита $\alpha \in \Sigma$; код строки $s = \alpha_1 \dots \alpha_N$ есть *конкатенация* (слева направо) кодов составляющих s символов:

$$f(\alpha_1 \dots \alpha_N) = f(\alpha_1) \dots f(\alpha_N). \tag{2.6}$$

Поскольку источник S фиксированный, в дальнейшем он не будет указываться явно.

Будем считать, что Σ — конечный алфавит. Так как все символы конечного алфавита могут быть занумерованы, то можно не проводить различия между символом алфавита и его номером; кроме того, такая нумерация одновременно вводит на Σ и Σ^* полный порядок: символ α_1 *лексикографически меньше* α_2, если номер α_1 меньше номера α_2; аналогично, строка $s_1 \in \Sigma^*$ *лексикографически меньше* $s_2 \in \Sigma^*$, если первые n символов s_1 и s_2 совпадают и длина s_1 равна n, а длина s_2 больше n или $(n+1)$-й символ s_1 меньше $(n+1)$-го символа s_2. Введенный таким образом полный порядок на символах алфавита называется *естественным*, а полный порядок на множестве строк — *естественным лексикографическим* порядком.

Определение 2.12. *Префиксом* строки $b = b_1 \dots b_k$ называется такая строка $a = a_1 \dots a_n$ $(n \le k)$, что $a_1 = b_1, \dots, a_n = b_n$. ■

Определение 2.13. *Код* называется *префиксным*, если ни одно кодовое слово не является префиксом другого. ■

Свойство префиксности очень важно по нескольким причинам. Во-первых, из однозначной декодируемости кода отнюдь не следует, что декодирование будет простым: код $\langle 0, 01, 011 \rangle$ однозначно декодируем, но при декодировании, например, строки 0110 требуется заглядывать вперед, чтобы выяснить, что последовательность 0 не является кодом символа 1, а последовательность 01 — кодом символа 2, т. к. за ними следует единица, и т. д. Префиксные же коды свободны от этого недостатка, т. к., начиная с некоторого момента, декодируемый символ определяется однозначно, причем заглядывание вперед не требуется никогда.

Во-вторых, Крафтом и Макмиллианом было показано, что для каждого однозначно декодируемого кода существует эквивалентный ему префиксный код (например, префиксный код $\langle 0, 10, 110 \rangle$ эквивалентен вышеприведенному однозначно декодируемому коду).

Крафт указал также на еще одно важное свойство префиксных кодов: для каждого префиксного кода существует бинарное дерево, листья которого помечены символами алфавита Σ, а ребра — нулями и единицами, так что конкатенация — слева направо —

пометок дуг пути от корня к листу и есть кодовое слово символа, помечающего данный лист дерева.

Определение 2.14. *Помеченное дерево*, соответствующее коду, называется *кодовым деревом.* ■

В дальнейшем без ограничения общности будем полагать, что *левые* ребра кодового дерева помечены *нулями*, а *правые — единицами.* С учетом этого между множеством кодовых деревьев и префиксных кодов можно установить взаимно-однозначное соответствие и в дальнейшем не проводить различия между префиксным кодом и соответствующим ему кодовым деревом.

Определение 2.15. *Префиксный код* называется *полным*, если для любой битовой строки s, являющейся префиксом некоторого кодового слова $f(\alpha)$, отличного от s, битовые строки $s0$ и $s1$ (полученные конкатенацией s со строками 0 и 1 соответственно) — также префиксы некоторых кодовых слов. ■

Префиксный код является полным тогда и только тогда, когда неравенство Крафта — Макмиллиана превращается в равенство, а префиксное дерево, соответствующее полному коду, также является полным, т. е. каждая промежуточная вершина дерева имеет в точности две выходящих из нее дуги.

Крафтом и Макмиллианом было доказано, что для любого префиксного кода f_1 существует *неудлиняющий* полный префиксный код f_2, т. е. такой, что $|f_2(\alpha)| \leq |f_1(\alpha)|\ \forall \alpha \in \Sigma$.

Шварц, Каллик [161] и Коннелл [67] ввели понятие *канонического кода.*

Определение 2.16. *Код* называется *каноническим* или *последовательным* [166], если он является полным и префиксным и, кроме того, удовлетворяет условиям:

- если кодовое слово $f(\alpha_1)$ короче кодового слова $f(\alpha_2)$, то $f(\alpha_1)$ лексикографически меньше $f(\alpha_2)$;
- если длины кодовых слов $f(\alpha_1)$ и $f(\alpha_2)$ равны, то $f(\alpha_1)$ лексикографически меньше $f(\alpha_2)$ тогда и только тогда, когда α_1 меньше α_2.

■

Например, кодовое дерево, изображенное на рис. 3.2 (стр. 63), — каноническое, в отличие от дерева на рис. 3.3.

Определение 2.17. *Значением v битовой строки $a = a_1 \ldots a_n$* называется неотрицательное целое число, запись которого в системе счисления по основанию два совпадает с a:

$$v = a_1 2^{n-1} + a_2 2^{n-2} + \ldots + a_n 2^0. \tag{2.7}$$

■

Замечание 2.1. Последовательность длин кодовых слов $\ell(\alpha) = |f(\alpha)|$ однозначно задает канонический код. Действительно, отсортировав в порядке возрастания последовательность длин кодовых слов устойчивым образом (т. е. с сохранением порядка символов при равной длине кодовых слов), получим перестановку символов $\alpha_1 \ldots \alpha_\Sigma$ алфавита Σ такую, что $\ell(\alpha_i) < \ell(\alpha_{i+1})$ или $\ell(\alpha_i) = \ell(\alpha_{i+1})$ и $\alpha_i < \alpha_{i+1}$; тогда $c(\alpha_0) = 0$ и

$$c(\alpha_{i+1}) = (c(\alpha_i) + 1)\, 2^{\ell(\alpha_{i+1}) - \ell(\alpha_i)}, \tag{2.8}$$

где битовые строки рассматриваются как запись значения в двоичной счисления. ■

В дальнейшем, если иного явно не оговорено, под *кодом* будет подразумеваться именно *полный префиксный* (но не обязательно канонический) код, а под *деревом* — соответствующее коду *полное префиксное* дерево.

2.5.1.1. КОД ШЕННОНА

В 1949 г. Шенноном [163] был предложен простой метод построения префиксного кода: длина кодового слова определяется как $\ell(\alpha) = \lceil -\log_2 p_S(\alpha) \rceil$, а кодовые слова могут быть получены последовательно, как описано в замечании 2.1; поскольку длины кодовых слов удовлетворяют неравенству Крафта — Макмиллиана, такой код будет однозначно декодируемым, а по выбору кодовых слов — префиксным. Код Шеннона не оптимален; его избыточность не превышает одного бита на символ: $\mathcal{R}_{Sh} \leq 1$.

Шенноном был также предложен другой способ построения кодовых слов в случае, если вероятности появления символов не возрастают ($p(\alpha_1) \geq p(\alpha_2) \geq \ldots \geq p(\alpha_{|\Sigma|})$):

$$f(\alpha) = \left\lfloor q(\alpha) 2^{\ell(\alpha)} \right\rfloor, \tag{2.9}$$

где

$$q(\alpha) = \left(\sum_{\beta \leq \alpha} p(\beta) \right) - p(\alpha). \tag{2.10}$$

2.5.1.2. КОД ШЕННОНА—ФАНО

В 1949 г. независимо друг от друга Фано [78] и Шенноном [163] был предложен следующий способ построения *полного* префиксного кода: сначала множество всех символов, отсортированных в порядке увеличения вероятностей появления, разбивается на два подмножества так, что сумма вероятностей символов первого подмножества примерно равна сумме вероятностей символов во втором подмножестве, каждое из подмножеств аналогичным образом снова разбивается еще на два подмножества и т. д. до тех пор, пока существуют подмножества, содержащие более одного символа; длина кодового слова символа есть глубина вложенности подмножества, состоящего из данного символа. Каждому множеству можно сопоставить вершину двоичного дерева так, что подмножества, составляющие множество, соответствуют сыновним вершинам, а если же множество состоит из символа, то этот символ помечает листовую вершину дерева; пометив дуги дерева нулями и единицами, получим полное кодовое дерево. Таким образом, построенный код — полный и префиксный; он также не оптимален, а его избыточность может достигать единицы: $\mathcal{R}_{SF} \leq 1$.

2.5.1.3. КОД ХАФФМАНА

В 1952 г. Хаффман [105] предложил способ построения *оптимального* префиксного кода[6] с избыточностью $\mathcal{R}_H < 1$. Сначала создается $N = |\Sigma|$ деревьев, состоящих из одной вершины, помеченных символами алфавита, и каждому дереву приписывается *вес*, равный вероятности появления соответствующего символа. До тех пор, пока не останется единственное дерево, два дерева с наименьшими весами сливаются в новое дерево с весом корня, равным сумме весов образующих поддеревьев. Единственное оставшееся дерево и является кодовым деревом Хаффмана; несмотря на неоднозначность построения, утверждается, что все коды Хаффмана, построенные для одного и

[6] Метод построения недвоичных кодов Хаффмана приводится в [18].

того же распределения символов, оптимальны (однако они могут быть не эквивалентными друг другу).

2.5.1.4. БЛОЧНЫЕ КОДЫ

Избыточность префиксных кодов может быть уменьшена применением *блочных кодов*, рассматривающих последовательность из n символов алфавита Σ как один символ другого алфавита $\Sigma' = \Sigma^n$ и кодирующих сообщение блоками по n символов одним из вышеперечисленных кодов; по построению, избыточность блочного кода не больше $\frac{1}{n} \to 0$ при $n \to \infty$.

2.5.2. АРИФМЕТИЧЕСКИЕ КОДЫ

Поскольку префиксные коды ставят в соответствие каждому символу битовую строку, длина каждого кодового слова — целое число, поэтому в некоторых случаях префиксные коды оказываются неэффективными (например, при небольшом размере алфавита или очень неравномерном распределении вероятностей).

В конце 50-х гг. Гильбертом и Муром был предложен алфавитный код с избыточностью $\mathcal{R}_{HM} < \frac{2}{N}$, где N — количество закодированных символов, а в начале 60-х гг. в книге Абрамсона [27] была приведена идея *арифметического кодирования* (в дальнейшем — АК), авторство которой приписывается Элиасу, с избыточностью $\mathcal{R}_{AC} < \frac{c}{N}$, где N — количество закодированных символов, а c — некоторая константа. Таким образом, избыточность стремится к нулю при $N \to \infty$, поэтому АК оптимально в смысле стоимости кодирования. К сожалению, АК не может быть реализовано на практике в классическом виде [27, 166], т. к. это потребует использования арифметики бесконечной точности и неограниченных объемов памяти; обычно используются различные модификации, использующие приближенные вычисления.

Арифметическому кодированию посвящено большое количество публикаций [4, 23, 24, 27, 46, 68, 79, 80, 81, 94, 103, 104, 122, 141, 147, 153, 156, 170, 195] и др.; учитывая достаточно высокую сложность построения АК и большое количество различных вариантов АК, в данной работе они не описываются.

2.6. КОДИРОВАНИЕ ИСТОЧНИКОВ БЕРНУЛЛИ С НЕИЗВЕСТНОЙ ВЕРОЯТНОСТНОЙ СТРУКТУРОЙ

Если частоты появления символов, порождаемых источником Бернулли, заранее не известны, то применяется *универсальное* [20] кодирование, называемое также *адаптивным* [38, 39, 166, 189] — в отличие от *статического* кодирования источников с известной вероятностной структурой; в данной работе будут использоваться термины *адаптивное* и *статическое* кодирование.

2.6.1. КОДИРОВАНИЕ СТАТИЧЕСКИМИ ПРЕФИКСНЫМИ КОДАМИ

Замечание 2.2. Так как максимальная длина кодового слова $\ell_{\max}$ (для полных кодов) заведомо меньше мощности алфавита $|\Sigma|$ (как правило, гораздо меньше), то одним из наиболее эффективных по стоимости кодирования способов передать декодировщику информацию об использованном *каноническом* префиксном коде является запись

последовательности длин кодовых слов. С учетом того, что при $\ell_{\max} \ll |\Sigma|$ среднеквадратичное уклонение длин кодовых слов от среднего значения мало, длины кодовых слов в свою очередь можно очень эффективно кодировать при помощи оптимальных префиксных кодов. ■

С учетом предыдущего замечания статическое кодирование может быть организовано следующим образом: предварительно просмотрев кодируемое сообщение и точно подсчитав вероятности появления символов, построить полный канонический префиксный код (например, канонический код Хаффмана) и, сообщив декодировщику длины кодовых слов, закодировать сообщение статическим кодом. Поскольку длина кодового слова меньше мощности алфавита, для ее записи нужно не более $\log_2 |\Sigma|$ битов, поэтому длина L'_{SH} сообщения, закодированного таким образом, удовлетворяет неравенству

$$L_{SH} < L'_{SS} < L_{SH} + |\Sigma| \log_2 |\Sigma|, \tag{2.11}$$

т. е. избыточность такого кодирования не превосходит

$$\mathcal{R}'_{SH} < 1 + \frac{|\Sigma| \log_2 |\Sigma|}{N}, \tag{2.12}$$

где N — длина кодируемого сообщения.

Избыточность блочного кодирования блоками по n символов не превышает

$$\mathcal{R}'_{BH} < \frac{1}{n} + \frac{n|\Sigma|^n \log_2 |\Sigma|}{N}; \tag{2.13}$$

очевидно, при

$$N < 0.5(n|\Sigma|^n - |\Sigma|) \log_2 |\Sigma|, \tag{2.14}$$

то посимвольного кодирование лучше блочного, а поскольку на практике $|\Sigma|$ достаточно велико (обычно в качестве алфавита используется расширенный ASCII [39, 166], состоящий из 256 символов), уже при небольших $n \geq 4$ необходимо, чтобы длина кодируемой последовательности была очень большой (256 Гб), что на практике бывает исключительно редко; при $|\Sigma| \geq 256$ и $N \sim 10^6 \div 10^8$ применение блочных кодов не оправданно.

2.6.2. АДАПТИВНЫЙ КОД ХАФФМАНА

При адаптивным кодировании по Хаффману [70, 120, 180, 181] сначала строится дерево Хаффмана с $|\Sigma|$ листьями помеченными символами алфавита Σ единичного веса.

Получив очередной символ, подлежащий кодированию, передается кодовое слово, описывающее путь из корня дерева до листа, помеченного кодируемым символом, вес листа увеличивается на единицу, и, если дерево перестало быть деревом Хаффмана, оно перестраивается.

Избыточность кодирования существенно зависит от алгоритма модификации дерева (см. [180, 181]). Для алгоритма, описанного в [70, 120],

$$\frac{1 - |\Sigma|}{N} < \mathcal{R}_{AH1} < \mathcal{E} + 1 + 2\frac{1 - 2|\Sigma|}{N}, \tag{2.15}$$

где $\mathcal{E}$ — энтропия кодируемого источника Бернулли, а для более сложного алгоритма, описанного в [180, 181],

$$\frac{1 - |\Sigma|}{N} < \mathcal{R}_{AH1} < 1 + \frac{1 - 2|\Sigma|}{N} \tag{2.16}$$

(предполагая, что кодируемое сообщение содержит все символы алфавита Σ).

Поскольку при записи или чтении каждого бита требуются достаточно нетривиальные действия для поддержания структуры дерева Хаффмана, адаптивный код Хафф-

мана при достаточно большой мощности кодируемого алфавита $|\Sigma| \geq 100$ на порядок медленнее статического кода Хаффмана [39] и поэтому почти никогда не используется.

2.6.3. СТАТИЧЕСКОЕ АРИФМЕТИЧЕСКОЕ КОДИРОВАНИЕ

При статическом АК сначала подсчитываются частоты появлений символов, сообщаемые декодировщику, а затем осуществляется кодирование, избыточность которого удовлетворяет [103, 104]

$$\frac{2}{N} < \mathcal{R}_{SA} < \frac{2 + |\Sigma| \log_2 N}{N}. \tag{2.17}$$

2.6.4. АДАПТИВНОЕ АРИФМЕТИЧЕСКОЕ КОДИРОВАНИЕ

При адаптивном АК сначала всем символам присваивается вес $w(\alpha) = \varepsilon$; очередной символ кодируется при помощи оценки вероятности его появления, равной

$$p'(\alpha) = \frac{w(\alpha)}{\sum w(\beta)}, \tag{2.18}$$

после чего $w(\alpha)$ увеличивается на единицу. Избыточность такого кодирования при $\varepsilon \to 0$ удовлетворяет [103, 104]

$$\frac{|\Sigma|}{N}(\mathcal{E} - \log_2 |\Sigma| - 1) < \mathcal{R}_{AA} < \frac{|\Sigma|}{N}(\mathcal{E} - \log_2 |\Sigma| + \log_2 N). \tag{2.19}$$

Поскольку статическое кодирование префиксными кодами сводится к конкатенации битовых строк, которую необходимо производить и при АК (помимо других весьма нетривиальных действий), при достаточно большой длине кодируемого сообщения АК медленнее кодирования по Хаффману (даже с учетом времени, требуемого для построения кодового дерева Хаффмана).

Кроме того, все известные автору АК (часть из которых описана в работах, процитированных в п. 2.5.2) при декодировании символов нуждаются в поиске такого символа, что

$$Q(\alpha) = \left(\sum_{\beta \leq \alpha} p(\beta) \right) - p(\alpha) \tag{2.20}$$

принадлежит заданному диапазону; даже без учета других действий, сложность декодирования ввиду необходимости поиска [при изменяющихся оценках вероятностей появления символов $p(\alpha)$] равна $O(\log_2 |\Sigma|)$, тогда как сложность декодирования символа при префиксном кодировании может быть сделана сколь угодно близкой к $O(1)$ и обычно требует одно-двух обращений к таблице декодирования (см. п. 3.5), поэтому при достаточно большой мощности алфавита $|\Sigma| \geq 100$ декодирование адаптивных АК порядок медленнее, чем декодирование префиксных кодов.

Следует отметить, что многие варианты АК и способы их реализации запатентованы (см. п. 2.11), поэтому при выборе реализуемого варианта АК, предназначенного для коммерческого применения, необходима консультация юриста.

По мнению автора и ряда других специалистов (см. п. 2.11), применение АК не всегда оправданно, поскольку при достаточно большой мощности алфавита $|\Sigma| \geq 100$ качество АК обычно ненамного – на 1–2% [39, 103] — лучше, чем при префиксном кодировании, однако скорость АК заметно ниже.

2.6.4.1. ПРЕФИКСНЫЕ КОДЫ ОГРАНИЧЕННОЙ ДЛИНЫ

Насмотря на то, что максимальная длина кодового слова, как правило, невелика (см. п. 3.1), в некоторых случаях желательно или необходимо (см. замечание 3.9) принудительно ограничить максимальную длину кодового слова.

Долгое время эта проблема была открытой, т. к. время работы известных алгоритмов решения этой задачи составляло $O(N^3)$ или $O(N^2)$ (см. [18]), пока в 1990 г. Лармор и Хиршберг [123] не предложили алгоритм построения префиксного кода с ограниченной длиной кодового слова за $O(LN)$ операций, где L — максимальная длина кодового слова. Тюрпин, Моффат, и Катаяйнен [110, 111, 173, 174, 175, 176] улучшили эти методы и показали, что задача построения префиксного кода минимальной избыточности с длиной кодового слова, не превышающей L, может быть решена за $O(rL)$ операций с использованием $O(rL)$ слов памяти, где r — количество групп символов с одинаковыми вероятностями; как правило, $r \ll N$.

2.6.5. КОДИРОВАНИЕ НАТУРАЛЬНЫХ ЧИСЕЛ

На практике нередко случается так, что необходимо закодировать натуральное (т. е. положительное целое) число n, причем диапазон возможных значений n слишком велик или заранее не известен, так что энтропийные методы кодирования малоэффективны или неприменимы.

2.6.5.1. РАВНОМЕРНОЕ КОДИРОВАНИЕ

Если известен диапазон $[1, N]$, которому принадлежит число n, то можно использовать *равномерное* кодирование, ставящее в соответствие числу n битовую строку $\phi(n)$ длины $k = \lceil \log_2 N \rceil$, являющуюся записью k младших значащих цифр числа $(n-1)$ в двоичной системе счисления,

$$\phi(n) = c_{k-1} \dots c_0, \tag{2.21}$$

так что

$$n - 1 = 2^{k-1} c_{k-1} + \dots + 2^0 c_0. \tag{2.22}$$

2.6.5.2. ШАГОВЫЕ КОДЫ

В ряде случаев качество кодирования можно улучшить использованием *шаговых* (step) кодов [39, 85, 166] (например, если частоты появления n не убывают или не возрастают по n). Выбрав монотонно возрастающую последовательность целых чисел $0 = n_0 < n_1 < n_2 \dots < n_k = N$, число $n \in [1, N]$, кодируется парой $(j, n - n_j)$, где j такое, что $n_j < n \le n_{j+1}$. Первый элемент пары кодируется с использованием энтропийного или равномерного кодирования, а второй элемента пары — равномерным образом $\lceil \log_2(n_{j+1} - n_j) \rceil$ битами.

С целью повышения качества кодирования рекомендуется выбирать n_j так, чтобы разность соседних элементов последовательности $(n_{j+1} - n_j)$ была целой степенью двойки.

2.6.5.3. ГРУППОВОЕ КОДИРОВАНИЕ

Если вероятности $p(n)$ появления числа n монотонно убывают, достаточно эффективно *групповое* кодирование, при котором каждое число n представляется парой

$(L(n), R(n))$, где $L(n) = \lfloor \log_2(n) \rfloor$ и $R(n) = n - 2^{L(n)}$; для кодирования первого элемента пары используется энтропийное кодирование, а для второго — равномерное (т. к. $R(n) \in [0, 2^{L(n)} - 1]$, то для передачи $R(n)$ достаточно $L(n)$ битов). В главе 4 автором показано, что групповое кодирование очень мало — менее 0.01% — отличается от оптимального, если распределение кодируемых чисел подчиняется очень часто встречающимся на практике закону Ципфа [199, 200] ($p(n) \approx \frac{C}{n}$) или более общему закону Шварца [160] с параметром θ ($p(n) \approx \frac{C}{n^{1+\theta}}$), $|\theta| < 1$).

Групповой код есть частный случай шагового кода при $n_0 = 0, n_j = 2^{j-1}$ при $j > 0$.

Впервые код, аналогичный вышеописанный, был описан в 1966 г. Голомбом [93] с тем отличием, что вместо энтропийного кодирования первый элемент $L(n)$ Голомб предложил записывать последовательностью $L(n)$ единиц, заканчивающейся нулем, так что длина кода числа n в точности равна $(2L(n) + 1)$.

2.6.5.4. БЕСКОНЕЧНЫЕ КОДЫ

Если верхняя граница возможных значений кодируемых чисел заранее не известна, то используются *бесконечные* коды.

Вышеописанный бесконечный код Голомба не оптимален в том смысле, что отношение длины кодового слова $|\phi(n)|$ и минимальной длины кодового слова $\log_2(n)$ при $n \to \infty$ стремится к двум, а не к единице. Впервые оптимальные или почти оптимальные бесконечные коды были предложены В.И. Левенштейном в 1968 г. [21] и в 1975 г. переоткрыты Элиасом. Ряд других оптимальных или почти оптимальных кодов разработали Ивен и Родэ [77], Райс [152], Фенвик [85] и др.

2.6.6. ДИСКРЕТНЫЕ РАСПРЕДЕЛЕНИЯ ВЕРОЯТНОСТЕЙ

Определение 2.18. Вектор $< p_S(1), \ldots, p_S(N) >$ вероятностей появления символов называется *дискретным*, если вероятность появления любого символа является рациональной дробью, $p_S(\alpha) \in \mathcal{Q}$. ■

В подавляющем большинстве случаев статистика источника заранее не известна и выясняется во время кодирования подсчетом частоты появления кодируемых символов, которые представляются *целыми* числами, поэтому вероятность появления символа α или ее оценка вычисляются как

$$p_S(\alpha) = \frac{w(\alpha)}{W}, \tag{2.23}$$

где $W = \sum w(\alpha)$ — общее количество просмотренных символов, $w(\alpha), W \in \mathcal{N}$.

Переход от использования произвольных действительных чисел к целым — очень важный шаг, т. к. позволяет существенно упростить и ускорить кодирование — не только потому, что операции с целыми числами быстрее операций с действительными, но и ввиду того, что целые числа представимы в виде строки бит, к которым применимы алгоритмы *цифровой* сортировки и поиска [18, 28, 29, 30, 32, 73, 90, 95, 117, 133, 151, 178], гораздо более эффективные, чем алгоритмы общего назначения (см. п. 3.4).

2.6.7. ЛОКАЛЬНО-АДАПТИВНОЕ КОДИРОВАНИЕ

Когда статистика источника очень быстро меняется, использование традиционные методы энтропийного кодирования не эффективны. В таких случаях используются

локально-адаптивные методы кодирования [20, 55, 39, 166]: самоорганизующиеся (self-adjusting и splay) деревья [45, 108, 165] или различные варианты интервального кодирования: собственно интервальное кодирование [76], метод стопки книг [25], модифицированное интервальное кодирование [34] и другие [98].

2.6.7.1. ИНТЕРВАЛЬНОЕ КОДИРОВАНИЕ

При интервальном кодировании [76] символ заменяется расстоянием до предыдущего своего вхождения в кодируемое сообщение (например, строка *(cba)aaabbbccca* будет преобразована в *(???)0004008006*); декодировщику достаточно заменять каждое декодированное число i на символ, декодированный i шагов назад.

Замечание 2.3. Можно заметить, что интервальное кодирование (и другие его варианты, описанные далее) не уменьшает длину кодируемой строки, а изменяет ее статистику, так что точнее было бы называть такие методы *преобразованиями*.

Авторы локально-адаптивных методов кодирования предлагали конкретные методы кодирования результатов преобразования, которые не описываются в данной работе (как и во многих других [39, 55, 84, 166]), поскольку обычно (см. [39, 55, 84, 166] и главу 10) используются отличные от оригинальных методы кодирования. Тем не менее, мы будем — следуя [39, 166] — сохранять авторское название, имея в виду только первый этап кодирования, описанного первооткрывателями. ■

Интервальные методы кодирования всегда используются в сочетании с энтропийными методами кодирования.

2.6.7.2. МЕТОД СТОПКИ КНИГ (MTF)

Интервальное кодирование может быть достаточно высокоизбыточным; для однозначного декодирования достаточно заменять символ количеством *различных* символов, расположенных между текущим вхождением символа и предыдущим. Это равносильно тому, что интервальному кодированию при условии, что после кодирования символа его предыдущее вхождение удаляется из кодируемой строки. Такая модификация была названа ее автором Б.Я. Рябко *методом стопки книг* [26], напоминая, как доставаемая из стопки книга складывается на вершину стопки, так что начиная с некоторого момента часто используемые книги оказываются сверху. Рябко [25, 26] приводит верхнюю оценку средней стоимости такого кодирования для источников Бернулли, равную $(\mathcal{E} + 1 + 2\log_2(1+N))$, а Элиас [76], переоткрывший этот метод и назвавший его Move-to-Front (MTF), — равную $(\mathcal{E} + \log\log N + O(1))$, где N — мощность алфавита, а $\mathcal{E}$ — энтропия источника (следует отметить, что оценка Рябко существенно точнее и лучше, оценка же Элиаса исключительно мало информативна). Таким образом, метод стопки книг преобразует строку *(cba)aaabbbccca* в *(???)0001002002*.

Нужно сказать, что алгоритм, функционально эквивалентный методу стопки книг, давно известен в программировании[7] под названием LRU (Last Recently Used) и широко используется для ускорения поиска данных. Хорспул и Кормак [98] показали, что эвристики LFU (Last Frequently Used), Move-One-Up и Move-Half-Up, которые перемещают элементы в последовательном массиве, заменяя символ индексом его текущего

[7] Кнут в своей монографии [18], написанной в 1968–1971 гг., при описании LRU на стр. 476 замечает: Простая схема, происхождение которой неизвестно, используется уже многие годы; на той же странице можно найти очень точный анализ поведения этого метода.

положения в массиве, также могут использоваться для локально-адаптивного кодирования и обладают примерно таким же качеством сжатия; однако, по собственному опыту автора, метод стопки книг — за чрезвычайно редким исключением — лучше этих эвристик.

2.6.7.3. INVERTED FREQUENCY CODING (IF)

Другая модификация интервального кодирования[8] [34] заключается в том, что символ заменяется количеством *лексикографически больших* символов, разделяющих текущее и предыдущее вхождение символа, так что строка *(cba)aaabbbccca* будет преобразована в *(c:0006) (b:000) (a:000)*. При таком подходе следует сначала закодировать вхождения последнего (лексикографически наибольшего) символа алфавита, затем предпоследнего и т. д.

2.6.7.4. КОДИРОВАНИЕ ДЛИН СЕРИЙ (RLE)

Ввиду того, что применение локально-адаптивных интервальных методов кодирования приводит к появлению длинных последовательностей повторений одного и того же символа, эти методы, как правило, применяются в сочетании с кодированием длин серий [93] (Run Length Encoding — RLE), которое заключается в замене последовательности

$$\underbrace{\alpha \ldots \alpha}_{N} \tag{2.24}$$

парой (α, N) (следуя [39, 166], конкретный метод кодирования N, предложенный в [93], не приводится; как правило — см. [55, 39, 166] и главу 10 — на практике применяются другие методы кодирования).

2.7. МЕТОДЫ СТАТИСТИЧЕСКОГО МОДЕЛИРОВАНИЯ

Методы статистического моделирования обычно основаны на том, что входные данные рассматриваются как посылаемые некоторым источником Маркова порядка k, и модель строится с учетом этих соображений.

Известные асимптотически оптимальные методы кодирования источников Маркова [20] обладают полиномиальной временной сложностью $t > O(n^2)$, а их избыточность убывает как $O\left(\sqrt{\frac{\log t}{t}}\right)$, где n — длина кодируемого сообщения, поэтому такие методы кодирования не применяются на практике [39].

2.7.1. АЛГОРИТМЫ СЕМЕЙСТВА PPM

Алгоритмы семейства PPM (от англ. Prediction by Partial Matching), получившие свое название от статьи [66], где они были впервые описаны, — достаточно прямолинейный адаптивный способ кодирования модели Маркова порядка k.

[8] В оригинальной работе этот вариант интервального кодирования назван Inverted Frequency Coding. По мнению автора, это название крайне неудачно по двум причинам: во-первых, оно ни о чем не говорит, а во-вторых, его перевод на русский язык (кодирование обратными частотами) еще менее информативен. В дальнейшем для обозначения этого метода будет использоваться аббревиатура IF.

Получив очередной символ, подлежащий кодированию, по предыдущим k символам находится описание статистики этого *контекста*, дающей оценки вероятностей появления следующих за этим контекстом символов. Ввиду большого количества контекстов (обычно несколько миллионов) используется адаптивное арифметическое кодирование. Если i-й кодируемый символ уже появлялся в данном контексте, то он кодируется согласно оценке вероятности своего появления

$$p_i(\alpha) = \frac{w_i(\alpha)}{W_i}, \tag{2.25}$$

где $w_i(\alpha)$ — количество появлений символа α, а

$$W_i = \sum_{\alpha \in \Sigma} w_i(\alpha). \tag{2.26}$$

Если же такого символа не было, то возникает проблема *нулевой вероятности*, т. к. символ с нулевой вероятностью не может быть закодирован. Обычно эта проблема решается присваиванием всем возможным символам некоторой ненулевой вероятности появления:

$$p_i(\alpha) = \frac{w_i + \varepsilon}{W_i + \varepsilon|\Sigma|}. \tag{2.27}$$

Другое решение основано на использовании особого символа алфавита, называемого символом убегания (escape symbol): если кодируемый символ отсутствует в контексте k-го порядка, посылается escape, отсутствующий символ вводится в текущий контекст и производится переход к контексту $(k-1)$-го порядка, затем — при необходимости — $(k-2)$-го и т. д.

Существует много различных вариантов основного алгоритма. Например, совершенно неочевидно, что вес символа необходимо увеличивать на единицу; более того, различные версии алгоритма PPM — PPMA, PPMB [66], PPMC [140], и PPMD [97] только этим и отличаются. Последний алгоритм считается наилучшим; он увеличивает веса всех символов, кроме escape, на два, а вес escape — на один.

Поскольку все символы, присутствующие в контексте k-го порядка, обязательно присутствуют в контекстая меньших порядков, при кодировании символа escape (т. е. отсутствии кодируемого символа в текущем контексте) можно временно исключать из текущего контекста символы, присутствующие в контексте большего порядка. По мнению автора, это может существенно улучшить качество сжатия (подобные идеи высказывались ранее, но не были реализованы на практике ввиду очевидных трудностей; было бы интересно узнать, каких результатов можно добиться при таком подходе).

Кроме того, при отсутствии кодируемого символа в текущем контексте совершенно необязательно переходить к контексту именно предыдущего порядка. Если порядок текущего контекста достаточно большой, то очень вероятно, что кодируемый символ отсутствует и в предыдущем контексте; возможно, в некоторых случаях имеет смысл переходить сразу же к контекстам существенно меньших порядков (эта идея также не нова и почти не изучена).

Недавно авторы PPM предложили его новый вариант [64, 65], названный PPM*, в котором они предложили использовать контексты *неограниченной* длины при условии, что такой контекст *уникален*; ими было замечено, что длинные уникальные контексты, как правило, однозначно определяют последующие символы. Хранение неуникальных контекстов, конечно, не ухудшает качество сжатия, однако в большинстве случаев объем требуемой памяти для PPM, например, 9-го порядка составляет сотни мегабайтов, в то время как PPM* 4-го порядка при использовании 10–30 Мб позволяет добиться примерно такого же качества сжатия.

Наконец, нужно сказать, что рядом авторов были предложены упрощенные варианты алгоритмов PPM [97, 101, 102, 124, 125], обладающие худшим качеством сжатия, но требующие меньше памяти или в 1.5–2 раза более быстрые.

2.7.2. АЛГОРИТМЫ СЕМЕЙСТВА DMC

Алгоритмы семейства DMC (от англ. Dynamic Markov Compression), названные по статье [71], также предназначены для моделирования источников Маркова, но основаны на несколько иных принципах. Имея некоторый конечный детерминированный автомат, кодирование производится путем перехода к следующему состоянию по дуге, помеченной кодируемым символом, согласно текущему весу, приписанному дуге. В некоторых случаях производится дублирование (так называемое *клонирование*) некоторых вершин со всеми их дугами, так что в новую вершину можно прийти только по уникальному пути; таким образом автомат адаптируется к кодируемому сообщению и автоматически учитывает его марковскую природу.

Белл [41] показал, что DMC осуществляет кодирование источника Маркова конечного порядка; порядок источника определяется условиями, при которых осуществляется клонирование вершин, а также длиной кодируемого сообщения.

В оригинальном варианте [71] кодируются символы бинарного алфавита и из каждой вершины выходит ровно две дуги, поэтому клонирование не вызывает проблем, однако приводит к достаточно большому объему требуемой памяти (обычно необходимо 10–20 Мб).

Теухола и Райта [168, 169] предложили вариант DMC, кодирующий символы произвольного алфавита. Так как при этом количество выходящих из вершины дуг может быть достаточно большим, то операция клонирования усложняется. Кроме того, с целью ограничения объема требуемой памяти имеет смысл иметь дуги только для тех символов, которые кодировались в данном состоянии. Это приводит к появлению выделенного символа escape, возвращающего кодировщик к начальному состоянию; новые дуги вводятся с помощью escape аналогично тому, как это делается в алгоритмах семейства PPM. Такой подход позволил за счет использования в качестве алфавита расширенного ASCII (по 8 битов на символ) уменьшить объем требуемой памяти, ускорить кодирование и — что довольно удивительно — несколько улучшить качество сжатия.

Объем памяти и время сжатия алгоритмов семейства DMC в 2–4 раза меньше, чем у PPM, а качество сжатия хуже на 10–15% (см. [39, 71, 168, 169] и главу 11).

2.7.3. АЛГОРИТМ ACB

В 1994 г. Г. Буяновский [1] предложил метод сжатия данных, названный им ACB (Associative Coding by Buyanovsky); им же был создан одноименный архиватор.

Статья [1] является самым невразумительным набором слов, который автору данной работы когда-либо удавалось встречать. Она состоит из трех страниц текста, описывающего алгоритм сжатия с помощью незнакомых автору терминов (воронка аналогий, стохастическая составляющая строки, вытяжка, жесткий информационный канал и т. д.) и заканчивается столь же малопонятным 7-страничным листингом программы, реализующей описанный алгоритм на смеси Си и Ассемблера с использованием жуткой смеси плохого русского и ломаного английского языков. К своему сожалению, автор признает, что ему не удалось понять принципов работы алгоритма ACB.

Как бы то ни было, алгоритм ACB действительно способен уменьшать объем кодируемых данных, причем качество сжатия сравнимо и часто лучше, чем у лучших вариантов алгоритмов сжатия семейства PPM. Скорость кодирования и декодирования алгоритма ACB чрезвычайно низкая (это один из самых медленных методов сжатия) при значительном объеме требуемой памяти.

Поскольку на практике алгоритм ACB является одним из лучших известных методов по качеству сжатия (см. главу 11 и [92]), автор счел необходимым упомянуть его.

2.7.4. МОДЕЛИРОВАНИЕ СТОХАСТИЧЕСКИХ ПРОЦЕССОВ И НЕЙРОННЫХ СЕТЕЙ

В последнее время появился ряд новых методов моделирования: алгоритм блочной сортировки (см. п. 2.9), использование обучаемых нейронных сетей для моделирования [127, 159] и применение стохастических процессов [49, 50, 51, 52, 53] для сжатия информации.

По мнению автора, использование нейронных сетей для сжатия данных бесперспективно (во всяком случае, в настоящее время). Во-первых, качество сжатия обычно хуже, чем у алгоритмов семейства PPM, при существенно большем объеме требуемой памяти; лишь при очень долгом (100–200 итераций) обучении качество сжатия начинает сравниваться с PPM. Во-вторых, что более важно, авторы [127, 159] игнорируют необходимость передачи состояния нейронной сети, установившегося в результате обучения; объем этой информации многократно превышает кодируемое сообщение. Таким образом, закодированное сообщение оказывается жестко привязанным к кодировщику.

Очень интересные результаты были получены Сьюзан Бантон [49, 50, 51, 52, 53], предложившей ряд методов улучшения и развития алгоритмов семейств PPM и DMC (скорее, это нужно называть именно так) на основе некоторых нетривиальных принципов несомненно заслуживают самого пристального изучения, т. к. качество сжатия алгоритмов С. Бантон лучше любых других вариантов PPM и DMC; правда, о скорости сжатия С. Бантон скромно умолчала, поэтому можно предположить, что скорость такого сжатия очень низкая.

2.8. СЛОВАРНЫЕ МЕТОДЫ СЖАТИЯ

Словарные методы сжатия (называемые также методами Лемпела — Зива), сокращенно LZ) впервые были описаны в конце 70-х гг. Лемпелом и Зивом [126, 201, 202], и с тех пор появилось большое количество их модификаций (LZ77 [201], LZ78 [202], LZR [155], LZSS [35, 167], LZB [36], LZH [47], LZBW [43], LZY [198], LZWZ [197], LZY2 [91], LZWW [196], LZW [184], LZT [172], LZJ [107] и др.). Великолепный обзор методов сжатия был написан Беллом, Клиари и Уиттеном [38]; т. к. объем обзора составляет 34 полных страницы текста мелким шрифтом, то в целях экономии места эти методы не будут рассматриваться во всех подробностях; мы ограничимся изучением только наиболее эффективных и перспективных вариантов.

В 1988 г. Сторер в своей монографии [166] (практически целиком посвященной алгоритмам семейства Лемпела — Зива) дал изящное описание обобщенного алгоритма словарного сжатия, которое и приводится ниже.

Допустим, что существует строка S над алфавитом Σ и некоторый словарь D строк алфавита Σ, содержащий все строки единичной длины. Просматривая строку слева направо, кодировщик находит некоторую строку словаря длины n, являющуюся началом

непросмотренной части кодируемой строки S, и заменяет эту подстроку ссылкой на соответствующую строку словаря, после чего сдвигается по кодируемому сообщению на n символов и, возможно, модифицирует словарь (так, чтобы словарь по-прежнему содержал все строки единичной длины).

Если декодировщик синхронизован с кодировщиком, т. е. они имеют одинаковые начальные словари и модифицируют их одинаковым образом, процесс декодирования сводится к последовательности замен (слева направо) ссылок в словарь соответствующими строками.

Понятно, что существует огромное количество различных способов организации словаря, методов его модификации, способов выбора подходящих строк в словаре, алгоритмов поиска подстрок в словаре и методов кодирования ссылок в словарь.

Беллом [42] было показано, что для алгоритмов словарного сжатия с простым методом выбора подходящей строки в словаре (самой длинной или первой подходящей) можно построить модель, аналогичную моделям PPM и DMC, обеспечивающую такое же или лучшее качество сжатия, поэтому такие алгоритмы словарного сжатия — подмножество методов статистического моделирования; Беллом также было высказано предположение о том, что вышеприведенное утверждение справедливо для *любых* алгоритмов словарного сжатия.

Таким образом, можно полагать, что для любого алгоритма словарного сжатия можно построить модель статистического моделирования, обеспечивающую такое же или лучшее качество сжатия, поэтому алгоритмы словарного сжатия не являются (и никогда не будут) лидерами по качеству сжатия. Однако это не мешает им обладать рядом исключительно ценных качеств, по которым они опережают почти все другие известные алгоритмы сжатия (см. [38, 39, 92] и главу 11), а именно:

- достаточно высокая скорость кодирования (в 5–20 раз выше, чем у методов статистического моделирования);
- чрезвычайно высокая скорость декодирования, практически равная скорости копирования подстрок (в сотни и тысячи раз выше скорости методов статистического моделирования);
- небольшой объем требуемой памяти (в 10–100 раз меньше необходимой для методов статистического моделирования);
- приемлемое качество сжатия (обычно в 1.2–1.7 раза хуже, чем у методов статистического моделирования);

По этим причинам подавляющее число систем сжатия данных построена на основе той или иной модификации алгоритма словарного сжатия.

2.8.1. АЛГОРИТМЫ СЕМЕЙСТВА LZ77

Алгоритмы семейства LZ77 — LZ77 [201], LZR [155], LZSS [35, 167], LZB [36], LZH [47], LZFG [87], LZBW [43], LZWZ [197], LZWW [196] и др. — отличаются от других алгоритмов словарного сжатия тем, что в качестве словаря выступает множество *всех* подстрок уже просмотренной части кодируемой подстроки (т. е. после просмотра двух символов строки *abcd* словарь будет содержать строки *a*, *ab*, *abc*, *abcd*, *b*, *bc*, *bcd*), а ссылка в словарь записывается парой *(смещение, длина)*, где *смещение* определяется разностью от текущей позиции в кодируемой строке до предыдущего вхождения такой же подстроки. Так как словарь не обязательно содержит все строки единичной длины, то в словаре может не оказаться строк, начинающихся с текущего символа. В таких случаях кодировщик оставляет текущий символ на месте и сдвигается к следующему.

Для повышения эффективности поиска подстрок, как правило, максимальное смещение ограничивается некоторым параметром W, так что подстроку, являющуюся началом еще незакодированного хвоста строки, необходимо искать только среди расположенных не более чем W символов левее текущей позиции; такая модификация алгоритма LZ77 называется LZ77 со скользящим окном (LZ77 with sliding window), т. к. множество начал возможных подстрок находится в окне ширины W, расположенном непосредственно перед текущей позицией. Нетрудно заметить, что LZ77-схемы без ограничений на смещение указателя –частный случай LZ77-схем со скользящим окном при $W = \infty$.

Если имеется несколько подстрок, являющихся началом текущей, обычно выбирается наиболее длинная подстрока, ближайшая к текущей позиции; такая эвристика называется Longest Match Heuristic — LMH.

Обозначим $S = s_0 \dots s_{L-1}$ кодируемую строку длины $L > 0$ над алфавитом Σ, а W — ширину окна, $W > 1$.

Определение 2.19. *Литералом* называется символ кодируемого алфавита $|\Sigma|$, а *указателем* — пара натуральных чисел (ℓ, p) такая, что $\ell, p > 0$ и $\ell, p < W$. ■

Определение 2.20. *LZ77-кодом непустой* строки $S \in \Sigma^*$ $(|S| > 0)$ называется *непустая* последовательность S' $(|S'| > 0)$ литералов и указателей, порождающая S при декодировании S' по следующему алгоритму:

1. Установить i и j в 0.
2. Увеличить j на единицу; если j-ый символ S' — литерал α, перейти к шагу 3, иначе j-ый символ S' — указатель (ℓ, p); в таком случае перейти к шагу 4.
3. Увеличить i на единицу, установить s_i в α и перейти к шагу 6.
4. Если $p > i$, то S' не является корректным LZ77-кодом; закончить декодирование и вернуть пустую строку.
5. Для $k = 1, \dots, \ell$ установить s_{i+k} в s_{i+k-p}, затем увеличить i на ℓ.
6. Если $j < |S'|$, то перейти к шагу 2, иначе декодирование закончено и его результатом является строка $s_1 \dots s_i$.

■

Таким образом, выходом кодировщика является последовательность указателей и литералов, называемая *LZ-кодом*. Например, LZ-кодом строки *aaaaa* может быть последовательность *a(1,1)(2,2)a* или *a(1,4)*; несмотря на рекурсию в последнем указателе, кодирование не будет двусмысленным (при условии, что декодировщик будет просто копировать указанную ему подстроку по одному символу слева направо).

В ряде случаев накладываются ограничения на возможную последовательность указателей и литералов (например, собственно алгоритм LZ77 требует, чтобы за указателем обязательно следовал литерал), а также запрещается рекурсия в указателях. Поскольку такие ограничения нужны только для облегчения теоретических изысканий и на практике приводят к ухудшению качества сжатия (не упрощая алгоритмы кодирования или декодирования), в дальнейшем будут рассматриваться так называемые *рекурсивные LZ77-схемы со свободной смесью* указателей и литералов в выходном потоке, т. е. без вышеперечисленных ограничений.

Следует отметить, что кодирование в текущей позиции самой длинной подстроки не всегда оптимально; иногда имеет смысл закодировать литерал даже при наличии подходящей подстроки. Например, строка *abcbcababcab* может быть закодирована двумя

способами: *abc(2,2)(2,5)(2,2)(3,7)(2,3)* или *abc(2,2)(2,5)a(4,5)*. Очевидно, если стоимость кодирования литерала меньше стоимости кодирования указателя (а это почти всегда так), то второй LZ77-код короче первого.

Способы построения *оптимального* LZ77-кода (т. е. LZ77-кода с минимальной стоимостью кодирования) рассматриваются в главе 5.

Эффективные методы поиска подстрок для LZ77-схем описаны в главе 6.

Эффективные способы энтропийного кодирования выхода LZ77 рассмотрены в главе 7.

2.8.2. СЖАТИЕ ТЕКСТОВ ЗАМЕНОЙ СЛОВ

Один из самых простых алгоритмов словарного сжатия текстов на естественных языках заключается в построении списка слов, используемых в тексте, и заменой слов ссылками в словарь. Такой способ сжатия был известен со времени появления первых ЭВМ; первая известная автору статья была написана еще в 1963 г. [160]. С тех пор были предложены сотни различных (отличающихся деталями) методов сжатия текстов заменой слов (см. [45, 76, 99, 108, 114, 115, 139, 143, 148, 149, 157, 160, 165, 182, 183, 192, 193] и др.): со статическим словарем (так что сжатие осуществляется за два прохода — построение словаря и собственно кодирование), адаптивным словарем, который строится во время кодирования, и методы сжатия с пополняемым словарем (адаптивное кодирование при непустом начальном словаре).

В настоящее время интерес к таким методам сжатия существенно вырос, т. к. заметно увеличились объемы хранимых и передаваемых данных, представляющих собой в основном текст на естественном языке: тексты подсказок и справочной информации, прилагаемые к большинству современных диалоговых программ, составляют (уже в сжатом виде) большую часть объема системы — от 15% до 30%, а пересылаемые по сетям Internet данные, за редким исключением, представляют собой текст, иногда сопровождаемый картинками. Кроме того, в связи с появлением мощных средств поиска данных в сети WWW (AltaVista, Yahoo, InfoSeek и др.), электронных библиотек и депозитариев технических отчетов (WAIS, NCSTRL, NZDL и др.), также предоставляющих возможность поиска информации среди всех хранимых документов по словам или фразам, особую актуальность приобрели задачи хранения, пополнения и модификации огромных массивов текстовой информации (иногда до нескольких сотен гигабайтов), объединенные с задачей поиска слов или фраз в хранимых данных и осложняемые необходимостью обеспечения почти произвольного доступа к сжатым данным (как правило, по границам параграфов).

Методы сжатия текстов заменой слов для решения таких задач подходят очень хорошо, т. к. при хорошем качестве сжатия они позволяют легко осуществлять поиск слов в сжатых данных и декодирование произвольного (с точностью до границы слова) участка данных.

Эффективность реализации сжатия заменой зависит от того, насколько эффективно кодируются

- слова в словаре;
- ссылки в словарь;
- разделители (пробелы, запятые и т. п.).

Обычно номера слов в словаре кодируются кодами Хаффмана с учетом частоты использования слов. Арифметические коды мало пригодны для этой цели, т. к. требуют хранения вместе со сжатыми данными не длин кодовых слов, а частот появления

(что менее эффективно и ухудшает качество сжатия) и, самое главное, при большой мощности словаря скорость декодирования арифметических кодов на порядок ниже скорости декодирования кодов Хаффмана.

Поскольку при больших объемах сжимаемых данных размер словаря может быть очень большим, задача построения кода Хаффмана для слов словаря и проблема хранения собственно словаря становятся достаточно нетривиальными. Как это ни удивительно, обычно используются очень прямолинейные подходы, т. е. словарь сжимается по принципу как-нибудь (обычно посимвольным кодированием по Хаффману), слова в словаре упорядочиваются в порядке их появления в кодируемом тексте и т. д. Кроме того, в некоторых работах необходимость кодирования разделителей (запятых, точек, пробелов и т. д.) полностью игнорируется, что на практике не приемлемо.

Эффективные по скорости кодирования и декодирования неискажающие методы сжатия текстов на естественных языках, обладающие высоким качеством сжатия, описаны в главе 4.

2.9. АЛГОРИТМ СЖАТИЯ СОРТИРОВКОЙ БЛОКОВ

Алгоритм сжатия сортировкой блоков (BSLDCA — Block Sorting Lossless Data Compression Algorithm), впервые описанный 1994 г. Барроузом и Уилером [55], прост и элегантен.

Для того, чтобы сжать строку S длины N, образуем матрицу M размера $N \times N$ из циклических вращений строки S на один символ, так что i-я строка матрицы содержит символы

$$M_i = S_i S_{i+1} \dots S_{N-2} S_{N-1} S_0 S_1 \dots S_{i-2} S_{i-1}, \tag{2.28}$$

после чего получим матрицу M' сортировкой M по строкам в лексикографическом порядке.

```
        Матрица M                          Матрица M'
a b c b c a b a b c a b .    0       a b a b c a b . a b c b c
b c b c a b a b c a b . a    1       a b c a b . a b c b c a b
c b c a b a b c a b . a b    2   ==> a b c b c a b a b c a b .
b c a b a b c a b . a b c    3       a b . a b c b c a b a b c
a b a b c a b . a b c b c    4       b a b c a b . a b c b c a
b a b c a b . a b c b c a    5       b c a b a b c a b . a b c
a b c a b . a b c b c a b    6       b c a b . a b c b c a b a
b c a b . a b c b c a b a    7       b c b c a b a b c a b . a
c a b . a b c b c a b a b    8       c a b . a b c b c a b a b
a b . a b c b c a b a b c    9       c b c a b a b c a b . a b
. a b c b c a b a b c a b    10      . a b c b c a b a b c a b
```

Барроуз и Уилер (см. [55], а также главу 9) показали, что знания последнего столбца матрицы M' (`cb.cacabbb`) и индекса исходной строки в M' (в данном примере — 2) достаточно для восстановления исходной строки за $O(N)$ операций и с использованием $O(N)$ слов дополнительной памяти.

Такое преобразование данных, по аналогии с преобразованием Фурье называемое преобразованием Барроуза — Уилера (BWT — Burrows—Wheeler Transformation), само по себе не уменьшает объем кодируемых данных, а увеличивает его (за счет необходимости передачи индекса исходной строки в M'), однако преобразованные

данные гораздо более приспособлены для сжатия из-за большого количества повторов символов (что хорошо видно уже на нашем примере).

Это объясняется тем, что две соседние строки матрицы M', скорее всего, имеют достаточно длинное общее начало, которое может рассматриваться как *контекст для предыдущего* символа, расположенного в последнем столбце M'; если контексты похожи, то следует ожидать совпадения и предыдущих символов.

Замечание 2.4. На практике сортировка строк производится сравнением их не слева направо, а справа налево (так что результатом является не последний, а первый столбец матрицы M'). Это несколько улучшает качество сжатия, т. к. предсказывается следующий за контекстом символ, а не предшествующий ему. Лучшее качество сжатия можно объяснить тем, что в большинстве случаев данные записываются слева направо и следующие символы зависят от предыдущих, а не наоборот (Шенноном [162] было показано, что при длине сообщения, стремящейся к бесконечности, разность качества сжатия при изменении направления просмотра стремится к нулю, однако для сообщений конечной длины разница может быть достаточно заметной [55, 83, 84]).

Поскольку сравнение строк слева направо представляется более естественным, для упрощения изложения будем предполагать обычное направление сравнения; при желании нетрудно модифицировать все алгоритмы, описанные в пп. 8 и 9, для варианта BWT со сравнением справа налево. ■

Уилером [55, 185] была создана реализация BWT с использованием для сжатия выхода BWT алгоритма стопки книг, разработанного Б. Я. Рябко [26, 25, 76, 98]; на примере этой реализации было показано, что скорость сжатия примерно в полтора раза уступает LZ77, в 3–5 раз опережая алгоритмы семейств PPM и DMC (однако в наихудшем случае время работы использованного Уилером метода сортировки может превысить $O(N\sqrt{N}\log_2 N)$ — см. главу 8); размер сжатых данных занимает промежуточное положение (примерно полусумма) между размером сжатых данных, полученных с использованием алгоритмов статистического моделирования и алгоритмов семейства LZ77.

К числу преимуществ алгоритма сжатия сортировкой блоков следует отнести высокую скорость сжатия (близкую к LZ77) при очень хорошем качестве сжатия (близком к PPM и DMC) и патентную чистоту этого алгоритма (см. п. 2.11), что очень важно для коммерческих приложений.

К числу недостатков следует отнести, в основном, слабую изученность данного метода сжатия (ввиду его новизны) и вытекающие отсюда сложности в реализации собственно BWT и выборе подходящего алгоритма сжатия результата преобразования, а также достаточно большой — линейный по длине блока — объем требуемой памяти; т. к. размер блока обычно выбирается достаточно большим — от сотен килобайтов до нескольких мегабайтов, то объем требуемой памяти может быть значительным.

Следует отметить, что алгоритм сжатия сортировкой блоков ввиду своей новизны очень слабо изучен; список всех известных автору публикаций исчерпывается [5, 34, 55, 82, 83, 84, 185], из которых большая часть содержит отрицательные результаты. В частности, теоретические оценки качества сжатия неизвестны, и можно лишь утверждать, что при хорошо подобранном алгоритме кодирования выхода BWT качество сжатия будет достаточно близко к достижимому применением статистических методов.

Реализация алгоритма сжатия сортировкой блоков состоит из двух почти независимых частей:

- эффективной реализации преобразования Барроуза — Уилера (алгоритм сортировки с временной сложностью $O\big((N+\sqrt{N})\log_2 N\big)$ с использованием $O(N)$ слов дополнительной памяти, разработанный автором, описан в п. 8);
- эффективного кодирования результата преобразования (применение описанных в п. 10 алгоритмов кодирования позволяет добиться качества сжатия, лишь незначительно — обычно менее 1–2% — уступающего достижимому при использовании наиболее эффективных вариантов статистического моделирования).

Автором также разработан эффективный вариант преобразования Барроуза — Уилера (изложенный в п. 9), который не требует полной сортировки матрицы M, ограничиваясь сортировкой лишь по нескольким первым символам строк. Это позволило существенно (в 1.5–2.5 раза) увеличить скорость сжатия при незначительном (1—5%) ухудшении качества сжатия.

2.10. СОВРЕМЕННЫЕ АРХИТЕКТУРЫ ЭВМ

При разработке программного обеспечения желательно учитывать особенности современных архитектур:

- ввиду наличия конвейера условные переходы и вызовы подпрограмм в 2–10 раз медленнее команд типа регистр-регистр;
- большая разность (в 3–6 раз) тактовых частот процессора и шины приводит к тому, что обращения в память в 8–15 раз медленнее команд типа регистр-регистр;
- наличие небольшого (обычно 8–16 Кб) внутрикристалльного кэша данных, доступ к которым осуществляется за один такт, и кэша второго уровня (обычно 256–512 Кб, на очень дорогих процессорах — MIPS R10000, Alpha 21664, — до 8 Мб), доступ к которому осуществляется за 2–5 тактов, заметно увеличивает скорость обработки данных небольшого объема;
- большая ширина линии данных — обычно 64–128–256 битов — позволяет за одно обращение в память извлекать 2–4 последовательно расположенных 32-битовых слова данных, поэтому последовательный доступ к данным гораздо эффективнее доступа в произвольном порядке;
- параллельная обработка двух и более последовательно расположенных независимых команд позволяет аккуратным перемешиванием вычислений увеличить скорость исполнения.

Таким образом, обработка данных небольшими порциями или в последовательном порядке в 2–5 раз быстрее обработки в произвольном порядке. К сожалению, большинство алгоритмов предназначено для работы со списками, деревьями или графами, поэтому без существенной переработки алгоритмов эти пожелания выполнить трудно или невозможно.

Тем не менее при реализации эти сведения можно (и нужно) учитывать: уменьшать размер структур данных и располагать данные последовательно (тем самым увеличивая коэффициент использования кэша и уменьшая количество обращений к основной памяти), избегать излишних команд перехода, перемешивать независимые вычисления. Опыт автора показывает, что аккуратная реализация может ускорить программу на 20–30%.

2.11. АЛГОРИТМЫ И ПАТЕНТЫ

Многие алгоритмы сжатия данных запатентованы в США; в большинстве случаев аналогичные патенты были получены в странах ЕЭС и Японии. Информация о патен-

тах США, хранящаяся в базе данных IBM, доступна на http://patent.womplex.ibm.com; точная информация о наличии или отсутствии эквивалентных патентов в других странах может быть получена у владельцев патентов.

Отметим, что большинство запатентованных алгоритмов целиком или частично были опубликованы до подачи заявки: согласно патентному праву, новое применение известным методов является изобретением, а патентное право США признает так называемое invention date rule, позволяющее подавать заявку на патент в течение одного года со дня изобретения, причем эта дата не обязана подтверждаться фактом публикации или каким-либо другим документальным способом.

Ниже перечислены ряд методов сжатия, описанных в данной главе, и перечислены номера защищающих их патентов, а также приведены известные случаи, широко освещавшиеся в периодической печати, связанные с нарушением патентных прав.

- Некоторые варианты RLE: 4,586,027; 4,872,009.
- Алгоритм LZRW1 [188, 190]: 4,701,745; 5,049,881.
- Алгоритмы LZFG [87], LZR [155] и другие алгоритмы семейства LZ77, применяющие для поиска подстрок дерево суффиксов: 4,906,991.
  Как отмечено в FAQ comp.compression, архиваторы ZOO, LHA, BOOZ и LHARC, применяющие дерево суффиксов для поиска подстрок, нарушают законы США; они не могут распространяться на территории США, равно как и данные, созданные с их помощью.
- Различные методы поиска подстрок в LZ77-схемах: 5,051,745; 5,016,009; 5,126,739; 5,140,321; 5,155,484.
  В 1993 г. Stac Inc. предъявила иск Mircosoft Corp., обвиняя последнюю в нарушении патента 5,016,009 утилитой DoubleSpace, входившей в состав MS DOS версии 6.0 и 6.02, на сумму в 200 млн. долларов в США. После месяца судебных разбирательств суд был досрочно прекращен, поскольку стороны договорились о том, что Microsoft выплачивает Stac 120 млн. долларов и заменяет все версии MS DOS 6.0 и 6.02 на 6.20 и 6.22.
- Алгоритм LZW [184], применяемый в форматах хранения изображений GIFF, TIFF, и протоколе V42bis для модемов: 4,366,551 (Klaus Holtz); 4,558,302 (Unisys); 4,814,746 (IBM); европейский патент 0,129,439 (British Telecom). Для производителей модемов годовая лицензия на патент 4,558,302 стоит около 25 тыс. долларов США, а на патент 0,129,439 — около 30 тыс. английских фунтов стерлингов. Unisys получает 1.5% от цена каждого проданного в США программного продукта, использующего формат GIF или алгоритм LZW. Доходы Клауса Хольца не известны, но он утверждает, что все производители модемов покупают у него лицензию на патент. До 30 декабря 1995 г. алгоритм LZW (который был запатентован в 1985–1987 гг.) свободно использовался в стандартной для каждой платформы UNIX утилите COMPRESS, применялся в формате хранения изображений GIFF и TIFF, протоколе V42bis для модемов, однако 30 декабря 1995 г. Unisys объявила, что применение LZW без приобретения лицензий незаконно, и потребовала (задним числом) от всех производителей программного и аппаратного обеспечения, использующего LZW, или заменить все такие продукты, проданные ранее, или оплатить лицензии на указанных выше условиях. Это требование было выполнено.
- Алгоритм PPM [66] в сочетании с арифметическим кодированием: 5,210,536 (IBM); 5,406,282 (Ricoh).
- Различные варианты арифметического кодирования и отдельных аспектов их реализации: 4,122,440; 4,286,256; 4,295,125; 4,463,342; 4,467,317; 4,633,490; 4,652,856;

4,792,954; 4,891,643; 4,901,363; 4,905,297; 4,933,883; 4,935,882; 5,045,852; 5,099,440; 5,142,283; 5,210,536; 5,414,423; 5,546,080 (все перечисленные патенты принадлежат IBM); 4,841,092; 4,973,961; 4,989,000; 5,003,307; 5,023,611; 5,025,258; 5,272,478; 5,307,062; 5,309,381; 5,311,177; 5,363,099; 5,404,140; 5,406,282; 5,414,423; 5,418,532; Japan 2-46275 (принадлежат AT&T и Mitsubishi). ELS-coder запатентован Pegasus Imaging Corp.; k-coder запатентован M.I.T.; номера патентов автору не известны, поскольку патентная база данных IBM обновляется достаточно редко, а Pegasus Imaging Corp. и M.I.T. предоставляют патентные данные только по письменному запросу в печатном виде с предоплатой стоимости пересылки и 10 долларов США за страницу — общей стоимостью более 100 долларов США.

Thomas Lane, один из руководителей Independent JPEG Group, в FAQ comp.jpeg пишет: Я не рекомендую применение арифметического кодирования; улучшение качества сжатия не настолько велико, чтобы компенсировать потенциальные потери из-за нарушения [патентных] прав. В частности, арифметическое кодирование не должно использоваться для любых образом, распространяемых через Internet. Даже если вас совершенно не заботит патентный закон США, это волнует других.

Учитывая весьма общие и расплывчатые формулировки патентов, автор присоединяется к этому мнению и также не рекомендует применение арифметического кодирования без приобретения лицензий на патенты (см. Appendix L of CCITT/ITU-T T-81 [JPEG standard], Appendix E of CCITT/ITU-T T-82 [JBIG standard]) 4,463,342; 4,467,317; 4,286,256; 4,491,643; 4,633,490; 4,652,856; 4,905,297; 4,935,88; 5,099,440 (все — IBM); 4,973,961 (AT&T). С июля 1997 г. между IBM и Motion Picture Expert Group ведутся переговоры о бесплатном применении арифметического кодирования в стандарте сжатия движущихся изображений и звука MPEG-2, применяемом в цифровом телевидении и цифровых видеодисках DVD; насколько известно автору, к декабрю 1997 г. эти переговоры не были закончены ввиду жеской позиции IBM.

Квалифицированные юридические консультации бесплатно оказываются в рекламных целях рядом известных юридических фирм США и Великобритании в группах news://comp.legal.computing и news://comp.misc.legal; юристы единодушны в том, что законы лучше не нарушать, а лицензии нужно покупать. Отметим, что закон США позволяет преследовать не только нарушителей патентного права, но и партнеров, клиентов и других лиц и организаций, связанных с нарушителем деловыми отношениями (это правило очень часто применялось во времена КОКОМ в отношении американских деловых партнеров иностранных компаний, нарушавших ограничения КОКОМ на торговлю с СССР); поскольку рынок программного обеспечения США составляет 50–80% мирового, игнорировать законодательство США неразумно.

Таким образом, применение запатентованных алгоритмов в коммерческих продуктах, распространяемых на территории США, Японии и стран ЕЭС, требует приобретения [часто очень дорогостоящих] лицензий, в связи с чем автор, как и комитеты JBIG, JPEG и MPEG, не рекомендует применение арифметического кодирования в коммерческом программном и аппаратном обеспечении, распространяемом в США, странах ЕЭС и Британского содружества.

Отметим, что патентное право подавляющего большинства стран разрешает применение запатентованных алгоритмов, устройств, машин, технологий и т. д. в исследовательских целях и для обучения без приобретения соответствующей лицензии.

Что же касается России, то ситуация не ясна: с одной стороны, патентное право России однозначно говорит о том, что ...алгоритмы, программы, математические методы... не являются изобретением и не защищаются патентным правом; с другой

стороны, известный закон от декабря 1995 г. о защите авторских прав включает в число защищаемых программы и алгоритмы. Автору не известно, насколько нужно модифицировать алгоритм и/или программу, чтобы суд, опирающийся на мнение экпертов, не признал факт плагиата и нарушения авторских прав.

Поскольку все алгоритмы и методы сжатия, а также способы их реализации, разработанные и примененные автором и описанные далее, ориентированы на применение в коммерческих продуктах (bCAD пр-ва ProPro Group Inc., XDS пр-ва XDS Inc., UC2 пр-ва AIP NL), распространяемых в США, Западной и Восточной Европе, Японии, Южной Корее, ЮАР, Бразилии, Австралии и Новой Зеландии и используемых правительственными организациями Великобритании и Южной Кореи, автор сталался быть предельно осторожным и не применять запатентованных алгоритмов и методов.

Часть II

Энтропийное кодирование

Глава 3

ПРЕФИКСНЫЕ КОДЫ И КОДЫ ХАФФМАНА

В этой главе для оценки сложности описываемых и предлагаемых алгоритмов по времени и памяти используется RAM-машина, описанная в п. 2.1.

Далее, если иного явно не оговорено, под префиксным кодом и/или кодом Хаффмана подразумевается *статический канонический* код — префиксный или Хаффмана соответственно (см. пп. 2.5.1 и 2.6.1), построенный для источника Бернулли с *неизвестной статистикой*, т. е. кодирование осуществляется в две стадии: сначала кодируемое сообщение просматривается и определяется количество появлений каждого символа, по которым строится код Хаффмана, а затем производится собственно кодирование, сводящееся к конкатенации битовых строк — кодовых слов символов; кодируемое сообщение предваряется информацией об использованном коде, записанной в виде последовательности длин кодовых слов всех символов алфавита (см. п. 2.6.1 и замечание 2.2).

3.1. НЕКОТОРЫЕ СВОЙСТВА КОДОВ ХАФФМАНА

3.1.1. ИЗБЫТОЧНОСТЬ КОДА ХАФФМАНА

Хаффманом было показано [105], что код, носящий его имя, оптимален, а его избыточность строго меньше одного бита на символ. Эта оценка избыточности достаточно груба: интуитивно понятно, что если символы распределены почти равномерно или нет ярко выраженной неравномерности распределения, то избыточность будет гораздо меньше единицы. В конце 80-х — начале 90-х гг. появился ряд интересных публикаций [56, 57, 58, 59, 74, 131, 158], посвященных более точному анализу избыточности кода Хаффмана.

Эти работы достаточно мало известны даже специалистам, поэтому имеет смысл привести самый важный результат:

$$\mathcal{R} \leq (p_{max} - p_{min}) + \log_2 \frac{2\log_2 e}{e} < (p_{max} - p_{min}) + 0.0861 < 0.253 \qquad (3.1)$$

при $p_{max} \leq \frac{1}{6}$, где p_{max} и p_{min} — наибольшая и наименьшая вероятности появления символов соответственно[1].

Таким образом, давно и хорошо известное практикам утверждение о том, что при отсутствии ярко выраженных пиков в распределении символов избыточность кода Хаффмана гораздо меньше единицы, в последние годы получило и теоретическое подтверждение. Поскольку с практической точки зрения важна не абсолютная, а относительная

[1] При $p_{max} > \frac{1}{6}$ эта оценка не точна и недостижима. В процитированных работах можно найти достаточно точные оценки для почти всех возможных случаев; ввиду громоздкости соответствующие формулы не приводятся.

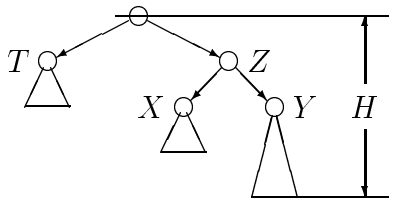

Рисунок 3.1. Дерево Хаффмана высоты H

[к энтропии] избыточность, модификацией метода кодирования и объединением нескольких символов в один обычно можно добиться очень низкой (менее 0.5–1.5%) относительной избыточности (см. главы 7, 10 и п. 2.5.1.4).

3.1.2. МАКСИМАЛЬНАЯ ДЛИНА КОДА ХАФФМАНА

Оценки длины кодов Хаффмана в зависимости от вероятности наименее вероятного символа и связь этих оценок с числами Фибоначчи были, наверное, впервые установлены Катоной и Немецом [116], затем их неоднократно (по мере роста интереса к кодам Хаффмана) переоткрывали [6, 7, 22, 54, 118, 158], в том числе и автор [6, 7]; поскольку в дальнейшем ряд свойств кодов Хаффмана будет очень существенно использоваться (см. п. 3.5), они приводятся в данной работе.

Следует отметить, что при статическом кодировании по Хаффману (см. п. 2.6.1) распределение вероятностей дискретно (см. п. 2.6.6) и вес листа кодового дерева Хаффмана есть не что иное, как количество появлений соответствующего символа в кодируемом сообщении; по построению дерева (см. п. 2.5.1.3) вес W корня в таком случае равен длине кодируемого сообщения.

Теорема 3.1. Если высота дерева Хаффмана равна H и веса двух смежных листьев уровня H равны p и P ($P \geq p$), то вес корня дерева Хаффмана не меньше

$$W \geq PF_{H+1} + pF_H, \tag{3.2}$$

где F_n — n-е число Фибоначчи, $F_1 = F_2 = 1$.

Доказательство. Если $H = 1$, то вес корня дерева равен $W = P + p = PF_2 + pF_1$; допустим, что утверждение (3.2) доказано для всех деревьев Хаффмана высоты не менее $(H-1)$ включительно.

Рассмотрим дерево высоты H (рис. 3.1). Без ограничения общности можно полагать, что смежные листья с весами P и p находятся в правом поддереве, поэтому по предположению индукции вес вершины Y не меньше $W(Y) \geq PF_{H-1} + pF_{H-2}$, а вес вершины Z — не меньше $W(Z) \geq PF_H + pF_{H-1}$.

Так как дерево Хаффмана строится слиянием в одно дерево двух поддеревьев *наименьшего* веса, то вес поддерева с корнем T не меньше веса вершин X и Y,

$$W(T) \geq \max\{W(X), W(Y)\} \geq W(Y) \geq PF_{H-1} + pF_{H-2}, \tag{3.3}$$

поэтому вес корня дерева Хаффмана высоты H, равный сумме весов составляющих его поддеревьев, не меньше

$$W = W(Z) + W(T) \geq \tag{3.4}$$

$$\geq (PF_H + pF_{H-1}p) + (PF_{H-1} + pF_{H-2}) = \tag{3.5}$$

$H_{\max}$	19	20	21	22	23
$W_{\max}$	17710	28656	46367	75024	121392
$H_{\max}$	24	25	26	27	28
$W_{\max}$	196417	317810	514228	832039	1346268
$H_{\max}$	39	30	31	32	33
$W_{\max}$	2178308	3524577	5702886	9227464	14930351

Таблица 3.1. Максимальная высота дерева Хаффмана

$$= PF_{H+1} + pF_H. \tag{3.6}$$

■

Теорема 3.2. Если высота дерева Хаффмана равна H и веса всех символов — целые и ненулевые, то вес корня дерева W не меньше F_{H+2}.

Доказательство. Приняв $P = p = 1$ и применив теорему 3.1, из (3.2) получаем доказываемое утверждение. ■

В табл. 3.1 приведены некоторые пары $(H_{\max}, W_{\max})$: например, если число просмотренных символов $W \leq 75024$, то высота дерева Хаффмана заведомо не больше 22, а если $W \leq 4.5 \cdot 10^{13} \approx 2^{45.36}$, то длина кодового слова не превышает 64 битов. Другими словами, можно быть уверенным, что длина кода Хаффмана не превышает 64 битов, если длина кодируемого сообщения меньше 2^{45} битов (32 Тб), чего вполне достаточно для практических нужд ввиду ограниченных объемов памяти и вычислительных мощностей современных ЭВМ.

Таким образом, высота кодового дерева Хаффмана на практике не бывает большой и обычно не превышает 32 битов.

Теорема 3.3. Максимальная глубина дерева Хаффмана H логарифмически зависит от суммы W весов символов, если они целые и ненулевые:

$$H \leq \lfloor \log_\phi W \rfloor \leq \lfloor 1.441 \log_2 W \rfloor, \tag{3.7}$$

где ϕ — отношение золотого сечения,

$$\phi = \frac{\sqrt{5}+1}{2}. \tag{3.8}$$

Доказательство. По известной формуле Бине [3]

$$F_n = \frac{\phi^n + \psi^n}{\sqrt{5}}, \tag{3.9}$$

где $\psi = -\frac{1}{\phi} = 1 - \phi$ — второй корень характеристического уравнения $x^2 - x - 1 = 0$, соответствующего рекуррентному соотношению, связывающему числа Фибоначчи, поэтому

$$n \leq \log_\phi F_n + 2, \tag{3.10}$$

и доказываемое утверждение следует из того, что $n \in \mathcal{N}$ и (3.2) ($W \geq F_{H+2}$).

Можно также воспользоваться более точной формулой

$$n \leq \log_\phi (F_n \sqrt{5}) + \frac{C}{F_n}, \tag{3.11}$$

где

$$C = 2 - \log_\phi \sqrt{5} < 0.328, \tag{3.12}$$

являющейся следствием того факта, что F_n есть ближайшее к $\frac{\phi^n}{\sqrt{5}}$ целое число. ■

Теорема 3.4. Если дерево Хаффмана имеет N листьев, то для того, чтобы высота дерева Хаффмана превысила $C \log_2 N$, где C — произвольная константа, необходимо, чтобы сумма целых и ненулевых весов символов W была не меньше $1.171 N^{0.695C}$.

Доказательство. $W \geq F_{H+2} \geq \frac{\phi^{H+2}}{\sqrt{5}} - 0.5$. ■

Теорема 3.5. Если вероятность появления символа равна p, то длина L кодового слова Хаффмана не превышает

$$L \leq \left\lceil \log_\phi \frac{1}{p} \right\rceil \leq \left\lceil 1.441 \log_2 \frac{1}{p} \right\rceil . \tag{3.13}$$

Доказательство. Оставим в дереве Хаффмана только остовное поддерево, порожденное путем из листа X, помеченного символом с вероятностью p, к корню дерева; полученное дерево Хаффмана будет *вырожденным* в том смысле, что все вершины, кроме находящихся на пути листу X, будет листовыми.

Обозначим вес вершины Y, братской к вершине X, через P; согласно теореме 3.1, вес корня дерева Хаффмана будет не меньше

$$1 = W \geq pF_{L+1} + PF_L > pF_{L+1}, \tag{3.14}$$

поэтому $F_{L+1} < \frac{1}{p}$ и

$$L < \log_\phi \frac{1}{p} + 1, \tag{3.15}$$

и ввиду того, что L — целое, получаем доказываемое утверждение. ■

Таким образом (см. теорему 3.4), при достаточно больших N и статическом кодировании по Хаффману можно полагать, что $H_{\max} = C \log_2 N$, где C — небольшая константа; этот факт будет очень существенно использоваться при анализе алгоритмов быстрого декодирования префиксных кодов (см. п. 3.5).

3.2. ВЫЧИСЛЕНИЕ СТОИМОСТИ КОДИРОВАНИЯ

Иногда кодирование не уменьшает, а *увеличивает* длину кодируемого сообщения. При использовании статического кодирования, как правило, необходимо знать заранее, является ли принятый способ кодирования (с учетом объема информации о принятом способе кодирования, которую необходимо сообщить декодировщику) сжимающим с тем, чтобы закодированное сообщение гарантированно поместилось в отведенном для него участке оперативной памяти. С этой целью необходимо вычислить стоимость кодирования

$$\mathcal{C}|S| = \sum_{\alpha \in \Sigma} w(\alpha)\ell(\alpha), \tag{3.16}$$

где $|S|$ — длина кодируемого сообщения S, $\mathcal{C}$ — средняя стоимость кодирования, $\ell(\alpha)$ — длина кодового слова символа α, а $w(\alpha)$ — количество появлений символа α в кодируемом сообщении.

В ряде случаев (например, на младших моделях RISC-процессоров) умножение реализовано не аппаратно, а программно и выполняется очень медленно (с учетом времени обработки возникающего прерывания). Однако существует способ вычислить стоимость кодирования вообще без умножений.

Построим массив

$$n_i = \sum_{\alpha\in\Sigma:\ \ell(\alpha)=i} w_i, \tag{3.17}$$

$i = 1, \ldots, H$, где H — максимальная длина кодового слова; другими словами, n_i есть сумма весов символов с длиной кода, равной i. Положим $S_{H+1} = S'_{H+1} = 0$ и вычислим для $i = H, \ldots, 1$ (в порядке убывания i)

$$S_i = S_{i+1} + n_i, \tag{3.18}$$

$$S'_i = S'_{i+1} + S_i. \tag{3.19}$$

Утверждается, что $S'_1 = \mathcal{C}|S|$. Действительно,

$$S_H = n_H, \tag{3.20}$$

$$S'_H = n_H, \tag{3.21}$$

$$S_{H-1} = n_H + n_{H-1}, \tag{3.22}$$

$$S'_{H-1} = 2n_H + n_{H-1}, \tag{3.23}$$

$$S_{H-2} = n_H + n_{H-1} + n_{H-2}, \tag{3.24}$$

$$S'_{H-2} = 3n_H + 2_{H-1} + n_{H-2}, \tag{3.25}$$

$$\ldots \quad \ldots \quad \ldots \tag{3.26}$$

$$S_1 = n_H + \ldots + n_1, \tag{3.27}$$

$$S'_1 = Hn_H + (H-1)n_{H-1} + \ldots + 2n_2 + n_1; \tag{3.28}$$

по определению n_i, $S'_1 = \mathcal{C}|S|$.

При таком способе вычисления стоимости кодирования требуется $H < N$ слов дополнительной памяти (для хранения массива n_i) и $(2H + N) < 3N$ сложений (вместо N сложений и N умножений при вычислениях по формуле (3.16)); таким образом, предложенный способ быстрее, если умножение медленнее сложения в два или больше раз (как упоминалось ранее в п. 2.1, обычно умножение в 15–30 раз медленнее сложения).

3.3. КОДИРОВАНИЕ КОДОВЫХ ДЕРЕВЬЕВ

При использовании статического кодирования необходимо сообщить декодировщику информацию о принятом способе кодирования, т. е. закодировать кодовое дерево.

Рядом авторов[2] [75, 100, 134] предлагалось кодировать дерево обходом его в глубину (слева направо) и записью информации о том, является текущая вершина промежуточной или листовой; в последнем случае необходимо также передать код символа, помечающего данный лист. Такая запись кодового дерева потребует ровно

$$(2N - 1) + N\lceil\log_2 N\rceil \tag{3.29}$$

битов, где N — количество символов в алфавите.

Однако, как уже отмечалось в замечании 2.2, для канонических кодов достаточно передать только длины кодовых слов, а поскольку максимальная длина кодового слова H равна $C\log_2 N$, где C — небольшая константа (см. [14]), такая запись будет более эффективной и потребует не более

$$N\lceil\log_2 C + \log_2\log_2 N\rceil \tag{3.30}$$

битов, что при больших N на порядок лучше предыдущего решения.

[2] Аналогичный способ был применен Е. Налимовым в 1989 г. при создании архиватора ARC для персональных компьютеров YAMAHA.

Однако можно заметить, что большинство символов, кроме нескольких наиболее часто встречающихся, расположены на последних, самых глубоких уровнях кодового дерева, поэтому при $l_i \geq \log_2 N$ вероятность того, что $l_i = l_{i+1}$, очень высока. Это позволяет передавать длины кодов парами (*длина кода, количество идущих подряд символов с такой длиной кода*), используя равномерное кодирование для записи элементов пар. Такой способ был, видимо, впервые предложен и описан (в файле appnote.txt, входящем в дистрибутив продукта) Филом Катцем, автором архиватора PKZIP, и практически без изменений изложен в RFC–1851 для использования в сетевых протоколах TCP/IP обмена данных со сжатием.

В 1991–1994 гг. автором разработан еще более эффективный метод. Вместо того, чтобы записывать *длину кода* очередного символа, предлагается записывать *разность длин кодов* очередного и предыдущего символа; т. к. большинство символов расположены на последних уровнях, то среднеквадратичное уклонение этой разности от нуля будет мало. Кроме того, при таком кодировании неравномерность появления различных разностей становится достаточно заметной (как правило, доля разностей 0, -1, и $+1$ составляет 60–90%), что позволяет использовать энтропийное кодирование (при достаточно большом N) для эффективной записи разностей длин кодов соседних символов.

Предлагается ограничиться использованием небольшого диапазона разностей (например $[-6, +6]$), а если разность выходит за границы выбранного диапазона, то передавать сначала некоторый выделенный символ (например, $+7$), а после этого — непосредственно длину кодового слова очередного символа, используя $\lceil \log_2 H \rceil$ битов, где H — максимальная длина кодового слова. Эмпирические исследования автора показали, что при таком диапазоне разностей необходимость непосредственной передачи длины кода символа не превышает 5–7% и не оказывает существенного влияния на качество сжатия кодовых деревьев.

Замечание 3.1. При необходимости передать последовательность из M кодовых деревьев имеет смысл слить все последовательности длин кодов символов в одну и передать одну длинную последовательность с помощью вышеописанного алгоритма, т. к. это позволит примерно в M раз понизить избыточность кодирования, вызванную необходимостью передавать (в свою очередь) кодовое дерево для кодов, присвоенным разностям. ■

Использование такой техники позволило добиться очень высокого качества сжатия кодовых деревьев, т. к. запись длины одного кодового слова составляет 1.3–1.8 битов при размере алфавита в 100 и более символов; например, запись кодового дерева для 512-символьного алфавита обычно занимает 80–120 байтов (640–960 битов), что примерно вдвое лучше, чем при использовании метода Катца, описанного в RFC–1851. Проведенные автором эксперименты показали, что для алфавитов достаточно большой мощности $|\Sigma| \geq 100$ средняя стоимость передачи длины одного кодового слова не превышает

$$\log_2 \log_2 |\Sigma| \tag{3.31}$$

битов.

3.4. ЭФФЕКТИВНЫЕ МЕТОДЫ ПОСТРОЕНИЕ КОДА ХАФФМАНА

Так как для построения дерева Хаффмана на каждом шаге необходимо сливать два

дерева с наименьшим весом, то самый простой (и в общем случае самый эффективный[3]) способ построения кода Хаффмана [18, 20, 39, 166] заключается в использовании *приоритетной очереди* [18] — структуры данных, позволяющей эффективно выполнять следующие операции:

- FindMinAndRemove — найти элемент с наименьшим ключом и удалить его из очереди;
- AddElem — добавить элемент к очереди;
- IsEmpty — проверить, что очередь непуста;
- Initialize — создать новую очередь.

В таком случае построение дерева Хаффмана выглядит следующим образом: создать очередь из тривиальных деревьев высоты 0, помеченных символами кодируемого алфавита и их вероятностями; извлечь из очереди дерево T_1 с минимальным весом, и если очередь стала пустой, то T_1 — искомое дерево Хаффмана; в противном случае извлечь из очереди еще одно дерево T_2 с минимальным весом среди оставшихся в очереди, слить T_1 и T_2 в новое дерево T (в качестве левого и правого поддеревьев соответственно), пометить T суммой весов T_1 и T_2, добавить T к очереди и перейти к следующей итерации.

Нетрудно заметить, что для построения дерева Хаффмана, имеющего N листьев, потребуется ровно $(N-1)$ итерация.

Приоритетная очередь может быть организована на основе сбалансированных деревьев или пирамид [18], при этом сложность операции Initialize равна $O(N \log_2 N)$, операции IsEmpty — 1, операций FindMinAndRemove и AddElem — $O(\log_2 K)$, где K — количество элементов, находящихся в очереди в настоящий момент. Таким образом, суммарное время построения дерева Хаффмана составляет $O(N \log_2 N)$.

Ввиду того, что на практике распределение вероятностей (см. п. 2.6.6) *всегда дискретно*, без ограничения общности можно полагать, что вместо вероятностей появления символов используются их целочисленные веса.

Может показаться, что переход к целочисленным весам не улучшает ситуацию, однако это не так: например, существуют алгоритмы и структуры данных для приоритетных очередей с целочисленными ключами [48, 171], для которых *среднее время* выполнения[4] вышеперечисленных операций получается заменой $\log_2 N$ на $\log_2 \log_2 N$ при условии, что ключи добавляемых элементов (и изначальной последовательности) распределены *случайно и равномерно*. К сожалению, ввиду того, что вставляемые ключи монотонно возрастают, они не являются случайными, поэтому время исполнения запросов к приоритетной очереди в таком случае становится существенно хуже $O(\log_2 N)$.

Тем не менее все не так плохо. В 1976 г. Van Leeuwen [179] обратил внимание, что если вероятности символов *монотонно возрастают* (или убывают, что несущественно), то можно построить код Хаффмана за линейное время по мощности алфавита.

Действительно, т. к. веса порождаемых деревьев монотонно возрастают, то для их извлечения в порядке возрастания весов достаточно FIFO-очереди (элементы добавляются в конец списка, а извлекаются из его начала). Таким образом, имея две FIFO-очереди — символов и деревьев, — дерево с минимальным весом можно найти за одно сравнение двух первых элементов этих очередей, поэтому время построения дерева Хаффмана равно $O(N)$.

---

[3] Здесь и далее все временные оценки сложности алгоритмов приводятся для RAM-машины, описанной в п. 2.1

[4] Алгоритмы [48, 171] относятся к числу так называемых *рандомизированных* (randomized), обеспечивающих оговоренную скорость с вероятностью $P(N) = 1 - O(1/N)$, где N — количество элементов в очереди.

В 1995–1997 гг. Кормак, Катаяйнен, Моффат и Тюрпин[5] развили эту идею [69, 109, 112, 113, 142, 144, 145, 146]; более того, они показали, что массив целочисленных весов символов можно всего за три прохода по нему преобразовать в массив длин кодовых слов (достаточный для построения кодов символов при использовании канонического кода), причем с использованием только фиксированного количества дополнительной памяти, необходимого для размещения простых переменных для организации циклов.

Кроме того, этими авторами было показано, что на практике при использовании эффективно реализованного алгоритма быстрой сортировки [44] для предварительной сортировки символов по весам и построения кода Хаффмана за линейное время общее время построения кода Хаффмана уменьшается в 4–6 раз по сравнению с алгоритмом, использующим приоритетные очереди, причем было отмечено, что для достаточно больших алфавитов время на собственно вычисление длин кодовых слов в 2–3 раза меньше времени сортировки (что, впрочем, достаточно очевидно). Однако такой подход не улучшает линейно-логарифмическую оценку времени построения кода Хаффмана.

Можно существенно воспользоваться тем фактом, что веса символов — целые числа. Поскольку длина кодируемого сообщения не превышает $C2^w$ ввиду ограниченной по сравнению с 2^w емкостью запоминающих устройств (см. 2.1), при кодировании статическим кодом Хаффмана (см. п. 2.6.1) веса символов есть не что иное, как количество появлений символов в кодируемом сообщении, поэтому без ограничения общности можно считать, что вес символа (и сумма весов всех символов) помещается в машинное слово ширины w битов RAM-машины, описанной в п. 2.1.

Фредман и Уиллард в 1993 г. в своей статье Преодолевая информационно-теоретический барьер с помощью fusion-деревьев [90] показали, что на RAM-машинах при $w = O(\log N)$, описанных в п. 2.1 (и для которых приводятся все временные оценки в данной работе), сортировка N целых чисел может быть выполнена за

$$O\left(N\frac{\log_2 N}{\log_2\log_2 N}\right) \tag{3.32}$$

операций с использованием $O(N)$ слов дополнительной памяти. В 1995 г. Андерссоном было показано [29], что использованием смеси fusion-деревьев и дерева van Emde Boas'a [178] можно уменьшить время сортировки Фредмана и Уилларда до

$$O\left(N\sqrt{\log_2 N}\right) \tag{3.33}$$

операций (правда, при этом потребуется более $O(N)$ слов дополнительной памяти). Наконец, в 1996 г. Нильссон [151] показал, что можно добиться времени целочисленной сортировки за

$$O\big(N\log_2\log_2 N\big) \tag{3.34}$$

операций при $N < 2^w$.

Замечание 3.2. Все предыдущие формулы справедливы лишь при ширине машинного слова $w = O(\log N)$ (см. п. 2.1). ■

Итак, использование целочисленных алгоритмов сортировки позволяет в ряде случаев существенно улучшить время построения кодов Хаффмана. Тем не менее применение вышеописанных алгоритмов сортировки на практике представляется неоправданным: как показали исследования Нильссона [151], при $N \le 10^6$ и при использовании 32-битовых ключей алгоритмы целочисленной сортировки лишь незначительно — примерно на 10% — опережают хорошо реализованный алгоритм быстрой сортировки [44];

[5] Несколько менее эффективный алгоритм был реализован автором в 1993 г.

учитывая существенно большую сложность реализации и достаточно большой объем требуемой дополнительной памяти (до $10N$ слов), автор полагает использование таких алгоритмов непрактичным — во всяком случае, в настоящее время.

3.4.1. ЭФФЕКТИВНЫЙ И ПРАКТИЧНЫЙ АЛГОРИТМ СОРТИРОВКИ

Можно воспользоваться тем, что на практике ключи заведомо помещаются в одно слово; поскольку длина кодируемой последовательности ограничена 2^w, где w — ширина машинного слова (обычно 32 бита), на практике можно полагать, что $K \le 24$ или $K \le 32$, где $K = \log_2 w_{\max}$ и $w_{\max}$ — наибольший вес символа.

Это позволяет разбить ключ сортировки на $U = \left\lceil \frac{K}{V} \right\rceil$ групп битов, где V — некоторая константа, и с использованием $(N+2^V)$ слов дополнительной памяти отсортировать ключи за U проходов МЦ-сортировки [18], каждый из которых потребует $\Theta(N + C2^V)$ операций, где C — некоторая константа (обычно много меньше единицы), так что суммарное время сортировки составит $\Theta(\left\lceil \frac{K}{V} \right\rceil (N + C2^V))$.

Ввиду того, что V и U — целые, анализ этой функции далеко не так прост, как кажется с первого взгляда. Можно показать, что оптимальное значение $U = 2$, если $N \ge 2^{0.5K}$, иначе же оптимальное значение $U = \left\lceil \frac{K}{\log_2 N} \right\rceil$; очевидно, оптимальное значение V равно $\left\lceil \frac{K}{U} \right\rceil$.

Таким образом, время сортировки не превышает $\Theta(\frac{2KN}{\log_2 N})$ (а при больших $N \ge 2^{0.5K}$ не превышает $\Theta(4N)$), что при $N \ge 2^{\sqrt{2K}}$ лучше времени сортировки сравнениями, а при $N \ge 2^{\sqrt{2K}-2}$ скорость целочисленной сортировки мало отличается от скорости сортировки сравнениями.

Например, при $K \le 24$ предложенный алгоритм сортировки не хуже сортировки сравнениями, начиная с $N = 32$, а при $K \le 32$ — начиная с $N = 64$ (при меньших значениях N следует использовать традиционные методы сортировки сравнениями).

При больших же значениях N, например $N \ge 10^6$ и $K \le 32$ (см. пример использования таких больших алфавитов в [143, 148, 149, 193, 194]), так что $2^V \ll N$ при $V = \frac{K}{2}$, время предложенной сортировки составит $\Theta(2N)$, тогда как время сортировки сравнениями составит $\Theta(N \log_2 N) > \Theta(20N)$ — в 10 раз больше времени предложенного алгоритма.

Применение такого алгоритма сортировки позволяет в 2–4 раза ускорить построение кода Хаффмана по сравнению с алгоритмом Катаяйнена и Моффата [109] и, соответственно, в 6–20 раз — по сравнению с традиционным подходом [18, 20, 39, 166].

Кроме того, к числу немаловажных достоинств данного алгоритма следует отнести простоту его реализации, которая занимает 20–30 строк, по сравнению с алгоритмом быстрой сортировки в варианте Бентли и МакИлроя [44], реализация которого требует сотни строк весьма нетривиального текста.

3.5. БЫСТРОЕ ДЕКОДИРОВАНИЕ ПРЕФИКСНЫХ КОДОВ

Как правило[20, 39, 166], декодирование префиксных кодов производится при помощи алгоритма *побитового декодирования*: начиная с вершины дерева, сдвигаться вниз по дереву по левой или правой ветви в зависимости от очередного бита входного потока; при достижении листовой вершины выдать символ, приписанный данному листу

дерева, и закончить декодирование символа. Следует отметить, что объем требуемой памяти равен $O(N)$, где N — мощность алфавита, которому принадлежат декодируемые символы.

Очевидно, что время такого декодирования прямо пропорционально длине закодированного сообщения, а среднее количество операций, необходимых для декодирования одного символа, в точности равно стоимости кодирования. Как правило, префиксные коды используются при достаточно высокой стоимости кодирования (обычно 4–10 битов), чтобы относительная избыточность кодирования была невелика, поэтому скорость побитового декодирования обычно на порядок ниже скорости кодирования.

Скорость декодирования можно существенно увеличить, если декодировщик будет за один раз обрабатывать не один, а сразу несколько последовательно идущих битов входного потока.

В 1985–1986 гг. Франкель и др. [60, 61] предложили способ быстрого декодирования префиксных кодов за счет использования $O(N^2)$ слов памяти, а в 1988 г. Симински [164] предложил еще более эффективный метод декодирования, также с использованием $O(N^2)$ слов памяти. Несмотря на достаточно высокую скорость декодирования, мало зависящую от формы префиксного дерева, эти алгоритмы редко применяются на практике, т. к. при большом объеме структур данных, используемых при декодировании, и длине закодированного сообщения, меньшим или сравнимым с N^2, время инициализации может оказаться сравнимым или даже превысить время собственно декодирования. Кроме того, квадратичная зависимость требуемых объемов памяти от мощности алфавита налагает на него существенные ограничения, и при $N \geq 256$ объем требуемой памяти (256 Кб и более) становится неприемлемым. К числу недостатков этих алгоритмов следует отнести то, что они не позволяют производить декодирование смешанных потоков данных, в которых могут быть перемешаны в произвольном порядке символы разных алфавитов, закодированных с использованием разных кодов (что является традиционной практикой).

В 1992 г. Хиршберг и Лельюер [96] предложили интересную модификацию алгоритма побитового декодирования, при которой декодировщик на каждом шаге выбирает из входного потока столько битов, каково кратчайшее расстояние от текущей вершины префиксного дерева до листовой, при этом объем требуемой памяти равен $O(N)$. К сожалению, часто бывает так, что префиксное дерево содержит листья на каждом или почти каждом уровне текущего поддерева, поэтому в таких случаях скорость декодирования мало отличается от скорости побитового декодирования.

В 1997 г. Кляйн [119] предложил алгоритм декодирования, аналогичный самому эффективному из рассмотренных Хиршбергом и Лельюер [96] алгоритму B2, за исключением деталей в организации памяти. Эксперименты, проведенные Кляйном, показали, что использование такого алгоритма позволяет ускоряет декодирование по сравнению с побитовым на 30–50%.

В 1991–1992 гг. автором [7] был разработан весьма эффективный — как по скорости, так и по памяти — метод декодирования полных[6] префиксных кодов при несущественных ограничениях на форму префиксного дерева. Как будет показано, использование этих алгоритмов позволяет уменьшить время декодирования символа в 3–8 раз при достаточно небольшом расходе дополнительной памяти.

[6] Результаты немедленно обобщаются на случай неполных префиксных кодов добавлением к алфавиту новых символов и кодовых слов до получения полного кода.

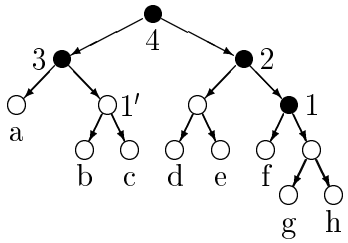

Рисунок 3.2. Каноническое дерево с минимальной 2-сетью

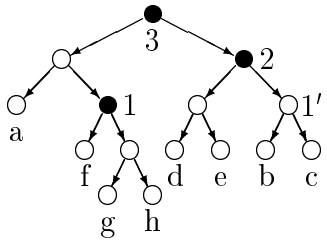

Рисунок 3.3. Эквивалентное 2-минимальное дерево

3.5.1. АЛГОРИТМ ТАБЛИЧНОГО ДЕКОДИРОВАНИЯ

Определение 3.1. *Q-сетью* ($Q \in \mathcal{N}$, $Q \geq 1$) на дереве называется множество вершин дерева, удовлетворяющее следующим условиям:

- корень дерева принадлежит сети;
- любой направленный путь (т. е. строго вниз или строго вверх по дереву), соединяющий любую вершину Q-сети с ближайшим листом дерева или другой вершиной Q-сети, содержит не более Q ребер;
- листья дерева не принадлежат Q-сети (за исключением дерева высоты 0).

■

Определение 3.2. *Мощностью Q-сети* называется множество вершин, принадлежащих Q-сети. ■

Определение 3.3. Если мощность некоторой Q-сети не больше мощности любой другой Q-сети на данном дереве, то такая сеть называется *минимальной* Q-сетью. ■

Определение 3.4. Дерево T называется *Q-минимальным*, если мощность минимальной Q-сети на T не больше мощности минимальной Q-сети на любом дереве, эквивалентном T. ■

Если на кодовом дереве имеется некоторая Q-сеть, то можно построить алгоритм быстрого табличного декодирования соответствующего кода. Действительно, создадим для каждой вершины v Q-сети таблицу размера 2^Q, каждый элемент которой указывает на u — другую вершина сети или лист дерева, достижимый из v, причем если путь от v до u описывается битовой строкой s, то s — префикс двоичной записи индекса элемента таблицы, содержащего ссылку на вершину v (следовательно, длина s не превышает Q). Заметим, что по определению Q-сети построение такой таблицы для каждой вершины Q-сети возможно.

Начиная с таблицы, соответствующей корню дерева, из входного потока выбирается Q битов, которыми индексируется таблица, соответствующая текущей вершине v, тем самым получается ссылка на другую вершину дерева u, после чего из входного потока удаляется k битов, где k — количество ребер дерева на пути от v к u. Если u — листовая вершина, то декодирование закончено и его результатом будет символ, помечающий вершину u. Если же u — промежуточная вершина, то по построению таблицы она принадлежит Q-сети и тогда необходимо продолжить декодирование, используя таблицу, соответствующую вершине Q-сети u.

3.5.2. ПОСТРОЕНИЕ ТАБЛИЦ ДЕКОДИРОВАНИЯ ПО Q-СЕТИ

Перед началом построения таблиц обходом дерева от корня к листьям пометим каждую вершину u (кроме корня дерева) ссылкой на ближайшую вышестоящую вершину Q-сети, используя следующий алгоритм:

```
void mark_nearest (Node *v, Node *n, int path, int length) {
  v->net    = n;
  v->path   = path;
  v->length = length;
  if (v->left == NULL)
    return;             // nothing more for leaf nodes
  if (v->net != 0) {
    n = v;              // "v" belongs to the Q-net
    path = length = 0;
  }
  mark_nearest (v->left,  n, path*2,   length+1);
  mark_nearest (v->right, n, path*2+1, length+1);
}

void mark_tree (Node *root) {
  mark_nearest (root, root, 0, 0);
}
```

Замечание 3.3. По определению Q-сети расстояние (количество ребер соединяющего пути) от любой вершины дерева (кроме корня) до ближайшей вышестоящей вершины Q-сети не превышает Q, поэтому значение пометки `length` для любой вершины дерева, кроме корня, находится в интервале $[1, Q]$; для корня дерева значение этой пометки равно 0, а пометка `net` указывает на корень дерева. ■

Замечание 3.4. Поскольку каждая вершина дерева посещается ровно один раз и полное дерево с N листьями ($N = |\Sigma|$) имеет $(2N - 1)$ вершин, время разметки равно $O(N)$. ■

Так как скорость декодирования символа прямо пропорциональна количеству таблиц, которые необходимо просмотреть, желательно, чтобы вершина u, записанная в элемент таблицы, была как можно более далекой от вершины v, которой соответствует таблица (однако длина пути от v до u не должна превышать Q), поэтому на самом деле необходимо пометить вершины, которые могут появиться в качестве элементов таблицы, не ближайшими вышестоящими вершинами Q-сети, а наиболее удаленными вышестоящими вершинами Q-сети, расстояние до которых не превышает Q.

Для того чтобы найти удаленную вершину, нужно пройти вверх по дереву по ближайшим вышестоящим вершинам Q-сети, пока не встретится корень дерева или длина пути не превысит Q, и выбрать последнюю подходящую вершину. Чтобы избежать избыточных вычислений, необходимо записать результат поиска в просмотренные вершины Q-сети, так что последующие поиски для других вершин того же поддерева, расположенных на том же уровне, заведомо не потребуют дополнительного перебора, а для расположенных на соседних уровнях листьев — скорее всего не потребуют перебора.

```
void find_remote (Node *v, int max_length) {
  Node *n = v->net;
  if (n->length == 0 || (max_length -= v->length))
    return;     // skip root of tree and too far nodes
  if (n->length < max_length) {
    find_remote (n, max_length, 0);
    if (n->length <= max_length) {
      v->net     = n->net[0];
      v->path    = (n->path << v->length) + v->path;
      v->length += n->length;
    }
  }
}

void handle_node (Node *v) {
  Node *n, *save_net;
  int   length, path, elems;
  find_remote (v, Q);
  n      = v->net;
  length = v->length;
  path   = v->path << (Q - length);
  for (elems = 1 << (Q - length); --elems >= 0; ++path) {
    n->table[path].node   = v;
    n->table[path].length = length;
  }
}
```

Для построения таблиц необходимо создать список листовых вершин и всех вершин Q-сети, отсортированный в порядке убывания глубины, и применить процедуру `handle_node` ко всем вершинам списка.

Замечание 3.5. Строго говоря, таблицы нужно создавать только для тех вершин Q-сети, которые упоминаются в других таблицах (для корня дерева создание таблицы необходимо). Это можно сделать, применив сначала процедуру `handle_node` к отсортированному по глубине списку листьев дерева, затем к отсортированному по глубине списку вершин Q-сети, элементы которых были затронуты во время первого прохода, и т. д. Для эффективной реализации необходимо помечать вершины Q-сети, для которых заполнялись элементы таблиц, и, если вершина не была помечена, добавлять ее в список вершин, которые будет необходимо обработать на следующем проходе. Однако при таком подходе временная сложность алгоритма может сильно увеличиться ввиду необходимости сортировки. ■

	T_4	T_3	T_2	T_1
00	(a,2)	—	(d,2)	(f,1)
01	(T_3,1)	—	(e,2)	(f,1)
01	(T_2,1)	(b,1)	—	(g,1)
11	(T_1,1)	(c,1)	—	(h,1)

Таблица 3.2. Таблицы декодирования дерева на рис. 3.2

Так как глубина вершины находится в интервале $[0, \ell_{\max}]$, $\ell_{\max} \leq N-1$, то для сортировки списка вершин в порядке убывания глубины вершин можно применить алгоритм сортировки односвязного списка расстановкой [18], используя $(\ell_{\max}+1)$ слов дополнительной памяти; время такой сортировки и последующего просмотра массива ссылок на список вершин равной глубины равно $O(\ell_{\max}+N+K)$, где K — мощность Q-сети.

Суммарное время работы процедуры `find_remote`, которая применяется к каждой листовой вершине и каждой вершине Q-сети, не превышает $O\left(Q(N+K)\right)$ (при $\ell_{\max} \ll N$ суммарное время работы близко к $O(N+K)$, т. к. цикл поиска наиболее удаленной вершины исполняется в большинстве случаев не более одного раза), а количество операций для заполнения таблиц равно $O(K2^Q)$.

Таким образом, суммарное время построения структур данных не превышает

$$O\left(Q(N+K)+K2^Q\right), \tag{3.35}$$

а объем памяти, занимаемый структурами данных, равен $O(N+K2^Q)$ слов.

В табл. 3.2 приведены таблицы декодирования, построенные по описанным алгоритмам для дерева с 2-сетью, изображенном на рис. 3.2. Следует обратить внимание на то, что некоторые элементы таблиц не были заполнены; это означает, что в процессе декодирования эти элементы не достижимы из таблицы, соответствующей корню кодового дерева (в данном примере это таблица T_4).

3.5.3. ПОСТРОЕНИЕ МИНИМАЛЬНОЙ Q-СЕТИ

Так как объем требуемой памяти и время построения структур данных линейно зависят от мощности Q-сети, то необходимо уметь строить минимальные Q-сети.

Алгоритм построения минимальной Q-сети очень прост. Изначально множество вершин сети полагается пустым. Если высота дерева не больше Q, то ко множеству вершин Q-сети добавляется корень дерева и построение Q-сети закончено. Если же высота дерева больше Q, необходимо взять произвольное *листовое* поддерево высоты Q (т. е. такое, все листья которого являются листьями исходного дерева), добавить его корень к Q-сети, а поддерево заменить на листовую вершину (на рис. 3.2 и 3.3 приведены примеры построения минимальных 2-сетей, при этом вершины 2-сети закрашены черным и занумерованы в порядке появления), после чего продолжить построение Q-сети для полученного сокращенного дерева.

По окончании построения Q-сети необходимо восстановить кодовое дерево, присоединяя удаленные поддеревья в обратном порядке (последнее удаленное поддерево присоединяется первым).

Теорема 3.6. Построенная сеть является минимальной.

Доказательство. Докажем индукцией по высоте и по количеству вершин исходного

дерева T, что построенная Q-сеть S действительно будет минимальна.

Если высота дерева T не превышает Q, то S содержит ровно одну вершину — корень дерева.

Допустим, что существует другая Q-сеть S' мощности меньше $|S|$ на дереве T высоты более Q. Рассмотрим дерево T', полученное из дерева T в процессе построения Q-сети удалением самого *первого* поддерева высоты Q. Если V — множество *промежуточных* вершин T', то $S \cap V$ и $S' \cap V$ — Q-сети на T', причем количество вершин дерева T' меньше количества листьев T как минимум на Q, т. к. дерево высоты Q (как полное, так и неполное) имеет не менее $(Q+1)$ вершин. По построению S и по предположению индукции $|S \cap V| \leq |S' \cap V|$, а т. к. удаленное поддерево было высоты Q, то $S' \cap V$ не может быть Q-сетью на дереве T, поэтому $|S'| \geq |S' \cap V| + 1 \geq |S \cap V| + 1 = |S|$. ■

3.5.4. ПОСТРОЕНИЕ Q-МИНИМАЛЬНЫХ ДЕРЕВЬЕВ

Как видно из примеров на рис. 3.2 и 3.3, мощность минимальной Q-сети зависит от формы дерева; несмотря на то, что канонические деревья очень удобны и широко используются на практике, они не оптимальны для построения Q-сетей.

Интуитивно понятно, что дерево Q-минимально среди эквивалентных ему, если на каждом шаге алгоритма удаляется поддерево, содержащее *максимальное* количество вершин.

Чтобы удаляемое поддерево содержало максимальное количество вершин, необходимо на каждом шаге алгоритма построения минимальной Q-сети преобразовывать дерево к каноническому виду и выбирать поддерево высоты Q (или корневое поддерево, если высота текущего дерева не превышает Q), содержащее самый правый и самый глубокий лист; по определению канонического дерева, такое поддерево содержит наибольшее количество вершин по сравнению с другими листовыми поддеревьями той же высоты.

По окончании построения Q-сети необходимо восстановить кодовое дерево, присоединяя удаленные поддеревья в обратном порядке (последнее удаленное поддерево присоединяется первым). Следует отметить, что восстановленное кодовое дерево (которое будет Q-минимальным) может не совпадать с исходным кодовым деревом (см. рис. 3.2 и 3.3: поддеревья с вершинами, помеченными 1 и $1'$, были обменены в процессе построения 2-минимального дерева).

Q-минимальность полученного кодового дерева немедленно следует из максимальности (по количеству вершин) удаляемых поддеревьев.

Замечание 3.6. Чтобы избежать построения канонических деревьев на каждом шаге построения Q-сети, достаточно хранить упорядоченный по высоте список имеющихся листьев, который однозначно определяет соответствующее каноническое дерево, при этом замена поддерева на листовую вершину выражается в удалении некоторого количества вершин из этого списка и добавлением нового листа. Чтобы избежать добавления в отсортированный список, можно поддерживать два списка листьев, второй из которых формируется за счет появления вершин Q-сети и по порядку построения отсортирован по глубине. ■

Замечание 3.7. К сожалению, Q_1-минимальное дерево может не быть Q_2-минимальным при $Q_1 \neq Q_2$, что сильно ограничивает область применения Q-минимальных деревьев, т. к. кодовые слова, построенные кодировщиком, существенно зависят от выбора Q. Так как Q обычно выбирается не только в зависимости от средней высоты

кодовых деревьев, средней стоимости кодирования и проч., но и в зависимости от аппаратных возможностей (например, для эффективной реализации битового ввода крайне желательно, чтобы Q было не больше половины количества битов, которые могут храниться в аппаратном регистре), то использование Q-минимальных деревьев представляется в большинстве случаев нежелательным. ■

3.5.5. Q-СЕТИ НА КАНОНИЧЕСКИХ ДЕРЕВЬЯХ

Как уже упоминалось, на практике используются именно канонические префиксные деревья[7] (см. замечания 2.1 и 2.2), поэтому декодирование канонических кодов представляет большой практический интерес, т. к. позволяет заменить имеющиеся процедуры декодирования более эффективными.

Замечание 3.8. Если на каждом шаге построения Q-сети для удаления будет выбираться поддерево, содержащее самый правый наиболее глубокий лист, то дерево, полученное на очередном шаге, будут выглядеть как каноническое дерево, у которого поддерево с корнем в одной из вершин на пути к самому правому листу заменено на другое каноническое поддерево меньшей высоты, так что получающиеся в процессе построения деревья если и не канонические, то мало от них отличаются. ■

При таком способе выбора корня удаляемого поддерева можно применить подход, описанный в замечании 3.6, и тем самым избежать обходов дерева при построении Q-сети и гарантировать построение минимальной Q-сети за линейное время по количеству вершин канонического дерева.

3.5.6. Q-ОПТИМАЛЬНЫЕ СЕТИ НА ДЕРЕВЬЯХ

Определение 3.5. Q-сеть на дереве T является *Q-оптимальной*, если путь от любого листа α дерева до корня можно разбить вершинами Q-сети так, что длина каждого участка не превышает Q, а количество участков в точности равно $\left\lceil \frac{\ell(\alpha)}{Q} \right\rceil$, где $\ell(\alpha)$ — глубина листа α. ■

Другими словами, Q-сеть оптимальна тогда и только тогда, когда количество итераций для декодирования любого символа в точности равно $\left\lceil \frac{\ell(\alpha)}{Q} \right\rceil$ (и не может быть улучшено при фиксированном Q).

В частности, тривиальная Q-сеть, состоящая из всех промежуточных вершин дерева, Q-оптимальна. Однако больший интерес представляют *минимальные Q-оптимальные* сети.

Определение 3.6. Q-сеть является *минимальной Q-оптимальной* сетью на дереве T, если ее мощность не превышает мощности любой другой Q-оптимальной сети на T. ■

Как будет показано при анализе скорости декодирования, при максимальной длине кодового слова $\ell_{\max} \leq 2Q$ любая Q-минимальная сеть на дереве является минимальной Q-оптимальной сетью.

К сожалению, автору не известно, как строить Q-оптимальные сети минимальной мощности в общем случае. Можно предположить, что введением не более одной дополнительной вершины между двумя соседними вершинами минимальной Q-сети ее удастся

[7] Возможно, исключения и существуют, но автору о них не известно.

преобразовать в Q-оптимальную сеть, так что мощность минимальной Q-оптимальной сети будет менее чем вдвое превышать мощность Q-минимальной сети. Тем не менее данный вопрос остается открытым и требует дальнейшего изучения.

Разработчик архиватора ZIP Жан-Лу Гэйли (Gean-loup Gially) в 1991–1992 гг. предложил помещать вершину в Q-сеть, если ее глубина кратна Q, что гарантирует оптимальность построенной сети (см. исходные тексты архиваторов GZIP и InfoZIP версии 4.1 и выше). К сожалению, мощность ее может быть очень велика, и нетрудно привести пример, когда мощность сети равна $O(N)$ (если все листья дерева расположены на уровнях $(kQ+1)$). Это вынуждает использовать таблицы переменного размера, что усложняет декодирование. Кроме того, на настоящий момент не известны оценки требуемой памяти при использовании таблиц переменного размера, и необходимость динамического заказа памяти в момент построения таблиц декодирования является серьезным недостатком данного подхода — например, если декодируемый поток состоит из нескольких сегментов, закодированных различным образом (как обычно и бывает), то в процессе декодирования может оказаться, что все ресурсы свободной памяти уже исчерпаны; это вызовет аварийной останов программы, что как минимум неприятно, а иногда и просто неприемлемо (при создании встроенных бортовых систем, систем контроля и обеспечения безопасности и т. д.).

3.5.7. ПОСТРОЕНИЕ ПОЧТИ ОПТИМАЛЬНОЙ Q-СЕТИ

Автором [7] был предложен несколько иной способ построения оптимальных Q-сетей: вершина попадает в Q-сеть тогда и только тогда, когда поддерево с корнем в этой вершине содержит лист на глубине, кратной Q. Фактически, это равносильно тому, что для всех листьев дерева в Q-сеть помещаются вершины, расположенные на Q, $2Q$ и т. д. уровней выше, тем самым гарантируя оптимальность построенной Q-сети.

Определение 3.7. Q-сеть, построенная таким образом, в дальнейшем будет именоваться *Q-оптимальной сетью почти минимальной мощности*, или, для краткости, *почти оптимальной* Q-сетью. ■

Очевидно, что построенная таким образом Q-оптимальная сеть не минимальна. Более того, в замечании 3.10 приведен пример, когда мощность даже Q-минимальной сети на дереве небольшой высоты будет равна $\left\lceil \frac{N}{Q} \right\rceil$.

3.5.8. ОЦЕНКА МОЩНОСТИ ПОЧТИ ОПТИМАЛЬНОЙ Q-СЕТИ ДЛЯ КАНОНИЧЕСКИХ ДЕРЕВЬЕВ

Теорема 3.7. Для канонического дерева высоты H, имеющего N листьев, мощность почти оптимальной Q-сети не превышает

$$1 + \frac{M}{2Q}\left(H - \frac{Q}{2} - \frac{1}{M}\right)^2, \tag{3.36}$$

где

$$M = \left\lfloor \frac{N-Q-1}{2^Q} \right\rfloor + 2, \tag{3.37}$$

поэтому при $Q \geq \log_2 N$ мощность почти оптимальной Q-сети не больше

$$1 + \frac{1}{Q}\left(H - \frac{Q+1}{2}\right)^2 \tag{3.38}$$

и в таком случае при $N \to \infty$ расход памяти не превышает

$$O(N \log N). \tag{3.39}$$

Доказательство. Рассмотрим все листья, расположенные на некотором уровне h, $h > Q$, и множество всех поддеревьев высоты Q, содержащих такие листья дерева. Пусть K — количество таких деревьев. Ввиду того, что кодовое дерево — каноническое, при $h = H$ должно выполняться неравенство

$$(K-1)2^Q + (Q+1) \le N \iff K \le \left\lfloor \frac{N-Q-1}{2^Q} \right\rfloor + 1 \le M - 1, \tag{3.40}$$

т. к. все поддеревья, кроме самого левого, — полные двоичные деревья высоты Q и имеют ровно 2^Q листьев, а самое левое поддерево имеет не менее $(Q+1)$ листьев.

Если же $h < H$, то

$$(K-2)2^Q + 2(Q+1) \le N \iff K \le \left\lfloor \frac{N-2(Q+1)}{2^Q} \right\rfloor + 2 \le M, \tag{3.41}$$

т. к. все поддеревья, кроме крайних, имеют ровно 2^Q листьев, а самое левое и самое правое — не менее $(Q+1)$ листьев.

Таким образом, общее количество поддеревьев высоты Q, имеющих листья дерева уровня $h > Q$, не превышает $(M-1)$ при $h = H$ и M при $h < H$.

Рассмотрим теперь цепочку вершин, расположенных на пути от некоторого листа уровня h к корню дерева на уровнях $h-Q$, $h-2Q$ и т. д.; длина этой цепочки (не включая корень дерева) равна $\left\lfloor \frac{h-1}{Q} \right\rfloor$.

Таким образом, потребуется не более $M \left\lfloor \frac{h-1}{Q} \right\rfloor$ таблиц декодирования для *всех* символов с длиной кодового слова h при $h < H$ и не более $(M-1) \left\lfloor \frac{h-1}{Q} \right\rfloor$ таблиц при $h = H$.

Итак, общее количество вершин Q-сети не превышает (с учетом корня дерева)

$$D(H) = 1 + (M-1) \left\lfloor \frac{H-1}{Q} \right\rfloor + \sum_{h=Q+1}^{H-1} M \left\lfloor \frac{h-1}{Q} \right\rfloor; \tag{3.42}$$

обозначив $H - 1 = kQ + b$ $(0 \le b < Q,\ k, b \in \mathcal{N})$ и воспользовавшись тем, что

$$\sum_{h=nQ+1}^{(n+1)Q} \left\lfloor \frac{h-1}{Q} \right\rfloor = nQ, \tag{3.43}$$

получаем, что

$$\begin{aligned}
D(H) &= 1 + k(M-1) + kbM + \frac{k(k-1)}{2} MQ = && (3.44)\\
&= 1 + k\left(M - 1 + \frac{M}{2}(kQ - Q + 2b)\right) = && (3.45)\\
&= 1 + k\left(M - 1 + \frac{M}{2}(H - 1 + b - Q)\right) = && (3.46)\\
&= 1 + k\left(\frac{M}{2} - 1 + \frac{M}{2}(H + b - Q)\right) = && (3.47)\\
&= 1 + \frac{H-b-1}{Q}\left(\frac{M}{2} - 1 + \frac{M}{2}(H + b - Q)\right). && (3.48)
\end{aligned}$$

Таким образом, функция $D(H)$ выпукла вверх по b и при

$$b = \frac{Q}{2} - \frac{M-1}{M} \tag{3.49}$$

достигает максимума, равного

$$1 + \frac{M}{2Q}\left(H - \frac{Q}{2} - \frac{1}{M}\right)^2, \tag{3.50}$$

поэтому при $Q \geq \log_2 N$ имеем $M = 2$ и

$$D(H) \leq 1 + \frac{1}{Q}\left(H - \frac{Q+1}{2}\right)^2. \tag{3.51}$$

Так как согласно теореме 3.4 высота дерева Хаффмана $H = O(\log_2 N)$, то $D(H) = O(\log_2 N)$ при $Q = O(\log_2 N) = O(H)$. ■

На самом деле эта оценка сильно завышена, т. к. при $Q \geq \log_2 N$ или некоторые уровни не содержат листьев (а они были учтены), или все листья одного уровня попадают в одно поддерево, так что одной цепочки вершин сети достаточно для их декодирования. Далее (см. замечание 3.14) показано, как построить гораздо более точную вычислимую оценку $D'(H)$. Эксперименты автора показали, что при $\log_2 N \leq Q$ и $H \leq Q^2$

$$D'(H) \leq \frac{H^2}{2Q}; \tag{3.52}$$

при $H \geq 2Q$ такая оценка достаточно точна.

Замечание 3.9. Автор не рекомендовал бы использование почти оптимальных Q-сетей (несмотря на гарантированно высокую скорость декодирования) ввиду существенно большего по сравнению с Q-минимальными сетями расхода памяти; далее будет показано, что, как правило, Q-минимальные сети незначительно — на доли процента — хуже Q-оптимальных сетей в смысле времени декодирования. Лишь в случае принудительного ограничения длины кодового слова (как и было сделано автором в [7]) до $2\log_2 N$–$3\log_2 N$ битов использование почти оптимальных Q-сетей может оказаться приемлемым решением. ■

3.5.9. ОЦЕНКА МАКСИМАЛЬНОЙ МОЩНОСТИ МИНИМАЛЬНОЙ Q-СЕТИ

Если высота дерева не превышает Q, то мощность минимальной Q-сети на нем равна единице.

Так как высота удаляемого поддерева на каждом шаге (кроме, возможно, последнего) алгоритма построения минимальной Q-сети в точности равна Q, то количество удаляемых листьев принадлежит интервалу $[Q+1, 2^Q]$ (подразумевается, что кодовое дерево — полное), при этом к дереву добавляется один лист. Итак, мощность K минимальной Q-сети на полном кодовом дереве с N листьями находится в пределах

$$\left\lceil \frac{N}{2^Q - 1} \right\rceil \leq K \leq \left\lceil \frac{N}{Q} \right\rceil. \tag{3.53}$$

Верхняя оценка $\left\lceil \frac{N}{Q} \right\rceil$ значения K весьма пессимистична, но достижима, — например, если кодовое дерево на каждом уровне имеет ровно один лист (а на последнем уровне — два листа); в таком случае на каждом шаге построения минимальной Q-сети количество вершин дерева уменьшается ровно на Q.

Как отмечено в п. 3.1.2, максимальная длина кодового слова для кода Хаффмана (т. е. максимальная глубина дерева Хаффмана) не бывает очень большой; для других кодов также существуют достаточно жесткие оценки максимальной высоты кодового дерева.

Замечание 3.10. К сожалению, в общем случае это не помогает уменьшить верхнюю оценку максимальной мощности минимальной Q-сети, т. к. можно построить кодовое дерево (последовательной заменой наименее глубокого листа полным поддеревом высоты Q, имеющим $(Q+1)$ листьев, начиная с пустого дерева, состоящего только из корня) высоты не более

$$Q + 1 + \left\lceil \log_2 \left\lceil \frac{N}{Q} \right\rceil \right\rceil \tag{3.54}$$

и имеющее ровно N листьев, для которого минимальная Q-сеть будет иметь ровно $\left\lceil \frac{N}{Q} \right\rceil$ вершин. ■

Этот пример наглядно показывает, что **форма** кодового дерева существенно влияет на мощность минимальной Q-сети, поэтому с целью ограничения мощности минимальной Q-сети желательно избегать использования плохих кодовых деревьев и при необходимости преобразовывать их к эквивалентным деревьям, более пригодным для эффективного декодирования. Как указано в замечании 3.7, Q-минимальные деревья мало применимы на практике, однако, учитывая активное использование канонической формы построении Q-минимального дерева, следует ожидать, что мощность минимальной Q-сети на каноническом дереве мало отличается от мощности минимальной Q-сети на Q-минимальном дереве, если Q сравнимо (не более чем в 3-5 раз меньше) с глубиной дерева.

Так как обычно наибольшее количество символов кодируемого алфавита и (см. табл. 3.1) максимальная высота кодового дерева известны заранее, получение верхней оценки $K(N,H)$ мощности минимальной Q-сети на множестве канонических деревьев высоты не более H и имеющих не более N листьев представляет большой практический интерес, потому что позволяет оценить объем требуемой декодировщику памяти и время построения таблиц декодирования.

Можно заметить, что оценка мощности почти оптимальной Q-сети (см. теорему 3.7) применима и к Q-минимальным сетям, но является очень грубой, хотя и позволяет получить верхнюю асимптотическую оценку объема требуемой памяти: $O(N \log_2 N)$.

Автору не удалось получить достаточно точных оценок $K(N,H)$, представимых в виде формул, однако удалось построить вычислимую функцию $k(N,H)$ такую, что $k(N,H) \geq K(N,H)$; значение этой функции может быть получено для конкретных N и H и использовано как верхняя оценка мощности минимальной Q-сети.

3.5.10. ПОСТРОЕНИЕ ВЫЧИСЛИМОЙ ОЦЕНКИ МОЩНОСТИ Q-СЕТИ

Сначала опишем построение функции $k'(N,H)$, дающей верхнюю оценку мощности минимальной Q-сети на множестве деревьев высоты H и имеющих не более N листьев. С этой оценкой $k(N,H)$ может быть получено как максимум значений $k'(N,h)$ при $h = 1, \ldots, H$.

Так как выбор удаляемого поддерева достаточно произволен, для однозначной определенности можно полагать, что удаляется поддерево, содержащее самый правый наиболее глубокий лист дерева. При таком подходе (см. замечание 3.8) дерево, полученное

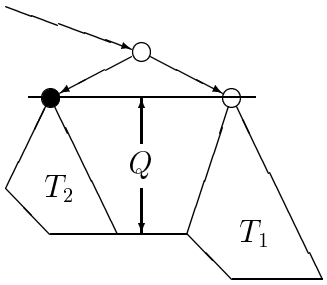

Рисунок 3.4. Удаляемое поддерево не захватывает вершину Q-сети

на очередном шаге, будет выглядеть как каноническое дерево, у которого поддерево с корнем в одной из вершин на пути к самому правому листу заменено на другое каноническое поддерево меньшей высоты.

Для простоты изложения будем полагать, что правое каноническое поддерево (содержащее вершины Q-сети) на каждом шаге содержит ровно одну вершину Q-сети (дальнейшие рассуждения легко обобщаются на произвольный случай, однако это приводит к весьма громоздким алгоритмам, которые не описываются в данной работе). Для обеспечения выполнения данного условия достаточно, чтобы $Q \geq \lceil \log_2 N \rceil$.

Действительно, рассмотрим ситуацию (см. рис. 3.4), при которой в *самый первый* раз корень поддерева (содержится в поддереве T_1), удаленного на предыдущем шаге, не попал в текущее удаляемое поддерево T_2.

Так как дерево каноническое, все листья исходного дерева, принадлежащие поддереву T_1, находятся не выше, чем листья, принадлежащие поддереву T_2; следовательно, все листья поддерева T_1 исходного дерева расположены как минимум Q уровней ниже корня поддерева, поэтому поддерево T_1 имеет не менее 2^Q листьев исходного дерева. Так как высота поддерева T_2 равна Q, то T_2 имеет не менее $(Q+1)$ листьев. Суммируя вышесказанное, получаем, что описанная ситуация возможно лишь тогда, когда исходное дерево имеет не менее

$$N \geq 2^Q + Q + 1 \tag{3.55}$$

листьев, поэтому выполнения условия $Q \geq \lceil \log_2 N \rceil$ достаточно, чтобы предотвратить появление нетривиального канонического поддерева из сетевых вершин в процессе построения минимальной Q-сети.

Допустим, нам известно, что высота дерева, полученного в процессе построения минимальной Q-сети, равна H, и оно содержит не более N вершин; кроме того, известна глубина R расположения корня поддерева, удаленного на предыдущем шаге, $H - Q + 1 \leq R \leq H$ (R не может быть равным $(H - Q)$, иначе корень удаленного поддерева не попадет в удаляемое поддерево).

Если $H \leq Q$, то мощность минимальной Q-сети равна единице, и в данном случае значение оценки является точным.

Если же $H > Q$, то необходимо рассмотреть все возможные значения L — глубины расположения самого высокого листа удаляемого поддерева. Если $R > L$, то удаляемое поддерево содержит не менее (рис. 3.5)

$$d = 2^{L-R'} + (H - L) \tag{3.56}$$

листьев, где $R' = H - Q$ — глубина корня удаляемого поддерева; если же $R \leq L$, то

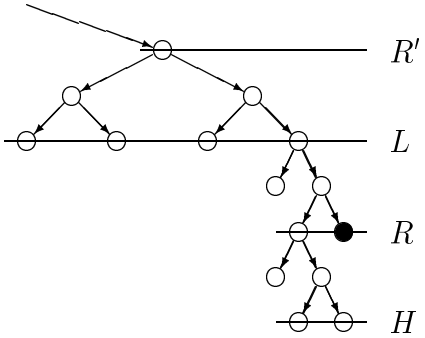

Рисунок 3.5. Удаляемое поддерево при $R > L$

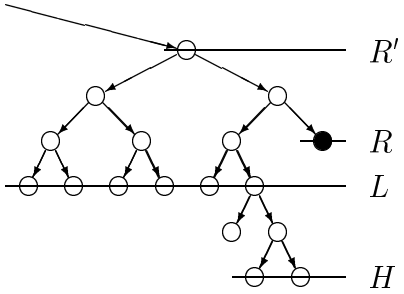

Рисунок 3.6. Удаляемое поддерево при $R \le L$

удаляемое поддерево имеет не менее (рис. 3.6)

$$d = (2^{L-R'} - 2^{L-R} + 1) + (H - L) \tag{3.57}$$

листьев; L может принимать значения в интервале $[R' + 1, H]$.

Высота H' дерева, полученное заменой текущего поддерева листовой вершиной, находится в интервале $[R', L]$; она не может быть меньше R', иначе корень удаляемого поддерева не имеет брата, что невозможно, если дерево полное, т. е. каждая промежуточная вершина имеет ровно двух сыновей; H' не может также превышать L, т. к. дерево — каноническое, и глубина листьев, расположенных в дереве левее, не может быть больше глубины более правых листьев. Кроме того, H' не может быть равной H.

Количество листьев N' нового дерева не превышает $(N - d + 1)$. Так как полное двоичное дерево высоты H' имеет не менее $(H' + 1)$ листьев, то N' должно быть больше $(H' + 1)$.

Итак, нам известна H' — (возможная) высота сокращенного дерева, наибольшее количество вершин дерева N' и глубина расположения нового корня дерева R'. Если k' — оценка максимума мощности минимальной Q-сети на деревьях с параметрами (N', H', Q'), то k — оценка максимума мощности минимальной Q-сети на деревьях с параметрами (N, H, Q) — может быть получена как максимум k' при $L \in [R' + 1, H]$, $H' \in [R', \min(L, H - 1)]$. Если в процессе перебора условие $N' > H' + 1$ нарушается, то такой вариант перебора следует опустить и не учитывать, т. к. это означает, что при заданном наибольшем количестве листьев текущее значение L невозможно.

Замечание 3.11. Перебирать значения H' при $H' \leq Q$ нет необходимости, т. к. $k' = 1$. ■

Замечание 3.12. Так как N' уменьшается с ростом L, то перебор по L можно прекращать, если L возрастает и N' становится меньшим $\min(H') = \max(Q, R')$. ■

Замечание 3.13. Если условие $N' > H'$ оказалось нарушенным при некотором H' и H' возрастает, перебирать остальные значения H' нет необходимости, т. к. условие $N' > H'$ не будет выполнено. ■

Нижеприведенная процедура вычисляет значение оценки k, как это описано выше (при первом вызове R полагается равным H, т. к. самый правый и глубокий лист дерева можно считать корнем поддерева, удаленного на предыдущем шаге).

```
static int Q;
static int maxk (int N, int H, int R) {
  int L, N1, H1, R1;
  int i, k, m;
  if (H <= Q || N <= H)
    return (1);
  R1 = H-Q;
  m = 1;
  for (L = R1 + 1; L <= H; ++L) {
    if (R > L)
      i = (1 << (L - R1)) + H - L;
    else
      i = (1 << (L - R1)) + H - L - (1 << (L - R)) + 1;
    N1 = N - i - 1;
    H1 = R1; if (H1 <= Q) H1 = Q + 1;
    if (N1 <= H1)
      break;
    i  = L;  if (i  >= H) i  = H - 1;
    for (; H1 <= i && N1 > H1; ++H1)
      if ((k = maxk (N1, H1, R1)) > m)
        m = k;
  }
  return (m+1);
}
```

Результаты вычисления оценок максимальной мощности Q-сети для типичных значений N и Q приведены в табл. 3.3, которой видно, что при $N \leq 256$ и $Q = 10$ мощность Q-сети заведомо не превышает 7, если все кодовые слова короче 35 битов; это позволяет предположить, что при $Q \sim H$ максимальная мощность Q-сети равна $O(Q)$.

Замечание 3.14. Аналогичным образом можно построить вычислимую оценку мощности почти оптимальной Q-сети с той разницей, что появление одной вершины Q-сети фактически приводит к появлению целой цепочки, причем необходимо просто перебрать все возможные уровни расположения листьев в текущем поддереве и все возможные уровни оставшихся листьев, отсекая случаи нарушения условия $N' \geq H'$. ■

Что же касается вычисления максимальной мощности Q-сети в общем случае (если

N	Q	2	3	4	5	6	7	8	9
$2^7 = 128$	7	8	9	10	13	16	21	26	32..32
	8	9	11	14	18	23	28	35	42..42
	9	10	13	17	22	28	35	43	51..51
$2^8 = 256$	8	9	10	11	13	17	21	25	30..32
	9	10	12	14	18	23	28	33	39..39
	10	11	14	18	23	28	35	41	48..48
$2^9 = 512$	9	10	11	12	14	17	21	25	30..32
	10	11	13	15	19	23	28	34	39..39
	11	12	15	18	23	29	35	41	47..47
$2^{10} = 1024$	10	11	12	13	15	18	22	26	30..32
	11	12	14	16	20	24	29	34	39..39
	12	13	16	19	24	29	35	42	48..48

Таблица 3.3. Оценки мощности Q-сети

Q не удовлетворяет условию (3.55)), то можно поступить аналогично тому, как было сделано выше, однако, учитывая возможность появления достаточно больших поддеревьев, состоящих из вершин Q-сети, требуется существенно более кропотливый анализ (который, к счастью, несколько упрощается ввиду того, что, в отличие от собственно дерева, про которое мало что известно, получающееся в процессе оценки поддерево вершин Q-сети известно точно).

Однако можно предложить несколько менее надежный, но более простой метод оценки, а именно: используя генератор распределений вероятностей[8] появлений символов, строить для порождаемых распределений коды Хаффмана (если необходимо, ограниченной длины), а для полученных кодов — минимальные Q-сети, вычисляя максимум мощностей получаемых Q-сетей. Кроме того, такие эксперименты совершенно необходимы для тестирования правильности реализации кодирования и декодирования при создании надежного программного обеспечения.

В частности, такие эксперименты проводились автором, при этом было перебрано (не автором, а ЭВМ) несколько сотен миллионов различных деревьев при различных параметрах кодировщика и декодировщика. При $Q \geq \log_2 N + 1$ подавляющее большинство оценок, приведенных в табл. 3.3, были получены, но при $Q = \lceil \log_2 N \rceil$ многие оценки никогда не получались: значения мощности сети были всегда меньше значений, указанных в табл. 3.3.

3.5.11. ОЦЕНКА ВРЕМЕНИ ТАБЛИЧНОГО ДЕКОДИРОВАНИЯ

Рассмотрим последовательность длин отрезков пути, ведущего от листа к корню, так что каждый отрезок заканчивается в вершине Q-сети, причем длина каждого отрезка максимальна, но не превышает Q. Другими словами, вершины Q-сети, порождающие это разбиение, — корни поддеревьев, которые необходимо пройти при декодировании символа, помечающего данный лист.

Пусть эта последовательность равна $(D_1, D_2, \ldots, D_{d(\alpha)})$, тогда выполняются условия $0 < D_i \leq Q < D_i + D_{i+1}$ и $\sum D_i = \ell(\alpha)$, где $\ell(\alpha)$ — длина кодового слова, соответ-

[8] Обычно дискретное распределение вероятностей появлений символов подчиняется законам Шварца, Ципфа, правилу 80–20 процентов [18], реже — законам Пуассона, Гаусса, нормальному.

ствующего символу α. В таком случае $d(\alpha)$ есть не что иное, как количество обращений к элементам таблиц, необходимых для декодирования символа, и время декодирования символа α будет пропорционально $d(\alpha)$.

Так как $D_i + D_{i+1} > Q$, то $D_i + D_{i+1} \geq Q + 1$, поэтому

$$d \leq 2 \left\lfloor \frac{\ell}{Q+1} \right\rfloor + 1, \tag{3.58}$$

и тогда среднее количество итераций, равное $\sum p(\alpha) d(\alpha)$, не превышает

$$\begin{aligned} d_{ave}(Q) &\leq \sum_{\alpha \in \Sigma} p(\alpha) \left(2 \left\lfloor \frac{\ell(\alpha)}{Q+1} \right\rfloor + 1 \right) \leq & (3.59) \\ &\leq 1 + 2 \sum_{\alpha \in \Sigma} \frac{p(\alpha)\ell(\alpha)}{Q+1} = & (3.60) \\ &= 1 + 2 \frac{\mathcal{C}}{Q+1}, & (3.61) \end{aligned}$$

где $\mathcal{C} = \sum p(\alpha)\ell(\alpha)$ — средняя стоимость кодирования.

Так как побитовое декодирование является частным случаем предложенного многотабличного декодирования при $Q = 1$ и требует ровно $\ell(\alpha)$ итераций для декодирования символа α, то среднее время побитового декодирования пропорционально

$$d_{ave}(1) = \sum_{\alpha \in \Sigma} p(\alpha)\ell(\alpha) = \mathcal{C}, \tag{3.62}$$

поэтому декодирование с использованием Q-сети как минимум в $\frac{Q+1}{2}$ быстрее побитового декодирования.

Замечание 3.15. На самом деле оценка (3.61) весьма пессимистична, т. к. совершенно не учитывает, что вероятность появления символа $p(\alpha)$ быстро убывает с ростом длины его кода $\ell(\alpha)$, а вероятность того, что для всех символов

$$D_1 + D_2 = D_3 + D_4 = \ldots = Q + 1, \tag{3.63}$$

очень мала. На самом деле средняя скорость декодирования в основном определяется количеством итераций, необходимых для декодирования нескольких наиболее часто встречающихся символов. Длины кодов таких символов не превышают $\lceil \log_2 N \rceil$ или $(\lceil \log_2 N \rceil + 1)$, поэтому при $Q \geq \lceil \log_2 N \rceil$ среднее время декодирования примерно равно

$$d_{ave}(Q) \approx \sum_{\alpha \in \Sigma} p(\alpha) \left\lceil \frac{\ell(\alpha)}{Q} \right\rceil < 1 + \frac{\mathcal{C}}{Q}. \tag{3.64}$$

■

Если максимальная длина кодового слова не превышает $2Q$ (а это достаточно часто случается при $Q \geq \lceil \log_2 N \rceil$), то каждый символ *гарантированно* декодируется в точности за

$$d(\alpha) = \left\lceil \frac{\ell(\alpha)}{Q} \right\rceil \tag{3.65}$$

итераций, и тогда декодирование почти в Q раз быстрее побитового декодирования, т. к. применима оценка (3.64). Действительно, глубина любой вершины Q-сети не превышает Q, поэтому любой символ декодируется за одну итерацию при $d(\alpha) \leq Q$ и за две итерации при $d(\alpha) > Q$.

3.5.12. (R,Q)-СЕТИ НА ДЕРЕВЬЯХ

Как уже отмечалось (см. замечание 3.15), время декодирования в основном определяется временем декодирования нескольких первых наиболее вероятных символов. Вместо того, чтобы декодировать все символы, применяя таблицы одинакового размера, можно использовать (R,Q)-сети ($R \geq Q$), отличающиеся от Q-сетей тем, что расстояние до ближайшей вышестоящей вершины сети может быть равным R, если эта вершина — корень дерева.

Другими словами, это означает, что все листья дерева глубины не более R будут доступны за одно обращение к таблице, что позволит улучшить — иногда существенно — скорость декодирования за счет незначительного увеличения объема требуемой памяти.

Нетрудно распространить все полученные результаты на (R,Q)-сети; т. к. корень дерева и деревья высоты не более Q всегда рассматривались как выделенный случай, в соответствующих местах достаточно заменить Q на R.

Поскольку Q-сети — частный случай (R,Q)-сетей при $R = Q$, все утверждения относительно (R,Q)-сетей немедленно распространяются на Q-сети.

3.5.13. ВЫБОР РАЗМЕРОВ ТАБЛИЦ

Желательно, чтобы наиболее часто встречающиеся символы декодировались за одну итерацию. Поскольку длины кодов таких символов (см. замечание 3.15), как правило, не превышают $\lceil \log_2 N \rceil$ или $(\lceil \log_2 N \rceil + 1)$, то размер корневой таблицы должен быть не меньше $\log_2 N$, т. е. параметр R при использовании (R,Q)-сетей нужно выбирать равным

$$R = \lceil \log_2 N \rceil + \varepsilon, \tag{3.66}$$

где ε — небольшое неотрицательное целое число (от 0 до 2). Выбор больших значений R нежелателен ввиду большого расхода памяти, который составляет — с учетом N слов дополнительной памяти для построения таблиц — $\left(N + 2^R + (K-1)2^Q\right)$ слов, а т. к. время инициализации структур данных тоже почти прямо пропорционально объему требуемой памяти, то выбор чрезмерно большого R может привести к тому, что время инициализации структур данных превысит время собственно декодирования.

Аналогично, параметр Q не следует выбирать слишком маленьким, чтобы не ухудшить среднее время декодирования, или слишком большим. Автору представляется, что наилучшими являются значения

$$Q = \lceil \log_2 N \rceil - \varepsilon, \tag{3.67}$$

где ε — небольшое неотрицательное целое число (от 0 до 2).

Эмпирические исследования для Q-сетей отношения $r = \frac{d_{ave}}{d_{opt}}$, где d_{ave} — среднее время декодирования при использовании Q-сети, а $d_{opt} = \sum p(\alpha) \left\lceil \frac{d(\alpha)}{Q} \right\rceil$ — минимальное возможное значение d_{ave}, показали, что при $Q = \lceil \log_2 N \rceil$ в среднем $r = 1.001$ при $N = 128 \ldots 1024$, а максимальное значение r не превышало 1.005; при $Q = \lceil \log_2 N \rceil + 2$ в среднем $r = 1.0004$, а максимальное значение r не превышало 1.0013.

Таким образом, в большинстве типичных случаев средняя скорость при использовании Q-минимальных сетей очень мало отличается от скорости, достижимой при использовании Q-оптимальных сетей. Тем не менее, при использовании описанных алгоритмов другими разработчиками автор рекомендовал бы им самостоятельно провести аналогичные исследования, т. к. типичные случаи (мощность алфавита, распределе-

ние символов и т. д.), исследованные автором, в некоторых случаях могут не быть таковыми.

3.5.14. ОРГАНИЗАЦИЯ БИТОВОГО ВВОДА

Как правило, работа с отдельными битами эффективно реализуется только на аппаратном уровне; при использовании же архитектур общего назначения необходимо помнить, что единицей обращения к памяти является байт — последовательность из 8 битов, а на самом деле при одном обращении к шине выбирается одновременно от 32 до 256 битов, поэтому пословная обработка данных (по 16, 32 или 64 бита одновременно) существенно быстрее побайтовой. Кроме того, если данные хранятся в регистре процессора, а не в основной памяти, то обращение к ним быстрее в 2–6 раз.

Для декодирования префиксных кодов необходимо реализовать три процедуры:

- `lookup(q)` — прочитать очередные `q` битов входного потока и вернуть целое число в интервале $[0, 2^q - 1]$;
- `remove(n)` — удалить `n` битов из входного потока;
- `initialize()` — подготовится к битовому вводу.

Для увеличения эффективности будем хранить в переменной `buff` несколько первых битов входного потока (не менее $B+1$ при $B = 0.5W$, где W — ширина слова в битах), а в переменной `bits` — счетчик количества битов в переменной `buff`; при аккуратной реализации эти переменные должны быть размещены на регистрах процессора.

Процедура `initialize()` тривиальна — необходимо прочитать W первых битов ($W/8$ байтов) закодированного потока и записать их в переменную `buff`, а переменную `bits` установить равной W.

Процедура `lookup(q)` ($q \le B$) возвращает значение переменной `buff`, сдвинутое вправо на $(W - q)$ битов, и может быть реализована одной-двумя командами (в зависимости от архитектуры). Важно отметить, что вычисление значения $(W - q)$, если q — константа, не требует вычислений во время исполнения; кроме того, на многих процессорах сдвиг на фиксированное число битов всегда или при некоторых выделенных значениях (как правило, кратных 8 и/или 16) гораздо эффективнее, чем сдвиг на переменное (определяемое во время исполнения) количество битов; эти соображения могут быть учтены при выборе Q и R.

Процедура `remove(n)` ($n \le B$) сдвигает содержимое переменной `buff` влево на `n` битов (так что `n` наиболее значимых битов теряется) и уменьшает значение переменной `bits` на `n`; если оно становится меньшим или равным B, то из входного потока дочитывается B битов ($B/8$ байтов). Как нетрудно заметить, данная процедура реализуется 7–10 командами, из которых при $bits - n \ge B$ исполняется лишь четыре (сдвиг, вычитание, сравнение, и условный переход) с возможным уменьшением до трех (сдвиг, вычитание, условный переход), если в переменной `bits` хранить количество битов в `buff`, уменьшенное на B, так что сравнение с нулем будет произведено процессором автоматически при уменьшении значения переменной `bits`.

Если использовать побайтовый ввод-вывод, то можно потребовать, чтобы в переменной `buff` всегда хранилось не менее $B' = W - 7$ битов, и если после обращения к процедуре `remove(n)` значение `bits` стало меньше $(W - 7) \iff bits \le (W - 8)$, то в переменную `buff` необходимо дочитать один или несколько следующих байтов входного потока, пока `bits` не станет больше либо равно $(W - 8)$.

Таким образом, одна итерация декодирования символа потребует одного обращения к процедуре `lookup(Q)` (или, при использовании (R, Q)-сетей, `lookup(R)` на первой итерации) и последующего обращения к процедуре `remove(n)`, где `n` — записанная

в поле `length` соответствующего элемента таблицы длина пути от корня поддерева, соответствующего таблице, до декодированного листа дерева или другой сетевой вершины. Затем, если достигнут лист дерева, то декодирование символа завершено, иначе необходимо продолжить декодирование, используя таблицу, соответствующую декодированной вершине, принадлежащей Q-сети.

Применение описанных алгоритмов позволяет добиться исключительно высокой скорости декодирования — до 10–20 млн. символов в секунду — на современных процессорах общего назначения, работающих на частоте 100–200 МГц, т. к. одна итерация выполняется за 10–15 тактов процессора.

3.5.15. ОРГАНИЗАЦИЯ СТРУКТУР ДАННЫХ

При использовании канонических кодов можно не хранить кодовое дерево в явном виде (воспользовавшись методами, описанными в замечаниях 3.6 и 3.8) и тем самым уменьшить время инициализации структур данных и объем требуемой памяти.

Поскольку максимальное количество таблиц и размер каждой таблицы могут быть определены заранее, можно расположить таблицы последовательно друг за другом.

Вместо хранения в таблицах явных ссылок на вершины дерева и т. д. можно упаковать в одно слово значение поля `length` (используя $\lceil \log_2 Q \rceil$ битов), номер символа или таблицы (используя $\lceil \log_2 \max(N, K) \rceil$ битов, где N — максимальное количество символов алфавита, а K — наибольший возможный номер таблицы); еще один бит необходим для того, чтобы отличать номера таблиц от номеров символов.

При использовании такого подхода расход памяти и время инициализации структур данных равны $O(N + K2^Q)$. С учетом того (см. п. 3.1.2), что $K \ll \ell_{max} \sim \log_2 N$ и $2^Q \sim N$, время инициализации и объем требуемой памяти не превышают $O(N \log_2 N)$.

3.6. ВЫВОДЫ

Предложенный автором алгоритм построения кода Хаффмана для дискретных распределений символов при ограниченной длине кодируемого сообщения (менее 2^{32} символов) и достаточно большой мощности кодируемого алфавита (более 100 символов) заметно превосходит по производительности все известные благодаря эффективному алгоритму сортировки символов в сочетании с линейным алгоритмом построения собственно дерева Хаффмана для отсортированных по частоте символов. Это позволяет в 2–4 раз ускорить построение кода Хаффмана по сравнению с алгоритмом Катаяйнена и Моффата [109] и, соответственно, в 6–20 раз — по сравнению с традиционным подходом [20, 39, 166].

Созданные автором эффективные методы декодирования префиксных кодов вообще и кодов Хаффмана в частности путем использования разработанных для этой цели Q-сетей позволяют ускорить декодирование почти в $\mathcal{E}$ раз ($\mathcal{E}$ — энтропия закодированного сообщения, обычно равная 5–8) по сравнению с побитовым декодированием, т. к. подавляющее большинство символов декодируется за одну итерацию; другие известные алгоритмы позволяют увеличить скорость лишь на 30–50% при $\mathcal{E} \approx 5$ бит на символ (т. е. скорость декодирования предложенного метода выше в 2–4 раза, и с ростом стоимости кодирования разрыв увеличивается).

Построенные автором оценки объема требуемой для декодирования памяти (как в явном виде, так и в виде программ для ЭВМ) показывают, что при использовании канонических кодов объем требуемой памяти невелик и сравним с требованиями других известных алгоритмов декодирования.

Следует отметить, что при выводе оценок объема требуемой памяти, в свою очередь, были существенно использованы свойства кодов Хаффмана, доказательства которых, полученные автором, также приводятся в данной работе.

Метод кодирования кодовых деревьев, предложенный автором, позволяет добиться высокого качества сжатия последовательности длин кодовых слов (обычно менее двух битов на слово) и тем самым существенно уменьшить стоимость передачи декодировщику информации о способе кодирования.

Часть III

Алгоритмы словарного сжатия

Глава 4

СЖАТИЕ ТЕКСТОВ НА ЕСТЕСТВЕННЫХ ЯЗЫКАХ

Стремительный рост возможностей современных компьютеров и расширение сферы их применения неквалифицированными пользователями приводит к тому, что практически все пакеты являются высокоинтерактивными и включают в себя развитую систему контекстных подсказок, что позволяет пользователю на любом этапе работы с программой по нажатию выделенной клавиши получить короткую подсказку о том, что он может делать в настоящий момент; при желании пользователь может получить более детальное и развернутое описание. Тексты подсказок занимают существенную часть объема всей системы (их размер нередко составляет несколько десятков мегабайтов), поэтому очень желательно сжатие данных.

К сожалению, универсальные методы мало применимы для этой цели, и основная проблема заключается в необходимости обеспечения **почти произвольного доступа к сжатым данным**. Методы, дающее высокое качество сжатия (словарные, статистического моделирования), допускают только последовательный доступ к данным, а разбиение данных на короткие блоки и сжатие каждого блока в отдельности приводит к существенному понижению качества сжатия. С другой стороны, методы сжатия, допускающие произвольный доступ к сжатым данным (посимвольное кодирование префиксными кодами, например, по Хаффману), обладают невысоким качеством сжатия.

Можно воспользоваться тем, что сжимаемые данные заведомо представляют собой текст на естественном языке (возможно, дополненный информацией о правилах форматирования, шрифтах, перекрестных ссылках на другие участки текста, и т. д.). Текст можно рассматривать как последовательность слов, перемежающихся последовательностями разделителей. Предлагаемый метод сжатия основан на замене слов, встретившихся в тексте, их номерами в словаре. С первого взгляда неочевидно, что при таком подходе можно получить хорошее качество сжатия, т. к. размер словаря обычно составляет 8–16 тыс. слов при средней длине слова 3–5 символов, поэтому равномерное кодирование номеров слов в словаре позволит получить качество сжатия в пределах 40–60%, что мало отличается от качества сжатия при посимвольном кодировании кодами Хаффмана [105]. Если добавить к этому размер собственно словаря, то хранение текста в таком виде теряет всякий смысл.

Все известные форматы хранения гипертекстов (Norton Guide, Tech Help, Microsoft Help и др.) обычно используют смешанную стратегию: наиболее часто используемые слова хранятся в словаре, а для кодирования остальных символов используется кодирование по Хаффману, что позволяет получить качество сжатия порядка 50%.

Предлагаемый алгоритм сжатия чисто текстовой информации основан на эмпирическом *законе Ципфа* [199, 200]: частота появления слов, отсортированных по частоте

употребления, обратно пропорциональна их номерам; другими словами,

$$p_k \approx \frac{1}{k}\frac{1}{\gamma(N)}, \tag{4.1}$$

где $(N-1)$ –количество различных слов, использованных в тексте, а

$$\gamma(N) = \sum_{k=1}^{N-1} \frac{1}{k}; \tag{4.2}$$

невзирая на существенные различия в грамматике и синтаксисе, закон Ципфа выполняется для очень большого количества языков, — английского, французского, немецкого, испанского, русского, китайского и многих других.

Если текст рассматривать как последовательность слов, являющихся символами некоторого алфавита (фактически частотного словаря), можно применить посимвольное кодирование для символов этого алфавита. Таким образом, сжатый текст представляет собой пару из словаря и собственно закодированного текста, состоящего из последовательности номеров слов в словаре.

Достоинства такого подхода очевидны:

- возможность декодирования произвольного участка текста (а не всего текста целиком);
- локальные изменения в тексте вызывают локальные изменения в закодированном представлении и не требуют перекодирования всего текста целиком;
- очень высокая скорость сжатия и/или декодирования текста, практически прямо пропорциональная количеству слов в тексте;
- существенная часть словаря зависит только от языка и может храниться в единственном экземпляре отдельно от закодированных текстов.

Такие методы кодирования текста давно известны [39, 99, 106, 139, 143, 148, 149, 160, 192, 193, 194] (большинство современных информационно-поисковых систем [143, 193] — AltaVista, InfoSeek, Yahoo, NZDL, NCSTRL и др. — хранят тексты, заменяя слова их номерами в словаре), однако предложенные автором методы решения ряда ключевых проблем не имеют аналогов и позволяют существенно улучшить качество и скорость кодирования и декодирования и уменьшить объем требуемой при декодировании памяти.

4.1. СТОИМОСТЬ КОДИРОВАНИЯ ТЕКСТА

Для начала стоит выяснить, насколько оптимальное кодирование натуральных чисел с распределением Ципфа лучше равномерного, чтобы принять решение о способе кодирования номеров слов в словаре. Далее будет часто использоваться известная формула из теории асимптотических функций (в частности, она приводится в [16] п. 1.2.11.2):

$$\sum_{k=1}^{N-1} f(k) = \int_1^N f(x)dx - \frac{1}{2}f(N) + \ldots + \frac{(-1)^m B_m}{m!} f^{(m-1)}(N) + R_m(N) + C_f, \tag{4.3}$$

где $R_m(N)$ — остаточный член ряда, B_m — m-е число Бернулли, C_f — константа, зависящая только от функции f. Прямым применением этой формулы можно получить известную асимптотическую оценку гармонического ряда:

$$\gamma(N) = \sum_{k=1}^{N-1} \frac{1}{k} = \ln N + C_\gamma - \frac{1}{2N} - R_\gamma(N), \tag{4.4}$$

где

$$0 \le R_\gamma(N) \le \frac{1}{12N^2}, \tag{4.5}$$

а $C_\gamma = 0.57721566\ldots$ — постоянная Эйлера.

Теорема 4.1. При $N \ge 2048$ оптимальная стоимость кодирования $\mathcal{C}(N)$ целых чисел из интервала $[1, N)$, вероятности появления которых подчиняются закону Ципфа, равна[1]

$$\mathcal{E}(N) = \frac{n}{2} + \log_2 n - \delta, \tag{4.6}$$

где δ удовлетворяет неравенству

$$0.82 < \delta < 0.95, \tag{4.7}$$

а $n = \log_2 N$ — стоимость равномерного кодирования; другими словами, оптимальное кодирование в полтора-два раза лучше равномерного.

Доказательство. Стоимость кодирования $\mathcal{C}(N)$ не может быть ниже энтропии

$$\mathcal{E}(N) = \sum_{k=1}^{N-1} -p_k \log_2 p_k \tag{4.8}$$

(см. п. 2.5.1) и при соответствующем кодировании может быть сколь угодно близка к энтропии.

В нашем случае

$$p_k = \frac{1}{k\gamma(N)}, \tag{4.9}$$

поэтому

$$\mathcal{E}(N) = \sum_{k=1}^{N-1} -\frac{1}{k\gamma(N)} \log_2 \frac{1}{k\gamma(N)} \tag{4.10}$$

$$= \frac{1}{\gamma(N)} \sum_{k=1}^{N-1} \frac{1}{k} (\log_2 \gamma(N) + \log_2 k) = \tag{4.11}$$

$$= \log_2 \gamma(N) + \frac{1}{\gamma(N)} \sum_{k=1}^{N-1} \frac{\log_2 k}{k}. \tag{4.12}$$

Согласно (4.3), ряд

$$L(N) = \sum_{k=1}^{N-1} \frac{\log_2 k}{k} \tag{4.13}$$

асимптотически сходится к

$$L(N) = \frac{\ln^2 N}{2 \ln 2} + C_L - \frac{\log_2 N}{2N} - R_L(N), \tag{4.14}$$

где $C_L = -0.105051\ldots$, а

$$0 \le R_L(N) \le \frac{\log_2 N}{6N^2}. \tag{4.15}$$

При $N \to \infty$

$$L(N) \to L_\infty(N) = \frac{\ln^2 N}{2 \ln 2} + C_L, \tag{4.16}$$

[1] Оценка $\mathcal{E}(N) = 0.5n + \log_2 n + O(1)$ хорошо известна [39, 166], однако поскольку в дальнейшем потребуются оценки $\mathcal{E}$ с точностью до 0.01, появление аддитивного $O(1)$ недопустимо.

а

$$\gamma(N) \to \gamma_\infty(N) = \ln N + C_\gamma, \tag{4.17}$$

поэтому

$$\mathcal{E}(N) \;=\; \log_2 \gamma(N) + \frac{L(N)}{\gamma(N)}, \tag{4.18}$$

$$\mathcal{E}(N) \;\to\; \mathcal{E}_\infty(N) = \frac{\log_2 N}{2} + \log_2 \log_2 N - C_\mathcal{E}, \tag{4.19}$$

где

$$C_\mathcal{E} = \frac{C_\gamma - \log_2 \ln 2}{2 \ln 2} = 0.945139\ldots \tag{4.20}$$

Обозначив стоимость равномерного кодирования через $n = \log_2 N$, получаем, что при $n \to \infty$

$$\mathcal{E}(N) \to \frac{n}{2} + \log_2 n - C_\mathcal{E}; \tag{4.21}$$

при $11 \le n \le 14$ ($2048 \le N \le 16384$) хорошим приближением является формула

$$\mathcal{E}(N) \approx \frac{n}{2} + \log_2 n - 0.836, \tag{4.22}$$

причем приближенное и точное значения $\mathcal{E}$ отличаются менее чем на 0.013, т. е. относительная ошибка меньше 0.15%. ■

4.2. КОДИРОВАНИЕ ЦЕЛЫХ ЧИСЕЛ С РАСПРЕДЕЛЕНИЕМ ЦИПФА

Поскольку количество слов в словаре обычно достигает нескольких десятков тысяч (а иногда и миллионов [194]), построение близких к оптимальному методов кодирования номеров слов не тривиально.

Адаптивные методы кодирования неприменимы, т. к. малый размер участков кодирования (100–200 слов) на порядок меньше размера словаря и вероятность появления одного и того же слова на участке кодирования очень мала. Кроме того, перед кодированием или декодированием каждого участка необходимо восстанавливать структуры данных в начальное состояние, что при большом размере алфавита на несколько порядков замедлит сжатие и декодирование. Наконец, структуры данных адаптивных алгоритмов сжатия требуют много памяти — 8–16 байтов на символ алфавита, поэтому размер словаря вместе с такими данными может превысить размер сжатого текста.

Таким образом, кодирование должно быть статическим. Практически безызбыточное арифметическое кодирование (см. п. 2.5.2) требует хранения таблицы частот слов, что увеличивает размер словаря в два-три раза и заметно снижает качество сжатия. Кроме того, алгоритмы арифметического кодирования обладают очень низкой скоростью кодирования и, главное, декодирования, а почти все известные способы их реализации запатентованы или защищены авторским правом (copyright) и не могут использоваться в коммерческих продуктах без приобретения соответствующих весьма дорогостоящих лицензий (авторы [143, 148, 149, 193, 194] приводят к таким же выводам).

Кодирование по статическому методу Хаффмана (см. главу 3) наиболее приемлемо с практической точки зрения; оно требует хранения длин кодовых слов, которых достаточно для построения и восстановления собственно кодовых слов.

Заметим, что слова в словаре отсортированы по частотам, поэтому последовательность длин кодовых слов будет неубывающей. Следовательно, все слова можно разбить

на группы по длине кодового слова, и хранить длины кодовых слов для всех слов словаря нет необходимости, т. к. последовательность пар *(длина кодового слова, количество слов)* позволяет однозначно восстановить длины кодовых слов и, следовательно, собственно кодовые слова. Согласно теореме 3.2, длина кодового слова не будет превышать 32 битов и последовательность пар будет короче 32, если текст содержит менее $F_{32+2} \leq 5.7$ млн. слов (F_k — k-е число Фибоначчи).

К числу недостатков такого подхода (описанного в [39, 193, 194] и применяемого подавляющим числом информационно-поисковых систем) следует отнести то, что словарь нельзя дополнять новыми словами, т. е. сжатый текст невозможно модифицировать, если использовались слова, отсутствующие в словаре. В качестве возможного решения можно предложить изначально завести в словаре место под достаточно большое количество пустых слов, которое будет постепенно заполняться новыми словами, или размещать новые слова непосредственно в закодированном тексте, что ухудшает качество сжатия [143, 148].

Друга проблема заключается в том, что если кодируемый текст очень длинный (более F_{34} слов) и мощность словаря велика (сотни тысяч слов), то длина кодового слова кода Хаффмана может быть очень большой и достигать $\lfloor 1.44 \log_2 W \rfloor$ (см. теорему 3.3), где W — длина текста в словах. Это вынуждает использовать оптимальные префиксные коды ограниченной длины [110, 111, 123, 173, 174, 175, 176], построение которых весьма нетривиально (отметим, что основным мотивом для разработки методов построения кодов ограниченной длины послужила как раз проблема кодирования номеров слов в очень большом словаре — см. процитированные работы).

Однако тот факт, что распределение частот номеров слов в словаре подчиняется закону Ципфа, позволяет построить компактную и простую схему кодирования, по качеству сжатия очень близкую к оптимальному.

Разобьем все слова на группы размером 2^m ($m = 0, \ldots, \lfloor \log_2(N-1) \rfloor$), т. е. в первую группу попадет самое часто употребляемое слово, во вторую — второе и третье, в третью — 4, 5, 6 и 7-е и т. д. Построим коды Хаффмана для частот q_m, где

$$q_m = \sum_{k=2^m}^{2^{m+1}-1} p_k = \frac{1}{\gamma(N)} \sum_{k=2^m}^{2^{m+1}-1} \frac{1}{k}, \tag{4.23}$$

и будем кодировать k-е слово парой *(номер старшего бита k, оставшиеся биты k)* (для кодирования оставшихся битов k используется равномерное кодирование), т. е. парой

$$\big(m : \ell_{m+1}, \quad (k - 2^m) : m\big), \tag{4.24}$$

где после двоеточия указано количество битов, необходимых для кодирования элемента пары, $m = \lfloor \log_2 k \rfloor$, а ℓ_m — стоимость кодирования символа с частотой q_m; при целом $n = \log_2 N$ стоимость такого кодирования равна

$$\mathcal{C}(N) = \sum_{m=0}^{n-1} \sum_{k=2^m}^{2^{m+1}-1} p_k(m + \ell_m) = \sum_{m=0}^{n-1} q_m(m + \ell_m). \tag{4.25}$$

Как известно, для статического кодирования по методу Хаффмана

$$\sum_{m=0}^{n-1} q_m \ell_m = \varepsilon + \sum_{m=0}^{n-1} -q_m \log_2 q_m, \tag{4.26}$$

где $0 \leq \varepsilon < 1$ — избыточность кодирования, вызванная тем, что кодовые слова имеют целые длины.

Теорема 4.2. Избыточность ε, вызванная применением кода Хаффмана, при $N \geq 2048$ ($n \geq 11$) меньше 0.1235.

Доказательство. Согласно (3.1),

$$\varepsilon < 0.0861 + (q_{max} - q_{min}), \tag{4.27}$$

где q_{max} и q_{min} — наибольшее и наименьшее q_m соответственно.

Обозначим

$$s_m = \sum_{k=2^m}^{2^{m+1}-1} p_k, \tag{4.28}$$

тогда

$$s_m = \gamma(2^{m+1}) - \gamma(2^m) = \ln 2 + d_m, \tag{4.29}$$

где

$$d_m = \sum_{k=1}^{\infty} \frac{B_k(2^k-1)}{k2^k} \frac{1}{2^{mk}} = \frac{1}{2^{m+2}} + \frac{1}{4^{m+2}} + \ldots \tag{4.30}$$

Так как d_m и s_m монотонно убывают с ростом m,

$$s_{max} = s_0 = 1, \tag{4.31}$$

$$s_{min} = s_{n-1} > \ln 2. \tag{4.32}$$

Поскольку $s_m = q_m \gamma(N)$,

$$\varepsilon < 0.0861 + \frac{1 - \ln 2}{\gamma(N)}, \tag{4.33}$$

поэтому $\varepsilon < 0.1235$ при $N \geq 2048$. ■

Использование других методов кодирования, в частности арифметического, позволит сделать ε сколь угодно малым, однако в этом нет необходимости, т. к. *относительная* избыточность кода Хаффмана для группового кодирования при $N \geq 2048$ меньше

$$\frac{\varepsilon}{\mathcal{E}(N)} < 0.0152, \tag{4.34}$$

т. е. меньше 1.5%, что с практической точки зрения не существенно. Кроме того, распределение q_m почти равномерное, поэтому оценка избыточности кода Хаффмана (3.1) является очень грубой и в нашем случае завышена на порядок. Таким образом, использование более сложных алгоритмов кодирования представляется не оправданным, поскольку код Хаффмана почти оптимален.

Теорема 4.3. Избыточность группового кодирования

$$\mathcal{R}(N) = \mathcal{C}(N) - \mathcal{E}(N) \tag{4.35}$$

не превышает

$$\mathcal{R}(N) \leq \varepsilon + 0.028766 + \frac{1.136520}{\log_2 N}. \tag{4.36}$$

Доказательство. При использовании кода Хаффмана

$$\mathcal{C} = \sum_{m=0}^{n-1} q_m(m + \ell_m) = \sum_{m=0}^{n-1} \frac{s_m}{\gamma(N)}(m + \varepsilon - \log_2 s_m), \tag{4.37}$$

и, т. к. $\sum s_m = \gamma(N)$ и $s_0 = 1$, то

$$\mathcal{C} = \varepsilon + \log_2 \gamma(N) + \frac{1}{\gamma(N)} \sum_{m=1}^{n-1} s_m(m - \log_2 s_m). \tag{4.38}$$

Вычислим

$$M(N) = \sum_{m=1}^{n-1} m s_m = \tag{4.39}$$

$$= \sum_{m=1}^{n-1} m(\ln 2 + d_m) = \tag{4.40}$$

$$= \frac{n(n-1)}{2} \ln 2 + \sum_{m=1}^{n-1} m d_m. \tag{4.41}$$

Для вычисления $\sum m d_m$ необходимо вычислить сумму ряда $\sum m x^{m-1}$, что легко сделать дифференцированием тождества

$$\sum_{m=1}^{n-1} x^m = \frac{x^n - 1}{x - 1}, \tag{4.42}$$

однако мы не будем пользоваться точным выражением, т. к. при $|x| \le \frac{1}{2}$ этот ряд очень быстро сходится, и, соответственно, ряд $\sum m d_m$ сходится к $0.52727384\ldots$, не превышая этого значения.

Таким образом,

$$M(N) = \sum_{m=1}^{n-1} m s_m \to \frac{n(n-1)}{2} \ln 2 + 0.52727384\ldots \tag{4.43}$$

Поскольку

$$(1 + x) \ln(1 + x) \ge x, \tag{4.44}$$

то при $x > 0$

$$(x + y) \ln(x + y) \ge y + (x + y) \ln x, \tag{4.45}$$

и поэтому

$$s_m \ln s_m = (\ln 2 + d_m) \ln(\ln 2 + d_m) \ge \tag{4.46}$$

$$\ge s_m \ln \ln 2 + d_m = \tag{4.47}$$

$$= s_m \ln \ln 2 + (s_m - \ln 2) = \tag{4.48}$$

$$= s_m(\ln \ln 2 + 1) - \ln 2; \tag{4.49}$$

тогда

$$Q(N) = \sum_{m=1}^{n-1} s_m \log_2 s_m \ge \tag{4.50}$$

$$\ge \frac{\ln \ln 2 + 1}{\ln 2} \sum_{m=1}^{n-1} s_m - \sum_{m=1}^{n-1} 1 = \tag{4.51}$$

$$= \frac{\ln \ln 2 + 1}{\ln 2} (\gamma(N) - 1) - (n - 1). \tag{4.52}$$

Так как

$$M(N) \le \frac{n(n-1)}{2} \ln 2 + 0.52727384 \tag{4.53}$$

то

$$M(N) - Q(N) \le \frac{n(n-1)}{2} \ln 2 - \gamma(N) \frac{\ln \ln 2 + 1}{\ln 2} + n + 0.441202. \tag{4.54}$$

Избыточность кодирования $\mathcal{R}$ равна

$$\mathcal{R}(N) = \mathcal{C}(N) - \mathcal{E}(N) = \tag{4.55}$$

$$= \varepsilon + \left(\log_2 \gamma(N) + \frac{M(N) - Q(N)}{\gamma(N)}\right) - \left(\log_2 \gamma(N) + \frac{L(N)}{\gamma(N)}\right) = \quad (4.56)$$

$$= \varepsilon + \frac{M(N) - Q(N) - L(N)}{\gamma(N)}. \quad (4.57)$$

Так как

$$L(N) \geq \frac{\ln^2 N}{2 \ln 2} - \frac{\ln 2}{2}, \quad (4.58)$$

то

$$M(N) - Q(N) - L(N) \leq -\gamma(N) \frac{\ln \ln 2 + 1}{\ln 2} + n - \frac{n \ln 2}{2} + 0.787776, \quad (4.59)$$

а поскольку

$$\frac{1}{\gamma(N)} \leq \frac{1}{n \ln 2}, \quad (4.60)$$

то

$$\frac{M(T) - Q(T) - L(T)}{\gamma(N)} \leq -\log_2 \ln 2 - \frac{1}{\ln 2} + \frac{1}{\ln 2} - \frac{1}{2} + \frac{1.136520}{n}. \quad (4.61)$$

Таким образом, избыточность группового кодирования заведомо не превышает

$$\mathcal{R}(N) \leq \varepsilon + 0.028766 + \frac{1.136520}{n}, \quad (4.62)$$

и относительная избыточность при $n \geq 11$ не превышает 4%. ■

Более аккуратные вычисления (которые не приводятся ввиду громоздкости промежуточных вычислений), учитывающие члены более высоких порядков в соответствующих асимптотических формулах, позволяют получить оценку

$$\mathcal{R}(N) \leq \varepsilon - \log_2 \ln 2 - \frac{1}{2} = \varepsilon + 0.028766, \quad (4.63)$$

поэтому

$$\mathcal{R}(N) < 0.16, \quad (4.64)$$

а поскольку $\mathcal{C} \geq 8.14$ при $N \geq 2048$, избыточность предложенного способа кодирования меньше 1.8%.

Замечание 4.1. Предложенное групповое кодирование близко к оптимальному не только для распределения Ципфа (p_k пропорционально $\frac{1}{k}$), но и для более общего распределения Шварца [160] с параметром θ (p_k пропорционально $k^{\theta-1}$), которое часто возникает на практике: распределение Ципфа ($\theta = 0$), равномерное распределение ($\theta = 1$); известное правило 80 и 20 процентов также приводит к распределению Шварца. ■

Теоретический анализ избыточности и стоимости кодирования для распределения Шварца требует большей аккуратности и точности, т. к. $\theta = 0$ — особая точка дзета-функции Римана

$$\zeta(N, \theta) = \sum_{k=1}^{N} \frac{1}{k^{1+\theta}}, \quad (4.65)$$

и других, описывающих асимптотическое поведение $\gamma_\theta(N) = \zeta(N, \theta)$, L_θ, $\mathcal{E}_\theta$ и т. д., поэтому анализ стоимости и избыточности кодирования источников Шварца приводит к чрезвычайно громоздким выкладкам, которые не приводятся в данной работе. Можно показать, что избыточность группового кодирования монотонно возрастает с ростом N и воспользоваться предельными значениями асимптотических рядов при

θ	$N = 1024$	$N = 2048$	$N = 4096$	$N = 8192$	$N = 16384$
-1.0	0.045194	0.045215	0.045225	0.045230	0.045233
-0.5	0.040666	0.040838	0.040959	0.041043	0.041102
-0.25	0.034623	0.034916	0.035150	0.035338	0.035490
-0.125	0.030468	0.030772	0.031025	0.031237	0.031418
0.0	0.025708	0.025968	0.026187	0.026374	0.026535
0.125	0.020637	0.020815	0.020962	0.021086	0.021190
0.25	0.015632	0.015726	0.015800	0.015859	0.015907
0.5	0.007123	0.007130	0.007134	0.007137	0.007138
1.0	0.000000	0.000000	0.000000	0.000000	0.000000

Таблица 4.1. Избыточность группового кодирования

$N = 2^n, n \to \infty$ для нахождения максимума избыточности; точные значения избыточности группового кодирования при некоторых θ и N приводятся в табл. 4.1.

4.3. ХРАНЕНИЕ СЛОВАРЯ

Задача оптимального кодирования словаря еще не решена и предположительно $\mathcal{NP}$-полна [39, 166], поэтому применяются различные эвристические методы.

Один из самых простых способов хранения словаря [39, 166] заключается в использовании методов универсального сжатия, позволяя сократить размер словаря примерно вдвое, и широко используется при сжатии текстов заменой слов [194]. Однако существуют лучшие способы добиться примерно такого же и часто лучшего качества сжатия словаря.

Заметим, что при использовании группового кодирования номеров слов в словаре качество сжатия не зависит от порядка следования в словаре слов, принадлежащих одной группе. Это позволяет отсортировать такие слова по алфавиту и эффективно кодировать общее начало двух соседних слов, которые, скорее всего, будут разными формами одного слова. Например, последовательность слов волк, волчий, волчонок будет закодирована как волк, 3 чий, 4 онок. Такое преобразование словаря сокращает его размер и улучшает качество сжатия примерно в 1.5 раза (до 35% вместо 50%). Отметим, что без использования группового кодирования выделение общего начала невозможно, т. к. если порядок следования слов в словаре определяется не лексикографическим порядком, а частотой использования слов, вероятность того, что два соседних слова словаря имеют достаточно длинное общее начало, очень мала.

Другая проблема, связанная с хранением словаря, заключается в том, что в течение всего времени работы декодировщика нужно хранить весь словарь в декодированном виде, что нежелательно, поскольку объем словаря может быть значительным [148, 194].

Предлагаемый подход заключается в следующем: разобьем отсортированные по алфавиту слова в группе на подгруппы по K слов. Внутри каждой подгруппы применим преобразование, выделяющее общее начало соседних слов (заметим, что для первого слова в группе заранее известно, что длина его общего начала с предыдущим равна нулю), а оставшиеся символы кодируются по Хаффману. Для декодирования i-го слова в словаре необходимо декодировать $\lfloor \frac{i}{K} \rfloor$-ю группу, а в ней — $(i \bmod K)$-е слово. K должно быть выбрано достаточно большим, чтобы ухудшение качества сжатия по сравнению с $K = \infty$ было небольшим, и, с другой стороны, K должно быть невелико,

чтобы декодирование подгруппы не занимало слишком много времени. Экспериментально установлено, что при $K \geq 8$ ухудшение качества сжатия по сравнению с $K = \infty$ менее пяти процентов, поэтому выбор $K = 8$ или $K = 16$ дает почти оптимальное качество сжатия словаря (35–40%).

С целью ускорения декодирования имеет смысл всегда хранить декодированными некоторую часть самых часто встречающихся слов. Так как распределение слов по частоте подчиняется закону Ципфа, то

$$p_i \approx \frac{1}{i \ln N}, \tag{4.66}$$

поэтому вероятность появления уже 32-го слова заведомо меньше 1%. Таким образом, если первые F слов декодированы, а к остальным словам доступ в D раз медленнее, то средневзвешенный коэффициент замедления равен

$$\sum_{i=1}^{F} p_i + D \sum_{i=F+1}^{N-1} p_i \approx D - (D-1)\frac{\ln F}{\ln N}, \tag{4.67}$$

поскольку

$$\sum_{i=1}^{F} p_i \approx \frac{\ln F}{\ln N}. \tag{4.68}$$

Таким образом, быстрый доступ к первым $\sqrt{N}$ слов ускоряет декодирование примерно вдвое, а дальнейшее ускорение декодирования требует экспоненциального роста F. Обычно [194] значение $F = 1000 > \sqrt{N}$, и использование 10–15 Кб оперативной памяти для хранения 1000 самых часто используемых слов представляется небольшой платой за значительное ускорение декодирования, скорость которого мало отличается от скорости декодирования при хранении словаря в незакодированном виде.

Кроме очевидного сокращения размера словаря, хранимого в оперативной памяти в сжатом виде, существует еще и скрытая экономия. Действительно, при хранении словаря в декодированном виде необходимо поддерживать массив из N указателей на начала слов словаря, что на практике вдвое увеличивает расход памяти. Если же хранить указатели только на каждое 8-е или 16-е слово, то объем требуемой памяти уменьшится почти вдвое.

Подход способ хранения словаря существенно лучше предлагавшимихся ранее [143, 148, 149, 193, 194], где указывается, что хранение словаря для $N = 92$ тыс. слов требует во время декодирования около $M = 1$ Мб оперативной памяти (с учетом размера таблицы ссылок на начала слов, занимающей 4 байта/слово). Если же хранить словарь предложенным выше способом, то объем оперативной памяти составит

$$M' \approx (M - 4N) * 0.35 + 4 * \frac{N}{8} \tag{4.69}$$

байтов, т. е. около $M' \approx 275$ Кб, т. е. вчетверо меньше. Как отмечалось в [143, 148, 149], для очень больших словарей в примерно 80% используются слова из первой тысячи наиболее используемых, а в 95% — из числа первых десяти тысяч. Таким образом, при хранении первой тысячи слов в декодированном виде и остальных слов — в сжатом, объем требуемой при декодировании памяти можно уменьшить почти в четыре раза при менее чем двукратном замедлении декодирования; если же хранить в декодированном виде первые 10 тыс. слов, то объем требуемой памяти можно уменьшить в 2.5–3 раза практически без изменения скорости декодирования.

4.4. КОДИРОВАНИЕ ТЕКСТА

Текст есть последовательность слов, разделенных знаками препинания. В дальнейшем для краткости последовательность знаков препинания между двумя словами будет называться *разделителем*.

Обычно предлагается [39, 143, 194] разбивать исходный текст на две последовательности — слов и разделителей — и кодировать текст, используя два различных и независимых словаря для слов и разделителей соответственно. Однако в 80–90% случаев (в зависимости от языка и стиля автора) разделителем является одиночный пробел, а остальные разделители менее вероятны и их распределение более равномерно, поэтому кодирование разделителей при использовании префиксных кодов будет высокоизбыточным: одиночный пробел получит самый короткий код длиной в один бит при оптимальном значении длины кодового слова порядка 0.2–0.3 бита.

Более эффективен подход с использованием единого словаря как для слов, так и для разделителей, создаваемого при первом просмотре текста. После создания словаря из него необходимо исключить самую часто встречающуюся лексему (как правило, это будет пробел с частотой появления 40–45%) и при втором просмотре *вообще не кодировать* удаленную лексему. В свою очередь, декодировщик, встретив последовательность двух лексем одного класса, непосредственно следующих друг за другом, должен вставить между ними исключенную из словаря лексему.

Оставшиеся разделители, за исключением запятой-пробела, точки-пробела и символа конца строки, достаточно маловероятны и не могут нарушить распределение вероятностей слов. В свою очередь, три вышеупомянутых разделителя достаточно хорошо вписываются в словарь и не нарушают распределения Ципфа.

Кроме того, можно заметить, что первое слово предложения, как правило, начинается с заглавной буквы. С целью сокращения словаря и улучшения качества сжатия имеет смысл отслеживать разделители, содержащие точки, вопросительные и/или восклицательные знаки, которые заканчивают предложение, т. к. с высокой вероятностью слово, следующее за таким разделителем, начинается с заглавной буквы. Если это так, то первая буква слова приводится к нижнему регистру, а если нет (что очень маловероятно), то перед следующим словом вставляется служебное слово предложение продолжается. Декодировщик после декодирования разделителя, содержащего символ конца предложения, получает следующее слово и, если это предложение продолжается, то переходит в обычный режим декодирования, иначе первая буква следующего слова делается заглавной. Это преобразование позволяет сократить размер словаря на 10–20%.

4.5. ДРУГИЕ ПРИЛОЖЕНИЯ

Предложенный метод сжатия текстов, обеспечивающий почти произвольный доступ к сжатым данным, может быть успешно использован не только для сжатия гипертекстов.

Заметим, что хранение словаря наиболее часто используемых слов какого-либо языка (размером в 10–15 тысяч слов) отдельно от сжатого текста позволяет существенно повысить качество сжатия до двух битов на символ; в настоящее время ни один другой известный автору метод сжатия, отличный от замены слов из номерами в словаре, не позволяет добиться такого качества (аналогичные сведения приводятся в [143, 149, 194]). При использовании отдельно хранимого статического словаря новые слова помещаются в динамический словарь (слова которого нумеруются после слов

	ostrov	book1
Исходный размер (Кб)	578	768
Слов в тексте (тыс)	110	172
Слов в словаре (тыс)	20	14
Размер словаря (Кб)	182	114
Сжатый текст (Кб)	144	202
Сжатый словарь (Кб)	62	44
Битов на слов	10.4	9.4
Избыточность (битов на слово)	0.1	0.06
Битов на символ (без словаря)	2.0	2.1
Битов на символ (со словарем)	2.8	2.6
Текст + словарь (Кб)	206 (7 сек)	246 (8 сек)
HC31 (Кб)	392 (21 мин)	481 (32 мин)
PKZIP 2.04G (Кб)	247 (12 сек)	316 (18 сек)

Таблица 4.2. Сравнение с другими методами сжатия текстов

статического словаря), который хранится вместе с текстом.

Такой сжатие может успешно использоваться для хранения деловой переписки учреждения, передачи по линиям связи и хранения почтовых сообщений (например, только в группу relcom.commerce ежедневно поступает несколько тысяч коротких — менее 10 строк — сообщений). В последнем случае сжатие с использованием статического словаря с качеством сжатия порядка 25% может существенно снизить нагрузку на линию связи, почтовый сервер, уменьшить время передачи данных, т. к. качество сжатия используемого в настоящее время алгоритма LZW [184] — лишь 50–60%.

Кроме того, возможна реализация сжатия текста, допускающая модификацию сжатого представления: при модификации участка текста измененный текст заново сжимается и заменяет старый участок, при этом слова, отсутствующие в словаре, добавляются к концу динамического словаря. При достаточно большом статическом словаре вероятность появления новых слов невелика, поэтому качество сжатия будет мало отличаться от теоретического (порядка 25%).

В последнем случае групповое кодирование исключительно удобно, т. к. при известном размере статического словаря нетрудно построить бесконечный код и избежать ограничений на размер динамического словаря. Например, можно использовать следующие длины кодовых слов для номеров групп, близкие к оптимальным:

```
4 4 4 3 3 3 3 4 4 4 5 5 5 6 7 8 9 10 ...
```

Нетрудно заметить, что при динамически растущем словаре локальные изменения исходного текста вызывают локальные изменения сжатого представления (в пределах одного предложения или параграфа). Это обстоятельство очень важно при применении сжатия в текстовых процессорах, предназначенных для работы с большими документами, т. к. остальные методы сжатия требуют перекодирования всего документа при внесении в него даже незначительных изменений.

4.6. СРАВНЕНИЕ С ДРУГИМИ МЕТОДАМИ

Автор использовал для тестов два художественных произведения: Обитаемый остров братьев Стругацких на русском языке и Far from the madding crowd Hardy

на английском языке (последняя книга входит в состав тестового набора данных Calgary Compression Corpus для сравнения методов сжатия между собой).

В табл. 4.2 приведены результаты[2] сжатия этих книг с использованием описанного метода (размер подгруппы словаря K был равен 8); кроме того, приводятся результаты сжатия с помощью утилиты HC31, которая сжимает тексты подсказок для Microsoft Windows и является лучшей программой сжатия гипертекстов, и программы PKZIP (одного из самых популярных в мире универсальных архиваторов).

Из табл. 4.2 видно, что стоимость кодирования (битов на слово) хорошо согласуется с теоретическими значениями, а избыточность группового кодирования несколько больше теоретической ввиду того, что распределение слов по частотам на практике несколько отличается от распределения Ципфа. Избыточность кодирования порядка 1%, вносимая групповым кодированием, на практике не оказывает существенного влияния на качество сжатия.

Нетрудно заметить, что даже при хранении словаря вместе с текстом предложенный метод кодирования уверенно лидирует как по качеству сжатия, так и по скорости. Аналогичный по назначению продукт фирмы Microsoft (HC31) не только в 150–200 раз медленнее разработанной автором системы сжатия, но и обладает гораздо худшим качеством сжатия при очень большом объеме требуемой памяти памяти, в 10 раз превышающем размер сжимаемого текста.

Сравнение специальных методов сжатия, использующих заранее известные особенности сжимаемых данных, с универсальными (PKZIP), строго говоря, не корректно и приводится для того, чтобы убедиться в том, что предложенный метод сжатия действительно достаточно эффективен.

Отметим ряд отличий предложенных методов кодирования от известных методов сжатия заменой слов их номерами в словаре, описанных в [39, 106, 139, 143, 148, 149, 160, 192, 193, 194]:

1. Вместо кода Хаффмана и/или кодов ограниченно длины для записи номеров слов в словаре используется групповое кодирование, что позволяет избежать серьезных проблем, связанных с необходимостью построения кодов ограниченной длины и/или обработки кодов Хаффмана большой длины (см. [110, 111, 123, 173, 174, 175, 176]).
2. При групповом кодировании сортировкой в лексикографическом порядке слов словаря, принадлежащих одной группе, и эффективным кодированием общего начала двух лексикографически соседних слов словаря качество сжатия словаря повышается в 1.5 раза (при прямолинейном использовании кодов Хаффмана, как описано в цитируемых работах, такая сортировка и, соответственно, кодирование невозможны).
3. При кодировании с пополняемым словарем редко встречающиеся и новые слова добавляются в словарь и эффективно кодируются, как описано в предыдущем пункте (обычно [143, 148, 194] редко встречающиеся и новые слов хранятся непосредственно в закодированном тексте, что менее эффективно и ухудшает качество сжатия).
4. Метод хранения словаря в сжатом виде позволяет в 3–4 раза снизить объем оперативной памяти, необходимой при декодировании, при почти такой же скорости декодирования (обычно [143, 148, 149, 193, 194] словарь хранится в декодированном виде, что при большом размере словаря вынуждает хранить редко встречающиеся непосредственно в закодированном тексте, что менее эффективно и негативно сказывается на качестве сжатия).

[2] Испытания проводились в 1994 г. на IBM PC с процессором Intel-486/DX4 с тактовой частотой 100 МГц и 8 Мб оперативной памяти.

5. Использование единого словаря для хранения всех лексем и исключение из процесса кодирования наиболее часто используемой лексемы позволяет увеличить качество сжатия на 10–15% по сравнению с описанным в [39, 143, 148, 149, 193, 194] применением раздельных словарей для слов и разделителей.

4.7. ВЫВОДЫ

Предложенный метод сжатия был реализован автором в конце 1993 г. в рамках проекта **bCAD** (введен в промышленную эксплуатацию в 1995 г.) и в настоящее время используется для хранения не только текстов подсказок, но и всех сообщений системы. Это позволило реализовать версии **bCAD** на русском, английском, итальянском, французском и немецком языках, причем изменение языка общения с пользователем не требует никаких изменений в самой системе — достаточно заменить тексты подсказок и набор шрифтов. Поскольку исходные тексты сообщений на каком-либо одном языке занимают примерно 20% объема всей системы, уменьшение их размера втрое заметно снизило размер всей системы.

К числу достоинств алгоритма следует отнести прозрачность и простоту реализации, очень хорошее качество и высокую скорость сжатия.

Что касается скорости восстановления исходных данных, то при хранении словаря в развернутом виде скорость декодирования находится очень высокая, а при хранении словаря в сжатом виде незначительно уступает скорости восстановления данных при использовании других методов сжатия.

Следует отметить, что предложенный алгоритм может быть с успехом использован и в других случаях, где требуется хранить или передавать большое число коротких сообщений на естественных языках. В частности, это относится к текстовым процессорам, системам поддержки служебной переписки, к почтовым системам и др.

Использование отдельно хранимого фиксированного словаря повышает качество сжатия до 25%, что в полтора-два раза лучше качества сжатия других методов.

Групповое кодирование в сочетании с использованием бесконечных кодов допускает неограниченный рост словаря, поэтому предложенный метод сжатия может применяться и для модифицируемых текстов.

Глава 5

ПОСТРОЕНИЕ ОПТИМАЛЬНЫХ LZ77-КОДОВ

Многие системы сжатия в своей основе содержат ту или иную модификацию методов *словарного сжатия*, описанных в пп. 2.8 и 2.8.1, сущность которых состоит в том, что подстроки входной строки заменяются указателем на то место в тексте, где они уже появлялись ранее. Это семейство алгоритмов носит имена Лемпела-Зива и обозначается как *LZ-схемы* [38, 87, 126, 167, 201, 202]. Применение LZ-схем позволяет получить хорошее качество сжатия, их важным свойством является очень быстрая работа декодировщика [39, 166].

Мы ограничимся рассмотрением только наиболее эффективных по качеству и скорости сжатия LZ77-схем (см. сравнение около 30 различных схем сжатия в [38, 39, 166]) — со свободной смесью указателей и символов в выходном потоке (возможно, с энтропийным кодированием выхода, как описано в главе 7), основанных на алгоритмах LZSS [167], LZH [47] и LZB [36]; основные определения, обозначения и термины введены в п. 2.8.1.

Как отмечалось в п. 2.8.1, кодируемая строка S может иметь не один LZ77-код, а несколько, стоимости кодирования которых различны; данная глава посвящена описанию разработанных автором эффективных методов построения оптимальных и близких к оптимальных LZ77-кодов (точное определение стоимости кодирования приводится в п. 5.1).

Насколько известно автору, попыток создания оптимальных методов сжатия для LZ77-схем до сих пор не предпринималось. В монографии Сторера [166] рассматриваются вопросы оптимизации порядка кодирования для очень общих случаев (при переменной стоимости кодирования литералов и указателей, для обобщенных LZ77-кодов — так называемых макро-схем — и т. д.); показано, что решение этой задачи для таких случаев $\mathcal{NP}$-полно.

Построение оптимальных схем для словарного сжатия с фиксированным словарем и постоянной стоимостью кодирования ссылок в словарь рассматривались Вагнером [182, 183] (позднее алгоритм Вагнера был существенно улучшен Рабином [157]). Миллером и Вегманом [137, 138] было предложено обобщение алгоритма Вагнера на случай фиксированного словаря при переменной стоимости кодирования ссылок; Катаяйнен и Райта [114] привели более эффективный вариант алгоритма Миллера — Вегмана (аналогично тому, как это было сделано Рабином для алгоритма Вагнера), позволяющий в несколько раз повысить производительность системы сжатия.

Автору неизвестны аналоги высокоэффективных алгоритмов оптимизации стоимости кодирования Рабина и/или Катаяйнена — Райты, применимых для LZ77-схем (оригинальные алгоритмы налагают и используют ряд существенных ограничений на словарь,

почти все из которых не выполняются при использовании LZ77-схем). Как правило, для повышения качества сжатия алгоритмов семейства LZ77 применяются различные эвристические методы.

5.1. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ И ОБОЗНАЧЕНИЯ

Обозначим $S = s_0 \dots s_{L-1}$ кодируемую строку длины $L > 0$ над алфавитом Σ, а W — ширину окна, $W > 1$; определение LZ77-кода и ряда других терминов дается в п. 2.8.1.

Определение 5.1. Обозначим $S[i : \ell]$ подстроку длины ℓ строки S, начинающуюся с i-го символа ($i + \ell \le L$):

$$S[i : \ell] = s_i \dots s_{i+\ell-1}. \tag{5.1}$$

■

Определение 5.2. i-ым *суффиксом* строки S ($i = 0, \dots, L-1$) называется подстрока $S[i : L - i] = s_i \dots s_{L-1}$. ■

Определим целочисленные функции $pos(i)$ и $len(i)$ как позицию и длину самой длинной и самой правой подстроки длины не более W, являющейся началом подстроки $S[i : L - i]$, и расположенной левее i: $0 \le pos(i) < i$. Более формально,

$$len(i) = \max_{l=0}^{\min\{L-i,W\}} \Big\{ \ell \mid \exists p \in range(i) \ : S[p : \ell] = S[i : \ell] \Big\}, \tag{5.2}$$

$$pos(i) = \max_{p \in range(i)} \Big\{ p \mid S\big[p : len(i)\big] = S\big[i : len(i)\big] \Big\}, \tag{5.3}$$

где W — ширина окна ($W = \infty$ означает, что ограничений не существует), а $range(i) = [\max\{i - W, 0\}, i - 1]$. Если искомой подстроки не существует ($len(i) = 0$), то положим $pos(i) = i$ и $len(i) = 1$.

Для строки может существовать несколько LZ77-кодов. Например, строка *abcbcababcab* может быть закодирована двумя способами: *abc (2,2) (2,5) (2,2) (3,7) (2,3)* или *abc (2,2) (2,5) a (4,5)*. Среди всех возможных представлений строки в сжатом виде есть одно или несколько, обладающих минимальной длиной.

Замечание 5.1. В дальнейшем будут рассматриваться только LZ77-схемы, порождающий LZ77-код согласно вышеприведенному определению функций *pos* и *len*, т. е. каждый указатель (ℓ, p) в LZ77-коде должен указывать на наиболее длинную и самую правую подстроку (см. также замечание 5.3). ■

Определение 5.3. Обозначим $send_i(\ell, p)$ стоимость кодирования ссылки на подстроку длины $\ell > 1$, расположенной p символами левее позиции i. В дальнейшем будем полагать, что при $\ell \le 1$ кодируется литерал s_i, так что значение функции $send_i(\ell, \cdot)$ при $\ell \le 1$ равно стоимости кодирования i-го литерала s_i, а не указателя. ■

Определение 5.4. Определим функцию $cost(i)$ как минимальную стоимость кодирования i-го суффикса $S[i : L - i]$ строки S. ■

Определение 5.5. Метод сжатия, гарантирующий минимальную длину сжатого представления кодируемой строки [при заданной функции *send*], называется *оптимальным*. ■

Для того чтобы вычислить $cost(i)$, определяющей оптимальное кодирование в позиции i, переберем все возможные способы кодирования i-го суффикса S. Если $1 \le$

$k \le len(i)$, то можно закодировать указатель на подстроку длины k (или литерал при $k = 1$), а затем — оптимальным образом — $(i + k)$-й суффикс строки.

Так как стоимость кодирования пустой строки равна нулю, то $cost(L) = 0$, а для i таких, что $0 \le i < L$,

$$cost(i) = \min_{k=1}^{len(i)} \bigl(send_i(k, pos(i)) + cost(i + k)\bigr). \tag{5.4}$$

Заметим, что $i + len(i) \le L$ по определению функции $len(i)$, поэтому определение функции $cost$ корректно.

В принципе, полная оптимизация стоимости кодирования — нахождение точного значения функции $cost(0)$ — требует применения динамического программирования. Сторер [166] приводит обобщение алгоритма динамического программирования Миллера — Вегмана [137, 138], которое заключается в следующем.

Вычислим значения функции $cost(i)$ для $i = L - 1, \ldots, 0$ согласно ее определению. Так как для каждого i значения $cost(i+1), \ldots, cost(L)$ уже известны и $i + len(i) \le L$, то вычисление $cost(i)$ возможно за $len(i)$ операций. Зная значение функции $cost(i)$ для всех i от 0 до $L - 1$, мы сможем для каждого i, начиная с $i = 0$, определить k, при котором достигается минимум выражения $\Bigl(send(k) + cost(i + k)\Bigr)$, и закодировать указатель на подстроку $\bigl(k, pos(i)\bigr)$ $(k \ne 1)$ или литерал $(k = 1)$, после чего увеличить i на k и продолжить кодирование до тех пор, пока i не станет равным L.

Такой подход имеет множество недостатков, из-за которых он и не применяется на практике. Вычисление значения $cost(0)$ и определение оптимального способа кодирования требует

$$\sum_{i=0}^{L-1} Clen(i) \tag{5.5}$$

операций. Кроме того, необходимо CL единиц памяти для хранения значений $len(i)$, $pos(i)$ и $cost(i)$ (расход памяти может быть уменьшен разбиением входной строки на несколько более коротких подстрок). Однако самая большая неприятность заключается в том, что требуется знание значения $len(i)$ для всех i. Поиск подстрок (т. е. вычисление $len(i)$ и $pos(i)$) — настолько медленная операция, что по сравнению с ней собственно стоимость вычисления $cost(0)$ несущественна.

Замечание 5.2. Время поиска (для RAM-машины, описанной в п. 2.1) наиболее длинной и самой правой подстроки, являющейся началом i-го суффикса, при использовании соответствующих структур данных и алгоритмов поиска (см. [39, 132, 155, 166, 177] и главу 6), почти константно; поскольку количество команд RAM-машины, исполняемых для собственно построения оптимальной последовательности кодирования, мало по сравнению с количеством команд, исполняемых во время поиска и прямо пропорционально количеству позиций, для которых осуществлялся поиск подстрок, **время кодирования** $T(S)$ строки S пропорционально количеству позиций i, где понадобилось вычисление значений $len(i)$ и $pos(i)$. ■

Если обозначить через $N_{lit}(S)$ и $N_{ptr}(S)$ количество литералов и указателей в сжатом представлении S, то при любом нетривиальном алгоритме выбора порядка кодирования потребуется не меньше

$$T_{min}(S) = N_{lit}(S) + N_{ptr}(S) \tag{5.6}$$

поисков подстрок, т. к. знание $len(i)$ необходимо хотя бы для того, чтобы определить, нужно ли (и можно ли) кодировать указатель или литерал.

Время работы алгоритма Миллера — Вегмана равно

$$T_{max}(S) = L = N_{lit}(S) + L_{ave}(S)N_{ptr}(S), \tag{5.7}$$

где $L_{ave}(S) \geq 2$ — средняя длина подстрок, замененных указателями в строке S.

Для текстов на естественных языках $L_{ave} = 5 \div 9$ и $N_{lit} \approx N_{ptr}$ [39, 166], поэтому сжатие с полной оптимизацией в 3–5 раз медленнее сжатия без оптимизации.

Основная идея предлагаемых автором алгоритмов выбора оптимальной (или близкой к оптимальной) последовательности кодирования совпадает с основными идеями алгоритмов Рабина [157] и Катаяйнена — Райты [114], и основана на том, что при аккуратном вычислении минимума $cost(0)$ (и тем самым определении оптимальной последовательности кодирования) значения $cost(i)$ в некоторых промежуточных точках *никогда* не будут использованы, т. е. при оптимальном кодировании начало i-го суффикса всегда будет захвачено при кодировании начала предшествующего суффикса.

Замечание 5.3. Известно много различных LZ77-схем, отличающихся от определенных в данной работе. Рассматриваемые LZ77-схемы называются [38, 39, 166] *рекурсивными LZ77-схемами со свободной смесью литералов и указателей в выходном потоке и выбором наиболее длинной подстроки*: они допускают пересечение подстрок $S[i : len(i)]$ и $S[pos(i) : len(i)]$ (поскольку алгоритм декодирования LZ77-кодов, описанный в п. 2.8.1, трактует такие рекурсивные указатели однозначным образом) и накладывают не накладывают ограничений на порядок следования литералов и указателей; в ряде работ [166, 201] рассматривались LZ77-схемы с различными ограничениями, не допускающие рекурсии в указателях и/или требующие, чтобы за указателем обязательно следовал литерал; такие ограничения не являются необходимыми и не улучшают качество сжатия [166].

Функции len и pos определены так, чтобы они были *однозначными*, выбирая самую длинную и самую правую подстроку, являющуюся началом текущего суффикса. Это ограничение необходимо: Сторером [166] было показано, что в общем случае отыскание LZ77-кода минимальной длины требует полного перебора всех LZ77-кодов, порождающих кодируемую строку. С другой стороны, Сторер, Белл и др. [38, 39, 166] отмечали, что в подавляющем большинстве случаев выбор наиболее длинной и самой правой подстроки оптимален, поэтому почти все известные LZ77-схемы [36, 38, 39, 47, 166, 167] кодируют указатели именно на такие подстроки. ■

Сначала, как более простой (и практически более важный), рассмотрим случай *фиксированной* стоимости кодирования литералов и указателей, т. е.

$$Send(\ell) = send_i(\ell, p) = \begin{cases} C, & \ell \leq 1, \\ P, & \ell > 1, \end{cases} \tag{5.8}$$

при $C < P$.

Как следует из определения функции $cost$, при вычислении $cost(0)$ вместо C и P можно использовать $C' = 1$ и $P' = r = \frac{P}{C}$. Полученная последовательность кодирования (например, при помощи алгоритма Миллера — Вегмана) будет оптимальной и при стоимостях кодирования литералов и указателей, равных C и P соответственно, поэтому в дальнейшем для упрощения выкладок (без ограничения общности) будем считать, что стоимость кодирования литерала равна $C' = 1$, а стоимость кодирования указателя — $P' = r$, $r > 1$.

5.2. КОДИРОВАНИЕ ПРИ $r = 2$

Сначала мы рассмотрим очень простой случай $C = 1$ и $P = 2$, т. е.

$$Send(\ell) = \begin{cases} 1, & \ell \le 1, \\ 2, & \ell > 1. \end{cases} \tag{5.9}$$

Такое определение функции *Send* логично с теоретической точки зрения (литерал — последовательность из одного символа алфавита Σ', содержащего символы алфавита Σ и все натуральные числа в интервале $[1, W-1]$, и стоимость его [равномерного] кодирования равна $\log_2 |\Sigma'|$ битов, а указатель — из двух; его кодирование требует $2\log_2 |\Sigma'|$ битов) и хорошо согласуется с практическими данными, т. к. при не очень большой ширине окна ($W \le 2^{20}$) средняя стоимость кодирования литерала (см. [39, 166]) примерно вдвое[1] меньше средней стоимости кодирования указателя (при использовании эффективных методов кодирования выхода LZ77, аналогичных описанным в главе 7).

Кроме того, как будет показано в дальнейшем, все или почти все известные эвристики выбора последовательности кодирования для LZ77-схем — частные случаи предложенного автором алгоритма оптимизации при $r = 2$ с различными ограничениями на допустимый объем используемой дополнительной памяти.

5.2.1. ОСНОВНАЯ ТЕОРЕМА ОПТИМИЗАЦИИ

Теорема 5.1. При $0 \le i < L - 1$

$$len(i+1) \ge len(i) - 1. \tag{5.10}$$

Доказательство. Заметим, что $len(i) \ge 1$ по определению функции *len*. Если $len(i) = 1$, то $len(i+1) \ge 1 > len(i) - 1 = 0$. Рассмотрим случай $len(i) > 1$.

Так как подстрока длины $len(i) > 1$, начинающаяся в позиции $pos(i)$, была началом строки $S\big[i : len(i)\big] = S\big[pos(i) : len(i)\big]$, то подстрока длины $len(i-1)$, начинающаяся в следующей позиции $pos(i) + 1$, является началом строки $S\big[i+1 : len(i) - 1\big] = S\big[pos(i) + 1 : len(i) - 1\big]$, также начинающейся в следующей позиции.

Смещение от позиции $(i+1)$ до начала подстроки $S\big[pos(i) + 1 : len(i) - 1\big]$ равно

$$(i+1) - \big(pos(i) + 1\big) = i - pos(i) \le W, \tag{5.11}$$

поэтому подстрока $S\big[pos(i) + 1 : len(i) - 1\big]$, начинающая строку $S\big[i + 1 : L - i - i\big]$, — допустимая. Следовательно, $len(i+1) \ge len(i) - 1$, что и требовалось доказать. ■

Теорема 5.2. Для всех i от 0 до $L - 1$ включительно выполняется

$$cost(i+1) \le cost(i) \le cost(i+1) + 1 \tag{5.12}$$

и

$$cost(i) = 1 + \min\Big\{cost(i+1), cost\big(i + len(i)\big) + 1\Big\}. \tag{5.13}$$

Другими словами, функция *cost* — невозрастающая, и ее значения в соседних точках отличаются не более чем на единицу. Кроме того, минимум в определении $cost(i)$ достигается при $k = 1$ или $k = len(i)$.

---

[1] Эмпирические исследования, проведенные автором для 50 тыс. текстовых и двоичных файлов, хранящихся на одном из ftp-серверов, показали, что при $W \in [2^{13}, 2^{20}]$ отношение r средней стоимости кодирования указателя P к средней стоимости литерала C находится в интервале $[2.25, 2.75]$.

Доказательство. Если $len(i) = 1$, то $cost(i) = cost(i+1) + 1$, и утверждение истинно. Так как $len(L-1) = 1$ по определению функции len, то утверждение теоремы можно доказывать индукцией по $i = L-1, \ldots, 0$.

Допустим, что утверждение теоремы доказано для $i+1, \ldots, L-1$. Рассмотрим k такое, что $2 \le k \le len(i)$. Так как при $j > i$

$$cost(j+1) \le cost(j) \le cost(j+1) + 1, \tag{5.14}$$

то $cost(j)$ — невозрастающая функция при $j > i$. Приняв $i+k$ в качестве j и воспользовавшись тем, что $Send(k) = Send\big(len(i)\big) = 2$, получим

$$Send(k) + cost(i+k) \ge Send\big(len(i)\big) + cost\big(i + len(i)\big), \tag{5.15}$$

т. к. $i < i + k \le len(i)$.

Итак, минимум в определении функции $cost$ достигается при $k = 1$ или $k = len(i)$, и для всех $j = i, \ldots, L-1$

$$cost(i) = 1 + \min\Big\{cost(i+1), cost\big(i + len(i)\big) + 1\Big\}. \tag{5.16}$$

Из этого равенства немедленно следует, что

$$cost(i) \le 1 + cost(i+1). \tag{5.17}$$

Покажем, что $cost(i) \ge cost(i+1)$ (интуитивно понятно, что стоимость кодирования любого суффикса подстроки не больше стоимости кодирования всей строки).

По определению функции $cost$ имеем

$$cost(i+1) \le Send\big(len(i+1)\big) + cost\big((i+1) + len(i+1)\big), \tag{5.18}$$

а т. к. $Send\big(len(i+1)\big) \le 2$ по определению функции $Send$, $len(i+1) \ge len(i) - 1 \ge 1$ по теореме 5.1 и функция $cost(j)$ — невозрастающая при $j > i$ по предположению индукции, то

$$cost(i+1) \le 2 + cost\big((i+1) + (len(i) - 1)\big) = cost\big(i + len(i)\big) + 2. \tag{5.19}$$

Следовательно,

$$\begin{aligned} cost(i) &= \min\Big\{cost(i+1) + 1, cost\big(i + len(i)\big) + 2\Big\} \ge & (5.20)\\ &\ge \min\Big\{cost(i+1) + 1, cost(i+1)\Big\} = & (5.21)\\ &= cost(i+1), & (5.22) \end{aligned}$$

что и требовалось доказать. ■

Теорема 5.3. Если $len(i) \le 2$, то

$$cost(i) = cost(i+1) + 1. \tag{5.23}$$

Доказательство. Если $len(i) = 1$, то $cost(i) = cost(i+1) + 1$ по определению функции $cost$.

Если $len(i) = 2$, то по теореме 5.2

$$cost(i) = 1 + \min\Big\{cost(i+1), cost(i+2) + 1\Big\}. \tag{5.24}$$

Другим следствием теоремы 5.2 является то, что

$$cost(i+2) \ge cost(i+1) - 1, \tag{5.25}$$

поэтому $cost(i+2) + 1 \ge cost(i+1)$, и

$$cost(i) = 1 + \min\Big\{cost(i+1), cost(i+2) + 1\Big\} = cost(i+1) + 1, \tag{5.26}$$

что и требовалось доказать. ■

Обозначим

$$g_n(i) = \begin{cases} i, & n = 0, \\ g_{n-1} + len(g_{n-1}), & n > 1. \end{cases} \tag{5.27}$$

Другими словами, $g_1(i) = i + len(i)$, $g_2(i) = \big(i + len(i)\big) + len\big(i + len(i)\big)$ и т. д.

При фиксированном i $(0 \le i \le L-1)$ определим функцию f_n:

$$f_n = \begin{cases} g_k(i), & n = 2k, \\ g_k(i+1), & n = 2k+1. \end{cases} \tag{5.28}$$

Теорема 5.4. Если

$$\begin{cases} f_j = f_{j-1} + 1, & j = 1, \\ f_j > f_{j-1} + 1, & j = 2, \ \ldots, \ m-1, \\ f_j \le f_{j-1} + 1, & j = m, \end{cases} \tag{5.29}$$

($f_1 = f_0 + 1$ по определению функции f), то для всех $k = 0, \ldots, m-2$ выполняется

$$\begin{cases} cost(f_k) \ge cost(f_{k+1}) + 1, & (m-k) = 2j, \\ cost(f_k) = cost(f_{k+2}) + 2, & (m-k) = 2j+1. \end{cases} \tag{5.30}$$

Доказательство. Докажем утверждение теоремы индукцией по $k = m-2, \ldots, 0$. Заметим, что $f_k + len(f_k) = f_{k+2}$ по определению функции f, поэтому по теореме 5.2

$$cost(f_k) = 1 + \min\Big\{cost(f_k + 1), cost(f_{k+2}) + 1\Big\}. \tag{5.31}$$

Так как $k \le m-2$, то по условию теоремы $f_k + 1 \le f_{k+1}$. Воспользовавшись тем, что функция $cost$ — невозрастающая, получим

$$cost(f_k + 1) \ge cost(f_{k+1}); \tag{5.32}$$

это неравенство будет использоваться в дальнейшем.

Докажем базис индукции ($k = m-2$). По условию $f_m \le f_{m-1} + 1$, поэтому

$$cost(f_m) \ge cost(f_{m-1} + 1). \tag{5.33}$$

Так как по теореме 5.2 $cost(i+1) + 1 \ge cost(i)$, то $cost(f_{m-1} + 1) \ge cost(f_{m-1}) - 1$. Следовательно,

$$cost(f_m) \ge cost(f_{m-1} + 1) \ge cost(f_{m-1}) - 1. \tag{5.34}$$

Таким образом, базис индукции ($k = m-2$) доказан, т. к.

$$cost(f_{m-2}) \ge 1 + \min\Big\{cost(f_{m-1}), cost(f_{m-1}) - 1 + 1\Big\} = cost(f_{m-1}) + 1. \tag{5.35}$$

Пусть $(m-k)$ — нечетное число. По теореме 5.2

$$cost(f_k) = 1 + \min\Big\{cost(f_k + 1), cost(f_{k+2}) + 1\Big\}. \tag{5.36}$$

Как было показано выше, $cost(f_k + 1) \ge cost(f_{k+1})$. Так как $\big(m - (k+1)\big)$ — четное и $k < k+1 \le m-2$, то по предположению индукции $cost(f_{k+1}) \ge cost(f_{k+2}) + 1$, и тогда

$$cost(f_k + 1) \ge cost(f_{k+1}) \ge cost(f_{k+2}) + 1. \tag{5.37}$$

Следовательно,

$$\min\Big\{cost(f_k + 1), cost(f_{k+2}) + 1\Big\} = cost(f_{k+2}) + 1, \tag{5.38}$$

поэтому

$$cost(f_k) = cost(f_{k+2}) + 2. \tag{5.39}$$

Пусть $m-k$ — четное число. По теореме 5.2

$$cost(f_k) = 1 + \min\Big\{cost(f_k + 1), cost(f_{k+2}) + 1\Big\}. \tag{5.40}$$

Так как $k \le m-4$ и $k+2 \le m-2$, то $cost(f_{k+2}) \ge cost(f_{k+3})+1$ по предположению индукции. Так как $\big(m-(k+1)\big)\big)$ — нечетное число, то $cost(f_{k+1}) = cost(f_{k+3})+2$. Следовательно,

$$cost(f_{k+2}) \ge cost(f_{k+3}) + 1 = cost(f_{k+1}) - 1. \tag{5.41}$$

В свою очередь $cost(f_k+1) \ge cost(f_{k+1})$, как было показано выше.

Подставив в выражение для $cost(f_k)$ полученные неравенства, получим

$$cost(f_k) \ge cost(f_{k+1}) + 1, \tag{5.42}$$

что и требовалось доказать. ■

5.2.2. АЛГОРИТМ ПОЛНОЙ ОПТИМИЗАЦИИ

Вышеприведенные теорема 5.3 и теорема 5.4 дают необходимые и достаточные условия для обнаружения и разрешения возможных неоднозначностей кодирования.

Действительно, если $len(i) \le 2$, то по теореме 5.3

$$cost(i) = cost(i+1) + 1, \tag{5.43}$$

и кодирование литерала в позиции i оптимально.

Если $len(i) \ge 3$, то необходимо вычислить f_k, $0 \le k \le m$. По определению,

$$f_k = \begin{cases} i, & k=0, \\ i+1, & k=1, \\ i+len(i), & k=2, \\ (i+1)+len(i+1), & k=3, \\ \dots \end{cases} \tag{5.44}$$

Так как $len(i)$ известно, то необходимо вычислить $len(i+1)$. Если

$$f_3 \le f_2+1 \iff (i+1)+len(i+1) \le i+len(i)+1 \iff len(i+1) \le len(i), \tag{5.45}$$

то по теореме 5.4

$$cost(i) = cost(f_0) = cost(f_2) + 2 = cost\big(i+len(i)\big)\big), \tag{5.46}$$

поэтому кодирование указателя в позиции i оптимально. Итак, кодирование неоднозначно тогда и только тогда, когда

$$2 < len(i) < len(i+1). \tag{5.47}$$

В общем случае нам необходимо продолжить вычисление последовательности

$$f_k = f_{k-2} + len(f_{k-2}) \tag{5.48}$$

для $k = 4, \dots, m$ до тех пор, пока $f_k > f_{k-1}+1$.

Теорема 5.5. Пусть m такое, что $f_m \le f_{m-1}+1$. Если m — нечетное, то оптимальным кодированием подстроки $S[i : f_{m-1}-i]$ является кодирование указателей

$$\big(len(f_{2k}), f_{2k} - pos(f_{2k})\big) \tag{5.49}$$

в позициях $f_0 = i$, $f_1 = i+len(i)$, $\dots$, f_{m-3} при $k = 0, \dots, \frac{m-3}{2}$.

Если m — четное, то оптимальным кодированием подстроки $S[i : f_{m-1}-i]$ будет кодирование литерала в позиции i и указателей

$$\big(len(f_{2k+1}), f_{2k+1} - pos(f_{2k+1})\big) \tag{5.50}$$

в позициях $f_1 = i+1$, $f_3 = (i+1)+len(i+1)$, $\dots$, f_{m-3} при $k = 0, \dots, \frac{m-4}{2}$.

Доказательство. Если m — нечетное число, то $(m-2k)$ — нечетное, и, согласно теореме 5.4, имеем

$$cost(f_{2k}) = cost(f_{2k+1}) + 2 = cost\big(f_{2k} + len(f_{2k})\big)\big) + 2. \tag{5.51}$$

Следовательно, в позициях $f_0 = i$, $f_2 = i + len(i)$, ..., f_{m-3} кодирование указателей оптимально (частный случай $m = 3$ рассмотрен выше).

Если m — четное число, то по теореме 5.4 имеем

$$cost(i) = cost(f_0) \geq cost(f_1) + 1 = cost(i+1) + 1. \tag{5.52}$$

Заметим, что по определению функции $cost$

$$cost(i) = 1 + \min\Big\{cost(i+1), cost\big(i + len(i)\big)\Big) + 1\Big\} \leq cost(i+1) + 1, \tag{5.53}$$

поэтому $cost(i) = cost(i+1) + 1$, и кодирование литерала в позиции i оптимально.

Так как

$$cost(f_{2k-1}) = cost(f_{2k+1}) + 2, \tag{5.54}$$

то кодирование указателей в позициях $f_1 = i + 1$, $f_3 = (i+1) + len(i+1)$, ..., f_{m-3} оптимально.

Осталось заметить, что, т. к. $f_{k+2} = f_k + len(f_k)$, мы оптимальным образом закодировали подстроку от i-й до f_{m-1}-й позиции включительно, причем корректным образом (без пропусков символов). ■

5.2.3. СКОРОСТЬ ОПТИМАЛЬНОГО СЖАТИЯ

Теорема 5.6. Количество поисков подстрок $T_{opt}(S)$, требуемых для оптимального сжатия строки S, равно

$$T_{opt}(S) = N_{lit}(S) + 2N_{ptr}(S). \tag{5.55}$$

Доказательство. Если $len(i) \leq 2$, то было применено кодирование литерала. В этом случае потребовался один поиск подстроки и длина выходного потока увеличилась на $Send(1) = 1$ символ.

Если $len(i) \geq 3$, то $len(f_k)$ было вычислено для f_k при $k = 0, \ldots, m-2$ — для $m-1$ позиций. При нечетном m было закодировано ровно $\frac{m-1}{2}$ указателей и длина входного потока увеличилась на $2\frac{m-1}{2} = m - 1$ символов. При четном m был закодирован один литерал и $\frac{m-2}{2}$ указателей, при этом длина выходного потока увеличилась на $2\frac{m-2}{2} + 1 = m - 1$ символов.

Итак, для оптимального сжатия строки S потребовалось ровно столько поисков подстрок, какова длина выходной строки, т. е.

$$T_{opt} = cost(0) = N_{lit} + 2N_{ptr}, \tag{5.56}$$

т. к. стоимость кодирования литерала равна единице, а стоимость кодирования указателя — двум. ■

Поскольку временная сложность алгоритма кодирования линейна по T_{opt} и время поиска подстрок (вычисления значений функций len и pos) константно (см. замечание 5.2), временая сложность кодирования равна $O(T_{opt}) = O(cost(0))$.

Так как алгоритм Миллера — Вегмана осуществляет поиск подстрок *во всех позициях*, его временная сложность равна $O(L)$, где L — длина кодируемой строки S, поэтому предложенный алгоритм быстрее алгоритма Миллера — Вегмана в L/T_{opt} раз, т. е. ровно во столько раз, насколько больше символов в исходной строке по отношению к количеству символов в LZ77-коде минимальной длины.

5.2.4. АЛГОРИТМ ПОЧТИ ОПТИМАЛЬНОГО СЖАТИЯ

Оценим объем оперативной памяти, необходимой для оптимального сжатия строки

S длины L. Так как $f_k > f_{k-1} + 1$, то $f_k - f_{k-1} \geq 2$. Кроме того, $len(i) \geq 3$, поэтому в самом худшем случае $m = \left\lfloor \frac{L-3}{2} \right\rfloor$, и расход памяти для хранения значений f_k и $pos(f_k)$ не превышает $2\frac{L}{2} = L$ слов (хранить значения $len(f_k)$ нет необходимости, т. к. $len(f_k) = f_{k+2} - f_k$ по определению функции f).

Можно разбить входную строку на подстроки фиксированного размера и тем самым обеспечить приемлемую границу для максимально возможного значения m.

Однако существует лучшее решение. Ограничим наибольшее возможное значение индекса m некоторым числом M. Заметим, что значение f_k устанавливается в тот момент, когда значение $len(f_{k-2})$ вычислено, и если $f_k \leq f_{k-1} + 1$, то $m = k$ и по накопленным значениям f_k и $pos(f_k)$ производится кодирование подстроки $S[i : f_{m-1} - i]$ в соответствии с теоремой 5.5.

Если значение $k \geq 2$ стало равным M — наибольшему допустимому значению индекса, — то необходимо закодировать подстроку $S[i : f_{m-1} - i]$ точно так же, как если бы выполнялось $f_k \leq f_{k-1} + 1$ (*форсированное кодирование*). Важно заметить, что вычислять значение f_k при $k = M$ *не требуется*, т. к. оно не будет использовано.

Метод такого кодирования при ограничении на объем используемой оперативной памяти назовем методом *почти оптимального сжатия*.

5.2.5. КАЧЕСТВО ПОЧТИ ОПТИМАЛЬНОГО СЖАТИЯ

При форсированном кодировании разность между стоимостью оптимального сжатия и полученным сжатием не превышает 1. Как было показано в теореме 5.6, при нечетном M будет закодировано $\frac{M-1}{2}$ указателей, и поэтому суммарная избыточность стоимости кодирования всей строки S не превысит $2\frac{N_{ptr}}{M-1}$ по сравнению с оптимальным сжатием. При четном M будет закодировано ровно $\frac{M-2}{2}$ указателей, поэтому суммарная ошибка в стоимости кодирования не превысит $2\frac{N_{ptr}}{M-2}$.

Как для четных, так и нечетных M избыточность кодирования не превышает $\varepsilon(M)N_{ptr}$, где

$$\varepsilon(M) = \frac{1}{\left\lceil \frac{M}{2} \right\rceil - 1}. \tag{5.57}$$

Так как длина оптимального сжатого представления есть $N_{lit} + 2N_{ptr}$, то относительная избыточность не превышает

$$\frac{\varepsilon(M)N_{ptr}}{N_{lit} + 2N_{ptr}} \leq \frac{\varepsilon(M)}{2} \leq \frac{1}{M-1}. \tag{5.58}$$

Таким образом, выбрав $M \approx 100 \div 1000$, можно гарантировать, что почти оптимальное сжатия будет хуже оптимального не более чем на $1 \div 0.1\%$.

5.2.6. РЕАЛИЗАЦИЯ

Отметим несколько важных свойств предложенных алгоритмов оптимального и почти оптимального сжатия.

Эти алгоритмы — *однопроходные*, т. к. вычисление значений функции len (поиск подстрок) требуется только в последовательном порядке, слева направо, что является обязательным требованием для многих алгоритмов поиска подстрок.

При заполнении f_k интерес представляют только подстроки такой длины, найденные для позиции f_{k-2}, что

$$f_k = f_{k-2} + len(f_{k-2}) > f_{k-1} + 1. \tag{5.59}$$

Это условие равносильно тому, что при вычислении $len(f_{k-2})$ можно пропускать подстроки, которые не длиннее

$$f_{k-1} + 1 - f_{k-2} \leq 3, \tag{5.60}$$

поэтому проверка на то, что $len(f_k) \geq 3$, не нужна.

Кроме того, во всех случаях при $len(i) \leq 2 \iff len(i) < 3$ нам не нужно знать $pos(i)$. Это позволяет использовать хэширование по первым трем символам подстроки.

Использование хэширования и знания минимальной интересующей длины искомой подстроки может значительно ускорить поиск подстрок и тем самым процесс сжатия в целом.

Как показали эмпирические испытания, ограничения на расход памяти $M = 128$ более чем достаточно для практического применения, т. к. во всех испытаниях m ни разу не достигло этого значения (следовательно, порядок кодирования был оптимальным).

В данной реализации кодировщика предполагается, что i-я подстрока должна быть добавлена к словарю до того, как начнется вычисление $len(i)$ и $pos(i)$ (для реализаций с другими соглашениями кодировщик нетрудно изменить).

```
int  match_len;                /* the value of len(i)     */
int  match_pos;                /* the value of pos(i)     */
void match_get (int i, int n); /* get pos(i) and len(i)>n */
void match_ins (int i, int n); /* insert i-th,..,(i+n-1)  */
void send_lit  (int i);        /* output i-th literal     */
void send_ptr  (int l, int p); /* output pointer <l,p>    */
#define M 128                  /* M=128 usually enough    */
void coder (int L)             /* encode string S[0:L-1]  */
{
  int f[M], p[M], i, k, m;
  for (i = 0; i < L;)
  {
    match_ins (i, 1);
    match_get (i, 2);
    if (match_len <= 2)
      send_lit (i++);
    else
    {
      f[0] = i; f[1] = i+1; m = 2;
      for (;;)
      {
        p[m-2] = match_pos;
        f[m] = f[m-2] + match_len;
        if (f[m] <= f[m-1]+1 || ++m == M) break;
        match_ins (i+1, f[m-2]-i);
        i = f[m-2];
        match_get (i, f[m-1]+1-i);
      }
      if ((m&1) == 0)
        send_lit (f[0]);
      for (k=(m+1)&1; k < m-1; k += 2)
        send_ptr (f[k+2] - f[k], f[k] - p[k]);
```

```
      match_ins (i+1, f[k]-1 - i); i = f[k];
    }
  }
}
```

5.2.7. ЭВРИСТИКИ

Прямолинейное (strait forward) сжатие заключается в том, что для текущего i вычисляется значение $len(i)$ и, если $len(i) \le 2$, кодируется литерал и i увеличивается на 1, а если $len(i) \ge 3$, то кодируется указатель на подстроку $\big(len(i), i - pos(i)\big)$ и i увеличивается на $len(i)$. Следовательно, при прямолинейном сжатии количество поисков подстрок равно

$$T_{min} = N_{lit} + N_{ptr}. \tag{5.61}$$

Прямолинейное сжатие есть не что иное, как почти оптимальное сжатие при $M = 3$. Обычно оптимальное сжатие на 5–10% лучше прямолинейного и на 20–40% медленнее (см. главу 11).

Ленивая оптимизация (lazy evaluation) давно известна разработчикам систем сжатия и используется практически повсеместно (Jean-loup Gially, разработчик архиватора ZIP, считает, что первооткрыватели — Jean-Mark Wams и Леонид Брухис, автор архиватора FREEZE).

Ленивая оптимизация заключается в том, что принятие решения, какое кодирование применить для предыдущей позиции, откладывается на один шаг. Если $len(i) \le len(i-1)$ и $len(i-1) \ge 3$, в позиции $(i-1)$ кодируется указатель на подстроку $\big(len(i-1), (i-1) - pos(i-1)\big)$ и i увеличивается на $len(i-1) - 1$. Если же $len(i) > len(i-1)$ или $len(i-1) \le 2$, то в позиции $i-1$ кодируется литерал и i увеличивается на 1.

Количество поисков подстрок есть 1 на каждый литерал и 2 на каждый указатель, поэтому

$$T_{lazy} = N_{lit} + 2N_{ptr}. \tag{5.62}$$

Заметим, что $T_{lazy} \ge T_{opt}$, несмотря на то, что выражения для T_{lazy} и T_{opt} совпадают, т. к. при оптимальном сжатии длина выходной последовательности, по теореме 5.6 равная T_{opt}, не больше длины выхода при ленивой оптимизации.

Таким образом, ленивая оптимизация только обнаруживает, но не разрешает возможную неоднозначность кодирования. Очень серьезным недостатком такого сжатия является то, что при $2 < len(i) < len(i+1)$ вычисляется значение $len(i+2)$. Из доказательства теоремы 5.4 видно, что значение $cost(i)$ не зависит от $len(i+2)$. Кроме того, обычно вероятность того, что $2 < len(i) < len(i+1) < len(i+2)$, очень мала.

Последнее соображение послужило основой для создания автором в 1992 г. **упрощенного (simplified) варианта ленивой оптимизации**, в котором решение о способе кодирования принимается немедленно при обнаружении неоднозначности кодирования. Очевидно,

$$N_{lit} + N_{ptr} \le T_{simp} \le N_{lit} + 2N_{ptr}. \tag{5.63}$$

Как правило, сжатие с использованием упрощенной ленивой оптимизации весьма незначительно (на доли процента) хуже сжатия с ленивой оптимизацией и на 5–20% быстрее.

Можно заметить, что сжатие с упрощенной ленивой оптимизацией есть частный случай почти оптимального сжатия при $M = 4$.

Как правило, оптимальное сжатие немного — на 2–3%, иногда до 10% — лучше сжатия с использованием ленивой оптимизации и примерно на столько же быстрее (см. главу 11).

5.3. КОДИРОВАНИЕ ПРИ ПРОИЗВОЛЬНОМ $r > 1$

Построим алгоритм выбора оптимальной последовательности кодирования, аналогичный описанному ранее для случая $r = 2$, для произвольных стоимостей кодирования указателей и литералов, равных P и C битов соответственно. Без ограничения общности можно полагать, что стоимость кодирования указателя равна

$$r = \frac{P}{C}, \tag{5.64}$$

а литерала — единице.

Определение 5.6. *Существенной* назовем позицию j, вычисление значений функций $len(j)$ и $pos(j)$ в которой необходимо; если же оптимальная последовательность кодирования может быть определена без вычисления $len(j)$ и $pos(j)$, то позицию j назовем *несущественной*. ■

Определение 5.7. Существенную позицию j, для которой значения функций $len(j)$ и $pos(j)$ еще не вычислены, назовем *активной*; если же $len(j)$ и $pos(j)$ уже известны, то позицию j назовем *пассивной*. ■

Определение 5.8. Обозначим $S[i \dots j]$ подстроку кодируемой строки S, начинающуюся в позиции i и заканчивающуюся в j:

$$S[i \dots j] = S_i \dots S_{j-1}. \tag{5.65}$$

■

Основная идея предлагаемого алгоритма заключается в следующем: проходя по списку активных позиций слева направо, будем вычислять для них значения функций len и pos и, если необходимо, добавлять новые активные позиции до тех пор, пока все активные позиции, кроме одной, не станут пассивными; а затем построим оптимальную последовательность кодирования подстроки, заканчивающейся в последней активной позиции.

Сначала мы опишем алгоритм кодирования, а затем докажем его корректность и проанализируем его свойства.

5.3.1. ОСНОВНОЙ АЛГОРИТМ

Допустим, что кодируемая последовательность закодирована до i-й позиции и необходимо продолжить кодирование, минимизируя как стоимость кодирования оставшейся части данных, так и количество позиций j, для которых необходимо вычислить значения функций $pos(j)$ и $len(j)$, т. е. осуществить поиск совпадающей подстроки.

Сначала необходимо вычислить $len(i)$ и $pos(i)$. Если

$$len(i) \le r \iff len(i) \le \lfloor r \rfloor, \tag{5.66}$$

то кодирование в позиции i литерала оптимально.

Если же

$$len(i) > r \iff len(i) \ge \lfloor r \rfloor + 1, \tag{5.67}$$

то оптимальным может быть как кодирование указателя, так и литерала; возможная неоднозначность кодирования должна быть обнаружена и разрешена.

Установим $a = 0$, $b = 1$, и заполним первые два элемента массивов Val, $Size$, $Link$, Pos, Len и следующим образом:

	0	1
$Link$	$-$	0
$Size$	0	r
Val	i	$i + len(i)$
Len	$-$	$len(i)$
Pos	$-$	$pos(i)$

(5.68)

После этого установим $k = 1$ и выполним следующие шаги:

1. Вычислить $\ell_k = len(Val[a] + k)$, $p_k = pos(Val[a] + k)$ и установить
$$d_k = Val[a] + k + \ell_k - Val[b], \tag{5.69}$$
$$c_k = Size[a] + k + r, \tag{5.70}$$
$$c'_k = Size[b] + \min(d_k, r). \tag{5.71}$$
2. Если $c_k \geq c'_k$, то перейти к шагу 4.
3. Установить
$$Link[b+1] = a, \tag{5.72}$$
$$Size[b+1] = Size[a] + k + r, \tag{5.73}$$
$$Val[b+1] = Val[a] + k + \ell_k, \tag{5.74}$$
$$Len[b+1] = \ell_k, \tag{5.75}$$
$$Pos[b+1] = p_k, \tag{5.76}$$
и увеличить b на единицу.
4. Увеличить k на единицу.
5. Если $k < Size[a+1] - Size[a]$, то перейти к шагу 1.
6. Установить $k = 0$ и увеличить a на единицу.
7. Если $a < b$, то перейти к шагу 1.
8. Установить $e = b$ и закончить алгоритм.

Позднее будет показано, что при $j > 0$ значение $Size[j]$ есть не что иное, как стоимость оптимального кодирования подстроки $S[i \ldots Val[j]]$, т. е.

$$cost(i) = Size[j] + cost(Val[j]), \tag{5.77}$$

поэтому вычисление $cost(Val[e])$ *не требуется*, т. к. оптимальная последовательность кодирования подстроки $S[i \ldots Val[e]]$ *не зависит* от $cost(Val[e])$ и определяется следующим рекурсивным образом: закодировать подстроку $S[i \ldots Val[e']$, где $e' = Link[e]$, затем

$$k = Val[e] - Val[e'] - Len[e] \tag{5.78}$$

литералов $S_{Val[e']}, \ldots, S_{Val[e']+k-1}$, а потом — указатель

$$\langle Len[e], Pos[e] \rangle. \tag{5.79}$$

На практике рекурсия не нужна, т. к. оптимальную последовательность кодирования нетрудно построить инвертированием списка указателей:

```
void encode (int e)
{
  int n, k;
  /* invert the list */
  n = -1;
  do
  {
```

```
      k = Link[e];
      Link[e] = n;
      n = e;
      e = k;
    }
    while (e > 0);
    /* walk through the list and encode */
    do
    {
      n = Val[n] - Val[e] - Len[n];
      for (k = 0; k < n; ++k)
        send_lit (S[ k + Val[e] ]);
      e = n;
      send_ptr (Len[e], Pos[e]);
      n = Link[e];
    }
    while (Link[e] >= 0);
  }
```

5.3.2. АНАЛИЗ ОСНОВНОГО АЛГОРИТМА

Покажем, что описанный алгоритм в каждой активной позиции $Val[j]$ вычисляет стоимость оптимального кодирования подстроки $S[i \ldots Val[j]]$, причем значения активных позиций монотонно возрастают. Кроме того, покажем, что этот алгоритм вычисляет значения функций len и pos только для существенных позиций.

Определение 5.9. Для краткости обозначений введем функцию $\mathcal{F}(j) = j + len(j)$.

■

5.3.2.1. ИНВАРИАНТНЫЕ СООТНОШЕНИЯ

Теорема 5.7. Для всех $j = 1, \ldots, b-1$

$$Size[j+1] - Size[j] < r, \tag{5.80}$$

а при $j = 0$

$$Size[1] - Size[0] = r. \tag{5.81}$$

Доказательство. Действительно, новая активная позиция добавляется тогда и только тогда, когда

$$Size[b+1] = c_k < c'_k = Size[b] + \min(d_k, r) \le Size[b] + r; \tag{5.82}$$

справедливость соотношения $Size[1] - Size[0] = r$ гарантируется построением алгоритма ($Size[1] = r$, $Size[0] = 0$). ■

Теорема 5.8. При $0 \le j \le b-1$

$$Size[j+1] - Size[j] < Val[j+1] - Val[j]. \tag{5.83}$$

Доказательство. При $j = 0$

$$Size[j+1] - Size[j] = r < Val[j+1] - Val[j], \tag{5.84}$$

а при $j > 0$ по построению алгоритма

$$Size[j+1] - Size[j] < \min(Val[j+1] - Val[j], r) \le Val[j+1] - Val[j]. \tag{5.85}$$

■

Итак, если $k < Size[a+1] - Size[a]$, то $Val[a] + k < Val[a+1]$, т. е. предложенный алгоритм является *однопроходным* и вычисляет значения функций len и pos слева направо, без возвращений назад.

Теорема 5.9.

$$d_k \geq 0. \tag{5.86}$$

Доказательство. Обозначим $a' = Link[a]$, тогда

$$Val[b] = \mathcal{F}\big(Val[a'] + k'\big), \tag{5.87}$$

где

$$k' = Val[b] - Val[a'] - Len[b], \tag{5.88}$$

причем (как следствие теоремы 5.8)

$$Val[a'] + k' < Val[a] + k < Val[b]. \tag{5.89}$$

По определению функции len, если

$$m \leq n \leq \mathcal{F}(m), \tag{5.90}$$

то

$$len(n) \geq \mathcal{F}(m) - n. \tag{5.91}$$

Приняв $n = Val[a] + k$ и $m = Val[a'] + k'$, имеем

$$\ell_k \geq Val[b] - (Val[a] + k), \tag{5.92}$$

а т. к.

$$d_k = Val[a] + k + \ell_k - Val[b], \tag{5.93}$$

то утверждение доказано. ■

Теорема 5.10. При $0 \leq j \leq b-1$

$$Size[j] < Size[j+1]. \tag{5.94}$$

Доказательство. При $j = 0$ это выполняется, т. к. $Size[0] = 0$ и $Size[1] = r$. При $j > 0$ рассмотрим m и m' такие, что

$$Val[j] = \mathcal{F}(Val[m] + k), \tag{5.95}$$

$$Val[j+1] = \mathcal{F}(Val[m'] + k'); \tag{5.96}$$

причем по теореме 5.8

$$Val[m] + k < Val[m'] + k'. \tag{5.97}$$

Если $m = m'$, то $Size[m] = Size[m']$ и $k < k'$, поэтому

$$Size[j] = Size[m] + k + r < Size[m] + k' + r = Size[j+1]. \tag{5.98}$$

Если же $m < m'$, то по выбору k

$$Size[m] + k < Size[m+1] \leq Size[m'], \tag{5.99}$$

поэтому

$$Size[j] = Size[m] + k + r < Size[m+1] + r \leq Size[m'] + k' + r = Size[j+1], \tag{5.100}$$

что и требовалось доказать. ■

Теорема 5.11. Для того, чтобы добавилась новая активная позиция, необходимо, чтобы длина найденной подстроки ℓ_k удовлетворяла условию

$$\ell_k > (Val[b] - Val[a]) - (Size[b] - Size[a]) + r > r. \tag{5.101}$$

Доказательство. Так как c_k должно быть меньше c'_k, имеем

$$Size[a] + k + r < Size[b] + \min(r, d_k) \iff \tag{5.102}$$

$$\iff \max(k, k + r - Val[a] - k - \ell_k + V[b]) < Size[b] - Size[a]. \tag{5.103}$$

По выбору k и вследствие теоремы 5.10

$$k < Size[a+1] - Size[a] \le Size[b] - Size[a], \tag{5.104}$$

поэтому должно выполняться

$$r - Val[a] - \ell_k + V[b] < Size[b] - Size[a], \tag{5.105}$$

откуда и следует первое доказываемое неравенство.

Осталось заметить, что согласно теореме 5.8

$$Size[j+1] - Size[j] < Val[j+1] - Val[j], \tag{5.106}$$

поэтому

$$V[b] - V[a] > Size[b] - Size[a], \tag{5.107}$$

откуда следует последнее доказываемое неравенство. ■

5.3.2.2. МИНИМАЛЬНОСТЬ СТОИМОСТИ КОДИРОВАНИЯ

Теорема 5.12. Если

$$len(j) \le r \iff len(j) \le \lfloor r \rfloor, \tag{5.108}$$

то в позиции j оптимально кодирование литерала, а если

$$len(j) > r \iff len(j) \ge \lfloor r \rfloor + 1, \tag{5.109}$$

то оптимальное кодирование начинается $k \le \lceil r \rceil - 1$ литералов, за которыми *обязательно* следует указатель.

Доказательство. По определению функции $cost$,

$$cost(j) = \min_{k \ge 1} \{ r + cost(j + len(j)), k + cost(j + k) \}. \tag{5.110}$$

Если $len(j) \le r$, то

$$r + cost(j + len(j)) \le len(j) + cost(j + len(j)) \tag{5.111}$$

и кодирование в позиции j указателя заведомо не лучше кодирования литерала.

По определению функции $cost$

$$cost(m) \le n + cost(m + n), \tag{5.112}$$

поэтому при $k \ge len(j) > r$ ($m = j + len(j)$ и $n = k - len(j)$)

$$cost(j + len(j)) \le k - len(j) + cost(k + j) \tag{5.113}$$

и

$$r + cost(j + len(j)) \le len(j) + cost(j + len(j)) \le k + cost(j + k), \tag{5.114}$$

т. е. кодирование, начинающееся с r и более литералов, заведомо не лучше начинающегося с указателя.

Таким образом, существует наибольшее k ($0 \le k < r$) такое, что

$$cost(j) = k + cost(j + k) < k + 1 + cost(j + k + 1), \tag{5.115}$$

поэтому при $len(j) > r$ оптимальное кодирование начинается с k литералов, за которыми обязательно следует указатель. Действительнom, оптимальное кодирование не может начинаться с $(k+1)$ литералов, поскольку $cost(j) < (k+1) + cost(j+k+1)$.

■

Теорема 5.13. Описанный алгоритм перебирает *все* существенные позиции.

Доказательство. Согласно теореме 5.12, для поиска оптимального кодирования в активной позиции $Val[a]$ достаточно перебрать все возможные способы кодирования, начинающиеся $k = 0, \ldots, \lceil r \rceil - 1$ литералов, за которыми следует указатель.

Прямолинейное применение такого способа оптимизации нежелательно, т. к. потребует вычисления значений функций len и pos во всех позициях. Чтобы сократить перебор, воспользуемся рядом соображений.

Во-первых, если длина ℓ_k найденной в позиции $Val[a] + k$ подстроки недостаточна, то позиция

$$F = \mathcal{F}(Val[a] + k) \tag{5.116}$$

не существенная. Недостаточность длины определяется просто. Согласно теореме 5.9, $F \geq Val[b]$, где $Val[b]$ — последняя активная в настоящий момент позиция. Подстрока $S[i \ldots F]$ может быть может быть закодирована как минимум двумя различными способами. Можно сначала закодировать подстроку $S[i \ldots Val[a]]$, затем k литералов, а потом указатель на подстроку $< \ell_k, p_k >$; стоимость такого кодирования равна

$$c_k = Size[a] + k + r. \tag{5.117}$$

Другое способ — сначала закодировать подстроку $S[i \ldots Val[b]]$, а затем — d_k литералов или указатель на подстроку длины $d_k = F - Val[b]$; стоимость такого кодирования равна

$$c'_k = Size[b] + \min(d_k, r). \tag{5.118}$$

Очевидно, если $c_k \geq c'_k$, то первый способ заведомо не лучше второго и не нуждается в дальнейшем рассмотрении, поэтому при $c_k \geq c'_k$ позиция F — несущественная.

Во-вторых, перебор нужно прекратить не когда k превысит $\lceil r \rceil - 1$, а при $k \geq Size[a+1] - Size[a]$.

С одной стороны, согласно теореме 5.7 $Size[a+1] - Size[a] \leq r$, поэтому такое ограничение заведомо не увеличивает перебор по k.

С другой стороны, если $k \geq Size[a+1] - Size[a]$, то оптимальное кодирование не может начинаться с k литералов. Действительно, если $F = \mathcal{F}(Val[a] + k) \leq Val[a+1]$, то подстрока $S[i \ldots Val[a+1]]$ может быть закодирована со стоимостью $Size[a+1]$, тогда как стоимость кодирования такой же или более короткой подстроки $S[i \ldots F]$ равна $Size[a] + k + r \geq Size[a+1]$, поэтому второй способ кодирования заведомо не лучше.

Если же $F > Val[a+1]$, то подстрока $S[i \ldots F]$ может быть закодирована двояко. Первый способ — закодировать подстроку $S[i \ldots Val[a]]$, затем k литералов, а потом — указатель; стоимость такого кодирования равна

$$c_k = Size[a] + k + r. \tag{5.119}$$

Другой способ — закодировать $S[i \ldots Val[a+1]]$, а затем $(F - Val[a+1])$ литералов или указатель на подстроку длины $(F - Val[a+1])$; стоимость такого кодирования равна

$$c'_k = Size[a+1] + \min(F - Val[a+1], r) \leq Size[a+1] + r. \tag{5.120}$$

Так как $Size[a] + k > Size[a+1]$, то

$$c_k = Size[a] + k + r > Size[a+1] + r \geq c'_k. \tag{5.121}$$

Итак, при $k \geq Size[a+1] - Size[a]$ кодирование подстроки $s(Val[a])$, за которым следует k литералов, заведомо не оптимально, поэтому перебор можно прекращать раньше. ■

Таким образом, значение $Size[j]$ есть не что иное, как минимальная стоимость кодирования подстроки $S[i \ldots Val[j]]$, поэтому если осталась *единственная* активная позиция e,

$$cost(i) = Size[e] + cost(Val[e]), \tag{5.122}$$

поэтому можно закодировать оптимальным образом подстроку $S[i \ldots Val[e]]$, *не вычисляя* $cost(Val[e])$, после чего установить i в $Val[e]$ и продолжить кодирование.

5.3.2.3. МИНИМАЛЬНОСТЬ ПЕРЕБОРА

Описанный алгоритм не только перебирает все существенные позиции, но и не затрагивает лишних.

Теорема 5.14. Все позиции, для которых алгоритм вычисляет значения функций len и pos, существенны.

Доказательство. Достаточно заметить, что если $F = \mathcal{F}(V[a]+k) > V[b]+r$ и для всех $j = V[a]+k+1, \ldots, F-1$

$$\mathcal{F}(j) \le F, \tag{5.123}$$

то оптимальное кодирование i-го суффикса S начинается с оптимального кодирования $S[i \ldots V[a]]$, за которым следуют k литералов и указатель на подстроку $< \ell_k, p_k >$, поэтому знание значений функций len и pos в позиции $Val[a]+k$ необходимо. ■

Замечание 5.4. Перебор можно сократить еще больше, если разрешить заглядывания вперед и аккуратно отслеживать особый случай, описанный в теореме 5.14.

Для каждой активной позиции $Val[j]$ сразу же вычислим $Len'[j] = len(Val[j])$ (если $len(Val[j]) \le r$, то будем считать, что $Len'[j] = 0$). Если на каком-то шаге $d_k \le Len'[b]$, то построение оптимального кодирования подстроки $S[i \ldots F]$ закончено, т. к. по определению функции len для всех $j = V[a]+k+1, \ldots, F-1$

$$\mathcal{F}(j) \le F. \tag{5.124}$$

■

К сожалению, автору не удалось получить при произвольном $r > 1$ оценку сложности алгоритма построения оптимальной последовательности кодирования в виде зависимости от длины LZ77-кода, как это было сделано при $r = 2$ (см. теорему 5.6).

5.3.3. ПРАКТИЧНАЯ ВЕРСИЯ ОСНОВНОГО АЛГОРИТМА

С целью ограничения объема требуемой памяти зададимся некоторым значением M и заменим шаг 5 следующими:

5_1 Если $b = M-1$, то перейти к шагу 8.

5_2 Если $k < Size[a+1] - Size[a]$, то перейти к шагу 1.

Таким образом, практичной версии вышеописанного алгоритма достаточно $5M$ слов памяти.

5.3.4. ИЗБЫТОЧНОСТЬ ПРАКТИЧНОГО АЛГОРИТМА

Если объем выделенной алгоритму памяти исчерпан, т. е. b стало равным M, практичная версия основного алгоритма производит форсированное кодирование, которого

хуже оптимальной менее чем на $r-1$. Так как кодируемая последовательность содержит ровно $(M-1)$ указателей, ее стоимость не меньше $r(M-1)$. Таким образом, относительная избыточность меньше

$$\frac{1}{M-1}, \tag{5.125}$$

поэтому при $M \approx 100 \div 1000$ качество кодирования практичного алгоритма уступает оптимальному менее 1–0.1%.

Автором был проведен эксперимент по сжатию около 50 тыс. различных файлов с данными, хранимых на одном из крупных узлов Internet. В ходе этого эксперимента значение $M = 128$ достигалось лишь единожды; таким образом, во всех остальных случаях кодирование было оптимальным.

5.3.5. СВОЙСТВА АЛГОРИТМОВ

Все свойства алгоритмов кодирования при произвольном r не отличаются от свойств описанных ранее алгоритмов для $r = 2$. Перечислим самые важные из них:

- вычисление значений pos и len производится последовательно, без возвращений назад (см. теорему 5.8);
- перед каждым обращением к pos и len известна минимальная длина интересующей строки (см. теорему 5.11), что может существенно использоваться процедурой поиска для ускорения перебора;
- указатели на подстроки длины $\lfloor r \rfloor$ и менее никогда не кодируются, что позволяет организовать хэширование по первым $(\lfloor r \rfloor + 1)$ символам подстрок.

Более того, можно показать, что предложенные алгоритмы при $r = 2$ являются частным случаем описанных алгоритмов для произвольного r.

5.4. КОДИРОВАНИЕ ПРИ ПРОИЗВОЛЬНОЙ СТОИМОСТИ

Мы не будем подробно описывать построение алгоритма оптимального алгоритма для случая, когда стоимость кодирования зависит от кодируемого элемента. Отметим лишь, что основные принципы его организации следуют из теоремы 5.13 с тем отличием, что условия прекращения перебора по k и условия порождения новых активных позиций необходимо изменить так, чтобы учитывать точную стоимость кодирования конкретных литералов и/или указателей.

Основная сложность реализации такого обобщенного алгоритма заключается в необходимости достаточно точной оценки стоимостей кодирования (в особенности при использовании гибридных схем кодирования, описанных в главе 7). Так как символ с вероятностью появления p кодируется не менее чем $\log_2 \frac{1}{p}$ битами, эту формулу можно использовать для вычисления оценки. Поскольку время ее вычисления очень значительно и должно проводиться в формате с плавающей точкой, скорость такого кодирования очень низкая; с практической точки зрения применение такого способа оптимизации не оправдано.

5.5. ДРУГИЕ МЕТОДЫ УЛУЧШЕНИЯ КАЧЕСТВА СЖАТИЯ

5.5.1. ИСПОЛЬЗОВАНИЕ МЕТОДА СТОПКИ КНИГ

При кодировании двоичных файлов (исполняемых программ, объектных модулей и их библиотек и т. д.) достаточно часто встречаются похожие подстроки, т. е. две достаточно длинные последовательности символов, отличающиеся в середине одним или двумя символами.

Для того, чтобы эффективно кодировать ссылки на такие подстроки, можно воспользоваться методом стопки книг (см. п. 2.6.7.2) или, что то же самое, LRU-очередью (от англ. Last Recently Used) для хранения смещений к началу последних найденных n подстрок (где n — небольшое целое число порядка 3–8-ми), и если текущее смещение p к началу текущей подстроки совпадает с i-м смещением, находящимся в стопке книг, то вместо p кодируется индекс смещения i в стопке книг. Если же текущее смещение p не обнаружено в стопке книг, то вместо смещения p кодируется смещение $(p + n)$, чтобы можно было отличать индексы в стопке книг от собственно смещений. После того, как смещение p закодировано, оно помещается в начало стопки книг с индексом 0 (возможно, с удалением последнего элемента стопки).

Таким образом, если смещения (стоимость кодирования которых составляет большую часть стоимости кодирования указателя) достаточно часто совпадают с находящимися в стопке книг значениями, то качество сжатия улучшается. В связи с тем, что стоимость кодирования индекса в стопке книг обычно в 3–5 раз ниже стоимости кодирования собственно смещения, имеет смысл принудительно увеличивать коэффициент использования смещений из стопки книг. Для этого перед началом поиска подстроки следует сравнить начало текущего суффикса кодируемой строки с подстроками, смещения к началу которых расположены в стопке книг, и при обнаружении совпадения достаточной длины (начиная с 3–4 символов) принудительно считать искомую подстроку найденной.

Эмпирические исследования автора показали, что применение такого способа кодирования смещений заметно — в среднем на 5–7%, иногда до 10–15%, — улучшает качество сжатия двоичных файлов, не влияя на качество сжатия текстовой информации, для которой вероятность появления похожих строк очень мала.

Вместо метода стопки книг LRU можно использовать деки или, что то же самое, LIFO-очередь (Last In First Out), что незначительно упрощает модификацию последовательности закодированных смещений, но ухудшает качество сжатия по сравнению с LRU.

Насколько известно автору, описание этого метода кодирования смещений никогда не появлялось в открытой печати. Видимо[2], впервые аналогичный описанному метод кодирования смещений был использован в 1995–1996 гг. Евгением Рошалом в архиваторе RAR версии 2.0 и позднее — в 1996 г. — был применен автором в разработанной им библиотеке сжатия данных PRS, а в 1997 г. — Иваном Павловым в архиваторе UFA32.

5.5.2. АЛГОРИТМ LZWW И ЕГО ВАРИАНТЫ

В 1995 г. Винерами[3] [196] был описан достаточно интересный метод кодирования

[2] Исходные тексты декодирования RAR 2.0 и выше — утилиты UNRAR 2.0 — с 1997 г. свободно распространяются; информация относительно UFA32 была получена автором путем сжатия специально подобранных тестов.

[3] Строго говоря, Винеры не являются первооткрывателями; аналогичные методы кодирования активно обсуждались в 1991–1992 г. в группе comp.compression, а в 1993 г. J.Y. Lim разработал архиватор LIMIT, использующий этот метод кодирования.

длин строк, который основан на следующем наблюдении.

Если первые k символов подстрок, начинающихся в позициях i_1 и i_2 ($i_1 < i_2$), совпадают (т. е. $S[i_1 : k] = S[i_2 : k]$), а k-ые символы отличаются ($S[i_1 + k] \neq S[i_2 + k]$), то если имеется ссылка на подстроку, начинающуюся в i_1-ой позиции, то длина ℓ указанной подстроки должна превышать k, т. к. иначе была бы закодирована ссылка на более правую подстроку, начинающуюся в позиции $i_2 > i_1$.

Таким образом, если для каждой позиции i, которая может быть указана в качестве начала подстроки, известна длина ℓ_i расположенной правее подстроки, совпадающей с i-й по наибольшему количеству первых символов, то при кодировании ссылки на подстроку длины ℓ, начинающуюся в i-й позиции, можно вместо ℓ закодировать $\ell' = \ell - \ell_i$, т. к. если декодировщик будет поддерживать последовательность ℓ_i таким же образом, как и кодировщик, он сможет восстановить действительную длину ℓ указанной подстроки по формуле $\ell = \ell' + \ell_i$.

Замечание 5.5. Если алгоритм сжатия допускает подстроки длины не менее n, то вместо ℓ_i в вышеуказанных формулах следует использовать $\ell'_i = \max\{n, \ell_i\}$. ■

Имея последовательность ℓ_i, вместо модификации длины указанной подстроки можно изменять значение смещения. Действительно, если длина указанной подстроки равна ℓ, то она может начинаться в позиции i тогда и только тогда, когда $\ell_i < \ell$, поэтому вместо смещения можно использовать количество позиций j таких, что $\ell_j < \ell$, расположенных между найденной подстрокой и текущей позицией.

Использование таких методов позволяет улучшить качество сжатия в среднем на 2–4% при существенно большей сложности кодирования и, самое главное, декодирования, вызванных необходимостью вычисления значений ℓ_i (и с этой точки зрения уменьшение значения кодируемого смещения вообще непригодно для практического использования). Действительно, построение точных ℓ_i требует перебора всех строк, которые имеют общее начало с текущей строкой, что усложняет алгоритм поиска подстрок и, самое главное, требует его применения в каждой позиции.

Согласно эмпирическим исследованиям автора, применение алгоритма Винеров замедляет кодирование в 5–10 раз, а декодирование — более чем в 100 (!) раз, тем самым сводя на нет два основных преимущества алгоритмов семейства LZ77 — высокую скорость кодирования и чрезвычайно высокую скорость декодирования. По мнению автора, такое радикальное ухудшение производительности не окупается незначительным улучшением качества сжатия на 2–4%.

5.5.2.1. ПРАКТИЧНЫЙ МЕТОД РЕАЛИЗАЦИИ

В 1992 г. автором был предложен практичный способ уменьшения длины указанной строки, который основан на том, что значения последовательности ℓ_i вычисляются не точно, а приближенно. При кодировании ссылки на строку длины ℓ, начинающуюся в i-й позиции, кодируется длина $\ell' = (\ell - \ell_i)$, после чего для $j = 0, \ldots, \ell - 1$ значение ℓ_{i+j} заменяется на $\max\{\ell_{i+j}, \ell - j, n\}$; декодировщик поступает аналогичным образом.

Улучшение качества сжатия, получаемое при использовании приближенного решения, несколько меньше, чем при точном решении, и по сравнению с традиционной версией LZ77 составляет (очень стабильно) около 1% при замедлении кодирования на 3–5% и на 15–25% — декодирования (см. главу 11). По мнению автора, такое незначительное улучшение качества сжатия не оправдывается большим объемом требуемой памяти и замедлением кодирования и декодирования.

Насколько известно автору, архиватор LIMIT (автор — J.Y. Lim) — единственная система сжатия, использующая метод Винеров (в его практичном варианте).

5.6. ВЫВОДЫ

Основное отличие разработанных алгоритмов от известных методов решения этой задачи (а именно, алгоритма Миллера — Вегмана) заключается в том, что:

- Значения функций len и pos вычисляются только в тех позициях, где это действительно необходимо (а не во всех позициях). Так как вычисление значения этих функций, которое сводится к поиску подстрок, занимает подавляющую часть (60–90%) времени работы алгоритма сжатия, уменьшение количества выполняемых поисков подстрок позволяет заметно — в 3–6 раз — ускорить процесс сжатия.
- Требуемый объем оперативной памяти при почти оптимальном кодировании известен заранее и не зависит от входных данных. При почти оптимальном сжатии относительная избыточность кодирования по сравнению с оптимальным сжатием может быть сделана сколь угодно малой и обратно пропорциональна объему доступной алгоритму памяти.

Доказано, что почти все применяющиеся эвристические методы управления порядком кодирования являются частными случаями алгоритма почти оптимального сжатия с постоянной стоимостью кодирования указателей и литералов при $r = 2$. Как правило, качество сжатия при оптимальном сжатии и энтропийном кодировании выхода LZ77, описанном в главе 7, на 2—5% лучше, чем при использовании эвристических методов (см. п. 5.2.7), а скорость сжатия такая же или немного лучше, чем у наиболее часто применяемых эвристик (см. результаты сравнения в главе 11).

Использование специфических особенностей алгоритма почти оптимального сжатия, перечисленных в п. 5.2.6, позволяет значительно ускорить поиск подстрок и собственно процесс сжатия (см. главы 6 и 11).

Глава 6

БЫСТРЫЕ АЛГОРИТМЫ ПОИСКА ПОДСТРОК В LZ77-СХЕМАХ

Данная глава во многом опирается на главу 5; как следствие, основные понятия, определения и терминология совпадают.

В дальнейшем для краткости будем использовать термин *образец* для обозначения i-го суффикса входной строки $S[i : L-1]$, для которого вычисляются значения функций $len(i)$ и $pos(i)$.

В работах [37, 40, 86] приводится подробный обзор методов поиска подстрок, применяемых в LZ77-схемах с шириной окна $W \leq 2^{13} = 8196$. В современных промышленных системах сжатия для улучшения качества последнего, как правило, используется окно шириной в $2^{15} = 32768$ и $2^{16} = 65536$ символов, поэтому объем перебора возрастает в 4–16 раз. Для того чтобы скорость сжатия оставалась приемлемой, необходимо использовать другие методы поиска подстрок и, соответственно, другие структуры данных. Многие из предложенных методов ускорения поиска и эвристик ранее не применялись.

В главе 5 построены алгоритмы оптимального кодирования для сжатия по методу LZ77, обладающие рядом важных свойств:

1. Множество позиций i таких, где необходимо знание $len(i)$ и $pos(i)$, пропорционально длине сжатого представления строки (в количестве символов алфавита B); если знание $len(i)$ не нужно, то необходимо только добавить i-ю подстроку к структуре данных, предназначенных для поиска подстрок.
2. Значения функции $len(i)$ вычисляются последовательно, слева направо.
3. Во время вычисления $len(i)$ заведомо известна минимальная длина подстроки (не меньше трех символов), интересующей кодировщик; алгоритм поиска может пропускать подстроки, которые являются более коротким началом образца, т. к. это не скажется на качестве кодирования.
4. Скорость оптимального сжатия лучше, чем у применяющихся эвристических методов управления порядком кодирования.

При построении алгоритма поиска подстрок мы постараемся максимально использовать всю доступную информацию для того, чтобы повысить скорость сжатия.

6.1. МЕТОДЫ ПОИСКА ПОДСТРОК

На множестве символов входного алфавита естественным образом определяется отношение порядка: $a_i \leq a_j \iff i \leq j$. В свою очередь, отношение порядка на символах задает отношение лексикографического порядка на множестве подстрок $S[k : L-k]$.

Рассмотрим в качестве ключа, заданного k-й подстрокой, k-ый суффикс кодируемой строки $S[k : L-k]$. Так как все ключи различны, то поиск самой длинной и самой

правой подстроки, являющейся началом образца, заключается в нахождении *самой правой* подстроки, *наиболее близкой* к образцу в смысле длины общего начала. Таким образом, для поиска подстрок можно применять любые методы поиска ключа, самого близкого к данному: если длина общего начала с образцом одинакова у нескольких ключей, то искомый — самый правый из них.

Для методов поиска подстрок и соответствующих структур данных, которые хорошо изучены, рассмотрены только их сильные и слабые стороны. Подробное описание этих методов приводится в процитированной литературе.

6.1.1. ДЕРЕВЬЯ ЦИФРОВОГО ПОИСКА

Алгоритм $PATRICIA$ [18, 150] изначально предназначен для решения проблемы поиска самой длинной подстроки, являющейся началом данной. Так как $PATRICIA$ ориентирован для поиска бинарных подстрок, используются его модификации: k-арный вариант $PATRICIA$ — структура данных *trie* [155] (от англ. *retrieve* — извлекать) или дерево суффиксов [132, 177], для которых время поиска почти константно (при условии, что поиск осуществляется во всех позициях последовательно, слева направо).

Поскольку точный поиск подстрок с использованием деревьев цифрового поиска оптимален по времени, эти методы поиска хорошо изучены [31, 33, 129, 132, 136, 150, 151, 155, 177]. К сожалению, они обладают одним очень серьезным недостатком — высокой сложностью структур данных — со всеми вытекающими отсюда последствиями: большим объемом требуемой памяти и высокой сложностью алгоритмов модификации этих структур данных при добавлении и в особенности при удалении элементов, поэтому во многих случаях использование более простых структур данных в сочетании с хэшированием дает заметно лучшие результаты — как по времени поиска, так и по объему требуемой памяти.

Самый главный недостаток деревьев цифрового поиска заключается в том, что добавление новой подстроки ничем не отличается от поиска. С учетом того, что при сжатии данных (например, текстов на естественных языках –см. [39, 166] и главу 5) с использованием алгоритмов семейства LZ77 добавления и удаления подстрок производятся в 5–10 раз чаще чем собственно поиск, это обстоятельство является существенным недостатком таких методов. Кроме того, во время поиска обычно известна минимальная длина подстроки, представляющая интерес для алгоритма сжатия (см. главу 5), но эта информация не может быть использована для ускорения поиска.

Несмотря на оптимальность, методы поиска подстрок, основанные на использовании деревьев суффиксов, почти никогда не используются на практике ввиду низкой производительности и большого объема требуемой памяти (см. [39, 40, 86, 167] и главу 11).

6.1.2. БИНАРНЫЕ ДЕРЕВЬЯ

На множестве подстрок $S[k : L - i]$ существует естественное отношение лексикографического порядка, что позволяет использовать бинарные деревья поиска [167]. Заметим, что при сравнении образца с k-й подстрокой длина общего начала — побочный результат сравнения подстрок.

Применение деревьев — сбалансированных, несбалансированных [2, 18], самонастраиваемых (self-adjusting) [108, 165] — в целом дает лучшие результаты, чем использование деревьев цифрового поиска: удаление происходит заметно быстрее, а процесс добавления и поиска подстроки — немного быстрее.

В целом бинарные деревья наследуют главный недостаток деревьев цифрового поиска: операция добавления новой подстроки по сложности такая же, как и поиск; плохая эффективность в среднем; знание минимальной длины искомой подстроки бесполезно.

6.1.3. ХЭШИРОВАНИЕ

Хэширование традиционно применяется для поиска информации [18], т. к. позволяет снизить объем необходимого перебора в сотни и тысячи раз.

Обозначим $h(a_{i_0}, \ldots, a_{i_{n-1}})$ хэш-функцию, построенную по первым n символам подстроки. Если требуется найти самую длинную (длиной не менее n символов) и самую правую подстроку в окне ширины W, являющуюся началом образца, то перебор можно сузить до множества подстрок, хэш-сумма которых совпадает с хэш-суммой образца.

Так как в нашем случае все ключи различны, то количество коллизий может быть значительным, и для их разрешения необходимо использовать списки или деревья. Использование для этих целей бинарных деревьев приводит к худшим результатам, чем использование списков, т. к. средняя длина цепочки мала и затраты на поддержание более сложных структур данных себя не оправдывают [18].

Как показано в [40], самым быстрый из методов поиска подстрок — прямой перебор подстрок хэш-списка, построенного по первым двум символам подстрок. В данной работе рассмотрены гораздо более быстрые методы перебора подстрок, основанные на той же идее: прямой перебор подстрок хэш-списка.

6.2. СРЕДНЕЕ ВРЕМЯ ПЕРЕБОРА ХЭШ-СПИСКА

Как правило, при оценке эффективности хэширования рассматриваются равномерные хэш-суммы и равномерное распределение ключей [18]. При таких предположениях средняя мощность множества элементов хэш-таблицы, имеющих одинаковую хэш-сумму, примерно равна коэффициенту загруженности хэш-таблицы $\alpha = \frac{W}{H}$, где H — размер, а W — количество элементов хэш-таблицы.

К сожалению, в нашем случае все обстоит иначе, потому что распределение ключей, имеющих одинаковую хэш-сумму, сильно отличается от равномерного. Обозначим $p_{i_0 \ldots i_{n-1}}$ вероятность того, что подстрока начинается с символов $a_{i_0} \ldots a_{i_{n-1}}$.

Будем считать, что h — *идеальная* хэш-функция, т. е. значения h на двух наборах из n символов совпадают тогда и только тогда, когда совпадают сами наборы символов. При таких предположениях в хэш-таблице находится в среднем

$$N_{i_0 \ldots i_{n-1}} = W p_{i_0 \ldots i_{n-1}} \tag{6.1}$$

подстрок, начинающихся с символов $a_{i_0} \ldots a_{i_{n-1}}$. Следовательно, среднее время перебора всех подстрок с одинаковой хэш-суммой (с учетом их вероятностей) пропорционально

$$P_n = \sum_{i_j=0}^{N-1} p_{i_0 \ldots i_{n-1}} N_{i_0 \ldots i_{n-1}} = \sum_{i_j=0}^{N-1} W p_{i_0 \ldots i_{n-1}}^2, \tag{6.2}$$

где N — мощность входного алфавита.

Предположим, что $p_{i_0 \ldots i_{n-1}} \approx p_{i_0} \ldots p_{i_{n-1}}$, тогда

$$P_n = \sum_{i_j=0}^{N-1} W p_{i_0 \ldots i_{n-1}}^2 \tag{6.3}$$

$$\approx \quad W \sum_{i_j=0}^{N-1} \left(p_{i_0} \dots p_{i_{n-1}}\right)^2 = W \sum_{i_j=0}^{N-1} p_{i_0}^2 \dots p_{i_{n-1}}^2 \tag{6.4}$$

$$= \quad W \sum_{i_0=0}^{N-1} p_{i_0}^2 \sum_{i_j=0}^{N-1} p_{i_1}^2 \dots p_{i_{n-1}}^2 \tag{6.5}$$

$$= \quad W \sum_{i_0=0}^{N-1} p_{i_0}^2 \sum_{i_1=0}^{N-1} p_{i_1}^2 \sum_{i_j=0}^{N-1} p_{i_2}^2 \dots p_{i_{n-1}}^2 \tag{6.6}$$

$$\dots \tag{6.7}$$

$$= \quad W \sum_{i_0=0}^{N-1} p_{i_0}^2 \sum_{i_{n-1}=0}^{N-1} p_{i_{n-1}}^2 = W \left(\sum_{i=0}^{N-1} p_i^2 \right)^n . \tag{6.8}$$

Итак, P_n с ростом n убывает как θ^n, где

$$\theta = \sum_{i=0}^{N-1} p_i^2 . \tag{6.9}$$

Согласно исследованию, проведенному автором, для большинства данных — текстов на естественных языках и языках программирования, объектных файлов и их библиотек, образов исполняемых задач и т. д., — значение θ находится в пределах от 0.05 до 0.075. Поскольку реальные данные порождаются не источником Бернулли, при $n \geq 5$ соотношение $p_{i_0 \dots i_{n-1}} \approx p_{i_0} \dots p_{i_{n-1}}$ не верно, однако при небольших $n \leq 4$ реальное значение P_n хорошо коррелирует с приближенной оценкой $W\theta^n$ и обычно отличается от нее не более чем на 30% (хотя, конечно, нетрудно привести пример, когда это будет не так уже при небольших n).

Обычно требуется искать подстроки длины 3 и более, следовательно, при $n = 3$ и ширине окна в 32–64 Кб ($W = 2^{15} \div 2^{16}$) в процессе поиска придется перебирать в среднем от 4 до 16 подстрок. С учетом того, что идеальных хэш-функций не бывает и коэффициент загруженности хэш-таблицы $\alpha \approx 0.5 \div 2$, в среднем потребуется перебирать несколько больше подстрок — от 16 до 64.

6.3. ОТСЕЧЕНИЕ НЕУДЛИНЯЮЩИХ ПОДСТРОК

Так как подстроки в хэш-списке располагаются справа налево, то при сравнении текущей подстроки хэш-списка с образцом ее необходимо запоминать как возможного кандидата на искомую подстроку только в том случае, если длина общего начала подстроки и образца строго больше уже найденной длины ℓ. Кроме того, в начале перебора обычно известно, что длина общего начала должна быть больше $\ell_0 \geq 2$, где значение ℓ_0 определяется кодировщиком (см. главу 5), и алгоритм поиска подстрок может не рассматривать подстроки, не удовлетворяющие этому условию.

Итак, приняв в начале перебора $\ell = \ell_0$, текущая подстрока хэш-списка становится кандидатом на искомую подстроку тогда и только тогда, когда длина общего начала подстроки и образца *строго больше* значения переменной ℓ. Это, в свою очередь, равносильно тому, что k, ..., $(k + \ell)$-й символы совпадают с символами в позициях i, ..., $(i + \ell)$, где k — начало текущей подстроки хэш-списка, а i — начало образца.

Для того, чтобы повысить эффективность перебора, можно выбрать несколько символов c_j подстроки $S[i : \ell + 1]$, расположенных в позициях $(i + n_j)$ $(n_j \leq \ell)$, и перед тем, как производить сравнение k-й и i-й подстрок, сравнить между собой символы c_j

и символы, расположенные в позициях $(k + n_j)$, и тогда при удачном выборе n_j сравнение подстрок будет очень редкой операцией. В идеальном случае сравнение должно производиться только тогда, когда первые $(\ell + 1)$ символов подстрок совпадают.

Так как хэш-суммы k-й и i-й подстрок равны, то с высокой вероятностью у них совпадают и первые n символов, где $n < \ell$ — количество символов, по которым строится хэш-сумма. Следовательно, проверка первых n символов очень часто бесполезна. Применение деревьев для поиска подстрок неэффективно именно потому, что сравнение первых n символов обязательно для выяснения отношения подстрок, но мало информативно.

Например, слово *символ* в этой работе встречается очень часто, и хэш-список, соответствующий символам *сим*, содержит все подстроки, начинающиеся со слова *символ*. Допустим, что образец начинается со слова *символов*. При переборе подстрок практически сразу же будет найдена подстрока длины $\ell = 6$, начинающаяся со слова *символ*. Очевидно, проверка любого из первых 6 символов также будет практически бесполезной; гораздо лучше проверять на то, что $(k + 6)$-й символ совпадает с буквой *о*.

Интуитивно понятно, что чем дальше символ находится от первых n, тем он меньше от них зависит. Более точно, определим условную вероятность

$$P_a^m(a_{j_0} \dots a_{j_{n-1}}) \tag{6.10}$$

того, что в подстроке $a_{j_0} \dots a_{j_m}$ символ a_{j_m} есть символ a при условии, что первые n символов совпадают с $a_{i_0} \dots a_{i_{n-1}}$. Как правило, P_a^m уменьшается с ростом m (хотя, конечно, нетрудно привести пример, когда это не так).

Если перед тем, как сравнивать символы k и i, $(k + 1)$ и $(i + 1)$ и т. д., сравнить последние $(k + \ell)$ и $(i + \ell)$ и предпоследние $(k + \ell - 1)$ и $(i + \ell - 1)$ символы, то для большинства подстрок проверка остальных символов окажется излишней, т. к. сравнение будет неудачным.

В таком случае перебор подстрок хэш-списка выглядит как

```
#define M               (W-1)
#define INIT(i)         ...
#define VALID(k)        ...
...
int match_get (int i, int l)
{
  int k, t;
  INIT(i);
  k = i;
  for (;;)
  {
    register char *b, c0, c1;
    b = match_buff + l;
    c0 = b[0];
    c1 = b[2];
    do
    {
      while (VALID (k=next[k]) && c0 != b[k]);
      if (!VALID (k)) return (l);
    }
```

```
      while (c1 != b[k-1]);
      if ((t = common_length (k, i)) > l)
        l = t, match_pos = k;
    }
  }
```

Предполагается, что макрос `INIT(i)` инициирует перебор с образцом i, а макрос `VALID(k)` определяет, является ли k-я подстрока допустимой (в частности, `VALID(k)` определяет конец списка). Считается, что $next[i]$ содержит индекс первой подстроки с хэш-суммой, равной хэш-сумме образца.

Как показывают экспериментальные данные, сравнение вначале последних символов позволяет отсечь 93–96% подстрок[1], а последующее сравнение еще и предпоследних символов — 96–98%. Такое отсечение неудлиняющих подстрок очень эффективно, т. к. при совпадении этих пар символов k-я подстрока оказывается более длинным началом образца примерно в 75–80%, т. е. вероятность ошибки менее 25%.

Таким образом, для проверки 94–96% подстрок достаточно исполнения 6–7 машинных команд (следует заметить, что переменные `k`, `c0` и `b` находятся на аппаратных регистрах).

Другой подход заключается в выборе среди первых $(\ell + 1)$ символов образца пары символов с самыми низкими вероятностями. Если n_1-й и n_2-й символы образца $(n_1, n_2 \leq \ell)$ имеют самые низкие вероятности появления p_1 и p_2, то количество подстрок в буфере, у которых n_1-й и n_2-й символы k-й подстроки совпадают с соответствующими символами образца, заведомо не превышает Wp_1p_2. Понятно, что по вышеприведенным соображениям желательно, чтобы или n_1, или n_2 было равно ℓ.

В большинстве случаев эффективность последнего метода немного выше, чем первого. При первом подходе вероятность ошибки (проверки подстроки на удлинение, которого не происходит) — около 20–25%; маловероятно, чтобы это значение можно было заметно улучшить, сравнивая только два символа подстрок. Кроме того, около 80–90% времени поиска приходится на самый внутренний цикл перебора

```
while (VALID (k=next[k]) && c0 != b[n]);
```

поэтому даже снижение ошибки до 0 ускорит перебор только на 2–4%.

6.4. УСКОРЕНИЕ ПОИСКА

Воспользуемся тем соображением, что совершенно необязательно находить точное значение $len(i)$. Если в подавляющем большинстве случаев найденное значение $len(i)$ совпадает с точным значением и при этом удастся сильно уменьшить перебор (за счет незначительной потери качества сжатия), то такой подход вполне приемлем.

6.4.1. ПРИНУДИТЕЛЬНОЕ ОГРАНИЧЕНИЕ ДЛИНЫ ХЭШ-СПИСКА

Средняя длина хэш-списка равна $\alpha = \frac{W}{H}$, где W — ширина окна, а H — размер хэш-таблицы. Очевидно, что небольшое количество очень длинных хэш-списков вносит наибольший вклад в общее время поиска.

[1] Качество отсечения также существенно зависит от коэффициента заполненности хэш-таблицы, качества выбранной хэш-функции, ширины окна, и собственно сжимаемых данных.

Однако перебор можно принудительно прекратить после просмотра первых K подстрок хэш-списка. Если хэш-список содержит очень много подстрок, то, скорее всего, среди них много похожих, и поэтому достаточно длинная подстрока все-таки будет найдена (хотя и необязательно самая длинная).

Как показывает опыт, при $K \approx 32 \div 64$ ухудшение качества сжатия не превышает 1–3%, а перебор сокращается в полтора-два раза.

6.4.2. МОДИФИКАЦИЯ ХЭШ-ФУНКЦИИ

Гораздо лучший способ ускорения перебора основан на том, что хэш-функцию необязательно вычислять только по трем символам подстроки и в некоторых случаях можно использовать большее количество символов. Это позволит выровнять длины хэш-списков и уменьшить среднеквадратичное уклонение от среднего значения.

Заметим, что самые длинные хэш-списки соответствуют подстрокам, у которых среди первых n символов имеется пробел или ноль, т. к. вероятности остальных символов не превышают 5–8% и длины хэш-списков при ширине окна $W = 2^{15} = 32768$ равны 4–16.

Будем вычислять хэш-функцию по первым $(3+s)$ символам подстроки, где s — количество пробелов среди первых трех символов; по аналогичным соображениям качество сжатия не должно заметно ухудшиться. Поскольку большинство строк, принадлежащих длинным хэш-спискам, содержит несколько пробелов, при использовании такой хэш-функции перебор сокращается в 1.4–1.7 раз, а качество сжатия (при ограничении длины хэш-списка) улучшается на 1–2%. Даже по сравнению с точным решением (перебор без ограничения длины хэш-списка, хэш-сумма вычисляется по первым трем символам) качество сжатия падает всего на 0.2–0.3%.

Модифицированная описанным образом хэш-сумма используется автором в библиотеке сжатия данных.

6.4.3. ВЫБОР ХЭШ-ФУНКЦИИ

Очевидно, выбор хэш-функции существенно влияет на время поиска: чем равномернее хэш-функция рассеивает ключи, тем короче длины хэш-списков и быстрее поиск, поэтому выбор более сложных, но более качественных хэш-функций часто бывает оправданным. С другой стороны, вычисление значения хэш-функции должно выполняться достаточно быстро, чтобы не оказать заметного влияния на время поиска. Таким образом, на практике выбор хэш-функции всегда приводит к некоторому компромиссу между эффективностью и вычислительной сложностью.

Известные методы вычисления хэш-суммы[2] суммированием [18] по формуле $h_0 = 0$, $h_i = (h_{i-1} + AS[i]) \bmod B$, где A и B — некоторые константы, по мнению автора, не выдерживают никакой критики.

Полиномиальное хэширование [18, 20], дающее остаток от деления полинома над полем Галуа $GF(2^n)$ на примитивный в этом поле полином, являющийся порождающим элементом поля, дает очень хорошие результаты (широко известные алгоритмы вычисления циклических контрольных сумм CRC фактически являются полиномиаль-

[2] Здесь и далее подразумевается, но не указывается в явном виде, достаточно очевидное действие — взятие значения хэш-функции по модулю размера хэш-таблицы в тех случаях, когда значение хэш-функции может выходить за допустимые границы индекса хэш-таблицы.

ными хэш-функциями), однако ввиду достаточно высокой вычислительной сложности их использование не всегда оправдано.

Неплохие результаты можно получить, применяя для вычисления значения хэш-суммы контрольную сумму Флетчера [88], вычисляемую по формулам $s_0 = s'_0 = 0$,

$$s_i = (s_{i-1} + S[i]) \bmod 255, \tag{6.11}$$

$$s'_i = (s'_{i-1} + s_i) \bmod 255, \tag{6.12}$$

$$h_i = 256 s_i + s'_i. \tag{6.13}$$

Однако, по мнению автора, наилучшим выбором является хэш-сумма Зобриста [203] (мало известная даже специалистам), устроенная следующим образом. Заполним двумерную таблицу $T[L][|\Sigma|]$ случайными числами, тогда значение хэш-функции строки символов алфавита Σ длины L вычисляется как

$$h = \sum_{i=1}^{L} T[i][S[i]]; \tag{6.14}$$

на практике вместо суммирования лучше использовать операцию симметричной побитовой разности XOR.

Замечание 6.1. Если возможная длина строки S может быть большой, то с целью ограничения объема памяти, занимаемого таблицей T, можно зафиксировать некоторую константу M и вычислять значение хэш-функции по формуле

$$h = \sum_{i=1}^{L} T[i \bmod M][S[i]], \tag{6.15}$$

что позволяет за счет некоторого снижения качества хэширования на очень длинных строках ограничить объем требуемой памяти до $M|\Sigma|$ слов. ■

Нетрудно заметить, что значение хэш-функции Зобриста удовлетворяет почти всем требованиям на хорошую хэш-функцию: изменение одного бита хэшируемой строки приводит к кардинальному изменению значения хэш-функции, так что даже мало отличающиеся друг от друга строки, строки, полученные перестановкой символов, и другие сильно коррелирующие последовательности строк порождают практически независимые и некоррелирующие значения хэш-функции.

Понятно, что качество хэширования по Зобристу также существенно зависит от того, насколько случайны значения, находящиеся в таблице: желательно, чтобы они удовлетворяли известным критериям случайности [17], причем по каждому биту или группе битов. Например, значения, порождаемые линейным конгруэнтным генератором [17] — аналогом хэширования суммированием, не удовлетворяют этому требованию. Генератор случайных чисел, построенный на сдвиговых регистрах (аналог полиномиального хэширования) — с задержкой или без — также не свободен от недостатков, хорошо известных в криптографии.

Вообще говоря, связь между методами вычисления хэш-функции, порождением псевдослучайных последовательностей и шифрованием сообщения очевидна и хорошо известна, однако метод, хорошо подходящий для решения задач одного класса, часто является не самым лучшим (хотя, как правило, и неплохим) для задач другого класса, что объясняется различием критериев оценки. В частности, для получения хэш-суммы можно использовать криптографические методы цифровой подписи, которые фактически и есть значение хэш-функции, причем гарантированно хорошей, или использовать в качестве значения хэш-функции зашифрованную хэшируемую строку, однако вычислительная сложность таких методов неприемлемо высока.

Возвращаясь к методу хэширования Зобриста, в свете вышесказанного предлагается использовать криптографические методы для генерации псевдослучайных последовательностей высокого качества. В частности, если таблицу T рассмотреть как последовательность байтов, а не слов, и сначала заполнить каким-либо произвольным образом (например, нулями), после чего зашифровать полученную последовательность байтов одним из простых в реализации, но очень надежных методов шифрования (RC4[3], RC5 [154], TEA[186, 187]), то полученная последовательность будет практически случайной ввиду высокой криптографический стойкости метода шифрования: если бы какие-либо закономерности можно было установить, то задача дешифрования не представляла бы трудности, в то время как достаточно достоверно известно, что это не так.

Эмпирические исследования автора показали, что использование хэш-функции Зобриста в описанном виде позволяет уменьшить количество строк, подлежащих перебору, в среднем на 20–30% по сравнению с использованием суммирующих хэш-сумм и тем самым уменьшить время кодирования на 5–15%.

6.5. БУФЕРИЗАЦИЯ

Для эффективного поиска в памяти необходимо хранить $B = W + N$ символов, где W — ширина окна, а N — максимальная длина подстроки:

$$0 \ldots i - W \boxed{i - W + 1 \ldots i - 1 \mid i \ldots i + N - 1} i + N \ldots L \tag{6.16}$$

Для того чтобы при увеличении i не модифицировать все структуры данных, обычно используются *кольцевые* буфера, т. е. $index(k) = k \bmod B$, где $index(k)$ — номер k-го символа входной строки относительно начала буфера, и для адресации в буфере все структуры данных используют $index(k)$ вместо k.

Для того, чтобы во время поиска убедиться в том, что $i - k < W$ (k-я подстрока находится в окне ширины W), необходимо вычислить значение $i-k$. При использовании кольцевых буферов может получиться так, что $index(k) > index(i)$ при $k < i$. Чтобы получить смещение $i - k$ правильно, необходимо использовать формулу

$$i - k = \big(index(i) - index(k)\big) \bmod B. \tag{6.17}$$

Если ширина окна W есть степень двойки ($W = 2^k$) и $N = W$, то $B = W + N = 2^{m+1}$ — тоже степень двойки, поэтому операцию взятия модуля можно заменить маскированием младших $(m + 1)$ битов по маске $2^{m+1} - 1 = B - 1 = 2W - 1$. Заметим, на $(m + 1)$-битных машинах нет необходимости в маскировании, т. к. оно выполнится автоматически, и условие $i - k < W$ равносильно условию $index(i) - index(k) < W$. Ввиду того, что в арифметике по модулю 2^{m+1} с дополнением до единицы $W = 2^m - 1$ есть последнее неотрицательное число,

$$i - k < W \iff index(i) - index(k) < W \iff index(i) - index(k) \geq 0. \tag{6.18}$$

Это наблюдение позволяет проверять условие $(i - n) \bmod B < W$ при $W = 2^{15} = 32768$ и использовании 16-битовой арифметики за одну машинную команду без излишних вычислений и применений регистров.

[3] Алгоритм потокового шифрования Рональда Ривеста RC4 ©RSA Laboratories Inc., USA, до 1995 г. был коммерческим секретом фирмы RSA, пока не был разглашен в sci.crypt анонимным источником. В настоящее время используется в протоколе SSL (Netscape) и для шифрования паролей в Windows NT.

6.6. ПОИСК ПОДСТРОК ПЕРЕБОРОМ ХЭШ-СПИСКОВ

Как было показано в главе 5, количество поисков подстрок, которые действительно необходимы для кодирования, примерно равно длине сжатого представления входной строки (как при использовании оптимального сжатия, так и различных эвристик). Следовательно, при сжатии в 3 раза поиск подстрок потребуется только в 1/3 всех случаев, а в 2/3 потребуется только добавлять и удалять из структуры данных соответствующие подстроки.

Если добавление и удаление элементов будут выполняться очень быстро, а поиск — примерно с такой же скоростью, как и при использовании деревьев, то процесс сжатия ускорится примерно в 3 раза. Линейные списки — двусвязные и односвязные — удовлетворяют первому требованию.

Что касается времени поиска, то в предыдущих пунктах было показано, каким образом последовательный перебор 16–64 подстрок хэш-списка можно сделать таким же быстрым, как поиск с использованием дерева поиска. Кроме того, при применении списков знание минимальной длины искомой подстроки может быть существенно использовано для быстрого отсечения неудлиняющих подстрок.

Так как интерес представляет только самая правая и самая длинная подстрока (см. главу 5), являющаяся началом образца, то строки в списке должны располагаться справа налево. Поиск должен быть прекращен, если k-я подстрока не попала в окно ширины W ($i - k \geq W$) или если список закончился (в этом случае можно считать, что $k = 0$).

6.6.1. ИСПОЛЬЗОВАНИЕ ДВУСВЯЗНЫХ СПИСКОВ

Использование двусвязных списков не представляет никаких проблем. Будем использовать три массива:

- $hash[h] = k \neq 0$ тогда и только тогда, когда k-я подстрока является последней подстрокой со значением хэш-суммы по первым n символам, равным h;
- $next[i \bmod B] = k$ — индекс следующей подстроки такой, что i-я и k-я подстроки имеют одинаковую хэш-сумму ($k = 0$ означает, что список закончился);
- $prev[i \bmod B] = k$ — номер k-го элемента в массиве $next$ ($k < W$) такого, что $next[k] = i$, или $(k - W)$-го в массиве $hash$ при $k \geq W$ такого, что $hash[k - W] = i$.

При использовании таких структур данных их модификация (удаление $(i - W)$-й и добавление i-й подстрок) описывается следующей программой на языке Си:

```
#define M     (W-1)
...
void update (int i)
{
  int h = hash_sum (i);
  if (prev[i&M] < W) next[ prev[i&M] ] = 0;
  else               hash[ prev[i&M] ] = 0;
  hash[h] = prev[ (next[i&M] = hash[h])&M ] = i;
}
```

Заметим, что можно разместить массивы $next$, $hash$ и $prev$ последовательно, один за другим. Это позволит избавиться от проверки на то, что `prev[i&M] < W` и, кроме того, использовать один и тот же регистр для адресации всех трех массивов:

```
#define M     (W-1)
#define hash  (next+W)
#define prev  (hash+H)
...
void update (int i)
{
  int h = hash_sum (i);
  next[ prev[i&M] ] = 0;
  hash[h] = prev[(next[i&M] = hash[h])&M] = i;
}
```

Соответствующие макросы для поиска подстрок есть

```
#define INIT(i)                         /* empty */
#define VALID(k)      ((k) != 0)        /* check for NIL */
```

При использовании двусвязных списков главный цикл можно ускорить, если перед началом перебора присвоить $next[0]$ значение i и удалить проверку `VALID(k)` из главного цикла. Получаемая экономия в две команды дает ускорение примерно в полтора раза, т. к. цикл сокращается до 4—6 команд и, самое главное, из него удаляется одна команда условного перехода, которая нарушает работу конвейера в современных процессорах.

```
#define INIT(i)       next[0] = i
#define VALID(k)      ((k) != 0)        /* check for NIL */
...
do
{
  while (c0 != b[ k=next[k&M] ]);
  if (!VALID (k)) return (1);
}
while (c1 != b[n-1]);
...
```

Использование двусвязных списков приводит к очень изящным и простым алгоритмам модификации структур данных и перебора подстрок. Однако размер массива $next$ должен быть не меньше чем W, $W+H$ или $2W+H$ слов, а машинное слово должно хранить, как минимум, число $H+W-1$. Следовательно, при $W \geq 2^{15} = 32768$ использование двусвязных списков на 16-битовых машинах затруднительно или невозможно.

Поиск подстрок с применением двусвязных списков используется в разработанной автором библиотеке сжатия данных.

6.6.2. ИСПОЛЬЗОВАНИЕ ОДНОСВЯЗНЫХ СПИСКОВ

Так как 16-битовые процессоры и архитектуры широко используются в настоящее время (например IBM PC под MS DOS, встроенные системы и т. д.), то желательно иметь структуры данных, аналогичные по свойствам двусвязным спискам, требующие меньших затрат памяти и позволяющие ограничиться 16-битовой арифметикой.

Кажется, что без массива $prev$ не обойтись, т. к. в противном случае при вставке i-го элемента (а он попадет на место $i \bmod W = i - W \bmod W$ в массиве $next$) может произойти (и, скорее всего, произойдет) срастание списков. Более того, может получится зацикленный список, что совсем неприятно.

Тем не менее выход существует. Допустим, мы вставили i-ю подстроку, а j-я подстрока указывает на подстроку $(i-W)$ как на следующую в хэш-списке: $next[j \bmod W] = i - W$. Затем вставили k-ю подстроку, $k \geq i$, и j-я подстрока находится в одном хэш-списке с k-й подстрокой. При итерировании хэш-списка для k-й подстроки после прохождения j-й подстроки следующая подстрока должна быть такой, индекс которой записан в массиве $next[j \bmod W] = (i - W)$. Если мы обработаем подстроку с началом в $(i - W)$ и попробуем продолжить перебор хэш-списка, то окажется, что мы уже находимся в хэш-списке с головой i, т. к. $next[i \bmod W] = next[(i - W) \bmod W]$, и при $k = i$ произойдет зацикливание.

6.6.2.1. ЗНАНИЕ ДЛИНЫ ХЭШ-СПИСКА

Один из способов корректно проитерировать хэш-список — зная его длину, перебрать ровно столько подстрок, сколько необходимо:

```
#define M             (W-1)
...
j = depth[hash_sum(i)];
...
do
{
  while (--j > 0 && c0 != b[ k=next[k] ]);
  if (j <= 0) return (1);
}
while (c1 != b[k-1]);
...
```

За счет разворачивания цикла можно вынести из него проверку `(--j != 0)`. Тем не менее скорость поиска уменьшится, т. к. при удалении подстроки $(i - W)$ будет необходимо или пересчитывать для нее хэш-сумму, или поддерживать массив $prev$ (аналогично тому, как это было сделано для двусвязных списков) для того, чтобы счетчики длин хэш-списков соответствовали действительности. Очевидно, последнее решение и по скорости, и по памяти заведомо уступает схеме с использованием двусвязных списков.

6.6.2.2. ОПРЕДЕЛЕНИЕ МЕСТА СРАСТАНИЯ СПИСКОВ

Другое решение основано на том, что при достижении места срастания двух списков ($(i-W)$-й подстроки) k станет равным $(i-W)$ и условие $(i-k) < W$ будет нарушено.

Следовательно, можно перебирать список до тех пор, пока смещение от i-й до k-й подстроки строго меньше W или пока k не станет равным нулю (заметим, что при добавлении i-й подстроки необходимо проверять, что следующая подстрока хэш-списка находится на расстоянии не более W от i).

Определение макросов, необходимых для перебора, есть

```
#define INIT(i)       next[0] = i
#define VALID(k)      ((((k)-i)&(2W-1)) < W)
```

В начале цикла $next[0]$ устанавливается равным i и используется обратное условие $(n - i) \bmod B \geq W$ для того, чтобы избавиться от отдельной проверки на конец списка, — после обработки 0-й строки в буфере n станет равным i, и условие окажется нарушенным.

Вставка i-й подстроки выглядит как

```
#define M             (W-1)
#define VALID(k)      ((((k)-i)&(2W-1)) < W)
...
void update (int i)
{
  int h, j;
  if (!VALID(j = hash[h = hash_sum(i)])) j = 0;
  next[i&M] = j;
  hash[h] = i;
}
```

Такой подход по скорости почти в полтора раза уступает двусвязным спискам, т. к. при переборе подстрок хэш-списка проверку `VALID(k)` из главного цикла выносить нельзя, хотя модификация структур данных и немного быстрее. К числу преимуществ такого подхода следует отнести экономию W слов памяти (с учетом буфера на $2W$ символов для односвязных списков потребуется $(4W + 2H) \approx 5W$ байтов оперативной памяти, а для двусвязных — $(6W + 2H) \approx 7W$ байтов, что почти в полтора раза больше).

6.6.2.3. НОРМАЛИЗАЦИЯ ХЭШ-СПИСКОВ

При предыдущем подходе на каждой итерации необходима проверка условия $(k - i) \bmod 2W \geq W$, которая может быть организована очень эффективно при $W = 2^{15}$ и использовании 16-битовой арифметики. В остальных случаях вычисление $(k - i) \bmod 2W$ требует нескольких команд (2 на трехадресных ЭВМ и 3 — на двухадресных).

Допустим, что $W \leq i < 2W = B$ и индексы всех подстрок в хэш-таблице меньше i (0 служит признаком конца списка). В таком случае можно вычислить значение $f = i - W$ (взятие по модулю $2W$ излишне, т. к. $i - W \geq 0$) и закончить перебор, если индекс k $(0 \leq k < i)$ текущей подстроки в буфере меньше или равен f, ввиду того, что

$$(i - k) \bmod B < W \iff (i - k) < W \iff f = i - W < k. \tag{6.19}$$

Заметим, что при достижении конца списка k станет равным 0 и условие $n > f$ будет нарушено, т. к. по предположению $i \geq W$.

При достижении указателем i конца буфера $(i = B = 2W)$ необходимо переместить последние W символов буфера в начало и модифицировать содержимое массивов $next$ и $hash$ так, чтобы оно соответствовало новому содержимому буфера (если $next[j] < W$, то установить $next[j]$ в 0, иначе в $next[j] - W$, т. к. $next[j]$-я подстрока в буфере теперь находится на W символов левее).

Итак, определение макросов, необходимых для перебора, есть

```
#define INIT(i)       f = i-W
#define VALID(k)      ((k) > f)
```

Модификация структур данных выглядит совсем просто:

```
#define M             (W-1)
...
void update (int i)
{
  int h;
  next[i&M] = hash[h = hash_sum(i)];
```

```
    hash[h] = i;
  }
```

Этот метод требует столько же памяти, как и предыдущий (примерно в полтора раза меньше, чем при использовании двусвязных списков), а по скорости почти такой же (добавление новой подстроки немного проще, но требуется модификация содержимого массивов $next$ и $hash$ каждые W символов).

6.7. ПОИСК ОЧЕНЬ КОРОТКИХ ПОДСТРОК

Иногда желательно осуществлять поиск подстрок длины 2. Например, такой поиск необходим при использовании упрощенного кодирования выхода алгоритма сжатия для того, чтобы максимально упростить декодировщик. Как правило, очень простые декодировщики используются для запуска саморазворачивающихся (self-extracting) задач, т. к. декодировщик является частью сжатых данных.

Если строить хэш-функцию по первым двум, а не трем символам, то это замедлит поиск подстрок примерно в $\frac{1}{\theta} \approx 13 \div 20$ раз, т. е. на порядок.

Гораздо лучшее решение — использовать еще одну хэш-таблицу $hash2$ для хранения индексов последних подстрок с хэш-суммой, построенной по первым двум символам. В начале поиска необходимо сравнить с образцом подстроку, чей индекс находится в ячейке $hash2[h]$, где h — хэш-сумма первых двух символов подстроки, после чего необходимо присвоить $hash2[h]$ значение i и начать обычный перебор подстрок так, как это описано раньше.

Для ограничения расхода памяти можно уменьшить размер дополнительной хэш-таблицы $hash2$ с 2^{16} слов (необходимый размер для поиска точного решения) до $2^{13} \div 2^{14}$ слов. При этом вероятность ошибки (т. е. подстрока длины 2 существует, но не была найдена в процессе перебора) обычно не превышает 1—2%. Соответственно качество сжатия ухудшается всего на 0.1—0.2% по сравнению с точным решением.

6.8. LZ77 С ПРЫГАЮЩИМ ОКНОМ

В 1995–1996 гг. автором была разработана высокоэффективная модификация алгоритма LZ77 со скользящим окном — LZ77 с прыгающим окном. Вместо того, чтобы поддерживать фиксированную ширину окна W, перемещая левую границу окна синхронно с правой, LZ77 с прыгающим окном перемещает левую границу скачкообразно, по позициям, кратным некоторому параметру w; в дальнейшем будет подразумеваться, что w нацело делит W.

Сначала кодировщик загружает W символов кодируемых данных в буфер и вставляет строки в позициях $0, \ldots, (W-1)$ (слева направо) в хэш-таблицу $hash$, используя для разрешения коллизий метод цепочек, т. е. провязывая элементы в односвязный список с использованием массива $next$.

Перебор подстрок хэш-списка осуществляется точно так же, как и для двусвязных списков (см. п. 6.6.1): сначала $next[0]$ устанавливается равным n, где n — смещение до начала незакодированного участка буфера (т. е. текущего суффикса), после чего производится перебор строк хэш-списка с использованием алгоритма быстрого отсечения неудлиняющих подстрок (см. п. 6.3), и если два последних символа i-й и n-й строк совпадают, то при $i = n$ перебор закончен, иначе производится полное сравнение строк.

После того, как закодированы первые W символов буфера, кодировщик *одновременно* удаляет w первых строк буфера из хэш-таблицы, после чего w первых симво-

лов буфера заменяются следующими символами входного потока данных, и w новых подстрок добавляется к хэш-таблице. После того, как закодированы следующие w символов, производится аналогичная операция с символами, расположенными в позициях $w, \ldots, (2w-1)$, и т. д.; по достижении конца буфера ширины W заменяются первые w символов буфера. Таким образом, в начале кодирования ширина окна сначала увеличивается с 0 до W символов, затем скачкообразно уменьшается до $(W-w)$ (левая граница окна прыгает на w символов вперед), после чего в процессе кодирования ширина окна плавно увеличивается до W (при этом положение левой границы окна остается неизменным), затем левая граница снова прыгает на w символов, и т. д.

Для одновременного удаления из хэш-таблицы подстрок с индексами, находящимися в диапазоне $[(k-1)w, kw-1]$ необходимо просмотреть хэш-таблицу $hash$ размера H и массив $next$ размера W и если значение какого-либо элемента таблицы находится в указанном диапазоне, то необходимо заменить значение такого элемента на признак окончания хэш-списка (в нашем случае это нулевой индекс).

Так как команды условных переходов нарушают работу конвейера, который имеется у всех современных процессоров, и поэтому исполняются заметно медленнее арифметических команд, вместо проверки двух условий

```
x = a[i];
if (x >= A && x < B)
  a[i] = 0;
```

можно ограничиться одной проверкой, если заметить, что отрицательные целые значения в ЭВМ представляются точно так же, как и большие беззнаковые целые, поэтому следующий код функционально эквивалентен предыдущему при почти вдвое лучшей эффективности:

```
C = B - A;
...
if (((unsigned) (a[i] - A)) < C)
  a[i] = 0;
```

С учетом того, что ширина шины современных ЭВМ в несколько раз больше ширины слова (см. п. 2.10), то обращение к k последовательно расположенным словам памяти (где k — отношение ширины шины к ширине слова, обычно от 2-х до 8-ми) требует столько же времени, сколько и обращение к одному слову по произвольному адресу. Более того, ряд современных процессоров отслеживают обращение к последовательно расположенным данным и автоматически включают предвыборку данных, загружая в кэш процессора следующие данные, не дожидаясь обращения к ним; это производится параллельно исполнению других команд, поэтому для таких процессоров обращение к последовательно расположенным данным мало отличается от обращения к регистровым переменным и происходит в 5–15 раз быстрее, чем обычное обращение в память (см. п. 2.10).

Таким образом, несмотря на то, что каждые w символов необходимо просматривать $(W+H)$ слов памяти, при достаточно больших $w \geq \frac{W}{16}$ эффективность массового удаления подстрок не меньше, а часто существенно выше, чем эффективность удаления w подстрок по одной.

Что же касается качества кодирования, то можно воспользоваться тем, что распределение длин смешений близко к распределению Шварца [160] с параметром $\theta =$

$0.1 \ldots 0.3$, т. е. вероятность того, что найденная подстрока расположена на i символов левее пропорциональная $\frac{1}{i^{1+\theta}}$, поэтому при выполнении этого соотношения вероятность того, что смещение до самой длинной подстроки находится в интервале $[W-w, W-1]$, не превышает (с учетом того, что при $\theta = 0$ распределение Шварца вырождается в распределение Ципфа [199, 200], при котором вероятность того, что $i \in [2^{k-1}, 2^k - 1]$, не превышает $\frac{1}{\log_2 W}$, а при $\theta \geq 0$ эта вероятность не выше вероятности для распределения Ципфа)

$$\frac{w}{W \log_2 W}, \tag{6.20}$$

поэтому при $w \leq 0.5W$ эта вероятность не превышает

$$\frac{1}{2 \log_2 W} \tag{6.21}$$

и при больших W эта вероятность очень мала[4] (например, при $W \geq 2^{15} = 32$ Кб эта вероятность не превышает 3%).

Таким образом, при достаточно больших W и $\frac{W}{16} \leq w \leq \frac{W}{2}$ использование LZ77 с прыгающим окном позволяет ограничить объем требуемой памяти до $W + H \leq 2W$ слов (как при использовании односвязных списков) и добиться такой же или лучшей скорости сжатия, как и при использовании двусвязных списков, при очень незначительном (менее 1%) ухудшении качества сжатия по сравнению с качеством сжатия алгоритма LZ77 со скользящим окном при равной ширине окна.

6.9. ВЫВОДЫ

В работе описаны методы поиска для использования в системах сжатия, основанных на модификациях алгоритма LZ77 со скользящим окном.

Показано, что алгоритмы прямого перебора подстрок хэш-списка в подавляющем большинстве случаев наиболее эффективны как по времени поиска, так и по объему требуемой оперативной памяти.

Методы быстрого отсечения неудлиняющих подстрок были разработаны автором в 1991–1992 гг.; аналогичный менее эффективный метод — сравнение только последнего символа — использовал в 1992 г. один из авторов архиватора InfoZIP Jean-loup Gially. В 1994 г. Фенвик и Гутман [86] независимо от автора разработали схожий алгоритм быстрого отсечения подстрок с той разницей, что сравниваются не два последних символа, а последний и средний. Автором такой вариант также рассматривался, но был признан существенно менее эффективным, т. к. сравнение средних символов при небольшой текущей длине найденной подстроки, как правило, бесполезно.

Принудительное ограничение длины хэш-списка применяется практически во всех системах сжатия, использующих для поиска перебор подстрок хэш-списка. Скорее всего, впервые эта идея была предложена Уильямсом [188, 189, 190]: предложенные им высокоскоростные методы сжатия LZRW1, LZRW2, LZRW3 и LZRW4 (последние три не были опубликованы в открытой печати) используют перебор очень коротких (1–4 элемента) хэш-списков.

Следует отметить ряд принципиальных особенностей, позволивших добиться высокой скорости сжатия при таком же качестве сжатия (при равной ширине окна), как при

[4] Именно близостью распределения смещений к распределению Шварца при небольшом положительном θ можно объяснить тот факт, что, начиная примерно с $W = 2^{16} = 64$ Кб, дальнейшее увеличение ширины окна очень незначительно — лишь на несколько процентов — улучшает качество сжатия.

использовании других известных методов поиска подстрок:

1. Поиск наиболее длинной самой правой подстроки, являющейся началом образца, осуществляется *не обязательно точно*; поиск прекращается после перебора достаточного количества кандидатов, что позволяет гарантировать (при фиксированной максимальной глубине перебора) линейное поведение алгоритма поиска и в среднем, и в наихудшем случае (возможно, при незначительном ухудшении качества сжатия).
2. Активно используется *хэширование* по первым символам образца, очень высокая эффективность которого обеспечивается выбором хэш-суммы переменной длины (так что количество символов, участвующих в вычислении хэш-суммы, увеличивается, если начало образца содержит часто встречающиеся символы); кроме того, использование табличного метода вычисления хэш-суммы по методу Зобриста позволяет существенно повысить качество собственно хэш-функции, не увеличивая времени ее вычисления.
3. Сравнение текущей подстроки с образцом начинается с *конца*, что позволяет сравнением одного-двух символов отсечь подавляющее число неудлиняющих подстрок, не проводя полного сравнения строк.
4. Использование тщательно разработанных *структур данных и алгоритмов*, в ряде случаев учитывающих особенности современных архитектур ЭВМ, позволяет в 1.3–1.8 раз повысить эффективность перебора по сравнению с традиционными решениями без увеличения объема требуемой памяти.

Применение описанных методов поиска подстрок в совокупности с приведенными эвристиками позволяет в десятки и сотни раз уменьшить объем вычислений и ускорить поиск подстрок практически без ущерба для качества сжатия. В свою очередь заметно более высокая скорость поиска подстрок дает возможность улучшить качество сжатия за счет увеличения ширины окна. Немаловажно, что объем необходимой оперативной памяти в несколько раз меньше, чем при использовании традиционных структур данных.

Самая высокая скорость сжатия текстовой информации, согласно [40], составляет 33 Кб/с при ширине окна в 8 Кб на Sun Sparcstation 4/75. Сжатие по описанным в данной работе алгоритмам позволяет достичь скорости сжатия в 200–300 Кб/с при ширине окна в 64 Кб, и более 1 Мб/с при ширине окна в 8 Кб.

Использование предложенного автором алгоритма LZ77 с прыгающим окном позволяет почти вдвое уменьшить объем требуемой памяти и/или увеличить скорость сжатия по сравнению с алгоритмом LZ77 со скользящим окном, при незначительном — обычно менее 1% — ухудшении качества сжатия.

Таким образом, применение описанных алгоритмов позволяет создание высокоэффективных кодировщиков семейства LZ77: авторская реализация (см. главу 11) в 1.5–2 раза быстрее архиватора PKZIP, в 2–3 раза — архиватора RAR, и в 3–4 раза — архиватора ARJ при равном качестве сжатия и при таком же или меньшем объеме требуемой памяти. Необходимо отметить, что вышеперечисленные архиваторы, считающиеся самыми быстрыми известными реализациями LZ77 со скользящим окном, написаны на языке ассемблера для процессора Intel, тогда как авторская версия целиком написана на языке Си (и перенесена на почти все известные платформы) и поэтому на 10–20% менее эффективна ввиду известных особенностей архитектуры Intel.

Глава 7

КОДИРОВАНИЕ ВЫХОДА LZ77

Почти все известные варианты алгоритма LZ77 [201] — LZ78 [202], LZR [155], LZSS [35, 167], LZB [36], LZH [47], LZBW [43], LZY [198], LZWZ [197], LZY2 [91], LZW [184], LZT [172], LZJ [107] и др. — отличаются друг от друга только очень незначительными (на взгляд автора) деталями. Часто [85] такой деталью оказывается способ кодирования выхода алгоритма LZ77, т. е. получающейся последовательности литералов и указателей.

Напомним, что выход алгоритма LZ77 представляет собой последовательность *литералов*, т. е. символов некоторого алфавита Σ, и *указателей* на уже встречавшиеся подстроки, т. е. пар (ℓ, p), где ℓ — длина подстроки, а p — смещение до ее начала.

Для того, чтобы при декодировании различать литералы и указатели, они предваряются *различающим битом*, т. е. кодируемая последовательность фактически состоит из пар $(0, \sigma)$, где $\sigma \in \Sigma$, и троек $(1, \ell, p)$.

Обычно используется равномерное кодирование (см. п. 2.6.5.1) для символов Σ и бесконечные коды (см. п. 2.6.5.4) — для кодирования длин и смещений.

Автором предлагается использование энтропийного кодирования для всех элементов выходной последовательности. Впервые эта идея была высказана в 1987 г. Брентом [47], однако предложенный им подход был далек от совершенства и в дальнейшем подвергнут достаточно жесткой критике [38]. С тех пор разработчиками промышленных систем сжатия[1] были созданы достаточно эффективные методы кодирования выхода LZ77 (которые не были опубликованы); описываемый метод был разработан автором в 1990–1994 гг. и по качеству кодирования несколько превосходит аналогичные разработки и гарантированно лучше использования любых фиксированных кодов.

Следует подчеркнуть, что автор не считает себя первооткрывателем описанных методов кодирования; они были разработаны — часто независимо друг от друга — большой группой разработчиков архиваторов (включающей автора), так что установить приоритет не представляется возможным. Особо следует отметить Хирухико Окумару (Hiruhiko Okumara), создателя архиватора ar002, который сумел достаточно эффективно использовать энтропийное кодирование и тем самым в начале 90-х гг. дал толчок к интенсивным исследованиям в данном направлении.

Автор счел необходимым включить данную главу в свою работу ввиду того, что многие из описанных методов кодирования, насколько известно автору, никогда ранее не публиковались в открытой печати и представляют собой ноу-хау.

[1] Исходные тексты многих систем сжатия (LZSS, LZHUF, HPACK, InfoZIP, LHA, LHARC, ZOO, HA, и др.) свободно распространяются; в ряде случаев (PKZIP, протокол TCP/IP RFC–1851) имеется точная спецификация алгоритмов кодирования и декодирования.

7.1. КОДИРОВАНИЕ СМЕЩЕНИЙ

Заметим, что смещение p до начала подстроки длины ℓ в уже декодированной части сообщения не превышает ширины окна W при использовании алгоритма LZ77 со скользящим окном, или i — количества уже декодированных символов — при использовании алгоритма LZ77 с без ограничений на смещение ($W = \infty$).

Таким образом, можно полагать, что *конечный* диапазон возможных значений смещения p в каждый момент времени известен: $p \in [1, W_i]$, где $W_i = \min\{W, i\}$.

Напомним, что бесконечные коды для натурального числа n представляют собой пару $\Big(L(n), R(n)\Big)$, где $L(n) = \lfloor \log_2 n \rfloor$, а $R(n) = n - 2^{L(n)}$, причем для кодирования $L(n)$, в свою очередь, используется какой-либо бесконечный код, а $R(n)$ кодируется равномерным образом $L(n)$ битами (для разделения $L(n)$ и $R(n)$ используется различающий бит).

Так как известен диапазон, в котором находится смещение, то можно использовать *групповое кодирование* (см. п. 2.6.5.3): ввиду того, что $L(p) \leq L(W_i)$, можно использовать *энтропийное* кодирование для записи $L(p)$ с учетом частоты появления $L(p)$.

Заметим, что т. к. длина кодируемой последовательности всегда конечна и на практике не превосходит $i_{max} < 2^{64}$, то можно считать, что $L(W_i) < 64$ (более того, часто $i_{max} < 2^{32} = 4$ Гб, так что $L(W_i) < 32$), поэтому можно заранее создать алфавит $P = \{\alpha_0, \ldots, \alpha_P\}$ и использовать $L(p)$-й символ P для кодирования $L(n)$.

Эмпирические исследования автора показали, что распределение смещений подчиняется закону Шварца [160], т. е. вероятность появления смещения p пропорциональна $\frac{1}{p^{1+\theta}}$, где $\theta \in [0.1, 0.3]$ и зависит от кодируемых данных.

Как показано в п. 4.2, групповое кодирование является почти оптимальным при распределении Шварца и его избыточность не превышает 0.03%. На практике избыточность группового кодирования при использования статического кода Хаффмана несколько выше — порядка 0.2–1% и объясняется как отличиями фактического распределения от распределения Ципфа, так и наличием избыточности, присущей коду Хаффмана.

Такое кодирование указателей (строго говоря, за исключением очень небольшого количества случаев) заведомо лучше использования любых фиксированных кодов.

7.2. КОДИРОВАНИЕ ДЛИН

Можно заметить, что длина ℓ совпадающей подстроки редко бывает большой; в подавляющем большинстве случаев (более 90%) она не превышает 32.

С целью повышения эффективности кодирования длин подстрок можно выбрать некоторую константу ℓ_1 и при $\ell \leq \ell_1$ оптимальным образом кодировать символ $\beta_\ell \in \Phi$, а при $\ell > \ell_1$ использовать групповое кодирование, как это было сделано для смещений, т. е. кодировать с использованием энтропийного кодирования символ $\beta_{\ell_1+1+L(\ell')}$ некоторого алфавита L, после чего равномерным образом кодировать $L(\ell')$ битов $R(\ell')$, где $\ell' = \ell - \ell_1$.

Так как ввиду буферизации реализацией налагаются естественные ограничения на наибольшую длину ℓ_{max} найденной подстроки (обычно $\ell_{max} < W$ или $\ell_{max} < 2W$), то мощность алфавита L не превышает $(L(\ell_{max} - \ell_1) + \ell_1 + 2)$.

Замечание 7.1. Несколько лучшее качество сжатия можно получить при использовании *шагового* кода (см. п. 2.6.5.2), т. е. выбрав монотонно возрастающую последовательность $0 = \ell_0 < \ell_1 < \ldots$, при $\ell_j < \ell \leq \ell_{j+1}$ закодировать символ β_j с исполь-

зованием энтропийного кодирования, после чего равномерным образом закодировать $\lceil \log_2(\ell_{j+1} - \ell_j) \rceil$ битов значения $(\ell - \ell_j)$. ■

Нетрудно заметить, что качество сжатия такого кодирования также заведомо не хуже достижимого при использовании любого фиксированного кода.

7.3. КОДИРОВАНИЕ ЛИТЕРАЛОВ И РАЗЛИЧАЮЩИХ БИТОВ

Так как количество литералов и указателей может быть различным, то равномерное кодирование в точности одного бита может быть избыточным. При применении арифметического кодирования можно эффективно кодировать и двоичный алфавит, однако при использовании кода Хаффмана (или других префиксных кодов) это невозможно.

Эффективное решение заключается в использовании *блочного кода*, который организуется следующим образом.

Объединив алфавит литералов Σ и алфавит длин L, получим новый алфавит $\Sigma_L = \Sigma \oplus L$ мощности $(|\Sigma| + |L|)$, так что декодированием очередного символа $x \in \Gamma$ нетрудно установить, какому алфавиту он принадлежит: $x \in \Sigma$ или $x \in L$; в первом случае закодирован литерал, а во втором — указатель, и необходимо дополнительно декодировать еще и смещение.

Нетрудно установить, что энтропия блочного кодирования совпадает с суммарной энтропией кодирования с использованием независимых алфавитов, т. е. блочный код сам по себе не улучшает и не ухудшает качество кодирования. Однако, при использовании префиксных кодов применение блочных кодов позволяет уменьшить избыточность кодирования, вызванную тем, что длины всех кодовых слов — целые.

С той же целью можно использовать блочный код для одновременного кодирования переменной части и длин, и смещений. Образуем новый алфавит Δ мощности $|L| \cdot |P|$ прямым произведением алфавитов L и P:

$$\Delta = L \otimes P = \{(x, y) : x \in L, y \in P\}. \tag{7.1}$$

Так как мощности алфавитов L и P невелики, то Δ будет содержать не более нескольких тысяч символов, что вполне приемлемо. После этого следует использовать блочный код для алфавитов Σ и Δ (вместо Σ и L).

Приятной особенностью такого способа блочного кодирования является то, что при плохом качестве сжатия (при малом количестве найденных алгоритмом LZ подстрок, т. е. указателей) LZ77 выродится в простой алгоритм энтропийного кодирования.

7.4. ПОСТРОЕНИЕ ДВУХПРОХОДНЫХ СХЕМ КОДИРОВАНИЯ

Использование фиксированных кодов часто мотивируется авторами их создания высокой эффективностью однопроходных систем сжатия, т. к. использование адаптивных методов кодирования заметно ухудшает скорость сжатия, а применение статических префиксных кодов (например, кода Хаффмана) требует двух проходов по кодируемому сообщению, что увеличивает время кодирования почти вдвое.

Если с первым утверждением автор согласен (целиком и полностью), то второе утверждение не совсем верно.

Заметим, что при сжатии подавляющую часть времени занимает собственно алгоритм LZ77; время кодирования несущественно и не превышает нескольких процентов от

общего времени работы.

Вместо того, чтобы производить два прохода по кодируемому сообщению (первый для сбора статистики, а второй — для собственно кодирования), можно (и нужно) ограничиться одних проходом алгоритма LZ77 и построить *внутреннее представление* полученного LZ77-кода, используя кодирование с фиксированными кодами (например, равномерное), одновременно собирая статистику о частоте использования литералов и т. д. После этого следует закодировать внутреннее представление с использованием полученной статистики. Так как построение кода Хаффмана может быть выполнено очень быстро (см. п. 3.4), а перекодирование содержимого буфера сводится к последовательности сдвигов и маскирований, то время второго прохода также будет несущественным по сравнению с общим временем сжатия.

С целью ограничения объема требуемой памяти можно использовать для записи внутреннего представления буфер постоянного объема; при его заполнении кодировать его содержимое и начинать заполнять его с начала, собирая новую статистику согласно новому содержимому буфера.

Понятно, что при таком подходе появится дополнительная избыточность, вызванная необходимостью сообщения декодировщику информации о принятом способе кодирования (т. е. длин кодовых слов). Однако, как показано в п. 3.3, такая информация может быть закодирована достаточно эффективно, а при достаточно большом объеме буфера относительная избыточность будет незначительна, так что результирующее качество сжатия все равно будет выше достижимого при использовании фиксированных кодов.

Таким образом, при использовании для кодирования выхода LZ77 статических префиксных кодов можно одновременно добиться и высокой скорости, и высокого качества кодирования.

7.5. СРЕДНЯЯ СТОИМОСТЬ КОДИРОВАНИЯ

Необходимо отметить, что при сжатии текстовых данных, исполняемых программ, объектных файлов и их библиотек (т. е. наиболее часто встречающихся типов данных) и использовании предложенного кодирования отношение r средних стоимостей кодирования указателей P и литералов C мало изменяется и находится в интервале

$$r = \frac{P}{C} \in [2.25, 2.75], \tag{7.2}$$

несмотря на то, что ширина окна W может варьироваться в широких пределах от 8 Кб до 1 Мб.

Этот на первых взгляд парадоксальный факт очень просто объясняется: при увеличении ширины окна средняя стоимость кодирования указателей P увеличивается (что достаточно очевидно), однако одновременно с этим растет количество найденных совпадающих подстрок и уменьшается доля литералов в выходном потоке, поэтому средняя стоимость кодирования литералов C увеличивается и отношение P и C остается практически неизменным.

Таким образом, применение разработанных автором алгоритмов построения оптимальной последовательности кодирования для фиксированных r, описанных в главе 5, обычно наиболее эффективно при $r = 2$, 2.25 или 2.5.

7.6. ВЫВОДЫ

Применение описанных способов организации энтропийного кодирования для выхо-

да алгоритма LZ77 позволяет добиться высокого качества кодирования, очень близкого к энтропии выхода LZ77; обычно энтропия меньше стоимости кодирования менее чем на 1–2%, поскольку избыточность кода Хаффмана (см. п. 3.1) не превышает $\varepsilon \leq 0.253$ битов на символ, а средняя стоимость кодирования $\mathcal{E} \geq 7$ при ширине окна $W \geq 2^{13}$ [39, 166], поэтому относительная избыточность $\frac{\varepsilon}{\mathcal{E}} < 0.04$, заведомо не превышая 4%.

Эмпирические исследования автора показали (ср. главу 11 и [41, 39]), что в среднем использование таких методов кодирования позволяет заметно — на 10–20% — улучшить качество сжатия по сравнению с применением фиксированных кодов, предлагаемых рядом авторов (см. LZR [155], LZSS [35, 167], LZB [36], LZBW [43], LZWZ [197] и др.).

При использовании статических префиксных кодов скорость кодирования и декодирования не отличается от скорости при использовании фиксированных кодов; более того, при применении высокоэффективных методов декодирования префиксных кодов, описанных в п. 3.5, скорость декодирования префиксных кодов может быть лучше скорости разбора фиксированных кодов. При адаптивном кодировании качество сжатия немного лучше при заметном ухудшении скорости сжатия (обычно вдвое) и очень значительном — в 5–10 раз — декодирования.

Часть IV

Алгоритмы сжатия сортировкой блоков

Глава 8

АЛГОРИТМЫ ПОЛНОЙ СОРТИРОВКИ БЛОКОВ

Реализация преобразования Барроуза — Уилера (см. п. 2.9) — весьма сложная задача, т. к. необходимо отсортировать по строкам матрицу M размера $N \times N$, содержащую N циклических вращений на $0, 1, \ldots, (N-1)$ символов строки S длины N, состоящей из символов некоторого алфавита Σ (см. п. 2.9).

Замечание 8.1. На самом деле построение отсортированной матрицы M' в явном не нужно, поскольку для всех $k = 0, \ldots, (N-1)$ k-я строка M' есть циклическое вращение исходной строки S на некоторое количество символов $m'(k)$, поэтому матрица M' однозначно задается последовательностью $m'(k)$. ■

Таким образом, алгоритм сортировки должен по строке S длины N построить вектор m' такой, что $m'(j)$ есть номер j-го циклического вращения S в отсортированной последовательности, т. е. $m'(j)$-я строка отсортированной матрицы M' есть j-е циклическое вращение строки S.

Можно применять любой из огромного количества известных алгоритмов сортировки, однако, как будет показано далее, в ряде случаев (а именно при наличии большого числа повторений) время сортировки может превысить все разумные границы. Так как желательно, чтобы алгоритм сжатия был работоспособен на произвольных данных, необходимо добиваться высокой производительности в среднем и приемлемой — в наихудшем случае, и как раз решение обеих задач в совокупности — весьма сложная проблема.

В настоящее время все известные системы сжатия, использующие BTW, реализуют или СЦ-сортировку (не менее $O(N\sqrt{N})$ операций[1] в наихудшем случае), как описано Барроузом и Уилером [55, 185], или сортировку удвоением (не менее $O(N \log_2^2 N)$ операций в наихудшем случае [5]).

8.1. АЛГОРИТМЫ СОРТИРОВКИ СТРОК

Хорошо известно [18], что задача сортировки N ключей сравнениями требует не менее $\log_2 N! \approx N \log_2 N - N$ сравнений. Более того, известны алгоритмы, которые сортируют последовательность N ключей за $N \log_2 N$ сравнений в наихудшем случае (сортировка слиянием MergeSort [18]) или в среднем (быстрая обменная сортировка QuickSort [18, 44]).

В нашем случае длина ключа очень велика ($N \log_2 |\Sigma|$ битов), поэтому время сравнения двух ключей может быть значительным. Например, если $S = \alpha \ldots \alpha\beta =$

[1] Все оценки сложности алгоритмов по времени и памяти приводятся для RAM-машины, описанной в п. 2.9.

$\alpha^{N-1}\beta$, то время сравнения двух ключей в среднем пропорционально N и время сортировки составит $O(N^2 \log_2 N)$, что неприемлемо уже при небольших N. В таких случаях применяют алгоритмы *цифровой сортировки* [18, 30, 32, 133, 151], существенно использующие тот факт, что ключи представляются в виде последовательности битов.

8.1.1. СОРТИРОВКА РАССТАНОВКОЙ

Сортировка расстановкой [18] применяется, если ключ состоит из одного символа и $|\Sigma| \ll N$. Если воспользоваться односвязными списками, то за один просмотр списка сортируемых записей можно построить $|\Sigma|$ списков записей с равными ключами, после чего слить их в один список в порядке возрастания ключей. Если всегда применяется одинаковый способ добавления записи к списку (первой или последней, что требует $O(1)$ операций), то результирующий список записей будет отсортирован устойчивым образом за $O(N+|\Sigma|)$ операций с использованием $(N+|\Sigma|)$ слов дополнительной памяти при добавлении записей в начало списка или $(N + 2|\Sigma|)$ слов при добавлении к концу списка.

Если ключи состоят из k символов, то можно их рассматривать как символы алфавита $\Sigma' = \Sigma^k$. К сожалению, если $|\Sigma|^k \geq \log_2 N$, то применение описанного алгоритма сортировки расстановкой становится непрактичным, т. к. алгоритмы сортировки сравнениями (за $O(N \log_2 N)$ операций) становятся более эффективными. Однако, как отмечалось выше, при больших k (если $k \log_2 |\Sigma|$ значительно превосходит ширину машинного слова в битах) время сравнения ключей становится пропорциональным k, и в таком случае применяются другие виды цифровой сортировки.

8.1.2. МЦ-СОРТИРОВКА

МЦ-сортировка (LSD [Less Significant Digit] RadixSort) упорядочивает ключи, начиная с *младшей цифры*; по мнению автора, это блестящая и очень неожиданная идея. Допустим, что ключ состоит из строк длины k символов алфавита Σ. Отсортируем расстановкой (устойчивым образом) строки сначала по $(k-1)$-му, затем по $(k-2)$-му, ..., и, наконец, 0-му символу. Ввиду устойчивости каждой сортировки последовательность строк, отсортированная по символам $(j+1), \ldots, (k-1)$ и отсортированная по j-му символу, будет отсортирована по символам $j, \ldots, (k-1)$. Таким образом, вся последовательность строк будет отсортирована за $O\big(k(N + |\Sigma|)\big)$ операций с использованием $(N + |\Sigma|)$ слов дополнительной памяти.

Несмотря на то, что при небольших k МЦ-сортировка существенно быстрее других алгоритмов сортировки, она неприемлема для реализации преобразования Барроуза — Уилера, т. к. при $k = N$ время МЦ-сортировки составит $O(N^2 + N|\Sigma|)$.

8.1.3. СЦ-СОРТИРОВКА

СЦ-сортировка (MSD [Most Significant Digit] RadixSort). упорядочивает ключи, начиная со *старшей цифры*. Отсортировав записи сначала по старшей, наиболее значимой цифре (или символу) ключа, разобьем все записи на $|\Sigma|$ групп, так что две записи принадлежат одной группе тогда и только тогда, когда первые символы соответствующих им ключей равны. После этого записи, принадлежащие одной группе, необходимо отсортировать по второй цифре (символу) и аналогичным образом разбить на группы второго порядка, так что у записей, принадлежащих одной группе второго порядка, совпадают две первые цифры (символа) ключа. Затем аналогичным образом сортируются

группы второго, третьего и т. д. порядка до тех пор, пока группа содержит более одной записи. Очевидно, результирующая последовательность является отсортированной.

На основе этого алгоритма Уилером [55, 185] был разработан алгоритм блочной сортировки, в котором Уилер существенно использовал то, что сортируемая последовательность строк получена циклическим вращением на один символ исходной строки S. Допустим, что последовательность строк, начинающихся с символа α, уже отсортирована, тогда строки, имеющие символ α вторым символом, сортировать не нужно: строка $S_i = c_i\alpha s'_i$ меньше $S_j = c_j\alpha s'_j$ тогда и только тогда, когда $c_j < c_j$ или $c_i = c_j$ и $s'_i < s'_j$. Таким образом, просматривая последовательность отсортированных строк, начинающихся с символа α, можно сразу же правильно расставить строки вида $c\alpha s$, причем c_i может быть равно α, т. е. строки, начинающиеся с $\alpha\alpha$, также не нужно сортировать. На практике это усовершенствование очень важно, т. к. позволяет избежать сортировки строк, начинающихся с повторов символа.

К сожалению, это очень ненамного улучшает временную оценку в наихудшем случае. Если S состоит из $O(\sqrt{N})$ одинаковых строк длины $O(\sqrt{N})$, разделенных случайными символами, то суммарное время сортировки составит не менее $O(N\sqrt{N})$, что неприемлемо много. Отметим, что для сортировки по одному символу следует применять алгоритм быстрой обменной сортировки QuickSort, как было сделано Уилером, а при достаточно большом количестве строк в группе — сортировку подсчетом и расстановкой (что почему-то не было сделано Уилером), т. к. это позволит быстро сортировать последовательности, содержащие большое количество одинаковых символов. При использовании других, *неадаптивных* алгоритмов сортировки (т. е. таких, для которых время сортировки последовательности ключей не зависит или почти не зависит от степени ее изначальной отсортированности) — например, сортировки слиянием MergeSort, — время сортировки в наихудшем случае может составить $O(N\sqrt{N}\log_2 N)$.

8.1.4. FORWARD RADIXSORT

В 1994–1996 гг. Андерссоном и Нильссоном [30, 32, 151] был предложен интересный алгоритм цифровой сортировки, названный Forward RadixSort, объединяющий преимущества МЦ– и СЦ–сортировок и лишенный их недостатков. Идея этого алгоритма заключается в следующем.

Отсортируем расстановкой[2] последовательность строк по первому символу, и образуем (в явном виде) список групп строк с равным первым символом ключа. Создадим список неотсортированных строк, в который попадут строки групп, содержащих более одной строки, и отсортируем этот список по второму символу, разбив его на подгруппы. Проходя по этому списку, вставим подгруппы (соответственно порядку просмотра) вместо групп, которым принадлежали неотсортированные строки; полученный таким образом список групп строк будет отсортирован уже по двум символам. После этого необходимо снова составить список неотсортированных строк, отсортировать его по третьему символу, заменить группы полученными подгруппами и т. д.

Чтобы избежать просмотра всех групп для отыскания неотсортированных строк, необходимо в процессе построения групп создать дополнительный список неотсортированных групп, содержащих две и более строк, так что список неотсортированных строк

[2] В действительности алгоритм Андерссона и Нильссона для сортировки n строк по k-му символу требует $O(n)$ операций, а не $O(n + |\Sigma|)$. По мнению автора, алгоритм чересчур сложен и требует чрезмерно много (около $15N$ слов) дополнительной памяти, поэтому описывается его упрощенный вариант, который тем не менее сохраняет все остальные свойства Forward RadixSort.

можно получить за один просмотр списка неотсортированных групп. Кроме того, можно совместить просмотр списка неотсортированных групп с расстановкой строк соответственно k-му символу, так что построение списка отсортированных групп не нужно.

Пусть B_k — общее количество строк, неотличимых по первым k символам (по определению, $B_0 = N$ и $B_{N+1} = 0$), тогда время работы описанного алгоритма равно[3]

$$O\left(\sum_{k=0}^{n} B_k + |\Sigma|\right), \tag{8.1}$$

где $n \leq N$ такое, что $B_{n+1} = 0$; т. к. B_k образуют невозрастающую последовательность,

$$N = B_0 \geq B_1 \geq \ldots \geq B_N \geq B_{N+1} = 0, \tag{8.2}$$

то такое n всегда существует.

Следует отметить, что алгоритм Forward RadixSort (во всяком случае, в варианте Андерссона — Нильссона) оптимален, т. к. он осматривает в точности $O\left(\sum B_k\right)$ символов; нетрудно заметить, что это *минимальное* количество символов, которые необходимо просмотреть в процессе сортировки.

К сожалению, этот алгоритм также мало пригоден для реализации преобразования Барроуза — Уилера, т. к. в наихудшем случае — для строки вида $S = \alpha^{N-1}\beta$ — время сортировки составит $O(N^2)$. Кроме того, объем требуемой памяти все равно остается достаточно большим — порядка $6N$–$8N$ слов.

8.2. АЛГОРИТМЫ СОРТИРОВКИ УДВОЕНИЕМ

8.2.1. АЛГОРИТМ МАНБЕРА—МАЙЕРСА

В 1990–1993 гг. Манбером и Майерсом [129, 130] был предложен интересный алгоритм построения так называемого *массива суффиксов* (suffix array) — отсортированной в лексикографическом порядке последовательности всех суффиксов (или хвостов) строки S длины N — за $O(N \log_2 N)$ операций с использованием $3N$ слов и $2N$ битов дополнительной памяти (за счет заметного усложнения алгоритма — без изменения асимптотической оценки его сложности — объем требуемой памяти может быть уменьшен до $2N$ слов).

Нетрудно заметить, что если строка S заканчивается символом, который появляется в S лишь единожды, то полученная применением алгоритма Манбера — Майерса последовательность суффиксов будет решением задачи блочной сортировки (возможно, для несколько другого лексикографического порядка символов).

Основная идея алгоритма Манбера — Майерса заключается в том, что если последовательность суффиксов отсортирована по первым n символам, то за $O(N)$ операций можно получить $2n$-отсортированную последовательность суффиксов.

Ввиду высокой сложности этого алгоритма он не описывается в данной работе; отметим лишь, что его средняя производительность [55] в 5–15 раз[4] ниже средней производительности других методов блочной сортировки, что ограничивает применение этого алгоритма на практике.

---

[3] Время работы оригинального алгоритма Андерссона и Нильссона равно $O\left(\sum B_k\right)$.

[4] Необходимо отметить, что Манбер и Майерс не ставили своей целью построение высокоэффективного алгоритма блочной сортировки; в своих работах [129, 130] они решали другие задачи.

К числу недостатков алгоритма Манбера — Майерса следует отнести также и то, что он *всегда* требует $\log_2 N$ проходов сложности $O(N)$ каждый. В 1993 г. Манбер и Майерс [130] предложили адаптивную версию алгоритма, для которого (при удачном стечении обстоятельств) построение массива суффиксов потребует менее $\log_2 N$ проходов, однако этот вариант еще сложнее исходного алгоритма и требует заметно больше памяти, так что средняя производительность все равно остается очень низкой.

8.2.2. АЛГОРИТМ ЗАМБАЛАЕВА

Гораздо более эффективный в среднем алгоритм сортировки удвоением (Sorting by Doubling), основанный на аналогичных идеях, был описан в 1997 г. Замбалаевым [5] и приводится ниже.

Отсортируем строки по первым F символам и разобьем на группы, так что две строки принадлежат одной группе тогда и только тогда, когда у них совпадают первые F символов. Пронумеруем группы в лексикографическом порядке, и пометим каждую строку номером группы, содержащей эту строку, так что $G(i)$ есть номер группы, содержащей i-ю строку s_i, т. е. строку, начинающуюся с i-го символа исходной строки S: $s_i = S_i S_{i+1} \dots S_{N-1} S_0 \dots S_{i-1}$.

Замечание 8.2. Начиная с нуля, номер следующей группы получается из номера предыдущей увеличением его на количество строк, содержащихся в предыдущей группе. Это нужно для того, чтобы при разбиении группы на подгруппы и перенумерации не возникала необходимость перенумерации всех остальных групп. ■

Таким образом, имеется два массива мощности N: массив I, где размещаются номера строк, отсортированных в неубывающем порядке по первым F символам, и массив G номеров групп, так что $G(i)$ есть номер группы строки s_i. Таким образом, при $i_1 < i_2$ выполняется $G(j_1) \le G(j_2)$, где $j_k = I(i_k)$, причем $G(j_1) = G(j_1)$ тогда и только тогда, когда первые F символов s_{j_1} и s_{j_2} совпадают, а $G(j_1) < G(j_1)$ тогда и только тогда, когда первые F символов s_{j_1} лексикографически меньше соответствующих символов s_{j_2}.

Допустим, что на k-м проходе все строки отсортированы по первым $n = F2^k$ символам (начиная с $k = 0$), и отсортируем их по первым $2n$ символам. Для этого воспользуемся тем, что если s_i и s_j неотличимы по первым n символам, $s_i =_n s_j$, то s_i и s_j принадлежат одной группе, $G(i) = G(j)$; если же $s_i <_n s_j$, то номер группы, содержащей s_j, строго меньше номера группы s_j: $G(i) < G(j)$.

Таким образом, чтобы отсортировать строки по первым $2n$ символам, достаточно отсортировать строки каждой группы, содержащей более одной строки, причем сравнивать строки по первым n символам не нужно, т. к. они заведомо совпадают.

Для того, чтобы сравнить две строки s_i и s_j одной группы по следующим n символам, достаточно сравнить номера групп, соответствующих строкам s'_i и s'_j, где

$$i' = (i + n) \bmod N, \tag{8.3}$$

$$j' = (j + n) \bmod N. \tag{8.4}$$

Таким образом, для каждой группы неединичной мощности необходимо отсортировать принадлежащие ей строки, используя в качестве ключа сортировки, соответствующего i-й строке, значение $G(i')$ номера группы, соответствующего i'-й строке, после чего группу необходимо разбить на подгруппы строк с одинаковым значением ключа сортировки, так что номер первой подгруппы равен номеру начальной группы, а номер каждой следующей подгруппы получается увеличением предыдущего значения на

количество строк, содержащихся в предыдущей подгруппе.

Замечание 8.3. При изменении номеров групп строк следует соблюдать определенную осторожность т. к. может случиться, что пометка i'-й строки уже изменена, и при обращении к значению $G(i')$ будет получено новое значение, в то время как пометки соседних строк — старые, что приведет к неправильному разбиению группы на подгруппы. Решение, предложенное Замбалаевым, достаточно сложное и запутанное; более эффективно перед началом сортировки строк группы заменить $G(i)$ на $G(i')+N$, сохранив при этом значение номера сортируемой группы. Если $G(i') \geq N$, то это означает, что следующие n символов i-й строки совпадают со следующими n символами, причем значение пометки i'-й строки уже заменено ключом сортировки, поэтому при $G(i') \geq N$ необходимо заменить $G(i)$ не на $G(i')+N$, а на $g+N$, где g — номер сортируемой группы. После этого строки группы сортируются использованием в качестве ключа i-й строки значения $G(i)$, а потом разбиваются на подгруппы так, как описано выше. Такое решение достаточно просто и, что самое главное, позволяет избежать многократных вычислений i', требующих взятия по модулю, в процессе сортировки. ■

Нетрудно заметить, что если группа состоит ровно из одной строки, то значение номера этой группы есть не что иное, как порядковый номер соответствующей строки в отсортированной матрице M'.

Итак, после k проходов все строки будут отсортированы по $2^k F$ первым символам, и сортировка закончена, если мощность всех групп равна единице или $F2^k \geq N$, для чего потребуется не более $\left\lceil \log_2 \frac{N}{F} \right\rceil$ проходов, сложность каждого из которых не превышает $N \log_2 N$, поэтому суммарное время сортировки не превышает $O(N \log_2^2 N)$, причем эта оценка достижима. Действительно, если S состоит из $O(\sqrt{N})$ одинаковых строк длины $O(\sqrt{N})$, разделенных случайными символами, то потребуется $O(\log_2 \sqrt{N})$ проходов по $O(\sqrt{N})$ группам со сложностью обработки каждой группы $O(\sqrt{N} \log_2 \sqrt{N})$, так что суммарное время работы составит $O(N \log_2^2 \sqrt{N}) = O(N \log_2^2 N)$.

Несмотря на то, что в наихудшем случае время работы алгоритма Замбалаева несколько больше алгоритма Манбера — Майерса [130] ($O(N \log_2^2 N)$ и $O(N \log_2 N)$ соответственно), ввиду своей простоты алгоритм Замбалаева даже при $N \sim 10^6$ всегда в 2–4 раза быстрее алгоритма Манбера — Майерса, а в среднем алгоритм Замбалаева 1.5–2.5 раза медленнее алгоритма BRED3, разработанного Уилером [185], что отчасти объясняется необходимостью просмотра всех N строк для обнаружения подлежащих сортировке групп строк мощности два и более.

Объем памяти, требуемый алгоритмом Замбалаева, составляет $2N$ слов и N байтов, что несколько превышает объем памяти, требуемой алгоритмом BRED3 (N слов и N байтов) и несколько меньше объема памяти, требуемого другими алгоритмами.

8.3. ИСПОЛЬЗОВАНИЕ ДЕРЕВА СУФФИКСОВ

Для эффективного поиска образца в строке можно использовать структуру данных PATRICIA [18, 150] и ее варианты — бор (trie), дерево суффиксов (suffix tree) и т. д. [31, 33, 132, 136, 150, 151, 177]. В отличие от бора (который может быть построен для произвольного набора строк) дерево суффиксов (как и следует из его названия) строится для набора строк, являющихся суффиксами исходной строки S, что позволяет добиться временной оценки сложности построения дерева, равной $O(N)$ операций и в среднем, и в наихудшем случае [132] (в отличие от бора, который строится за $O(N^2)$ операций в наихудшем случае), при использовании $O(N)$ слов дополнительной памяти.

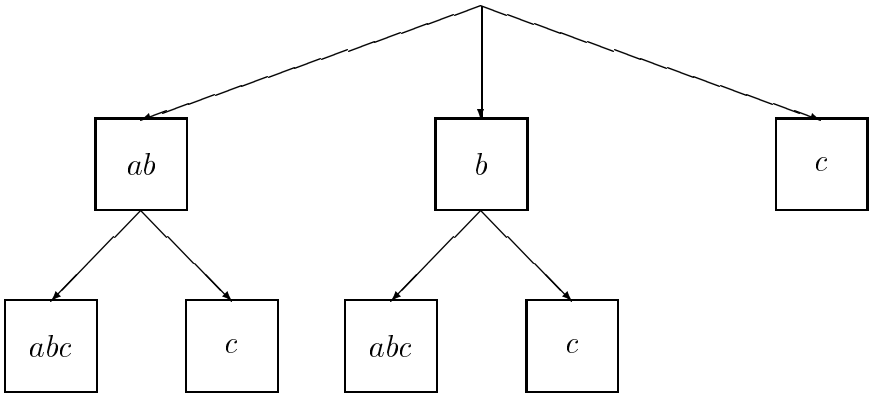

Рисунок 8.1. Дерево суффиксов для строки *ababc*

Пример дерева суффиксов для строки *ababc* приведен на рисунке.

Понятно, что задача блочной сортировки может быть решена обходом дерева суффиксов в лексикографическом порядке (при условии, что строка S заканчивается уникальным в S символом).

Дерево суффиксов применяется на практике исключительно редко по двум причинам:

- объем требуемой памяти составляет $5N$–$8N$ слов, что неприемлемо много при больших N (например, при $N \sim 10^6$ требуется не менее 24 Мб оперативной памяти);
- из-за высокой сложности алгоритма построения дерева суффиксов постоянный множитель во временной оценке $O(N)$ очень велик. Нетрудно заметить, что при $N < 2^{20} \approx 10^6$ алгоритм, требующий $N \log_2 N$ единиц времени, *быстрее* асимптотически более оптимального алгоритма, требующего $20N$ единиц времени. Таким образом, асимптотически менее эффективные функционально эквивалентные алгоритмы на практике иногда оказываются эффективнее асимптотически оптимальных алгоритмов.

Другая сложность заключается в том, что дерево суффиксов является *сильно ветвящимся*, т. е. каждая промежуточная вершина имеет от двух до $|\Sigma|$ сыновей, поэтому для эффективной работы требуется хэширование [62, 63, 72, 132], т. к. применение односвязных списков ухудшает время работы алгоритма до $O(N|\Sigma|)$ операций, а сбалансированных деревьев — до $O(N \log_2 |\Sigma|)$.

Кроме того, чтобы иметь возможность обхода дерева, необходимо иметь список всех сыновей для каждой промежуточной вершины, который недоступен при хэшировании, так что обход дерева требует поиска *всех* возможных сыновей вершины, поэтому время обхода дерева составляет $O(N|\Sigma|)$ единиц времени в наихудшем случае. Применение же сбалансированных деревьев неприемлемо, т. к. такой метод построения дерева суффиксов уступает другим методам блочной сортировки по всем параметрам.

Можно одновременно с использованием хэширования для представления дерева суффиксов хранить *неотсортированный* односвязный список всех сыновей промежуточной вершины. При обходе дерева необходимо сначала построить список всех сыновей всех промежуточных вершин, после чего сортировкой расстановкой (устойчивым образом) отсортировать его по первому символу подстроки, указываемой вершиной, а затем восстановить структуру дерева обходом уже отсортированного списка и обойти дерево в лексикографическом порядке; как можно заметить, все операции выполняются за $O(N)$ единиц времени.

Из-за необходимости хранить дополнительные ссылки на братскую и отцовскую вершины такое решение увеличивает объем требуемой памяти на $2N$ слов, поэтому при больших $N \sim 10^6$ объем требуемой памяти ($7N$–$10N$ слов) становится неприемлемым (не менее 30–40 Мб), а ввиду высокой сложности алгоритма построения и обхода дерева суффиксов средняя производительность такого алгоритма блочной сортировки низкая.

8.4. ЭФФЕКТИВНЫЙ АЛГОРИТМ БЛОЧНОЙ СОРТИРОВКИ

Разместим кодируемую строку S длины N в массиве $buff$. Кроме того, нам потребуется массив $node$ из N записей, так что i-я запись $node[i]$ соответствует подстроке, начинающейся в i-й позиции, что позволяет в дальнейшем не проводить различия между записью и соответствующей ей строкой (и наоборот). Каждая запись имеет два поля:

- $next$ — указатель на следующую запись в односвязном списке;
- $value$ — номер группы, которой принадлежит соответствующая записи строка.

Изначально все N записей провязаны в один односвязный список в последовательном порядке, т. е. $head = node + 0$, …, $node[i].next = node + i$, …, $node[N-1].next = NULL$.

Отсортируем этот список МЦ-сортировкой по первым F символам подстрок, соответствующих записям, после чего присвоим каждой записи номер группы, полученный следующим образом: начиная с нулевого номера группы и первой записи отсортированного списка, новая группа порождается последовательностью записей списка, соответствующие которым строки не отличимы по первым F символам. Каждая из записей одной группы получает текущий номер группы, после чего номер группы увеличивается на количество записей в последней группе.

Таким образом, если обозначить $=_F$, $<_F$, $>_F$ сравнение строк по первым F символам, то

$$i.value = j.value \iff s_i =_F s_j, \tag{8.5}$$

$$i.value < j.value \iff s_i <_F s_j, \tag{8.6}$$

$$i.value > j.value \iff s_i >_F s_j, \tag{8.7}$$

номера групп расположены в списке в неубывающем порядке и находятся в интервале от 0 до $(N-1)$ включительно.

Заметим, что если группа состоит ровно из одной строки, то присвоенный данной строке номер группы есть не что иное, как порядковый номер строки в отсортированной матрице M', поэтому такие строки не нуждаются в дальнейшей обработке и должны быть исключены из списка.

Допустим, что на k-м проходе список содержит строки, неотличимые по первым $n = F2^k$ символам, начиная с $k = 0$. Введем операцию

$$i' = (i + n) \bmod N \tag{8.8}$$

и воспользуемся тем, что для того, чтобы сравнить две строки s_i и s_j, принадлежащие одной группе, по первым $2n$ символам, достаточно сравнить номера групп, соответствующих строкам $s_{i'}$ и $s_{j'}$.

Вместо того, чтобы сортировать строки каждой группы в отдельности, как сделано в алгоритме Замбалаева, *отсортируем все строки одновременно.*

Временно добавим к записям поле $index$ (в дальнейшем будет показано, что на самом деле оно не нужно) и будем предполагать, что его значение у всех элементов списка равно нулю. Кроме того, заведем массив G из $(\lceil \frac{N}{2} \rceil + 1)$ слов и, начиная с $j = 1$,

для каждой группы строк запишем в $G[j]$ номер текущей группы, а для всех строк группы запишем в поле $index$ значение j, а в поле $i.value$ (для i-й строки) — значение номера группы, соответствующей $s_{i'}$, после чего увеличим j на 1 и перейдем к обработке следующей группы.

Замечание 8.4. Так как при обработке записи, соответствующей i-й строке, может получиться так, что значение поля $i'.value$ уже изменено и равно $i''.value$, то

$$i.value = \begin{cases} i'.value, & i'.index = 0, \\ G[i'.index], & i'.index > 0. \end{cases} \tag{8.9}$$

■

Так как ключ сортировки — значение поля $value$ — целое число в диапазоне $[0, N-1]$, то элементы списка можно отсортировать *устойчивым образом* МЦ-сортировкой сначала по L младшим значащим битам ключа, а затем по H старшим битам, используя $2^L + 2^H \approx 2\sqrt{N}$ слов дополнительной памяти, где

$$L = \left\lceil \frac{\lceil \log_2 N \rceil}{2} \right\rceil, \tag{8.10}$$

$$H = \left\lfloor \frac{\lceil \log_2 N \rceil}{2} \right\rfloor. \tag{8.11}$$

Замечание 8.5. Первый проход МЦ-сортировки и изменение содержимого поля $value$ строк списка можно и нужно совместить. ■

После этого необходимо сформировать новые группы, проходя по отсортированному списку следующим образом. Новая группа образуется последовательностью строк, у которых значения полей $value$ и $index$ совпадают, а номер новой группы есть не что иное, как $G[i.index]$, где i — произвольная запись, принадлежащая новой группе. После того, как перенумерование строк новой группы окончено, значение $G[i.index]$ необходимо увеличить на количество строк в новой группе для обеспечения корректности порождения следующей группы строк, которые ранее находились с текущими в одной группе.

Очевидно, если новая группа содержит только одну строку, то эту строку необходимо удалить из списка строк, не отличимых уже по первым $2n = F2^{k+1}$ символам.

Процесс разбиения групп и порождения новых продолжается до тех пор, пока список строк, подлежащих сортировке, не станет пустым и пока текущее значение n (глубины отсортированности) строго меньше N.

Замечание 8.6. Если $n > N$ и подлежащий сортировке список не пуст, то это означает, что кодируемая строка получена многократным повторением более короткой строки, длина которой N' является делителем N. ■

Замечание 8.7. Временная сложность k-го прохода определяется только длиной сортируемого списка, равной B_{F2^k}, и параметров H и L и не превышает

$$T_k(N) = O(B_{F2^k} + \sqrt{N}). \tag{8.12}$$

■

Так как глубина отсортированности увеличивается вдвое на каждой итерации, то максимальное количество итераций не может превысить

$$K = \left\lceil \log_2 \frac{N}{F} \right\rceil, \tag{8.13}$$

а каждая итерация требует не более двух проходов по списку длины $B_{F2^k} \le N$, поэтому суммарное время (с учетом времени МЦ-сортировки по первым F символам) близко к оптимальному

$$T_{opt}(N) = O\left((N + |\Sigma|)F + \sum_{k=0}^{K-1} B_{F2^k}\right) \tag{8.14}$$

и составляет

$$\begin{aligned} T(N) &= O\left((N + |\Sigma|)F + \sum_{k=0}^{K-1} T_k(N)\right) = & (8.15)\\ &= O\left((N + |\Sigma|)F + \sum_{k=0}^{K-1} (B_{F2^k} + \sqrt{N})\right) \le & (8.16)\\ &\le O\left((N + |\Sigma|)F + (N + \sqrt{N}) \log_2 \frac{N}{F}\right). & (8.17) \end{aligned}$$

Отметим, что алгоритмы Замбалаева и Манбера — Майерса не оптимальны, т. к. у них сложность каждого прохода равна $O(N)$, а не $O(B_{F2^k}) + o(N)$. В наихудшем случае время работы продложенного алгоритма — не более $O(N \log N)$ операций — лучше, чем у алгоритма Замбалаева ($O(N \log^2 N)$) и такое же, как у алгоритма Манбера — Майерса, однако поскольку предложенный алгоритм на порядок проще описанного Манбером и Майерсом [129, 130], его реальная производительность на порядок выше.

Объем памяти, требуемой предложенному алгоритму, составляет

$$3.5N + 2\max\{\sqrt{N}, |\Sigma|\} \tag{8.18}$$

слов. Далее будет показано, что если N не очень велико ($\log_2 N \le \frac{2}{3}w + \varepsilon$, где w — размер слова в битах), то, не ухудшая скорости сортировки, объем требуемой памяти можно уменьшить на $1.5N$ слов и тем самым добиться такого же объема требуемой памяти (около $2N$ слов), что и у других алгоритмов сортировки.

8.5. УЛУЧШЕНИЕ ОСНОВНОГО АЛГОРИТМА

Можно заметить, что для хранения значения поля *value* достаточно $v = \lceil \log_2 N \rceil$ битов, а т. к. обычно N — размер блока — выбирается равным 1–2 Мб, то $v \approx 20$ меньше ширины машинного слова $w = 32$ или 64, поэтому старшие $(w - v)$ бит поля *value* никогда не используются и равны нулю.

Именно на этом месте старших неиспользуемых $(w - v)$ битов поля *value* и нужно хранить значение поля *index*. Однако количество различных групп может достигать значения $\lfloor \frac{N}{2} \rfloor$, в то время как самое большое целое, которое может быть размещено в $(w - v)$ битах, равно $(2^{w-v} - 1)$.

Положим

$$g = \min\{w - v, \left\lceil \log_2 \sqrt{N} \right\rceil\} \tag{8.19}$$

и отведем под массив G ровно 2^g слов, а сортируемый список будем обрабатывать не весь сразу, а участками по $(2^g - 1)$ групп. Если g_k — количество различных групп в списке на k-й итерации, то сортировка потребует

$$T'_k(N) = O(B_{F2^k} + \sqrt{N} \left\lceil \frac{g_k}{g} \right\rceil) \tag{8.20}$$

операций, а т. к.

$$g_k \le \left\lfloor \frac{B_{F2^k}}{2} \right\rfloor \tag{8.21}$$

то при $g \approx \sqrt{N}$

$$T'_k(N) = O(1.5B_{F2^k} + \sqrt{N}) = T_k(N), \tag{8.22}$$

и асимптотическая временная оценка не изменится.

Однако можно пойти несколько дальше и заметить, что группы строк малой мощности можно сортировать каким-либо другим алгоритмом *по одной*, как в алгоритме Замбалаева. Например, можно использовать *устойчивый алгоритм сортировки односвязных списков восходящим слиянием*, который сортирует n элементов не более чем за $n \log_2 n$ сравнений и при аккуратной реализации часто быстрее других классических методов сортировки.

Замечание 8.8. Перед началом сортировки списка необходимо заменить значение $i.value$ (предварительно сохранив номер текущей группы) на $i'.value$ с тем, чтобы i' вычислялось лишь единожды, аналогично предложенному автором решению для алгоритма Замбалаева (см. замечание 8.3). Однако в нашем случае можно ограничиться простым копированием значения $i'.value$ в $i.value$. Чтобы гарантировать корректность такого подхода, необходимо и достаточно, чтобы строка i' или принадлежала другой группе, или i' следовала за i в сортируемом списке строк группы, так что значение $i'.value$ всегда содержит корректное значение номера группы.

Заметим, что если i и i' принадлежат одной группе, то это означает, что $s_{i'} = Xs_i$, а $s_i = XX \dots XY$, где X и Y — некоторые строки, причем длина X равна $F2^k$, поэтому необходимо гарантировать, что *строки одной группы всегда следуют в порядке возрастания индексов*, т. е. $i < i.next$. Ввиду того, что и МЦ-сортировка при четном количестве проходов, и сортировка слиянием являются устойчивыми, выполнения вышеуказанного инварианта несложно добиться соответствующим порядком вставления строк на первом проходе МЦ-сортировки по первым F символам. ■

Ввиду того, что сортировка n ключей слиянием требует не более $O(n \log_2 n)$ сравнений, то при небольших $n \le 2^4 \dots 2^5$ группы мощности n более эффективно сортируются слиянием ($\log_2 n$ обращений к ключу), а не все сразу МЦ-сортировкой (4–5 обращений к ключу). Кроме того, сортировка слиянием обрабатывает данные *локально*, эффективно используя кэш-память процессора, обращение к которой в 3–5 раз быстрее обращения в оперативную память, поэтому сортировка слиянием групп мощности $n \le 2^6 \dots 2^7$ *быстрее* МЦ-сортировки всех неотсортированных групп одновременно.

Итак, строки групп мощности меньше 64–128 (скажем, 100), эффективнее сортировать слиянием, а строки групп большой мощности — вышеописанным алгоритмом, обрабатывающим за один проход g групп.

Так как количество различных групп мощности больше 100 на k-м проходе не превышает

$$g'_k = \left\lfloor \frac{B_{F2^k}}{100} \right\rfloor, \tag{8.23}$$

то при $g \approx \sqrt{N}$ время k-го прохода не превышает

$$T''_k(N) = C_1 B_{F2^k} + C_2 \left\lceil \frac{g'_k}{g} \right\rceil \sqrt{N} < \tag{8.24}$$

$$< C_1 B_{F2^k} + C_2 (\frac{B_{F2^k}}{100g} + 1)\sqrt{N} \approx \tag{8.25}$$

$$\approx (C_1 + 0.01C_2)B_{F2^k} + C_2\sqrt{N} \approx \tag{8.26}$$
$$\approx C_1 B_{F2^k} + C_2\sqrt{N} = \tag{8.27}$$
$$= T_k(N), \tag{8.28}$$

т. е. производительность модифицированного алгоритма с раздельной сортировкой групп малой и большой мощности менее чем на 1% хуже производительности первоначального варианта (а на практике несколько выше за счет несколько более быстрой сходимости), при этом объем требуемой памяти составляет

$$2N + \max\{7\sqrt{N}, 2|\Sigma|\} \tag{8.29}$$

слов, т. е. равен объему памяти, требуемой другим алгоритмам сортировки удвоением.

8.5.1. СРЕДНЯЯ ПРОИЗВОДИТЕЛЬНОСТЬ АЛГОРИТМА

Известно [130], что средняя длина общего начала двух соседних строк отсортированной матрицы циклических вращений строки S длины N, порожденной источником Бернулли с энтропией $\mathcal{E}$, равна $D(N) + \varepsilon$, где $D(N) = 2\log_{\mathcal{E}} N$ и $\varepsilon = O(1)$. Эксперименты, проведенные автором для большого собрания текстов на русском и английском языках (около 3.5 тыс. текстов общей длиной более 2 Гб) показали, что это правило с высокой точностью выполняется и для текстовых данных, причем в подавляющем большинстве случаев $|\varepsilon| \le 0.5$ (Манбер и Майерс также указывали на этот эмпирический факт [130]).

Таким образом, количество циклических вращений B_k строки S, неотличимых по первым k символам, при $k \ge D(N)$ мало, и в таком случае подавляющую часть времени сортировки (при $N \sim 10^6$ обычно более 95%) занимает сортировка по первым $D(N)$ символам, для чего требуется $\lceil \log_2 D(N) \rceil$ проходов. Поскольку $B_k \le N$, среднее время сортировки равно

$$T_{ave}(N) = O\big(N \log_2(2\log_{\mathcal{E}} N)\big), \tag{8.30}$$

а так как для текстов на естественных языках $\mathcal{E} \ge 4$ [163, 39, 166], то[5] $T_{ave}(N) = O(N \log\log N)$; этим и объясняется высокая средняя производительность алгоритмов сортировки удвоением и СЦ-сортировок, предложенных Уилером.

8.5.2. МЕТОДЫ УСКОРЕНИЯ АЛГОРИТМА

Для начальной сортировки по первым F символам лучше всего использовать алгоритм сортировки расстановкой для списков, как описано в п. 8.1.1. Так как, скорее всего, $|\Sigma| \ll \sqrt{N}$, то можно использовать вышеописанные алгоритмы вставки строки в голову списка, где в качестве ключа используется k-й символ ($k = F-1, \ldots, 1, 0$) текущей строки. Ввиду того, что вставка элемента в голову списка меняет начальный порядок строк на обратный, F должно быть четным. Кроме того, т. к. каждая итерация вышеописанного алгоритма удваивает глубину отсортированности, то F не следует выбирать слишком большим; обычно $F = 4$ является оптимальным значением.

Так как операция взятия по модулю, используемая при вычислении $i' = (i + n) \bmod N$, требует деления — очень медленной операции, — то лучше воспользоваться тем, что $0 \le i, n < N$, поэтому

$$i' = (i+n) \bmod N = \begin{cases} i+n, & i+n < N, \\ i+n-N, & i+n \ge N. \end{cases} \tag{8.31}$$

[5] Аналогичные оценки средней скорости сортировки удвоением получены Манбером и Майерсом [130].

Кроме того, можно заметить, что B_k (длина сортируемого списка) быстро — как правило, экспоненциально — убывает с ростом k, поэтому подавляющая часть работы (более 95% времени) обычно приходится на первые две-три итерации. Чтобы на этих итерациях избежать взятия по модулю вообще, можно скопировать первые $F2^k$ элементов (где k — номер текущей итерации) массива *node* в его конец, так что $(i + F2^k)$ будет разрешенным значением индекса. Автор рекомендовал бы поступать таким образом до тех пор, пока $k \leq 0.5 \log_2 N = \log_2 \sqrt{N}$, так что первые проходы выполнятся очень эффективно. Такое решение увеличивает объем требуемой памяти на $\sqrt{N}$ слов; при $N \sim 10^6$ расход дополнительных $\sqrt{N} \sim 10^3 \ll N$ слов памяти можно не принимать во внимание, т. к. это составляет менее 0.05% от общего объема требуемой памяти.

Аналогично, необходимо скопировать F первых символов массива $buff$ в его конец, чтобы $(i + F - 1)$ было разрешенным значением индекса массива $buff$.

Следует отметить, что Уилер в своей реализации BWT BRED3 использовал большинство приемов, описанных выше, чем во многом и объясняется чрезвычайно высокая производительность его программы.

Подавляющая часть работы алгоритма сортировки сводится к цифровой сортировке списков расстановкой, описываемой следующим псевдокодом (просмотр списка и вставка текущего элемента списка в голову подсписка согласно цифровому ключу):

```
do {
  q = p->next;
  t = & TABLE [KEY(p)];
  p->next = *t;
  *t = p;
  p = q;
} while (p != NULL);
```

Так как большинство современных процессоров способны одновременно исполнять две и более команд, следующих друг за другом, то при одновременной обработке двух элементов списка можно ускорить сортировку почти вдвое. Для этого необходимо использовать следующую последовательность команд:

```
do {
  p2 = p1->next;
  t1 = & TABLE [KEY(p1)];
  t2 = & TABLE [KEY(p2)];
  p1->next = *t1;
  tmp = p2->next;
  *t1 = p1;
  p1 = tmp;
  p2->next = *t2;
  *t2 = p2;
} while (p1 != nil);
if (p2 == nil) {
  *t2 = p2->next;
  p2->next = p2;
}
```

Вычисление значения функции `KEY(p)` также следует разбить на примитивные операции вида `A operation B` и перемешать вычисления для двух элементов списка ана-

логично тому, как это сделано выше.

Так как при нечетной длине списка будет обработан элемент, следующий за последним элементом списка, то в качестве признака окончания списка следует использовать указатель `nil` на особую запись, так что указатель на следующий элемент снова указывает на эту же запись, `nil->next = nil`, а остальные поля записи заполнены так, чтобы значение функции `KEY(nil)` вычислялось корректным образом и `KEY(nil)` было допустимым значением индекса массива `TABLE`. Кроме того, при нечетной длине списка запись `nil` будет вставлена в таблицу первым элементом списка, и в таком случае необходимо восстановить правильное значение этого элемента таблицы (указываемого переменной `t2`), которое хранится в поле `nil->next`, и установить значение этого поля в `nil`.

8.6. ВЫВОДЫ

Автором построен алгоритм сортировки последовательности N циклических вращений строки S длины N на $0, \ldots, (N-1)$ символ, являющийся центральной частью алгоритма сжатия данных сортировкой блоков, скорость работы которого и объем требуемой памяти в основном (более чем на 80%) определяются соответствующими параметрами алгоритма сортировки.

Показано, что время работы алгоритма не превышает

$$O\Big((N+|\Sigma|)F + \sum_{k=0}^{K-1}(B_{F2^k} + \sqrt{N})\Big), \tag{8.32}$$

что близко к оптимальному значению

$$O\Big(NF + \sum_{k=0}^{K-1} B_{F2^k}\Big), \tag{8.33}$$

где B_n — количество строк–циклических вращений кодируемой строки, не отличимых по первым n символам, а

$$K = \left\lceil \log_2 \frac{N}{F} \right\rceil \tag{8.34}$$

— максимальное количество проходов. Так как $B_n \le N$, то максимальное время работы предложенного алгоритма блочной сортировки заведомо не превышает

$$O\Big(N \log_2 N\Big); \tag{8.35}$$

показано, что среднее время сортировки для текстов на естественных языках равно

$$O\Big(N \log_2 \log_2 N\Big). \tag{8.36}$$

Объем памяти, требуемый алгоритмом, составляет $(2N + \max\{7\sqrt{N}, 2|\Sigma|\} + O(1))$ слов и $(N+F)$ байтов.

Преимуществом предложенных алгоритмов является простота реализации и, в связи с этим, высокая практическая скорость сортировки.

Эмпирические исследования показали (см. главу 11), что при таком же или сравнимом объеме требуемой памяти время сортировки в 1.5–3 раза меньше времени работы всех других известных алгоритмов блочной сортировки (Манбера — Майерса, Замбалаева, Барроуза — Уилера) и в среднем и в наихудшем случае; о высокой производительности предложенного алгоритма также свидетельствует тот факт, что время МЦ-сортировки строк по первым четырем символам лишь в 1.5–2.5 раза меньше времени полной сортировки.

Единственным исключением является алгоритм BRED3 Уилера, который требует вдвое меньше памяти при почти такой же средней скорости сортировки. Однако (см. главу 11) BRED3 очень чувствителен к входным данным (наличие повторящихся строк немедленно вызывает увеличение времени сортировки в сотни раз); ввиду того, что двоичные данные часто содержат много повторов, алгоритм BRED3 неприемлем при создании надежного программного обеспечения.

Таким образом (см. главу 11), предложенный алгоритм при сравнительно небольшом объеме требуемой памяти обладает очень высокой средней производительностью, опережая по этому параметру все другие известные алгоритмы, а в наихудшем случае скорость работы остается на приемлемом уровне.

Глава 9

СЖАТИЕ ЧАСТИЧНОЙ СОРТИРОВКОЙ БЛОКОВ

Один из способов эффективной реализации преобразования Барроуза — Уилера, которое заключается в сортировке по строкам матрицы M размера $N \times N$ и полученной циклическими вращениями исходной строки S длины N на $0, 1, \ldots, (N-1)$ символов, представлен в п. 8. Однако можно заметить, что если отсортировать строки матрицы M не полностью, а лишь по первым F символам, то при достаточно большом F коэффициент сжатия будет мало отличаться от коэффициента сжатия при полной сортировке, т. к. достаточно длинные контексты (длины F) должны обеспечивать мало отличающееся от контекстов длины N качество предсказаний, т. к. фактическая средняя длина контекста (т. е. количество первых совпадающих символов у соседних строк отсортированной матрицы M') обычно достаточно небольшая.

Таким образом, можно отсортировать строки матрицы M по первым F символам (с чего начинаются все известные алгоритмы сортировки матрицы M) и, не досортировывая M, выдать символы последнего столбца и индекс начальной строки в частично отсортированной матрице M.

Преимущества такого подхода очевидны:

- чрезвычайно малое среднее время[1] кодирования (CF команд на символ, где константа C зависит от целевой архитектуры и обычно равна 5–8), что существенно меньше аналогичного показателя для почти всех других известных методов сжатия;
- низкий расход требуемой памяти — $(N + |\Sigma|)$ слов памяти (для организации сортировки расстановкой) и $(N + F)$ байтов памяти для хранения кодируемой строки;
- достаточно хорошее качество сжатия, (предположительно) мало отличающееся от качества сжатия, достижимого при использовании преобразования Барроуза — Уилера с полной сортировкой.

Кроме того, такой вариант BWT (в дальнейшем обозначаемый PBWT — Partial Burrows–Wheeler Transformation) обладает одним достаточно редким и иногда чрезвычайно полезным качеством: время сжатия блока данных — практически постоянная величина и очень мало зависит от собственно сжимаемых данных. Это позволит использовать данный метод в системах сжатия с жесткими временными ограничениями: встроенных (embedded) системах реального времени, системах связи, файловых системах со сжатием и т. д., где в настоящее время обычно используются алгоритмы семейства LZ77 с небольшой шириной окна и/или очень жесткими ограничениями на длину перебираемого хеш-списка, что отрицательно сказывается на качестве сжатия.

[1] Все оценки памяти и времени работы алгоритмов приводятся для RAM-машины, описанной в п. 2.1.

Для того чтобы алгоритм сжатия данных, основанный на частичном преобразовании Барроуза — Уилера, можно было использовать на практике, необходимо убедиться, что качество сжатия PBWT действительно мало отличается от качества сжатия, достижимого при использовании BWT, и, самое главное, построить обратное преобразование и убедиться, что время восстановления исходных данных и объем требуемой памяти находятся в приемлемых пределах.

9.1. ПОСТРОЕНИЕ ОБРАТНОГО ПРЕОБРАЗОВАНИЯ

В дальнейшем будем полагать, что матрица M'_F получена из матрицы M циклических вращений исходной строки S длины N сортировкой M по первым F символам строк (например за F проходов МЦ-сортировки расстановкой), причем сортировка *устойчивая*, т. е. если первые F символов строк s_i и s_k совпадают, то в матрице M'_F строка s_i предшествует s_k тогда и только тогда, когда $i < k$.

Обозначим через L последний столбец матрицы M'_F, т. е. L — это та последовательность символов, которая передается декодировщику. Будем полагать, что декодировщику также сообщается индекс в M' строки $s_{N-1} = s{-}1$, полученной вращением исходной строки S на один символ вправо.

Первый этап восстановления исходной строки S заключается в восстановлении F первых столбцов матрицы M'_F. Так как каждый столбец матрицы M и, соответственно, M'_F содержит те же символы, что и строка S (возможно, в ином порядке), то для того, чтобы восстановить первый столбец $M'_F[0]$ матрицы M'_F, достаточно отсортировать столбец L по возрастанию символов (проще и эффективнее всего это сделать сортировкой расстановкой, как описано в п. 8.1.1).

Итак, нам известны последний и первый столбцы матрицы M'_F. Ввиду того, что каждая строка матрицы M'_F получена циклическим вращением строки S на некоторое количество символов, все пары символов, встречающиеся в строке S, известны. Действительно, пара символов $\alpha\beta$ встречается в S столько раз, сколько раз выполняется условие $L[i] = \alpha$ и $M'_F[i][0] = \beta$ при $i = 0, \ldots, N-1$.

Если $F > 1$, то необходимо восстановить второй столбец $M'_F[1]$. Пусть первый символ первого столбца M'_F повторяется n_0 раз, т. е. $\alpha_0 = M'_F[0][0] = \ldots = M'_F[n_0 - 1][0] < M'_F[n_0][0]$. Рассмотрим все n_0 пар символов, начинающихся с α_0, и отсортируем их по второму символу; полученная последовательность вторых символов и есть первые n_0 символов второго столбца.

На самом деле рассматривать все такие пары символов и производить сортировку нет необходимости. Пусть $T(0)$ — индекс первого, $T(1)$ — второго, $\ldots$, $T(n_0 - 1)$ — последнего вхождения α_0 в столбец L, тогда указанная этими индексами последовательность символов первого столбца

$$M'_F[T(0)][0], \ldots, M'_F[T(n_0 - 1)][0] \tag{9.1}$$

и есть искомая (уже отсортированная по построению первого столбца M'_F) последовательность вторых символов пар, т. е. для $i = 0, \ldots, (n_0 - 1)$

$$M'_F[i][1] = M'_F[T(i)][0]. \tag{9.2}$$

Аналогично, если следующий за α_0 символ первого столбца $\alpha_1 = M'_F[n_0][0]$ встретился n_1 раз и $T(n_0)$ — индекс первого, $\ldots$, $T(n_0 + n_1 - 1)$ — индекс последнего вхождения символа α_1 в столбец L, то для $i = n_0, \ldots, (n_0 + n_1 - 1)$ следующие n_1 символов второго столбца определяются согласно

$$M'_F[i][1] = M'_F[T(i)][0]. \tag{9.3}$$

Проведя аналогичные рассуждения для символов третьего столбца (при $F > 2$), получаем

$$M'_F[i][2] = M'_F[T(i)][1] = M'_F[T^2(i)]][0], \tag{9.4}$$

поэтому при $k = 1, \ldots, F-1$

$$M'_F[i][k] = M'_F[T(i)][k-1] = M'_F[T^k(i)][0], \tag{9.5}$$

а т. к. $M'_F[i][0] = L[T(i)]$, то для $k = 0, \ldots, F-1$

$$M'_F[i][k] = L[T^{k+1}(i)]. \tag{9.6}$$

Итак, первые F столбцов матрицы M'_F восстановлены, и следующий этап заключается в том, чтобы для всех $i = 0, 1, \ldots, (N-1)$ найти месторасположение строки s_i, полученной вращением исходной строки S на i символов влево.

Разобьем строки M'_F на группы так, что две строки принадлежат одной группе тогда и только тогда, когда у них совпадают первые F символов, и занумеруем группы следующим образом: номер первой группы равен нулю, а номер следующей получается увеличением номера предыдущей группы на количество строк, содержащихся в предыдущей группе. Пусть $G[i]$ — номер группы, которой принадлежит i-я строка матрицы M'_F. Кроме того, для каждой группы с номером n запишем в ячейку $I[n]$ массива I индекс последней строки, принадлежащей этой группе.

По определению, декодировщику известен номер j_{N-1} строки s_{N-1} в матрице M'_F. Заметим, что s_{N-1} — последняя строка в своей группе с номером $G[j_{N-1}]$, т. е. $I[G[j_{N-1}]] = j_{N-1}$. Итак, j_{N-1}-ая строка матрицы M'_F есть s_{N-1}; присвоим ячейке $R_{N-1} = j_{N-1}$. Так как последняя строка группы $G[j_{N-1}]$ установлена, уменьшим $I[G[j_{N-1}]]$ на единицу, т. е. в дальнейшем $I[n]$ будет содержать индекс в M'_F последней строки n-й группы, номер которой еще не установлен.

Так как известны первые F символов строки s_{N-1}, равные

$$M'_F[j_{N-1}][0] \ldots M'_F[j_{N-1}][F-1], \tag{9.7}$$

и предшествующий им символ $L[j_{N-1}]$, то строка s_{N-2} начинается с символов

$$L[j_{N-1}]M'_F[j_{N-1}][0] \ldots M'_F[j_{N-1}][F-1]. \tag{9.8}$$

Кроме того, s_{N-2} является последней строкой среди всех строк, первые F символов которых совпадают с первыми F символами s_{N-2}, если первые F символов s_{N-2} и s_{N-1} различны, иначе s_{N-2} предшествует s_{N-1}. В любом случае, индекс s_{N-2} в M'_F равен $j_{N-2} = I[G[n]]$, где n — индекс в M'_F произвольной строки, первые F символов которой равны

$$L[j_{N-1}]M'_F[j_{N-1}][0] \ldots M'_F[j_{N-1}][F-2]. \tag{9.9}$$

Установим R_{N-2} в j_{N-2} и уменьшим $I[G[n]]$ на единицу.

Заметим, что если $T(n) = j_{N-1}$, то по построению первых F столбцов матрицы M'_F n-я строка M'_F начинается с символов

$$L[j_{N-1}]M'_F[j_{N-1}][0] \ldots M'_F[j_{N-1}][F-2]; \tag{9.10}$$

таким образом, $n = T^{-1}(j_{N-1})$.

Аналогичным образом отыскивается положение остальных строк: $n' = T^{-1}(j_{N-2})$ дает индекс в M'_F строки, начинающейся с символов

$$L[j_{N-2}]M'_F[j_{N-2}][0] \ldots M'_F[j_{N-2}][F-2]; \tag{9.11}$$

так что индекс строки s_{N-3} в M'_F равен

$$j_{N-3} = I[G[n']] = I[G[T^{-1}(j_{N-2})]]. \tag{9.12}$$

Установив R_{N-3} в j_{N-3} и уменьшив $I[G[n']]$ на единицу, перейдем к отысканию $n'' = T^{-1}(j_{N-3})$ и т. д.

Таким образом, R_i содержит индекс i-й строки s_i в M'_F, и т. к. первый символ i-й строки s_i, равный $M'_F[R[i]][0] = L[T(R_i)]$, есть не что иное, как i-й символ исходной строки

$$S_i = L[T(R_i)] = M'[R_i][0], \tag{9.13}$$

восстановление исходной строки закончено.

Замечание 9.1. Так как полное преобразование Барроуза — Уилера есть частичное преобразование при $F = N$, то обратное преобразование BWT^{-1} может рассматриваться как частный случай $PBWT^{-1}$ с тем отличием, что строить массив номеров групп и т. д. не нужно, т. к. ввиду полной отсортированности M' по строкам размер каждой группы равен единице, т. е. $G[n] = I[n] = n$ для всех $n = 1, \ldots, (N-1)$, поэтому $R_{N-1} = j_{N-1}$, $R_{N-2} = T^{-1}(R_{N-1})$, $\ldots$, $R_i = T^{-1}(R_{i+1})$, поэтому даже построение массива R не требуется. Действительно, если известен индекс $R_0 = n$ исходной строки S в отсортированном массиве M', то

$$\begin{array}{rcrcrcl} R_{N-1} & = & T^{-1}(R_0) & \Longrightarrow & S_{N-1} & = & L[n], \\ R_{N-2} & = & (T^{-1})^2(R_0) & \Longrightarrow & S_{N-2} & = & L[T^{-1}(n)], \\ \ldots & & & & & & \\ R_{N-k} & = & (T^{-1})^k(R_0) & \Longrightarrow & S_{N-k} & = & L[(T^{-1})^{k-1}(n)]; \end{array} \tag{9.14}$$

данный алгоритм во многом похож на несколько более сложный метод, предложенный Барроузом и Уилером [55]. ■

9.2. ЛОГАРИФМИЧЕСКИЙ АЛГОРИТМ

Внимательный анализ формулы

$$M'_F[i][k] = M'_F[T^j(i)][k-j] \tag{9.15}$$

позволяет заключить, что при $k = 2^n$ и $m = 0, 1, \ldots, k-1$

$$M'_F[i][k+m] = M'_F[T^k(i)][m]. \tag{9.16}$$

Таким образом, построив за $O(N)$ операций T и T^2, можно по последнему столбцу L восстановить нулевой и первый столбцы M'_F, затем, копируя нулевой и первый символы $T^2(i)$-й строки, восстановить второй и третий символы i-й строки, потом возвести T^2 в квадрат и копированием символов 0–3 строки $T^4(i)$ восстановить 4–7-й символы i-й строки и т. д.

Итак, первые F символов строк матрицы M'_F можно восстановить за $\lceil \log_2 F \rceil$ проходов, однако автор не рекомендует использование асимптотически более оптимальной версии обратного преобразования на практике, т. к. при небольших $F \leq 8$ скорость логарифмического алгоритма ниже скорости линейного при почти вдвое большем объеме требуемой памяти.

9.3. ЭФФЕКТИВНАЯ РЕАЛИЗАЦИЯ ОБРАТНОГО ПРЕОБРАЗОВАНИЯ

Первое, что нужно сделать — это получить перестановку t, обратную к T. Для этого необходимо подсчитать n_i — количество появлений i-го символа в столбце L,

после чего построить массив

$$n'_i = \sum_{k=0}^{i} n_k = n_i + n'_{i-1} \tag{9.17}$$

($n'_0 = 0$) и, просматривая столбец L снизу вверх, для каждого символа $L[k]$ при $k = N-1, \ldots, 0$ уменьшать $n'_{L[k]}$ на единицу и устанавливать $t(n'_{L[k]})$ в k; временная сложность этой операции не превышает $O(N + |\Sigma|)$.

Заметим, что, строго говоря, знание содержимого первых F столбцов матрицы M'_F необходимо лишь для того, чтобы построить разбиение строк на группы, поэтому имеет смысл сразу же строить это разбиение. Сначала построим разбиение, соответствующее первому столбцу, следующим образом: $g[n'_0] = n'_0 = 0$, $g[n'_1] = n'_1$ и т. д., где n'_i получены согласно (9.17) (т. е. до построения перестановки t). Кроме того, восстановим первый столбец M'_F в массив X_0, заполнив n_0 первых элементов первым символом алфавита, n_1 следующих элементов вторым символом алфавита и т. д.

Если $F > 1$, то необходимо восстановить второй столбец матрицы M'_F в массив X_1 согласно формуле

$$M'_F[i][1] = M'_F[T(i)][0] \iff X_1[i] = X_0[T(i)] \iff X_1[t[i]] = X_0[i] \tag{9.18}$$

и разбить группы, порожденные первым столбцом, на подгруппы. Для этого необходимо, проходя по массиву g (начиная с $i = 0$), просмотреть $k = g[i]$ символов массива X_1, начиная с i-го, и присвоить $g[i']$ значение k', если, начиная с i'-го символа, следует подгруппа из k' одинаковых символов, т. е.

$$X_1[i'-1] \neq X_1[i'] = X_1[i'+1] = \ldots X[i'+k'-1] \neq X_2[i'+k'], \tag{9.19}$$

$i \leq i' < i' + k' \leq i + k$, после чего увеличивать i на k до тех пор, пока $i \neq N$.

Если $F > 2$, то необходимо восстановить третий столбец M'_F, записав его на место строки L (сохраняя первый столбец X_0),

$$X_2[t[i]] = X_1[i], \tag{9.20}$$

и снова разбить группы на подгруппы; четвертый столбец надо записать на место второго и т. д.

Таким образом, после F разбиений $g[0]$ содержит количество элементов в первой группе, $g[g[0]]$ — во второй с номером $g[0]$ и т. д.

Вместо того чтобы использовать дополнительный массив I, воспользуемся тем, что элементы группы заполняются с конца, поэтому можно установить $g[i]$ в n, где n — номер группы, которой принадлежит i-я строка матрицы M'_F, а если $i = n$ (т. е. i — первая строка группы), то установить $g[i]$ в номер последней строки этой группы. Итак,

$$I[G[n]] = \begin{cases} g[g[n]], & g[n] < n, \\ g[n], & g[n] \geq n, \end{cases} \tag{9.21}$$

что позволяет правильно заполнить массив R следующим образом. Предварительно установив $R[N-1] = j_{N-1}$, присвоим n значение индекса j_{N-1} строки s_{N-1} в M'_F. Для всех $k = N-2, \ldots, 0$ вычислим $m = g[t(n)]$, присвоим n и $R[k]$ значение $g[m]$ и уменьшим $g[m]$ на единицу.

Наконец, необходимо восстановить исходную строку S по формуле

$$S_i = M'_F[R_i] = X_0[R_i]. \tag{9.22}$$

Таким образом, реализация обратного преобразования требует $(10N + 8|\Sigma|)$ байтов (около $2.5N$ слов) памяти для размещения массивов t, g, L, X_0, n и n'.

9.4. ВЫВОДЫ

Автору не удалось получить теоретических оценок качества сжатия частичной сортировкой блоков; поскольку такие оценки не известны и для сжатия полной сортировкой блоков, сравнение этих методов между собой, равно как и с другими методами сжатия, возможно только эмпирически. Результаты такого сравнения приводятся в главе 11.

Предложенный автором алгоритм сжатия данных путем частичной сортировки блоков и кодированием выхода алгоритма, как описано в главе 10, позволяет добиться очень высокого качества сжатия (лишь на 1–5% уступающего лучшим алгоритмам сжатия) при исключительно высокой скорости работы, превышающей скорость подавляющего числа известных алгоритмов сжатия данных. Существенным (и достаточно уникальным) преимуществом данного алгоритма является то, что время его работы практически не зависит от собственно кодируемых данных и, при фиксированной глубине сортировки и мощности кодируемого алфавита, линейно зависит только от длины кодируемого сообщения, что позволяет использовать этот алгоритм сжатия в системах с жесткими временными ограничениями: встроенных системах, коммуникационных системах, дисковых массивах и т. п.

К числу недостатков этого алгоритма относится достаточно большой объем требуемой оперативной памяти, примерно в 10–20 раз превышающий таковой для алгоритмов сжатия семейства LZ77 — единственных конкурентов данного алгоритма по соотношению качества и скорости сжатия, которые при гораздо худшем качестве сжатия несколько быстрее предложенных алгоритмов.

Кроме того, данный алгоритм сжатия на порядок уступает алгоритмам семейства LZ77 в скорости декодирования (которая практически не отличается от скорости кодирования). Впрочем, последние уникальны в смысле скорости декодирования, сравнимой со скоростью копирования участков памяти; для подавляющего числа других алгоритмов сжатия время декодирования мало отличается от времени кодирования, поэтому тот факт, что алгоритм сжатия частичной сортировкой блоков также обладает этим качеством, не следует относить к числу его недостатков.

Глава 10

ИНТЕРВАЛЬНОЕ КОДИРОВАНИЕ

Как отмечалось в п. 2.9, результат применения преобразования Барроуза — Уилера к строке S длины N над алфавитом Σ — последний столбец L матрицы M', являющейся отсортированной по строкам матрицей M, полученной циклическими вращениями S на $0, 1, \ldots, (N-1)$ символов. Обычно эта последовательность символов содержит длинные серии повторений одного символа, иногда перемежающегося другими символами, и из-за очевидной избыточности L можно достичь высокого качества сжатия, применяя достаточно простые методы локально-адаптивного интервального кодирования [26, 25, 76, 98].

Ввиду того, что результирующее качество сжатия определяется именно качеством конечного кодирования, разработке методов кодирования выхода BWT следует уделить самое пристальное внимание, добиваясь как высокого качества сжатия, так и высокой скорости кодирования и декодирования.

Следует отметить, что в данном случае автора мало интересовали теоретические оценки качества сжатия, т. к. статистические свойства выхода алгоритма блочной сортировки в настоящее время не известны, а выводить оценки на основе некоторых предположений — достаточно неблагодарное и часто бесполезное занятие, потому что фактическое положение дел может оказаться весьма далеким от предполагаемого. Кроме того, для локально-адаптивных алгоритмов качество сжатия очень сильно зависит от кодируемой последовательности (даже для адаптивного кода Хаффмана качество сжатия может измениться почти вдвое при перестановке символов в кодируемой последовательности) и незначительные изменения кодируемых данных могут заметно повлиять на качество сжатия. Учитывая тот факт, что в настоящее время качество сжатия относительно слабых алгоритмов лишь на 30–50% уступает качеству кодирования лучших, автор предпочел опираться на эмпирические данные.

10.1. МЕТОД СТОПКИ КНИГ

Барроуз и Уилер [55] показали, что применение метода стопки книг [26, 25, 76, 98], разработанного Б.Я. Рябко, дает хорошее качество сжатия, лишь на 15–20% уступающее качеству сжатия лучших вариантов статистического моделирования (см. пп. 2.7, 2.9 и главу 11). Метод стопки книг заменяет текущее вхождение символа количеством различных символов, разделяющих текущий символ и предыдущее его вхождение, что равносильно тому, что символ заменяется его номером в перестановке символов алфавита Σ, после чего закодированный переставляется на первое место в этой перестановке.

10.1.1. КОДИРОВАНИЕ НУЛЕЙ

Так как большое количество повторений одного символа приводит к появлению большого количества нулей, необходимо применять кодирование длин серий RLE [93] (Run-Length Encoding), заменяя

$$\underbrace{0\ldots0}_{n} \tag{10.1}$$

парой $(0, n)$, в свою очередь используя энтропийное кодирование (код Хаффмана [18, 105] или арифметический код [147, 195]) для передачи длины серии n. Так как n может быть очень большим, то для ограничения размера требуемых структур данных (а при статическом кодировании — и объема сообщаемой декодировщику информации о принятом способе кодирования) выбирают некоторую константу C и при $n \geq C$ вместо пары $(0, n)$ передают последовательность

$$0\underbrace{C\ldots C}_{k}n', \tag{10.2}$$

так что $n = Ck + n'$.

Обычно при кодировании выхода BWT вероятность $p(n)$ того, что длина серии нулей равна n, подчиняется закону Ципфа, т. е. $p(n)$ пропорционально $\frac{1}{n}$ (см. [83, 84]), поэтому для повышения эффективности кодирования можно расширить алфавит символами 0_2, 0_3, …, 0_C и кодировать последовательности 00, 000 и т. д. символами 0_2, 0_3, …, 0_C, а при $n > C$ применять *групповое кодирование* (см. п. 4.2), заменяя n парой $\left(L(n), n - 2^{L(n)}\right)$, где $L(n) = \lfloor \log_2 n \rfloor$, используя энтропийное кодирование для первого элемента пары и равномерное кодирование (т. е. в точности $L(n)$ младших значащих битов) для второго.

Замечание 10.1. Кодировать собственно символ 0 не нужно, т. к. достаточно расширить алфавит символами α_1, α_2, …и кодировать $L(n)$ символом $\alpha_{L(n)}$. ■

Довольно остроумный способ кодирования длин серий был реализован Уилером [185]. Вместо того, чтобы передавать пару $(0, n)$, расширим кодируемый алфавит символами α_0 и α_1 и, представив $(n + 1)$ в системе счисления по основанию 2 ($n + 1 = c_k 2^k + \ldots + c_0 2^0$, $c_k = 1$, $c_j \in \{0, 1\}$ при $j = 0, 1, \ldots, k - 1$), закодируем n, передавая символы α_{c_j} при $j = 0, 1, \ldots, k - 1$ (c_k передавать не нужно, т. к. достоверно известно, что $c_k = 1$ и $n \geq 1 \iff n + 1 \geq 2$); собственно символ 0 кодировать также не нужно.

Обычно качество кодирования Уилера примерно на 5% хуже описанного варианта RLE при кодировании выхода BWT для текстов на естественных языках и примерно на 5% лучше при кодировании двоичных данных (исполняемых программ, объектных файлов и т. д.). Автор рекомендовал бы использовать смешанную стратегию, кодируя символы 0_n при небольших n и применяя кодирование Уилера при больших значениях n, что позволяет несколько улучшить качество сжатия.

10.1.2. ЭФФЕКТИВНАЯ РЕАЛИЗАЦИЯ МЕТОДА СТОПКИ КНИГ

Реализация метода стопки книг требует реализации двух операций:

- $index(\alpha)$, возвращающей номер символа α в текущей перестановке символов Σ и затем перемещающей его на первое место (используется при кодировании);
- $symbol(i)$, возвращающей i-й символ перестановки и затем переставляющей его на первое место (используется при декодировании).

Традиционно (см. [39, 45, 55, 76, 166], а также исходные тексты архиваторов BRED, BRED3, BZIP и TZIP, использующих метод стопки книг) эти операции реализуются прямым перебором перестановки символов кодируемого алфавита, что, по мнению автора, неприемлемо при создании надежных систем сжатия, т. к. при плохом качестве сжатия среднее время поиска символа составит $O(|\Sigma|)$ и время кодирования N символов составит $O(N|\Sigma|)$, что при $|\Sigma| = 256$ (расширенный код ASCII) неприемлемо много. Таким образом, при плохом качестве сжатия время кодирования и декодирования может стать чрезмерно большим.

Однако существует[1] достаточно элегантное решение этой проблемы, основанное на способе представления списков с произвольным доступом, описанным Кнутом [18].

Создадим сбалансированное дерево (AVL, red-black, 2-3, 1-2-3 и т. п.) с $|\Sigma|$ листьями и пометим его листья символами алфавита Σ. Кроме того, пометим каждую вершину дерева количеством листьев в поддереве с корнем в данной вершине. Будем полагать, что символ α_1 предшествует в перестановке символу α_2, если лист, помеченный α_1, расположен левее листа, помеченного α_2.

Для того, чтобы реализовать операцию $index(\alpha)$, достаточно, проходя по дереву снизу вверх, уменьшать на единицу счетчик количества листьев текущей вершины и суммировать значения счетчиков братских к текущей *левых* вершин дерева; при достижении корня дерева сумма счетчиков братских левых вершин и будет номером символа α в перестановке. После этого лист, помеченный α, следует удалить из дерева (обычно эту операцию можно совместить с обходом дерева от листа к корню) и вставить в дерево *самой левой вершиной*, увеличив на единицу значения счетчиков листьев вершин, расположенных на новом пути к корню (что, как правило, можно совместить со вставкой).

Для реализации операции $symbol(i)$ $(i = 0, 1, \ldots, |\Sigma| - 1)$ необходимо, начиная с корня дерева, переходить к левому поддереву, если значение ℓ счетчика листьев его корня больше i, и к правому, если $\ell \le i$; при переходе к правому поддереву i необходимо уменьшить на ℓ. При достижении листа дерева i должно быть равно нулю, а символ, помечающий данный лист, и является искомым. После этого необходимо исключить найденный лист из дерева, уменьшить значения счетчиков листьев и вставить лист в дерево самым левым, увеличив значения счетчиков листьев вершин на новом пути к корню.

Так как сбалансированные деревья обычно используются для реализации деревьев двоичного поиска, максимальная глубина которых не превышает $C \log_2 n$, а время исключения и добавления вершины не превышает $C' \log_2 n$, где n — количество листьев дерева, то при использовании сбалансированных деревьев временная сложность операций $index(\alpha)$ и $symbol(i)$ не превышает $O(\log_2 |\Sigma|)$.

Для повышения средней эффективности поиска индексов и символов автор рекомендовал бы использовать прямой перебор для первых m элементов перестановки и поиск по дереву для символов с индексами больше m, где m — небольшая константа. Так как при сжатии сортировкой блоков с кодированием по методу стопки книг индекс кодируемого символа в подавляющем числе случаев (более 80%) не превышает 4, то при $m = 4$ можно добиться высокой производительности метода стопки книг как в среднем ($O(1)$ операций на символ), так и наихудшем случае ($O(\log_2 |\Sigma|)$ операций на символ).

Эмпирические исследования, проведенные автором в 1994–1995 гг. и, независимо от

[1] Автор не претендует на право первооткрывателя описываемого метода; поскольку он опирается на хорошо известные алгоритмы, возможно, аналогичные решения предлагались ранее, однако автору о них не известно.

него, Фенвиком [82, 83, 84], показали, что качество сжатия ухудшается при использовании других эвристик (LFU, Move-One-Up и Move-Half-Up [98]) для локально-адаптивного кодирования, аналогичных эвристике LRU, на которой основан метод стопки книг.

10.2. INVERTED FREQUENCY CODING

К сожалению, метод стопки книг — не наилучший способ кодирования выхода алгоритма блочной сортировки, т. к. разбиение последовательности L символов последнего столбца отсортированной матрицы M' на сегменты размером 16–20 Кб, кодируемых независимо, улучшает качество сжатия (как и было сделано Барроузом и Уилером [55]). Это означает, что статистика выхода алгоритма стопки книг, в свою очередь, также быстро изменяется, что однозначно говорит о том, что метод стопки книг — не лучший предсказатель кодируемой последовательности.

В 1997 г. Арнавутом и Магливеросом [34] был предложен интересный вариант интервального кодирования — Inverted Frequency Coding (IF), при котором сначала передается количество разделяющих символов для каждого вхождения первого символа алфавита, затем второго, и т. д.; это равносильно тому, что каждое вхождение символа заменяется количеством лексикографически больших символов, разделяющих текущее и предыдущее вхождение символа (для предотвращения появления проблемы нулевой вероятности в дальнейшем предполагается, что для первого вхождения предыдущее вхождение непосредственно предшествует кодируемой последовательности символов). Например, строка *bbaaccaabb* будет преобразована в последовательность *(a,2020)(b,0020)(c,00)*.

Декодирование производится в обратном порядке — от лексикографически старших символов к младшим вставкой текущего символа после пропуска соответствующего количества символов предыдущей строки, так что в нашем примере для символов *c*, *b*, и *a* будут последовательно получены строки *cc*, *bbccbb*, и *bbaaccaabb*.

Как указано авторами алгоритма, качество сжатия сортировкой блоков с последующим IF-преобразованием обычно незначительно (менее 1%) лучше кодирования методом стопки книг. Казалось бы, столь несущественное улучшение качества сжатия (при заметном усложнении алгоритма кодирования) не заслуживает внимания, если бы не одна деталь: разбиение кодируемой последовательности на сегменты *не улучшает* качество сжатия, т. е. IF-преобразование является достаточно хорошим предсказателем. Самое важное, это допускает создание моделей высоких порядков, позволяющих существенно улучшить качество сжатия, которое уже очень мало (менее 1–2%) отличается от качества сжатия, достижимого при использовании лучших алгоритмов статистического моделирования.

10.2.1. УЛУЧШЕНИЕ КАЧЕСТВА СЖАТИЯ ПРИ IF-ПРЕОБРАЗОВАНИИ

10.2.1.1. КОДИРОВАНИЕ ВЫХОДА IF-ПРЕОБРАЗОВАНИЯ

Так как IF-преобразование порождает много нулей, следующих друг за другом, то для повышения качества сжатия имеет смысл применять кодирование длин серий RLE одним из способов, описанных в п. 10.1.1. Эмпирические исследования автора показали (см. главу 11), что хорошие результаты достигаются, если длина серии n кодируется

выделенным символом α_n при небольших $n \leq n_{min} = 32$), а при больших значениях $n > n_{min}$ используется групповое кодирование с использованием выделенных символов $\beta_{L(n')}$, где $n' = n - n_{min}$ и $L(x) = \lfloor \log_2 x \rfloor$, т. е. сначала передается символ $\beta_{L(n')}$, после чего $L(n')$ младших значащих битов n'.

Смещения, отличные от нуля, кодируются аналогичным образом: если смещение i мало (например, $i \leq i_{min} = 64$), оно кодируется выделенным символом α'_i, иначе используется групповое кодирование с использованием выделенных символов $\beta'_{L(i-i_{min})}$.

Применение вышеописанных методов кодирования позволяет использовать для кодирования выхода IF-преобразования алфавиты очень малой мощности ($|\Sigma'| \sim 128$), уменьшая тем самым время кодирования (пропорциональное $\log_2 |\Sigma'|$) при использовании адаптивного арифметического кода или объем передаваемой декодировщику информации о принятом способе кодирования (в не более $|\Sigma'| \log_2 |\Sigma'|$ битов) при использовании статического кода Хаффмана.

10.2.1.2. ПОСТРОЕНИЕ МОДЕЛЕЙ ВЫСОКИХ ПОРЯДКОВ

Стандартные методы построения контекстно-зависимых моделей высоких порядков (см. п. 2.7) не годятся для кодирования выхода BWT даже после применения IF: объем требуемой памяти и время работы таких алгоритмов существенно превосходят аналогичные показатели для BWT, так что применение методов статистического моделирования делает бессмысленным проведение преобразования Барроуза — Уилера (лучше их сразу же применить к исходным данным). Кроме того, начиная с очень небольших значений порядка модели (1–2), дальнейшее увеличение порядка приводит к ухудшению качества сжатия, т. к. многие последовательности смещений встречаются лишь единожды ввиду большого диапазона возможных значений смещений.

С другой стороны, прямолинейный способ построения модели k-го порядка применением $|\Sigma'|^k$ независимых моделей нулевого порядка также не подходит, т. к. объем требуемой памяти (и стоимость передачи длин кодовых слов при использовании статического кода Хаффмана) экспоненциально возрастает с ростом k и уже при небольших $k \geq 2$ становится чрезмерно большим; при адаптивном кодировании многие модели нулевого порядка используются очень редко, поэтому часто применяется равномерное кодирование и качество сжатия с ростом порядка модели ухудшается.

Эмпирические исследования показали, что обычно вероятность $p(i)$ появления смещения $i \geq 1$ очень быстро убывает с ростом i почти по закону Ципфа, так что более 95% всех смещений не превышает 64 (см. табл. 10.1, где приводятся значения $p(i)$ для файлов из Calgary Data Compression Corpus; хорошо заметно, что $p(i) \approx \frac{p(1)}{i}$ как для текстовых, так и для двоичных файлов). Используя этот факт, модель первого порядка можно организовать следующим образом.

Поставим каждому смещению в соответствие некоторое *состояние*, определяемое следующим образом. Последовательности нулей длины 1 или 2 ставится в соответствие состояние $state = 0$, длины 3 и более — $state = 1$. Если текущее смещение i отлично от нуля, то $state = \lfloor \log_2(i) \rfloor$, если $i < 2^s$, где s — небольшая целая константа, а при $i \geq 2^s$ состояние, соответствующее смещению i, принимается равным $(s-1)$, так что общее количество различных состояний равно s (автор использовал $s = 9$).

Вместо того, чтобы использовать в качестве контекста сами смещения, будем использовать состояния, соответствующие этим смещениям. Таким образом, модель k-го порядка требует $s^k |\Sigma'|$ слов памяти, где Σ' — алфавит, содержащий все символы α_n, $\beta_{L(n')}$, α'_i и $\beta'_{L(i-i_{min})}$ (т. е. те символы, для которых будет использоваться энтропийное

Файл	1	2	3	4	5	6	7	8	9	≥ 10
book1	71684	41892	27914	20117	15217	12227	9673	7889	6752	58047 (16%)
book2	38554	21839	15152	10874	8500	6463	5597	4236	3719	40869 (18%)
news	18981	11322	7738	5904	4474	3836	3093	2603	2239	34897 (21%)
pic	15444	7108	4859	3456	2662	1991	1769	1424	1254	14062 (17%)
obj2	10241	5206	3555	2682	1941	1647	1366	1298	986	20972 (21%)
trans	2006	1195	674	518	425	327	224	262	142	4150 (17%)

Таблица 10.1. Зависимость частоты появления смещения p(i) от i

кодирование).

К сожалению, уже при $k = 2$ и $s \sim 10$ объем требуемой памяти и, что более важно, стоимость передачи длин кодовых слов при использовании статического кода Хаффмана становятся слишком большими, а при $k \geq 3$ применение как адаптивных, так и статических методов кодирования становится невозможным.

Тем не менее, рассмотрим процесс контекстно-зависимого кодирования более внимательно. Разместим модели нулевого порядка (т. е. бернуллиевские модели, для которых и производится энтропийное кодирование) в k-мерной матрице размерности s по каждому измерению, так что каждая бернуллиевская модель однозначно определяется последовательностью предыдущих состояний.

Применив сначала преобразование Барроуза — Уилера, а затем IF, соберем статистику использования кодируемых символов для каждой бернуллиевской модели, подсчитав количество использований каждого символа в каждом контексте.

Заметим, что ввиду достаточно большого количества состояний модели k-го порядка многие бернуллиевские модели будут или идентичными (в смысле вероятностей появления символов), или очень похожими.

Предлагается разбить все множество s^k простых бернуллиевских моделей на *множество классов эквивалентности* (желательно, существенно меньшей s^k мощности) и использовать одну и ту же бернуллиевскую модель для каждого класса эквивалентности.

Разбиение же на классы эквивалентности можно произвести следующим образом. Допустим, что i-й символ алфавита Σ' встретился u_i раз в бернуллиевской модели U и v_i раз — в модели V. Стоимость кодирования последовательности символов, встретившихся $w_1, \ldots, w_{|\Sigma'|}$ раз, равна

$$C(\overline{w}) = \sum_{w_i \geq 1} w_i \log_2 \frac{\sum w_j}{w_i} = \tag{10.3}$$

$$= \left(\sum w_j\right) \log_2 \left(\sum w_j\right) - \sum_{w_i \geq 1} w_i \log_2 w_i. \tag{10.4}$$

Известно [163], что $C(\overline{u}) + C(\overline{v}) \leq C(\overline{u} + \overline{v})$, причем равенство достигается тогда и только тогда, когда

$$\frac{u_i}{\sum u_j} = \frac{v_i}{\sum v_j}. \tag{10.5}$$

Таким образом, при использовании одной бернуллиевской модели вместо двух стоимость кодирования, строго говоря, не улучшится, однако за счет уменьшения объема передаваемой декодировщику информации о принятом способе кодирования при статическом кодирования (т. е. длин кодовых слов) или улучшении адаптивности

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	1	2	3	4	5
1	0	0	6	6	7	7	7	5	5
2	8	8	0	1	9	2	3	10	5
3	11	8	1	1	9	2	3	10	5
4	11	11	9	9	2	2	3	10	12
5	11	11	2	2	2	3	3	10	12
6	13	11	3	3	3	3	10	10	12
7	13	13	3	10	10	10	10	10	12
8	14	14	4	4	4	12	12	12	15

Таблица 10.2. Статическая модель первого порядка

бернуллиевской модели при адаптивном кодировании результирующее качество сжатия улучшится, если разность стоимостей кодирования

$$\Delta(\overline{u},\overline{v}) = C(\overline{u}) + C(\overline{v}) - \Big(C(\overline{u}) + C(\overline{v})\Big) \tag{10.6}$$

мала.

Итак, две бернуллиевские модели U и V можно считать эквивалентными, если $\Delta(\overline{u},\overline{v}) \le \varepsilon$, где ε — некоторое пороговое значение.

Собственно разбиение на классы эквивалентности можно производить последовательным отысканием пары моделей U и V с наименьшей разностью стоимостей кодирования $\Delta(\overline{u},\overline{v})$ и слиянием их в одну модель до тех пор, пока $\Delta(\overline{u},\overline{v}) \le \varepsilon$.

Замечание 10.2. Так как вычисление $\Delta(\overline{u},\overline{v})$ занимает довольно большое время и может измениться только при добавлении к U или V другой модели, то с целью ускорения перебора имеет смысл запоминать значения $\Delta(\overline{u},\overline{v})$ для всех пар различных моделей U и V, так что время разбиения $K = s^k$ бернуллиевских моделей на небольшое число классов эквивалентности становится пропорциональным не K^3, а K^2 (т. е. при увеличении порядка модели на единицу время вычислений увеличивается не в s^3, а в s^2 раза). ■

С использованием такого подхода построение классов эквивалентности для модели третьего порядка и $s = 9$ (т. е. для 9^3 бернуллиевских моделей) занимает менее одной минуты на современных ЭВМ, а для модели четвертого порядка — несколько часов (при использовании около 100 Мб памяти, что допустимо в единичных случаях).

Замечание 10.3. Скорее всего, полученное таким образом разбиение не будет оптимальным в смысле стоимости кодирования среди всех возможных разбиений равной мощности, однако автору неизвестен метод построения оптимального разбиения за разумное время. ■

Замечание 10.4. Задача построения оптимального разбиения очень похожа на задачу оптимальной установки в комнатах предметов разной формы — известной модификации $\mathcal{NP}$-полной задачи об укладке рюкзака. Автору не удалось доказать $\mathcal{NP}$-полноту задачи построения оптимального разбиения; это является лишь предположением. ■

Разбиение на классы эквивалентности позволяет снизить объем требуемой памяти с $s^k|\Sigma'|$ до $K|\Sigma'|$ слов, где K — количество различных классов эквивалентности. Эмпирические исследования автора показали, что наилучшее результирующее качество сжатия достигается при $K \sim 16 - 24$ (см. главу 11).

К сожалению, описанный выше метод нельзя применять на практике по одной простой причине: его временная сложность на много порядков превышает время, затрачиваемое на преобразования Барроуза — Уилера и IF.

Автор предлагает использовать не динамически, а *статически* построенное разбиение на классы эквивалентности. Для того чтобы построить такое разбиение, необходимо применить преобразование Барроуза — Уилера к достаточно большому (сотни мегабайтов) набору типичных данных (например, содержимому жесткого диска персональной ЭВМ), после чего применением полной модели высокого порядка и последовательным слиянием бернуллиевских моделей нулевого порядка, как описано выше, построить требуемое разбиение, которое и использовать в дальнейшем.

В табл. 10.2 приводится пример построенной таким образом модели первого порядка для $s = 9$ состояний и $K = 16$ бернуллиевских моделей; если текущее состояние равно s_{curr}, а предыдущее s_{prev}, то для кодирования используется модель $T[s_{prev}][s_{curr}]$. Хорошо заметно, что матрица T достаточно регулярна, причем часто бернуллиевская модель однозначно определяется $\max\{s_{curr}, s_{prev}\}$.

Эксперименты, проведенные автором, показали (см. главу 11), что качество сжатия при преобразовании Барроуза — Уилера в сочетании с последующим IF-преобразованием, кодируемым с применением статической модели третьего-четвертого порядка, практически не уступает качеству сжатия лучших методов статистического моделирования, опережая последние по скорости в 3–20 раз.

10.2.1.3. ИЗМЕНЕНИЕ ПОРЯДКА СИМВОЛОВ

В отличие от многих других методов качество сжатия IF-преобразования (равно как и преобразования Барроуза — Уилера) сильно зависит от *лексикографического* порядка символов, изменение которого может изменить качество сжатия. Так как отношение полного порядка на конечном алфавите мощности $|\Sigma|$ можно задать произвольным образом одним из $|\Sigma|!$ способов, то, вообще говоря, существует $|\Sigma|!$ различных IF-кодов одной и той же последовательности символов, статистика которых (и, соответственно, стоимость кодирования) может заметно отличаться.

К сожалению, автору не удалось найти способ задания лексикографического порядка, оптимального в смысле стоимости кодирования. Тем не менее можно привести несколько эвристик, улучшающих качество сжатия выхода IF-преобразования.

Заметим, что IF-преобразование эквивалентно последовательности $|\Sigma|$ интервальных кодирований: сначала строятся смещения для лексикографически наименьшего символа алфавита Σ, все вхождения которого затем удаляются из кодируемой последовательности символов; затем — последовательность смещений для следующего символа Σ, который удаляется из кодируемой последовательности, и т. д. Таким образом, имеет смысл удалять символы в порядке убывания частоты их использования, т. к. удаление символа уменьшает среднюю длину смещения (и, таким образом, стоимость кодирования) для следующих символов.

С другой стороны, последовательность смещений для последнего символа алфавита (лексикографически наиболее старшего) содержит только нули и может не передаваться вообще — достаточно передать количество появлений такого символа, поэтому иногда имеет смысл наиболее часто используемый символ считать последним символом алфавита (а остальные символы упорядочить по убыванию частоты их использования). Такое изменение порядка символов для текстовых файлов позволяет улучшить качество сжатия на 1–3%, однако для двоичных данных такое преобразование ухудшает качество кодирования на 1–2%.

Для того, чтобы отличать текстовые и двоичные данные, автором использовалась простая эвристика: у текстовых данных наиболее часто используемым символом всегда является пробел, в то время как у двоичных наиболее — символ с нулевым кодом.

Автором также изучалась другая эвристика определения лексикографического порядка символов на основе средней стоимости кодирования последовательности смещений при простом интервальном кодировании. Использование этой эвристики позволяет незначительно (на 0.1–0.5%) улучшить качество сжатия по сравнению с применением вышеописанных эвристик. Тем не менее автор считает применение такого метода определения порядка символов нецелесообразным, т. к. это почти вдвое замедляет IF-преобразование, а при использовании моделей больших порядков разница в стоимости кодирования быстро убывает с ростом порядка модели и становится практически несущественной (например, для модели третьего порядка разность в качестве сжатия не превышает 0.1%).

10.2.2. РЕАЛИЗАЦИЯ ПРЯМОГО IF-ПРЕОБРАЗОВАНИЯ

Прямое IF-преобразование ставит в соответствие каждому символу α последовательность *смещений* $i_1, i_2, \ldots$, так что i_k есть количество лексикографически больших символов, разделяющих k-е и $(k-1)$-е вхождения α в кодируемую строку (предполагается, что 0-е вхождение α предшествует кодируемой строке).

Авторами IF [34] было предложено реализовывать IF построением для каждого символа алфавита (в лексикографическом порядке) последовательности смещений между текущим и предыдущим вхождениями символов в кодируемую строку с последующим удалением всех вхождений символа; сложность такого алгоритма равна $O(N|\Sigma|)$ как в среднем, так и в наихудшем случае, что неприемлемо уже при достаточно небольшой мощности алфавита $|\Sigma| \geq 100$.

Вместо того, чтобы строить последовательность смещений для каждого символа в отдельности, лучше построить их все одновременно за один просмотр кодируемой строки, заменяя символ смещением к *следующему* его вхождению и позицией следующего вхождения; для последнего вхождения символа следующее вхождение отсутствует, поэтому для обозначения последнего вхождения символа его нужно заменить парой *(-1,-1)*. Кроме того, добавим к кодируемой строке (с начала) последовательность всех символов алфавита в обратном порядке, чтобы избежать появления проблемы нулевой вероятности (так что каждый символ имеет как минимум одно вхождение). При таком подходе кодируемая последовательность символов $L = (cba)aaccaabbaa$ будет преобразована в

	-3	-2	-1	0	1	2	3	4	5	6	7	8	9
L	c	b	a	a	a	c	c	a	a	b	b	a	a
$offset$	0	2	0	0	2	0	—	0	2	0	—	0	—
$next$	2	5	0	1	4	3	—	5	8	6	—	9	—

так что для получения результата IF-преобразования для i-го символа достаточно просмотреть односвязный список пар значений.

Заметим, что значение поля $offset(k)$ в k-й позиции $(k = 0, \ldots, (N-1))$ равно

$$offset(k) = \sum_{i=prev(k)}^{k} gtr(L_i, L_k), \tag{10.7}$$

где $prev(k)$ — индекс предыдущего вхождения символа L_k (для первого вхождения

положим $prev(k) = -1$), а

$$gtr(\alpha, \beta) = \begin{cases} 1, & \alpha > \beta, \\ 0, & \alpha \leq \beta. \end{cases} \tag{10.8}$$

Заметим, что

$$\begin{aligned} offset(k) &= \sum_{i=prev(k)}^{k} gtr(L_i, L_k) = & (10.9)\\ &= \sum_{i=0}^{k} gtr(L_i, L_k) - \sum_{i=0}^{prev(k)} gtr(L_i, L_k) = & (10.10)\\ &= G\big(k\big) - G\big(prev(k)\big) = & (10.11)\\ &= G(k) - G'(L_k), & (10.12) \end{aligned}$$

где

$$G\big(k\big) = \sum_{i=0}^{k} gtr(L_i, L_k); \tag{10.13}$$

другими словами, $G(k)$ есть общее количество больших L_k символов среди первых k символов строки L.

Имея эффективный алгоритм вычисления $G(k)$, кодируемую строку L легко преобразовать в требуемую последовательность пар следующим образом (предполагается наличие свободного места перед строкой L для $|\Sigma|$ пар ($offset$,$next$)). Сначала для всех $i = 0, \ldots, (|\Sigma| - 1))$ положим $prev(i) = -(i + 1)$ и $G'(i) = 0$. После этого за один просмотр строки L слева направо ($k = 0, \ldots, N - 1$) вычислим $G(k)$ и положим

$$\begin{aligned} offset\big(prev(L_k)\big) &= G(k) - G'(L_k), & (10.14)\\ next\big(prev(L_k)\big) &= k, & (10.15) \end{aligned}$$

после чего установим значение $prev(L_k)$ в k, а значение $G'(L_k)$ — в $G(k)$. По окончании просмотра строки для всех $i = 0, \ldots, |\Sigma| - 1$ установим значения $offset\big(prev(L_k)\big)$ и $next\big(prev(S_k)\big)$ в (-1) для обозначения конца списка, соответствующего i-му символу (этот список начинается в позиции $-(i + 1)$).

Для того чтобы вычислить значение $G(k)$, создадим идеально сбалансированное дерево с $|\Sigma|$ листьями, помеченными символами алфавита Σ слева направо, так что более левым листьям соответствуют лексикографически меньшие символы. Кроме того, с каждой вершиной дерева свяжем счетчики, изначально установленные в 0, в которых будет храниться общее количество просмотренных символов, помечающих листья данного поддерева. Для вычисления $G(k)$ необходимо пройти по пути от листа, помеченного символом L_k, и подсчитать сумму счетчиков *правых* братских вершин пути (т. е. если текущая вершина пути — левая, то сумму счетчиков необходимо увеличить на значение счетчика правой вершины, братской к текущей), одновременно увеличивая на единицу значения счетчиков проходимых вершин на пути к корню дерева. Таким образом, значение $G(k)$ может быть вычислено за $O(\log_2 |\Sigma|)$ операций с использованием $O(N)$ слов дополнительной памяти.

Замечание 10.5. Ввиду того, что кодируемая последовательность символов L, скорее всего, содержит большое количество повторений символов, то имеет смысл обрабатывать сразу же последовательность повторений одного символа. Вышеописанный алгоритм и соответствующие структуры данных нетрудно преобразовать с тем, чтобы последовательность k повторений одного символа обрабатывалась за $O(\log_2 |\Sigma|)$ опера-

ций (увеличивая счетчики вершин дерева не на единицу, а на k, и записывая не только значения полей $offset$ и $next$, но и количество последующих повторений смещения 0, равное $(k-1)$). ■

Замечание 10.6. При использовании статического кода Хаффмана можно (и нужно) совмещать построение последовательности смещений и сбор статистики, так что частоты использования символов кодируемого алфавита известны к концу IF-преобразования. ■

Итак, общая сложность IF-преобразования составляет

$$O(N \log_2 |\Sigma|), \tag{10.16}$$

что существенно лучше временной оценки $O(N|\Sigma|)$ прямолинейного алгоритма вычисления IF-преобразования; более того, несмотря на кажущуюся большую сложность, на практике прямое IF-преобразование *быстрее* метода стопки книг.

10.2.3. РЕАЛИЗАЦИЯ ОБРАТНОГО IF-ПРЕОБРАЗОВАНИЯ

Эффективная реализация обратного IF-преобразования — существенно более сложная задача, чем реализация прямого. Сразу же отметим, что предложенный авторами IF-преобразования прямолинейный метод восстановления закодированной строки и в наихудшем и в среднем случае требует $O(N|\Sigma|)$ операций, что чересчур много с практической точки зрения и гораздо хуже временной оценки $O(N \log_2 |\Sigma|)$ операций для прямого и обратного MTF-преобразований (метод стопки книг), поэтому, чтобы использовать IF-преобразование на практике, необходимо построить более эффективный алгоритм обратного IF-преобразования.

Как и прямое IF-преобразование, восстановление исходной строки лучше производить последовательным восстановлением очередного символа. Заметим, что для IF-преобразования текущее значение смещения для символа α есть не что иное, как количество символов, лексикографически больших α, которые должны появиться между предыдущим и следующим появлениями α. Исходя из этого можно построить алгоритм декодирования следующим (рекурсивным) образом.

10.2.3.1. АЛГОРИТМ IF-1

Допустим, что имеется запрос на печать k символов, лексикографически больших или равных j-го (изначально $k = N$ и $j = 0$).

Если n_j — текущее значение смещения j-го символа — равно нулю, то необходимо выдать j-й символ, заменить n_j следующим смещением, соответствующим j-му символу, и уменьшить k на единицу.

Если значение k стало равным нулю, то запрос исполнен.

Если же k не равно нулю, то необходимо снова сравнить n_j с нулем и при равенстве поступить так, как описано выше. Если же $n_j \neq 0$, то необходимо сравнить n_j и k.

Если $n_j \geq k$, то n_j нужно уменьшить на k и передать запрос на выдачу k символов, больших или равных $(j+1)$-го, по исполнении которого считать исполненным и текущий запрос.

При $n_j < k$ необходимо (рекурсивно) отправить запрос на печать n_j символов, лексикографически больших или равных $(j+1)$-му символу, и после его выполнения напечатать j-й символ, уменьшить k на (n_j+1) (т. к. было напечатано n_j символов, больших j-го, и сам j-й символ) и заменить n_j следующим смещением для j-го символа. Если k стало равным нулю, то запрос исполнен, иначе выполнение запроса продолжается.

Замечание 10.7. Если напечатанный j-й символ — последний j-й символ в строке, то можно считать n_j равным ∞, что гарантирует, что j-ый символ никогда не будет напечатан вновь.

Другое, несколько более эффективное решение заключается в том, что для каждого *активного* символа (т. е. такого, который обязательно еще появится в кодируемой строке) хранятся номера следующего и предыдущего активных символов (перед первым символом алфавита вводится фиктивный активный символ; т. к. последовательность смещения для последнего символа алфавита содержит только нули, то можно считать, что последний символ всегда активный). Другими словами, активные символы алфавита провязаны в двусвязный список, первый и последний элементы которого никогда не удаляются, так что удаление промежуточного элемента списка не требует никаких предосторожностей и сводится к двум присваиваниям. При использовании списка активных символов запросы посылаются не к $(j+1)$-му символу, а к *следующему активному*. ■

Нетрудно заметить, что производительность алгоритма в наихудшем случае, к сожалению, равна $O(N|\Sigma|)$ (если кодируемая последовательность содержит очень много чередований первого и последнего символов алфавита), однако с учетом особенностей выхода преобразования Барроуза — Уилера вероятность получить такую последовательность чрезвычайно мала. Обычно *средняя* производительность этого алгоритма очень высока и практически совпадает со скоростью прямого IF-преобразования ($O(N\log_2|\Sigma|)$ операций), что [при кодировании выхода BWT] объясняется наличием большого количества повторений одного и того же символа в кодируемой последовательности, которые обрабатываются алгоритмом IF-1 очень эффективно.

10.2.3.2. АЛГОРИТМ IF-2

Несмотря на хорошую среднюю производительность вышеописанного алгоритма, автору хотелось получить алгоритм с временной оценкой $O(N\log_2|\Sigma|)$. Это важно для применения IF-преобразования для кодирования выхода *частичного преобразования Барроуза — Уилера*, разработанного автором (см. п. 9), которое требует и в среднем и в наихудшем случае $O(F)$ операций на символ, где F — небольшая константа (4–8), поэтому ограничение времени работы IF-преобразования $O(\log|\Sigma|)$ операциями допустит построение высокоэффективных алгоритмов сжатия, время работы которых и в среднем, и в наихудшем случае пропорционально длине кодируемой последовательности и практически не зависит от собственно кодируемых данных, что является существенным требованием в ряде приложений.

Заметим, что текущий символ, который должен быть напечатан, может быть определен следующим образом. Для каждого символа определим g_j — количество символов, больших j-го, напечатанных со времени последнего появления j-го символа (или начала строки, если появлений j-го символа еще не было). Рассмотрим множество значений $(n_j - g_j) \geq 0$, которое должно содержать не менее одного нуля (например, для последнего символа алфавита n_j и g_j всегда равны нулю). Номер j символа, который должен быть напечатан следующим, определяется наименьшим номером j таким, что $(n_j - g_j) = 0$. Вышеописанные условия равносильны тому, что номер символа равен j такому, что $f(j) \to \min$, где

$$f_j = \big((n_j - g_j)|\Sigma| + j\big). \tag{10.17}$$

К сожалению, после того, как напечатан j-й символ, соответствующее ему значение n_j изменяется (практически произвольным образом), равно как и значения g_i для всех

$i < j$ (которые увеличиваются на единицу), поэтому задача отыскания минимума $f(j)$ по j за $O(\log_2 |\Sigma|)$ операций представляется весьма нетривиальной.

Для того чтобы определить значения g_j для произвольного j в текущей позиции, можно воспользоваться структурой данных, описанной в п. 10.2.2 для прямого IF-преобразования. Однако в дальнейшем будет удобнее пользоваться немного модифицированным ее вариантом.

Возьмем произвольное идеально сбалансированное дерево (все листья которого расположены на двух смежных уровнях, причем каждая промежуточная вершина имеет ровно двух сыновей), имеющее $|\Sigma|$ листьев, помеченных символами Σ так, что более левые листья помечены лексикографически меньшими символами. В каждой вершине дерева поместим счетчики количества напечатанных лексикографически больших символов, содержащихся в *правом* братском к данной вершине поддереве (если таковое имеется; если же нет, то значение счетчика всегда будет равно нулю). Таким образом, для того, чтобы подсчитать общее количество напечатанных символов, больших данного, достаточно просуммировать значения счетчиков, помечающих вершины дерева на пути от соответствующего листа до корня. После того, как напечатан некоторый символ, необходимо увеличить на единицу значения счетчиков *левых* братских вершин дерева на пути от соответствующего листа до корня дерева.

Заметим, что

$$g_j = G_j - G'_j, \tag{10.18}$$

где G_j — общее количество появлений символов, больших j-го, а G'_j — значение G_j при предыдущем появлении j-го символа. Таким образом, задача сводится к нахождению минимума по j функции

$$f(j) = ((n_j + G'_j) - G_j)|\Sigma| + j; \tag{10.19}$$

изначально $G_j = G'_j = 0$ для всех $j = 0, \ldots, |\Sigma| - 1$.

Обходом дерева снизу вверх в ширину пометим каждую промежуточную вершину v дерева минимальным в данном поддереве значением $f(j)$, которое равно

$$w(v) = \min_j\{f(j)\} = \min\{w(L_v) - G(L_v)|\Sigma|, w(R_v)\}, \tag{10.20}$$

где $w(L_v)$ и $w(R_v)$ — пометки левого и правого сыновей вершины v, а $G(L_v)$ — значения счетчика больших символов (по построению алгоритма, $G(R_v)$ всегда равно нулю, поэтому вычитать из $w(R_v)$ значение $G(R_v) = 0$ нет необходимости).

Таким образом, пометка $w(r)$ корня дерева r и есть минимальное значение $f(j)$. По определению функции f,

$$j = \min_{i=0}^{|\Sigma|-1}\{f(i)\} \bmod |\Sigma| = w(r) \bmod |\Sigma|. \tag{10.21}$$

Выдадим j-й символ и заменим n_j новым значением соответствующего j-му символу смещения (если таковое отсутствует, то n_j будем считать равным ∞). Вычислим новое значение пометки $w(v_j)$ соответствующего j-му символу листа дерева v_j

$$w(v_j) = (n_j + G_j)|\Sigma| + j, \tag{10.22}$$

где n_j — новое значение смещения для j-го символа, а G_j — общее количество напечатанных символов, больших j-го (равное сумме счетчиков всех вершин на пути от соответствующего листа дерева до корня).

Заметим, что выдача j-го символа изменила также значения счетчиков больших символов левых братских вершин пути (и только этих вершин), поэтому для нахождения нового минимума значения функции f достаточно пройти по дереву от листа, помеченного j-м символов, до корня и для каждой промежуточной вершины v пересчитать

значение пометки $w(v)$ согласно

$$w(v) = \min\{w(L_v) - G(L_v)|\Sigma|, w(R_v)\}; \tag{10.23}$$

после этого определяется номер следующего выдаваемого символа $j = w(r) \bmod |\Sigma|$, вычисляется текущее значение G_j и новое значение n_j, новое значение пометки соответствующей j-му символу вершины $w(v_j)$ и перевычисляются значения пометок $w(v)$ промежуточных вершин дерева на пути от v_j к корню дерева r и т. д.

Таким образом, нахождение текущего значения $\min\{f(j)\}$ требует двух проходов по дереву — для вычисления G_j и перевычисления пометок промежуточных вершин, поэтому обратное IF-преобразование требует не более $O(\log_2 |\Sigma|)$ операций на символ с использованием $O(|\Sigma|)$ слов дополнительной памяти.

Замечание 10.8. Если декодируется серия повторений одного и того же символа, то перевычислять значение G_j на каждом шаге не требуется, т. к. оно не изменяется. ■

Замечание 10.9. Если символы отсортированы по частоте использования, т. е. вероятности появления символов $p(i)$ не возрастают с ростом i, то вместо идеально сбалансированного дерева лучше использовать дерево Хаффмана, что позволит снизить количество операций, требуемых для декодирования одного символа, с $O(\log_2 |\Sigma|)$ до $O(\mathcal{E})$, где $\mathcal{E}$ — энтропия (исходного) сообщения,

$$\mathcal{E} = \sum_{i=0}^{|\Sigma|-1} p(i) \log_2 \frac{1}{p(i)}. \tag{10.24}$$

Так как обычно для кодирования используется расширенный 256-символьный код ASCII, то, например, для текстов на естественных языках, энтропия которых составляет 4.5–5.5 битов на символ, использование дерева Хаффмана может в 1.5–2 раза ускорить декодирование. ■

Вышеописанный алгоритм обратного IF-преобразования IF-2 при использовании идеально сбалансированных деревьев обычно в 1.5–1.8 раза медленнее алгоритма IF-1, описанного в п. 10.2.3.1, а при использовании дерева Хаффмана — в 1.2–1.3 раза. Это объясняется наличием [в выходе преобразования Барроуза — Уилера] большого количества повторений одного и того же символа, более эффективно обрабатываемых алгоритмом IF-1.

10.3. ВЫВОДЫ

Разработанные автором методы реализации алгоритмов интервального кодирования — метода стопки книг и Inverted Frequency Coding — позволяют улучшить время работы этих алгоритмов (по сравнению с известными реализациями) с $O(|\Sigma|)$ до $O(\log_2 |\Sigma|)$ операций при таком же объеме требуемой оперативной памяти. Получаемое увеличение скорости работы метода стопки книг (в среднем в 1.5–3 раза) и IF (в 10–20 раз) при $|\Sigma| = 256$ (для расширенного ASCII) существенно расширяет область применения этих алгоритмов. Более того, до сих пор Inverted Frequency Coding не использовалось на практике именно из-за чрезмерно низкой скорости работы.

Предложенный автором метод построения моделей высокого порядка в сочетании с IF-преобразованием позволил заметно (на 10–20%) улучшить качество сжатия сортировкой блоков и достичь качества сжатия, лишь на 1–2% уступающего достижимому при использовании наилучших известных методов сжатия статистическим моделированием при таком же и часто меньшем объеме требуемой памяти. Скорость сжатия

описанных методов выше в 3–20 раз и совпадает со скоростью сжатия алгоритмов семейства LZ77 (при заметно лучшем качестве сжатия), которые до сих пор считались наилучшими по соотношению производительности и качества сжатия.

Часть V

Результаты исследований

Глава 11

СРАВНЕНИЕ МЕТОДОВ СЖАТИЯ

11.1. АРХИВАТОРЫ

Для сравнения были выбраны наилучшие реализации (1995–1997 гг.) самых эффективных по качеству и/или скорости алгоритмов сжатия.

Исполняемые программы вышеперечисленных архиваторов находятся на многих ftp-серверах, например, на **ftp.elf.skuba.sk** и др.

11.1.1. АЛГОРИТМЫ СЛОВАРНОГО СЖАТИЯ

- **PKZIP 2.04G** (автор — P. Katz) реализует LZ77 со скользящим окном в 32 Кб.
- **RAR 2.02** (автор — Е. Рошал) реализует LZ77 со скользящим окном в 64 Кб.

Скорость декодирования этих архиваторов — 1 Мб/с, а объем требуемой памяти — 300 и 500 Кб соответственно.

Другие архиваторы семейства LZ77 (LHA, LHARC, ZOO, ARJ, UFA, LIMIT, HPACK, UC2 и т. д.) заметно уступают по скорости PKZIP и RAR при таком же и часто худшем качестве сжатия и большем объеме требуемой памяти (несмотря на такую же или меньшую ширину окна):

- **LHA 2.13** (автор — H. Yoshizaki) реализует LZ77 со скользящим окном в 8 Кб;
- **LIMIT 1.02** (автор — J. Lim) реализует вариант LZWW алгоритма LZ77 со скользящим окном в 32 Кб;
- **ARJ 2.41** (автор — R. Jung) реализует LZ77 со скользящим окном в 32 Кб.

11.1.2. АЛГОРИТМЫ СТАТИСТИЧЕСКОГО МОДЕЛИРОВАНИЯ

- **DMC1** (автор — А. Кадач) реализует DMC согласно [71] и примерно в 1.5–2 раза быстрее оригинальной версии [68] при прочих равных параметрах.
- **PPMZ 8.3** (автор — C. Bloom) реализует PPM/PPM* 3–5 порядка.
- **HA 0.999C** (автор — H. Hirvola) реализует PPM 4 порядка;
- **X1 0.94L** (автор — S. Valentini) реализует PPM/PPM* 5–6 порядка (опции *am3* и *am4*).
- **RKIVE 1.4** (автор — M. Taylor) реализует PPM 6–8 порядка.
- **ACB 1.23C** (автор — Г. Буяновский) реализует алгоритм ACB.

Скорость декодирования этих методов практически идентична скорости кодирования, а объем требуемой памяти составляет 16–24 Мб.

11.1.3. АЛГОРИТМЫ СЖАТИЯ СОРТИРОВКОЙ БЛОКОВ

- **BRED** и **BRED3** (автор — D. Wheeler) реализуют BWT согласно [55] и [185] соответственно;
- **X1 0.94L** (автор — S. Valentini) реализует BWT согласно [55] (опция *am7*).

Размер блока составляет 1 Мб, объем требуемой памяти — 9–10 Мб (кроме BRED3 — 6 Мб), скорость декодирования — 400–450 Кб/с.

11.1.4. АВТОРСКИЕ АЛГОРИТМЫ (PRS)

Алгоритмы, описанные в данной работе, представлены программой **PRS**.

11.1.4.1. АЛГОРИТМЫ СЕМЕЙСТВА LZ77

Во всех случаях для поиска подстрок использовался разработанный автором вариант LZ77 со скользящим окном — LZ77 с прыгающим окном, прямым перебором строк хэш-списка и быстрым отсечением неудлиняющих подстрок по двум последним символам (см. главу 6); методы сжатия полученного LZ77-кода описаны в главе 7.

При опциях L0–L3 использовалось прямолинейное кодирование (см. п. 5.2.7) при максимальной глубине перебора хэш-списка, равной 20, 30, 50 и 90 соответственно.

При опциях L4–L6 использовался алгоритм оптимального кодирования для $r = 2$ (см. пп. 5.2.4 и 5.2.6) при максимальной глубине перебора хэш-списка, равной 50, 80 и 130 соответственно.

При опции L7 использовался алгоритм оптимального кодирования для $r = 2.5$ (см. п. 5.3) при максимальной глубине перебора хэш-списка, равной 100.

Ширина окна составляла 48–64 Кб при опциях L0–L4 и 56–64 Кб при L5–L7.

Скорость декодирования (включая время ввода-вывода, вычисления циклических контрольных сумм и т. д.) авторской реализации алгоритмов семейства LZ77 — 2.5 Мб/с.

11.1.4.2. АЛГОРИТМЫ СЖАТИЯ СОРТИРОВКОЙ БЛОКОВ

Автором были реализованы алгоритмы сжатия полной и частичной сортировкой блоков (см. главы 8 и 9); для кодирования выхода алгоритма использовались методы, описанные в главе 10.

Ширина блока составляла в 1 и 2 Мб (первый символ соответствующего значения опции), при этом требовалось 9 и 18 Мб оперативной памяти соответственно. Полному преобразованию Барроуза — Уилера соответствует опция **F**ull, частичному — **P**artial; следующая за буквой P цифра означает глубину сортировки (по первым четырем или восьми символам); таким образом, *prs 2P8* означает PBWT с сортировкой по первым восьми символам при размере блока в 2 Мб.

Скорость декодирования авторской версии BWT — 350 Кб/с — несколько ниже, чем у метода Уилера, основанного на применении метода стопки книг; это объясняется более сложной схемой кодирования (см. п. 10.2), качество сжатия которой на 10–20% лучше.

При сжатии частичной сортировкой блоков время декодирования в 1.2–1.3 раза превышает время кодирования при сортировке по первым четырем символам, и в 1.3–1.4 раза — по восьми символам.

Название	Содержание
book1	Hardy Far from the madding crowd, повесть
book2	Witten Principles of computer speech, монография
bible.txt	Библия, новый завет (на англ. языке)
world.txt	Краткая географическая энциклопедия
amber110.txt	Зилазни Девять принцев Амбера, романы
mobydick.txt	Melville Moby Dick, роман
excel.exe	Mircrosoft Excel 7.0 for Windows 95/NT
winword.exe	Mircrosoft Word 7.0 for Windows 95/NT

Таблица 11.1. Файлы с тестовыми данными

11.2. ТЕСТОВЫЕ ДАННЫЕ

Были использованы данные[1] из стандартных наборов для сравнения систем сжатия —- Calgary и Canterbury Data Compression Corpus(es); кроме того, некоторые файлы добавлены автором:

- большие текстовые файлы (bible.txt и world.txt) обладают нетипично высокой избыточностью, поэтому были добавлены amber110.txt и mobydick.txt, которые написаны богатым языком и в гораздо большей степени отвечают критерию типичности;
- оба стандартных набора данных не содержат достаточно больших образцов исполняемых файлов, которые составляют значительную часть хранимой на большинстве персональных ЭВМ данных, поэтому были добавлены файлы excel.exe и winword.exe.

Замечание 11.1. Для файлов длины менее 1 Мб результаты сжатия сортировкой блоков шириной 2 Мб не приводятся, т. к. и время, и качество сжатия не отличаются от соответствующих параметров PRS с блоком в 1 Мб. ■

11.3. МЕТОДИКА СРАВНЕНИЯ

Для сравнения использовалась IBM PC с процессором Pentium с тактовой частотой 100 МГц и 32 Мб оперативной памяти под операционной системой Windows 95.

Каждая программа запускалась 11 раз, наибольшее время исполнения не учитывалось, а в качестве результата принималось усредненное общее время работы по 10 запускам; т. к. использовался таймер в 1000 Гц, точность полученных значений достаточно высока.

Следует отметить, что время исполнения включает время загрузки кода программы и чтения/записи данных, однако поскольку данные и программы быстро попадают в кэш операционной системы, время загрузки кода и ввода-вывода очень мало, не превышая нескольких сотых долей секунды, и не оказывает заметного влияния на результат.

Время вычисления циклических контрольных сумм CRC (около 0.2 с/Мб) также включено в общее время. Ряд программ не вычисляют CRC вообще (DMC1, BRED, BRED3, PPMZ), остальные вычисляют только CRC сжатых данных, тогда как библиотека PRS подсчитывает CRC как сжатых, так и исходных данных, поэтому учет времени

[1] Необходимо сказать, что автором не делалось никаких попыток — явных или неявных — улучшить качество сжатия на файлах, использованных для тестирования; аналогичные результаты получаются и на других данных.

вычисления контрольных сумм в первую очередь негативно сказывается на авторских программах, т. к. их реальное быстродействие на 5–10% выше.

Наконец, необходимо учесть, что в подавляющем большинстве случаев (кроме DMC1, BRED, BRED3, PPMZ) архиваторы реализованы на языке ассемблера, тогда как библиотека PRS реализована целиком на языке Си для обеспечения переносимости; известные особенности архитектуры процессоров Intel приводят к тому, что программы на Си на 10–20% медленнее эквивалентных реализаций на ассемблере.

11.4. ОФОРМЛЕНИЕ РЕЗУЛЬТАТОВ

Результаты сжатия были отсортированы по качеству сжатия и разбиты на группы с примерно одинаковым качеством сжатия, а внутри каждой группы отсортированы по времени сжатия. Наилучший по качеству результат внутри каждой группы выделен жирным шрифтом.

book1 (768771 байтов)			**book2 (610856 байтов)**		
Архи-ватор и опции	Время, *с*	Длина сжатых данных, *байтов*	Архи-ватор и опции	Время, *с*	Длина сжатых данных, *байтов*
prs 1P8	4.648	219905	**prs** 1P8	3.480	152616
prs 1F	5.753	219708	**prs** 1F	4.450	152465
x1 am4l9	18.265	219688	x1 am4l9	13.815	152539
x1 am3l9	19.250	219824	x1 am3l9	14.500	152832
rkive -mt	29.880	219672	rkive -mt	19.880	149881
rkive -mtx	48.610	219913	rkive -mtx	32.300	**149682**
ppmz -c4	83.710	**218775**	acb b	51.250	150591
prs 1P4	2.896	227652	ppmz -c4	59.700	150138
acb b	86.830	226609	acb u	82.280	149743
acb u	142.150	**224525**	**prs** 1P4	2.181	**158853**
dmc1 c	9.650	235619	ha a2	13.210	163566
ppmz -c5	17.215	**234400**	rkive -mb	15.190	164570
ha a2	17.245	235764	acb B	24.500	159133
acb B	40.200	241260	bred3 -M1	3.827	**167255**
bred3 -M1	5.355	**249971**	x1 am7	5.313	167422
x1 am7	7.413	250121	bred -M1	5.492	167387
bred -M1	7.633	250086	dmc1 c	7.507	167244
rkive -mb	23.780	252317	ppmz -c5	12.990	171144
prs L4	3.947	304737	**prs** L4	2.603	200965
prs L5	5.013	301107	**prs** L5	3.310	198574
prs L6	5.945	299393	**prs** L6	3.928	197589
prs L7	6.960	**296524**	**prs** L7	4.482	**195879**
rar -m3	7.377	305906	rar -m3	4.712	199440
rar -m4	13.375	301445	rar -m4	7.857	196853
rar -m5	14.115	301298	rar -m5	8.183	196803
prs L2	2.794	314942	**prs** L2	1.884	208465
prs L3	3.327	311627	**prs** L3	2.270	206029
pkzip -en	4.020	316107	pkzip -en	2.547	209169
rar -m2	4.132	319837	rar -m2	2.808	208902
arj -m2	5.315	322381	pkzip -ex	3.990	206621
limit -ms	5.355	320090	arj -m2	3.397	212659
pkzip -ex	6.015	312598	limit -ms	3.497	210064
arj -m1	7.890	319166	arj -m1	4.756	210431
limit -m1	10.600	311184	limit -m1	6.523	204707
limit -mx	11.425	**310869**	limit -mx	6.793	**204616**
pkzip -es	0.923	372380	pkzip -es	0.659	255206
prs L0	1.647	360294	pkzip -ef	1.187	226625
pkzip -ef	1.768	341585	**prs** L0	1.169	246829
prs L1	2.011	**330325**	**prs** L1	1.407	**220968**
arj -m3	2.615	341071	arj -m3	1.796	226982
rar -m1	2.849	337069	rar -m1	2.066	221510
lha a	8.970	339107	lha a	6.753	228475

bible.txt (4047392 байтов)			**world192.txt (2473400 байтов)**		
Архи-ватор и опции	Время, *с*	Длина сжатых данных, *байтов*	Архи-ватор и опции	Время, *с*	Длина сжатых данных, *байтов*
prs 2P8	22.198	780312	acb b	178.670	426871
prs 2F	35.082	779502	acb u	276.060	**422301**
acb b	471.210	768207	**prs** 2F	22.802	**449435**
acb u	694.260	**757921**	rkive -mtx	92.930	455613
prs 1P8	20.700	809972	acb B	95.900	456483
prs 1F	32.250	807908	**prs** 2P8	14.048	469776
rkive -mt	112.980	794974	**prs** 1F	20.650	486608
rkive -mtx	192.740	**787493**	rkive -mt	59.160	**469415**
acb B	280.230	826840	ppmz -c4	309.780	471820
ppmz -c4	671.030	816005	**prs** 1P8	13.100	503555
prs 2P4	13.974	849914	x1 am7	26.250	494960
x1 am4l9	69.150	846628	rkive -mb	51.690	**492849**
x1 am3l9	73.600	**846138**	bred3 -M1	18.730	509980
prs 1P4	12.385	**872292**	bred -M1	22.950	509208
bred3 -M1	30.260	898319	x1 am3l9	56.350	514716
bred -M1	40.100	898581	x1 am4l9	61.680	**507735**
x1 am7	42.120	899514	**prs** 1P4	8.073	574337
dmc1 c	60.090	898174	**prs** 2P4	9.231	549953
ha a2	64.480	884287	dmc1 c	33.780	**540998**
rkive -mb	88.980	913206	ha a2	51.800	602180
prs L2	12.250	1205930	**prs** L4	8.130	683615
prs L3	15.410	1184700	**prs** L5	10.185	674255
prs L4	17.215	1161743	**prs** L6	11.730	671879
pkzip -en	17.660	1197802	**prs** L7	13.405	665467
rar -m2	17.690	1218946	rar -m3	15.000	672873
limit -ms	21.810	1228508	rar -m4	23.780	666117
prs L5	22.960	1142038	rar -m5	26.200	665641
arj -m2	24.820	1231886	ppmz -c5	62.670	**647297**
prs L6	29.720	1129884	**prs** L2	6.303	709167
pkzip -ex	30.590	1179372	**prs** L3	7.140	**702552**
rar -m3	32.130	1157528	pkzip -en	7.690	739775
prs L7	32.410	1120439	rar -m2	9.667	708270
arj -m1	43.940	1209412	arj -m2	10.520	756524
limit -m1	52.340	1177599	pkzip -ex	12.800	723473
limit -mx	68.870	1171782	limit -ms	12.440	735053
rar -m4	73.980	1126499	arj -m1	15.625	746090
ppmz -c5	83.980	**989642**	limit -m1	21.200	716929
rar -m5	86.450	1123800	limit -mx	24.170	715799
pkzip -es	3.652	1532952	pkzip -es	2.326	952380
prs L0	7.380	1459265	pkzip -ef	4.098	827837
pkzip -ef	8.037	1310234	**prs** L0	4.558	883037
prs L1	8.643	**1291564**	**prs** L1	5.123	**759478**
arj -m3	11.620	1315129	arj -m3	6.180	832900
rar -m1	12.410	1310512	rar -m1	7.450	769122
lha a	44.710	1311857	lha a	26.810	992394

amber110.txt (3494983 байтов)			**mobydick.txt (1245579 байтов)**		
Архи-ватор и опции	Время, *с*	Длина сжатых данных, *байтов*	Архи-ватор и опции	Время, *с*	Длина сжатых данных, *байтов*
prs 2P8	22.143	973067	**prs** 2P8	8.110	354281
prs 2F	28.846	973683	**prs** 2F	10.330	**353996**
x1 am4l9	87.770	961872	x1 am4l9	29.110	359415
x1 am3l9	90.410	986231	x1 am3l9	31.800	360742
rkive -mt	135.390	**959347**	rkive -mt	48.830	358517
rkive -mtx	225.080	959505	rkive -mtx	79.700	359007
acb b	499.060	976598	ppmz -c4	144.240	359172
acb u	770.720	965178	acb u	243.050	361026
prs 1P4	12.605	1054827	**prs** 2P4	5.316	371751
prs 2P4	14.396	1022397	**prs** 1P8	7.397	365873
prs 1P8	20.540	1012811	**prs** 1F	9.153	365603
prs 1F	25.820	1012666	acb b	149.400	**364632**
acb B	271.940	1046923	**prs** 1P4	4.560	**380969**
ppmz -c4	453.460	**991295**	ppmz -c5	28.730	385237
bred3 -M1	24.120	1133439	acb B	68.380	387804
bred -M1	34.330	1135130	x1 am7	13.235	403192
x1 am7	35.100	1119330	dmc1 c	19.085	393118
dmc1 c	50.480	1090089	ha a2	27.740	**392738**
ha a2	80.250	1097246	bred3 -M1	8.643	**411945**
ppmz -c5	94.850	**1075916**	bred -M1	12.415	412197
rkive -mb	127.810	1174649	rkive -mb	38.830	412901
prs L3	14.775	1405459	**prs** L5	8.293	488992
prs L4	17.575	1373615	**prs** L6	9.777	486686
prs L5	21.970	1355154	**prs** L7	11.535	**481913**
prs L6	25.430	1347727	rar -m3	12.495	497322
prs L7	30.320	**1334427**	rar -m4	23.400	490292
rar -m3	39.000	1398708	rar -m5	25.150	490005
rar -m4	69.150	1373682	**prs** L2	4.602	510927
rar -m5	72.070	1373337	**prs** L3	5.508	505949
prs L2	12.550	**1422301**	**prs** L4	6.510	**494963**
pkzip -en	18.180	1458085	rar -m2	6.810	518103
rar -m2	20.540	1468109	pkzip -en	6.940	512786
limit -ms	25.700	1469417	pkzip -ex	10.270	507773
arj -m2	26.800	1482178	arj -m1	13.375	517913
pkzip -ex	27.080	1438956	arj -m2	8.860	522640
arj -m1	38.510	1465364	limit -m1	17.905	506078
limit -m1	44.930	1425025	limit -ms	8.917	520115
limit -mx	45.530	1424809	limit -mx	19.825	505481
pkzip -es	4.184	1724514	pkzip -es	1.445	601319
pkzip -ef	6.937	1609273	pkzip -ef	2.973	550246
prs L0	7.617	1651173	**prs** L0	2.658	583135
prs L1	9.120	**1500353**	**prs** L1	3.217	**534729**
arj -m3	12.795	1571589	arj -m3	4.306	550495
rar -m1	13.375	1570174	rar -m1	4.580	544801
lha a	40.200	1573850	lha a	14.610	546411

excel.exe (4828160 байтов)			**winword.exe (3852288 байтов)**		
Архи-ватор и опции	Время, *с*	Длина сжатых данных, *байтов*	Архи-ватор и опции	Время, *с*	Длина сжатых данных, *байтов*
rkive -mb	462.310	**2437764**	rkive -mb	360.310	1959094
acb u	1124.050	2439000	acb b	570.290	1982975
prs 2P4	25.714	2538712	acb u	897.920	**1958932**
prs 2F	35.165	2523014	**prs** 2P4	20.549	2048219
prs 2P8	37.473	2523793	**prs** 2F	28.159	2039437
rkive -mt	537.440	2491445	**prs** 2P8	30.247	2039641
x1 am4l9	621.590	2507613	rkive -mt	425.240	2001135
rkive -mtx	672.290	2490952	x1 am3l9	449.680	2039593
acb b	714.140	**2468654**	x1 am4l9	500.040	2003405
prs 1P4	23.620	2585865	rkive -mtx	530.640	**2000786**
prs 1F	31.860	2575472	**prs** 1P4	17.880	2072305
prs 1P8	35.430	2576161	**prs** 1F	24.560	**2065084**
acb B	362.460	2575690	**prs** 1P8	27.580	2065281
x1 am3l9	546.230	**2549026**	acb B	288.030	2069021
rar -m3	41.960	2758212	ha a2	347.130	2144475
bred -M1	46.030	2721920	rar -m3	33.120	2226409
x1 am7	46.530	**2719323**	bred -M1	36.690	2189051
rar -m4	69.320	2753449	x1 am7	37.730	2185643
rar -m5	79.310	2752864	rar -m4	54.600	2222022
dmc1 c	82.280	2785515	rar -m5	62.890	2221746
ha a2	429.900	2684733	dmc1 c	63.490	**2182453**
prs L1	16.700	2889544	**prs** L1	13.485	2334480
prs L2	19.250	2860777	**prs** L2	15.515	2310663
prs L3	20.600	2856629	**prs** L3	16.830	2306806
rar -m1	21.640	2817456	rar -m1	17.160	2276405
prs L4	23.780	2828680	**prs** L4	19.445	2283646
pkzip -en	25.980	2900351	pkzip -en	20.210	2334149
arj -m2	26.910	2909221	arj -m2	21.530	2342433
limit -ms	27.250	2902052	limit -ms	21.970	2335850
rar -m2	28.070	**2795738**	rar -m2	22.130	**2257446**
prs L5	28.570	2820347	**prs** L5	23.290	2275649
prs L7	31.470	2818552	**prs** L6	26.750	2271946
prs L6	32.620	2817296	**prs** L7	25.710	2273745
pkzip -ex	39.650	2891639	pkzip -ex	30.980	2327258
arj -m1	40.320	2902799	arj -m1	32.300	2337231
limit -m1	42.400	2893329	limit -m1	34.170	2327468
limit -mx	49.650	2891846	limit -mx	40.860	2327086
pkzip -es	7.507	3155387	pkzip -es	5.933	2540327
pkzip -ef	11.920	2969341	pkzip -ef	9.540	2392621
prs L0	14.965	2985688	**prs** L0	12.085	2411807
arj -m3	15.820	2965856	arj -m3	12.635	2389276
lha a	46.630	**2958344**	lha a	37.020	**2385021**
bred3 -M1	186.800	3024004	bred3 -M1	149.230	2434006

11.5. ВЫВОДЫ

Предложенные автором методы сжатия и способы их реализации позволяют создание систем сжатия данных, наилучших по отношению качества и скорости.

Алгоритмы семейства LZ77 используют небольшой объем оперативной памяти, однако уступают в качестве алгоритмам сжатия сортировки блоков, которым нужно на порядок больше памяти.

Предложенные методы сжатия сортировкой блоков идеально подходят для применения на универсальных ЭВМ общего назначения (персональных компьютерах, рабочих станциях, сетевых узлах, серверах и т. д.).

Алгоритмы семейства LZ77 могут использоваться, если:

- существуют жесткие ограничениях на объем требуемой алгоритму сжатия памяти (например, во всевозможных встроенных системах);
- необходима очень высокая скорость декодирования (например, в файловых системах со сжатием);
- декодирование осуществляется заметно чаще кодирования, а объем сжатых данных не является определяющим фактором (например, при хранении данных на CD-ROM).

Замечание 11.2. Если на текстовых данных качество сжатия алгоритмов семейства LZ77 заметно — в 1.4–1.7 раза — хуже, чем у методов статистического моделирования и сжатия сортировкой блоков, то на двоичных данных разница гораздо менее заметна (1.15–1.25 раза). Это объясняется тем, что двоичные данные устроены гораздо проще текстовых, поэтому достичь достаточно высокого качества сжатия двоичных данных можно более простыми средствами. В частности, средняя длина найденных подстрок (и определяющих контекстов) для двоичных данных — 4–5 символов, тогда как для текстовых — 7–10, поэтому поведение двоичных данных менее предсказуемо, а качество их сжатия заметно хуже качества сжатия текстов.

Кроме того, структура исполняемых файлов очень нерегулярна (в отличие от текстов), т. к. они состоят из относительно коротких участков кода, данных, таблиц экспортируемых и импортируемых объектов, списков используемых ресурсов и т. д., перемежающихся друг с другом, поэтому исполняемые файлы обладают очень быстро меняющейся стстистикой, что также затрудняет сжатие. ■

11.5.1. АЛГОРИТМЫ СЕМЕЙСТВА LZ77

При равном качестве сжатия на текстовых данных авторская реализация быстрее других в 2–5 раз, причем разрыв увеличивается с ростом качества сжатия; на двоичных быстрее лишь в 1.5–3 раза, поскольку длины совпадающих подстрок (равно как и длины хэш-списков) малы, поэтому большая эффективность предложенных методов реализации LZ77 менее заметна.

Предложенные автором методы при равном качестве сжатия несколько медленнее высокоскоростных вариантов LZ77 (PKZIP -es, PKZIP -ef, RAR -m1). Основном причина этого заключается в том, что PKZIP и RAR реализованы на ассемблере, а PRS — на Си, и поскольку большая часть времени приходится не на поиск подстрок или кодирование, а на их организацию, вычисление CRC-32 и другие накладные расходы, PKZIP и RAR более эффективны. С другой стороны, PKZIP и RAR обычно используются с опциями -en/-ex и -m3/-m5, рекомендованных авторами этих программ, и

практически никогда не применяются в высокоскоростном режиме из-за чрезвычайно низкого качества сжатия, несколько худшую скорость сжатия авторской реализации в высокоскоростном режиме не следует считать ее недостатком.

Замечание 11.3. LHA — единственный из рассмотренных архиваторов, который использует для поиска совпадающих подстрок дерево суффиксов (см. п. 6.1.1). Несмотря на очень малый размер окна (8 Кб) и плохое в связи с этим качество сжатия, LHA уступает всем другим архиваторам по скорости, в том числе и с окном в 64 Кб. Это служит наглядным подтверждением высказанного в п. 6.1.1 тезиса о том, что дерево суффиксов и ряд других асимптотически оптимальных методов поиска подстрок не пригодны для практического использования. ■

Замечание 11.4. LIMIT — единственный из рассмотренных архиваторов, реализующий практичный вариант LZWW алгоритма LZ77 (см. п. 5.5.2.1). LIMIT заметно уступает другим реализациям в скорости, а если ширина окна LZ77 вдвое больше, чем у LZWW, то и по качеству сжатия. Очевидно, реализация LZWW в классическом варианте еще больше замедлит кодирование и декодирование при незначительно лучшем качестве сжатия, поэтому LZWW следует признать не пригодным для практического применения. ■

11.5.2. АЛГОРИТМЫ СЖАТИЯ СОРТИРОВКОЙ БЛОКОВ

По скорости сжатия предложенные методы уверенно лидируют; их скорость сжатия мало отличается и часто лучше скорости сжатия алгоритмов семейства LZ77, а по сравнению с алгоритмами статистического моделирования предложенные методы быстрее на текстовых данных в 4–20 раз, а на двоичных — в 15–30 раз.

Качество предложенных методов сжатия сортировкой блоков на 15–25% лучше качества других реализаций BWT и практически не отличается от качества сжатия лучших методов статистического моделирования; более того, на текстовых данных авторские методы по качеству сжатия обычно являются наилучшими.

Сжатие частичной сортировкой блоков по первым восьми символам практически не отличается от полной сортировки ни по качеству, ни по скорости (различия по скорости становятся заметными лишь при наличии в сжимаемых данных большого количества повторений одной и той же достаточно длинной строки).

Качество сжатия *частичной* сортировкой блоков по первым четырем символам хуже полной сортировки примерно на 0.5% на двоичных данных и на 3–5% — на текстовых, однако скорость частичной сортировки вдвое выше и, самое главное, не зависит от сжимаемых данных. Таким образом, **алгоритм сжатия данных частичной сортировкой блоков по первым четырем символам является наилучшим по соотношению скорости и качества сжатия**.

Замечание 11.5. Алгоритм ACB продемонстрировал наилучшее качество сжатия двоичных данных (excel.exe и winword.exe) и высокоизбыточных текстов (bible.txt и world.txt), которое на 3–5% лучше, чем у авторских методов, занявших второе место; поскольку ACB всегда в 20–30 раз медленнее, с практической точки зрения алгоритмы сжатия, предложенные автором, безусловно лучше.

Отчасти такое отставание авторских алгоритмов объясняется применением кодов Хаффмана; использованием арифметических кодов можно улучшить качество сжатия двоичных и высокоизбыточных данных на 1–3%, однако это увеличит общее время

сжатия в полтора раза, а декодирования — вдвое, что представляется чрезмерно высокой ценой за незначительное улучшение качества сжатия. ■

Замечание 11.6. Алгоритм BRED3 по средней скорости сжатия разделяет первое и второе места с авторскими алгоритмами (среди методов сжатия сортировкой блоков), однако его поведение становится квадратичным по размеру блока, если сжимаемые данные содержат много повторений. Уилер, автор BRED3, утверждает (см. [185]), что на практике такие данные встречаются очень редко. Проведенные эксперименты продемонстрировали несостоятельность этого заявления, поскольку почти все исполняемые файлы для Microsoft Windows достаточной длины являются для алгоритма BRED3 наихудшим случаем; таким образом, этот алгоритм нельзя применять в промышленных продуктах. ■

11.5.3. КОДЫ ХАФФМАНА

Исключительно высокая скорость кодирования и декодирования вышеперечисленных методов сжатия отчасти объясняется большой скоростью построения и декодирования канонических кодов Хаффмана, достигаемой при помощи разработанных автором алгоритмов.

Скорость декодирования очень важна при восстановлении данных, сжатых при помощи LZ77, а также при декодировании текстов на естественных языках (см. главу 4), поскольку в этих случаях декодирование кодов Хаффмана занимает 60–90% общего времени. Использование разработанных автором алгоритмов и структур данных, описанных в п. 3.5, позволило в 2–5 раз ускорить декодирование по сравнению с известными методами и тем самым существенно уменьшить общее время декодирования.

Скорость построения кодов Хаффмана оказывает заметное влияние на скорость сжатия сортировкой блоков, т. к. предложенные автором высокоэффективные методы кодирования результата преобразования (см. п. 10.2) используют много различных алфавитов (16–32), содержащих 128–256 символов, поэтому суммарное время построения кодов может быть значительным; особенно важно повышение скорости построения кодов Хаффмана для методов сжатия частичной сортировкой блоков, обладающих очень высокой скоростью сжатия, не зависящей от сжимаемых данных. Применение методов, описанных в п. 3.4, позволило уменьшить время построения кодов Хаффмана в 3–4 раза, а общее время сжатия — на 15%.

Глава 12

ЗАКЛЮЧЕНИЕ

Автором разработаны, исследованы и детально обоснованы как с теоретической, так и с практической точки зрения **неискажающие методы сжатия текстовой информации и способы их реализации**. Доказано, что при линейном объеме требуемой памяти временная сложность всех предложенных алгоритмов — линейная или линейно-логарифмическая.

1. Разработаны и исследованы новые методы построения кодов Хаффмана и декодирования префиксных кодов для последовательностей ограниченной сверху длины.
   - Изучены некоторые свойства кодов Хаффмана; полученные результаты (о логарифмической зависимости максимальной длины кодового слова от суммы целочисленных весов символов) были использованы при анализе алгоритмов построения и декодирования кодов Хаффмана.
   - Разработаны и исследованы новые методы построения кода Хаффмана. Показано, что если длина кодируемого сообщения не очень велика (менее 2^{32}), а мощность алфавита достаточно большая (более 100 символов), предложенные алгоритмы быстрее известных.
   - Предложены и изучены (с применением разработанных автором Q-сетей на деревьях) новые алгоритмы декодирования префиксных кодов. Доказано, что они линейны по количеству закодированных символов, тогда как время декодирования известными алгоритмами пропорционально длине закодированной последовательности в битах, что в $\mathcal{C}$ больше, где $\mathcal{C}$ — средняя стоимость кодирования в битах (обычно 5–8). Доказано, что почти все известные методы декодирования являются частными случаями предложенных.
2. Разработаны и исследованы новые методы сжатия текстов на естественных языках — русском, английском, немецком и т. д. Доказано, что использование описываемых законами Ципфа и Шварца лингвистических особенностей, присущих *всем* естественным языкам, позволяет добиться очень высокого качества сжатия, часто недостижимого при сжатии другими методами. Доказано, что предложенные методы кодирования почти оптимальны по качеству сжатия.
3. Разработаны и исследованы новые эффективные варианты алгоритмов семейства LZ77 и способы их реализации, предложены новые методы оптимального разрешения неоднозначностей кодирования.
   - Разработаны и изучены новые однопроходные алгоритмы управления порядком поиска идентичных подстрок и замены их ссылками. Доказано, что они оптимальны по стоимости кодирования, линейны по количеству неоднозначностей кодирования и минимальны по количеству необходимых поисков подстрок, определяющих скорость сжатия. Доказано, что все используемые в настоящее время эвристические методы решения проблемы оптимального кодирования — частные случаи алгоритмов, предложенных автором.

- Предложены и обоснованы методы поиска совпадающих подстрок и новые эффективные варианты алгоритма LZ77. Показано, что они используют в 2–5 раз меньше оперативной памяти и в 2–5 раз быстрее известных.

4. Разработана новая методика построения и реализации алгоритмов сжатия сортировкой блоков.
   - Предложены и исследованы новые алгоритмы сортировки блоков, требующие небольшого объема оперативной памяти ($2N$–$3N$ слов, где N — размер блока). Доказано, что они оптимальны по скорости; среднее время сортировки блока размера N почти линейно и равно $O(N \log\log N)$, а в наихудшем случае не превышает $O(N \log N)$; при таком же объеме требуемой памяти время других известных эффективных в среднем алгоритмов сортировки может достигать $O(N\sqrt{N}\log N)$.
   - Разработан и исследован новый класс алгоритмов сжатия *частичной* сортировкой блоков. Доказано, что такие алгоритмы обладают достаточно редким и очень важным качеством: время сжатия или восстановления не зависит от исходных данных и прямо пропорционально их длине, при этом требуется $2N$ слов дополнительной памяти. Показано, что при вдвое большей скорости качество сжатия *частичной* сортировкой лишь незначительно — на 1–5% — хуже качества сжатия полной сортировкой блоков.
   - Предложены и исследованы новые методы кодирования выхода преобразования сортировкой блоков (полной или частичной). Доказано, что они оптимальны по скорости. Показано, что качество такого сжатия мало отличается и часто лучше качества наиболее эффективных статистических методов сжатия.

Таким образом, **разработанные и обоснованные автором методы неискажающего сжатия текстовой информации и способы их реализации являются наилучшими по всем параметрам как среди требующих очень небольшого объема памяти алгоритмов (словарного сжатия), так и среди более ресурсоемких (сжатия сортировкой блоков и статистическим моделированием)**.

СПИСОК ЛИТЕРАТУРЫ

1. **Буяновский Г.** Ассоциативное кодирование // Монитор. — 1994. — N 8. — С. 10–19.
2. **Вирт Н.** Алгоритмы и структуры данных. — М.: Мир, 1989.
3. **Воробьев Н.Н.** Числа Фибоначчи. — М.: Наука, 1984.
4. **Гильберт Е.Н., Мур Э.Ф.** Двоичные кодовые системы переменной длины // Кибернетический сборник. — М.: ИЛ. — 1961. — N 3. — С. 103–141.
5. **Замбалаев Т.Ж.** Алгоритмы сжатия данных сортировкой блоков. — Квалификационная работа на соискание звания бакалавра. — Механико-математический факультет Новосибирского государственного университета, 1997.
6. **Кадач А.В.** Re: максимальная глубина дерева Хаффмана. — news://comp.compression, 1991.
7. **Кадач А.В.** Методы сжатия изображений. — Квалификационная работа на соискание звания бакалавра. — Механико-математический факультет Новосибирского государственного университета, 1992.
8. **Кадач А.В.** Эффективные алгоритмы неискажающего сжатия данных. — Квалификационная работа на соискание звания магистра. — Механико-математический факультет Новосибирского государственного университета, 1994.
9. **Кадач А.В.** Оптимизация качества сжатия в LZ77-схемах // В сб. Средства и инструменты окружений программирования под ред. И.В. Поттосина. — Новосибирск, 1995. — С. 127–142.
10. **Кадач А.В.** Поиск подстрок в LZ77-схемах // В сб. Средства и инструменты окружений программирования под ред. И.В. Поттосина. — Новосибирск, 1995. — С. 143–160.
11. **Кадач А.В.** Эффективные алгоритмы неискажающего сжатия данных сортировкой блоков. Ч. I: Преобразование Барроуза—Уилера и его модификации. — Новосибирск, 1997. — 43 с. — (Препр. / Сиб. отд-ние РАН; Институт систем информатики им. А.П. Ершова; N 42/1).
12. **Кадач А.В.** Эффективные алгоритмы неискажающего сжатия данных сортировкой блоков. Ч. II: Кодирование выхода преобразования Барроуза—Уилера. — Новосибирск, 1997. — 39 с. — (Препр. / Сиб. отд-ние РАН; Институт систем информатики им. А.П. Ершова; N 42/2).
13. **Кадач А.В.** Эффективные методы создания и передачи префиксных кодов. — Новосибирск, 1997. — 27 с. — (Препр. / Сиб. отд-ние РАН; Институт систем информатики им. А.П. Ершова; N 44).
14. **Кадач А.В.** Свойства кодов Хаффмана и эффективные алгоритмы декодирования префиксных кодов. — Новосибирск, 1997. — 44 с. — (Препр. / Сиб. отд-ние РАН; Институт систем информатики им. А.П. Ершова; N 45).
15. **Кадач А.В.** Сжатие текстов и гипертекстов // Программирование. — 1997. — N 4. — С. 47–57.
16. **Кнут Д.Е.** Искусство программирования для ЭВМ. Т. 1: Основные алгоритмы. — М.: Мир, 1976.
17. **Кнут Д.Е.** Искусство программирования для ЭВМ. Т. 2: Получисленные алгоритмы. — М.: Мир, 1977.
18. **Кнут Д.Е.** Искусство программирования для ЭВМ. Т. 3: Сортировка и поиск. — М.: Мир, 1978.
19. **Колмогоров А.Н.** Три подхода к определению понятия количество информации // Проблемы передачи информации. — 1965. — Т. 1, N 1. — С. 3–11.
20. **Кричевский Р.Е.** Сжатие и поиск информации. — М.: Радио и связь, 1989.
21. **Левенштейн В.И.** Об избыточности и замедлении разделимого кодирования натуральных чисел // Проблемы кибернетики. — 1968. — N 20. — С. 173–179.
22. **Налимов Е.Д.** Оценка максимальной длины кода Хаффмана. — Не опубликовано, 1990.
23. **Рябко Б.Я.** Эффективный метод кодирования источников информации, использующий алгоритм быстрого умножения // Проблемы передачи информации. — 1995. — Т. 31, N 3. — С. 3–13.

24. **Рябко Б.Я.** Fast and efficient coding of information sources // IEEE Trans. Inform. Theory. — 1994. — Vol. 40, N 1. — P. 96–99.
25. **Рябко Б.Я.** Comments on A locally adoptive data compression scheme // Communs. ACM. — 1987. — Vol. 16, N 2. — P. 792–792.
26. **Рябко Б.Я.** Сжатие данных методом стопки книг // Проблемы передачи информации. — 1980. — Т. 16, N 4. — С. 16–21.
27. **Abramson N.** Information Theory and Coding. — New York: McGraw-Hill, 1963.
28. **Albers S., Hangerup T.** Improved parallel integer sorting without concurrent writing // 3rd ACM-SIAM Symposium on Descrete Algorithms (SODA). — 1992. — P. 463–462.
29. **Andersson A.** Sorting and searching revisted // Lect. Notes Computer Sci. — 1996. — Vol. 1097. — P. 185–197.
30. **Andersson A., Nilsson S.** — A new efficient radix sort. Lund, 1994. — (Tech. Rep. / Dept. of Comp. Sci., Lund Univ., N LU-CS-94).
31. **Andersson A., Nilsson S.** Efficient implementation of suffix trees // Software — Practice and Experience. — 1995. — Vol. 25, N 2. — P. 129–141.
32. **Andersson A., Nilsson S., Hagerup T., Raman R.** — Sorting in linear time? Lund, 1995. — (Tech. Rep. / Dept. of Comp. Sci., Lund Univ., N LU-CS-95).
33. **Andresson A., Larsson N.J., Swanson K.** Suffix trees on words // Lect. Notes Computer Sci. — 1996. — Vol. 1075. — P. 102–115.
34. **Arnavut Z., Magliveras S.S.** Block sorting and data compression // Proc. IEEE Data Compression Conference, Snowbird, Utah. — 1997. — P. 181–190.
35. **Bell T.C.** Better OPM/L text compression // IEEE Trans. Communs. — 1986. — Vol. 34, N 12. — P. 1176–1182.
36. **Bell T.C.** A Unifying Theory and Improvements for Existing Approaches to Text Compression // PhD thesis. — Dept. of Comp. Sci., Univ. of Canterbury. — 1987.
37. **Bell T.C.** — Longest match string searching for Ziv-Lempel compression. Canterbury, 1989. — (Tech. Rep. / Dept. of Comp. Sci., Univ. of Canterbury, N 6/89).
38. **Bell T.C., Cleary J.G., Witten I.H.** Modelling for text compression // ACM Computing Surveys. — 1989. — Vol. 21, N 4. — P. 557–591.
39. **Bell T.C., Cleary J.G., Witten I.H.** Text Compression. — Englewood Cliffs, NJ: Prentice-Hall, 1990.
40. **Bell T.C., Kulp D.** Longest-match string searching for Ziv-Lempel compression // Software — Practice and Experience. — 1993. — Vol. 23, N 7. — P. 757–772.
41. **Bell T.C., Moffat A.M.** A note on DMC data compression scheme // Computer J. — 1989. — Vol. 32, N 1. — P. 16–20.
42. **Bell T.C., Witten I.H.** The relationship between greedy parsing and symbol-wise text compression // J. ACM. — 1994. — Vol. 41, N 4. — P. 708–724.
43. **Bender P.E., Wolf J.K.** New asymptotic bounds and improvements on the Lempel-Ziv data compression algorithm // IEEE Trans. Inform. Theory. — 1991. — Vol. 37, N 3. — P. 721–729.
44. **Bentley J.L., McIlroy M.D.** Engineering a sort function // Software — Practice and Experience. — 1993. — Vol. 23, N 11. — P. 1249–1265.
45. **Bentley J.L., Sleator D.D., Tarian R.E., Wei V.K.** A locally adoptive data compression scheme // Communs. ACM. — 1986. — Vol. 29, N 4. — P. 320–330.
46. **Bloom C.** New Techniques of Context Modelling and Arithmetic Encoding. — http://www.utexas.edu/~cboom, 1995.
47. **Brent R.P.** A linear algorithm for data compression // Aust. Computer J. — 1987. — Vol. 19, N 2. — P. 64–68.
48. **Brodnik A., Munro J.J.** Membership in constant time and minimum space // Lect. Notes Computer Sci. — 1994. — Vol. 855. — P. 72–81.
49. **Bunton S.** — An executable taxonomy of on-line modeling algorithms. Washington, 1997. — (Tech. Rep. / Dept. of Comp. Sci., Univ. of Washington, N TR–97–02–05).
50. **Bunton S.** — A generalization and improvement to PPM's 'blending'. Washington, 1997. — (Tech. Rep. / Dept. of Comp. Sci., Univ. of Washington, N TR–97–01–10).
51. **Bunton S.** On-Line Stochastic Processes in Data Compression // PhD thesis. — Dept. of Comp. Sci., Univ. of Washington. — 1997.
52. **Bunton S.** — On-line stochastic processes in data compression. Washington, 1997. — (Tech. Rep. / Dept. of Comp. Sci., Univ. of Washington, N TR–97–03–02).
53. **Bunton S.** — A percolating state selector for suffix-tree context models. Washington, 1997. — (Tech. Rep. / Dept. of Comp. Sci., Univ. of Washington, N TR–97–02–06).

54. **Buro M.** On the maximum length of Huffman codes // Inform. Proc. Letters. — 1993. — Vol. 45, N 5. — P. 219–223.
55. **Burrows M., Wheeler D.J.** A block-sorting lossless data compression algorithm. — Palo Alto, 1994. — (Tech. Rep. / DEC Systems Research Center, N 124).
56. **Capocelli R. M., De Santis A.** Tight upper bounds on the redundancy of Huffman codes // IEEE Trans. Inform. Theory. — 1989. — Vol. 35, N 5. — P. 1084–1091.
57. **Capocelli R. M., De Santis A.** Minimum codeword length and redundancy of Huffman codes // Lect. Notes Computer Sci. — 1991. — Vol. 514. — P. 309–319.
58. **Capocelli R. M., De Santis A.** New bounds on the redundancy of Huffman codes // IEEE Trans. Inform. Theory. — 1991. — Vol. 37, N 4. — P. 1095–1104.
59. **Capocelli R. M., Giancarlo R., Taneja I.J.** Bounds on the redundancy of Huffman codes // IEEE Trans. Inform. Theory. — 1986. — Vol. 32, N 6. — P. 854–857.
60. **Choueka Y., Fraenkel A.S., Klein S.T., Perl Y.** — Huffman coding without bit manipulation. Rehovot, 1986. — (Tech. Rep. / Dept. of Appl. Math., Weizmann Inst. Sci., N CS86-05).
61. **Choueka Y., Klein S.T., Perl Y.** Huffman coding without bit manipulation // Proc. 8-th ACM-SIGIR Conference. — 1985. — P. 122–130, Montreal.
62. **Cleary J.G.** Compact hash tables using bidirectional linear probing // IEEE Trans. Computers. — 1984. — Vol. 33, N 9. — P. 824–834.
63. **Cleary J.G., Darragh J.J.** — A fast compact representation of trees using hash tables. Calgary, 1993. — (Tech. Rep. / Dept. of Comp. Sci., Univ. of Calgary, N 162/20).
64. **Cleary J.G., Teahan W.J.** Unbounded length contexts for PPM // Computer J. — 1993. — Vol. 36, N 5. — P. ??–??
65. **Cleary J.G., Teahan W.J., Witten I.H.** Unbounded length contexts for PPM // Proc. IEEE Data Compression Conference, Snowbird, Utah. — 1995. — P. 52–61.
66. **Cleary J.G., Witten I.H.** Data compression using adaptive coding and partial string matching // IEEE Trans. Communs. — 1984. — Vol. 32, N 4. — P. 396–402.
67. **Connell J.B.** A Huffman-Shannon-Fano code // Proc. IEEE. — 1973. — Vol. 61, N 7. — P. 1046–1047.
68. **Cormack G.V.** Implementation of DMC. — ftp://ftp.uwaterloo.ca/pub/dmc, 1987.
69. **Cormack G.V.** Linear Time Huffman Codes. — news://comp.compression, 1995.
70. **Cormack G.V., Horspool R.N.** Algorithms for adaptive Huffman codes // Inform. Proc. Letters. — 1984. — Vol. 18, N 3. — P. 159–165.
71. **Cormack G.V., Horspool R.N.** Data compression using dynamic Markov modelling // Computer J. — 1987. — Vol. 30, N 6. — P. 541–550.
72. **Darragh J.J., Cleary J.G., Witten I.H.** Bonsai: A compact representation of trees // Software — Practice and Experience. — 1993. — Vol. 23, N 3. — P. 277–291.
73. **Davis I.J.** A fast radix sort // Computer J. — 1992. — Vol. 35, N 6. — P. 636–642.
74. **De Prisco R., De Santis A.** On the redundancy achieved by Huffman codes // Inform. Sciences. — 1996. — Vol. 88, N 1. — P. 131–148.
75. **Dreizhen H.** Comments on 'Data compression using static Huffman code-decode tables' // Communs. ACM. — 1996. — Vol. 29, N 2. — P. 149–150.
76. **Elias P.** Interval and recency rank source coding: Two on-line variable-length schemes // IEEE Trans. Inform. Theory. — 1987. — Vol. 33, N 1. — P. 3–10.
77. **Even S., Rodeh M.** Economical encoding of commas between strings // Communs. ACM. — 1978. — Vol. 21, N 4. — P. 315–317.
78. **Fano R.M.** Transmission of Information. — Cambridge, MA: M.I.T. Press, 1949.
79. **Fenwick P.M.** — A new data structure for cumulative frequency tables. Auckland, 1993. — (Tech. Rep. / Dept. of Comp. Sci., Auckland Univ., N CS–93–88).
80. **Fenwick P.M.** A new data structure for cumulative frequency tables // Software — Practice and Experience. — 1994. — Vol. 24, N 3. — P. 327–336.
81. **Fenwick P.M.** — A new data structure for cumulative frequency tables: an improved frequency-to-symbol algorithm. Auckland, 1995. — (Tech. Rep. / Dept. of Comp. Sci., Auckland Univ., N CS–95–110).
82. **Fenwick P.M.** — Experiments with a block sorting text compression algorithm. Auckland, 1995. — (Tech. Rep. / Dept. of Comp. Sci., Auckland Univ., N CS–95–111).
83. **Fenwick P.M.** — Improvements to the block sorting text compression algorithm. Auckland, 1995. — (Tech. Rep. / Dept. of Comp. Sci., Auckland Univ., N CS–95–120).
84. **Fenwick P.M.** — Block sorting text compression — final report. Auckland, 1996. — (Tech. Rep. / Dept. of Comp. Sci., Auckland Univ., N CS–96–130).

85. **Fenwick P.M.** — Punctured elias codes for variable-length coding of the integers. Auckland, 1996. — (Tech. Rep. / Dept. of Comp. Sci., Auckland Univ., N 137).
86. **Fenwick P.M., Gutmann P.C.** — Fast LZ77 string matching. Auckland, 1994. — (Tech. Rep. / Dept. of Comp. Sci., Auckland Univ., N TR–102).
87. **Fiala E.R., Green D.H.** Data compression with finite windows // Communs. ACM. — 1988. — Vol. 32, N 4. — P. 490–505.
88. **Fletcher J.G.** An arithmetic checksum for serial transmission // IEEE Trans. Communs. — 1982. — Vol. 30, N 1. — P. 247–252.
89. **Fraenkel A.S., Klein S.T.** Bounding the depth of search trees // Computer J. — 1993. — Vol. 36. — P. 668–678.
90. **Fredman M.L., Willard D.E.** Surpassing the information theoretic bound with fusion trees // J. Computer and System Sciences. — 1993. — Vol. 47, N 3. — P. 424–436.
91. **Gavish A.** Match-length functions for data compression // IEEE Trans. Inform. Theory. — 1996. — Vol. 42, N 5. — P. 1375–1380.
92. **Gilkrist J.** Archiver Comparison Test. — http://www.geocities.com/SiliconValley/Park/4264, 1992–1997.
93. **Golomb S.W.** Run-length encodings // IEEE Trans. Inform. Theory. — 1966. — Vol. 12, N 3. — P. 399–401.
94. **Guazzo M.** A general minimum-redundancy source-coding algorithm // IEEE Trans. Inform. Theory. — 1980. — Vol. 26, N 1. — P. 15–25.
95. **Han Y., Shen X.** Conservative algorithms for parallel and sequential sorting // Lect. Notes Computer Sci. — 1995. — Vol. 959. — P. 324–333.
96. **Hirschberg D.S., Lelewer D.A.** — Efficient decoding of prefix codes. Irvine, 1989. — (Tech. Rep. / Dept. of Inform. and Comp. Sci., Univ. of California, N ICS–TR–89–09).
97. **Hirschberg D.S., Lelewer D.A.** — Context modelling for text compression. Pomona, 1992. — (Tech. Rep. / Dept. of Comp. Sci., California State Polytechnic Univ.).
98. **Horspool R.N., Cormack G.V.** Comments on A locally adoptive data compression scheme // Communs. ACM. — 1987. — Vol. 16, N 2. — P. 792–794.
99. **Horspool R.N., Cormack G.V.** Constructing word-based text compression algorithms // Proc. IEEE Data Compression Conference, Snowbird, Utah. — 1992. — P. 62–71.
100. **Horspool R.N., Cormack G.V.** Comments on 'Data compression using static Huffman code-decode tables' // Communs. ACM. — 1996. — Vol. 29, N 2. — P. 150–152.
101. **Howard P.G.** — The design and analysis of efficient lossless data compression systems. Providence, Rhode Island, 1993. — (Tech. Rep. / Dept. of Comp. Sci., Brown Univ., N CS–93–28).
102. **Howard P.G.** The Design and Analysis of Efficient Lossless Data Compression Systems // PhD thesis. — Dept. of Comp. Sci., Brown Univ., Providence, Rhode Island. — 1993.
103. **Howard P.G., Vitter J.S.** — Analisys of arithemic coding for data compression. Providence, Rhode Island, 1991. — (Tech. Rep. / Dept. of Comp. Sci., Brown Univ., N CS–91–03).
104. **Howard P.G., Vitter J.S.** — Analisys of arithemic coding for data compression. Providence, Rhode Island, 1992. — (Tech. Rep. / Dept. of Comp. Sci., Brown Univ., N CS–92–17).
105. **Huffman D.A.** A method for the construction of minimum-redundancy codes // Proc. IRE. — 1952. — Vol. 40. — P. 1098–1101.
106. **Jacobson G.** Random access in Huffman-coded files // Proc. IEEE Data Compression Conference, Snowbird, Utah. — 1992. — P. 368–377.
107. **Jacobsson M.** Compression of character strings by an adaptive dictionary // BIT. — 1985. — Vol. 25, N 4. — P. 593–603.
108. **Jones D.W.** Application of splay trees to data compression // Communs. ACM. — 1988. — Vol. 31, N 8. — P. 996–1007.
109. **Katajainen J., Moffat A.M.** In-Place Calculation of Minimum-Redundancy Codes. — submitted, 1997.
110. **Katajainen J., Moffat A.M., Turpin A.** A fast and space-econimical for length-limited coding // Lect. Notes Computer Sci. — 1995. — Vol. 1004. — P. 12–21.
111. **Katajainen J., Moffat A.M., Turpin A.** — A fast and space-economical algorithm for length-limited coding. Parkville, 1995. — (Tech. Rep. / Dept. of Comp. Sci., Univ. of Melbourne, N 95/17).
112. **Katajainen J., Moffat A.M., Turpin A.** — In-place calculation of minimum-redundacy codes. Copenhagen, 1995. — (Tech. Rep. / Dept. of Comp. Sci., Univ. of Copenhagen, N 95/12).

113. **Katajainen J., Moffat A.M., Turpin A.** — Space-efficient construction of optimal prefix codes. Copenhagen, 1995. — (Tech. Rep. / Dept. of Comp. Sci., Univ. of Copenhagen, N 95/10).
114. **Katajainen J., Raita T.** An approximation algorithm for space-optimal encoding of a text // Computer J. — 1989. — Vol. 32, N 3. — P. 228–237.
115. **Katajainen J., Raita T.** An analysis of the longest match and the greedy heuristics in text encoding // J. ACM. — 1992. — Vol. 39, N 2. — P. 281–294.
116. **Katona G.H.O., Nemetz T.O.H.** Huffman codes and self-information // IEEE Trans. Inform. Theory. — 1976. — Vol. 22, N 2. — P. 337–340.
117. **Kirkpatrick D., Reisch S.** Upper bounds for sorting integers on random access machines // Theor. Computer Sci. — 1984. — Vol. 28, N 3. — P. 263–276.
118. **Kirrinnis P.** An optimal bound for path weights in Huffman trees // Inform. Proc. Letters. — 1994. — Vol. 51, N 2. — P. 107–110.
119. **Klein S.T.** Space- and time-efficient decoding with canonical Huffman trees // Lect. Notes Computer Sci. — 1997. — Vol. 1027. — P. 65–75.
120. **Knuth D.E.** Dynamic Huffman coding // J. Algorithms. — 1985. — Vol. 6. — P. 163–180.
121. **Kraft L.G.** A device for quantizing, grouping, and coding amplitutde modulated pulses // MS thesis. — Dept. of E. Eng., Massachusetts Inst. Technology, 1949.
122. **Langdon G.G., Rissanen Jr., Rissanen J.J.** A simple general binary source code // IEEE Trans. Inform. Theory. — 1982. — Vol. 28, N 5. — P. 800–803.
123. **Larmore L.L., Hirschberg D.S.** A fast algorithm for optimal length-limited codes // J. ACM. — 1990. — Vol. 37, N 3. — P. 464–473.
124. **Lelewer D.A., Hirschberg D.S.** — An order-2 context model for data compression with reduced time and space requirements. Irvine, 1990. — (Tech. Rep. / Dept. of Inform. and Comp. Sci., Univ. of California, N ICS–TR–90–33).
125. **Lelewer D.A., Hirschberg D.S.** — Streamlining context models for data compression. Proc. IEEE Data Compression Conference, Snowbird, Utah. — 1991. — P. 313–322.
126. **Lempel A., Ziv J.** On the complexity of finite sequences // IEEE Trans. Inform. Theory. — 1976. — Vol. 22, N 1. — P. 75–81.
127. **Long P.M., Natsev A. I., Vitter J.S.** Text compression via alphabet re-representation // Proc. IEEE Data Compression Conference, Snowbird, Utah. — 1997. — P. 161–170.
128. **MacMillian B.** Two inequalities implied by unique decipherability // IRE Trans. Inform. Theory. — 1956. — Vol. 2. — P. 115–116.
129. **Manber U., Myers G.** Suffix arrays: A new method for on-line string searching // Proc. 1st ACM-SIAM Symposium on Descrete Algorithms (SODA). — 1990. — P. 319–327.
130. **Manber U., Myers G.** Suffix arrays: A new method for on-line string searches // SIAM J. Computing. — 1993. — Vol. 22, N 5. — P. 935–948.
131. **Manstetten R.** Tight bounds on the redundancy of Huffman codes // IEEE Trans. Inform. Theory. — 1992. — Vol. 38, N 1. — P. 144–151.
132. **McCreight E.M.** A space-economical suffix tree construction algorithm // J. ACM. — 1976. — Vol. 23, N 2. — P. 262–272.
133. **McIlroy P.M., Bostic K., McIlroy M.D.** Engineering radix sort // Computing Systems. — 1993. — Vol. 6, N 1. — P. 5–27.
134. **McIntyre D.R., Pechura M.A.** Data compression using static Huffman code-decode tables // Communs. ACM. — 1985. — Vol. 28, N 6. — P. 612–616.
135. **Mehlhorn K.** Data structures and algorithms. V 1: Sorting and searching. — Berlin: Springer-Verlag, 1984.
136. **Merret T.H., Shang H.** Trie methods for representing text // Foundations of Data Organization and Algorithms: 4th International Conference (FODO '93). — 1993. — P. 130–145, Chicago, Il.
137. **Miller V.S., Wegman M.N.** — Variations on a theme by Ziv and Lempel. 1984. — (Tech. Rep. / IBM Research Division, N RC 10630).
138. **Miller V.S., Wegman M.N.** Variations on a theme by Ziv and Lempel // Combinatorial Algorithms on Words, Vol. F12 of NATO ASI Series. — P. 131–140 // Berlin: Springer-Verlag, 1985.
139. **Moffat A.M.** Word based text compression // Software — Practice and Experience. — 1989. — Vol. 19, N 2. — P. 185–198.
140. **Moffat A.M.** Implementing the PPM data compression scheme // IEEE Trans. Inform. Theory. — 1990. — Vol. 36, N 11. — P. 1917–1921.
141. **Moffat A.M.** Linear time adaptive arithmetic coding // IEEE Trans. Inform. Theory. — 1990. — Vol. 36, N 2. — P. 401–406.

142. **Moffat A.M., Katajainen J.** In-place calculation of minimum-redundancy codes // Lect. Notes Computer Sci. — 1995. — Vol. 955. — P. 393–402.
143. **Moffat A.M., Sharman N., Zobel J.** — Static compression for dynamic text. Parkville, 1993. — (Tech. Rep. / Dept. of Comp. Sci., Univ. of Melbourne, N 93/28).
144. **Moffat A.M., Turpin A.** — On the implementation of minimum-redundancy prefix codes. Parkville, 1995. — (Tech. Rep. / Dept. of Comp. Sci., Univ. of Melbourne, N 95/38).
145. **Moffat A.M., Turpin A.** On the implementation of minimum-redundancy prefix codes // Proc. IEEE Data Compression Conference, Snowbird, Utah. — 1996. — P. 170–179.
146. **Moffat A.M., Turpin A., Katajainen J.** Space-efficient construction of optimal prefix codes // Proc. IEEE Data Compression Conference, Snowbird, Utah. — 1995. — P. 192–201.
147. **Moffat A.M., Witten I.H., Neal R.M.** Arithmetic coding revisited // Proc. IEEE Data Compression Conference, Snowbird, Utah. — 1995. — P. 202–211.
148. **Moffat A.M., Zobel J.** — Compression and fast indexing for multi-gigabyte text databases. Parkville, 1993. — (Tech. Rep. / Dept. of Comp. Sci., Univ. of Melbourne, N 93/2).
149. **Moffat A.M., Zobel J., Sharman N.B.** Static compression for dynamic texts (extended abstract) // Proc. IEEE Data Compression Conference, Snowbird, Utah. — 1994. — P. 126–135.
150. **Morrison D.R.** PATRICIA — Practical Algorithm To Retrieve Information Coded in Alphanumeric // J. ACM. — 1968. — Vol. 15, N 3. — P. 514–534.
151. **Nilsson S.** Radix Sorting and Searching // PhD thesis. — Dept. of Comp. Sci., Lund Univ. — 1996.
152. **Rice R.F.** — Some practical universal noiseless coding techniques. Pasadena, 1979. — (Tech. Rep. / Jet Propulsion Laboratory, N JPL 79–22).
153. **Rissanen J.J., Langdon G.G.** Universal modelling and coding // IEEE Trans. Inform. Theory. — 1981. — Vol. 27, N 1. — P. 12–23.
154. **Rivest R.** The RC5 encryption algorithm // Lect. Notes Computer Sci. — 1995. — Vol. 1008. — P. 86–96.
155. **Rodeh M., Pratt V.R., Even S.** Linear algorithm for data compression via string matching // J. ACM. — 1981. — Vol. 28, N 1. — P. 16–24.
156. **Rubin F.** Arithmetic stream coding using fixed precision registers // IEEE Trans. Inform. Theory. — 1979. — Vol. 25, N 6. — P. 672–675.
157. **Rubin F.** Experiments in text file compression // Communs. ACM. — 1996. — Vol. 19, N 11. — P. 617–623.
158. **Schack R.** The length of a typical Huffman codeword // IEEE Trans. Inform. Theory. — 1994. — Vol. 40, N 4. — P. 1246–1247.
159. **Schmidhuber J., Heil S.** Sequential neural text compression // IEEE Trans. Neural Networks. — 1996. — Vol. 7, N 1. — P. 142–146.
160. **Schwartz E.S.** A dictionary for minimum redundancy encoding // J. ACM. — 1963. — Vol. 10, N 4. — P. 413–439.
161. **Schwartz E.S., Kallick B.** Generating a canonical prefix encoding // Communs. ACM. — 1964. — Vol. 7, N 3. — P. 166–169.
162. **Shannon C.E.** Prediction and entropy of printed English // Bell Syst. Tech. J. — 1951. — Vol. 30. — P. 50–64.
163. **Shannon C.E., Weaver W.** The Mathematical Theory of Communication. — Urbana, IL: University of Illinois Press, 1949.
164. **Sieminski A.** Fast decoding of the Huffman codes // Inform. Proc. Letters. — 1988. — Vol. 26, N 5. — P. 237–241.
165. **Sleator D.S., Tarjan R.E.** Self-adjusting binary search trees // J. ACM. — 1985. — Vol. 32, N 3. — P. 652—686.
166. **Storer J.A.** Data Compression: Methods and Theory. — Rockville, ML: Computer Science Press, 1988.
167. **Storer J.A., Szymansky T.G.** Data compression via textual substitution // J. ACM. — 1982. — Vol. 25, N 4. — P. 928–951.
168. **Teuhola J., Raita T.** — Application of a finite-state model to text compression. Turku, 1993. — (Tech. Rep. / Dept. of Comp. Sci., Univ. of Turku, N TR–93–5).
169. **Teuhola J., Raita T.** Application of a finite-state model to text compression // Computer J. — 1993. — Vol. 36, N 7. — P. 607–614.
170. **Teuhola J., Raita T.** Arithmetic coding into fixed-length codewords // IEEE Trans. Inform. Theory. — 1994. — Vol. 40, N 1. — P. 219–223.

171. **Thorup M.** — An $o(\log\log n)$ priority queue. Copenhagen, 1995. — (Tech. Rep. / Dept. of Comp. Sci., Univ. of Copenhagen, N DIKU–TR–95/5).
172. **Tischer P.** A modified Lempel-Ziv-Welch data compression scheme // Aust. Computer Sci. Communs. — 1987. — Vol. 9, N 1. — P. 262–272.
173. **Turpin A., Moffat A.M.** — Practical length-limited coding for large alphabets. Parkville, 1995. — (Tech. Rep. / Dept. of Comp. Sci., Univ. of Melbourne, N 94/16).
174. **Turpin A., Moffat A.M.** Practical length-limited coding for large alphabets // Aust. Computer Sci. Communs. — 1995. — Vol. 17, N 1. — P. 523–532.
175. **Turpin A., Moffat A.M.** Practical length-limited coding for large alphabets // Computer J. — 1995. — Vol. 38, N 5. — P. 339–347.
176. **Turpin A., Moffat A.M.** Efficient implementation of the package-merge algorithm for length-limited codes // Proc. Australian Computing Theory Symposium. — 1996. — P. 187–195, Melbourne.
177. **Ukkonen E.** On-line construction of suffix trees // Algorithica. — 1995. — Vol. 14, N 3. — P. 249–260.
178. **van Emde Boas P.** Preserving order in a forest in less than logarithmic time and linear space // Inform. Proc. Letters. — 1977. — Vol. 6, N 3. — P. 80–82.
179. **Van Leeuwen J.** On the construction of Huffman trees // Proc. 3rd International Conference on Automata, Languages, and Programming. — 1976. — P. 382–410, Edinburgh University.
180. **Vitter J.S.** Design and analisys of dynamic Huffman algorithm // J. ACM. — 1987. — Vol. 34, N 4. — P. 825–845.
181. **Vitter J.S.** Algorithm 673: Dynamic Huffman coding // ACM Trans. Math. Software. — 1989. — Vol. 15, N 2. — P. 158–167.
182. **Wagner R.A.** Algorithm 444: An algorithm for extracting phrases in a space-optimal fashion // Communs. ACM. — 1973. — Vol. 16, N 3. — P. 148–152.
183. **Wagner R.A.** Common phrases and minimum-space text storage // Communs. ACM. — 1973. — Vol. 16, N 3. — P. 148–152.
184. **Welch T.A.** A technique for high-performance data compression // IEEE Computer. — 1984. — Vol. 17, N 6. — P. 8–19.
185. **Wheeler D.J.** Bred: A Fast Block-Sorting Algorithm. — ftp.cl.cam.ac.uk/~djw/bred3.ps, 1996.
186. **Wheeler D.J., Needham R.M.** TEA, a tiny encryption algorithm // Lect. Notes Computer Sci. — 1995. — Vol. 1008. — P. 363–366.
187. **Wheeler D.J., Needham R.M.** TEA Extensions. — ftp.cl.cam.ac.uk/~djw/xtea.ps, 1997.
188. **Williams R.N.** Dynamic history predictive compression // Inform. Systems. — 1988. — Vol. 13, N 1. — P. 129–140.
189. **Williams R.N.** Adaptive Data Compression. — Kluwer Academic Publishers, Norwell, MA, 1991.
190. **Williams R.N.** An extremely fast Ziv-Lempel data compression algorithm // Proc. IEEE Data Compression Conference, Snowbird, Utah. — 1991. — P. 362–371.
191. **Witten I.H., Bell T.C.** The zero frequency problem: Estimating the probabilities of novel events in adaptive data compression // IEEE Trans. Inform. Theory. — 1991. — Vol. 37, N 3. — P. 324–326.
192. **Witten I.H., Bell T.C., Nevill C.G.** Models for compression in full-text retrieval systems // Proc. IEEE Data Compression Conference, Snowbird, Utah. — 1991. — P. 23–32.
193. **Witten I.H., Cunningham S.J., Vallabh M., Bell T.C.** — A New Zealand digital library for computer science research. Hamilton, 1994. — (Tech. Rep. / Dept. of Comp. Sci., Waikato Univ., N TR–95–6).
194. **Witten I.H., Moffat A.M., Bell T.C.** Managing Gigabytes: Compressing and Indexing Documents and Images. — New York: Van Nostrand Reinhold, 1994.
195. **Witten I.H., Neal R.M., Cleary J.G.** Arithmetic coding for data compression // Communs. ACM. — 1987. — Vol. 30, N 6. — P. 520–540.
196. **Wyner A.D., Wyner A.J.** Improved redundancy of a version of the Lempel-Ziv algorithm // IEEE Trans. Inform. Theory. — 1995. — Vol. 41, N 3. — P. 723–731.
197. **Wyner A.D., Ziv J.** The sliding-window Lempel-Ziv algorithm is asymptotically optimal // Proc. IEEE. — 1994. — Vol. 82, N 6. — P. 872–877.
198. **Yokoo H.** Improved variations relating the Ziv-Lempel and Welch-type algorithms for sequential data compression // IEEE Trans. Inform. Theory. — 1992. — Vol. 38, N 1. — P. 73–81.
199. **Zipf G.K.** The Psicho-Biology of Language. — Boston, MA: Houghton Muffling, 1935.

200. **Zipf G.K.** Human Behavior and the Principle of Least Efford. — Reading, MA: Addison-Wesley, 1949.
201. **Ziv J., Lempel A.** A universal algorithm for sequential data compression // IEEE Trans. Inform. Theory. — 1977. — Vol. 23, N 3. — P. 337–343.
202. **Ziv J., Lempel A.** Compression of individual sequences via variable-rate coding // IEEE Trans. Inform. Theory. — 1978. — Vol. 24, N 5. — P. 530–536.
203. **Zobrist A.L.** — A new hashing method with application for game playing. Madison, 1970. — (Tech. Rep. / Dept. of Comp. Sci., Univ. of Wisconsin, N 88).