Д. Ватолин, А. Ратушняк,
М. Смирнов, В. Юкин

МЕТОДЫ СЖАТИЯ ДАННЫХ

УСТРОЙСТВО АРХИВАТОРОВ, СЖАТИЕ ИЗОБРАЖЕНИЙ И ВИДЕО

МОСКВА ■ "ДИАЛОГ-МИФИ" ■ 2003

УДК 681.3
ББК 32.97
В21

Ватолин Д., Ратушняк А., Смирнов М., Юкин В.

В21 Методы сжатия данных. Устройство архиваторов, сжатие изображений и видео. – М.: ДИАЛОГ-МИФИ, 2003. – 384 с.

ISBN 5-86404-170-x

В книге описаны основные классические и современные методы сжатия: метод Хаффмана, арифметическое кодирование, LZ77, LZW, PPM, BWT, LPC и т. д. Разбираются алгоритмы, использующиеся в архиваторах Zip, HA, CabArc (*.cab-файлы), RAR, BZIP2, RK. Отдельный раздел посвящен алгоритмам сжатия изображений, использующимся в форматах PCX, TGA, GIF, TIFF, CCITT G-3, JPEG, JPEG2000. Рассмотрено фрактальное сжатие, вэйвлет-сжатие и др. Изложены принципы компрессии видеоданных, дан обзор стандартов MPEG, MPEG-2, MPEG-4, H.261 и H.263.

Некоторые методы повышения сжатия на русском языке публикуются впервые. Книга содержит большое количество примеров и упражнений и ориентирована на студентов и преподавателей вузов. Материал книги позволяет самостоятельно несколькими способами написать архиватор с характеристиками, превосходящими программы типа pkzip и arj. Ответы на вопросы для самоконтроля и исходные тексты программ можно найти на сайте http://compression.graphicon.ru/.

Учебно-справочное издание
Ватолин Д., Ратушняк А., Смирнов М., Юкин В.
Методы сжатия данных. Устройство архиваторов, сжатие изображений и видео

Редактор О. А. Голубев
Корректор В. С. Кустов
Макет Н. В. Дмитриевой

Лицензия ЛР N 071568 от 25.12.97. Подписано в печать 22.09.2003
Формат 60x84/16. Бум. офс. Печать офс. Гарнитура Таймс.
Усл. печ. л. 22,32. Уч.-изд. л. 17. Тираж 3 000 экз. Заказ 1468

ЗАО "ДИАЛОГ-МИФИ", ООО "Д и М"
115409, Москва, ул. Москворечье, 31, корп. 2. Т.: 320-43-55, 320-43-77
Http://www.bitex.ru/~dialog. E-mail: dialog@bitex.ru

Подольская типография
142100, г. Подольск, Московская обл., ул. Кирова, 25

ISBN 5-86404-170-x

Предисловие

Основная задача, которая ставилась при написании этой книги, – изложить в одном издании и достаточно единообразно современные методы сжатия данных. Тема эта необъятная. Все разделы достойны отдельных книг, а развернутое описание методов сжатия видеоданных требует нескольких томов. Поэтому в изложении основной упор делался на базовые идеи и концепции, используемые при сжатии. Нам бы хотелось, чтобы, изучив изложенные в книге методы, читатель мог понять принципы работы большинства компрессоров и разобраться в методах сжатия данных по исходным текстам программ.

Многие в первую очередь зададут вопрос, где взять исходные тексты архиваторов? И чтобы степень сжатия была хорошая. Это не проблема. Сейчас в любом достаточно крупном городе России можно без труда приобрести компакт-диски с дистрибутивом Linux. В составе исходных текстов этой операционной системы есть текст на языке программирования Си, позволяющий работать со сжатыми дисками файловой системы NTFS. Следовательно, можно посмотреть, как организовано сжатие в операционных системах типа Windows NT. Также в состав практически всех дистрибутивов Linux входят исходные тексты утилит сжатия gzip и bzip2, дающих неплохие результаты. Поэтому исходные тексты обычных программ сжатия, имеющих хорошие характеристики, можно найти буквально на каждом лотке с CD-ROM и на десятках тысяч сайтов с дистрибутивами Linux в Интернете. Кроме того, есть сайты с исходными текстами эффективных и качественно реализованных архиваторов, которые пока менее известны и распространены. Например, сайт проекта 7-Zip http://www.7-zip.org. Примечательно, что такая ситуация складывается не только со сравнительно простыми универсальными архиваторами (10–200 Кб исходных текстов), но и с такими сложными системами, как видеокодеки (400–4000 Кб текста). Свободно доступны тексты кодеков вполне промышленного качества, таких, как OpenDivX и On2 VP3, тексты конвертера VirtualDub, поддерживающего несколько форматов видео. Таким образом, у человека, имеющего доступ к Интернету и умеющего искать (попробуйте начать с http://dogma.net/DataCompression), возникает проблема не найти исходные тексты программ, а понять, ***как*** и ***почему*** эти программы работают. Именно в этом мы и пытаемся помочь нашим читателям.

Мы старались не только охватить все основные подходы, используемые при сжатии разных типов данных, но и рассказать о современных специальных техниках и модификациях алгоритмов, зачастую малоизвестных. Естественно, полнее всего раскрыты темы, в области которых авторы наиболее компетентны. Поэтому некоторые алгоритмы рассмотрены глубоко и основательно, в то время как другие, также достойные, описаны конспективно или лишь упомянуты. Тем не менее главная цель достигнута: изложение получилось достаточно интересным, практическим и охватывающим ключевые методы сжатия данных различных типов.

В тексте книги встречается большое количество русских имен. Для многих будет неожиданностью, что очень много интересных и эффективных компрессоров написано программистами, живущими в России и других странах СНГ. В частности, это RAR (автор Евгений Рошал), 7-Zip, BIX, UFA, 777 (Игорь Павлов), BMF, PPMD, PPMonstr (Дмитрий Шкарин), YBS (Вадим Юкин), ARI, ERI (Александр Ратушняк), PPMN (Максим Смирнов), PPMY (Евгений Шелвин) и многие другие. Причем по

данным тестирования, например, на сжатие исполняемых файлов (http://compression.ca/act-executable.html) на 31.01.2002 в рейтинге, содержащем 154 архиватора, программы наших соотечественников занимают 8 позиций в двадцатке лучших. Авторам этой книги искренне хотелось бы, чтобы наших архиваторов в таких рейтингах становилось все больше.

Написание материалов книги и ответственность были распределены следующим образом:

Д. Ватолиным написан "Классический вариант алгоритма" в п. "Арифметическое сжатие" (разд. 1, гл. 1), разд. 2 за исключением подразд. "Методы обхода плоскости", разд. 3, а также приложение 2;

А. Ратушняком – подрад. "Разделение мантисс и экспонент", "Нумерующее кодирование", "Векторное квантование", "Методы обхода плоскости", гл. 2 разд. 1, гл. 6 а также части Введения "Определения. Аббревиатуры и классификации методов сжатия", "Замечание о методах, алгоритмах и программах";

М. Смирновым – гл. 3 и 4 разд.1, подразд. "Препроцессинг текстов" (гл. 7) и приложение 1;

В. Юкиным – гл. 5, "Интервальное кодирование" в подразд. "Арифметическое сжатие" (гл. 1 разд. 1), подразд. "Препроцессинг нетекстовых данных" и "Выбор метода сжатия" гл. 7.

Часть введения "Сравнение алгоритмов по степени сжатия" написана совместно А. Ратушняком и М. Смирновым.

В заключение мы хотим выразить свою благодарность Дмитрию Шкарину, Евгению Шелвину, Александру Жиркову, Владимиру Вежневцу, Алексею Игнатенко: их конструктивная критика, интересные дополнения, замечания и советы способствовали существенному улучшению качества книги. Также мы благодарны лаборатории компьютерной графики ВМиК МГУ и персонально Юрию Матвеевичу Баяковскому и Евгению Викторовичу Шикину за организационную и техническую поддержку.

Мы будем признательны читателям за их отзывы и критические замечания, отправленные непосредственно нам по электронной почте по адресу compression@graphicon.ru.

Исходные тексты программ, ответы на контрольные вопросы и упражнения вы сможете получить по адресу http://compression. graphicon.ru/.

Дмитрий Ватолин, Александр Ратушняк,
Максим Смирнов, Вадим Юкин
Москва – Новосибирск – Санкт-Петербург, 2002 г.

ВВЕДЕНИЕ

Обзор тем

Структура книги отвечает принятой авторами классификации методов сжатия данных и определенным традициям, сложившимся в литературе по сжатию.

В 1 разд. – "Методы сжатия без потерь" описаны основные подходы, применяющиеся при кодировании источников без памяти, источников типа "аналоговый сигнал" и источников с памятью, а также методы предобработки типичных данных, обеспечивающие улучшение сжатия для распространенных алгоритмов.

Ввиду особой практической и теоретической важности, а также распространенности универсальные методы кодирования источников с памятью были рассмотрены в соответствующих отдельных главах: "Словарные методы сжатия данных", "Методы контекстного моделирования", "Преобразование Барроуза–Уилера". Описаны не только базовые варианты алгоритмов, но и множество специфических техник улучшения сжатия, в том числе малоизвестных. Материалы этих глав могут быть полезны не только для неискушенного читателя, но и для специалиста.

Ввиду актуальности и определенной новизны излагаемых приемов была написана гл. 7 – "Предварительная обработка данных".

В разд. 2 – "Методы сжатия изображений" объяснена специфика кодирования растровых изображений, описаны основные классические и современные алгоритмы сжатия изображений без потерь и с потерями. В частности, изложены особенности сравнительно нового алгоритма JPEG-2000.

В разд. 3 – "Методы сжатия видео" указаны особенности задач компрессии видеоданных, изложены базовые идеи, лежащие в основе алгоритмов сжатия видео, дан обзор ряда известных стандартов, в частности MPEG-4. Необъятность темы и необходимость удержания размера книги в разумных пределах определили конспективный характер этого раздела. Интересующиеся читатели могут продолжить изучение вопросов сжатия видеоданных, руководствуясь изложенными в разделе базовыми принципами, в чем им поможет предложенный список литературы.

Определения, аббревиатуры и классификации методов сжатия

Базовые определения

Бит – это "атом" цифровой информации: переменная, которая может принимать ровно два различных значения:

- "1" (единица, да, истина, существует);
- "0" (нуль, нет, ложь, не существует).

Любая система, которую можно перевести в одно из двух различных задаваемых состояний и удержать в нем в течение требуемого промежутка времени, может быть использована для хранения 1 бита информации.

Емкость для хранения бита можно представлять себе как небольшой "ящик" где-то в пространстве-времени (в микросхеме, на магнитном/оптическом диске, линии связи) с двумя возможными состояниями: *полный* – "1" и *пустой* – "0".

Данные – информация в цифровом виде.

Объем данных измеряется в битах, но может быть и рациональным числом, а не только целым.

R-битовый элемент – совокупность R битов – имеет 2^R возможных значений-состояний. Большинство источников цифровой информации порождает элементы одного **размера** R. А в большинстве остальных случаев – элементы нескольких размеров: R_1, R_2, R_3... (например, 8, 16 и 32).

Байт – это 8-битовый элемент: совокупность 8 битов.

Входная **последовательность** в общем случае бесконечна, но ее элементы обязательно пронумерованы, поэтому имеют смысл понятия "предыдущие" и "последующие" элементы. В случае многомерных данных есть много способов создания последовательности из входного множества.

Блок – конечная последовательность цифровой информации.

Поток – последовательность с неизвестными границами: данные поступают маленькими блоками, и нужно обрабатывать их сразу, не накапливая. Блок – последовательность с произвольным доступом, а поток – с последовательным.

Сжатием блока называется такое его описание, при котором создаваемый **сжатый** блок содержит меньше битов, чем исходный, но по нему возможно однозначное восстановление каждого бита исходного блока. Обратный процесс, восстановление по описанию, называется **разжатием**.

Используют и такие пары терминов: компрессия/декомпрессия, кодирование/декодирование, упаковка/распаковка.

Под просто *сжатием* будем далее понимать сжатие без потерь (lossless compression).

Сжатие с потерями (lossy compression) – это два разных процесса:

1) выделение сохраняемой части информации с помощью модели, зависящей от цели сжатия и особенностей источника и приемника информации;
2) собственно сжатие, без потерь.

При измерении физических параметров (яркость, частота, амплитуда, сила тока и т. д.) неточности неизбежны, поэтому "округление" вполне допустимо. С другой стороны, приемлемость сжатия изображения и звука со значительными потерями обусловлена особенностями восприятия такой информации органами чувств человека. Если же предполагается компьютерная обработка изображения или звука, то требования к потерям гораздо более жесткие.

Конечную последовательность битов назовем **кодом**[1], а количество битов в коде – **длиной кода**.

Конечную последовательность элементов назовем **словом**, а количество элементов в слове – **длиной слова**. Иногда используются синонимы **строка** и **фраза**. В общем случае слово построено из R-битовых элементов, а не 8-битовых. Таким образом, код – это слово из 1-битовых элементов.

Например, в блоке из 14 элементов "кинчотсихыннад" одно слово длиной 14 элементов, два слова длиной 13 элементов, и т. д., 13 слов длиной 2 и 14 слов длиной 1. Аналогично в блоке из семи битов "0100110" один код длиной 7 бит, два кода длиной 6 бит, и т. д., семь кодов длиной 1 бит.

Символ – это "атом" некоторого языка (например, буквы, цифры, ноты, символы шахматных фигур, карточных мастей). Во многих случаях под символом имеют в виду R-битовый элемент (обычно байт), однако элементы мультимедийных данных все-таки не стоит называть символами: они содержат количественную, а не качественную информацию.

Качественными можно называть данные, содержащие элементы-указатели на символы внутри таблиц или указатели на ветви алгоритма (и таким образом "привязанные" к некоторой структуре: таблице, списку, алгоритму и т. п.). А **количественными** – множества элементов, являющиеся записями значений каких-либо величин.

ASCII (American Standard Code for Information Interchange – Американский стандартный код для обмена информацией) каждому значению байта

---

[1] *В теории информации кодом называется совокупность всех битовых последовательностей, применяемых для представления порождаемых источником символов. Авторы сознательно пошли на использование слова "код" в обыденном значении.*

ставит в соответствие символ. Но чтобы построить однозначное соответствие для всех необходимых символов из множества национальных алфавитов народов мира, требуется больше: по крайней мере 16 бит на символ (что и обеспечивает стандарт **Unicode**).

Множество всех различных символов, порождаемых некоторым источником, называется **алфавитом**, а количество символов в этом множестве – **размером алфавита**. Источники *данных* порождают только элементы, но *физические* источники информации – символы или элементы.

Размер алфавита таблицы ASCII равен 2^8=256, а Unicode – 2^{16}=65 536. Это две самые распространенные таблицы символов.

Источник данных порождает поток либо содержит блок данных. Вероятности порождения элементов определяются состоянием источника. У источника данных **без памяти** состояние одно, у источника **с памятью** – множество состояний, и вероятности перехода из одного состояния в другое зависят от совокупности предыдущих и последующих (еще не реализованных, в случае потока) состояний.

Можно говорить, что источник без памяти порождает *"элементы"*, а источник данных с памятью – *"слова"*, поскольку во втором случае

- учет значений соседних элементов (**контекста**) улучшает сжатие, т. е. имеет смысл трактовать данные как слова;
- поток данных выглядит как поток слов.

В первом же случае имеем дело с перестановкой элементов и рассматривать данные как слова нет смысла.

Кавычки показывают, что это условные названия способов интерпретации входных данных: *"слова"*, *"элементы"*, *"биты"*.

По традиции бинарный источник без памяти называют обычно **источником Бернулли**, а важнейшим частным случаем источника данных с памятью является **источник Маркова** (*N*-го порядка): состояние на i-м шаге зависит от состояний на N предыдущих шагах: *i*-1, *i*-2,..., *i*-N.

Третья важнейшая применяемая при сжатии данных математическая модель – **аналоговый сигнал**:

- данные считаются количественными;
- источник данных считается источником Маркова 1-го порядка.

Если использовать модель "аналоговый сигнал" с $N > 1$, то при малых N эффективность сжатия неизменна или незначительно лучше, но метод существенно сложнее, а при дальнейшем увеличении N эффективность резко уменьшается.

Эффективность сжатия учитывает не только **степень сжатия** (отношение длины несжатых данных к длине соответствующих им сжатых данных),

но и скорости сжатия и разжатия. Часто пользуются обратной к степени сжатия величиной – **коэффициентом сжатия**, определяемым как отношение длины сжатых данных к длине соответствующих им несжатых.

Еще две важные характеристики алгоритма сжатия – объемы памяти, необходимые для сжатия и для разжатия (для хранения данных, создаваемых и/или используемых алгоритмом).

Названия методов

CM (Context Modeling) – контекстное моделирование.

DMC (Dynamic Markov Compression) – динамическое марковское сжатие (является частным случаем CM).

PPM (Prediction by Partial Match) – предсказание по частичному совпадению (является частным случаем CM).

LZ-методы – методы Зива – Лемпела, в том числе LZ77, LZ78, LZH и LZW.

PBS (Parallel Blocks Sorting) – сортировка параллельных блоков.

ST (Sort Transformation) – частичное сортирующее преобразование (является частным случаем PBS).

BWT (Burrows–Wheeler Transform) – преобразование Барроуза – Уилера (является частным случаем ST).

RLE (Run Length Encoding) – кодирование длин повторов.

HUFF (Huffman Coding) – кодирование по методу Хаффмана.

SEM (Separate Exponents and Mantissas) – разделение экспонент и мантисс (представление целых чисел).

UNIC (Universal Coding) – универсальное кодирование (является частным случаем SEM).

ARIC (Arithmetic Coding) – арифметическое кодирование.

RC (Range Coding) – интервальное кодирование (вариант арифметического).

DC (Distance Coding) – кодирование расстояний.

IF (Inversion Frequences) – "обратные частоты" (вариант DC).

MTF (Move To Front) – "сдвиг к вершине", "перемещение стопки книг".

ENUC (Enumerative Coding) – нумерующее кодирование.

FT (Fourier Transform) – преобразование Фурье.

DCT (Discrete Cosine Transform) – дискретное Косинусное Преобразование, ДКП (является частным случаем FT).

DWT (Discrete Wavelet Transform) – дискретное вэйвлетное преобразование, ДВП.

LPC (Linear Prediction Coding) – линейно-предсказывающее кодирование, ЛПК (к нему относятся дельта-кодирование, ADPCM, CELP и MELP).

SC (Subband Coding) – субполосное кодирование.
VQ (Vector Quantization) – векторное квантование.

Карта групп методов сжатия

	Статистические		Преобразующие	
	Поточные	Блочные	Поточные	Блочные
Для *"слов"*, модель "Источник с памятью"	CM, DMC, все PPM	CMBZ, pre-conditioned PPMZ	Все LZ, в т.ч. LZH и LZW	ST, в т. ч. BWT
Для *"элементов"*, модели "Источник без памяти" или "Аналоговый сигнал"	Адаптивный HUFF	Статический HUFF	SEM, VQ, MTF, DC, SC, DWT	DCT, FT, фрактальные методы
Для *"элементов"* или *"битов"*	Адаптивный ARIC	Статический ARIC	RLE, LPC, в т.ч. дельта	PBS, ENUC

Каждая группа (ветвь, семейство) содержит множество методов. Исключением является блочно-ориентированный CM – это относительно мало исследованная область. Авторам не известны другие практические реализации, кроме компрессоров CM Булата Зиганшина и "pre-conditioned PPMZ" Чарльза Блума.

Статистические методы оперируют величинами вероятностей элементов напрямую (или величинами относительных частот[1], что по сути то же самое), а преобразующие используют статистические свойства данных опосредованно. Есть и методы смешанного типа, но их меньше.

Все поточные методы применимы и к блокам, но обратное неверно. Блочные методы неприменимы к потокам, поскольку не могут начать выполнение, пока не задана длина блока, заполненного данными, подлежащими сжатию.

В первой строке "карты групп" – методы для источников с памятью, порождаемые ими данные выгодно трактовать как слова. Однако методы для потоков *"слов"* оперируют, как правило, элементами заданного размера, а не словами, поскольку разбиение потока элементов на слова заранее в общем случае неизвестно.

Во второй строке – методы для источников без памяти и аналоговых сигналов. Эти данные при сжатии невыгодно рассматривать как слова.

[1] ***Относительная частота элемента X** в блоке Z – это количество элементов со значением X, деленное на количество всех элементов в блоке Z: rel_freq(X) = =count(X)/length(Z).*

Не все методы для потоков R-битовых *"элементов"* применимы к *"битам"* (только те, которые в третьей строке "карты").

Очевидно, что невыгодно применять методы для *"элементов"* – к *"словам"* или *"битам"*. Менее очевидно, что невыгодно и обратное: применять методы для потоков *"слов"* к данным без значимых вероятностных взаимосвязей, к *"элементам"* или *"битам"*.

Базовые стратегии сжатия

Базовых стратегий сжатия три:

1. **Преобразование потока** ("Скользящее окно-словарь"). Описание поступающих данных через уже обработанные. Сюда входят LZ-методы для потоков *"слов"*, т. е. когда комбинации поступающих элементов предсказуемы по уже обработанным комбинациям. Преобразование по таблице, RLE, LPC, DC, MTF, VQ, SEM, Subband Coding, Discrete Wavelet Transform – для потоков *"элементов"*, т. е. когда не имеет смысла рассматривать комбинации длиной два и более элемента или запоминать эти комбинации, как в случае Linear Prediction Coding.

   Никаких вероятностей, в отличие от второй стратегии, не вычисляется. В результате преобразования может быть сформировано несколько потоков. Даже если суммарный объем потоков увеличивается, их структура улучшается и последующее сжатие можно осуществить проще, быстрее и лучше.
2. Статистическая стратегия.

   а) **Адаптивная** (поточная). Вычисление вероятностей для поступающих данных на основании статистики по уже обработанным данным. Кодирование с использованием этих вычисленных вероятностей. Семейство PPM-методов – для потоков *"слов"*, адаптивные варианты методов Хаффмана и Шеннона – Фано, арифметического кодирования – для потоков *"элементов"*. В отличие от первого случая, давно собранная статистика имеет тот же вес, что и недавняя, если метод не борется с этим специально, что гораздо сложнее, чем в случае LZ. Кроме того, считаются вероятными все комбинации, даже те, которые еще не встречались в потоке и скорее всего никогда не встретятся.

   б) **Блочная**. Отдельно кодируется и добавляется к сжатому блоку его статистика. Статические варианты методов Хаффмана, Шеннона – Фано и арифметического кодирования – для потоков *"элементов"*. Статическое CM – для *"слов"*.
3. **Преобразование блока**. Входящие данные разбиваются на блоки, которые затем трансформируются целиком, а в случае блока однородных данных лучше брать весь блок, который требуется сжать. Это методы

сортировки блоков ("BlockSorting"-методы: ST, BWT, PBS), а также Fourier Transform, Discrete Cosine Transform, фрактальные преобразования, Enumerative Coding.

Как и при первой стратегии, в результате могут формироваться несколько блоков, а не один. Опять же, даже если суммарная длина блоков не уменьшается, их структура значительно улучшается и последующее сжатие происходит проще, быстрее и лучше.

Резюмируя одним предложением: метод сжатия может быть или статистическим, или трансформирующим и обрабатывать данные либо поточно, либо блоками, причем

- чем больше и однороднее данные и память[1], тем эффективнее блочные методы;
- чем меньше и неоднороднее данные и память, тем эффективнее поточные методы;
- чем сложнее источник, тем сильнее улучшит сжатие оптимальная преобразование;
- чем проще источник, тем эффективнее прямолинейное статистическое решение (математические модели "источник Бернулли" и "источник Маркова").

Сравнение алгоритмов по степени сжатия

Решение задачи сравнения алгоритмов по достигаемой ими степени сжатия требует введения некоторого критерия, так как нельзя сравнивать производительность реализаций на каком-то абстрактном файле. Следует осторожно относиться к теоретическим оценкам, так как они вычисляются с точностью до констант. Величины этих констант на практике могут колебаться в очень больших пределах, особенно при сжатии небольших файлов.

В 1989 г. группа исследователей предложила оценивать коэффициент сжатия с помощью набора файлов, получившего название Calgary Compression Corpus[2] (CalgCC). Набор состоит из 14 файлов, большая часть которых представляет собой тексты на английском языке или языках программирования. Позже к этим 14 файлам были добавлены еще 4 текста на английском

[1] *Однородной назовем память, выделенную одним блоком: никаких особенностей при обращении к ней нет. Если память не однородна, доступ по произвольному адресу (random access) замедляется, как правило, в несколько раз.*

[2] *Bell T. C., Witten I. H. Cleary, J. G. Modeling for text compression // ACM Computer Survey. 1989. Vol. 24, № 4. P. 555–591.*

зыке. Тем не менее обычно оценка производится на наборе из 14 файлов назовем такой набор стандартным CalgCC), а не из 18 (назовем его полным CalgCC).

За последние 10 лет CalgCC сыграл значительную роль в развитии методов сжатия данных без потерь. С одной стороны, он обеспечил исследователей и разработчиков простым критерием качества алгоритма с точки зрения коэффициента сжатия, но, с другой стороны, его использование привело к широкому распространению порочной практики, когда универсальный алгоритм сжатия "настраивался" под файлы набора на этапе разработки и настройки. В итоге прилагательное "универсальный" можно было применять к такому алгоритму лишь с натяжкой. Хотя скорее всего даже "настроенный" алгоритм будет работать достаточно хорошо в реальных условиях, поскольку, несмотря на преобладание текстовой информации, в CalgCC входят файлы различных типов данных.

В таблице приведено описание файлов, составляющих стандартный CalgCC.

Файл	Размер, байт	Описание
Bib	111261	Библиографический список в формате UNIX "refer", ASCII
Book1	768771	Художественная книга: T.Hardy. "Far from the madding crowd", неформатированный текст ASCII. Содержит большое количество OCR-опечаток (неправильно распознанных символов)
Book2	610856	Техническая книга: Witten. "Principles of computer speech", формат UNIX "troff", ASCII
Geo	102400	Геофизические данные, 32-битовые числа
News	377109	Набор сообщений электронных конференций Usenet, формат ASCII
Obj1	21504	Объектный файл для ЭВМ типа VAX
Obj2	246814	Объектный файл для ПК Apple Macintosh
Paper1	53161	Техническая статья: Witten, Neal, Cleary. "Arithmetic coding for data compression", формат UNIX "troff", ASCII
Paper2	82199	Техническая статья: Witten. "Computer (in)security", формат UNIX "troff", ASCII
Pic	513216	Факсимильная двухцветная картинка, 1728x2376 точек, представляет собой две страницы технической книги на французском языке, отсканированные с разрешением 200 точек на дюйм
Progc	39611	Программа на языке Си, ASCII
Progl	71646	Программа на языке Лисп, ASCII

Файл	Размер, байт	Описа...
Progp	49379	Программа на языке Паскаль,
Trans	93695	Расшифровка терминальной сес ра "EMACS", ASCII

Размер стандартного CalgCC составляет 3,141,622 ба...
занимает 3,251,493 байт.

Единственная кодировка текстовой информации в Calg...
поэтому все символы – 8-битовые. Нет ни одного файла с
символами или символами в другой кодировке.

Очевидно, что набор серьезно устарел. Типы входящих
отнюдь не являются типами файлов, обычно подвергаемы
временным пользователем ПК. Поэтому с учетом данного вы...
режения о настройке некоторых алгоритмов под CalgCC к ...
сравнения на этом наборе нужно относиться осторожно. CalgCC п...
лишь часть правды.

Среди конкурентов CalgCC отметим:

- Canterbury Compression Corpus (CantCC), состоящий из дву... -
  стандартного набора "Standard Set" (11 файлов общей длиной 2
  байт) и набора больших файлов "Large Set" (4 файла, 16,005,61...
  предложен той же группой исследователей, что и CalgCC, в ка...
  альтернативы морально устаревшему CalgCC;
- наборы файлов из Archive Comparison Test (ACT): 3 текстовых фа...
  3 исполнимых, 2 звуковых и 8 полноцветных 24-битовых изображе...
  а также вышеописанные CalgCC полный, CantCC стандартный, и по-
  следний (седьмой) набор – это демо-версия игры Worms2 (159 файло...
  общим размером 17 Мб);
- файлы из Compressors Comparison Test Вадима Юкина (VYCCT, 8 фай-
  лов разных типов);
- наборы файлов из тестов Art Of Lossless Data Compression (ARTest):
  - 627 полноцветных изображений, 2066 Мб в 12 наборах;
  - 1231 текстовый файл общей длиной 500 Мб в 6 наборах, в том числ...
    CantCC "Large Set" и 663 русских текста;
  - 5960 разнородных файлов, 382 Мб в 10 наборах.

Среди стандартных наборов тестовых изображений наиболее известны
четыре: JPEG Set, PNG Set, Waterloo Images и Kodak True Color Images.

Все тестовые файлы хранятся на WWW и FTP-серверах Интернета, то...
ные ссылки на них – в описаниях тестов:

ACT: http://compression.ca
ARTest: http://go.to/artest, http://artst.narod.ru

CalgCC: http://links.uwaterloo.ca/calgary.corpus.html
CantCC: http://corpus.canterbury.ac.nz
VYCCT: http:// compression.graphicon.ru./ybs

Замечание о методах, алгоритмах и программах

В ЧЕМ РАЗНИЦА МЕЖДУ МЕТОДОМ И АЛГОРИТМОМ?

Метод – это совокупность действий, а *алгоритм* – конкретная последовательность действий.

1. Алгоритм более подробен, чем метод. Иллюстрация алгоритма – блок-схема, а иллюстрация метода – устройство, компоненты которого работают одновременно.
2. Один и тот же метод могут реализовывать несколько алгоритмов. И чем сложнее метод, тем больше возможно реализаций в виде алгоритмов.
3. По описанию алгоритма можно понять метод, но описание метода даст более полное представление об идеях, реализованных в алгоритме.
4. В методе ошибок быть не может. Но с другой стороны, ошибочным может быть выбор метода. На тех же данных может всегда давать лучший результат другой метод, преимущество которого может казаться не очевидным на первый взгляд. Ошибочным может быть и выбор алгоритма.
5. Разные алгоритмы, реализующие один и тот же метод, могут давать совершенно разные результаты! Покажем это на примере.

ПРИМЕР, ПОКАЗЫВАЮЩИЙ НЕЭКВИВАЛЕНТНОСТЬ АЛГОРИТМОВ МЕТОДА

Метод содержит процедуру Z, поворачивающую двумерное изображение на заданный угол A и добавляющую яркость точкам изображения на величину B, зависящую от расстояния до заданной точки C: $B=B(x-x_0, y-y_0)$. "Выделенная" точка C может лежать как внутри, так и снаружи границ изображения, это дела не меняет. При повороте она получает новые координаты: $x`_0, y`_0$.

Очевидно, что возможны два алгоритма:

- сначала развернуть на заданный угол, затем добавить яркость;
- сначала добавить яркость, затем развернуть.

Результаты работы этих двух алгоритмов могут незначительно отличаться из-за округления результатов вычисления расстояний: $D=((x-x_0)^2+(y-y_0)^2)^{1/2}$, а $D`=((x`-x`_0)^2+(y`-y`_0)^2)^{1/2}$, и в общем случае эти расстояния до и после поворота D и D` не равны.

При извлечении квадратного корня возникают иррациональные числа, т. е. бесконечные дроби. Поэтому, какова бы ни была точность арифме-

тики – 16 знаков или 1024, все равно D и D` придется округлять после какого-то знака, отбрасывая остальные знаки. Увеличение точности приведет лишь к уменьшению вероятности того, что после округления D и D` будут неравны.

Если на основании результата работы процедуры поворота с добавлением яркости вычисляется критерий и в соответствии с его величиной выбирается один из нескольких вариантов дальнейших действий, то результаты работы двух алгоритмов могут отличаться уже не "совсем чуть-чуть", а кардинально.

Например, критерий имеет вид $T_{new}<3\cdot T_{old}$ [1], где T_{old} – суммарная яркость изображения до процедуры Z, а T_{new} – после нее. И если в первом алгоритме $T_{old}/T_{new} = 0.3333$, а во втором 0.3334, то после проверки критерия выполнятся разные ветви алгоритма. Результат неэквивалентности алгоритмов будет хорошо заметен.

Даже если никакого критерия нет, ошибка может накапливаться постепенно, на каждом шаге некоторого цикла.

Таким образом, два алгоритма, реализующих один и тот же метод, могут иногда давать совершенно разные результаты.

Реализация алгоритма – программа

Программа – это реализация, "воплощение" алгоритма на одном из языков программирования. Таким образом, общая схема написания программы сжатия (кодека, т. е. компрессора и декомпрессора), равно как и любой программы вообще, следующая:

1) постановка задачи;
2) выбор **метода**;
3) создание **алгоритма**;
4) написание **программы**;
5) тестирование, оптимизация и настройка.

В этой книге описаны именно методы, но для их иллюстрации приводятся конкретные алгоритмы для одного процессора, иллюстрируемые текстами на языке программирования Си.

[1] *Даже если констант в явном виде нет (например, A/B<C/D, где A, B, C и D – вычисляемые величины), всегда есть умножение на единицу и прибавление нуля.*

РАЗДЕЛ 1
МЕТОДЫ СЖАТИЯ БЕЗ ПОТЕРЬ

В основе всех методов сжатия лежит простая идея: если представлять часто используемые элементы короткими кодами, а редко используемые – длинными кодами, то для хранения блока данных требуется меньший объем памяти, чем если бы все элементы представлялись кодами одинаковой длины. Данный факт известен давно: вспомним, например, азбуку Морзе, в которой часто используемым символам поставлены в соответствие короткие последовательности точек и тире, а редко встречающимся – длинные.

Точная связь между вероятностями и кодами установлена в теореме Шеннона о кодировании источника, которая гласит, что элемент s_i, вероятность появления которого равняется $p(s_i)$, выгоднее всего представлять $-\log_2 p(s_i)$ битами. Если при кодировании размер кодов всегда в точности получается равным $-\log_2 p(s_i)$ битам, то в этом случае длина закодированной последовательности будет минимальной для всех возможных способов кодирования. Если распределение вероятностей $F = \{p(s_i)\}$ неизменно, и вероятности появления элементов независимы, то мы можем найти среднюю длину кодов как среднее взвешенное

$$H = -\sum_i p(s_i) \cdot \log_2 p(s_i) . \qquad (1.1)$$

Это значение также называется *энтропией распределения вероятностей F* или *энтропией источника в заданный момент времени*.

Обычно вероятность появления элемента является условной, т. е. зависит от какого-то события. В этом случае при кодировании очередного элемента s_i распределение вероятностей F принимает одно из возможных значений F_k, т. е. $F = F_k$ и соответственно $H = H_k$. Можно сказать, что источник находится в состоянии k, которому соответствует набор вероятностей $p_k(s_i)$ генерации всех возможных элементов s_i. Поэтому среднюю длину кодов можно вычислить по формуле

$$H = -\sum_k P_k \cdot H_k = -\sum_{k,i} P_k \cdot p_k(s_i) \log_2 p_k(s_i) , \qquad (1.2)$$

где P_k – вероятность того, что F примет k-е значение, или, иначе, вероятность нахождения источника в состоянии k.

Итак, если нам известно распределение вероятностей элементов, генерируемых источником, то мы можем представить данные наиболее компактным образом, при этом средняя длина кодов может быть вычислена по формуле (1.2).

Но в подавляющем большинстве случаев истинная структура источника нам неизвестна, поэтому необходимо строить *модель* источника, которая позволила бы нам в каждой позиции входной последовательности оценить вероятность $p(s_i)$ появления каждого элемента s_i алфавита входной последовательности. В этом случае мы оперируем оценкой $q(s_i)$ вероятности элемента s_i.

Методы сжатия могут строить модель источника адаптивно по мере обработки потока данных или использовать фиксированную модель, созданную на основе априорных представлений о природе типовых данных, требующих сжатия.

Процесс моделирования может быть либо явным, либо скрытым. Вероятности элементов могут использоваться в методе как явным, так и неявным образом. Но всегда сжатие достигается за счет устранения **статистической избыточности** в представлении информации.

Ни один компрессор не может сжать **любой** файл. После обработки **любым** компрессором размер части файлов уменьшится, а оставшейся части – увеличится или останется неизменным. Данный факт можно доказать исходя из неравномерности кодирования, т. е. разной длины используемых кодов, но наиболее прост для понимания следующий комбинаторный аргумент.

Существует 2^n различных файлов длины n бит, где $n = 0, 1, 2, \ldots$ Если размер **каждого** такого файла в результате обработки уменьшается хотя бы на 1 бит, то 2^n исходным файлам будет соответствовать самое большее 2^n-1 различающихся сжатых файлов. Тогда по крайней мере одному архивному файлу будет соответствовать несколько различающихся исходных, и, следовательно, его декодирование без потерь информации **невозможно в принципе**.

Вышесказанное предполагает, что файл отображается в один файл и объем данных указывается в самих данных. Если это не так, то следует учитывать не только суммарный размер архивных файлов, но и объем информации, необходимой для описания нескольких взаимосвязанных архивных файлов и/или размера исходного файла. Общность доказательства при этом сохраняется.

Поэтому невозможен "вечный" архиватор, который способен бесконечное число раз сжимать свои же архивы. "Наилучшим" архиватором является программа копирования, поскольку в этом случае мы можем быть всегда уверены в том, что объем данных в результате обработки не увеличится.

Регулярно появляющиеся заявления о создании алгоритмов сжатия, "обеспечивающих сжатие в десятки раз лучшее, чем у обычных архиваторов", являются либо ложными слухами, порожденными невежеством и погоней за сенсацией, либо рекламой аферистов. В области сжатия без потерь,

т. е. собственно сжатия, такие революции невозможны. Безусловно, степень сжатия компрессорами типичных данных будет неуклонно расти, но улучшения составят в среднем десятки или даже единицы процентов, при этом каждый последующий этап эволюции будет обходиться значительно дороже предыдущего. С другой стороны, в сфере сжатия с потерями, в первую очередь компрессии видеоданных, все еще возможно многократное улучшение сжатия при сохранении субъективной полноты получаемой информации.

Глава 1. Кодирование источников данных без памяти

Разделение мантисс и экспонент

Английское название метода – Separate Exponents and Mantissas (SEM).

Цель – сжатие потока *R*-битовых элементов. В общем случае никаких предположений о свойствах значений элементов не делается, поэтому эту группу методов называют также представлением целых чисел (Representation of Integers).

Основная идея состоит в том, чтобы отдельно описывать порядок значения элемента X_i ("экспоненту" E_i) и отдельно – значащие цифры значения ("мантиссу" M_i).

Значащие цифры начинаются со старшей ненулевой цифры: например, в числе $000001101_2 = 1\cdot2^0+0\cdot2^1+1\cdot2^2+1\cdot2^3+0\cdot2^4+0\cdot\ldots = 13$ это последние 4 цифры. Порядок числа определяется позицией старшей ненулевой цифры в записи числа. Как и при обычной записи в десятичной системе, он равен числу цифр в записи числа без предшествующих незначащих нулей. В данном примере порядок равен четырем. В этом пункте экспонентой называем старшие биты, мантиссой – младшие.

Методы этой группы являются **трансформирующими и поточными** (т. е. могут применяться даже в том случае, когда длина блока с данными не задана). В общем случае скорость работы компрессора (содержащего прямое, "сжимающее" преобразование) равна скорости декомпрессора (реализующего обратное, "разжимающее" преобразование) и зависит только от объема исходных данных: входных при сжатии, выходных при разжатии. Памяти требуется всего лишь несколько байтов.

Поскольку никаких величин вероятностей не вычисляется, никаких таблиц вероятностей не формируется, методы более эффективны в случае простой зависимости вероятности P появления элемента со значением Z от самого значения Z: $P=P(Z)$, и функция $P(Z)$ относительно проста.

Из краткого описания общей идеи видно, что

1) формируется один либо два выходных потока (в зависимости от варианта метода) с кодами меньшего размера;
2) каждый из них может быть либо фиксированного размера (под запись числа отводится C бит, C<R), либо переменной (размер битовой записи зависит от ее содержания);
3) к каждому из них можно итеративно применять метод этого же семейства SEM (чаще всего это полезно применять к потоку с экспонентами).

Прямое преобразование

В самом простом случае под запись экспонент и мантисс отводится фиксированное число битов: E и M. Причем $E \geq 1$, $M \geq 1$, $E+M=R$, где R – число битов в записи исходного числа.

Этот первый из четырех вариантов метода условно обозначим

- Fixed+Fixed (Фиксированная длина экспоненты – Фиксированная длина мантиссы), а остальные три:
- Fixed+Variable (Фиксированная длина экспоненты – Переменная длина мантиссы),
- Variable+Variable (Переменная длина экспоненты – Переменная длина мантиссы) и
- Variable+Fixed (Переменная длина экспоненты – Фиксированная длина мантиссы).

Базовый алгоритм **первого** варианта:

```
#define R 15    //исходные элементы - 15-битовые
#define E 7     //задано число битов под экспоненты
#define M (R-E) //и мантиссы
for( i=0; i<N; i++) {//с каждым элементом исходного блока:
  M[i]=((unsigned)S[i])&((1<<M)-1);  //мантиссы в массив M
  E[i]=((unsigned)S[i]) >> M;        //экспоненты в массив E
}
```

где N – количество элементов во входном блоке;
S[N] – входной блок;
E[N] – блок с экспонентами;
M[N] – блок с мантиссами.

Побитовый логический сдвиг влево на единицу эквивалентен умножению на 2, поэтому $(1<<M) = 2^M$.

Если имеем распределение вероятностей, близкое к "плоскому": $P(Z) \approx \approx \text{const}$, то только первый рассмотренный вариант – Fixed+Fixed – может оказаться полезным: при правильном выборе числа E результат сжатия бло-

ков E[*N*], M[*N*] будет лучше, чем если сжимать исходный блок S[*N*]. Но если вероятности в целом убывают с ростом значений элементов и их распределение близко к такому:

$$P(Z) \geq P(Z+1), \text{ при любом } Z, \qquad (1.3)$$

то полезны два варианта с переменным числом битов под мантиссы, т. е. схемы Fixed+Variable и Variable+Variable.

Если справедливо (1.3), кодирование таких чисел называется *универсальным* (Universal Coding of Integers).

Алгоритм **второго** варианта (Fixed+Variable):

```
#define R 15 // исходные элементы - 15-битовые
#define E 4  // 4 бита под экспоненты, так как 23 ≤ R < 24
             // S[i] - беззнаковые числа
for(i=0; i<N; i++) { // с каждым элементом исходного блока:
  j=0;
  while (S[i]>=1<<j) j++; // найдем такое j, что S[i]<(1<<j)
  E[i]=j;                 // запишем j, т. е. порядок
                          // числа S[i],в массив E
  if (j>1)                // если j>1,
  WriteBits(Output,S[i],j-1);// запишем (j-1) младших бит
                    //числа S[i] в битовый блок с мантиссами
}
```

Поскольку (двоичный) порядок числа сохраняем в массиве с экспонентами, первую значащую цифру (в двоичной записи в случае беззнакового числа это может быть только "1") в битовый блок с мантиссами записывать не нужно.

Упражнение. Составьте таблицу из 16 строк и двух столбцов: в левом – число битов, необходимых для записи чисел из диапазона D, справа – соответствующий диапазон D.

Третий вариант (Variable+Variable) будет отличаться лишь тем, что вместо

```
E[i]=j; //запишем j, т. е. порядок числа S[i], в массив E
```

будет

```
WriteBits(Output,1,j+1);
```

В выходной битовый блок Output записываем подряд несколько нулевых битов, количество которых равно значению экспоненты, и в качестве признака окончания экспоненты записываем единичный бит:

```
for (k=j; k>0; k--)
    WriteBit(Output,0); //j бит "0"
WriteBit(Output,1);     //и один бит "1"
```

Такая запись числа N, последовательность из N нулевых бит и одного единичного, называется унарным кодом.

Если исходные элементы – 32-битовые и почти все равны нулю, степень сжатия может доходить до 32:1.

Упражнение. Какой будет степень сжатия блока 16-битовых элементов в случае применения варианта Variable+Variable: 3, 8, 0, 15, 257, 11, 57867, 2, 65, 18?

Последний, **четвертый** вариант (Variable+Fixed), отводящий переменное число битов под экспоненты и фиксированное – под мантиссы, будет рассмотрен чуть ниже в подразд. про коды Райса.

Важное замечание. Если коды переменной длины записываются в один поток, они должны генерироваться так, чтобы любые два кода A и B из группы генерируемых кодов удовлетворяли условию: A не является началом B, B не является началом A. Такие группы кодов называются **префиксными** кодами.

Обратное преобразование

Обратное преобразование не сложнее прямого. В **первом** варианте внутри цикла будет:

```
for( i=0; i<N; i++) {//с каждым элементом исходного блока:
S[i]=(E[i]<<M)+M[i]; //старшие биты в массиве E, младшие - в M
}
```

Во **втором** варианте:

```
for( i=0; i<N; i++) {// с каждым элементом исходного блока:
  j=E[i];             // возьмем j, т. е. порядок
                      // числа S[i],из массива E
  S[i]=1<<(j-1);      // запишем первую единицу в позиции,
                      // определяемой порядком числа
  if (j>1)            // если j>1,
  S[i]+=GetBits(Input,j-1); // возьмем (j-1) младших бит
                      //S[i] из битового блока с мантиссами
}
```

В **третьем** вместо

```
j=E[i]; //возьмем j, т. е. порядок числа S[i],из массива E
```

будет

```
j=0;                                  // j - счетчик числа нулевых
while (GetBit(Input)==0) j++;// битов в битовом блоке Input
```

а дальше выполняются те же действия, что и во втором варианте:

```
S[i]=1<<(j-1);  // запишем первую единицу в позицию,
                // определяемую порядком числа
if (j>1)        // если j>1,
S[i]+=GetBits(Input,j-1);//возьмем (j-1) младших бит S[i]
                         //из битового блока с мантиссами
```

Пути увеличения степени сжатия

В каждом из рассмотренных выше четырех вариантов (Fixed+Fixed, Variable+Variable, Fixed+Variable, Variable+Fixed) можно пробовать улучшать сжатие за счет:

- отказа от "классической" схемы с диапазонами длиной 2^K и границами, выровненными по 2^L (K, L – константы схемы);
- использования априорного знания диапазона допустимых значений исходных элементов;
- применения хорошо исследованных схем кодирования (Элиаса, Райса, Голомба, Фибоначчи).

При сжатии с потерями можно просто ограничивать число битов мантиссы M: сохранять только не более чем M_1 бит мантиссы (а остальные (M[i]–M_1) бит – удалять, если M[i]>M_1).

Коды переменной длины (VARIABLE+VARIABLE)

Гамма- и дельта-коды Элиаса

Эти коды генерируются так:

Диапазон	Гамма-коды	Длина кода, бит	Дельта-коды	Длина кода, бит
1	1	1	1	1
2...3	01x	3	010x	4
4...7	001xx	5	011xx	5
8...15	0001xxx	7	00100xxx	8
16...31	00001xxxx	9	00101xxxx	9
32...63	000001xxxxx	11	00110xxxxx	10
64...127	0000001xxxxxx	13	00111xxxxxx	11
128...255	00000001xxxxxxx	15	0001000xxxxxxx	14

и т. д., символами "x" здесь обозначены биты мантиссы без старшей единицы.

Для диапазона [2^K, 2^{K+1}–1] коды формируются следующим образом:

γ-код: 00...(K раз)...001x..(K раз)..x ; длина: 2·K+1 бит;

δ-код: n...(2·L+1 раз)...nx..(K раз)..x ; длина: 2·L+K+1 бит,

где $L = [\log_2(K+1)]$ – целая часть значения логарифма числа $(K+1)$ по основанию 2; n – биты, относящиеся к записи экспоненты δ-кода; их число $2 \cdot L+1$.

Единственное отличие между γ- и δ-кодами состоит в том, что в γ-кодах экспоненты записываются в унарном виде, а в δ-кодах к ним еще раз применяется γ-кодирование.

Видно, что γ-коды первых 15 чисел короче δ-кодов, а γ-коды первых 31 не длиннее δ-кодов. То есть чем неравномернее распределение вероятностей, чем круче возрастает вероятность чисел при приближении их значения к нулю, тем выгоднее γ-коды по сравнению с δ-кодами.

Как соотносятся γ-коды и наш базовый алгоритм третьего варианта (Variable+Variable)? Если к γ-коду слева добавить столбец, состоящий из одной единицы и последовательности нулей, то получим такое соответствие кодов числам:

Диапазон	Гамма-коды	Диапазон	Дельта-коды
0	-	0	1
1	1	1	0 1
2-3	01x	2-3	0 01x
4-7	001xx	4-7	0 001xx
8-15	0001xxx	8-15	0 0001xxx
16-31	00001xxxx	16-31	0 00001xxxx
32-63	000001xxxxx (до добавления)	32-63	0 000001xxxxx (после добавления)

Оно как раз и соответствует базовому алгоритму третьего варианта.

Если еще раз прибавить такой столбец и к значениям чисел прибавить 2, то соответствие примет такой вид:

Диапазон	Гамма-коды
1	1
2	0 1
3	0 0 1
4-5	0 0 01x
6-9	0 0 001xx
10-17	0 0 0001xxx
18-33	0 0 00001xxxx
34-65	0 0 000001xxxxx
	γ(3)

Таким образом, единственный параметр обобщенных $\gamma(X)$-кодов – число кодов *без битов мантиссы*. Традиционный γ-код – это γ(1). У обобщенных δ-кодов два параметра.

Упражнение. Напишите функцию, создающую γ(3)-код задаваемого числа, а затем функцию для δ(3,3)-кода.

Коды Райса и Голомба

Коды Райса и Голомба изначально задаются с одним параметром и выглядят так:

Код Голомба:	m=1	m=2	m=3	m=4	m=5	m=6	m=7	m=8
Код Райса:	k=0	k=1		k=2				k=3
n=1	0	00	00	000	000	000	000	0000
2	10	01	010	001	001	001	0010	0001
3	110	100	011	010	010	0100	0011	0010
4	1110	101	100	011	0110	0101	0100	0011
5	11110	1100	1010	1000	0111	0110	0101	0100
6	111110	1101	1011	1001	1000	0111	0110	0101
7	1111110	11100	1100	1010	1001	1000	0111	0110
8	...	11101	11010	1011	1010	1001	1000	0111
9	...	111100	11011	11000	10110	10100	10010	10000

Алгоритм построения кодов можно понять с помощью следующих двух таблиц:

m=2	m=3	m=4	m=8
0x	00	0xx	0xxx
10x	01x	10xx	10xxx
110x	100	110xx	110xxx
1110x	101x	1110xx	1110xxx
11110x	1100	11110xx	11110xxx
	1101x		
	11100		
	11101x		

Видно, что коды Голомба при $m=2^S$ – это коды Райса с $k = S$, экспоненты записываются в унарном виде, а под мантиссы отведено S бит.

Далее:

m=5	m=6	m=7
00x	00x	000
010	01xx	001x

011x	100x	01xx
100x	101xx	1000
1010	1100x	1001x
1011x	1101xx	101xx
1100x	11100x	11000
	11101xx	11001x
	111100x	1101xx
		111000

Если $m<2^S$, первые m кодов начинаются с 0, вторая группа из m кодов начинается с 10, третья – 110 и т. д. Диапазоны длиной m не выровнены по границам, равным 2^L, как в γ-кодах Элиаса. Экспоненты вычисляются как $e = (n-1)/m$ (деление целочисленное) и записываются в унарной системе счисления: e бит 1 и в конце бит 0. Под мантиссы $r = n-em-1$ отводится либо $(S-1)$, либо S бит.

Очевидно, что к экспонентам, как и в случае с γ-кодами Элиаса, можно применять либо γ(X)-, либо Golomb(m)-кодирование. Аналогично и к экспонентам γ-кода – не только γ(X), но и Golomb(m).

Упражнение. Напишите функцию, создающую γ(3)-Golomb(2)-код задаваемого числа. То есть γ(3), к экспонентам которого применен Golomb(2)-код.

Омега-коды Элиаса и коды Ивэн-Родэ

Английские названия кодов: omega (ω) Elias codes и Even-Rodeh codes соответственно.

Эти коды определены так:

Диапазон	ω-код	Битов	Ивэн-Родэ-код	Битов
1	0	1	00	2
2...3	1x 0	3	01x	3
4...7	10 1xx 0	6	1xx 0	4
8...15	11 1xxx 0	7	100 1xxx 0	8
16...31	10 100 1xxxx 0	11	101 1xxxx 0	9
32...63	10 101 1xxxxx 0	12	110 1xxxxx 0	10
64...127	10 110 1xxxxxx 0	13	111 1xxxxxx 0	11
128...255	10 111 1xxxxxxx 0	14	100 1000 1xxxxxxx0	16

И те и другие состоят из последовательности групп длиной $L_1, L_2, L_3, \ldots, L_m$, начинающихся с бита 1. Конец последовательности задается битом 0. Длина каждой следующей $(n+1)$-й группы задается значением битов предыдущей n-й группы. Значение битов последней группы является итоговым значением всего кода, т. е. всей последовательности групп. Иначе говоря, все первые $m-1$ групп служат лишь для указания длины последней группы.

В ω-кодах Элиаса длина первой группы – 2 бита и далее длина следующей группы равна значению предыдущей плюс один. *Первое значение* задано отдельно.

В Ивэн-Родэ-кодах длина первой группы – 3 бита и далее длина каждой следующей группы равна значению предыдущей. *Первые 3 значения* заданы особым образом.

При кодировании формируется сначала последняя группа, затем предпоследняя и т. д., пока процесс не будет завершен.

При декодировании, наоборот, сначала считывается первая группа, по значению ее битов определяется длина следующей группы (если первая группа начинается с единицы) или итоговое значение кода (если группа начинается с нуля).

Упражнение. Как будут выглядеть коды ω Элиаса и Ивэн-Родэ для чисел из диапазонов 256...511 и 512...1023 ?

Старт-шаг-стоп (start-step-stop) коды

Старт-шаг-стоп-коды задаются тремя параметрами: (i,j,k).

Экспонента может занимать 1,2,3,...,m–1,m бит, а мантисса – i, i+j, i+2j, i+3j, …, k.

Если (i,j,k)=(3,2,11), то коды выглядят так:

Диапазон	(3,2,11)-код	Длина кода, бит
1...8	0xxx	4
9...40	10xxxxx	7
41...168	110xxxxxxx	10
169...680	1110xxxxxxxxx	13
681...2728	1111xxxxxxxxxxx	15

Таким образом, это соответствие можно использовать для чисел из диапазона [1,2728]. Экспонента записывается в унарной системе счисления: конец поля экспоненты указывается с помощью нуля. Для пятой группы, имеющей максимальную длину мантиссы – 11 бит, разделяющий нуль не нужен, так как вне зависимости от его наличия декодирование однозначно.

Если максимальное значение кодируемых чисел не задано, то и третий параметр не задается. Такие коды называются Старт-Шаг-кодами (Start-Step-codes).

Коды Фибоначчи

Самые интересные, нетривиальные коды. Исходное число N раскладывается в сумму чисел Фибоначчи f_i (f_1=1, f_2=2, f_i=f_{i-1}+f_{i-2}). Известно, что любое число **однозначно** представимо в виде суммы чисел Фибоначчи. Поэтому

можно построить код числа как последовательность битов, каждый из которых указывает на факт наличия в N определенного числа Фибоначчи.

Заметим также, что если в N есть f_i, то в нем не может быть f_{i+1}. Поэтому если единичное значение бита указывает на использование какого-то числа f_i, то мы можем обозначать конец записи текущего кода и начало следующего последовательностью из двух единиц:

f_i:	1	2	3	5	8	13	21	34	55
n=1	1	(1)							
2	0	1	(1)						
3	0	0	1	(1)					
4	1	0	1	(1)					
5	0	0	0	1	(1)				
6	1	0	0	1	(1)				
7	0	1	0	1	(1)				
8	0	0	0	0	1	(1)			
...	...	...	...	...	...	...			
12	1	0	1	0	1	(1)			
13	0	0	0	0	0	1	(1)		
...	...	...	...	...	...	...	...		
20	0	1	0	1	0	1	(1)		
21	0	0	0	0	0	0	1	(1)	
...	...	...	...	...	...	...	...	...	
27	1	0	0	1	0	0	1	(1)	

Как и в случае с унарной записью, нет четкого разделения на мантиссы и экспоненты. Можно считать, что при записи в унарном виде все биты, кроме последнего, экспоненты. А в кодах Фибоначчи наоборот: все биты, кроме двух последних, мантиссы.

Рассмотрим утверждение подробнее.

Если в потоке γ-кодов или кодов Райса будет искажен 1 бит, длина и содержимое остального потока не изменится, если этот бит мантисса (и "испорчен" окажется только один код). Но если "сломанный" бит экспонента, то будет неправильно декодирована значительная часть потока.

Если в потоке унарных кодов изменить один бит, кодов станет либо на один больше, если этот бит экспонента:

...00000001000000001... – было
...00100001000000001... – стало

либо на один код меньше, если этот бит мантисса:

...00000000000000001...

С другой стороны, для потока кодов Фибоначчи кодов станет на один меньше, если "сломавшийся" бит экспонента, т. е. один из двух единичных

битов, обозначающих конец кода, либо на один больше, если и "сбойный" бит стал таким, как бит экспоненты (1), и хотя бы один из двух соседних со сбойным битов экспоненты единичный. В остальных же случаях "сломается" только один код (в некоторых случаях может "сломаться" пара ссседних кодов). Но весь поток, как может быть в случае γ- и Голомб-кодов, никогда[1].

Упражнение. Приведите пример, показывающий, как из-за сбоя в одном бите "сломается" 3 кода Фибоначчи.

Замечание по унарным кодам. *Если, например, мы сжимаем поток элементов со значениями:*

1,3,4,7,9,10,13,15,16,20,25,26,28,30,33...

так называемым методом флагов:

10110010110010110001000011010100l...

то реально здесь имеем два метода: линейно-предсказывающее кодирование (LPC) плюс унарные коды (см. подразд. "Линейно-предсказывающее кодирование").

Коды фиксированной длины (fixed+fixed)

Несмотря на недостатки с точки зрения сравнительно низкой степени сжатия, коды фиксированной длины широко используются на практике. Положительными особенностями данного варианта SEM являются:

- высокая скорость кодирования;
- поток закодированных данных обладает строгой регулярностью, что облегчает дальнейшее сжатие данных в случае использования сложных многопроходных алгоритмов обработки.

Fixed+Fixed – это самый простой, самый подходящий вариант для последующей сортировки параллельных блоков (PBS).

Предварительная обработка данных с помощью варианта SEM Fixed+Fixed может также улучшить степень сжатия RLE, SC или LPC. Весьма вероятно, что во входных данных разность между экспонентами соседних кодов является в основном D-битовой (т. е. ее можно записать с помощью D бит) и эти D-битовые элементы далее несжимаемы. А после SEM Fixed+Fixed с $M=D-1$ разность между соседними экспонентами в основном равна нулю. И таким образом, от каждого элемента остается в среднем примерно $D-1$ бит, а не D.

Пример: поток элементов, у которых младшие 8 бит меняются хаотично, а старшие 8 – константа. Если делать LPC без SEM, в выходном потоке будет оставаться в среднем по 9 бит от каждого элемента, а после SEM+LPC – по 8 бит.

---

[1] *Поток назовем "сломанным", если возможность восстановить исходные данные утрачена.*

Улучшит сжатие выбор оптимальных размеров экспонент и мантисс. Правильное разделение мантисс и экспонент во многом аналогично выделению шума из аналогового сигнала.

Коды смешанные (fixed+variable)

Поможет улучшить сжатие знание диапазона, особенно если его длина не равна степени двойки: $L<2^{S+1}$, $L=2^S+C$. Тогда при максимальном значении экспоненты под запись мантиссы потребуется на 1 бит меньше в (2^S-C) случаях при $C\geq 2^{S-1}$, на 2 бита меньше в $(2^{S-1}-C)$ случаях при $2^{S-2}\leq C<2^{S-1}$ и т. д.

Например, если диапазон $L=2^{15}$, то при E[i]=15 под запись мантиссы нужно 15 бит, а если $L=2^{14}+2^{13}-7$, то при E[i]=15 достаточно 14 бит, а в семи случаях – 13 бит.

PBS в данном случае уже не столь прост и тривиален, как в предыдущем случае Fixed+Fixed, но все-таки применим, а LPC, RLE или SC для экспонент ничуть не сложнее.

Применимы также методы типа **DAKX** (по имени первоисследователя: D. A. Kopf), отводящие на каждом шаге фиксированное число битов под экспоненту, но помещающие в единый выходной поток "флаги" с информацией об изменении числа битов экспоненты.

Заметим, что в общем случае "флагом" в потоке может быть не только условие на значение одного элемента вида "S[i]=F?" или "f(S[i])=F?", но и условие на значение функции от нескольких последних элементов: "f(S[i],S[i–1],S[i–2],...)=F?". И битов, относящихся исключительно к записи "флага", в потоке может и не быть.

Пути увеличения скорости сжатия и разжатия

Если памяти достаточно, имеет смысл при инициализации алгоритма строить таблицу. Для алгоритма сжатия – содержащую соответствующие E[i] и M[i] по адресу, задаваемому значением S[i]. Для разжатия, наоборот, – содержащую S[i] по адресам, задаваемым значениями E[i] и M[i].

В результате внутренний цикл (если он есть) вида

```
j=0;
while (S[i] >= 1<<j) j++; //найдем такое j, что S[i]< (1<<j)
```

(см. алгоритм сжатия, второй вариант)

преобразуется в вид

```
j=T[S[i]];              //возьмем такое j, что S[i]< (1<<j)
```

Упражнение. Напишите функцию, строящую таблицу T для этого варианта (Fixed+ +Variable).

Очевидно, что размер таблицы T будет равен размеру диапазона 2^R возможных значений элементов входного потока S. Поэтому компромисс между объемом памяти и скоростью работы может находиться где-то посередине: например если $L=2^{16}$, то вместо

```
j=T[S[i]];              //возьмем такое j, что S[i]< (1<<j)
```

можно реализовать вычисление *j* так:

```
j=T[S[i]>>8]+8;       // j>8, если S[i]>=256
if (j==8) j=T[S[i]]; // j=8, если S[i]<256
```

На несколько операций дольше по сравнению с первым рассмотренным вариантом использования таблицы, но и размер T уже не 2^{16}=65536, а 2^8=256.

Характеристики методов семейства SEM:
Степень сжатия: до R:1, где R – размер исходных элементов.
Типы данных: любые данные, лучше количественные.
Симметричность по скорости: в общем случае 1:1.
Характерные особенности: традиционно используется для эффективного кодирования источников без памяти.

Канонический алгоритм Хаффмана

Один из классических алгоритмов, известных с 60-х гг. Использует только частоту появления одинаковых байтов во входном блоке данных. Ставит в соответствие символам входного потока, которые встречаются чаще, цепочку битов меньшей длины. И напротив, встречающимся редко – цепочку большей длины. Для сбора статистики требует двух проходов по входному блоку (также существуют однопроходные адаптивные варианты алгоритма).

Для начала введем несколько определений.

Определение. Пусть задан алфавит $\Psi=\{a_1, ..., a_r\}$, состоящий из конечного числа букв. Конечную последовательность символов из Ψ

$$A = a_{i_1} a_{i_2} ... a_{i_n}$$

будем называть *словом* в алфавите Ψ, а число n – *длиной слова A*. Длина слова обозначается как *l(A)*.

Пусть задан алфавит Ω, $\Omega=\{b_1, ..., b_q\}$. Через *B* обозначим слово в алфавите Ω, и через $S(\Omega)$ – множество всех непустых слов в алфавите Ω.

Пусть $S=S(\Psi)$ – множество всех непустых слов в алфавите Ψ и S' – некоторое подмножество множества *S*. Пусть также задано отображение *F*, которое каждому слову *A*, $A \in S(\Psi)$ ставит в соответствие слово

$$B=F(A),\ B\in S(\Omega).$$

Слово B будем называть *кодом сообщения* A, а переход от слова A к его коду – *кодированием*.

Определение. Рассмотрим соответствие между буквами алфавита Ψ и некоторыми словами алфавита Ω:

$$a_1 - B_1,$$
$$a_2 - B_2,$$
$$\ldots$$
$$a_r - B_r.$$

Это соответствие называют *схемой* и обозначают через Σ. Оно определяет кодирование следующим образом: каждому слову $A = a_{i_1} a_{i_2} \ldots a_{i_n}$ из $S'(\Omega)=S(\Omega)$ ставится в соответствие слово $B = B_{i_1} B_{i_2} \ldots B_{i_n}$, называемое *кодом слова A*. Слова $B_1 \ldots B_r$ называются *элементарными кодами*. Данный вид кодирования называют *алфавитным кодированием*.

Определение. Пусть слово B имеет вид

$$B=B'B''.$$

Тогда слово B' называется *началом* или *префиксом слова* B, а B'' – *концом слова* B. При этом пустое слово Λ и само слово B считаются началами и концами слова B.

Определение. Схема Σ *обладает свойством префикса*, если для любых i и j ($1\le i, j\le r$, $i\ne j$) слово B_i не является префиксом слова B_j.

Теорема 1. *Если схема* Σ *обладает свойством префикса, то алфавитное кодирование будет взаимно-однозначным.*

Доказательство теоремы можно найти в [4].

Предположим, что задан алфавит $\Psi=\{a_1, \ldots, a_r\}$ ($r>1$) и набор вероятностей $p_1, \ldots, p_r$ $\left(\sum_{i=1}^{r} p_i = 1\right)$ появления символов $a_1, \ldots, a_r$. Пусть, далее, задан алфавит Ω, $\Omega=\{b_1, \ldots, b_q\}$ ($q>1$). Тогда можно построить целый ряд схем Σ алфавитного кодирования

$$a_1 - B_1,$$
$$\ldots$$
$$a_r - B_r,$$

обладающих свойством взаимной однозначности.

Для каждой схемы можно ввести среднюю длину $l_{ср}$, определяемую как математическое ожидание длины элементарного кода:

$$l_{ср} = \sum_{i=1}^{r} l_i p_i, \qquad l_i = l(B_i) - \text{длины слов.}$$

Длина $l_{ср}$ показывает, во сколько раз увеличивается средняя длина слова при кодировании с помощью схемы Σ.

Можно показать, что $l_{ср}$ достигает величины своего минимума l_* на некоторой Σ и определяется как

$$l_* = \min_{\Sigma} l_{ср}^{\Sigma}.$$

Определение. Коды, определяемые схемой Σ с $l_{ср} = l_*$, называются *кодами с минимальной избыточностью* или *кодами Хаффмана.*

Коды с минимальной избыточностью дают в среднем минимальное увеличение длин слов при соответствующем кодировании.

В нашем случае алфавит $\Psi=\{a_1, ..., a_r\}$ задает символы входного потока, а алфавит $\Omega=\{0,1\}$, т. е. состоит всего из нуля и единицы.

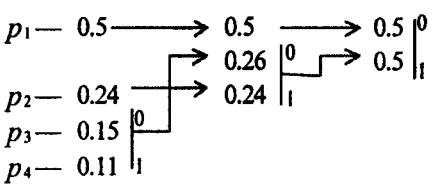
Алгоритм построения схемы Σ можно представить следующим образом:

Шаг 1. Упорядочиваем все буквы входного алфавита в порядке убывания вероятности. Считаем все соответствующие слова B_i из алфавита $\Omega=\{0,1\}$ пустыми.

Шаг 2. Объединяем два символа $a_{i_{r-1}}$ и a_{i_r} с наименьшими вероятностями $p_{i_{r-1}}$ и p_{i_r} в псевдосимвол $a'\{a_{i_{r-1}} a_{i_r}\}$ с вероятностью $p_{i_{r-1}}+p_{i_r}$. Дописываем 0 в начало слова $B_{i_{r-1}}$ ($B_{i_{r-1}}=0B_{i_{r-1}}$) и 1 в начало слова B_{i_r} ($B_{i_r}=1B_{i_r}$).

Шаг 3. Удаляем из списка упорядоченных символов $a_{i_{r-1}}$ и a_{i_r}, заносим туда псевдосимвол $a'\{a_{i_{r-1}}a_{i_r}\}$. Проводим шаг 2, добавляя при необходимости 1 или 0 для всех слов B_i, соответствующих псевдосимволам, до тех пор пока в списке не останется 1 псевдосимвол.

Пример. Пусть у нас есть 4 буквы в алфавите $\Psi=\{a_1,..., a_4\}$ ($r=4$), $p_1=0.5$, $p_2=0.24$, $p_3=0.15$, $p_4=0.11$ $\left(\sum_{i=1}^{4} p_i = 1\right)$. Тогда процесс построения схемы можно представить так:



Производя действия, соответствующие 2-му шагу, мы получаем псевдосимвол с вероятностью 0.26 (и приписываем 0 и 1 соответствующим словам). Повторяя же эти действия для измененного списка, мы получаем псевдосимвол с вероятностью 0.5. И наконец, на последнем этапе мы получаем суммарную вероятность 1.0.

Для того чтобы восстановить кодирующие слова, нам надо пройти по стрелкам от начальных символов к концу получившегося бинарного дерева. Так, для символа с вероятностью p_4 получим B_4=101, для p_3 получим B_3=100, для p_2 получим B_2=11, для p_1 получим B_1=0. Что соответствует схеме:

$$a_1 - 0$$
$$a_2 - 11$$
$$a_3 - 100$$
$$a_4 - 101$$

Эта схема представляет собой префиксный код, являющийся кодом Хаффмана. Самый часто встречающийся в потоке символ a_1 мы будем кодировать самым коротким словом 0, а самый редко встречающийся a_4 – длинным словом 101.

Для последовательности из 100 символов, в которой символ a_1 встретится 50 раз, символ a_2 – 24 раза, символ a_3 – 15 раз, а символ a_4 – 11 раз, данный код позволит получить последовательность из 176 бит ($100 \cdot l_{cp} = \sum_{i=1}^{4} p_i l_i$). То есть в среднем мы потратим 1.76 бита на символ потока.

Доказательства теоремы, а также того, что построенная схема действительно задает код Хаффмана, заинтересованный читатель найдет в [4].

Как стало понятно из изложенного выше, канонический алгоритм Хаффмана требует помещения в файл со сжатыми данными таблицы соответствия кодируемых символов и кодирующих цепочек.

На практике используются его разновидности. Так, в некоторых случаях резонно либо использовать постоянную таблицу, либо строить ее адаптивно, т. е. в процессе архивации/разархивации. Эти приемы избавляют нас от двух проходов по входному блоку и необходимости хранения таблицы вместе с файлом. Кодирование с фиксированной таблицей применяется в качестве последнего этапа архивации в JPEG и в алгоритме CCITT Group, рассмотренных в разд. 2.

Характеристики канонического алгоритма Хаффмана:

Степени сжатия: 8, 1.5, 1 (лучшая, средняя, худшая степени).

Симметричность по времени: 2:1 (за счет того, что требует двух проходов по массиву сжимаемых данных).

Характерные особенности: один из немногих алгоритмов, который не увеличивает размера исходных данных в худшем случае (если не считать необходимости хранить таблицу перекодировки вместе с файлом).

Арифметическое сжатие

Классический вариант алгоритма

Сжатие по методу Хаффмана постепенно вытесняется арифметическим сжатием. Свою роль в этом сыграло то, что закончились сроки действия патентов, ограничивающих использование арифметического сжатия. Кроме того, алгоритм Хаффмана приближает относительные частоты появления символов в потоке частотами, кратными степени двойки (например, для символов *a, b, c, d* с вероятностями 1/2, 1/4, 1/8, 1/8 будут использованы коды 0, 10, 110, 111), а арифметическое сжатие дает лучшую степень приближения частоты. По теореме Шеннона наилучшее сжатие в двоичной арифметике мы получим, если будем кодировать символ с относительной частотой f с помощью $-\log_2(f)$ бит.

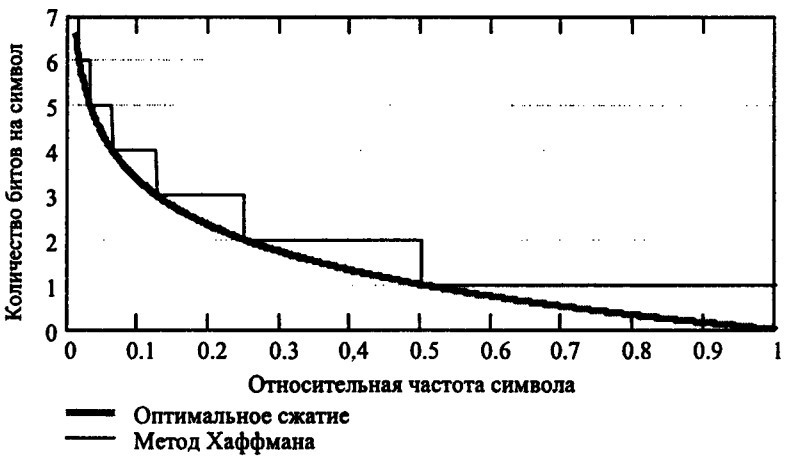


Рис. 1.1. График сравнения оптимального кодирования и кодирования по методу Хаффмана

На графике выше приводится сравнение оптимального кодирования и кодирования по методу Хаффмана. Хорошо видно, что в ситуации, когда относительные частоты не являются степенями двойки, сжатие становится менее эффективным (мы тратим больше битов, чем это необходимо). Например, если у нас два символа *a* и *b* с вероятностями 253/256 и 3/256, то в идеале мы должны потратить на цепочку из 256 байт $-\log_2(253/256)\cdot 253 - \log_2(3/256)\cdot 3 = 23.546$, т. е. **24** бита. При кодировании по Хаффману мы закодируем *a* и *b* как 0 и 1 и нам придется потратить $1\cdot 253+1\cdot 3=$**256** бит, т. е. в 10 раз больше. Рассмотрим алгоритм, дающий результат, близкий к оптимальному.

Арифметическое сжатие – достаточно изящный метод, в основе которого лежит очень простая идея. Мы представляем кодируемый текст в виде дроби, при этом строим дробь таким образом, чтобы наш текст был представлен как можно компактнее. Для примера рассмотрим построение такой дроби на интервале [0, 1) (0 – включается, 1 – нет). Интервал [0, 1) выбран потому, что он удобен для объяснений. Мы разбиваем его на подынтервалы с длинами, равными вероятностям появления символов в потоке. В дальнейшем будем называть их диапазонами соответствующих символов.

Пусть мы сжимаем текст "КОВ.КОРОВА" (что, очевидно, означает "коварная корова"). Распишем вероятности появления каждого символа в тексте (в порядке убывания) и соответствующие этим символам диапазоны:

Символ	Частота	Вероятность	Диапазон
О	3	0.3	[0.0; 0.3)
К	2	0.2	[0.3; 0.5)
В	2	0.2	[0.5; 0.7)
Р	1	0.1	[0.7; 0.8)
А	1	0.1	[0.8; 0.9)
"."	1	0.1	[0.9; 1.0)

Будем считать, что эта таблица известна в компрессоре и декомпрессоре. Кодирование заключается в уменьшении рабочего интервала. Для первого символа в качестве рабочего интервала берется [0, 1). Мы разбиваем его на диапазоны в соответствии с заданными частотами символов (см. таблицу диапазонов). В качестве следующего рабочего интервала берется диапазон, соответствующий текущему кодируемому символу. Его длина пропорциональна вероятности появления этого символа в потоке. Далее считываем следующий символ. В качестве исходного берем рабочий интервал, полученный на предыдущем шаге, и опять разбиваем его в соответствии с таблицей диапазонов. Длина рабочего интервала уменьшается пропорционально вероятности текущего символа, а точка начала сдвигается вправо пропорционально началу диапазона для этого символа. Новый построенный диапазон берется в качестве рабочего и т. д.

Используя исходную таблицу диапазонов, кодируем текст "КОВ.КОРОВА":

Исходный рабочий интервал [0, 1).
Символ "К" [0.3; 0.5) получаем [0.3000; 0.5000).
Символ "О" [0.0; 0.3) получаем [0.3000; 0.3600).
Символ "В" [0.5; 0.7) получаем [0.3300; 0.3420).
Символ "." [0.9; 1.0) получаем [0,3408; 0.3420).

Графический процесс кодирования первых трех символов можно представить так, как на рис. 1.2.

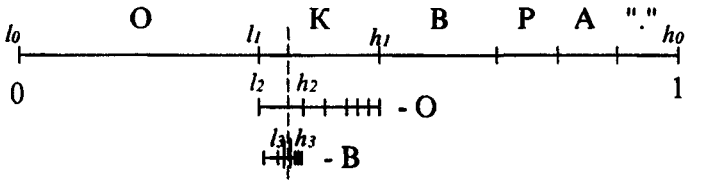


Рис. 1.2. Графический процесс кодирования первых трех символов

Таким образом, окончательная длина интервала равна произведению вероятностей всех встретившихся символов, а его начало зависит от порядка следования символов в потоке. Если обозначить диапазон символа c как [$a[c]$; $b[c]$), а интервал для i-го кодируемого символа потока как [l_i, h_i), то алгоритм сжатия может быть записан как

```
l0=0; h0=1; i=0;
while(not DataFile.EOF()){
   c = DataFile.ReadSymbol(); i++;
   li = li-1 + a[c]·(hi-1 - li-1);
   hi = li-1 + b[c]·(hi-1 - li-1);
};
```

Большой вертикальной чертой на рисунке выше обозначено произвольное число, лежащее в полученном при работе интервале [l_i, h_i). Для последовательности "КОВ.", состоящей из четырех символов, за такое число можно взять 0.341. Этого числа достаточно для восстановления исходной цепочки, если известна исходная таблица диапазонов и длина цепочки.

Рассмотрим работу алгоритма восстановления цепочки. Каждый следующий интервал вложен в предыдущий. Это означает, что если есть число 0.341, то первым символом в цепочке может быть только "К", поскольку только его диапазон включает это число. В качестве интервала берется диапазон "К" – [0.3; 0.5) и в нем находится диапазон [$a[c]$; $b[c]$), включающий 0.341. Перебором всех возможных символов по приведенной выше таблице находим, что только интервал [0.3; 0.36), соответствующий диапазону для "О", включает число 0.341. Этот интервал выбирается в качестве следующего рабочего и т. д. Алгоритм декомпрессии можно записать так:

```
l0=0; h0=1; value=File.Code();
for(i=1; i<=File.DataLength(); i++){
   for(для всех cj){
      li = li-1 + a[cj]·(hi-1 - li-1);
      hi = li-1 + b[cj]·(hi-1 - li-1);
      if ((li <= value) && (value < hi)) break;
   };
   DataFile.WriteSymbol(cj);
};
```

где value – прочитанное из потока число (дробь), а c – записываемые в выходной поток распаковываемые символы. При использовании алфавита из 256 символов c_j внутренний цикл выполняется достаточно долго, однако его можно ускорить. Заметим, что поскольку $b[c_{j+1}]=a[c_j]$ (см. приведенную выше таблицу диапазонов), то l_i для c_{j+1} равно h_i для c_j, а последовательность h_i для c_j строго возрастает с ростом j. То есть количество операций во внутреннем цикле можно сократить вдвое, поскольку достаточно проверять только одну границу интервала. Также если у нас мало символов, то, отсортировав их в порядке уменьшения вероятностей, мы сокращаем число итераций цикла и таким образом ускоряем работу декомпрессора. Первыми будут проверяться символы с наибольшей вероятностью, например в нашем примере мы с вероятностью 1/2 будем выходить из цикла уже на втором символе из шести. Если число символов велико, существуют другие эффективные методы ускорения поиска символов (например, бинарный поиск).

Хотя приведенный выше алгоритм вполне работоспособен, он будет работать медленно по сравнению с алгоритмом, оперирующим двоичными дробями. Двоичная дробь задается как $0.a_1a_2a_3...a_i = a_1\cdot1/2 + a_2\cdot1/4 + a_3\cdot1/8+... a_i\cdot1/2^i$. Таким образом, при сжатии нам необходимо дописывать в дробь дополнительные знаки до тех пор, пока получившееся число не попадет в интервал, соответствующий закодированной цепочке. Получившееся число полностью задает закодированную цепочку при аналогичном алгоритме декодирования (рис. 1.3).

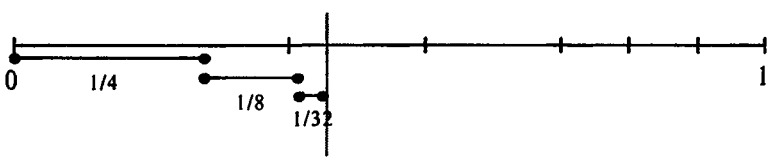


Рис. 1.3

Упражнение. Восстановить исходный текст из закодированной цепочки в 2 бита, равных "11" (число $0.11_{(2)}=0.75_{(10)}$), используя приведенную выше таблицу диапазонов, если известно, что длина текста 10 символов.

Интересной особенностью арифметического кодирования является способность сильно сжимать отдельные длинные цепочки. Например, **1 бит "1"** (двоичное число "0.1") для нашей таблицы интервалов однозначно задает цепочку "BOOOOOOOOO..." **произвольной длины** (например, 1000000000 символов). То есть если наш файл заканчивается одинаковыми символами, например массивом нулей, то этот файл может быть сжат с весьма впечатляющей степенью сжатия. Очевидно, что длину исходного

файла при этом следует передавать декомпрессору явным образом перед сжатыми данными, как это делалось в приведенных выше примерах.

Приведенный выше алгоритм может сжимать только достаточно короткие цепочки из-за ограничений разрядности всех переменных. Чтобы избежать этих ограничений, реальный алгоритм работает с целыми числами и оперирует с дробями, числитель и знаменатель которых являются целыми числами (например, знаменатель равен 10000h = 65536). При этом с потерей точности можно бороться, отслеживая сближение l_i и h_i и умножая числитель и знаменатель представляющей их дроби на какое-то число (удобно на 2). С переполнением сверху можно бороться, записывая старшие биты в l_i и h_i в файл тогда, когда они перестают меняться (т. е. реально уже не участвуют в дальнейшем уточнении интервала). Перепишем таблицу диапазонов с учетом сказанного выше:

J	Символ (c_i)	Накопленная частота	b[c_i]
0	–	–	0
1	O	3	3
2	K	2	5
3	B	2	7
4	P	1	8
5	A	1	9
6	"."	1	10

Теперь запишем алгоритм сжатия, используя целочисленные операции. Минимизация потерь по точности достигается благодаря тому, что длина целочисленного интервала всегда не менее половины всего интервала. Когда l_i или h_i одновременно находятся в верхней или нижней половине (Half) интервала, то мы просто записываем их одинаковые верхние биты в выходной поток, вдвое увеличивая интервал. Если l_i и h_i приближаются к середине интервала, оставаясь по разные стороны от его середины, то мы также вдвое увеличиваем интервал, записывая биты "условно". "Условно" означает, что реально эти биты выводятся в выходной файл позднее, когда становится известно их значение. Процедура изменения значений l_i и h называется *нормализацией*, а вывод соответствующих битов – *переносом*. Знаменатель дроби в приведенном ниже алгоритме будет равен 10000h = = 65536, т. е. максимальное значение h_0=65535.

```
l0=0; h0=65535; i=0; delitel= b[c_last]; // delitel=10
First_qtr = (h0+1)/4;    // = 16384
Half = First_qtr*2;      // = 32768
Third_qtr = First_qtr*3;// = 49152
bits_to_follow =0;       // Сколько битов сбрасывать
```

```
while(not DataFile.EOF()) {
      c = DataFile.ReadSymbol();  // Читаем символ
      j = IndexForSymbol(c); i++; // Находим его индекс
      l_i = l_i-1 + b[j-1]*(h_i-1 - l_i-1 + 1)/delitel;
      h_i = l_i-1 + b[j  ]*(h_i-1 - l_i-1 + 1)/delitel - 1;
      for(;;) {              // Обрабатываем варианты
          if(h_i < Half)     // переполнения
              BitsPlusFollow(0);
          else if(l_i >= Half) {
              BitsPlusFollow(1);
              l_i-= Half; h_i-= Half;
          }
          else if((l_i >= First_qtr)&&(h_i < Third_qtr)){
              bits_to_follow++;
              l_i-= First_qtr; h_i-= First_qtr;
          } else break;
          l_i+=l_i; h_i+= h_i+1;
      }
}
          // Процедура переноса найденных битов в файл
void BitsPlusFollow(int bit)
{
      CompressedFile.WriteBit(bit);
      for(; bits_to_follow > 0; bits_to_follow--)
          CompressedFile.WriteBit(!bit);
}
```

Упражнение. Покажите, что BitsPlusFollow работает правильно и записывает в выходной файл значения, попадающие внутрь рабочего интервала.

I	Символ (c_j)	l_i	h_i	Нормализованный l_i	Нормализованный h_i	Результат
0		0	65535			
1	K	19660	32767	13104	65535	01
2	O	13104	28832	26208	57665	010
3	B	41937	48227	7816	58143	010101
4	.	53111	58143	15836	35967	01010111
5	K	21875	25901	21964	38071	0101011101
6	O	21964	26795	22320	41647	010101110101

Упражнение. Выведите самостоятельно, какими битами нужно закончить сжатый файл, чтобы при декомпрессии были корректно получены последние 2– 3 символа цепочки.

На символ с меньшей вероятностью у нас тратится в целом большее число битов, чем на символ с большой вероятностью. Алгоритм декомпрессии в целочисленной арифметике можно записать так:

```
l0=0; h0=65535; delitel= b[clast];
First_qtr = (h0+1)/4;       // = 16384
Half = First_qtr*2;         // = 32768
Third_qtr = First_qtr*3;    // = 49152

value=CompressedFile.Read16Bit();
for(i=1; i< CompressedFile.DataLength(); i++){
   freq=((value-li-1+1)*delitel-1)/(hi-1 - li-1 + 1);
   for(j=1; b[j]<=freq; j++);    // Поиск символа

   li = li-1 + b[j-1]*(hi-1 - li-1 + 1)/delitel;
   hi = li-1 + b[j  ]*(hi-1 - li-1 + 1)/delitel - 1;

       for(;;) {             // Обрабатываем варианты
           if(hi < Half)     // переполнения
               ; // Ничего
           else if(li >= Half) {
               li-= Half; hi-= Half; value-= Half;
           }
           else if((li >= First_qtr)&&(hi < Third_qtr)){
               li-= First_qtr; hi-= First_qtr;
               value-= First_qtr;
           } else break;
           li+=li; hi+= hi+1;
           value+=value+CompressedFile.ReadBit();
       }
   DataFile.WriteSymbol(c);
};
```

Упражнение. Предложите примеры последовательностей, сжимаемых алгоритмом с максимальным и минимальным коэффициентом.

Как видно, с неточностями арифметики мы боремся, выполняя отдельные операции над l_i и h_i синхронно в компрессоре и декомпрессоре.

Незначительные потери точности (доли процента при достаточно большом файле) и, соответственно, уменьшение степени сжатия по сравнению с идеальным алгоритмом происходят во время операции деления, при округлении относительных частот до целого, при записи последних битов в файл. Алгоритм можно ускорить, если представлять относительные частоты так, чтобы делитель был степенью двойки (т. е. заменить деление операцией побитового сдвига).

Для того чтобы оценить степень сжатия арифметическим алгоритмом конкретной строки, нужно найти минимальное число N, такое, чтобы длина рабочего интервала при сжатии последнего символа цепочки была бы меньше $1/2^N$. Этот критерий означает, что внутри нашего интервала заведомо найдется хотя бы одно число, в двоичном представлении которого после N-го знака будут только 0. Длину же интервала посчитать просто, поскольку она равна произведению вероятностей всех символов.

Рассмотрим приводившийся ранее пример строки из двух символов a и b с вероятностями 253/256 и 3/256. Длина последнего рабочего интервала для цепочки из 256 символов a и b с указанными вероятностями равна:

$$h_{256} - l_{256} = \left(\frac{253}{256}\right)^{253} \cdot \left(\frac{3}{256}\right)^{3} = \frac{253^{253} \cdot 9}{2^{2048}} \approx 8.15501 \cdot 10^{-8}.$$

Легко подсчитать, что искомое N=24 ($1/2^{24} \approx 5.96 \cdot 10^{-8}$), поскольку 23 дает слишком большой интервал (в 2 раза шире), а 25 не является *минимальным* числом, удовлетворяющим критерию. Выше было показано, что алгоритм Хаффмана кодирует данную цепочку в 256 бит. То есть для рассмотренного примера арифметический алгоритм дает десятикратное преимущество перед алгоритмом Хаффмана и требует менее 0.1 бита на символ.

Упражнение. Подсчитайте оценку степени сжатия для строки "КОВ.КОРОВА".

Следует сказать пару слов об адаптивном алгоритме арифметического сжатия. Его идея заключается в том, чтобы перестраивать таблицу вероятностей $b[j]$ по ходу упаковки и распаковки непосредственно при получении очередного символа. Такой алгоритм не требует сохранения значений вероятностей символов в выходной файл и, как правило, дает бóльшую степень сжатия. Так, например, файл вида $a^{1000}b^{1000}c^{1000}d^{1000}$ (где степень означает число повторов данного символа) адаптивный алгоритм сможет сжать эффективнее, чем потратив 2 бита на символ. Приведенный выше алгоритм достаточно просто превращается в адаптивный. Ранее мы сохраняли таблицу диапазонов в файл, а теперь мы считаем прямо по ходу работы компрессора и декомпрессора, пересчитываем относительные частоты, корректируя в соответствии с ними таблицу диапазонов. Важно, чтобы изменения в таблице происходили в компрессоре и декомпрессоре синхронно, т. е., например, после *кодирования* цепочки длины 100 таблица диапазонов должна быть точно такой же, как и после *декодирования* цепочки длины 100. Это условие легко выполнить, если изменять таблицу *после* кодирования и декодирования очередного символа. Подробнее об адаптивных алгоритмах смотрите в гл. 4.

Характеристики арифметического алгоритма:

Лучшая и худшая степень сжатия: лучшая > 8 (возможно кодирование менее бита на символ), худшая – 1.

Плюсы алгоритма: обеспечивает лучшую степень сжатия, чем алгоритм Хаффмана (на типичных данных на 1–10%).

Характерные особенности: так же как кодирование по Хаффману, не увеличивает размера исходных данных в худшем случае.

Интервальное кодирование

В отличие от классического алгоритма, интервальное кодирование предполагает, что мы имеем дело с целыми дискретными величинами, которые могут принимать ограниченное число значений. Как уже было отмечено, начальный интервал в целочисленной арифметике записывается в виде $[0,N)$ или $[0,N-1]$, где N – число возможных значений переменной, используемой для хранения границ интервала.

Чтобы наиболее эффективно сжать данные, мы должны закодировать каждый символ s посредством $-\log_2(f_s)$ бит, где f_s – частота символа s. Конечно, на практике такая точность недостижима, но мы можем для каждого символа s отвести в интервале диапазон значений $[N(F_s),N(F_s+f_s))$, где F_s – накопленная частота символов, предшествующих символу s в алфавите, $N(f)$ – значение, соответствующее частоте f в интервале из N возможных значений. И чем больше будет $N(f_s)$, тем точнее будет представлен символ s в интервале. Следует отметить, что для всех символов алфавита должно соблюдаться неравенство $f_s>0$.

Задачу увеличения размера интервала выполняет процедура, называемая *нормализацией*. Практика показывает, что можно отложить выполнение нормализации на некоторое время, пока размер интервала обеспечивает приемлемую точность. Микаэль Шиндлер (Michael Schindler) предложил в работе [3] рассматривать выходной поток как последовательность байтов, а не битов, что избавило от битовых операций и позволило производить нормализацию заметно реже. И чаще всего нормализация обходится без выполнения переноса, возникающего при сложении значений нижней границы интервала и размера интервала. В результате скорость кодирования возросла в полтора раза при крайне незначительной потере в степени сжатия (размер сжатого файла обычно увеличивается лишь на сотые доли процента).

Выходные данные арифметического кодера можно представить в виде четырех составляющих:

1. Составляющая, записанная в выходной файл, которая уже не может измениться.

2. Один элемент (бит или байт), который может быть изменен переносом, если последний возникнет при сложении значений нижней границы интервала и размера интервала.
3. Блок элементов, имеющих максимальное значение, через которые по цепочке может пройти перенос.
4. Текущее состояние кодера, представленное нижней границей интервала.

Например:

Составляющая, записанная в файл	Элемент, который может быть изменен переносом	Блок элементов, имеющих максимальное значение	Нижняя граница интервала
D7 03 56 E4	3A	FF FF	35 38 B1
			+
Размер интервала			EA 12 1A
			=
Перенос	3B	00 00	1F 4A CB

При выполнении нормализации возможны следующие действия:

1. Если интервал имеет приемлемый для обеспечения заданной точности размер, нормализация не нужна.
2. Если при сложении значений нижней границы интервала и размера интервала не возникает переноса, составляющие 2 и 3 могут быть записаны в выходной файл без изменений.
3. В случае возникновения переноса он выполняется в составляющих 2 и 3, после чего они также записываются в выходной файл.
4. Если элемент, претендующий на запись в выходной файл, имеет максимальное значение (в случае бита – 1, в случае байта – 0xFF), то он может повлиять на предыдущий при возникновении переноса. Поэтому этот элемент записывается в блок, соответствующий третьей составляющей.

Ниже приведен исходный текст алгоритма, реализующего нормализацию для интервального кодирования[3].

```
// Максимальное значение, которое может принимать
// переменная. Для 32-разрядной арифметики
// CODEBITS = 31. Один бит отводится для
// определения факта переноса.
#define TOP (1<<CODEBITS)

// Минимальное значение, которое может принимать
// размер интервала. Если значение меньше,
// требуется нормализация
#define BOTTOM (TOP>>8)
```

```
// На сколько битов надо сдвинуть значение нижней
// границы интервала, чтобы остался 1 байт
#define SHIFTBITS (CODEBITS-8)

// Если для хранения значений используется 31 бит,
// каждый символ сдвинут на 1 байт вправо
// в выходном потоке и при декодировании приходится
// его считывать в 2 этапа.
#define EXTRABITS ((CODEBITS-1)%8+1)

// Используемые глобальные переменные:
// next_char  - символ, который может быть изменен
// переносом (составляющая 2).
// carry_counter - число символов, через которые
// может пройти перенос до символа next_char
// (составляющая 3).
// low - значение нижней границы интервала,
// начальное значение равно нулю.
// range - размер интервала,
// начальное значение равно TOP.

void encode_normalize( void ) {
  while( range <= BOTTOM ) {

    // перенос невозможен, поэтому возможна
    // запись в выходной файл (ситуация 2)
    if( low < 0xFF << SHIFTBITS ) {
      output_byte( next_char );
      for(;carry_counter;carry_counter--)
        output_byte(0xFF);
      next_char = low >> SHIFTBITS;

    // возник перенос (ситуация 3)
    } else if( low >= TOP ) {
      output_byte( next_char+1 );
      for(;carry_counter;carry_counter--)
        output_byte(0x0);
      next_char = low >> SHIFTBITS;

    // элемент, который может повлиять на перенос
    // (ситуация 4)
    } else {
      carry_counter++;
    }
    range <<= 8;
    low = (low << 8) & (TOP-1);
  }
}
```

```
void decode_normalize( void ) {
  while( range <= BOTTOM ) {
    range <<= 1;
    low = low<<8 |
          ((next_char<<EXTRABITS) & 0xFF);
    next_char = input_byte();
    low |= next_char >> (8-EXTRABITS);
    range <<= 8;
  }
}
```

Для сравнения приведем текст функции, оперирующей с битами, из работы [2]:

```
#define HALF (1<<(CODEBITS-1))
#define QUARTER (HALF>>1)

void bit_plus_follow( int bit ) {
  output_bit( bit );
  for(;carry_counter;carry_counter--)
    output_bit(!bit);
}

void encode_normalize( void ) {
  while( range <= QUARTER ) {
    if( low >= HALF ) {
      bit_plus_follow(1);
      low -= HALF;
    } else if( low + range <= HALF ) {
      bit_plus_follow(0);
    } else {
      carry_counter++;
      low -= QUARTER;
    }
    low   <<= 1;
    range <<= 1;
  }
}

void decode_normalize( void ) {
  while( range <= QUARTER ) {
    range <<= 1;
    low = low<<1 |input_bit();
  }
}
```

Процедура интервального кодирования очередного символа выглядит следующим образом:

```
void encode(
    int symbol_freq, // частота кодируемого символа
    int prev_freq,   // накопленная частота символов,
                     // предшествующих кодируемому
                     // в алфавите
    int total_freq   // частота всех символов
  ) {
  int r = range / total_freq;
  low += r*prev_freq;
  range = r*symbol_freq;
  encode_normalize();
}
```

Упражнение. Написать процедуру интервального декодирования, использующую приведенные выше функции нормализации.

Рассмотрим пример интервального кодирования строки "КОВ.КОРОВА". Частоты символов задаются следующим образом:

Индекс	Символ	Symbol_freq	Prev_freq
0	О	3	0
1	К	2	3
2	В	2	5
3	Р	1	7
4	А	1	8
5	"."	1	9
total_freq		10	

Для кодирования строки будем использовать функцию compress:

```
void compress(
DATAFILE *DataFile // файл исходных данных
) {
  low = 0;
  range = TOP;
  next_char = 0;
  carry_counter = 0;
  while( !DataFile.EOF ()) {
    c = DataFile.ReadSymbol()  // очередной символ
    encode( Symbol_freq[c], Prev_freq[c], 10 );
  }
}
```

Символ	Symbol _freq	Prev _freq	Low	Range	Результат
			0	0x7FFFFFFF	
K	2	3	0x26666664	0x19999998	
O	3	0	0x26666664	0x051EB850	
B	2	5	0x28F5C28A	0x010624DC	
.	1	9	0x29E1B07C	0x001A36E2	
Нормализация			0x61B07C00	0x1A36E200	0x53
K	2	3	0x698DC232	0x053E2ECC	0x53
O	3	0	0x698DC232	0x0192A7A3	0x53
P	1	7	0x6AA79DEC	0x002843F6	0x53
Нормализация			0x279DEC00	0x2843F600	0x53D5
O	3	0	0x279DEC00	0x0C146364	0x53D5
B	2	5	0x2DA81DAF	0x026A7A46	0x53D5
A	1	8	0x2F96E5E7	0x003DD907	0x53D5
Нормализация			0x16E5E700	0x3DD90700	0x53D55F

Как уже было отмечено, чаще всего при нормализации не происходит переноса. Исходя из этого, Дмитрий Субботин[1] предложил отказаться от переноса вовсе. Оказалось, что потери в сжатии совсем незначительны, порядка нескольких байтов. Впрочем, выигрыш по скорости тоже оказался не очень заметен. Главное достоинство такого подхода – в простоте и компактности кода. Вот как выглядит функция нормализации для 32-разрядной арифметики:

```
#define CODEBITS 24
#define TOP (1<<CODEBITS)
#define BOTTOM (TOP>>8)
#define BIGBYTE (0xFF<<(CODEBITS-8))

void encode_normalize( void ) {
  while( range <  BOTTOM ) {
    if( low & BIGBYTE == BIGBYTE &&
        range + (low & BOTTOM-1) >= BOTTOM )
      range = BOTTOM - (low & BOTTOM-1);
    output_byte(low>>24);
    range<<=8;
    low<<=8;
  }
}
```

Можно заметить, что избежать переноса нам позволяет своевременное принудительное уменьшение значения размера интервала. Оно происходит

[1] *Dmitry Subbotin. русский народный rangecoder// Сообщение в эхо-конференции FIDO RU.COMPRESS. 1 мая 1999.*

тогда, когда второй по старшинству байт low принимает значение 0xFF, а при добавлении к low значения размера интервала range возникает перенос. Так выглядит оптимизированная процедура нормализации:

```
void encode_normalize( void ) {
  while((low ^ low+range)<TOP ||
        range < BOTTOM &&
      ((range = -low & BOTTOM-1),1)) {
    output_byte(low>>24);
    range<<=8;
    low<<=8;
  }
}
void decode_normalize( void ) {
  while((low ^ low+range)<TOP ||
        range<BOTTOM &&
      ((range= -low & BOTTOM-1),1)) {
    low = low<<8 | input_byte();
    range<<=8;
  }
}
```

Упражнение. Применить интервальное кодирование без переноса для строки "КОВ.КОРОВА".

Литература

1 Martin G. N. N. Range encoding: an algorithm for removing redundancy from digitized message // Video & Data Recording Conference, Southampton. July 24–27, 1979.

2 Moffat A. Arithmetic Coding Revisited // Proceedings of Data Compression Conference. Snowbird, Utah, 1995.

3 Schindler M. A byte oriented arithmetic coding // Proceedings of Data Compression Conference. 1998. http://www.compressconsult.com /rangecoder/.

4 Яблонский С. В. Введение в дискретную математику. М.: Наука, 1986. Разд. "Теория кодирования".

Нумерующее кодирование

Английское название метода – Enumerative Coding, или ENUC.

Цель – сжатие блока *R*-битовых элементов в предположении, что у него есть одна важная характеристика *C*, которую выгодно сжимать отдельно от остальных.

Такой характеристикой *C* может быть, например, сумма всех элементов блока (или же произведение), или максимальное значение элемента, а менее

важной – вклад конкретных элементов в эту сумму (произведение), или положение элемента с максимальным значением внутри блока.

Основная идея состоит в том, чтобы формировать два блока: сохраняющий самую важную характеристику C и содержащий остальные данные D (необходимые для восстановления исходного блока) – так, чтобы все комбинации D были почти равновероятны. И далее обрабатывать эти блоки раздельно: их характеристики существенно различны.

Например, сжимая блок из 3 битов (R=1, длина L=3), метод сохраняет сумму битов S, $0 \le S \le 3$ (для ее записи требуется уже 2 бита), а также записывает положение единицы Z_1, если S=1, положение нуля Z_0, если S=2, или ничего, если S=0 или S=3.

S	Возможные блоки	Z_0	Z_1
0	000	-	-
1	100	-	0
	010	-	1
	001	-	2
2	011	0	-
	101	1	-
	110	2	-
3	111	-	-

И далее считаем, что все 3 значения Z_0 равновероятны, а также все 3 значения Z_1 – равновероятны.

Аналогично при сжатии блока X из 7 бит: сохраняем его сумму $0 \le S \le 7$ в 3 бита и далее, в зависимости от этой суммы, номер k-й комбинации битов (во множестве всех возможных $K_S[k]$, считающихся равновероятными), которая описывает текущий блок X известной длины L=7 с известной суммой S.

- Если S=0 или S=7, сохранять k не нужно.
- Если S=1 или S=6, есть 7 вариантов для k.
- Если S=2 или S=5, есть 21 вариант.
- Если S=3 или S=4, есть 35 вариантов.

В общем случае, если длина *битового блока* L, а сумма его битов S, то вариантов для k существует

$$V = L!/(S! \cdot (L-S)!\ ), \qquad (1.4)$$

т. е. $(L-(S-1))\cdot(L-1)\ldots L\ /\ (1\cdot 2\cdot 3\ldots S)$

Как легко заметить, при любых L и S числитель дроби кратен знаменателю и деление будет без остатка. Произведение первых n сомножителей числителя кратно произведению первых n сомножителей знаменателя.

Номера вариантов метод ENUC считает равновероятными.

Таким образом, всего требуется $\log_2(L+1)$ бит для записи S (в данном примере самая важная характеристика C – это S, сумма битов блока) плюс $\log_2(V)$ бит для записи остальных (равновероятных) данных D (в нашем примере – это V, вычисляемая по формуле (1.4)).

Если требуется сжать блок *R-битовых элементов* X[i], известно два подхода:

1. Преобразовать его в блок битов B, помещая в B на каждом i-м шаге X[i] нулей и одну единицу (или, наоборот, X[i] единиц и один нуль).
2. Преобразовать его в R блоков битов: в первом блоке – старшие биты, во втором – следующие (возможно, сортированные по первым, методом сортировки параллельных блоков), затем третьи по старшинству и т. д., до R-х (все j-е можно сортировать по всем предыдущим, старшим (j–1) битам элементов блока X).

Размер данных в результате применения ENUC уменьшается, если данные соответствуют ENUC-модели: важна одна характеристика, значения остальных равновероятны.

Методы этой группы являются **трансформирующими и блочными** (т. е. могут применяться только в том случае, когда задана длина блока с данными, подлежащими сжатию).

В общем случае скорость работы компрессора равна скорости декомпрессора и зависит и от размера данных и от их содержания. Оба идут по создаваемому множеству вариантов $K_S[k]$: компрессор – чтобы найти k – положение сохраняемого варианта (его *номер*) в создаваемом множестве K_S, декомпрессор – чтобы найти вариант, зная его номер k.

Впрочем, если сжимаемых блоков много, скорость работы декомпрессора выше скорости компрессора. Скорость обоих выше (и уже не зависит от содержания данных), поскольку не требуется каждый раз создавать множество вариантов. Но памяти необходимо больше, так как требуется хранить описания множеств.

Так, в рассмотренном выше случае сжатия 7-битового блока это будет три массива: из 7, 21 и 35 элементов. Если сжимаем поток 7-битовых блоков, имеет смысл один раз создать эти 3 массива $K_1[7]$, $K_2[21]$, $K_3[35]$. Компрессор будет делать поиск заданной комбинации битов W в этих K_1, K_2, K_3 и сохранять номер k, а декомпрессор, получая k, восстанавливать исходную комбинацию W. С целью еще более повысить скорость можно сделать все 3 массива длиной 2^7=128.

Векторное квантование

Цель **скалярного** квантования – преобразование потока *R*-битовы ментов, такое, чтобы в формируемом выходном потоке оставалось не чем (заданное число) *N* значений.

Иллюстрация скалярного квантования, *N*=5, – на рис. 1.4.

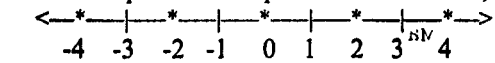


Рис. 1.4

Входное значение из диапазона	На выход записывается
(–∞,–3]	–4
(–3, –1]	–2
(–1,+1)	0
[+1,+3)	+2
[+3,+∞)	+4

Аналогично цель векторного квантования (Vector Quantization преобразование потока *групп элементов* (векторов), такое, чтобы ной поток записывался один из *N* векторов.

Иллюстрация векторного квантования для двумерного случая рис. 1.5.

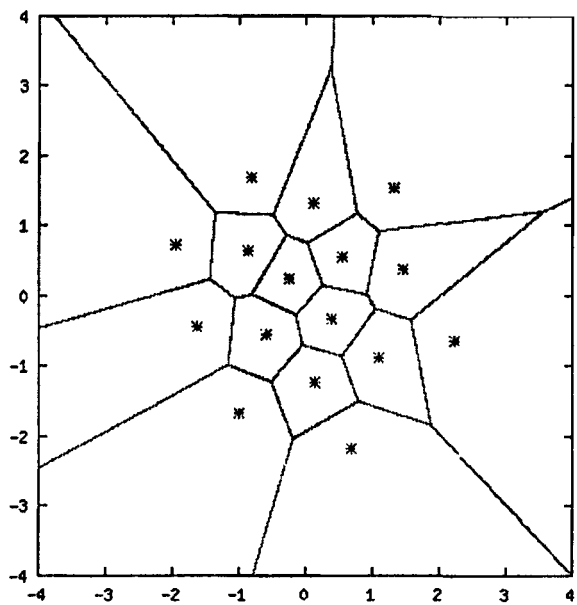


Рис. 1.5. Иллюстрация векторного квантования для двух измерени

Выходные векторы, обозначаемые звездочками *, называются кодирующими векторами (или код-векторами), а множество всех код-векторов называется *код-книгой*.

Основная идея состоит в том, чтобы каждый входной вектор $X = (X_{i,1}, X_{i,2},\ldots, X_{i,k})$ заменять адресом (в код-книге) того код-вектора $C = (C_{j,1}, C_{j,2},\ldots ,C_{j,k})$, отклонение которого от входного, определяемое как $D = (X_{i,1} - C_{j,1})^2+(X_{i,2}-C_{j,2})^2+\ldots+(X_{i,k}-C_{j,k})^2$, минимально.

Размер данных в результате применения VQ уменьшается, если данные соответствуют модели, либо если сжимаем с потерями.

Методы этой группы являются **трансформирующими** и **поточными** (т. е. могут применяться даже в том случае, когда длина блока с данными не задана).

Из краткого описания общей идеи видно, что

1. Процесс сжатия сводится к поиску по код-книге и в общем случае сложнее процесса разжатия, выполняющего копирование задаваемых векторов из код-книги в разжатый поток.
2. Задача метода может формулироваться двояко:
   а) задан размер код-книги и требуется заполнять ее так, чтобы суммарное отклонение было минимальным;
   б) задана верхняя граница суммарного отклонения (среднего по заданному числу векторов) и требуется заполнять код-книгу так, чтобы ее размер был минимальным.
3. Размеры элементов внутри k-мерного вектора X могут быть разными.
4. В еще более сложном случае размеры N векторов разные: $k_1, k_2, \ldots, k_3$, но одинаковы размеры элементов внутри этих N векторов переменной длины. И задача состоит в оптимальном разбиении входного потока на векторы.

Прямое преобразование

В простейшем случае – просто деление компонент вектора на заданные числа B_i. Если компонента одна, а $B_i = B = 4$:

```
S[i]=S[i]/4;
```

Следующий по сложности вариант – поиск код-вектора (одномерного) в код-книге (массиве CodeBook размером CB_Size, в котором код-векторы сортированы по возрастанию значений):

```
                    // найдем минимальный код-вектор
                    // со значением, большим S[i]
for (cvector=0, step=CB_Size/2; step>0; step/=2)
  if (S[i]<CodeBook[cvector+step]) cvector+=step;
                    // определим, к нему или к
```

```
                        // предыдущему ближе кодируемый S[i]
if ( (S[i]-CodeBook[cvector-1])<(CodeBook[cvector]-S[i]) )
      cvector--;
```

Обратное преобразование тривиально: из код-книги берутся элементы по адресам, задаваемым значениями элементов входного сжатого потока.

Один из самых распространенных методов, использующих векторное квантование, – **палитризация изображений**.

Из R_0-битовых элементов, применяя VQ, создаем R_1-битовые, являющиеся указателями на вектора-цвета (в таблице-палитре), причем

1) так, чтобы расхождение между исходным множеством S_0 и восстановленным по палитре множеством S_1 было минимально возможным;
2) $R_1<R_0$.

В данном случае код-книга содержит палитру, а код-векторы есть индексы цветов в этой палитре. Степень сжатия будет равна R_0/R_1. Например, если R_0=24, R_1=8, то получаем сжатие в 3 раза.

Однако применение векторного квантования не обязательно означает, что сжатие будет с потерями. Можно сохранять полностью и информацию об отклонениях реальных векторов из входного потока от аппроксимирующих их векторов из код-книги.

Увеличение скорости сжатия

Если памяти достаточно, а размер входного блока существенно больше множества возможных значений векторов входного блока X_i, имеет смысл заранее, при инициализации, создать таблицу, ставя в соответствие всевозможным X_i оптимальные для них (по критерию D) код-векторы.

Глава 2. Кодирование источников данных типа "аналоговый сигнал"

Линейно-предсказывающее кодирование

Английское название метода – Linear Prediction Coding (LPC).

Цель – сжатие потока R-битовых элементов в предположении, что значение каждого из них является линейной комбинацией значений h предыдущих элементов:

$$S_i = \sum_{j=i-1}^{j=i-h} K_j S_j ,$$

где S_i – i-й R-битовый элемент; K_j – некоторые коэффициенты, в общем случае непостоянные.

Основная идея состоит в том, чтобы в формируемый поток записывать ошибки предсказаний: разности между реальными S_i и предсказанными значениями:

$$D_i = S_i - \sum_{j=i-h}^{j=i-1} K_j S_j .$$

Размер данных в результате применения LPC не изменяется. Более того, размер элементов R может даже увеличиваться на единицу: $R`=R+1$ – за счет добавления бита для сохранения знака разности.

Чем точнее предсказание, тем больше в преобразованной последовательности элементов с близкими к нулю значениями, тем лучше можно сжать такую последовательность.

Для сжатия результата работы метода может быть применена любая комбинация методов – RLE, MTF, DC, PBS, HUFF, ARIC, ENUC, SEM...

Методы этой группы являются **трансформирующими** и **поточными** (т. е. могут применяться даже в том случае, когда длина блока с данными не задана). Скорость выполнения прямого преобразования равна скорости обратного преобразования и зависит только от размера данных, но не от их содержания, т. е. Скорость=O(Размер). Памяти требуется порядка h байт.

Из краткого описания общей идеи видно, что

1) можно один раз, при инициализации, задать как h, так и K_j;
2) с точки зрения сложности вычислений и качества сжатия полезно ограничить h, но периодически определять оптимальные K_j;
3) теоретически одна и та же зависимость может быть задана несколькими формулами, например
   $S_i=(S_{i-1}+S_{i-2})/2$ и $S`_i= S_{i-1}/2 + S_{i-2}/2$,
   но реально, из-за использования целочисленной арифметики, они не эквиваленты; в данном случае при вычислении $S`_i$ используется два округления вместо одного, поэтому мы получим различие.

Прямое преобразование

В базовом простейшем случае иллюстрируется одной строкой:

```
//предполагаем, что S[i]≈S[i-1] и, следовательно, D[i]≈0
D[i]=S[i]-S[i-1];
```

где S[N] – исходный массив (Source, источник); D[N] – выходной массив преобразованных данных (Destination, приемник, с отклонениями, т. е. разностями, дельтами).

Такой вариант обычно называется дельта-кодированием (Delta Coding), поскольку записываем приращения элементов относительно значений предыдущих элементов. Если в значениях элементов имеется ярко выраженная линейная тенденция, то в результате получаем ряд чисел с близкими значениями.

Три строки, если требуется писать в тот же массив:

```
current=S[i];      //возьмем очередной элемент
S[i]=current-last; //вычислим его разность с прошлым
last=current;      //теперь очередной стал прошлым
```

Например, из блока (12, 14, 15, 17, 16, 15, 13, 11, 9), применив этот простейший вариант преобразования, получим: (2, 1, 2, –1, –1, –2, –2, –2).

Более сложный вариант: h=4, предполагаем, что K_i=1/4:

$S[i] \approx (S[i-1] + S[i-2] + S[i-3] + S[i-4])/4$. Тогда в алгоритме:

```
D[i]=S[i] - (S[i-1]+S[i-2]+S[i-3]+S[i-4])/4;
```

Здесь мы предполагаем, что явно выраженная линейная тенденция отсутствует и значение очередного элемента примерно равно значению предыдущего, причем D[i] приобретает отрицательные и положительные значения с равной вероятностью. Чтобы компенсировать влияние случайных возмущений, значение следующего элемента принимаем равным среднему арифметическому нескольких последних элементов. С другой стороны, чем больше элементов усредняем, тем консервативнее наша модель, тем с большим запаздыванием она будет адаптироваться к существенным изменениям (необязательно линейного характера) во входных данных.

Если результат преобразования требуется записывать в тот же входной массив S, то это можно реализовать следующим образом. Введем вспомогательный массив Last, в котором будем хранить исходные значения последних четырех преобразованных элементов массива S.

```
int Last[4]={ 0,0,0,0 };
int last_pos=0;
```

И далее в цикле преобразования для каждого элемента S[i] выполняем:

```
current=S[i]; // возьмем очередной элемент и вычислим
              // его разность с предсказываемым значением:
S[i]=current-(Last[0]+Last[1]+Last[2]+Last[3])/4;
Last[last_pos]=current;  // внесем current в массив Last
last_pos=(last_pos+1)&3; // новое значение позиции в Last
```

Упражнение. Как будет выглядеть алгоритм для модели $S[i]=(S[i-1] + S[i-2] + S[i-3])/3$? Будет ли он выполняться быстрее или медленнее рассмотренного варианта с h=4 и K_i=1/4?

Обратное преобразование

Алгоритм очень похож на алгоритм прямого преобразования. Единственное отличие – теперь вычисляем не разность, а сумму предсказываемого значения и отклонения:

```
S[i]=D[i]+S[i-1]; // по предположенному S[i]=S[i-1]
```

Если для D и S используем один и тот же массив (S):

```
S[i]=S[i]+S[i-1]; // сумма дельты с предыдущим элементом
```

В случае h=4 , K_f=1/4 :

```
S[i]=D[i] + (S[i-1]+S[i-2]+S[i-3]+S[i-4])/4;
```

Если пишем в тот же массив, то и в этом случае, в отличие от прямого преобразования, не требуется каких-то ухищрений:

```
S[i]=S[i] + (S[i-1]+S[i-2]+S[i-3]+S[i-4])/4;
```

Действительно, к моменту обратного преобразования очередного элемента S[i] в ячейках S[i–1], ..., S[i–4] уже находятся восстановленные значения, а не отклонения, что нам как раз и нужно. Таким образом, использовать при разжатии два массива совершенно необязательно.

Пути увеличения скорости сжатия

Если можно записывать в тот же массив, то можно обойтись без излишнего массива Last[h] и связанных с ним операций, выполняемых на каждом шаге цикла преобразования.

Рассмотрим все тот же пример с h=4 , K_f=1/4. Сделаем так, чтобы внутри цикла было присваивание:

```
S[i-4]=S[i] - (S[i-1]+S[i-2]+S[i-3]+S[i-4])/4;
```

Тогда первые 4 элемента нам придется записывать отдельно:

```
First[0]=S[0];
First[1]=S[1] - S[0];
First[2]=S[2] - (S[1]+S[0])/2;
First[3]=S[3] - (S[2]+S[1]+S[0])/3;
```

Далее с помощью цикла преобразуем цепочку из последних (N–4) элементов и запишем результаты в первые (N–4) ячейки массива S:

```
for (i=4; i<N; i++)
   S[i-4]=S[i] - (S[i-1]+S[i-2]+S[i-3]+S[i-4])/4;
```

Или даже так:

```
sum=S[3]+S[2]+S[1]+S[0];
for (i=4; i<N; i++)
   j=S[i-4], S[i-4]=S[i] - sum/4, sum=sum-j+S[i];
```

Теперь мы можем переписать первые 4 преобразованных элемента из вспомогательного массива First в S:

```
for (i=0; i<4; i++)  S[N-4+i]=First[i];
```

Таким образом, дополнительной памяти вне функции преобразования не требуется, внутри – только h элементов, скорость преобразования в общем случае та же, какая была бы при записи во второй массив.

Единственное осложнение – метод, сжимающий данные, полученные в результате преобразования LPC, должен сначала обработать h элементов из конца массива S, а лишь затем (N–h) из начала.

Если работаем не с потоком, а с блоком и длина массива S известна заранее, можно обрабатывать из конца в начало: в цикле по i от N–1 до 0 делаем: S[i+1] –=S[i].

Увеличение скорости разжатия

Если памяти достаточно и входных данных много, может оказаться выгодным сделать табличную реализацию каких-то арифметических операций (функций). С целью ускорения работы основного цикла можно, например, за счет этого избежать операции деления, выполняемой обычно гораздо дольше, чем операции сложения/вычитания и записи/чтения в/из памяти.

Например, вместо

```
S[i-3]=S[i] - (S[i-1]+S[i-2]+S[i-3])/3;
```

намного быстрее на большинстве процессоров и для большинства компиляторов будет

```
S[i-3]=S[i]- Value[ S[i-1]+S[i-2]+S[i-3] ];
```

или даже

```
                // R - размер элементов массива S в битах
S[i-3]=Value2[ (S[i-1]+S[i-2]+S[i-3])<<R + S[i] ];
```

Упражнение. Напишите циклы, заполняющие вспомогательные массивы Value[ ] и Value2[ ].

Естественно, такой подход применим и для увеличения скорости сжатия.

Пути улучшения степени сжатия

Как мы уже отмечали, целесообразно повторно вычислять коэффициенты K_j через каждые P шагов. На основании последних обработанных данных находим значения коэффициентов, оптимальные с точки зрения точности предсказания значений элементов, встреченных в блоке последних обработанных данных. Если P=1, такой вариант обычно называется *адаптивным*. При большом значении P вариант называется статическим, если значения K_j записываются в сжатый поток, и "полуадаптивным", "блочно-

адаптивным" или "квазистатическим", если в явном виде значения K_j не записываются.

При сжатии с потерями можем сохранять не точные значения каждой разности $D_i = S_i - \Sigma(K_j \cdot S_j)$, а только первые B бит или, например, номер первого ненулевого бита. Но тогда при сжатии нужно использовать на следующих шагах то же значение S_i, которое получится при разжатии, а именно $S_i` = \Sigma(K_j \cdot S_j) + D_i`$, т. е. результат предсказания плюс сохраненная часть разности-дельты.

Метод LPC применяется обычно для сжатия аналоговых сигналов. И как правило, каждый элемент сигнала отклоняется от своего предсказываемого значения не только из-за "сильных" обусловленных изменений – эволюции, но и из-за "слабых" фоновых колебаний, т. е. шума. Поэтому возможно **два противоположных типа моделей**:

- вклад шума невелик по сравнению с вкладом эволюции;
- вклад эволюции невелик по сравнению с вкладом шума.

В первом случае мы будем предсказывать значение S[*i*] на основании сложившейся линейной тенденции, во втором – как равное среднему арифметическому *h* предыдущих элементов.

> *Тенденция может быть и нелинейной, например S[i]=S[i–1]+2*(S[i–1]–S[i–2]), но такие модели на практике в большинстве случаев менее выгодны и мы их рассматривать не будем.*

Если ресурсов достаточно, можно вычислять предсказываемые значения, даваемые несколькими моделями, и затем либо синтезировать эти несколько предсказаний, либо выбирать (через каждые *P* шагов) ту модель, которая оптимальна на заданном числе предыдущих элементов.

Требуется минимизировать ошибки предсказаний, поэтому рациональнее (выгоднее) та модель, которая дает наименьшую сумму абсолютных значений этих ошибок:

$$\min\left(\sum|D_i|\right).$$

> *Эта задача имеет элементарное аналитическое решение: min\|D\| = min (\|Y–K*X\|) → K = Y*X^{-1}, где Y – вектор из S[i] (реально наблюдаемых), K – матрица коэффициентов и X – матрица S[i-x] (x = 1,...,h). Но 1) сложность вычисления обратной матрицы не менее O(h^2) и перевычислять ее невыгодно, 2) высокая точность прогноза обычно не нужна, так как на практике стационарные последовательности встречаются редко.*

Рассмотрим подробнее на примерах.

Общий случай

Если *h*=1, то учитывается только один последний элемент: S[*i*]=*K*·S[*i*–1]. Об эволюции не известно ничего, но можно предполагать, что *K*=1.

Упражнение. Изобразите сигнал, оптимально сжимаемый с помощью модели с *h*=1 и *K*= –1.

Все это, однако, не помешает нам действовать следующим образом: выберем начальное значение K_0, шаг *StepK* изменения *K*, размер окна *F*, а также *P*. Будем следить, с какой из трех моделей – $K=K_0$, $K_p=K_0+StepK$, $K_m=K_0-StepK$ – сжатие последних *F* элементов лучше (меньше суммарная ошибка предсказания), и через каждые *P* элементов выбирать лучшую (в данный момент) из трех моделей, с помощью которой и будут сжиматься следующие *P* элементов.

Если это не текущая, а одна из двух с ±*StepK*, то изменяются и K_p с K_m, если, конечно, их значения не совпадают с границами: –1 или 1.

Например, можно использовать K_0=0, *StepK*=1/4, *F*=10 и *P*=2. Имеет смысл задать еще один параметр *M* – минимальный размер выигрыша оптимальной модели у текущей. Если выигрыш меньше *M*, то отказываемся от смены модели.

Следующий по сложности вариант использует переменный шаг для *K*. Также мы можем похожим образом корректировать *F* (например, изменять с шагом *StepF*, через каждые *Q* обработанных элементов).

Если *h*=2

Выражение $S_i = \sum_{j=i-1}^{j=i-h} K_j S_j$ можно всегда переписать в таком виде:

$$S_i = K`_1 S_{i-1} + \sum_{j=1}^{j=h-1} K`_{j+1}(S_{i-j} - S_{i-j-1}).$$

Поэтому модель при *h*=2 можно представить как

$$S[i]=K_1\cdot S[i-1] + K_2\cdot(S[i-1] - S[i-2]).$$

Найдем значения коэффициентов **эволюционной** модели. Если шум стремится к нулю, то изменения элементов объясняются только линейной тенденцией (эволюцией). Иначе говоря:

$$S[i]-S[i-1]=S[i-1]-S[i-2].$$

Отсюда получаем: K_1=1, K_2=1.

Для **шумовой** модели S[*i*]=(S[*i*-1]+S[*i*-2])/2, и коэффициенты K_j получим из системы:

K_1+K_2=1/2 (коэффициент при S[*i*-1])

$-K_2=1/2$ (коэффициент при S[i-2])

Находим: $K_1=1$, $K_2=-1/2$. Таким образом, имеет смысл корректировать K_2 в диапазоне не от 0 до 1, как может показаться на первый взгляд, а от -1/2 до 1.

Если h=3

$$S[i]=K_1\cdot S[i-1] + K_2\cdot(S[i-1]-S[i-2]) + K_3\cdot(S[i-2]-S[i-3]). \qquad (2.1)$$

У эволюционных моделей уже нет однозначного ответа о K_1, K_2 и K_3: $K_1=1$, $K_2+K_3=1$ (так как в общем случае через 3 точки нельзя провести прямую). Например, можно взять $K_2=3/4$ и $K_3=1/4$, т. е. последнее приращение имеет втрое больший вес, чем предпоследнее. Или же $K_2=2$, $K_3=-1$ (вторая производная постоянна).

Шумовая модель в одномерном случае одна при любых h, и при h=3:

$$S[i]=(S[i-1] + S[i-2] + S[i-3])/3 \qquad (2.2)$$

Попробуем совместить обе модели – шумовую и эволюционную.

При каких K_1, K_2 и K_3 эволюционная (2.1) становится шумовой (2.2)?

Из системы

$K_1+K_2=1/3$ (коэффициент при S[i-1]);
$-K_2+K_3=1/3$ (коэффициент при S[i-2]);
$-K_3=1/3$ (коэффициент при S[i-3])

находим: $K_1=1$, $K_2=-2/3$, $K_3=-1/3$.

Имеет смысл пытаться корректировать K_3 от -1 до 1, при этом: (K_2+K_3) от -1 (шумовая модель) до 1 (эволюционная), т. е. K_2 от ($-1-K_3$) до ($1-K_3$):

$K_2 \setminus K_3$	-1	...	...	1
$-1-K_3$		Ш		
...				
$1-K_3$	Э	Э	Э	Э

Ш – такая пара значений ($K_2=-2/3$, $K_3=-1/3$) соответствует шумовой модели; Э – такие пары $K_2+K_3=1$ соответствуют эволюционной (а остальные – "средней").

При h = 2 существует только одна эволюционная модель (через две точки всегда можно провести только одну прямую), по которой предсказываемое значение $S[i] = S^{(2)}[i]$ (рис. 2.1.);

При h = 3 возможно использование множества моделей: например, если считать изменение $S[i]$-$S[i-1]$ более важным, чем $S[i-2]$-$S[i-3]$, то такая модель может дать, например, предсказание $S_1^{(3)}[i]$ или, если считать $S[i-2]$-$S[i-3]$ и $S[i]$-$S[i-1]$ равноправными, прогнозное значение $S_2^{(3)}[i]$.

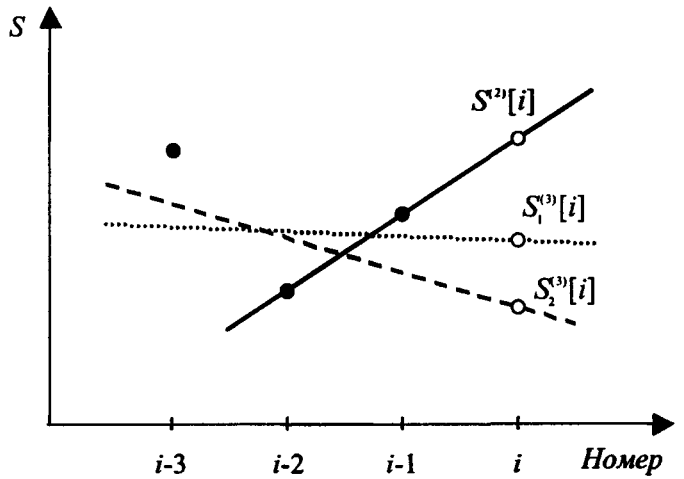


Рис. 2.1. Эволюционная модель при h=2 и h=3

Кроме того, при h=3 возникает возможность **уменьшать шум**. Метод заключается в следующем. После того как найдено среднее значение (S[i-1]+S[i-2]+S[i-3])/3, определим, какой из трех элементов больше отклоняется от него. Будем считать, что именно этот элемент содержит больше шума, чем остальные два, поэтому именно его учитывать не будем (или, что то же самое, положим его равным полусумме двух остальных). В качестве предсказания, даваемого шумовой моделью, возьмем полусумму двух остальных элементов.

Аналогично при h=4 есть возможность взять 3 приращения ("дельты") и оставить те два, которые ближе к среднему всех трех.

Остается добавить, что ряд методов для сжатия речи с потерями (CELP, CS-ACELP, MELP) использует LPC для вычисления нескольких характеристик фрагментов сигнала (процесс называется LPC analysis). После обратного процесса (LPC synthesis) получается сигнал с теми же характеристиками, но совершенно другими данными, т. е. значениями элементов.

Двумерный случай

Рассмотрим только самый простой обход плоскости[1] – строками.

В каждой точке плоскости уже обработанные, известные элементы – выше данной точки, а также на том же уровне, но левее ее.

---

[1] *Задача обхода плоскости возникает при обработке двумерных данных. Цель обхода – создание одномерного массива D из двумерного S. Подробнее смотри в разд. 2.*

S[i-L-1]	S[i-L]	S[i-L+1]
S[i-1]	S[i]	...

Здесь L – число элементов в строке. Ближайших к S[i] элементов – четыре. Обозначим для краткости так:

A	B	C
D	E	...

У шумовой модели есть 4 варианта с h=1, 6 вариантов с h=2, 4 варианта с h=3 и один с h=4:

$$E=K_1\cdot D + K_2\cdot C + K_3\cdot B + K_4\cdot A \qquad (2.3)$$

Можно изначально задать $K_1=K_2=K_3=K_4=1/4$ и корректировать K_i вышеизложенным методом. Границы будут заданы условиями: $K_i>0$, $K_1+K_2+K_3+K_4=1$, и, например, $K_1=K_3$, $K_2=K_4$ (и таким образом, $K_1+K_2=1/2$, достаточно корректировать K_1).

У вариантов с h>2 есть возможность уменьшать шум тем же методом, что и в одномерном случае.

Таким образом, если использовать только 4 или меньше ближайших элементов, шумовая модель дает 20 вариантов: 4 – с h=1, 6 – с h=2, 8 – с h=3 и 2 – с h=4.

Эволюционная модель делает то же предположение, что и в одномерном случае: сохраняется линейная тенденция, приращение на текущем шаге примерно равно приращению на предыдущем. Но теперь, в двумерном случае, необходимо как минимум 3 элемента, а не 2:

$$E=K_1\cdot B + K_2\cdot (D-A);$$

$$E=K_1\cdot D + K_2\cdot (B-A); \qquad (2.4)$$

$$E=K_1\cdot D + K_2\cdot (B-A) + K_3\cdot (C-B). \qquad (2.5)$$

Первые два варианта, по предположению эволюционной модели, сводятся к одному: $K_1=K_2=1$. В третьем, при коррекции в рамках эволюционной модели, границы таковы: $K_1=1$, $K_2+K_3=1$.

Попробуем построить модель с h=4, объединяющую два противоположных полюса – шумовой и эволюционный.

При каких K_1, K_2 и K_3 эволюционная модель (2.5) становится шумовой: (2.3) с K_i=1/4?

K_1=1/4 (коэффициент при D)
K_3=1/4 (коэффициент при C)
K_2-K_3=1/4 (коэффициент при B)
$-K_2$= 1/4 (коэффициент при A)

Решений нет, и правильный ответ: ни при каких. Потому что в (2.3) коэффициенты при C, B и A должны быть $K_i>0$. Применяя это условие к (2.5), получаем:

(C): $K_3>0$;

(A): $K_2<0$;

(B): $K_2-K_3>0$ (это несовместимо с результатами по C и A).

Тем не менее синтез шумовой и эволюционной модели при $h=4$ возможен. Будем брать предсказание того из 20 вариантов шумовой модели, который ближе всего к предсказанию эволюционной, т. е. разность между ними минимальна.

Как раз такой путь реализован в популярном алгоритме **PNG**, а именно в самом сложном, самом интеллектуальном его фильтре Paeth: 3 варианта шумовой модели (с использованием элементов A, B и D) и эволюционная модель (2.4). По сути это тот же метод, что и описанный выше в подразд. "Общий случай", только вместо среднего по трем точкам берется "двумерное эволюционное" значение и исключаются два элемента, а не один.

Другой синтез возможен, если эволюционную модель модифицировать:

$$E=K_1\cdot D+K_2\cdot(B-A)+K_3\cdot(C-B)+K_4\cdot A; \quad (2.6)$$

$$E=K_1\cdot D+K_2\cdot(B-A)+K_3\cdot(C-B)+K_4\cdot B; \quad (2.7)$$

$$E=K_1\cdot D+K_2\cdot(B-A)+K_3\cdot(C-B)+K_4\cdot C. \quad (2.8)$$

Чтобы формулы описывали эволюционную модель, должно быть $K_4=0$, $K_1=K_2+K_3=1$. В шумовой модели $K_1=1/4$, а K_2, K_3 и K_4 найдем из условий: коэффициенты при A, B и C должны быть равны 1/4:

в случае (2.6) C: $K_3=1/4$, B: $K_2=1/2$, A: $K_4=3/4$;

в случае (2.7) C: $K_3=1/4$, A: $K_2=-1/4$, B: $K_4=3/4$;

в случае (2.8) A: $K_2=-1/4$, B: $K_3=-1/2$, C: $K_4=¾$.

Рассмотрим, например, (2.7). Имеет смысл корректировать K_1 от 1/4 (шумовая модель) до 1 (эволюционная); K_4 найдется из условия $K_1+K_4=1$; $K_3=1/4$; K_2+K_3 от 0 (шумовая) до 1 (эволюционная), т. е. K_2 от $-1/4$ до 3/4.

Таким образом, из (2.7) получается эволюционная модель при $K_1=1$, $K_2=3/4$ и шумовая при $K_1=1/4$, $K_2=-1/4$.

Упражнение. Каким будет синтез эволюционной и шумовой моделей в случаях (2.6) и (2.7) ?

LPC в алгоритме PNG

Для сжатия изображения можно выбрать одну из следующих LPC-моделей (называемых также фильтрами):

1) E=0 (нет фильтра);

2) E=D (элемент слева);
3) E=B (элемент сверху);
4) E=(B+D)/2;
5) вышеописанный алгоритм Paeth.

Все 5 моделей – варианты шумовой модели, и только последняя модель является комбинированной, учитывающей эволюцию.

LPC в алгоритме Lossless Jpeg

Может быть задан один из восьми предсказателей-фильтров:
1) E=0 (нет фильтра);
2) E=B (элемент сверху);
3) E=D (элемент слева);
4) E=A (выше и левее);
5) E=B+D–A;
6) E=B+(D–A)/2;
7) E=D+(B–A)/2;
8) E=(B+D)/2.

Пятый, шестой и седьмой – варианты эволюционной модели, остальные – шумовой.

Выбор фильтра

Итак, имеем множество фильтров $Sp_n[X,Y]=S_n(K_{i,j}\cdot S[X-i,Y-j])$ дающих оценки-предсказания Sp_n для значения элемента в позиции (X,Y) как функцию S_n от значений элементов контекста, т. е. элементов, соседних с текущим $S[X,Y]$.

Два уже рассмотренных (в подразд. "Общий случай") пути дальнейших действий:

1) выбрать одно значение Sp_n, даваемое той функцией S_n, которая точнее на заданном числе предыдущих элементов;
2) выбрать значение, вычисляемое как сумма всех Sp_n , взятых с разными весами: $S_p = \Sigma(W_n\cdot Sp_n)$, $0\le W_n\le 1$.

Кроме довольно очевидного способа – корректировки W_n, весов, с которыми берутся предсказания, даваемые разными фильтрами, есть и еще варианты:

- Брать те K значений Sp_n, которые дают S_n, лучшие на K (задаваемом числе) ближайших элементов контекста (некоторые из этих S_n могут совпадать, и их останется меньше чем K).
- Все функции S_n сортируются по значению точности предсказания на ближайших элементах контекста, и выбирается несколько функций, дающих лучшую точность.

Четвертый вариант является простым расширением первого. Третий же предполагает постановку в соответствие каждому элементу одной S_n, дающей для него самое точное предсказание. В обоих случаях, если требуется выбрать одну S_n из нескольких S_i с одинаковым значением точности, можно посмотреть значения точности этих S_i на большем числе элементов.

Во 2, 3 и 4-м часто полезно уменьшить шум вышеописанным способом. И в конечном итоге сложить оставшиеся $S_p = \Sigma(W_n \cdot Sp_n)$. В простейшем случае $W_n=1/H$, где H – количество оставшихся S_n.

И еще три простых, но важных замечания.

Во-первых, сжатие обычно лучше, если значения оценок-предсказаний Sp_n не выходят за **границы диапазона** допустимых значений элементов входного потока S[i]. Если $A<S[i]<B$, то и $Sp_n[i]$ должны быть $A<Sp_n[i]<B$.

Во-вторых, если известно, что **доля шума постоянна** во всем блоке данных, очень полезно делать SEM до LPC, если нет другого метода для отделения шума.

И в-третьих, в каждой точке (x,y) веса $K_{a,b}$ значений $S_{a,b}$ из (x–a, y–b) должны зависеть от **расстояния** до (x,y), вычисляемого как $R=(a^2+b^2)^{1/2}$. Эти веса должны убывать с увеличением расстояния R. Из четырех ближайших элементов контекста, рассмотренных в подразд. "Двумерный случай", D и B находятся на расстоянии 1, а A и C на расстоянии $2^{1/2}$. Поэтому, если как в формуле (2.3), шумовая модель складывает значения четырех ближайших элементов контекста с разными весами K_i:

$$E=K_1 \cdot D + K_2 \cdot C + K_3 \cdot B + K_4 \cdot A$$

имеет смысл задавать $K_1=K_3$, $K_2=K_4$, и еще из-за учета расстояний $K_2/K_1=2^{1/2}$ (и конечно, $K_1+K_2+K_3+K_4=1$).

Точно так же при вычислении точностей фильтров на элементах контекста ошибки их предсказаний $V_{a,b}$ (а точнее, абсолютные значения этих ошибок) должны учитываться с весами $K_{a,b}$, зависящими от расстояний: $K_{a,b}=K((a^2+b^2)^{1/2})$.

У эволюционной модели, использующей 4 элемента контекста (A, B, C, D), приращения (B–A), (D–A) и (C–B) находятся на равном расстоянии и имеют одинаковый вес. Но если используется больше четырех элементов, тоже необходимо вычислять расстояния.

Характеристики методов семейства LPC:

Степень сжатия: увеличивается примерно в 1.1–1.9 раза.

Типы данных: методы предназначены для сжатия количественных данных.

Симметричность по скорости: в общем случае 1:1.

Характерные особенности: чем меньше доля "шума" в данных, тем выгоднее применение методов LPC.

Субполосное кодирование

Английское название метода – Subband Coding (SC). Дословный перевод – кодирование поддиапазонов.

Цель метода – сжатие потока R-битовых элементов в предположении, что значение каждого из них отличается от значений соседних элементов незначительно: $S_i \approx S_{i-1}$.

Основная идея состоит в том, чтобы формировать два потока: для каждой пары S_{2i}, S_{2i+1} сохранять полусумму $(S_{2i} + S_{2i+1})/2$ и разность $(S_{2i} - S_{2i+1})$. Далее эти потоки следует обрабатывать раздельно, поскольку их характеристики существенно различны.

В случае модели "аналоговый сигнал" физический смысл потока с полусуммами – низкие частоты, а с разностями – высокие. Методы SC предназначены для сжатия элементов с "количественными" данными, а не "качественными". Самый распространенный вид количественных данных – мультимедийные. Но не единственный: например, потоки смещений и длин, формируемые методом семейства LZ77, тоже содержат количественные данные.

Размер данных в результате применения SC не изменяется. Более того, в потоке с разностями размер элементов R может даже увеличиваться на 1 бит: R`=R+1, поскольку для сохранения разностей R-битовых элементов требуется (R+1) бит.

Для сжатия результата работы метода может быть применена любая комбинация методов – RLE, MTF, DC, PBS, HUFF, ARIC, ENUC, SEM...

Методы этой группы являются **трансформирующими** и **поточными** (т. е. могут применяться даже в том случае, когда длина блока с данными не задана). В общем случае скорость работы компрессора равна скорости декомпрессора и зависит только от размера данных, но не от их содержания: Скорость = O(Размер). Памяти требуется всего лишь несколько байтов.

Из краткого описания общей идеи видно, что

1) к обоим получающимся потокам можно повторно применять метод этого же семейства SC;
2) метод можно применять и к параллельным потокам (например, левый и правый каналы стереозвука);
3) при сжатии аналогового сигнала с потерями степень сжатия обратно пропорциональна ширине сохраняемого диапазона частот;
4) можно сохранять не полусуммы и разности, а суммы и полуразности (поскольку сумма и разность двух чисел – либо обе четны, либо обе нечетны, одну из них можно делить на 2 без всяких осложнений).

Прямое преобразование

В базовом простейшем случае иллюстрируется тремя строками:

```
for (i=0; i<N/2; i++) { // цикл по длине исходного массива
  D[2*i]=(S[2*i]+S[2*i+1])/2;   // четные - с полусуммами
  D[2*i+1]=S[2*i]-S[2*i+1];     // нечетные - с разностями
}                               // (1)
```

где S[*N*] – исходный массив (Source, источник); D[*N*] – выходной массив преобразованных данных (Destination, приемник, с дельтами, т. е. разностями, и полусуммами).

Если результат пишем в тот же массив S:

```
for (i=0; i<N/2; i++) {       // (2)
  d=S[2*i]-S[2*i+1];          // разность, и дальше так же:
  S[2*i]=(S[2*i]+S[2*i+1])/2;// четные будут с полусуммами,
  S[2*i+1]=d;                 // нечетные элементы массива -
}                             // с разностями
```

Если же формируем два выходных массива:

```
for (i=0; i<N/2; i++) {       // (3)
  A[i]=(S[2*i]+S[2*i+1])/2;  // полусумма (Average)
  D[i]=S[2*i]-S[2*i+1];      // разность (Delta)
}
```

то каждый из них – вдвое короче, чем исходный S.

Если его длина *N* нечетна, добавим к концу S элемент S[*N*], равный последнему S[*N*–1]; в результате он добавится к массиву A, а к D добавится нуль:

```
  A[N/2]=S[N-1];   // последняя полусумма
  D[N/2]=0;        // последняя разность
```

Если известно, что в результате разности может возникнуть **переполнение**, т. е. невозможно будет записать полученное значение, используя исходное число битов *R* (*R* – размер элементов исходного массива S):

```
if ( (S[2*i]-S[2*i+1])!=(S[2*i]-S[2*i+1])mod(n) )
{} // n=(2 в степени R)
```

то можем поступить одним из следующих **четырех** способов:

1. Изначально отвести под разности массив элементов большего размера:

```
char A[N];   // если под полусуммы 8 бит,
short D[N];  // то под разности 9 бит, а реально даже 16
```

2. Формировать третий (битовый) массив с флагами переполнений:

```
D[i]= S[2*i]-S[2*i+1];    //сохраним разность;
```

```
if(D[i]!=S[2*i]-S[2*i+1])
//если не хватило битов для записи,
    Overflow[i]=1;          //установим флаг в 1
else Overflow[i]=0;         //иначе его значение - нуль
```

3. Использовать текущее значение разности для вычисления полусуммы:

(4)

```
D[i]= S[2*i]-S[2*i+1];
                  // реально D[i]=(S[2*i]-S[2*i+1])mod(n);
A[i]= S[2*i]-D[i]/2;
                  //это полусумма, если не было переполнения
```

4. Наоборот, использовать текущее значение суммы для вычисления полуразности:

(5)

```
A[i]= S[2*i]+S[2*i+1];
             // реально A[i]=(S[2*i]+S[2*i+1])mod(n);
D[i]= S[2*i]-A[i]/2;
             // это полуразность, если не было переполнения
```

Упражнение. Можно ли использовать полуразности для вычисления сумм? А полусуммы для вычисления разностей?

Обратное преобразование

Обратное преобразование ничуть не сложнее прямого:

```
for (i=0; i<N/2; i++) {              // (3`)
  S[2*i]=  ( 2*A[i]+D[i] ) /2 ;
  S[2*i+1]=( 2*A[i]-D[i] ) /2 ;
}
```

Если использовалась защита от переполнения и брались разности для вычисления полусумм:

```
S[2*i]= A[i]+D[i]/2;                 // (4`)
S[2*i+1]= S[2*i]-D[i];
                    // реально D[i]=(S[2*i]-S[2*i+1])mod(n)
```

Пути улучшения степени сжатия

Общий случай

К получаемым потокам можно и дальше рекурсивно применять разложение на полусуммы (Average) и разности (Delta). При сжатии аналоговых сигналов, как правило, полезно дальнейшее разложение полусумм.

Есть два пути создавать 3 выходных потока или блока:

1. Original↓

   Average1+Delta1↓

   DA+DD

(**DA** – Average2, получаемая при разложении разностей Delta1, **DD** – Delta2, получаемая при разложении Delta1. Аналогично с остальными обозначениями).

2. Original↓
   Average1↓+**Delta1**
   AA+AD

Пять вариантов создания четырех:

1. Original↓
   Average1+Delta1↓
   DA+DD↓
   DDD+DDA

2. Original↓
   Average1+Delta1↓
   DA↓+**DD**
   DAD+DAA

3. Original1↓
   verage1↓ + Delta1↓
   AA+AD DA+DD

4. Original↓
   Average↓+**Delta**
   AA↓+**AD**
   AAA+AAD

5. Original↓
   Average↓+**Delta**
   AA+AD↓
   ADA+ADD

Четырнадцать вариантов создания пяти и т. д.

Упражнение. Изобразите эти 14 вариантов.

При принятии решения о целесообразности разложения некоторого потока или блока S на A и D оказывается полезен следующий прием. Будем заглядывать на несколько шагов вперед: если непосредственные результаты разложения A и D сжимаемы суммарно хуже, чем исходный S, может оказаться, что если разложить дальше A и/или D, то только тогда результат – три или четыре потока, по которым восстановим S, – сжимаем лучше, чем S.

Если же и эти AA, AD, DA, DD дают в сумме худшее сжатие, заглянем еще на шаг вперед, т. е. попытаемся разложить эти вторичные результаты,

и т. д., пока глубина такого просмотра не достигнет заданного предела, или же дальнейшее разложение окажется невозможным из-за уменьшения длин в 2 раза на каждом шаге.

Еще одно усложнение – поиск границ фрагментов потока, разбивающих его на такие блоки, чтобы их дальнейшее субполосное кодирование давало лучшее сжатие.

При сжатии параллельных потоков

Если имеем два параллельных потока X и Y, каждая пара (X_i, Y_i) описывает один объект. Например, X_i – адрес, Y_i – длина или же X – угол, Y – расстояние. В случае "аналоговый сигнал" пара (X_i,Y_i) относится к одному моменту времени t_i.

Число потоков и их длина N остаются неизменными. Можно оставлять один из двух исходных потоков вместо потока с полусуммами (а при сжатии с потерями и вместо разностей): X+D или Y+D вместо A+D.

Теперь можно искать границы таких блоков, внутри которых один из этих четырех вариантов – X+Y, X+D, Y+D, A+D – существенно выгоднее остальных трех.

Заметим, что применение LPC, в том числе дельта-кодирования, к потоку полусумм A выгоднее, чем к X и/или Y: первая производная потока с полусуммами A`[i] не превосходит (по абсолютному значению) максимума первых производных X`[i] и Y`[i]:

$$A`_i = A_{i+1}-A_i = (X_{i+1}+Y_{i+1}-\alpha_{i+1})/2 - (X_i+Y_i-\alpha_i)/2\ , \qquad (2.9)$$

"издержки округления" α_k равны единице, если сумма соответствующих X_k+Y_k нечетна, а арифметика используется целочисленная; иначе $\alpha_k=0$.

$$A`_i = (X_{i+1}-X_i+Y_{i+1}-Y_i+\alpha_i-\alpha_{i+1})/2 = (X`_i+Y`_i+\alpha_i-\alpha_{i+1})/2. \qquad (2.10)$$

Таким образом, значение $A`_i$ лежит внутри интервала $[X`_i, Y`_i]$ (рис. 2.2).

Заметим также, что (при использовании целочисленной арифметики из-за округлений) важен порядок действий: дельта-кодирование потока полусумм A даст другой результат, чем полусумма результатов дельта-кодирования X и Y: в первом случае (2.9) и (2.10), во втором $A``_i = (X`_i+Y`_i)/2$.

Если сумма $(X`_i+Y`_i)$ нечетна, а $\alpha_i-\alpha_{i+1}=1$, то $A`_i \neq A``_i$, $A`_i=A``_i+1$.

Например, если $X_i=0$, $X_{i+1}=Y_i=Y_{i+1}=1$, то $A`_i=1$ (по формуле (2.9)), а $A``_i=(1+0)/2=0$.

Если потоков более двух, например 5 (сжатие именно пятиканального звука становится все актуальнее), возникает задача нахождения оптимальных пар для применения к ним субполосного кодирования. Здесь тоже возможны оба вышеописанных приема: и "просмотр на несколько шагов впе-

ред", еще более усложненный, и "поиск границ интервалов времени", оптимальных для выделяемых затем пар. Например, может выясниться, что сжатие существенно лучше, если использовать знание того, что на интервале между 32765-м и 53867-м элементами очень близки первая разность (при первом разложении) внутри полусуммы (1,4), т. е. полусуммы 1-го потока с 4-м, и первая разность внутри полусуммы (3,7).

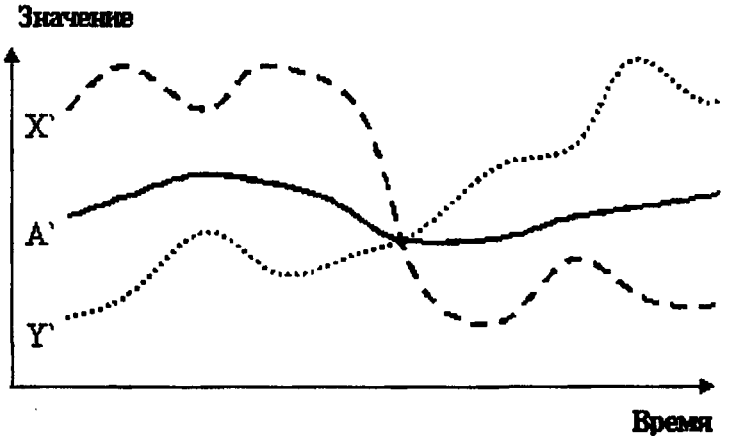


Рис. 2.2. Значение $A`_i$ лежит внутри интервала $[X`_i, Y`_i]$

Двумерный случай

Раньше на каждом этапе можно было принять одно из двух решений: применить SC-преобразование к блоку или не применять. Теперь же – одно из пяти:

1) применить SC к строкам блока;
2) применить SC к столбцам блока;
3) применить SC к строкам блока, а затем к столбцам;
4) применить SC к столбцам блока, а затем к строкам;
5) не применять SC к блоку вообще.

Пункты 3 и 4 не совсем эквивалентны при использовании целочисленной арифметики из-за округлений, но математическое ожидание расхождения их результатов равно нулю (рис. 2.3).

При сжатии аналоговых сигналов

Появляются дополнительные возможности:

- сжимая с потерями, округлять с заданной точностью: на каждом шаге SC и/или после всех этапов SC;

- сохранять только заданную субполосу, многократно применяя SC-разложение и оставляя только верхнюю половину частот (разности) либо нижнюю (полусуммы);
- использовать дискретное вэйвлетное преобразование для нахождения низкочастотной и высокочастотной компонент.

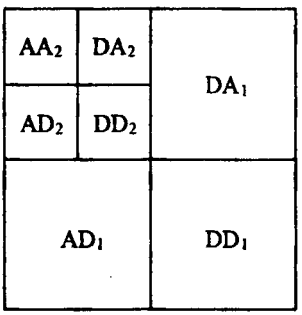


Рис. 2.3. Вариант применения SC в двумерном случае

Дискретное вэйвлетное преобразование

ДВП отличается от SC лишь тем, что в нем для вычисления низко- и высокочастотной компоненты используется не два соседних элемента сигнала, а произвольное задаваемое число элементов D>2. Причем в отличие от других контекстных методов, в частности LPC, контекст элемента S[x] состоит из элементов по обе стороны от него: S[x+i] и S[x–i], i=1,2,3,…,D/2.

Основная же идея – та же, что в SC: сформировать два потока – с низкими H[x] и высокими частотами L[x] – так, чтобы по половине значений H[x] и половине значений L[x] можно было восстановить исходный поток S[x]. Оставляются либо четные H[2·x] и нечетные L[2·x+1], либо наоборот.

При разжатии сначала по H[2·x] и L[2·x+1] восстанавливаем четные S[2·x], затем по S[2·x] и L[2·x+1] находим нечетные S[2·x+1].

Если D=3, L[x]=S[x]–(S[x–1]+S[x+1])/2,

H[x]=S[x]+hi(*x*).

Функция hi(S[i]) должна быть выбрана так, чтобы она была выражаема через L[x+j], j=2k+1. Например,

$hi_1(x)$=(L[x–1]+L[x+1])/2,

$hi_2(x)$=(L[x–1]+L[x+1])/4,

$hi_3(x)$= –(L[x–1]+L[x+1])/2,

$hi_4(x)$= –(L[x–1]+L[x+1])/4,

$hi_5(x)$=(L[x–3]+L[x–1]+L[x+1]+L[x+3])/4, и т. д.

Возьмем вторую функцию hi_2 (именно этот вариант используется в алгоритме JPEG 2000) и посмотрим, каким образом H выражается через S.

H[x]=S[x]+(L[x–1]+L[x+1])/4 =
=S[x]+ (S[x–1] – (S[x–2]+S[x])/2 + S[x+1] – (S[x]+S[x+2])/2) /4 =
=S[x]+ (2·S[x–1] – S[x–2] – S[x] + 2·S[x+1] – S[x] – S[x+2]) /8 =
=(–S[x–2] + 2·S[x–1] + 6·S[x] + 2·S[x+1] – S[x+2]) /8.

ДВП обычно задается в виде набора коэффициентов, в данном случае (-1/8, 2/8, 6/8, 2/8, -1/8).

Графически это выглядит так, как на рис. 2.4 и 2.5.

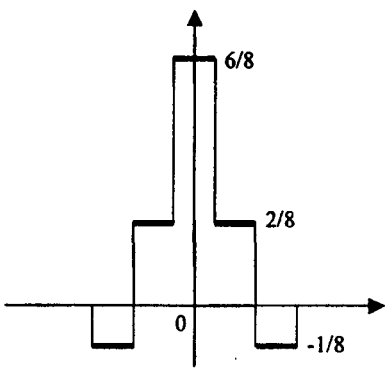


Рис. 2.4. ДВП для набора коэффициентов (–1/8, 2/8, 6/8, 2/8, –1/8)

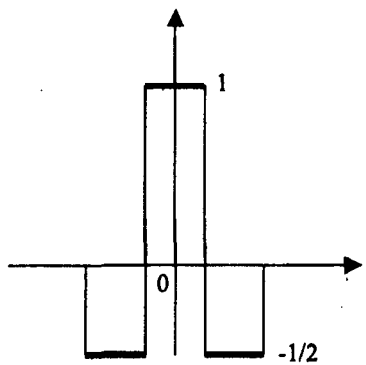


Рис. 2.5. ДВП для набора коэффициентов (–1/2, 1, –1/2)

Упражнение. Каким будет набор коэффициентов, если используем $hi_5(L[i])$?

Итак, при таком построении вэйвлет-фильтров формируется два выходных потока:

$$L[2\cdot x+1]=S[2\cdot x+1]+lo(S[2\cdot x+j]),\ j\text{ – четные: } j=\pm 2k; \qquad (2.11)$$

$$H[2\cdot x]=S[2\cdot x]+hi(L[2\cdot x+i]),\ i\text{ – нечетные: } i=\pm(2m+1).$$

Если сжимаем с потерями, функция lo может использовать любые S[x], а не только четные.

Обратное преобразование: сначала найдем все четные элементы:

$$S[2\cdot x]=H[2\cdot x]-hi(L[2\cdot x+i]),\ i\text{ – нечетные;}$$

затем, на основании найденных четных, восстановим все нечетные элементы:

$$S[2\cdot x+1]=L[2\cdot x+1]-lo(S[2\cdot x+j]),\ j\text{ – четные.}$$

Если, наоборот, берем четные L и нечетные H, это ничего по сути не меняет.

Заметим, что при сжатии с потерями существует и альтернативный путь. В прямом преобразовании сначала вычисляется

H[2·x+1]=C·S[2·x+1]+hi(S[2·x+j]), j – четные: j= ±2k, 0<C<1; затем

L[2·x]=S[2·x]+lo(H[2·x+i]), i – нечетные: i= ±(2m+1). Сравните с (2.1'1).

Например, если D=3, H[x]=(S[x]+(S[x-1]+S[x+1])/2) /2.

И еще заметим, что для улучшения сжатия применимо все то, что написано выше для SC. Но, кроме того, здесь, как и в ЛПК, можно через каждые K элементов выбирать фильтр, оптимальный для последних M обработанных элементов (K, M – задаваемые параметры). Причем метод может даже не записывать номер выбранного фильтра в поток в явном виде, поскольку аналогичный анализ может выполняться на этапе обратного преобразования.

Характеристики методов семейства SC:

Степень сжатия: увеличивается обычно в 1.1–1.9 раза.

Типы данных: методы предназначены для сжатия количественных данных.

Симметричность по скорости: в общем случае 1:1.

Характерные особенности: может быть целесообразно многократное применение.

Глава 3. Словарные методы сжатия данных

Идея словарных методов

Входную последовательность символов можно рассматривать как последовательность строк, содержащих произвольное количество символов. Идея словарных методов состоит в замене строк символов на такие коды, что их можно трактовать как индексы строк некоторого словаря. Образующие словарь строки будем далее называть *фразами*. При декодировании осуществляется обратная замена индекса на соответствующую ему фразу словаря.

Можно сказать, что мы пытаемся преобразовать исходную последовательность путем ее представления в таком алфавите, что его "буквы" являются фразами словаря, состоящими, в общем случае из произвольного количества символов входной последовательности.

Словарь – это набор таких фраз, которые, как мы полагаем, будут встречаться в обрабатываемой последовательности. Индексы фраз должны быть построены таким образом, чтобы в среднем их представление занимало меньше места, чем требуют замещаемые строки. За счет этого и происходит сжатие.

Уменьшение размера возможно в первую очередь за счет того, что обычно в сжимаемых данных встречается лишь малая толика всех возможных строк длины n, поэтому для представления индекса фразы требуется, как правило, меньшее число битов, чем для представления исходной строки. Например, рассмотрим количество взаимно различных строк длины от 1 до 5 в тексте на русском языке (роман Ф.М. Достоевского "Бесы", обычный неформатированный текст, размер около 1.3 Мб):

Длина строки	Количество различных строк	Использовано комбинаций, % от всех возможных
5	196969	0.0004
4	72882	0.0213
3	17481	0.6949
2	2536	13.7111
1	136	100.0000

Иначе говоря, размер (мощность) алфавита равен 136 символам, но реально используется только $2536/(136 \cdot 136) \cdot 100\% \approx 13.7$ % от всех возможных двухсимвольных строк и т. д.

Далее, если у нас есть заслуживающие доверия гипотезы о частоте использования тех или иных фраз либо проводился какой-то частотный анализ обрабатываемых данных, то мы можем назначить более вероятным фразам коды меньшей длины. Например, для той же электронной версии романа "Бесы" статистика встречаемости строк длины 5 имеет вид:

N	Количество строк длины 5, встретившихся ровно N раз	Количество относительно общего числа всех различных строк длины 5, %
1	91227	46.3
2	30650	15.6
3	16483	8.4
4	10391	5.3
5	7224	3.7
≥6	40994	20.7
Всего	**196969**	**100.0**

Заметим, что из всех 197 тыс. различных строк длины 5 почти половина встретилась лишь один раз, поэтому они вообще не будут использованы как фразы при словарном кодировании в том случае, если словарь строится только из строк обработанной части потока. Наблюдаемые частоты оставшейся части строк быстро уменьшаются с увеличением N, что указывает на выгодность применения статистического кодирования, когда часто используемым фразам ставятся в соответствие коды меньшей длины.

Обычно же просто предполагается, что короткие фразы используются чаще длинных. Поэтому в большинстве случаев индексы строятся таким

образом, чтобы длина индекса короткой фразы была меньше длины индекса длинной фразы. Такой прием обычно способствует улучшению сжатия.

Очевидно, что процессы моделирования и кодирования, рассматриваемые в гл. 4, для словарных методов сливаются. Моделирование в явном виде может выполняться уже только для индексов. Заметим, что апологеты идеи универсального моделирования и кодирования последовательно изучают любой метод, не вписывающийся явно в их модель, и обычно достаточно убедительно доказывают, что для него можно построить аналог в виде статистического метода. Так, например, доказано, что несколько рассматриваемых ниже словарных алгоритмов семейства Зива – Лемпела могут быть воспроизведены в рамках контекстного моделирования ограниченного порядка либо с помощью моделей состояний [6, 9].

Ниже будут рассмотрены алгоритмы словарного сжатия, относимые к классу методов Зива – Лемпела. В качестве примера словарного алгоритма иного класса укажем [7].

Методы Зива – Лемпела ориентированы на сжатие качественных данных, причем эффективность применения достигается в том случае, когда статистические характеристики обрабатываемых данных соответствуют модели источника с памятью.

Классические алгоритмы Зива – Лемпела

Алгоритмы словарного сжатия Зива-Лемпела появились во второй половине 70-х гг. Это были так называемые алгоритмы LZ77 и LZ78, разработанные совместно Зивом (Ziv) и Лемпелом (Lempel). В дальнейшем первоначальные схемы подвергались множественным изменениям, в результате чего мы сегодня имеем десятки достаточно самостоятельных алгоритмов и бессчетное количество модификаций.

LZ77 и LZ78 являются универсальными алгоритмами сжатия, в которых словарь формируется на основании уже обработанной части входного потока, т. е. адаптивно. Принципиальным отличием является лишь способ формирования фраз. В модификациях первоначальных алгоритмов это свойство сохраняется. Поэтому словарные алгоритмы Зива – Лемпела разделяют на два семейства – алгоритмы типа LZ77 и алгоритмы типа LZ78. Иногда также говорят о словарных методах LZ1 и LZ2.

Публикации Зива и Лемпела носили чисто теоретический характер, так как эти исследователи на самом деле занимались проблемой измерения "сложности" строки и применение выработанных алгоритмов к сжатию данных явилось скорее лишь частным результатом. Потребовалось некоторое время, чтобы идея организации словаря, часто в переложении уже других людей, достигла разра-

ботчиков программного и аппаратного обеспечения. Поэтому практическое использование алгоритмов началось спустя пару лет.

С тех пор методы данного семейства неизменно являются самыми популярными среди всех методов сжатия данных, хотя в последнее время ситуация начала меняться в пользу BWT и PPM, как обеспечивающих лучшее сжатие. Кроме того, практически все реально используемые словарные алгоритмы относятся к семейству Зива – Лемпела.

Необходимо сказать несколько слов о наименованиях алгоритмов и методов. При обозначении семейства общепринятой является аббревиатура LZ, но расшифровываться она должна как Ziv – Lempel, поэтому и алгоритмы Зива – Лемпела, а не Лемпела – Зива. Согласно общепринятому объяснению этого курьеза, Якоб Зив внес больший вклад в открытие соответствующих словарных схем и исследование их свойств и, таким образом, заслужил, чтобы первым стояла его фамилия, что мы и видим в заголовках статей [12, 13]. Но случайно была допущена ошибка, и прикрепилось сокращение LZ (буквы упорядочены в алфавитном порядке). Иногда, кстати, встречается и обозначение ZL (порядок букв соответствует порядку фамилий авторов в публикациях [12, 13]). В дальнейшем, если некий исследователь существенно изменял какой-то алгоритм, относимый к семейству LZ, то в названии полученной модификации к строчке LZ обычно дописывалась первая буква его фамилии, например: алгоритм LZB, автор Белл (Bell).

Подчеркнем также наличие большой путаницы с классификацией алгоритмов. Обычно она проявляется в нежелании признавать существование двух самостоятельных семейств LZ, а также в неправильном отнесении алгоритмов к конкретному семейству. Беспорядку часто способствуют сами разработчики: многим невыгодно раскрывать, на основе какого алгоритма создана данная модификация из-за коммерческих, патентных или иных меркантильных соображений. Например, в случае коммерческого программного обеспечения общепринятой является практика классификации используемого алгоритма сжатия как "модификации LZ77". И в этом нет ничего удивительного, ведь алгоритм LZ77 не запатентован.

Алгоритм LZ77

Этот словарный алгоритм сжатия является самым старым среди методов LZ. Описание было опубликовано в 1977 г. [12], но сам алгоритм разработан не позднее 1975 г.

Алгоритм LZ77 является родоначальником целого семейства словарных схем – так называемых *алгоритмов со скользящим словарем*, или *скользящим окном*. Действительно, в LZ77 в качестве словаря используется блок уже закодированной последовательности. Как правило, по мере выполнения

обработки положение этого блока относительно начала последовательности постоянно меняется, словарь "скользит" по входному потоку данных.

Скользящее окно имеет длину N, т. е. в него помещается N символов, и состоит из двух частей:

- последовательности длины $W = N-n$ уже закодированных символов, которая и является словарем;
- упреждающего буфера, или буфера предварительного просмотра (lookahead), длины n; обычно n на порядки меньше W.

Пусть к текущему моменту времени мы уже закодировали t символов S_1, S_2, ..., S_t. Тогда словарем будут являться W предшествующих символов $S_{t-(W-1)}$, $S_{t-(W-1)+1}$, ..., S_t. Соответственно, в буфере находятся ожидающие кодирования символы S_{t+1}, S_{t+2}, ..., S_{t+n}. Очевидно, что если $W \geq t$, то словарем будет являться вся уже обработанная часть входной последовательности.

Идея алгоритма заключается в поиске самого длинного совпадения между строкой буфера, начинающейся с символа S_{t+1}, и всеми фразами словаря. Эти фразы могут начинаться с любого символа $S_{t-(W-1)}$, $S_{t-(W-1)+1}$, ..., S_t и выходить за пределы словаря, вторгаясь в область буфера, но должны лежать в окне. Следовательно, фразы не могут начинаться с S_{t+1}, поэтому буфер не может сравниваться сам с собой. Длина совпадения не должна превышать размера буфера. Полученная в результате поиска фраза $S_{t-(i-1)}$, $S_{t-(i-1)+1}$, ..., $S_{t-(i-1)+(j-1)}$ кодируется с помощью двух чисел:

1) смещения (offset) от начала буфера, i;
2) длины соответствия, или совпадения (match length), j.

Смещение и длина соответствия играют роль указателя (ссылки), однозначно определяющего фразу. Дополнительно в выходной поток записывается символ s, непосредственно следующий за совпавшей строкой буфера.

Таким образом, на каждом шаге кодер выдает описание трех объектов: смещения и длины соответствия, образующих код фразы, равной обработанной строке буфера, и одного символа s (литерала). Затем *окно* смещается на $j+1$ символов вправо и осуществляется переход к новому циклу кодирования. Величина сдвига объясняется тем, что мы реально закодировали именно $j+1$ символов: j с помощью указателя на фразу в словаре и 1 с помощью тривиального копирования. Передача одного символа в явном виде позволяет разрешить проблему обработки еще ни разу не виденных символов, но существенно увеличивает размер сжатого блока.

Пример

Попробуем сжать строку "кот_ломом_колол_слона" длиной 21 символ. Пусть длина буфера равна семи символам, а размер словаря больше длины сжимаемой строки. Условимся также, что:

- нулевое смещение зарезервировали для обозначения конца кодирования;
- символ s_t соответствует единичному смещению относительно символа s_{t+1}, с которого начинается буфер;
- если имеется несколько фраз с одинаковой длиной совпадения, то выбираем ближайшую к буферу;
- в неопределенных ситуациях – когда длина совпадения нулевая – смещению присваиваем единичное значение.

Таблица 3.1

Шаг	Скользящее окно		Совпадающая фраза	Закодированные данные		
	Словарь	Буфер		i	j	s
1	-	кот_лом	-	1	0	"к"
2	к	от_ломо	-	1	0	"о"
3	ко	т_ломом	-	1	0	"т"
4	кот	_ломом_	-	1	0	"_"
5	кот_	ломом_к	-	1	0	"л"
6	кот_л	омом_ко	о	4	1	"м"
7	кот_лом	ом_коло	ом	2	2	"_"
8	кот_ломом_	колол_с	ко	10	2	"л"
9	кот_ломом_кол	ол_слон	ол	2	2	"_"
10	..._ломом_колол_	слона	-	1	0	"с"
11	...ломом_колол_с	лона	ло	5	2	"н"
12	...ом_колол_слон	а	-	1	0	"а"

Для кодирования i нам достаточно 5 бит, для j нужно 3 бита, и пусть символы требуют 1 байта для своего представления. Тогда всего мы потратим 12·(5+3+8) = 192 бита. Исходно строка занимала 21·8 = 168 бит, т. е. LZ77 кодирует нашу строку еще более расточительным образом. Не следует также забывать, что мы опустили шаг кодирования конца последовательности, который потребовал бы еще как минимум 5 бит (размер поля i = 5 битам).

Процесс кодирования можно описать следующим образом.

```
while ( ! DataFile.EOF() ){
  /*найдем максимальное совпадение; в match_pos получим
    смещение i, в match_len - длину j, в unmatched_sym -
  первый несовпавший символ s_{t+1+j}; считаем также, что в
    функции find_match учитывается ограничение на длину
    совпадения
  */
  find_match (&match_pos, &match_len, &unmatched_sym);
  /*запишем в файл сжатых данных описание найденной
```

```
      фразы, при этом длина битового представления i
      задается константой OFFS_LN, длина представления
      j - константой LEN_LN, размер символа s принимаем
      равным 8 битам
   */
   CompressedFile.WriteBits (match_pos, OFFS_LN);
   CompressedFile.WriteBits (match_len, LEN_LN);
   CompressedFile.WriteBits (unmatched_sym, 8);
   for (i = 0; i <= match_len; i++){
      // прочтем очередной символ
      c = DataFile.ReadSymbol();
      //удалим из словаря одну самую старую фразу
      DeletePhrase ();
      /*добавим в словарь одну фразу, начинающуюся с
        первого символа буфера
      */
      AddPhrase ();
      /*сдвинем окно на 1 позицию, добавим в конец буфера
        символ c
      */
      MoveWindow(c);
   }
}
CompressedFile.WriteBits (0, OFFS_LN);
```

Пример подтвердил, что способ формирования кодов в LZ77 неэффективен и позволяет сжимать только сравнительно длинные последовательности. До некоторой степени сжатие небольших файлов можно улучшить, используя коды переменной длины для смещения *i*. Действительно, даже если мы используем словарь в 32 Кб, но закодировали еще только 3 Кб, то смещение реально требует не 15, а 12 бит. Кроме того, происходит существенный проигрыш из-за использования кодов одинаковой длины при указании длин совпадения *j*. Например, для уже упоминавшейся электронной версии романа "Бесы" были получены следующие частоты использования длин совпадения:

j	Количество раз, когда максимальная длина совпадения была равна *j*
0	136
1	1593
2	4675
3	11165
4	20047
5	26939
6	28653

j	Количество раз, когда максимальная длина совпадения была равна j
7	24725
8	19702
9	14767
10	10820
≥11	27903

Из таблицы следует, что в целях минимизации закодированного представления для $j = 6$ следует использовать код наименьшей длины, так как эта длина совпадения встречается чаще всего.

Хотя авторы алгоритма и доказали, что LZ77 может сжать данные не хуже, чем любой специально на них настроенный полуадаптивный словарный метод, из-за указанных недостатков это выполняется только для последовательностей достаточно большого размера.

Что касается декодирования сжатых данных, то оно осуществляется путем простой замены кода на блок символов, состоящий из фразы словаря и явно передаваемого символа. Естественно, декодер должен выполнять те же действия по изменению окна, что и кодер. Фраза словаря элементарно определяется по смещению и длине, поэтому важным свойством LZ77 и прочих алгоритмов со скользящим окном является очень быстрая работа декодера.

Алгоритм декодирования может иметь следующий вид:

```
for (;;) {
   // читаем смещение
   match_pos = CompressedFile.ReadBits (OFFS_LN);
   if (!match_pos)
      // обнаружен признак конца файла, выходим из цикла
      break;
   // читаем длину совпадения
   match_len = CompressedFile.ReadBits (LEN_LN);
   for (i = 0; i < match_len; i++) {
      /*находим в словаре очередной символ совпавшей
        фразы
      */
      c = Dict (match_pos + i);
      /*сдвигаем словарь на одну позицию, добавляем в его
        начало c
      */
      MoveDict (c)
      /*записываем очередной раскодированный символ
        в выходной файл
      */
      DataFile.WriteSymbol (c);
```

```
    }
    /*читаем несовпавший символ, добавляем его в словарь
      и записываем в выходной файл
    */
    c = CompressedFile.ReadBits (8);
    MoveDict (c)
    DataFile.WriteSymbol (c);
}
```

Алгоритмы со скользящим окном характеризуются сильной несимметричностью по времени – кодирование значительно медленнее декодирования, поскольку при сжатии много времени тратится на поиск фраз.

Упражнение. Предложите несколько более эффективных способов кодирования результатов работы LZ77, чем использование простых кодов фиксированной длины.

Алгоритм LZSS

Алгоритм LZSS позволяет достаточно гибко сочетать в выходной последовательности символы и указатели (коды фраз), что до некоторой степени устраняет присущую LZ77 расточительность, проявляющуюся в регулярной передаче одного символа в прямом виде. Эта модификация LZ77 была предложена в 1982 г. Сторером (Storer) и Жимански (Szymanski) [10].

Идея алгоритма заключается в добавление к каждому указателю и символу 1-битового префикса *f*, позволяющего различать эти объекты. Иначе говоря, 1-битовый флаг *f* указывает тип и, соответственно, длину непосредственно следующих за ним данных. Такая техника позволяет:

- записывать символы в явном виде, когда соответствующий им код имеет большую длину и, следовательно, словарное кодирование только вредит;
- обрабатывать ни разу не встреченные до текущего момента символы.

Пример

Закодируем строку "кот_ломом_колол_слона" из предыдущего примера и сравним коэффициент сжатия для LZ77 и LZSS.

Пусть мы переписываем символ в явном виде, если текущая длина максимального совпадения буфера и какой-то фразы словаря меньше или равна единице. Если мы записываем символ, то перед ним выдаем флаг со значением 0, если указатель – то со значением 1. Если имеется несколько совпадающих фраз одинаковой длины, то выбираем ближайшую к буферу.

Таблица 3.2

Шаг	Скользящее окно		Совпадающая фраза	Закодированные данные			
	Словарь	Буфер		f	i	j	s
1	-	кот_лом	-	0	-	-	"к"
2	к	от_ломо	-	0	-	-	"о"
3	ко	т_ломом	-	0	-	-	"т"
4	кот	_ломом_	-	0	-	-	"_"
5	кот_	ломом_к	-	0	-	-	"л"
6	кот_л	омом_ко	о	0	-	-	"о"
7	кот_ло	мом_кол	-	0	-	-	"м"
8	кот_лом	ом_коло	ом	1	2	2	-
9	кот_ломом	_колол_	_	0	-	-	"_"
10	кот_ломом_	колол_с	ко	1	10	2	-
11	кот_ломом_ко	лол_сло	ло	1	8	2	-
12	...от_ломом_коло	л_слона	л	0	-	-	"л"
13	...т_ломом_колол	_слона	_	0	-	-	"_"
14	..._ломом_колол_	слона	-	0	-	-	"с"
15	...ломом_колол_с	лона	ло	1	5	2	-
16	...мом_колол_сло	на	-	0	-	-	"н"
17	...ом_колол_слон	а	-	0	-	-	"а"

Таким образом, для кодирования строки по алгоритму LZSS нам потребовалось 17 шагов: 13 раз символы были переданы в явном виде и 4 раза мы применили указатели. Заметим, что при работе по алгоритму LZ77 нам потребовалось всего лишь 12 шагов. С другой стороны, если задаться теми же длинами для *i* и *j*, то размер закодированных по LZSS данных равен 13·(1+8) + 4·(1+5+3) = 153 битам. Это означает, что строка действительно была сжата, так как ее исходный размер 168 бит.

Рассмотрим алгоритм сжатия подробнее.

```
const int
   // порог для включения словарного кодирования
   THRESHOLD = 2,
   // размер представления смещения, в битах
   OFFS_LN = 14,
   // размер представления длины совпадения, в битах
   LEN_LN = 4;
const int
   WIN_SIZE = (1 << OFFS_LN), // размер окна
   BUF_SIZE = (1 << LEN_LN) - 1; // размер буфера
//функция вычисления реального положения символа в окне
```

```
inline int MOD (int i) { return i & (WIN_SIZE-1); };

...
//собственно алгоритм сжатия
int buf_sz = BUF_SIZE;
/* инициализация: заполнение буфера, поиск совпадения
    для первого шага
*/
...
while ( buf_sz ) {
   if ( match_len > buf_size) match_len = buf_size;
   if ( match_len <= THRESHOLD ) {
      /*если длина совпадения меньше порога (2 в примере),
        то запишем в файл сжатых данных флаг и символ;
        pos определяет позицию начала буфера
      */
      CompressedFile.WriteBit (0);
      CompressedFile.WriteBits (window [pos], 8);
      // это понадобится при обновлении словаря
      match_len = 1;
   }else{
      /*иначе запишем флаг и указатель, состоящий из
        смещения и длины совпадения
      */
      CompressedFile.WriteBit (1);
      CompressedFile.WriteBits (match_offs, OFFS_LN);
      CompressedFile.WriteBits (match_len, LEN_LN);
   }
   for (int i = 0; i < match_len; i++) {
      /*удалим из словаря фразу, начинающуюся в позиции
        MOD (pos+buf_sz)
      */
      DeletePhrase ( MOD (pos+buf_sz) );
      if ( (c = DataFile.ReadSymbol ()) == EOF)
        // мы в конце файла, надо сократить буфер
        buf_sz--;
      else
        /*иначе надо добавить в конец буфера новый
          символ
        */
        window [MOD (pos+buf_sz)] = c;
      pos = MOD (pos+1); // сдвиг окна на 1 символ
      if (buf_sz)
        /*если буфер не пуст, то добавим в
          словарь новую фразу, начинающуюся в позиции pos;
          считаем, что в функции AddPhrase одновременно
```

```
            выполняется поиск максимального совпадения
            между буфером и фразами словаря
         */
         AddPhrase (pos, &match_offs, &match_len)
      ;
   }
}
CompressedFile.WriteBit (1);
CompressedFile.WriteBits (0, OFFS_LN); // знак конца файла
```

Скользящее окно можно реализовывать с помощью "циклического" массива, что и было проделано в вышеприведенном учебном фрагменте программы сжатия. Использованный подход не является лучшим, но сравнительно прост для понимания.

Алгоритм декодирования может быть реализован следующим образом:

```
for (;;) {
   if ( !CompressedFile.ReadBit () ){
      /*это символ, просто выведем его в файл и запишем в
         конец словаря (символ будет соответствовать смещению
         i = 1)
      */
      c = CompressedFile.ReadBits (8);
      DataFile.WriteSymbol (c);
      window [pos] = c;
      pos = MOD (pos+1);
   }else{
      // это указатель, прочитаем его
      match_pos = CompressedFile.ReadBits (OFFS_LN);
      if (!match_pos)
         break; // конец файла
      match_pos = MOD(pos - match_pos);
      match_len = CompressedFile.ReadBits (LEN_LN);
      // цикл копирования совпавшей фразы словаря в файл
      for (int i = 0; i < match_len; i++) {
         //выдаем очередной совпавший символ c
         c = window [MOD (match_pos+i)];
         DataFile.WriteSymbol (c);
         window [pos] = c;
         pos = MOD (pos+1);
      }
   }
}
```

Упражнение. Из-за наличия порога THRESHOLD часть допустимых значений длины реально не используется, поэтому размер буфера BUF_SIZE может быть увеличен при неизменном LEN_LN. Проделайте соответствующие модификации фрагментов программ кодирования и декодирования.

Алгоритм LZ78

Алгоритм LZ78 был опубликован в 1978 г. [13] и впоследствии стал "отцом" семейства словарных методов LZ78.

Алгоритмы этой группы не используют скользящего окна и в словарь помещают не все встречаемые при кодировании строки, а лишь "перспективные" с точки зрения вероятности последующего использования. На каждом шаге в словарь вставляется новая фраза, которая представляет собой сцепление (конкатенацию) одной из фраз *S* словаря, имеющей самое длинное совпадение со строкой буфера, и символа *s*. Символ *s* является символом, следующим за строкой буфера, для которой найдена совпадающая фраза *S*. В отличие от семейства LZ77 в словаре не может быть одинаковых фраз.

Кодер порождает только последовательность кодов фраз. Каждый код состоит из номера (индекса) *n* "родительской" фразы *S*, или префикса, и символа *s*.

В начале обработки словарь пуст. Далее, теоретически, словарь может расти бесконечно, т. е. на его рост сам алгоритм не налагает ограничений. На практике при достижении определенного объема занимаемой памяти словарь должен очищаться полностью или частично.

Пример

И еще раз закодируем строку "кот_ломом_колол_слона" длиной 21 символ. Для LZ78 буфер в принципе не требуется, поскольку достаточно легко так реализовать поиск совпадающей фразы максимальной длины, что последовательность незакодированных символов будет просматриваться только один раз. Поэтому буфер показан только с целью большей доходчивости примера. Фразу с номером 0 зарезервируем для обозначения конца сжатой строки, номером 1 будем задавать пустую фразу словаря.

Таблица 3.3

Шаг	Добавляемая в словарь фраза		Буфер	Совпадающая фраза *S*	Закодированные данные	
	Сама фраза	Ее номер			*n*	*s*
1	к	2	кот_лом	-	1	"к"
2	о	3	от_ломо	-	1	"о"
3	т	4	т_ломом	-	1	"т"
4	_	5	_ломом_	-	1	"_"

Шаг	Добавляемая в словарь фраза		Буфер	Совпадающая фраза *S*	Закодированные данные	
	Сама фраза	Ее номер			*n*	*s*
5	л	6	ломом_к	-	1	"л"
6	ом	7	омом_ко	о	3	"м"
7	ом_	8	ом_коло	ом	7	"_"
8	ко	9	колол_с	к	2	"о"
9	ло	10	лол_сло	л	6	"о"
10	л_	11	л_слона	л	6	"_"
11	с	12	слона	-	1	"с"
12	лон	13	лона	ло	10	"н"
13	а	14	а	-	1	"а"

Строку удалось закодировать за 13 шагов. Так как на каждом шаге выдавался один код, сжатая последовательность состоит из 13 кодов. Возможно использование 15 номеров фраз (от 0 до 14), поэтому для представления *n* посредством кодов фиксированной длины нам потребуется 4 бита. Тогда размер сжатой строки равен 13·(4+8) = 156 битам.

Ниже приведен пример реализации алгоритма сжатия LZ78.

```
n = 1;
while ( ! DataFile.EOF() ){
   s = DataFile.ReadSymbol; // читаем очередной символ
   /*пытаемся найти в словаре фразу, представляющую
      собой конкатенацию родительской фразы с номером n и
      символа s; функция возвращает номер искомой фразы
      в phrase_num; если же фразы нет, то phrase_num
      принимает значение 1, т. е. указывает на пустую фразу
   */
   FindPhrase (&phrase_num, n, s);
   if (phrase_num != 1)
      /*такая фраза имеется в словаре, продолжим поиск
         совпадающей фразы максимальной длины
      */
      n = phrase_num;
   else {
      /*такой фразы нет, запишем в выходной файл код;
         INDEX_LN - это константа, определяющая длину
         битового представления номера n
      */
```

```
      CompressedFile.WriteBits (n, INDEX_LN);
      CompressedFile.WriteBits (s, 8);
      AddPhrase (n, s); // добавим фразу в словарь
      n = 1; // подготовимся к следующему шагу
   }
}
// признак конца файла
CompressedFile.WriteBits (0, INDEX_LN);
```

При декодировании необходимо обеспечивать такой же порядок обновления словаря, как и при сжатии. Реализуем алгоритм следующим образом:

```
for (;;){
   // читаем индекс родительской фразы
   n = CompressedFile.ReadBits (INDEX_LN);
   if (!n)
      break; // конец файла
   // читаем несовпавший символ s
   s = CompressedFile.ReadBits (8);
   /*находим в словаре позицию начала фразы с индексом n
      и ее длину
   */
   GetPhrase (&pos, &len, n)
   /*записываем фразу с индексом n в файл
      раскодированных данных
   */
   for (i = 0; i < len; i++)
      DataFile.WriteSymbol (Dict[pos+i]);
   // записываем в файл символ s
   DataFile.WriteSymbol (s);
   AddPhrase (n, s); // добавляем новую фразу в словарь
}
```

Очевидно, что скорость раскодирования для алгоритмов семейства LZ78 потенциально всегда меньше скорости для алгоритмов со скользящим окном, так как в случае последних затраты по поддержанию словаря в правильном состоянии минимальны. С другой стороны, для LZ78 и его потомков, например LZW, существуют эффективные реализации процедур поиска и добавления фраз в словарь, что обеспечивает значительное преимущество над алгоритмами семейства LZ77 в скорости сжатия.

Несмотря на относительную быстроту кодирования LZ78, при грамотной реализации алгоритма оно все же медленнее декодирования, соотношение скоростей равно обычно 3:2.

Интересное свойство LZ78 заключается в том, что если исходные данные порождены источником с определенными характеристиками (он дол-

жен быть стационарным[1] и эргодическим[2]), то коэффициент сжатия приближается по мере кодирования к минимальному достижимому [13]. Иначе говоря, количество битов, затрачиваемых на кодирование каждого символа, в среднем равно так называемой *энтропии источника*. Но, к сожалению, сходимость медленная и на данных реальной длины алгоритм ведет себя не лучшим образом. Так, например, коэффициент сжатия текстов в зависимости от их размера обычно колеблется от 3.5 до 5 бит/символ. Кроме того нередки ситуации, когда обрабатываемые данные порождены источником с ярко выраженной нестационарностью. Поэтому при оценке реального поведения алгоритма следует относиться с большой осторожностью к теоретическим выкладкам, обращая внимание на выполнение соответствующих условий.

Доказано, что аналогичным свойством сходимости обладает и классический алгоритм LZ77, но скорость приближения к энтропии источника меньше, чем у алгоритма LZ78 [12].

Другие алгоритмы LZ

В табл. 3.4 приведены несколько достаточно характерных алгоритмов LZ. Указанный список далеко не полон, реально существует в несколько раз больше алгоритмов, основанных либо на LZ77, либо на LZ78. Известны и гибридные схемы, сочетающие оба подхода к построению словаря.

Таблица 3.4

№	Названия	Авторы, год	Тип алгоритма
1	LZMW	Миллер (Miller) и Уэгнам (Wegman), 1984	Алгоритм семейства LZ78
2	LZW	Уэлч (Welch), 1984	Модификация LZ78
3	LZB	Белл (Bell), 1987	Модификация LZSS
4	LZH	Брент (Brent), 1987	Модификация LZSS
5	LZFG	Файэлэ (Fiala) и Грини (Greene), 1989	Модификация LZ77
6	LZBW	Бендер (Bender) и Вулф (Wolf), 1991	Способ модификация алгоритмов семейства LZ77
7	LZRW1	Уилльямс (Williams), 1991	Модификация LZSS
8	LZCB	Блум (Bloom), 1995	Модификация LZ77

[1] *Многомерные распределения вероятностей генерации последовательностей (слов) из* n *символов не меняются во времени, причем* n *– любое конечное число.*

[2] *Среднее по времени равно среднему по числу реализаций; иначе говоря, для оценки свойств источника достаточно только одной длинной сгенерированной последовательности.*

№	Названия	Авторы, год	Тип алгоритма
9	LZP	Блум (Bloom), 1995	Основан на LZ77
10	LZ77-PM, LZFG-PM, LZW-PM	Хоанг (Hoang), Лонг (Long) и Виттер (Vitter), 1995	Модификации алгоритмов LZ

1. **LZMW.** Алгоритм семейства LZ78. Интересен способом построения словаря: новая фраза создается с помощью конкатенации двух последних использованных фраз, а не конкатенации фразы и символа, т. е. словарь наполняется "агрессивнее". Практические реализации LZMW в программах универсального сжатия неизвестны. Причина, по-видимому, состоит в том, что усложнение алгоритма не приводит к адекватному улучшению сжатия по сравнению с исходным LZW. С другой стороны, выгода от быстрого заполнения и обновления словаря проявляется главным образом при обработке неоднородных данных. Но для сжатия данных такого типа заведомо лучше подходит метод скользящего словаря (семейство LZ77).
2. **LZW.** Модификация LZ78. За счет предварительного занесения в словарь всех символов алфавита входной последовательности результат работы LZW состоит только из последовательности индексов фраз словаря. Из-за устранения необходимости регулярной передачи одного символа в явном виде LZW обеспечивает лучшее сжатие, чем LZ78. Подробнее об LZW см. в разд. 2, а также в [11].
3. **LZB.** Модификация LZSS. Изменения затрагивают кодирование указателей. Смещение задается переменным количеством битов в зависимости от реального размера словаря (в начале сжатия он мал). Длина совпадения записывается γ-кодами Элиаса. И первый и второй механизм часто применяются при разработке простых и обеспечивающих высокую скорость алгоритмов семейства LZ77.
4. **LZH.** Модификация LZSS. Аналогична LZB, но для сжатия смещений и длин соответствия используются коды Хаффмана. Заметим, что большинство современных архиваторов также применяют кодирование по методу Хаффмана в этих целях.
5. **LZFG.** Модификация LZ77. На самом деле представляет собой несколько алгоритмов. Идея самого сложного (C2 в обозначении авторов LZFG) заключается в кодировании фразы не парой <длина, смещение>, а индексом соответствующего фразе узла дерева цифрового поиска. Одинаковые фразы словаря имеют один и тот же индекс, что и обеспечивает более экономное кодирование строк. Алгоритмы LZFG не получили распространения на практике. В значительной степени этому способствовали патентные ограничения.

6. **LZBW.** Способ модификации алгоритмов семейства LZ77. За счет учета имеющихся в словаре одинаковых фраз позволяет уменьшить количество битов, требуемых для кодирования длины совпадения. Подробнее см. в подразд. "Пути улучшения сжатия алгоритмов LZ".
7. **LZRW1.** Алгоритм является модификацией LZSS, точнее, алгоритма A1 группы LZFG и разработан с целью обеспечения максимальной скорости сжатия и разжатия. Коды фраз состоят из 16 бит: 12 бит указывают смещение i и 4 бита задают длину совпадения j, а символы s передаются как байты (требуют 8 бит). Флаги f задаются сразу для последовательности из 16 кодов и литералов, т. е. сначала выдается 2-байтовое слово значений флагов, затем группа из 16 кодов и/или литералов. Для поиска в словаре при сжатии используется хеш-таблица со смешиванием по трем символам. Хеш-цепочки как таковые отсутствуют, т. е. каждая новая фраза заменяет в таблице старую с таким же значением хеш-функции. За счет устранения побитового ввода-вывода и использования словаря малого размера достигается высокая скорость кодирования и декодирования. Степень сжатия LZRW1 равна примерно 1.5–2.
8. **LZCB.** По сути, целая группа алгоритмов, являющихся той или иной модификацией LZ77. Основная идея заключается во введении достаточно сильного ограничения на минимальную длину совпадения – от четырех символов и более. Если фраза удовлетворяющей длины не найдена, то кодируется один символ (литерал). Характер типичных данных таков, что литералы и успешно закодированные с помощью словаря строки имеют тенденцию объединяться в группы с себе подобными, т. е., например, на выходе LZCB могут появиться 10 литералов подряд, затем 5 закодированных строк, затем опять несколько литералов и т. д. Эта особенность позволяет реализовать достаточно эффективное статистическое кодирование потоков литералов и указателей фраз. Тем не менее, растеряв преимущество в скорости, LZCB уступает современным алгоритмам PPM по коэффициенту сжатия.
9. **LZP.** Основан на LZ77. Для каждого входного символа строка из предшествующих L символов рассматривается как контекст C длины N. С помощью хеш-функции в словаре находится одна из совпадающих с контекстом C фраз, назовем эту фразу C': $C' = C$. Строка S буфера сравнивается с фразой, непосредственно следующей за C'. Если длина совпадения $L > 0$, то выдается флаг успеха $f = 1$ и S кодируется через длину L. Так как C' находится детерминированным образом, то смещение кодировать не надо. Если $L = 0$ или C' не была найдена, то выдается $f = 0$ и первый символ S кодируется непосредственно. Декодер должен использовать такой же механизм для поиска C'. Изложенный алгоритм спра-

ведлив для алгоритма LZP1, достоинством которого является высокая скорость. В более сложных модификациях процесс поиска C' повторяется для контекста длины N–1 и т. д. В LZP1 литералы просто копируются, а для кодирования флагов и длин используются коды переменной длины. В LZP3 применяется достаточно сложная схема кодирования потоков флагов, длин и литералов на основе алгоритма Хаффмана. В сочетании с PPM техника LZP обеспечивает высокую степень сжатия при неплохих скоростных характеристиках.

10. **LZ77-PM, LZFG-PM, LZW-PM.** Модификации алгоритмов LZ. Используется несколько контекстнозависимых словарей, а не один словарь. В контекстнозависимый словарь порядка L с номером i попадают только строки, встречаемые после контекста[1] C_i длины L. Кодирование строки буфера S производится с помощью одного или нескольких словарей, номера которых определяются последними закодированными перед S символами. Учет контекста позволяет существенно улучшить сжатие исходных словарных схем. Подробнее см. в подразд. "Пути улучшения сжатия алгоритмов LZ".

В табл. 3.5 приведены результаты сравнения нескольких алгоритмов по степени сжатия на наборе CalgCC.

Таблица 3.5

	LZP1	LZSS	LZW	LZB	LZW-PM	LZFG	LZFG-PM	LZP3
Bib	1.98	2.81	2.48	2.52	3.07	2.76	3.28	3.32
Book1	1.42	2.33	2.52	2.07	2.92	2.21	2.44	2.50
Book2	1.76	2.68	2.61	2.44	3.14	2.62	2.88	3.01
Geo	1.19	1.19	1.38	1.30	1.23	1.40	1.42	1.51
News	1.76	2.28	2.21	2.25	2.52	2.33	2.46	2.78
Obj1	1.55	1.57	1.58	1.88	1.56	1.99	1.85	1.83
Obj2	2.21	2.28	1.96	2.55	2.34	2.70	2.63	2.79
Paper1	1.72	2.47	2.21	2.48	2.60	2.64	2.92	2.73
Paper2	1.62	2.50	2.37	2.33	2.72	2.53	2.86	2.72
Pic	6.30	3.98	8.51	7.92	8.16	9.20	8.60	9.30
Progc	1.86	2.44	2.15	2.60	2.52	2.77	2.92	2.73
Progl	2.66	3.28	2.71	3.79	3.38	4.06	4.44	4.19
Progp	2.82	3.31	2.67	3.85	3.33	4.21	4.42	3.98
Trans	3.27	3.46	2.53	3.77	3.46	4.55	4.79	4.79
Итого	**2.29**	**2.61**	**2.71**	**2.98**	**3.07**	**3.28**	**3.42**	**3.44**

[1] *Контекст – это в данном случае конечная последовательность символов. См. также главу "Сжатие данных с помощью контекстных методов моделирования".*

Видно, что за счет оптимизации структуры словаря достигается значительное улучшение сжатия. Но, с другой стороны, сложные алгоритмы построения словаря обычно существенно замедляют работу декодера, что сводит на нет достоинства LZ.

Приведенные ниже характеристики степени сжатия и скорости алгоритмов семейств LZ77 и LZ78 относятся к типичным представителям семейств – LZH и LZW соответственно.

Характеристики алгоритмов семейства LZ77:

Степени сжатия: определяются данными, обычно 2–4.

Типы данных: алгоритмы универсальны, но лучше всего подходят для сжатия разнородных данных, например файлов ресурсов.

Симметричность по скорости: примерно 10:1; если алгоритм обеспечивает хорошее сжатие, то декодер обычно гораздо быстрее кодера.

Характерные особенности: обычно медленно сжимают высокоизбыточные данные; из-за высокой скорости разжатия идеально подходят для создания дистрибутивов программного обеспечения.

Характеристики алгоритмов семейства LZ78:

Степени сжатия: определяются данными, обычно 2–3.

Типы данных: алгоритмы универсальны, но лучше всего подходят для сжатия текстов и тому подобных однородных данных, например рисованных картинок; плохо сжимают разнородные данные.

Симметричность по скорости: примерно 3:2, декодер обычно в полтора раза быстрее кодера.

Характерные особенности: из-за относительно небольшой степени сжатия и невысокой скорости декодирования уступают по распространенности алгоритмам семейства LZ77.

Формат Deflate

Формат словарного сжатия Deflate, предложенный Кацем (Katz), используется популярным архиватором PKZIP [3]. Сжатие осуществляется с помощью алгоритма типа LZH, иначе говоря, указатели и литералы кодируются по методу Хаффмана. Формат специфицирует только работу декодера, т. е. определяет алгоритм декодирования, и не налагает серьезных ограничений на реализацию кодера. В принципе в качестве алгоритма сжатия может применяться любой работающий со скользящим окном, лишь бы он исходил из стандартной процедуры обновления словаря для алгоритмов семейства LZ77 и использовал задаваемые форматом типы кодов Хаффмана.

Особенности формата:

- является универсальным, т. е. не ориентирован на конкретный тип данных;
- прост в реализации;
- де-факто является одним из промышленных стандартов на сжатие данных.

Существует множество патентов, покрывающих весь формат полностью или какие-то детали его реализации. С другой стороны, Deflate используется в огромном количестве программных и аппаратных приложений и отработаны методы защиты в суде в случае предъявления соответствующего иска от компании, обладающей патентом на формат или его часть.

Поучительная ремарка. *В 1994 г. корпорация Unisys "вспомнила" о своем патенте в США на алгоритм LZW, зарегистрированном в 1985 г., и объявила о незаконности использования LZW без соответствующей лицензии. В частности, Unisys заявила о нарушении своих прав как патентовладельца в случае нелицензированного использования алгоритма LZW в формате GIF. Было объявлено, что производители, программы или аппаратное обеспечение которых читают или записывают файлы в формате GIF, должны покупать лицензию на использование, а также выплачивать определенный процент с прибыли при коммерческом применении. Далее Unisys в лучших традициях тоталитарной пропаганды последовательно продолжала переписывать историю, неоднократно меняя задним числом свои требования к лицензируемым продуктам и условия оплаты. В частности, было оговорено, что плата взимается только в случае коммерческих продуктов, но в любом случае требуется получить официальное разрешение от Unisys. Но, с другой стороны, Unisys требует у всех владельцев Интернет (интранет)-сайтов, использующих рисунки в формате GIF, приобрести лицензию стоимостью порядка $5000 в том случае, если ПО, с помощью которого были созданы эти файлы GIF, не имеет соответствующей лицензии Unisys. С точки зрения некоторых независимых экспертов по патентному праву, чтение (декодирование) GIF-файлов не нарушает права Unisys, но, судя по всему, сама корпорация придерживается другой точки зрения. Также достаточно странно поведение корпорации CompuServ, разработавшей формат GIF и опубликовавшей его в 1987 г., т. е. уже после регистрации патента на LZW, как открытый и свободный от оплаты. По состоянию на 2001 г., LZW запатентован Unisys по меньшей мере в следующих странах: США, Канаде, Великобритании, Германии, Франции, Японии. Текущее состояние дел можно выяснить на сайте компании www.unisys.com. Срок действия основного патента в США истекает не ранее 19 июня 2003 г.*

Общее описание

Закодированные в соответствии с форматом Deflate данные представляют собой набор блоков, порядок которых совпадает с последовательностью соответствующих блоков исходных данных. Используется 3 типа блоков закодированных данных:

1) состоящие из несжатых данных;
2) использующие фиксированные коды Хаффмана;
3) использующие динамические коды Хаффмана.

Длина блоков первого типа не может превышать 64 Кб, относительно других ограничений по размеру нет. Каждый блок типа 2 и 3 состоит из двух частей:

- описания двух таблиц кодов Хаффмана, использованных для кодирования данных блока;
- собственно закодированных данных.

Коды Хаффмана каждого блока не зависят от использованных в предыдущих блоках.

Само описание динамически создаваемых кодов Хаффмана является, в свою очередь, также сжатым с помощью фиксированных кодов Хаффмана, таблица которых задается форматом.

Алгоритм словарного сжатия может использовать в качестве словаря часть предыдущего блока (блоков), но величина смещения не может быть больше 32 Кб. Данные в компактном представлении состоят из кодов элементов двух типов:

- литералов (одиночных символов);
- указателей имеющихся в словаре фраз; указатели состоят из пары <длина совпадения, смещение>.

Длина совпавшей строки не может превышать 258 байт, а смещение фразы – 32 Кб. Литералы и длины совпадения кодируются с помощью одной таблицы кодов Хаффмана, а для смещений используется другая таблица; иначе говоря, литералы и длины совпадения образуют один алфавит. Именно эти таблицы кодов и передаются в начале блока третьего типа.

Алгоритм декодирования

Сжатые данные декодируются по следующему алгоритму.

```
do{
   Block.ReadHeader (); // читаем заголовок блока
   /*определяем необходимые действия по разжатию в
     соответствии с типом блока
   */
   switch (Block.Type) {
   case NO_COMP:// данные не сжаты, просто копируем их
     /*заголовок блока не выровнен на границу байта,
       сделаем это
     */
     Block.SeekNextByte ();
```

```
      Block.ReadLen (); // читаем длину блока
      /*копируем данные блока из входного файла
        сжатых данных в результирующий DataFile
      */
      PutRawData (Window, Block, DataFile);
      break;
    case DYN_HUF:
      /*блок данных сжат с помощью динамически
        построенных кодов Хаффмана, прочитаем их
      */
      Block.ReadHuffmanCodes ();
    case FIXED_HUF:
      for (;;) {
        /*прочтем один символ алфавита литералов и
          длин совпадения
        */
        value = Block.DecodeSymbol ();
        if ( value < 256)
          // это литерал, запишем его в выходной файл
          DataFile.WriteSymbol (value);
        else if ( value == 256)
          // знак конца блока
          break;
        else {
          // это закодированный указатель
          match_len = Block.DecodeLen ();
          match_pos = Block.DecodePos ();
          /*скопируем соответствующую фразу из словаря
            в выходной файл
          */
          CopyPhrase (Window, match_len, match_pos,
                      DataFile);
        }
      };
      break;
    default:
      // Ошибка в блоке данных
      throw BadData (Block);
}while ( !IsLastBlock );
```

Предполагается, что используемые алгоритмы поддерживают правильное обновление скользящего окна.

Кодирование длин и смещений

Как уже указывалось, литералы и длины совпадения объединены в единый алфавит символов со значениями {0, 1, ..., 285}, так что 0–255 отведены под литералы, 256 указывает на конец текущего блока, а 257–285 определяют длины совпадения. Код длины совпадения состоит из кода числа, лежащего в диапазоне 257–285 (базы длины совпадения), и, возможно, дополнительно читаемых битов, непосредственно следующих за кодом этого числа. База определяет квантованную длину совпадения, поэтому одному значению базы может соответствовать несколько длин. Значение поля дополнительно читаемых битов используется для доопределения длины совпадения. Таблица отображения значений кодов в длины фраз приведена ниже (табл. 3.6).

Таблица 3.6

Значение базы	Число доп. битов	Длина совпадения	Значение базы	Число доп. битов	Длина совпадения
257	0	3	272	2	31...34
258	0	4	273	3	35...42
259	0	5	274	3	43...50
260	0	6	275	3	51...58
261	0	7	276	3	59...66
262	0	8	277	4	67...82
263	0	9	278	4	83...98
264	0	10	279	4	99...114
265	1	11,12	280	4	115...130
266	1	13,14	281	5	131...162
267	1	15,16	282	5	163...194
268	1	17,18	283	5	195...226
269	2	19...22	284	5	227...257
270	2	23...26	285	0	258
271	2	27...30			

Поле дополнительных битов представляет собой целое число заданной длины, в котором первым записан самый старший бит. Длина совпадения 258 кодируется с помощью небольшого числа битов, поскольку это максимально допустимая в Deflate длина совпадения, и такой прием позволяет увеличить степень сжатия высокоизбыточных файлов.

Таким образом, функция Block.DecodeSymbol, упомянутая в предыдущем подразд., читает символ, который может быть либо литералом, либо знаком конца блока, либо базой совпадения. В последнем случае, т. е. когда значение символа > 256, дополнительные биты читаются с помощью функции Block.DecodeLen.

Представление смещений также состоит из двух полей: базы и поля дополнительных битов (табл. 3.7). Объединение смещений в группы позволяет использовать достаточно эффективные коды Хаффмана небольшой длины, канонический алгоритм построения которых обеспечивает быстрое декодирование. Объединение целесообразно также потому, что распределение величин смещений в отдельной группе обычно носит случайный характер.

Таблица 3.7

Значение базы	Число доп. битов	Значение смещения	Значение базы	Число доп. битов	Значение смещения
0	0	1	15	6	193...256
1	0	2	16	7	257...384
2	0	3	17	7	385...512
3	0	4	18	8	513...768
4	1	5,6	19	8	769...1024
5	1	7,8	20	9	1025...1536
6	2	9...12	21	9	1537...2048
7	2	13...16	22	10	2049...3072
8	3	17...24	23	10	3073...4096
9	3	25...32	24	11	4097...6144
10	4	33...48	25	11	6145...8192
11	4	49...64	26	12	8193...12288
12	5	65...96	27	12	12289...16384
13	5	97...128	28	13	16385...24576
14	6	129...192	29	13	24577...32768

Функция `Block.DecodePos` должна выполнить два действия: прочитать базу смещения и, на основании значения базы, прочитать необходимое количество дополнительных битов. Как и в случае литералов/длин совпадения, дополнительные биты соответствуют целым числам заданной длины, в которых первым записан самый старший бит.

Кодирование блоков фиксированными кодами Хаффмана

В этом случае для сжатия символов алфавита литералов и длин совпадения используются заранее построенные коды Хаффмана, известные кодеру и декодеру, т. е. нам не нужно передавать их описания. Длины кодов определяются значением символов (табл. 3.8).

Таблица 3.8

Значение символа	Длина кода, бит	Значение кода (в двоичной системе счисления)
0–143	8	00110000 ... 10111111
144–255	9	110010000 ... 111111111
256–279	7	0000000 ... 0010111
280–287	8	11000000 ... 11000111

Символы со значениями 286 и 287 не должны появляться в сжатых данных, хотя для них и отведено кодовое пространство.

Базы смещений кодируются 5-битовыми числами так, что 00000 соответствует 0, 11111 – 31. При этом запрещается использовать базы со значениями 30 и 31, так как их смысл не определен.

Пример

Покажем, как выглядит в закодированном виде такая последовательность замещенных строк и литерала:

Элемент	Смещение *i*	Длина совпадения *j*	Литерал
1	260	5	-
2	45	20	-
3	-	-	3

Длина совпадения раскладывается на два поля, поэтому для $j = 5$ получаем (см. табл. 3.6):

Длина совпадения	Соответствующий символ в алфавите литералов/длин	База	Число дополнительных битов
5	259	259	0

Длина совпадения кодируется как 0000011 (см. табл. 3.8).

В общем случае смещение состоит также из двух полей, и для $i = 260$ получаем:

Смещение	База	Число дополнительных битов	Значение дополнительных битов
260	16	7	3

Смещение 260 записывается как последовательность битов 10000_0000011 (подчеркиванием показана граница чисел), где первое число – база, второе – поле дополнительных битов, равное 260 – 257 = 3.

Действуя аналогичным образом, на основании табл. 3.6, 3.7, 3.8 находим отображение всей последовательности:

Элемент	Последовательность битов	Комментарии
1	0000011 10000_0000011	База длины совпадения = 259. Поле дополнительных битов совпадения отсутствует. База смещения = 16. Поле дополнительных битов смещения = 3. Длина поля = 7
2	0001101_01 01010_1100	База длины совпадения = 269. Поле дополнительных битов совпадения = 1. Длина поля = 2. База смещения = 10. Поле дополнительных битов смещения = 12. Длина поля = 4
3	00110011	Литерал = 3

Кодирование блоков динамически создаваемыми кодами Хаффмана

В этом случае перед собственно сжатыми данными передается описание кодов литералов/длин и описание кодов смещений. В методе Deflate используется канонический алгоритм Хаффмана, поэтому для указания кода символа достаточно задать только длину этого кода. Неявным параметром является положение символа в упорядоченном списке символов. Таким образом, динамические коды Хаффмана описываются цепочкой длин кодов, упорядоченных по значению соответствующего кодам числа (литерала/длины совпадения в одном случае и смещения в другом). При этом алфавит *CWL* (codeword lengths) длин кодов имеет вид, описанный в табл. 3.9.

Таблица 3.9

Значение символа алфавита CWL	Что определяет
0...15	Соответствует длинам кодов от 0 до 15
16	Копировать предыдущую длину кода x раз, где x определяется значением 2 битов, читаемых после кода символа 16; можно указать необходимость повторить предыдущую длину 3–6 раз

Значение символа алфавита CWL	Что определяет
17	Позволяет задать для x кодов, начиная с текущего, длину 0; x может принимать значения от 3 до 10 и определяется таким же образом, как и для 16
18	Аналогично 17, но x может быть от 11 до 138

Например, если в результате декодирования описания кодов была получена цепочка длин 2, 3, 16 (x = 4), 17 (x=3), 4,..., то длина кодов символов будет равна:

Значение символа	0	1	2	3	4	5	6	7	8	9	...
Длина кода символа	2	3	3	3	3	3	0	0	0	4	?

Обратите внимание, что символы упорядочены по возрастанию их значений.

С целью увеличения сжатия сами эти цепочки длин кодируются с помощью кодов Хаффмана. Если длина кода символа (т. е. длина кода длин кодов) равна нулю, то это значит, что соответствующий символ в данных не встречается и код ему не отводится.

Формат блока с динамическими кодами Хаффмана описан в табл. 3.10.

Таблица 3.10

Поле	Описание	Размер
HLIT	Хранит количество кодов литералов/длин минус 257	5 бит
HDIST	Хранит количество кодов смещений минус 1	5 бит
HCLEN	Хранит количество кодов длин кодов минус 4	4 бита
Таблица описания кодов длин кодов (коды 2)	Содержит описание (последовательность длин кодов) длин кодов в следующем порядке: 16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15. Иначе говоря, это порядок символов в алфавите CWL. Длина кода любого символа CWL задается с помощью 3 бит; таким образом, длина кода длины кода может быть от 0 (соответствующий символ из CWL не используется) до $2^3 - 1 = 7$ бит. Длины однозначно задают совокупность кодов	(HCLEN+4)· 3 бита
Таблица описания кодов литералов/ длин (коды 1)	Содержит описание HLIT+257 кодов литералов/длин совпадения, закодировано кодами длин кодов	Переменный

Поле	Описание	Размер
Таблица описания кодов смещений	Содержит описание HDIST+1 кодов смещений, закодировано кодами длин кодов	Переменный
Сжатые данные	Содержит данные, сжатые с помощью двух заданных выше совокупностей кодов	"
Знак конца блока	Число 256, сжатое с помощью кодов литералов/длин	"

На основании вышесказанного можно дать такое описание алгоритма кодирования литералов/длин:

- литералы/длины совпадения кодируются с помощью кодов Хаффмана для литералов/длин, назовем их кодами 1;
- описание кодов 1 передается в виде цепочки длин этих кодов;
- при указании длин кодов могут использоваться специальные символы (см. алфавит *CWL*);
- длины кодов, т. е. символы алфавита *CWL*, кодируются с помощью кодов Хаффмана для длин кодов, назовем их кодами 2;
- коды 2 также описываются через последовательность их длин;
- длины кодов 2 задаются с помощью 3-битовых чисел.

Для кодирования смещений используется такой же подход, при этом длины кодов смещений также сжимаются с помощью кодов 2.

Упражнение. Объясните, почему размеры полей, хранящих величины HLIT, HDIST и HCLEN, именно таковы, как это указано в табл. 3.10.

Алгоритм словарного сжатия для Deflate

Как уже указывалось, формат Deflate не имеет четкой спецификации алгоритма словарного сжатия. Разработчики могут использовать какие-то свои алгоритмы, подходящие для решения специфических задач.

Рассмотрим свободный от патентов алгоритм сжатия для Deflate, используемый в разрабатываемой Info-ZIP group утилите Zip.

Для поиска фраз используется метод хеш-цепочек. Хеш-функция вычисляется на основании 3 байт данных (напомним, что формат Deflate не позволяет кодировать строки длиной менее 3 байт). Функция может принимать значения от 0 до заданного числа HASH_MASK – 1 и имеет вид выражения, последовательно вычисляемого для каждого очередного символа:

```
int UPDATE_HASH (int h, char c) {
   return ( (h<<H_SHIFT) ^ c ) & HASH_MASK;
}
```

где h – текущее значение хеш-функции; c – очередной символ; H_SHIFT – параметр сдвига значения функции.

H_SHIFT назначается таким образом, чтобы после очередного сдвига значение самого старого байта не влияло на значение хеш-функции.

На каждом шаге компрессор читает очередную 3-байтовую строку, располагающуюся в начале буфера. После соответствующего обновления хеш-функции производится обращение к первому элементу хеш-цепочки, адрес которого определяется значением функции. Если цепочка пуста, то кодируется литерал, производится сдвиг окна на 1 символ и осуществляется переход к следующему шагу. Иначе хеш-цепочка анализируется с целью найти самое длинное совпадение между буфером и фразами, на которые ссылаются элементы (узлы) хеш-цепочки. Обновление хеш-цепочек организовано так, что поиск начинается с самых "новых" узлов, что позволяет сместить распределение частот смещений кодируемых фраз в пользу коротких смещений и, следовательно, улучшить сжатие, так как небольшие смещения имеют коды малой длины.

Для ускорения кодирования в случае обработки избыточных данных очень длинные хеш-цепочки обрубаются до определенной длины, задаваемой параметрами алгоритма. Усечение производится в зависимости от длины уже найденного совпадения: чем оно длиннее, тем больше отрезаем.

Сжатие может быть улучшено за счет механизма "ленивого" сравнения (lazy matching, или lazy evaluation). Этот подход позволяет отойти от прямолинейного, "жадного" разбора входной последовательности и повысить эффективность сжатия путем более аккуратного выбора фраз словаря. После того как определяется совпадающая фраза *match* (t+1) длины *match_len* (t+1) = L для строки s_{t+1}, s_{t+2}, s_{t+3}, ..., находящейся в начале буфера, выполняется поиск совпадения *match* (t+2) для строки s_{t+2}, s_{t+3}, s_{t+4}... Если *match_len* (t+2) > *match_len* (t+1) = L, то отказываемся от замещения строки s_{t+1}, s_{t+2}, s_{t+3}... Решение о том, следует кодировать *match* (t+2) или нет, принимается на шаге t+3 по результатам аналогичной проверки. Иначе кодирование протекает обычным образом, но с "запаздыванием" на один шаг. Подробнее:

```
// минимальная длина совпадения
const int THRESHOLD = 3;
// смещение и длина совпадения для match(t+1)
int   prev_pos,
      prev_len;
// смещение и длина совпадения для match(t+2)
int   match_pos,
      match_len = THRESHOLD - 1;
// признак отложенного кодирования фразы match(t+1)
```

```
int  match_available = 0;
...
prev_pos = match_pos;   prev_len = match_len;
/*найдем максимальное (или достаточно длинное) совпадение
   для позиции t+2
*/
find_phrase (&match_pos, &match_len);
if ( prev_len >= THRESHOLD &&
     match_len <= prev_len ) {
   /* считаем, что выгоднее закодировать фразу
      match(t+1)
   */
   encode_phrase (prev_pos, prev_len);
   match_available = 0;
   match_len = THRESHOLD - 1;
   // сдвинем окно на match_len(t+1)-1 символов
   move_window (prev_len-1);
   t += prev_len-1;
} else {
   // отложим решение о кодировании на один шаг
   if (match_available) {
      /*кодирование литерала s(t+1) или фразы match(t+1)
         было отложено; кодируем литерал s(t+1)
      */
      encode_literal (window[t+1]);
   }else{
      match_available = 1;
   }
   move_window (1);
   t++;
}
```

Можно сказать, что это одна из возможных реализаций схемы ленивого сравнения с просмотром на один символ вперед. В зависимости от параметров алгоритма для обеспечения желаемого соотношения скорости и коэффициента сжатия механизм ленивого сравнения может запускаться при различных значениях L.

Недостатком рассмотренной реализации ленивого сравнения является порождение длинной цепочки литералов, если на каждом последующем шаге длина совпадения больше, чем на предыдущем.

Упражнение. Объясните, почему использование ленивого сравнения при сжатии не требует внесения изменений в алгоритм декодера.

Кодер обрывает текущий блок данных, если определяет, что изменение кодов Хаффмана может улучшить сжатие, или когда происходит переполнение буфера хранения блока.

Пути улучшения сжатия для методов LZ

Улучшать сжатие алгоритмов семейства Зива – Лемпела можно двумя путями:

1) уменьшением количества указателей при неизменной или большей общей длине закодированных фраз за счет более эффективного разбиения входной последовательности на фразы словаря;
2) увеличением эффективности кодирования индексов фраз словаря и литералов, т. е. уменьшением количества битов, в среднем требуемых для кодирования индекса или литерала.

Идея приемов, относящихся к первому пути, была продемонстрирована на примере ленивого сравнения при описании Deflate. Действительно, для одного и того же словаря мы имеем огромное количество вариантов построения набора фраз для замещения им сжимаемой последовательности. Естественным является так называемый "жадный" разбор (greedy parsing), при котором на каждом шаге кодер выбирает самую длинную фразу. Заметим, что такой способ разбиения данных используют все рассмотренные нами классические алгоритмы LZ. Если поиск ведется по всему словарю, то жадный разбор обеспечивает наибольшую скорость, но и практически всегда наихудшее сжатие. Стратегии оптимального разбора позволяют значительно улучшить сжатие, до 10% и более, но серьезным образом замедляют работу компрессора. Алгоритмы оптимального разбора для алгоритмов семейства LZ77 рассмотрены, например, в [1, 10], а для семейства LZ78 – в [8].

Добиваться большей компактности представления индексов словаря можно за счет:

- применения более сложных алгоритмов сжатия, например на базе контекстных методов моделирования;
- минимизации объема словаря путем удаления излишних совпадающих фраз.

В качестве способа увеличения эффективности кодирования литералов может выступать явное статистическое моделирования вероятностей появления литералов в сочетании с арифметическим кодированием, которое собственно и обеспечивает сжатие. При этом, как показывают эксперименты, для улучшения сжатия целесообразно использовать контекстное моделирование 1-го или 2-го порядка.

Идеи нескольких способов увеличения степени сжатия для методов Зива – Лемпела достаточно подробно описаны ниже.

Стратегия разбора LFF

Как возможную технику улучшения качества разбора входной последовательности на фразы словаря LZ77 укажем метод кодирования самой длинной строки первой – Longest Fragment First, или LFF. Суть LFF заключается в том, что рассматривается несколько вариантов разбиения буфера на фразы словаря и первой замещается строка, с которой совпала фраза максимальной длины, причем строка может начинаться в любой позиции буфера.

Пример

Допустим, у нас в буфере находится строка "абракадабра". Пусть в словаре для каждой позиции буфера можно найти следующие совпадающие фразы максимальной длины:

№ позиции	Совпадающая фраза максимальной длины	Длина фразы
1	аб	2
2	брак	4
3	рак	3
4	ака	3
5	кадаб	5
6	адаб	4
7	даб	3
...		

Поиск прерван на 7-й позиции, поскольку уже никакое совпадение не может быть длиннее максимального встреченного 5. Так как способ кодирования самой длинной строки "кадаб" определен, то теперь для оставшейся подстроки "абра" снова ищется самая длинная совпадающая фраза. Это будет "бра". Оставшаяся слева строка "а" может быть закодирована как литерал. Тогда разбор буфера будет иметь вид:

"абракадаб..." → <а> <бра> <кадаб>.

Эта последовательность из литерала и двух фраз кодируется; окно смещается на 9 символов, а буфер приобретает вид "ра...".

Заметим, что в случае жадного разбора буфер был бы разбит таким образом:

"абракадаб..." → <аб> <рак> <адаб>.

LFF позволяет улучшить сжатие на 0.5–1% по сравнению с жадным разбором.

Оптимальный разбор для методов LZ77

Эвристические техники повышения эффективности разбора входной последовательности, например рассмотренные ленивое сравнение и метод LFF, не решают проблемы получения очень хорошего разбиения в общем случае. Очевидно, что хотелось бы иметь оптимальную стратегию разбора, *во всех случаях* обеспечивающую минимизацию длины закодированной последовательности, порождаемой компрессорами с алгоритмом словарного сжатия типа LZ77 или LZ78.

Для алгоритмов семейства LZ77 эта задача была рассмотрена в [10] и признана *NP*-полной, т. е. требующей полного перебора всех вариантов для нахождения оптимального решения. В диссертации [1] был изложен однопроходной алгоритм, который, как утверждается, позволяет получить оптимальное разбиение при затратах времени не больших, чем вносимых ленивым сравнением.

Мы рассмотрим алгоритм, обеспечивающий получение почти оптимального решения для методов LZ77. Похожая схема используется, например, в компрессоре CABARC.

В общем случае для каждой позиции *t* находим все совпадающие фразы длины от 2 до максимальной MAX_LEN. На основании информации о кодах, используемых для сжатия фраз и литералов, мы можем оценить, сколько примерно битов потребуется для кодирования каждой фразы (литерала), или какова цена price ее кодирования. Тогда мы можем найти близкое к оптимальному решение за один проход следующим образом.

В массив offsets будем записывать ссылки на фразы, подходящие для замещения строки буфера, длины от 2 до max_len, где max_len является длиной максимального совпадения для текущей позиции t. В поле offs[len] сохраняем смещение фразы с длиной len. Если имеется несколько вариантов фраз с одной и той же len, то выбираем фразу с наименьшей ценой price: offsets [t].offs[len] = offset_min_price (t, len). Размер обрабатываемого блока равен MAX_T. Для обеспечения эффективности разбора MAX_T должно быть достаточно большим – несколько сотен байт и более.

```
struct Node {
   int max_len;
   int offs[MAX_LEN+1];
} offsets[MAX_T+1];
```

В ячейках path[t] массива path будем хранить информацию о том, как добраться до позиции *t*, "заплатив" минимальную цену.

```
struct {
   /*цена самого "дешевого" пути до t (из пока известных путей)
```

```
   */
   int price;
   //с какой позиции мы попадаем в t
   int prev_t;
   /*если мы попадем в t, кодируя фразу, то здесь
     хранится смещение этой фразы в словаре
   */
   int offs;
} path[MAX_T+1];
```

Основной цикл разбора:

```
path[0].price = 0;
for (t = 1; t < MAX_T; t++){
   //установим недостижимо большую цену для всех t
   path[t].price = INFINITY;
}
for (t = 0; t < MAX_T; t++){
   /*найдем все совпадающие фразы для строки,
     начинающейся в позиции t
   */
   find_matches (offsets[t]);
   for (len = 1; len <= offsets[t].max_len; len++){
      /*определяем цену рассматриваемого перехода на len
        символов вперед
      */
      if (len == 1)
         // найдем цену кодирования литерала
         price = get_literal_price (t);
      else
         // найдем цену кодирования фразы длины len
         price = get_match_price (len,
                  offsets[t].offs[len]);
      // вычислим цену пути до t + len
      new_price = path[t].price + price;
      if (new_price < path[t+len].price){
         /*рассматриваемый путь до t + len выгоднее
           хранящегося в path[t+len]
         */
         path[t+len].price = new_price;
         path[t+len].prev_t = t;
         if (len > 1)
            path[t+len].offs = offsets[t].offs[len];
      }
   }
}
```

В результате работы алгоритма получаем почти оптимальное решение, записанное в виде односвязного списка. Если предположить, что длина max_len ограничивается в функции find_matches так, чтобы мы не "перепрыгнули" позицию MAX_T, то головой списка является элемент path[MAX_T]. Путь записан в обратном порядке, т. е. фраза со смещением path[MAX_T].offs и длиной MAX_T – path[MAX_T].prev_t (или литерал в позиции MAX_T–1) должна кодироваться самой последней.

На практике применяется несколько эвристических правил, ускоряющих поиск за счет уменьшения числа рассматриваемых вариантов ветвления. Описанный алгоритм позволяет улучшить сжатие для алгоритмов семейства LZ77 на несколько процентов.

Упражнение. Придумайте алгоритм кодирования найденной последовательности фраз и литералов.

Алгоритм Бендера–Вулфа

Очевидно, что классические алгоритмы семейства LZ77 обладают большой избыточностью словаря. Так, например, в словаре может быть несколько одинаковых фраз длины от *match_len* и менее, совпадающих со строкой в начале буфере. Но классический LZ77 никак не использует такую информацию и фактически отводит каждой фразе одинаковый объем кодового пространства, считает их равновероятными. В 1991 г. Бендер (Bender) и Вулф (Wolf) описали прием, позволяющий до некоторой степени компенсировать данную "врожденную" избыточность алгоритмов LZ со скользящим окном [2]. В этом алгоритме (LZBW) после нахождения фразы *S*, имеющей самое длинное совпадение с буфером, производится поиск самой длинной совпадающей фразы *S'* среди добавленных в словарь позже *S* и полностью находящихся в словаре (не вторгающихся в область буфера). Длина *S* передается декодеру как разница между длиной *S* и длиной *S'*. Например, если *S* = "абсд" и *S'* = "аб", то длина *S* передается с помощью разностной длины 2.

Пример

Модифицируем LZSS с помощью техники Бендера – Вулфа. Рассмотрим процесс кодирования строки "колол_кот_ломом_колесо" начиная с первого появления символа "м". В отличие от рассмотренного выше примера LZSS словарное кодирование будем применять при длине совпадения 1 и более (табл. 3.11).

Таблица 3.11

Шаг	Скользящее окно		Совпадающая фраза	Закодированные данные			
	Словарь	Буфер		*f*	*i*	*j*	*s*
k	колол_кот_ло	мом_кол	-	0	-	-	"м"
k+1	колол_кот_лом	ом_коле	ом	1	2	2	-
k+2	колол_кот_ломом	_колесо	_	1	6	1	-
k+3	колол_кот_ломом_	колесо	кол	1	16	1	-
k+4	...л_кот_ломом_кол	есо	-	0	-	-	"е"

На шаге *k* мы встретили символ "м", отсутствующий в словаре, поэтому передаем его в явном виде. Сдвигаем окно на одну позицию.

На шаге *k*+1 совпадающая фраза равна "ом", и в словаре нет никаких других фраз, добавленных позже "ом". Поэтому положим длину *S*' равной нулю, тогда передаваемая разность *j* есть 2. Сдвигаем окно на 2 символа.

На шаге *k*+2 совпадающая фраза состоит из одного символа "_", последний раз встреченного 6 символов назад. Здесь, как и на шаге *k*+1, разностная длина совпадающей фразы равна ее длине, т. е. единице. Сдвигаем окно на 1 символ.

На шаге *k*+3 совпадающая фраза максимальной длины есть "кол", но среди фраз, добавленных в словарь после "кол", имеется фраза "ко", поэтому длина "кол" кодируется как разница 3 и 2. Если бы в части словаря, ограниченной фразой "кол" слева и началом буфера справа, не нашлось бы фразы "ко" с длиной совпадения 2, а была бы обнаружена, например, только фраза "к" с длиной совпадения 1, то длина "кол" была бы представлена как 3 – 1 = 2.

При декодировании необходимо поддерживать словарь в таком же виде, что и при кодировании. После получения смещения *match_pos* декодер производит сравнение фразы, начинающейся с найденной позиции *t*–(*match_pos*–1) (*t*+1 – позиция начала буфера), со всеми более "новыми" фразами словаря, т. е. лежащими в области *t*–(match_pos–1), ..., *t*. Длина фразы восстанавливается как сумма максимального совпадения и полученной от кодера разностной длины *j*. Так, например, процедура декодирования для шага *k*+3 будет выглядеть следующим образом. Декодер читает смещение *i* = 16, разностную длину *j* = 1 и начинает сравнивать фразу "колол_...", начало которой определяется данным смещением *i*, со всеми фразами словаря, начало которых имеет смещение от 15 до 1. Максимальное совпадение имеет фраза "ко", расположенная по смещению 10. Поэтому длина закодированной фразы равна 2 + *j* = 2 + 1 = 3, т. е. кодер передал указатель на фразу "кол".

Использование техники Бендера – Вулфа позволяет улучшить сжатие для алгоритмов типа LZH примерно на 1%. При этом декодирование сильно замедляется (в разы!), поскольку мы вынуждены выполнять такой же дополнительный поиск для определения длины, что и при сжатии.

Алгоритм Файэлэ – Грини

Еще один способ борьбы с избыточностью словаря LZ77 предложен Файэлэ (Fiala) и Грини (Greene) [4]. В разработанном ими алгоритме LZFG для просмотра словаря используется дерево цифрового поиска, и любая фраза кодируется не парой <длина, смещение>, а индексом узла, соответствующего этой фразе. Иначе говоря, всем одинаковым фразам соответствует один и тот же индекс. Устранение повторяющихся фраз из словаря позволяет уменьшить среднюю длину кодов фраз. Обычно коэффициент сжатия LZFG лучше коэффициента сжатия LZSS примерно на 10%.

Контекстно-зависимые словари

Объем словаря и, соответственно, средняя длина закодированного индекса могут быть уменьшены за счет использования приемов контекстного моделирования. Было установлено, что применение контекстно-зависимых словарей улучшает сжатие для алгоритмов семейства LZ77 и, в особенности, семейства LZ78 [5]. Идея подхода состоит в следующем. Пусть мы только что закодировали последовательность символов S небольшой длины L, тогда на текущем шаге в качестве словаря используются только строки, встреченные в уже обработанной части потока непосредственно после строк, равных S. Эти строки образуют контекстно-зависимый словарь порядка L для S. Если в словаре порядка L совпадение обнаружить не удалось, то происходит уход к словарю порядка L–1. В конечном итоге если не было найдено совпадающих фраз ни в одном из доступных словарей, то текущий символ передается в явном виде.

Например, при использовании техники контекстно-зависимых словарей порядка в сочетании с LZ77 после обработки последовательности "абракадабра" получаем следующий состав словарей порядков 2, 1 и 0 для текущего контекста:

Порядок L	Состав контекстно-зависимого словаря порядка L (фразы словаря могут начинаться только в первой позиции указанных последовательностей)
2 (контекст "ра")	кадабра
1 (контекст "а")	бракадабра кадабра дабра бра

0 (пустой контекст)	абракадабра бракадабра ракадабра акадабра кадабра адабра дабра абра бра ра а (обычный словарь LZ77)

Эксперименты показали, что наилучшие результаты достигаются при $L = 1$ или 2, т. е. в качестве контекста достаточно использовать один или два предыдущих символа. Применение контекстно-зависимых словарей позволяет улучшить сжатие LZSS на 1–2%, LZFG – на 5%, LZW – примерно на 10%. Потери в скорости в случае модификации LZFG составляют порядка 20–30%. Соответствующие версии алгоритмов известны как LZ77-PM, LZFG-PM, LZW-PM [5].

Недостатком техники является необходимость поддерживать достаточно сложную структуру контекстно-зависимых словарей не только при кодировании, но и при декодировании.

Буферизация смещений

Если была закодирована фраза со смещением i, то увеличивается вероятность того, что вскоре нам может потребоваться закодировать фразы с приблизительно таким же смещением $i \pm \delta$, где δ – небольшое число. Это особенно часто проявляется при обработке двоичных данных (исполнимых файлов, файлов ресурсов), поскольку для них характерно наличие сравнительно длинных последовательностей, отличающихся лишь в нескольких позициях.

Смещение обычно представляется посредством двух (иногда более) полей: базы (сегмента) и поля дополнительных битов, уточняющего значение смещения относительно базы. Поэтому в потоке закодированных данных, порождаемом алгоритмами семейства LZ77, коды фраз с одним и тем же значением базы смещения часто располагаются недалеко друг от друга. Это свойство можно использовать для улучшения сжатия, применив технику буферизации баз смещений.

В буфере запоминается m последних использованных баз смещений, различающихся между собой. Буфер обновляется по принципу списка LRU, т. е. самая последняя использованная база имеет индекс 0, самая "старая" –

индекс m–1. Если база B смещения текущей фразы совпадает с одной из содержащихся в буфере баз B_i, то вместо B кодируется индекс i буфера. Затем B_i перемещается в начало списка LRU, т. е. получает индекс 0, а все B_0, B_1, ..., B_{i-1} сдвигаются на одну позицию к концу списка. Иначе кодируется собственно база B, после чего она добавляется в начало буфер как B_0, а все буферизованные базы смещаются на одну позицию к концу списка LRU, при этом B_{m-1} удаляется из буфера.

Пример

Примем m равным двум. Если база не совпадает ни с одной содержащейся в буфере, то кодируемое значение базы равно абсолютному значению плюс m. Пусть также содержимое буфера равно {0, 1}, тогда пошаговое преобразование последовательности баз смещений 15, 14, 14, 2, 3, 2,... будет выглядеть следующим образом:

Номер шага	Абсолютное значение базы	Содержимое буфера в начале шага		Кодируемое значение базы
		B_0	B_1	
1	15	0	1	17
2	14	15	0	16
3	14	14	15	0
4	2	14	15	4
5	3	2	14	5
6	2	3	2	1
7	?	2	3	?

Обратите внимание на состояние буфера после шага 3. Оно не изменилось, поскольку все элементы буфера должны быть различны. Иначе мы ухудшим сжатие из-за внесения избыточности в описание баз смещений, поскольку в этом случае одна и та же база может задаваться несколькими числами.

Как показывают эксперименты, оптимальное значение m лежит в пределах 4–8.

Стоимость кодирования индекса буфера обычно ниже стоимости кодирования базы смещения непосредственно. Поэтому имеет смысл принудительно увеличивать частоту использования буферизованных смещений за счет подбора фраз с "нужным" расположением в словаре.

Применение рассмотренной техники заметно улучшает сжатие двоичных файлов – до нескольких процентов, но слабо влияет в случае обработки текстов.

Буферизация смещений используется практически во всех современных архиваторах, реализующих алгоритмы семейства LZ77, например: 7-Zip, CABARC, WinRAR.

Совместное кодирование длин и смещений

Между величиной смещения и длиной совпадения имеется незначительная корреляция, величина которой возрастает в случае применения буферизации смещений. Это свойство можно использовать, объединив в один метасимвол длину совпадения `match_len` и базу смещения `offset_base`, и, таким образом, кодировать метасимвол на основании статистики совместного появления определенных длины и смещения. Как и в случае смещения, в метасимвол лучше включать не полностью длину, а ее квантованное значение.

Так, например, в формате LZX (используется в компрессоре CABARC) длины совпадения от 2 до 8 входят в состав метасимвола непосредственно, а все длины `match_len` > 8 отображаются в одно значение. В последнем случае длина совпадения доопределяется путем отдельной передачи величины `match_len` – 9. Метасимволы длина/смещение и литералы входят в один алфавит, поэтому в упрощенном виде алгоритм кодирования таков:

```
if (match_len >= 2) {
   // закодируем фразу
   if ( match_len <= 8 )
      metasymbol = (offset_base<<3) || (match_len-2);
   else
      metasymbol = (offset_base<<3) || 7;
   /*закодируем метасимвол, указав, что это не литерал,
      для правильного отображения значения метасимвола в
      алфавит длин/смещений и литералов
   */
   encode_symbol (metasymbol, NON_LITERAL);
   if (match_len > 8)
      // доопределим длину совпадения
      encode_length_footer (match_len - 9);
   // закодируем младшие биты смещения
   encode_offset_footer (...);
   ...
}else{
   // закодируем литерал в текущей позиции t+1
   encode_symbol (window[t+1], LITERAL)
   ...
}
```

Упражнение. Напишите упрощенный алгоритм декодирования.

Архиваторы и компрессоры, использующие алгоритмы LZ

Среди огромного количества LZ-архиваторов отметим следующие:

3) 7-Zip, автор Игорь Павлов (Pavlov);
4) ACE, автор Маркел Лемке (Lemke);
5) ARJ, автор Роберт Джанг (Jung);
6) ARJZ, автор Булат Зиганшин (Ziganshin);
7) CABARC, корпорация Microsoft;
8) Imp, фирма Technelysium Pty Ltd.;
9) JAR, автор Роберт Джанг (Jung);
10) PKZIP, фирма PKWARE Inc.;
11) RAR, автор Евгений Рошал (Roshal);
12) WinZip, фирма Nico Mak Computing;
13) Zip, Info-ZIP group.

Эти архиваторы являются или одними из самых эффективных в классе применяющих методы Зива – Лемпела, или пользуются популярностью, или оказали существенное влияние на развитие словарных алгоритмов, или интересны с точки зрения нескольких указанных критериев. За исключением 7-Zip, словарные алгоритмы всех указанных архиваторов можно рассматривать как модификации LZH. В алгоритме LZMA, реализованном в 7-Zip, совместно со словарными заменами используется контекстное моделирование и арифметическое кодирование.

В табл. 3.12 представлены результаты сравнения некоторых архиваторов по степени сжатия файлов набора CalgCC.

Таблица 3.12

	ARJ	PKZIP	ACE	RAR	CABARC	7-Zip
Bib	3.08	3.16	3.38	3.39	3.45	3.62
Book1	2.41	2.46	2.78	2.80	2.91	2.94
Book2	2.90	2.95	3.36	3.39	3.51	3.59
Geo	1.48	1.49	1.56	1.53	1.70	1.89
News	2.56	2.61	3.00	3.00	3.07	3.16
Obj1	2.06	2.07	2.19	2.18	2.20	2.26
Obj2	3.01	3.04	3.39	3.38	3.54	3.96
Paper1	2.84	2.85	2.91	2.93	2.99	3.07
Paper2	2.74	2.77	2.86	2.88	2.95	3.01
Pic	9.30	9.76	10.53	10.39	10.67	11.76
Progc	2.93	2.94	3.00	3.01	3.04	3.15

Progl	4.35	4.42	4.49	4.55	4.62	4.76
Progp	4.32	4.37	4.55	4.57	4.62	4.73
Trans	4.65	4.79	5.19	5.23	5.30	5.56
Итого	**3.47**	**3.55**	**3.80**	**3.80**	**3.90**	**4.10**

Использованные версии архиваторов: ARJ 2.50a, PKZIP 2.04g, WinRAR 2.71, ACE 2.04, 7-Zip 2.30 beta 7. Во всех случаях применялся тот алгоритм LZ, который обеспечивал наилучшее сжатие. Заметим, что 7-Zip использует специальные методы препроцессинга нетекстовых данных, "отключить" которые не удалось, что до некоторой степени исказило картину. Тем не менее преимущество этого архиватора на данном тестовом наборе несомненно. В случае WinRAR и ACE режим мультимедийной компрессии намеренно не включался.

> *При сравнении программ 7-Zip версии 2.3 и RAR версии 3 с другими LZ-архиваторами необходимо следить, чтобы 7-Zip и RAR использовали алгоритм типа LZ, поскольку они имеют в своем арсенале алгоритм PPMII, обеспечивающий высокую степень сжатия текстов.*

При сравнении следует учитывать, что скорость сжатия ARJ и PKZIP была примерно в 4.5 раза выше, чем у RAR и ACE, которые, в свою очередь, были быстрее CABARC и 7-Zip приблизительно на 30%. Размер словаря в ARJ и PKZIP в десятки раз меньше, чем в остальных программах.

Вопросы для самоконтроля[1]

1. Какие свойства данных определяют принципиальную возможность их сжатия с помощью LZ-методов?
2. В чем основная разница между алгоритмами семейства LZ77 и семейства LZ78?
3. Какие особенности строения словаря LZ77 позволяют создавать для одного и того же входного файла несколько различных архивных, которые затем можно разжать без потерь информации с помощью одного и того же декодера LZ77? Возможно ли это в случае алгоритма LZ78?
4. Почему в алгоритмах семейства LZ77 короткие строки часто выгоднее сжимать не с помощью словарной замены, а через кодирование как последовательности литералов? Каким образом это связано с величиной смещения фразы, совпадающей со строкой?
5. Приведите пример блока данных, которые в общем случае выгоднее сжимать алгоритмом семейства LZ77, нежели семейства LZ78, а также

[1] *Ответы на вопросы, выполненные упражнения и исходные тексты программ вы можете найти на http://compression.graphicon.ru/.*

обратный пример. На основании каких критериев можно сделать предварительный выбор между алгоритмами семейства LZ77 или семейства LZ78, если задаваться только целью максимизации степени сжатия?

6. Почему дистрибутивы программного обеспечения целесообразно архивировать с помощью алгоритмов семейства LZ77?
7. В каких случаях имеет смысл использовать методы кодирования целых чисел – коды Элиаса, Голомба и т. п. – для сжатия потоков смещений и длин совпадения?
8. Применим ли алгоритм "почти оптимального" разбора для методов LZ77, рассмотренный в подразд. "Пути улучшения сжатия для методов LZ", для обработки потоков? Можно ли однозначно сказать, что любая стратегия получения оптимального разбора требует поблочной обработки данных?
9. Почему применение контекстно-зависимых словарей улучшает степень сжатия для алгоритмов семейства LZ78 значительно больше, чем для алгоритмов семейства LZ77?

Литература

1. Кадач А. В. Эффективные алгоритмы неискажающего сжатия текстовой информации. – Дис. к. ф.-м. н. – Ин-т систем информатики им. А. П. Ершова. М., 1997.
2. Bender P. E., Wolf J. K. New asymptotic bounds and improvements on the Lempel-Ziv data compression algorithm // IEEE Transactions on Information Theory. May 1991. Vol. 37(3). P. 721–727.
3. Deutsch L. P. (1996) DEFLATE Compressed Data Format Specification v.1.3 (RFC1951) http://sochi.net.ru/~maxime/doc/rfc1951.ps.gz.
4. Fiala E. R., Greene D. H. Data compression with finite windows. Commun // ACM. Apr. 1989. Vol. 32(4). P. 490–505.
5. Hoang D. T., Long P. M., Vitter J.S. Multiple-dictionary compression using partial matching // Proceedings of Data Compression Conference. March 1995. P. 272–281, Snowbird, Utah.
6. Langdon G. G. A note on the Ziv-Lempel model for compressing individual sequences // IEEE Transactions on Information Theory. March 1983. Vol. 29(2). P. 284–287.
7. Larsson J., Moffat A. Offline dictionary-based compression // Proceedings IEEE. Nov. 2000. Vol. 88(11). P. 1722–1732.
8. Matias Y., Rajpoot N., Sahinalp S.C. Implementation and experimental evaluation of flexible parsing for dynamic dictionary based data compression // Proceedings WAE'98. 1998.

9. Rissanen J.J., Langdon G.G. Universal modeling and coding // IEEE Transactions on Information Theory. Jan. 1981. Vol. 27(1). P. 12–23.
10. Storer J. A., Szymanski T. G. Data compression via textual substitution // Journal of ACM. Oct. 1982. Vol. 29(4). P. 928–951.
11. Welch T.A. A technique for high-performance data compression // IEEE Computer. June 1984. Vol. 17(6). P. 8–19.
12. Ziv J., Lempel A. A universal algorithm for sequential data compression // IEEE Transactions on Information Theory. May 1977. Vol. 23(3). P. 337–343.
13. Ziv J. and Lempel A. Compression of individual sequences via variable-rate coding // IEEE Transactions on Information Theory. Sept. 1978. Vol. 24(5). P. 530–536.

Список архиваторов и компрессоров

1. Info-ZIP group. Info-ZIP's portable Zip – C sources. http://www.infozip.org
2. Jung R. ARJ archiver. http://www.arjsoftware.com
3. Jung R. JAR archiver. ftp://ftp.elf.stuba.sk/pub/pc/pack/jar102x.exe
4. Lemke M. ACE archiver. http://www.winace.com
5. Microlog Cabinet Manager 2001 for Win9x/NT – Compression tool for .cab files. ftp://ftp.elf.stuba.sk/pub/pc/pack/cab2001.zip
6. Microsoft Corporation. Cabinet Software Development Tool. http://msdn.microsoft.com/library/en-us/dnsamples/cab-sdk.exe
7. Nico Mak Computing. WinZip archiver. http://www.winzip.com
8. Pavlov I. 7-Zip archiver. http://www.7-zip.org
9. PKWARE Inc. PKZIP archiver. ftp://ftp.elf.stuba.sk/pub/pc/pack /pk250dos.exe
10. Roshal E. RAR for Windows. http://www.rarsoft.com
11. Technelysium Pty Ltd. IMP archiver. ftp://ftp.elf.stuba.sk/pub/pc/pack /imp112.exe
12. Ziganshin B. ARJZ archiver. ftp://ftp.elf.stuba.sk/pub/pc/pack/arjz015.zip

Глава 4. Методы контекстного моделирования

Применение методов контекстного моделирования для сжатия данных опирается на парадигму сжатия с помощью универсального моделирования и кодирования (universal modelling and coding), предложенную Риссаненом (Rissanen) и Лэнгдоном (Langdon) в 1981 г. [12]. В соответствии с данной идеей процесс сжатия состоит из двух самостоятельных частей:

- **моделирования**;
- **кодирования**.

Под моделированием понимается построение модели информационного источника, породившего сжимаемые данные, а под кодированием – отображение обрабатываемых данных в сжатую форму представления на основании результатов моделирования (рис. 4.1). "**Кодировщик**" создает выходной поток, являющийся компактной формой представления обрабатываемой последовательности, на основании информации, поставляемой ему "**моделировщиком**".

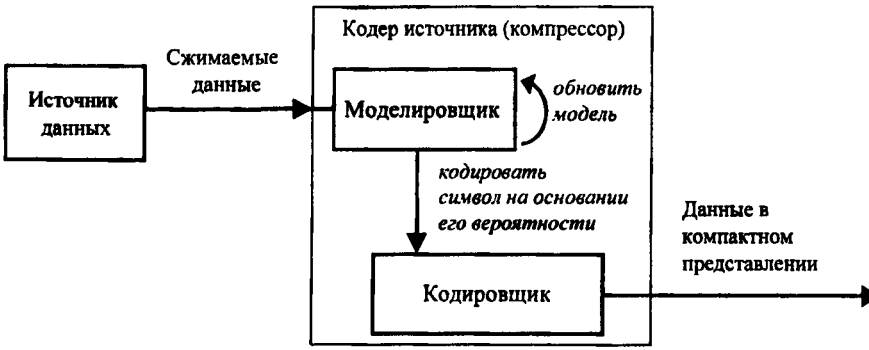


Рис. 4.1. Схема процесса сжатия данных в соответствии с концепцией универсального моделирования и кодирования

Следует заметить, что понятие "кодирование" часто используют в широком смысле для обозначения всего процесса сжатия, т. е. включая моделирование в данном нами определении. Таким образом, необходимо различать понятие кодирования в широком смысле (весь процесс) и в узком (генерация потока кодов на основании информации модели). Понятие "статистическое кодирование" также используется, зачастую с сомнительной корректностью, для обозначения того или иного уровня кодирования. Во избежание путаницы ряд авторов применяет термин "энтропийное кодирование" для кодирования в узком смысле. Это наименование далеко от совершенства и встречает вполне обоснованную критику. Далее в этой главе процесс кодирования в широком смысле будем именовать "кодированием", а в узком смысле – "статистическим кодированием" или "собственно кодированием".

Из теоремы Шеннона о кодировании источника [13] известно, что символ s_i, вероятность появления которого равняется $p(s_i)$, выгоднее всего представлять $-\log_2 p(s_i)$ битами, при этом средняя длина кодов может быть вычислена по приводившейся ранее формуле (1.1 и 1.2). Практически всегда истинная структура источника скрыта, поэтому необходимо строить **модель источника**, которая позволила бы нам в каждой позиции входной последо-

вательности найти оценку $q(s_i)$ вероятности появления каждого символа s_i алфавита входной последовательности.

Оценка вероятностей символов при моделировании производится на основании известной статистики и, возможно, априорных предположений, поэтому часто говорят о задаче статистического моделирования. Можно сказать, что моделировщик **предсказывает** вероятность появления каждого символа в каждой позиции входной строки, отсюда еще одно наименование этого компонента – "предсказатель" или "предиктор" (от predictor). На этапе статистического кодирования выполняется замещение символа s_i с оценкой вероятности появления $q(s_i)$ кодом длиной $-\log_2 q(s_i)$ бит.

Рассмотрим пример. Предположим, что мы сжимаем последовательность символов алфавита {"0","1"}, порожденную источником без памяти, и вероятности генерации символов следующие: $p("0") = 0.4$, $p("1") = 0.6$. Пусть наша модель дает такие оценки вероятностей: $q("0") = 0.35$, $q("1") = 0.65$. Энтропия H источника равна

$$-p('0')\log_2 p('0') - p('1')\log_2 p('1') =$$
$$= -0.4\log_2 0.4 - 0.6\log_2 0.6 \approx 0.971 \ \textit{бита}.$$

Если подходить формально, то "энтропия" модели получается равной

$$-q('0')\log_2 q('0') - q('1')\log_2 q('1') =$$
$$= -0.35\log_2 0.35 - 0.65\log_2 0.65 \approx 0.934 \ \textit{бита}.$$

Казалось бы, что модель обеспечивает лучшее сжатие, чем это позволяет формула Шеннона. Но истинные вероятности появления символов не изменились! Если исходить из вероятностей p, то "0" следует кодировать – $\log_2 0.4 \approx 1.322$ бита, а для "1" нужно отводить $-\log_2 0.6 \approx 0.737$ бита. Для оценок вероятностей q мы имеем $-\log_2 0.35 \approx 1.515$ бита и $-\log_2 0.65 \approx 0.621$ бита соответственно. При каждом кодировании на основании информации модели в случае "0" мы будем терять 1.515 – 1.322 = 0.193 бита, а в случае "1" выигрывать 0.737 – 0.621 = 0.116 бита. С учетом вероятностей появления символов средний проигрыш при каждом кодировании составит $0.4 \cdot 0.193 - 0.6 \cdot 0.116 = 0.008$ бита.

Вывод. ***Чем точнее оценка*** вероятностей появления символов, тем больше коды соответствуют оптимальным, ***тем лучше сжатие.***

Правильность декодирования обеспечивается использованием точно такой же модели, которая была применена при кодировании. Следовательно, при моделировании для сжатия данных нельзя пользоваться информацией, которая неизвестна декодеру.

Осознание двойственной природы процесса сжатия позволяет осуществлять декомпозицию задач компрессии данных со сложной структурой и нетривиальными взаимозависимостями, обеспечивать определенную самостоятельность процедур, решающих частные проблемы, сосредоточивать больше внимания на деталях реализации конкретного элемента.

Задача статистического кодирования была в целом успешно решена к началу 80-х г. Арифметический кодер позволяет сгенерировать сжатую последовательность, длина которой обычно всего лишь на десятые доли процента превышает теоретическую длину, рассчитанную с помощью формулы (1.1). Более того, применение современной модификации арифметического кодера – интервального кодера – позволяет осуществлять собственно кодирование очень быстро. Скорость статистического кодирования составляет миллионы символов в секунду на современных ПК.

В свете вышесказанного повышение точности моделей является фактически единственным способом существенного улучшения сжатия.

Классификация стратегий моделирования

Перед рассмотрением контекстных методов моделирования следует сказать о классификации стратегий моделирования источника данных по способу построения и обновления модели. Выделяют 4 варианта моделирования:

- статическое;
- полуадаптивное;
- адаптивное (динамическое);
- блочно-адаптивное.

При статическом моделировании для любых обрабатываемых данных используется одна и та же модель. Иначе говоря, не производится адаптация модели к особенностям сжимаемых данных. Описание заранее построенной модели хранится в структурах данных кодера и декодера; таким образом достигается однозначность кодирования, с одной стороны, и отсутствие необходимости в явной передаче модели, с другой. Недостаток подхода также очевиден: мы можем получать плохое сжатие и даже увеличивать размер представления, если обрабатываемые данные не соответствуют выбранной модели. Поэтому такая стратегия используется только в специализированных приложениях, когда тип сжимаемых данных неизменен и заранее известен.

Полуадаптивное сжатие является развитием стратегии статического моделирования. В этом случае для сжатия заданной последовательности выбирается *или* строится модель на основании анализа именно обрабатываемых данных. Понятно, что кодер должен передавать декодеру не только зако-

дированные данные, но и описание использованной модели. Если модель выбирается из заранее созданных и известных как кодеру, так и декодеру, то это просто порядковый номер модели. Иначе если модель была настроена или построена при кодировании, то необходимо передавать либо значения параметров настройки, либо модель полностью. В общем случае полуадаптивный подход дает лучшее сжатие, чем статический, так как обеспечивает приспособление к природе обрабатываемых данных, уменьшая вероятность значительной разницы между предсказаниями модели и реальным поведением потока данных.

Адаптивное моделирование является естественной противоположностью статической стратегии. По мере кодирования модель изменяется по заданному алгоритму после сжатия каждого символа. Однозначность декодирования достигается тем, что, во-первых, изначально кодер и декодер имеют идентичную и обычно очень простую модель и, во-вторых, модификация модели при сжатии и разжатии осуществляется одинаковым образом. Опыт использования моделей различных типов показывает, что адаптивное моделирование является не только элегантной техникой, но и обеспечивает по крайней мере не худшее сжатие, чем полуадаптивное моделирование. Понятно, что если стоит задача создания "универсального" компрессора для сжатия данных несходных типов, то адаптивный подход является естественным выбором разработчика.

Блочно-адаптивное моделирование можно рассматривать как частный случай адаптивной стратегии (или наоборот, что сути дела не меняет). В зависимости от конкретного алгоритма обновления модели, оценки вероятностей символов, метода статистического кодирования и самих данных изменение модели после обработки каждого символа может быть сопряжено со следующими неприятностями:

- потерей устойчивости (робастности) оценок, если данные "зашумлены", или имеются значительные локальные изменения статистических взаимосвязей между символами обрабатываемого потока; иначе говоря, чересчур быстрая, "агрессивная" адаптация модели может приводить к ухудшению точности оценок;
- большими вычислительными расходами на обновление модели (как пример – в случае адаптивного кодирования по алгоритму Хаффмана);
- большими расходами памяти для хранения структур данных, обеспечивающих быструю модификацию модели.

Поэтому обновление модели может выполняться после обработки целого блока символов, в общем случае переменной длины. Для обеспечения правильности разжатия декодер должен выполнять такую же последовательность действий по обновлению модели, что и кодер, либо кодеру необ-

ходимо передавать вместе со сжатыми данными инструкции по модификации модели. Последний вариант достаточно часто используется при блочно-адаптивном моделировании для ускорения процесса декодирования в ущерб коэффициенту сжатия.

Понятно, что приведенная классификация является до некоторой степени абстрактной, и на практике часто используют гибридные схемы.

Контекстное моделирование

Итак, нам необходимо решить задачу оценки вероятностей появления символов в каждой позиции обрабатываемой последовательности. Для того чтобы разжатие произошло без потерь, мы можем пользоваться только той информацией, которая в полной мере известна как кодеру, так и декодеру. Обычно это означает, что оценка вероятности очередного символа должна зависеть только от свойств уже обработанного блока данных.

Пожалуй, наиболее простой способ оценки реализуется с помощью полуадаптивного моделирования и заключается в предварительном подсчете безусловной частоты появления символов в сжимаемом блоке. Полученное распределение вероятностей используется для статистического кодирования всех символов блока. Если, например, такую модель применить для сжатия текста на русском языке, то в среднем на кодирование каждого символа будет потрачено примерно 4.5 бита. Это значение является средней длиной кодов для модели, базирующейся на использовании безусловного распределения вероятностей букв в тексте. Заметим, что уже в этом простом случае достигается степень сжатия 1.5 по отношению к тривиальному кодированию, когда всем символам назначаются коды одинаковой длины. Действительно, размер алфавита русского текста превышает 64 знака, но меньше 128 знаков (строчные и заглавные буквы, знаки препинания, пробел), что требует 7-битовых кодов.

Анализ распространенных типов данных, – например, тех же текстов на естественных языках, – выявляет сильную зависимость вероятности появления символов от непосредственно им предшествующих. Иначе говоря, бóльшая часть данных, с которыми мы сталкиваемся, порождается источниками с памятью. Допустим, нам известно, что сжимаемый блок является текстом на русском языке. Если, например, строка из трех только что обработанных символов равна "_цы" (подчеркиванием здесь и далее обозначается пробел), то текущий символ скорее всего входит в следующую группу: "г" ("цыган"), "к" ("цыкать"), "п" ("цыпочки"), "ц" ("цыц"). Или, в случае анализа сразу нескольких слов, если предыдущая строка равна "Вставай,_проклятьем_заклейменный,", то продолжением явно будет "весь_мир_". Следовательно, учет зависимости частоты появления символа (в общем

случае – блока символов) от предыдущих должен давать более точные оценки и в конечном счете лучшее сжатие. Действительно, в случае посимвольного кодирования при использовании информации об одном непосредственно предшествующем символе достигается средняя длина кодов в 3.6 бита для русских текстов, при учете двух последних – уже порядка 3.2 бита. В первом случае моделируются условные распределения вероятностей символов, зависящие от значения строки из одного непосредственно предшествующего символа, во втором – зависящие от строки из двух предшествующих символов.

Любопытно, что модели, оперирующие безусловными частотами и частотами в зависимости от одного предшествующего символа, дают примерно одинаковые результаты для всех европейских языков (за исключением, быть может, самых экзотических) – 4.5 и 3.6 бита соответственно.

Улучшение сжатия при учете предыдущих элементов (пикселов, сэмплов, отсчетов, чисел) отмечается и при обработке данных других распространенных типов: объектных файлов, изображений, аудиозаписей, таблиц чисел.

Терминология

Под контекстным моделированием будем понимать оценку вероятности появления символа (элемента, пиксела, сэмпла, отсчета и даже набора качественно разных объектов) в зависимости от непосредственно ему предшествующих, или контекста.

Заметим, что в быту понятие "контекст" обычно используется в глобальном значении – как совокупность символов (элементов), окружающих текущий обрабатываемый. Это контекст в широком смысле. Выделяют также "левосторонние" и "правосторонние" контексты, т. е. последовательности символов, непосредственно примыкающих к текущему символу слева и справа соответственно. Здесь и далее под контекстом будем понимать именно классический левосторонний: так, например, для последнего символа "о" последовательности "...молоко..." контекстом является "...молок".

Если длина контекста ограничена, то такой подход будем называть *контекстным моделированием ограниченного порядка* (finite-context modeling), при этом под порядком понимается максимальная длина используемых контекстов N. Например, при моделировании порядка 3 для последнего символа "о" в последовательности "...молоко..." контекстом максимальной длины 3 является строка "лок". При сжатии этого символа под "текущими контекстами" могут пониматься "лок", "ок", "к", а также пустая строка "". Все эти контексты длины от N до 0 назовем *активными контекстами* в том смысле,

что при оценке символа может быть использована накопленная для них статистика.

Далее вместо "контекст длины o, $o \le N$" мы будем обычно говорить "контекст порядка o".

В силу объективных причин – ограниченности вычислительных ресурсов – техника контекстного моделирования именно ограниченного порядка получила наибольшее развитие и распространение, поэтому далее под контекстным моделированием будем понимать именно ее. Дальнейшее изложение также учитывает специфику того, что контекстное моделирование практически всегда применяется как адаптивное.

Оценки вероятностей при контекстном моделировании строятся на основании обычных счетчиков частот, связанных с текущим контекстом. Если мы обработали строку "абсабвбабс", то для контекста "аб" счетчик символа "с" равен двум (говорят, что символ "с" *появился в контексте* "аб" 2 раза), символа "в" – единице. На основании этой статистики можно утверждать, что вероятность появления "с" после "аб" равна 2/3, а вероятность появления "в" – 1/3, т. е. оценки формируются на основе уже просмотренной части потока.

В общем случае для каждого контекста конечной длины $o \le N$, встречаемого в обрабатываемой последовательности, создается контекстная модель (КМ). Любая КМ включает в себя счетчики всех символов, встреченных в соответствующем ей контексте, т. е. сразу после строки контекста. После каждого появления какого-то символа s в рассматриваемом контексте производится увеличение значения счетчика символа s в соответствующей контексту КМ. Обычно счетчики инициализируются нулями. На практике счетчики обычно создаются по мере появления в заданном контексте новых символов, т. е. счетчиков, ни разу не виденных в заданном контексте символов, просто не существует.

Под порядком КМ будем понимать длину соответствующего ей контекста. Если порядок КМ равен o, то будем обозначать такую КМ как "КМ(o)".

Кроме обычных КМ, часто используют контекстную модель минус 1-го порядка КМ(–1), присваивающую одинаковую вероятность всем символам алфавита сжимаемого потока.

Понятно, что для 0-го и минус 1-го порядка контекстная модель одна, а КМ большего порядка может быть несколько, вплоть до q^N, где q – размер алфавита обрабатываемой последовательности. КМ(0) и КМ(–1) всегда активны.

Заметим, что часто не делается различий между понятиями "контекст" и "контекстная модель". Авторы этой книги такое соглашение не поддерживают.

Часто говорят о "родительских" и "дочерних" контекстах. Для контекста "к" дочерними являются "ок" и "лк", поскольку они образованы сцеплением (конкатенацией) одного символа и контекста "к". Аналогично для контекста "лок" родительским является контекст "ок", а контекстами-предками – "ок", "к", "". Очевидно, что "пустой" контекст "" является предком для всех. Аналогичные термины применяются для КМ, соответствующих контекстам.

Совокупность КМ образует модель источника данных. Под порядком модели понимается максимальный порядок используемых КМ.

Виды контекстного моделирования

Пример обработки строки "абсабвбабс" иллюстрирует сразу две проблемы контекстного моделирования:

- как выбирать подходящий контекст (или контексты) среди активных с целью получения более точной оценки, ведь текущий символ может лучше предсказываться не контекстом 2-го порядка "аб", а контекстом 1-го порядка "б";
- как оценивать вероятность символов, имеющих нулевую частоту (например, "г").

Выше были приведены цифры, в соответствии с которыми при увеличении длины используемого контекста сжатие данных улучшается. К сожалению, при кодировании блоков типичной длины – одного или нескольких мегабайтов и меньше – это справедливо только для небольших порядков модели, так как статистика для длинных контекстов медленно накапливается. При этом также следует учитывать, что большинство реальных данных характеризуется неоднородностью, нестабильностью силы и вида статистических взаимосвязей, поэтому "старая" статистика контекстно-зависимых частот появления символов малополезна или даже вредна. Поэтому модели, строящие оценку только на основании информации КМ максимального порядка N, обеспечивают сравнительно низкую точность предсказания. Кроме того, хранение модели большого порядка требует много памяти.

Если в модели используются для оценки только КМ(N), то иногда такой подход называют "чистым" (pure) контекстным моделированием порядка N. Из-за вышеуказанного недостатка "чистые" модели представляют обычно только научный интерес.

Действительно, реально используемые файлы обычно имеют сравнительно небольшой размер, поэтому для улучшения их сжатия необходимо учитывать оценки вероятностей, получаемые на основании статистики контекстов разных длин. Техника объединения оценок вероятностей, соответствующих отдельным активным контекстам, в одну оценку называется

смешиванием (blending). Известно несколько способов выполнения смешивания.

Рассмотрим модель произвольного порядка N. Если $q(s_i|o)$ есть вероятность, присваиваемая в активной КМ(o) символу s_i алфавита сжимаемого потока, то смешанная вероятность $q(s_i)$ вычисляется в общем случае как

$$q(s_i) = \sum_{o=-1}^{N} w(o) q(s_i \mid o),$$

где $w(o)$ – вес оценки КМ(o).

Оценка $q(s_i|o)$ обычно определяется через частоту символа s_i по тривиальной формуле

$$q(s_i \mid o) = \frac{f(s_i \mid o)}{f(o)},$$

где $f(s_i|o)$ – частота появления символа s_i в соответствующем контексте порядка o; $f(o)$ – общая частота появления соответствующего контекста порядка o в обработанной последовательности.

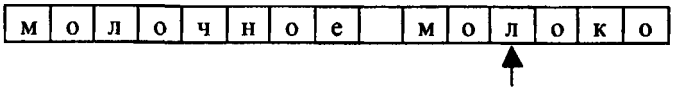
Заметим, что правильнее было бы писать не, скажем, $f(s_i|o)$, а $f(s_i|C_{j(o)})$, т. е. "частота появления символа s_i в КМ порядка o с номером $j(o)$", поскольку контекстных моделей порядка o может быть огромное количество. Но при сжатии каждого текущего символа мы рассматриваем только *одну* КМ для каждого порядка, так как контекст определяется *непосредственно* примыкающей слева к символу строкой определенной длины. Иначе говоря, для каждого символа мы имеем набор из N+1 активных контекстов длины от N до 0, каждому из которых однозначно соответствует только одна КМ, если она вообще есть. Поэтому здесь и далее используется сокращенная запись.

Если вес $w(-1) > 0$, то это гарантирует успешность кодирования любого символа входного потока, так как наличие КМ(–1) позволяет всегда получать ненулевую оценку вероятности и, соответственно, код конечной длины.

Различают модели с полным смешиванием (fully blended), когда предсказание определяется статистикой КМ всех используемых порядков, и с частичным смешиванием (partially blended) – в противном случае.

Пример 1

Рассмотрим процесс оценки отмеченного на рисунке стрелкой символа "л", встретившегося в блоке "молочное_молоко". Считаем, что модель работает на уровне символов.



Пусть мы используем контекстное моделирование порядка 2 и делаем полное смешивание оценок распределений вероятностей в КМ 2-го, 1-го и 0-го порядка с весами 0.6, 0.3 и 0.1. Считаем, что в начале кодирования в КМ(0) создаются счетчики для всех символов алфавита {"м", "о", "л", "ч", "н", "е", "_", "к"} и инициализируются единицей; счетчик символа после его обработки увеличивается на единицу.

Для текущего символа "л" имеются контексты "мо", "о" и пустой (0-го порядка). К данному моменту для них накоплена статистика, показанная в табл. 4.1.

Таблица 4.1

Символы		"м"	"о"	"л"	"ч"	"н"	"е"	"_"	"к"
КМ порядка 0 (контекст "")	Частоты	3	5	2	2	2	2	2	1
	Накопленные частоты	3	8	10	12	14	16	18	19
КМ порядка 1 (контекст "о")	Частоты	-	-	1	1	-	1	-	-
	Накопленные частоты	-	-	1	2	-	3	-	-
КМ порядка 2 ("мо")	Частоты	-	-	1	-	-	-	-	-
	Накопленные частоты	-	-	1	-	-	-	-	-

Тогда оценка вероятности для символа "л" будет равна

$$q('л') = 0.1 \cdot \frac{2}{19} + 0.3 \cdot \frac{1}{3} + 0.6 \cdot \frac{1}{1} = 0,71 .$$

В общем случае для однозначного кодирования символа "л" такую оценку необходимо проделать для всех символов алфавита. Действительно, с одной стороны, декодер не знает, чему равен текущий символ, с другой стороны, оценка вероятности не гарантирует уникальности кода, а лишь задает его длину. Поэтому статистическое кодирование выполняется на основании накопленной частоты (см. подробности в примере 2 и в подразд. "Арифметическое сжатие" гл. 1). Например, если кодировать на основании статистики только 0-го порядка, то существует взаимно-однозначное соответствие между накопленными частотами из диапазона (8,10] и символом "л", что не имеет места в случае просто частоты (частоту 2 имеют еще 4 символа). По-

нятно, что аналогичные свойства остаются в силе и в случае оценок, получаемых частичным смешиванием.

Упражнение. Предложите способы увеличения средней скорости вычисления оценок для методов контекстного моделирования со смешиванием, как полным, так и частичным.

Очевидно, что успех применения смешивания зависит от способа выбора весов $w(o)$. Простой путь состоит в использовании заданного набора фиксированных весов КМ разных порядков при каждой оценке; этот способ был применен в примере 2. Естественно, альтернативой является адаптация весов по мере кодирования. Приспособление может заключаться в придании все большей значимости КМ все больших порядков или, скажем, попытке выбрать наилучшие веса на основании определенных статистических характеристик последнего обработанного блока данных. Но так исторически сложилось, что реальное развитие получили методы неявного взвешивания. Это объясняется в первую очередь их меньшей вычислительной сложностью.

Техника неявного взвешивания связана с введением вспомогательного символа ухода (escape). Символ ухода является квазисимволом и не должен принадлежать к алфавиту сжимаемой последовательности. Фактически он используется для передачи декодеру указаний кодера. Идея заключается в том, что если используемая КМ не позволяет оценить текущий символ (его счетчик равен нулю в этой КМ), то на выход посылается закодированный символ ухода и производится попытка оценить текущий символ в другой КМ, которой соответствует контекст иной длины. Обычно попытка оценки начинается с КМ наибольшего порядка N, затем в определенной последовательности осуществляется переход к контекстным моделям меньших порядков.

Естественно, статистическое кодирование символа ухода выполняется на основании его вероятности, так называемой вероятности ухода. Очевидно, что символы ухода порождаются не источником данных, а моделью. Следовательно, их вероятность может зависеть от характеристик сжимаемых данных, свойств КМ, с которой производится уход, свойств КМ, на которую происходит уход, и т. д. Как можно оценить эту вероятность, имея в виду, что конечный критерий качества – улучшение сжатия? Вероятность ухода – это вероятность появления в контексте нового символа. Тогда фактически необходимо оценить правдоподобность наступления ни разу не происходившего события. Теоретического фундамента для решения этой проблемы, видимо, не существует, но за время развития техник контекстного моделирования было предложено несколько подходов, хорошо работающих в большинстве реальных ситуаций. Кроме того, эксперименты показывают, что модели с неявным взвешиванием устойчивы относительно используемого метода оценки вероятности ухода, т. е. выбор какого-то способа вы-

числения этой величины не влияет на коэффициент сжатия кардинальным образом.

Алгоритмы PPM

Техника контекстного моделирования Prediction by Partial Matching (предсказание по частичному совпадению), предложенная в 1984 г. Клири (Cleary) и Уиттеном (Witten) [5], является одним из самых известных подходов к сжатию качественных данных и уж точно самым популярным среди контекстных методов. Значимость подхода обусловлена и тем фактом, что алгоритмы, причисляемые к PPM, неизменно обеспечивают в среднем наилучшее сжатие при кодировании данных различных типов и служат стандартом, "точкой отсчета" при сравнении универсальных алгоритмов сжатия.

Перед собственно рассмотрением алгоритмов PPM необходимо сделать замечание о корректности используемой терминологии. На протяжении примерно 10 лет – с середины 80-х гг. до середины 90-х – под PPM понималась группа методов с вполне определенными характеристиками. В последние годы, вероятно из-за резкого увеличения числа всевозможных гибридных схем и активного практического использования статистических моделей для сжатия, произошло смешение понятий и термин "PPM" часто используется для обозначения контекстных методов вообще.

Ниже будет описан некий обобщенный алгоритм PPM, а затем особенности конкретных распространенных схем.

Как и в случае многих других контекстных методов, для каждого контекста, встречаемого в обрабатываемой последовательности, создается своя контекстная модель КМ. При этом под контекстом понимается последовательность элементов одного типа – символов, пикселов, чисел, но не набор разнородных объектов. Далее вместо слова "элемент" мы будем использовать слово "символ". Каждая КМ включает в себя счетчики всех символов, встреченных в соответствующем контексте.

PPM относится к адаптивным методам моделирования. Исходно кодеру и декодеру поставлена в соответствие начальная модель источника данных. Будем считать, что она состоит из КМ(–1), присваивающей одинаковую вероятность всем символам алфавита входной последовательности. После обработки текущего символа кодер и декодер изменяют свои модели одинаковым образом, в частности наращивая величину оценки вероятности рассматриваемого символа. Следующий символ кодируется (декодируется) на основании новой, измененной модели, после чего модель снова модифицируется и т. д. На каждом шаге обеспечивается идентичность модели кодера и декодера за счет применения одинакового механизма ее обновления.

В PPM используется неявное взвешивание оценок. Попытка оценки символа начинается с КМ(N), где N является параметром алгоритма и называется порядком PPM-модели. В случае нулевой частоты символа в КМ текущего порядка осуществляется переход к КМ меньшего порядка за счет использования механизма уходов (escape strategy), рассмотренного в предыдущем подразд.

Фактически, вероятность ухода – это суммарная вероятность всех символов алфавита входного потока, еще ни разу не появлявшихся в контексте. Любая КМ должна давать отличную от нуля оценку вероятности ухода. Исключения из этого правила возможны только в тех случаях, когда значения всех счетчиков КМ для всех символов алфавита отличны от нуля, т. е. любой символ может быть оценен в рассматриваемом контексте. Оценка вероятности ухода традиционно является одной из основных проблем алгоритмов с неявным взвешиванием, и она будет специально рассмотрена ниже в подразд. "Оценка вероятности ухода".

Вообще говоря, способ моделирования источника с помощью классических алгоритмов PPM опирается на следующие предположения о природе источника:
1) источник является марковским с порядком N, *т. е. вероятность генерации символа зависит от* N *предыдущих символов и только от них;*
2) источник имеет такую дополнительную особенность, что чем ближе располагается один из символов контекста к текущему символу, тем больше корреляция между ними.

Таким образом, механизм уходов первоначально рассматривался лишь как вспомогательный прием, позволяющий решить проблему кодирования символов, ни разу не встречавшихся в контексте порядка N. В идеале, достигаемом после обработки достаточно длинного блока, никакого обращения к КМ порядка меньше N происходить не должно. Иначе говоря, причисление классических алгоритмов PPM к методам, производящим взвешивание, пусть и неявным образом, является не вполне корректным.

При сжатии очередного символа выполняются следующие действия.

Если символ s обрабатывается с использованием PPM-модели порядка N, то, как мы уже отмечали, в первую очередь рассматривается КМ(N). Если она оценивает вероятность s числом, не равным нулю, то сама и используется для кодирования s. Иначе выдается сигнал в виде символа ухода, и на основе меньшей по порядку КМ(N–1) производится очередная попытка оценить вероятность s. Кодирование происходит через уход к КМ меньших порядков до тех пор, пока s не будет оценен. КМ(–1) гарантирует, что это в конце концов произойдет. Таким образом, каждый символ кодируется серией кодов символа ухода, за которой следует код самого символа. Из этого

следует, что вероятность ухода также можно рассматривать как вероятность перехода к контекстной модели меньшего порядка.

Если в процессе оценки обнаруживается, что текущий рассматриваемый контекст встречается в первый раз, то для него создается КМ.

При оценке вероятности символа в КМ порядка $o < N$ можно исключить из рассмотрения все символы, которые содержатся в КМ(o+1), поскольку ни один из них точно не является символом s. Для этого в текущей КМ(o) нужно замаскировать, т. е. временно установить в нуль, значения счетчиков всех символов, имеющихся в КМ(o+1). Такая техника называется методом исключения (exclusion).

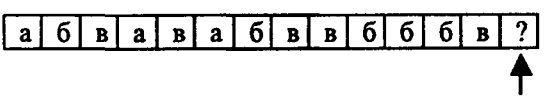
После собственно кодирования символа обычно осуществляется обновление статистики всех КМ, использованных при оценке его вероятности, за исключением статической КМ(–1). Такой подход называется методом исключения при обновлении. Простейшим способом модификации является инкремент счетчиков символа в этих КМ. Подробнее о стратегиях обновления будет сказано в подразд. "Обновление счетчиков символов".

Пример работы алгоритма PPM

Рассмотрим подробнее работу алгоритма PPM с помощью примера.

Пример 2

Имеется последовательность символов "абвавабввбббв" алфавита {"а", "б", "в", "г"}, которая уже была закодирована.



Пусть счетчик символа ухода равен единице для всех КМ, при обновлении модели счетчики символов увеличиваются на единицу во всех активных КМ, применяется метод исключения и максимальная длина контекста равна трем, т. е. $N = 3$.

Первоначально модель состоит из КМ(–1), в которой счетчики всех четырех символов алфавита имеют значение 1. Состояние модели обработки последовательности "абвавабввбббв" представлено на рис. 4.2, где прямоугольниками обозначены контекстные модели, при этом для каждой КМ указан курсивом контекст, а также встречавшиеся в контексте символы и их частоты.

Пусть текущий символ равен "г", т. е. "?" = "г", тогда процесс его кодирования будет выглядеть следующим образом.

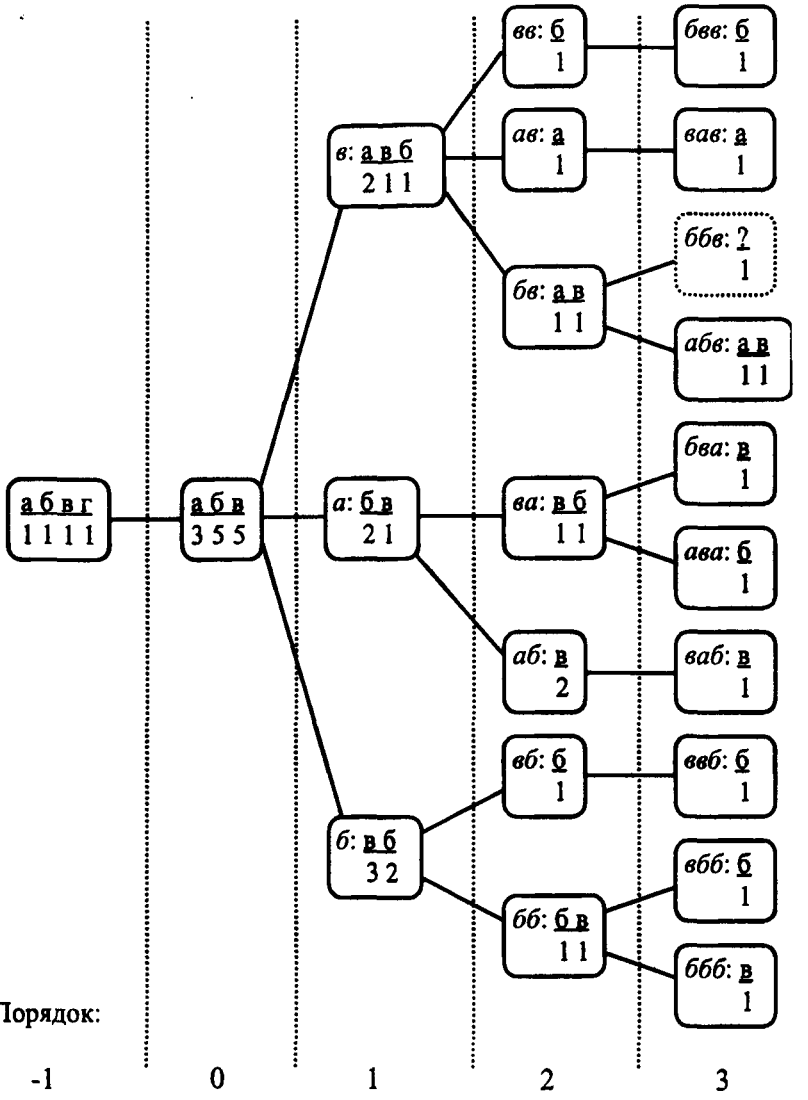


Рис. 4.2. Состояние модели после обработки последовательности "абвавабввббв"

Сначала рассматривается контекст 3-го порядка "ббв". Ранее он не встречался, поэтому кодер, ничего не послав на выход, переходит к анализу статистики для контекста 2-го порядка. В этом контексте ("бв") встречались символ "а" и символ "в", счетчики которых в соответствующей КМ равны 1

каждый, поэтому символ ухода кодируется с вероятностью 1/(2+1), где в знаменателе число 2 – это наблюдавшаяся частота появления контекста "бв", 1 – это значение счетчика символа ухода. В контексте 1-го порядка "в" дважды встречался символ "а", который исключается (маскируется), один раз также исключаемый "в" и один раз "б", поэтому оценка вероятности ухода будет равна 1/(1+1). В КМ(0) символ "г" также оценить нельзя, причем все имеющиеся в этой КМ символы "а", "б", "в" исключаются, так как уже встречались нам в КМ более высокого порядка. Поэтому вероятность ухода получается равной единице. Цикл оценивания завершается на уровне КМ(–1), где "г" к этому времени остается единственным до сих пор не попадавшимся символом, поэтому он получает вероятность 1 и кодируется посредством 0 бит. Таким образом, при использовании хорошего статистического кодировщика для представления "г" потребуется в целом примерно 2.6 бита.

Перед обработкой следующего символа создается КМ для строки "ббв" и производится модификация счетчиков символа "г" в созданной и во всех просмотренных КМ. В данном случае требуется изменение КМ всех порядков от 0 до *N*.

Табл. 4.2 демонстрирует оценки вероятностей, которые должны были быть использованы при кодировании символов алфавита {"а", "б", "в", "г"} в текущей позиции.

Таблица 4.2

Символ s	Последовательность оценок для КМ каждого порядка от 3 до –1					Общая оценка вероятности q(s)	Представление требует битов
	3	2	1	0	–1		
	"ббв"	"бв"	"в"	""			
"а"	-	$\frac{1}{2+1}$	-	-	-	$\frac{1}{3}$	1.6
"б"	-	$\frac{1}{2+1}$	$\frac{1}{1+1}$	-	-	$\frac{1}{6}$	2.6
"в"	-	$\frac{1}{2+1}$	-	-	-	$\frac{1}{3}$	1.6
"г"	-	$\frac{1}{2+1}$	$\frac{1}{1+1}$	1	1	$\frac{1}{6}$	2.6

Алгоритм декодирования абсолютно симметричен алгоритму кодирования. После декодирования символа в текущей КМ проверяется, не является ли он символом ухода; если это так, то выполняется переход к КМ порядком ниже. Иначе считается, что исходный символ восстановлен, он записы-

вается в декодированный поток и осуществляется переход к следующему шагу. Содержание процедур обновления счетчиков, создания новых контекстных моделей, прочих вспомогательных действий и последовательность их применения должны быть строго одинаковыми при кодировании и декодировании. Иначе возможна рассинхронизация копий модели кодера и декодера, что рано или поздно приведет к ошибочному декодированию какого-то символа. Начиная с этой позиции вся оставшаяся часть сжатой последовательности будет разжата неправильно.

Разница между кодами символов, оценки вероятности которых одинаковы, достигается за счет того, что PPM-предсказатель передает кодировщику так называемые накопленные частоты (или накопленные вероятности) оцениваемого символа и его соседей или кодовые пространства символов. Так, например, для контекста "бв" из примера 2 можно составить табл. 4.3.

Таблица 4.3

Символ	Частота	Оценка вероятности	Накопленная вероятность (оценка)	Кодовое пространство
"а"	1	1/3	1/3	[0 ... 0.33)
"б"	0	-	-	-
"в"	1	1/3	2/3	[0.33 ... 0.66)
"г"	0	-	-	-
Уход	1	1/3	1	[0.66 ... 1)

Хороший кодировщик должен отобразить символ "s" с оценкой вероятности $q(s)$ в код длины $\log_2 q(s)$, что и обеспечит сжатие всей обрабатываемой последовательности в целом.

В обобщенном виде алгоритм кодирования можно записать так.

```
/*инициализация контекста длины N (в смысле строки предыдущих
   символов), эта строка должна содержать N предыдущих
   символов, определяя набор активных контекстов длины o ≤ N
*/
context = "";
while ( ! DataFile.EOF() ){
   c = DataFile.ReadSymbol(); // текущий символ
   order = N; // текущий порядок КМ
   success = 0; // успешность оценки в текущей КМ
   do{
      // найдем КМ для контекста текущей длины
      CM = ContextModel.FindModel (context, order);
      /*попробуем найти текущий символ c в этой КМ, в
        CumFreq получим его накопленную частоту (или
        накопленную частоту символа ухода), в counter -
        ссылку на счетчик символа; флаг success указывает
```

```
        на отсутствие ухода
      */
      success = CM.EvaluateSymbol (c, &CumFreq, counter);
      /*запомним в стеке КМ и указатель на счетчик для
        последующего обновления модели
      */
      Stack.Push (CM, counter);
      // закодируем с или символ ухода
      StatCoder.Encode (CM, CumFreq, counter);
      order--;
    }while ( ! success );
    /*обновим модель: добавим КМ в случае необходимости,
      изменим значения счетчиков и т. д.
    */
    UpdateModel (Stack);
    // обновим контекст: сдвинем влево, справа добавим с
    MoveContext (c);
}
```

Пример реализации PPM-компрессора

Рассмотрим основные моменты реализации компрессора PPM для простейшего случая с порядком модели $N = 1$ без исключения символов. Будем также исходить из того, что статистическое кодирование выполняется арифметическим кодером.

При контекстном моделировании 1-го порядка нам не требуются сложные структуры данных, обеспечивающие эффективное хранение и доступ к информации отдельных КМ. Можно просто хранить описания КМ в одномерном массиве, размер которого равен количеству символов в алфавите входной последовательности, и находить нужную КМ, используя символ ее контекста как индекс. Мы используем байт-ориентированное моделирование, поэтому размер массива для контекстных моделей порядка 1 будет равен 256. Чтобы не плодить лишних сущностей, мы, во-первых, откажемся от КМ(–1) за счет соответствующей инициализации КМ(0) и, во-вторых, будем хранить КМ(0) в том же массиве, что и КМ(1). Считаем, что КМ(0) соответствует индекс 256.

В структуру контекстной модели `ContextModel` включим массив счетчиков `count` для всех возможных 256 символов. Для символа ухода введем в структуру КМ специальный счетчик `esc`, а также добавим поле `TotFr`, в котором будет содержаться сумма значений счетчиков всех обычных символов. Использование поля `TotFr` не обязательно, но позволит ускорить обработку данных.

С учетом сказанного структуры данных компрессора будут такими:

```
struct ContextModel{
   int   esc,
         TotFr;
   int   count[256];
};
ContextModel cm[257];
```

Если размер типа int равен 4 байтам, то нам потребуется не менее 257 Кб памяти для хранения модели.

Опишем стек, в котором будут храниться указатели на требующие модификации КМ, а также указатель стека SP и контекст context.

```
ContextModel *stack[2];
int          SP,
             context [1]; //контекст вырождается в 1 символ
```

Больше никаких глобальных переменных и структур данных нам не нужно.

Инициализацию модели будем выполнять в общей для кодера и декодера функции init_model.

```
void init_model (void){
   /*Так как cm является глобальной переменной, то значения
     всех полей равны нулю. Нам требуется только распределить кодовое пространство в КМ(0) так, чтобы
     все символы,включая символ ухода, всегда бы имели
     ненулевые оценки.
     Пусть также символы будут равновероятными
   */
   for ( int j = 0; j < 256; j++ )
     cm[256].count[j] = 1 ;
   cm[256].TotFr = 256;
   /*Явно запишем, что в начале моделирования мы считаем
     контекст равным нулю. Число не имеет значения, лишь бы
     кодер и декодер точно следовали принятым
     соглашениям. Обратите на это внимание
   */
   context [0] = 0;
   SP = 0;
}
```

Функции обновления модели также будут общими для кодера и декодера. В update_model производится инкремент счетчиков просмотренных КМ, а в rescale осуществляется масштабирование счетчиков. Необходимость масштабирования обусловлена особенностями типичных реализаций ариф-

метического кодирования и заключается в делении значений счетчиков пополам при достижении суммы значений всех счетчиков TotFr+esc некоторого порога. Подробнее об этом рассказано в подразд. "Обновление счетчиков символов" этой главы.

```
const int MAX_TotFr = 0x3fff;
void rescale (ContextModel *CM){
   CM->TotFr = 0;
   for (int i = 0; i < 256; i++){
      /*обеспечим отличие от нуля значения
        счетчика после масштабирования
      */
      CM->count[i] -= CM->count[i] >> 1;
      CM->TotFr += CM->count[i];
   }
}

void update_model (int c){
   while (SP) {
      SP--;
      if ((stack[SP]->TotFr + stack[SP]->esc) >= MAX_TotFr)
         rescale (stack[SP]);
      if (!stack[SP]->count[c])
         /*в этом контексте это новый символ, увеличим
           счетчик уходов
         */
         stack[SP]->esc += 1;
      stack[SP]->count[c] += 1;
      stack[SP]->TotFr += 1;
   }
}
```

Собственно кодер реализуем функцией encode. Эта функция управляет последовательностью действий при сжатии данных, вызывая вспомогательные процедуры в требуемом порядке, а также находит нужную КМ. Оценка текущего символа производится в функции encode_sym, которая передает результаты своей работы арифметическому кодеру.

```
int encode_sym (ContextModel *CM, int c){
   // КМ потребует инкремента счетчиков, запомним ее
   stack [SP++] = CM;

   if (CM->count[c]){
      /*счетчик сжимаемого символа не равен нулю, тогда
        его можно оценить в текущей КМ; найдем
        накопленную частоту предыдущего в массиве count
        символа
```

```
      */
      int CumFreqUnder = 0;
      for (int i = 0; i < c; i++)
         CumFreqUnder += CM->count[i];
      /*передадим описание кодового пространства,
         занимаемого символом c, арифметическому кодеру
      */
      AC.encode (CumFreqUnder, CM->count[c],
                 CM->TotFr + CM->esc);
      return 1; // возвращаемся в encode с победой
   }else{
      /*нужно уходить на КМ(0);
         если текущий контекст 1-го порядка встретился первый
         раз, то заранее известно, что его КМ пуста (все
         счетчики равны нулю) и кодировать уход не только не
         имеет смысла, но и нельзя, так как TotFr+esc = 0
      */
      if (CM->esc)
         AC.encode (CM->TotFr, CM->esc, CM->TotFr + CM->esc)
      ;
      return 0; // закодировать символ не удалось
   }
}

void encode (void){
   int  c, // текущий символ
        success;  // успешность кодирования символа в КМ
   init_model ();
   AC.StartEncode ();   // проинициализируем арифм. кодер
   while (( c = DataFile.ReadSymbol() ) != EOF) {
      // попробуем закодировать в КМ(1)
      success = encode_sym (&cm[context[0]], c);
      if (!success)
         /*уходим на КМ(0), где любой символ получит
            ненулевую оценку и будет закодирован
         */
         encode_sym (&cm[256], c);
      update_model (c);
      context [0] = c; // сдвинем контекст
   }
   // закодируем знак конца файла символом ухода с КМ(0)
   AC.encode (cm[context[0]].TotFr, cm[context[0]].esc,
      cm[context[0]].TotFr + cm[context[0]].esc);
   AC.encode (cm[256].TotFr, cm[256].esc,
      cm[256].TotFr + cm[256].esc);
   // завершим работу арифметического кодера
   AC.FinishEncode();
}
```

Реализация декодера выглядит аналогично. Внимания заслуживает разве что только процедура поиска символа по описанию его кодового пространства. Метод get_freq арифметического кодера возвращает число x, лежащее в диапазоне [CumFreqUnder, CumFreqUnder+CM->count[i]), т. е. CumFreqUnder $\leq x <$ < CumFreqUnder+CM->count[i]. Поэтому искомым символом является i, для которого выполнится это условие.

```
int decode_sym (ContextModel *CM, int *c){
   stack [SP++] = CM;
   if (!CM->esc) return 0;

   int cum_freq = AC.get_freq (CM->TotFr + CM->esc);
   if (cum_freq < CM->TotFr){
      /*символ был закодирован в этой КМ; найдем символ и
        его точное кодовое пространство
      */
      int CumFreqUnder = 0;
      int i = 0;
      for (;;){
         if ( (CumFreqUnder + CM->count[i]) <= cum_freq)
            CumFreqUnder += CM->count[i];
         else break;
         i++;
      }
      /*обновим  состояние арифметического кодера
        на основании точной накопленной частоты символа
      */
      AC.decode_update (CumFreqUnder, CM->count[i],
                        CM->TotFr + CM->esc);
      *c = i;
      return 1;
   }else{
      /*обновим  состояние арифметического кодера на
        основании точной накопленной частоты символа,
        оказавшегося символом ухода
      */
      AC.decode_update (CM->TotFr, CM->esc,
                        CM->TotFr + CM->esc);
      return 0;
   }
}

void decode (void){
   int  c,
        success;
   init_model ();
```

```
    AC.StartDecode ();
    for (;;){
      success = decode_sym (&cm[context[0]], &c);
      if (!success){
        success = decode_sym (&cm[256], &c);
        if (!success) break; //признак конца файла
      }
      update_model (c);
      context [0] = c;
      DataFile.WriteSymbol (c);
    }
}
```

Характеристики созданного компрессора, названного Dummy, приведены в подразд. "Компрессоры и архиваторы, использующие контекстное моделирование" (см. "Производительность на тестовом наборе Calgary Compression Corpus"). Полный текст реализации Dummy оформлен в виде приложения 1.

Оценка вероятности ухода

На долю символов ухода обычно приходится порядка 30% и более от всех оценок, вычисляемых моделировщиком PPM. Это определило пристальное внимание к проблеме оценки вероятности символов с нулевой частотой. Львиная доля публикаций, посвященных PPM, прямо касаются оценки вероятности ухода (ОВУ).

Можно выделить два подхода к решению проблемы ОВУ: априорные методы, основанные на предположениях о природе сжимаемых данных, и адаптивные методы, которые пытаются приспособить оценку к данным. Понятно, что первые призваны обеспечить хороший коэффициент сжатия при обработке типичных данных в сочетании с высокой скоростью вычислений, а вторые ориентированы на обеспечение максимально возможной степени сжатия.

Априорные методы

Введем обозначения:

C – общее число просмотров контекста, т. е. сколько раз он встретился в обработанном блоке данных;

S – количество разных символов в контексте;

$S^{(i)}$ – количество таких разных символов, что они встречались в контексте ровно i раз;

$E^{(x)}$ – значение ОВУ по методу x.

Изобретатели алгоритма PPM предложили два метода ОВУ: так называемые метод A и метод B. Использующие их алгоритмы PPM были названы PPMA и PPMB соответственно.

В дальнейшем было описано еще 5 априорных подходов к ОВУ: методы C, D, P, X и XC [8, 10, 17]. По аналогии с PPMA и PPMB алгоритмы PPM, применяющие методы C и D, получили названия PPMC и PPMD соответственно.

Идея методов и их сравнение представлены в табл. 4.4 и табл. 4.5.

Таблица 4.4

Метод	$E^{(x)} =$
A	$\frac{1}{C+1}$
B	$\frac{S-S^{(1)}}{C}$
C	$\frac{S}{C+S}$
D	$\frac{S}{2C}$
P	$\frac{S^{(1)}}{C}-\frac{S^{(2)}}{C^2}+\frac{S^{(3)}}{C^3}-\ldots$
X	$\frac{S^{(1)}}{C}$
XC	$\begin{cases}\frac{S^{(1)}}{C}, & \text{при } 0<S^{(1)}<C \\ E^{(C)}, & \text{в противном случае}\end{cases}$

Кстати, в примере 2 был использован метод A, а в компрессоре Dummy – метод C.

При реализации метода B воздерживаются от оценки символов до тех пор, пока они не появятся в текущем контексте более одного раза. Это достигается за счет вычитания единицы из счетчиков. Методы P, X, XC базируются на предположении о том, что вероятность появления в обрабатываемых данных символа s_i подчиняется закону Пуассона с параметром λ_i.

Таблица 4.5

Тип файлов	Точность предсказания						
	Лучше			→			Хуже
Тексты	XC	D	P	X	C	B	A
Двоичные файлы	C	X	P	XC	D	B	A

Места в табл. 4.5 очень условны. Так, например, при сжатии текстов методы XC, D, P, X показывают весьма близкие результаты и многое зависит от порядка модели и используемых для сравнения файлов. В большинстве случаев существенным является только отставание точности ОВУ по способам A и B от других методов.

Упражнение. Выполните действия, описанные в примере 2, используя ОВУ по методу C. Если текущий символ "б", то точность его предсказания улучшится, останется неизменной или ухудшится?

Адаптивные методы

Чтобы улучшить оценку вероятности ухода, необходимо иметь такую модель оценки, которая бы адаптировалась к обрабатываемым данным. Подобный адаптивный механизм получил название Secondary Escape Estimation (SEE), т. е. дополнительной оценки ухода или вторичной оценки ухода. Метод заключается в тривиальном вычислении вероятности ухода из текущей КМ через частоту появления новых символов (или, что то же, символов ухода) в контекстных моделях со схожими характеристиками:

$$E^{(SEE)}(i) = \frac{f_i(esc)}{n_i},$$

где $f_i(esc)$ – число наблюдавшихся уходов из контекстных моделей типа i; n_i – число просмотров контекстных моделей типа i.

Вразумительные обоснования выбора этих характеристик и критериев "схожести" при отсутствии априорных знаний о характере сжимаемой последовательности дать сложно, поэтому известные алгоритмы адаптивной оценки базируются на эмпирическом анализе типовых данных.

Метод Z

Одна из самых ранних попыток реализации SEE известна как метод Z, а использующая его разновидность алгоритма PPM – PPMZ [3]. Для точности описания этой техники SEE объект "контекст" ниже будет также именоваться "PPM-контекстом".

Для нахождения ОВУ строятся так называемые контексты ухода (escape contexts) (КУ), формируемые из четырех полей. В полях КУ содержится информация о значениях следующих величин: последние 4 символа PPM-

контекста, порядок PPM-контекста, количество уходов и количество успешных оценок в соответствующей КМ. Нескольким КМ может соответствовать один КУ.

Информация о фактическом количестве уходов и успешных кодирований во всех контекстных моделях, имеющих общий КУ, запоминается в счетчиках контекстной модели уходов КМУ, построенной для данного КУ. Эта информация определяет ОВУ для текущей КМ. ОВУ находится путем взвешивания оценок, которые дают три КМУ (КМУ порядка 2, 1 и 0), отвечающие характеристикам текущей КМ.

КУ порядка 2 наиболее точно соответствует текущей КМ, контексты ухода порядком ниже формируются главным образом путем выбрасывания части информации из полей КУ порядка 2. Компоненты КУ порядка 2 определяются в соответствии с табл. 4.6 [3].

Таблица 4.6

Номер поля	Размер, бит	Способ формирования значения поля	
		Параметр и его значения	Значение поля
1	2	Порядок КМ	Порядок КМ/2 (с округлением до младшего)
2	2	Количество уходов из КМ	
		1	0
		2	1
		3	2
		> 3	3
3	3	Количество успешных оценок в КМ	
		0	0
		1	1
		2	2
		3, 4	3
		5, 6	4
		7, 8, 9	5
		10, 11, 12	6
		> 12	7
4	9	X_1 = 7 младших бит последнего (только что обработанного) символа PPM-контекста; X_2 = шестой и пятый биты предпоследнего символа; т. е., если расписать байт как совокупность 8 бит xXXxxxxx, то это биты XX	$((X_2\&0x60) << 2) \mid X_1$

В состав КУ всех порядков входят поля 1, 2, 3. Для КУ порядка 1 поле 4 состоит из 8 бит и строится из шестых и пятых битов последних четырех обработанных символов. У КУ порядка 0 четвертое поле отсутствует. Очевидно, что алгоритм построения поля 4 для КУ порядков 1 и 2 призван улучшить предсказание ухода для текстов на английском языке в кодировке ASCII. Аналогичный прием, хотя и в не столь явном виде, используется в адаптивных методах ОВУ SEE-d1 и SEE-d2, рассмотренных ниже.

При взвешивании статистики КМУ(n) используются следующие веса w_n:

$$\frac{1}{w_n} = e \cdot \log_2\left(\frac{1}{e}\right) + (1-e) \cdot \log_2\left(\frac{1}{1-e}\right),$$

где e – ОВУ, которую дает данная взвешиваемая КМУ(n); формируется из фактического количества уходов и успешных кодирований в контекстных моделях, соответствующих этой КМУ, или, иначе, определяется наблюдавшейся частотой ухода из таких КМ.

Окончательная оценка:

$$E^{(Z)} = \frac{\sum_{n=0}^{2} e_n w_n}{\sum_{n=0}^{2} w_n}.$$

После ОВУ выполняется поиск текущего символа среди имеющихся в КМ. По результатам поиска (символ найден или нет) обновляются счетчики соответствующих трех КМУ порядка 0, 1 и 2.

Методы SEE-d1 и SEE-d2

Современным адаптивным методом является подход, предложенный Шкариным и успешно применяемый в компрессорах PPMd и PPMonstr [1]. При каждой оценке используется КУ только какого-то одного типа (в методе Z их 3), но в зависимости от требований, предъявляемых к компрессору, автор предлагает применять один из двух методов. Назовем первый метод SEE-d1, а второй – SEE-d2.

Метод SEE-d1 используется в компрессоре PPMd и ориентирован на хорошую точность оценки при небольших вычислительных затратах. Рассмотрим его подробно.

Все КМ разобьем на 3 типа:

- детерминированные (или бинарные), содержащие только один символ; назовем их контекстными моделями типа d;
- с незамаскированными символами, т. е. ни один из символов, имеющихся в данной КМ, не встречался в КМ бóльших порядков; обычно это КМ

максимального порядка или КМ, с которой началась оценка; назовем их контекстными моделями типа nm;

- с замаскированными символами; назовем их контекстными моделями типа m.

Как показали эксперименты, вероятность ухода из КМ разных типов коррелирует с разными характеристиками.

Случай КМ типа d является наиболее простым, так как у такой модели малое число степеней свободы. Естественно, наибольшее влияние на вероятность ухода оказывает счетчик частоты единственного символа.

Ниже изложен способ формирования КУ, при этом описание полей отсортировано в порядке убывания степени их влияния на точность оценки ухода.

1. Счетчик частоты символа квантуется до 128 значений.
2. Существует сильная взаимосвязь между родительскими и дочерними КМ, поэтому число символов в родительской КМ квантуется до четырех значений.
3. При сжатии реальных данных характерно чередование блоков хорошо предсказуемых символов с плохо предсказуемыми. Включим в КУ оценку вероятности предыдущего символа, чтобы отслеживать переходы между такими блоками. Величина квантуется до двух значений.
4. Существует сильная статистическая взаимосвязь между текущим и предыдущим символами. Введем 1-битовый флаг, принимающий значение 0, если 2 старших бита предыдущего символа нулевые, и значение 1 в прочих случаях.
5. Кроме чередования блоков хорошо и плохо предсказуемых символов, часто встречаются длинные блоки очень хорошо предсказуемых данных. Это обычно бывает в случае множественных повторов длинных строк в сжимаемом потоке. Часто PPM-модели небольших порядков плохо работают в таких ситуациях. Введем 1-битовый флаг, свидетельствующий о нахождении в таком блоке. Флаг принимает значение 1, если при обработке предыдущих символов ни разу не происходил уход и оценки вероятности превышали 0.5 для L или большего количества этих символов. L обычно равно порядку PPM-модели.
6. Возможность ухода зависит от единственного символа КМ типа d. Пусть соответствующий флаг равен нулю, если 2 старших бита символа нулевые, и единице в остальных случаях.

Таким образом, всего возможно $128 \cdot 4 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 8192$ контекста ухода для КМ типа d.

В случае КМ типа nm адаптивная оценка затруднена из-за небольшой частоты их использования, что приводит к отсутствию представительной статистики в большинстве случаев. Поэтому применим полуадаптивный метод, доказавший на практике свою эффективность. Допустим, распределение символов геометрическое:

$$p(s_i^k \mid o) = p^k(1-p),$$

где $p(s_i^k \mid o)$ – вероятность появления символа s_i в заданной КМ(o) после серии из k иных символов.

Иначе говоря, $p(s_i^k \mid o)$ – вероятность успеха в первый раз после ровно k *испытаний по схеме Бернулли при вероятности успеха (1 –* p*).*

Параметр p геометрического распределения может быть найден через оценку для соответствующей КМ типа d. Получив p, можно оценить частоту счетчика числа уходов. Это делается только для КМ, содержащих 1 символ, при добавлении нового символа, т. е. когда КМ перестает быть детерминированной. Далее значение счетчика символа ухода изменяется только при добавлении в КМ новых символов. Величина инкремента δ счетчика равна

$$\begin{cases} 1/2, \text{ при } 4S(o) < S(o-k) \\ 1/4, \text{ при } 2S(o) < S(o-k) \\ 0, \text{ для всех прочих случаев} \end{cases},$$

где $S(o)$ – число символов в модифицируемой КМ(o); $S(o\text{-}k)$ – число символов в КМ(o-k), в которой реально был оценен текущий символ.

Если новому символу соответствует небольшая вероятность, то δ увеличивается на $1-f^0(s_i \mid o)$, где $f^0(s_i \mid o)$ есть наследуемая частота нового символа s_i (см. подразд. "Наследование информации" в подразд. "Повышение точности оценок в контекстных моделях высоких порядков").

Вероятность ухода из КМ типа m больше всего зависит от суммы значений всех счетчиков. Но эта сумма должна представляться достаточно точно, что приведет к большому количеству используемых КМУ и, как следствие, к недостатку статистики в каждой КМУ. Поэтому будем моделировать не вероятность ухода, а величину счетчика символа ухода, которая слабо зависит от суммы частот. Поля КУ формируются следующим образом:

1. Имеется сильная взаимосвязь между частотой уходов и числом незамаскированных символов; произведем квантование этого числа до 25 значений.

2. Результат сравнения числа незамаскированных символов $S(o)-S(o+1)$ в КМ(o) с числом символов $S(o+1)$ в дочерней КМ($o+1$) запишем в виде 1-битового флага.
3. Аналогично полю 2 введем флаг результата сравнения числа незамаскированных символов $S(o)-S(o+1)$ с числом символов $S(o-1)-S(o)$, которые останутся незамаскированными в родительской КМ($o-1$), если текущая КМ(o) не сможет оценить обрабатываемый символ.
4. Существует сильная статистическая взаимосвязь между текущим и предыдущим символами. Введем 1-битовый флаг, принимающий значение 0, если 2 старших бита предыдущего символа нулевые, и значение 1 в прочих случаях.
5. Существует статистическая взаимосвязь между частотой уходов и средней частотой символов $\frac{f(o)}{S(o)+1}$ в КМ, включая символ ухода, где $f(o)$ есть сумма значений всех счетчиков текущей КМ(o).

Всего может использоваться до $25\cdot2\cdot2\cdot2\cdot2=400$ контекстов ухода для КМ типа m.

Метод SEE-d2, используемый в компрессоре PPMonstr, отличается от SEE-d1 следующим образом:

1) добавлено 4 поля в КУ для КМ типа d;
2) добавлено 2 поля в КУ для КМ типа m;
3) для КМ типа nm используется адаптивная оценка.

Данная модификация позволяет улучшить сжатие на 0.5–1%, но вычисление значений дополнительных полей существенно замедляет работу компрессора.

Как SEE-d2, так и SEE-d1 обычно эффективнее SEE по методу Z с точки зрения точности оценок и вычислительной сложности.

Пример реализации адаптивной ОВУ

Заменим в нашем компрессоре Dummy априорный метод ОВУ на адаптивный. С целью максимального упрощения контекст ухода будем формировать на основании только частоты появления контекста TotFr.

Рассмотрим, как изменится программа. Для подсчета числа успешных кодирований и числа уходов создадим структуры данных:

```
struct SEE_item { //счетчики для контекста ухода
   int e, s;
};
int  TF2Index [MAX_TotFr+1]; //таблица квантования TotFr
SEE_item  *SEE;
```

Алгоритм квантования TotFr описывается следующим образом:

Значение TotFr	Номер КУ
1	1
2...3	2
4...7	3
8...15	4
...	...

Иначе говоря, по мере роста TotFr диапазон возможных значений TotFr группы КМ, относимых к одному и тому же КУ, в 2 раза больше предыдущего. Если MAX_TotFr = 0x3fff, то всего может использоваться до 14 КУ.

Изменим соответствующим образом функцию init_model.

```
void init_model (void){
   ... // ранее описанные действия по инициализации
   int  ind = 0,   //номер КУ
        i = 1,     //значение TotFr
        size = 1;  //размер диапазона
   do{
      int j = 0;
      do{
         TF2Index [i+j] = ind;
      }while (++j < size);
      i += j;
      size <<= 1;
      ind++;
   }while ((i + size) <= MAX_TotFr);
   for (; i <= MAX_TotFr; i++)
      TF2Index [i] = ind;
   /*на всякий случай отнесем КМ с TotFr = 0 к КУ
      с номером 0
   */
   TF2Index [0] = 0;
   SEE = (SEE_item*) new SEE_item[ind+1];
   //проинициализируем счетчики КМУ
   for (i = 0; i <= ind; i++) {
      SEE[i].e = 0;
      //это предотвратит деление на 0, хотя и сместит оценку
      SEE[i].s = 1;
   }
}
```

Функция encode_sym примет такой вид (изменения выделены жирным шрифтом):

```
int encode_sym (ContextModel *CM, int c){
   int  esc;
   stack [SP++] = CM;
   SEE_item  *E;
   E = calc_SEE (CM, &esc); //находим адаптивную ОВУ
   if (CM->count[c]){
      int CumFreqUnder = 0;
      for (int i = 0; i < c; i++)
         CumFreqUnder += CM->count[i];
      AC.encode (CumFreqUnder, CM->count[c],
                 CM->TotFr + esc);
      /* увеличиваем счетчик успешных кодирований для
         текущего КУ
      */
      E->s++;
      return 1;
   }else{
      if (CM->esc){
         AC.encode (CM->TotFr, esc, CM->TotFr + esc);
         //увеличиваем счетчик уходов для текущего КУ
         E->e++;
      }
      return 0;
   }
}
```

В функции декодирования символа decode_sym произведем аналогичные изменения.

```
int decode_sym (ContextModel *CM, int *c){
   stack [SP++] = CM;
   if (!CM->esc) return 0;
   int  esc;
   SEE_item  *E = calc_SEE (CM, &esc);
   int cum_freq = AC.get_freq (CM->TotFr + esc);
   if (cum_freq < CM->TotFr){
      int CumFreqUnder = 0;
      int i = 0;
      for (;;){
         if ( (CumFreqUnder + CM->count[i]) <= cum_freq)
            CumFreqUnder += CM->count[i];
         else break;
         i++;
      }
      AC.decode_update (CumFreqUnder, CM->count[i],
```

```
                              CM->TotFr + esc);
      *c = i;
      E->s++;
      return 1;
   }else{
      AC.decode_update (CM->TotFr, esc,
                              CM->TotFr + esc);
      E->e++;
      return 0;
   }
}
```

Зависимость между требуемым значением счетчика уходов, с одной стороны, и количеством уходов и успешных кодирований, с другой стороны, имеет вид

$$\frac{esc}{TotFr + esc} = \frac{e}{e + s},$$

откуда

$$esc = \frac{e}{s} \cdot TotFr.$$

Поэтому функцию `calc_SEE`, в которой собственно и осуществляется адаптивная ОВУ, реализуем так:

```
const int SEE_THRESH1 = 10;
const int SEE_THRESH2 = 0x7ff;
SEE_item* calc_SEE (ContextModel* CM, int *esc){
   SEE_item* E = &SEE[TF2Index[CM->TotFr]];
   if ((E->e + E->s) > SEE_THRESH1){
      *esc = E->e * CM->TotFr / E->s; //адаптивная оценка
      if (!(*esc)) *esc = 1;
      if ((E->e + E->s) > SEE_THRESH2){
         E->e -= E->e >> 1;
         E->s -= E->s >> 1;
      }
   }
   else *esc = CM->esc; //априорная оценка
   return E;
}
```

Константа `SEE_THRESH1` определяет порог частоты использования КУ, ниже которого предпочтительнее все же применение априорного метода оценки, так как не набралось еще значительного объема статистики для текущего КУ. Константа SEE_THRESH2 налагает ограничение на значения

счетчиков успешных кодирований s и ухода e чтобы, с одной стороны, предотвратить переполнение при умножении E->e * CM->TotFr, а с другой – улучшить адаптацию к локальным изменениям характеристик сжимаемого потока.

Предложенная реализация вычисления средней частоты уходов крайне неэкономна. Так, например, можно избежать умножения на CM->TotFr, так как обычно в пределах группы КМ, относимых к одному КУ, эта величина изменяется не сильно, поэтому имеет смысл заложить неявное умножение на соответствующую константу в сам алгоритм изменения счетчиков e и s. Практичная реализация адаптивной оценки среднего изложена в [1].

Также необходимо следить за величиной esc, поскольку достаточно большая сумма CM->TotFr + esc может привести к переполнению в арифметическом кодере. Мы не делали соответствующих проверок лишь с целью упрощения описания.

Упражнение. Добавление адаптивного метода ОВУ требует изменения действий по кодированию знака конца файла. Предложите вариант такой модификации.

Если сравнивать с помощью CalgCC, то применение описанного метода адаптивной ОВУ улучшает сжатие всего лишь приблизительно на 0.3%. Небольшой эффект объясняется не только простым механизмом вычисления ОВУ, но и малым порядком используемой модели.

Заключение. Даже сложные адаптивные методы ОВУ улучшают сжатие обычно лишь на 1–2% по сравнению с априорными методами, но требуют существенных вычислительных затрат.

Обновление счетчиков символов

Модификация счетчиков после обработки очередного символа может быть реализована по-разному. После кодирования каждого символа естественно изменять соответствующие счетчики во всех КМ порядков 0, 1, ..., *N*, что и предлагается, в частности, делать в алгоритмах PPMA и PPMB. Такой подход называется полным обновлением (full updates). Но в случае классического, не использующего наследование информации PPM лучшие результаты достигаются, когда счетчики оцененного символа увеличиваются только в КМ порядков *o*, *o*+1, ..., *N*, где *o* – порядок КМ, в которой символ был закодирован. Иначе говоря, счетчик обработанного символа не увеличивается в какой-то активной КМ, если он был оценен в КМ более высокого порядка. Эта техника имеет самостоятельное название – исключение при обновлении (update exclusion).

Термин "исключение при обновлении" не следует путать с исключением (exclusion), под которым понимают сам механизм уходов с маскированием счетчиков тех символов, которые встречались в контекстах большего порядка.

Применение исключения при обновлении позволяет улучшить сжатие обычного PPM-компрессора примерно на 1–2% по сравнению с тем случаем, когда производится обновление счетчиков символа во всех КМ. Одновременно ускоряется работа компрессора. В случае применения наследования информации, а также для алгоритма PPM* (описание PPM* приведено ниже) польза от исключения при обновлении не столь очевидна.

Роль контекстов-предков сравнительно небольших порядков значительно возрастает при использовании техники наследования информации, поэтому необходимо более быстрое обновление их статистики. Как показывают эксперименты, полное обновление работает все же плохо и в этом случае. Поэтому обычно следует использовать решение, промежуточное между исключением при обновлении и полным обновлением. Например, помимо увеличения с весом 1 в рамках реализации исключения при обновлении, имеет смысл инкрементировать с весом $1/(o-i+1)$ счетчики символа в контекстных моделях меньших порядков i. Под КМ(i) понимаются предки той КМ(o), в которой символ был оценен. Например, в компрессоре PPMd делается модификация счетчика с весом 1/2 только в родительской КМ и только в определенных случаях. При этом основное условие выполнения такой модификации требует, чтобы счетчик оцененного символа в КМ(o) был меньше некоторого порога.

В алгоритме PPM* применяется частичное исключение при обновлении (partial update exclusion). В этом случае производится увеличение счетчиков во всех так называемых детерминированных КМ, а если же их нет, то только в недетерминированной КМ с самым большим порядком. Под детерминированной понимается такая КМ, что в соответствующем ей контексте до данного момента встречался только один символ (любое число раз). Аналогично такой контекст называется детерминированным.

Для собственно сжатия в связке с PPM практически всегда используется арифметическое кодирование. Увеличение значений счетчиков КМ может привести к ошибке переполнения в арифметическом кодере. Во избежание этого обычно производят деление значений пополам при достижении заданного порога. Максимальная величина порога определяется особенностями конкретной реализации арифметического кодера. С другой стороны, масштабирование счетчиков дает побочный эффект в виде улучшения сжа-

тия при кодировании данных с достаточно быстро меняющейся статистикой контекстных моделей. Чем нестабильнее КМ, тем чаще следует масштабировать ее счетчики, отбрасывая таким образом часть информации о поведении данной КМ в прошлом. В свете этого естественной является идея об увеличении счетчиков с неравным шагом так, чтобы недавно встреченные символы получали большие веса, чем "старые" символы. В качестве полумеры можно применять масштабирование счетчика последнего встреченного символа, которое эксплуатирует такую же особенность типичных данных (см. ниже "Масштабирование счетчика последнего встреченного символа" в подразд. "Различные способы повышения точности предсказания"). Существенному улучшению сжатия в таких случаях также способствует вторичная оценка вероятности символов (см. "Увеличение точности предсказания наиболее вероятных символов" в том же подразд.).

Использование в качестве шага прироста счетчиков величин, больших единицы, необходимо для успешной работы сложных методов обновления, а также способствует лучшей адаптации модели при масштабировании. В качестве добавки веса 1 хорошо работают 4 или 8, при этом все еще отсутствует проблема переполнения даже при использовании для счетчиков 16-битовых машинных слов. Например, если шаг прироста равен четырем, то счетчик символа может принимать значения: 4 (инициализация при первом появлении символа в контексте), 8, 12, 16... В компрессоре Dummy используется единичный шаг прироста.

Повышение точности оценок в контекстных моделях высоких порядков

Наряду с задачей оценки вероятности ухода серьезной проблемой PPM является недостаточный объем статистики в КМ высоких порядков, что приводит к большим погрешностям оценок. Как побочный результат имеется неприятная зависимость порядка обычной PPM-модели, обеспечивающего наилучшее сжатие, от вида данных. Как правило, оптимальный порядок обычной модели колеблется от 0 до 16 (для текстов в районе 4–6), кроме того, часто существуют значительные локальные изменения внутри файла. Например, на рис. 4.3 приведен типичный для классического PPM-алгоритма график зависимости степени сжатия текста от порядка модели. Видно, что максимум достигается при $N = 4...5$, после чего наблюдается плавное падение степени сжатия.

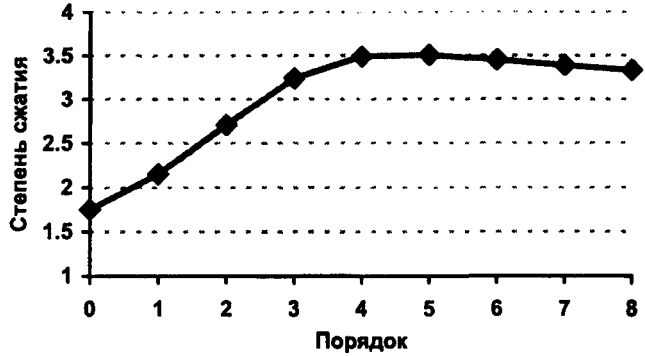


Рис. 4.3. Зависимость степени сжатия от порядка модели для классического PPM-алгоритма

Можно выделить два подхода к решению проблемы:

- выбор порядка модели, обеспечивающего наилучшее сжатие, при оценивании каждого символа;
- использование статистики контекстов-предков.

Выбор локального порядка модели

Механизм выбора порядка модели для кодирования каждого символа получил название Local Order Estimation (LOE) – оценки локального порядка. Схемы LOE носят чисто эвристический характер и заключаются в том, что мы задаем некую функцию "доверия" и пробуем предсказать с ее помощью эффективность кодирования текущего символа в той или иной доступной – соответствующей активному контексту и физически существующей – КМ порядка от 0 до заданного *N*. Можно выделить 3 типа схем LOE:

1) ищем оптимальный порядок сверху вниз от КМ максимального порядка *N* к КМ минимального порядка; прекращаем поиск, как только КМ меньшего порядка кажется нам более "подозрительной", чем текущая, которую и используем для оценки вероятности символа;
2) поиск снизу вверх;
3) оценка всех доступных КМ.

Если в выбранной КМ закодировать символ не удалось, то, вообще говоря, процедуру поиска следующей оптимальной по заданному критерию КМ можно повторить. Тем не менее обычно ищут только начальный порядок, а в случае ухода просто спускаются на уровень ниже, т. е. дальше используют обычный алгоритм PPM.

Выбор той или иной функции доверия зависит от особенностей конкретной реализации PPM, характеристик данных, для сжатия которых разрабатывается компрессор, и, как нередко бывает, личных пристрастий разработчика. Как показал опыт, различные "наивные" энтропийные оценки надежности КМ работают плохо. Эти оценки основываются на сравнении средней длины кодов для текущей КМ(o) и родительской КМ(o–1). Неудача объясняется, видимо, тем, что в силу чистой случайности функция распределения в КМ(o) может быть более плоской, чем в следующей рассматриваемой КМ(o–1). Поэтому для получения правдоподобной оценки надо сравнивать среднюю длину кодов для всех дочерних контекстных моделей текущей КМ(o) со средней длиной кодов для всех дочерних контекстных моделей родительской КМ(o–1).

В [3] был предложен эффективный и простой метод, дающий оценку надежности КМ исходя из оценки вероятности Q наиболее вероятного в КМ символа (Most Probable Symbol's Probability, MPS-P) и количества уходов E из КМ. Обобщенно формулу можно записать так:

$$a \cdot Q \log_2 Q + b \cdot E(\log_2 E - c) + d \cdot (1-Q)(\log_2 E - c), \qquad (4.1)$$

где константы a, b, c, d подбираются эмпирическим путем.

Критерием выбора КМ является минимальное значение выражения (4.1) среди всех доступных КМ.

К счастью, оценка только по Q дает хорошие результаты уже в том случае, когда просто выбираем КМ с максимальным Q (соответствует варианту обобщенной формулы при $b = d = 0$).

Можно дать следующий пример, демонстрирующий недостатки "наивного" энтропийного подхода и подхода MPS-P. Пусть мы кодируем с помощью PPM-моделирования порядка 2 последовательность алфавита {"0", "1"}, порожденную источником с такими вероятностями генерации символов:

$$p(0|00) = 0, p(1|00) = 1,$$
$$p(0|10) = p(1|10) = 0.5.$$

Пусть появление строк "00" и "10" равновероятно. Если уже обработан большой блок входной последовательности и контекст текущего символа равен "10", то в соответствии с заданным описанием источника в КМ(2) оценки вероятностей

$$q(0|10) = q(1|10) = 0.5,$$

а в КМ(1) оценки составляют

$$q(0|0) = 0.25, q(1|0) = 0.75.$$

Средняя длина кодов для КМ(2) равна

$$-q(0|10)\log_2 q(0|10) - q(1|10)\log_2 q(1|10),$$

что составляет 1 бит. В то же время средняя длина для КМ(1) равна

$$-q(0|0)\log_2 q(0|0) - q(1|0)\log_2 q(1|0),$$

что соответствует примерно 0.8 бита. Поэтому, если пользуемся "наивным" энтропийным критерием, то мы должны выбрать для кодирования КМ(1). Критерий MPS-P также указывает на КМ(1). Этот выбор ошибочен. Действительно, если мы кодируем в КМ(2), то символы "0" и "1" требуют по $-\log_2 0.5 = 1$ биту. Если в КМ(1), то для кодирования "0" нужно $-\log_2 0.25 = 2$ бита, а для "1" требуется $-\log_2 0.75 \approx 0.4$ бита. В соответствии с принятым описанием источника вероятности появления "0" и "1" после строки "10" одинаковы, поэтому при каждом оценивании на основании статистики для контекста "0" вместо статистики для контекста "10" мы будем терять в среднем $0.5\cdot(2-1) + 0.5\cdot(0.4-1) = 0.2$ бита.

Добавим, что известные варианты реализации подхода LOE плохо сочетаются в смысле улучшения сжатия с механизмом наследованием информации, так как эти техники эксплуатируют примерно одинаковые недостатки классического алгоритма PPM.

Наследование информации

Метод наследования информации позволяет существенно улучшить точность оценок. На конец 2001 г. имеется по крайней мере одна очень эффективная реализация PPM, обеспечивающая высокую степень сжатия типовых данных, в особенности текстов, в первую очередь из-за применения наследования информации [1].

Метод наследования информации, предложенный Шкариным в [1], борется с неточностью оценок символов в КМ бо́льших порядков и основан на очень простой идее. Логично предположить, что распределения частот символов в родительских и дочерних КМ похожи. Тогда при появлении в дочерней КМ(o) нового символа s_i целесообразно инициализировать его счетчик $f(s_i|o)$ некоторой величиной $f^0(s_i|o)$, зависящей от информации о данном символе в родительской КМ (или нескольких контекстных моделях-предках). Пусть в результате серии уходов мы спустились с КМ(o) на КМ($o-k$), где символ и был успешно оценен. Тогда начальное значение счетчика символа в КМ(o) разумно вычислять, исходя из равенства

$$\frac{f^0(s_i|o)}{f(o)} = \frac{f(s_i|o-k)}{f(o-k)+F_{o-k,o}}, \qquad (4.2)$$

где $f^0(s_i \mid o)$ – *наследуемая частота*; $f(o)$ – сумма частот всех символов КМ(o), включая символ ухода; $F_{o-k,o}$ – сумма частот всех символов, реально встреченных в контекстах порядков o–k+1…o.

$$F_{o-k,o} = \sum_{m=o-k+1}^{o} \left(f(m) - f(esc \mid m) - \sum_{j=1}^{S(m)} f^0(s_j \mid m) \right),$$

где $f(esc|m)$ – частота для символа ухода в КМ(m); $S(m)$ – количество символов в КМ(m), не включая символ ухода.

Упражнение. Объясните смысл слагаемого $F_{o-k,o}$ в выражении (4.2).

Выражение (4.2) обладает большой вычислительной сложностью и требует существенных затрат памяти для организации вычисления. Кроме того, имеет смысл учитывать только статистику КМ(o–k), так как все равно в большинстве случаев k равно единице, т. е. успешное кодирование происходит в родительской КМ, если же это не так, то скорее всего контексты порядков o–1…o–k+1 являются "молодыми" и для них еще не накоплено полезной статистики. Поэтому на практике целесообразно использовать приближенную формулу

$$f^0(i \mid o) = \frac{f(o) \cdot f(s_i \mid o-k)}{f(o) - s(o) + f(o-k) - f(s_i \mid o-k)}.$$

При самой простой реализации величина $f^0(s_i \mid o)$ вычисляется и присваивается сразу при создании нового счетчика в КМ(o). Но можно отложить наследование частоты до следующего появления символа s_i в этом контексте порядка o. Вероятнее всего, к тому времени родительская КМ будет обладать бóльшим объемом статистики, что должно дать более точную оценку $f^0(s_i \mid o)$ и в конечном итоге улучшить сжатие. Такое "отложенное" наследование требует повторного поиска родительской КМ и символа s_i в ней, что может существенно замедлить обработку.

В зависимости от порядка модели, особенностей реализации и собственно сжимаемых данных наследование информации позволяет улучшить сжатие примерно на 1–10%. Типичной является цифра порядка 2–4%.

Заключение. Существующие реализации компрессоров, использующих механизм наследования информации, обеспечивают лучшее сжатие, чем компрессоры с LOE.

Различные способы повышения точности предсказания

Масштабирование счетчика последнего встреченного символа

Если в последний раз в текущем контексте был встречен некий символ *s*, то вероятность того, что и текущий символ также *s*, вырастает, причем часто значительно. Этот факт позволяет улучшить предсказание за счет умножения счетчика последнего встреченного символа (ПВС) на некий коэффициент. Судя по всему, впервые такая техника описана в [15], где она называется recency scaling" – масштабированием новизны.

Техника была исследована М. А. Смирновым при разработке компрессора PPMN. По состоянию на 2001 г. неизвестны другие программы сжатия, использующие recency scaling.

В большинстве случаев хорошо работает множитель 1.1–1.2, т. е. при таком значении наблюдается улучшение сжатия большинства файлов и маловероятно ухудшение сжатия какого-то "привередливого" файла. Но часто оптимальная величина масштабирующего коэффициента колеблется в районе 2–2.5, так что можно улучшить оценку за счет адаптивного изменения множителя. Подходящее значение выбирается на основе анализа сжимаемых данных с помощью эмпирически полученных формул. Исследования показали, что величина наблюдаемой частоты совпадения последнего встреченного и текущего символов в недетерминированных контекстах позволяет подбирать коэффициент с хорошей точностью.

Для повышения точности оценки также следует учитывать расстояние между текущей позицией и моментом последней встречи текущего контекста. Если расстояние больше 5000 символов, то скорее всего масштабирование счетчика ПВС только повредит. Но следует понимать, что учет этой характеристики требует существенных затрат памяти.

Рассмотрим реализацию простейшего алгоритма recency scaling на примере нашего компрессора Dummy.

В описание структуры КМ внесем поле `last`, описывающее ПВС для заданного контекста.

```
struct ContextModel{
   int   esc,
         TotFr,
         last;
   int   count[256];
   };
```

Значение last будем вычислять как сумму номера символа и 1, для того чтобы нулевое значение last указывало на отсутствие необходимости вы-

полнять масштабирование. В начале работы программы last будет равно нулю, так как cm является глобальной переменной.

В начало функций encode_sym и decode_sym добавим действия по умножению счетчика ПВС на постоянный коэффициент 1.25.

```
int  addition;
if (CM->last){
   addition = CM->count[CM->last-1]>>2;
   CM->count[CM->last-1] += addition;
   CM->TotFr += addition;
}
```

Кроме того добавим сразу же после вызова методов AC.encode и AC.decode_update условные операторы, исправляющие значения счетчиков на первоначальные.

```
if (CM->last){
   CM->count[CM->last-1] -= addition;
   CM->TotFr -= addition;
}
```

И, наконец, обеспечим правильное обновление last в функции update_model.

```
if (!stack[SP]->count[c])
   stack[SP]->esc += 1;
else stack[SP]->last = c+1;
```

Обращаем внимание, что необходимо контролировать величину addition, так как масштабирование может привести к переполнению в арифметическом кодере. Мы не реализовали эту проверку лишь с целью упрощения описания.

Описанная модификация компрессора позволяет улучшить степень сжатия CalgCC примерно на 0.5%. Но оптимальная величина коэффициента масштабирования существенно меняется от файла к файлу. Так, например, для файла Obj2 максимальное улучшение сжатия – на 6% – было отмечено при коэффициенте 4.

Вывод. Масштабирование счетчика последнего встреченного символа (recency scaling) позволяет улучшить сжатие в среднем лишь на 0.5–1%, но может дать существенный выигрыш при обработке данных, статистические характеристики которых подвержены частым изменениям.

Масштабирование в детерминированных контекстах

Известно, что методы ОВУ A, B, C и, в меньшей степени, другие рассмотренные априорные методы назначают завышенную вероятность ухода из детерминированных контекстов [15]. Это можно исправить, умножая

счетчик единственного символа на определенный коэффициент. Такой прием был впервые описан в [15] под наименованием "deterministic scaling" - "детерминистического масштабирования". Нетрудно заметить, что этот механизм есть частный случай recency scaling.

Эффект от deterministic scaling увеличивается, если при этом используется частичное исключение при обновлении, а не обычное исключение при обновлении [15].

Deterministic scaling имеет смысл применять только в сочетании с априорными методами ОВУ, ведь чем точнее вычисляется вероятность ухода, тем пользы от этого масштабирования меньше.

Увеличение точности предсказания наиболее вероятных символов

Контексты большой длины встречаются редко, и статистика, набираемая в соответствующих КМ, обычно является недостаточной для получения надежной оценки даже в случае использования механизма наследования информации. Наиболее негативно это проявляется при предсказании наиболее вероятных символов (НбВС) и наименее вероятных символов (НмВС), так как нахождение их истинной частоты требует большого числа наблюдений. Очевидно, что при этом основную избыточность кодирования мы вносим из-за неточной оценки не НмВС, а НбВС в силу их более частой встречаемости.

В этих случаях естественным является учет информации родительской КМ, что можно рассматривать как переход от неявного взвешивания к явному.

Под НбВС будем здесь понимать символы с оценками вероятностей $p(s_i|o) \geq p(s_{mps}|o)/2$, где $p(s_{mps}|o)$ соответствует самому вероятному символу (most probable symbol) s_{mps} в текущей КМ. Частоты НбВС будем корректировать, если $f(o) < f(o-1)$. В этих случаях КМ(o) "молода" и частота $f(s_i|o)$ символа s_i еще, как правило, меньше частоты $f(s_i|o-1)$. Уточненное значение $f_{скорр}(s_i|o)$ счетчика представляет собой среднее взвешенное $f(s_i|o)$ и "приведенной" частоты $f'(s_i \mid o-1)$:

$$f_{скорр}(s_i \mid o) = \frac{f(o) \cdot f(s_i \mid o) + f(o-1) \cdot f'(s_i \mid o-1)}{f(o) + f(o-1)},$$

где $f'(s_i \mid o-1) = f(s_i \mid o-1) \cdot \dfrac{f(o) - f(s_i \mid o)}{f(o-1) - f(s_i \mid o-1)}$.

В случае использования наследуемых частот аналогичную коррекцию имеет смысл применять при наследовании.

Очевидно, что степень сжатия сильно зависит от точности предсказания НбВС, поэтому после выполнения коррекции частоты целесообразно делать адаптивную оценку вероятности символа по аналогии с SEE. Назовем такую технику Secondary Symbol Estimation (SSE) – вторичной оценкой символа. В компрессоре PPMonstr версии H вторичная оценка выполняется для КМ с замаскированными символами (КМ типа m) и недетерминированных КМ с незамаскированными символами (КМ типа nm). Поля контекстов для SSE строятся следующим образом.

Для КМ типа nm:

1) частота $f(s_{mps}|o)$ самого вероятного символа, квантуемая до 68 значений;
2) 1-битовый флаг, указывающий, что было произведено масштабирование счетчиков контекстной модели;
3) 1-битовый флаг, хранящий результат сравнения порядка o текущей КМ со средним порядком контекстных моделей, в которых были закодированы последние 128 символов;
4) 1-битовый флаг, принимающий значение 0, если 2 старших бита предыдущего обработанного символа нулевые, и значение 1 в прочих случаях (аналог поля 4 контекста ухода для КМ типа d в SEE-d1);
5) 1-битовый флаг, равный нулю, если 2 старших бита символа s_{mps} нулевые, и единице в остальных случаях (аналог поля 6 КУ для КМ типа d в SEE-d1).

Для КМ типа m:

1) оценка вероятности $p(s_{mps}|o)$, квантуемая до 40 значений;
2) 1-битовый флаг результата сравнения средней частоты замаскированных и незамаскированных символов;
3) 1-битовый флаг, указывающий, что только один символ остался незамаскированным;
4) аналог поля 2 контекста SSE для КМ типа nm (см. предыдущий список);
5) аналог поля 4 в предыдущем списке.

Техника SSE разработана Шкариным и используется для уточнения предсказания НбВС в компрессоре PPMonstr версии H.

Заключение. Применение SSE к НбВС дает выигрыш в районе 0.5–1% и особенно полезно при сжатии неоднородных данных, т. е. либо порожденных источником, быстро меняющим свои характеристики, либо состоящих из фрагментов, сгенерированных существенно отличающимися источниками.

Общий случай применения вторичной оценки символа

Естественной является идея применения SSE ко всем символам контекста. Действительно, этот прием обеспечивает существенное улучшение сжатия неоднородных данных.

Рассмотрим обобщенный алгоритм применения SSE подробнее[1]. Пусть символы КМ хранятся в виде упорядоченного списка, так что символ с наибольшей частотой записан первым, т. е. имеет ранг 1. Процедура оценки очередного символа s приобретает вид цепочки последовательных оценок альтернатив "да"/"нет": "да" – символ с текущим рангом равен s, "нет" – символы различаются, необходимо увеличить ранг на единицу и продолжить поиск. Вероятность "да" для ранга i находится как

$$Q_i(o) = q^{SSE}(s_i \mid o),$$

где $q^{SSE}(s_i \mid o)$ – оценка вероятности символа с рангом i, полученная с помощью SSE *после* оценивания символа с рангом i–1; вычисляется на основании обычной оценки $q(s_i \mid o)$ и значений $w_i(o)$ полей контекста для SSE:

$$q^{SSE}(s_i \mid o) = SSE(q(s_i \mid o), w_1(o), w_2(o), \ldots w_p(o)),$$

$$q(s_i \mid o) = \frac{f(s_i \mid o)}{f(o) - \sum_{r=1}^{i-1} f(s_r \mid o)}.$$

Например, пусть список символов текущей КМ(o) имеет вид "в", "б", "а", "г". Если обрабатывается символ "а", то он будет оценен посредством цепочки ответов "нет", "нет", "да" и получит вероятность $(1 - Q_1(o))\cdot(1 - Q_2(o))\cdot Q_3(o)$.

Дальнейшее улучшение сжатия обеспечивается соединением SSE с техникой recency scaling. В этом случае символом с рангом 1 считается последний встреченный в контексте символ (ПВС), если, конечно, он не замаскирован. Такая "адаптация новизны" позволяет значительно улучшить приспособляемость модели к потоку данных с меняющимися вероятностными характеристиками.

Положительный эффект обеспечивает внесение в контексты для SSE флагов, указывающих на совпадение ПВС текущего контекста и контекстов-предков. Аналогично присваивать ПВС ранг 1 лучше только в том случае, если он равен ПВС "ближайших" предков.

В табл. 4.7 приведено сравнение характеристик сжатия компрессора PPMonstr версии H и его модификации, использующей обобщенный алго-

[1] *Алгоритм разработан и реализован Д. Шкариным в ноябре 2001 г.*

ритм SSE в сочетании с адаптацией новизны. В сравнении были использованы файлы из наборов CalgCC и VYCCT. Все файлы, за исключением текстового Book1, содержат неоднородные бинарные данные или смесь текста и бинарных данных. Использовались модели 64-го порядка.

Таблица 4.7

Название файла	Размер сжатого файла, байт		Улучшение сжатия, %	Падение скорости сжатия, раз
	для версии H	для модификации версии H		
Book1	205144	203669	0.7	2.0
Fileware.doc	112111	100454	10.4	2.5
Obj2	65093	58466	10.2	2.1
OS2.ini	94218	83405	11.5	2.2
Samples.xls	70912	64666	8.8	2.0
Wcc386.exe	278045	255719	8.0	3.3

Вывод. Применение SSE ко всем символам контекста в сочетании с адаптацией новизны значительно улучшает сжатие неоднородных данных. Это позволяет получать стабильно наилучшую степень сжатия по сравнению с другими универсальными методами (BWT, LZ) при обработке файлов самых разнообразных типов, а не только текстов. Ценой увеличения степени сжатия является падение скорости в 2 и более раза.

PPM и PPM*

При фиксировании максимального порядка контекстов в районе 5–6 PPM даже без наследования информации дает отличные результаты на текстах, но не очень хорошо работает на высокоизбыточных данных с большим количеством длинных повторяющихся строк. В середине 90-х гг. был предложен метод борьбы с этим недостатком [6]. Предложенный алгоритм, PPM* (произносится как "пи-пи-эм ста"), был основан на использовании контекстов неограниченной длины. Авторы алгоритма предложили следующую стратегию выбора максимального порядка на каждом шаге: максимальный порядок соответствует порядку самого короткого детерминированного контекста. Под детерминированным понимается контекст, в котором до данного момента встречался только один символ (любое число раз). Если детерминированных контекстов нет, то выбирается самый длинный среди имеющихся. После выбора максимального порядка процедура оценки вероятности символа в алгоритме PPM* ничем не отличается от применяемой в алгоритмах обычного PPM, т. е. PPM-моделирования ограниченного порядка.

Реализация PPM*, описанная в [6], имела невпечатляющие характеристики: сжатие на уровне PPMC порядка пяти, скорость кодирования, как утверждается, также сопоставима, но памяти расходуется значительно больше. Судя по

всему, авторам очень хотелось доказать превосходство их схемы над другими методами PPM, и стандартным PPMC в частности. Читатель может самостоятельно сравнить степень сжатия PPM* с другими алгоритмами PPM, пользуясь табл. 4.8 и 4.9.

В принципе расходы памяти для PPM и PPM* могут быть одинаковы, что показано в [4].

Вывод. Преимущество подхода PPM* над обычным PPM не очевидно.

Достоинства и недостатки PPM

Вот уже в течение полутора десятков лет представители семейства PPM остаются наиболее мощными практическими алгоритмами с точки зрения степени сжатия. По-видимому, добиться лучших результатов смогут только более изощренные контекстные (в широком смысле) методы, которые, несомненно, будут появляться, так как производятся все более быстрые процессоры, а объем оперативной памяти ЭВМ становится все больше.

Наилучшие результаты алгоритмы PPM показывают на текстах: отличный коэффициент сжатия при высокой скорости, чему наглядным примером являются компрессоры PPMd и PPMonstr. Кроме того, если стоит задача максимизации степени сжатия определенных данных, то скорее всего PPM-подобный алгоритм будет наилучшим выбором в качестве основы специализированного компрессора.

Если выйти за рамки частной проблемы сжатия данных, то несомненным достоинством PPM является возможность получения хорошей статистической модели обработанной последовательности качественных данных (или сгенерировавшего ее источника). Действительно, модель, позволяющую эффективно предсказывать неизвестные символы сообщения, можно применять не только для сжатия, но и для решения задач коррекции текста в системах OCR, распознавания речи, классификации типа текста, семантического анализа текста, шифрования [18].

Недостатки реализаций подхода PPM заключаются в следующем:

- медленное декодирование (обычно на 5–10% медленнее кодирования);
- несовместимость кодера и декодера в случае изменения алгоритма оценки; в то же время алгоритмы семейства LZ77 допускают серьезную модификацию кодера без необходимости исправления декодера;
- медленная обработка малоизбыточных данных (скорость может падать в разы);
- наилучшее сжатие различных файлов достигается при порядках модели PPM в районе 4–12 для моделей, не применяющих технику наследования информации и/или LOE, и при порядках 16–32 в противном случае; поэтому при выборе какого-то фиксированного порядка модели мы можем

терять либо в степени сжатия, либо использовать чересчур много ресурсов ЭВМ;

- в общем случае недостаточно хорошее сжатие файлов, статистические характеристики которых подвержены частым изменениям такого типа, что оценки распределений вероятностей в контекстных моделях быстро устаревают (так называемая нестабильность статистик контекстов); с точки зрения такой адаптации обычные алгоритмы PPM уступают алгоритмам типа LZ77, хотя известны способы ослабления или вообще устранения этого неприятного эффекта при помощи вторичной оценки символа (см. "Общий случай применения вторичной оценки символа" в подразд. "Различные способы повышения точности предсказания");
- большие запросы памяти – десятки мегабайт – в случае использования сложных моделей высокого порядка в сочетании с симметричностью алгоритма препятствуют организации эффективного доступа к сжатым данным;
- заметный проигрыш в эффективности по сравнению с алгоритмами типа LZ77 при сжатии файлов, имеющих длинные повторяющиеся блоки символов.

Практически всегда можно подобрать и настроить такую PPM-модель, точнее, контекстную модель с неявным взвешиванием, что она будет давать лучшее сжатие, чем LZ или BWT. Несмотря на это, применение PPM-компрессоров целесообразно главным образом для сжатия текстов на естественных языках и подобных им данных, поскольку при обработке малоизбыточных файлов велики временные затраты. Избыточные файлы с длинными повторяющимися строками (например, тексты программ) имеет смысл сжимать с помощью BWT-компрессоров и даже словарных компрессоров, так как соотношение сжатие–скорость–память обычно лучше. Для сильно избыточных данных предпочтительнее все-таки использовать PPM, так как методы LZ и BWT, особенно не использующие предобработку, работают при этом сравнительно медленно из-за деградации структур данных.

Характеристики алгоритмов семейства PPM:

Степени сжатия: определяются данными, для текстов обычно 3–4, для объектных файлов 2–3.

Типы данных: алгоритмы универсальны, но лучше всего подходят для сжатия текстов.

Симметричность: близка к единице; обычно декодер немного медленнее кодера.

Характерные особенности: медленная обработка малоизбыточных данных.

Компрессоры и архиваторы, использующие контекстное моделирование

HA

Программа HA явилась, пожалуй, первым публично доступным архиватором, использующим контекстное моделирование. Не исключено, что HA стал бы очень популярным архиватором, если бы его автор, Гарри Хирвола (Hirvola), не прекратил работать над проектом.

В HA реализованы алгоритм семейства LZ77 и алгоритм типа PPM.

Алгоритм PPM представляет собой хорошо продуманную модификацию классического PPMC. Метод OBY является априорным и основывает оценку ухода из КМ на количестве имеющихся в ней символов с небольшой частотой. LOE не производится, последовательность спуска с КМ высоких порядков является обычной. Максимальный порядок КМ равен четырем, минимальный – минус единице. Для организации поиска КМ применяются хеш-цепочки. Хеширование осуществляется по символам контекста и его длине.

Результаты тестирования на CalgCC, представленные в табл. 4.8, получены для версии 0.999c. Сжатие файлов осуществлялось с помощью метода 2 программы, который как раз и соответствует PPM.

Архиватор разрабатывался для работы в MS DOS, и размер используемой памяти ограничен примерно 400 Кб, что, вообще говоря, мало для модели 4-го порядка, поэтому сжатие можно существенно улучшить за счет увеличения объема доступной памяти.

CM

CM Булата Зиганшина (Ziganshin) является компрессором, применяющим блочно-адаптивное контекстное моделирование.

Алгоритм работы кодера следующий. Читается блок входных данных, по умолчанию до 1 Мб, и на основании его статистики строится модель заданного порядка N. Модель сохраняется в компактном виде в выходном файле, после чего с ее использованием кодируется сам считанный блок данных. Затем, если еще не весь входной файл обработан, производятся аналогичные действия для следующего блока и т. д.

Идея построения модели заключается в следующем:

- первоначально строится модель порядка N, содержащая статистику для всех встреченных в блоке контекстов длиной от 0 до N;
- из модели удаляются контекстные модели и счетчики символов с частотой меньше порога f_{min}, являющегося параметром алгоритма.

Дерево оставшихся КМ записывается в выходной файл, при этом описания символов и их частот в определенных КМ сжимаются на основании информации КМ-предков.

Оценка вероятности ухода из КМ при кодировании самих данных зависит от количества ее дочерних КМ, удаленных при "прочистке" модели. Собственно алгоритм оценки вероятности символов не отличается от классического PPM.

Программа написана достаточно давно и не отвечает современным требованиям на соотношение скорости и степени сжатия. Тем не менее CM демонстрирует интересный подход и при соответствующей доработке практическое использование такой техники может быть целесообразно.

Характеристики степени сжатия компрессора, приведенные в табл. 4.8, были получены при запуске программы с параметрами –o4 и –m10000000, т. е. был задан порядок $N = 4$, а максимальный объем памяти для хранения модели был увеличен с 5 Мб, используемых по умолчанию, до 10 Мб, что обеспечило отсутствие переполнения при обработке всех файлов CalgCC.

RK и RKUC

С точки зрения коэффициента сжатия разрабатываемый Малькольмом Тейлором (Taylor) архиватор RK является одним из лучших среди существующих на момент написания этой книги. Но достигается это не столько за счет очень хорошего PPM-компрессора, сколько благодаря большому количеству применяемых техник предварительного преобразования данных, позволяющих значительно улучшить сжатие файлов определенных типов. Именно грамотно реализованный препроцессинг и позволяет показывать RK стабильно хорошие результаты при сжатии таких типовых данных, как объектные файлы, файлы ресурсов, документы MS Word и таблицы MS Excel, тексты на английском языке.

В RK реализовано два алгоритма: статистический типа PPM и словарный типа Зива – Лемпела. В качестве PPM-компрессора в RK применяется облегченный вариант программы RKUC, созданной также Тейлором.

С учетом сказанного мы исключили RK из таблицы сравнения контекстных компрессоров по степени сжатия (табл. 4.8).

RKUC реализует контекстное моделирование с максимальным порядком 16. Порядок КМ может быть равен 16, 12, 8, 5, ..., 0 и, вероятно, –1. Иначе говоря, в RKUC используется отличающийся от классического механизм выбора порядка следующей КМ в случае ухода. В дополнение к этому в зависимости от параметров вызова программы может выполняться LOE.

Еще одна из опций компрессора разрешает использовать при оценке вероятности статистику, накопленную для разбросанных (sparse) контекстов,

или бинарных (binary) в терминологии автора компрессора. Идея заключается в том, что несколько обычных контекстов одинаковой длины могут считаться одним контекстом, если в определенных позициях их символы одинаковы. Например, если для контекстов длины 4 требуется совпадение первого[1] (последнего обработанного), второго и четвертого символов, то строки "абсд" и "аасд" являются одним и тем же разбросанным контекстом "аХсд", где X – любой символ. Таким образом, техника разбросанных контекстов заключается в объединении информации, собираемой для нескольких классических контекстов. Применение этого механизма часто позволяет заметно улучшить сжатие блоков данных с регулярной структурой.

В RKUC применяется улучшенный вариант адаптивной ОВУ по методу Z.

При тестировании использовался RKUC версии 1.04, который запускался с параметрами -m10 -o16 -x -b, т. е. использовалась модель 16-го порядка, ограниченная 10 Мб памяти, и были включены механизмы LOE и разбросанных контекстов. Если отказаться от использования разбросанных контекстов, то степень сжатия текстовых файлов улучшается примерно на 1%, а бинарных (Geo, Obj1, Obj2) – ухудшается на несколько процентов.

PPMN

Компрессор PPMN разработан Максимом Смирновым (Smirnov) в экспериментальных целях. Основная цель разработки состояла в создании PPM-компрессора, обеспечивающего степень сжатия на уровне компрессоров, использующих разновидности алгоритма PPMZ, и значительно более высокую скорость кодирования при меньших ограничениях на объем используемой памяти.

В PPMN реализован алгоритм PPM с ограниченным порядком контекстов. Максимальный порядок N модели равен 7, но также реализованы варианты модели с "псевдопорядками" 8 и 9, при которых каждой КМ(N), N = = 8–9, в действительности может соответствовать несколько контекстов порядка N. Иначе говоря, осуществляется смешивание статистики, набираемой для нескольких похожих контекстов.

Для ОВУ используется адаптивный механизм, который можно рассматривать как упрощенный метод Z. Используется только один тип контекста ухода, поля которого определяются:

- порядком КМ;
- количеством просмотров КМ;
- количеством символов в КМ;
- последним обработанным символом;

[1] *Символы контекста обычно нумеруются справа налево, поэтому первый символ контекста соответствует последнему обработанному.*

- 1-битовым флагом, указывающим на то, что при кодировании строки последних обработанных символов заданной длины $L = 16$ ни разу не происходил уход.

Если предыдущий символ был закодирован в КМ меньшего порядка, чем у текущей рассматриваемой, или величина оценки вероятности ухода значительно меньше вычисленной для предыдущего символа, то производится увеличение оценки. Обновление счетчиков контекстной модели уходов имеет следующую специфику: если рассматривается КМ максимального порядка среди имеющихся активных, т. е. не производился уход, то шаг увеличения счетчиков КМУ имеет вес 4, иначе – 1. Как показывают эксперименты, используемая схема превосходит метод Z по точности ОВУ и при этом требует меньших вычислительных затрат.

Как и в компрессоре RKUC, последовательность выбора порядка контекстной модели в случае ухода отличается от классической – часть порядков просто не используется. Например, при $N = 5$ PPM-модель состоит из контекстных моделей 5, 3, 2, 1, 0, –1 порядков, поэтому в случае ухода из КМ(5) рассматривается КМ(3). Такой прием позволяет ускорить обработку, а уменьшение степени сжатия либо незначительно, либо, наоборот, наблюдается ее увеличение.

Как уже упоминалось ранее, в PPMN используется масштабирование счетчика последнего встреченного символа. Улучшению сжатия неоднородных файлов также способствует ограничение на количество КМ порядков 1 и 2, приводящее к интенсивному обновлению этих КМ – статистика не используемых дольше всего контекстов просто удаляется.

В PPMN применяется упрощенный способ наследования информации: при добавлении символа в КМ(o) начальное значение его счетчика определяется частотой символа в той КМ(o–k), $k > 0$, в которой он был закодирован. Механизм инкремента счетчиков соответствует обычному исключению при обновлении.

По аналогии с НА в качестве структуры данных для доступа к КМ используются хеш-цепочки. Для ускорения поиска контекстных моделей каждый символ КМ(N) имеет внешний указатель на КМ(N), соответствующую следующему символу обрабатываемого потока.

PPMN содержит механизмы предварительной обработки текстов на английском и русском языках и исполнимых файлов для процессоров типа Intel x86 (см. гл. 7). Кроме того, имеются средства кодирования таблиц 8-, 16- и 32-битовых элементов, а также кодирования длин повторов (RLE). Параметры SEE, механизмов наследования информации и обновления счетчиков настроены на обеспечение наилучшей степени сжатия в режиме обработки текстов с использованием средств препроцессинга.

В табл. 4.8 указаны результаты тестирования PPMN версии 1.00 beta 1 в режиме сжатия без использования предварительной обработки (опция –da компрессора). Порядок модели равнялся 6 (опция –o6), а ее размер был ограничен 20 Мб.

PPMd и PPMonstr

Программы PPMd и PPMonstr Дмитрия Шкарина (Shkarin) реализуют разработанные им же алгоритмы PPMII и cPPMII (complicated PPMII, т. е. "усложненный" PPMII) [1]. Оба компрессора используют механизм наследования информации. В PPMd применяется адаптивный метод ОВУ SEE-d1, а в PPMonstr – SEE-d2. Кроме этого отличия, в PPMonstr реализовано несколько дополнительных приемов улучшения сжатия, основными из которых являются отложенное наследование и улучшение точности предсказания наиболее вероятных символов.

PPMII является одним из алгоритмов, используемых в архиваторе RAR версии 3.

В тестировании были использованы версии H компрессоров. PPMd запускался с опцией -o8, а PPMonstr – с опцией -o1, что соответствует моделям 8-го и 64-го порядка. Максимальный размер модели был ограничен 20 Мб (параметр -m20) в обоих случаях, что предотвратило сброс статистики модели, осуществляемый при полном заполнении выделенного объема памяти.

PPMY

PPMY является экспериментальным компрессором, разрабатываемым Евгением Шелвиным (Shelwien). Данная программа относится к немногочисленному классу компрессоров, использующих полное смешивание.

При обработке каждого символа производится смешивание оценок распределений вероятностей из КМ всех доступных порядков от некоторого N до -1. Величина N ограничивается сверху значением одного из параметров программы. Если позволяет данное ограничение, то в качестве КМ максимального порядка выбирается КМ с самым длинным контекстом, ранее встречавшимся по крайней мере один раз. Компрессор осуществляет моделирование на уровне байтов, поэтому КМ(-1) содержит счетчики для всех 256 возможных символов; значение каждого счетчика равно единице. Кроме счетчиков встречаемых в ее контексте символов, каждая КМ(o) содержит еще две переменные:

- число символов в контексте; будем обозначать здесь как $f(esc|o)$; принятое обозначение характеристики обусловлено тем, что ее значение совпадало бы с количеством уходов из КМ(o), если бы мы имели дело с обычным алгоритмом PPM;

- число случаев, когда вероятность обрабатываемого символа в данной КМ(o) была максимальной по сравнению с прочими контекстными моделями; обозначим как $Opt(o)$.

Оценки вероятности символа, даваемые разными КМ некоторого порядка o, взвешиваются с помощью адаптивно вычисляемых весов w_o. Процедуру нахождения w_o можно описать следующим образом:

$$W_o^{(esc)} = \frac{\min(f(o), ET) - f(esc \mid o)}{f(o)},$$

$$W_o^{(opt)} = \frac{(w^{(opt)} + Opt(o))}{w^{(opt)} + f(o)},$$

$$w_o = w^{(E)} \cdot W^{(opt)} + (1 - w^{(E)}) \cdot W_o^{(esc)} \cdot W_o^{(opt)},$$

где $f(o)$ – общее количество просмотров соответствующей КМ(o); ET, $w^{(opt)}$, $w^{(E)}$ – некоторые параметры.

Коэффициентам $W_o^{(esc)}$ и $W_o^{(opt)}$ можно дать такую упрощенную интерпретацию:

- $W_o^{(esc)}$ – это вероятность того, что символ присутствует в текущей КМ(o), или, в терминах классического PPM, вероятность того, что ухода не будет;
- $W_o^{(opt)}$ – это вероятность того, что именно данная КМ(o) имеет максимальную оценку вероятности текущего символа.

Алгоритм нахождения значений параметров ET, $w^{(opt)}$, $w^{(E)}$, обеспечивающий хороший коэффициент сжатия, был получен эмпирическим путем, и результаты вычислений сведены в таблицы. Для каждой КМ(o) величины параметров выбираются из соответствующей таблицы по адресу, определяемому значением пары (o, N).

Вообще говоря, в версии PPMY 3b используется до 120 специально подобранных параметров. Возможно, использование других функций построения весов позволит уменьшить количество параметров без ущерба для степени сжатия.

В табл. 4.8 приведены результаты сжатия набора файлов CalgCC для PPMY версии 3b. Кодер запускался с опцией /o64.

Производительность на тестовом наборе Calgary Compression Corpus

Все рассмотренные компрессоры и архиваторы были протестированы на наборе CalgCC, описанном в Введении в подразд. "Сравнение алгоритмов

по степени сжатия". Для сравнения добавлены характеристики тривиального компрессора Dummy, на примере которого мы объясняли идею PPM.

В таблице указаны степени сжатия отдельных файлов набора, средняя степень сжатия, общее время кодирования $T_{код}$ и декодирования $T_{дек}$. Средняя степень вычислялась как средняя не взвешенная по размеру файлов степень сжатия. Для $T_{код}$ и $T_{дек}$ за единицу принято время сжатия всего CalgCC компрессором PPMd. Чтобы внести ясность, заметим, что единица примерно соответствует скорости кодирования 700 Кб/с для ПК с процессором типа Pentium III 733 МГц.

Таблица 4.8

	Dummy	CM	HA	PPMY	RKUC	PPMN	PPMd	PPMonstr
Bib	2.31	3.01	4.12	4.55	4.55	4.57	4.62	4.76
book1	2.22	2.99	3.27	3.72	3.62	3.64	3.65	3.74
book2	2.12	3.19	3.74	4.35	4.30	4.31	4.35	4.49
Geo	1.68	1.74	1.72	1.74	2.12	1.96	1.84	1.92
News	1.92	2.52	3.05	3.60	3.57	3.58	3.62	3.74
obj1	1.76	1.69	2.19	2.09	2.25	2.32	2.26	2.29
obj2	1.94	2.32	3.07	3.46	3.79	3.65	3.62	3.79
Paper1	2.08	2.37	3.39	3.59	3.51	3.59	3.65	3.74
Paper2	2.20	2.61	3.43	3.67	3.57	3.62	3.67	3.77
Pic	9.41	9.30	10.00	10.26	10.67	11.01	10.53	11.43
Progc	2.06	2.27	3.36	3.51	3.45	3.54	3.60	3.70
Progl	2.40	3.07	4.68	5.30	5.37	5.26	5.44	5.76
Progp	2.36	2.99	4.71	5.33	5.30	5.19	5.26	5.76
Trans	2.28	2.96	5.23	6.35	6.40	6.17	6.35	6.84
Итого	**2.62**	**3.07**	**4.00**	**4.39**	**4.46**	**4.46**	**4.46**	**4.70**
$T_{код}$	**0.5**	**2.4**	**3.0**	**14.6**	**6.5**	**1.9**	**1.0**	**2.3**
$T_{дек}$	**0.6**	**0.8**	**3.1**	**15.6**	**6.7**	**2.0**	**1.0**	**2.4**

Другие архиваторы и компрессоры

Существует множество компрессоров, применяющих контекстное моделирование, которые не были охвачены данным обзором. Отметим следующие архиваторы:

- BOA, автор Ян Саттон (Sutton);
- ARHANGEL, автор Юрий Ляпко (Lyapko);
- LGHA, являющийся реализацией архиватора HA на языке Ассемблера, выполненной Юрием Ляпко;
- UHARC, автор Уве Херклоц (Herklotz);
- X1, автор Стиг Валентини (Valentini),

а также компрессор PPMZ, автор Чарльз Блум (Bloom).

Обзор классических алгоритмов контекстного моделирования

LOEMA

Судя по всему, впервые алгоритм контекстного моделирования был реализован в 1982 г. Робертсом (Roberts) [2]. Автор назвал свой алгоритм Local Order Estimation Markov Analysis (марковский анализ посредством оценивания локального порядка). В LOEMA используется полное смешивание оценок КМ различного порядка, при этом веса представляют собой значения уровня доверия к оценке в том смысле, как это понимается в математической статистике. Сравнение степени сжатия LOEMA с другими алгоритмами затруднено, так как, с одной стороны, программа, реализующая алгоритм, не стала публично доступной и, с другой стороны, файлы, на которых Робертс доказывал эффективность своего подхода, также недоступны. Имеются сторонние отчеты, по которым LOEMA обеспечивает компрессию примерно на уровне PPMC (см. табл. 4.9) при значительно меньшей скорости, что выглядит вполне правдоподобно. Судя по всему, LOEMA Робертса позволял только оценивать, но не сжимать, т. е. выполнял исключительно статистическое моделирование.

DAFC

Алгоритм Double-Adaptive File Compression (дважды адаптивное сжатие файлов), разработанный Лэнгдоном (Langdon) и Риссаненом (Rissanen) в 1983 г., сыграл серьезную роль в развитии контекстных методов [2]. Во-первых, в нем впервые, если не считать LOEMA, была реализована идея разделения процесса кодирования на моделирование и статистическое кодирование, а также идея одновременной адаптации структуры модели (т. е. набора КМ) и частот символов. Во-вторых, простота алгоритма обеспечивала возможность его реального применения при относительно скромных возможностях вычислительной техники 80-х гг. В-третьих, DAFC полюбился научным работникам, охотно использовавшим его результаты для сравнения при написании статей.

В DAFC используются контекстная модель 0-го порядка и n контекстных моделей 1-го порядка. В начале сжатия используется КМ(0), в которой все символы алфавита обрабатываемой последовательности имеют отличные от нуля счетчики. По мере хода кодирования КМ(1) создаются только для первых n символов, встретившихся в уже обработанном блоке m раз. Из соображений экономии памяти авторы предлагали использовать $n = 31$ и $m = 50$. Далее если текущий символ встречается в контексте C и для этого контекста существует КМ(1), то производится попытка закодировать сим-

вол в ней, иначе выдается символ ухода с вероятностью, оцениваемой по методу A, и символ кодируется в КМ(0).

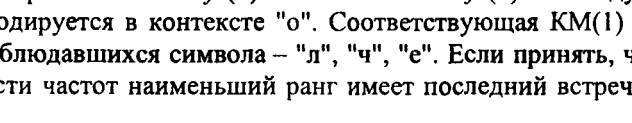
В DAFC также применяется кодирование длин повторов (RLE), которое "запускается" при встрече последовательности из трех одинаковых символов.

Упражнения: Сравните DAFC с алгоритмом работы компрессора Dummy. Если пренебречь RLE, то какие изменения следует внести в Dummy, чтобы получить реализацию DAFC?

Пользуясь приведенными в тексте таблицами, сравните DAFC и Dummy по степени сжатия файлов набора CalgCC (см. табл. 4.8 и 4.9).

ADSM

Алгоритм Adaptive Dependency Source Model (модель источника с адаптирующейся зависимостью) Абрахамсона (Abrahamson) представляет собой образец интересного подхода к реализации идеи контекстного моделирования [2]. Здесь осуществляется чистое контекстное моделирование 1-го порядка, но собственно оценка строится на основании только одного распределения частот, общего для всех КМ. Достигается это следующим образом. В каждой КМ(1) счетчики символов хранятся в виде упорядоченного по величине частот списка. Счетчики ранжируются так, что символ с наибольшей частотой имеет наименьший ранг 1, а с наименьшей частотой – наибольший ранг. При обработке текущего символа находится его ранг (это может быть просто номер символа в упорядоченном списке символов текущей контекстной модели) и оценка определяется частотой использования этого ранга. Частоты рангов изменяются после кодирования каждого символа. Таким образом, статистическое кодирование осуществляется не на базе частоты *символа*, а на основании количества появлений символов с соответствующим *рангом* частоты. Рассмотрим сжатие строки "молочное_молоко" начиная с отмеченного на рисунке стрелкой символа.



В контексте "м" реально встречался только один символ "о", соответствующий текущему, поэтому мы кодируем "о", опираясь на частоту *fr*(1) использования ранга 1. Затем *fr*(1) обновляется как *fr*(1)++. Следующий символ, "л", кодируется в контексте "о". Соответствующая КМ(1) содержит 3 реально наблюдавшихся символа – "л", "ч", "е". Если принять, что в случае одинаковости частот наименьший ранг имеет последний встреченный сим-

вол, то "л" кодируется на базе частоты *fr*(3) применения ранга 3. Производится обновление *fr*(3)++ и переход к обработке следующей буквы, "о".

Ни разу не встреченные в соответствующей КМ(1) символы, т. е. имеющие нулевую частоту, не трактуются как особый случай, им также присваивается какой-то ранг.

Были исследованы расширения алгоритма ADSM для 2-го и 3-го порядков [9]. Результаты экспериментов показывают, что, в отличие от многих других классических алгоритмов контекстного моделирования, ADSM актуален и по сей день, так как обеспечивает хорошее соотношение скорости работы, объема используемой памяти и коэффициента сжатия. Так, например, техника ADSM использована в программе сжатия изображений без потерь BMF, являющейся одной из лучших в своем классе.

Упражнение. Сожмите строку с самого начала, найдите действительные значения *fr*(1) и *fr*(3), используемые при кодировании "о" и "л" в примере.

DHPC

Техника Dynamic-History Predictive Compression (сжатие на основе предсказания по динамической истории), предложенная Уильямсом (Williams), интересна в сравнении с DAFC как использующая иной способ ограничения роста модели. В этом алгоритме происходит создание КМ(*o*), только если родительская КМ(*o*+1) достаточно часто использовалась. Если представить контекст дочерней КМ в виде сцепления (конкатенации) *aC* какого-то символа *a* и контекста *C* родительской, в случае классического DAFC являющегося пустой строкой, то можно дать такую сравнительную характеристику. В DAFC образование дочерней КМ возможно в случае превышения частотой появления образующего контекст символа "а" заданного порога, а в DHPC – при превышении порога частотой появления родительского контекста *C*, т. е. только "заслуженные" КМ могут иметь "детей".

При исчерпании доступной памяти рост модели DHPC прекращается, далее возможна адаптация только за счет изменения счетчиков символов.

Благодаря использованию КМ бóльших порядков DHPC дает лучшее сжатие, чем DAFC, но уступает по этому показателю классическим представителям семейства PPM (PPMC и PPMD).

Алгоритмы PPMA и PPMB

Алгоритмы PPMA и PPMB являются самыми ранними среди представителей семейства PPM [5]. Они были предложены авторами техники PPM одновременно в одной и той же статье и могут рассматриваться как единственные "чистокровные" PPM.

В PPMA и PPMB применяется ОВУ по методам A и B соответственно. После кодирования символа производится полное обновление счетчиков. Значения счетчиков символов не масштабируются, что требует достаточно больших счетчиков (авторы алгоритмов использовали 32-битовые).

WORD

Алгоритм WORD был создан Моффатом (Moffat) в конце 80-х гг. специально для сжатия текстов и является ярким представителем особого семейства контекстных методов.

В алгоритме используются КМ не только для символов, но и для последовательностей (строк) конечной длины. Весь алфавит сжимаемого блока делится на "буквы" и "не-буквы". Последовательность букв называется "словом", а не-букв – "не-словом". Для оценивания применяются КМ 1-го и 0-го порядка, при этом буква предсказывается буквой, а слово – словом, аналогично для не-букв и не-слов. Если обрабатываемое слово ни разу не встречалось в КМ(1) для слов, то производится уход на уровень КМ(0); если же и там оценивание невозможно, т. е. такая строка встретилась впервые, то слово передается как последовательность букв. Для этого сначала кодируется длина слова, а затем составляющие его буквы с использованием КМ первого, нулевого и минус первого порядков. При оценке вероятностей букв используется техника уходов. Обработка не-слов и не-букв осуществляется аналогично. Таким образом, в WORD используется всего 12 типов КМ: 1-го и 0-го порядка для слов (не-слов), 1-го, 0-го и -1-го порядка для букв (не-букв), 0-го порядка для длин слов (не-слов).

Длина слова (не-слова) ограничивается 20 символами. Как и в случае DHPC, при достижении моделью заданного размера удаление всей структуры или ее части не производится, просто прекращается рост.

Сравнение алгоритмов контекстного моделирования

В табл. 4.9 представлены сведения о степени сжатия файлов набора CalgCC компрессорами, реализующими соответствующие алгоритмы контекстного моделирования. В первой строке указано название алгоритма, во второй, по необходимости, порядок использованной модели – строка "o-*N*" указывает, что использовалась модель порядка *N*. В строке "Итого" указана средняя не взвешенная по размеру файлов степень сжатия всего CalgCC.

Алгоритм cPPMII реализует механизм наследования информации и использует SEE-d2. Описание прочих алгоритмов было дано выше.

Таблица 4.9

	ADSM	DAFC	WORD	PPMC o-3	PPMC o-5	PPM*	PPMD o-5	cPPMII o-64
Bib	2.07	2.08	3.65	3.79	4.17	4.19	4.26	4.76
Book1	2.11	2.17	2.96	3.23	3.42	3.33	3.48	3.74
Book2	2.03	2.04	3.19	3.54	4.00	3.96	4.06	4.49
Geo	1.46	1.72	1.58	1.67	1.69	1.66	1.70	1.92
News	1.84	1.84	2.60	3.02	3.33	3.31	3.39	3.74
Obj1	1.60	1.55	1.78	2.13	2.14	2.00	2.14	2.29
Obj2	1.81	1.39	1.84	2.97	3.27	3.29	3.31	3.79
Paper1	1.96	1.90	3.10	3.23	3.38	3.38	3.42	3.74
Paper2	2.08	2.08	3.35	3.27	3.39	3.39	3.46	3.77
Pic	7.77	8.89	8.99	7.34	9.76	9.41	9.88	11.43
Progc	1.90	1.81	2.95	3.21	3.32	3.33	3.36	3.70
Progl	2.18	2.22	4.21	4.21	4.62	4.79	4.73	5.76
Progp	2.14	2.08	4.17	4.35	4.57	4.94	4.65	5.76
Trans	2.06	1.95	4.19	4.52	5.19	5.52	5.33	6.84
Итого	**2.36**	**2.41**	**3.47**	**3.61**	**4.02**	**4.04**	**4.08**	**4.70**

Таким образом, изощренные модели большого порядка обеспечивают лучшее сжатие данных, но разница в производительности схем обычно составляет лишь десятки, а то и единицы процентов. Поэтому обоснованный выбор алгоритма моделирования следует делать на базе комплексной оценки, включающей также объем используемой памяти, скорости кодирования и декодирования и, конечно же, вычисляемой именно для тех данных, которые требуется сжимать.

Другие методы контекстного моделирования

Среди нерассмотренных остался интересный метод универсального сжатия Context Tree Weighting (взвешивание контекстного дерева), или CTW, который потенциально обеспечивает лучшую степень сжатия среди всех известных алгоритмов [16]. В CTW при оценке вероятности символа используется явное взвешивание.

Контекстное моделирование ограниченного порядка хорошо работает на практике, обеспечивая высокую степень сжатия при терпимых требованиях к вычислительным ресурсам. Но оно представляет собой всего лишь один из типов контекстного моделирования в широком смысле. Можно отметить другие методы:

- модели состояний; в качестве конкретного алгоритма можно указать динамическое марковское сжатие (Dynamic Markov Compression, или DMC) [4, 7];
- грамматические модели; конкретный алгоритм – SEQUITUR [11];
- модели с использованием искусственных нейронных сетей для построения предсказателя [14].

Рассмотрение алгоритмов моделирования данных видов выходит за рамки этой книги.

Вопросы для самоконтроля[1]

1. Объясните связь между точностью предсказания значений данных и степенью сжатия.
2. Что собой представляет модель источника данных в случае использования для моделирования PPM-алгоритма?
3. Почему технику уходов можно охарактеризовать как способ неявного взвешивания статистики контекстных моделей?
4. В каких случаях оценка вероятности ухода может равняться нулю?
5. Приведите пример блока данных, которые выгоднее сжимать, предсказывая вероятность символов на базе их безусловных частот, а не с помощью PPM-моделирования порядка 1.
6. Почему применение метода исключения всегда улучшает степень сжатия?
7. Как вы думаете, целесообразно ли применение метода наследования информации для обновления статистики контекстных моделей ухода?
8. Почему в случае использования наследования информации применение метода исключения при обновлении обычно не позволяет достигать потенциально возможной степени сжатия?
9. Почему метод наследования информации является потенциально более мощным способом компенсации недостатка статистики в контекстных моделях высоких порядков, чем метод выбора локального порядка?
10. В чем идея способов улучшения точности предсказания при обработке неоднородных данных?
11. Почему чем избыточнее со статистической точки зрения данные, тем скорость их обработки PPM-алгоритмами выше?
12. Укажите задачи, при решении которых может использоваться PPM-моделирование.

[1] *Ответы на вопросы, выполненные упражнения и исходные тексты программ вы можете найти на http://compression.graphicon.ru/.*

Литература

1. Шкарин Д. Повышение эффективности алгоритма PPM // Проблемы передачи информации. 2001. Т. 34(3). С. 44–54.
2. Bell T. C., Witten I. H., Cleary, J. G. Modeling for text compression // ACM Computer Surveys. 1989. Vol. 24(4). P. 555–591.
3. Bloom C. Solving the problems of context modeling // California Institute of Technology. 1996.
4. Bunton S. On-Line Stochastic Processes in Data Compression // PhD thesis, University of Washington. 1996.
5. Cleary J. G., Witten I. H. Data compression using adaptive coding and partial string matching // IEEE Transactions on Communications April 1984. Vol. 32(4). P. 396–402.
6. Cleary J. G., Teahan W. J. Unbounded length contexts for PPM // The Computer Journal. 1997. Vol. 40 (2/3). P. 67– 75.
7. Cormack G. V., Horspool R. N. Data compression using dynamic Markov modelling // The Computer Journal. Dec. 1987. Vol. 30(6). P. 541–550.
8. Howard P. G. The design and analysis of efficient lossless data compression systems // PhD thesis, Brown University, Providence, Rhode Island. 1993.
9. Lelewer D. A., Hirschberg D. S. Streamlining Context Models for Data Compression. // Proceedings of Data Compression Conference, Snowbird, Utah. 1991. P. 313–322.
10. Moffat A. Implementing the PPM data compression scheme // IEEE Transactions on Communications. Nov. 1990. Vol. 38(11). P. 1917–1921.
11. Nevill-Manning C. G., Witten I. H. Linear-time, incremental hierarchy inference for compression // Proceedings of Data Compression Conference /By ed. J. A. Storer and M. Cohn. Los Alamitos, CA: IEEE Press. 1997. P. 3–11.
12. Rissanen J. J., Langdon G. G. Universal modeling and coding // IEEE Transactions on Information Theory. Jan. 1981. Vol. 27(1). P. 12–23.
13. Shannon C. E. A mathematical theory of communication // The Bell System Technical Journal. July, October, 1948. Vol. 27. P. 379–423, 623–656.
14. Schmidhuber J. Sequential neural text compression // IEEE Transactions on Neural Networks. 1996. Vol. 7(1) P. 142–146.
15. Teahan W. J. Probability estimation for PPM // Proceedings of the New Zealand Computer Science Research Students Conference, 1995. University of Waikato, Hamilton, New Zealand.
16. Willems F. M. J., Shtarkov Y. M., Tjalkens T.J. The context-tree weighting method: Basic properties // IEEE Transactions on Information Theory. May 1995. Vol. 41(3). P. 653–664.

17. Witten I. H., Bell T. C. The zero-frequency problem: estimating th probabilities of novel events in adaptive text compression // IEEE Transactions on Information Theory. July 1991. Vol. 37(4). P. 1085–1094.
18. Witten I. H., Bray Z., Mahoui M., Teahan B. Text mining: a new frontier fo lossless compression // University of Waikato, NZ. 1999.
19. Ryabko B. Ya. Twice Universal Coding// Problems of Information Transmission. Vol. 20(3). 1984. P. 173–177.

Список архиваторов и компрессоров

1. Bloom C. (1996) PPMZ – PPM Based Compressor – Win95/NT version. ftp://ftp.elf.stuba.sk/pub/pc/pack/ppmz_ntx.zip
2. Herklotz U. (2001) UHARC – high compression multimedia archiver. ftp://ftp.elf.stuba.sk/pub/pc/pack/uharc04.zip
3. Hirvola H. (1995) HA – file archiver utility. ftp://ftp.elf.stuba.sk/pub/pc/pack/ha0999.zip
4. Lyapko G. (2000) ARHANGEL – file archiving utility. ftp://ftp.elf.stuba.sk/pub/pc/pack/arh140.zip
5. Lyapko G. (1997) LGHA – archive processor. ftp://ftp.elf.stuba.sk/pub/pc/pack/lgha11g.zip
6. Shelwien E. (2001) PPMY – context modelling compressor. http://www.pilabs.org.ua/sh/ppmy_3b.zip
7. Shkarin D. (1999) BMF – lossless image compressor. ftp://ftp.elf.stuba.sk/pub/pc/pack/bmf_1_10.zip
8. Shkarin D. (2001) PPMDH – fast PPM compressor for textual data. ftp://ftp.elf.stuba.sk/pub/pc/pack/ppmdh.rar
9. Smirnov M. (2001) PPMN – not so fast PPM compressor. http://msmirn.newmail.ru/ppmnb1.zip
10. Sutton I. (1998) BOA constrictor archiver. ftp://ftp.elf.stuba.sk/pub/pc/pack/boa058.zip
11. Taylor M. (2000) RK – file archiver. ftp://ftp.elf.stuba.sk/pub/pc/pack/rk104a1d.exe
12. Taylor M. (1999) RKUC – universal compressor. ftp://ftp.elf.stuba.sk/pub/pc/pack/rkuc104.zip
13. Valentini S. (1996) X1 archiver. ftp://ftp.elf.stuba.sk/pub/pc/pack/x1dos95a.zip
14. Ziganshin B. (1997). CM – static context modeling archiver. http://sochi.net.ru/~maxime/src/cm.cpp.gz

Глава 5. Преобразование Барроуза – Уилера

Введение

Преобразование Барроуза – Уилера применяется в алгоритмах сжатия качественных данных. Для эффективного использования преобразования необходимо, чтобы характеристики данных соответствовали модели источника с памятью.

Как и многие другие применяемые в алгоритмах сжатия преобразования, преобразование Барроуза – Уилера предназначено для того, чтобы сделать сжатие данных входного блока более эффективным. Посредством перестановки элементов данное преобразование превращает входной блок данных со сложными зависимостями в блок, структуру которого моделировать гораздо легче, причем отображение происходит без потерь информации.

Этот метод появился сравнительно недавно. Впервые он был опубликован 10 мая 1994 г. в [13]. Авторами метода являются Дэвид Уилер и Майк Барроуз. Причем первый из них придумал этот метод еще в 1983 г., когда работал в компании AT&T Bell Laboratories. Но, видимо, либо тогда не придал значения своему открытию, либо посчитал чрезмерными требования к вычислительным ресурсам.

По имени авторов, алгоритм был назван преобразованием Барроуза – Уилера (Burrows – Wheeler Transform, далее – BWT). Метод был объявлен компромиссным между быстрыми словарными алгоритмами, с одной стороны, и статистическими алгоритмами, дающими более сильное сжатие, но малоприменимыми в то время на практике, с другой стороны.

Благодаря таким свойствам описываемое преобразование стало довольно популярным и среди разработчиков архиваторов, и среди научных работников. Уже опубликовано более сотни статей на разных языках, посвященных этому методу, и написано столько же программ, его реализующих.

Преобразование Барроуза – Уилера

Как уже указывалось, преобразование Барроуза – Уилера предназначено для того, чтобы преобразовать входной блок в более удобный для сжатия вид. Причем, как показывает практика, полученный в результате преобразования блок обычные методы сжимают не так эффективно, как методы, специально для этого разработанные.

Поэтому нельзя рассматривать описываемый алгоритм отдельно от соответствующих специфических методов кодирования данных.

В оригинальной статье была предложена как одна из возможных реализаций сжатия на основе BWT совокупность из трех алгоритмов:

- преобразования Барроуза – Уилера;
- преобразования Move-To-Front (известного в русскоязычной литературе как перемещение стопки книг);
- статистического кодера для сжатия данных, полученных в результате последовательного применения двух вышеупомянутых преобразований.

Дальнейшие исследования показали, что второе из перечисленных преобразований не является необходимым и может быть заменено альтернативным. Или даже исключено вовсе за счет усложнения кодирующей фазы. Впрочем, еще сами авторы BWT упоминали о такой возможности.

Итак, начнем с описания главной составляющей части процесса сжатия данных при помощи методов на основе BWT – с самого преобразования.

Прежде всего следует отметить одну из его особенностей. Он оперирует сразу целым блоком данных. То есть ему заранее известны сразу все элементы входного потока или по крайней мере достаточно большого блока. Это делает затруднительным использование алгоритма в тех областях применения, где требуется сжатие данных "на лету", символ за символом. В этом отношении BWT даже более требователен, чем методы семейства LZ77, использующие для сжатия скользящее окно.

Следует отметить, что возможна реализация сжатия данных на основе BWT, обрабатывающая данные последовательно по символам, а не по блокам. Но скоростные характеристики программ, использующих такую реализацию, будут очень далеки от совершенства.

Таким образом, мы пришли к первой и самой легкой из фаз преобразования – к выделению из непрерывного потока блока данных.

Де-факто описывать BWT стало принято с помощью примера преобразования строки символов "абракадабра".

Далее нужно из полученного блока данных создать матрицу всех возможных его циклических перестановок. Первой строкой матрицы будет исходная последовательность, второй строкой – она же, сдвинутая на один символ влево, и т. д. Таким образом, получим матрицу, изображенную на рис. 5.1.

```
абракадабра
бракадабраа
ракадабрааб
акадабраабр
кадабраабра
адабраабрак
дабраабрака
абраабракад
браабракада
раабракадаб
аабракадабр
```

Рис. 5.1. Множество циклических перестановок строки "абракадабра"

Пометим в этой матрице исходную строку и отсортируем все строки в соответствии с лексикографическим порядком символов. Будем считать, что одна строка должна находиться в матрице выше другой в том случае, если в самой левой из позиций, начиная с которой строки отличаются, в этой строке находится символ лексикографически меньший, чем у другой строки. Другими словами, следует отсортировать символы сначала по первому символу, затем строки, у которых первые символы равны, – по второму и т. д. (рис. 5.2).

```
 0  аабракадабр
 1  абраабракад
 2  абракадабра – исходная строка
 3  адабраабрак
 4  акадабраабр
 5  браабракада
 6  бракадабраа
 7  дабраабрака
 8  кадабраабра
 9  раабракадаб
10  ракадабрааб
```

Рис. 5.2. Матрица циклических перестановок строки "абракадабра", отсортированная слева направо в соответствии с лексикографическим порядком символов ее строк

Теперь остался последний шаг – выписать символы последнего столбца и запомнить номер исходной строки среди отсортированных. Итак, "рдакраааабб",2 – это результат, полученный в результате преобразования Барроуза – Уилера.

Теперь нам осталось сделать "всего" 3 вещи:

1) доказать, что преобразование обратимо;
2) показать, что оно не требует огромного количества ресурсов;
3) показать, что оно полезно для последующего сжатия.

Упражнение. Проделайте преобразование Барроуза – Уилера строки "карабас".

Доказательство обратимости преобразования Барроуза – Уилера

Возможно, Дэвид Уилер не опубликовал описание алгоритма в 1983 г. потому, что ему самому показалось странным, что можно восстановить начальную строку из столь сильно перемешанных между собой символов. Но так или иначе, это не фокус и обратное преобразование действительно возможно.

Пусть нам известны только результат преобразования, т. е. последний столбец матрицы, и номер исходной строки. Рассмотрим процесс восстановления исходной матрицы. Для этого отсортируем все символы последнего столбца (рис. 5.3).

Нам известно, что строки матрицы были отсортированы по порядку, начиная с первого символа. Поэтому, очевидно, в результате такой сортировки мы получили первый столбец исходной матрицы.

Поскольку последний столбец по условию задачи нам известен, добавим его в полученную матрицу (рис. 5.4).

0	а
1	а
2	а
3	а
4	а
5	б
6	б
7	д
8	к
9	р
10	р

Рис. 5.3. Отсортированные символы исходной строки

0	а.....р
1	а.....д
2	а.....а
3	а.....к
4	а.....р
5	б.....а
6	б.....а
7	д.....а
8	к.....а
9	р.....б
10	р.....б

Рис. 5.4. Первый и последний столбцы матрицы циклических перестановок

Теперь самое время вспомнить, что строки матрицы были получены в результате циклического сдвига исходной строки. То есть, символы последнего и первого столбцов образуют друг с другом пары. И нам ничто не может помешать отсортировать эти пары, поскольку обязательно существуют такие строки в матрице, которые начинаются с этих пар. Заодно допишем в матрицу и известный нам последний столбец (рис. 5.5).

0	аа.....р
1	аб.....д
2	аб.....а
3	ад.....к
4	ак.....р
5	бр.....а
6	бр.....а
7	да.....а
8	ка.....а
9	ра.....б
10	ра.....б

Рис. 5.5. Первый, второй и последний столбцы матрицы

Таким образом, два столбца нам уже известны. Легко заметить, что отсортированные пары вместе с символами последнего столбца составляют тройки. Аналогично восстанавливается вся матрица. А на основании записанного заранее номера исходной строки в матрице – и сама исходная строка (рис. 5.6).

0	ааб.....р	аабр.....р		аабракада.р	аабракадабр
1	абр.....д	абра.....д		абраабрак.д	абраабракад
2	абр.....а	абра.....а		абракадаб.а	абракадабра
3	ада.....к	адаб.....к		адабраабр.к	адабраабрак
4	ака.....р	акад.....р		акадабраа.р	акадабраабр
5	бра.....а	браа.....а	...	браабрака.а	браабракада
6	бра.....а	брак.....а		бракадабр.а	бракадабраа
7	даб.....а	дабр.....а		дабраабра.а	дабраабрака
8	кад.....а	када.....а		кадабрааб.а	кадабраабра
9	раа.....б	рааб.....б		раабракад.б	раабракадаб
10	рак.....б	рака.....б		ракадабра.б	ракадабрааб

Рис. 5.6. Процесс определения всех столбцов матрицы

Упражнение. В результате преобразования Барроуза – Уилера получена строка "тпрооппо". Номер исходной строки в матрице преобразований – 5 (считая с нуля). Восстановите исходную строку.

Вектор обратного преобразования

После того как мы доказали принципиальную возможность обратного преобразования, можно показать, что для его осуществления нет необходимости посимвольно выписывать все строки матрицы перестановок. Если бы мы хранили в памяти всю матрицу, требуемую для мегабайтового блока данных, нам потребовалось бы... В общем, видно, что затея была бы бесполезной.

Для обратного преобразования нам дополнительно к собственно данным нужен только вектор обратного преобразования, представляющий собой массив чисел, размер которого равен числу символов в блоке. Поэтому за-

траты памяти при выполнении обратного преобразования линейно зависят от размера блока.

Обратите внимание, что в процессе выявления очередного столбца матрицы мы совершали одни и те же действия. А именно получали новую строку, сливая символ из последнего столбца старой строки с известными символами первых столбцов этой же строки. И новая строка после сортировки перемещалась в другую позицию в матрице.

Из строки 0 мы получали строку 9, из первой – седьмую и т. п., (рис. 5.7).

Номер строки		Номер новой строки	
0	а.....р	9	
1	а.....д	7	
2	а.....а	0	– исходная строка
3	а.....к	8	
4	а.....р	10	
5	б.....а	1	
6	б.....а	2	
7	д.....а	3	
8	к.....а	4	
9	р.....б	5	
10	р.....б	6	

Рис. 5.7. Способ получения первого столбца матрицы из последнего

Важно, что при добавлении любого столбца перемещения строк на новую позицию были одинаковы. Нулевая строка перемещалась в девятую позицию, первая – в седьмую и т. д.

Чтобы получить вектор обратного преобразования, следует определить порядок получения символов первого столбца из символов последнего. То есть, отсортировать матрицу по номерам новых строк (рис. 5.8).

Номер строки		Номер новой строки	Переносим последний столбец в начало
2	а.....а	0	аа.....р
5	б.....а	1	аб.....д
6	б.....а	2	аб.....а
7	д.....а	3	ад.....к
8	к.....а	4	ак.....р
9	р.....б	5	бр.....а
10	р.....б	6	бр.....а
1	а.....д	7	да.....а
3	а.....к	8	ка.....а
0	а.....р	9	ра.....б
4	а.....р	10	ра.....б

Рис. 5.8. Получение вектора обратного преобразования

Полученные значения номеров строк T = { 2, 5, 6, 7, 8, 9, 10, 1, 3, 0, 4 } и есть искомый вектор, содержащий номера позиций символов в строке, которую нам надо декодировать (символы упорядочены в соответствии с положением в алфавите).

Теперь получить исходную строку совсем просто. Первым делом возьмем элемент вектора обратного преобразования, соответствующий номеру исходной строки в матрице циклических перестановок, T[2] = 6. Иначе говоря, в качестве первого символа в исходной строке следует взять шестой символ из строки "рдакрааааббб". Это символ "а".

Затем нужно определить, какой символ заставил опуститься найденный символ "а" на вторую позицию среди равных. Искомый символ находится в последнем столбце шестой строки матрицы, изображенной на рис. 5.8. А поскольку T[6] = 10, в преобразованной строке он находится в десятой позиции. Это символ "б". И т. д. (рис. 5.9).

6⇒	10⇒	4⇒	8⇒	3⇒	7⇒	1⇒	5⇒	9⇒	0⇒	2
а	б	р	а	к	а	д	а	б	р	а

Рис. 5.9. Декодирование исходной строки при помощи вектора обратного преобразования

Упражнение. В результате преобразования Барроуза – Уилера получена строка "тпрооппо". Номер исходной строки в матрице преобразований – 5 (считая с нуля). Постройте вектор обратного преобразования.

Реализация обратного преобразования

Получение исходной строки из преобразованной можно проиллюстрировать при помощи небольшой программы.

Введем следующие обозначения:

n – количество символов в блоке входного потока;
N – количество символов в алфавите;
pos – номер исходной строки в матрице перестановок;
in – входной блок;
count – массив частот каждого символа алфавита во входном блоке;
T – вектор обратного преобразования, размер вектора равен *n*.

Первым делом следует посчитать частоты символов и пронумеровать все исходные символы в порядке их появления в алфавите. По сути, это эквивалентно построению первого столбца матрицы циклических перестановок.

```
for( i=0; i<N; i++) count[i]=0;
for( i=0; i<n; i++) count[in[i]]++;
sum = 0;
for( i=0; i<n; i++) {
   sum += count[i];
   count[i] = sum - count[i];
}
```

После выполнения этих действий count[*i*] указывает на первую позицию символа с кодом *i* в первом столбце матрицы. Следующий шаг – создание вектора обратного преобразования.

```
for( i=0; i<n; i++) T[count[in[i]]++]]=i;
```

Далее при помощи полученного вектора восстановим исходный текст.

```
for( i=0; i<n; i++) {
   putc( in[pos], output );
   pos = T[pos];
}
```

Использование BWT в сжатии данных

Теперь, после того как выяснилось, что наши действия вполне обратимы и данные мы не исказим, можно вернуться к вопросу рассмотрения полезности преобразования. Как уже отмечалось выше, главная задача преобразования Барроуза – Уилера заключается в том, чтобы ловко переставить символы. Переставить так, чтобы их можно было легко сжать, не ломая голову над их взаимосвязями. Потому что преобразование как раз тем и занимается: "вытаскивает" все взаимосвязи наружу. Точнее, очень многие.

Для понимания этого процесса достаточно представить поток данных состоящим из набора некоторых стабильных сочетаний символов. Например, как этот текст, состоящим из слов. Сочетание символов, позволяющее предсказать некоторый неизвестный доселе символ, называется контекстом. Например, если нам известна последовательность символов "реобразование", то скорее всего ей предшествует символ "п". Назовем *устойчивым* (*стабильным*) такой контекст, для которого распределение частот символов, непосредственно примыкающих к нему слева или справа, меняется незначительно в пределах блока.

Если нам потребуется подвергнуть преобразованию данную главу, можно с уверенностью сказать, что строки, начинающиеся с символов "реобразование", будут располагаться рядом в отсортированной матрице. И в соответствующих этим строкам позициях последнего столбца матрицы будет находиться символ "п".

Таким образом, главное свойство преобразования в том, что оно собирает вместе символы, соответствующие похожим контекстам. Чем больше стабильных контекстов в блоке данных, тем лучше будет сжиматься полученный в результате преобразования блок. Практика показывает, что в результате преобразования обычных текстов более половины из всех символов следует за такими же.

Упражнение. Какие еще символы помимо "п" могут оказаться в конце строк, начинающихся с последовательности символов "реобразование", в результате преобразования данной главы?

Частичное сортирующее преобразование

Некоторое время спустя после появления первых архиваторов, использующих преобразование Барроуза – Уилера, было опубликовано описание еще одного алгоритма, также основанного на сортировке блока данных [33, 1]. Отличие от BWT заключается в том, что сортировка строк матрицы осуществляется не по всей длине строк, а только по некоторому фиксированному количеству символов. В том случае, если у нескольких строк эти символы одинаковы, выше в списке помещается строка, первый символ которой встретился во входном блоке раньше первых символов остальных рассматриваемых строк.

Можно сказать, что позиция первого символа строки во входном блоке участвует в сортировке. Число символов строки, участвующих в преобразовании, называется порядком сортирующего преобразования. Легко заметить, что в результате преобразования нулевого порядка, ST(0), получается исходная строка. В качестве примера выполним преобразования 1-го и 2-го порядка строки "абракадабра" (рис. 5.10).

№ строки	ST(1)	ST(2)	BWT
	Первый шаг. Построение матрицы преобразования		
0	а< 0>бракадабра	аб< 0>ракадабра	абракадабра
1	б< 1>ракадабраа	бр< 1>акадабраа	бракадабраа
2	р< 2>акадабрааб	ра< 2>кадабрааб	ракадабрааб
3	а< 3>кадабраабр	ак< 3>адабраабр	акадабраабр
4	к< 4>адабраабра	ка< 4>дабраабра	кадабраабра
5	а< 5>дабраабрак	ад< 5>абраабрак	адабраабрак
6	д< 6>абраабрака	да< 6>браабрака	дабраабрака
7	а< 7>браабракад	аб< 7>раабракад	абраабракад
8	б< 8>раабракада	бр< 8>аабракада	браабракада
9	р< 9>аабракадаб	ра< 9>абракадаб	раабракадаб
10	а<10>абракадабр	аа<10>бракадабр	аабракадабр
	Второй шаг. Сортировка		
0	а< 0>бракадабра	аа<10>бракадабр	аабракадабр
1	а< 3>кадабраабр	аб< 0>ракадабра	абраабракад
2	а< 5>дабраабрак	аб< 7>раабракад	абракадабра
3	а< 7>браабракад	ад< 5>абраабрак	адабраабрак
4	а<10>абракадабр	ак< 3>адабраабр	акадабраабр
5	б< 1>ракадабраа	бр< 1>акадабраа	браабракада
6	б< 8>раабракада	бр< 8>аабракада	бракадабраа
7	д< 6>абраабрака	да< 6>браабрака	дабраабрака
8	к< 4>адабраабра	ка< 4>дабраабра	кадабраабра
9	р< 2>акадабрааб	ра< 2>кадабрааб	раабракадаб
10	р< 9>аабракадаб	ра< 9>абракадаб	ракадабрааб
	Результат		
	аркдрааааббб,5	радкрааааббб,5	рдакрааааббб,2

Рис. 5.10. Частичные сортирующие преобразования 1-го и 2-го порядка

Так же как и в BWT, результатом преобразования являются последний столбец матрицы и номер строки, последний символ которой является начальным символом исходной строки.

Упражнение. Выполните преобразование ST(3) строки "абракадабра".

Легко заметить, что различие между частичным сортирующим преобразованием и преобразованием Барроуза – Уилера можно увидеть только при сортировке устойчивых контекстов длиннее порядка преобразования ST. В методе BWT в этом случае продолжается процесс сравнения символов, а в ST – сравнение символов прекращается и выше располагается та строка, первый символ которой встретился во входном блоке раньше. Следовательно, те данные, в которых встречаются длинные повторы, более эффективно сжимаются преобразованием Барроуза – Уилера. К примеру, типичные текстовые файлы на английском языке теряют в сжатии около 5% при выполнении сортировки по четырем символам.

С другой стороны, частичное сортирующее преобразование не требует полной сортировки всех строк матрицы и свободно от тех проблем, которые возникают при преобразовании очень избыточных данных с использованием полной сортировки. Как правило, данное преобразование выполняется быстрее, чем BWT.

Но при выполнении обратного преобразования наблюдается иная картина. Если в случае BWT оно выполняется легко и просто, то для восстановления исходных данных после частичного сортирующего преобразования необходимо приложить дополнительные усилия. А именно вести учет количества одинаковых контекстов. И чем порядок преобразования больше, тем требуется больше времени на подсчет.

Методы, используемые совместно с BWT

Как уже было сказано, само по себе преобразование Барроуза – Уилера не сжимает. Эту работу проделывают другие методы, призванные толково распорядиться теми свойствами, которыми обладают преобразованные данные.

Среди таких методов можно отметить следующие:

- кодирование длин повторов (RLE);
- метод перемещения стопки книг [35] (MTF);
- кодирование расстояний (DC);
- метод Хаффмана;
- арифметическое кодирование.

Последовательность применения методов, используемых совместно с BWT:

Шаг	Используемый алгоритм	
1	Кодирование длин повторов (необязательно)	
2	Преобразование Барроуза – Уилера	Частичное сортирующее преобразование
3	Перемещение стопки книг	Кодирование расстояний
4	Кодирование длин повторов (необязательно)	
5	Метод Хаффмана	Арифметическое кодирование

Перемещение стопки книг

Метод также известен под названием MTF (Move To Front). Суть его легко понять, если представить процесс перемещения книг в стопке, из которой время от времени достают нужную книгу и кладут сверху. Таким образом, через некоторое время наиболее часто используемые книги оказываются ближе к верхушке стопки.

Введем следующие обозначения:

N – число символов в алфавите;

M – упорядоченный список символов размером N; M[0] соответствует верхней книге стопки, M[N-1] – нижней;

x – очередной символ.

```
int tmp1, tmp2, i=0;
tmp1 = M[i];
M[i] = x;
while( tmp1 != x ) {
   i++;
   tmp2 = tmp1;
   tmp1 = M[i];
   M[i] = tmp2;
}
```

Для примера произведем преобразование над последовательностью "рдакрааааббб", полученной нами после BWT. Предположим, что мы имеем дело с алфавитом, содержащим только эти 5 символов и в упорядоченном списке символов они расположены в следующем порядке: M = {"а", "б", "д", "к", "р"}.

Первый из символов последовательности, "р", находится в списке под номером 4. Это число мы и записываем в выходной блок. Затем мы изменяем список, перенося этот символ в вершину списка, при этом сдвигая все остальные элементы, находившиеся до этого выше.

Следующий символ, "д", после этого сдвига оказывается в списке под номером 3. И т. д. (рис. 5.11).

Символ	Список	Номер
р	абдкр	4
д	рабдк	3
а	драбк	2
к	адрбк	4
р	кадрб	3
а	ркадб	2
а	аркдб	0
а	аркдб	0
а	аркдб	0
б	аркдб	4
б	баркд	0

Рис. 5.11. MTF-преобразование строки "рдакрааааббб"

Таким образом, в результате преобразования по методу перемещения стопки книг мы получили последовательность "43243200040".

Вспомним, что результат преобразования Барроуза – Уилера представляет собой последовательность символов, среди которых часто попадаются идущие подряд одинаковые символы. Поэтому, чтобы эффективно сжать такую последовательность, статистическому кодеру необходимо вовремя отслеживать смену одного частого символа другим. MTF предназначен для того, чтобы облегчить задачу статистическому кодеру.

Рассмотрим последовательность "bbbbccccccddddddaaaaab", обладающую такими свойствами. Попробуем обойтись без MTF и закодировать ее по методу Хаффмана. Для упрощения будем исходить из предположения, что затраты на хранение дерева, требуемого для обеспечения декодирования, будут равны и с использованием MTF и без него.

Вероятности всех четырех символов в данном примере равны 1/4, т. е. для кодирования каждого из символов нам потребуется 2 бита. Легко посчитать, что в результате кодирования мы получим последовательность длиной 20·2 = 40 бит.

Теперь проделаем то же самое со строкой, подвергнутой MTF-преобразованию. Предположим, что начальный список символов выглядит как {"a", "b", "c", "d"} (рис. 5.12).

```
bbbbccccccddddddaaaaab      исходная строка
10002000030000300003        строка после MTF
```

Символ	Частота	Вероятность	Код Хаффмана
0	15	3/4	0
3	3	3/20	10
1	1	1/20	110
2	1	1/20	111

Рис. 5.12. Кодирование методом Хаффмана строки после MTF-преобразования

В результате кодирования получаем последовательность длиной 15*1 + + 3*2 + 1*3 + 1*3 = 27 бит.

Упражнение. Произведите преобразование методом стопки книг последовательности "bbbbbccccdddcceaaaaa" и определите, будет ли использование MTF давать преимущество при кодировании методом Хаффмана. Начальный упорядоченный список символов установить {"a", "b", "c", "d", "e"}. Исходите из предположения, что алфавит состоит только из указанных пяти символов.

Кодирование длин повторов

Этот метод также называется Run Length Encoding (RLE). Это один из наиболее старых методов сжатия. Суть этого метода заключается в замене идущих подряд одинаковых символов числом, характеризующим их количество. Конечно, также мы должны и указать признак "включения" механизма кодирования длин повторов, который можем распознать при декодировании.

Один из возможных вариантов – включать кодирование, когда число повторяющихся символов превысит некоторый порог. Например, если мы условимся, что порог равняется трем символам, то последовательность "aaaaabbbccccdd" в результате кодирования будет выглядеть как "aaa2bbb0ccc1dd". Если мы выберем в качестве порога 4 символа, то получим "aaaa1bbbcccc0dd".

Упражнение. Что получится, если закодировать повторы в данной строке, используя порог, равный двум символам?

Главное назначение кодирования длин повторов в связке с BWT – увеличить скорость сжатия и разжатия.

RLE можно применить дважды – до преобразования и после. До преобразования данный метод может пригодиться, если мы имеем дело с потоком, содержащим много повторов одинаковых символов. Сортировка строк матрицы перестановок – наиболее длительная из процедур, необходимых для сжатия при помощи BWT. В случае высокоизбыточных данных время выполнения этой процедуры может существенно (в разы) возрастать. Сейчас разработаны методы сортировки, устойчивые к такого рода избыточности данных, но ранее метод кодирования длин повторов широко использовался на этом этапе ценой небольшого ухудшения сжатия. RLE следует применять, если указанных повторов уж слишком много.

Не является обязательным и другое применение RLE – кодирование длин повторов после преобразования Барроуза – Уилера. Оно довольно эффективно реализовано в szip [33] и BA, но известны архиваторы, в которых RLE не требуется, например такие, которые используют кодирование расстояний (DC, YBS). Ряд архиваторов использует некую разновидность кодирования длин повторов – 1–2 кодирование, описанное ниже. В любом

случае, если не воспользоваться каким-нибудь из перечисленных методов сокращения количества выходных символов, скорость работы будет оставлять желать лучшего, особенно в случае архиваторов, в которых используется арифметическое кодирование.

Усовершенствование метода перемещения стопки книг

MTF является вполне самостоятельным преобразованием и, помимо использования вкупе с BWT, применяется и в других областях. Но сейчас мы рассмотрим модификации метода перемещения стопки книг, которые помогают улучшить сжатие данных, полученных именно после преобразования Барроуза – Уилера.

Когда мы выписываем символы последнего столбца матрицы перестановок, относящиеся к весьма близким контекстам, мы можем с достаточно большой долей уверенности утверждать, что для многих типов данных эти символы будут одинаковы. В частности, к таким типам данных относятся файлы, содержащие текст на естественном языке.

Однако возможны небольшие нарушения такой закономерности – за счет ошибок, за счет наличия больших букв в начале предложения, переносов слов и т. д. Эти нарушения на выходе BWT часто выглядят как небольшие вкрапления посторонних символов среди длинной цепочки одинаковых. Очевидно, вкрапления из одного редкого символа будут встречаться чаще двойных, тройных и более длинных.

Можно заметить, что при MTF-преобразовании такие одиночные символы приводят к двойному появлению единичных кодов, что ухудшает нам статистику. Способ преодоления такой неприятности довольно прост. Следует отложить продвижение символа на верхушку списка в том случае, если этот символ не находится в позиции, соответствующей коду 1. На рис. 5.13 представлено, как будет выглядеть преобразование последовательности "рдакрааааббв" в этом случае.

Символ	Список	Выход
р	абдкр	4
д	арбдк	3
а	адрбк	0
к	адрбк	4
р	акдрб	3
а	аркдб	0
а	аркдб	0
а	аркдб	0
а	аркдб	0
б	аркдб	4
б	абркд	1

Рис. 5.13. Модифицированное MTF-преобразование строки "рдакрааааббв"

Преимущество такой модификации видно на примере MTF-преобразования типичной для английских текстов последовательности "ttttTtttwtt", полученной из части последнего столбца матрицы перестановок:

```
he_   ...   t
he_   ...   t
he_   ...   t
he_   ...   t
he_   ...   T
he_   ...   t
he_   ...   t
hen_  ...   t
hen_  ...   w
hen_  ...   t
hen_  ...   t
```

Предположив, что упорядоченный список содержит символы в порядке {"t", "w", "T" ...}, в результате применения метода перемещения стопки книг получаем следующие результаты:

ttttTtttwtt	
00002100210	Обычный MTF
00002000200	Модифицированный MTF

Как видно из примера, на типах данных, обладающих описанными свойствами, усовершенствованный метод перемещения стопки книг дает распределение кодов MTF, которое лучше поддается сжатию за счет большего количества нулевых кодов.

Но в случае появления двойных редких символов, этот метод дает результаты хуже классического MTF. Например, если нам попалась последовательность "ttttTTtttwwtt":

ttttTTtttwwtt	
0000201002010	Обычный MTF
0000211002110	Модифицированный MTF

Но, как уже было замечено, вероятность появления двух идущих подряд редких символов меньше, чем вероятность появления одного.

Картину также может испортить ситуация, когда мы имеем дело с символами, соответствующими не устойчивым контекстам, а участку смены одного контекста другим. В этом случае задержка в перемещении символа к вершине списка может сослужить плохую службу при отслеживании появления нового устойчивого контекста. Справедливости ради стоит отметить, что и в этом случае модифицированное MTF-преобразование все же обычно реагирует достаточно быстро.

Резюмируя, можно сказать, что описанный вариант метода перемещения стопки книг на текстовых данных оказывается лучше классического, а однозначно поручиться за сохранение такого преимущества при обработке данных другого типа нельзя. Возможно, более правильным выбором будет предварительный анализ данных или подсчет числа отложенных перемещений на вершину списка для принятия решения, какой из вариантов использовать.

Упражнение. Какой из методов MTF-преобразования будет эффективнее для последовательности символов "aaacbaaa"? Предположим, что начальный упорядоченный список символов выглядит как {"a", "b", "c"}.

Преимущество модифицированного алгоритма для текстовых данных можно оценить на примере файла book1 из набора файлов CalgCC, часто используемого для оценки архиваторов. На рис. 5.14 приводятся частоты рангов для обоих методов перемещения стопки книг.

Ранги	MTF обычный, %	MTF модифицированный, %
0	49.77	51.43
1	15.36	14.93
2	7.91	7.45
3	5.28	4.96
4	3.79	3.67
5	2.91	2.81
6	2.35	2.29
7	1.98	1.94
8	1.68	1.63
9	1.45	1.44
10	1.26	1.23
11	1.06	1.05
12	0.91	0.90
13...255	4.33	4.31

Рис. 5.14. Статистика рангов MTF-преобразования для файла book1

1–2-КОДИРОВАНИЕ

При кодировании длин повторов символов применительно к преобразованию Барроуза – Уилера стоят две задачи:

- обеспечить кодирование чисел любой величины;
- адаптироваться к изменению величины избыточности данных.

Данные задачи успешно решает алгоритм, нашедший применение в ряде архиваторов (bzip2, IMP, BWC) и названный 1–2-кодированием. Суть его заключается в том, что число, соответствующее количеству повторов, кодируется посредством двухсимвольного алфавита.

При использовании 1–2 кодирования в связке с MTF мы отводим под нулевой ранг не один, а два символа (назовем их z_1 и z_2), увеличивая таким образом алфавит до 257 символов. Символ, отличный от z_1 и z_2, является признаком окончания записи числа повторов.

Число повторов кодируется так, как это представлено на рис. 5.15.

Число повторов	Код
1	z_1
2	z_2
3	z_1z_1
4	z_1z_2
5	z_2z_1
6	z_2z_2
7	$z_1z_1z_1$
8	$z_1z_1z_2$
9	$z_1z_2z_1$
10	$z_1z_2z_2$
11	$z_2z_1z_1$
12	$z_2z_1z_2$
13	$z_2z_2z_1$
14	$z_2z_2z_2$
15	$z_1z_1z_1z_1$
...	

Рис. 5.15. 1–2-кодирование чисел

Упражнение. Закодируйте посредством описанного алгоритма число 30. Какому числу соответствует последовательность $z_2z_2z_1z_1z_2$?

Как можно видеть, данный способ обеспечивает кодирование чисел в любом диапазоне. Этот метод соответствует и второму из предъявляемых требований. Увеличение избыточности данных, приводящее к возрастанию концентрации нулевых рангов MTF, приводит к увеличению частот символов z_1 и z_2. Причем если преобладают короткие последовательности MTF-0, то частота символа z_1 превосходит частоту символа z_2.

Предвидя возможное замечание о том, что, например, число 7 приводит к большему возрастанию частоты символа z_1, чем число 1, отметим следующее. Как правило, распределение частот чисел повторов в среднем убывает с ростом числа, а частоты появлений близких по значению чисел близки. Причем обычно с ростом чисел различие частот уменьшается. Поэтому:

- частота числа 7 больше частоты числа 1 только в редких, скорее вырожденных случаях;
- увеличение частоты числа 7 за счет преобладания над частотами чисел 8, 9 и 10 приводит к более интенсивному использованию символа z_1, так

как число символов z_1 в коде, соответствующем числу 7, максимально среди всех трехсимвольных кодов.

В целом преобладание частоты символа z_1 над частотой символа z_2 определяется преобладанием частоты одной группы символов над другой (рис. 5.16).

Группа, приводящая к росту доли z_1	Группа, приводящая к росту доли z_2
1	2
3	4
5	6
7	8
...	...
3,4	5,6
7,8	9,10
11,12	13,14
...	...
7...10	11...14
15...18	19...22
...	...

Рис. 5.16. Свойства 1–2-кодирования

После двух преобразований строка "абракадабра" выглядела как {4,3,2,4,3,2,0,0,0,4,0}. Подвергнем ее 1–2-кодированию. Воспользовавшись рис. 5.16, получим {4,3,2,4,3,2,z_1,z_1,4,z_1}.

Упражнение. Как будет выглядеть исходная строка в результате указанных двух преобразований и 1–2-кодирования, если вместо обычного применить модифицированный MTF? Для справки: после BWT и модифицированного MTF была получена последовательность {4,3,0,4,2,0,0,0,0,4,1}.

Реализация 1–2-кодирования

```
// Функция 1-2-кодирования.
// Выводит последовательность символов z1 и
// z2, соответствующую числу count.

void z1z2( int count ) {

  // длина последовательности
  int len=0;

  // число 0 не кодируется
  if( !count ) return;

  // находим длину последовательности
  { int t = count+1;
    do { len++;
         t >>= 1;
```

```
    } while( t > 1 );
  }

  // кодирование последовательности
  do { len--;
    putc((count & (1<<len))? z2:z1, output );
  } while( len );
}

// кодирование
// использует функцию z1z2()
void encode( void ) {
  unsigned char c;
  int count = 0;           // число повторов
  while( !feof( input ) ) {
    c = getc( input );
    if( c == 0 ) count++; // считаем MTF-0
    else {
      // вводим число MTF-0
      z1z2( count );
      count = 0;
      // выводим символ, отличный от MTF_0
      putc( c, output );
    }
  }
  z1z2( count );
}

// декодирование
void decode( void ) {
  unsigned char c;
  int count = 0;           // число повторов
  while( !feof( input ) ) {
    c = getc( input );

    // чтение последовательности z1z2
    if       ( c == z1 ) {
      count += count + 1;
    } else if( c == z2 ) {
      count += count + 2;

    } else {
      // вывод MTF-0, заданные числом count
      while( count-- ) putc( 0, output );
      putc( in[i], output );
    }
    i++;
  }
  while( count-- ) putc( 0, output );
}
```

Кодирование расстояний

В течение нескольких лет метод перемещения стопки книг был непременным атрибутом архиватора, построенного на основе преобразования Барроуза – Уилера. В силу того, что этот алгоритм недостаточно точно учитывал соотношение частот символов, соответствующих разным рангам, разработчики постоянно стремились найти достойную замену.

Одним из наиболее распространенных был подход, при котором отдельно строилась модель для наиболее часто используемых символов. Для таких символов вероятности просчитывались отдельно. Впервые этот подход был описан Петером Фенвиком [18], но автору не удалось превзойти результаты модели, использующей традиционный подход. Более удачным было применение кеширования наиболее частых символов в архиваторе szip, разработанном Шиндлером [33].

Самым поздним из методов, пришедших на смену MTF, является метод кодирования расстояний (Distance Coding). Первый же из архиваторов, использующих этот метод, сумел превзойти своих конкурентов по степени сжатия большинства типовых данных. Теперь, помимо архиватора DC, кодирование расстояний используют YBS и SBC.

Были публикации, посвященные родственному методу, названному Inversion Frequencies авторами одной из работ [5,1]. Помимо метода кодирования расстояний, ниже мы рассмотрим и его.

Возьмем для примера ту же строку "рдакрааааббб", полученную в результате преобразования Барроуза – Уилера.

В качестве первого, подготовительного этапа нам следует определить первые вхождения символов в данную строку. Для этого в начало строки припишем все символы алфавита в произвольном, например в лексикографическом, порядке. Поскольку при декодировании мы можем проделать то же самое, данная операция является обратимой.

Для удобства добавим в конец строки символ, означающий конец блока. При ссылке на этот символ мы можем распознать окончание цепочки символов, от которых эта ссылка сделана. Обозначим символ конца блока как "$".

Итак, после этих приготовлений наша строка выглядит как "абдкррдакрааааббб$". Причем при декодировании нам на этом подготовительном этапе известны первые символы строки, "абдкр", и последний, "$".

строка:	абдкррдакрааааббб$
известные символы:	абдкр.....$

Теперь займемся собственно кодированием расстояний. Для этого берем первый символ, "а", и ищем ближайший такой же. Расстояние до него равно шести – это число иных символов между "а". Но из этих шести символов

нам уже известны 4 и при декодировании мы заранее знаем, что очередной символ "а" никак не может попасть в эти позиции. Наша задача – закодировать номер той вакантной позиции, на которую выпадает этот символ. Это номер 6 – 4 = 2.

строка:	абдкррдакрааааббS
известные символы:	**а**бдкр..**а**.....$
расстояние:	**2**

Аналогично кодируем еще несколько символов по очереди, подсчитывая число точек, символизирующих незанятые позиции, в строке известных символов.

строка:	**абдкррдакрааааббS**
известные символы:	**а**бдкр..а.....**б**.$
расстояние:	**28**
известные символы:	**а**б**д**кр.**д**а.....б.$
расстояние:	**281**
известные символы:	абд**к**р.да**к**.....б.$
расстояние:	**2811**
известные символы:	абдк**рр**дак.....б.$
расстояние:	2811-

При кодировании первого из символов "р" вместо ссылки на следующий символ поставлен прочерк, потому что сразу после символа "р" находится вакантная позиция, и это означает, что никакой другой символ не сможет на эту позицию сослаться. Значит, нам нет необходимости выполнять кодирование этой ссылки. В данном случае мы наблюдаем специфический эффект, присущий методу кодирования расстояний, который позволяет избежать применения RLE.

строка:	абдкррдакрааааббS
известные символы:	абдкр**р**дак**р**....б.$
расстояние:	2811-**0**

Поскольку очередной символ "р" занимает ближайшую вакантную позицию, мы кодируем его числом 0. Благодаря такому свойству метода кодирования расстояний, в нем достаточно легко решается проблема случайной смены контекста, ради которой требовалось специально совершенствовать метод перемещения стопки книг.

строка:	абдкррдакрааааббS
известные символы:	абдкр**р**дакр....б.**$**
расстояние:	2811-**05**

Кодируя символ "д", мы сделали ссылку на конец строки. При декодировании такая ссылка позволит понять, что символы "д" закончились (рис. 5.17).

строка:	абдкррдакрааааббS
известные символы:	абдкррда**к**ра...б.$
расстояние:	2811-05**0**
известные символы:	абдкррда**к**ра...б.**$**
расстояние:	2811-050**4**
известные символы:	абдкррдак**р**а...б.**$**
расстояние:	2811-0504**4**
известные символы:	абдкррдакр**аааа**б.$
расстояние:	2811-05044**---**
известные символы:	А**б**дкррдакрааа**а**б.**$**
расстояние:	2811-05044---**1**
известные символы:	А**б**дкррдакрааа**аб**б$
расстояние:	2811-05044---1-

Рис. 5.17. Кодирование расстояний

В итоге получаем последовательность { 2,8,1,1,0,5,0,4,4,1 }.

Упражнение. Вычислите расстояния для строки "бррааааакаакдрр".

Обратные частоты

Есть еще один метод, похожий на описанный выше. В нем также кодируются расстояния между одинаковыми символами. Отличие только в том, что символы, для которых определяются расстояния, берутся не в порядке поступления, а исходя из некоторого фиксированного порядка. Например, по алфавиту.

Авторы данного алгоритма, названного Inversion Frequencies (IF), исходили из того, что расстояние между одинаковыми символами характеризует частоту использования этих символов на данном отрезке символьной последовательности. Чем расстояние меньше, тем выше частота. Поясним работу алгоритма на примере.

Предположим, нам нужно определить IF для строки "рдакрааааббб", а расчет расстояний будем проводить в соответствии с положением символов в алфавите "абдкр".

Сначала запишем для символа "а" положение первого из таких символов в исходной строке и их количество.

Символ	Первая позиция	Число символов
а	2	5

Следующим шагом – для каждого из символов "а" в качестве расстояния запишем число иных символов до следующего "а".

исходная строка:	рдакрааааббр
расстояния:	2 000

После того как все позиции символа "а" определены, мы можем удалить их из исходной строки и продолжить обработку строки, как будто их и не было (рис. 5.18).

Символ	Первая позиция	Число символов	Строка	Расстояния
а	2	5	рдакрааааббр	2,0,0,0
б	4	2	рдкрббр	0
д	1	1	рдкрр	
к	1	1	ркрр	

Рис. 5.18. Вычисление обратных частот

В кодировании символа "р" нет необходимости, так как заведомо известно, что он занимает оставшиеся позиции в строке. Таким образом, мы получили такую последовательность: {{2,5,2,0,0,0}, {4,2,0}, {1,1}, {1,1}}.

Легко заметить, что способ и эффективность кодирования зависят от того порядка, в котором мы будем обрабатывать символы. Еще одно важное отличие данного метода от кодирования расстояний заключается в том, что он не освобождает от необходимости кодирования длинных последовательностей нулевых значений обратных частот.

Упражнение. Определите обратные частоты для строки "бррааааакаакдрр".

Способы сжатия преобразованных с помощью BWT данных

Перейдем к рассмотрению способов сжатия данных, полученных в результате описанных выше преобразований. Для начала надо разобраться с данными, которые нам необходимо сжать.

Рассмотрим фрагмент данных, типичный для текста, преобразованного посредством BWT. Для примера взят файл book1 из набора "Calgary Corpus" (рис. 5.19).

```
eteksehendeynkrtdserttnregensksngsgsedeneyswmessrne
xgynystslgyegsgstssrhmsstetehselxtptneessthndesddy
htksthwtpfdttegedmmhysyresprssneenselgetdemsetse,t
reehsetrtteseeeeeeesssdeedmnlendeedgtdgttdsgtteeysy
tddentnrxsltshtghnteeernsdpwlttensedehsteeswekheee
teneeeeseslteenestrngsthsdgeyeyrteetklrdtetteyodth
eegeercesyttesedenrtresnyssgttsslsawssygsssrewmsht
gt,etssgehnehessssehneesesdnrnekhtrsslthdsseestste
nbgeeessesdesyndtrdhpeesehesetsrerhyesdnwtlrrhoses
hsetdrptttsdhaynenetyntpgstesknhysftsgssdftgteeedu
```

Рис. 5.19. Однородный фрагмент

Можно заметить, что распределение символов на этом фрагменте (рис. 5.19) меняется незначительно. Совсем другую картину мы можем наблюдать на рис. 5.20, когда преобладание одних символов сменяется преобладанием других.

```
ygeldsd,,ttyogdodgdedndygnmotedgwkgodoowtdoddtotet
tndmeggkrdsgtohtdegteddrsttttegtddewdddoootdgdntet
stsstttstt!uetttdtte-eIttetny,hettItgrlttyIttnttyrt
rttttttttettttdtttstottttttttttntttttttn,ddtkgnde;
ed,d,stefoefssfrsnsstyslw,fnoeadere,rteeynsfofhynn
nyoytted,yfnedhddtoldtnyhnnhyrtyryttmeryesfoyedney
oymd,sedesgnrrsysnssmydsspdyt'-dssehs,gynsdydgee'o
defddeynt,,tdnd,os;sttyysofy-nnetognfetdnyldlewhe-
odsttemshsdtsyteny,ngdefs,-offsnntettseyesgleay:es
dtsdgllredksyryes,rldosts;dtdeefgghsfrergkdgngkenw
```

Рис. 5.20. Изменяющийся фрагмент

Также бывают случаи, когда на протяжении всего фрагмента явно превалирует один определенный символ (рис. 5.21).

```
edeeeeeIeydeyeeet'eeeIeeeeeeeeeeeeeeeeeeeeeeleeeee
eeeeeee!eeeeleeeeeeeeeeeeeeeeeeeeeeeyedeeereeeleee
eeeeeeeeeeslyadeeeeeeeeeyeeeeeeeeeeeeeedyereseeeee
eIueeeeeeyeeeeeeIsseseeIl'eeIetenhyalehcesyyseessn
s'dlesffao,dffeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeTeeee
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeteeeeeeeeeeeeee
eeeeeeeeeeaerre,see,eywesldsysdhedeeetaeeesaeeyedo
```

Рис. 5.21. Фрагмент с преобладанием одного символа

Итак, выделим 3 вида данных:

1) на протяжении всего фрагмента несколько символов имеют постоянную частоту;
2) промежуток в фрагменте, когда преобладание одних символов сменяется преобладанием других;
3) один из символов встречается намного чаще других.

Одна из задач выбираемой модели заключается в том, чтобы позволить оперативно настроиться на текущий вид данных.

Сжатие при помощи кодирования по алгоритму Хаффмана

Указанные свойства преобразованных данных использует алгоритм, реализованный в архиваторе bzip2 и позднее позаимствованный также разработчиками архиватора IMP.

После преобразований BWT и MTF блок данных делится на равные 50-символьные куски. Полученные куски объединяются в группы по степени близости распределений MTF-рангов. Количество групп зависит от размера файла. Например, для мегабайтового файла таких групп будет шесть.

Группирование кусков выполняется итеративно. Изначально куски приписываются группам в порядке следования в блоке так, чтобы каждой группе соответствовало примерно равное количество кусков. Для каждой группы строится отдельное дерево Хаффмана. В архив записываются деревья Хаффмана, номер группы для каждого из 50-символьного кусков, а затем все куски сжимаются по алгоритму Хаффмана в соответствии с номером той группы, к которой они относятся.

Выбор группы для куска делается на основе подсчета длины закодированного куска, который получится при использовании каждого из построенных деревьев Хаффмана. Выбирается та группа, при выборе которой код получается короче. Поскольку после этого статистика использования символов в группах меняется, по завершении обработки блока деревья для каждой группы строятся заново. Практика показывает, что вполне достаточно четырех итераций для получения приемлемого сжатия. С каждой новой итерации прирост эффективности резко уменьшается.

Такой способ кодирования называется полуадаптивным алгоритмом Хаффмана. Полуадаптивность заключается в том, что адаптация происходит за счет выбора подходящего дерева Хаффмана для очередного куска данных, а не за счет перестройки текущего дерева.

Сжатие по алгоритму Хаффмана довольно эффективно, хоть и уступает алгоритмам, в которых реализовано арифметическое сжатие, но зато заметно быстрее при декодировании.

Алгоритмы, использующие кодирование по методу Хаффмана, довольно эффективны, хоть и уступают алгоритмам, в которых реализовано арифметическое сжатие

Структурная и иерархическая модели

Отметим основное свойство таких преобразований, как MTF, DC и IF: на большинстве данных, полученных в результате преобразования Барроуза – Уилера, малые значения встречаются гораздо чаще больших и легче подда-

ются предсказанию. Это свойство очень важно учитывать при построении адаптивных моделей, использующих арифметическое кодирование.

Анализируя перечисленные выше 3 вида фрагментов, можно отметить некоторые особенности, которыми можно воспользоваться при моделировании:

- большое количество идущих подряд одинаковых символов свидетельствует о том, что вероятно появление такой же длинной их последовательности и в будущем;
- появление символа, довольно давно встречавшегося последний раз, говорит о том, что вероятна смена устойчивого контекста, а следовательно, и наиболее частые до этого символы могут смениться другими.

Указанные свойства послужили причиной того, что практически все реализации сжатия на основе преобразования Барроуза – Уилера уделяют повышенное внимание наиболее частым в текущем фрагменте символам. Для обработки редких символов важна лишь приблизительная оценка возможности их появления.

Рассмотрим две модели, обладающие описанными качествами, – структурную и иерархическую [18].

Структурная модель. Кодирование символа осуществляется в два этапа. Сначала кодируется номер группы, к которой этот символ относится. А затем – номер символа внутри группы.

Размеры групп различны. Наиболее частые символы помещаются в группы, состоящие из небольшого числа символов. Предложенная Петером Фенвиком структурная модель предполагает самостоятельное существование 0-го и 1-го ранга, т. е. каждый из них представляет собой отдельную группу. В следующую (третью по счету) группу помещаются 2-й и 3-й ранг, затем – с четвертого по седьмой и т. д. (рис. 5.22).

Группа	Кодируемые ранги			
0	0			
1	1			
2	2	3		
3	4	5	6	7
. . .	. . .			
8	128	129	...	255

Рис. 5.22. Структурная модель

Ниже, рис. 5.23, приводятся частоты групп для файла book1 при использовании обычного и модифицированного MTF.

Номер группы	MTF обычный, %	MTF модифицированный, %
0	49.77	51.44
1	15.36	14.93
2	13.19	12.41
3	11.05	10.72
4	8.28	8.16
5	2.14	2.13
6	0.20	0.20
7	0.01	0.01
8	0.00	0.00

Рис. 5.23. Статистика распределения рангов в группах структурной модели

Иерархическая модель. Эта модель также делит символы на группы. И по такому же принципу. Отличие заключается в том, что в каждую группу добавляется символ, означающий уход к группе, находящейся на более низкой ступени иерархии (рис. 5.24).

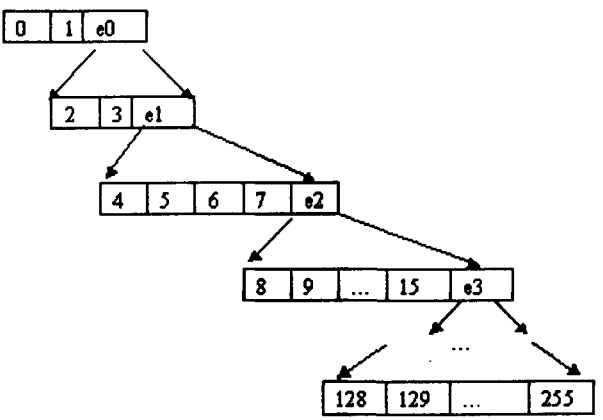


Рис. 5.24. Иерархическая модель

Эксперимент [16] показал, что структурная модель позволяет достичь немного лучшего сжатия и, в силу того, что для каждого символа в среднем требуется меньше операций арифметического кодирования, чуть быстрее.

Размер блока

Один из важных вопросов, который встает перед разработчиком архиватора на основе преобразования Барроуза – Уилера, заключается в выборе размера блока.

BWT – алгоритм, дающий наилучшую результативность при сжатии однородных данных. А однородные данные предполагают наличие устойчи-

вых сочетаний символов. А раз сочетания не меняются, то увеличение размера блока приведет только к увеличению числа одинаковых символов, находящихся рядом в результате преобразования. А увеличение количества находящихся рядом символов вдвое в идеале требует всего одного дополнительного бита при кодировании. Таким образом, для однородных данных можно сделать однозначный вывод: чем блок больше, тем сжатие будет сильнее. Ниже приведены экспериментальные данные сжатия файла book1 из набора CalgCC. Тестирование производилось на компьютере следующей конфигурации:

Процессор	Intel Pentium III 840 МГц
Частота шины	140 МГц
Оперативная память	256 Мб SDRAM
Жесткий диск	Quantum FB 4.3 Гб
Операционная система	Windows NT 4.0 Service Pack 3

Размер блока, Кб	bzip2 1.01 Размер сжатого файла, байт	bzip2 1.01 Время сжатия, с	YBS 0.03e Размер сжатого файла, байт	YBS 0.03e Время сжатия, с
100	270,508	0.61	255,428	0.65
200	256,002	0.66	239,392	0.66
300	249,793	0.71	232,681	0.71
400	243,402	0.72	225,659	0.73
500	242,662	0.73	224,782	0.74
600	241,169	0.75	222,782	0.76
700	237,993	0.76	218,826	0.77
800	**232,598**	0.77	**213,722**	0.77

Что касается данных неоднородных, то здесь картина иная. Размер блока надо выбирать таким, чтобы его край приходился на то место, где данные резко меняются. Причем для BWT страшна не сама неоднородность как таковая, а изменение статистики символов, предсказываемых устойчивыми контекстами. Например, нас не должен пугать тот факт, что в начале файла часто встречаются строки "abcd", а в конце их место занимают "efgh". Гораздо неприятнее, когда вместо строк "abcd" начинают появляться строки "abgh". Это не означает, что файл, в котором наблюдается вторая ситуация, сожмется при помощи BWT-компрессора очень плохо. Но разница в сжатии по сравнению с архиваторами, использующими, например, словарные методы, на таких данных будет не в пользу BWT.

Для иллюстрации зависимости эффективности сжатия неоднородных данных от размера блока сожмем исполнимый файл из дистрибутива компилятора Watcom 10.0, wcc386.exe (536,624 байта).

Размер блока, Кб	bzip2 1.01 размер сжатого файла, байт	bzip2 1.01 время сжатия, с	YBS 0.03e размер сжатого файла, байт	YBS 0.03e время сжатия, с
100	309,716	0.66	279,596	0.60
200	308,668	0.66	277,312	0.61
300	308,812	0.66	277,285	0.61
400	309,163	0.66	277,374	0.60
500	307,131	0.66	274,351	0.60
600	308,624	0.66	276,026	0.60

Переупорядочение символов

В отличие от многих других методов сжатие, основанное на преобразовании Барроуза – Уилера, заметно зависит от лексикографического порядка следования символов. Замечено, что символы, имеющие сходное использование в словообразовании, лучше располагать поблизости.

Возьмем такие часто встречающиеся окончания слов русского языка, как -ый и -ий. Легко заметить, что им в большинстве случаев предшествуют одни и те же символы, например буква "н". Если эти окончания в результате сортировки каким-то образом окажутся в начале соседних строк матрицы перестановок, мы получим в последнем столбце рядом стоящие одинаковые символы.

Этот эффект особенно заметен на текстовых файлах. Для разных языков нюансы выбора лучшего переупорядочения символов могут отличаться, но общее правило таково – все символы надо поделить на 3 лексикографически отдельных группы: гласные, согласные и знаки препинания.

Файл	Размер архива, байт	
	bzip2 1.01	YBS 0.03e
book1	232,598	213,722
book1, после переупорядочения символов	231,884	212,975

Направление сортировки

Можно сортировать строки слева направо и в качестве результата преобразования использовать последний столбец матрицы отсортированных строк. А можно – справа налево и использовать первый столбец. Как делать лучше с точки зрения степени сжатия?

Первая стратегия подразумевает предсказание символа исходя из того, какие символы за ним следуют, а вторая – исходя из того, какие были раньше. В литературе для обозначения этих двух направлений используются словосочетания "following contexts" и "preceding contexts" соответственно (правосторонние и левосторонние контексты).

Практика показывает, что большинство архиваторов, использующих традиционные BWT и MTF, достигают лучшего сжатия на текстовых данных при использовании правосторонних контекстов. Для данных, имеющих "не лингвистическую" природу, лучше использовать левосторонние контексты. Например, это справедливо для исполнимых файлов.

Для компрессоров, которые не используют MTF, а проблему адаптации кодера к потоку преобразованных данных решают как-то иначе, выбор направления сортировки может быть иным. Например, DC и YBS многие исполнимые файлы, как и текстовые, сжимают лучше при сортировке слева направо.

Проделать самостоятельное сравнение очень просто. Возьмите ваш любимый архиватор и сожмите с его помощью ваш любимый файл. Затем переверните данные этого файла наоборот, сожмите полученное и сравните результаты.

Файл	Направление сортировки	Размер сжатого файла, байт	
		bzip2 1.01	YBS 0.03e
book1	following contexts	232,598	213,722
book1	preceding contexts	234,538	214,890
wcc386.exe	following contexts	308,624	276,026
wcc386.exe	preceding contexts	306,020	279,198

Некоторые программы самостоятельно пытаются определить тип данных и выбрать направление сортировки. Иногда, впрочем, ошибаясь. Простейший способ обмануть такой архиватор – дать ему сжать русский текст.

Сортировка, используемая в BWT

Сортировка – это очень важный компонент архиватора, реализующего сжатие на основе преобразования Барроуза – Уилера. Именно от нее зависит скорость сжатия. До недавнего времени именно сортировка была узким местом. В настоящее время моделирование стало достаточно сложным, чтобы конкурировать по времени работы с процедурой сортировки, реализации которой, напротив, совершенствуются в сторону ускорения. Но и теперь возможна ситуация, когда характеристики сжимаемых данных таковы, что могут существенно замедлить сортировку.

Основные требования к сортировке заключаются в том, что она должна обеспечивать быстрое сжатие обычных (преимущественно текстовых) данных и не приводить к существенному замедлению на очень избыточных данных.

Помешать сортировке могут два вида избыточности – когда в сортируемых данных содержатся:

- длинные одинаковые строки;
- короткие одинаковые строки в большом количестве.

Отдельный случай представляют собой большое число идущих подряд одинаковых символов или длинные последовательности перемежающихся символов типа "абабабаб".

Что касается текстовых файлов, наиболее часто встречаемая длина повторяющихся строк – 3–5 символов. Для файлов с исходными текстами программ, как правило, эта длина несколько больше – 8–12 символов.

Рассмотрим алгоритмы сортировок, получившие наибольшую известность.

Сортировка Бентли – Седжвика

Данный алгоритм получил, пожалуй, наибольшее распространение среди всех известных сортировок. Впервые он был применен еще одним из основоположников – Уилером. Затем эта сортировка была реализована в bzip2 и других архиваторах (BWC, BA, YBS).

Сортировка Бентли – Седжвика (Bentley – Sedgewick) представляет собой модификацию быстрой сортировки (quick-sort), ориентированной на сравнение длинных строк, среди которых может оказаться значительное количество похожих.

Главная идея описываемой сортировки заключается в том, что все сравниваемые с эталонной строки делятся не на 2, а на 3 группы. В третью группу входит сама эталонная строка и строки, сравниваемые символы которых равны соответствующим символам эталонной строки.

Выделение третьей группы помогает нам уменьшить число операций сравнения строк, имеющих много совпадающих подстрок. В эту группу попадают строки с одинаковыми начальными символами, что избавляет нас от необходимости сравнивать эти символы еще раз.

Работу алгоритма можно описать при помощи следующего исходного текста на языке Си:

```
void sort(char **s, int n, int d) {
  char **s_less, **s_eq, **s_greater;
  int   *n_less,  *n_eq,  *n_greater;

  // выбор значения, с которым будут
  // сравниваться d-е символы строк
  char v = choose_value(&s,d);

  // осталась только одна строка
  if( n <= 1 ) return;

  // деление всех строк на группы
```

```
  compare(&s, v, d,
    // строки, d-й символ которых меньше v
    &s_less,    &n_less,
    // строки, d-й символ которых равен  v
    &s_eq,      &n_eq,
    // строки, d-й символ которых больше v
    &s_greater, &n_greater);

  sort(&s_less,    n_less,    d);
  sort(&s_eq,      n_eq,      d+1);
  sort(&s_greater, n_greater, d);
}
```

Данная рекурсивная функция сортирует последовательность из n строк s, имеющих d одинаковых начальных символов. Самый первый вызов выглядит как sort(s,n,0);

Разумеется, с течением времени придумывалось все больше ухищрений, ускоряющих работу сортировки Бентли – Седжвика применительно к BWT. Перечислим основные из них.

1. Поразрядная сортировка. При большом количестве сортируемых строк предварительно осуществляется поразрядная сортировка по нескольким символам. По результату поразрядной сортировки строки делятся на пакеты, каждый из которых обрабатывается при помощи сортировки Бентли – Седжвика.
2. Использование результата предыдущих сравнений для последующих. После окончания сортировки некоторого количества строк можно легко отсортировать строки, начинающиеся на один символ раньше отсортированных. Для этого достаточно сравнить только первые их символы, а дальше – воспользоваться результатами предыдущей сортировки.
3. Сравнение не одиночных символов, а одновременно нескольких. Большая разрядность современных компьютеров позволяет выполнять операции сразу над несколькими символами, обычно представляемыми байтами. Например, если команды процессора позволяют оперировать 32-разрядными данными, то можно осуществлять одновременное сравнение 4 байт.
4. Неполная сортировка. В результате преобразования Барроуза – Уилера мы должны получить последовательность символов последнего столбца матрицы перестановок. Нам не важно, какая из двух строк будет лексикографически меньше, если им соответствуют одинаковые символы последнего столбца. Поэтому мы можем избежать лишних сравнений при сортировке строк.

Сортировка суффиксов

Применяя данную сортировку, мы исходим из того, что нам приходится сортировать строки, каждая из которых является частью другой, начинающейся с более ранней позиции в блоке, т. е. является ее суффиксом. Задача сортировки суффиксов неразрывно связана с построением дерева суффиксов, которое помимо сжатия данных может быть также использовано для быстрого поиска строк в блоке.

Главное свойство всех суффиксных сортировок заключается в том, что время сортировки почти не зависит от данных. Опишем одну из получивших большую известность суффиксных сортировок, которая была опубликована в 1998 г. Кунихико Садакане.

Рассмотрим алгоритм на примере. Введем обозначения:

X – массив суффиксов X[*i*], каждый из которых представляет собой строку, начинающуюся с *i*-й позиции в блоке;

I – массив индексов суффиксов; положение индексов в этом массиве должно соответствовать порядку лексикографически отсортированных суффиксов;

V[*i*] – номер группы, к которой относится суффикс X[*i*]; сортировка выполняется до тех пор, пока все значения в V не станут разными;

S[*i*] – число суффиксов, относящихся к группе *i*;

k – порядок сортировки.

Как всегда, будем производить преобразование Барроуза – Уилера строки "абракадабра$" (добавим к строке символ "$", означающий конец строки).

Шаг 1. Упорядочим все символы строки (для определенности предположим, что символ конца строки имеет наименьшее значение).

Затем заполним массив I значениями, равными позициям этих символов в исходной строке.

В массив V запишем по порядку номера групп, оставляя место для еще не упорядоченных элементов. Так, для символа "б" укажем номер группы, равный шести, чтобы оставить возможность для упорядочения всех пяти строк, начинающихся с символа "а".

И наконец, в массив S запишем размеры полученных групп. Небольшая хитрость, придуманная автором описываемого алгоритма, заключается в том, чтобы для групп, в которых осталась только одна строка, записывать отрицательное значение, равное количеству упорядоченных суффиксов до ближайшей неотсортированной группы. Это существенно ускоряет работу в случаях, когда у нас становится много отсортированных групп (на текстах такой момент, как правило, наступает уже на 3–4 шаге сортировки).

Позиции:	0	1	2	3	4	5	6	7	8	9	10	11
Исходная строка:	а	б	р	а	к	а	д	а	б	р	а	$
Упорядоченные символы:	$	а	а	а	а	а	б	б	д	к	р	р
I[i]	11	0	3	5	7	10	1	8	6	4	2	9
V[I[i]]	0	1	1	1	1	1	6	6	8	9	10	10
S[i]	-1	5					2		-2		2	

Шаг 2. Таким образом, завершена обработка суффиксов порядка k=0, т. е. одиночных символов. Теперь отсортируем пары (k=1). Для сортировки суффиксов каждой группы нам потребуются только значения номеров групп суффиксов (обозначаемых далее как Xk), находящихся на k символов правее сортируемых суффиксов. Например, для сортировки группы 1, соответствующей символу "а", нам потребовались группы 6, 9, 8, 6 и 0 от символов "б", "к", "д", "б" и "$".

Позиции:	0	1	2	3	4	5	6	7	8	9	10	11
Исходная строка:	а	б	р	а	к	а	д	а	б	р	а	$
Упорядоченные символы:	$	а	а	а	а	а	б	б	д	к	р	р
Суффикс Xk	а	б	к	д	б	$	р	р	а	а	а	а
V[I[i]+1]	1	6	9	8	6	0	10	10	1	1	1	1
Пары:	$а	аб	ак	ад	аб	а$	бр	бр	да	ка	ра	ра

После сортировки номеров групп:

V[I[i]+1]	1	0	6	6	8	9	10	10	1	1	1	1
Упорядоченные пары:	$а	а$	аб	аб	ад	ак	бр	бр	да	ка	ра	ра
I[i]	11	10	0	7	5	3	1	8	6	4	2	9
V[I[i]]	0	1	2	2	4	5	6	6	8	9	10	10
S[i]	-2		2		-2		2		-2		2	

Шаг 3. Поскольку все пары теперь отсортированы, можно упорядочить четверки. Для этого каждую группу (которые представляют собой суффиксы, начинающиеся с одинаковых пар) отсортируем в соответствии с парой символов, следующих после этих одинаковых пар. Прочерками отмечены уже отсортированные суффиксы, которые дальше обрабатывать нет необходимости.

Позиции:	0	1	2	3	4	5	6	7	8	9	10	11
Исходная строка:	а	б	р	а	к	а	д	а	б	р	а	$
Упорядоченные пары:	$а	а$	аб	аб	ад	ак	бр	бр	да	ка	ра	ра
Суффикс Xk	-	-	ра	ра	-	-	ак	а$	-	-	ка	$а
V[I[i]+2]			10	10			5	1			9	0

После сортировки номеров групп:

Суффикс Xk	-	-	ра	ра	-	-	а$	ак	-	-	$а	ка
V[I[i]+2]			10	10			1	5			9	0
I[i]	11	10	0	7	5	3	8	1	6	4	9	2
V[I[i]]	0	1	2	2	4	5	6	7	8	9	10	11
S[i]	-2		2		-8							

Шаг 4. Как можно заметить, нам осталось отсортировать всего одну группу, состоящую из двух элементов.

Позиции:	0	1	2	3	4	5	6	7	8	9	10	11
Исходная строка:	а	б	р	а	к	а	д	а	б	р	а	$
Упорядоченные четверки:	$абр	-	абра	абра	-	-	-	-	-	када	-	-
Суффикс Xk	-	-	када	$абр	-	-	-	-	-	-	-	-
V[I[i]+4]			9	0								

После сортировки номеров групп:

V[I[i]+4]			0	9			1	5			9	0
I[i]	11	10	7	0	5	3	8	1	6	4	9	2
V[I[i]]	0	1	2	3	4	5	6	7	8	9	10	11
S[i]	-12											

Таким образом, мы получили упорядоченные суффиксы, индексы которых записаны в массиве I (рис. 5.25)

i	I[i]	Суффикс
0	11	$
1	10	а$
2	7	абра$
3	0	абракадабра$
4	5	адабра$
5	3	акадабра$
6	8	бра$
7	1	бракадабра$
8	6	дабра$
9	4	кадабра$
10	9	ра$
11	2	ракадабра$

Рис. 5.25. Результат сортировки суффиксов

Упражнение. Выполните сортировку строки "tobeormottobe", используя описанный алгоритм.

Сравнение алгоритмов сортировки

Поскольку скорость сортировки во многом определяет быстродействие компрессоров, осуществляющих сжатие при помощи преобразования Барроуза – Уилера, над совершенствованием алгоритмов сортировок постоянно ведется работа. Можно отметить, что исследователи, тяготеющие к практическому применению, уделяют особое внимание сжатию типичных файлов, в то время как большинство публикаций в научных изданиях посвящено методам, позволяющим реализовать сортировку, устойчивую к вырожденным данным.

Для сравнения методов сортировок полезно ввести понятие средней длины совпадений (Average Match Length, AML), вычисляемой по следующей формуле:

$$AML = \frac{1}{n-1}\sum_{i=1}^{n-1} D(X[I[i]],X[I[i+1]]),$$

где N – размер блока данных, подвергаемого преобразованию; X – массив суффиксов X[i], каждый из которых представляет собой строку, начинающуюся с i-й позиции в блоке; I – массив индексов лексикографически упорядоченных суффиксов; D(x,y) – число совпадающих символов в одинаковых позициях строк x и y, начиная от первого символа строки и заканчивая первым несовпадением.

Исследования показали, что время, затрачиваемое суффиксной сортировкой, пропорционально логарифму средней длины совпадений, в то время как алгоритмы, основанные на быстрой сортировке или сортировке слиянием, демонстрируют, как правило, линейно-пропорциональную AML зависимость. Впрочем, справедливости ради стоит отметить, что усовершенствования алгоритма сортировки Бентли – Седжвика, описанные выше, существенно сокращают время сортировки, и поэтому этот алгоритм зачастую не уступает суффиксной сортировке даже на ряде вырожденных данных.

Цена такой устойчивости – повышенные накладные расходы на ее обеспечение.

Для сравнения ниже приведены экспериментальные данные, полученные на компьютере с процессором Intel Pentium 233 МГц и оперативной памятью 64 Мб. Были выбраны одни из наиболее оптимизированных представителей BWT-компрессоров для выявления слабых и сильных сторон различных методов. Требования к памяти участвующих в эксперименте программ довольно близки (от 6 до 8 размеров блока).

- DC 0.99.015b (автор – Эдгар Биндер). В данном архиваторе использованы поразрядная сортировка и сортировка слиянием.

- BA 1.01br5 (Микаель Лундквист), YBS 0.03e (Вадим Юкин), ARC (Ян Саттон). Во всех трех программах нашла свое применение сортировка Бентли – Седжвика вместе с поразрядной. Еще стоит упомянуть SBC 0.910 (Сами Мякинен), которая не участвовала в эксперименте по причине невозможности выделить сортировку отдельно от всех остальных процедур.
- QSuf (Ларссон и Садакане). В этой программе реализована только суффиксная сортировка, немного улучшенная по сравнению с описанной выше.

Для эксперимента использовались следующие файлы:

- book1 из тестового набора "Calgary Corpus", как файл, обладающий основными свойствами типичных текстов. Размер файла – 768 771 байт;
- file2 (1 000 000 байт). Этот файл был составлен из нескольких больших одинаковых частей, позаимствованных из файла book1. Был сконструирован для проверки умения алгоритмов сортировки упорядочивать длинные одинаковые строки;
- wat.c (1 890 501 байт). Файл исходных текстов, полученный путем слияния исходных текстов, поставляемых с дистрибутивом компилятора Watcom C 10.0. Как уже отмечалась выше, средняя длина устойчивого контекста у таких файлов немного выше, чем у типичных текстов;
- kennedy.xls (1 029 744 байта). Данный файл использовался для анализа способности сортировщиков обрабатывать большое количество одинаковых строк небольшой длины.

	book1	file2	wat_c	kennedy
QSuf *	3.30	9.23	10.00	4.23
YBS	3.13	4.77	6.76	4.15
DC	2.36	4:23.59	6.92	2.97
ARC	4.17	4.34	6.43	5.00
BA	4.45	5.82	7.36	4.73
bzip2 **	3.03	4.23	6.81	4.07

** В программе QSuf реализована только сортировка, без сжатия и записи информации в файл. Это необходимо учитывать при сравнении быстродействия. Например, архиватор YBS потратил на сжатие преобразованного файла book1 примерно 1 с.*

*** Архиватор bzip2 приведен только для справки, так как в нем реализовано сжатие на основе метода Хаффмана, а в остальных используется арифметическое, которое заметно медленнее, но дает существенно лучшее сжатие (на 5–10 %). Кроме того, максимальный размер блока, который позволяет использовать bzip2, – 900 Кб, что не позволяет достичь должного сжатия всех файлов, кроме book1. Хотя и дает небольшой прирост в скорости (2–3 % на файле wat.c). Увеличение размера блока до размера файла могло бы замедлить скорость сжатия файла wat.c на 2–3 %.*

По результатам эксперимента можно сделать наблюдения:

1. Представитель алгоритмов, реализующих сортировку суффиксов, вел себя довольно ровно, хотя на типичных файлах оказался на последнем-предпоследнем месте. Как уже отмечалось, суффиксная сортировка хороша, но уж больно велики накладные расходы.
2. DC провалился именно там, где ожидалось, – на длинных совпадени... Архиваторы, использующие быструю сортировку, потенциально м... отстать от конкурентов на данных с большим количеством лекси... фически упорядоченных совпадений. В принципе можно успе... роться и с теми и с другими слабостями. Но можно предпол... на типичных файлах скорее всего сортировка слиянием оста... на коротких контекстах, а сортировка Бентли – Седжвика ... (что видно на примере файла wat.c).
3. Можно было рассмотреть частичное сортирующее преобр... честве альтернативного преобразования, не требующего та... при сортировке. Но здесь уже появляются другие требов... ковке – бóльшие затраты памяти и времени. И как правил... шее сжатие.

Хотя сортировка суффиксов и не зависит от избыточно... типичных данных методы, использующие быструю сорти... ровку слиянием, оказываются быстрее.

Архиваторы, использующие BWT и ST

Довольно быстро после опубликования статьи Барро... появляться первые компрессоры. Это объясняется, во-... метод оказался хорошим компромиссом между быс... использующими словарное сжатие, и медленными п... стическими компрессорами. Во-вторых, авторы сра... шают некоммерческое использова...

С тех пор количество программ, использующих ... уза – Уилера, непрерывно растет. Ниже приведена ... мянуты наиболее интересные из них.

Компрессор и версия	Даты	Автор	
BRed*	06.1997	D.J. Wheeler	ftp://ftp.cl.cam...
X1 –m7 0.95	05.1997	Stig Valentini	mailto:x1develope... http://www.saunalah...
BWC 0.99	01.1999	Willem Monsuwe	mailto:willem@stack.n... ftp://ftp.stack.nl/pub/us...

Компрессор и версия	Даты	Автор	Адреса
IMP -2 1.12 WinImp 1.21	01.2000 09.2000	Conor McCarthy	mailto:imp@technelysium.com.au http://www.technelysium.com.au/ http://www.winimp.com
szip 1.12	03.2000	Michael Schindler	mailto:michael@compressconsult.com http://www.compressconsult.com/
bzip2 1.01 (bzip 0.21)	06.2000	Julian Seward	mailto:jseward@acm.org http://sourceware.cygnus.com/bzip2
DC 0.99.298b	08.2000	Edgar Binder	mailto:EdgarBinder@t-online.de ftp://ftp.elf.stuba.sk/pub/pc/pack
YBS 0.03e	09.2000	Vadim Yoockin	mailto:yoockinv@mtu-net.ru mailto:vy@thermosyn.com http:// compression.graphicon.~/ybs
BA 1.01br5	10.2000	Mikael Lundqvist	mailto:mikael@2.sbbs.se http://hem.spray.se/mikael.lundqvist
Zzip 0.36c	06.2001	Damien Debin	mailto:damien.debin@via.ecp.fr http://www.zzip.f2s.com/
SBC 0.910b	11.2001	Sami Makinen	mailto:sjm@pp.inet.fi http://www.geocities.com/sbcarchiver
ERI 5.0re	12.2001	Alexander Ratushnyak	mailto:artest@inbox.ru http://geocities.com/eri32
GCA 0.9k	12.2001	Shin-ichi Tsuruta	mailto:synsyr@pop21.odn.ne.jp http://www1.odn.ne.jp/~synsyr/
7-Zip 2.30b12	01.2002	Igor Pavlov	Mailto: support@7-zip.org Http://www.7-zip.org

Семейство программ BRed, BRed и BRed3 написано одним из родоначальников BWT – Дэвидом Уилером. Многие идеи, использованные в этих компрессорах, описаны в работах Петера Фенвика [18] и нашли свое применение в ряде других программ, например в X1.

Bzip использует адаптированную к BWT сортировку Бентли – Седжвика, во многом позаимствованную из вышеупомянутого семейства. После BWT выполняется преобразование методом стопки книг, выходные данные которого сжимаются при помощи интервального кодирования (аналог арифметического сжатия) с использованием 1–2-кодирования и структурной модели Петера Фенвика.

Для того чтобы создать программу, которую можно свободно использовать в некоммерческих целях, в bzip2 интервальное кодирование было заменено на сжатие по методу Хаффмана. Видимо, благодаря этому bzip2 находит все большее распространение в различных областях применения и де-факто уже становится одним из стандартов. Сортировка в bzip2 изменена

незначительно по сравнению с bzip. В основном повышена устойчивость к избыточным данным и оптимизирован ряд процедур.

В BWC используются такие же методы, что и bzip и bzip2. А именно оптимизированная сортировка, MTF, 1–2-кодирование и интервальное кодирование.

IMP использует собственную сортировку, очень быструю на обычных текстах, но буквально "зависающую" на данных, в которых встречаются длинные одинаковые последовательности символов. Сжатие полностью позаимствовано из bzip2.

Алгоритм сжатия, используемый в bzip2, также включен и в архиватор 7-Zip.

В szip, помимо упоминавшегося частичного сортирующего преобразования, реализована и возможность использования BWT. Реализована, прямо скажем, только для примера, без затей. А вот для сжатия используются очень интересные решения, представляющие собой некий гибрид MTF-преобразования и адаптивного кодера, берущий статистику из короткого окна преобразованных с помощью BWT данных. С участием автора szip и с использованием описанных решений был также создан архиватор ICT UC.

В Zzip применяются все те же испытанные временем структурная модель, сортировка Бентли – Седжвика и кодирование диапазонов.

BA использует аналогичную сортировку. Но повышение устойчивости реализовано в BA другим способом. Деление строк по ключу прекращается в том случае, когда оказывается, что этим строкам предшествуют одинаковые символы. Еще одно новшество, реализованное в BA, – это выбор структурной модели MTF в отдельном проходе. Также за счет динамического определения размера блока улучшено сжатие неоднородных файлов. Для усиления сжатия английских текстов используется переупорядочение алфавита.

В DC впервые реализован целый ряд новаторских идей. Во-первых, конечно, это модель сжатия, отличная от MTF, – кодирование расстояний. Во-вторых, новый метод сортировки, очень быстрый на текстах, хотя и дающий слабину на сильно избыточных данных. И наконец, большой набор методов препроцессинга текстовых данных, позволяющий добиться особенного успеха на английских текстах.

Отличительная особенность SBC – наличие мощной криптосистемы. Ни в одном из архиваторов, пожалуй, не реализовано столько алгоритмов шифрования, как в SBC. В SBC используется алгоритм BWT, ориентированный на большие блоки избыточных данных и позволяющий очень быстро сортировать данные с большим количеством длинных похожих строк. Вместо

MTF в архиваторе используется кодирование расстояний, хотя пока не так эффективно, как в YBS и DC, но это компенсируется большим количеством фильтров (методов препроцессинга), настроенных на определенные типы данных.

ARC (автор – Ian Sutton, которому также принадлежит PPM-архиватор BOA). Как и многие другие, использует BWT на основе сортировки Бентли – Седжвика и MTF. Как и в SBC, дополнительно отслеживаются очень длинные повторы данных.

Первые версии YBS также использовали перемещение стопки книг, которое затем было заменено на кодирование расстояний. Что дало заметный выигрыш в степени сжатия.

Среди не распространяемых свободно компрессоров, описание которых опубликовано в научных трудах, можно отметить BKS98 и BKS99, которые принадлежат сразу трем авторам [10]. Эти компрессоры используют суффиксную сортировку и многоконтекстовую модель MTF по трем последним кодам.

Сравнение BWT-архиваторов

Параметры, используемые для указания режима работы архиваторов, выбраны таким образом, чтобы добиться наилучших результатов в сжатии без особого ухудшения скорости.

Тестирование производилось на компьютере со следующей конфигурацией:

Процессор	Intel Pentium III 840 МГц
Частота шины	140 МГц
Оперативная память	256 Мб SDRAM
Жесткий диск	Quantum FB 4.3 Гб
Операционная система	Windows NT 4.0 Service Pack 3

Начнем со сжатия русских текстов, потому что BWT-архиваторы особенно эффективны именно для сжатия текстов. А русские тексты выбраны для того, чтобы показать эффективность сжатия в чистом виде, без использования текстовых фильтров, которые для русских текстов еще не созданы авторами описываемых программ. Файл имеет длину 1,639,139 байт.

Архиватор, версия и параметры	Размер сжатого файла, байт	Время сжатия, с	Время разжатия, с
YBS 0.03e	446,151	1.81	0.93
DC 0.99.298b -a	449,403	1.21	1.00
SBC 0.860	451,240	1.69	0.87
ARC (I.Sutton) b20	459,409	2.08	1.37
Compressia' b2048	462,873	2.92	2.66

Архиватор, версия и параметры	Размер сжатого файла, байт	Время сжатия, с	Время разжатия, с
BA 1.01b5 -24-m	463,214	2.17	1.26
Zzip 0.36 -mx -b8	467,383	1.96	1.65
szip 1.12 b21 o0	470,894	3.34	0.78
ICT UC 1.0	472,556	2.54	1.27
szip 1.12 b21 o8	472,577	2.32	1.12
GCA 0.90g -v	477,999	2.17	1.17
BWC/PGCC 0.99 m2m	479,162	1.69	0.83
BWC/PGCC 0.99 m900k	503,556	1.56	0.83
szip 1.12 b21 o4	506,348	0.48	0.94
IMP 1.10 -2 u1000	506,524	1.07	0.64
bzip2/PGCC 1.0b7 -9	507,828	1.55	0.66

Как можно заметить, первенство удерживают компрессоры, использующие кодирование расстояний.

На английском тексте (2 042 760 байт) некоторые архиваторы используют фильтры, тем самым заметно улучшая сжатие. Ниже приведены результаты, принадлежащие тем программам, которые показали наилучшие результаты в первом тесте.

Архиватор, версия и параметры	Размер сжатого файла, байт	Время сжатия, с	Время разжатия, с	Использование фильтров
DC 0.99.298b	476,215	1.58	1.28	+
SBC 0.860 b3m1	489,612	1.59	0.96	+
YBS 0.03e	496,703	2.32	1.09	
DC 0.99.298b -a	500,421	1.50	1.18	
ARC (I.Sutton) b20	508,737	2.62	1.71	
BA 1.01b5 -24	512,696	2.87	1.53	+
Zzip 0.36 -mx -b8	515,672	2.84	2.08	+
Compressia b2048	517,484	3.67	2.12	
BA 1.01b5 -24-x	517,626	2.75	1.42	

При сжатии данных, представляющих собой исходные тексты программ, распределение среди лидеров тестов практически не меняется.

Для иллюстрации поведения BWT-архиваторов на неоднородных данных применен исполнимый модуль из дистрибутива компилятора Watcom 10.0 wcc386.exe (536,624 байта). Для того чтобы можно было судить об эффективности различных методов и режимов, некоторые строки помечены специальными знаками:

Архиватор, версия и параметры	Размер сжатого файла, байт	Время сжатия, с	Время разжатия, с	Фильтры	Уменьшенный размер блока	Автоматическое определение размера блока
YBS 0.03e -m512k	275,396	0.66	0.51	+	+	
YBS 0.03e	276,035	0.66	0.57	+		
SBC 0.860 m3a	278,061	0.98	0.69	+		+
ARC b5mm1	278,392	1.33	0.48	+	+	
DC 0.99.298b -b512	279,424	0.67	0.36	+	+	
DC 0.99.298b	279,759	0.66	0.37	+		
ARC mm1	280,052	1.34	0.46	+		
Zzip 0.36 -mx	291,199	0.74	0.66			
ARC (I.Sutton) b5	291,345	0.58	0.48		+	
ARC (I.Sutton)	292,979	0.58	0.48			
BA 1.01b5 -24-z	293,489	0.82	0.64			+
DC 0.99.298b -a	293,807	0.52	0.39			
IMP 1.10 -2 u1000	294,679	0.38	0.18	+		
Compressia b512	297,647	0.97	1.16			
ICT UC 1.0	298,348	0.75	0.53			
BA 1.01b5	298,617	0.82	0.66			
szip 1.12 b21 o0	298,668	0.76	0.31			
szip 1.12 b21 o4	299,249	0.27	0.39			
BWC/PGCC 0.99 m600k	304,996	0.58	0.37			
bzip2/PGCC 1.0b7 -6	308,624	0.63	0.26			

В качестве файла, содержащего смесь текстовых и бинарных данных, использовался Fileware.doc размером 427,520 байт из поставки русского MS Office'95. Данный пример показывает, что иногда модель, использующая MTF, оказывается достаточно эффективной.

Архиватор, версия и параметры	Размер сжатого файла, байт	Время сжатия, с	Время раз-жатия, с	Фильтры	Уменьшенный размер блока	Автоматическое определение размера блока
SBC 0.860 m3a	126,811	0.69	0.42	+		+
DC 0.99.298b	127,377	0.38	0.18	+		
ARC b2	128,685	0.38	0.23	+	+	
YBS 0.03e -m256k	130,356	0.37	0.24		+	
Compressia b256	131,737	0.61	0.40		+	
BA 1.01b5 -24-r	132,651	0.41	0.30			
Zzip 0.36 -a1	132,711	0.65	0.40			
DC 0.99.298b -a	133,825	0.34	0.23			
YBS 0.03e	133,915	0.37	0.25			
BWC/PGCC 0.99 m600k	134,183	0.33	0.19		+	
bzip2/ PGCC 1.0b7 -6	134,932	0.44	0.14		+	
szip 1.12 b21 o0	134,945	0.90	0.15			
IMP 1.10 -2 u1000	135,431	0.30	0.12			
ICT UC 1.0	136,842	0.41	0.29			
BA 1.01b5 -24-z	137,566	0.49	0.31			+
szip 1.12 b21 o4	141,784	0.17	0.18			

Заключение

Несмотря на то что преобразование Барроуза – Уилера было опубликовано сравнительно недавно, оно пользуется большим вниманием со стороны разработчиков архиваторов. И пожалуй, еще впереди новые исследования, которые позволят повысить эффективность сжатия на основе BWT еще в большей степени.

Можно отметить характерные особенности архиваторов, использующих описанное преобразование, по сравнению с другими.

Скорость сжатия – на уровне архиваторов, применяющих словарные методы, например LZ77. Разжатие, как правило, в 3–4 раза быстрее сжатия. Степень сжатия сильно зависит от типа данных.

Наиболее эффективно применение BWT-архиваторов для текстов и любых данных со стабильными контекстами. В этом случае рассматриваемые компрессоры по своим характеристикам близки к программам, использующим PPM. На неоднородных данных существующие архиваторы на основе BWT немного уступают лучшим современным компрессорам, использую-

щим словарные методы или PPM. Впрочем, существуют способы компенсировать этот недостаток.

Расходы памяти в режиме максимального сжатия довольно близки у всех современных архиваторов. Наибольшее отличие наблюдается при декодировании. Наиболее скромными в этом отношении являются архиваторы, использующие алгоритмы семейства LZ77, а наиболее расточительными – PPM-компрессоры, требующие столько же ресурсов, сколько им нужно при сжатии. Архиваторы на основе BWT занимают промежуточное положение.

Литература

1. Кадач А. В. Эффективные алгоритмы неискажающего сжатия текстовой информации: Дис. к. ф-м. н. – Ин-т систем информатики им. А. П. Ершова. М., 1997.
2. Albers S., v.Stengel B., Werchner R. A combined BIT and TIMESTAMP Algorithm for the List Update Problem. 1995.
3. Ambuhl C., Gartnerm B., v.Stengel B. A New Lower Bound for the List Update Problem in the Partial Cost Model. 1999.
4. Arimura M., Yamamoto H. Asymptotic Optimality of the Block Sorting Data Compression Algorithm. 1998.
5. Arnavaut Z., Magliveras S. S. Block Sorting and Compression // Proceedings of Data Compression Conference. 1999.
6. Arnavaut Z., Magliveras S. S. Lexical Permutation Sorting Algorithm.
7. Baik H-K., Ha D.S., Yook H-G., Shin S-C., Park M-S. A New Method to Improve the Performance of JPEG Entropy Coding Using Burrows-Wheeler Transformation. 1999.
8. Baik H., Ha D. S., Yook H-G., Shin S-C., Park M-S. Selective Application of Burrows-Wheeler Transformation for Enhancement of JPEG Entropy Coding. 1999.
9. Balkenhol B., Kurtz S. Universal Data Compression Based on the Burrows and Wheeler-Transformation: Theory and Practice.
10. Balkenhol B., Kurtz S., Shtarkov Y.M. Modifications of the Burrows and Wheeler Data Compression Algorithm // Proceedings of Data Compression Conference. Snowbird: Utah, IEEE Computer Society Press, 1999. P. 188–197.
11. Balkenhol B., Shtarkov Y.M. One attempt of a compression algorithm using the BWT.
12. Baron D., Bresler Y. Tree Source Identification with the BWT. 2000.
13. Burrows M., Wheeler D.J. A Block-sorting Lossless Data Compression Algorithm // SRC Research Report 124, Digital Systems Research Center, Palo Alto, May 1994. http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/SRC-124.ps.Z.

14. Chapin B. Switching Between Two On-line List Update algorithms for Higher Compression of Burrows-Wheeker Transformed Data // Proceedings of Data Compression Conference. 2000.
15. Chapin B., Tate S. Higher Compression from the Burrows-Wheeler Transform by Modified Sorting. 2000.
16. Deorowicz S. An analysis of second step algorithms in the Burrows-Wheeler compression algorithm. 2000.
17. Deorowicz S. Improvements to Burrows-Wheeler Compression Algorithm. 2000.
18. Fenwick P. M. Block sorting text compression // Australasian Computer Science Conference, ACSC'96, Melbourne, Australia, Feb 1996. ftp://ftp.cs.auckland.ac.nz/out/peter-f/ACSC96.ps.
19. Ferragina P., Manzini G. An experimental study of an opportunistic index. 2001.
20. Kruse H., Mukherjee A. Improve Text Compression Ratios with Burrows-Wheeler Transform. 1999.
21. Kurtz S. Reducing the Space Requirement of Suffix Trees.
22. Kurtz S. Space efficient linear time computation of the Burrows and Wheeler Transformation // Proceedings of Data Compression Conference. 2000.
23. Kurtz S., Giegerich R., Stoye J. Efficient Implementation of Lazy Suffix Trees. 1999.
24. Larsson J. Attack of the Mutant Suffix Tree.
25. Larsson J. The Context Trees of Block Sorting Compression.
26. Larsson J., Sadakane K. Faster Suffix Sorting.
27. Manzini G. The Burrows-Wheeler Transform: Theory and Practice. 1999.
28. Nelson P.M. Data Compression with the Burrows Wheeler Transform // Dr. Dobbs Journal. Sept. 1996. P 46–50. http://web2.airmail.net/markn/articles/bwt/bwt.htm.
29. Sadakane K. A Fast Algorithm for Making Suffix Arrays and for BWT.
30. Sadakane K. Comparison among Suffix Array Constructions Algorithms.
31. Sadakane K. On Optimality of Variants of Block-Sorting Compression.
32. Sadakane K. Text Compression using Recency Rank with Context and Relation to Context Sorting, Block Sorting and PPM.
33. Schindler M. A Fast Block-sorting Algorithm for lossless Data Compression // Vienna University of Technology. 1997.
34. Schulz F. Two New Families of List Update Algorithms. 1998.
35. Ryabko B. Ya. Data Compression by Means of a "Book Stack"// Problems of Information Transmission. Vol. 16(4). 1980. P. 265–269.

Глава 6. Обобщенные методы сортирующих преобразований

Сортировка параллельных блоков

Английское название метода – Parallel Blocks Sorting (PBS).

Два блока A и B называются параллельными, если каждому элементу A[i] первого блока поставлен в соответствие один элемент B[i] второго блока и наоборот. Длины блоков L_A и L_B равны: $L_A = L_B = L$. Размеры элементов блоков R_A и R_B могут быть разными.

Основная идея метода PBS состоит в сортировке элементов In[i] входного блока In и их раскладывании в несколько выходных блоков Out_j на основании атрибутов A[i] этих элементов. Атрибут A[i] есть значение функции A, определяемой значениями предшествующих элементов In[j] и/или элементов P[k] из параллельного блока P:

```
A[i]=A(i, In[j], P[k]), i=0...L-1; j=0,...,i-1; k=0,...,L-1
```

При декодировании осуществляется обратное преобразование: элементы из нескольких блоков Out_j собираются в один результирующий, соответствующий несжатому блоку In (рис. 6.1).

Чем лучше значения In[i] предсказуемы по значениям A[i], тем эффективнее последующее сжатие блоков Out_j с помощью простых универсальных методов.

Методы этой группы **преобразующие** и **блочные**, т. е. могут применяться только в том случае, когда известна длина блока с данными.

Размер данных в результате применения PBS, как и при ST/BWT, не изменяется. Для сжатия результата работы метода может быть применена любая комбинация методов – RLE, LPC, MTF, DC, HUFF, ARIC, ENUC, SEM...

В общем случае скорость V_C работы компрессора (реализующего прямое, "сжимающее" преобразование) равна скорости V_D декомпрессора (реализующего обратное, "разжимающее" преобразование) и зависит только от размера данных, но не от их содержания: $V_C = V_D = O(L)$. Памяти требуется $2 \cdot L + C$. Константа C определяется особенностями реализации и может быть довольно большой. Операций умножения и деления нет.

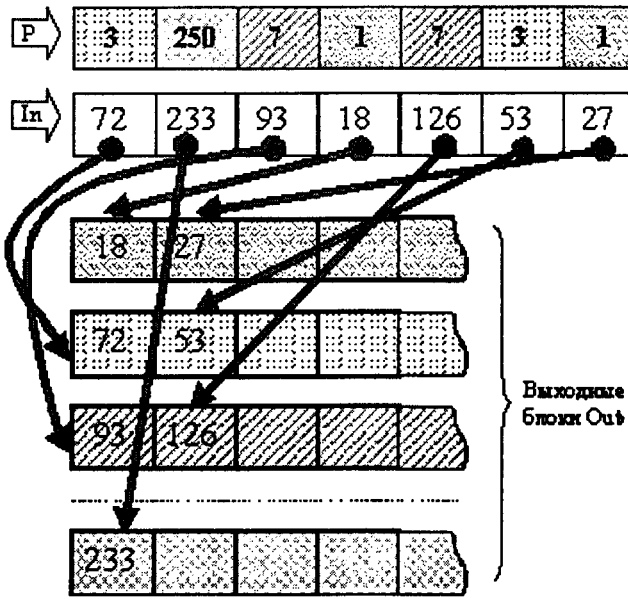


Рис. 6.1. Иллюстрация PBS: атрибут A[i] определяется значением элемента P[i]; значение A[i] показано штриховкой

Из краткого описания общей идеи видно, что

- для распаковки нужно иметь не только содержимое сортированных блоков Out_j, но и их размеры;
- параллельных блоков может быть и два, и больше;
- если функция A не использует те P[k], которые находятся после текущей позиции i, т. е. k=i+1, ..., L , то можно создавать параллельный блок P одновременно с текущим сортируемым In;
- если A задана так, что принимает мало значений: от 2 до, допустим, 16, метод вполне может быть применим и к потокам данных;
- если параллельного блока P нет, получаем ST/BWT как частный случай PBS.

Основной алгоритм преобразования

В простейшем случае сортирующего преобразования ST(1) значение атрибута вычисляется так: A[i]=In[i-1]; при ST(2): A[i]=In[i-1]·2^R + In[i-2] и т. д.

В простейшем случае PBS A[i]=P[i], т. е. значение атрибута равно значению элемента из параллельного блока. Выходных блоков столько, сколько значений принимают P[i]. Делая проход по P, считаем частоты значений атрибута и в ре-

зультате получаем размеры сортированных блоков (далее – контейнеров), в которые будем раскладывать элементы из входного блока In:

```
for( a=0; a<n; a++) Attr[a]=0;     //инициализация    (1)
for( i=0; i<L; i++) Attr[P[i]]++; //подсчет частот   (2)
```

In – входной сортируемый блок длины L;

P – параллельный блок длины L;

L – количество элементов во входном блоке;

$n=2^R$ – число всех возможных значений атрибута;

Attr[n] – частоты значений атрибута.

Например, в P содержатся старшие 8 бит массива 16-битовых чисел, а в In – младшие 8. Заметим, что этот процесс "дробления" можно продолжать и дальше, создавая вплоть до 16 параллельных блоков: содержащий первые биты, вторые, и т. д. Кроме того, при сжатии мультимедийных данных часто уже имеются параллельные блоки, например левый и правый каналы стереозвука, красная, зеленая и синяя компоненты изображения и т. п.

Если функция A задана иначе, то и при подсчете частот формула будет иной:

```
// A[i]=P[i]&252 :
for( i=0; i<L; i++) Attr[ P[i]&252 ]++;// (2a)

// A[i]=255-P[i] :
for( i=0; i<L; i++) Attr[ 255-P[i] ]++;// (2b)

// A[i]=(P[i]+P[i-1])/2 :
Attr[ P[0] ]++;          // (2c)
for( i=1; i<L; i++) Attr[ (P[i]+P[i-1])/2 ]++;
```

Итак, после двух циклов – инициализации и подсчета частот – мы получили в массиве Attr длины контейнеров (сортированных блоков). Теперь можно вычислить, где какой контейнер будет начинаться, если их последовательно сцепить в один блок в порядке возрастания значения атрибута:

```
for( a=0, pos=0; a<n; a++) {    //          (3)
  tmp=Attr[a]; // длина текущего a-го контейнера
  Attr[a]=pos;  // теперь там адрес-указатель на его начало
                // (указатель на начало свободного места в нем)
  pos+=tmp;   // начало следующего - дальше на длину текущего
}
```

В том же массиве Attr[n] теперь адреса-указатели на начала контейнеров. Остается только пройти по входному блоку, раскладывая элементы из него по контейнерам:

```
for(i=0; i<L; i++) Out[Attr[P[i]]++]=In[i]; //(4c)
```

Out – выходной отсортированный блок (sorted, transformed), содержащий все контейнеры. Подробнее:

```
for( i=0; i<L; i++) { // На каждом шаге:
 s=In[i];// берем следующий элемент s из входного блока In,
  a=P[i];// берем его атрибут a из параллельного блока P,
// и адрес свободного места в контейнере, задаваемом этим a;
  pos=Attr[a];
// кладем элемент s в выходной блок Out по этому адресу,
  Out[pos]=s;
  pos++;  //теперь в этом контейнере на один элемент больше,
  Attr[a]=pos;//a свободное место - на один элемент дальше.
}
```

Функция A – та же, что и во втором цикле. То есть, для выше рассмотренных примеров:

```
(2a) a=P[i]&252;
(2b) a=255-P[i];
(2c) a=(P[i]+P[i-1])/2;
       //причем i=1...L-1, а когда i=0, a=P[0].
```

Обратное преобразование

Совершенно идентичны первые 3 цикла: инициализируем, в результате прохода по параллельному блоку находим длины контейнеров, затем определяем адреса начал контейнеров во входном отсортированном блоке In. Именно эти адреса – цель выполнения первых трех циклов. И только в четвертом цикле, наоборот, берем из отсортированного блока с контейнерами In, кладем в единый выходной блок Out:

```
for(i=0;i<L; i++) Out[i]=In[Attr[P[i]]++];   //(4d)
```

In – входной отсортированный блок со всеми контейнерами;

Out – создаваемый выходной блок.

Подробнее:

```
for( i=0; i<L; i++) {
  a=P[i];
  pos=Attr[a];
//так было при прямом (сжимающем) преобразовании:
//Out[pos]=In[i];
        //(в текущих обозначениях это записывается так:
        //In[pos]=Out[i];)
//а так делаем сейчас, при разжимающем преобразовании:
  Out[i]=In[pos];
  pos++;
  Attr[a]=pos;
```

```
}
```

Пути увеличения скорости сжатия

Итак, в обоих случаях выполняем два цикла от 0 до $n=2^R$ и два цикла от 0 до L. Какой из них займет больше времени? Чаще всего R=8, n=256, а L – от нескольких килобайт до десятков мегабайт. И четвертый, главный цикл – самый долгий, второй выполняется быстрее, а третий и особенно первый – совсем быстро.

Но возможна и обратная ситуация: если, например, R=16, $n=2^{16}$=65536, а L=512, т. е. требуется сжать поток блоков из 512 16-битовых элементов (поток "фреймов"). Тогда, наоборот, самым долгим будет третий цикл, затем первый, четвертый и второй.

Два цикла при больших R

Если памяти хватает, поступим так: будем наполнять контейнеры не одновременно, проходя по In, P и записывая элементы In[i] в вычисляемые адреса блока Out, а последовательно: проходя по Out и вычисляя адреса в In, P, из которых нужно брать значения атрибута и элементы, помещаемые затем в Out. Останутся два цикла от 0 до L, а циклы от 0 до $n=2^R$ исчезнут.

При первом проходе заполняем вспомогательный массив Anext, содержащий цепочки указателей на позиции с одинаковыми атрибутами (A-цепочки). То есть в Anext[i] записываем указатель на следующую позицию (i+k) с таким же значением атрибута A[i], если таковая (i+k) в оставшейся части блока есть. В противоположном случае, если в оставшейся части блока атрибут не принимает значение A[i], в Anext[i] записываем i.

```
// при инициализации: вспомогательные массивы
int Anext[L];
int Beg[n];  // начала,
int End[n];  // концы и
int Flg[n];  // флаги наличия A-цепочек в текущем фрейме

// в цикле для каждого фрейма:
FrameNumber++;          // следующий L-элементный фрейм
for( i=0; i<L; i++) {
a=P[i];                 //вычислим значение атрибута a
if(Flg[a]==FrameNumber) //если цепочка, определяемая этим a,
  {          // в текущем фрейме уже есть,
  e=End[a];  // то конец ее - в массиве концов цепочек
  Anext[e]=i;// теперь прошлый конец указывает на новый
  }
else
  {  // иначе
  Flg[a]=FrameNumber;// запишем, что такая цепочка есть
```

```
  Beg[a]=i;           // и сохраним адрес ее начала
  }
Anext[i]=i;  // текущий элемент A-цепочки - ее конец
End[a]=i;    // укажем конец в массиве концов цепочек
}
```

Массив Flg нужен только затем, чтобы инициализировать не Beg и End перед *каждым* фреймом, а только Flg перед *всеми* фреймами. Если создать еще два массива Fprev и Fnext, соединяющие все элементы Flg с одним значением в одну F-цепочку (причем Fnext указывает на следующий элемент F-цепочки, Fprev – на предыдущий), то не придется при втором проходе линейно искать в Flg появлявшиеся в текущем фрейме A-цепочки. И цикла от 0 до *n* удастся избежать. Соответствующие изменения читатель легко внесет в алгоритм сам.

При втором проходе выполняем сортировку:

```
pos=f=0;
while (Fnext[f]!=f) {   // (линейный поиск следующей
  f=Fnext[f];           // A-цепочки в Flg, если без Fnext)
  i=Beg[f];             // первый элемент текущей A-цепочки,
  do {                  // и далее для всех ее элементов:
    Out[pos]=In[i];     // берем из In, кладем в Out
    pos++;              // последовательно,
    i=Anext[i];         //и т. д., проходя всю A-цепочку
  } while(i!=Anext[i])  //до конца
}
```

На этот раз во втором цикле уже нет обращений к P, поэтому можно совместить A и P. Памяти же требуется не

$2 \cdot L + n \cdot$sizeof(*char), а

$2 \cdot L + 5 \cdot n \cdot$sizeof(*char),

так как Beg, End, Flg, Fprv, Fnxt занимают по $n \cdot$sizeof(*char) байт.

Два цикла при малых R

Если памяти доступно больше, чем необходимо для хранения $2 \cdot L + n \cdot C$ элементов, и C достаточно велико – 256 или, например, 512, можно сразу "расформатировать" память под начала контейнеров:

```
#define K 256;
for( i=0; i<n; i++) Attr[i]=i*K;        // (1sm)
// под каждый из n контейнеров отведено K элементов
//  (1 "сектор")
```

И затем при проходе по In, P, если свободное место в каком-то контейнере кончается, будем записывать в конце "сектора" указатель на следующий сектор, выделяемый под продолжение этого контейнера:

```
FreeSector=n;
for( i=0; i<L; i++) {          // (2sm)
  a=P[i];  // берем атрибут a из параллельного блока P,
  pos=Attr[a];// и адрес свободного места в контейнере,
              // задаваемом этим a;
  Out[pos]=In[i];
  pos++;
if(pos%K+sizeof(*char)==K)//если свободное место кончилось,
  {
  Out[pos]=FreeSector;//пишем указатель на следующий сектор
     // если FreeSector типа int32, а Out[i] типа int8, то:
     // Out[pos] =FreeSector%256;
     // Out[pos+1]=(FreeSector>>8)%256;
     // Out[pos+2]=(FreeSector>>16)%256;
     // Out[pos+3]=(FreeSector>>24)%256;
  pos=FreeSector*K;//свободное место= начало нового сектора
    FreeSector++; // свободный сектор на один сектор дальше
    }
  Attr[a]=pos;
}
```

Упражнение. *Сравните этот цикл с (4c) и оцените, насколько он будет* медленнее.

Это очень выгодно, если $n \leq 256$, и, кроме того, после PBS выполняется еще один проход по всем элементам блока Out. Как правило, это так. Таким образом, весь алгоритм сводится к одному тривиальному циклу (1sm) и одному проходу по данным (2sm).

Увеличение скорости разжатия

Ускорение разжатия возможно, если длины контейнеров сжимать и передавать отдельно. Если L существенно больше n, описание длин занимает незначительную часть сжатого блока, так что потери в степени сжатия будут невелики. Тогда в массив Attr[n] при разжатии можно сразу записывать не длины, а начала контейнеров. Заметим, что и в случае ST/BWT это даст заметное увеличение скорости: подсчет частот элементов требует при разжатии дополнительного прохода по данным.

Пути улучшения сжатия

Общий случай

Если содержимое каждого контейнера (или некоторых из них) подвергается преобразованию, зависящему от его атрибута, то длины контейнеров надо обязательно передавать декомпрессору, иначе он неправильно построит таблицу частот по содержимому контейнеров.

Если параллельный блок уже существует и не изменяется при прямом и обратном преобразовании, функцию A можно строить без всяких ограничений, используя любую комбинацию арифметических, логических и других операций. Например,

```
A[i]=(int)P[i]/2; // деление целочисленное
```

Если сравнивать со случаем A[*i*] = P[*i*], то можно сказать, что каждая пара контейнеров объединяется в один "двойной" контейнер.

```
A[i]=P[i]^(n-1); // или, что то же самое, A[i]=n-1-P[i].
```

То есть контейнеры будут лежать в Out в обратном порядке по сравнению с A[*i*]=P[*i*]. Это полезно для второго отсортированного блока Out2, записываемого за первым Out1, особенно если данные мультимедийные, поскольку в этом случае значения In[*i*] и In[*i*+1], как правило, близки.

Еще два варианта вычисления атрибута:

```
A[i]=(P[i] & 16 == 0)? P[i] : (P[i]^15);
```

Младшие 4 бита изменяются только на +1 или –1:
abs(P[i]&15 – P[i–1]&15)=1.

```
A[i]=(P[i]<0)? (-P[i]*2+1 ) : P[i]*2;
```

Расстояние до нуля, а знак – в младшем бите.

Упражнение. Напишите к последним двум формулам формулы обратного преобразования.

Функция A может быть и адаптивной, т. е. производится периодическая корректировка каких-то ее коэффициентов, чтобы выходной блок больше соответствовал заданному критерию. Например, известно, что последние 20–30 контейнеров из 256 полезно объединять в четверки, а остальные – в пары:

```
// последние 256-232=24 контейнера объединяем в четверки
K=232;
A[i]=(P[i]>K)? P[i]/4 : P[i]/2; // деление - целочисленное
```

При этом критерий используется такой: сумма первой производной выходного блока Out должна стремиться к нулю:

$$Sum = \sum_{i=1}^{L-1} (Out[i] - Out[i-1]) = 0,$$

где Out[*i*]–Out[*i*–1] – первая производная блока.

Тогда можно вычислять

```
SumCurrent+=Out[i]-Out[i-1], при текущем значении K,
SumMinus+=Outm[i]-Outm[i-1] при K=K-4,
SumPlus+=Outp[i]-Outp[i-1] при K=K+4,
```

и через каждые *StepK* шагов выбирать новое значение *K* по результату сравнения *SumCurrent*, *SumMinus* и *SumPlus*:

```
if (SumMinus<SumCurrent && SumMinus<SumPlus) K=K-4;
else if (SumPlus<SumCurrent) K=K+4;
```

Может оказаться полезным сохранять блок A вместо P, особенно если параллельный блок строится одновременно с сортируемым, например, P[*i*] вычисляется по P[*i*-1] и In[*i*-1]. Тогда P должен быть восстановим по A и функция A должна быть биективной: по значению A[*i*]=A(P[*i*]) однозначно находится P[*i*]. Реально такая функция A сводится к перестановке: имеем контейнеры с атрибутами A[*i*]=P[*i*], затем перетасовываем их, меняя местами внутри единого выходного блока Out, но не изменяя их содержимого. Но формально A может выглядеть очень по-разному, содержать прибавление константы:

```
A[i]=P[i]+123; // результат берется по модулю n
```

логический XOR:

```
A[i]=P[i]^157;
```

сравнения и перестановки битов, в том числе циклическим вращением:

```
A[i]=(P[i]&(n/2))? ( (P[i]-(n/2))*2+1 ) : P[i]*2;
```

Упражнение. Как будет выглядеть A, собирающая вместе элементы In[*i*], соответствующие тем P[*i*], остаток которых от деления на заданное *K* одинаков? Начните со случаев *K*=4 и *K*=8.

Если функция A неадаптивна или зависит только от элементов параллельного блока, сами контейнеры, поскольку их длины известны, можно сортировать внутри единого выходного блока по их длинам. Это может улучшить сжатие даже при ST/BWT, особенно в случае качественных данных, а не количественных. Часто оказывается выгодным группировать короткие контейнеры в одном конце выходного блока, а длинные – в другом. И при сжатии, и при разжатии процедура сортировки контейнеров по длинам добавится между циклами (2) и (3).

Рассмотрим последний вариант: весь блок A сохраняется и передается (компрессором декомпрессору), поэтому и строится он именно с учетом та-

кой особенности. Если есть параллельный блок P, можно строить A так, чтобы и P преобразовывать на основе A, и чтобы суммарно блоки A, P, In сжимались лучше, чем P и In. Если нет P, то получается, что In распадается на две компоненты: "идеальную" – моделируемые, предсказываемые A[*i*], и "реальную" – отклонения In[*i*] от предсказанных значений A[*i*].

Двумерный случай

Параллельным блоком может быть предыдущая, уже обработанная строка таблицы. Либо предсказываемая (текущая), формируемая на основе одной или нескольких предыдущих строк. В обоих случаях целесообразно вычислять атрибут как среднее арифметическое соседних элементов, особенно для мультимедийных данных:

```
//если известны те, что выше и левее, то соседних - четыре:
//    P[i-2]  P[i-1]  P[i] P[i+1]
//    In[i-2] In[i-1] In[i]
A[i]=(P[i-1]+  P[i]+P[i+1])/3;
A[i]=(P[i-1]+2*P[i]+P[i+1])/4;
A[i]=(P[i-1]+  P[i]+In[i-1])/3;
A[i]=(P[i-1]+2*P[i]+In[i-1])/4;
A[i]=(P[i-1]+  P[i]+2*In[i-1])/4;
A[i]=(P[i-1]+  P[i]+P[i+1]+In[i-1])/4;
```

Тогда опять же длины контейнеров (частоты значений атрибутов) надо обязательно передавать декомпрессору. Перед делением полезно делать округление до ближайшего числа, кратного делителю. Еще сложнее будут формулы, если использовать более одной предыдущей строки.

Заметим важное полезное свойство метода: увеличение сложности вычислений не влечет увеличения объема необходимой памяти.

Если рассматривать множество строк как единый блок, то в простейшем случае A[*i*]=P[*i*]=In[*i*–*W*] (*W* – число элементов в строке), т. е. в качестве атрибута используется значение "верхнего" элемента. Тогда таблицу длин сохранять не нужно, но первые *W* элементов в неизменном виде – обязательно.

Упражнение. Опишите подробно алгоритм сортировки в случае A[i]=In[i-W]. Начните с W=1 и W=2.

Характеристики методов семейства PBS:
Степень сжатия: увеличивается в 1.0–2.0 раза.
Типы данных: многомерные или многоуровневые данные.
Симметричность по скорости: в общем случае 1:1.
Характерные особенности: необходимо наличие как минимум двух параллельных блоков данных.

Фрагментирование

Цель: разбиение исходного потока или блока на фрагменты с разными статистическими свойствами. Такое разбиение должно увеличить степень последующего сжатия. В простейшем случае битового потока необходимо находить границы участков с преобладанием нулей, участков с преобладанием единиц и участков с равномерным распределением нулей и единиц. Если поток символьный, может оказаться выгодным разбить его на фрагменты, отличающиеся распределением вероятностей элементов, например на фрагменты с русским текстом и фрагменты с английским.

Основная идея состоит в том, чтобы для каждой точки потока X (лежащей между парой соседних элементов) находить значение функции отличия FO(X) предыдущей части потока от последующей. В базовом простейшем варианте используется "скользящее окно" **фиксированной длины**: Z элементов до точки X и Z элементов после нее.

Иллюстрация (Z=7):

...8536429349586436542 | 9865332 | X | 6564387 | 58676780674389...

Цифрами обозначены элементы потока, вертикальными чертами "|" границы левого и правого окон. X не элемент потока, а точка между парой элементов, в данном случае между "2" и "6".

При фиксированной длине окон Z и **одинаковой значимости, или весе,** всех элементов внутри окон (независимо от их удаленности от точки X) значение функции отличия может быть легко вычислено на основании разности частот элементов в левом и правом окнах. Это сумма по всем 2^R возможным значениям V_i элементов. Суммируются абсолютные значения разностей: число элементов с текущим значением V в **левом** окне длиной Z **минус** число элементов с данным значением V в **правом** окне длиной Z. Суммируя 2^R модулей разности, получаем значение FO(X) в данной точке X:

$$FO(X) = \sum_{V=0}^{2^R-1} \left| CountLeft[V] - CountRight[V] \right|,$$

где CountLeft[V] – число элементов со значением V в **левом** окне; CountRight[V] – число элементов со значением V в **правом** окне.

Видно, что если в обоих окнах элементы одинаковые, то сумма будет стремиться к нулю, если же элементы совершенно разные – к $2 \cdot Z$.

Для приведенного выше примера:

V=	2	3	4	5	6	7	8	9
FO(X) =	+|1-0|	+|2-1|	+|0-1|	+|1-1|	+|1-2|	+|0-1|	+|1-1|	+|1-0|
	=6							

После того как все значения FO(X) найдены, остается отсортировать их по возрастанию и выбрать границы по заданному критерию (заданное число границ и/или пока FO(X)>F_{min}).

Размер данных в результате фрагментирования увеличивается: либо появляется второй поток с длинами фрагментов, либо флаги границ в одном результирующем потоке.

Для сжатия результата работы метода может быть применена любая комбинация методов – RLE, LPC, MTF, DC, PBS, HUFF, ARIC, ENUC, SEM...

Методы этой группы **преобразующие** и **поточные**, т. е. могут применяться даже в том случае, когда длина блока с данными не задана.

Обратного преобразования не требуется.

В общем случае скорость работы компрессора зависит только от размера данных, но не от их содержания: Скорость=O(Размер). Памяти требуется порядка 2^{R+1}. Для левого окна – O(2^R), и столько же для правого. Операций умножения и деления нет.

С точки зрения сегментации данных метод отличается от RLE лишь тем, что выделяет из потока (или блока) не только цепочки одинаковых элементов, но и вообще отделяет друг от друга фрагменты с разными распределениями вероятностей значений элементов.

Из краткого описания общей идеи видно, что

- порождается либо второй поток с длинами фрагментов, либо в исходный поток добавляются флаги границ (флаг может отвечать условию не только на значение одного элемента, но и условию на значение функции нескольких элементов);
- задаваемыми параметрами могут быть (максимальная) длина окна Z, F_{min} – минимальное значение FO(X), минимальное расстояние между границами R_{min} (а в случае фрагментирования блока заданной длины – еще и число границ NG, минимальное и/или максимальное число границ: N_{min}, N_{max});
- вычисление функции отличия при нескольких длинах окон – Z_1 , Z_2 , ..., Z_n – в общем случае улучшит качество фрагментирования;
- уменьшение веса элементов окна с удалением от точки X также в общем случае улучшит качество фрагментирования.

Основной алгоритм преобразования

Каков бы ни был размер окон Z, при проходе по входным данным In[N] в каждой точке X при переходе к следующей точке (X+1) достаточно следить за тремя элементами: вышедшим из левого окна (из-за сдвига точки X вправо), вошедшим в правое окно и перешедшим из правого окна в левое. Но

сначала разберем самый понятный алгоритм: с проходом по всем элементам окон для каждой точки X.

```
#define Z 32
for( i=0; i<n; i++)
  CountLeft[i]=CountRight[i]=0; //инициализация          (1)

for( x=0; x<Z; x++) {          //цикл по длине окон:     (2)
  i=In[x];   //возьмем очередной элемент левого окна In[x]
  CountLeft[i]++;     //в левом окне элементов на 1 больше
  i=In[x+Z];          //и аналогично с правым окном
  CountRight[i]++;
  }
```

In[N] – входной фрагментируемый блок;
N – количество байтов во входном блоке;
$n=2^R$ – каждый байт может иметь 2^R значений;
FO[N–2·Z] – значения функции отличия;
CountLeft[n] – частоты элементов в левом окне;
CountRight[n] – частоты элементов в правом окне.

После двух циклов, инициализирующих оба окна, начинаем проход по входному блоку. Изначально левая граница левого окна совпадает с началом данных.

```
for(x=Z; x<N-Z-1; x++) {
//точка X скользит с позиции Z до N-Z-1                 (3)
  f=0;                        //текущее значение функции отличия
  for( i=0; i<n; i++)   //вычислим по формуле как сумму
    f+=abs(CountLeft[i]-CountRight[i]);/*т. е.если без abs()
                            if (CountLeft[i]>CountRight[i])
                              f+=CountLeft[i]-CountRight[i];
                          else f+=CountRight[i]-CountLeft[i];
                            */
  FO[x]=f;       //и запишем в массив
  for(i=0;i<n;i++) CountLeft[i]=CountRight[i]=0;//как в (1)
 for(j=x+1-Z;j<x+1;j++){ //цикл по длине окон: как в (2)
 CountLeft[In[j]]++;//подсчет частот элементов в левом окне
 CountRight[In[j+Z]]++; //и аналогично в правом
 }//но теперь левая граница левого окна - в позиции (x+1-Z)
}
```

Наконец, последний цикл – это либо сортировка FO[N] с целью нахождения заданного числа максимальных значений (заданного числа самых "четких" границ), либо просто нахождение таких FO[k], значение которых больше заданного. Последнее можно делать в основном цикле.

Упражнение. Внесите необходимые изменения в (3), если задано минимальное значение отличия F_{min}.

Пути улучшения скорости

Если последние два цикла внутри (3) перенести и выполнять их до первого, вычисляющего FO, то (1) и (2) не нужны. Но поскольку внутри цикла-прохода (3) по In достаточно следить за тремя элементами, в нем вообще не будет внутренних циклов. После выполнения (1) и (2) поступим так:

```
f=0;                    //исходное значение функции отличия
for( i=0; i<n; i++)  //вычислим по формуле как сумму      (2a)
   f+= abs(CountLeft[i]-CountRight[i]);
FO[0]=f;      //и запишем в массив
```

А внутри основного цикла, проходящего по входному блоку In[N]:

```
for(x=Z; x<N-Z-1;x++) {
//точка X скользит с позиции Z до N-Z-1                    (3a)
  i=In[x-Z]; //элемент, вышедший из левого скользящего окна
  f-=abs(CountLeft[i]-CountRight[i]);
//сумма без 1 слагаемого
  CountLeft[i]--; //теперь в левом окне таких на 1 меньше
  f+=abs(CountLeft[i]-CountRight[i]);
//обратно к полной сумме

 i=In[x+Z];  //элемент, вошедший в правое окно
//совершенно аналогично обновлению для левого окна
 f-=abs(CountLeft[i]-CountRight[i]);
 CountRight[i]++; //теперь в правом окне таких на 1 больше
  f+=abs(CountLeft[i]-CountRight[i]);

  i=In[x]; //элемент, перешедший из правого окна в левое
  f-=abs(CountLeft[i]-CountRight[i]);
  CountLeft[i]++;  //теперь в левом окне таких на 1 больше
  CountRight[i]--; //а в правом - на 1 меньше
  f+=abs(CountLeft[i]-CountRight[i]);

  FO[x]=f; //запишем значение функции отличия в массив
}
```

Упражнение. Как изменится значение f, если все 3 элемента одинаковы, причем в левом окне таких было 7, а в правом 8?

Если Z существенно меньше n, вместо **(2a)** лучше делать другой цикл, до 2*Z. Внутри него будет прибавление модуля разности для текущего элемента к сумме, если этот модуль еще не вошел в сумму, и процедура, запоминающая какие элементы уже вошли в сумму:

```
f=0;              // исходное значение функции отличия
stacksize=0;      // и размера стека
for (x=0; x<2*Z; x++)                                    (2z)
{
   i=In[x];       // текущий элемент
   s=0;           // просмотрим все уже обработанные:
   while(s<stacksize) if (Stack[s]==i) goto NEXT
   // если элемент еще не вошел
   f+= abs(CountLeft[i]-CountRight[i]);
   Stack[stacksize++]=i;  // то вносим сейчас
NEXT:
}
FO[0]=f;       // запишем в массив
```

Точно так же можно модифицировать первый цикл внутри **(3)**, вычисляющий FO: вместо него будет **(2z)**, т. е. цикл не до *n*, а до 2·*Z*.

Чтобы избавиться от второго цикла до n внутри **(3)**, заведем два массива с флагами FlagLeft и FlagRight, параллельные CountLeft и CountRight. В них будем записывать, в какой позиции *x* делалась запись в соответствующую позицию массива Count. При чтении же из Count будем читать и из Flag. Если значение в Flag меньше текущего *x*, то в соответствующей ячейке массива Count должен быть нуль.

Упражнение. Перепишите **(3)** так, чтобы не было циклов до *n*.

Пути улучшения сжатия

Общий случай

Самый выгодный путь – использование **окна с убыванием веса элементов** при удалении от точки *X*. Линейное убывание, например: вес двух элементов, между которыми лежит точка *X*, равен *Z*, вес двух следующих, на расстоянии 1 от *X*, равен *Z*-1, и т. д., (*Z*-*d*) у элементов на расстоянии *d*, нуль у элемента, только что вышедшего из левого окна, и у элемента, который войдет в правое окно на следующем шаге.

Внутри основного цикла будет цикл либо до *Z* – по длине окна, либо до *n* – по всем возможным значениям элементов. В зависимости от значений *Z* и *n*, один из этих двух вариантов может оказаться существенно эффективнее другого.

В первом случае – *Z* существенно меньше *n*, выгоднее цикл до *Z* – вместо инкрементирования будем добавлять веса, т. е. расстояния от *X* до дальней границы (самой левой в случае левого окна, самой правой в случае правого):

```
for( x=0; x<Z; x++) {   //цикл по длине окон:          (2`)
```

```
//  CountLeft[In[x]]++;    //так было в цикле                (2)
//  CountRight[In[x+Z]]++;//а теперь: прибавляем
CountLeft[In[x]]+=x+1;  // расстояние от левой границы до x,
CountRight[In[x+Z]]+=Z-x;// от x+Z до правой границы:
                        // 2*Z-(x+Z)
}
```

И аналогичные модификации в цикле (3).

Во втором случае – *n* существенно меньше *Z*, выгоднее цикл до *n* – реализация гораздо сложнее: в дополнение к массиву со счетчиками появляется второй массив с весами. Зато сохраняется возможность действовать так, как в (3а), не делая цикла по длине окна *Z* внутри основного цикла:

```
for(x=Z;x<N-Z-1;x++) {//точка X - с позиции Z до N-Z-1  3a`)
 f=0;
 for(i=0;i<n; i++)
             // как изменятся веса элементов со значением
  {          // i при переходе к позиции x+1?
  WeightLeft[i]-=CountLeft[i];     // всего их CountLeft[i],
                                   // каждый изменится на -1
  WeightRight[i]+=CountRight[i]); // а каждый правый - на +1
   // считаем значение отличия
  f+=abs(WeightLeft[i]-WeightRight[i]);
  }
   // теперь сдвинем x на 1 вправо, внося и вынося элементы

   i=In[x-Z]; //элемент, вышедший из левого скользящего окна
   CountLeft[i]--; // теперь в левом окне таких на 1 меньше
         //вес его уже 0, поэтому WeightLeft[i] не изменен

   i=In[x+Z];   //элемент, вошедший в правое окно
   f-=abs(WeightLeft[i]-WeightRight[i]);
// сумма без 1 слагаемого
  CountRight[i]++;//теперь в правом окне таких на 1 больше
  WeightRight[i]++;//и вес их на 1 больше
// обратно к полной сумме
  f+=abs(WeightLeft[i]-WeightRight[i]);

  i=In[x];   // элемент, перешедший из правого окна в левое
f-=abs(WeightLeft[i]-WeightRight[i]);
// сумма без 1 слагаемого
  CountLeft[i]++; // теперь в левом окне таких на 1 больше
  WeightLeft[i]+=Z;     // вес их на Z больше
  CountRight[i]--;   // а в правом - на 1 меньше
  WeightRight[i]-=Z;    // вес их на Z меньше
// обратно к полной сумме
  f+=abs(WeightLeft[i]-WeightRight[i]);
```

```
  FO[x]=f;    // запишем значение функции отличия в массив
}
```

Упражнение. Как будут выглядеть циклы (1) и (2) ?

Второй путь улучшения качества фрагментирования – находить максимальное значение функции отличия при **нескольких длинах окна**: Z_1, Z_2, … …, Z_q . Тогда эти вычисляемые значения FO_i придется делить на длину соответствующего окна Z_i, при котором это FO_i вычислено. И предварительно умножать на 2^K, чтобы получались величины не от 0 до 2, а от 0 до 2^{K+1}.

Третий путь – периодическое (через каждые *Y* точек) нахождение оптимальной длины окна *Z0*.

Четвертый – анализ массива FO после того, как он полностью заполнен.

Пятый – использование и других критериев при вычислении FO(*X*), например

не $\Sigma|$ CountLeft[V] – CountRight[V] $|$,

а Σ(CountLeft[V] – CountRight[V])^2.

D-мерный случай

В двумерном случае появляется возможность следить за изменением характеристик элементов при перемещении через точку *X* по *K* разным прямым. В простейшем варианте – по двум перпендикулярным: горизонтальной и вертикальной.

Алгоритм может выглядеть, например, так: сначала проходим по строкам и заполняем массив FO значениями функции отличия при горизонтальном проходе: FO[*X*]=$FO_{ГОР}(X)$. Затем идем по столбцам, вычисляя отличия при вертикальном проходе $FO_{ВЕР}(X)$ и записывая в массив максимальное из этих двух значений – $FO_{ГОР}(X)$ и $FO_{ВЕР}(X)$. Дальше можно таким же образом добавить FO при двух диагональных проходах (рис. 6.2), причем не для каждой точки, а в окрестностях точек с максимумами FO[*X*].

3		2		4
	3	2	4	
1	1	*X*	1	1
	4	2	3	
4		2		3

Рис. 6.2. Иллюстрация двумерного случая с K=4

Кроме того, если хватает ресурсов, можно рассматривать отличие не отрезков длиной *Z*, отмеченных на *K* прямых, а секторов круга радиуса *Z*, разбитого на *K* секторов. Это могут быть, например, две половины круга или же 4 четвертинки круга.

Теперь можно варьировать не только Z, пытаясь найти оптимальное значение, но также и K.

Таким образом, будут найдены "контуры" – границы, отделяющие области с разными характеристиками совокупности содержащихся в этих областях элементов.

В трехмерном случае: либо используем $K_П$ прямых, либо $K_К$ секторов круга, либо $K_С$ секторов сферы. Причем оптимальные (с точки зрения вычисления) значения $K_П$ не 2, 4, 8..., как в двумерном, а 3, 7, 13...

Упражнение. Нарисуйте эти варианты с тремя прямыми, семью и тринадцатью. Как продолжить процесс дальше?

Совершенно аналогично и в D-мерном случае.

Характеристики метода фрагментирования:

Степень сжатия: увеличивается в 1.0–1.2 раза.

Типы данных: неоднородные данные, особенно с точными границами фрагментов.

Симметричность по скорости: более чем 10:1.

Характерные особенности: может применяться не только для сжатия данных, но и для анализа их структуры.

Глава 7. Предварительная обработка данных

Предварительная обработка данных выполняется до их сжатия как такового и призвана улучшить коэффициент сжатия. Схема кодирования в этом случае приобретает вид:

Исходные данные → препроцессор → кодер → сжатые данные

а схема декодирования:

Сжатые данные → декодер → постпроцессор → восстановленные данные

Препроцессор должен так видоизменить входной поток, чтобы коэффициент сжатия преобразованных данных был в среднем лучше коэффициента сжатия исходных, "сырых" данных. Система препроцессор-постпроцессор работает автономно; кодер "не знает", что он сжимает уже преобразованные данные. Постпроцессор восстанавливает исходные данные без потерь информации, поэтому результаты работы и само существование системы препроцессор-постпроцессор могут быть не заметны для внешнего наблюдателя.

В общем случае преобразования, выполняемые препроцессором, могут быть реализованы в кодере и при этом быть такими же эффективными с

точки зрения улучшения сжатия. Но предварительная обработка позволяет существенно упростить алгоритм работы кодера и декодера, дает возможность создания достаточно гибкой специализированной системы сжатия данных определенного типа на основе универсальных алгоритмов. Модули препроцессор-постпроцессор обычно можно применять в сочетании с различными кодерами и архиваторами, повышая их степень сжатия и, возможно, скорость работы.

В этой главе рассмотрено несколько способов предварительной обработки типичных данных. Предполагается, что алгоритм сжатия предназначен для кодирования источников с памятью. Описываемые методы препроцессинга достаточно хорошо известны в кругу разработчиков программ сжатия, но не получили широкого освещения в литературе.

Препроцессинг текстов

В последние несколько лет приобрели популярность и были существенно развиты методы предварительной обработки текстовой информации. В настоящее время специализированный препроцессинг текстов, позволяющий заметно улучшить сжатие, используется в таких архиваторах, как ARHANGEL, JAR, RK, SBC, UHARC, в компрессорах DC, PPMN.

Использование словарей

Идея преобразования данных с помощью словаря заключается в замене каких-то строк текста на коды фраз словаря, соответствующих этим строкам. Часто указывается просто номер фразы в словаре. Пожалуй, метод словарной замены является самым старым и известным среди техник предварительного преобразования текстов, да и любых данных вообще. Сама словарная замена может приводить как к сжатию представления информации, так и к его расширению. Главное, чтобы при этом достигалась цель преобразования – изменение структуры данных, позволяющее повысить эффективность последующего сжатия.

Можно выделить несколько стратегий построения словаря. По способу построения словари бывают:

- статическими, т. е. заранее построенными и полностью известными как препроцессору, так и постпроцессору;
- полуадаптивными, когда словарь выбирается из нескольких заранее сконструированных и известных препроцессору и постпроцессору *или* достраивается, при этом один из имеющихся словарей берется за основу;
- адаптивными, т. е. целиком создаваемыми специально для сжимаемого файла (блока) данных на основании его анализа.

В качестве фраз обычно используются [4]:
- целые слова;
- последовательности из двух символов (биграфы);
- пары букв, фонетически эквивалентных одному звуку;
- пары букв согласная – гласная или гласная – согласная;
- последовательности из *n* символов (*n*-графы).

Как правило, существует огромное количество последовательностей, которые в принципе могут стать фразами, поэтому необходимо применять какие-то критерии отбора. Обычно в словарь добавляются:
- последовательности, чаще всего встречающиеся в сжимаемом тексте или в текстах определенного класса;
- одни из самых часто используемых последовательностей, удовлетворяющие некоторым ограничениям;
- слова, без которых едва ли сможет получиться связный текст: предлоги, местоимения, союзы, артикли и т. п.

Словарь n-графов

Судя по всему, наибольшее распространение в современных архиваторах и компрессорах получила стратегия статического словаря, состоящего из последовательностей букв длины от 2 до небольшого числа *n* (обычно 4–5). В большинстве случаев размер словаря равен примерно 100 таким фразам. К достоинствам данного типа словаря можно отнести:
- малый размер;
- отсутствие жесткой привязки к определенному языку;
- обеспечение существенного прироста степени сжатия;
- простота реализации.

Небольшой размер словаря обусловлен двумя причинами:
- это упрощает кодирование фраз словаря;
- дальнейшее увеличение размера словаря улучшает сжатие лишь незначительно (справедливо для BWT и в меньшей степени для LZ) либо даже вредит в большинстве случаев (справедливо для PPM).

Обычно тексты представлены в формате "plain text", когда 1 байт соответствует одному символу. Так как размер словаря мал, то в качестве индекса фраз выступают неиспользуемые или редко используемые значения байтов. Например, если обрабатывается текст на английском языке, то появление не-ASCII байтов со значением 0x80 и больше маловероятно. Поэтому мы можем заместить все биграфы "th" на число 0x80. Если байт 0x80 все же встретится в обрабатываемых данных, то он может быть передан как пара (<флаг исключения>, <0x80>), где флаг исключения может быть равен,

например, 0xFF. Это обеспечивает однозначность восстановления текста постпроцессором.

Упражнение. Каким образом будет передан байт, равный самому флагу исключения?

Недостатком данного типа словаря является отсутствие формализованного алгоритма построения. Как показали эксперименты, добавление в словарь самых часто используемых последовательностей букв небольшой длины действительно улучшает сжатие, но такой подход не приводит к получению оптимального состава словаря. Поэтому построение словаря делается в полуавтоматическом режиме, когда для заданного кодера и заданного тестового набора текстов эмпирически определяется целесообразность внесения или удаления из словаря той или иной фразы, исходя из изменения коэффициента сжатия тестового набора.

Например, для английского языка хорошо работает словарь, представленный в табл. 7.1. В него входят 45 последовательностей длины 2 (биграфов), 25 последовательностей длины 3 (триграфов) и 16 последовательностей длины 4 (тетраграфов). Список отсортирован в примерном порядке убывания полезности фраз (внутри каждой группы *n*-графов – слева направо и сверху вниз).

Таблица 7.1

Биграфы	Триграфы	Тетраграфы
th er in ou an en	the ing and	ight self
ea or ll is on ar	for ess ver	ward this
st gh ed ee om oo	was igh ous	have been
ow ss ur ld at sh	our ell een	able nder
id sa ic tr al il	had ich ugh	ttle with
as ir ec ul ly et	her out his	ound reat
ai ch ot it av im	ead ard ome	that what
ol to qu	est ght rom	from ther
	ith	

Пример

Строка

he took his vorpal sword in hand

будет преобразована в последовательность:

he <to>ok <his> v<or>p<al> sw<or>d <in> h<and>

В угловых скобках показаны *n*-графы, заменяемые на их индекс в словаре.

Сначала отображаются тетраграфы (в разбираемой строке их нет), затем триграфы и в самую последнюю очередь биграфы.

Рассмотрим пример реализации простейшей словарной замены для английских текстов. Пусть словарь состоит только из 10 биграфов, в качестве кодов биграфов используются не-ASCII байты со значениями 128–137, исключительные ситуации не отслеживаются, т. е. предполагается, что во входном файле отсутствуют не-ASCII символы.

```
const int BIGRAPH_NUM = 10;
const char blist [BIGRAPH_NUM][3] = {
   "th", "er", "in", "ou", "an",
   "en", "ea", "or", "ll", "is"
};
/*предполагаем, что bnum - глобальная переменная и
   инициализируется нулями; в этом массиве будем хранить коды
   биграфов
*/
unsigned char bnum [256][256];

int code = 128;
for (int i = 0; i < BIGRAPH_NUM; i++){ //заполним bnum
   bnum [ blist[i][0] ] [ blist[i][1] ] = code++;
}
int c1, c2;
c1 = DataFile.ReadSymbol();
while ( c1 != EOF ) {
   c2 = DataFile.ReadSymbol();
   if ( c2 == EOF ) {
      PreprocFile.WriteSymbol (c1);
      break;
   }
   if (bnum[c1][c2]){
   //такой биграф имеется в словаре, произведем замену
      PreprocFile.WriteSymbol (bnum[c1][c2]);
      c1 = DataFile.ReadSymbol();
   }else{
      PreprocFile.WriteSymbol (c1);
      c1 = c2;
   }
}
```

Для русского языка неплохие результаты показывает словарь, описанный в табл. 7.2. Заметим, что словарь для русских текстов имеет меньший размер, чем для английских.

Таблица 7.2

Биграфы	Триграфы	Тетраграфы
то ст ов ен по аз ак ер ол ор	ств был при	лько врем
он ел ет ам от ом ас ан ин ск	про ере ого	тобы огда
на за ар ик пр ев ив ит ил ед	ост ись енн	азал ольш
ем ть ал ат ав ся ес об од ос	вет	еред отор
ис ог им ег ич сь		

За счет описанной словарной замены достигается значительное улучшение сжатия текстов (табл. 7.3). В качестве тестового файла был использован роман на английском языке "Three men in a boat (to say nothing of the dog)" ("Трое в лодке, не считая собаки"), электронный вариант которого занимал около 360 Кб в исходном виде. Отказ от сравнения с помощью больших текстовых файлов Book1 и Book2, входящих в состав стандартного набора CalgCC, был обусловлен тем, что они являются не вполне типичными текстами. Первый файл содержит значительное количество опечаток, а второй – большой объем служебной информации о форматировании текста.

Таблица 7.3

Архиватор	Тип метода сжатия	Размер архива исходного файла, байт	Размер архива обработанного файла	Улучшение сжатия %
Bzip2, вер. 1.00	BWT	109736	107904	1.7
WinRAR, вер. 2.71	LZ77	130174	126026	3.2
HA a2, вер. 0.999c	PPM	108443	106831	1.5

Заметим, что в случае Bzip2 и HA сжатие могло быть улучшено. С одной стороны, коэффициент сжатия алгоритма BWT зависит от номеров символов в алфавите, а нами не делалось никаких попыток переупорядочить более подходящим образом ASCII-символы и номера добавленных нами фраз. С другой стороны, HA использует небольшой объем памяти и часть накапливаемой в PPM-модели статистики периодически выбрасывается, что ухудшает предсказание и, соответственно, сжатие.

Эффективность использования *n*-графового словаря с другим составом фраз совместно с BWT архиваторами оценена в [2].

Обычно применение *n*-графового словаря улучшает сжатие компрессоров, использующих PPM или BWT, на 2%.

Степень сжатия компрессоров на базе методов Зива – Лемпела может быть заметно улучшена за счет увеличения размера словаря препроцессора и использования фраз большей длины. В принципе фразы могут включать

не только буквы, но и пробелы, что, кстати, приводит к ухудшению сжатия в случае BWT или PPM.

Словарь LIPT

В качестве примера словаря, в котором фразами являются слова, рассмотрим схему Length Index Preserving Transformation (LIPT) – преобразование с сохранением индекса длины [1]. В словарь включаются самые часто используемые слова, определяемые исходя из анализа большого количества текстов на заданном языке и, возможно, определенной тематики. Под словом здесь понимается последовательность букв, ограниченная с двух сторон символами, не являющимися буквами ("не-букв"). Весь словарь делится на части (подсловари). В зависимости от своей длины L_i слово попадает в часть словаря с номером i. В пределах подсловаря фразы отсортированы в порядке убывания частоты, т. е. самое часто используемое слово имеет минимальный индекс 0. Каждое слово исходной последовательности, которому соответствует какая-та фраза словаря, кодируется следующим образом:

флаг	длина слова (номер подсловаря)	индекс в подсловаре

В качестве алфавита для записи длины слова и индекса авторами алгоритма предлагается использовать алфавит языка. Например, если мы работаем с английским языком, то "a" соответствует 1, "b" – 2, ..., "z" – 26, "A" – 27, ..., "Z" – 52 и далее "aa" – 53, "ab" – 54... Нулевой индекс явным образом не передается. Если, допустим, слово "mere" имеет индекс 29 в своем подсловаре 4, то оно будем преобразовано так (для большей доходчивости различные части кода фразы выделены подчеркиванием):

"mere" → "<флаг>_d_C".

Если индекс равен 56, то отображение будет таким:

"mere" → "<флаг>_d_ad".

Индекс указан как "ad", поскольку он записывается в позиционной системе счисления и первая буква соответствует старшему порядку. Конец последовательности, передающей индекс, нет нужды указывать явно, поскольку ссли мы рассматриваем "mere" как слово, то оно должно ограничиваться какой-то "не-буквой", которая и станет маркером конца записи индекса.

Если слово отсутствует в словаре, то оно без изменений копируется в файл преобразованных данных.

Эксперименты показывают, что для различных алгоритмов сжатия выгоднее использовать разные алфавиты длины слова и индекса. Также иногда имеет смысл использовать иной принцип разбиения словаря. Рассмотрим результаты сжатия текста "Three men in a boat (to say nothing of the dog)" для следующих трех алгоритмов LIPT:

1) алгоритма 1 – практически соответствует авторскому, но алфавит ограничен только строчными английскими буквами;

2) алгоритма 2 – алфавиты длины слова и индекса отличаются от алфавита букв и не пересекаются между собой;
3) алгоритма 3 – словарь разбивается на подсловари не только по критерию длины фраз, но и по соответствию слова определенной части речи[1]; используются такие же алфавиты, что и в алгоритме 3.

Словарь LIPT был построен на основании анализа примерно 50 Мб английских текстов различного характера. Общий объем словаря составил около 53 тыс. фраз, или 480 Кб. Для реализации алгоритма 3 примерно 11.6 тысяч. слов был присвоен атрибут принадлежности к определенной части речи, например "the" – артикль и т. д. Классификация была очень груба – использовалось всего лишь 9 категорий. Если слово могло относиться к нескольким частям речи, то выбиралась самая часто употребляемая форма (понятно, здесь был определенный произвол). Слова, не получившие такого лексического атрибута, трактовались как существительные. Схема кодирования для алгоритма 3 имела вид:

флаг	часть речи	длина слова	индекс в подсловаре

Например:

"mere" → "<флаг><прилагательное><длина = 4><индекс>",

где индекс определяет положение фразы в подсловаре 4-буквенных прилагательных.

Результаты эксперимента приведены в табл. 7.4.

Таблица 7.4

Архиватор	Тип метода сжатия	Алгоритм LIPT 1		Алгоритм LIPT 2		Алгоритм LIPT 3	
		Размер архива, байт	Улучшение сжатия	Размер архива, байт	Улучшение сжатия	Размер архива, байт	Улучшение сжатия, %
Bzip2, вер. 1.00	BWT	102797	6.3	100106	8.8	101397	7.6
WinRAR, вер. 2.71	LZ77	118647	8.9	122118	6.2	125725	3.4
HA a2, вер. 0.999c	PPM	100342	7.5	98838	8.9	98173	9.5

Таким образом, для LZ77 лучше всего подходит обычный алгоритм LIPT. Очевидно, что LZ77 плохо использует корреляцию между строками и

[1] *Эта модификация LIPT разработана и реализована М. А. Смирновым летом 2001 г.*

основной выигрыш достигается за счет уменьшения длин совпадения и величин смещений. Алгоритм 2 можно признать компромиссным вариантом. Для PPM-компрессоров имеет смысл использовать алгоритм 3.

Преобразование заглавных букв

Заглавные буквы существенно увеличивают число встречающихся в тексте последовательностей и, соответственно, приводят к ухудшению сжатия по сравнению с тем случаем, если бы их не было вообще. Способ частичного устранения этого неприятного явления очевиден. Если слово начинается с заглавной буквы, то будем преобразовывать его так, как показано на рис. 7.1 [2].

флаг	первая буква, преобразованная в строчную	оставшаяся часть слова

Рис. 7.1. Преобразованный вид слова, начинавшегося с заглавной буквы

При этом под словом понимается последовательность букв, ограниченная с двух сторон "не-буквами".

Например, если в качестве флага используется байт 0x00, то преобразование может иметь вид:

"_Если_" → "_<0x00>если_"

Для BWT- и PPM-компрессоров отмечается улучшение сжатия, если после флага вставляется пробел, т. е. когда результат преобразования имеет вид типа "_<0x00>_если_". Невыгодно преобразовывать слова, состоящие только из одной заглавной буквы.

Иногда в текстах встречается много слов, набранных полностью заглавными буквами. Очевидно, что в этом случае описанное преобразование не только не помогает, но и, возможно, даже вредит. Поэтому целесообразно использовать еще одно отображение, переводящее слова, состоящие целиком из заглавных букв, в соответствующие слова из строчных букв (рис. 7.2).

флаг 2	последовательность букв слова, преобразованных в строчные

Рис. 7.2. Преобразованный вид слова, целиком состоявшего из заглавных букв

Если в роли флага 2 выступает байт 0x01, то справедлив такой пример отображения:

"_АЛГОРИТМЫ_" → "_<0x01>алгоритмы_".

Как уже указывалось, добавление пробела после флагов улучшает сжатие BWT- и PPM-компрессоров.

Может показаться, что описанный алгоритм вносит избыточность за счет использования флага даже в тех случаях, когда в нем нет особой необходимости. Если текущее слово является первым встреченным после точки, восклицательного или вопросительного знака, то его начальная буква наверняка является заглавной и флаг излишен. Естественно, для обеспечения правильности декодирования необходимо одно из двух:

- особо обрабатывать ситуации, когда первая буква все-таки строчная, например использовать специальный флаг, сигнализирующий об исключении;
- делать компенсирующее преобразование, отображающее все строчные буквы, встречаемые после знаков конца предложения, в заглавные.

Несмотря на кажущуюся эффективность, в случае компрессоров BWT и PPM такая техника работает хуже ранее рассмотренных, поскольку нарушает регулярность в использовании флагов и искажает контекстно-зависимую статистику частот символов. Если же необходимо разработать препроцессор текстовых данных исключительно для LZ-архиваторов, то, действительно, следует избавиться от ненужных флагов.

В табл. 7.5 описывается эффект от использования преобразования заглавных букв для архиваторов различных типов на примере сжатия уже упоминавшегося электронного варианта книги "Three men in a boat (to say nothing of the dog)".

Чтобы продемонстрировать роль вставки пробела после флагов, мы рассмотрели два алгоритма предварительной обработки. Алгоритм 1 преобразует:

- слова, начинающиеся с заглавной буквы, но далее состоящие из одной или более строчных, в соответствии с правилом, изображенным на рис. 7.1;
- слова из двух и более символов, содержащие только заглавные буквы, в соответствии с правилом, приведенным на рис. 7.2.

Алгоритм 2 использует эти же два отображения, но добавляет после флагов пробел.

Таблица 7.5

Архиватор	Тип метода сжатия	Алгоритм 1		Алгоритм 2	
		Размер архива, байт	Улучшение сжатия, %	Размер архива, байт	Улучшение сжатия, %
Bzip2, вер. 1.00	BWT	108883	0.8%	108600	1.0%
WinRAR, вер. 2.71	LZ77	128351	1.4%	128563	1.2%
HA a2, вер. 0.999c	PPM	107285	1.1%	107137	1.2%

Добавление пробела заметно улучшило сжатие для алгоритма BWT, благоприятно повлияло на эффективность PPM, но сказалось отрицательно в случае LZ77.

Модификация разделителей

Текст содержит не только буквы, но и символы-разделители. Разделители бывают:

- естественными – это знаки препинания, пробелы;
- связанными с форматированием текста – символы конца строки (СКС), символы табуляции.

Символы-разделители в большинстве случаев плохо предсказываются на основании контекстно-зависимой статистики. Особенно плохо предсказываются символы конца строки, т. е. пара символов {перевод каретки, перевод строки} CR/LF или символ перевода строки LF.

Коэффициент сжатия BWT- и PPM-компрессоров может быть улучшен, если преобразовать знаки препинания и СКС, выделив их из потока букв. Наиболее эффективным и простым способом модификации является добавление пробела перед этими разделителями [2]. Например:

"очевидно," → "очевидно_,",

при этом:

"очевидно_," → "очевидно__,".

Преобразование однозначно: когда постпроцессор встречает пробел, он смотрит на следующий символ и, если это разделитель (знак препинания или СКС), пробел на выход не выдается.

Положительный эффект отображения объясняется тем, что пробел встречается в таких же контекстах, в которых встречаются и преобразуемые знаки. Поэтому выполнение преобразования приводит к следующему:

- уменьшается количество используемых контекстов и, следовательно, увеличивается точность накапливаемой статистики;
- пробел сжимается часто сильнее, чем соответствующий знак препинания или СКС, и при этом предоставляет несколько лучший с точки зрения точности предсказания контекст для последующего разделителя.

Укажем несколько способов повышения производительности схемы:

- в случае СКС пробел в большинстве случае выгодно добавлять и после символа (-ов) конца строки; при этом лучше воздержаться от преобразования, если первый символ новой строки не является ни буквой, ни пробелом;
- сжатие обычно улучшается в среднем, если не делается преобразование знаков препинания, за которыми не следует ни пробел, ни СКС;

- не следует вставлять пробел перед знаком препинания, если предыдущий символ не является ни буквой, ни пробелом.

Упражнение. Почему необходимо делать модификацию в том случае, когда предыдущий символ является пробелом?

В целом эффект от модификации разделителей менее стабилен, чем от словарной замены или перевода заглавных букв в строчные. Выигрыш практически всегда достигается главным образом за счет преобразования СКС и составляет 1–2 %. В случае BWT преобразование знаков препинания дает неустойчивый эффект, лучше обрабатывать таким образом только СКС.

Результаты сжатия различными архиваторами преобразованного текста книги "Three men in a boat (to say nothing of the dog)" приведены в табл. 7.6.

Таблица 7.6

Архиватор	Тип метода сжатия	Размер архива обработанного файла, байт	Улучшение сжатия, %
Bzip2, вер. 1.00	BWT	108864	0.8
WinRAR, вер. 2.71	LZ77	130835	-0.5
HA a2, вер. 0.999c	PPM	106582	1.7

Данные табл. 7.6 подтверждают, что использование описанного преобразования в сочетании с методов LZ77 бессмысленно.

Специальное кодирование символов конца строки

Как уже отмечалось, СКС плохо сжимаются сами и ухудшают сжатие окружающих их символов.

Очевидно, что если бы мы заменили СКС на пробелы, то сжатие текстов улучшилось бы существенным образом. Этого можно достигнуть, искусственно разбив исходный файл на два блока: собственно текст, в котором СКС заменены на пробелы, и сведения о расположении СКС в файле, т. е. фактически информация о длинах строк. Если в расположении СКС имеется достаточно строгая регулярность, то сумма размеров сжатого блока преобразованного текста и сжатого блока длин строк будет меньше размера архива исходного файла [2].

Эффективность специального кодирования СКС напрямую зависит от характера распределения длин строк в тексте. Если текст отформатирован по ширине строки, то сведения о длинах строк могут быть представлены очень компактно. Напротив, если в тексте много коротких строк, максимальная длина строки в символах не выдерживается постоянной, то скорее всего, специальное кодирование СКС будет малополезно или вовсе невыгодно.

Одним из возможных способов задания длины строки является количество пробелов, лежащих между предыдущим и текущим СКС. Например:

Строка	Длина строки в пробелах
Twas_brillig,_and_the_slithy_toves	5
Did_gyre_and_gimble_in_the_wabe;	6
All_mimsy_were_the_borogoves,	4
And_the_mome_raths_outgrabe.	4

Поскольку мы измеряем длину строки не в символах, а скорее в словах, то количество наблюдаемых длин строк не будет велико и с помощью арифметического кодирования можно достаточно компактно представить информацию о расположении СКС в тексте. В простейшем случае достаточно кодировать длины на основании безусловных частот их использования.

Постпроцессор получит два блока данных:

Блок	Данные блока
Текст	Twas_brillig,_and_the_slithy_toves_ Did_gyre_and_gimble_in_the_wabe;_ All_mimsy_…
Длины строк	5, 6, …

Так как постпроцессору известно, что первая строка содержит 5 пробелов, то он заменит 6-й по счету пробел на СКС:

"…toves_ Did…" → "…toves<СКС>Did…"

Для следующей строки он заменит на СКС 7-й пробел и т. д. Таким образом текст будет полностью реконструирован.

Укажем несколько способов повышения степени сжатия.

Оценка вероятности длины строки может быть улучшена, если принимать во внимание символы, примыкающие к пробелу слева и справа, и если учитывать текущую длину строки в символах. Это позволит компактнее представить информацию о длинах строк с помощью арифметического кодирования.

Эксперименты показывают, что можно заменять на пробелы не все СКС, а только соответствующие достаточно длинным строкам. Зачастую это способствует повышению общей степени сжатия. При этом, однако, появляется проблема определения, при каких же именно длинах строк выгодно "запускать" преобразование. В компрессоре PPMN используется следующий способ нахождения порога, при превышении которого длиной L строки включается преобразование СКС. Величина L принимается за постоянную на протяжении обработки всего файла. Для вычисления L анализируется достаточно большой блок (до 32 Кб) исходного файла и собирается информация о количестве (частоте) строк с определенной длиной L, измеряемой в символах. В подавляющем большинстве случаев частоты длин строк максимальны в районе наибольшей наблюдаемой длины L_{max} строк, а затем дос-

таточно резко падают с уменьшением L. Поэтому, двигаясь от L_{max} к нулевой длине, находим сначала максимум частоты, а затем, продолжая уменьшать L, ищем пороговое значение. Порог принимается равным L, для которой частота впервые становится меньше средней частоты длин строк. С другой стороны, в качестве порога целесообразно выбирать точку, в которой произошло многократное падение частоты, что характерно для текстов со строгим выравниванием по ширине.

Данная техника позволяет определять порог с очень высокой точностью в случае текстов достаточно простым форматированием.

На рис. 7.3 изображено распределение частот длин строк, полученное для первых 32 Кб текста "Three men in a boat (to say nothing of the dog)". Видно, что $L_{max} = 73$, максимум частоты находится в точке $L = 72$, а порог равен 65 символам, поскольку при этом частота впервые становится меньше средней частоты, если считать от точки $L = 72$.

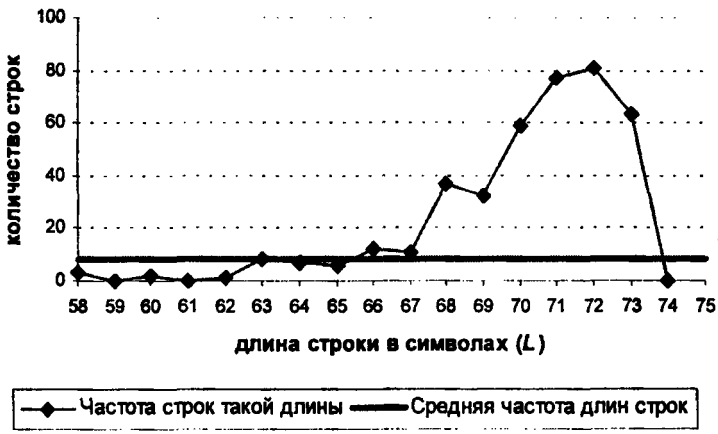


Рис. 7.3. Распределение частот длин строк, полученное для первых 32 Кб текста "Three men in a boat (to say nothing of the dog)"

В качестве примера укажем в табл. 7.7 результаты сжатия тремя выбранными нами архиваторами преобразованного текста "Three men in a boat (to say nothing of the dog)". Длина строк измерялась в пробелах, преобразовывались только СКС в строках с длиной 65 символов и более. Длины строк сжимались с помощью арифметического кодера на основании их безусловных частот. Размер закодированного описания длин получился равным 793 байтам, и это число добавлялось к размеру архива обработанного файла при сравнении эффективности преобразования относительно разных методов сжатия (BWT, LZ77, PPM).

Таблица 7.7

Архиватор	Тип метода сжатия	Размер архива исходного файла, байт	Размер архива обработанного файла плюс размер описания длин строк, байт	Улучшение сжатия, %
Bzip2, вер. 1.00	BWT	109736	106069	3.3
WinRAR, вер. 2.71	LZ77	130174	126042	3.2
HA a2, вер. 0.999c	PPM	108443	105018	3.2

Видно, что специальное кодирование СКС выгодно использовать в сочетании с любым универсальным методом сжатия.

Оценка общего эффекта от использования предварительной обработки

До сих мы рассматривали различные методы препроцессинга текстов независимо друг от друга. Естественно, что практический интерес требует их одновременного использования. Получим ли мы при этом увеличение сжатия равным сумме улучшений, обеспечиваемых каждым способом препроцессинга по отдельности? Ответ на этот вопрос дают табл. 7.8 и 7.9, в которых приведены результаты сжатия текста "Three men in a boat (to say nothing of the dog)", преобразованного с помощью последовательного применения четырех техник:

- специального кодирования СКС;
- преобразования заглавных букв;
- модификации разделителей;
- словарной замены *n*-графов (словарь из табл. 7.1).

Таблица 7.8

Архиватор	Тип метода сжатия	Размер архива исходного файла, байт	Размер архива обработанного файла плюс размер описания длин строк, байт	Улучшение сжатия, %	Улучшение сжатия без выполнения модификации разделителей, %
Bzip2, вер. 1.00	BWT	109736	103047	6.1	6.0
WinRAR, вер. 2.71	LZ77	130174	120632	7.3	7.8
HA a2, вер. .999c	PPM	108443	99788	8.0	7.3

В табл. 7.9 ожидаемое улучшение сжатия вычислялось как сумма улучшений для всех четырех типов препроцессинга относительно размера архива исходного непреобразованного файла (см. табл. 7.3, 7.5, 7.6, 7.7).

Таблица 7.9

Архиватор	Тип метода сжатия	Улучшение сжатия, %	Ожидаемое улучшение сжатия, %
Bzip2, вер. 1.00	BWT	6.1	6.8
WinRAR, вер. 2.71	LZ77	7.3	7.1
HA a2, вер. 0.999c	PPM	8.0	7.6

Из таблиц видно, что для HA и, в меньшей степени, WinRAR, проявился даже положительный кумулятивный эффект от применения нескольких алгоритмов предварительной обработки. Это достаточно странный на первый взгляд результат, так как чем совершеннее алгоритм сжатия, тем меньший выигрыш должно давать использование дополнительных механизмов, что, собственно, мы и наблюдаем в случае архиватора BZIP2. До некоторой степени это можно объяснить тем, что после преобразования заглавных букв большее количество *n*-графов может быть заменено на соответствующие им индексы словаря, что уменьшает разнообразие используемых строк и способствует увеличению сжатия. Возможно, сочетание словарной замены со специальным кодированием СКС настолько уменьшает общее количество строк, сжимаемых с помощью словаря LZ77, при одновременном уменьшении их фиктивной длины, что это компенсирует падение общего процента улучшения сжатия. Вставка пробелов или замена СКС на пробелы уменьшает количество контекстов и соответственно уменьшает размер PPM-модели, поэтому HA, ограниченный всего лишь примерно 400 Кб памяти, может использовать для оценки большее количество статистики, что улучшает сжатие. Судя по всему, реализация BWT и сопутствующих методов в BZIP2 и принципиальные особенности алгоритма блочной сортировки не позволили BWT "воспользоваться" ситуацией так же эффективно, как LZ77 и PPM.

Рассмотрим, что произойдет при использовании LIPT вместо словарной замены *n*-графов. В табл. 7.10 представлены результаты сжатия преобразованного текста, полученного с помощью трех упоминавшихся техник препроцессинга с последующим использованием LIPT, алгоритм 2 (см. "Использование словарей" в начале подразд. "Препроцессинг текстов").

Таблица 7.10

Архиватор	Тип метода сжатия	Размер архива исходного файла, байт	Размер архива обработанного файла плюс размер описания длин строк, байт	Улучшение сжатия, %	Ожидаемое улучшение сжатия, %
Bzip2, вер. 1.00	BWT	109736	93882	14.4	12.9
WinRAR, вер. 2.71	LZ77	130174	114566	12.0	10.1
HA a2, вер. 0.999c	PPM	108443	94191	13.1	14.9

И опять мы видим, что различные способы препроцессинга дополняют друг друга, обеспечивая рост степени сжатия. Хотя теперь ситуация до некоторой степени поменялась: увеличение сжатия больше ожидаемого для BWT и LZ77, а в случае PPM наблюдается эффект "насыщения". Отметим, что использованная схема предварительной обработки далеко не самая лучшая, если ее предполагается использовать совместно с LZ-компрессором. В этом случае за счет упоминавшихся модификаций можно повысить степень сжатия еще на несколько процентов.

Вывод. Одновременное применение рассмотренных способов предварительной обработки текстов позволяет улучшить сжатие на 5–8 % в случае простой словарной схемы препроцессинга и на 12–15 % при использовании громоздкого словаря.

Препроцессинг нетекстовых данных

Было замечено, что многие файлы содержат данные, записанные не в самом удобном виде для сжатия традиционными универсальными компрессорами. И если изменить форму представления этих данных, то эффективность сжатия файлов заметно увеличится. Многие компрессоры уже оснащены препроцессорами регулярных структур и исполнимых файлов. Среди них можно отметить 7-Zip, ACE, CABARC, DC, IMP, PPMN, SBC, UHARC, YBS.

Преобразование относительных адресов

Как известно, в системе команд процессоров Intel адреса меток в ряде случаев записываются в виде смещения от адреса текущей команды до адреса соответствующей метки. Так записываются команды CALL (код операции 0xE8), JMP (код операции 0xE9). В результате, если ряд команд ссылаются на одну и ту же метку, каждый раз адрес этой метки записывается по-разному.

Главная идея преобразования заключается в замене относительных адресов на абсолютные. Обычно подпрограмма вызывается несколько раз из разных участков программы. Тогда несколько относительных адресов будут преобразованы в один абсолютный адрес, вследствие чего количество различных строк в файле уменьшится, а степень сжатия возрастет. Причем не обязательно получать истинные абсолютные адреса меток. Лишь бы преобразование было обратимо.

Рассмотрим, например, преобразование адресов 32-разрядной команды CALL. Команда записывается в виде последовательности 5 байт:

0xE8	R_0	R_1	R_2	R_3

Относительное смещение R вычисляется по формуле

$$R = R_0 + (R_1 << 8) + (R_2 << 16) + (R_3 << 24).$$

Зная смещение команды CALL от начала файла и относительное смещение R, можно вычислить абсолютный адрес. При этом следует учитывать, что не все символы с кодом 0xE8 являются началом команды CALL. Чтобы отличить настоящие команды, предлагается довольно простой способ, примененный в архиваторе CABARC [3].

Введем следующие обозначения:

C – величина смещения анализируемой команды от начала файла (а именно смещение байта кода операции – 0xE8 или 0xE9);

N – размер файла;

R – смещение метки, на которую указывает операнд команды, относительно самой команды CALL;

A – абсолютный адрес метки.

Разделим все значения относительных смещений на 4 диапазона.

Первым делом определим диапазон значений смещений, которые имеет смысл подвергать преобразованию. Минимальное значение этого диапазона соответствует ссылке команды на начало файла, максимальное – на конец.

Смещения, принимающие значения меньше вышеуказанных, нецелесообразно подвергать преобразованию, поскольку очевидно, что они не принадлежат командам. Это второй диапазон.

Прежде чем определять остальные два диапазона, рассмотрим процесс преобразования относительных значений из первого диапазона в абсолютные:

$$A = R + C.$$

В результате преобразования получим величину, которая может принимать значения от нуля до N–1. Таким образом, в результате преобразования мы отображаем значения из отрезка [–C,N–C) на значения отрезка [0,N).

Для обеспечения возможности однозначного декодирования введем третий диапазон, [N-C,N), над которым будем осуществлять компенсирующее преобразование, т. е. [N-C,N) → [–C,0).

Оставшиеся значения смещений поместим в четвертый диапазон. Эти значения в результате преобразования будут оставаться неизменными.

Преобразование относительных адресов можно представить в виде рис. 7.4. Преобразование относительных значений [–C,N–C) в абсолютные значения [0,N) показано толстой сплошной стрелкой, компенсирующее преобразование – толстой пунктирной.

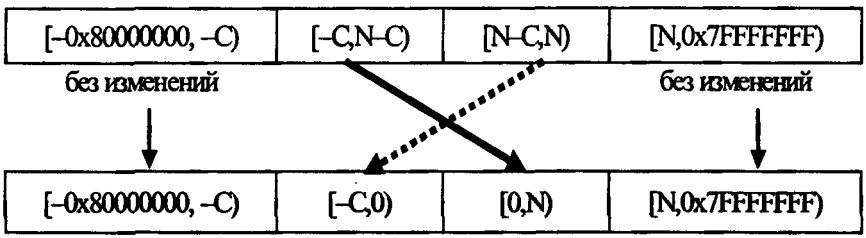


Рис. 7.4. Схема преобразования относительных значений в абсолютные

После преобразования заменим запись команды CALL такой последовательностью:

0xE8	A_0	A_1	A_2	A_3

где

A_0 = A & 0xFF;
A_1 = (A>>8) & 0xFF;
A_2 = (A>>16) & 0xFF;
A_3 = (A>>24) & 0xFF.

Функция преобразования выглядит следующим образом:

```
// in - указатель на найденную команду
// op - адрес в буфере, где записан операнд
// cur_pos - номер позиции команды в файле
// file_size - размер файла

op = (long *)&in[1];
if( *op >= -cur_pos &&
    *op <  file_size - cur_pos ) {
    *op += cur_pos;
} else if( *op  >  0 &&
           *op  < file_size ) {
           *op -= file_size;
}
```

Аналогичное преобразование выполняется и для команды безусловного перехода, JMP. Правда, замена относительных значений адресов для этой команды не так эффективна. В частности, из-за того, что, как правило, число команд относительного перехода на один и тот же абсолютный адрес в среднем меньше, чем количество вызовов одной и той же подпрограммы. А также из-за того, что относительные адреса обычно принимают меньшие значения, чем в случае с командой CALL, т. е. могут быть эффективно сжаты и без какого-либо преобразования. Поэтому преобразование может даже ухудшить сжатие. Решение о применении данного преобразования можно принимать на основе оценки статистики его выполнения. Критерием целесообразности выполнения преобразования может выступать следующее условие: доля компенсирующих преобразований адресов незначительна и в результате преобразования получается большое число совпадающих значений абсолютных адресов.

Сравним влияние описываемого преобразования команд CALL и JMP на сжатие данных компрессорами, представляющими различные методы. В качестве тестового файла был использован исполнимый модуль wcc386.exe из дистрибутива Watcom C 10.0 (табл. 7.11).

Таблица 7.11

Архиватор	Тип метода сжатия	Размер архива, байт	Замена операндов команды CALL		Замена операндов команд CALL и JMP	
			Размер архива, байт	Улучшение сжатия, %	Размер архива, байт	Улучшение сжатия, %
Bzip2, вер 1.00	BWT	308624	291492	5.55	292051	5.37
WinRAR, вер 2.70	LZ77	298959	281584	5.81	280995	6.01
HA a2, вер. 0.999c	PPM	296769	280316	5.54	279959	5.66

Следует отметить, что данное преобразование, хотя и является достаточно простым и эффективным, может быть усовершенствовано для достижения более сильного сжатия. Например, можно заметить, что код операции 0xE8 соседствует с младшим байтом значения абсолютного адреса A_0, в то время как более логичным было бы расположить рядом более связанные друг с другом 0xE8 и A_3. То есть, можно записывать значение абсолютного адреса старшими байтами вперед. Другой способ повышения эффективности заключается в предварительном составлении списка всех процедур, на которые делаются ссылки в программе, и указании номеров процедур в списке вместо адресов.

Упражнение. Напишите процедуру обратного преобразования абсолютного значения адреса в относительное.

Преобразование табличных структур

В файлах зачастую можно встретить регулярные структуры, такие, как, например, различного рода служебные таблицы и т. п. Очевидно, такие структуры требуют особого внимания. Причем необязательно их выносить в отдельный файл и сжимать при помощи специализированного алгоритма. Иногда достаточно преобразовать их в такой вид, который будет приемлем и для универсального архиватора.

Например, рассмотрим преобразование таблиц, представляющих собой упорядоченный по возрастанию список 32-разрядных значений. Чем нам могут помешать эти структуры в первоначальном виде? Статистические характеристики таких табличных последовательностей обычно сильно отличаются от характеристик файла в целом. Например, в таблицах рядом находятся байты, имеющие разный вес в составе 32-разрядных значений и статистически слабо связанные между собой. Таким образом, эти таблицы не только сами хранятся в неудобном для сжатия виде, но и ухудшают сжатие окружающих их данных за счет искажения вероятностных характеристик.

Первая задача, которая встает перед нами, – распознать такие таблицы среди остальных данных. Учтем, что таблицы могут быть произвольного размера и располагаться в любом месте файла. Будем рассматривать какую-то область данных как таблицу при выполнении следующих условий:

- Если нам в файле встретились подряд три 32-разрядных числа, идущие в неубывающем порядке, будем считать это началом таблицы. Например (1-й байт самый младший):
  0x05 0x00 0x80 0x3f
  0x05 0x00 0x80 0x3f
  0x35 0x00 0x80 0x3f
- Если эти числа могут быть закодированы при помощи RLE, отказываемся от применения преобразования таблиц.
- Концом таблицы будем считать число, которое превышает предыдущее более чем на 2^8 - 2 или меньше предыдущего.

Выполним преобразование таких таблиц следующим образом:

- Первые 3 числа таблицы записываем в неизменном виде.
- Для записи остальных чисел нам требуется только величина, на которую каждое число отличается от предыдущего. Для записи этой разницы достаточно 1 байта.
- В качестве признака конца таблицы записываем дополнительно символ с кодом 2^8 - 1 = 0xFF.

В качестве тестового файла возьмем тот же исполнимый модуль wcc386.exe из дистрибутива Watcom C 10.0.

Таблица 7.12

Архиватор	Тип метода сжатия	Размер архива, байт	Преобразование 32-разрядных таблиц	
			Размер архива, байт	Улучшение сжатия, %
Bzip2, вер 1.00	BWT	308624	306791	0.59
WinRAR, вер 2.70	LZ77	298959	298342	0.21
HA a2, вер. 0.999c	PPM	296769	295240	0.52

Можно заметить, что для архиватора, использующего LZ77, выигрыш в сжатии оказался существенно меньше, чем для других методов. Это объясняется особенностью алгоритма кодирования указателей, используемого в **WinRAR, позволяющего эффективно обрабатывать такого рода структуры.**

Описанный алгоритм не является самым эффективным с точки зрения сжатия, но позволяет оценить полезные свойства такого рода преобразований. Способов улучшения довольно много. Например, большие таблицы следует кодировать отдельно от всего файла. Кроме того, помимо 32-разрядных таблиц, в файле могут присутствовать и, например, 16-разрядные; в таблицах могут находиться как возрастающие, так и убывающие последовательности чисел и т. п. И надо сказать, зачастую суммарный эффект от применения такого рода преобразований оказывается довольно ощутимым.

Упражнение. Придумайте способ преобразования для 16-разрядных таблиц.

Вопросы для самоконтроля[1]

1. Почему невыгодно включение длинных фраз в словарь *n*-графов?
2. Каковы недостатки и преимущества динамического составления словаря *n*-графов?
3. Объясните, почему использование при словарной замене фраз, содержащих пробелы, приводит к уменьшению эффективности предварительной обработки в случае алгоритмов класса PPM и BWT.
4. Почему при организации LIPT для записи индекса фраз удобно использовать только те символы, которые входят в словарь букв?

[1] *Ответы на вопросы, выполненные упражнения и исходные тексты программ вы можете найти на http://compression.graphicon.ru/.*

5. В каких случаях при специальном кодировании символов конца строки выгоднее указывать длину строки не через количество символов, а через количество пробелов?
6. Почему при преобразовании относительного адреса подпрограмм, вызываемых командой CALL, обычно выгодно записывать значение абсолютного адреса старшими байтами вперед?
7. Почему преобразование относительных адресов для команды JUMP, как правило, менее эффективно, чем для CALL?

Литература

1. Awan F., Motgi N. Zh. N., Iqbal R., Mukherjee A. LIPT: A Reversible Lossless Text Transform to Improve Compression Performance // Proceedings of Data Compression Conference, Snowbird, Utah, March 2001.
2. Grabowski Sz. Text preprocessing forBurrows-Wheeler block sorting compression // VIIKonferencja "Sieci i Systemy Informatyczne" (7thConference "Networks and IT Systems"). Lodz. Oct. 1999. Conf. proc. P. 229 – 239. http://www.dogma.net/DataCompression /ArchiveFormats/BWT _paper.rtf.
3. Microsoft Corporation. Microsoft LZX Data Compression Format. 1997.
4. Teahan W.J. Modelling English Texts // PhD thesis, Department of Computer Science. The University of Waikato. Hamilton, New Zealand. May 1998.

Выбор метода сжатия

В заключение разд. скажем несколько слов о процедуре выбора метода сжатия.

Выбор метода – это первая задача, которую должен решить разработчик программных средств сжатия данных. Выбор зависит от типа данных, которые предстоит обрабатывать, аппаратных ресурсов, требований к степени сжатия и ограничений на время работы программы. Дадим ряд рекомендаций, предполагая, что необходимо сжимать качественные данные, между элементами которых имеются сильные и достаточно протяженные статистические связи, т. е. данные порождены источником с памятью.

Прежде всего, следует учитывать, что каждый из рассмотренных в разделе универсальных методов сжатия данных источников с памятью допускает модификации, позволяющие существенно изменить параметры компрессора. Так, увеличение порядка модели PPM приводит к заметному усилению сжатия ценой замедления работы и увеличения расхода памяти. К аналогичному результату приводит увеличение размера словаря LZ77-методов, но при этом время разжатия остается практически неизменным. Свойства компрессоров на основе BWT варьируются в меньшей степени.

Ограничения на время работы программы возникают в зависимости от условий, в которых предстоит работать архиватору. Например, при сжатии данных для формирования дистрибутива программного обеспечения ограничение на время компрессии практически отсутствует, в то время как требуется максимально уменьшить время восстановления исходных данных. При резервном сохранении данных положение вещей обратно, поскольку ситуация, когда требуется выполнить разжатие, довольно редка. Необходимость оперативной передачи данных по сети обусловливает одинаковые требования к временам сжатия и разжатия; если данные передаются нескольким абонентам, некоторое преимущество имеют архиваторы, обладающие более быстрым алгоритмом разжатия.

Методы семейства LZ77 обладают наибольшей скоростью декомпрессии. Превышение над скоростью сжатия при использовании метода Хаффмана для кодирования результатов работы LZ77-метода – десятикратное. Меньшая разница у методов на основе BWT – в среднем скорость разжатия в 2–4 раза выше скорости сжатия. Декодирование при использовании PPM на 5–10% медленнее кодирования. Компрессоры на базе частичных сортирующих преобразований малого порядка характеризуются еще большим отставанием разжатия – на некоторых файлах оно в несколько раз медленнее сжатия.

Похожая картина наблюдается, если сравнивать использование памяти при декодировании. В случае применения LZ77 расходы памяти минимальны. Архиваторы на основе PPM наиболее требовательны – им необходимо столько же памяти, сколько и при кодировании. Следует отметить, что при сжатии требования к памяти у программ-представителей разных методов примерно близки, хотя и могут изменяться для разных типов сжимаемых данных.

Таким образом, если можно пренебречь степенью сжатия, методы семейства LZ77 наиболее эффективны для создания дистрибутивов, а методы на основе частичных сортирующих преобразований – для резервного копирования.

Типичные данные качественного характера можно условно разделить на 4 типа:

1) однородные данные (например, типичные тексты);
2) однородные данные с большой избыточностью в виде длинных повторяющихся строк (например, набор исходных текстов);
3) неоднородные данные, в которых имеется выраженная нестабильность контекстно-зависимой статистики (например, исполнимые файлы);

4) данные с малой избыточностью (например, файлы, содержащие уже сжатые блоки).

Рассмотрим, как ведут себя различные методы при сжатии данных разных типов. При анализе будем ориентироваться на наилучших представителей методов.

Для сжатия таких однородных данных, как тексты на естественных языках, наиболее подходящими являются PPM-методы и методы на основе BWT. Первые позволяют достичь большей степени сжатия, вторые обладают большей скоростью декодирования. Программы, использующие методы семейства LZ77, сжимают указанные данные заметно хуже и, при увеличении длины словаря, существенно медленнее.

Если в однородных данных есть длинные повторяющиеся строки, у программ на основе LZ77 есть шанс себя реабилитировать. Впрочем, в этом случае наиболее выгодным будет использовать гибрид LZ77 и любого из двух его конкурентов или применять LZ77-препроцессинг.

При сжатии неоднородных данных пальма первенства принадлежит семейству методов LZ77, которые при сохранении своих высоких скоростных качеств настигают по степени сжатия заметно более медленных лучших представителей PPM-компрессоров. Если не использовать фрагментирование, BWT-компрессоры показывают не очень высокую степень сжатия неоднородных данных.

На данных с малой избыточностью все методы выступают не лучшим образом. Некоторое преимущество в степени сжатия имеют архиваторы на основе LZ77 и PPM. Но последние при этом требуют значительного расхода памяти и скорость их работы заметно падает.

В заключение приведем таблицу, в которой описаны свойства методов при сжатии качественных данных различных типов.

Параметр	Метод	Однородные данные (типичный текст)	Однородные данные с большой избыточностью (исходные тексты программ)	Неоднородные данные	Данные с малой избыточностью
Степень сжатия	PPM	Высокая	Высокая	Высокая	Невысокая
	BWT	Близкая к PPM	Близкая к PPM	Без фрагментирования – худшая	"
	LZ77	Заметно худшая	При большом количестве длинных повторов довольно высокая	Близкая к PPM	"

Параметр	Метод	Однородные данные (типичный текст)	Однородные данные с большой избыточностью (исходные тексты программ)	Неоднородные данные	Данные с малой избыточностью
Скорость кодирования	BWT	Высокая	Средняя	Высокая	Высокая
	PPM	При большом порядке модели – самая низкая; при небольшом – немного быстрее BWT	Если не использовать сложное моделирование – высокая	Средняя	Низкая
	LZ77	Средняя, а при малом словаре – самая высокая	Средняя, при малом словаре – высокая	Высокая	Высокая
Скорость декодирования	LZ77	Примерно в 10 раз выше скорости кодирования; разница еще больше на избыточных данных			
	PPM	Обычно на 5–10% медленнее кодирования			
	BWT	В 2–4 раза выше скорости кодирования			
Требуемый объем памяти при сжатии	BWT	Постоянный при сжатии данных любого типа			
	PPM	Варьируется в широких пределах, в зависимости от сложности моделирования и порядка модели; вырастает для очень неоднородных данных; в зависимости от структуры хранения контекстной информации может увеличиваться для малоизбыточных данных			
	LZ77	Пропорционален размеру словаря			
Требуемый объем памяти при разжатии	LZ77	Минимальный			
	PPM	Максимальный; если процесс моделирования симметричен, то примерно равен расходу памяти при сжатии			
	BWT	Средний			

РАЗДЕЛ 2
АЛГОРИТМЫ СЖАТИЯ ИЗОБРАЖЕНИЙ

Введение

Алгоритмы сжатия изображений – бурно развивающаяся область машинной графики. Основной объект приложения усилий в ней – изображения – своеобразный тип данных, характеризуемый тремя особенностями.

1. Изображение (как и видео) *обычно требует для хранения гораздо большего объема памяти, чем текст.* Так, скромная не очень качественная иллюстрация на обложке книги размером 500x800 точек занимает 1,2 Мб – столько же, сколько художественная книга из 400 страниц (60 знаков в строке, 42 строки на странице). В качестве примера можно рассмотреть также, сколько тысяч страниц текста мы сможем поместить на CD-ROM и как мало там поместится несжатых фотографий высокого качества. Эта особенность изображений **определяет актуальность алгоритмов архивации** графики.
2. Второй особенностью изображений является то, что человеческое зрение при анализе изображения оперирует контурами, общим переходом цветов и сравнительно нечувствительно к малым изменениям в изображении. Таким образом, мы можем создать эффективные алгоритмы архивации изображений, в которых декомпрессированное изображение не будет совпадать с оригиналом, однако человек этого не заметит. **Данная особенность человеческого зрения позволила создать специальные алгоритмы сжатия, ориентированные только на изображения.** Эти алгоритмы позволяют сжимать изображения с высокой степенью сжатия и незначительными с точки зрения человека потерями.
3. Мы можем легко заметить, что изображение в отличие, например, от текста обладает избыточностью в двух измерениях. То есть как правило, соседние точки, как по горизонтали, так и по вертикали, в изображении близки по цвету. Кроме того, мы можем использовать подобие между цветовыми плоскостями R, G и B в наших алгоритмах, что дает возможность создать еще более эффективные алгоритмы. Таким образом, **при создании алгоритма компрессии графики мы используем особенности структуры изображения.**

Всего на данный момент известно минимум 3 семейства алгоритмов, которые разработаны исключительно для сжатия изображений, и применяемые в них методы практически невозможно применить к архивации еще каких-либо видов данных.

Для того чтобы говорить об алгоритмах сжатия изображений, мы должны ответить на несколько важных вопросов:

1. Какие критерии мы можем предложить для сравнения различных алгоритмов?
2. Какие классы изображений существуют?
3. Какие классы приложений, использующих алгоритмы компрессии графики, существуют и какие требования они предъявляют к алгоритмам?

Рассмотрим эти вопросы подробнее.

Классы изображений

Статические растровые изображения представляют собой двумерный массив чисел. Элементы этого массива называют *пикселами* (от английского pixel – picture element). Все изображения можно подразделить на две группы – с палитрой и без нее. У изображений с палитрой в пикселе хранится число – индекс в некотором одномерном векторе цветов, называемом *палитрой*. Чаще всего встречаются палитры из 16 и 256 цветов.

Изображения без палитры бывают в какой-либо системе цветопредставления и *в градациях серого* (grayscale). Для последних значение каждого пиксела интерпретируется как яркость соответствующей точки. Чаще всего встречаются изображения с 2, 16 и 256 уровнями серого. Одна из интересных практических задач заключается в приведении цветного или черно-белого изображения к двум градациям яркости, например для печати на лазерном принтере. При использовании некой *системы цветопредставления* каждый пиксел представляет собой запись (структуру), полями которой являются компоненты цвета. Самой распространенной является *система RGB*, в которой цвет передается значениями интенсивности красной (R), зеленой (G) и синей (B) компонент. Существуют и другие системы цветопредставления, такие, как CMYK, CIE XYZccir60-1, YVU, YCrCb и т. п. Ниже мы увидим, как они используются при сжатии изображений с потерями.

Для того чтобы корректнее оценивать степень сжатия, нужно ввести понятие *класса изображений*. Под классом будет пониматься совокупность изображений, применение к которым алгоритма архивации дает качественно одинаковые результаты. Например, для одного класса алгоритм дает очень высокую степень сжатия, для другого – почти не сжимает, для третьего – увеличивает файл в размере. (Например, все алгоритмы сжатия без потерь в худшем случае увеличивают файл.)

Дадим неформальное определение наиболее распространенных классов изображений.

Класс 1. Изображения с небольшим количеством цветов (4–16) и большими областями, заполненными одним цветом. Плавные переходы цветов отсутствуют. *Примеры*: деловая графика – диаграммы, графики и т. п.

Класс 2. Изображения с плавными переходами цветов, построенные на компьютере. *Примеры*: графика презентаций, эскизные модели в САПР, изображения, построенные по методу Гуро.

Класс 3. Фотореалистичные изображения. *Пример*: отсканированные фотографии.

Класс 4. Фотореалистичные изображения с наложением деловой графики. *Пример*: реклама.

Развивая данную классификацию, в качестве отдельных классов могут быть предложены некачественно отсканированные в 256 градаций серого цвета страницы книг или растровые изображения топографических карт. (Заметим, что этот класс не тождественен классу 4). Формально являясь 8- или 24-битовыми, они несут не растровую, а чисто векторную информацию. Отдельные классы могут образовывать и совсем специфичные изображения: рентгеновские снимки или фотографии в профиль и фас из электронного досье.

Достаточно сложной и интересной задачей является поиск наилучшего алгоритма для конкретного класса изображений.

Итог. Нет смысла говорить о том, что какой-то алгоритм сжатия лучше другого, если мы не обозначили **классы изображений**, на которых сравниваются наши алгоритмы.

Классы приложений

Примеры приложений, использующих алгоритмы компрессии графики

Рассмотрим следующую простую классификацию приложений, использующих алгоритмы компрессии:

Класс 1. Характеризуются высокими требованиями ко времени архивации и разархивации. Нередко требуется просмотр уменьшенной копии изображения и поиск в базе данных изображений. *Примеры*: издательские системы в широком смысле этого слова, причем как готовящие качественные публикации (журналы) с заведомо высоким качеством изображений и использованием алгоритмов архивации без потерь, так и готовящие газеты, и WWW-серверы, где есть возможность оперировать изображениями меньшего качества и использовать алгоритмы сжатия с потерями. В подобных системах приходится иметь дело с полноцветными изображениями самого разного размера (от 640x480 – формат цифрового фотоаппарата, до

3000x2000) и с большими двуцветными изображениями. Поскольку иллюстрации занимают львиную долю от общего объема материала в документе, проблема хранения стоит очень остро. Проблемы также создает большая разнородность иллюстраций (приходится использовать универсальные алгоритмы). Единственное, что можно сказать заранее, – это то, что будут преобладать фотореалистичные изображения и деловая графика.

Класс 2. Характеризуется высокими требованиями к степени архивации и времени разархивации. Время архивации роли не играет. Иногда подобные приложения также требуют от алгоритма компрессии легкости масштабирования изображения под конкретное разрешение монитора у пользователя. *Пример*: справочники и энциклопедии на CD-ROM. С появлением большого количества компьютеров, оснащенных этим приводом (в США уровень в 50% машин достигнут еще в 1995 г.), достаточно быстро сформировался рынок программ, выпускаемых на лазерных дисках. Несмотря на то что емкость одного диска довольно велика (примерно 650 Мб), ее, как правило, не хватает. При создании энциклопедий и игр бóльшую часть диска занимают статические изображения и видео. Таким образом, для этого класса приложений актуальность приобретают существенно асимметричные по времени алгоритмы (*симметричность по времени* – отношение времени компрессии ко времени декомпрессии).

Класс 3. Характеризуется очень высокими требованиями к степени архивации. Приложение клиента получает от сервера информацию по сети. *Пример*: новая быстро развивающаяся система "Всемирная информационная паутина" – WWW. В этой гипертекстовой системе достаточно активно используются иллюстрации. При оформлении информационных или рекламных страниц хочется сделать их более яркими и красочными, что, естественно, сказывается на размере изображений. Больше всего при этом страдают пользователи, подключенные к сети с помощью медленных каналов связи. Если страница WWW перенасыщена графикой, то ожидание ее полного появления на экране может затянуться. Поскольку при этом нагрузка на процессор мала, то здесь могут найти применение эффективно сжимающие сложные алгоритмы со сравнительно большим временем разархивации. Кроме того, мы можем видоизменить алгоритм и формат данных так, чтобы просматривать огрубленное изображение файла до его полного получения.

Можно привести множество более узких классов приложений. Так, свое применение машинная графика находит и в различных информационных системах. Например, уже становится привычным исследовать ультразвуковые и рентгеновские снимки не на бумаге, а на экране монитора. Постепенно в электронный вид переводят и истории болезней. Понятно, что хранить эти материалы логичнее в единой картотеке. При этом без использования

специальных алгоритмов бóльшую часть архивов займут фотографии. *Поэтому при создании эффективных алгоритмов решения этой задачи нужно учесть специфику рентгеновских снимков – преобладание размытых участков.*

В геоинформационных системах – при хранении аэрофотоснимков местности – специфическими проблемами являются большой размер изображения и *необходимость выборки лишь части изображения по требованию.* Кроме того, может потребоваться *масштабирование.* Это неизбежно накладывает свои ограничения на алгоритм компрессии.

В электронных картотеках и досье различных служб для изображений характерно подобие между фотографиями в профиль и подобие между фотографиями в фас, которое также необходимо учитывать при создании алгоритма архивации. *Подобие между фотографиями* наблюдается и в любых других специализированных справочниках. В качестве примера можно привести энциклопедии птиц или цветов.

Итог. Нет смысла говорить о том, что какой-то конкретный алгоритм компрессии лучше другого, если мы не обозначили **класс приложений**, относительно которого мы эти алгоритмы собираемся сравнивать.

Требования приложений к алгоритмам компрессии

В предыдущем подразделе мы определили, какие приложения являются потребителями алгоритмов архивации изображений. Однако заметим, что приложение определяет характер использования изображений (либо большое количество изображений хранится и используется, либо изображения скачиваются по сети, либо изображения велики по размерам и нам необходима возможность получения лишь части...). Характер использования изображений задает степень важности нижеследующих противоречивых требований к алгоритму.

Высокая степень компрессии. Заметим, что далеко не для всех приложений актуальна высокая степень компрессии. Кроме того, некоторые алгоритмы дают лучшее соотношение качества к размеру файла при высоких степенях компрессии, однако проигрывают другим алгоритмам при низких степенях.

Высокое качество изображений. Выполнение этого требования напрямую противоречит выполнению предыдущего.

Высокая скорость компрессии. Это требование для некоторых алгоритмов с потерей информации является взаимоисключающим с первыми двумя. Интуитивно понятно, что чем больше времени мы будем анализировать изображение, пытаясь получить наивысшую степень компрессии, тем лучше будет результат. И соответственно, чем меньше мы времени потра-

тим на компрессию (анализ), тем ниже будет качество изображения и больше его размер.

Высокая скорость декомпрессии. Достаточно универсальное требование, актуальное для многих приложений. Однако можно привести примеры приложений, где время декомпрессии далеко не критично.

Масштабирование изображений. Данное требование подразумевает легкость изменения размеров изображения до размеров окна активного приложения. Дело в том, что одни алгоритмы позволяют легко масштабировать изображение прямо во время декомпрессии, в то время как другие не только не позволяют легко масштабировать, но и увеличивают вероятность появления неприятных артефактов после применения стандартных алгоритмов масштабирования к декомпрессированному изображению. Например, можно привести пример "плохого" изображения для алгоритма JPEG – это изображение с достаточно мелким регулярным рисунком (пиджак в мелкую клетку). Характер вносимых алгоритмом JPEG искажений таков, что уменьшение или увеличение изображения может дать неприятные эффекты.

Возможность показать огрубленное изображение (низкого разрешения), использовав только начало файла. Данная возможность актуальна для различного рода сетевых приложений, где перекачивание изображений может занять достаточно большое время и желательно, получив начало файла, корректно показать preview. Заметим, что примитивная реализация указанного требования путем записывания в начало изображения его уменьшенной копии заметно ухудшит степень компрессии.

Устойчивость к ошибкам. Данное требование означает локальность нарушений в изображении при порче или потере фрагмента передаваемого файла. Данная возможность используется при широковещании (broadcasting – передача по многим адресам) изображений по сети, т. е. в тех случаях, когда невозможно использовать протокол передачи, повторно запрашивающий данные у сервера при ошибках. Например, если передается видеоряд, то было бы неправильно использовать алгоритм, у которого сбой приводил бы к прекращению правильного показа всех последующих кадров. Данное требование противоречит высокой степени архивации, поскольку интуитивно понятно, что мы должны вводить в поток избыточную информацию. Однако для разных алгоритмов объем этой избыточной информации может существенно отличаться.

Учет специфики изображения. Более высокая степень сжатия для класса изображений, которые статистически чаще будут применяться в нашем приложении. В предыдущих подразделах это требование уже обсуждалось.

Редактируемость. Под редактируемостью понимается минимальная степень ухудшения качества изображения при его повторном сохранении

после редактирования. Многие алгоритмы с потерей информации могут существенно испортить изображение за несколько итераций редактирования.

Небольшая стоимость аппаратной реализации. Эффективность программной реализации. Данные требования к алгоритму реально предъявляют не только производители игровых приставок, но и производители многих информационных систем. Так, декомпрессор фрактального алгоритма очень эффективно и коротко реализуется с использованием технологии MMX и распараллеливания вычислений, а сжатие по стандарту CCITT Group 3 легко реализуется аппаратно.

Очевидно, что для конкретной задачи нам будут очень важны одни требования и менее важны (и даже абсолютно безразличны) другие.

Итог. На практике для каждой задачи мы можем сформулировать **набор приоритетов из требований**, изложенных выше, который и определит наиболее подходящий в наших условиях алгоритм (либо набор алгоритмов) для ее решения.

Критерии сравнения алгоритмов

Заметим, что характеристики алгоритма относительно некоторых требований приложений, сформулированные выше, зависят от конкретных условий, в которые будет поставлен алгоритм. Так, *степень компрессии* зависит от того, на каком классе изображений алгоритм тестируется. Аналогично *скорость компрессии* нередко зависит от того, на какой платформе реализован алгоритм. Преимущество одному алгоритму перед другим может дать, например, возможность использования в вычислениях алгоритма технологий нижнего уровня, типа MMX, а это возможно далеко не для всех алгоритмов. Так, JPEG существенно выигрывает от применения технологии MMX, а LZW нет. Кроме того, нам придется учитывать, что некоторые алгоритмы распараллеливаются легко, а некоторые нет.

Таким образом, **невозможно составить универсальное сравнительное описание известных алгоритмов**. Это можно сделать только для типовых классов приложений при условии использования типовых алгоритмов на типовых платформах. Однако такие данные необычайно быстро устаревают.

Так, например, еще в 1994 г., интерес к **показу огрубленного изображения**, используя только начало файла (требование 6), был чисто абстрактным. Реально эта возможность практически нигде не требовалась, и класс приложений, использующих данную технологию, был крайне невелик. Со взрывным распространением Интернета, который характеризуется передачей изображений по сравнительно медленным каналам связи, использо-

вание Interlaced GIF (алгоритм LZW) и Progressive JPEG (вариант алгоритма JPEG), реализующих эту возможность, резко возросло. То, что новый алгоритм (например, wavelet) поддерживает такую возможность, существеннейший плюс для него сегодня.

В то же время мы можем рассмотреть такое редкое на сегодня требование, как **устойчивость к ошибкам**. Можно предположить, что в скором времени (через 5–10 лет) с распространением широковещания в сети Интернет для его обеспечения будут использоваться именно алгоритмы, устойчивые к ошибкам, даже не рассматриваемые в сегодняшних статьях и обзорах.

Со всеми сделанными выше оговорками, выделим несколько наиболее важных для нас критериев сравнения алгоритмов компрессии, которые и будем использовать в дальнейшем. Как легко заметить, мы будем обсуждать меньше критериев, чем было сформулировано выше. Это позволит избежать лишних деталей при кратком изложении данного материала.

Худшая, средняя и лучшая степень сжатия. То есть доля, на которую возрастет изображение, если исходные данные будут наихудшими; некая среднестатистическая степень для того класса изображений, *на который ориентирован алгоритм*; и, наконец, лучшая степень. Последняя необходима лишь теоретически, поскольку показывает степень сжатия наилучшего (как правило, абсолютно черного) изображения, иногда фиксированного размера.

Класс изображений, на который ориентирован алгоритм. Иногда указано также, почему на других классах изображений получаются худшие результаты.

Симметричность. Отношение характеристики алгоритма кодирования к аналогичной характеристике при декодировании. Характеризует ресурсоемкость процессов кодирования и декодирования. Для нас наиболее важной является симметричность по времени: отношение времени кодирования ко времени декодирования. Иногда нам потребуется симметричность по памяти.

Есть ли потери качества? И если есть, то за счет чего изменяется степень сжатия? Дело в том, что у большинства алгоритмов сжатия с потерей информации существует возможность изменения степени сжатия.

Характерные особенности алгоритма и изображений, к которым его применяют. Здесь могут указываться наиболее важные для алгоритма свойства, которые могут стать определяющими при его выборе.

Используя данные критерии, приступим к рассмотрению алгоритмов архивации изображений.

Прежде чем непосредственно начать разговор об алгоритмах, хотелось бы сделать оговорку. Один и тот же алгоритм часто можно реализовать разными способами. Многие известные алгоритмы, такие, как RLE, LZW или JPEG, имеют десятки различающихся реализаций. Кроме того, у алгоритмов бывает несколько явных параметров, варьируя которые можно изменять характеристики процессов архивации и разархивации. (См. примеры в подразделе о форматах). При конкретной реализации эти параметры фиксируются, исходя из наиболее вероятных характеристик входных изображений, требований на экономию памяти, требований на время архивации и т. д. Поэтому у алгоритмов одного семейства лучшая и худшая степени сжатия могут отличаться, но качественно картина не изменится.

Методы обхода плоскости

Задача обхода плоскости возникает при обработке двумерных данных. Цель: создание одномерного массива D из двумерного массива S. Причем если предполагается последующее сжатие D, то желательно создавать его так, чтобы "разрывов" было как можно меньше: каждый следующий элемент D_i, заносимый в D на i-м шаге, является соседним (в плоскости) для предыдущего, занесенного в D на (i-1)-м шаге, D_{i-1}.

Змейка (зигзаг-сканирование)

Обход массива S начинается с одного угла плоскости, заканчивается в противоположном по диагонали. Например, из левого верхнего в правый нижний:

1	2	6	7	15	16
3	5	8	14	17	24
4	9	13	18	23	25
10	12	19	22	26	29
11	20	21	27	28	30

На иллюстрации показан порядок выборки элементов из плоскости (с последующим занесением в одномерный массив). Значение из ячейки массива S, помеченной на рисунке номером i, заносится в $D[i]$.

Змейка выгодна в случаях, когда в одном из углов "особенность", например сосредоточены самые крупные коэффициенты. Применяется в алгоритме JPEG для обхода квадрантов (размером 8x8 точек).

Обход строками

Самый тривиальный метод. Именно он используется в самых распространенных графических форматах (BMP, TGA, RAS...) для хранения элементов изображений.

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30

В варианте строк с разворотами для каждой второй строки делаем выборку в обратном направлении:

1	2	3	4	5	6
12	11	10	9	8	7
13	14	15	16	17	18
24	23	22	21	20	19
25	26	27	28	29	30

Точек "разрыва" нет, в отличие от варианта без разворотов.

Совершенно аналогично можно делать обход столбцами.

Упражнение. Нарисуйте пример обхода плоскости столбцами с разворотами.

Обход полосами

Чаще всего сжатие лучше, если каждая область двумерного массива S не рассредоточена (равномерно "размазана") по всему одномерному D, а сконцентрирована в D компактно. В случае обхода строками понятие "области" отсутствует: каждый элемент считается "областью". Пытаясь обходить плоскость квадратами размером NxN, приходим к идее обхода горизонтальными "полосами" шириной N:

1	4	7	10	13
2	5	8	11	14
3	6	9	12	15
16	19	22	25	28
17	20	23	26	29
18	21	24	27	30

В данном примере ширина полосы N=3. Если N=1, получаем обход строками.

Полосами с разворотами

То же самое, но с разворотами и столбцов внутри полос, и направлений самих полос:

1	6	7	12	13
2	5	8	11	14
3	4	9	10	15
28	27	22	21	16
29	26	23	20	17
30	25	24	19	18

Разрывов опять нет, но теперь еще и каждая точка принадлежит к области, записанной в *D* **компактно**, без разрывов: ее элементы расположены внутри одного интервала (*D*[*i*], *D*[*i*+1], ..., *D*[*i*+*j*]) и элементов из других областей внутри этого интервала нет. Примеры таких областей – каждый из четырех углов размером 3x3 элемента.

Упражнение. Нарисуйте схему обхода квадрата 7x7 полосами шириной 3 с разворотами. Какой вариант лучше: 3+3+1 или 3+2+2? Какие области заносятся в *D* компактно?

Обход решетками

Для первой порции берем элементы из каждого N-го столбца каждой M-й строки. Для второй – то же, но со сдвигом на один столбец. Так же и для следующих, а затем – со сдвигом на одну, две, (M–1) строки. Например, если M=N=2, то имеем 4 порции:

1	13	2	14	3	15	4	16
25	37	26	38	27	39	28	40
5	17	6	18	7	19	8	20
29	41	30	42	31	43	32	44
9	21	10	22	11	23	12	24
33	45	34	46	35	47	36	48

То есть плоскость разбивается на прямоугольники размера MxN, задается обход плоскости прямоугольниками, а также обход внутри самих прямоугольников и далее делается "одновременный" обход по каждому из них: сначала выбираются их первые элементы, затем вторые, третьи, и т. д., до последнего.

Обход решетками с учетом значений элементов

После того как обработаны первые элементы прямоугольников (в нашем примере – квадратов 2x2), если предположить, что они являются атрибутами своих областей, выгодно (для улучшения сжатия) группировать области

с одинаковыми атрибутами, т. е. с одинаковыми значениями этих первых элементов.

Допустим, атрибуты распределены так:

R		G		G		G	
L		R		G		G	
L		L		R		R	

Тогда имеет смысл дальше действовать так:

R	13	G	25	G	28	G	31
15	14	27	26	30	29	33	32
L	40	R	16	G	34	G	37
42	41	18	17	36	35	39	38
L	43	L	46	R	19	R	22
45	44	48	47	21	20	24	23

То есть сначала обходим квадраты с атрибутом "R", затем с атрибутом "G" и, наконец, с "L".

Контурный обход

Часть элементов принадлежит к одной группе, часть – к другой, причем контур задан:

1	1	1	1	1	1	1	1
1	1	2	2	1	1	1	1
1	2	2	2	2	1	1	1
1	2	2	2	2	2	1	1
1	1	1	2	1	1	1	1
1	1	1	1	1	1	1	1

36 элементов – из группы "1", а 12 – из группы "2".

Очевидно, что имеет смысл отдельно оформить элементы группы "1":

1	29	28	27	26	22	21	31
2	30			25	23	20	32
3					24	19	33
4						18	34
5	8	9		13	14	17	35
6	7	10	11	12	15	16	36

и затем точно так же остальные элементы, принадлежащие к группе "2".

Контурный обход с неизвестными контурами

Рассмотрим предыдущий пример, т. е. такое же распределение элементов групп по плоскости, но изначально, при начале обхода плоскости, это распределение неизвестно. Будем действовать таким методом:

1	44	41	40	35	34	29	28
2	43	42	39	36	33	30	27
3	45	46	38	37	32	31	26
4	9	10	47	48	19	20	25
5	8	11	14	15	18	21	24
6	7	12	13	16	17	22	23

Обходим плоскость "столбцами с разворотами" и, обнаруживая элемент другой группы (в элементах 9, 10, 14...), также делаем разворот на 180°. Затем (шаги 45–48) обходим оставшуюся часть плоскости, содержащую (предположительно) элементы другой группы.

В итоге имеем:

- среди первых 36 элементов 4 из группы "2", а 32 из группы "1";
- из последних 12 элементов 8 из группы "2", а 4 из группы "1".

В первой части 4/36=1/9 "исключений", во второй – 4/12=1/3.

А если бы делали просто обход "столбцами с разворотами":

1	12	13	24	25	36	37	48
2	11	14	23	26	35	38	47
3	10	15	22	27	34	39	46
4	9	16	21	28	33	40	45
5	8	17	20	29	32	41	44
6	7	18	19	30	31	42	43

В итоге:

- среди первых 33 элементов 12 из группы "2", а 21 из группы "1";
- из последних 15 элементов все из группы "1".

То есть в большей части получившегося блока – более чем 1/3 "исключений".

"Квадратная змейка"

Рекурсивный метод для квадратных областей.

Если принять левый верхний элемент за первый, то для **квадрата 2x2** возможны два варианта обхода без разрывов:

1	4
2	3

и

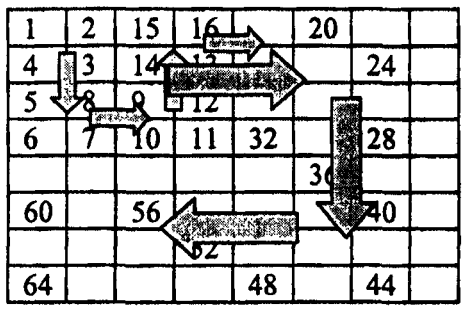
1	2
4	3

То есть либо первый переход внутри квадрата был сверху **вниз**, тогда пятым шагом будет переход к левому верхнему элементу квадрата **справа**

(или к правому нижнему элементу квадрата **сверху**). Либо, наоборот, первый переход внутри квадрата был **вправо**, тогда пятым шагом будет переход к квадрату **снизу** (или **слева**). Переформулируем так:

- если нужно выйти **к правому** (или верхнему) квадрату, то первый шаг – **вниз**, если **к нижнему** (или левому), то первый шаг – **вправо**;
- пройденным путем однозначно задается, к какому квадрату нужно выйти в каждый конкретный момент;
- только в самом начале есть выбор одного из двух вариантов обхода.

Например:



Первый шаг внутри квадрата 2x2 был вправо, – значит, в результате обхода этого квадрата 2x2 мы должны выйти к нижнему квадрату 2x2.

Первый шаг перехода от 2x2 к 2x2 внутри 4x4 был вниз, – значит, мы должны выйти к правому 4x4.

Первый переход внутри квадрата 16x16 был вправо, – значит, в результате обхода 16x16 мы должны прийти к нижнему, и т. д.

Разрывов нет, и каждый элемент принадлежит к области, записанной компактно.

Упражнение. Определите, какие числа должны быть в незаполненных клетках примера. Нарисуйте другой пример: при выборе второго элемента квадрата делаем переход не слева направо, а сверху вниз.

Для квадрата 3x3 можно взять такие два "шаблона":

1	4	5
2	3	6
9	8	7

и

1	2	9
4	3	8
5	6	7

Первое правило будет "противоположным" первому правилу случая 2x2, но остальные два – такие же:

- если нужно выйти **к правому** (или верхнему) квадрату, то первый шаг – **вправо**, если **к нижнему** (или левому), то первый шаг – **вниз**;

- пройденным путем однозначно задается, к какому квадрату нужно выйти в каждый конкретный момент;
- только в самом начале есть выбор одного из двух вариантов.

Но для 3x3 есть еще и такие варианты:

1	6	7
2	5	8
3	4	9

и

1	2	3
6	5	4
7	8	9

Видно, что они совершенно эквивалентны, т. е. взаимозаменяемы, поскольку в обоих случаях мы выходим в правый нижний угол. Точно так же эквивалентны и два варианта обхода ими квадрата 9x9:

1					18	19		
		9	10					27
		46	45					28
54					37	36		
55					72	73		
		63	64					81

и

1					54	55		
		9	46					63
		10	45					64
18					37	72		
19					36	73		
		27	28					81

Совершенно аналогично для 5x5, и затем 25x25 и т. д.

Если требуется обойти квадрат размера NxN, сначала определяем, произведением каких простых чисел является N, затем задаем порядок этих чисел в произведении (а для каждого нечетного множителя еще и направление обхода). Таким образом будет задан процесс обхода.

Если K, равное числу двоек в произведении, больше одного: K>1, то сторона самого мелкого квадрата должна быть 2^{K-1} или 2^K элементов. Например, если K=3, то обход без разрывов 2x3x2x2 (от самого мелкого 2x2 к 6x6, затем к 12x12 и, наконец, к 24x24) невозможен:

		63	64					9	10		
	59			68			5			14	
55					72	1					18
54					37	36					19
	50			41			32			23	
		46	45					28	27		
...	...	...	...	...	...	73	...	...	...	...	...

Каждая ячейка этой таблицы – квадрат 2x2; нижние 5 строк пропущены: перейти к двум нижним квадратам 12x12 без разрыва не сможем.

Других ограничений при обходе "квадратной змейкой" нет.

Каждый квадрат со стороной L>2 можно обходить обычной змейкой, а не "квадратной". Это выгодно в том случае, если наиболее различающиеся элементы сгруппированы в противоположных углах квадрата.

Упражнение. Нарисуйте порядок обхода квадрата 25x25, а затем – квадрата 12x12. Убедитесь, что разрывов нет и каждый элемент принадлежит к компактно записанной области.

Обход по спирали

Обход по спирали довольно тривиален. Строится квадрат 3x3, затем 5x5, затем 7x7, 9x9 и т. д.:

43	44	45	46	47	48	49
42	21	22	23	24	25	26
41	20	7	8	9	10	27
40	19	6	1	2	11	28
39	18	5	4	3	12	29
38	17	16	15	14	13	30
37	36	35	34	33	32	31

Если же строить круги радиуса 2, 3, 4 и т. д., неизбежно будут присутствовать точки разрыва. Спираль может быть и сходящейся. Суть ее можно показать с помощью этой же иллюстрации, только направление движения обратное: от 49 к 1. Кроме того, она может быть с разворотами:

1	2	3	4	5	6	7
24	25	40	39	38	37	8
23	26	41	42	43	36	9
22	27	48	49	44	35	10
21	28	47	46	45	34	11
20	29	30	31	32	33	12
19	18	17	16	15	14	13

Направление изменяется в точках, расположенных на диагонали: в данном примере это "25" и "41".

Общие моменты для прямоугольных методов

В любом случае можно начать с одного из четырех углов и дальше двигаться в одном из двух направлений: по вертикали и по горизонтали. Первое, т. е. положение первого угла, влияния на степень сжатия почти не оказывает, особенно при сжатии изображений. А вот второе, выбор направления, может существенно улучшить сжатие, поскольку области в этих двух случаях (основное направление по вертикали или по горизонтали) будут сгруппированы по-разному.

Например, отсканированный текст лучше сжимать, обходя по вертикали, поскольку в нем больше длинных сплошных вертикальных линий, чем горизонтальных.

Общие моменты для методов сложной формы

Может возникнуть необходимость помечать уже обработанные точки плоскости, чтобы избежать лишних вычислений, предотвращающих повторное их занесение в D. Тогда есть два основных варианта: либо добавить по одному "флаговому" биту для каждой точки плоскости, либо выбрать (или добавить) значение для "флага", показывающего, что точка уже внесена в D, и записывать это значение на место уже внесенных точек.

Вопросы для самоконтроля

1. Какие параметры надо определить, прежде чем сравнивать два алгоритма компрессии?
2. Почему некорректно сравнивать временные параметры реализаций алгоритмов компрессии, оптимально реализованных на разных компьютерах? Приведите примеры ситуаций, когда архитектура компьютера дает преимущества тому или иному алгоритму.
3. Предложите пример своего класса изображений.
4. Какими свойствами изображений мы можем пользоваться, создавая алгоритм компрессии? Приведите примеры.
5. Что такое редактируемость?
6. Назовите основные требования приложений к алгоритмам компрессии.
7. Что такое симметричность?
8. Предложите пример своего класса приложений.
9. Приведите примеры аппаратных реализаций алгоритма сжатия изображений, с которыми вам приходилось сталкиваться (повседневные и достаточно новые).

10. Почему высокая скорость компрессии, высокое качество изображений и высокая степень компрессии взаимно противоречивы? Покажите противоречивость каждой пары условий.

Глава 1. Сжатие изображения без потерь

Алгоритм RLE

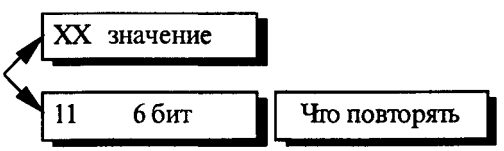
Первый вариант алгоритма

Данный алгоритм необычайно прост в реализации. Кодирование длин повторов – от английского Run Length Encoding (RLE) – один из самых старых и самых простых алгоритмов архивации графики. Изображение в нем (как и в нескольких алгоритмах, описанных ниже) вытягивается в цепочку байтов по строкам растра. Само сжатие в RLE происходит **за счет того, что в исходном изображении встречаются цепочки одинаковых байтов.** Замена их на пары <счетчик повторений, значение> уменьшает избыточность данных.

Алгоритм *декомпрессии* при этом выглядит так:

```
Initialization(...);
do {
   byte = ImageFile.ReadNextByte();
   if(является счетчиком(byte)) {
      counter = Low6bits(byte)+1;
      value = ImageFile.ReadNextByte();
      for(i=1 to counter)
         DecompressedFile.WriteByte(value)
   }
   else {
   DecompressedFile.WriteByte(byte)
} while(!ImageFile.EOF());
```

В данном алгоритме признаком счетчика (counter) служат единицы в двух верхних битах считанного файла:



Соответственно оставшиеся 6 бит расходуются на счетчик, который может принимать значения от 1 до 64. Строку из 64 повторяющихся байт мы превращаем в 2 байта, т. е. сожмем в 32 раза.

Упражнение. Составьте алгоритм *компрессии* для первого варианта алгоритма RLE.

Алгоритм рассчитан на деловую графику – изображения с большими областями повторяющегося цвета. Ситуация, когда файл увеличивается, для этого простого алгоритма не так уж редка. Ее можно легко получить, применяя групповое кодирование к обработанным цветным фотографиям. Для того чтобы увеличить изображение в 2 раза, его надо применить к изображению, в котором значения всех пикселов больше двоичного 11000000 и подряд попарно не повторяются.

Упражнение. Предложите 2–3 примера "плохих" изображений для алгоритма RLE. Объясните, почему размер сжатого файла больше размера исходного файла.

Данный алгоритм реализован в формате PCX. См. пример в приложении 2.

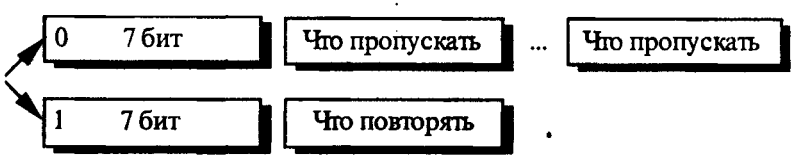
Второй вариант алгоритма

Второй вариант этого алгоритма имеет большую максимальную степень сжатия и меньше увеличивает в размерах исходный файл.

Алгоритм декомпрессии для него выглядит так:

```
Initialization(...);
do {
   byte = ImageFile.ReadNextByte();
   counter = Low7bits(byte)+1;
   if(если признак повтора(byte)) {
      value = ImageFile.ReadNextByte();
      for (i=1 to counter)
         CompressedFile.WriteByte(value)
   }
   else {
      for(i=1 to counter){
         value = ImageFile.ReadNextByte();
         CompressedFile.WriteByte(value)
         }
} while(!ImageFile.EOF());
```

Признаком повтора в данном алгоритме является единица в старшем разряде соответствующего байта:



Как можно легко подсчитать, в лучшем случае этот алгоритм сжимает файл в 64 раза (а не в 32 раза, как в предыдущем варианте), в худшем увеличивает на 1/128. Средние показатели степени компрессии данного алгоритма находятся на уровне показателей первого варианта.

Упражнение. Составьте алгоритм компрессии для второго варианта алгоритма RLE.

Похожие схемы компрессии использованы в качестве одного из алгоритмов, поддерживаемых форматом TIFF, а также в формате TGA.

Характеристики алгоритма RLE:

Степени сжатия: первый вариант: 32, 2, 0,5. Второй вариант: 64, 3, 128/129. (Лучшая, средняя, худшая степени).

Класс изображений: ориентирован алгоритм на изображения с небольшим количеством цветов: деловую и научную графику.

Симметричность: примерно единица.

Характерные особенности: к положительным сторонам алгоритма, пожалуй, можно отнести только то, что он не требует дополнительной памяти при архивации и разархивации, а также быстро работает. Интересная особенность группового кодирования состоит в том, что степень архивации для некоторых изображений может быть существенно повышена всего лишь за счет изменения порядка цветов в палитре изображения.

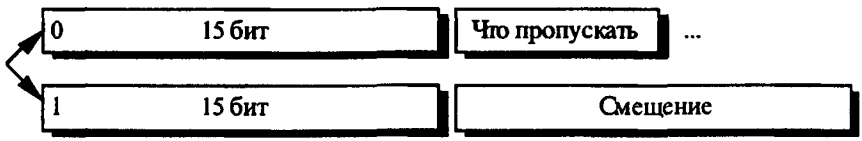
Алгоритм LZW

Название алгоритм получил по первым буквам фамилий его разработчиков – Lempel, Ziv и Welch. Сжатие в нем, в отличие от RLE, осуществляется уже за счет одинаковых цепочек байтов. Алгоритм LZW является самым известным представителем семейства словарных методов LZ78 (см. гл. 3 подразд. 1).

Алгоритм LZ

Существует довольно большое семейство LZ-подобных алгоритмов, различающихся, например, методом поиска повторяющихся цепочек. Один из достаточно простых вариантов этого алгоритма, например, предполагает, что в выходном потоке идет либо пара <длина совпадения, смещение относительно текущей позиции>, либо просто <длина совпадения> "пропускаемых" байтов и сами значения байтов (как во втором варианте алгоритма RLE). При разжатии для пары <длина совпадения, смещение> копируются <длина совпадения> байт из выходного массива, полученного в результате разжатия, на <смещение> байт раньше, а <длина совпадения> (т. е. число,

равное длина совпадениюу) значений "пропускаемых" байтов просто копируются в выходной массив из входного потока. Данный алгоритм является несимметричным по времени, поскольку требует полного перебора буфера (скользящего окна) при поиске одинаковых подстрок. В результате нам сложно задать большой буфер из-за резкого возрастания времени компрессии. Однако потенциально построение алгоритма, в котором на <длина совпадения> и на <смещение> будет выделено по 2 байта (старший бит старшего байта длины совпадения – признак повтора строки / копирования потока), даст нам возможность сжимать все повторяющиеся подстроки размером до 32Кб в буфере размером 64Кб.



При этом мы получим увеличение размера файла в худшем случае на 32770/32768 (в 2 байтах записано, что нужно переписать в выходной поток следующие 2^{15} байт), что совсем неплохо. Максимальная степень сжатия составит в пределе 8192 раза. В пределе, поскольку максимальное сжатие мы получаем, превращая 32 Кб буфера в 4 байта, а буфер ("словарь" в терминах LZ) такого размера мы накопим не сразу. Однако минимальная подстрока, для которой нам выгодно проводить сжатие, должна состоять в общем случае минимум из 5 байт, что и определяет малую ценность данного алгоритма. К достоинствам этого варианта LZ можно отнести чрезвычайную простоту алгоритма декомпрессии.

Упражнение. Предложите другой вариант алгоритма LZ, в котором на пару <счетчик, смещение> будет выделено 3 байта, и подсчитайте основные характеристики своего алгоритма.

Алгоритм LZW

Рассматриваемый нами ниже вариант алгоритма будет использовать дерево для представления и хранения цепочек (фраз словаря в терминах разд. 1). Очевидно, что это достаточно сильное ограничение на вид цепочек и далеко не все одинаковые подцепочки в нашем изображении будут использованы при сжатии. Однако в предлагаемом алгоритме выгодно сжимать даже цепочки, состоящие из 2 байт.

Процесс сжатия выглядит достаточно просто. Мы считываем последовательно символы входного потока и проверяем, есть ли в созданной нами таблице строк такая строка. Если строка есть, то мы считываем следующий

символ, а если строки нет, то мы заносим в поток код для предыдущей найденной строки, заносим строку в таблицу и начинаем поиск снова.

Функция InitTable() очищает таблицу и помещает в нее все строки единичной длины.

```
InitTable();
CompressedFile.WriteCode(ClearCode);
CurStr=пустая строка;

while(не ImageFile.EOF()){     //Пока не конец файла
   C=ImageFile.ReadNextByte();
   if(CurStr+C есть в таблице)
           CurStr=CurStr+C;//Приклеить символ к строке
       else {
           code=CodeForString(CurStr);//code-не байт!
           CompressedFile.WriteCode(code);
           AddStringToTable (CurStr+C);
           CurStr=C;      // Строка из одного символа
   }
}
code=CodeForString(CurStr);
CompressedFile.WriteCode(code);
CompressedFile.WriteCode(CodeEndOfInformation);
```

Как говорилось выше, функция InitTable() инициализирует таблицу строк так, чтобы она содержала все возможные строки, состоящие из одного символа. Например, если мы сжимаем байтовые данные, то таких строк в таблице будет 256 ("0", "1", ... , "255"). Для кода очистки (ClearCode) и кода конца информации (CodeEndOfInformation) зарезервированы значения 256 и 257. В рассматриваемом варианте алгоритма используется 12-битовый код, и, соответственно, под коды для строк нам остаются значения от 258 до 4095. Добавляемые строки записываются в таблицу последовательно, при этом индекс строки в таблице становится ее кодом.

Функция ReadNextByte() читает символ из файла. Функция WriteCode() записывает код (не равный по размеру байту) в выходной файл. Функция AddStringToTable () добавляет новую строку в таблицу, приписывая ей код. Кроме того, в данной функции происходит обработка ситуации переполнения таблицы. В этом случае в поток записывается код предыдущей найденной строки и код очистки, после чего таблица очищается функцией InitTable(). Функция CodeForString() находит строку в таблице и выдает код этой строки.

Пример

Пусть мы сжимаем последовательность 45, 55, 55, 151, 55, 55, 55. Тогда, согласно изложенному выше алгоритму, мы поместим в выходной поток сначала код очистки <256>, потом добавим к изначально пустой строке "45" и проверим, есть ли строка "45" в таблице. Поскольку мы при инициализации занесли в таблицу все строки длиной в один символ, то строка "45" есть в таблице. Далее мы читаем следующий символ 55 из входного потока и проверяем, есть ли строка "45, 55" в таблице. Такой строки в таблице пока нет. Мы заносим в таблицу строку "45, 55" (с первым свободным кодом 258) и записываем в поток код <45>. Можно коротко представить архивацию так:

"45" – есть в таблице;

"45, 55" – нет. Добавляем в таблицу <258>"45, 55". В поток: <45>;

"55, 55" – нет. В таблицу: <259>"55, 55". В поток: <55>;

"55, 151" – нет. В таблицу: <260>"55, 151". В поток: <55>;

"151, 55" – нет. В таблицу: <261>"151, 55". В поток: <151>;

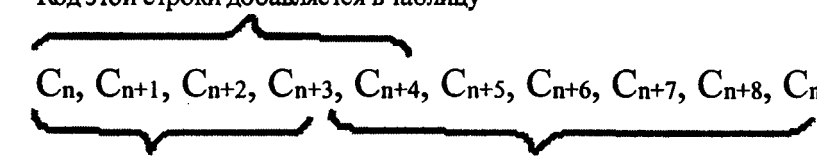
"55, 55" – есть в таблице;

"55, 55, 55" – нет. В таблицу: "55, 55, 55" <262>. В поток: <259>.

Последовательность кодов для данного примера, попадающих в выходной поток: <256>, <45>, <55>, <55>, <151>, <259>.

Особенность LZW заключается в том, что для декомпрессии нам не надо сохранять таблицу строк в файл для распаковки. Алгоритм построен таким образом, что мы в состоянии восстановить таблицу строк, пользуясь только потоком кодов.

Мы знаем, что для каждого кода надо добавлять в таблицу строку, состоящую из уже присутствующей там строки и символа, с которого начинается следующая строка в потоке.



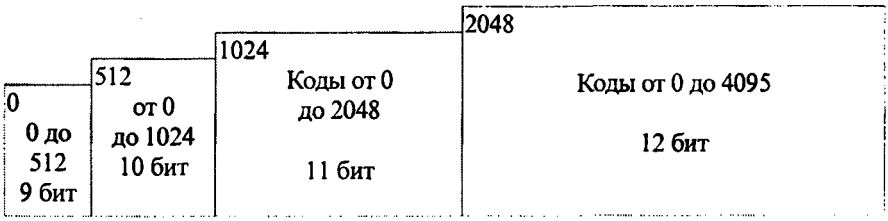
Алгоритм декомпрессии, осуществляющий эту операцию, выглядит следующим образом:

```
code=File.ReadCode();
while(code != CodeEndOfInformation){
      if(code == ClearCode) {
           InitTable();
           code=File.ReadCode();
      }
      if(code == CodeEndOfInformation) {
           закончить работу;
       }
       else {
           if(InTable(code)) {
               ImageFile.WriteString(FromTable(code));
               AddStringToTable(StrFromTable(old_code)+
                   FirstChar(StrFromTable(code)));
               old_code=code;
           }
           else {
               OutString= StrFromTable(old_code)+
                   FirstChar(StrFromTable(old_code));
               ImageFile.WriteString(OutString);
               AddStringToTable(OutString);
               old_code=code;
           }
       }
       code=File.ReadCode();
}
```

Здесь функция ReadCode() читает очередной код из декомпрессируемого файла. Функция InitTable() выполняет те же действия, что и при компрессии, т. е. очищает таблицу и заносит в нее все строки из одного символа. Функция FirstChar() выдает нам первый символ строки. Функция StrFrom Table() выдает строку из таблицы по коду. Функция AddStringToTable() добавляет новую строку в таблицу (присваивая ей первый свободный код). Функция WriteString() записывает строку в файл.

Замечание 1. Как вы могли заметить, записываемые в поток коды постепенно возрастают. До тех пор пока в таблице не появится, например, в первый раз код 512, все коды будут меньше 512. Кроме того, при компрессии и при декомпрессии коды в таблице добавляются при обработке одного и того же символа, т. е. это происходит "синхронно". Мы можем воспользоваться этим свойством алгоритма для того, чтобы повысить степень компрессии. Пока в таблицу не добавлен 512-й символ, мы будем писать в выходной битовый поток коды из 9 бит, а сразу при добавлении 512 – коды из 10 бит. Соответственно декомпрессор также должен будет воспринимать все коды входного потока 9-битовыми до момента добавления в таблицу кода 512, после чего будет вос-

принимать все входные коды как 10-битовые. Аналогично мы будем поступать при добавлении в таблицу кодов 1024 и 2048. Данный прием позволяет примерно на 15% поднять степень компрессии:



Замечание 2. *При сжатии изображения нам важно обеспечить быстроту поиска строк в таблице. Мы можем воспользоваться тем, что каждая следующая подстрока на один символ длиннее предыдущей, кроме того, предыдущая строка уже была нами найдена в таблице. Следовательно, достаточно создать список ссылок на строки, начинающиеся с данной подстроки, как весь процесс поиска в таблице сведется к поиску в строках, содержащихся в списке для предыдущей строки. Понятно, что такая операция может быть проведена очень быстро.*

Заметим также, что реально нам достаточно хранить в таблице только пару <код предыдущей подстроки, добавленный символ>. Этой информации вполне достаточно для работы алгоритма. Таким образом, массив от 0 до 4095 с элементами <код предыдущей подстроки; добавленный символ; список ссылок на строки, начинающиеся с этой строки> решает поставленную задачу поиска, хотя и очень медленно.

На практике для хранения таблицы используется такое же быстрое, как в случае списков, но более компактное по памяти решение – хеш-таблица. Таблица состоит из 8192 (2^{13}) элементов. Каждый элемент содержит <код предыдущей подстроки; добавленный символ; код этой строки>. Ключ для поиска длиной 20 бит формируется с использованием двух первых элементов, хранимых в таблице как одно число (key). Младшие 12 бит этого числа отданы под код, а следующие 8 бит под значение символа.

В качестве хеш-функции при этом используется

```
Index(key)= ((key >> 12) ^ key) & 8191;
```

где >> – побитовый сдвиг вправо (key >> 12 – мы получаем значение символа); ^ – логическая операция побитового исключающего ИЛИ; & логическое побитовое И.

Таким образом, за считанное количество сравнений мы получаем искомый код или сообщение, что такого кода в таблице нет.

Подсчитаем лучшую и худшую степень сжатия для данного алгоритма. Лучшее сжатие, очевидно, будет получено для цепочки одинаковых байт большой длины (т. е. для 8-битового изображения, все точки которого имеют, для определенности, цвет 0). При этом в 258 строку таблицы мы запи-

шем строку "0, 0", в 259 – "0, 0, 0", ... в 4095 – строку из 3839 (=4095-256) нулей. При этом в поток попадет (проверьте по алгоритму!) 3840 кодов, включая код очистки. Следовательно, посчитав сумму арифметической прогрессии от 2 до 3839 (т. е. длину сжатой цепочки) и поделив ее на 3840*12/8 (в поток записываются 12-битовые коды), мы получим лучшую степень сжатия.

Упражнение. Вычислить точное значение лучшей степени сжатия. Более сложное задание: вычислить ее с учетом замечания 1.

Худшее сжатие будет получено, если мы ни разу не встретим подстроку, которая уже есть в таблице (в ней не должно встретиться ни одной одинаковой пары символов).

Упражнение. Составить алгоритм генерации таких цепочек.

В случае, если мы постоянно будем встречать новую подстроку, мы запишем в выходной поток 3840 кодов, которым будет соответствовать строка из 3838 символов. Без учета замечания 1 это составит увеличение файла почти в 1.5 раза.

LZW реализован в форматах GIF и TIFF.

Характеристики алгоритма LZW:

Степени сжатия: примерно 1000, 4, 5/7 (лучшее, среднее, худшее сжатие). Сжатие в 1000 раз достигается только на одноцветных изображениях размером кратным примерно 7 Мб. (Почему 7 Мб – становится понятно при расчете наилучшей степени сжатия.)

Класс изображений: ориентирован LZW на 8-битовые изображения, построенные на компьютере. Сжимает за счет одинаковых подцепочек в потоке.

Симметричность: почти симметричен, при условии оптимальной реализации операции поиска строки в таблице.

Характерные особенности: ситуация, когда алгоритм увеличивает изображение, встречается крайне редко. Универсален.

Алгоритм Хаффмана

Алгоритм Хаффмана с фиксированной таблицей CCITT Group 3

Классический алгоритм Хаффмана был рассмотрен в разд. 1 данной книги. Он практически не применяется к изображениям в чистом виде, а используется как один из этапов компрессии в более сложных схемах.

Близкая модификация алгоритма используется при сжатии черно-белых изображений (1 бит на пиксел). Полное название данного алгоритма CCITT Group 3. Это означает, что данный алгоритм был предложен третьей группой по стандартизации Международного консультационного комитета по телеграфии и телефонии (Consultative Committee International Telegraph and Telephone). Последовательности подряд идущих черных и белых точек в нем заменяются числом, равным их количеству. А этот ряд уже, в свою очередь, сжимается по Хаффману с фиксированной таблицей.

Определение. Набор идущих подряд точек изображения одного цвета называется *серией*. Длина этого набора точек называется *длиной серии*.

В таблицах, приведенных ниже (табл. 1.1 и 1.2), заданы два вида кодов:

- *коды завершения серий* – заданы с 0 до 63 с шагом 1;
- *составные (дополнительные) коды* – заданы с 64 до 2560 с шагом 64.

Каждая строка изображения сжимается независимо. Мы считаем, что в нашем изображении существенно преобладает белый цвет, и все строки изображения начинаются с белой точки. Если строка начинается с черной точки, то мы считаем, что строка начинается белой серией с длиной 0. Например, последовательность длин серий 0, 3, 556, 10, ... означает, что в этой строке изображения идут сначала 3 черные точки, затем 556 белых, затем 10 черных и т. д.

На практике в тех случаях, когда в изображении преобладает черный цвет, мы инвертируем изображение перед компрессией и записываем информацию об этом в заголовок файла.

Алгоритм компрессии выглядит так:

```
for(по всем строкам изображения) {
      Преобразуем строку в набор длин серий;
      for(по всем сериям) {
       if(серия белая) {
          L= длина серии;
          while(L > 2623) { // 2623=2560+63
             L=L-2560;
             ЗаписатьБелыйКодДля(2560);
          }
          if(L > 63) {
          L2=МаксимальныйСостКодМеньшеL(L);
             L=L-L2;
             ЗаписатьБелыйКодДля(L2);
          }
          ЗаписатьБелыйКодДля(L);
                    //Это всегда код завершения
      }
```

```
        else {
        [Код, аналогичный белой серии,
              с той разницей, что записываются
              черные коды]
    }
}
// Окончание строки изображения
}
```

Поскольку черные и белые серии чередуются, то реально код для белой и код для черной серии будут работать попеременно.

В терминах регулярных выражений мы получим для каждой строки нашего изображения (достаточно длинной, начинающейся с белой точки) выходной битовый поток вида:

((<Б-2560>)*[<Б-сст.>]<Б-зв.>(<Ч-2560>)*[<Ч-сст.>]<Ч-зв.>)$^{+}$
[(<Б-2560>)*[<Б-сст.>]<Б-зв.>]

где ()* – повтор 0 или более раз; ()$^{+}$.– повтор 1 или более раз; [] – включение 1 или 0 раз.

Для приведенного ранее примера: 0, 3, 556, 10... алгоритм сформирует следующий код: <Б-0><Ч-3><Б-512><Б-44><Ч-10>, или, согласно таблице, 00110101100110010100101101**0000100** (разные коды в потоке выделены для удобства). Этот код обладает свойством префиксных кодов и легко может быть свернут обратно в последовательность длин серий. Легко подсчитать, что для приведенной строки в 569 бит мы получили код длиной в 33 бита, т. е. степень сжатия составляет примерно 17 раз.

Упражнение. Во сколько раз увеличится размер файла в худшем случае? Почему? (Приведенный в характеристиках алгоритма ответ не является полным, поскольку возможны большие значения худшей степени сжатия. Найдите их.)

Заметим, что единственное "сложное" выражение в нашем алгоритме: L2=МаксимальныйДопКодМеньшеL(L) – на практике работает очень просто: L2=(L>>6)*64, где >> – побитовый сдвиг L влево на 6 бит (можно сделать то же самое за одну побитовую операцию & – логическое И).

Упражнение. Дана строка изображения, записанная в виде длин серий – 442, 2, 56, 3, 23, 3, 104, 1, 94, 1, 231, размером 120 байт ((442+2+..+231)/8). Подсчитать степень сжатия этой строки алгоритмом CCITT Group 3.

Табл. 1.1 и 1.2 построены с помощью классического алгоритма Хаффмана (отдельно для длин черных и белых серий). Значения вероятностей появления для конкретных длин серий были получены путем анализа большого количества факсимильных изображений.

Таблица 1.1. Коды завершения

Длина серии	Код белой подстроки	Код черной подстроки	Длина серии	Код белой подстроки	Код черной подстроки
0	00110101	0000110111	32	00011011	000001101010
1	00111	010	33	00010010	000001101011
2	0111	11	34	00010011	000011010010
3	1000	10	35	00010100	000011010011
4	1011	011	36	00010101	000011010100
5	1100	0011	37	00010110	000011010101
6	1110	0010	38	00010111	000011010110
7	1111	00011	39	00101000	000011010111
8	10011	000101	40	00101001	000001101100
9	10100	000100	41	00101010	000001101101
10	00111	0000100	42	00101011	000011011010
11	01000	0000101	43	00101100	000011011011
12	001000	0000111	44	00101101	000001010100
13	000011	00000100	45	00000100	000001010101
14	110100	00000111	46	00000101	000001010110
15	110101	000011000	47	00001010	000001010111
16	101010	0000010111	48	00001011	000001100100
17	101011	0000011000	49	01010010	000001100101
18	0100111	0000001000	50	01010011	000001010010
19	0001100	00001100111	51	01010100	000001010011
20	0001000	00001101000	52	01010101	000000100100
21	0010111	00001101100	53	00100100	000000110111
22	0000011	00000110111	54	00100101	000000111000
23	0000100	00000101000	55	01011000	000000100111
24	0101000	00000010111	56	01011001	000000101000
25	0101011	00000011000	57	01011010	000001011000
26	0010011	000011001010	58	01011011	000001011001
27	0100100	000011001011	59	01001010	000000101011
28	0011000	000011001100	60	01001011	000000101100
29	00000010	000011001101	61	00110010	000001011010
30	00000011	000001101000	62	00110011	000001100110
31	00011010	000001101001	63	00110100	000001100111

Таблица 1.2. Составные коды

Длина серии	Код белой подстроки	Код черной подстроки	Длина серии	Код белой подстроки	Код черной подстроки
64	11011	0000001111	1344	011011010	0000001010011
128	10010	000011001000	1408	011011011	0000001010100
192	01011	000011001001	1472	010011000	0000001010101
256	0110111	000001011011	1536	010011001	0000001011010
320	00110110	000000110011	1600	010011010	0000001011011
384	00110111	000000110100	1664	011000	0000001100100
448	01100100	000000110101	1728	010011011	0000001100101
512	01100101	0000001101100	1792	00000001000	совп. с белой
576	01101000	0000001101101	1856	00000001100	–//–
640	01100111	0000001001010	1920	00000001101	–//–
704	011001100	0000001001011	1984	000000010010	–//–
768	011001101	0000001001100	2048	000000010011	–//–
832	011010010	0000001001101	2112	000000010100	–//–
896	011010011	0000001110010	2176	000000010101	–//–
960	011010100	0000001110011	2240	000000010110	–//–
1024	011010101	0000001110100	2304	000000010111	–//–
1088	011010110	0000001110101	2368	000000011100	–//–
1152	011010111	0000001110110	2432	000000011101	–//–
1216	011011000	0000001110111	2496	000000011110	–//–
1280	011011001	0000001010010	2560	000000011111	–//–

Если в одном столбце встретятся два числа с одинаковым префиксом, то это опечатка.

Этот алгоритм реализован в формате TIFF.

Характеристики алгоритма CCITT Group 3

Степени сжатия: лучшая стремится в пределе к 213.(3), средняя 2, в худшем случае увеличивает файл в 5 раз.

Класс изображений: двуцветные черно-белые изображения, в которых преобладают большие пространства, заполненные белым цветом (рис. 1.1 и 1.2).

Симметричность: близка к единице.

Характерные особенности: данный алгоритм чрезвычайно прост в реализации, быстр и может быть легко реализован аппаратно.

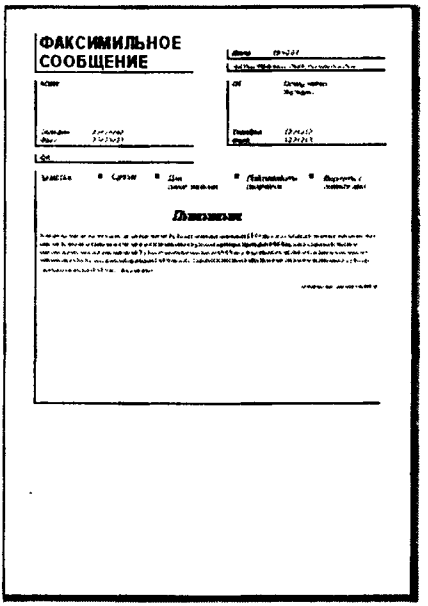


Рис. 1.1. Изображение, для которого очень выгодно применение алгоритма CCITT-3. (Большие области заполнены одним цветом)

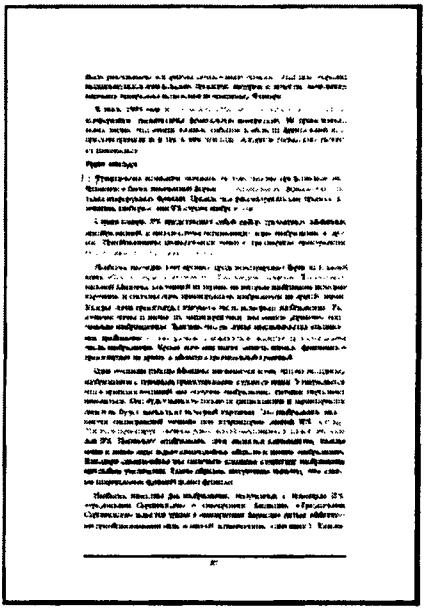


Рис. 1.2. Изображение, для которого менее выгодно применение алгоритма CCITT-3. (Меньше областей, заполненных одним цветом. Много коротких "черных" и "белых" серий)

JBIG

Алгоритм разработан группой экспертов ISO (Joint Bi-level Experts Group) специально для сжатия 1-битовых черно-белых изображений [5]. Например, факсов или отсканированных документов. В принципе может применяться и к 2-, и к 4-битовым картинкам. При этом алгоритм разбивает их на отдельные битовые плоскости. JBIG позволяет управлять такими параметрами, как порядок разбиения изображения на битовые плоскости, ширина полос в изображении, уровни масштабирования. Последняя возможность позволяет легко ориентироваться в базе больших по размерам изображений, просматривая сначала их уменьшенные копии. Настраивая эти параметры, можно использовать описанный выше эффект "огрубленного изображения" при получении изображения по сети или по любому другому каналу, пропускная способность которого мала по сравнению с возможностями процессора. Распаковываться изображение на экране будет постепен-

но, как бы медленно "проявляясь". При этом человек начинает анализировать картинку задолго до конца процесса разархивации.

Алгоритм построен на базе Q-кодировщика [6], патентом на который владеет IBM. Q-кодер, так же как и алгоритм Хаффмана, использует для чаще появляющихся символов короткие цепочки, а для реже появляющихся – длинные. Однако, в отличие от него, в алгоритме используются и последовательности символов.

Lossless JPEG

Этот алгоритм разработан группой экспертов в области фотографии (Joint Photographic Expert Group). В отличие от JBIG, Lossless JPEG ориентирован на полноцветные 24- или 8-битовые картинки в градациях серого изображения без палитры. Он представляет собой специальную реализацию JPEG без потерь. Степени сжатия: 20, 2, 1. Lossless JPEG рекомендуется применять в тех приложениях, где необходимо побитовое соответствие исходного и декомпрессированного изображений. Подробнее об алгоритме сжатия JPEG см. разд. 3.

Заключение

Попробуем на этом этапе сделать некоторые обобщения. С одной стороны, приведенные выше алгоритмы достаточно универсальны и покрывают все типы изображений, с другой – у них, по сегодняшним меркам, слишком маленькая степень сжатия. Используя один из алгоритмов сжатия без потерь, можно обеспечить архивацию изображения примерно в 2 раза. В то же время алгоритмы сжатия с потерями оперируют с коэффициентами 10–200 раз. Помимо возможности модификации изображения, одна из основных причин подобной разницы заключается в том, что традиционные алгоритмы ориентированы на работу с цепочкой. Они не учитывают так называемую *когерентность областей* в изображениях. Идея когерентности областей заключается в малом изменении цвета и структуры на небольшом участке изображения. Все алгоритмы, о которых речь пойдет ниже, были созданы позднее специально для сжатия графики и используют эту идею.

Справедливости ради следует отметить, что и в классических алгоритмах можно использовать идею когерентности. Существуют алгоритмы обхода изображения по *фрактальной* кривой, при работе которых оно также вытягивается в цепочку; но за счет того, что кривая обегает области изображения по сложной траектории, участки близких цветов в получающейся цепочке удлиняются.

Вопросы для самоконтроля

1. На какой класс изображений ориентирован алгоритм RLE?
2. Приведите два примера "плохих" изображений для первого варианта алгоритма RLE, для которых файл максимально увеличится в размере.
3. На какой класс изображений ориентирован алгоритм CCITT G-3?
4. Приведите пример "плохого" изображения для алгоритма CCITT G-3, для которого файл максимально увеличится в размере. (Приведенный в характеристиках алгоритма ответ не является полным, поскольку требует более "умной" реализации алгоритма.)
5. Приведите пример "плохого" изображения для алгоритма Хаффмана.
6. Сравните алгоритмы сжатия изображений без потерь.
7. В чем заключается идея когерентности областей?

Глава 2. Сжатие изображений с потерями

Проблемы алгоритмов сжатия с потерями

Первыми для сжатия изображений стали применяться привычные алгоритмы. Те, что использовались и используются в системах резервного копирования, при создании дистрибутивов и т. п. Эти алгоритмы архивировали информацию без изменений. Однако основной тенденцией в последнее время стало использование новых классов изображений. Старые алгоритмы перестали удовлетворять требованиям, предъявляемым к сжатию. Многие изображения практически не сжимались, хотя "на взгляд" обладали явной избыточностью. Это привело к созданию нового типа алгоритмов – сжимающих с потерей информации. Как правило, степень сжатия и, следовательно, степень потерь качества в них можно задавать. При этом достигается компромисс между размером и качеством изображений.

Одна из серьезных проблем машинной графики заключается в том, что до сих пор не найден адекватный критерий оценки потерь качества изображения. А теряется оно постоянно – при оцифровке, при переводе в ограниченную палитру цветов, при переводе в другую систему цветопредставления для печати и, что для нас особенно важно, при сжатии с потерями. Можно привести пример простого критерия: среднеквадратичное отклонение значений пикселов (L_2 мера, или root mean square – RMS):

$$d(x,y)=\sqrt{\frac{\sum_{i=1,j=1}^{n,n}\left(x_{ij}-y_{ij}\right)^2}{n^2}}.$$

По нему изображение будет сильно испорчено при понижении яркости всего на 5% (глаз этого не заметит – у разных мониторов настройка яркости варьируется гораздо сильнее). В то же время изображения "со снегом" – резким изменением цвета отдельных точек, слабыми полосами или "муаром" – будут признаны "почти не изменившимися" (Объясните почему.). Свои неприятные стороны есть и у других критериев.

Рассмотрим, например, максимальное отклонение:

$$d(x,y) = \max_{i,j} \left| x_{ij} - y_{ij} \right| .$$

Эта мера, как можно догадаться, крайне чувствительна к биению отдельных пикселов. То есть во всем изображении может существенно измениться только значение 1 пиксела (что практически незаметно для глаза), однако согласно этой мере изображение будет сильно испорчено.

Мера, которую сейчас используют на практике, называется *мерой отношения сигнала к шуму* (peak-to-peak signal-to-noise ratio – PSNR):

$$d(x,y) = 10 \cdot \log_{10} \frac{255^2 \cdot n^2}{\sum_{i=1,j=1}^{n,n} \left(x_{ij} - y_{ij} \right)^2} .$$

Данная мера, по сути, аналогична среднеквадратичному отклонению, однако пользоваться ей несколько удобнее за счет логарифмического масштаба шкалы. Ей присущи те же недостатки, что и среднеквадратичному отклонению.

Лучше всего потери качества изображений оценивают наши глаза. Отличным считается сжатие, при котором невозможно на глаз различить первоначальное и распакованное изображения. Хорошим – когда сказать, какое из изображений подвергалось сжатию, можно только сравнивая две находящиеся рядом картинки. При дальнейшем увеличении степени сжатия, как правило, становятся заметны побочные эффекты, характерные для данного алгоритма. На практике, даже при отличном сохранении качества, в изображение могут быть внесены регулярные специфические изменения. Поэтому алгоритмы сжатия с потерями не рекомендуется использовать при сжатии изображений, которые в дальнейшем собираются либо печатать с высоким качеством, либо обрабатывать программами распознавания образов. Неприятные эффекты с такими изображениями, как мы уже говорили, могут возникнуть даже при простом масштабировании изображения.

Алгоритм JPEG

JPEG – один из новых и достаточно мощных алгоритмов. Практически он является стандартом де-факто для полноцветных изображений [1]. Оперирует алгоритм областями 8x8, на которых яркость и цвет меняются сравнительно плавно. Вследствие этого при разложении матрицы такой области в двойной ряд по косинусам (см. формулы ниже) значимыми оказываются только первые коэффициенты. Таким образом, сжатие в JPEG осуществляется за счет плавности изменения цветов в изображении.

Алгоритм разработан группой экспертов в области фотографии специально для сжатия 24-битовых изображений. JPEG – Joint Photographic Expert Group – подразделение в рамках ISO – Международной организации по стандартизации. Название алгоритма читается как ['jei'peg]. В целом алгоритм основан на дискретном косинусоидальном преобразовании (в дальнейшем – ДКП), применяемом к матрице изображения для получения некоторой новой матрицы коэффициентов. Для получения исходного изображения применяется обратное преобразование.

ДКП раскладывает изображение по амплитудам некоторых частот. Таким образом, при преобразовании мы получаем матрицу, в которой многие коэффициенты либо близки, либо равны нулю. Кроме того, благодаря несовершенству человеческого зрения можно аппроксимировать коэффициенты более грубо без заметной потери качества изображения.

Для этого используется квантование коэффициентов (quantization). В самом простом случае – это арифметический побитовый сдвиг вправо. При этом преобразовании теряется часть информации, но может достигаться большая степень сжатия.

Как работает алгоритм

Итак, рассмотрим алгоритм подробнее (рис. 2.1). Пусть мы сжимаем 24-битовое изображение.

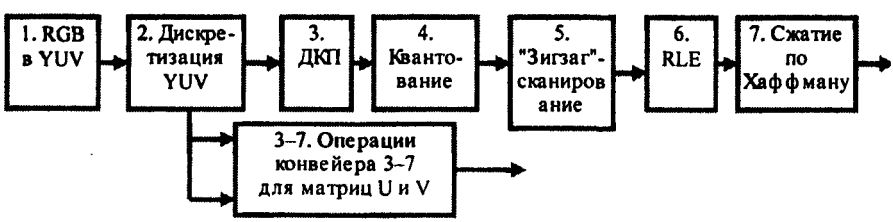


Рис. 2.1. Конвейер операций, используемый в алгоритме JPEG

Шаг 1. Переводим изображение из цветового пространства RGB, с компонентами, отвечающими за красную (Red), зеленую (Green) и синюю

(Blue) составляющие цвета точки, в цветовое пространство YCrCb (иногда называют YUV).

В нем Y – яркостная составляющая, а Cr, Cb – компоненты, отвечающие за цвет (хроматический красный и хроматический синий). За счет того, что человеческий глаз менее чувствителен к цвету, чем к яркости, появляется возможность архивировать массивы для Cr и Cb компонент с большими потерями и, соответственно, большими степенями сжатия. Подобное преобразование уже давно используется в телевидении. На сигналы, отвечающие за цвет, там выделяется более узкая полоса частот.

Упрощенно перевод из цветового пространства RGB в цветовое пространство YCrCb можно представить с помощью матрицы перехода:

$$\begin{vmatrix} Y \\ Cr \\ Cb \end{vmatrix} = \begin{vmatrix} 0.2990 & 0.5870 & 0.1140 \\ 0.5000 & -0.4187 & -0.0813 \\ -0.1687 & -0.3313 & 0.5000 \end{vmatrix} * \begin{vmatrix} R \\ G \\ B \end{vmatrix} + \begin{vmatrix} 0 \\ 128 \\ 128 \end{vmatrix}$$

Обратное преобразование осуществляется умножением вектора YUV на обратную матрицу.

$$\begin{vmatrix} R \\ G \\ B \end{vmatrix} = \begin{vmatrix} 1 & 0 & 1.402 \\ 1 & -0.34414 & -0.71414 \\ 1 & 1.772 & 0 \end{vmatrix} * \left(\begin{vmatrix} Y \\ Cb \\ Cr \end{vmatrix} - \begin{vmatrix} 0 \\ 128 \\ 128 \end{vmatrix} \right)$$

Шаг 2. Разбиваем исходное изображение на матрицы 8x8. Формируем из каждой 3 рабочие матрицы ДКП – по 8 бит отдельно для каждой компоненты. При больших степенях сжатия этот шаг может выполняться чуть сложнее. Изображение делится по компоненте Y, как и в первом случае, а для компонент Cr и Cb матрицы набираются через строчку и через столбец. То есть из исходной матрицы размером 16x16 получается только одна рабочая матрица ДКП. При этом, как нетрудно заметить, мы теряем 3/4 полезной информации о цветовых составляющих изображения и получаем сразу сжатие в 2 раза. Мы можем поступать так благодаря работе в пространстве YCrCb. На результирующем RGB-изображении, как показала практика, это сказывается несильно.

Шаг 3. В упрощенном виде ДКП при n=8 можно представить так:

$$Y[u,v] = \frac{1}{4} \sum_{i=0}^{7} \sum_{j=0}^{7} C(i,u) \times C(j,v) \times y[i,j],$$

где $$C(i,u) = A(u) \times \cos\left(\frac{(2\times i+1)\times u\times \pi}{2\cdot n}\right)$$

$$A(u) = \begin{cases} \frac{1}{\sqrt{2}}, & \text{for } u \equiv 0, \\ 1, & \text{for } u \neq 0. \end{cases}$$

Применяем ДКП к каждой рабочей матрице. При этом мы получаем матрицу, в которой коэффициенты в левом верхнем углу соответствуют низкочастотной составляющей изображения, а в правом нижнем – высокочастотной. Понятие частоты следует из рассмотрения изображения как двумерного сигнала (аналогично рассмотрению звука как сигнала). Плавное изменение цвета соответствует низкочастотной составляющей, а резкие скачки – высокочастотной.

Шаг 4. Производим квантование. В принципе это просто деление рабочей матрицы на матрицу квантования поэлементно. Для каждой компоненты (Y, U и V) в общем случае задается своя матрица квантования q[u,v] (далее – МК).

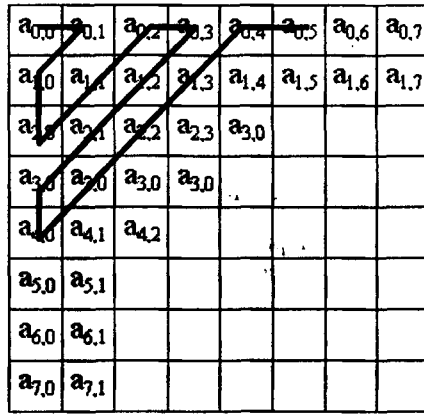
$$Yq[u,v] = \text{IntegerRound}\left(\frac{Y[u,v]}{q[u,v]}\right).$$

На этом шаге осуществляется управление степенью сжатия и происходят самые большие потери. Понятно, что, задавая МК с большими коэффициентами, мы получим больше нулей и, следовательно, большую степень сжатия.

В стандарт JPEG включены рекомендованные МК, построенные опытным путем. Матрицы для большей или меньшей степени сжатия получают путем умножения исходной матрицы на некоторое число gamma.

С квантованием связаны и специфические эффекты алгоритма. При больших значениях коэффициента gamma потери в низких частотах могут быть настолько велики, что изображение распадется на квадраты 8x8. Потери в высоких частотах могут проявиться в так называемом *эффекте Гиббса*, когда вокруг контуров с резким переходом цвета образуется своеобразный "нимб".

Шаг 5. Переводим матрицу 8x8 в 64-элементный вектор при помощи "зигзаг"-сканирования, т. е. берем элементы с индексами (0,0), (0,1), (1,0), (2,0)...



Таким образом, в начале вектора мы получаем коэффициенты матрицы, соответствующие низким частотам, а в конце – высоким.

Шаг 6. Свертываем вектор с помощью алгоритма группового кодирования. При этом получаем пары типа <пропустить, число>, где "пропустить" является счетчиком пропускаемых нулей, а "число" – значение, которое необходимо поставить в следующую ячейку. Так, вектор 42 3 0 0 0 -2 0 0 0 0 1 ... будет свернут в пары (0,42) (0,3) (3,-2) (4,1) ... .

Шаг 7. Свертываем получившиеся пары кодированием по Хаффману с фиксированной таблицей.

Процесс восстановления изображения в этом алгоритме полностью симметричен. Метод позволяет сжимать некоторые изображения в 10–15 раз без серьезных потерь.

Существенными положительными сторонами алгоритма является то, что:

- задается степень сжатия;
- выходное цветное изображение может иметь 24 бита на точку.

Отрицательными сторонами алгоритма является то, что:

- При повышении степени сжатия изображение распадается на отдельные квадраты (8x8). Это связано с тем, что происходят большие потери в низких частотах при квантовании и восстановить исходные данные становится невозможно.
- Проявляется эффект Гиббса – ореолы по границам резких переходов цветов.

Как уже говорилось, стандартизован JPEG относительно недавно – в 1991 г. Но уже тогда существовали алгоритмы, сжимающие сильнее при меньших потерях качества. Дело в том, что действия разработчиков стандарта были ограничены мощностью существовавшей на тот момент техни-

ки. То есть даже на ПК алгоритм должен был работать меньше минуты на среднем изображении, а его аппаратная реализация должна быть относительно простой и дешевой. Алгоритм должен был быть симметричным (время разархивации примерно равно времени архивации).

Выполнение последнего требования сделало возможным появление таких устройств, как цифровые фотоаппараты, снимающие 24-битовые фотографии на 8–256 Мб флеш-карту. Потом эта карта вставляется в разъем на вашем ноутбуке и соответствующая программа позволяет считать изображения. Не правда ли, если бы алгоритм был несимметричен, было бы неприятно долго ждать, пока аппарат "перезарядится" – сожмет изображение.

Не очень приятным свойством JPEG является также то, что нередко горизонтальные и вертикальные полосы на дисплее абсолютно не видны и могут проявиться только при печати в виде муарового узора. Он возникает при наложении наклонного растра печати на горизонтальные и вертикальные полосы изображения. Из-за этих сюрпризов JPEG не рекомендуется активно использовать в полиграфии, задавая высокие коэффициенты матрицы квантования. Однако при архивации изображений, предназначенных для просмотра человеком, он на данный момент незаменим.

Широкое применение JPEG долгое время сдерживалось, пожалуй, лишь тем, что он оперирует 24-битовыми изображениями. Поэтому для того, чтобы с приемлемым качеством посмотреть картинку на обычном мониторе в 256-цветной палитре, требовалось применение соответствующих алгоритмов и, следовательно, определенное время. В приложениях, ориентированных на придирчивого пользователя, таких, например, как игры, подобные задержки неприемлемы. Кроме того, если имеющиеся у вас изображения, допустим, в 8-битовом формате GIF перевести в 24-битовый JPEG, а потом обратно в GIF для просмотра, то потеря качества произойдет дважды при обоих преобразованиях. Тем не менее выигрыш в размерах архивов зачастую настолько велик (в 3–20 раз), а потери качества настолько малы, что хранение изображений в JPEG оказывается очень эффективным.

Несколько слов необходимо сказать о модификациях этого алгоритма. Хотя JPEG и является стандартом ISO, формат его файлов не был зафиксирован. Пользуясь этим, производители создают свои, несовместимые между собой форматы и, следовательно, могут изменить алгоритм. Так, внутренние таблицы алгоритма, рекомендованные ISO, заменяются ими на свои собственные. Кроме того, легкая неразбериха присутствует при задании степени потерь. Например, при тестировании выясняется, что "отличное" качество, "100%" и "10 баллов" дают существенно различающиеся картинки. При этом, кстати, "100%" качества не означает сжатия без потерь. Встречаются также варианты JPEG для специфических приложений.

Как стандарт ISO JPEG начинает все шире использоваться при обмене изображениями в компьютерных сетях. Поддерживается алгоритм JPEG в форматах Quick Time, PostScript Level 2, Tiff 6.0 и на данный момент занимает видное место в системах мультимедиа.

Характеристики алгоритма JPEG:

Степень сжатия: 2–200 (задается пользователем).

Класс изображений: полноцветные 24 битовые изображения или изображения в градациях серого без резких переходов цветов (фотографии).

Симметричность: 1.

Характерные особенности: в некоторых случаях алгоритм создает "ореол" вокруг резких горизонтальных и вертикальных границ в изображении (эффект Гиббса). Кроме того, при высокой степени сжатия изображение распадается на блоки 8x8 пикселов.

Фрактальный алгоритм

Идея метода

Фрактальное сжатие основано на том, что мы представляем изображение в более компактной форме – с помощью коэффициентов системы итерируемых функций (Iterated Function System – далее по тексту как IFS). Прежде чем рассматривать сам процесс архивации, разберем, как IFS строит изображение, т. е. процесс декомпрессии.

Строго говоря, IFS представляет собой набор трехмерных аффинных преобразований, в нашем случае переводящих одно изображение в другое. Преобразованию подвергаются точки в трехмерном пространстве (x_координата, y_координата, яркость).

Наиболее наглядно этот процесс продемонстрировал M. F. Barnsley в книге [22]. Там введено понятие "фотокопировальной машины", состоящей из экрана, на котором изображена исходная картинка, и системы линз, проецирующих изображение на другой экран (рис. 2.2):

- линзы могут проецировать часть изображения **произвольной формы** в любое другое место нового изображения;
- области, **в которые** проецируются изображения, **не пересекаются**;
- линза может **менять яркость и уменьшать контрастность**;
- линза может **зеркально отражать и поворачивать** свой фрагмент изображения;
- линза **должна масштабировать** (причем только уменьшая) свой фрагмент изображения.

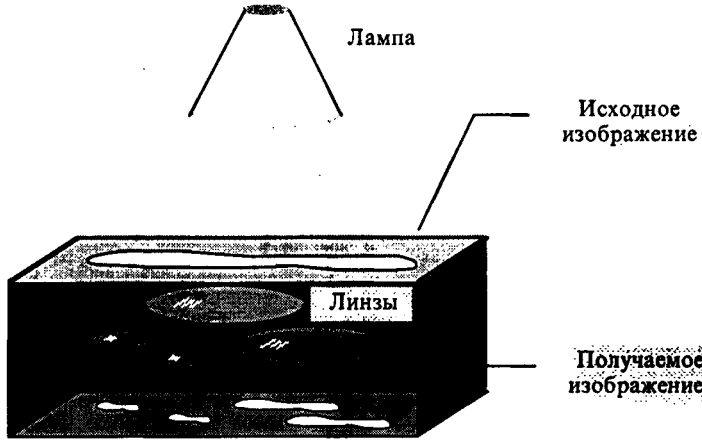


Рис. 2.2. Машина Барнсли

Расставляя линзы и меняя их характеристики, мы можем управлять получаемым изображением. Одна итерация работы машины заключается в том, что по исходному изображению с помощью проектирования строится новое, после чего новое берется в качестве исходного. Утверждается, что в процессе итераций мы получим изображение, которое перестанет изменяться. Оно будет зависеть только от расположения и характеристик линз и не будет зависеть от исходной картинки. Это изображение называется *неподвижной точкой* или *аттрактором* данной IFS. Соответствующая теория гарантирует наличие ровно одной неподвижной точки для каждой IFS.

Поскольку отображение линз является сжимающим, каждая линза в явном виде задает *самоподобные* области в нашем изображении. Благодаря самоподобию мы получаем сложную структуру изображения при любом увеличении. Таким образом, интуитивно понятно, что система итерируемых функций задает *фрактал* (нестрого – самоподобный математический объект).

Наиболее известны два изображения, полученные с помощью IFS: *треугольник Серпинского* (рис. 2.3) и *папоротник Барнсли* (рис. 2.4).

Треугольник Серпинского задается тремя, а папоротник Барнсли – четырьмя *аффинными преобразованиями* (или, в нашей терминологии, "линзами"). Каждое преобразование кодируется буквально считанными байтами, в то время как изображение, построенное с их помощью, может занимать и несколько мегабайт.

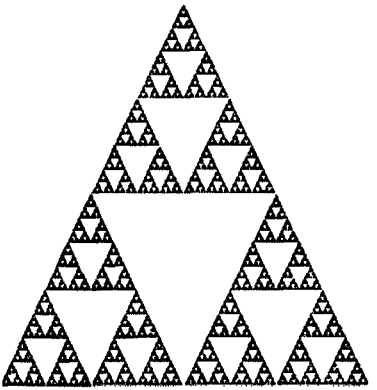


Рис. 2.3. Треугольник Серпинского

Рис. 2.4. Папоротник Барнсли

Упражнение. Укажите в изображении 4 области, объединение которых покрывало бы все изображение и каждая из которых была бы подобна всему изображению (не забывайте о стебле папоротника).

Из вышесказанного становится понятно, как работает архиватор и почему ему требуется так много времени. Фактически фрактальная компрессия – это поиск самоподобных областей в изображении и определение для них параметров аффинных преобразований (рис. 2.5).

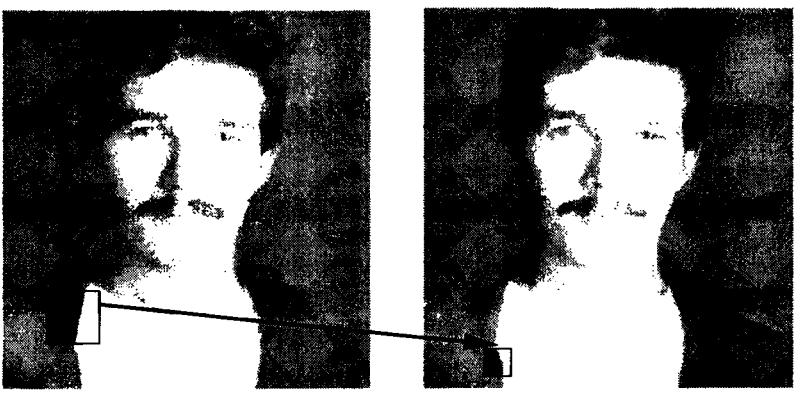


Рис. 2.5. Самоподобные области изображения

В худшем случае, если не будет применяться оптимизирующий алгоритм, потребуется перебор и сравнение всех возможных фрагментов изображения разного размера. Даже для небольших изображений при учете дискретности мы получим астрономическое число перебираемых вариантов. Причем даже резкое сужение классов преобразований, например за счет

масштабирования только в определенное количество раз, не дает заметного выигрыша во времени. Кроме того, при этом теряется качество изображения. Подавляющее большинство исследований в области фрактальной компрессии сейчас направлены на уменьшение времени архивации, необходимого для получения качественного изображения.

Далее приводятся основные определения и теоремы, на которых базируется фрактальная компрессия. Этот материал более детально и с доказательствами рассматривается в [3] и [4].

Определение. Преобразование $w: R^2 \to R^2$, представимое в виде

$$w(\overline{x}) = w\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix},$$

где a, b, c, d, e, f – действительные числа и $\begin{pmatrix} x & y \end{pmatrix} \in R^2$ – называется *двумерным аффинным преобразованием.*

Определение. Преобразование $w: R^3 \to R^3$, представимое в виде

$$w(\overline{x}) = w\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a & b & t \\ c & d & u \\ r & s & p \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} e \\ f \\ q \end{pmatrix}$$

где $a, b, c, d, e, f, p, q, r, s, t, u$ – действительные числа и $\begin{pmatrix} x & y & z \end{pmatrix} \in R^3$, называется *трехмерным аффинным преобразованием.*

Определение. Пусть $f: \mathrm{X} \to \mathrm{X}$ – преобразование в пространстве X. Точка $x_f \in \mathrm{X}$, такая, что $f(x_f) = x_f$, называется *неподвижной точкой* (*аттрактором*) преобразования.

Определение. Преобразование $f: \mathrm{X} \to \mathrm{X}$ в метрическом пространстве (X, d) называется *сжимающим*, если существует число s: $0 \le s < 1$, такое, что

$$d(f(x), f(y)) \le s \cdot d(x, y), \ \forall x, y \in \mathrm{X}.$$

Замечание. *Формально мы можем использовать любое сжимающее отображение при фрактальной компрессии, но реально используются лишь трехмерные аффинные преобразования с достаточно сильными ограничениями на коэффициенты.*

Теорема. (О сжимающем преобразовании.) *Пусть* $f: \mathrm{X} \to \mathrm{X}$ *– сжимающее преобразование в полном метрическом пространстве (X, d). Тогда существует в точности одна неподвижная точка* $x_f \in \mathrm{X}$ *этого преобразо-*

вания и для любой точки $x \in X$ *последовательность* $\{f^n(x): n = 0,1,2...\}$ *сходится к* x_f.

Более общая формулировка этой теоремы гарантирует нам сходимость.

Определение. *Изображением* называется функция S, определенная на единичном квадрате и принимающая значения от 0 до 1 или $S(x,y) \in [0...1]\ \forall x,y \in [0...1]$.

Пусть трехмерное аффинное преобразование $w_i : R^3 \to R^3$, записано в виде

$$w_i(\overline{x}) = w_i\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & p \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} e \\ f \\ q \end{pmatrix}$$

и определено на компактном подмножестве D_i декартова квадрата [0...1]x[0...1] (мы пользуемся особым видом матрицы преобразования, чтобы уменьшить размерность области определения с R^3 до R^2). Тогда оно переведет часть поверхности S в область R_i, расположенную со сдвигом *(e,f)* и поворотом, заданным матрицей

$$\begin{pmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

При этом, если интерпретировать значения функции $S(x,y) \in [0...1]$ как яркость соответствующих точек, она уменьшится в p раз (преобразование обязано быть сжимающим) и изменится на сдвиг q.

Определение. Конечная совокупность W сжимающих трехмерных аффинных преобразований w_i, определенных на областях D_i, таких, что $w_i(D_i) = R_i$ и $R_i \cap R_j = \phi \quad \forall i \neq j$, называется *системой итерируемых функций* (*IFS*).

Системе итерируемых функций однозначно ставятся в соответствие неподвижная точка – изображение. Таким образом, процесс компрессии заключается в поиске коэффициентов системы, а процесс декомпрессии – в проведении итераций системы до стабилизации полученного изображения (неподвижной точки IFS). На практике бывает достаточно 7–16 итераций. Области R_i в дальнейшем будут именоваться *ранговыми*, а области D_i – *доменными* (рис. 2.6).

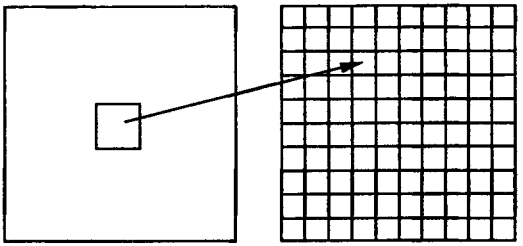


Рис. 2.6. Перевод доменного блока в ранговый

Построение алгоритма

Как уже стало очевидным из изложенного выше, основной задачей при компрессии фрактальным алгоритмом является нахождение соответствующих аффинных преобразований. В самом общем случае мы можем переводить любые по размеру и форме области изображения, однако в этом случае получается астрономическое число перебираемых вариантов разных фрагментов, которое невозможно обработать на текущий момент даже на суперкомпьютере.

В ***учебном варианте алгоритма***, изложенном далее, сделаны следующие ограничения на области:

1. Все области *являются квадратами со сторонами, параллельными сторонам изображения*. Это ограничение достаточно жесткое. Фактически мы собираемся аппроксимировать все многообразие геометрических фигур лишь квадратами.
2. При переводе доменной области в ранговую уменьшение размеров производится *ровно в 2 раза* (см. рис. 2.6). Это существенно упрощает как компрессор, так и декомпрессор, так как задача масштабирования небольших областей является нетривиальной.
3. Все доменные блоки – квадраты и имеют *фиксированный размер*. Изображение равномерной сеткой разбивается на набор доменных блоков.
4. Доменные области берутся *"через точку"* и по X и по Y, что сразу уменьшает перебор в 4 раза.
5. При переводе доменной области в ранговую поворот куба возможен *только на 0, 90, 180 или* 270^0. Также допускается зеркальное отражение. Общее число возможных преобразований (считая пустое) – 8.
6. Масштабирование (сжатие) по вертикали (яркости) осуществляется *в фиксированное число раз* – 0.75.

Эти ограничения позволяют:

1. Построить алгоритм, для которого требуется сравнительно малое число операций даже на достаточно больших изображениях.

2. Очень компактно представить данные для записи в файл. Нам требуется на каждое аффинное преобразование в IFS:
   - Два числа для того, чтобы задать смещение доменного блока. Если мы ограничим входные изображения размером 512x512, то достаточно будет по 8 бит на каждое число.
   - Три бита для того, чтобы задать преобразование симметрии при переводе доменного блока в ранговый.
   - 7–9 бит для того, чтобы задать сдвиг по яркости при переводе.

Информацию о размере блоков можно хранить в заголовке файла. Таким образом, мы затратили менее 4 байт на одно аффинное преобразование. В зависимости от того, каков размер блока, можно высчитать, сколько блоков будет в изображении. Таким образом, мы можем получить оценку степени компрессии.

Например, для файла в градациях серого 256 цветов 512x512 пикселов при размере блока 8 пикселов аффинных преобразований будет 4096 (512/8·512/8). На каждое потребуется 3.5 байта. Следовательно, если исходный файл занимал 262144 (512·512) байт (без учета заголовка), то файл с коэффициентами будет занимать 14336 байт. Степень сжатия – 18 раз. При этом мы не учитываем, что файл с коэффициентами тоже может обладать избыточностью и сжиматься без потерь с помощью, например, LZW.

Отрицательные стороны предложенных ограничений:

1. Поскольку все области являются квадратами, невозможно воспользоваться подобием объектов, по форме далеких от квадратов (которые встречаются в реальных изображениях достаточно часто.)
2. Аналогично мы не сможем воспользоваться подобием объектов в изображении, коэффициент подобия между которыми сильно отличается от двух.
3. Алгоритм не сможет воспользоваться подобием объектов в изображении, угол между которыми не кратен 90^0.

Такова плата за **скорость компрессии** и за простоту упаковки коэффициентов в файл.

Сам алгоритм упаковки сводится к перебору всех доменных блоков и подбору для каждого соответствующего ему рангового блока. Ниже приводится схема этого алгоритма.

```
for (all range blocks) {
   min_distance = MaximumDistance;
   Rij = image->CopyBlock(i,j);
   for (all domain blocks) { // С поворотами и отр.
      current=Координаты тек. преобразования;
      D=image->CopyBlock(current);
```

```
      current_distance = Rij.L2distance(D);
      if(current_distance < min_distance) {
         // Если коэффициенты best хуже:
         min_distance = current_distance;
         best = current;
      }
   } // Next domain block
   Save_Coefficients_to_file(best);
} // Next range block
```

Как видно из приведенного алгоритма, для каждого рангового блока делаем его проверку со всеми возможными доменными блоками (в том числе с прошедшими преобразование симметрии), находим вариант с наименьшей мерой L_2 (наименьшим среднеквадратичным отклонением) и сохраняем коэффициенты этого преобразования в файл. Коэффициенты – это (1) координаты найденного блока, (2) число от 0 до 7, характеризующее преобразование симметрии (поворот, отражение блока), и (3) сдвиг по яркости для этой пары блоков. Сдвиг по яркости вычисляется как.

$$q = \left[\sum_{i=1}^{n} \sum_{j=1}^{n} 0.75 \cdot d_{ij} - \sum_{i=1}^{n} \sum_{j=1}^{n} r_{ij} \right] / n^2,$$

где r_{ij} – значения пикселов рангового блока (R), а d_{ij} – значения пикселов доменного блока (D). При этом мера считается как

$$d(R,D) = \sum_{i=1}^{n} \sum_{j=1}^{n} (r_{ij} + q - 0.75 \cdot d_{ij})^2 .$$

Мы не вычисляем квадратного корня из L_2 меры и не делим ее на n, поскольку данные преобразования монотонны и не помешают нам найти экстремум, однако мы сможем выполнять на две операции меньше для каждого блока.

Посчитаем количество операций, необходимых нам для сжатия изображения в градациях серого 256 цветов 512x512 пикселов при размере блока 8 пикселов:

Часть программы	Число операций
for (all range blocks)	4096 (=512/8·512/8)
for (all domain blocks) + symmetry transformation	492032 (=(512/2-8)* (512/2-8)*8)
Вычисление q и $d(R,D)$	> 3*64 операций "+" > 2*64 операций "·"
Итог:	> 3* 128.983.236.608 операций "+" > 2* 128.983.236.608 операций "·"

Таким образом, нам удалось уменьшить число операций алгоритма компрессии до вполне вычисляемых величин.

Схема алгоритма декомпрессии изображений

Декомпрессия алгоритма фрактального сжатия чрезвычайно проста. Необходимо провести несколько итераций трехмерных аффинных преобразований, коэффициенты которых были получены на этапе компрессии.

В качестве начального может быть взято абсолютно любое изображение (например, абсолютно черное), поскольку соответствующий математический аппарат гарантирует нам сходимость последовательности изображений, получаемых в ходе итераций IFS, к неподвижному изображению (близкому к исходному). Обычно для этого достаточно 16 итераций.

```
Прочитаем из файла коэффициенты всех блоков;
Создадим черное изображение нужного размера;
Until(изображение не станет неподвижным){
   For(every range (R)){
      D=image->CopyBlock(D_coord_for_R);
      For(every pixel(i,j) in the block{
         Rij = 0.75Dij + q;
      } //Next pixel
   } //Next block
}//Until end
```

Поскольку мы записывали коэффициенты для блоков R_{ij} (которые, как мы оговорили, в нашем частном случае являются квадратами одинакового размера) *последовательно*, то получается, что мы последовательно заполняем изображение по квадратам сетки разбиения использованием аффинного преобразования.

Как можно подсчитать, количество операций на 1 пиксел изображения в градациях серого при восстановлении необычайно мало (N операций сложения "+" и N операций умножения "·", где N – количество итераций, т. е. 7–16). Благодаря этому декомпрессия изображений для фрактального алгоритма проходит быстрее декомпрессии, например, для алгоритма JPEG. В простой реализации JPEG на точку приходится 64 операции сложения "+" и 64 операции умножения "·". При реализации быстрого ДКП можно получить 7 сложений и 5 умножений на точку, но это без учета шагов RLE, квантования и кодирования по Хаффману. При этом для фрактального алгоритма умножение происходит на рациональное число, одно для каждого блока. Это означает, что мы можем, во-первых, использовать целочисленную рациональную арифметику, которая быстрее арифметики с плавающей точкой. Во-вторых, можно использовать умножение вектора на число – более простую и быструю операцию, часто закладываемую в архитектуру процессора (процессоры SGI, Intel MMX, векторные операции Athlon

и т. д.). Для полноцветного изображения ситуация качественно не изменяется, поскольку перевод в другое цветовое пространство используют оба алгоритма.

Оценка потерь и способы их регулирования

При кратком изложении упрощенного варианта алгоритма были пропущены многие важные вопросы. Например, что делать, если алгоритм не может подобрать для какого-либо фрагмента изображения подобный ему. Достаточно очевидное решение – разбить этот фрагмент на более мелкие и попытаться поискать для них. В то же время понятно, что эту процедуру нельзя повторять до бесконечности, иначе количество необходимых преобразований станет так велико, что алгоритм перестанет быть алгоритмом компрессии. Следовательно, мы допускаем потери в какой-то части изображения.

Для фрактального алгоритма компрессии, как и для других алгоритмов сжатия с потерями, очень важны механизмы, с помощью которых можно будет регулировать степень сжатия и степень потерь. К настоящему времени разработан достаточно большой набор таких методов. Во-первых, можно ограничить количество аффинных преобразований, заведомо обеспечив степень сжатия не ниже фиксированной величины. Во-вторых, можно потребовать, чтобы в ситуации, когда разница между обрабатываемым фрагментом и наилучшим его приближением будет выше определенного порогового значения, этот фрагмент дробился обязательно (для него обязательно заводится несколько линз). В-третьих, можно запретить дробить фрагменты размером меньше, допустим, четырех точек. Изменяя пороговые значения и приоритет этих условий, мы будем очень гибко управлять коэффициентом компрессии изображения в диапазоне от побитового соответствия до любой степени сжатия. Заметим, что эта гибкость будет гораздо выше, чем у ближайшего "конкурента" – алгоритма JPEG.

Характеристики фрактального алгоритма:

Степень сжатия: 2–2000 (задается пользователем).

Класс изображений: полноцветные 24 битовые изображения или изображения в градациях серого без резких переходов цветов (фотографии). Желательно, чтобы области большей значимости (для восприятия) были более контрастными и резкими, а области меньшей значимости – неконтрастными и размытыми.

Симметричность: 100–100 000.

Характерные особенности: может свободно масштабировать изображение при разжатии, увеличивая его в 2–4 раза без появления "лестничного эффекта". При увеличении степени компрессии появляется "блочный" эффект на границах блоков в изображении.

Рекурсивный (волновой) алгоритм

Английское название рекурсивного сжатия – wavelet. На русский язык оно переводится как волновое сжатие, как сжатие с использованием всплесков, а в последнее время и как вэйвлет-сжатие. Этот вид сжатия известен довольно давно и напрямую исходит из идеи использования когерентности областей. Ориентирован алгоритм на цветные и черно-белые изображения с плавными переходами. Идеален для картинок типа рентгеновских снимков. Степень сжатия задается и варьируется в пределах 5–100. При попытке задать больший коэффициент на резких границах, особенно проходящих по диагонали, проявляется лестничный эффект – ступеньки разной яркости размером в несколько пикселов.

Идея алгоритма заключается в том, что мы сохраняем в файл разницу – число между средними значениями соседних блоков в изображении, которая обычно принимает значения, близкие к нулю.

Так, два числа a_{2i} и a_{2i+1} всегда можно представить в виде $b^1_i=(a_{2i}+a_{2i+1})/2$ и $b^2_i=(a_{2i}-a_{2i+1})/2$. Аналогично последовательность a_i может быть попарно переведена в последовательность $b^{1,2}_i$.

Разберем конкретный пример: пусть мы сжимаем строку из восьми значений яркости пикселов (a_i): (220, 211, 212, 218, 217, 214, 210, 202). Мы получим следующие последовательности b^1_i и b^2_i: (215.5, 215, 215.5, 206) и (4.5, -3, 1.5, 4). Заметим, что значения b^2_i достаточно близки к нулю. Повторим операцию, рассматривая b^1_i как a_i. Данное действие выполняется как бы рекурсивно, откуда и название алгоритма. Мы получим из (215.5, 215, 215.5, 206): (215.25, 210.75) (0.25, 4.75). Полученные коэффициенты, округлив до целых и сжав, например, с помощью алгоритма Хаффмана с фиксированными таблицами, мы можем поместить в файл.

Заметим, что мы применяли наше преобразование к цепочке только 2 раза. Реально мы можем позволить себе применение wavelet-преобразования 4–6 раз. Более того, дополнительное сжатие можно получить, используя таблицы алгоритма Хаффмана с неравномерным шагом (т. е. нам придется сохранять код Хаффмана для ближайшего в таблице значения). Эти приемы позволяют достичь заметных степеней сжатия.

Упражнение. Мы восстановили из файла цепочку (215, 211) (0, 5) (5, -3, 2, 4) (см. пример). Постройте строку из восьми значений яркости пикселов, которую воссоздаст алгоритм волнового сжатия.

Алгоритм для двумерных данных реализуется аналогично. Если у нас есть квадрат из четырех точек с яркостями $a_{2i,2j}$, $a_{2i+1,2j}$, $a_{2i,2j+1}$ и $a_{2i+1,2j+1}$, то

$$b^1_{i,j} = (a_{2i,2j} + a_{2i+1,2j} + a_{2i,2j+1} + a_{2i+1,2j+1})/4;$$
$$b^2_{i,j} = (a_{2i,2j} + a_{2i+1,2j} - a_{2i,2j+1} - a_{2i+1,2j+1})/4;$$
$$b^3_{i,j} = (a_{2i,2j} - a_{2i+1,2j} + a_{2i,2j+1} - a_{2i+1,2j+1})/4;$$
$$b^4_{i,j} = (a_{2i,2j} - a_{2i+1,2j} - a_{2i,2j+1} + a_{2i+1,2j+1})/4.$$

Используя эти формулы, мы для изображения 512x512 пикселов получим после первого преобразования 4 матрицы размером 256x256 элементов (рис. 2.7).

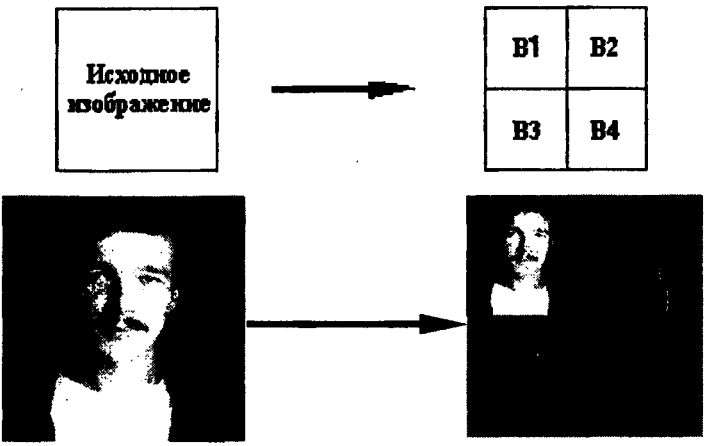


Рис. 2.7. Вид двумерного wavelet-преобразования

В первой, как легко догадаться, будет храниться уменьшенная копия изображения. Во второй – усредненные разности пар значений пикселов по горизонтали. В третьей – усредненные разности пар значений пикселов по вертикали. В четвертой – усредненные разности значений пикселов по диагонали. По аналогии с двумерным случаем мы можем повторить наше преобразование и получить вместо первой матрицы 4 матрицы размером 128x128. Повторив наше преобразование в третий раз, мы получим в итоге: 4 матрицы 64x64, 3 матрицы 128x128 и 3 матрицы 256x256. На практике при записи в файл значениями, получаемыми в последней строке ($b^4_{i,j}$), обычно пренебрегают (сразу получая выигрыш примерно на треть размера файла – 1- 1/4 - 1/16 - 1/64...).

К достоинствам этого алгоритма можно отнести то, что он очень легко позволяет реализовать возможность постепенного "проявления" изображения при передаче изображения по сети. Кроме того, поскольку в начале

изображения мы фактически храним его уменьшенную копию, упрощается показ "огрубленного" изображения по заголовку.

В отличие от JPEG и фрактального алгоритма данный метод не оперирует блоками, например, 8x8 пикселов. Точнее, мы оперируем блоками 2x2, 4x4, 8x8 и т. д. Однако за счет того, что коэффициенты для этих блоков мы сохраняем независимо, мы можем достаточно легко избежать дробления изображения на "мозаичные" квадраты.

Характеристики волнового алгоритма:
Степень: 2–200 (задается пользователем).
Класс изображений: как у фрактального и JPEG.
Симметричность: ~1.5.
Характерные особенности: кроме того, при высокой степени сжатия изображение распадается на отдельные блоки.

Алгоритм JPEG 2000

Алгоритм JPEG 2000 разработан той же группой экспертов в области фотографии, что и JPEG. Формирование JPEG как международного стандарта было закончено в 1992 г. В 1997 г. стало ясно, что необходим новый, более гибкий и мощный стандарт, который и был доработан к зиме 2000 г. Основные отличия алгоритма в JPEG 2000 от алгоритма в JPEG заключаются в следующем.

Лучшее качество изображения при сильной степени сжатия. Или, что то же самое, бóльшая степень сжатия при том же качестве для высоких степеней сжатия. Фактически это означает заметное уменьшение размеров графики "Web-качества", используемой большинством сайтов.

Поддержка кодирования отдельных областей с лучшим качеством. Известно, что отдельные области изображения критичны для восприятия человеком (например, глаза на фотографии), в то время как качеством других можно пожертвовать (например, задним планом). При "ручной" оптимизации увеличение степени сжатия проводится до тех пор, пока не будет потеряно качество в какой-то важной части изображения. Сейчас появляется возможность задать качество в критических областях, сжав остальные области сильнее, т. е. мы получаем еще большую окончательную степень сжатия при субъективно равном качестве изображения.

Основной алгоритм сжатия заменен на wavelet. Помимо указанного повышения степени сжатия это позволило избавиться от 8-пиксельной блочности, возникающей при повышении степени сжатия. Кроме того, плавное проявление изображения теперь изначально заложено в стандарт

(Progressive JPEG, активно применяемый в Интернете, появился много позднее JPEG).

Для повышения степени сжатия в алгоритме используется арифметическое сжатие. Изначально в стандарте JPEG также было заложено арифметическое сжатие, однако позднее оно было заменено менее эффективным сжатием по Хаффману, поскольку арифметическое сжатие было защищено патентами. Сейчас срок действия основного патента истек и появилась возможность улучшить алгоритм.

Поддержка сжатия без потерь. Помимо привычного сжатия с потерями новый JPEG теперь будет поддерживать и сжатие без потерь. Таким образом, становится возможным использование JPEG для сжатия медицинских изображений, в полиграфии, при сохранении текста под распознавание OCR системами и т. д.

Поддержка сжатия 1-битовых (2-цветных) изображений. Для сохранения 1-битовых изображений (рисунки тушью, отсканированный текст и т. п.) ранее повсеместно рекомендовался формат GIF, поскольку сжатие с использованием ДКП весьма неэффективно для изображений с резкими переходами цветов. В JPEG при сжатии 1-битовая картинка приводилась к 8-битовой, т. е. увеличивалась в 8 раз, после чего делалась попытка сжимать, нередко менее чем в 8 раз. Сейчас можно рекомендовать JPEG 2000 как универсальный алгоритм.

На уровне формата поддерживается прозрачность. Плавно накладывать фон при создании WWW-страниц теперь можно будет не только в GIF, но и в JPEG 2000. Кроме того, поддерживается не только 1 бит прозрачности (пиксел прозрачен/непрозрачен), а отдельный канал, что позволит задавать плавный переход от непрозрачного изображения к прозрачному фону.

Кроме того, на уровне формата поддерживаются включение в изображение информации о копирайте, поддержка устойчивости к битовым ошибкам при передаче и широковещании, можно запрашивать для декомпрессии или обработки внешние средства (plug-ins), можно включать в изображение его описание, информацию для поиска и т. д.

Идея алгоритма

Базовая схема JPEG 2000 очень похожа на базовую схему JPEG. Отличия заключаются в следующем:

- вместо дискретного косинусного преобразования (DCT) используется дискретное wavelet -преобразование (DWT);
- вместо кодирования по Хаффману используется арифметическое сжатие;
- в алгоритм изначально заложено управление качеством областей изображения;

- не используется явно дискретизация компонент U и V после преобразования цветовых пространств, поскольку при DWT можно достичь того же результата, но более аккуратно.

Рассмотрим алгоритм по шагам (рис. 2.8).

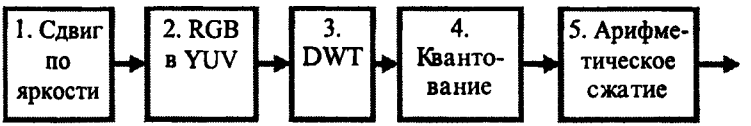


Рис. 2.8. Конвейер операций, используемый в алгоритме JPEG 2000

Шаг 1. В JPEG 2000 предусмотрен сдвиг яркости (DC level shift) каждой компоненты (RGB) изображения перед преобразованием в YUV. Это делается для выравнивания динамического диапазона (приближения к нулю гистограммы частот), что приводит к увеличению степени сжатия. Формулу преобразования можно записать как:

$$I'(x, y) = I(x, y) - 2^{ST-1}.$$

Значение степени ST для каждой компоненты R, G и B свое (определяется при сжатии компрессором). При восстановлении изображения выполняется обратное преобразование:

$$I'(x, y) = I(x, y) + 2^{ST-1}.$$

Шаг 2. Переводим изображение из цветового пространства RGB с компонентами, отвечающими за красную (Red), зеленую (Green) и синюю (Blue) составляющие цвета точки, в цветовое пространство YUV. Этот шаг аналогичен JPEG (см. матрицы преобразования в описании JPEG), за тем исключением, что кроме преобразования с потерями предусмотрено также и преобразование без потерь. Его матрица выглядит так:

$$\begin{vmatrix} Y \\ U \\ V \end{vmatrix} = \begin{pmatrix} \left\lfloor \frac{R + 2G + B}{4} \right\rfloor \\ R-G \\ B-G \end{pmatrix}.$$

Обратное преобразование осуществляется с помощью обратной матрицы:

$$\begin{vmatrix} R \\ G \\ B \end{vmatrix} = \begin{vmatrix} U + G \\ Y - \left\lfloor \frac{U + V}{4} \right\rfloor \\ V + G \end{vmatrix}.$$

Шаг 3. Дискретное wavelet-преобразование (DWT) также может быть двух видов – для случая сжатия с потерями и для сжатия без потерь. Его коэффициенты задаются табл. 2.1 и 2.2.

Таблица 2.1. Коэффициенты для сжатия с потерями

Коэффициенты при упаковке		
i	Низкочастотные коэффициенты $h_L(i)$	Высокочастотные коэффициенты $h_H(i)$
0	1.115087052456994	0.6029490182363579
±1	0.5912717631142470	-0.2668641184428723
±2	-0.05754352622849957	-0.07822326652898785
±3	-0.09127176311424948	0.01686411844287495
±4	0	0.02674875741080976
Другие i	0	0
Коэффициенты при распаковке		
i	Низкочастотные коэффициенты $g_L(i)$	Высокочастотные коэффициенты $g_H(i)$
0	0.6029490182363579	1.115087052456994
±1	-0.2668641184428723	0.5912717631142470
±2	-0.07822326652898785	-0.05754352622849957
±3	0.01686411844287495	-0.09127176311424948
±4	0.02674875741080976	0
Другие *i*	0	0

Таблица 2.2. Коэффициенты для сжатия без потерь

	При упаковке		При распаковке	
i	Низкочастотные коэффициенты $h_L(i)$	Высокочастотные коэффициенты $h_H(i)$	Низкочастотные коэффициенты $g_L(i)$	Высокочастотные коэффициенты $g_H(i)$
0	6/8	1	1	6/8
±1	2/8	-1/2	1/2	-2/8
±2	-1/8	0	0	-1/8

Само преобразование в одномерном случае представляет собой скалярное произведение коэффициентов фильтра на строку преобразуемых значений (в нашем случае – на строку изображения). При этом четные выходящие значения формируются с помощью низкочастотного преобразования, а нечетные – с помощью высокочастотного:

$$y_{output}(2n) = \sum_{j=0}^{N-1} x_{input}(j) \cdot h_L(j-2n),$$

$$y_{output}(2n+1) = \sum_{j=0}^{N-1} x_{input}(j) \cdot h_H(j-2n-1).$$

Поскольку большинство $h_L(i)$, кроме окрестности i=0, равны нулю, то можно переписать приведенные формулы с меньшим количеством операций. Для простоты рассмотрим случай сжатия без потерь.

$$y_{out}(2n) = \frac{-x_{in}(2n-2) + 2 \cdot x_{in}(2n-1) + 6 \cdot x_{in}(2n) + 2 \cdot x_{in}(2n+1) - x_{in}(2n+2)}{8}$$

$$y_{out}(2n+1) = -\frac{x_{in}(2n)}{2} + x_{in}(2n+1) - \frac{x_{in}(2n+2)}{2}.$$

Легко показать, что данную запись можно эквивалентно переписать, уменьшив еще втрое количество операций умножения и деления (однако теперь необходимо будет подсчитать сначала все нечетные y). Добавим также операции округления до ближайшего целого, не превышающего заданное число a, обозначаемые как $\lfloor a \rfloor$:

$$y_{out}(2n+1) = x_{in}(2n+1) - \left\lfloor \frac{x_{in}(2n) + x_{in}(2n+2)}{2} \right\rfloor,$$

$$y_{out}(2n) = x_{in}(2n) + \left\lfloor \frac{y_{out}(2n-1) + y_{out}(2n+1) + 2}{4} \right\rfloor.$$

Упражнение. Самостоятельно уменьшите количество операций для случая без потерь.

Рассмотрим на примере, как работает данное преобразование. Для того чтобы преобразование можно было применять к крайним пикселам изображения, оно симметрично достраивается в обе стороны на несколько пикселов, как показано на рисунке ниже. В худшем случае (сжатие с потерями) нам необходимо достроить изображение на 4 пиксела (рис. 2.9).

Рис. 2.9. Симметричное расширение изображения (яркости АБ...Е) по строке вправо и влево

Пусть мы преобразуем строку из 10 пикселов. Расширим ее значения вправо и влево и применим DWT-преобразование:

n	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11
x_{in}	3	2	1	2	3	7	10	15	12	9	10	5	10	9
y_{out}		0	1	0	3	1	11	4	13	-2	8	-5		

Получившаяся строка 1, 0, 3, 1, 11, 4, 13, -2, 8, -5 и является цепочкой, однозначно задающей исходные данные. Совершив аналогичные преобразования с коэффициентами для распаковки, приведенными выше в таблице, получим необходимые формулы:

$$x_{out}(2n) = y_{out}(2n) - \left\lfloor \frac{y_{out}(2n-1) + y_{out}(2n+1) + 2}{4} \right\rfloor,$$

$$x_{out}(2n+1) = y_{out}(2n+1) + \left\lfloor \frac{x_{out}(2n) + x_{out}(2n+2)}{2} \right\rfloor,$$

Упражнение. Докажите, что во всех случаях округления мы будем получать одинаковые входную и выходную цепочки.

Легко проверить (используя преобразование упаковки), что значения на концах строк в y_{out} также симметричны относительно n=0 и 9. Воспользовавшись этим свойством, расширим нашу строку вправо и влево и применим обратное преобразование:

n	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11
y_{out}		0	1	0	3	1	11	4	13	-2	8	-5	8	-2
x_{out}			1	2	3	7	10	15	12	9	10	5	10	

Как видим, мы получили исходную цепочку ($x_{in} = x_{out}$).

Упражнение. Примените прямое и обратное DWT-преобразования к цепочке из 10 байт: 121, 107, 98, 102, 145, 182, 169, 174, 157, 155.

Далее к строке применяется чересстрочное преобразование, суть которого заключается в том, что все четные коэффициенты переписываются в начало строки, а все нечетные – в конец. В результате этого преобразования в начале строки формируется "уменьшенная копия" всей строки (низкочастотная составляющая), а в конце строки – информация о колебаниях значений промежуточных пикселов (высокочастотная составляющая).

y_{out}	1	0	3	1	11	4	13	-2	8	-5
y'_{out}	1	3	11	13	8	0	1	4	-2	-5

Это преобразование применяется сначала ко всем строкам изображения, а затем ко всем столбцам изображения. В результате изображение делится на 4 квадранта (примеры смотрите в описании рекурсивного сжатия). В первом квадранте будет сформирована уменьшенная копия изображения, а в остальных трех – высокочастотная информация. После чего преобразование повторно

применяется уже только к первому квадранту изображения по тем же правилам (преобразование второго уровня) (рис. 2.10).

Рис. 2.10. Вид DWT

Для корректного сохранения результатов под данные 2-го и 3-го квадрантов выделяется на 1 бит больше, а под данные 4-го квадранта – на 2 бита больше. То есть если исходные данные были 8-битовые, то на 2-й и 3-й квадранты нужно 9 бит, а на 4-й – 10, независимо от уровня применения DWT. При записи коэффициентов в файл можно использовать иерархическую структуру DWT, помещая коэффициенты преобразований с большего уровня в начало файла. Это позволяет получить "изображение для предварительного просмотра", прочитав небольшой участок данных из начала файла, а не распаковывая весь файл, как это приходилось делать при сжатии изображения целиком. Иерархичность преобразования может также использоваться для плавного улучшения качества изображения при передаче его по сети.

Шаг 4. Так же как и в алгоритме JPEG, после DWT применяется квантование. Коэффициенты квадрантов делятся на заранее заданное число. При увеличении этого числа снижается динамический диапазон коэффициентов, они становятся ближе к нулю, и мы получаем большую степень сжатия. Варьируя эти числа для разных уровней преобразования, для разных цветовых компонент и для разных квадрантов, мы очень гибко управляем степенью потерь в изображении. Рассчитанные в компрессоре оптимальные коэффициенты квантования передаются в декомпрессор для однозначной распаковки.

Шаг 5. Для сжатия получающихся массивов данных в JPEG 2000 используется вариант арифметического сжатия, называемый MQ-кодером, прообраз которого (QM-кодер) рассматривался еще в стандарте JPEG, но реально не использовался из-за патентных ограничений. Подробнее об алгоритме арифметического сжатия читайте в гл.1 разд. 1.

Области повышенного качества

Основная задача, которую мы решаем, – повышение степени сжатия изображений. Когда практически достигнут предел сжатия изображения в целом и различные методы дают очень небольшой выигрыш, мы можем существенно (в разы) увеличить степень сжатия за счет изменения качества разных участков изображения (рис. 2.11).

Проблемой этого подхода является то, что необходимо каким-то образом получать расположение наиболее важных для человека участков изображения. Например, таким участком на фотографии человека является лицо, а на лице – глаза. Если при сжатии портрета с большими потерями будут размыты предметы, находящиеся на заднем плане – это будет несущественно. Однако если будет размыто лицо или глаза – экспертная оценка степени потерь будет хуже.

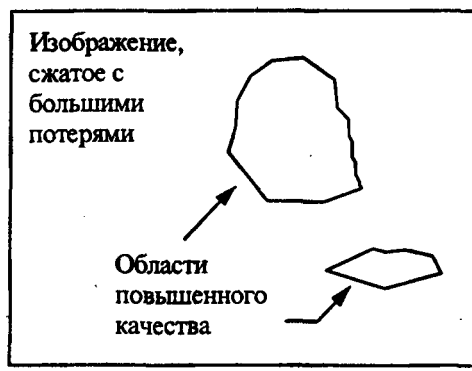


Рис. 2.11. Локальное улучшение качества областей изображения

Работы по автоматическому выделению таких областей активно ведутся. В частности, созданы алгоритмы автоматического выделения лиц на изображениях. Продолжаются исследования методов выделения наиболее значимых (при анализе изображения мозгом человека) контуров и т. д. Однако очевидно, что универсальный алгоритм в ближайшее время создан не будет, поскольку для этого требуется построить полную схему восприятия изображений мозгом человека.

На сегодня вполне реально применение полуавтоматических систем, в которых качество областей изображения будет задаваться интерактивно. Данный подход уменьшает количество возможных областей применения модифицированного алгоритма, но *позволяет достичь большей степени сжатия*.

Такой подход логично применять, если:

- для приложения должна быть критична (максимальна) степень сжатия, причем настолько, что возможен индивидуальный подход к каждому изображению;
- изображение сжимается один раз, а разжимается множество раз.

В качестве примеров приложений, удовлетворяющих этим ограничениям, можно привести практически все мультимедийные продукты на CD-ROM. И для CD-ROM энциклопедий, и для игр важно записать на диск как можно больше информации, а графика, как правило, занимает до 70% всего объема диска. При этом технология производства дисков позволяет сжимать каждое изображение индивидуально, максимально повышая степень сжатия.

Интересным примером являются WWW-сервер. Для них тоже, как правило, выполняются оба изложенных выше условия. При этом совершенно не обязательно индивидуально подходить к каждому изображению, поскольку, по статистике, 10% изображений будут запрашиваться 90% раз. То есть для крупных справочных или игровых серверов появляется возможность уменьшать время загрузки изображений и степень загруженности каналов связи адаптивно.

В JPEG 2000 используется 1-битовое изображение-маска, задающее повышение качества в данной области изображения. Поскольку за качество областей у нас отвечают коэффициенты DWT-преобразования во 2, 3 и 4-м квадрантах, то маска преобразуется таким образом, чтобы указывать на все коэффициенты, соответствующие областям повышения качества (рис. 2.12)

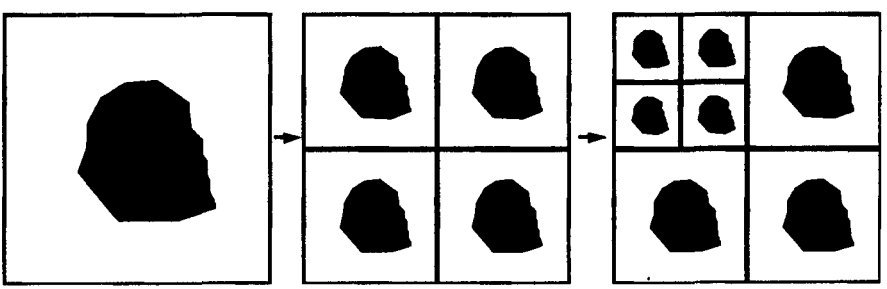


Рис. 2.12. Преобразование маски области повышения качества для обработки DWT-коэффициентов

Эти области обрабатываются далее другими алгоритмами (с меньшими потерями), что и позволяет достичь искомого баланса по общему качеству и степени сжатия.

Характеристики алгоритма JPEG 2000:

Степень сжатия: 2–200 (задается пользователем). Возможно сжатие без потерь.

Класс изображений: полноцветные 24-битовые изображения. Изображения в градациях серого без резких переходов цветов (фотографии). 1-битовые изображения.

Симметричность: 1–1.5.

Характерные особенности: позволяет удалять визуально неприятные эффекты, повышая качество в отдельных областях. При сильном сжатии появляется блочность и большие волны в вертикальном и горизонтальном направлениях.

Заключение

В заключение рассмотрим табл. 2.3 и 2.4, в которых сводятся воедино параметры различных алгоритмов сжатия изображений, рассмотренных нами выше.

Таблица 2.3

Алгоритм	Особенности изображения, за счет которых происходит сжатие
RLE	Подряд идущие одинаковые цвета: 2 2 2 2 2 2 15 15 15
LZW	Одинаковые подцепочки: 2 3 15 40 2 3 15 40
Хаффмана	Разная частота появления цвета: 2 2 3 2 2 4 3 2 2 2 4
CCITT-3	Преобладание белого цвета в изображении, большие области, заполненные одним цветом
Рекурсивный	Плавные переходы цветов и отсутствие резких границ
JPEG	Отсутствие резких границ
Фрактальный	Подобие между элементами изображения

Таблица 2.4

Алгоритм	Коэффициенты сжатия	Симметричность по времени	На что ориентирован	Потери	Размерность
RLE	32, 2, 0.5	1	3,4-х битовые	Нет	1D
LZW	1 000, 4, 5/7	1.2-3	1-8 битовые	"	1D
Хаффмана	8, 1.5, 1	1-1.5	8 битовые	"	1D
CCITT-3	213(3), 5, 0.25	~1	1-битовые	"	1D
JBIG	2–30 раз	~1	1-битовые	"	2D
Lossless JPEG	2 раза	~1	24-бит. сер.	"	2D
Рекурсивное сжатие	2–200 раз	1.5	24-битовые, серые	Да	2D
JPEG	2–200 раз	~1	24-битовые, сер.	"	2D
Фрактальный	2–2 000 раз	1 000–10 000	24-бит. сер.	"	2.5D

В табл. 2.5 отчетливо видны тенденции развития алгоритмов сжатия изображения последних лет:

- ориентация на фотореалистичные изображения с 16 млн. цветов (24 бита);
- использование сжатия с потерями, возможность за счет потерь регулировать качество сжатых изображений;
- использование избыточности изображений в двух измерениях;
- появление существенно несимметричных алгоритмов;
- увеличивающаяся степень сжатия изображений.

Вопросы для самоконтроля

1. В чем разница между алгоритмами с потерей информации и без потери информации?
2. Приведите примеры мер потери информации и опишите их недостатки.
3. За счет чего сжимает изображения алгоритм JPEG?
4. В чем заключается идея алгоритма фрактального сжатия?
5. В чем заключается идея рекурсивного (волнового) сжатия?
6. Можно ли применять прием перевода в другое цветовое пространство алгоритма JPEG в других алгоритмах компрессии?
7. Сравните приведенные в этой главе алгоритмы сжатия изображений.

Глава 3. Различия между форматом и алгоритмом

Напоследок несколько замечаний относительно разницы в терминологии, путаницы при сравнении рейтингов алгоритмов и т. п.

Посмотрите на краткий перечень *форматов*, достаточно часто используемых **на PC, Apple и UNIX платформах**: ADEX, Alpha Microsystems BMP, Autologic, AVHRR, Binary Information File (BIF), Calcomp CCRF, CALS, Core IDC, Cubicomp PictureMaker, Dr. Halo CUT, Encapsulated PostScript, ER Mapper Raster, Erdas LAN/GIS, First Publisher ART, GEM VDI Image File, GIF, GOES, Hitachi Raster Format, PCL, RTL, HP-48sx Graphic Object (GROB), HSI JPEG, HSI Raw, IFF/ILBM, Img Software Set, Jovian VI, JPEG/JFIF, Lumena CEL, Macintosh PICT/PICT2, MacPaint, MTV Ray Tracer Format, OS/2 Bitmap, PCPAINT/Pictor Page Format, PCX, PDS, Portable BitMap (PBM), QDV, QRT Raw, RIX, Scodl, Silicon Graphics Image, SPOT Image, Stork, Sun Icon, Sun Raster, Targa, TIFF, Utah Raster Toolkit Format, VITec, Vivid Format, Windows Bitmap, WordPerfect Graphic File, XBM, XPM, XWD.

В оглавлении вы можете видеть список *алгоритмов компрессии*. Единственным совпадением оказывается JPEG, а это, согласитесь, не повод, чтобы повсеместно использовать слова *формат* и *алгоритм компрессии* как синонимы (что, увы, можно часто наблюдать).

Между этими двумя множествами нет взаимно-однозначного соответствия. Так, ***различные модификации*** *алгоритма* RLE реализованы в огромном количестве *форматов*. В том числе в TIFF, BMP, PCX. И **если в определенном формате какой-либо файл занимает много места, это не означает, что плох соответствующий алгоритм** компрессии. Это означает зачастую лишь то, что *реализация алгоритма*, использованная в этом *формате*, дает *для данного изображения* плохие результаты. Не более того. (См. примеры в приложении 2.)

В то же время многие современные *форматы* поддерживают запись с использованием нескольких *алгоритмов архивации* либо без использования архивации. Например, *формат* TIFF 6.0 может сохранять изображения с использованием *алгоритмов* RLE-PackBits, RLE-CCITT, LZW, Хаффмана с фиксированной таблицей, JPEG, а может сохранять изображение без архивации. Аналогично *форматы* BMP и TGA позволяют сохранять файлы как с использованием *алгоритма* компрессии *RLE* (разных модификаций!), так и без использования оного.

Вывод 1. Для многих *форматов*, говоря о размере файлов, **необходимо** указывать, использовался ли *алгоритм* компрессии, и если использовался, то какой.

Можно пополнить перечень ситуаций некорректного сравнения алгоритмов. При сохранении абсолютно черного изображения в формате 1000x1000x256 цветов в формате BMP без компрессии мы получаем, как и положено, файл размером чуть более 1 000 000 байт, а при сохранении с компрессией RLE, можно получить файл размером 64 байта. Это был бы превосходный результат – сжатие в 15 тыс. раз(!), если бы к нему имела отношение компрессия. Дело в том, что данный файл в 64 байта состоит только из заголовка изображения, в котором указаны все его данные. Несмотря на то что такая короткая запись изображения стала возможна именно благодаря особенности реализации RLE в BMP, еще раз подчеркнем, что в данном случае *алгоритм* компрессии *даже не применялся*. И то, что для абсолютно черного изображения 4000x4000x256 мы получаем коэффициент сжатия 250 тыс. раз, совсем не повод для продолжительных эмоций по поводу эффективности RLE. Кстати, данный результат возможен лишь при определенном положении цветов в палитре и далеко не на всех программах, которые умеют записывать BMP с архивацией RLE (однако все стандартные

средства, в том числе средства системы Windows, читают такой сжатый файл нормально).

Всегда полезно помнить, что на размер файла оказывают существенное влияние большое количество параметров (вариант реализации алгоритма, параметры алгоритма – как внутренние, так и задаваемые пользователем, – порядок цветов в палитре и многое другое). Например, для **абсолютно черного** изображения 1000x1000x256 градаций серого в формате JPEG с помощью одной программы при различных параметрах *всегда* получался файл примерно в 7 Кб. В то же время, меняя опции в другой программе, я получил файлы размером от 4 до 68 Кб (всего-то на порядок разницы). При этом декомпрессированное изображение для всех файлов было одинаковым – *абсолютно* черный квадрат (яркость 0 для всех точек изображения).

Дело в том, что даже для простых форматов *одно и то же **изображение** в одном и том же **формате*** с использованием *одного и того же **алгоритма*** архивации можно записать в файл ***несколькими корректными способами***. Для сложных форматов и алгоритмов архивации возникают ситуации, когда многие программы сохраняют изображения разными способами. Такая ситуация, например, сложилась с форматом TIFF (в силу его большой гибкости). Долгое время по-разному сохраняли изображения в *формат* JPEG, поскольку соответствующая группа ISO (Международной организации по стандартизации) подготовила только стандарт *алгоритма*, но не стандарт *формата*. Сделано так было для того, чтобы не вызывать "войны форматов". Абсолютно противоположное положение сейчас с фрактальной компрессией, поскольку есть стандарт де-факто на сохранение фрактальных коэффициентов в файл (стандарт *формата*), но *алгоритм* их нахождения (быстрого нахождения!) является технологической тайной создателей программ-компрессоров. В результате для вполне стандартной программы-декомпрессора могут быть подготовлены файлы с коэффициентами, существенно различающиеся как по размеру, так и по качеству получающегося изображения.

Приведенные примеры показывают, что встречаются ситуации, когда алгоритмы записи изображения в файл в различных программах различаются. Однако гораздо чаще причиной разницы файлов являются разные *параметры алгоритма*. Как уже говорилось, многие алгоритмы позволяют в известных пределах менять свои параметры, но не все программы позволяют это делать пользователю.

Вывод 2. Если вы не умеете пользоваться программами архивации или пользуетесь программами, в которых "для простоты использования" убрано управление параметрами алгоритма, не удивляйтесь, что для отличного алгоритма компрессии в результате получаются большие файлы.

Литература по алгоритмам сжатия

1. *Wallace G. K.* The JPEG still picture compression standard // Communication of ACM. April 1991. Vol. 34, № 4.
2. *Smith B., Rowe L.* Algorithm for manipulating compressed images // Computer Graphics and applications. September 1993.
3. *Jacquin A.*Fractal image coding based on a theory of iterated contractive image transformations // Visual Comm. and Image Processing. 1990. Vol. SPIE-1360.
4. *Fisher Y.* Fractal image compression // SigGraph-92.
5. Progressive Bi-level Image Compression, Revision 4.1 // ISO/IEC JTC1/SC2/WG9, CD 11544. 1991. September 16.
6. *Pennebaker W. B., Mitchell J. L., Langdon G. G., Arps R. B.* An overview of the basic principles of the Q-coder adaptive binary arithmetic coder // IBM Journal of research and development. November 1988. Vol.32, No.6. P. 771–726.
7. *Huffman D. A.* A method for the construction of minimum redundancy codes. // Proc. of IRE. 1952. Vol.40. P. 1098–1101.
8. Standardisation of Group 3 Facsimile apparatus for document transmission. CCITT Recommendations. Fascicle VII.2. 1980. T.4.
9. *Александров В. В., Горский Н. Д.* Представление и обработка изображений: рекурсивный подход // Л.: Наука, 1985. 190 с.
10. *Климов А. С.* Форматы графических файлов // С.-Пб.: ДиаСофт. 1995.
11. *Ватолин Д. С.* MPEG – стандарт ISO на видео в системах мультимедиа // Открытые системы. Лето 1995. № 2.
12. *Ватолин Д. С.* Тенденции развития алгоритмов архивации графики // Открытые системы. Зима 1995. № 4.
13. *Ватолин Д. С.* Алгоритмы сжатия изображений // М.: Диалог-МГУ, 1999.
14. *Добеши И.* Десять лекций по вейвлетам / Пер. с анг. Е. В. Мищенко, под ред. А. П. Петухова. М.; Ижевск, 2001. 464 с.
15. *Яншин В. В.* Анализ и обработка изображений (принципы и алгоритмы) // М.: Машиностроение,1995.
16. *Павлидис Т.* Алгоритмы машинной графики и обработка изображений // М.: Радио и связь 1986, 400 с.
17. *Претт У.* Цифровая обработка изображений в двух томах // М.: Мир, 1982. 790 с.
18. *Розеншельд А.* Распознавание и обработка изображений // М.: Мир, 1972. 232 с.
19. Материалы конференции "Графикон" (статьи по сжатию публиковались практически ежегодно) доступны в научных библиотеках и, частично, на http://www.graphicon.ru.

20. *Ярославский Л. П.* Введение в цифровую обработку изображений // М.: Сов. радио, 1969. 312 с.
21. *Яблонский С. В.* "Введение в дискретную математику" // М.: Наука, 1986. Раз. "Теория кодирования".
22. Barnsley M. F., Hurd L. P. Fractal Image Compression //A. K. Press Wellesley, Mass. 1993.
23. Более 150 статей по сжатию изображений можно найти на http://graphics.cs.msu.su/library/.

Литература по форматам изображений

24. *Климов А. С.* Форматы графических файлов // М.: НИПФ "ДиаСофт Лтд.", 1995.
25. *Романов В. Ю.* Популярные форматы файлов для хранения графических изображений на IBM PC // М.: Унитех, 1992.
26. *Сван Т.* "Форматы файлов Windows // М.: Бином, 1995.
27. *Hamilton E.* JPEG File Interchange Format // Version 1.2. September 1, 1992, San Jose CA: C-Cube Microsystems, Inc.
28. Aldus Corporation Developer's Desk. TIFF – Revision 6.0, Final. 1992. June 3.

РАЗДЕЛ 3
СЖАТИЕ ВИДЕОДАННЫХ

Введение

Основной сложностью при работе с видео являются большие объемы дискового пространства, необходимого для хранения даже небольших фрагментов. Причем даже применение современных алгоритмов сжатия не изменяет ситуацию кардинально. При записи на один компакт-диск "в бытовом качестве" на него можно поместить несколько тысяч фотографий, примерно 10 ч музыки и всего полчаса видео. Видео "телевизионного" формата 720x576 пикселов 25 кадров в секунду в системе RGB требует потока данных примерно в 240 Мбит/с (т. е. 1.8 Гб/мин). При этом традиционные алгоритмы сжатия изображений, ориентированные на отдельные кадры, не спасают ситуации, поскольку даже при уменьшении потока в 10 раз он составляет достаточно большие величины.

В результате подавляющее большинство сегодняшних алгоритмов сжатия видео являются алгоритмами с потерей данных. При сжатии используется несколько типов избыточности:

1) ***когерентность областей изображения*** – малое изменение цвета изображения в соседних пикселах (свойство, которое эксплуатируют все алгоритмы сжатия изображений с потерями);
2) ***избыточность в цветовых плоскостях*** – используется большая важность яркости изображения для восприятия;
3) ***подобие между кадрами*** – использование того факта, что на скорости 25 кадров в секунду, как правило, соседние кадры изменяются незначительно.

Первые два пункта знакомы вам по алгоритмам сжатия графики. Использование подобия между кадрами в самом простом и наиболее часто используемом случае означает кодирование не самого нового кадра, а его разности с предыдущим кадром. Для видео типа "говорящая голова" (передача новостей, видеотелефоны) бóльшая часть кадра остается неизменной и даже такой простой метод позволяет значительно уменьшить поток данных. Более сложный метод заключается в нахождении для каждого блока в сжимаемом кадре наименее отличающегося от него блока в кадре, используемом в качестве базового. Далее кодируется разница между этими блоками. Этот метод существенно более ресурсоемкий.

Основные понятия

Определимся с основными понятиями, которые используются при сжатии видео. Видеопоток характеризуется *разрешением*, *частотой кадров* и *системой представления цветов*. Из телевизионных стандартов пришли разрешения в 720x576 и 640x480 и частоты в 25 (стандарты PAL или SECAM) и 30 (стандарт NTSC) кадров в секунду. Для низких разрешений существуют специальные названия CIF – Common Interchange Format, равный 352x288, и QCIF – Quartered Common Interchange Format, равный 176x144. Поскольку CIF и QCIF ориентированы на крайне небольшие потоки, то с ними работают на частотах от 5 до 30 кадров в секунду.

Требования приложений к алгоритму

Для алгоритмов сжатия видео характерны большинство тех же требований приложений, которые предъявляются к алгоритмам сжатия графики, однако есть и определенная специфика:

Произвольный доступ – подразумевает возможность найти и показать любой кадр за ограниченное время. Обеспечивается наличием в потоке данных так называемых *точек входа* – кадров, сжатых независимо (т. е. как обычное статическое изображение). Приемлемым временем поиска произвольного кадра считается 1/2 с.

Быстрый поиск вперед/назад – подразумевает быстрый показ кадров, не следующих друг за другом в исходном потоке. Требует наличия дополнительной информации в потоке. Эта возможность активно используется всевозможными проигрывателями.

Показ кадров фильма в обратном направлении. Редко требуется в приложениях. При жестких ограничениях на время показа очередного кадра выполнение этого требования может резко уменьшить степень сжатия.

Аудиовизуальная синхронизация – самое серьезное требование. Данные, необходимые для того, чтобы добиться синхронности аудио и видео дорожек, существенно увеличивают размер фильма. Для видеосистемы это означает, что если мы не успеваем достать и показать в нужный момент времени некий кадр, то мы должны уметь корректно показать, например, кадр, следующий за ним. Если мы показываем фильм без звука, то можно позволить себе чуть более медленный или более быстрый показ. Во времена сравнительно несовершенного немого кино кадры шли настолько неравномерно, насколько неравномерно крутил ручку камеры оператор. Показ без звука фильма, *снятого* столь несовершенными методами, воспринимается нормально даже при условии, что частота *показываемых* кадров постоянна (и герои фильма передвигаются то карикатурно быстро, то медленно). Од-

нако смотреть фильм (например, боевик), в котором видеосистема не успевает за звуком, становится мучением.

Устойчивость к ошибкам – требование, обусловленное тем, что большинство каналов связи ненадежны. Испорченное помехой изображение должно быстро восстанавливаться. Требование достаточно легко удовлетворяется необходимым числом независимых кадров в потоке. При этом также уменьшается степень сжатия, так как на экране 2–3 с (50–75 кадров) может быть одно и то же изображение, но мы будем вынуждены нагружать поток независимыми кадрами.

Время кодирования/декодирования. Во многих системах (например, видеотелефонах) общая задержка на кодирование-передачу-декодирование должна составлять не более 150 мс. Кроме того, в приложениях, где необходимо редактирование, нормальная интерактивная работа невозможна, если время реакции системы составляет более 1 с.

Редактируемость. Под редактируемостью понимается возможность изменять все кадры так же легко, как если бы они были записаны независимо.

Масштабируемость – простота реализации концепции "видео в окне". Мы должны уметь быстро изменять высоту и ширину изображения в пикселах. Масштабирование способно породить неприятные эффекты в алгоритмах, основанных на ДКП (дискретном косинусном преобразовании). Корректно реализовать эту возможность для MPEG на данный момент можно, пожалуй, лишь при достаточно сложных аппаратных реализациях, только тогда алгоритмы масштабирования не будут существенно увеличивать время декодирования. Интересно, что масштабирование достаточно легко осуществляется в так называемых *фрактальных алгоритмах*. В них, даже при увеличении изображения в несколько раз, оно не распадается на квадраты, т. е. отсутствует эффект "зернистости". Если необходимо уменьшать изображение (что хоть и редко, но бывает нужно), то с такой задачей хорошо справляются алгоритмы, основанные на wavelet-преобразовании (см. описание JPEG 2000).

Небольшая стоимость аппаратной реализации. При разработке хотя бы приблизительно должна оцениваться и учитываться конечная стоимость. Если эта стоимость велика, то даже при использовании алгоритма в международных стандартах производители будут предлагать свои, более конкурентоспособные алгоритмы и решения. На практике это требование означает, что алгоритм должен реализовываться небольшим набором микросхем.

Упражнение. Покажите, что требования произвольного доступа, быстрого поиска, показа в обратном направлении, аудиовизуальной синхронизации и устойчивости к ошибкам противоречат условию высокой степени сжатия потока.

Описанные требования к алгоритму противоречивы. Очевидно, что высокая степень сжатия подразумевает архивацию каждого последующего кадра с использованием предыдущего. В то же время требования на аудиовизуальную синхронизацию и произвольный доступ к любому кадру за ограниченное время не дают возможности вытянуть все кадры в цепочку. И тем не менее можно попытаться прийти к некоторому компромиссу. Сбалансированная реализация, учитывающая систему противоречивых требований, может достигаться на практике за счет настроек компрессора при сжатии конкретного фильма.

Определение требований

Под процедурой определения требований, предъявляемых к алгоритму, понимается уяснение классов программного и аппаратного обеспечения, на которые он ориентирован и, соответственно, выработка требований к нему.

Носители информации, на которые ориентирован алгоритм:

- DVD-ROM – сравнительно низкая стоимость, очень высокая плотность записи информации делают его наиболее перспективным устройством для хранения оцифрованного видео.
- CD-ROM – низкая стоимость при высокой плотности записи информации делают его наиболее привлекательным устройством для хранения оцифрованного видео. К недостаткам относится сравнительно малый объем, однако диск обладает рекордно низким отношением стоимости диска к объему.
- Жесткий диск – наиболее быстрое и гибкое устройство, обладающее очень малым временем поиска, что необходимо для некоторых приложений. Однако он имеет высокое соотношение стоимости диска к объему.
- Перезаписываемый оптический диск – одно из наиболее перспективных устройств, способных сочетать в себе достоинства CD-ROM (низкая себестоимость хранения информации, большой объем, произвольный доступ) и жесткого диска (возможность перезаписи).
- Компьютерные сети (как глобальные, так и локальные). Характеризуются возможностью быстро получать практически неограниченные объемы информации. К недостаткам сетей, с которыми борются так называемые *технологии QoS* (Quality of Service – гарантированное качество сервиса), относятся возможные задержки пакетов и произвольное изменение пропускной способности канала.

Программное обеспечение, использующее видеокомпрессию, можно подразделить на две группы – *симметричное* и *асимметричное*.

Асимметричные приложения предъявляют серьезные требования к декодеру (как правило, по времени и памяти), но для них безразличны затраты ресурсов при кодировании. Примером являются различные мультимедиа-энциклопедии, путеводители, справочники, игры и просто фильмы. При такой постановке задачи появляется возможность применить сложные алгоритмы компрессии, позволяющие получить большую степень сжатия данных.

Симметричные приложения предъявляют одинаково жесткие требования на время, память и другие ресурсы как при кодировании, так и при декодировании. Примерами такого рода приложений могут служить видеопочта, видеотелефон, видеоконференции, редактирование и подготовка видеоматериалов.

Обзор стандартов

В 1988 г. в рамках Международной организации по стандартизации (ISO) начала работу группа MPEG (Moving Pictures Experts Group) – группа экспертов в области цифрового видео (ISO-IEC/JTC1/SC2/WG11/MPEG). Группа работала в направлениях, которые можно условно назвать MPEG-Video – сжатие видеосигнала в поток со скоростью до 1.5 Мбит/с, MPEG-Audio – сжатие звука до 64, 128 или 192 Кбит/с на канал и MPEG-System – синхронизация видео- и аудиопотоков [1]. Нас в основном будут интересовать достижения MPEG-Video, хотя очевидно, что многие решения в этом направлении принимались с учетом требований синхронизации.

Как *алгоритм* MPEG имеет несколько предшественников. Это, прежде всего, универсальный алгоритм JPEG. Его универсальность означает, что JPEG показывает неплохие результаты на широком классе изображений.

Если быть более точным, то *стандарт* MPEG, как и другие стандарты на сжатие, описывает лишь выходной битовый поток, *неявно* задавая *алгоритмы* кодирования и декодирования. При этом их реализация перекладывается на программистов-разработчиков. Такой подход открывает широкие горизонты для тех, кто желает оптимально реализовать алгоритм для конкретного вычислительного устройства (контроллера, ПК, распределенной вычислительной системы), операционной системы, видеокарты и т. п. [2]. При специализированных реализациях могут быть учтены весьма специфические требования на время работы, расход памяти и качество получаемых изображений. Алгоритмы сжатия видео весьма гибки, и зачастую для разных подходов к реализации можно получить существенную разницу по *качеству* видео при одной и той же *степени сжатия*. Более того, для одного и того же сжатого файла с помощью разных алгоритмов декодирования можно получить существенно различающиеся по визуальному качеству фильмы. Зачастую "простая" реализация стандарта дает дергающий видеоряд

с хорошо заметными блоками, в то время как программы известных производителей проигрывают этот же файл вполне плавно и без бросающейся в глаза блочности. Эти нюансы необходимо хорошо себе представлять, когда речь заходит о сравнении различных стандартов.

В сентябре 1990 г. был представлен предварительный стандарт кодирования MPEG-1. В январе 1992 г. работа над MPEG-1 была завершена и начата работа над MPEG-2, в задачу которого входило описание потока данных со скоростью от 3 до 10 Мбит/с [3]. Практически в то же время была начата работа над MPEG-3, который был предназначен для описания потоков 20-40 Мбит/с. Однако вскоре выяснилось, что алгоритмические решения для MPEG-2 и MPEG-3 принципиально близки и можно безболезненно расширить рамки MPEG-2 до потоков в 40 Мбит/с. В результате работа над MPEG-3 была прекращена. MPEG-2 был окончательно доработан к 1995 г.

В 1991 г. группой экспертов по видеотелефонам (EGVT) при Международном консультативном комитете по телефонии и телеграфии (CCITT) предложен стандарт видеотелефонов px64 Kbit/s [4,9]. Запись px64 означает, что алгоритм ориентирован на параллельную передачу оцифрованного видеоизображения по p-каналам с пропускной способностью 64 Кбита/с. Таким образом, захватывая несколько телефонных линий, можно получать изображение вполне приемлемого качества. Одним из главных ограничений при создании алгоритма являлось время задержки, которое должно было составлять не более 150 мс. Кроме того, уровень помех в телефонных каналах достаточно высок, и это, естественно, нашло отражение в алгоритме. Можно считать, что px64 Kbits – предшественник MPEG для потоков данных менее 1,5 Мбит/с и специфического класса видео.

В группе при CMTT (совместный комитет при CCITT и CCIR – International Consultative Committee on bRoadcasting) работы были направлены на передачу оцифрованного видео по выделенным каналам с высокой пропускной способностью и радиолиниям. Соответствующие стандарты H21 и H22 ориентированы на 34 и 45 Мбит/с, и сигнал передается с очень высоким качеством.

MPEG-4 изначально был задуман как стандарт для работы со сверхнизкими потоками. Однако в процессе довольно долгой подготовки стандарт претерпел совершенно революционные изменения, и сейчас собственно сжатие с низким потоком входит в него как одна составная часть, причем достаточно небольшая по размеру. Например, сам формат сегодня включает в себя такие вещи, как синтез речи, рендеринг изображений и описания параметров визуализации лица на стороне программы просмотра.

Разработка MPEG-7 была начата в 1996 г. Собственно к алгоритмам сжатия видео этот стандарт имеет еще меньшее отношение, чем MPEG-4,

поскольку его основная задача заключается в описании контента и управлении им. Описание MPEG-7 выходит за рамки это книги.

Параллельно все это время существовали форматы Motion-JPEG и недавно появившийся Motion-JPEG 2000, предназначенные в основном для удобства обработки сжатого видео. Рассмотрим основные стандарты и лежащие в их основе алгоритмы поподробнее.

Глава 1. Базовые технологии сжатия видеоданных

Описание алгоритма компрессии

Технология сжатия видео в MPEG распадается на две части: уменьшение избыточности видеоинформации во временном измерении, основанное на том, что соседние кадры, как правило, отличаются несильно, и сжатие отдельных изображений.

Для того чтобы удовлетворить противоречивым требованиям и увеличить гибкость алгоритма, рассматривается 4 типа кадров:

- I-кадры – кадры, сжатые независимо от других кадров (I-Intra pictures);
- P-кадры – сжатые с использованием ссылки на одно изображение (P-Predicted);
- B-кадры – сжатые с использованием ссылки на два изображения (B-Bidirection);
- DC-кадры – независимо сжатые с большой потерей качества (используются только при быстром поиске).

I-кадры обеспечивают возможность произвольного доступа к любому кадру, являясь своеобразными входными точками в поток данных для декодера. P-кадры используют при архивации ссылку на один I- или P-кадр, повышая тем самым степень сжатия фильма в целом. B-кадры, используя ссылки на два кадра, находящиеся впереди и позади, обеспечивают наивысшую степень сжатия. Сами в качестве ссылки использоваться не могут. Последовательность кадров в фильме может быть, например, такой: IBBPBBPBBPBBIBBPBB... Или, если мы не экономим на степени сжатия, такой, как на рис. 1.1

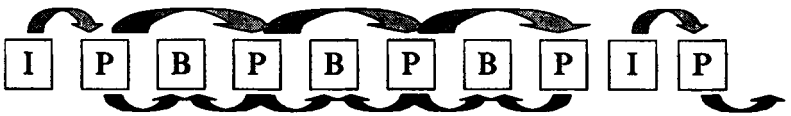


Рис. 1.1. I-кадры – независимо сжатые (I-Intrapictures); P-кадры – сжатые с использованием ссылки на одно изображение (P-Predicted); B-кадры – сжатые с использованием ссылки на два изображения (B-Bidirection)

Частота I-кадров выбирается в зависимости от требований на время произвольного доступа и надежности потока при передаче через канал с ошибками. Соотношение P- и B-кадров подбирается, исходя из требований к величине компрессии и ограничений декодера. Как правило, декодирование B-кадров требует больше вычислительных мощностей, однако позволяет повысить степень сжатия. Именно варьирование частоты кадров разных типов обеспечивает алгоритму необходимую гибкость и возможность расширения. Понятно, что для того, чтобы распаковать B-кадр, мы должны уже распаковать те кадры, на которые он ссылается. Поэтому для последовательности IBBPBBPBBPBBIBBPBB кадры в фильме будут записаны так: 0**312645..., где цифры – номера кадров, а звездочкам соответствуют либо B-кадры с номерами -1 и -2, если мы находимся в середине потока, либо пустые кадры (ничего), если мы в начале фильма. Подобный формат обладает достаточно большой гибкостью и способен удовлетворять самым различным наборам требований.

Одним из основных понятий при сжатии нескольких изображений является понятие *макроблока*. При сжатии кадр из цветового пространства RGB переводится в цветовое пространство YUV. Каждая из плоскостей сжимаемого изображения (Y, U, V) разделяется на блоки 8x8, с которыми работает ДКП. Причем плоскости U и V, соответствующие компоненте цветности, берутся с разрешением в 2 раза меньшим (по вертикали и горизонтали), чем исходное изображение. Таким образом, мы сразу получаем сжатие в 2 раза, пользуясь тем, что глаз человека хуже различает цвет отдельной точки изображения, чем ее яркость (подробнее об этих преобразованиях смотрите в описании алгоритма JPEG). Блоки 8x8 группируются в макроблоки. Макроблок – это группа из четырех соседних блоков в плоскости яркостной компоненты Y (матрица пикселов 16x16 элементов) и два соответствующих им по расположению блока из плоскостей цветности U и V. Таким образом, кадр разбивается на независимые единицы, несущие полную информацию о части изображения. При этом размер изображения должен быть кратен 16.

Отдельные макроблоки сжимаются независимо, т. е. в B-кадрах мы можем сжать макроблок конкретный как I-блок, P-блок со ссылкой на предыдущий кадр, P-блок со ссылкой на последующий кадр и, наконец, как B-блок.

Алгоритм сжатия отдельных кадров в MPEG похож на соответствующий алгоритм для статических изображений – JPEG. Если говорить коротко, то сам алгоритм сжатия представляет собой конвейер преобразований. Это дискретное косинусное преобразование исходной матрицы 8x8, квантование матрицы и вытягивание ее в вектор v11,v12,v21,v31,v22, ..., v88 (зигзаг-сканирование), сжатие вектора групповым кодированием и, наконец, сжатие по алгоритму Хаффмана.

Общая схема алгоритма

В целом весь конвейер преобразований можно представить так:

1. Подготовка макроблоков. Для каждого макроблока определяется, каким образом он будет сжат. В I-кадрах все макроблоки сжимаются независимо. В P-кадрах блок либо сжимается независимо, либо представляет собой разность с одном из макроблоков в предыдущем опорном кадре, на который ссылается P-кадр.
2. Перевод макроблока в цветовое пространство YUV. Получение нужного количества матриц 8x8.
3. Для P- и B-блоков производится вычисление разности с соответствующим макроблоком в опорном кадре.
4. ДКП
5. Квантование.
6. Зигзаг-сканирование.
7. Групповое кодирование.
8. Кодирование Хаффмана.

При декодировании весь конвейер повторяется для обратных преобразований, начиная с конца.

Использование векторов смещений блоков

Простейший способ учитывать подобие соседних кадров – это вычитать каждый блок сжимаемого кадра из соответствующего блока предыдущего. Однако более гибким является алгоритм поиска векторов, на которые сдвинулись блоки текущего кадра по отношению к предыдущему. Для каждого блока в изображении мы находим блок, близкий по некоторой метрике (например, по сумме квадратов разности пикселов), в предыдущем кадре в некоторой окрестности текущего положения блока. Если минимальное расстояние по выбранной метрике с блоками в предыдущем кадре больше выбранного порога, блок сжимается независимо (рис. 1.2.).

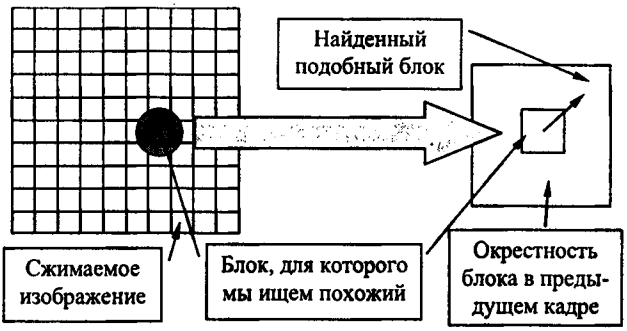


Рис. 1.2. Поиск векторов смещения блока

Таким образом, вместе с каждым блоком в поток теперь сохраняются координаты смещения максимально похожего блока в предыдущем I- или P-кадре, либо признак того, что данные сжаты независимо. Эти координаты задают вектор смещения блока (motion vector). В ситуациях, когда камера наезжает на объект или дает панораму, использование векторов смещений блоков позволяет значительно уменьшить амплитуду разности кадров и, как следствие, значительно поднять степень сжатия.

Если мы проанализируем реальные фильмы, то окажется, что часто блок сдвигается не на кратное число пикселов, а, например, на 10.4 пиксела (камера быстро движется вправо, план съемки сдвигается равномерно и проходит полный кадр размером 352x240 за 1.35 с). При этом оказывается, что для повышения степени сжатия выгодно строить 4 области поиска векторов смещений: исходную, сдвинутую на полпиксела по горизонтали, сдвинутую на полпиксела по вертикали и сдвинутую на полпиксела по горизонтали и по вертикали (по диагонали), которые строятся с помощью достаточно быстрых алгоритмов билинейной или кусочно-линейной аппроксимации. Этот прием также позволяет уменьшить разность между блоками и повысить степень сжатия при минимальной дополнительной информации, которую надо сохранять в файл (плюс 2 бита на каждый блок). Правда, строить аппроксимированные блоки придется и при декомпрессии, однако это сравнительно дешевая по времени операция, которая весьма незначительно увеличивает общее время декомпрессии.

Также надо понимать, что алгоритм поиска оптимальных векторов смещения заключается, вообще говоря, в переборе. Существуют различные методы уменьшения этого перебора, и настройки видеокодеков, регулирующие скорость сжатия, нередко варьируют именно параметры метода перебора.

Возможности по распараллеливанию

Даже беглый взгляд на этот обобщенный алгоритм позволяет заметить, что он сравнительно легко распараллеливается. Изображение 320x288 содержит 330 макроблоков, которые можно кодировать и декодировать независимо. Каждый макроблок, в свою очередь, содержит 6 блоков данных для ДКП. Распараллелить ДКП очень важно, так как, не считая поиска векторов смещения, это самая медленная операция. Заметим также, что остальные преобразования легко конвейеризуются. В результате мы получаем параллельно-конвейерную схему обработки потока видеоданных.

Достаточно заманчиво выглядит возможность распараллелить обработку различных кадров, но здесь мы сталкиваемся со сложностями. Как правило, компрессор строится таким образом, чтобы после сжатия изображение подвергалось обратным преобразованиям. Таким образом, мы получаем кадр с потерями и архивируем остальные кадры, отталкиваясь от него. Это позволяет не накапливать ошибки, получаемые еще при квантовании. Таким образом, если на экране между кадрами наблюдались большие изменения и качество изображения пришлось понизить, то при стабилизации изображения качество быстро повышается практически до качества исходного видеоряда. Неприятный эффект, порождаемый этим приемом, заключается в том, что появляется мерцание отдельных точек (или областей) изображения, значение цвета в которых округляется то в большую, то в меньшую сторону.

При распаковке наши возможности по параллельной обработке различных кадров достаточно ограничены, поскольку велика зависимость между кадрами в потоке (велик процент P- и B-кадров).

Другие пути повышения степени сжатия

Описанный выше алгоритм в целом крайне близок к большинству применяемых сейчас на практике алгоритмам сжатия видео. Однако новые (или хорошо забытые старые) идеи появляются ежегодно. Если для алгоритмов сжатия без потерь можно говорить о росте степени сжатия на 1 % в год (относительно предыдущего года) для достаточно большого тестового массива данных, то для алгоритмов сжатия видео речь обычно идет о 3–5 % прибавки степени сжатия для достаточно большого видеофрагмента при том же визуальном качестве.

Если, с одной стороны, повышается степень сжатия, то, с другой стороны, растет сложность программы и падает скорость работы как при компрессии так и при декомпрессии.

Перечислим основные пути повышения степени сжатия.

Изменение алгоритма сжатия I-кадров. Выше приведен алгоритм, основанный на ДКП. Сегодня все чаще используются алгоритмы, основанные на вэйвлетах (см. описание JPEG 2000).

Изменение алгоритма сжатия без потерь. Выше приведен алгоритм, использующий сжатие по алгоритму Хаффмана. Однако недавно закончился основной патент на арифметическое сжатие (дающее преимущество 2–15 %), и соответственно, все чаще используется именно оно.

Изменение алгоритма работы с векторами смещения блоков. Подбор векторов смещений блоков по наименьшему среднеквадратичному смещению не является оптимальным. Существуют алгоритмы, дающие лучший результат при некоторых дополнительных затратах времени при сжатии.

Применение обработки коэффициентов. Можно пытаться получить больше информации об изображении из сохраненных коэффициентов. Например, возможно быстрое сравнение коэффициентов после ДКП в соседних блоках и их усреднение по достаточно сложным алгоритмам. Этот прием заметно снижает количество артефактов, вносимых в изображение ДКП, при этом допуская реализацию, работающую в реальном времени.

Применение обработки получающихся кадров. Известная беда алгоритмов сжатия изображений – неизбежная "блочность" (хорошо заметные границы макроблоков). В принципе существуют алгоритмы, работающие с кадром совсем без применения блоков (даже без векторов смещения блоков), но такой подход пока не оправдывает себя ни по степени сжатия, ни по скорости работы декодера. Однако можно построить достаточно быстрые алгоритмы постобработки, которые достаточно аккуратно уберут видимые границы между блоками, не внося существенных помех в само изображение. Существуют также алгоритмы, устраняющие на лету эффект Гиббса (см. описание JPEG), и т. п. Таким образом, существенно улучшается визуальное качество изображения. Это также означает, что можно повысить степень сжатия при том же визуальном качестве.

Улучшение алгоритмов масштабирования изображений. Как правило, видео на компьютере просматривают во весь экран. При этом даже применение очень простого и быстрого кусочно-линейного масштабирования способно кардинально снизить скорость проигрывания ролика. То есть примитивная операция масштабирования изображения будет работать заметно дольше, чем сложный алгоритм декодера. Однако скорости современных компьютеров быстро растут, и те алгоритмы, что вызывали падение скорости до 7 кадров в секунду на Celeron-300, дают живое видео – больше 30 кадров – на P4-1200 (начинает использоваться расширенный набор команд и т. п.). Появляется возможность использовать более сложные и качественные алгоритмы масштабирования на весь экран, получая более высо-

кое качество, чем для использовавшейся ранее билинейной интерполяции (в большинстве видеокарт реализованной аппаратно).

Применение предварительной обработки видео. Если мы хотим получить достаточно высокую степень сжатия, то можно заранее предсказать, что в нашем изображении пострадают высокие частоты. Оно станет сглаженным, пропадут многие мелкие детали. При этом, как правило, появляются дополнительные артефакты в виде полосок, ореолов у резких границ, волн. Значительно повысить качество изображения после кодирования позволяет предобработка с удалением высоких частот. При этом существуют алгоритмы, обрабатывающие поток таким образом, что визуально качество изображения не изменяется, однако после декодера мы получаем существенно более качественное изображение.

Выше перечислены лишь отдельные направления работы. Фактически за 90-е гг. изменения и улучшения коснулись всех модулей алгоритма, построенного по классической схеме. Свою лепту в этот процесс вносят также производители микропроцессоров, и в особенности Intel. Процессоры, начиная с P4, специально предназначены для обработки потоковых данных. Особенности архитектуры процессора (в частности, небольшой по сравнению с процессорами AMD кеш первого уровня) дают значительное преимущество одним алгоритмам и делают неэффективными другие. Однако общее совершенствование алгоритмов сжатия видео идет очень быстро, и в ближайшее время можно ожидать только увеличения скорости появления новых разработок.

Глава 2. Стандарты сжатия видеоданных

Motion-JPEG

Motion-JPEG (или M-JPEG) является наиболее простым алгоритмом сжатия видеоданных. В нем каждый кадр сжимается независимо алгоритмом JPEG. Этот прием дает высокую скорость доступа к произвольным кадрам как в прямом, так и в обратном порядке следования. Соответственно легко реализуются плавные "перемотки" в обоих направлениях, аудиовизуальная синхронизация и, что самое главное, редактирование. Типичные операции JPEG сейчас поддерживаются на аппаратном уровне большинством видеокарт, и данный формат позволяет легко оперировать большими объемами данных при монтаже фильмов. Независимое сжатие отдельных кадров позволяет накладывать различные эффекты, не опасаясь, что взаимное влияние соседних кадров внесет дополнительные искажения в фильм.

Характеристики Motion-JPEG:

Поток, разрешение (сжатие): поток и разрешение произвольные, сжатие в 5–10 раз.

Плюсы: обеспечивает быстрый произвольный доступ. Легко редактировать поток. Низкая стоимость аппаратной реализации.

Минусы: сравнительно низкая степень сжатия.

MPEG-1

Алгоритм MPEG-1 в целом соответствует описанной выше общей схеме построения алгоритмов сжатия.

Характеристики MPEG-1:

Поток, разрешение: 1.5 Мбит/с, 352x240x30, 352x288x25.

Плюсы: сравнительно прост в аппаратной реализации, содержит преобразования, поддерживаемые на аппаратном уровне большим количеством видеокарт.

Минусы: невысокая степень сжатия. Малая гибкость формата.

H.261

Стандарт H.261 специфицирует кодирование и декодирование видеопотока для передачи по каналу px64 Кбит, где p=1...30. В качестве канала может выступать, например, несколько телефонных линий.

Входной формат изображения – разрешения CIF или QCIF в формате YUV (CCIR 601), частота кадров от 30 fps и ниже. Используется уменьшение разрешения в 2 раза для компонент цветности.

В выходной поток записываются два типа кадров: INTRA – сжатые независимо (соответствуют I-кадрам) и INTER – сжатые со ссылкой на предыдущий кадр (соответствуют P-кадрам). В передаваемом кадре не обязательно присутствуют все макроблоки изображения; если блок изменился незначительно передавать его обычно нет смысла. Сжатие в INTRA–кадрах осуществляется по схеме сжатия отдельного изображения. В INTER–кадрах производится аналогичное сжатие разности каждого передаваемого макроблока с "наиболее похожим" макроблоком из предыдущего кадра (компенсация движения). Для сглаживания артефактов ДКП предусмотрена возможность применения размытия внутри каждого блока 8x8 пикселов. Стандарт требует, чтобы INTRA-кадры встречались в потоке не реже чем через каждые 132 INTER-кадра (чтобы не накапливалась погрешность кодирования и была возможность восстановиться в случае ошибки в потоке).

Степень сжатия зависит в основном от метода нахождения "похожих" макроблоков в предыдущем кадре, алгоритма решения, передавать ли конкретный макроблок, выбора способа кодирования каждого макроблока (INTER/INTRA) и выбора коэффициентов квантования результатов ДКП. Ни один из перечисленных вопросы стандартом не регламентируются, оставляя свободу для построения собственных оптимальных алгоритмов.

Характеристики H.261:
Поток, разрешение: px64 Кбит, p=1...30, CIF или QCIF.
Плюсы: прост в аппаратной реализации.
Минусы: невысокая степень сжатия. Ограничения на формат.

H.263

Данный стандарт является расширением, дополнением и значительным усложнением H.261. Он содержит "базовый" стандарт кодирования, практически не отличающийся по алгоритмам сжатия от H.261, плюс множество опциональных его расширений.

Кратко перечислим наиболее важные отличия.

Использование арифметического кодирования вместо кодов Хаффмана. Дает возможность на 5–10 % повысить степень сжатия.

Возможность задания векторов смещения, указывающих за границы изображения. При этом граничные пикселы используются для предсказания пикселов вне изображения. Данный прием усложняет алгоритм декодирования, но позволяет значительно улучшить изображение при резкой смене плана сцены.

Возможность задания вектора смещения для каждого блока 8x8 в макроблоке, что в ряде случаев существенно увеличивает сжатие и снижает блочность изображения.

Появление B-кадров, которые позволяют увеличить степень сжатия, за счет усложнения и увеличения времени работы декодера.

Поддержка большого числа форматов входных видеоданных: sub-QCIF, QCIF, CIF, 4CIF, 16CIF и отдельно настраиваемых. Основное отличие от более универсальных форматов заключается в адаптации для нескольких фиксированных разрешений, что позволяет делать менее универсальные, но более быстрые процедуры обработки кадров. Построенный таким образом декодер работает несколько быстрее.

Компенсация движения с субпиксельной точностью. Возможность сдвинуть блок на полпиксела также увеличивает степень сжатия, но увеличивает время работы декодера.

Особый режим сжатия INTRA макроблоков со ссылкой на соседние макроблоки в обрабатываемом кадре, особый режим квантования и специальная таблица Хаффмана для улучшения сжатия I-кадров в ряде случаев.

Сглаживание границ блоков декодированного изображения для уменьшения эффекта "блочности". Зачастую при резком движении в кадре при сжатии алгоритм оказывается вынужден повысить степень квантования блоков после ДКП, чтобы уложиться в отведенный на передачу битовый поток. При этом в кадре возникают хорошо вам знакомые по JPEG блоки размером 8x8. Как показала практика, "сращивание" границ, когда крайние пикселы блоков сдвигают по яркости так, чтобы уменьшить разницу, позволяет зачастую заметно повысить визуальное качество фильма.

Изменение разрешения и деформирование базового кадра, использующегося в качестве базового при сжатии.

Различные режимы квантования и кодирования по Хаффману.

Характеристики H.263:

Поток, разрешение: 0.04–20 Мбит/с, sub-QCIF, QCIF, CIF, 4CIF, 16CIF и отдельно настраиваемые разрешения.

Плюсы: алгоритм H.263, так же как H.261, допускает быструю аппаратную реализацию, однако при этом позволяет добиться большей степени сжатия при том же качестве. Поддерживает сжатие звука.

Минусы: по количеству заложенных идей находится между MPEG-2 и MPEG-4.

MPEG-2

Как уже говорилось, MPEG-2 занимается сжатием оцифрованного видео при потоке данных от 3 до 10 Мбит/с. Многое в нем заимствовано из формата CCIR-601. CCIR-601 представляет собой стандарт цифрового видео с размером передаваемого изображения 720x486 при 60 полукадрах в секунду. Строки изображения передаются с чередованием, и два полукадра составляют кадр. Этот прием нередко применяют для уменьшения мерцания. Хроматические каналы (U и V в YUV) передаются размером 360x243 60 раз в секунду и также чередуются, уже между собой. Подобное деление называется 4:2:2. Перевод из CCIR-601 в MPEG-I прост: надо поделить в 2 раза яркостную компоненту по горизонтали, поделить поток в 2 раза во временном измерении (убрав чередование), добавить вторую хроматическую компоненту и выкинуть "лишние" строки, чтобы размер по вертикали делился на 16. Мы получим поток YUV кадров размером 352x240 с частотой 30 кадров в секунду. Здесь все просто.

Проблемы начинаются, когда появляется возможность увеличить поток данных и довести качество изображения до CCIR-601. Это не такая простая задача, как кажется. Проблема состоит в чередовании полукадров во входном формате. Тривиальное решение – работать с кадрами 720x486 при 30 кадрах в секунду, как с обычным видео. Этот путь приводит к неприятным эффектам при быстром движении объектов на экране. Между двумя исходными полукадрами 720x243 сдвиг становится заметным, а так как наш кадр формируется из исходных полукадров через строку, то при сжатии происходит размывание движущегося объекта. Виновно в этом эффекте ДКП, и как-то исправить ситуацию, не уменьшив степени сжатия видео или не потеряв в визуальном качестве, нельзя. Достаточно распространенным является применение "деинтерлейсинга" (от английского deinterlacing – удаление чередования строк). Эта операция позволяет удалить чередование, смещая четные строки в одном направлении, а нечетные в другом, пропорционально относительному движению объекта в данной области экрана. В результате мы получаем визуально более качественное изображение, но несколько более длительную предобработку перед сжатием.

Другим решением является архивация четных и нечетных кадров в потоке CCIR-601 независимо. При этом мы, конечно, избавимся от артефактов, возникающих при быстром движении объектов, но существенно уменьшим степень сжатия, так как не будем использовать важнейшей вещи – избыточности между соседними кадрами, которая очень велика.

Характеристики MPEG-2:

Поток, разрешение: 3–15 Мбит/с, универсальный.

Плюсы: поддержка достаточно серьезных звуковых стандартов Dolby Digital 5.1, DTS, высокая универсальность, сравнительная простота аппаратной реализации.

Минусы: недостаточная на сегодня степень сжатия, недостаточная гибкость.

MPEG-4

MPEG-4 кардинально отличается от принимаемых ранее стандартов. Рассмотрим наиболее интересные и полезные нововведения.

Расчет трехмерных сцен и работа с синтетическими объектами. В состав декодера MPEG-4 как составная часть входит блок визуализации трехмерных объектов (Animation Framework eXtension – AFX – то, что в просторечии называют данными для трехмерного движка). Те, кто кодировал видео, знают, сколько проблем доставляют титры и вообще любые на-

кладываемые поверх фильма объекты (логотипы, заставки и т. п.). Если хорошо выглядит основной план – будут подпорчены накладываемые объекты, если хорошо смотрятся они – будет низкой общая степень сжатия. В MPEG-4 предлагается решить проблему кардинально. Накладываемые объекты рассчитываются отдельно и накладываются потом. Кроме того, можно использовать видеопоток даже как текстуру, накладываемую на поверхности рассчитываемых объектов. Такая гибкая работа с трехмерными объектами позволяет существенно поднять степень сжатия при заметно лучшем качестве изображения. Более того, никто не мешает делать видеоролики вообще без живого видео, а состоящие только из рассчитанных (синтетических) объектов. Размер их описания будет в разы меньше, чем размер аналогичных фильмов, сжатых просто как поток кадров. Кстати, отдельно в стандарте предусмотрена работа со "спрайтами" – статическими изображениями, накладываемыми на кадр. При этом размер спрайта может быть как совсем маленький (логотип канала в уголке экрана), так и превышать размер кадра и "прокручиваться" (т. е. в качестве спрайта может быть задан фон, а небольшие видеообъекты, например голова диктора, будут на него накладывать). Это дает значительную гибкость при создании MPEG-4-фильмов и позволяет заметно уменьшить объем кодируемой информации.

Объектно-ориентированная работа с потоком данных. Теперь работа с потоком данных становится объектно-ориентированной. При этом данные могут быть живым видео, звуковыми данными, синтетическими объектами и т. д. Из них создаются сцены, этими сценами можно управлять. Для простых смертных при этом мало что изменится, однако для программистов объектная среда означает кардинальное упрощение работы с возникающими сложными структурами.

Помещение в поток двоичного кода "C++ подобного" языка BIFS. С помощью BIFS в поток добавляются описания объектов, классов объектов и сцен. Также на нем можно менять координаты, размеры, свойства, поведение и реакцию объектов на действия пользователя. В свое время Flash был назван революцией 2D графики в Интернете. Аналогичный прорыв в области видео совершает MPEG-4.

Активная зрительская позиция. Как было замечено выше, BIFS позволяет задавать реакцию объектов сцены на действия пользователя. Потенциально возможно удаление, добавление или перемещение объектов, ввод команд с клавиатуры. Событийная модель заимствована из развивавшегося уже долгое время языка моделирования виртуальной реальности VRML. Для тех, кто играл в написанные на VRML игры, очевидно, что в MPEG-4 будет совершенно реально создавать "квест"-подобные (и не только) игры. Широчайший простор открывается для создания обучающих и развлека-

тельных программ. Представляете, скачиваете из Интернета один файл, который сразу в себе содержит все, что необходимо для небольшого курса лекций, причем вы можете прослушать его, видя говорящую голову преподавателя, или, отключив его, увеличить фрагменты (спрайты) с материалами. А в конце – пройти короткий тест на понимание предмета. Кстати, в стандарте предусмотрена обработка команд на стороне сервера, т. е. программа-просмотрщик может отослать данные на сервер и получить оттуда оценку. Отличие от предыдущих стандартов революционное.

Синтезатор лиц и фигур. В стандарт заложен интерфейс к модулю синтеза лиц и фигур. Например, в файле сохраняются ключевые данные о профиле лица и текстуры лица, а при записи фильма сохраняются только коэффициенты изменения формы. Для передач типа новостей этот прием позволяет в десятки раз сократить размер файла при замечательном качестве.

Синтезатор звуков и речи. Помимо синтеза лиц в стандарт MPEG-4 также заложены алгоритмы синтеза звуков, и даже речи(!).

Улучшенные алгоритмы сжатия видео. В стандарте предусмотрены блоки, отвечающие за потоки 4.8–65 Кбит/с с прогрессивной разверткой и большие потоки с поддержкой чересстрочной развертки. Для передачи по ненадежным каналам возможно использование помехоустойчивых методов кодирования (за счет незначительного увеличения объема передаваемых данных резко снижается вероятность искажения изображения). При передаче видео с одновременным просмотром заложена возможность огрубить изображение, если декодер из-за ограничений канала связи не успевает получить всю информацию. Всего в стандарт заложено 3 уровня детализации. Эта возможность позволит легко адаптировать алгоритм для трансляций видео по сети.

Поддержка профилей на уровне стандарта. Понятно, что реализация всех возможностей стандарта превращает декодер в весьма сложную и большую конструкцию. При этом далеко не для всех приложений необходимы какие-то сложные специфические функции (например, синтез речи). Создатели стандарта поступили просто: они оговорили наборы профилей, каждый из которых включает в себя набор обязательных функций. Если в фильме записано, что ему для проигрывания необходим такой-то профиль и декодер этот профиль поддерживает, то стандарт гарантирует, что фильм будет проигран правильно.

Выше кратко перечислены некоторые отличия MPEG-4 от предыдущих стандартов. Надо отметить, что на момент создания стандарта острой потребности в описанных выше вещах еще не было. Иначе говоря, мы имеем дело с хорошо продуманной работой по формированию стандарта, которая была закончена к тому времени, когда в нем возникла первая необходимость.

Создателями MPEG-4 учтен опыт предшественников (в частности, VRML), когда слишком раннее появление стандарта и отсутствие в нем механизма профилей серьезно подорвало его массовое применение. Будем надеяться, что массовому применению MPEG-4 такие проблемы не грозят.

Характеристики MPEG-4:

Поток, разрешение: 0,0048–20 Мбит/с, поддерживаются все основные стандарты видеопотоков.

Плюсы: поддержка достаточно прогрессивных звуковых стандартов, высокая степень универсальности, поддержка новых технологий (различные виды синтеза звука и изображения).

Минусы: высокая сложность реализации.

Сравнение стандартов

Название	Годы	Разрешения и поток	Аудио	Применение
MPEG-1	1992	352x240x30, 352x288x25, 1.5 Мбит/с	MPEG-1 Layer II	VideoCD первого поколения
H.261	1993	352x288x30, 176x144x30, 0,04–2 Мбит/с (px64 Кбит/с, где p от 1 до 30)	–	Аппаратно реализованные кодеки, видеоконференции
MPEG-2	1995	Универсальный, 3–15 Мбит/с	MPEG-1 Layer II, Dolby Digital 5.1, DTS	DVD
H.263	1998	sub-QCIF, QCIF, CIF, 4CIF, 16CIF и настраиваемые особо	Поддерживается	Аппаратно реализованные кодеки, видеотелефоны, видеоконференции
MPEG-3 не принят	1993-1995	Телевидение высокой четкости, 20–40 Мбит/с		HDTV
MPEG-4	1999	Универсальный, 0,0048–20 Мбит/с	MPEG-1 Layer II, MPEG-1 Layer III, Dolby Digital 5.1, DTS	VideoCD второго поколения

Название	За счет чего достигается сжатие	Дополнительные возможности
MPEG-1	ICT, DCT	
H.261	ICT, DCT, MC	Передача потока данных по p-каналам с пропускной способностью 64 Кбит (телеконференции по нескольким телефонным линиям)
MPEG-2	ICT, DCT, MC	
MPEG-4	ICT, Wavelet, MC, спрайты, объекты с прозрачным фоном, 3d-рендеринг	Встроенный язык описания BIFS, синтезатор речи, функции анимации лиц, 3D-рендеринг и т. д.

Вопросы для самоконтроля

1. Какие параметры надо определить, прежде чем сравнивать два алгоритма сжатия видео?
2. Приведите примеры ситуаций, когда архитектура компьютера дает преимущество тому или иному алгоритму сжатия видео.
3. Какими свойствами видеопотока мы можем пользоваться, создавая алгоритм сжатия? Приведите примеры.
4. Что такое аудиовизуальная синхронизация? Почему выполнение ее требований значительно снижает степень сжатия?
5. Назовите основные требования к алгоритмам сжатия видео.
6. Что такое I-кадры, P-кадры?
7. Приведите примеры программно-аппаратной реализации алгоритмов сжатия видео (повседневные и достаточно новые).
8. Приведите примеры областей использования видео, НЕкритичных к требованию "устойчивости к ошибкам".

Литература

1. *Le Gall D. J.* The MPEG Video Compression Algorithm // Signal Processing: Image Communication. 1992. Vol. 4, № 2. P. 129– 140.
2. *Wallach D. S., Kunapalli S., Cohen M.F.* Accelerated MPEG compression of Dynamic poligonal scenes // ACM. Jun 1994.
3. FAQ-статья по MPEG. Ver 2.0–3.0. May 1993. PHADE SOFTWARE. Berlin.
4. *Liou Ming.* Overview of the px64 kbit/s video coding standart // Communication of ACM. April 1991. Vol. 34. № 4.
5. *Anderson M.* VCR quality video at 1,5 Mbit/s // National Communication Forum. Chicago, Oct. 1990.

6. *Chen C. T. and Le Gall D.A. A* Kth order adaptive transform coding algorithm for high-fidelity reconstruction of still images // Proceedings of the SPIE. San Diego, Aug. 1989.
7. Coding of moving pictures and associated audio // Committee Draft of Standart ISO11172: ISO/MPEG 90/176 Dec. 1990.
8. JPEG digital compression and coding of continuous-tone still images. ISO 10918. 1991.
9. Video codec for audio visual services at px64 Kbit/s. CCITT Recommendation H.261. 1990.
10. MPEG proposal package description. Document ISO/WG8/MPEG 89/128. July 1989.
11. International Telecommunication Union. Video Coding for Low Bitrate Communication, ITU-T Recommendation H.263. 1996.
12. RTP Payload Format for H.263 Video Streams, Intel Corp. Sept. 1997. http://www.faqs.org/rfcs/rfc2190.html.
13. *Mitchell J.L., Pennebaker W.B., Fogg C.E. and LeGall D.J.* "MPEG Video Compression Standard" // Digital Multimedia Standards Series. New York, NY: Chapman & Hall, 1997.
14. *Haskell B. G., Puri A. and Netravali A. N.* Digital Video: An Introduction to MPEG-2 // ISBN: 0-412-08411-2. New York, NY: Chapman & Hall, 1997.
15. Image and Video Compression Standards Algorithms and Architectures, Second Edition by Vasudev Bhaskaran. Boston Hardbound: Kluwer Academic Publishers, 472 p.
16. *Sikora T.*, MPEG-4 Very Low Bit Rate Video // Proc. IEEE ISCAS Conference, Hongkong, June 1997. http://wwwam.hhi.de/mpeg-video/papers /sikora/vlbv.htm.
17. *Rao K. R. and Yip P.* Discrete Cosine Transform – Algorithms, Advantages, Applications // London: Academic Press, 1990.
18. ISO/IEC JTC1/SC29/WG11 N4030 Overview of the MPEG-4 Standard // Vol. 18 – Singapore Version, March 2001. http://mpeg.telecomitalialab.com /standards/mpeg-4/mpeg-4.htm.
19. ISO/IEC JTC1/SC29/WG11N MPEG-4 Video Frequently Asked Questions // March 2000 http://mpeg.telecomitalialab.com/faq/mp4-vid/mp4-vid.htm.
20. ITU. Chairman of Study Group 11. The new approaches to quality assessment and measurement in digital broadcasting. Doc. 10–11Q/9, 6. October 1998.
21. *Kuhn Peter,* Algorithms, Complexity Analysis and Vlsi Architectures for Mpeg-4 Motion Estimation // Boston Hardbound: Kluwer Academic Publishers, June 1999. ISBN 0792385160, 248 p.
22. Цифровое телевидение // Под ред. М. И. Кривошеева. М.: Связь, 1980.

23. Кривошеев М. И. Основы телевизионных измерений. 3-е изд., доп. и перераб. М.: Радио и связь, 1989.

Ссылки на программы и реализации алгоритмов

24. OpenDivX For Windows, Linux и т. д. – качественный открытый кодек MPEG-4) http://www.projectmayo.com/projects/
25. MPEG4IP: Open Source MPEG4 encoder & decoder – другой вариант MPEG-4-декодера, доступный для изучения в исходных текстах http://www.mpeg4ip.net/
26. Open-Source On3 кодек – открытая часть проекта ON2, коммерческий кодек, тексты которого открыты для модификаций всеми желающими http://www.vp3.com/
27. Универсальный конвертер VirtualDub, Open Source. http://www.virtualdub.org/
28. Public Source Code Release of Matching Pursuit Video Codec http://www-video.eecs.berkeley.edu/download/mp/
29. Исходные тексты MPEG, JPEG и H.261 (простые варианты) http://www-set.gmd.de/EDS/SYDIS/designenv/applications/analysis/

ПРИЛОЖЕНИЯ

П-1. Контекстный компрессор Dummy

```
/* Контекстный компрессор Dummy, автор М.Смирнов.
 * Исходный текст этой программы также можно найти на сайт
 * http://compression.graphicon.ru/
 *
 * Применение:
 * e infile outfile  - закодировать infile в outfile
 * d infile outfile  - декодировать infile в outfile
 */

#include <stdio.h>

/* Класс для организации ввода/вывода данных */

class DFile {
   FILE *f;
public:
   int ReadSymbol (void) {
      return getc(f);
   };
   int WriteSymbol (int c) {
      return putc(c, f);
   };
   FILE* GetFile (void) {
      return f;
   }
   void SetFile (FILE *file) {
      f = file;
   }
} DataFile, CompressedFile;

/* Реализация range-кодера, автор Е.Шелвин
 * http://www.pilabs.org.ua/sh/aridemo6.zip
 */

typedef unsigned int  uint;

#define  DO(n)     for (int _=0; _<n; _++)
#define  TOP     (1<<24)

class RangeCoder
{
 uint code, range, FFNum, Cache;
 __int64 low; // Microsoft C/C++ 64-bit integer type
```

```
  FILE *f;
public:
  void StartEncode( FILE *out )
  {
    low=FFNum=Cache=0;  range=(uint)-1;
    f = out;
  }
  void StartDecode( FILE *in )
  {
    code=0;
    range=(uint)-1;
    f = in;
    DO(5) code=(code<<8) | getc(f);
  }
  void FinishEncode( void )
  {
    low+=1;
    DO(5) ShiftLow();
  }
  void FinishDecode( void ) {}
  void encode(uint cumFreq, uint freq, uint totFreq)
  {
    low += cumFreq * (range/= totFreq);
    range*= freq;
    while( range<TOP ) ShiftLow(), range<<=8;
  }
  inline void ShiftLow( void )
  {
    if ( (low>>24)!=0xFF ) {
       putc ( Cache + (low>>32), f );
       int c = 0xFF+(low>>32);
       while( FFNum ) putc(c, f), FFNum--;
       Cache = uint(low)>>24;
    } else FFNum++;
    low = uint(low)<<8;
  }
  uint get_freq (uint totFreq) {
    return code / (range/= totFreq);
  }
  void decode_update (uint cumFreq, uint freq, uint totFreq)
  {
```

```
   code -= cumFreq*range;
   range *= freq;
   while( range<TOP ) code=(code<<8)|getc(f), range<<=8;
 }
} AC;

/* конец реализации range-кодера */

/* Структуры данных, глобальные переменные, константы */

struct ContextModel{
   int  esc,
      TotFr;
   int  count[256];
};

ContextModel cm[257],
             *stack[2];

int  context [1],
     SP;

const int MAX_TotFr = 0x3fff;

/* Собственно реализация компрессора */

void init_model (void){
   for ( int j = 0; j < 256; j++ )
      cm[256].count[j] = 1
   ;
   cm[256].TotFr = 256;
   cm[256].esc = 1;
   context [0] = SP = 0;
}

int encode_sym (ContextModel *CM, int c){
   stack [SP++] = CM;
   if (CM->count[c]){
      int CumFreqUnder = 0;
      for (int i = 0; i < c; i++)
         CumFreqUnder += CM->count[i];
      AC.encode (CumFreqUnder, CM->count[c],
                 CM->TotFr + CM->esc);
      return 1;
   }else{
      if (CM->esc){
         AC.encode (CM->TotFr, CM->esc, CM->TotFr +
                    CM->esc);
      }
```

```
      return 0;
   }
}

int decode_sym (ContextModel *CM, int *c){
   stack [SP++] = CM;
   if (!CM->esc) return 0;

   int cum_freq = AC.get_freq (CM->TotFr + CM->esc);
   if (cum_freq < CM->TotFr){
      int CumFreqUnder = 0;
      int i = 0;
      for (;;){
         if ( (CumFreqUnder + CM->count[i]) <= cum_freq)
            CumFreqUnder += CM->count[i];
         else break;
         i++;
      }
      AC.decode_update (CumFreqUnder, CM->count[i],
                        CM->TotFr + CM->esc);
      *c = i;
      return 1;
   }else{
      AC.decode_update (CM->TotFr, CM->esc,
                        CM->TotFr + CM->esc);
      return 0;
   }
}

void rescale (ContextModel *CM){
   CM->TotFr = 0;
   for (int i = 0; i < 256; i++){
      CM->count[i] -= CM->count[i] >> 1;
      CM->TotFr += CM->count[i];
   }
}

void update_model (int c){
   while (SP) {
      SP--;
      if (stack[SP]->TotFr >= MAX_TotFr)
         rescale (stack[SP]);
      stack[SP]->TotFr += 1;
      if (!stack[SP]->count[c])
         stack[SP]->esc += 1;
      stack[SP]->count[c] += 1;
   }
}

void encode (void){
```

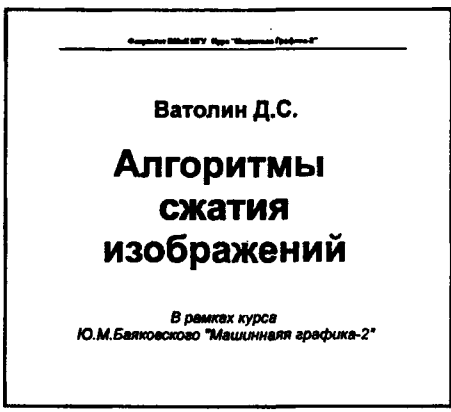
```
   int  c,
        success;
   init_model ();
   AC.StartEncode (CompressedFile.GetFile());
   while (( c = DataFile.ReadSymbol() ) != EOF) {
      success = encode_sym (&cm[context[0]], c);
      if (!success)
         encode_sym (&cm[256], c);
      update_model (c);
      context [0] = c;
   }
   AC.encode (cm[context[0]].TotFr, cm[context[0]].esc,
      cm[context[0]].TotFr + cm[context[0]].esc);
   AC.encode (cm[256].TotFr, cm[256].esc,
      cm[256].TotFr + cm[256].esc);
   AC.FinishEncode();
}
void decode (void){
   int  c,
        success;
   init_model ();
   AC.StartDecode (CompressedFile.GetFile());
   for (;;){
      success = decode_sym (&cm[context[0]], &c);
      if (!success){
         success = decode_sym (&cm[256], &c);
         if (!success) break;
      }
      update_model (c);
      context [0] = c;
      DataFile.WriteSymbol (c);
   }
}

void main (int argc, char* argv[]){
   FILE *inf, *outf;
   if (argv[1][0] == 'e'){
      inf = fopen (argv[2], "rb");
      outf = fopen (argv[3], "wb");
      DataFile.SetFile (inf);
      CompressedFile.SetFile (outf);
      encode ();
      fclose (inf);
      fclose (outf);
   }else if (argv[1][0] == 'd'){
      inf = fopen (argv[2], "rb");
      outf = fopen (argv[3], "wb");
```

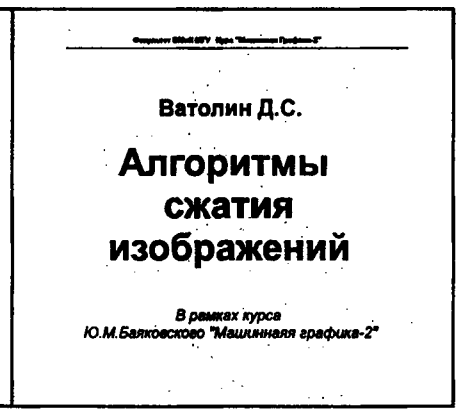
```
        CompressedFile.SetFile (inf);
        DataFile.SetFile (outf);
        decode ();
        fclose (inf);
        fclose (outf);
    }
}
```

П-2. Сжатие цветного изображения

Сжатие двуцветного изображения



Изображение 1000x1000x2 цвета 125.000 байт



Изображение 1000x1000x2 цвета 125.000 байт с внесенными в него помехами

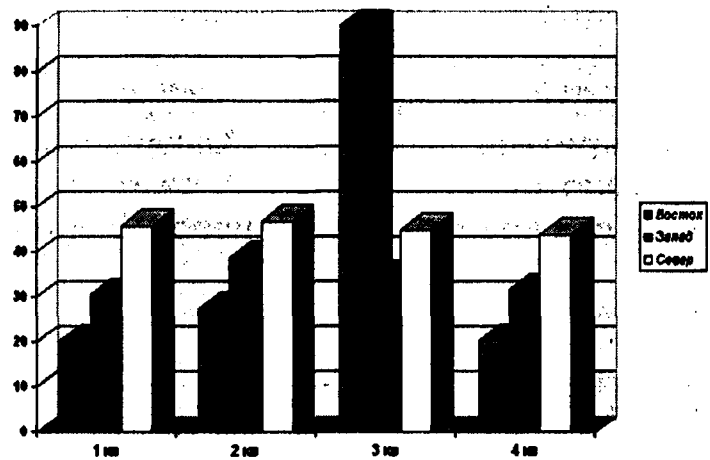
Ниже приведена степень компрессии изображений в зависимости от применяемого алгоритма.

	Алгоритм RLE	Алгоритм LZW	CCITT Group 3	CCITT Group 4
Без помех	10,6 (TIFF-CCITT RLE) 6,6 (TIFF-PackBits) 4,9 (PCX) 2,99 (BMP) 2,9 (TGA)	12 (TIFF-LZW) 10,1 (GIF)	9,5 (TIFF)	3[illegible] (TIFF)
С помехами	5 (TIFF-CCITT RLE) 2,49 (TIFF-PackBits) 2,26 (PCX) 1,7 (TGA) 1,69 (BMP)	5,4 (TIFF-LZW) 5,1 (GIF)	4,7 (TIFF)	5,12 (TIFF)

Выводы, которые можно сделать, анализируя данную таблицу:

- Лучшие результаты показал алгоритм, оптимизированный для этого класса изображений CCITT Group 4, и модификация универсального алгоритма LZW.
- Даже в рамках одного алгоритма велик разброс значений алгоритма компрессии. Заметим, что реализации RLE и LZW для TIFF показали заметно лучшие результаты, чем в других форматах. Более того, **во всех колонках все варианты** алгоритмов сжатия, реализованные в формате TIFF, лидируют.

Сжатие 16-цветного изображения



Изображение 619x405x16 цвета 125 350 байт

В таблице приведена степень компрессии изображений в зависимости от применяемого алгоритма.

	Алгоритм RLE	Алгоритм LZW
Первое изображение	5,55 (TIFF-PackBits) 5,27 (BMP) 4,8 (TGA) 2,37 (PCX)	11 (TIFF-LZW)

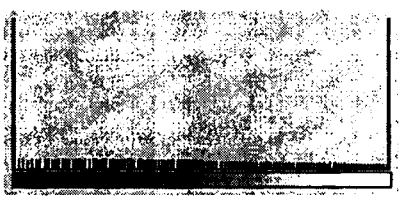
Вывод, который можно сделать, анализируя данную таблицу: несмотря на то что данное изображение относится к классу изображений, на которые ориентирован алгоритм RLE (отвечает критериям "хорошего" изображения для алгоритма RLE), заметно лучшие результаты для него дает более универсальный алгоритм LZW.

Сжатие изображения в градациях серого

Изображение 600x700x256 градаций серого сразу после сканирования. 420.000 байт

То же изображение с выровненной гистограммой плотности серого

На гистограмме хорошо видны равномерные большие значения в области темных и "почти белых" тонов



После выравнивания пики есть только в значениях 0 и 255. В изображении присутствуют далеко не все значения яркости

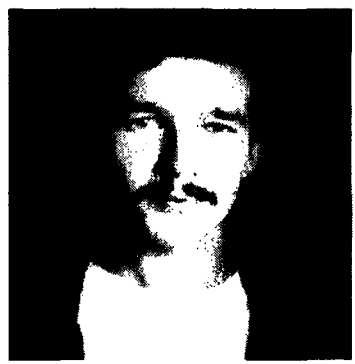
	Алгоритм RLE	Алгоритм LZW	Алгоритм JPEG
Оригинал	0,99 (TIFF-PackBits) 0,98 (TGA) 0,88 (BMP) 0,74 (PCX)	0,976 (TIFF-LZW) 0,972 (GIF)	7,8 (JPEG q=10) 3,7 (JPEG q=30) 2,14 (JPEG q=100)
После обработки	2,86 (TIFF-PackBits) 2,8 (TGA) 0,89 (BMP) 0,765 (PCX)	3,02 (TIFF-LZW) 0,975 (GIF)[1]	6,9 (JPEG q=10) 3,7 (JPEG q=30) 2,4 (JPEG q=100)

[1] *Для формата GIF в этом случае можно получить изображение меньшего размера, используя дополнительные параметры.*

Выводы, которые можно сделать анализируя таблицу:

- Лучшие результаты показал алгоритм сжатия с потерей информации. Для оригинального изображения только JPEG смог уменьшить файл. Заметим, что увеличение контрастности уменьшило степень компрессии при максимальном сжатии – врожденное свойство JPEG.
- Реализации RLE и LZW для TIFF опять показали заметно лучшие результаты, чем в других форматах. Степень сжатия для них после обработки изображения возросла в 3 раза(!). В то время как GIF, PCX и BMP и в этом случае увеличили размер файла.

Сжатие полноцветного изображения



Изображение 320x320xRGB – 307 200 байт

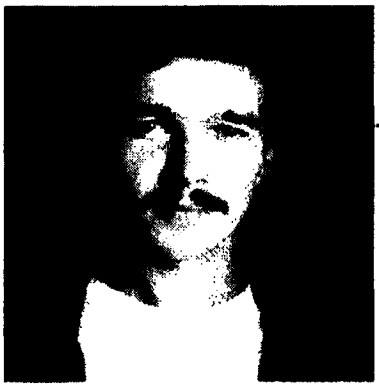
Ниже приведена степень компрессии изображений в зависимости от применяемого алгоритма.

	Алгоритм RLE	Алгоритм LZW	Алгоритм JPEG
Первое изображение	1,046 (TGA) 1,037 (TIFF-PackBits)	1,12 (TIFF-LZW) 4,65 (GIF) *С потерями!* Изображение в 256 цветах	47,2 (JPEG q=10) 23,98 (JPEG q=30) 11,5 (JPEG q=100)

Выводы, которые можно сделать, анализируя таблицу:

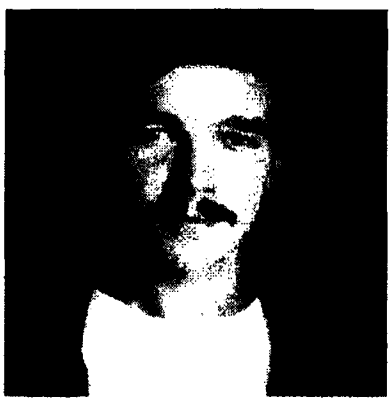
- Алгоритм JPEG при визуально намного меньших потерях (q=100) сжал изображение в 2 раза сильнее, чем LZW с использованием перевода в изображение с палитрой.
- Алгоритм LZW, примененный к 24-битовому изображению практически на дает сжатия.
- Минимальное сжатие, полученное алгоритмом RLE, можно объяснить тем, что изображение в нижней части имеет сравнительно большую область однородного белого цвета (полученную после обработки изображения).

Сжатие полноцветного изобр[illegible]



320x320xRGB – 307 200 байт

Сжати[illegible]



Сжатие в 100 раз (3.04Kb) фрактальным алгоритмом

Сжати[illegible]

На данном примере хорошо видно, что при высоких [illegible] сии алгоритм JPEG оказывается полностью неконкурентос[illegible] ство изображения для фрактального алгоритма визуально н[illegible] чем у wavelet-алгоритма, однако для него не используется по[illegible] изображения (достаточно "разумное" сглаживание), из-за которо[illegible] вого алгоритма размываются мелкие детали изображения.

УКАЗАТЕЛЬ ТЕРМИНОВ

ОГЛАВЛЕНИЕ