

# программирования

д. сэломон

Сжатие данных,  
изображений  
и звука



ТЕХНОСФЕРА



# М И Р

## программирования

цифровая обработка сигналов

д. сэломон

## Сжатие данных, изображений и звука

Перевод с английского  
В. В. Чепыжкова

Рекомендовано ИППИ РАН в  
качестве учебного пособия для  
студентов высших учебных заведений,  
обучающихся по направлению  
подготовки "Прикладная математика"

ТЕХНОСФЕРА  
Москва  
2004

Д.Сэломон

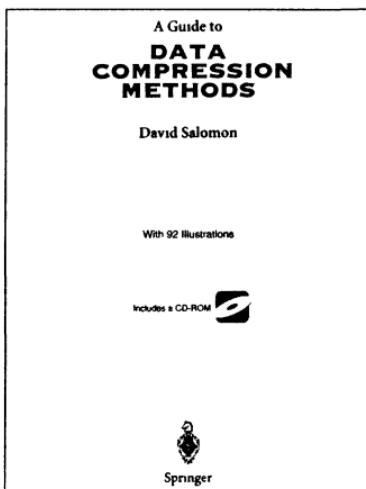
Сжатие данных, изображений и звука

Москва:

Техносфера, 2004. – 368с. ISBN 5-94836-027-X

В учебном пособии изложены как общие идеи и основы теории сжатия информации, так и практические методы с подробным описанием конкретных алгоритмов компрессии различных типов цифровых данных. Общие концепции описываются вполне строго и основываются на четких научных принципах. Все алгоритмы проиллюстрированы подробными примерами, снабженны таблицами, диаграммами и рисунками. В книге рассматриваются различные методы сжатия самой разнообразной информации: текстов, графических изображений, звука, анимации, оцифрованных аудио- и видео данных. В руководстве приводятся многие популярные стандарты и протоколы сжатия, такие как JPEG, MPEG, которые часто сопровождаются готовыми к употреблению текстами программ для системы MATLAB.

Книга рассчитана на многочисленную аудиторию программистов и Web-дизайнеров, разработчиков телекоммуникационных и информационных систем



ORIGINAL ENGLISH LANGUAGE  
EDITION PUBLISHED BY  
Springer-Verlag  
© 2002 All Rights Reserved.  
© 2004, ЗАО «РИЦ «Техносфера»  
перевод на русский язык,  
оригинал-макет, оформление.

ISBN 5-94836-027-X

ISBN 0-387-95260-8 (англ.)

# Содержание

Предисловие переводчика .....	6
Предисловие автора .....	10
Введение .....	15
<b>Глава 1.</b>	
<b>Статистические методы .....</b>	<b>25</b>
1.1. Энтропия .....	25
1.2. Коды переменной длины .....	26
1.3. Декодирование .....	29
1.4. Кодирование Хаффмана .....	30
1.4.1. Декодирование Хаффмана .....	38
1.4.2. Средняя длина кода .....	39
1.5. Адаптивные коды Хаффмана .....	40
1.5.1. Несжатые коды .....	43
1.5.2. Модификация дерева .....	43
1.5.3. Переполнение счетчика .....	46
1.5.4. Кодовое переполнение .....	47
1.5.5. Вариант алгоритма .....	51
1.6. Факсимильное сжатие .....	52
1.6.1. Одномерное кодирование .....	53
1.6.2. Двумерное кодирование .....	58
1.7. Арифметическое кодирование .....	62
1.7.1. Детали реализации метода .....	69
1.7.2. Потеря значащих цифр .....	73
1.7.3. Заключительные замечания .....	74
1.8. Адаптивное арифметическое кодирование .....	76
<b>Глава 2.</b>	
<b>Словарные методы .....</b>	<b>81</b>
2.1. LZ77 (скользящее окно) .....	84
2.1.1. Циклическая очередь .....	87
2.2. LZSS .....	88
2.2.1. Недостатки .....	92
2.3. LZ78 .....	93
2.4. LZW .....	97
2.4.1. Декодирование LZW .....	101



2.4.2. Структура словаря LZW .....	104
2.4.3. LZW в практических приложениях .....	110
2.5. Заключение .....	110
<b>Глава 3.</b>	
<b>Сжатие изображений .....</b>	<b>111</b>
3.1. Введение .....	112
3.2. Типы изображений .....	118
3.3. Подходы к сжатию изображений .....	120
3.3.1. Коды Грэя .....	125
3.3.2. Метрики ошибок .....	134
3.4. Интуитивные методы .....	137
3.4.1. Подвыборка .....	137
3.4.2. Квантование .....	138
3.5. Преобразование изображений .....	139
3.5.1. Ортогональные преобразования .....	145
3.5.2. Матричные преобразования .....	148
3.5.3. Дискретное косинус-преобразование .....	150
3.5.4. Пример .....	162
3.5.5. Дискретное синус-преобразование .....	162
3.5.6. Преобразование Уолша–Адамара .....	167
3.5.7. Преобразование Хаара .....	170
3.5.8. Преобразование Кархунена–Лоэвэ .....	172
3.6. Прогрессирующее сжатие .....	174
3.7. JPEG .....	182
3.7.1. Светимость .....	187
3.7.2. DCT .....	189
3.7.3. Практическое DCT .....	191
3.7.4. Квантование .....	192
3.7.5. Кодирование .....	195
3.7.6. Мода без потери данных .....	200
3.7.7. Сжатый файл .....	201
3.7.8. JFIF .....	202
3.8. JPEG-LS .....	205
3.8.1. Коды Голомба .....	205
3.8.2. Основы метода JPEG-LS .....	205
3.8.3. Кодер .....	207
<b>Глава 4.</b>	
<b>Вейвлетные методы .....</b>	<b>214</b>
4.1. Вычисление средних и полуразностей .....	215

4.1.1. Обобщение на двумерный случай .....	218
4.1.2. Свойства преобразования Хаара .....	225
4.2. Преобразование Хаара .....	232
4.2.1. Матричная форма .....	233
4.3. Поддиапазонные преобразования .....	236
4.4. Банк фильтров .....	242
4.5. Нахождение коэффициентов фильтра .....	250
4.6. Преобразование DWT .....	252
4.7. Примеры .....	260
4.8. Вейвлеты Добеши .....	264
4.9. SPIHT .....	267
4.9.1. Алгоритм сортировки разделением множеств .....	274
4.9.2. Пространственно ориентированное дерево .....	276
4.9.3. Кодирование в алгоритме SPIHT .....	278
4.9.4. Пример .....	282
4.9.5. QTCQ .....	284
<b>Глава 5.</b>	
<b>Сжатие видео .....</b>	<b>286</b>
5.1. Основные принципы .....	287
5.2. Методы подоптимального поиска .....	294
<b>Глава 6.</b>	
<b>Сжатие звука .....</b>	<b>304</b>
6.1. Звук .....	305
6.2. Оцифрованный звук .....	309
6.3. Органы слуха человека .....	312
6.4. Общепризнанные методы .....	316
6.5. Сжатие звука в стандарте MPEG-1 .....	320
6.5.1. Кодирование частотной области .....	324
6.5.2. Формат сжатых данных .....	328
6.5.3. Психоакустические модели .....	331
6.5.4. Кодирование: слой III .....	332
<b>Литература .....</b>	<b>340</b>
<b>Глоссарий .....</b>	<b>346</b>
<b>Сообщество сжатия данных .....</b>	<b>358</b>
<b>Список алгоритмов .....</b>	<b>359</b>
<b>Предметный указатель .....</b>	<b>361</b>

## Предисловие переводчика

Никто не будет отрицать, что политическая и экономическая активность в современном обществе в значительной степени держится на надежных коммуникациях, в которые вовлечены огромные объемы информации. С целью обеспечения информационных средств сообщения были разработаны и продолжают разрабатываться всевозможные электронные устройства и программные комплексы для передачи, отображения и хранения информации. Это телеграф, радио, телефония, телевидение, модемы, космические линии связи, оптические диски и многое другое. Основная проблема, которую необходимо решить при построении системы коммуникации, была впервые сформулирована Клодом Шенноном в 1948 году:

*Главное свойство системы связи заключается в том, что она должна точно или приблизенно воспроизвести в определенной точке пространства и времени некоторое сообщение, выбранное в другой точке. Обычно, это сообщение имеет какой-то смысл, однако это совершенно не важно для решения поставленной инженерной задачи. Самое главное заключается в том, что посыпаемое сообщение выбирается из некоторого семейства возможных сообщений.*

Такая точная и ясная постановка проблемы коммуникации оказала огромное воздействие на развитие средств связи. Возникла новая научная отрасль, которая стала называться *теорией информации*. Главная идея, обоснованная Шенноном, заключается в том, что надежные коммуникации должны быть цифровыми, т.е. задачу связи следует рассматривать как передачу двоичных цифр (битов).

На рис. 1 приведена общая схема передачи цифровой информации. Заметим, что любой физический канал передачи сигналов не может быть абсолютно надежным. На рис. 1 это проиллюстрировано шумом, который портит канал и вносит ошибки в передаваемую цифровую информацию. Шеннон показал, что при выполнении некоторых достаточно общих условий имеется принципиальная возможность использовать ненадежный канал для передачи информации со сколь угодно большой степенью надежности. Поэтому нет необходимости пытаться очистить канал от шумов, например, повышая

мощность сигналов (это дорого и зачастую невозможно). Вместо этого следует разрабатывать эффективные схемы кодирования и декодирования цифровых сигналов.



Рис. 1. Блок-схема системы связи.

Кроме того, Шеннон доказал, что задачу надежной связи можно разложить на две подзадачи без уменьшения ее эффективности. Эти две подзадачи называются *кодированием источника* и *кодированием канала* (см. рис. 2).



Рис. 2. Системы связи с раздельным кодированием.

Задача кодирования канала заключается в построении на основе известных характеристик канала кодера, посылающего в канал входные символы, которые будут декодированы приемником с максимальной степенью надежности. Это достигается с помощью добавления в передаваемую цифровую информацию некоторых дополнительных проверочных символов. На практике каналом может служить телефонный кабель, спутниковая антенна, оптический диск, память компьютера или еще что-то.

Задачей кодирования источника является создание кодера источника, который производит компактное (укороченное) описание исходного сигнала, который необходимо передать адресату. Источником сигналов может служить текстовый файл, цифровое изображение, оцифрованная музыка или телевизионная передача. Это сжа-

тое описание сигналов источника может быть неточным, тогда следует говорить о расхождении между восстановленным после приема и декодирования сигналом и его оригиналом. Это обычно происходит при преобразовании (квантовании) аналогового сигнала в цифровую форму.

Предлагаемая книга посвящена задачам кодирования источников. Она написана известным специалистом в этой области Дэвидом Сэломоном. В ней приводятся различные методы решения проблем сжатия цифровых данных, возникающих в информационных технологиях. Поскольку целью кодирования источников является создание компактного, сжатого описания цифровой информации, эту технику также принято называть *сжатием* или *компрессией* цифровых данных. Этот термин является весьма удачным, так как он содержит в себе интуитивное описание, поэтому он будет широко использоваться в данной книге.

Идея сжатия информации очень естественна, она проявляется и в обычном языке в виде различных сокращений. Главная причина использования сжатия данных в коммуникациях заключается в желании передавать или хранить информацию с наибольшей эффективностью. В качестве примера можно привести азбуку Морзе, которая сильно ускорила телеграфную связь. В некотором смысле задача сжатия данных заключается в извлечении из потока данных самой значимой и уникальной его части, которая позволяет целиком восстановить всю исходную информацию. Сжатые данные должны содержать только самую существенную информацию, здесь не должно быть места избыточным данным.

Обратимся к кодированию источников. Как и в обычном тексте в различных источниках цифровых данных имеются избыточные данные, то есть, некоторые участки, которые, "содержатся" в других частях данных источника. Поэтому возникает вопрос: что есть наиболее компактное представление данных источника? На этот вопрос ответ был дан Шенноном. В своей пионерской работе по теории информации он ввел числовую характеристику источника, которая называется *энтропией*. Фундаментальное значение этой величины состоит в том, что она задает нижнюю границу возможного сжатия. Кроме того, имеется теорема, которая утверждает, что к этой границе можно приблизиться сколь угодно плотно с помощью подходящего метода кодирования источника. Например, известно, что энтропия усредненного английского текста равна примерно 3.2 бит/букву. В компьютерном представлении одна буква занимает 8 бит (стандартный код ASCII). Значит, при сжатии ти-



личного текстового файла на английском языке достигается сжатие примерно в 2.5 раза.

Энтропия сжатых данных совпадает с энтропией исходного источника. При этом предполагается, что по сжатым данным можно полностью восстановить исходную информацию. Такой подход принято называть *сжатием без потерь*. Это сжатие, например, применяется в дистрибутивах программных продуктов. Это наиболее изученная область сжатия данных. Она является весьма важной, поскольку большинство методов компрессии самых разных типов цифровой информации часто используют на определенных стадиях алгоритмы сжатия без потерь. Такое сжатие еще называется *энтропийным сжатием*. *Избыточностью* данных можно назвать разность между их объемом и энтропией. Можно сказать, что компрессия без потерь является экстремальным случаем сжатия, при котором энтропия данных остается неизменной.

Здесь мы приходим к другой важной проблеме: каково наиболее компактное представление информации, если допускается неточное восстановление сжатых данных. Такое сжатие называется сжатием с *частичной потерей информации*. Сжатие с потерями, по существу, предполагает уменьшение энтропии исходной информации. На практике компрессия с потерями применяется, например, при сжатии цифровых изображений и оцифрованного звука. Величина допустимого расхождения оригинальных данных и их разжатой копии часто определяется субъективно по их неразличимости глазом или ухом человека, который, в конечном итоге является получателем данного рода информации.

Эти проблемы будут подробно разбираться в книге с привлечением большого числа примеров, таблиц и иллюстраций. Многие важные методы и алгоритмы сжатия данных представлены в виде программ, написанных на языке популярных пакетов Matlab и Mathematica. Большинство из них будут тщательно разобраны по шагам применительно к конкретным сжимаемым образцам и фрагментам.

Читателю понадобятся лишь базовые знания по математике и информатике для успешного усвоения материала книги. Для более глубокого изучения теории информации и других смежных вопросов информационных технологий можно обратиться к дополнительному списку литературы на русском языке, приведенному на стр. 344.

## Предисловие автора

В 1829 году Луи Брайль (Louis Braille), молодой органист одного парижского собора, который потерял зрение в раннем детстве в возрасте трех лет, изобрел знаменитый шрифт для слепых. Этот шрифт, названный его именем, широко распространился по всему миру, позволяя слепым людям читать и писать. Брайль слегка усовершенствовал свои коды в 1834 году. Было еще несколько незначительных изменений, однако осталось неизменным основное начертание символов или букв, каждая из которых представлена в виде блока точек  $3 \times 2$ . Эти точки выдавливаются на листе плотной бумаги, причем каждая может быть поднята вверх или опущена, что означает, она присутствует или отсутствует. Таким образом, каждая точка эквивалентна одному биту информации. В результате, шрифт Брайля (см. рис.1) представляет собой шестибитовый код, с помощью которого можно представить ровно 64 символа (блок, в котором все точки опущены, обозначает пустой символ или пробел).

Рис. 1. Алфавит Брайля.

Последователи Брайля расширили возможности его шрифта несколькими путями. Прежде всего они ввели сокращения. Некоторые отдельно стоящие буквы стали обозначать целые слова. Например, отдельно стоящая буква «b» (или со знаком препинания) обозначает слово «but» (мы будем вести речь об английском варианте шрифта Брайля). Одиночная буква «e» означает «every», а буква «p» – «people».

Другим полезным правилом стало использование сокращенных форм некоторых часто используемых слов, то есть, комбинации двух и более символов стали обозначать целые слова. Например, «ab» означает «about», «rcv» – «receive», а буквосочетание «(the)mvs» – это



«themselves». (Символы «the» в круглых скобках тоже имеют свой специальный код, в котором подняты точки 2-3-4-6.) На рис. 2 показаны некоторые специальные коды и соответствующие им слова или части слов.

ING not and for of the with ch gh sh th

Рис. 2. Некоторые сокращения и короткие слова.

## Немного о Луи Брайле—

Луи Брайль родился 4 января 1809 года в городе Купврэ недалеко от Парижа. Несчастный случай лишил его зрения в возрасте 3 лет и сделал слепым на всю жизнь. Когда Луи исполнилось 10 лет, родители послали его в парижскую школу для слепых, где он учился читать с помощью специального рельефно-точечного шрифта. Этот шрифт был изобретен Шарлем Барбье для использования в армии. Он назывался «ночное письмо». С его помощью офицеры и солдаты могли обмениваться сообщениями в темноте. В основе шрифта ночного письма лежали блоки из 12 точек, две точки в длину и шесть точек в высоту. Каждая точка или комбинация выпуклых точек в одном блоке обозначала букву или фонетический звук. В таком представлении, однако, имелось одно существенное неудобство, замедлившее чтение сообщений: было невозможно пропускать весь блок точек за одно быстрое касание пальцем.

Брайль, потратив 9 лет (а ведь он был слепым!) на развитие и усовершенствование ночного письма, разработал свою систему выпуклых точек, которая теперь носит его имя. Главное усовершенствование кода заключалось в уменьшении блока символа. Вместо матрицы  $6 \times 2$  он предложил использовать матрицу  $3 \times 2$  точек. Прежде всего это делалось для того, чтобы читающий мог легко распознать символы за одно легкое касание, быстро перемещая палец по строчке символов.

Шрифт Брайля начал вводиться в Соединенных Штатах Америки в 1860 году. С его помощью стали учить незрячих детей в школе святого Луи для слепых. В 1868 году была основана британская ассоциация слепых. Позже стали образовываться объединения незрячих людей в других странах. Шрифт Брайля стал применяться в Англии и в других странах. Начало развиваться книгопечатание на брайле, стали распространяться книги для слепых.

В США существует Североамериканское Общество Брайля (Braille Authority of North America, BANA) с сайтом <http://www.brailleauthority.org/index.html>. Целью этой организации является продвижение шрифта Брайля, содействие в его использовании при обучении. Общество регулярно публикует стандарты и правила языка брайля и следит за правильностью его употребления.

Сокращения, укороченные слова и некоторые другие приемы использования шрифта Брайля – все это примеры *интуитивного сжатия информации*. Люди, занимавшиеся развитием шрифта Брайля и приспособлением его к другим языкам, осознали, что некоторые

часто встречающиеся слова и буквосочетания можно заменить специальными знаками или короткими кодами для ускорения чтения и письма. В основе современных методов сжатия информации лежит та же идея: *чем чаще встречаются объекты в массиве сжимаемых данных, тем короче сопоставляемый им код*.

Предшественницей этой книги послужила монография «Сжатие данных: полное руководство» («Data Compression: The Complete Reference»), опубликованная в 1977 году и переизданная в конце 2000 года. Быстрые и весьма благожелательные читательские отклики на это издание побудили меня к написанию этой небольшой книги. В первой книге я стремился самым подробным образом осветить как основные принципы сжатия данных, так и все детали множества специфических методов. Поэтому она вышла такой объемной. При написании новой книги я был менее амбициозен. В ней я заился целью провести неискушенного читателя по полям сжатия, чтобы он ощущал чудесный аромат этих полей. Это будет сделано с помощью представления основных приемов сжатия информации, а также с помощью описания ключевых алгоритмов. В книге совсем немного математики, нет упражнений, но зато имеется множество примеров, которые помогут проиллюстрировать основные методы.

Во введении объясняется, почему информацию можно сжимать, рассматриваются некоторые простые примеры, а также обсуждаются основные технические термины, которые будут использоваться в следующих главах.

В главе 1 обсуждаются статистические методы сжатия (компрессии) информации. В основе этих методов лежит оценка вероятностей появления элементарных символов в сжимаемом массиве информации, которая определяет коды переменной длины сопоставляемые этим символам. Элементарными символами могут быть биты, ASCII-коды, байты, пиксели, аудио-фрагменты или другие компоненты. Главной идеей этой главы является использование кодов переменной длины, так называемых, префикс-кодов. Среди описываемых методов: кодирование Хаффмана, факсимильное сжатие и арифметическое кодирование.

Популярная техника словарного сжатия является предметом главы 2. Метод словарного сжатия основан на сохранении байтов и фрагментов сжимаемого файла в виде специальной структуры, называемой словарем. Для каждого нового фрагмента данных делается поиск в словаре. Если этот фрагмент находится в словаре, то в сжатый файл записывается ссылка на этот фрагмент. В этой гла-

ве описываются следующие известные алгоритмы компрессии этого типа: LZ77, LZSS, LZ78 и LZW.

Чрезвычайно важными объектами сжатия в компьютерных приложениях являются всевозможные оцифрованные изображения (графики, рисунки, картинки или фотографии), которые обычно имеют большой объем. Глава 3 посвящена сжатию изображений. В основном в этой главе обсуждаются различные подходы к решению этой проблемы, такие как кодирование длинных серий, вероятность контекста, предсказание пикселов и преобразование изображений. Из конкретных алгоритмов обсуждаются JPEG и JPEG-LS.

В главе 4 рассматривается преобразование вейвлетов. Этот метод становится все более важным в сжатии изображений и в компрессии аудио-видео информации. Для чтения этой главы необходимы некоторые математические сведения, которые для удобства читателя здесь же излагаются. Глава начинается описанием интуитивной техники, основанной на вычислении среднего и разностей, которая имеет прямое отношение к вейвлетному преобразованию Хаара. Затем вводится понятие банка фильтров, которое рассматривается после дискретного преобразования вейвлетов. В конце главы излагается алгоритм компрессии SPIHT, основанный на преобразовании вейвлетов.

Мультфильм является, в некотором смысле, обобщением одной единственной картинки. Мультфильмы и анимация, быстро заполняющие компьютерные мультимедийные программы, обусловливают пристальное внимание к сжатию видео информации. Анимационный файл неизмеримо объемнее одной картинки, поэтому эффективное сжатие видео становится просто необходимым. При этом сжатие (компрессия) видео должно быть простым и иметь быструю декомпрессию, чтобы аппаратура успевала сделать необходимую декомпрессию в реальном времени, иначе зритель увидит лишь дергающееся изображение с выпадающими кадрами. Принципы сжатия видео излагаются в главе 5.

В последней главе, главе 6, рассматривается аудиокомпрессия. Звук является одним из «медиа» в компьютерных мультимедиа приложениях. Он очень популярен среди компьютерных пользователей. Перед сохранением и использованием на компьютере звук необходимо оцифровать. В результате получается исходный аудиофайл очень большого объема. В этой главе описаны основные операции, используемые в знаменитом методе аудиосжатия MP3, который, на самом деле, является звуковой частью алгоритма MPEG-1. В главе имеется

ся короткое введение в теорию звука и его восприятия человеком, а также в аудиосэмплирование.

Эта книга прежде всего предназначена тем читателям, которые хотят узнать и освоить основные методы сжатия информации, но у которых нет времени вдаваться во все обширные технические детали множества алгоритмов компрессии и декомпрессии. Я также надеюсь, что использование математики позволит полнее раскрыть «тайны» сжатия информации читателю, который не является экспертом в этой важной области компьютерной науки.

Я зарегистрировал домен BooksByDavidSalomon.com, на котором всегда будут находиться свежие ссылки по тематике сжатия информации. Мне можно писать по адресу: david.salomon@csun.edu, также читатели могут использовать более удобный переадресуемый адрес <anynname>@BooksByDavidSalomon.com.

Читатели, которые заинтересовались сжатием информации, могут обратиться к небольшому разделу «Сообщество сжатия данных» в конце этой книги, а также посетить сайты в интернете со следующими адресами: <http://www.internz.com/compression-pointers.html> и [http://www.hn.is.uec.ac.jp/~arimura/compression\\_links.html](http://www.hn.is.uec.ac.jp/~arimura/compression_links.html).

Northridge, California

David Salomon

*Математика это трудно.*  
— Барби

*Non mi legga chi non è matematico.*  
(Пусть не читает меня не математик.)  
— Леонардо да Винчи

## Введение

Всем, кто использует компьютерные программы сжатия информации, хорошо знакомы такие слова, как «zip», «implode», «stuffit», «dien» и «squeeeze». Всё это имена программ или названия методов для компрессии компьютерной информации. Перевод этих слов в той или иной степени означает застегивание, уплотнение, набивку или сжатие. Однако обычный, языковый смысл этих слов или их перевод не в полной мере отражают истинную природу того, что происходит с информацией в результате компрессии. На самом деле, при компрессии компьютерной информации ничего не набивается и не ужимается, но лишь удаляется некоторый *избыток информации*, присущий в исходных данных. *Избыточность* – вот центральное понятие в теории сжатия информации. Любые данные с избыточной информацией можно сжать. Данные, в которых нет избыточности, сжать нельзя, точка.

Мы все хорошо знаем, что такое информация. Интуитивно нам это вполне понятно, однако это скорее качественное восприятие предмета. Информация представляется нам одной из сущностей, которые невозможно точно определить и тем более измерить количественно. Однако, существует область математики, которая называется *теорией информации*, где информацию изучают именно количественно. Другим важным достижением теории информации является вполне строгое определение избыточности. Сейчас мы попытаемся истолковать это понятие интуитивно, указав, что есть избыточность для двух простых типов компьютерных данных, и что представляют собой данные, из которых удалена избыточность.

Первый тип информации – это текст. Текст представляет собой важнейший вид компьютерных данных. Огромное количество компьютерных программ и приложений являются по своей природе нечисловыми; они работают с данными, у которых основными элементарными компонентами служат символы текста. Компьютер способен сохранять и обрабатывать лишь двоичную информацию, состоящую из нулей и единиц. Поэтому каждому символу текста необходимо сопоставить двоичный код. Современные компьютеры используют так называемые коды ASCII (произносится «аски», а само

слово ASCII является сокращением от «American Standard Code for Information Interchange»), хотя все больше компьютеров и приложений используют новые коды Unicode. ASCII представляет код фиксированной длины, где каждому символу присваивается 8-битовая последовательность (сам код занимает семь битов, а восьмой – проверочный, который изначально был задуман для повышения надежности кода). Код фиксированной длины представляется наилучшим выбором, поскольку позволяет компьютерным программам легко оперировать с символами различных текстов. С другой стороны, код фиксированной длины является по существу избыточным.

В случайном текстовом файле мы ожидаем, что каждый символ встречается приблизительно равное число раз. Однако файлы, используемые на практике, навряд ли являются случайными. Они содержат осмысленные тексты, и по опыту известно, что, например, в типичном английском тексте некоторые буквы, такие, как «E», «T» и «A», встречаются гораздо чаще, чем «Z» и «Q». Это объясняет, почему код ASCII является избыточным, а также указывает на пути устранения избыточности. ASCII избытен прежде всего потому, что независимо присваивает каждому символу, часто или редко используемому, одно и то же число бит (восемь). Чтобы удалить такую избыточность, можно воспользоваться кодами переменной длины, в котором короткие коды присваиваются буквам, встречающимся чаще, а редко встречающимся буквам достаются более длинные коды. Точно так работает кодирование Хаффмана (см. § 1.4).

Представим себе два текстовых файла *A* и *B*, содержащие один и тот же текст, причем файл *A* использует коды ASCII, а *B* записан с помощью некоторых кодов переменной длины. Мы ожидаем, что размер файла *B* меньше размера *A*. Тогда можно сказать, что *файл A сжат в файл B*. Понятно, что степень сжатия зависит от избыточности взятого текста, а также от используемых кодов переменной длины. Текст, в котором одни символы встречаются очень часто, а другие – очень редко, имеет большую избыточность; он будет сжиматься хорошо, если коды переменной длины выбраны подходящим образом. В соответствующем файле *B* коды часто встречающихся символов будут короткими, а коды редких символов – длинными. Длинные коды не смогут понизить степень сжатия, так как они будут встречаться в файле *B* достаточно редко. Большая часть *B* будет состоять из коротких кодов. С другой стороны, случайный текстовый файл не получает преимущество при замене кодов

ASCII кодами переменной длины, потому что сжатие, достигнутое с использованием коротких кодов, будет аннулировано длинными кодами. В этом частном примере работает общее правило, которое гласит: *случайные данные невозможно сжать, так как в них нет избыточности*.

### Из новостей FIDO, 23 апреля 1990

---

Вокруг нас разгорается пламя войны, в которой спорным вопросом является: чья программа компрессии данных является самой лучшей. Я решил тоже в ней поучаствовать и написать СВОЮ собственную программу.

Вы конечно слышали про программы, которые скрывают, глушат, ужимают, вдавливают, пакуют, дробят и т.д.

Теперь есть программа TRASH (в мусор).

TRASH сжимает файл до наименьшего возможного размера: 0 байт! Ничто не сожмет файл лучше, чем TRASH. Атрибуты дата/время не затрагиваются, и поскольку файл имеет нулевую длину, он совсем не займет место на вашем винчестере!

И TRASH очень быстра. Ваши файлы будут скомканы за микросекунды! Вы потратите больше времени взглядываясь в задаваемые параметры, в то время как файл будет уже обработан.

Это предпродажная версия моей программы. Вы можете хранить и испытывать ее. Я бы вам рекомендовал сделать резервную копию ваших файлов перед тем, как вы впервые запустите мой TRASH, хотя...

Следующая версия TRASH будет иметь графический интерфейс и принимать кредитные карты.

TRASH C:\PAYROOL\\*.\*

... и работать с целыми дисками

TRUSH D:

... и быть первой, чтобы заблокировать сбрасывание в мусор вашей системы НАРОЧНО!

TRUSH ALL

Мы даже надеемся научить нашу программу восстанавливать заTRASHенные файлы!

---

Второй тип компьютерных данных – это оцифрованные изображения: фотографии, рисунки, картинки, графики и т.п. Цифровое изображение – это прямоугольная матрица окрашенных точек, называемых *пикселями*. Каждый пиксель представляется в компьютере с помощью цветового кода. (До конца этого параграфа термин «пиксель» используется только для цветового кода.) Для упрощения цифровой обработки изображений предполагается, что все пиксели имеют один и тот же размер. Размер пикселя зависит от числа цветов в изображении, которое, обычно, является степенью 2. Если в нем содержится  $2^k$  разных цветов, то каждый пиксель – это  $k$ -битовое число.

Имеется два вида избыточности в цифровых изображениях. Первый вид похож на избыточность в текстовом файле. В каждом неслучайном изображении некоторые цвета могут преобладать, а другие встречаться редко. Такая избыточность может быть удалена с помощью кодов переменной длины, присваиваемых разным пикселям, точно также как и при сжатии текстовых файлов. Другой вид избыточности гораздо более важен, он является результатом *корреляции пикселов*. Когда наш взгляд перемещается по картинке, он обнаруживает в большинстве случаев, что соседние пиксели окрашены в близкие цвета. Представим себе фотографию, на которой изображено голубое небо, белые облака, коричневые горы и зеленые деревья. Пока мы смотрим на горы, близкие пиксели имеют похожий цвет; все или почти все из них имеют разные оттенки коричневого цвета. Про близкие пиксели неба можно сказать, что они носят различные оттенки голубого цвета. И только на горизонте, там, где горы встречаются с небом, соседние пиксели могут иметь совершенно разные цвета. Таким образом, отдельные пиксели не являются совершенно независимыми. Можно сказать, что ближайшие пиксели изображения коррелируют между собой. С этим видом избыточности можно бороться разными способами. Об этом будет рассказано в главе 3.

Независимо от метода, которым сжимается изображение, эффективность его компрессии определяется прежде всего количеством избыточности, содержащимся в нем. Предельный случай – это однотонное изображение. Оно имеет максимальную избыточность, потому что соседние пиксели тождественны. Понятно, такое изображение не интересно с практической точки зрения, оно, если встречается, то крайне редко. Тем не менее, оно будет очень хорошо сжиматься любым методом компрессии. Другим экстремальным случаем является изображение с некоррелированными, то есть, случайными пикселями. В таком изображении соседние пиксели, как правило, весьма различаются по своему цвету, а избыточность этого изображения равна нулю. Его невозможно сжать никаким методом. Оно выглядит как случайная мешанина окрашенных точек, и потому не интересно. Нам вряд ли понадобится сохранять и обрабатывать подобные изображения, и нет смысла пытаться их сжимать.

Следующее простое наблюдение поясняет существование утверждения: «Сжатие данных достигается сокращением и удалением из них избыточности». Оно также дает понять, что большинство файлов невозможно сжать никакими методами. Это может показаться странным, так как мы постоянно сжимаем свои файлы. Причина заключа-

ется в том, что большинство файлов являются случайными или почти случайными, и поэтому, в них совсем нет избыточности. Существует относительно немного файлов, которые можно сжимать, и именно с ними мы хотим это сделать, с этими файлами мы работаем все время. В них есть избыточность, они не случайны, а потому полезны и интересны.

Пусть два разных файла  $A$  и  $B$  сжаты в файлы  $C$  и  $D$ , соответственно. Ясно, что  $C$  и  $D$  должны также отличаться друг от друга. В противном случае было бы невозможно по ним восстановить исходные файлы  $A$  и  $B$ .

Предположим, что файл состоит из  $n$  бит, и мы хотим его сжать эффективным образом. Будем приветствовать любой алгоритм, который сожмет этот файл, скажем, в 10 бит. Компрессия в 11 или 12 бит тоже была бы замечательна. Сжатие файла до половины его размера будет для нас вполне удовлетворительным. Всего существует  $2^n$  различных файлов размера  $n$ . Их надо бы сжать в  $2^n$  различных файлов размера не больше  $n/2$ . Однако, общее число таких файлов равно

$$N = 1 + 2 + 4 + \cdots + 2^{n/2} = 2^{1+n/2} - 1 \approx 2^{1+n/2},$$

поэтому только  $N$  исходных файлов имеют шанс сжаться эффективным образом. Проблема в том, что число  $N$  существенно меньше числа  $2^n$ . Вот два примера соотношения между ними.

Для  $n = 100$  (файл всего в 100 бит) общее число файлов равно  $2^{100}$ , а число файлов, эффективно сжимаемых, равно  $2^{51}$ . А их частное просто до смешного малая дробь  $2^{-49} \approx 1.78 \cdot 10^{-15}$ .

Для  $n = 1000$  (файл из 1000 бит, т.е., около 125 байт) число всех файлов равно  $2^{1000}$ , а число сжимаемых всего  $2^{501}$ . Их доля просто катастрофически мала, она равна  $2^{-499} \approx 9.82 \cdot 10^{-91}$ .

Большинство интересных файлов имеют размер по крайней мере в несколько тысяч байт. Для таких размеров доля эффективно сжимаемых файлов настолько мала, что ее невозможно выразить числом с плавающей точкой даже на суперкомпьютере (результат будет нуль).

Из этих примеров становится ясно, что не существует методов и алгоритмов, способных эффективно сжимать **ЛЮБЫЕ** файлы, или даже существенную часть их. Для того, чтобы сжать файл, алгоритм компрессии должен сначала изучить его, найти в нем избыточность, а потом попытаться удалить ее. Поскольку избыточность зависит от типа данных (текст, графика, звук и т.д.), методы компрессии должны разрабатываться с учетом этого типа. Алгоритм

будет лучше всего работать именно со своими данными. В этой области не существует универсальных методов и решений.

В конце введения напомним некоторые важные технические термины, которые используются в области сжатия информации.

- *Компрессор* или *кодер* – программа, которая сжимает «сырой» исходный файл и создает на выходе файл со сжатыми данными, в которых мало избыточности. *Декомпрессор* или *декодер* работает в обратном направлении. Отметим, что понятие кодера является весьма общим и имеет много значений, но поскольку мы обсуждаем сжатие данных, то у нас слово кодер будет означать компрессор. Термин *кодек* иногда используется для объединения кодера и декодера.
- Метод *неадаптивного сжатия* подразумевает неспособность алгоритма менять свои операции, параметры и настройки в зависимости от сжимаемых данных. Такой метод лучше всего сжимает однотипные данные. К ним относятся методы группы 3 и группы 4 сжатия факсимильных сообщений (см. § 1.6). Они специально разработаны для сжатия в факс-машинах и будут весьма слабо работать на других типах данных. Напротив, *адаптивные* методы сначала тестируют «сырые» исходные данные, а затем подстраивают свои параметры и/или операции в соответствии с результатом проверки. Примером такого алгоритма может служить кодирование Хаффмана из § 1.5. Некоторые методы компрессии используют двухпроходные алгоритмы, когда на первом проходе по файлу собирается некоторая статистика сжимаемых данных, а на втором проходе происходит непосредственно сжатие с использованием параметров, вычисленных на первой стадии. Такие методы можно назвать *полуадаптивными*. Методы компрессии могут быть также *локально адаптивными*, что означает способность алгоритма настраивать свои параметры исходя из локальных особенностей файла и менять их, перемещаясь от области к области входных данных. Пример такого алгоритма приведен в [Salomon 2000].
- *Компрессия без потерь/с потерей*: некоторые методы сжатия допускают потери. Они лучше работают, если часть информации будет опущена. Когда такой декодер восстанавливает данные, результат может не быть тождественен исходному файлу. Такие методы полезны, когда сжимаемыми данными

является графика, звук или видео. Если потери невелики, то бывает невозможно обнаружить разницу. А текстовые данные, например, текст компьютерной программы, могут стать совершенно непригодными, если потеряется или изменится хоть один бит информации. Такие файлы можно сжимать только методами без потери информации. (Здесь стоит отметить два момента относительно текстовых файлов. (1) Если текстовый файл содержит исходный код компьютерной программы, то из него можно безболезненно удалить большинство пробелов, так как компилятор, обычно, их не рассматривает. (2) Когда программа текстового процессора сохраняет набранный текст в файле, она также сохраняет в нем информацию об используемых шрифтах. Такая информация также может быть отброшена, если автор желает сохранить лишь текст своего произведения).

- *Симметричное сжатие* – это когда и кодер, и декодер используют один и тот же базовый алгоритм, но используют его в противоположных направлениях. Такой метод имеет смысл, если постоянно сжимается и разжимается одно и то же число файлов. При *асимметричном* методе или компрессор или де-компрессор должны проделать существенно большую работу. Таким методам тоже находится место под солнцем, они совсем не плохи. Метод, когда компрессия делается долго и тщательно с помощью сложнейшего алгоритма, а декомпрессия делается быстро и просто, вполне оправдан при работе с архивами, когда приходится часто извлекать данные. То же происходит и при создании и прослушивании аудиофайлов формата mp3. Обратный случай, это когда внешние файлы часто меняются и сохраняются в виде резервных копий. Поскольку вероятность извлечения резервных данных невелика, то декодер может работать существенно медленнее своего кодера.
- *Производительность сжатия*: несколько величин используются для вычисления эффективности алгоритмов сжатия.

1) *Коэффициент сжатия* определяется по формуле

$$\text{Коэффициент сжатия} = \frac{\text{размер выходного файла}}{\text{размер входного файла}}.$$

Коэффициент 0.6 означает, что сжатые данные занимают 60% от исходного размера. Значения большие 1 говорят

рят о том, что выходной файл больше входного (отрицательное сжатие). Коэффициент сжатия принято изменять в  $bpb$  (bit per bit, бит на бит), так как он показывает, сколько в среднем понадобится бит сжатого файла для представления одного бита файла на входе. При сжатии графических изображений аналогично определяется величина  $bpp$  (bit per pixel, бит на пиксел). В современных эффективных алгоритмах сжатия текстовой информации имеет смысл говорить о похожей величине  $bpc$  (bit per character, бит на символ), то есть, сколько в среднем потребуется бит для хранения одной буквы текста.

Здесь следует упомянуть еще два понятия, связанные с коэффициентом сжатия. Термин **битовая скорость (bit-rate)** является более общим, чем  $bpb$  и  $bpc$ . Целью компрессии информации является представление данных с наименьшей битовой скоростью. **Битовый бюджет (bit budget)** означает некоторый довесок к каждому биту в сжатом файле. Представьте себе сжатый файл, в котором 90% размера занимают коды переменной длины, соответствующие конкретным символам исходного файла, а оставшиеся 10% используются для хранения некоторых таблиц, которые будут использоваться декодером при декомпрессии. В этом случае битовый бюджет равен 10%.

- 2) Величина, обратная коэффициенту сжатия, называется *фактором сжатия*:

$$\text{Фактор сжатия} = \frac{\text{размер входного файла}}{\text{размер выходного файла}}.$$

В этом случае значения большие 1 означают сжатие, а меньшие 1 – расширение. Этот множитель представляется большинству людей более естественным показателем: чем больше фактор, тем лучше компрессия. Эта величина отдаленно соотносится с коэффициентом разреженности, который будет обсуждаться в § 4.1.2.

- 3) Выражение  $100 \times (1 - k)$ , где  $k$  – коэффициент сжатия, тоже отражает качество сжатия. Его значение равное 60 означает, что в результате сжатия занимает на 60% меньше места, чем исходный файл.
- 4) Для оценивания эффективности алгоритмов сжатия образов используется величина  $bpp$ . Она показывает, сколько



необходимо в среднем использовать битов для хранения одного пикселя. Это число удобно сравнивать с его значением до компрессии.

- *Вероятностная модель.* Это понятие очень важно при разработке статистических методов сжатия данных. Зачастую, алгоритм компрессии состоит из двух частей, вероятностной модели и непосредственно компрессора. Перед тем как приступить к сжатию очередного объекта (бита, байта, пикселя и т.п.), активируется вероятностная модель и вычисляется вероятность этого объекта. Эта вероятность и сам объект сообщаются компрессору, который использует полученную информацию при сжатии. Чем выше вероятность, тем лучше сжатие.

Приведем пример простой модели для черно-белого изображения. Каждый пиксель такого изображения – это один единственный бит. Предположим, что алгоритм уже прочитал и сжал 1000 пикселов и читает 1001-ый пиксель. Какова вероятность того, что пиксель будет черным? Модель может просто сосчитать число черных пикселов в уже прочитанном массиве данных. Если черных пикселов было 350, то модель приписывает 1001-ому пикселу вероятность  $350/1000=0.35$  быть черным. Эта вероятность вместе с пикселиом (черным или белым) передаются компрессору. Главным моментом является то, что декодер может также легко вычислить вероятность 1001-ого пикселя.

- Слово *алфавит* означает множество символов в сжимаемых данных. Алфавит может состоять из двух символов, 0 и 1, из 128 символов ASCII, из 256 байтов по 8 бит в каждом или из любых других символов.
- Возможности каждого алгоритма сжатия имеют ограничения. Никакой алгоритм не может эффективно сжимать любые файлы. Как было уже показано, любой алгоритм может эффективно сжимать лишь малую долю файлов, которые не являются случайными. Подавляющее же число случайных или близких к случайным файлов не поддается сжатию.

Последнее заключение выглядит немного обескураживающим, но я надеюсь, оно не заставит читателя закрыть эту книгу со вздохом и обратиться к другим многообещающим занятиям. Это удивительно, но среди  $2^n$  всевозможных файлов размера  $n$ , именно те

файлы, которые мы захотим сжать, будут сжиматься хорошо. Те же, которые сжимаются плохо, являются случайными, а значит, неинтересными и неважными для нас. Мы не будем их сжимать, пересыпать или хранить.

*Чудесные дни перед свадьбой сродни живому вступлению к скучной книге.*

— Уилсон Мизнер

# ГЛАВА I

## СТАТИСТИЧЕСКИЕ МЕТОДЫ

Статистические методы компрессии используют статистические свойства сжимаемых данных и присваивают всем символам коды с переменной длиной. Под «статистическими свойствами» обычно понимается вероятность (или, что то же самое, частота появления) каждого символа в потоке данных, однако этот термин может иметь иное, более сложное значение. Пару последовательных символов будем называть биграммой. Долгие наблюдения показали, что в типичном английском тексте некоторые биграммы, например, «ta», «he», «са», встречаются очень часто, а другие, например, «ха», «hz», «qe», – редко. Поэтому разумный статистический метод может присваивать коды переменной длины многим биграммам (и даже триграммам), а не только индивидуальным символам.

### 1.1. Энтропия

Основы теории информации были заложены Клодом Шеноном в 1948 году в лаборатории Белла. Эта теория оказалась настоящим сюрпризом, поскольку все привыкли воспринимать информацию лишь качественно. Для наших целей сжатия данных нам необходимо освоить только одно теоретико-информационное понятие, а именно, энтропию. Под энтропией символа  $a$ , имеющего вероятность  $P$ , подразумевается количество информации, содержащейся в  $a$ , которая равна  $-P \log_2 P$ . Например, если вероятность  $P_a$  символа  $a$  равна 0.5, то его энтропия  $-P_a \log_2 P_a = 0.5$ .

Если символы некоторого алфавита с символами от  $a_1$  до  $a_n$  имеют вероятности от  $P_1$  до  $P_n$ , то энтропия всего алфавита равна сумме  $\sum_n -P_i \log_2 P_i$ . Если задана строка символов этого алфавита, то для нее энтропия определяется аналогично.

С помощью понятия энтропии теория информации показывает, как вычислять вероятности строк символов алфавита, и предсказывает ее наилучшее сжатие, то есть, наименьшее, в среднем, число бит, необходимое для представления этой строки символов.

Продемонстрируем это на простом примере. Для последовательности символов «ABCDE» с вероятностями 0.5, 0.2, 0.1, 0.1 и 0.1, соответственно, вероятность строки «AAAAAABCDE» равна  $P = 0.5^5 \times 0.2^2 \times 0.1^3 = 1.25 \times 10^{-6}$ . Логарифм по основанию 2 этого числа  $\log_2 P = -19.6096$ . Тогда наименьшее в среднем число требуемых бит для кодирования этой строки равно  $-\lceil \log_2 P \rceil$ , то есть, 20. Кодер, достигающий этого сжатия называется *энтропийным кодером*.

**Пример:** Проанализируем энтропию алфавита, состоящего всего из двух символов  $a_1$  и  $a_2$  с вероятностями  $P_1$  и  $P_2$ . Поскольку  $P_1 + P_2 = 1$ , то энтропия этого алфавита выражается числом  $-P_1 \log_2 P_1 - (1 - P_1) \log_2(1 - P_1)$ . В табл. 1.1 приведены различные значения величин  $P_1$  и  $P_2$  вместе с соответствующей энтропией. Когда  $P_1 = P_2$ , необходим по крайней мере один бит для кодирования каждого символа. Это означает, что энтропия достигла своего максимума, избыточность равна нулю, и данные невозможно сжать. Однако, если вероятности символов сильно отличаются, то минимальное число требуемых бит на символ снижается. Мы, скорее всего, не сможем непосредственно предъявить метод сжатия, который использует 0.08 бит на символ, но мы точно знаем, что при  $P_1 = 0.99$  такой алгоритм теоретически возможен.

$P_1$	$P_2$	Энтропия
0.99	0.01	0.08
0.90	0.10	0.47
0.80	0.20	0.72
0.70	0.30	0.88
0.60	0.40	0.97
0.50	0.50	1.00

Табл. 1.1. Вероятности и энтропии двух символов.

## 1.2. Коды переменной длины

Первое правило построения кодов с переменной длиной вполне очевидно. Короткие коды следует присваивать часто встречающимся символам, а длинные – редко встречающимся. Однако есть другая проблема. Эти коды надо назначать так, чтобы их было возможно декодировать однозначно, а не двусмысленно. Маленький пример прояснит это.

Рассмотрим четыре символа  $a_1$ ,  $a_2$ ,  $a_3$  и  $a_4$ . Если они появляются в последовательности данных с равной вероятностью ( $=0.25$

каждая), то мы им просто присвоим четыре двухбитовых кода 00, 01, 10 и 11. Все вероятности равны, и поэтому коды переменной длины не сожмут эти данные. Для каждого символа с коротким кодом найдется символ с длинным кодом и среднее число битов на символ будет не меньше 2. Избыточность данных с равновероятными символами равна нулю, и строку таких символов невозможно сжать с помощью кодов переменной длины (или любым иным методом).

Предположим теперь, что эти четыре символа появляются с разными вероятностями, указанными в табл. 1.2, то есть  $a_1$  появляется в строке данных в среднем почти в половине случаев,  $a_2$  и  $a_3$  имеют равные вероятности, а  $a_4$  возникает крайне редко. В этом случае имеется избыточность, которую можно удалить с помощью переменных кодов и сжать данные так, что потребуется меньше 2 бит на символ. На самом деле, теория информации говорит нам о том, что наименьшее число требуемых бит на символ в среднем равно 1.57, то есть, энтропии этого множества символов.

Символ	Вероятность	Code1	Code2
$a_1$	0.49	1	1
$a_2$	0.25	01	01
$a_3$	0.25	010	000
$a_4$	0.01	001	001

Табл. 1.2. Коды переменной длины.

В табл. 1.2 предложен код *Code1*, который присваивает самому часто встречающемуся символу самый короткий код. Если закодировать данный с помощью *Code1*, то среднее число бит на символ будет равно  $1 \times 0.49 + 2 \times 0.25 + 3 \times 0.25 + 3 \times 0.01 = 1.77$ . Это число весьма близко к теоретическому минимуму. Рассмотрим последовательность из 20 символов

$a_1a_3a_2a_1a_3a_3a_4a_2a_1a_1a_2a_2a_1a_1a_3a_1a_1a_2a_3a_1$ ,

в которой четыре символа появляются, примерно, с указанными частотами. Этой строке будет соответствовать кодовая строка кода *Code1* длины 37 бит

1|010|01|1|010|010|001|01|1|1|01|01|1|1|010|1|1|01|010|1|,

которая для удобства разделена черточками. Нам понадобилось 37 битов, чтобы закодировать 20 символов, то есть, в среднем 1.85

бит/символ, что не слишком далеко от вычисленной выше средней величины. (Читатель должен иметь в виду, что эта строка весьма коротка, и для того чтобы получить результат, близкий к теоретическому, необходимо взять входной файл размером несколько тысяч символов).

Однако, если мы теперь попробуем декодировать эту двоичную последовательность, то немедленно обнаружим, что *Code1* совершенно не годен. Первый бит последовательности равен 1, поэтому первым символом может быть только  $a_1$ , так как никакой другой код в таблице для *Code1* не начинается с 1. Следующий бит равен 0, но коды для  $a_2$ ,  $a_3$  и  $a_4$  все начинаются с 0, поэтому декодер должен читать следующий бит. Он равен 1, однако коды для  $a_2$  и  $a_3$  оба имеют в начале 01. Декодер не знает, как ему поступить. То ли декодировать строку как 1|010|01..., то есть,  $a_1a_3a_2\dots$ , то ли как 1|01|001..., то есть  $a_1a_2a_4\dots$  Причем заметим, что дальнейшие биты последовательности уже не помогут исправить положение. Код *Code1* является двусмысленным. В отличие от него, код *Code2* из табл. 1.2 дает при декодировании всегда однозначный результат.

*Code2* имеет одно важное свойство, которое делает его лучше, чем *Code1*, которое называется *свойством префикса*. Это свойство можно сформулировать так: если некоторая последовательность битов выбрана в качестве кода какого-то символа, то ни один код другого символа не должен иметь в начале эту последовательность (не может быть *префиксом*, то есть, приставкой). Раз строка «1» уже выбрана в качестве целого кода для  $a_1$ , то ни один другой код не может начинаться с 1 (то есть, они все должны начинаться на 0). Раз строка «01» является кодом для  $a_2$ , то другие коды не должны начинаться с 01. Вот почему коды для  $a_3$  и  $a_4$  должны начинаться с 00. Естественно, они будут «000» и «001».

Значит, выбирая множество кодов переменной длины, необходимо соблюдать два принципа: (1) следует назначать более короткие коды чаще встречающимся символам, и (2) коды должны удовлетворять свойству префикса. Следуя эти принципам, можно построить короткие, однозначно декодируемые коды, но не обязательно наилучшие (то есть, *самые короткие*) коды. В дополнение к этим принципам необходим алгоритм, который всегда порождает множество самых коротких кодов (которые в среднем имеют наименьшую длину). Исходными данными этого алгоритма должны быть частоты (или вероятности) символов алфавита. К счастью, такой простой алгоритм существует. Он был придуман Давидом Хаффманом и называется его именем. Этот алгоритм будет описан в § 1.4.

Следует отметить, что не только статистические методы компрессии используют коды переменной длины при кодировании индивидуальных символов. Замечательным примером служат арифметические коды, о которых будет рассказано в § 1.7.

### 1.3. Декодирование

Перед тем как описывать статистические методы сжатия данных, важно понять способ взаимодействия кодера и декодера (компрессора и декомпрессора). Предположим, что некоторый файл (с текстом, изображением или еще чем-то) был сжат с помощью кодов переменной длины (префиксных кодов). Для того, чтобы сделать декомпрессию, декодер должен знать префиксный код каждого символа. Эту проблему можно решить тремя способами.

1. Множество префиксных кодов один раз выбрано и используется всеми кодерами и декодерами. Такой метод используется в факсимильной связи (см. § 1.6). Создатели стандарта сжатия для факс-машин выбрали восемь «эталонных» документов, проанализировали их статистические свойства и взяли их за основу при отборе подходящего префиксного кода, представленного в табл. 1.20. Говоря техническом языком, эталонные документы были выбраны для обучения и тренировки алгоритма. Обучение алгоритма является простейшим подходом к статистическому сжатию, и его качество сильно зависит от того, насколько реальные сжимаемые файлы походят на выбранные для тренировки и обучения образцы.

2. Кодер делает свою работу в два прохода. На первом проходе он читает сжимаемый файл и собирает необходимые статистические сведения. На втором проходе происходит собственно сжатие. В перерыве между проходами декодер на основе собранной информации создает наилучший префиксный код именно для этого файла. Такой метод дает замечательные результаты по сжатию, но он, обычно, слишком медленен для практического использования. Кроме того, у него имеется еще один существенный недостаток. Необходимо добавлять таблицу построенных префиксных кодов в сжатый файл, чтобы ее знал декодер. Это ухудшает общую производительность алгоритма. Такой подход в статистическом сжатии принято называть полуадаптивной компрессией.

3. Адаптивное сжатие применяется как кодером, так и декодером. Кодер начинает работать, не зная статистических свойств объекта сжатия. Поэтому первая часть данных сжимается не оптималь-

ным образом, однако, по мере сжатия и сбора статистики, кодер улучшает используемый префиксный код, что приводит к улучшению компрессии. Алгоритм должен быть разработан таким образом, чтобы декодер мог повторить каждый шаг кодера, собрать ту же статистику и улучшить префиксный код в точности тем же способом. Пример адаптивного сжатия рассмотрен в § 1.5.

## 1.4. Кодирование Хаффмана

Кодирование Хаффмана является простым алгоритмом для построения кодов переменной длины, имеющих минимальную среднюю длину. Этот весьма популярный алгоритм служит основой многих компьютерных программ сжатия текстовой и графической информации. Некоторые из них используют непосредственно алгоритм Хаффмана, а другие берут его в качестве одной из ступеней многоуровневого процесса сжатия. Метод Хаффмана [Huffman 52] производит идеальное сжатие (то есть, сжимает данные до их энтропии), если вероятности символов точно равны отрицательным степеням числа 2. Алгоритм начинает строить кодовое дерево снизу вверх, затем скользит вниз по дереву, чтобы построить каждый индивидуальный код справа налево (от самого младшего бита к самому старшему). Начиная с работ Д.Хаффмана 1952 года, этот алгоритм являлся предметом многих исследований. (Последнее утверждение из § 3.8.1 показывает, что наилучший код переменной длины можно иногда получить без этого алгоритма.)

Алгоритм начинается составлением списка символов алфавита в порядке убывания их вероятностей. Затем от корня строится дерево, листьями которого служат эти символы. Это делается по шагам, причем на каждом шаге выбираются два символа с наименьшими вероятностями, добавляются наверх частичного дерева, удаляются из списка и заменяются вспомогательным символом, представляющим эти два символа. Вспомогательному символу приписывается вероятность, равная сумме вероятностей, выбранных на этом шаге символов. Когда список сокращается до одного вспомогательного символа, представляющего весь алфавит, дерево объявляется построенным. Завершается алгоритм спуском по дереву и построением кодов всех символов.

Лучше всего проиллюстрировать этот алгоритм на простом примере. Имеется пять символов с вероятностями, заданными на рис. 1.3а.

Символы объединяются в пары в следующем порядке:

1.  $a_4$  объединяется с  $a_5$ , и оба заменяются комбинированным символом  $a_{45}$  с вероятностью 0.2;
2. осталось четыре символа,  $a_1$  с вероятностью 0.4, а также  $a_2$ ,  $a_3$  и  $a_{45}$  с вероятностями по 0.2. Произвольно выбираем  $a_3$  и  $a_{45}$ , объединяем их и заменяем вспомогательным символом  $a_{345}$  с вероятностью 0.4;
3. теперь имеется три символа  $a_1$ ,  $a_2$  и  $a_{345}$  с вероятностями 0.4, 0.2 и 0.4, соответственно. Выбираем и объединяем символы  $a_2$  и  $a_{345}$  во вспомогательный символ  $a_{2345}$  с вероятностью 0.6;
4. наконец, объединяем два оставшихся символа  $a_1$  и  $a_{2345}$  и заменяем на  $a_{12345}$  с вероятностью 1.

Дерево построено. Оно изображено на рис. 1.3а, «лежа на боку», с корнем справа и пятью листьями слева. Для назначения кодов мы произвольно приписываем бит 1 верхней ветке и бит 0 нижней ветке дерева для каждой пары. В результате получаем следующие коды: 0, 10, 111, 1101 и 1100. Распределение битов по краям – произвольное.

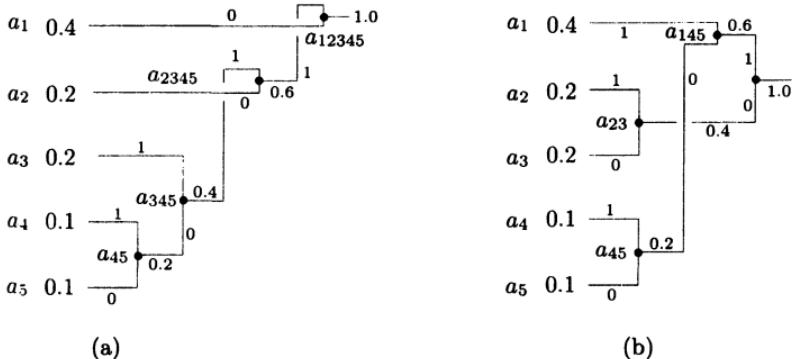


Рис. 1.3. Коды Хаффмана.

Средняя длина этого кода равна  $0.4 \times 1 + 0.2 \times 2 + 0.2 \times 3 + 0.1 \times 4 + 0.1 \times 4 = 2.2$  бит/символ. Очень важно то, что кодов Хаффмана бывает много. Некоторые шаги алгоритма выбирались произвольным образом, поскольку было больше символов с минимальной вероятностью. На рис. 1.3б показано, как можно объединить символы по-другому и получить иной код Хаффмана (11, 01, 00, 101 и 100). Средняя длина равна  $0.4 \times 2 + 0.2 \times 2 + 0.2 \times 2 + 0.1 \times 3 + 0.1 \times 3 = 2.2$  бит/символ как и у предыдущего кода.

**Пример:** Дано 8 символов  $A, B, C, D, E, F, G$  и  $H$  с вероятностями  $1/30, 1/30, 1/30, 2/30, 3/30, 5/30, 5/30$  и  $12/30$ . На рис. 1.4a,b,c изображены три дерева кодов Хаффмана высоты 5 и 6 для этого алфавита. Средняя длина этих кодов (в битах на символ) равна

$$(5 + 5 + 5 + 5 \cdot 2 + 3 \cdot 3 + 3 \cdot 5 + 3 \cdot 5 + 12) / 30 = 76/30,$$

$$(5 + 5 + 4 + 4 \cdot 2 + 4 \cdot 3 + 3 \cdot 5 + 3 \cdot 5 + 12) / 30 = 76/30,$$

$$(6 + 6 + 5 + 4 \cdot 2 + 3 \cdot 3 + 3 \cdot 5 + 3 \cdot 5 + 12) / 30 = 76/30.$$

**Пример:** На рис. 1.4d показано другое дерево высоты 4 для восьми символов из предыдущего примера. Следующий анализ показывает, что соответствующий ему код переменной длины – плохой, хотя его длина меньше 4.

(Анализ.) После объединения символов  $A, B, C, D, E, F$  и  $G$  остаются символы  $ABEF$  (с вероятностью  $10/30$ ),  $CDG$  (с вероятностью  $8/30$ ) и  $H$  (с вероятностью  $12/30$ ). Символы  $ABEF$  и  $CDG$  имеют наименьшую вероятность, поэтому их необходимо было слить в один, но вместо этого были объединены символы  $CDG$  и  $H$ . Полученное дерево не является деревом Хаффмана.

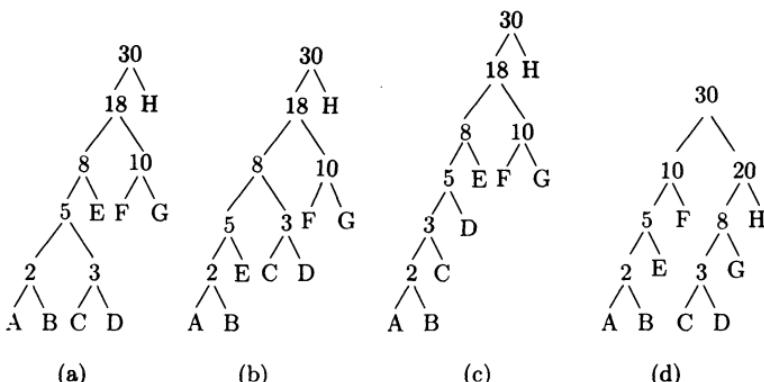


Рис. 1.4. Три дерева Хаффмана для восьми символов.

Таким образом, некоторый произвол в построении дерева позволяет получать разные коды Хаффмана с одинаковой средней длиной. Напрашивается вопрос: «Какой код Хаффмана, построенный для данного алфавита, является наилучшим?» Ответ будет простым, хотя и неочевидным: лучшим будет код с наименьшей дисперсией.

Дисперсия показывает насколько сильно отклоняются длины индивидуальных кодов от их средней величины (это понятие разъяс-

няется в любом учебнике по статистике). Дисперсия кода 1.3а равна  $0.4(1-2.2)^2+0.2(2-2.2)^2+0.2(3-2.2)^2+0.1(4-2.2)^2+0.1(4-2.2)^2=1.36$ , а для кода 1.3б

$$0.4(2-2.2)^2+0.2(2-2.2)^2+0.2(2-2.2)^2+0.1(3-2.2)^2+0.1(3-2.2)^2=0.16.$$

Код 1.3б является более предпочтительным (это будет объяснено ниже). Внимательный взгляд на деревья показывает, как выбрать одно, нужное нам. На дереве рис. 1.3а символ  $a_{45}$  сливается с символом  $a_3$ , в то время как на рис. 1.3б он сливается с  $a_1$ . Правило будет такое: когда на дереве имеется более двух узлов с наименьшей вероятностью, следует объединять символы с наибольшей и наименьшей вероятностью; это сокращает общую дисперсию кода.

Если кодер просто записывает сжатый файл на диск, то дисперсия кода не имеет значения. Коды Хаффмана с малой дисперсией более предпочтительны только в случае, если кодер будет передавать этот сжатый файл по линиям связи. В этом случае, код с большой дисперсией заставляет кодер генерировать биты с переменной скоростью. Обычно данные передаются по каналам связи с постоянной скоростью, поэтому кодер будет использовать буфер. Биты сжатого файла помещаются в буфер по мере их генерации и подаются в канал с постоянной скоростью для передачи. Легко видеть, что код с нулевой дисперсией будет подаваться в буфер с постоянной скоростью, поэтому понадобится короткий буфер, а большая дисперсия кода потребует использование длинного буфера.

Следующее утверждение можно иногда найти в литературе по сжатию информации: *длина кода Хаффмана символа  $a_i$  с вероятностью  $P_i$  всегда не превосходит  $[-\log_2 P_i]$* . На самом деле, не смотря на справедливость этого утверждения во многих примерах, в общем случае оно не верно. Я весьма признателен Гаю Блелоку, который указал мне на это обстоятельство и сообщил пример кода, приведенного в табл. 1.5. Во второй строке этой таблицы стоит символ с кодом длины 3 бита, в то время как  $[-\log_2 0.3] = [1.737] = 2$ .

$P_i$	Код	$-\log_2 P_i$	$[-\log_2 P_i]$
0.01	000	6.644	7
*	0.30	001	1.737
0.34	01	1.556	2
0.35	1	1.515	2

Табл. 1.5. Пример кода Хаффмана.

Длина кода символа  $a_i$ , конечно, зависит от его вероятности  $P_i$ . Однако она также неявно зависит от размера алфавита. В большом алфавите вероятности символов малы, поэтому коды Хаффмана имеют большую длину. В маленьком алфавите наблюдается обратная картина. Интуитивно это понятно, поскольку для малого алфавита требуется всего несколько кодов, поэтому все они коротки, а большому алфавиту необходимо много кодов и некоторые из них должны быть длинными.

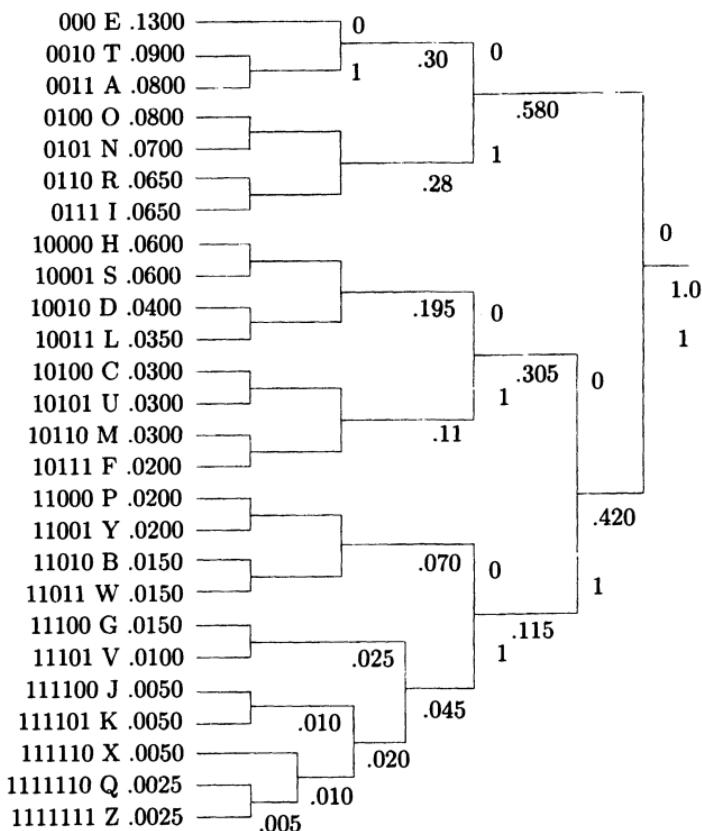


Рис. 1.6. Код Хаффмана для английского алфавита.

На рис. 1.6 показан код Хаффмана для всех 26 букв английского алфавита.

Случай алфавита, в котором символы равновероятны, особенно интересен. На рис. 1.7 приведены коды Хаффмана для алфавита с

5, 6, 7 и 8 равновероятными символами. Если размер алфавита  $n$  является степенью 2, то получаются просто коды фиксированной длины. В других случаях коды весьма близки к кодам с фиксированной длиной. Это означает, что использование кодов переменной длины не дает никаких преимуществ. В табл. 1.8 приведены коды, их средние длины и дисперсии.

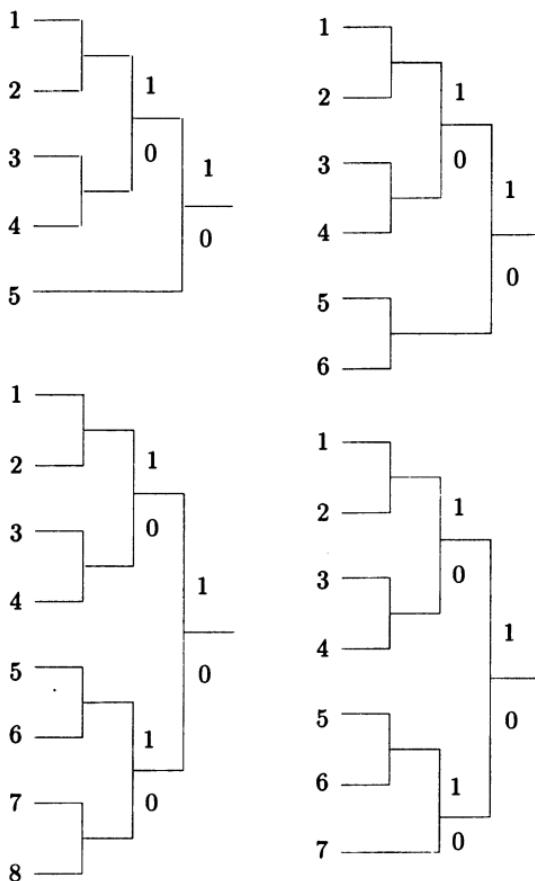


Рис. 1.7. Коды Хаффмана с равными вероятностями.

Тот факт, что данные с равновероятными символами не сжимаются методом Хаффмана может означать, что строки таких символов являются совершенно случайными. Однако, есть примеры строк, в которых все символы равновероятны, но не являются случайными, и их можно сжимать. Хорошим примером является последо-

вательность  $a_1a_1 \dots a_1a_2a_2 \dots a_2a_3a_3 \dots$ , в которой каждый символ встречается длинными сериями. Такую строку можно сжать методом RLE, но не методом Хаффмана. (Буквосочетание RLE означает «run-length encoding», т.е. «кодирование длин серий». Этот простой метод сам по себе мало эффективен, но его можно использовать в алгоритмах сжатия со многими этапами, см. [Salomon, 2000].)

$n$	$p$	$a_1$	$a_2$	$a_3$	$a_4$	$a_5$	$a_6$	$a_7$	$a_8$	Ср.длина	Дисперсия
5	0.200	111	110	101	100	0				2.6	0.64
6	0.167	111	110	101	100	01	00			2.672	0.2227
7	0.143	111	110	101	100	011	010	00		2.86	0.1226
8	0.125	111	110	101	100	011	010	001	000	3	0

Табл. 1.8. Коды Хаффмана для 5–8 символов.

Заметим, что метод Хаффмана не работает в случае двухсимвольного алфавита. В таком алфавите одному символу придется присвоить код 0, а другому – 1. Метод Хаффмана не может присвоить одному символу код короче одного бита. Если исходные данные (источник) состоят из индивидуальных битов, как в случае двухуровневого (монохромного) изображения, то возможно представление нескольких бит (4 или 8) в виде одного символа нового недвичного алфавита (из 16 или 256 символов). Проблема при таком подходе заключается в том, что исходные битовые данные могли иметь определенные статистические корреляционные свойства, которые могли быть утеряны при объединении битов в символы. При сканировании монохромного рисунка или диаграммы по строкам пиксели будут чаще встречаться длинными последовательностями одного цвета, а не быстрым чередованием черных и белых. В итоге получится файл, начинающийся с 1 или 0 (с вероятностью 1/2). За нулем с большой вероятностью следует нуль, а за единицей – единица. На рис. 1.9 изображен конечный автомат, иллюстрирующий эту ситуацию. Если биты объединять в группы, скажем, по 8 разрядов, то биты внутри группы будут коррелированы, но сами группы не будут иметь корреляцию с исходными пикселями. Если входной файл содержит, например, две соседние группы 00011100 и 00001110, то они будут кодироваться независимо, игнорируя корреляцию последних нулей первой группы и начальных нулей второй. Выбор длинных групп улучшает положение, но увеличивает число возможных групп, что влечет за собой увеличение памяти для хранения таблицы кодов и удлиняет время создания этой таблицы. (Напомним, что

если группа длины  $s$  увеличивается до длины  $s + n$ , то число групп растет экспоненциально с  $2^s$  до  $2^{s+n}$ .)

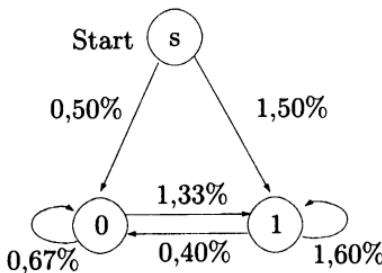


Рис. 1.9. Конечный автомат.

Более сложный подход к сжатию изображений с помощью кодов Хаффмана основан на создании нескольких полных множеств кодов Хаффмана. Например, если длина группы равна 8 бит, то порождается несколько семейств кодов размера 256. Когда необходимо закодировать символ  $S$ , выбирается одно семейство кодов, и  $S$  кодируется из этого семейства.

#### Давид Хаффман (1925–1999)

Давид начал свою научную карьеру студентом в Массачусетском технологическом институте (MIT), где построил свои коды в начале пятидесятых годов прошлого века.

Он поступил на факультет MIT в 1953 году. В 1967 году Хаффман перешел в университет Санта Круз на факультет компьютерных наук. Он играл заметную роль в развитии академической программы факультета и в подборе преподавателей, возглавляя его с 1970 по 1973 года. Выйдя на пенсию в 1994 году, Давид Хаффман продолжил свою научную деятельность в качестве заслуженного профессора в отставке, читая курсы по теории информации и по анализу сигналов. Он умер в 1999 году в возрасте 74 лет.

Хаффман сделал важный вклад во многих областях науки, включая теорию информации и теорию кодирования. Он много занимался прикладными задачами, например, проектированием сигналов для радаров, разработкой процедур для асинхронных цепей и другими коммуникационными проблемами. Побочным результатом его работы по математическим свойствам поверхностей «нулевой кривизны» стало развитие им оригинальной техники свертывания из бумаги необычных скульптурных форм [Grafica 96].

**Пример:** Представим себе изображение из 8-и битовых пикселов, в котором половина пикселов равна 127, а другая половина имеет значение 128. Проанализируем эффективность метода RLE для индивидуальных битовых областей по сравнению с кодированием Хаффмана.

(Анализ.) Двоичная запись 127 равна 0111111, а 128 – 10000000. Половина пикселов поверхности будет нулем, а вторая половина – единицей. В самом плохом случае область будет походить на шахматную доску, то есть, иметь много серий длины 1. В этом случае каждая серия требует кода в 1 бит, что ведет к одному кодовому биту на пиксел на область, или 8 кодовых битов на пиксел для всего изображения. Это приводит к полному отсутствию сжатия. А коду Хаффмана для такого изображения понадобится всего два кода (поскольку имеется всего два разных пикселя), и они могут быть длины 1. Это приводит к одному кодовому биту на пиксел, то есть к восьмикратному сжатию.

#### 1.4.1. Декодирование Хаффмана

Перед тем как начать сжатие потока данных, компрессор (кодер) должен построить коды. Это делается с помощью вероятностей (или частот появления) символов. Вероятности и частоты следует записать в сжатый файл для того, чтобы декомпрессор (декодер) Хаффмана мог сделать декомпрессию данных (см. различные подходы в §§ 1.3 и 1.5). Это легко сделать, так как частоты являются целыми числами, а вероятности также представимы целыми числами. Обычно это приводит к добавлению нескольких сотен байтов в сжатый файл. Можно, конечно, записать в сжатый файл сами коды, однако это вносит дополнительные трудности, поскольку коды имеют разные длины. Еще можно записывать в файл само дерево Хаффмана, но это потребует большего объема, чем простая запись частот.

В любом случае, декодер должен прочесть начало файла и построить дерево Хаффмана для алфавита. Только после этого он может читать и декодировать весь файл. Алгоритм декодирования очень прост. Следует начать с корня и прочитать первый бит сжатого файла. Если это нуль, следует двигаться по нижней ветке дерева; если это единица, то двигаться надо по верхней ветке дерева. Далее читается второй бит и происходит движение по следующей ветке по направлению к листьям. Когда декодер достигнет листа дерева, он узнает код первого несжатого символа (обычно это символ ASCII). Процедура повторяется для следующего бита, начиная опять из корня дерева.

Описанная процедура проиллюстрирована на рис. 1.10 для алфавита из 5 символов. Входная строка  $a_4a_2a_5a_1$  кодируется последовательностью 1001100111. Декодер начинает с корня, читает первый бит «1» и идет вверх. Второй бит «0» направляет его вниз. То же самое делает третий бит. Это приводит декодер к листу  $a_4$ . Получен



первый несжатый символ. Декодер возвращается в корень и читает 110, движется вверх, вверх и вниз и получает символ  $a_2$ , и так далее.

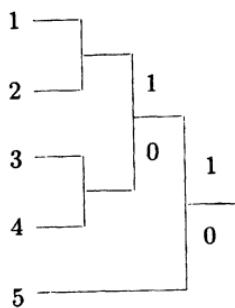


Рис. 1.10. Коды Хаффмана с равными вероятностями.

#### 1.4.2. Средняя длина кода

На рис. 1.13а представлено множество из пяти символов с их вероятностями, а также типичное дерево Хаффмана. Символ  $A$  возникает в 55% случаев и ему присваивается однобитовый код, что вносит вклад  $0.55 \times 1$  в среднюю длину. Символ  $E$  возникает лишь в 2% случаев. Ему присваивается код длины 4, и его вклад равен  $0.02 \times 4 = 0.08$ . Тогда средняя длина кода равна

$$0.55 \times 1 + 0.25 \times 2 + 0.15 \times 3 + 0.03 \times 4 + 0.02 \times 4 = 1.7 \text{ бит на символ.}$$

Удивительно, но тот же результат получится, если сложить значения вероятностей четырех внутренних узлов кодового дерева:  $0.05 + 0.2 + 0.45 + 1 = 1.7$ . Это наблюдение дает простой способ вычисления средней длины для кодов Хаффмана без использования операции умножения. Надо просто сложить значения внутренних узлов дерева. Табл. 1.11 иллюстрирует, почему этот метод будет работать.

$$\begin{aligned}
 0.05 &= &= 0.02 + 0.03 \\
 0.20 &= 0.05 + 0.15 &= 0.02 + 0.03 + 0.15 \\
 0.45 &= 0.20 + 0.25 &= 0.02 + 0.03 + 0.15 + 0.25 \\
 1.00 &= 0.45 + 0.55 &= 0.02 + 0.03 + 0.15 + 0.25 + 0.55
 \end{aligned}$$

Табл. 1.11. Состав узлов.

(В таблице внутренние узлы выделены.) В левом столбце выпи-саны величины всех внутренних узлов. В правых столбцах показано,

как величины складываются из величин предыдущих узлов и из величин листьев. Если сложить числа в левом столбце, то получится 1.7, а складывая числа в остальных столбцах, убеждаемся, что это число 1.7 есть сумма четырех 0.02, четырех 0.03, трех 0.15, двух 0.25 и одного 0.55.

Это рассуждение годится и в общем случае. Легко видеть, что в дереве типа Хаффмана (т.е., дереве, в котором каждый узел есть сумма своих потомков) взвешенная сумма листьев, где вес листа – это его расстояние до корня, равна сумме всех внутренних узлов. (Это свойство было мне сообщено Джоном Мотилом.)

<b>0.05</b>	=	$= 0.02 + 0.03 + \dots$
$a_1$	=	$0.05 + \dots = 0.02 + 0.03 + \dots$
$a_2$	=	$a_1 + \dots = 0.02 + 0.03 + \dots$
$\vdots$	=	
$a_{d-2}$	=	$a_{d-3} + \dots = 0.02 + 0.03 + \dots$
<b>1.0</b>	=	$a_{d-2} + \dots = 0.02 + 0.03 + \dots$

Табл. 1.12. Состав узлов.

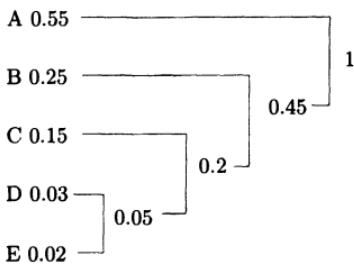
На рис. 1.13b показано такое дерево, где предполагается, что два листа 0.02 и 0.03 имеют коды Хаффмана длины  $d$ . Внутри дерева эти листья являются потомками внутреннего узла 0.05, который, в свою очередь, связан с корнем с помощью  $d-2$  внутренних узлов от  $a_1$  до  $a_{d-2}$ . Табл. 1.12 состоит из  $d$  строк и показывает, что две величины 0.02 и 0.03 включаются в разные внутренние узлы ровно  $d$  раз. Сложение величин всех внутренних узлов дает вклад от этих двух листьев равный  $0.02d + 0.03d$ . Поскольку эти листья выбраны произвольно, ясно, что эта сумма включает в себя аналогичный вклад от всех остальных узлов, то есть, равна средней длине кода. Эта число также равно сумме левого столбца или сумме всех внутренних узлов, которая и определит среднюю длину кода.

Отметим, что в доказательстве нигде не предполагалось, что дерево – двоичное. Поэтому это свойство выполнено для любого дерева, в котором узлы являются суммой своих потомков.

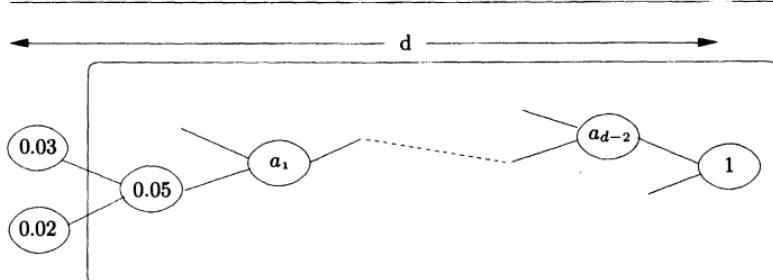
## 1.5. Адаптивные коды Хаффмана

Метод Хаффмана предполагает, что частоты символов алфавита известны декодеру. На практике это бывает крайне редко. Одно возможное решение для компрессора — это прочитать исходные дан-

ные два раза. В первый раз, чтобы вычислить частоты, а во второй раз, чтобы сделать сжатие. В промежутке компрессор строит дерево Хаффмана. Такой метод называется полуадаптивным (см. § 1.3), и он работает медленно для практического использования. На практике применяется метод адаптивного (или динамического) кодирования Хаффмана. Этот метод лежит в основе программы `compress` операционной системы UNIX.



(a)



(b)

Рис. 1.13. Деревья для кодов Хаффмана.

Более подробно про адаптивный метод, излагаемый ниже, можно прочитать в [Lelewer, Hirschberg 87], [Knuth 85] и [Vitter 87].

Основная идея заключается в том, что и компрессор, и декомпрессор начинают с пустого дерева Хаффмана, а потом модифицируют его по мере чтения и обработки символов (в случае компрессора обработка означает сжатие, а для декомпрессора – декомпрессию). И компрессор и декомпрессор должны модифицировать дерево совершенно одинаково, чтобы все время использовать один и тот же код, который может изменяться по ходу процесса. Будем говорить, что компрессор и декомпрессор *синхронизованы*, если их

работа жестко согласована (хотя и не обязательно выполняется в одно и то же время). Слово *зеркально*, возможно, лучше обозначает суть их работы. Декодер зеркально повторяет операции кодера.

В начале кодер строит пустое дерева Хаффмана. Никакому символу код еще не присвоен. Первый входной символ просто записывается в выходной файл в несжатой форме. Затем этот символ помещается на дерево и ему присваивается код. Если он встретится в следующий раз, его текущий код будет записан в файл, а его частота будет увеличена на единицу. Поскольку эта операция модифицировала дерево, его надо проверить, является ли оно деревом Хаффмана (дающее наилучшие коды). Если нет, то это влечет за собой перестройку дерева и перемену кодов (§ 1.5.2).

Декомпрессор зеркально повторяет это действие. Когда он читает несжатый символ, он добавляет его на дерево и присваивает ему код. Когда он читает сжатый код (переменной длины), он использует текущее дерево, чтобы определить, какой символ отвечает данному коду, после чего модифицирует дерево тем же образом, что и кодер.

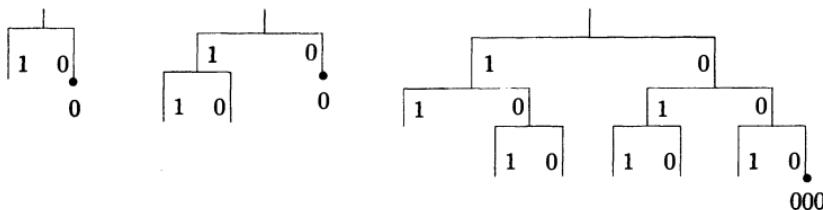


Рис. 1.14. Коды *ecs*.

Однако, имеется одно тонкое место. Декодер должен знать является ли данный образчик просто несжатым символом (обычно, это 8-битный код ASCII, однако, см. § 1.5.1) или же это код переменной длины. Для преодоления двусмысленности, каждому несжатому символу будет предшествовать специальный *esc* (escape) код переменной длины. Когда декомпрессор читает этот код, он точно знает, что за ним следуют 8 бит кода ASCII, который впервые появился на входе.

Беспокойство вызывает то, что *esc* код не должен быть переменным кодом, используемым для символов алфавита. Поскольку это коды все время претерпевают изменения, то и код для *esc* следует также модифицировать. Естественный путь – это добавить к

дереву еще один *пустой* лист с постоянно нулевой частотой, чтобы ему все время присваивалась ветвь из одних нулей. Поскольку этот лист будет все время присутствовать на дереве, ему все время будет присваиваться код переменной длины. Это и будет код *esc*, предшествующий каждому новому несжатому символу. Дерево будет все время модифицироваться, будет изменяться положение на нем пустого листа и его кода, но сам этот код будет идентифицировать каждый новый несжатый символ в сжатом файле. На рис. 1.14 показано движение этого *esc* кода по мере роста дерева.

Этот метод также используется в известном протоколе V.32 передачи данных по модему со скоростью 14400 бод.

### 1.5.1. Несжатые коды

Если сжимаемые символы являются кодами ASCII, то им можно просто присвоить свои значения для представления в несжатом виде. В общем случае, когда алфавит имеет произвольный размер, несжатые коды двух разных размеров можно также легко построить. Рассмотрим, например, алфавит размера  $n = 24$ . Первым 16 символам можно присвоить числа от 0 до 15 в их двоичном представлении. Эти символы потребуют только 4 бита, но мы закодируем их пятью битами. Символам с номерами от 17 до 24 присвоим числа  $17 - 16 - 1 = 0$ ,  $18 - 16 - 1 = 1$ , и до  $24 - 16 - 1 = 7$  в двоичном представлении из 4 бит. Итак, мы получим шестнадцать 5-битовых кода 00000, 00001,...,01111, за которыми следуют восемь 4-битовых кода 0000, 0001,...,0111.

В общем случае, если имеется алфавит  $a_1, a_2, \dots, a_n$ , состоящий из  $n$  символов, выбираются такие числа  $m$  и  $r$ , что  $2^m \leq n < 2^{m+1}$  и  $r = n - 2^m$ . Первые  $2^m$  символов кодируются как  $(m+1)$ -битовые числа от 0 до  $2^m - 1$ , а остальные символы кодируются  $m$ -битовыми последовательностями так, что код символа  $a_k$  равен  $k - 2^m - 1$ . Такие коды называются *синфазными двоичными кодами*.

*Гибель одного человека – это трагедия, а смерть миллионов людей – это статистика.*

— Иосиф Сталин

### 1.5.2. Модификация дерева

Основная идея заключается в проверке дерева при появлении каждого нового входного символа. Если дерево не является деревом Хаффмана, его следует подправить. Рис. 1.15 дает представление

о том, как это делается. Дерево на рис. 1.15а содержит пять символов А, В, С, Д и Е. Для всех символов указаны их текущие частоты (в круглых скобках). Свойство Хаффмана означает, что при изучении дерева по всем уровням слева направо и снизу вверх (от листьев к корню) частоты будут упорядочены по возрастанию (неубыванию). Таким образом, нижний левый узел (А) имеет наименьшую частоту, а верхний правый (корень) имеет наибольшую частоту. Это свойство принято называть *свойством соперничества*.

Причина этого свойства заключается в том, что символы с большими частотами должны иметь более короткие коды и, следовательно, находиться на дереве на более высоком уровне. Требование для частот быть упорядоченными на одном уровне устанавливается для определенности. В принципе, без него можно обойтись, но оно упрощает процесс построения дерева.

Приведем последовательность операций при модификации дерева. Цикл начинается в текущем узле (соответствующем новому входному символу). Этот узел будет листом, который мы обозначим  $X$ , а его частота пусть будет  $F$ . На каждой следующей итерации цикла требуется сделать три действия:

1. сравнить  $X$  с его ближайшими соседями на дереве (справа и сверху). Если непосредственный сосед имеет частоту  $F+1$  или выше, то узлы остаются упорядоченными и ничего делать не надо. В противном случае, некоторый сосед имеет такую же или меньшую частоту, чем  $X$ . В этом случае следует поменять местами  $X$  и последний узел в этой группе (за исключением того, что  $X$  не надо менять с его родителем);
2. увеличить частоту  $X$  с  $F$  до  $F+1$ . Увеличить на единицу частоту всех его родителей;
3. если  $X$  является корнем, то цикл останавливается; в противном случае он повторяется для узла, являющегося родителем  $X$ .

На рис. 1.15б показано дерево после увеличения частоты узла А с 1 до 2. Легко проследить, как описанные выше три действия влияют на увеличение частоты всех предков А. Места узлов не меняются в этом простом случае, так как частота узла А не превысила частоту его ближайшего соседа справа В.

На рис. 1.15с показано, что произойдет при следующем увеличении частоты А с 2 до 3. Три узла, следующие за А, а именно, В, С и Д, имели частоту 2, поэтому А переставлен с Д. После чего частоты всех предков А увеличены на 1, и каждый сравнен со своими соседями, но больше на дереве никого переставлять не надо.

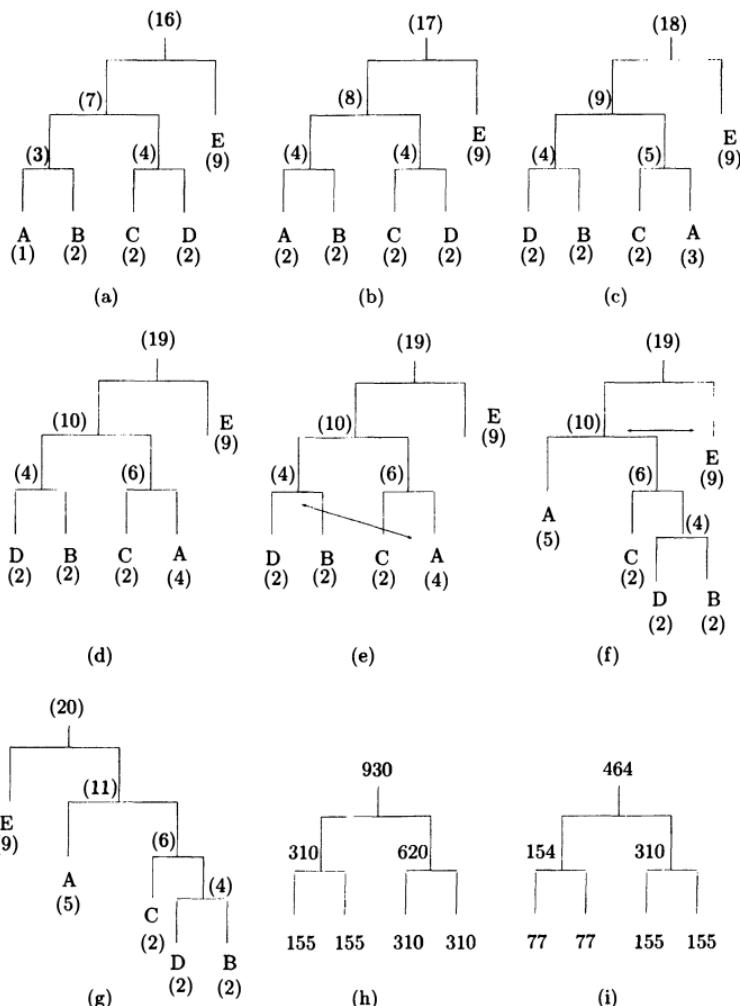


Рис. 1.15. Обновление дерева Хаффмана.

Рис. 1.15d изображает дерево после увеличения частоты A до 4. Поскольку узел A является текущим, его частота (равная пока еще 3) сравнивается с частотой соседа сверху (4), и дерево не меняется. Частота A увеличивается на единицу вместе с частотами всех потомков.

На рис. 1.15e узел A опять является текущим. Его частота (4) равна частоте соседа сверху, поэтому их следует поменять местами.

Это сделано на рис. 1.15f, где частота А уже равна 5. Следующий шаг проверяет родителя А с частотой 10. Его следует переставить с его соседом справа Е, у которого частота 9. В итоге получается конечное дерево, изображенное на рис. 1.15g.

*Тот, кто хочет отведать плод, должен влезть на дерево.*  
— Томас Фуллер

### 1.5.3. Переполнение счетчика

Счетчики частот символов сохраняются на дереве в виде чисел с фиксированной разрядностью. Поля разрядов могут переполниться. Число без знака из 16 двоичных разрядов может накапливать счетчик до числа  $2^{16} - 1 = 65535$ . Простейшее решение этой проблемы заключается в наблюдении за ростом счетчика в корне дерева, и, когда он достигает максимального значения, следует сделать изменение масштаба всех счетчиков путем их целочисленного деления на 2. На практике это обычно делается делением счетчиков листьев с последующим вычислением счетчиков всех внутренних узлов дерева. Каждый внутренний узел равен сумме своих потомков. Однако при таком методе целочисленного деления понижается точность вычислений, что может привести к нарушению свойства соперничества узлов дерева.

Простейший пример приведен на рис. 1.15h. После уполовинивания счетчиков листьев, три внутренних узла пересчитаны, как показано на рис. 1.15i. Полученное дерево, однако, не удовлетворяет свойству Хаффмана, поскольку счетчики перестали быть упорядоченными. Поэтому после каждой смены масштаба требуется перестройка дерева, которая, по счастью, будет делаться не слишком часто. Поэтому компьютерные программы общего назначения, использующие метод сжатия Хаффмана должны иметь многоразрядные счетчики, чтобы их переполнение случалось не слишком часто. Счетчики разрядности в 4 байта будут переполняться при значении равном  $2^{32} - 1 \approx 4.3 \times 10^9$ .

Стоит отметить, что после смены масштаба счетчиков, новые поступившие для сжатия символы будут влиять на счетчики сильнее, чем сжатые ранее до смены масштаба (примерно с весом 2). Это не критично, поскольку из опыта известно, что вероятность появления символа сильнее зависит от непосредственно предшествующих символов, чем от тех, которые поступили в более отдаленном прошлом.

### 1.5.4. Кодовое переполнение

Более серьезную проблему может создать кодовое переполнение. Это случится, если на дерево поступит слишком много символов и оно станет слишком высоким. Сами коды не хранятся на дереве, так как они меняются все время, и компрессор должен вычислять код символа  $X$  каждый раз заново при его появлении. Вот что при этом происходит:

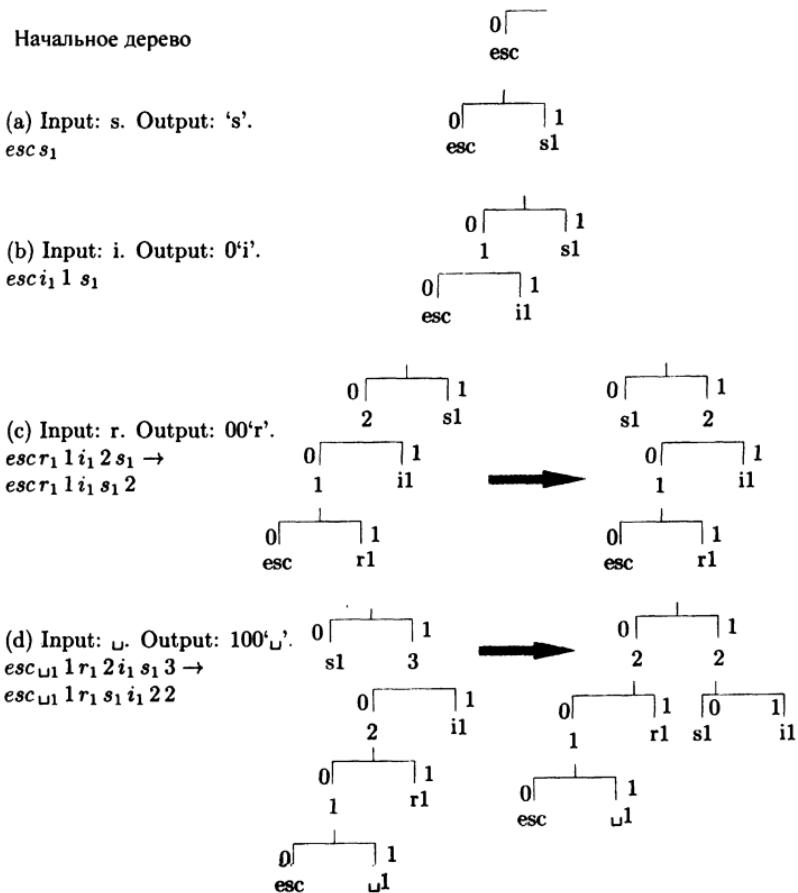
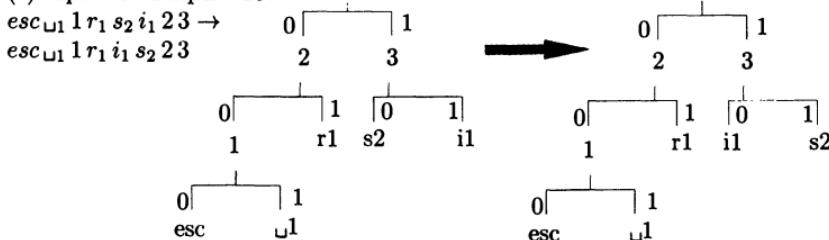


Рис. 1.16. Адаптивный код Хаффмана, пример: часть I.

1. Кодер должен обнаружить символ  $X$  на дереве. Дерево следует реализовать в виде массива структур, состоящих из узлов. Поиск в этом массиве будет линейным.

2. Если  $X$  не найден, то вырабатывается код  $esc$ , за которым следует несжатый код символа. Затем символ  $X$  добавляется на дерево.

(e) Input:  $s$ . Output: 10.



(f) Input:  $i$ . Output: 10.



(g) Input:  $d$ . Output: 000'd'.

$esc d_1 1 \sqcup_1 2 r_1 i_2 s_2 3 4 \rightarrow$   
 $esc d_1 1 \sqcup_1 r_1 2 i_2 s_2 3 4$

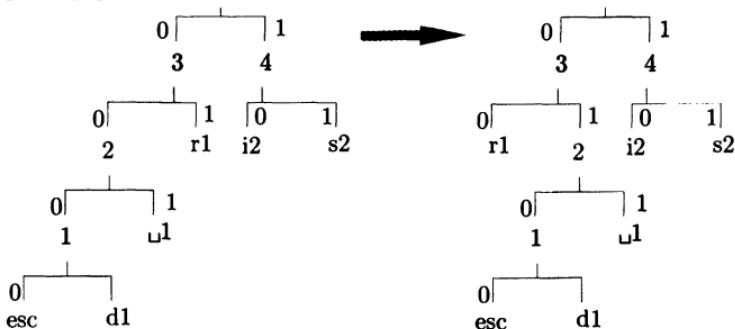


Рис. 1.16. Адаптивный код Хаффмана, пример: часть II.

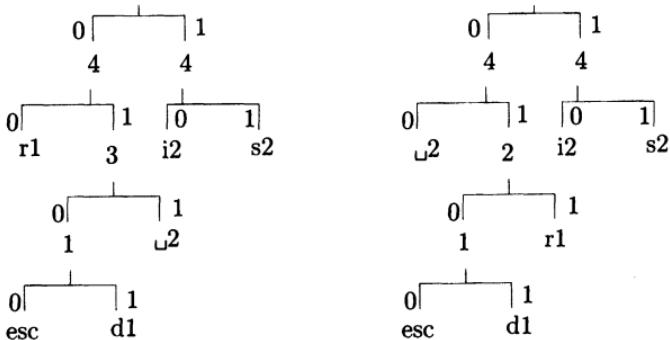
3. Если  $X$  найден, то компрессор движется от узла  $X$  назад к корню, выстраивая его код бит за битом. На каждом шаге, двигаясь влево от потомка к родителю он добавляет к коду «1», а двигаясь вправо, добавляет «0» (или наоборот, но это должно быть четко определено, поскольку декодер будет делать то же самое). Эту последовательность битов надо где-то хранить, поскольку она будет записываться

в обратном порядке. Когда дерево станет высоким, коды тоже удлиняются. Если они накапливаются в виде 16-ти разрядного целого, то коды длиннее 16 бит вызовут сбой программы.

(h) Input:  $\text{u}$ . Output: 011.

$\text{esc } d_1 \ 1 \ \text{u}_2 \ r_1 \ 3 \ i_2 \ s_2 \ 4 \ 4 \rightarrow$

$\text{esc } d_1 \ 1 \ r_1 \ \text{u}_2 \ 2 \ i_2 \ s_2 \ 4 \ 4$



(i) Input: i. Output: 10.

$\text{esc } d_1 \ 1 \ r_1 \ \text{u}_2 \ 2 \ i_3 \ s_2 \ 4 \ 5 \rightarrow$

$\text{esc } d_1 \ 1 \ r_1 \ \text{u}_2 \ 2 \ s_2 \ i_3 \ 4 \ 5$

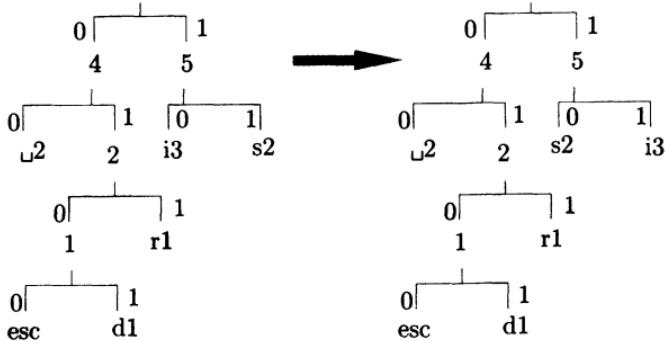
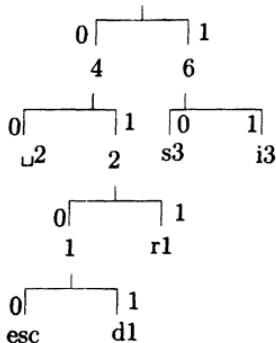


Рис. 1.16. Адаптивный код Хаффмана, пример: часть III.

Правильное решение может заключаться в накоплении битов кода в связанным списке, к которому можно добавлять новые узлы. Тогда ограничением будет служить лишь объем доступной памяти. Это общее решение, но медленное. Другое решение состоит в накапливании кода в длинной целой переменной (например из 50 разрядов). При этом следует документировать программу ограничением в 50 бит для максимальной длины кодов.

(j) Input: s. Output: 10.  
 $esc d_1 1 r_{1+2} 2 s_3 i_3 4 6$



(k) Input:  $\sqcup$ . Output: 00.

$\text{esc} d_1 1 r_1 \sqcup_3 2 s_3 i_3 5 6 \rightarrow$   
 $\text{esc} d_1 1 r_1 2 \sqcup_3 s_3 i_3 5 6$

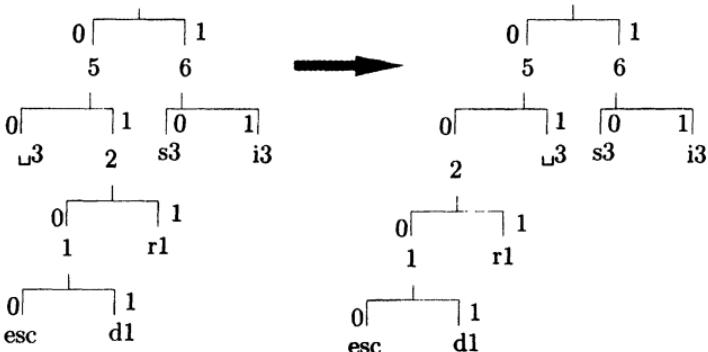


Рис. 1.16. Адаптивный код Хаффмана, пример: часть IV.

К счастью, эта проблема не оказывает влияние на процесс декодирования. Декодер читает сжатый код бит за битом и использует каждый бит, чтобы идти шаг за шагом влево или вправо вниз по дереву, пока он не достигнет листа с символом. Если листом служит *esc* код, декодер прочтет несжатый символ из сжатого файла (и добавит этот символ на дерево). В противном случае несжатый символ будет на этом листе дерева.

**Пример:** Применим адаптивное кодирование Хаффмана к строке «sir.sid.is». Для каждого входного символа найдены код на выходе, дерево после добавления этого символа, дерево после модификации (если необходимо), а также пройденные узлы слева направо снизу вверх.

Рис. 1.16 показывает начальное дерево и как оно изменялось за 11 шагов от (a) до (k). Обратите внимание на то, как символ *esc* получает все время разные коды, и как разные символы перемещаются по дереву, меняя свои коды. Код 10, например, кодирует символ «i» на шагах (f) и (i), этот же код присваивается символу «s» на шагах (e) и (j). Пустой символ имеет код 011 на шаге (h), но он же имеет код 00 на шаге (k).

На выходе кодера будет строка «s0i00r100\_1010000d011101000». Общее число битов равно  $5 \times 8 + 22 = 62$ . Коэффициент сжатия равен  $62/88 \approx 0.7$ .

### 1.5.5. Вариант алгоритма

Этот вариант адаптивного кодирования Хаффмана очень прост, но менее эффективен. Его идея заключается в построении множества из  $n$  кодов переменной длины на основе равных вероятностей и случайном присвоении этих кодов  $n$  символам. После чего смена кодов делается «на лету» по мере считывания и сжатия символов. Метод не слишком производителен, поскольку коды не основаны на реальных вероятностях символов входного файла. Однако его проще реализовать, и он будет работать быстрее описанного выше алгоритма, поскольку переставляет строки таблицы быстрее, чем первый алгоритм перестраивает дерево при изменении частот символов.

Симв.	Сч-к	Код									
$a_1$	0	0	$a_2$	1	0	$a_2$	1	0	$a_4$	2	0
$a_2$	0	10	$a_1$	0	10	$a_4$	1	10	$a_2$	1	10
$a_3$	0	110									
$a_4$	0	111	$a_4$	0	111	$a_1$	0	111	$a_1$	0	111

(a)

(b)

(c)

(d)

Рис. 1.17. Четыре шага алгоритма типа Хаффмана.

Основная структура данных – это таблица размера  $n \times 3$ , в которой три столбца хранят, соответственно, имена  $n$  символов, частоты символов и их коды. Таблица все время упорядочена по второму столбцу. Когда счетчики частот во втором столбце изменяются, строки переставляются, но перемещаются только первый и второй столбцы. Коды в третьем столбце не меняются. На рис. 1.17 показаны примеры для четырех символов и поведение метода при сжатии строки « $a_3a_4a_4$ ».

На рис. 1.17а изображено начальное состояние. После считывания символа  $a_2$  его счетчик увеличивается, и поскольку он теперь наибольший, строки 1 и 2 меняются местами (рис. 1.17б). Далее считывается второй символ  $a_4$ , его счетчик увеличивается, и строки 2 и 4 переставляются (рис. 1.17в). Наконец, после считывания третьего символа  $a_4$ , его счетчик становится наибольшим, что приводит к перестановке строк 1 и 2 (рис. 1.17г).

В этом алгоритме только одно место может вызвать проблему – это переполнение счетчиков. Если переменные счетчиков имеют разрядность  $k$  бит, их максимальное значение равно  $2^k - 1$ . Поэтому наступит переполнение после  $2^k$ -го увеличения. Это может случиться, если размер сжимаемого файла заранее не известен, что бывает часто. К счастью, нам не надо знать точные значения счетчиков. Нам нужен лишь их порядок, поэтому эту проблему переполнения легко разрешить.

Можно, например, считать входные символы, и после  $2^k - 1$  символа делать целочисленное деление счетчиков на 2 (или сдвигать их содержимое на одну позицию влево, что проще).

Другой близкий способ – это проверять каждый счетчик после его увеличения и после достижения максимального значения делать деление на 2 всех счетчиков. Этот подход требует более редкого деления, но более сложных проверок.

В любом случае, все операции должны делаться синхронно кодером и декодером.

*Сначала соберите ваши факты, а затем можете передергивать их по своему усмотрению. (Факты упрямая вещь, а статистика более гговорчива.)*

— Марк Твен

## 1.6. Факсимильное сжатие

Сжатие информации бывает особенно важным при передаче изображений по линиям связи, потому что получатель, обычно, ждет на приемном конце и желает поскорее увидеть результат. Документы, пересылаемые с помощью факс-машин, представляются в виде последовательности битов, поэтому сжатие было просто необходимо для популяризации этого метода сообщений. Несколько мето-

дов было развито и предложено для стандарта факсимильной<sup>1</sup> связи организации ITU-T. [Anderson et al. 87], [Hunter, Robinson 80], [Marking 90] и [McConnell 92] – вот несколько из множества ссылок с описанием этого стандарта. Формальное описание можно найти в [Ohio-state 01] и в файлах 7\_3\_01.ps.gz и 7\_3\_02.ps.gz на ftp-сайте [ccitt 01].

ITU-T – это одно из подразделений Международного Союза Телекоммуникаций (International Telecommunications Union, ITU), которое располагается в Женеве (Швейцария) (<http://www.itu.ch/>). Эта организация издает рекомендации по стандартам, работающим в модемах. Несмотря на то, что у этой авторитетной организации нет никаких властных полномочий, ее рекомендации, обычно, принимаются и используются производителями по всему миру. До марта 1993 года, ITU-T была известна как Консультативный Комитет по Международной Телефонии и Телеграфии (Comité Consultatif International Télégraphique et Téléphonique, CCITT).

Первые разработанные в ITU-T стандарты называются T2 (известный также как Group 1) и T3 (Group 2). Теперь они устарели и заменены алгоритмами T4 (Group 3) и T6 (Group 4). В настоящее время Group 3 используется в факс-машинах, разработанных для сетей PSTN (Public Switched Telephone Network). Эти аппараты работают на скорости 9600 бод. Алгоритм Group 4 разработан для эксплуатации в цифровых сетях, таких как ISDN. Они обычно работают на скорости 64К бод. Оба метода обеспечивают фактор сжатия 10:1 и выше, сокращая время передачи страницы типичного документа до минуты в первом случае, и до нескольких секунд во втором.

### 1.6.1. Одномерное кодирование

Факс-машина сканирует документ по строчкам, одну за другой, переводя каждую строку в последовательность черных и белых точек, называемых *пелами* (от pel, Picture Element). Горизонтальное разрешение всегда составляет 8.05 пелов на миллиметр (примерно 205 пелов на дюйм). Таким образом, строка стандартной длины 8.5 дюймов конвертируется в 1728 пелов. Стандарт T4, однако, предписывает сканирование строки длиной около 8.2 дюймов, что производит 1664 пела. (Все величины в этом и других параграфах приводятся с точностью  $\pm 1\%$ .)

<sup>1</sup>Слово «факсимиле» происходит от двух латинских *facere* (делать) и *similis* (похожий).

Вертикальное разрешение составляет 3.85 линий на миллиметр (стандартная мода) или 7.7 линий на миллиметр (тонкая мода). Во многих факс-машинах имеется также сверх тонкая мода, при которой сканируется 15.4 линий на миллиметр. В табл. 1.18 приведен пример для страницы высотой в 10 дюймов (254 мм), и показано общее число пелов на страницу и типичное время передачи для всех трех мод без компрессии. Время очень большое, что показывает, насколько важно сжатие при пересылке факсов.

Число линий	Число пелов в линии	Число пелов на странице	Время (сек)	Время (мин)
978	1664	1.670M	170	2.82
1956	1664	3.255M	339	5.65
3912	1664	6.510M	678	11.3

Табл. 1.18. Время передачи факсов.

Изображение	Описание
1	Типичное деловое письмо на английском
2	Рисунок электрической цепи (от руки)
3	Печатная форма, заполненная на машинке на французском
4	Плотно напечатанный отчет на французском
5	Техническая статья с иллюстрациями на французском
6	График с напечатанными подписями на французском
7	Плотный документ
8	Рукописная записка белым по черному на английском

Табл. 1.19. Восемь эталонных документов.

Для того чтобы сгенерировать коды Group 3, было сосчитано распределение последовательностей черных и белых пелов в восьми эталонных документах, которые содержали типичные тексты с изображениями, которые обычно посылают по факсу. Потом использовался алгоритм Хаффмана для построения префиксных кодов переменной длины, которые кодировали серии черных и белых пелов. (Эти 8 текстов описаны в табл. 1.19, они здесь не опубликованы, так как на них распространяется авторское право ITU-T. Их можно скачать из [funet 91].) Было обнаружено, что чаще всего встречаются серии из 2, 3 и 4 черных пелов, поэтому им были присвоены самые короткие коды (табл. 1.20). Потом шли серии из 2–7 белых пелов, которым были присвоены несколько более длинные коды. Большинство же остальных длин серий встречались реже, и им

были назначены длинные, 12-битные коды. Таким образом, в стандарте Group 3 была использована комбинация кодирования RLE и метода Хаффмана.

Длина серий	Белые коды	Черные коды	Длина серий	Белые коды	Черные коды
0	00110101	0000110111	32	00011011	000001101010
1	000111	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	0000011011010
11	01000	0000101	43	00101100	0000011011011
12	001000	0000111	44	00101101	000001010100
13	000011	00000100	45	00000100	000001010101
14	110100	00000111	46	00000101	000001010110
15	110101	000011000	47	000001010	000001010111
16	101010	0000010111	48	000001011	000001100100
17	101011	0000011000	49	01010010	000001100101
18	0100111	0000001000	50	01010011	0000001010010
19	0001100	000001100111	51	01010100	0000001010011
20	0001000	000001101000	52	01010101	0000000100100
21	0010111	000001101100	53	00100100	0000000110111
22	0000011	000000110111	54	00100101	0000000111000
23	0000100	000000101000	55	01011000	0000000100111
24	0101000	000000010111	56	01011001	0000000101000
25	0101011	000000011000	57	01011010	00000001011000
26	0010011	0000011001010	58	01011011	00000001011001
27	0100100	0000011001011	59	01001010	00000001010111
28	0011000	0000011001100	60	01001011	00000001011100
29	00000010	0000011001101	61	00110010	0000001011010
30	00000011	0000001101000	62	00110011	0000001100110
31	00011010	0000001101001	63	00110100	0000001100111

Табл. 1.20 (а). Коды Group 3 и 4 для факсов.

Интересно отметить, что строке из 1664 белых пелов был присвоен короткий код 011000. Дело в том, что при сканировании часто попадаются пустые строки, которым соответствует это число пелов.

Поскольку длина серий одинаковых пелов может быть большой, алгоритм Хаффмана был модифицирован. Сначала коды были присвоены остаткам – сериям длины от 1 до 63 пелов (они показаны в

табл. 1.20а). Другие коды были даны сериям, длины которых кратны 64 (они приведены в табл. 1.20б). Таким образом, Group 3 – это *модифицированные коды Хаффмана* (коды МН). Каждый код соответствует либо короткой остаточной серии длины до 64, либо длинной серии, кратной 64. Вот некоторые примеры:

Длина серии	Белые коды	Черные коды	Длина серии	Белые коды	Черные коды
64	11011	0000001111	1344	011011010	0000001010011
128	10010	000011001000	1408	011011011	0000001010100
192	010111	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	000000001000	с этого места как и белые
576	01101000	0000001101101	1856	000000001100	
640	01100111	0000001001010	1920	000000001101	
704	011001100	0000001001011	1984	0000000010010	
768	011001101	0000001001100	2048	0000000010011	
832	011010010	0000001001101	2112	0000000010100	
896	011010011	0000001110010	2176	0000000010101	
960	011010100	0000001110011	2240	0000000010110	
1024	011010101	0000001110100	2304	0000000010111	
1088	011010110	0000001110101	2368	0000000011100	
1152	011010111	0000001110110	2432	0000000011101	
1216	011011000	0000001110111	2496	0000000011110	
1280	011011001	0000001010010	2560	0000000011111	

Табл. 1.20 (б). Коды Group 3 и 4 для факсов.

1. Серия из 12 белых пелов кодируется как 001000.
2. Серия из 76 белых пелов ( $=64+12$ ) кодируется как 11011|001000 (без вертикальной черты).
3. Серия из 140 белых пелов ( $=128+12$ ) получает код 10010|001000.
4. Код 64 черных пелов ( $=64+0$ ) равен 0000001111|0000110111.
5. Код 2561 черных пелов ( $2560+1$ ) – 000000011111|010.

Дотошный читатель заметит, что разные коды были также присвоены пустым сериям белых и черных пелов. Эти коды необходимы для того, чтобы обозначить серии, длины которых равны 64, 128 или любому числу, кратному 64. Он также может заметить, что серии длины 2561 быть не может, так как в строке длины 8.5 дюймов помещается только 1728 пелов, поэтому коды для более длинных серий



не нужны. Однако, могут быть (или появиться в будущем) факс-машины для широкой бумаги, поэтому коды Group 3 были созданы с учетом этой возможности.

Каждая строка пелов кодируется отдельно, заканчиваясь специальным 12-битным EOL-кодом 000000000001. К каждой строке также добавляется слева один белый пел. Это делается для того, чтобы избежать неопределенности в декодировании при получении конца.

Прочитав код EOL для предыдущей строки, приемник знает, что новая строка начнется с одного белого пела и игнорирует первого из них. Примеры:

1. Стока из 14 пелов  кодируется сериями 1w 3b 2w 2b 7w EOL, и ей присваивается следующий двоичный код: 000111|10|0111|11|1111|000000000001. Декодер игнорирует одиночный белый пел в начале.

2. Строке  ставится в соответствие серия 3w 5b 5w 2b EOL, и она получает следующий двоичный код: 1000|0011|1100|11|000000000001.

Заметим, что коды из табл. 1.20 должны удовлетворять свойству префикса только в каждом столбце. Дело в том, что каждая отсканированная строка начинается белым пелом, и декодер знает, какого цвета будет следующая серия. Примером нарушения свойства префикса служит код для пяти черных пелов (0011), которым начинаются коды белых серий длины 61, 62 и 63.

Коды Group 3 не могут исправлять ошибки, но они могут обнаружить многие из них. Дело в том, что по природе кодов Хаффмана, даже один плохо переданный бит вынудит приемник потерять синхронизацию и породить последовательность неправильных пелов. Поэтому последовательные строки следует кодировать независимо друг от друга. Если декодер обнаруживает ошибку, он пропускает биты, разыскивая EOL. При таком методе одиночная ошибка может испортить не более одной строки. Если декодер не может долгое время обнаружить EOL, он предполагает высокую зашумленность канала и прерывает процесс, сообщая об этом передатчику. Поскольку длины кодов лежат в диапазоне от 2 до 12, то приемник обнаруживает ошибку, если он не в состоянии выявить правильный код, прочитав 12 бит.

В начале каждой страницы передается код EOL, а в конце страницы ставится 6 кодов EOL. Поскольку все строки кодируются независимо, эта схема называется схемой *одномерного кодирования*. Коэффициент сжатия зависит от передаваемого изображения. Если

оно состоит из крупных соприкасающихся белых и черных областей (текст, таблицы или графики), то он будет сильно сжат. А изображения, в которых присутствует много коротких серий могут вызвать отрицательное сжатие. Это может случиться при передаче полутоновых изображений, например, фотографий. Такие изображения при сканировании порождают множество коротких пелов длины 1 или 2.

Поскольку стандарт Т4 основан на длинных сериях, он может давать плохое сжатие, если все серии будут короткими. Экстремальный случай – это, когда все пели имеют длину 1. Белый пел имеет код 000111, а черный – 010. Поэтому при кодировании двух последовательных разноцветных пелов требуется 9 бит, тогда как без кодирования можно обойтись двумя битами (01 или 10). Значит, коэффициент сжатия равен  $9/2=4.5$  (сжатый файл будет в 4.5 раза длиннее исходного).

Стандарт Т4 допускает добавление нулевых бит между кодом данных и кодом EOL. Это делается, если необходимо сделать паузу, например, в связи с тем, что число передаваемых битов кодирующих целую строку должно делиться на 8.

**Пример:** Двоичная строка 000111|10|0111|11|1111|000000000001 становится немного длиннее 000111|10|0111|11|1111|00|000000000001 после добавления 2 нулей, чтобы общая длина строки была 32 бит ( $= 8 \times 4$ ). Декодер обнаружит это добавление перед одиннадцатью нулями кода EOL.

98% всей статистики выдумана.  
— Неизвестный

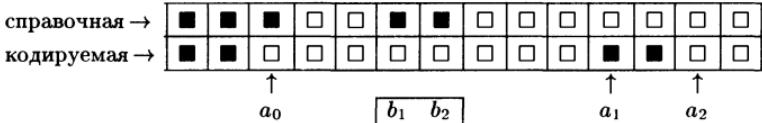
### 1.6.2. Двумерное кодирование

Двумерное кодирование было разработано, чтобы преодолеть недостатки одномерного кодирования при сжатии изображений, содержащих серые области. Этот метод является опционным дополнением к Group 3 и используется только при работе в цифровых сетях. Если факс-машина поддерживает двумерное кодирование, то за кодом EOL следует еще один бит, указывающий на метод кодирования следующей строки. Если он равен 1, то будет использоваться одномерное кодирование, а 0 указывает на двумерную схему.

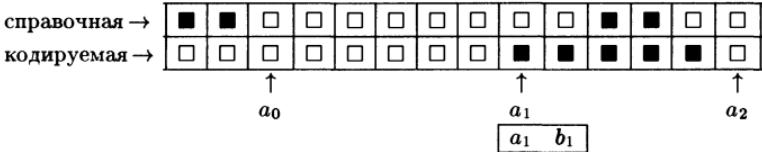
Метод двумерного кодирования также называется MMR (*modified modified READ*, то есть, дважды модифицированный *READ*, а *READ* расшифровывается как *relative element adress designate* –

**(a)**

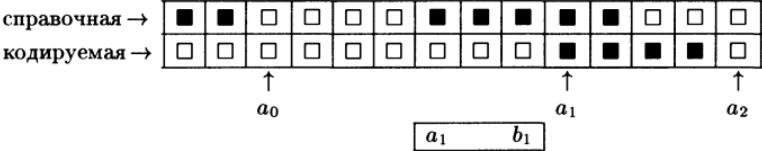
Строки:

Кодируется длина  $b_1 b_2$  серии. Новая  $a_0$  становится старой  $b_2$ .**(b1)**

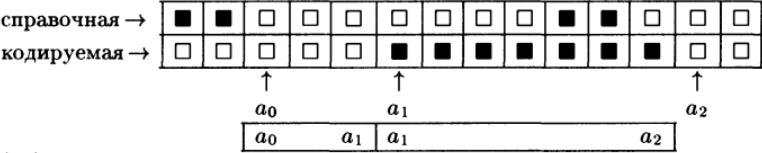
Строки:

**(b2)**

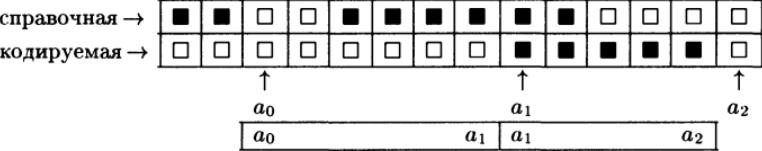
Строки:

Кодируется длина  $a_1 b_1$  серии. Новая  $a_0$  становится старой  $a_1$ .**(c1)**

Строки:

**(c2)**

Строки:

Кодируются длины  $a_0 a_1$  (белой) и  $a_1 a_2$  (черной) серий.Новая  $a_0$  становится старой  $a_2$ .

---

Примечания

1.  $a_0$  – первый пел нового кода; он может быть черным или белым.
  2.  $a_1$  – первый пел другого цвета справа от  $a_0$ .
  3.  $a_2$  – первый пел другого цвета справа от  $a_1$ .
  4.  $b_1$  – первый пел справочной строки другого цвета справа от  $a_0$ .
  5.  $b_2$  – первый пел справочной строки другого цвета справа от  $b_1$ .
- 

Рис. 1.21. Пять конфигураций мод.

обозначение относительного адреса элемента). Такое странное словосочетание объясняется тем, что этот алгоритм является модификацией одномерной схемы, которая, в свою очередь, получена модификацией оригинального метода Хаффмана. Метод работает, сравнивая текущую отсканированную строку, называемую *кодируемой*, со строкой, отсканированной на предыдущем проходе, которая называется *справочной* строкой. При этом будет сжиматься *разность* этих строк. Алгоритм исходит из логичного предположения, что две соседние строки обычно отличаются всего несколькими пелами. При этом предполагается, что документ начинается строкой белых пелов, которая служит начальной справочной строкой. После кодирования первая строка становится справочной, а вторая строка – кодируемой. Как и при одномерном кодировании предполагается, что строка начинается белым пелом, который игнорируется приемником.

Метод двумерного кодирования менее надежен, чем одномерный метод, поскольку ошибка в декодировании некоторой строки вызовет ошибки при декодировании последующих строк, и эта волна ошибок может распространиться до конца по всему документу. Вот почему стандарт Т4 (Group 3) включает требование, что после строки, закодированной одномерным методом, следует не более  $K - 1$  строк, закодированных двумерной схемой. Для стандартного разрешения  $K = 2$ , а для тонкого  $K = 4$ . Стандарт Т6 не содержит этого требования и использует только двумерное кодирование.

Сканирование кодируемой строки и ее сравнение со справочной строкой делается в трех случаях или модах. Мода определяется при сравнении очередной серии пелов справочной строки [( $b_1b_2$ ) на рис. 1.21] с текущей серией ( $a_0a_1$ ), а также со следующей серией ( $a_1a_2$ ) кодируемой строки. Каждая из этих серий может быть белой или черной. Опишем эти три моды.

**1. Проходная мода.** Это случай, когда ( $b_1b_2$ ) находится слева от ( $a_1a_2$ ), а  $b_2$  – слева от  $a_1$  (рис. 1.21а). Эта мода не включает случай, когда  $b_2$  находится над  $a_1$ . Когда эта мода установлена, то блок ( $b_1b_2$ ) кодируется с помощью кодов табл. 1.22 и передается. Указатель  $a_0$  устанавливается под  $b_2$ , а четыре величины  $b_1$ ,  $b_2$ ,  $a_1$  и  $a_2$  обновляются.

**2. Вертикальная мода.** В этом случае ( $b_1b_2$ ) частично перекрываетяется с ( $a_1a_2$ ), но не более чем тремя пелами (рис. 1.21б1 и 1.21б2). Если предположить, что соседние строки отличаются не сильно, то это будет самый частый случай. При обнаружении этой моды генерируется один из семи кодов (табл. 1.22) и посыпается. Производи-

тельность двумерной схемы зависит от того, насколько часто имеет место эта мода.

Мода	Кодируемая серия	Сокращение	Код
Проходная	$b_1b_2$	P	0001+код для длины $b_1b_2$
Горизонтальная	$a_0a_1, a_1a_2$	H	001+код для длины $a_0a_1$ и $a_1a_2$
Вертикальная	$a_1b_1 = 0$	V(0)	1
	$a_1b_1 = -1$	VR(1)	011
	$a_1b_1 = -2$	VR(2)	000011
	$a_1b_1 = -3$	VR(3)	0000011
	$a_1b_1 = +1$	VL(1)	010
	$a_1b_1 = +2$	VL(2)	000010
	$a_1b_1 = +3$	VL(3)	0000010
Расширенный			0000001000

Табл. 1.22. Двумерные коды для метода Group 4.

**3. Горизонтальная мода.** Серия  $(b_1b_2)$  перекрывается с  $(a_1a_2)$  более чем по трем пелам (рис. 1.21c1 и 1.21c2). При обнаружении этой моды серии  $(a_0a_1)$  и  $(a_1a_2)$  кодируются с помощью табл. 1.22 и передаются. Указатели обновляются как в случаях 1 и 2.

В начале сканирования указатель  $a_0$  устанавливается на воображаемый белый пел слева от кодируемой строки, а указатель  $a_1$  указывает на первый черный пел. (Поскольку  $a_0$  указывает на воображаемый пел, то длина первой белой серии равна  $|a_0a_1| - 1$ .) Указатель  $a_2$  устанавливается на следующий первый белый пел. Указатели  $b_1$  и  $b_2$  указывают на начало первой и второй серии справочной строки, соответственно.

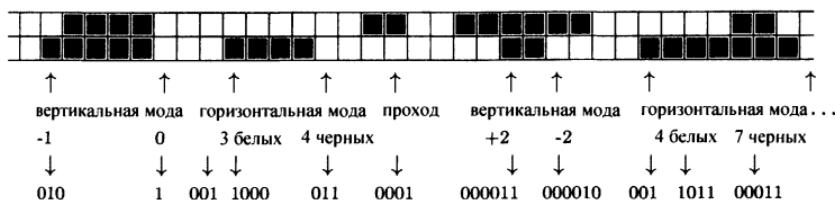


Рис. 1.23. Пример двумерного кодирования.

После идентификации текущей моды и передачи кода в соответствии с табл. 1.22, указатель  $a_0$  обновляется как показано на блок-схеме, а остальные указатели обновляются в соответствии с новым

положением  $a_0$ . Процесс продолжается до тех пор, пока не достигнет конец кодируемой строки. Кодер предполагает дополнительный пел справа от строки, цвет которого противоположен цвету последнего пела.

Расширенный код в табл. 1.22 используется для преждевременного обрывания процесса кодирования до достижения конца страницы. Это необходимо сделать, если оставшаяся часть страницы будет передаваться другими кодами или в несжатой форме.

**Пример:** Рис. 1.23 изображает, какие моды и какие коды соответствуют двум соседним строкам пелов.

*Статистика здравого смысла говорит, что каждый американец из четырех – сумасшедший. Подумайте о трех ваших лучших друзьях: если они OK, то – это вы.*

— Rita Mae Braun

## 1.7. Арифметическое кодирование

Метод Хаффмана является простым и эффективным, однако, как было замечено в § 1.4, он порождает наилучшие коды переменной длины (коды, у которых средняя длина равна энтропии алфавита) только когда вероятности символов алфавита являются степенями числа 2, то есть равны  $1/2, 1/4, 1/8$  и т.п. Это связано с тем, что метод Хаффмана присваивает каждому символу алфавита код с целым числом битов. Теория информации предсказывает, что при вероятности символа, скажем, 0,4, ему в идеале следует присвоить код длины 1.32 бита, поскольку  $-\log_2 0.4 \approx 1.32$ . А метод Хаффмана присвоит этому символу код длины 1 или 2 бита.

(Перед тем как углубиться в теорию арифметического кодирования, стоит указать две работы [Moffat et al. 98] и [Witter 87], где рассмотрены основные принципы этого метода и приведено множество примеров.)

Арифметическое кодирование решает эту проблему путем присвоения кода всему, обычно, большому передаваемому файлу вместо кодирования отдельных символов. (Входным файлом может быть текст, изображение или данные любого вида.) Алгоритм читает входной файл символ за символом и добавляет биты к сжатому файлу. Чтобы понять метод, полезно представлять себе получающийся код в виде числа из полуинтервала  $[0, 1]$ . (Здесь  $[a, b)$  обозначает числа от  $a$  до  $b$ , включая  $a$  и исключая  $b$ . Конец  $a$  замкнут, а конец

б открыт.) Таким образом код «9746509» следует интерпретировать как число «0.9746509» однако часть «0.» не будет включена в передаваемый файл.

На первом этапе следует вычислить или, по крайней мере, оценить частоты возникновения каждого символа алфавита. Наилучшего результата можно добиться, прочитав весь входной файл на первом проходе алгоритма сжатия, состоящего из двух проходов. Однако, если программа может получить хорошие оценки частот символов из другого источника, первый проход можно опустить.

В первом примере мы рассмотрим три символа  $a_1$ ,  $a_2$  и  $a_3$  с вероятностями  $P_1 = 0.4$ ,  $P_2 = 0.5$  и  $P_3 = 0.1$ , соответственно. Интервал  $[0, 1)$  делится между этими тремя символами на части пропорционально их вероятностям. Порядок следования этих подинтервалов не существенен. В нашем примере трем символам будут соответствовать подинтервалы  $[0, 0.4)$ ,  $[0.4, 0.9)$  и  $[0.9, 1.0)$ . Чтобы закодировать строку « $a_2a_2a_2a_3$ », мы начинаем с интервала  $[0, 1)$ . Первый символ  $a_2$  сокращает этот интервал, отбросив от него 40% в начале и 10% в конце. Результатом будет интервал  $[0.4, 0.9)$ . Второй символ  $a_2$  сокращает интервал  $[0.4, 0.9)$  в той же пропорции до интервала  $[0.6, 0.85)$ . Третий символ  $a_2$  переводит его в  $[0.7, 0.825)$ . Наконец, символ  $a_3$  отбрасывает от него 90% в начале, а конечную точку оставляет без изменения. Получается интервал  $[0.8125, 0.8250)$ . Окончательным кодом нашего метода может служить любое число из этого промежутка. (Заметим, что подинтервал  $[0.6, 0.85)$  получен из  $[0.4, 0.9)$  с помощью следующих преобразований его концов:  $0.4 + (0.9 - 0.4) \times 0.4 = 0.6$  и  $0.4 + (0.9 - 0.4) \times 0.9 = 0.85$ .)

На этом примере легко понять следующие шаги алгоритма арифметического кодирования:

1. Задать «текущий интервал»  $[0, 1)$ .

2. Повторить следующие действия для каждого символа  $s$  входного файла.

**2.1.** Разделить текущий интервал на части пропорционально вероятностям каждого символа.

**2.2.** Выбрать подинтервал, соответствующий символу  $s$ , и назначить его новым текущим интервалом.

3. Когда весь входной файл будет обработан, выходом алгоритма объявляется любая точка, которая однозначно определяет текущий интервал (то есть, любая точка внутри этого интервала).

После каждого обработанного символа текущий интервал становится все меньше, поэтому требуется все больше бит, чтобы выра-

зить его, однако окончательным выходом алгоритма является единственное число, которое не является объединением индивидуальных кодов последовательности входных символов. Среднюю длину кода можно найти, разделив размер выхода (в битах) на размер входа (в символах). Отметим, что вероятности, которые использовались на шаге 2.1, могут каждый раз меняться, и это можно использовать в адаптивной вероятностной модели (см. § 1.8).

Следующий пример будет немного более запутанным. Мы продемонстрируем шаги сжатия для строки «SWISS\_MISS». В табл. 1.24 указана информация, приготовленная на предварительном этапе (*статистическая модель данных*). Пять символов на входе можно упорядочить любым способом. Для каждого символа сначала вычислена его частота, затем найдена вероятность ее появления (частота, деленная на длину строки, 10). Область [0, 1) делится между символами в любом порядке. Каждый символ получает кусочек или подобласть, равную его вероятности. Таким образом, символ «S» получает интервал [0.5, 1.0) длины 0.5, а символу «I» достается интервал [0.2, 0.4) длины 0.2. Столбец CumFreq (накопленные частоты) будет использоваться алгоритмом декодирования на стр. 71. CumFreq равно сумме частот всех предыдущих символов, например, для «W», CumFreq=1+1+2.

Символ	Частота	Вероятность	Область	CumFreq
Общая CumFreq =				10
S	5	5/10=0.5	[0.5,1.0)	5
W	1	1/10=0.1	[0.4,0.5)	4
I	2	2/10=0.2	[0.2,0.4)	2
M	1	1/10=0.1	[0.1,0.2)	1
-	1	1/10=0.1	[0.0,0.1)	0

Табл. 1.24. Частоты и вероятности 5 символов.

Символы и частоты табл. 1.24 записываются в начало выходного файла до битов кода сжатия.

Процесс кодирования начинается инициализацией двух переменных *Low* и *High* и присвоением им 0 и 1, соответственно. Они определяют интервал  $[Low, High]$ . По мере поступления и обработки символов, переменные *Low* и *High* начинают сближаться, уменьшая интервал.

После обработки первого символа «S», переменные *Low* и *High* равны 0.5 и 1, соответственно. Окончательным сжатым кодом все-

го входного файла будет число из этого интервала ( $0.5 \leq Code < 1.0$ ). Для лучшего понимания процесса кодирования хорошо представить себе, что новый интервал  $[0.5,1)$  делится между пятью символами нашего алфавита в той же пропорции, что и начальный интервал  $[0,1)$ . Результатом будет пять подинтервалов  $[0.50,0.55)$ ,  $[0.55,0.60)$ ,  $[0.60,0.70)$ ,  $[0.70,0.75)$  и  $[0.75,1.0)$ . После поступления следующего символа «W» выбирается четвертый интервал, который снова делится на пять подинтервалов.

Символ		Вычисление переменных Low и High		
S	L	$0.0 + (1.0 - 0.0) \times 0.5$	=	0.5
	H	$0.0 + (1.0 - 0.0) \times 1.0$	=	1.0
W	L	$0.5 + (1.0 - 0.5) \times 0.4$	=	0.70
	H	$0.5 + (1.0 - 0.5) \times 0.5$	=	0.75
I	L	$0.7 + (0.75 - 0.70) \times 0.2$	=	0.71
	H	$0.7 + (0.75 - 0.70) \times 0.4$	=	0.72
S	L	$0.71 + (0.72 - 0.71) \times 0.5$	=	0.715
	H	$0.71 + (0.72 - 0.71) \times 1.0$	=	0.72
S	L	$0.715 + (0.72 - 0.715) \times 0.5$	=	0.7175
	H	$0.715 + (0.72 - 0.715) \times 1.0$	=	0.72
-	L	$0.7175 + (0.72 - 0.7175) \times 0.0$	=	0.7175
	H	$0.7175 + (0.72 - 0.7175) \times 0.1$	=	0.71775
M	L	$0.7175 + (0.71775 - 0.7175) \times 0.1$	=	0.717525
	H	$0.7175 + (0.71775 - 0.7175) \times 0.2$	=	0.717550
I	L	$0.717525 + (0.71755 - 0.717525) \times 0.2$	=	0.717530
	H	$0.717525 + (0.71755 - 0.717525) \times 0.4$	=	0.717535
S	L	$0.717530 + (0.717535 - 0.717530) \times 0.5$	=	0.7175325
	H	$0.717530 + (0.717535 - 0.717530) \times 1.0$	=	0.717535
S	L	$0.7175325 + (0.717535 - 0.7175325) \times 0.5$	=	0.71753375
	H	$0.7175325 + (0.717535 - 0.7175325) \times 1.0$	=	0.717535

Табл. 1.25. Процесс арифметического кодирования.

С каждым поступившим символом переменные Low и High пересчитываются по правилу:

```
NewHigh:=OldLow+Range*HighRange(X);
NewLow:=OldLow+Range*LowRange(X);
```

где Range=OldHigh-OldLow, а HighRange(X) и LowRange(X) обозначают верхний и нижний конец области символа X. В нашем примере второй символ – это «W», поэтому новые переменные будут Low:=  $0.5 + (1.0 - 0.5) \times 0.4 = 0.7$ , High :=  $0.5 + (1.0 - 0.5) \times 0.5 = 0.75$ . Новый интервал  $[0.70,0.75)$  покрывает кусок [40%,50%) подобласти

символа «S». В табл. 1.25 приведены все шаги, связанные с кодированием строки «SWISS\_MISS». Конечный код – это последнее значение переменной *Low*, равное 0.71753375, из которого следует взять лишь восемь цифр 71753375 для записи в сжатый файл (другой выбор кода будет обсуждаться позже).

Декодер работает в обратном порядке. Сначала он узнаёт символы алфавита и считывает табл. 1.24. Затем он читает цифру «7» и узнаёт, что весь код имеет вид 0.7... Это число лежит внутри интервала [0.5,1) символа «S». Значит, первый символ был S. Далее декодер удаляет эффект символа S из кода с помощью вычитания нижнего конца интервала S и деления на длину этого интервала, равную 0.5. Результатом будет число 0.4350675, которое говорит декодеру, что второй символ был «W» (поскольку область «W» – [0.4,0.5)).

Чтобы удалить влияние символа X, декодер делает следующее преобразование над кодом:  $Code := (Code - LowRange(X)) / Range$ , где *Range* обозначает длину интервала символа X. В табл. 1.26 отражен процесс декодирования рассмотренной строки.

Символ	Code-Low	Область
S	0.71753375 – 0.5	= 0.21753375 / 0.5 = 0.4350675
W	0.4350675 – 0.4	= 0.0350675 / 0.1 = 0.350675
I	0.350675 – 0.2	= 0.150675 / 0.2 = 0.753375
S	0.753375 – 0.5	= 0.253375 / 0.5 = 0.50675
S	0.50675 – 0.5	= 0.00675 / 0.5 = 0.0135
-	0.0135 – 0	= 0.0135 / 0.1 = 0.135
M	0.135 – 0.1	= 0.035 / 0.1 = 0.35
I	0.35 – 0.2	= 0.15 / 0.2 = 0.75
S	0.75 – 0.5	= 0.25 / 0.5 = 0.5
S	0.5 – 0.5	= 0 / 0.5 = 0

Табл. 1.26. Процесс арифметического декодирования.

Следующий пример имеет дело с тремя символами, вероятности которых приведены в табл. 1.27а. Заметим, что эти вероятности сильно отличаются друг от друга. Одна – большая, 0.975, а другие – существенно меньше. Это случай *асимметричных вероятностей*.

Кодирование строки  $a_2a_2a_1a_3a_3$  выдаёт числа с точностью в 16 знаков, приведенные в табл. 1.28, в которой для каждого символа в двух строках записаны последовательные значения *Low* и *High*.

На первый взгляд кажется, что полученный код длиннее исходной строки, однако в § 1.7.3 показано, как правильно определять степень сжатия, достигаемого арифметическим кодированием.

Декодирование этой строки показано в табл. 1.29. Оно обнаруживает особую проблему. После удаления эффекта от  $a_1$  в строке 3 стоит число 0. Ранее мы неявно предполагали, что это означает конец процесса декодирования, но теперь мы знаем, что еще имеется два символа  $a_3$ , которые следует декодировать. Это показано в строках 4 и 5 таблицы. Эта проблема всегда возникает, если последний символ входного файла является символом, чья область начинается в нуле. Для того, чтобы различить такой символ и конец входного файла, необходимо ввести один дополнительный символ «eof», обозначающий конец файла (end of file). Этот символ следует добавить в таблицу частот, приписав ему маленькую вероятность (см. табл. 1.27б), и закодировать его в конце входного файла.

Симв.	Вероятн.	Область	Симв.	Вероятн.	Область
$a_1$	0.001838	[0.998162, 1.0)	eof	0.000001	[0.999999, 1.0)
$a_2$	0.975	[0.023162, 0.998162)	$a_1$	0.001837	[0.998162, 0.999999)
$a_3$	0.023162	[0.0, 0.023162)	$a_2$	0.975	[0.023162, 0.998162)

(a)

(b)

Табл. 1.27. (Асимметрические) вероятности трех символов.

$a_2$	$0.0 + (1.0 - 0.0) \times 0.023162 = 0.023162$	$0.0 + (1.0 - 0.0) \times 0.998162 = 0.998162$
$a_2$	$0.023162 + 0.975 \times 0.023162 = 0.04574495$	$0.023162 + 0.975 \times 0.998162 = 0.99636995$
$a_1$	$0.04574495 + 0.950625 \times 0.998162 = 0.99462270125$	$0.04574495 + 0.950625 \times 1.0 = 0.99636995$
$a_3$	$0.99462270125 + 0.00174724875 \times 0.0 = 0.99462270125$	$0.99462270125 + 0.00174724875 \times 0.023162 = 0.994663171025547$
$a_3$	$0.99462270125 + 0.00004046977554749998 \times 0.0 = 0.99462270125$	$0.99462270125 + 0.00004046977554749998 \times 0.023162 = 0.994623638610941$

Табл. 1.28. Кодирование строки  $a_2a_2a_1a_3a_3$ .

Символ	Code-Low	Область
$a_2$	$0.99462270125 - 0.023162 = 0.97146170125$	$/ 0.975 = 0.99636995$
$a_2$	$0.99636995 - 0.023162 = 0.97320795$	$/ 0.975 = 0.998162$
$a_1$	$0.998162 - 0.998162 = 0.0$	$/ 0.00138 = 0.0$
$a_3$	$0.0 - 0.0 = 0.0$	$/ 0.023162 = 0.0$
$a_3$	$0.0 - 0.0 = 0.0$	$/ 0.023162 = 0.0$

Табл. 1.29. Декодирование строки  $a_2a_2a_1a_3a_3$ .

В табл. 1.30 и 1.31 показано, как строка *a3a3a3eof* будет закодирована маленькой дробью 0.0000002878086184764172, а потом корректно декодирована. Без символа eof строка из одних *a3* была закодирована числом 0.

Обратите внимание на то, что нижнее значение было 0, пока не дошли до eof, а верхнее значение быстро стремится к нулю. Как уже отмечалось, окончательным кодом может служить любое число из промежутка от нижнего до верхнего конечного значения, а не обязательно нижний конец. В примере с *a3a3a3eof* кодом может служить более короткое число 0.0000002878086 (или 0.0000002878087, или даже 0.0000002878088).

<i>a</i> <sub>3</sub>	0.0 + (1.0 - 0.0) × 0.0 = 0.0
	0.0 + (1.0 - 0.0) × 0.023162 = 0.023162
<i>a</i> <sub>3</sub>	0.0 + 0.023162 × 0.0 = 0.0
	0.0 + 0.023162 × 0.023162 = 0.000536478244
<i>a</i> <sub>3</sub>	0.0 + 0.000536478244 × 0.0 = 0.0
	0.0 + 0.000536478244 × 0.023162 = 0.000012425909087528
<i>a</i> <sub>3</sub>	0.0 + 0.000012425909087528 × 0.0 = 0.0
	0.0 + 0.000012425909087528 × 0.023162 = 0.0000002878089062853235
<i>eof</i>	0.0 + 0.0000002878089062853235 × 0.999999 = 0.0000002878086184764172
	0.0 + 0.0000002878089062853235 × 1.0 = 0.0000002878089062853235

Табл. 1.30. Кодирование строки *a3a3a3eof*.

(На рис. 1.32 приведена программа для системы *Mathematica*, которая вычисляет табл. 1.28.)

Сим.	Code-Low	Область
<i>a</i> <sub>3</sub>	0.00000028780861847642-0	= 0.00000028780861847642 /0.023162 = 0.000012425896661618912
<i>a</i> <sub>3</sub>	0.000012425896661618912-0	= 0.000012425896661618912 /0.023162 = 0.0005364777075218
<i>a</i> <sub>3</sub>	0.0005364777075218 - 0	= 0.000536477707521756 /0.023162 = 0.023161976838
<i>a</i> <sub>3</sub>	0.023161976838 - 0.0	= 0.023161976838 /0.023162 = 0.999999
<i>eof</i>	0.999999 - 0.999999	= 0.0 /0.000001 = 0.0

Табл. 1.31. Декодирование строки *a3a3a3eof*.

**Пример:** В табл. 1.33 приведены шаги кодирования строки одинаковых символов *a2a2a2a2*. Из-за высокой вероятности символа *a2* переменные Low и High начинаются с сильно различающихся начальных значений и приближаются друг к другу достаточно медленно.

Если размер входного файла известен, тогда можно его кодировать без символа eof. Кодер может начать с записи его размера (в несжатом виде) в заголовок выходного файла. Декодер прочитает этот размер, начнет процесс декомпрессии и остановится после извлечения всех символов. Если декодер читает сжатый файл байт за байтом, то кодер может добавить в конце несколько нулевых битов, чтобы сжатый файл мог быть прочитан блоками по 8 бит.

```

lowRange={0.998162,0.023162,0.};
highRange={1.,0.998162,0.023162};
low=0.; high=1.;
enc[i_]:=Module[{nlow,nhigh,range},
range=high-low;
nhigh=low+range highRange[[i]];
nlow=low+range lowRange[[i]];
low=nlow; high=nhigh;
Print["r=",N[range,25],"l=",N[low,17],"h=",N[high,17]]];
enc[2]
enc[2]
enc[1]
enc[3]
enc[3]

```

Рис. 1.32. Программа для вычисления табл. 1.28.

$a_2$	$0.0 + (1.0 - 0.0) \times 0.023162$	=	0.023162
	$0.0 + (1.0 - 0.0) \times 0.998162$	=	0.998162
$a_2$	$0.023162 + 0.975 \times 0.023162$	=	0.04574495
	$0.023162 + 0.975 \times 0.998162$	=	0.99636995
$a_2$	$0.04574495 + 0.950625 \times 0.023162$	=	0.06776332625
	$0.04574495 + 0.950625 \times 0.998162$	=	0.99462270125
$a_2$	$0.06776332625 + 0.926859375 \times 0.023162$	=	0.08923124309375
	$0.06776332625 + 0.926859375 \times 0.998162$	=	0.99291913371875

Табл. 1.33. Кодирование строки  $a_2a_2a_2a_2$ .

### 1.7.1. Детали реализации метода

Описанный выше процесс кодирования невозможно реализовать на практике, так как в нем предполагается, что в переменных Low и High хранятся числа с неограниченной точностью. Процесс декодирования, описанный на стр. 66 («Далее декодер удаляет эффект символа S из кода с помощью вычитания ... и деления ...») по своей сути очень прост, но совершенно не практичен. Код, который является одним числом, обычно будет длинным, очень длинным числом.

Файл объема 1 МБ будет сжиматься, скажем, до 500 КБ, в котором будет записано всего одно число. Деление чисел в 500 КБ делается очень сложно и долго.

Любое практическое применение арифметического кодирования должно основываться на оперировании с целыми числами (арифметика чисел с плавающей запятой работает медленно и при этом проходит потеря точности), которые не могут быть слишком длинными. Мы опишем такую реализацию с помощью двух целых переменных *Low* и *High*. В нашем примере они будут иметь длину в 4 десятичные цифры, но на практике будут использоваться целые длиной 16 или 32 бита. Эти переменные будут хранить верхний и нижний концы текущего подинтервала, но мы не будем им позволять неограниченно расти. Взгляд на табл. 1.25 показывает, что как только самые левые цифры переменных *Low* и *High* становятся одинаковыми, они уже не меняются в дальнейшем. Поэтому мы будем выдвигать эти числа за пределы переменных *Low* и *High* и сохранять их в выходном файле. Таким образом, эти переменные будут хранить не весь код, а только самую последнюю его часть. После сдвига цифр мы будем справа дописывать 0 в переменную *Low*, а в переменную *High* – цифру 9. Для того, чтобы лучше понять весь процесс, можно представлять себе эти переменные как левый конец бесконечно длинного числа. Число *Low* имеет вид *xxxx00...* а число *High= yyyy99...*

Проблема состоит в том, что переменная *High* в начале должна равняться 1, однако мы интерпретируем *Low* и *High* как десятичные дроби меньшие 1. Решение заключается в присвоении переменной *High* значения 9999..., которое соответствует бесконечной дроби 0.9999..., равной 1.

(Это легко доказать. Если  $0.9999\dots < 1$ , то среднее число  $a = (0.9999\dots + 1)/2$  должно лежать между 0.9999... и 1; однако, его невозможно выразить десятичной дробью. Нельзя добавить ни одной цифры к числу 0.9999..., поскольку в нем бесконечно много цифр. Также ни одну из его цифр нельзя увеличить, так как все они равны 9. Значит, число 0.9999... равно 1. Рассуждая аналогично, легко показать, что число 0.5 имеет два представления в виде двоичной дроби: 0.1000... и 0.0111...).

В табл. 1.34 представлен процесс кодирования строки символов «SWISS\_MISS». В первом столбце записаны символы на входе. Столбец 2 содержит новые значения переменных *Low* и *High*. В третьем столбце эти числа представлены в измененном масштабе с уменьшением *High* на 1. Стока 4 показывает следующую цифру, посы

ляемую на выход, а в пятом столбце записаны **Low** и **High** после их сдвига влево на одну цифру. Заметьте, что на последнем шаге в выходной файл было записано четыре цифры 3750. Окончательный выходной файл имеет вид 717533750.

Декодер работает в обратном порядке. В начале сделаны присвоения **Low=0000**, **High=9999**, и **Code=7175** (первые четыре цифры сжатого файла). Эти переменные обновляются на каждом шаге процедуры декодирования. **Low** и **High** приближаются друг к другу (и к **Code**) до тех пор, пока их самые значимые цифры не будут совпадать. Тогда они сдвигаются влево, что приводит к их разделению, **Code** тоже сдвигается. На каждом шаге вычисляется переменная **index**, которая используется для нахождения столбца **CumFreq** табл. 1.24, позволяющего определить текущий символ.

1	2	3	4	5
S	L= 0.0 + (1.0 - 0.0) × 0.5 = 0.5	5000	5000	
	H= 0.0 + (1.0 - 0.0) × 1.0 = 1.0	9999	9999	
W	L= 0.5 + (1.0 - 0.5) × 0.4 = 0.7	7000	7	0000
	H= 0.5 + (1.0 - 0.5) × 0.5 = 0.75	7499	7	4999
I	L= 0.0 + (0.5 - 0.0) × 0.2 = 0.1	1000	1	0000
	H= 0.0 + (0.5 - 0.0) × 0.4 = 0.2	1999	1	9999
S	L= 0.0 + (1.0 - 0.0) × 0.5 = 0.5	5000	5000	
	H= 0.0 + (1.0 - 0.0) × 1.0 = 1.0	9999	9999	
S	L= 0.5 + (1.0 - 0.5) × 0.5 = 0.75	7500	7500	
	H= 0.5 + (1.0 - 0.5) × 1.0 = 1.0	9999	9999	
-	L= 0.75 + (1.0 - 0.75) × 0.0 = 0.75	7500	7	5000
	H= 0.75 + (1.0 - 0.75) × 0.1 = 0.775	7749	7	7499
M	L= 0.5 + (0.75 - 0.5) × 0.1 = 0.525	5250	5	2500
	H= 0.5 + (0.75 - 0.5) × 0.2 = 0.55	5499	5	4999
I	L= 0.25 + (0.5 - 0.25) × 0.2 = 0.3	3000	3	0000
	H= 0.25 + (0.5 - 0.25) × 0.4 = 0.35	3499	3	4999
S	L= 0.0 + (0.5 - 0.0) × 0.5 = 0.25	2500	2500	
	H= 0.0 + (0.5 - 0.0) × 1.0 = 0.5	4999	4999	
S	L= 0.25 + (0.5 - 0.25) × 0.5 = 0.375	3750	3750	
	H= 0.25 + (0.5 - 0.25) × 1.0 = 0.5	4999	4999	

Табл. 1.34. Кодирование «SWISS\_MISS» сдвигами.

Каждая итерация цикла состоит из следующих шагов:

1. Вычислить число **index:=((Code-Low)\*10-1)/(High-Low+1)** и округлить его до ближайшего целого (в нашем примере общее **CumFreq** равно 10).
2. Использовать **index** для нахождения следующего символа путем сравнения с **CumFreq** в табл. 1.24. В следующем примере первое зна-

чение `index` равно 7.1759, а после округления – 7. Число 7 находится в таблице между 5 и 10, поэтому выбран символ «S».

**3.** Изменить `Low` и `High` по формулам:

`Low:=Low+(High-Low+1)*LowCumFreq[X]/10;`

`High:=Low+(High-Low+1)*HighCumFreq[X]/10-1;`

где `LowCumFreq[X]` и `HighCumFreq[X]` – накопленные частоты символа `X` и символа над `X` в табл. 1.24.

**4.** Если самые левые цифры в переменных `Low` и `High` совпадают, то сдвинуть `Low`, `High` и `Code` на одну позицию влево. В самую правую позицию `Low` записать 0, в `High` – 9, а в `Code` считать следующую цифру из сжатого файла.

Приведем все шаги декодирования для нашего примера.

**0.** Инициализация `Low= 0000`, `High= 9999` и `Code= 7175`.

**1.**  $\text{index} = ((7175 - 0 + 1) \times 10 - 1) / (9999 - 0 + 1) = 7.1759 \rightarrow 7$ . Символ «S» выбран.

$\text{Low}=0+(9999-0+1) \times 5/10 = 5000$ .  $\text{High}=0+(9999-0+1) \times 10/10-1 = 9999$ .

**2.**  $\text{index} = ((7175 - 5000 + 1) \times 10 - 1) / (9999 - 5000 + 1) = 4.3518 \rightarrow 4$ . Символ «W» выбран.

$\text{Low}=5000+(9999-5000+1) \times 4/10 = 7000$ .  $\text{High}=5000+(9999-5000+1) \times 5/10 - 1 = 7499$ .

После сдвига и выхода цифры 7 имеем `Low= 0000`, `High = 4999` и `Code= 1753`.

**3.**  $\text{index} = ((1753 - 0 + 1) \times 10 - 1) / (4999 - 0 + 1) = 3.5078 \rightarrow 3$ . Символ «I» выбран.

$\text{Low}=0+(4999-0+1) \times 2/10 = 1000$ ,  $\text{High}=0+(4999-0+1) \times 4/10-1 = 1999$ .

После сдвига и выхода цифры 1 имеем `Low= 0000`, `High = 9999` и `Code= 7533`.

**4.**  $\text{index} = ((7533 - 0 + 1) \times 10 - 1) / (9999 - 0 + 1) = 7.5339 \rightarrow 7$ . Символ «S» выбран.

$\text{Low}=0+(9999-0+1) \times 5/10 = 5000$ ,  $\text{High}=0+(9999-0+1) \times 10/10-1 = 9999$ .

**5.**  $\text{index} = ((7533 - 5000 + 1) \times 10 - 1) / (9999 - 5000 + 1) = 5.0678 \rightarrow 5$ . Символ «S» выбран.

$\text{Low}= 5000 + (9999 - 5000 + 1) \times 5/10 = 7500$ ,  $\text{High}=5000+(9999-5000+1) \times 10/10-1 = 9999$ .

**6.**  $\text{index} = ((7533 - 7500 + 1) \times 10 - 1) / (9999 - 7500 + 1) = 0.1356 \rightarrow 0$ . Символ «\_» выбран.

$\text{Low} = 7500 + (9999 - 7500 + 1) \times 0/10 = 7500$ ,  $\text{High} = 7500 + (9999 - 7500 + 1) \times 1/10 - 1 = 7749$ .

После сдвига и выхода цифры 7 имеем  $\text{Low} = 5000$ ,  $\text{High} = 7499$  и  $\text{Code} = 5337$ .

7.  $\text{index} = ((5337 - 5000 + 1) \times 10 - 1)/(7499 - 5000 + 1) = 1.3516 \rightarrow 1$ . Символ «М» выбран.

$\text{Low} = 5000 + (7499 - 5000 + 1) \times 1/10 = 5250$ ,  $\text{High} = 5000 + (7499 - 5000 + 1) \times 2/10 - 1 = 5499$ .

После сдвига и выхода цифры 5 имеем  $\text{Low} = 2500$ ,  $\text{High} = 4999$  и  $\text{Code} = 3375$ .

8.  $\text{index} = ((3375 - 2500 + 1) \times 10 - 1)/(4999 - 2500 + 1) = 3.5036 \rightarrow 3$ . Символ «I» выбран.

$\text{Low} = 2500 + (4999 - 2500 + 1) \times 2/10 = 3000$ ,  $\text{High} = 2500 + (4999 - 2500 + 1) \times 4/10 - 1 = 3499$ .

После сдвига и выхода цифры 3 имеем  $\text{Low} = 0000$ ,  $\text{High} = 4999$  и  $\text{Code} = 3750$ .

9.  $\text{index} = ((3750 - 0 + 1) \times 10 - 1)/(4999 - 0 + 1) = 7.5018 \rightarrow 7$ . Символ «S» выбран.

$\text{Low} = 0 + (4999 - 0 + 1) \times 5/10 = 2500$ ,  $\text{High} = 0 + (4999 - 0 + 1) \times 10/10 - 1 = 4999$ .

10.  $\text{index} = ((3750 - 2500 + 1) \times 10 - 1)/(4999 - 2500 + 1) = 5.0036 \rightarrow 5$ . Символ «S» выбран.

$\text{Low} = 2500 + (4999 - 2500 + 1) \times 5/10 = 3750$ ,  $\text{High} = 2500 + (4999 - 2500 + 1) \times 10/10 - 1 = 4999$ .

Теперь ясно, что символ eof следует добавлять в исходную таблицу частот и вероятностей. Этот символ будет кодироваться и декодироваться последним, что будет служить сигналом для остановки процесса.

### 1.7.2. Потеря значащих цифр

В табл. 1.35 даны шаги кодирования строки *азазазазаз*. Она похожа на табл. 1.34 и иллюстрирует проблему потери значащих разрядов. Переменные *Low* и *High* сближаются, и поскольку в этом примере *Low* всегда равна 0, переменная *High* теряет свои значащие цифры при приближении к *Low*.

Потеря значащих цифр происходит не только в этом случае, но всегда, когда *Low* и *High* должны близко сходиться. Из-за своего конечного размера переменные *Low* и *High* могут достигнуть значений, скажем 499996 и 500003, а затем, вместо того, чтобы получить значения с одинаковыми старшими цифрами, они станут равны 499999

и 500000. Поскольку самые значимые цифры остались разными, алгоритм ничего не даст на выход, не будет сдвигов, и следующая итерация только добавит цифры в младший разряд переменных. Старшие цифры будут потеряны, а первые 6 цифр не изменятся. Алгоритм будет работать, не производя выходных цифр, пока не достигнет eof.

Решить эту проблему можно, если заранее распознать эту ситуацию и поменять масштаб обеих переменных. В приведенном примере это следует сделать, когда обе переменные достигнут значений 49xxxx и 50yyyy. Необходимо *удалить вторую значащую цифру*, получить значения 4xxxx0 и yyyy9, а затем увеличить счетчик cntr. Возможно, смену масштаба придется делать несколько раз до тех пор, пока не сравняются самые значащие цифры. После этого самая значащая цифра (это будет 4 или 5) подается на выход, за которой следует cntr нулей (если переменные сходятся к 4) или девяток (если они стремятся к 5).

1	2	3	4	5
1	$L = 0 + (1 - 0) \times 0.0$	= 0.0	000000	0 000000
	$H = 0 + (1 - 0) \times 0.023162$	= 0.023162	023162	0 231629
2	$L = 0 + (0.0231629 - 0) \times 0.0$	= 0.0	00000	0 000000
	$H = 0 + (0.0231629 - 0) \times 0.023162$	= 0.000536478244	005364	0 053649
3	$L = 0 + (0.053649 - 0) \times 0.0$	= 0.0	000000	0 000000
	$H = 0 + (0.053649 - 0) \times 0.023162$	= 0.00124261813	001242	0 012429
4	$L = 0 + (0.012429 - 0) \times 0.0$	= 0.0	000000	0 000000
	$H = 0 + (0.012429 - 0) \times 0.023162$	= 0.00028788049	000287	0 002879
5	$L = 0 + (0.002879 - 0) \times 0.0$	= 0.0	000000	0 000000
	$H = 0 + (0.002879 - 0) \times 0.023162$	= 0.00006668339	000066	0 000669

Табл. 1.35. Кодирование «азазазазаз» сдвигами.

### 1.7.3. Заключительные замечания

Во всех приведенных примерах мы использовали десятичную систему исчисления для лучшего понимания арифметического метода сжатия. Оказывается, что все эти алгоритмы и правила применимы и к двоичному представлению чисел с одним изменением: каждое появление цифры 9 надо заменить цифрой 1 (наибольшей двоичной цифрой).

Может показаться, что приведенные выше примеры не производят никакого сжатия, и во всех трех рассмотренных примерах строки «SWISS\_MISS», «a<sub>2</sub>a<sub>2</sub>a<sub>1</sub>a<sub>3</sub>a<sub>3</sub>» и «азазазазeof» кодируются слишком

длинными последовательностями. Похоже, что длина окончательного кода сжатия зависит от вероятностей символов. Длинные числа вероятностей табл. 1.27а породят длинные цифры при кодировании, а короткие цифры табл. 1.24 породят более разумные значения переменных *Low* и *High* табл. 1.25. Это поведение требует дополнительных разъяснений.

Для того, чтобы выяснить степень компрессии, достигаемую с помощью арифметического сжатия, необходимо рассмотреть два факта: (1) на практике все операции совершаются над двоичными числами, поэтому следует переводить все результаты в двоичную форму перед тем, как оценивать эффективность компрессии; (2) поскольку последним кодируемым символом является eof, окончательный код может быть любым числом между *Low* и *High*. Это позволяет выбрать более короткое число для окончательного сжатого кода.

Табл. 1.25 кодирует строку «SWISS\_MISS» некоторым числом в интервале от *Low*=0.71753375 до *High*=0.717535, чьи двоичные значения будут приблизительно равны 0.10110111101100000100101010111 и 0.101101111011000001011111011. Тогда декодер может выбрать строку «10110111101100000100» в качестве окончательного сжатого кода. Стока из 10 символов сжата в строку из 20 битов. Хорошее ли это сжатие?

Ответ будет «да». Используя вероятности из табл. 1.24, легко вычислить вероятность строки «SWISS\_MISS». Она равна  $P = 0.5^5 \times 0.1 \times 0.2^2 \times 0.1 \times 0.1 = 1.25 \times 10^{-6}$ . Энтропия этой строки  $-\log_2 P = -\log_2 (1.25 \times 10^{-6}) = 19.6096$ . Значит, двадцать бит – это минимальное число, необходимое для практического кодирования этой строки.

Вероятности символов из табл. 1.27а равны 0.975, 0.001838 и 0.023162. Эти величины требуют довольно много десятичных цифр для записи, а конечные значения *Low* и *High* в табл. 1.28 равны 0.99462270125 и 0.994623638610941. Опять кажется, что тут нет никакого сжатия, однако анализ энтропии показывает отличное сжатие и в этом случае.

Вычисляем вероятность строки «*a<sub>2</sub>a<sub>2</sub>a<sub>1</sub>a<sub>3</sub>a<sub>3</sub>*» и получаем число  $0.975^2 \times 0.001838 \times 0.023162^2 \approx 9.37361 \times 10^{-7}$ , а ее энтропия будет равна  $-\log_2 (9.37361 \times 10^{-7}) \approx 20.0249$ .

В двоичном представлении значения переменных *Low* и *High* равны 0.11111101001111110010111111001 и 0.111111010011111100111101. Можно выбрать любое число из этого промежутка, и мы выбираем «111111010011111100». Этот код имеет длину 19 (он и теоретически должен быть 21-битным, но числа в табл. 1.28 имеют ограниченную точность).

**Пример:** Даны три символа  $a_1$ ,  $a_2$  и eof с вероятностями  $P_1 = 0.4$ ,  $P_2 = 0.5$  и  $P_{\text{eof}} = 0.1$ . Кодируем строку « $a_2a_2a_2\text{eof}$ ». Размер конечного кода будет равен теоретическому минимуму.

Шаги кодирования просты (см. первый пример на стр. 63). Начинаем с интервала  $[0, 1]$ . Первый символ сокращает интервал до  $[0.4, 0.9]$ , второй – до  $[0.6, 0.85]$ , третий – до  $[0.7, 0.825]$ , а четвертый eof – до  $[0.8125, 0.8250]$ . Приблизительные двоичные значения концов равны 0.1101000000 и 0.1101001100, поэтому в качестве кода выбирается 7-битовое число «1101000».

Вероятность этой строки равна  $0.5^3 \times 0.1 = 0.0125$ , тогда число  $-\log_2 0.0125 \approx 6.322$ , то есть, теоретический минимум длины кода равен 7 бит. (Конец примера.)

Следующее рассуждение показывает, почему арифметическое кодирование может быть весьма эффективным методом сжатия. Обозначим через  $s$  последовательность кодируемых символов, а через  $b$  – число бит, требуемых для кодирования  $s$ . Если длина  $s$  растет, то ее вероятность  $P(s)$  уменьшается, а число  $b$  растет. Поскольку логарифм является функцией информации, легко видеть, что число  $b$  растет с той же скоростью, что  $-\log_2 P(s)$  убывает. Значит, их произведение остается постоянным. В теории информации показано, что

$$2 \leq 2^b P(s) \leq 4,$$

что влечет неравенство

$$1 - \log_2 P(s) \leq b \leq 2 - \log_2 P(s). \quad (1.1)$$

Когда  $s$  растет, ее вероятность уменьшается, а число  $-\log_2 P(s)$  становится большим положительным числом. Двойное неравенство (1.1) показывает, что в пределе  $b$  стремится к  $-\log_2 P(s)$ . Вот почему арифметическое кодирование может, в принципе, сжимать строку до ее теоретического предела.

## 1.8. Адаптивное арифметическое кодирование

Две характерные черты арифметического кодирования позволяют легко расширить этот метод сжатия.

1. Основной шаг кодирования состоит из двух операций

```
Low := Low+(High-Low+1)*LowCumFreq[X]/10;
High := Low+(High-Low+1)*HighCumFreq[X]/10-1;
```

Это означает, что для кодирования символа  $X$  необходимо сообщить кодеру накопленные частоты этого символа и символа, расположенного над ним (см. табл. 1.24 с примером накопленных частот). Но тогда частота  $X$  (или ее вероятность) может изменяться каждый раз после кодирования, при условии, что кодер и декодер будут это делать согласованно.

2. Порядок символов в табл. 1.24 не имеет значения. Символы могут даже менять свое место в таблице, если это делать согласованно с декодером.

Имея это в виду, легко понять, как может работать процесс адаптивного арифметического кодирования. Алгоритм кодирования состоит из двух частей: вероятностная модель и арифметический кодер. Вероятностная модель читает следующий символ из входного файла и вызывает кодер, сообщая ему символ и две текущие накопленные частоты. Затем модель увеличивает на единицу счетчик символов и изменяет накопленные частоты. Главным здесь является то, что вероятности символов определяются моделью по *старым* значениям счетчиков, которые увеличиваются только *после* кодирования данного символа. Это позволяет декодеру делать зеркальные действия. Кодер знает символ, который ему предстоит закодировать, а декодер должен его распознать по сжатому коду, поэтому декодер знает только старые значения счетчиков, но может увеличивать и изменять их значения точно так же как и кодер.

Модель должна накапливать символы, их счетчики (частоты появления) и накопленные частоты в некотором массиве. Этот массив следует хранить упорядоченно по значениям счетчиков символов. При чтении очередного символа происходит увеличение его счетчика, затем модель обновляет накопленные частоты и смотрит, надо ли менять порядок символов для соблюдения упорядоченности всего массива.

$a_1$	$a_2$	$a_3$	$a_4$	$a_5$	$a_6$	$a_7$	$a_8$	$a_9$	$a_{10}$
11	12	12	2	5	1	2	19	12	8
(a)									
$a_8$	$a_2$	$a_3$	$a_9$	$a_1$	$a_{10}$	$a_5$	$a_4$	$a_7$	$a_6$
19	12	12	12	11	8	5	2	2	1
(b)									

Табл. 1.36. Алфавит из 10 символов и их счетчики.

Оказывается, что такой массив легко организовать в виде структуры, которая позволит делать поиск и обновление этого массива.

Такой структурой является двоичное сбалансированное дерево. (Сбалансированным двоичным деревом служит полное дерево, в котором некоторые правые нижние узлы отсутствуют.) На дереве должны быть узлы для каждого символа алфавита, и поскольку оно сбалансировано, его высота равна  $\lceil \log_2 n \rceil$ , где  $n$  – размер алфавита. При  $n = 256$  высота сбалансированного дерева равна 8, поэтому начав с корня, поиск символа займет не более 8 шагов. Дерево организовано так, что более вероятный символ (у которого самый большой счетчик) находится ближе к корню дерева, что ускоряет процесс его поиска. В табл. 1.36а показан пример алфавита из 10 символов вместе со счетчиками. В табл. 1.36б этот же алфавит упорядочен по значениям счетчиков.

Упорядоченный массив размещен на дереве, изображенном на рис. 1.38а. Это простой и элегантный способ построения дерева. Сбалансированное двоичное дерево можно построить без использования указателей. Правило такое: первый элемент массива (с индексом 1) находится в корне, два узла-потомка элемента с индексом  $i$  имеют индексы  $2i$  и  $2i + 1$ , а родительский узел узла  $j$  имеет индекс  $\lfloor j/2 \rfloor$ . Легко видеть, как процесс упорядочения массива разместит символы с большими счетчиками ближе к корню.

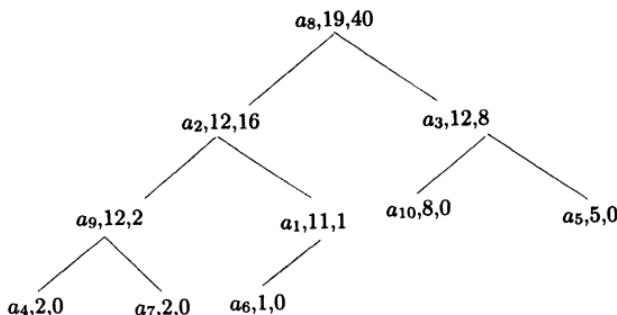
В дополнение к символу и его счетчику в каждом узле дерева будет храниться общая сумма счетчиков его левого поддерева. Эти величины будут использоваться при подсчете накопленных частот. Соответствующий массив показан в табл. 1.37а.

$a_8$	$a_2$	$a_3$	$a_9$	$a_1$	$a_{10}$	$a_5$	$a_4$	$a_7$	$a_6$
19	12	12	12	11	8	5	2	2	1
40	16	8	2	1	0	0	0	0	0
(a)									
$a_8$	$a_9$	$a_3$	$a_2$	$a_1$	$a_{10}$	$a_5$	$a_4$	$a_7$	$a_6$
19	13	12	12	11	8	5	2	2	1
41	16	8	2	1	0	0	0	0	0
(b)									

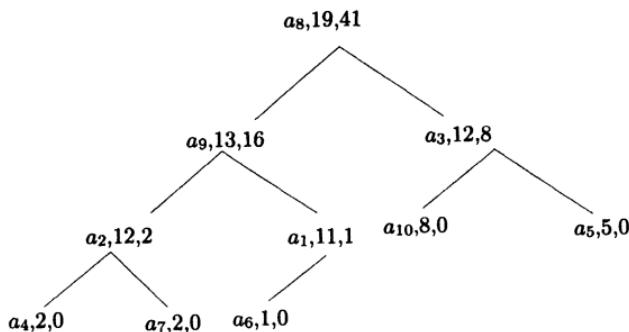
Табл. 1.37. Алфавит из 10 символов и их счетчики.

Предположим, что следующий прочитанный из входного файла символ был  $a_9$ . Его счетчик изменяется с 12 до 13. Модель, имея упорядоченный массив, ищет самый дальний слева от  $a_9$  элемент, у которого счетчик меньше или равен счетчику  $a_9$ .

Этот поиск может быть просто линейным, если массив короткий, или двоичным, если массив длинный. В нашем случае это бу-



(a)



(b)

$a_4$	2	0—1
$a_9$	12	2—13
$a_7$	2	14—15
$a_2$	12	16—27
$a_6$	1	28—28
$a_1$	11	29—39
$a_8$	19	40—58
$a_{10}$	8	59—66
$a_3$	12	67—78
$a_5$	5	79—83

(c)

Рис. 1.38. Адаптивное арифметическое кодирование.

дет элемент  $a_2$ , который следует переставить с  $a_9$  (табл. 1.37b). На рис. 1.38b показано дерево после этой перестановки. Обратите внимание на то, как меняются счетчики левых поддеревьев.

Наконец покажем, как вычисляются накопленные частоты с помощью этого дерева. Когда необходимо найти эту величину для символа  $X$ , модель идет по веткам из корня до узла содержащего  $X$ , добавляя числа в переменную  $af$ . Каждый раз, когда выбирается правая ветвь из внутреннего узла  $N$ , к переменной  $af$  добавляются два числа (счетчик символа и счетчик левого дерева), лежащие в этом узле. При выборе левой ветви переменная  $af$  не меняется. Когда узел, содержащий  $X$  найден, счетчик левого дерева этого узла добавляется к  $af$ . Теперь значение переменной  $af$  равно в точности величине  $LowCumFreq[X]$ .

Для примера проследим на рис. 1.38a из корня дерева до символа  $a_6$ , чья накопленная частота равна 28. Правая ветвь выбрана в узле  $a_2$ , поэтому в  $af$  добавлена сумма  $12+16$ . В узле  $a_1$  выбрана левая ветвь, поэтому к  $af$  ничего не добавлено. В результате  $af=12+16=28$ , что можно проверить на рис. 1.38c. Величина  $HighCumFreq[X]$  получается прибавлением счетчика  $a_6$  (равного 1) к  $LowCumFreq[X]$ .

При проходе по дереву от корня до символа  $a_6$  алгоритм выполняет следующие действия.

1. Находит  $a_6$  в массиве с помощью двоичного поиска по дереву. В нашем примере  $a_6$  находится в позиции 10 массива.
2. Делит 10 на 2. Остаток равен 0. Это означает, что  $a_6$  является левым потомком своего родителя. Частное 5 равно индексу в массиве его родителя.
3. Находит в позиции 5 символ  $a_1$ . Делит 5 пополам. Остаток 1 говорит о том, что  $a_1$  является правым потомком своего родителя, имеющего индекс 2.
4. Элемент массива с индексом 2 равен  $a_2$ . Алгоритм делит 2 пополам. Остаток равен 0, то есть,  $a_2$  – левый потомок родителя, который имеет индекс 1 и находится в корне, поэтому процесс останавливается.

Метод компрессии PPM, описанный в [Cleary, Witten 84] и [Moffat 90], является хорошим примером статистической модели, использующей арифметическое кодирование.

*Статистика подобна бикини. То, что она показывает, соблазнительно, а то, что скрывает — жизненно важно.*

— Аарон Левинштейн

## ГЛАВА 2

# СЛОВАРНЫЕ МЕТОДЫ

Статистические методы компрессии используют статистическую модель данных, и качество сжатия информации напрямую зависит от того, насколько хороша была эта модель. Методы, основанные на словарном подходе, не рассматривают статистические модели, они также не используют коды переменной длины. Вместо этого они выбирают некоторые последовательности символов, сохраняют их в словаре, а все последовательности кодируются в виде *меток*, используя словарь. Словарь может быть статическим или динамическим (адаптивным). Первый является постоянным; иногда в него добавляют новые последовательности, но никогда не удаляют. Динамический словарь содержит последовательности, ранее поступившие из входного файла, при этом разрешается и добавление, и удаление данных из словаря по мере чтения входного файла.

Простейшим примером статического словаря может служить словарь английского языка, используемый для сжатия английских текстов. Представьте себе словарь, в котором содержится полмиллиона слов (без их описания или перевода). Слово (последовательность символов, заканчивающаяся пробелом или знаком препинания) читается из входного файла и ищется в словаре. Если оно найдено в словаре, его индекс, или словарная метка, записывается в выходной файл. В противном случае записывается само это слово без сжатия. (Это пример *логического сжатия*.)

В результате выходной файл состоит из индексов и рядов слов, поэтому необходимо уметь делать различие между ними. Один возможный способ – это добавление к записываемому в файл образчику еще один дополнительный бит. В принципе, 19-битовый индекс будет достаточен для точного указания слов в словаре объема  $2^{19} = 524288$  элементов. Поэтому, если прочитанное слово найдено в словаре, то можно записать 20-битную ссылку, состоящую из флагового бита (возможно, нулевого), за которым следует 19 битов индекса. Если данное слово не обнаружено в словаре, записывается флаг 1, потом длина слова и, наконец, само это слово.

**Пример:** Предположим, что слово `bet` находится в словаре под номером 1025. Тогда оно будет закодировано 20-битовым числом `0|00000000100000000001`. Пусть слово `het` не обнаружено в словаре. Тогда в выходной файл будет записана последовательность битов `1|0000011|01111000|01100101|01110100`. В этом четырехбайтовом числе 7-битовое поле `0000011` означает, что за ним следуют еще три байта.

Если считать, что размер записывается в виде 7-битового числа и что средний размер слова состоит из 5 букв, то несжатое слово будет в среднем занимать 6 байт (= 48 бит) в выходном файле. Сжатие 48 бит в 20 является замечательным, при условии, что это делается достаточно часто. При этом, возникает вопрос: «Сколько совпадений слов надо обнаруживать в словаре, чтобы иметь общее сжатие?». Пусть вероятность обнаружения слова в словаре равна  $P$ . После чтения и сжатия  $N$  слов размер выходного файла будет равен  $N(20P + 48(1 - P)) = N(48 - 28P)$ . Размер входного файла равен (при условии, что слово имеет 5 букв)  $40N$ . Тогда сжатие будет достигаться, если  $N(48 - 28P) < 40N$ , то есть, при  $P > 0.29$ . Следовательно, надо иметь 29% или больше успешных нахождений слов в словаре для положительного сжатия.

**Пример:** Если вероятность обнаружения выше, например,  $P = 0.9$ , то размер выходного файла будет равен  $N(48 - 28P) = N(48 - 25.2P) = 22.8N$ . Размер входного файла равен, как и раньше,  $40N$ . Фактор сжатия равен  $40/22.8 \approx 1.75$ .

До тех пор, пока входной файл содержит английский текст, большинство слов будут обнаружены в полмиллионном словаре. Для других типов данных, разумеется, это не будет выполняться. В файле, содержащем код компьютерной программы будут находиться «слова» типа `cout`, `xor`, `malloc`, которые, возможно, не удастся обнаружить в словаре английского языка. А бинарный файл, если его рассматривать в ASCII кодах, обычно, содержит всякий «мусор», который едва ли удастся обнаружить в этом словаре. В результате произойдет расширение вместо сжатия.

Все это говорит о том, что статический словарь – не самый лучший выбор для общего алгоритма сжатия. Однако в ряде специальных случаев, это будет хорошим методом. Возьмем сеть компьютерных магазинов. Их файлы во множестве будут содержать слова типа `nut`, `bolt`, `paint`. В то же время, слова `peanut`, `lightning`, `painting` будут встречаться реже. Поэтому специальное компрессионное программное обеспечение этой фирмы может использовать очень маленький специализированный словарь, состоящий всего из

нескольких сотен слов. Каждый сетевой компьютер будет иметь копию этого словаря, что позволит легко сжимать файлы и пересыпать их по сети между магазинами и офисами фирмы.

В общем случае, использование динамического словаря является более предпочтительным. Такой метод сжатия начинает с пустого или маленького исходного словаря, добавляет в него слова по мере поступления из входного файла и удаляет старые слова, поскольку поиск по большому словарю делается медленно. Такой алгоритм состоит из цикла, каждая итерация которого начинается с чтения входного файла и с его (грамматического) разбиения на слова и фразы. Потом делается поиск каждого слова в словаре. Если слово там найдено, то в выходной файл пишется соответствующая ему метка (индекс, указатель, ярлык и т.п.). В противном случае в выходной файл записывается несжатое слово и оно же добавляется в словарь. На последнем шаге делается проверка, не надо ли удалить из словаря старое слово. Все это может показаться сложным, однако имеется два важных преимущества.

1. Алгоритм использует только операции над последовательностями строк типа поиска и сравнения и не использует арифметические вычисления.

2. Декодер очень прост (значит, мы имеем дело с асимметричным методом компрессии). В статистическом методе сжатия декодер в точности противоположен кодеру (делается симметричная компрессия). В адаптивном методе словарного сжатия декодер должен прочитать файл, распознать, является ли прочитанный образчик меткой или несжатыми данными, использовать метку для извлечения данных из словаря и выдать исходный разжатый файл. Не придется производить сложный грамматический разбор входного файла, также не нужно делать поиск по словарю для обнаружения совпадений. Многие программисты тоже не любят делать это.

Большая часть для ученого, когда его имя присваивается научному открытию, методу или явлению. Еще более почетно, когда имя присваивается целому научному направлению. Как раз это случилось с Якобом Зивом (Jacob Ziv) и Абрахамом Лемпэлом (Abraham Lempel). В 1970 году эти ученые разработали два первых метода, называемые LZ77 и LZ78, для словарного сжатия данных. Их замечательные идеи вдохновили многих исследователей, которые стали улучшать, обобщать и комбинировать их с другими методами (например, с RLE или со статистическим методом) для создания многих широко используемых на практике алгоритмов сжатия без потери информации для компрессии текстов, изображений и звука.

Далее в этой главе описано несколько общих LZ методов компрессии и показано, как они развивались из основополагающих идей Зива и Лемпзла.

## 2.1. LZ77 (скользящее окно)

Основная идея этого метода (его еще часто называют методом LZ1, см. [Ziv 77]) состоит в использовании ранее прочитанной части входного файла в качестве словаря. Кодер создает окно для входного файла и двигает его справа налево в виде строки символов, требующих сжатие. Таким образом, метод основан на *скользящем окне*. Окно разбивается на две части. Часть слева называется *буфером поиска*. Она будет служить текущим словарем, и в ней всегда содержатся символы, которые недавно поступили и были закодированы. Правая часть окна называется *упреждающим буфером*, содержащим текст, который будет сейчас закодирован. На практике буфер поиска состоит из нескольких тысяч байт, а длина упреждающего буфера равна нескольким десяткам символов. Вертикальная черта | между символами `t` и `e` означает текущее разделение между двумя буферами. Итак, предположим, что текст `sir.sid.eastman.easily.t` уже сжат, а текст `eases.sea.sick.seals` нуждается в сжатии.

← выход ... sir.sid.eastman.easily.t eases.sea.sick.seals ... ← вход

Кодер просматривает буфер поиска в обратном порядке (справа налево) и ищет в нем первое появление символа `e` из упреждающего буфера. Он обнаруживает такой символ в начале слова `easily`. Значит, `e` находится (по смещению) на расстоянии 8 от конца буфера поиска. Далее кодер определяет, сколько совпадающих символов следует за этими двумя символами `e`. В данном случае совпадение по символам `eas` имеет длину 3. Кодер продолжает поиск, пытаясь найти более длинные совпадающие последовательности. В нашем случае имеется еще одна совпадающая последовательность той же длины 3 в слове `eastman` со смещением 16. Декодер выбирает самую длинную из них, а при совпадении длин – самую удаленную, и готовит метку `(16, 3, «e»)`.

Выбор более удаленной ссылки упрощает кодер. Интересно отметить, что выбор ближайшего совпадения, несмотря на некоторое усложнение программы, также имеет определенные преимущества. При этом выбирается наименьшее смещение. Может показаться, что в этом нет преимущества, поскольку в метке будет предусмотрено

достаточно разрядов для хранения самого большого смещения. Однако, можно следовать LZ77 с кодированием Хаффмана или другого статистического кодирования меток, при котором более коротким смещениям присваиваются более короткие коды. Этот метод, предложенный Бернардом Хердом (Bernard Herd), называется LZH. Если получается много коротких смещений, то при LZH достигается большее сжатие.

Следующую идею очень легко усвоить. Декодер не знает, какое именно совпадение выбрал кодер, ближайшее или самое дальнее, но ему это и не надо знать! Декодер просто читает метки и использует смещение для нахождения текстовой строки в буфере поиска. Для него не важно, первое это совпадение или последнее.

В общем случае, метка из LZ77 имеет три поля: смещение, длина и следующий символ в упреждающем буфере (в нашем случае, это будет **второе e** в слове **teases**). Эта метка записывается в выходной файл, а окно сдвигается вправо (или входной файл сдвигается влево) на четыре позиции: три позиции для совпадающей последовательности и одна позиция для следующего символа.

... sir \_sid\_eastman\_easily\_tease s\_sea\_sick\_seals... ...

Если обратный поиск не обнаружил совпадение символов, то записывается метка со смещением 0 и длиной 0. По этой же причине метка будет иметь третью компоненту. Метки с нулевыми смещениями и длиной всегда стоят в начале работы, когда буфер поиска пуст или почти пуст. Первые семь шагов кодирования нашего примера приведены ниже.

	sir_sid_eastman_r	$\Rightarrow (0,0,«s»)$
s	ir_sid_eastman_e	$\Rightarrow (0,0,«i»)$
si	r_sid_eastman_ea	$\Rightarrow (0,0,«r»)$
sir	_sid_eastman_eas	$\Rightarrow (0,0,«_»)$
sir_	sid_eastman_easi	$\Rightarrow (4,2,«d»)$
sir_sid	_eastman_easily	$\Rightarrow (4,1,«e»)$
sir_sid_e	astman_easily_te	$\Rightarrow (0,0,«a»)$

Ясно, что метки вида  $(0,0,\dots)$ , которые кодируют единичные символы, не обеспечивают хорошее сжатие. Легко оценить их длину. Размер смещения равен  $\lceil \log_2 S \rceil$ , где  $S$  – длина буфера поиска. На практике этот буфер имеет длину в несколько сотен байт, поэтому смещение имеет длину 10 – 12 бит. Поле «длина» имеет размер

равный  $\lceil \log_2(L - 1) \rceil$ , где  $L$  – длина упреждающего буфера (в следующем абзаце будет объяснено почему надо вычесть 1). Обычно упреждающий буфер имеет длину нескольких десятков байтов. Поэтому длина этого поля равна нескольким битам. Размер поля «символ», обычно, равен 8 бит, но в общем случае –  $\lceil \log_2 A \rceil$ , где  $A$  – размер алфавита. Полный размер метки  $(0,0,\dots)$  одиночного символа равен  $11+5+8=24$  бит, что много больше длины 8 «сырого» символа без нулевых элементов кода.

Следующий пример показывает, почему поле «длина» может быть длиннее размера упреждающего буфера:

...Mr..	alf_eastman_easily_grows.alf	alfa_in_his_	garden...
---------	------------------------------	--------------	-----------

Первый символ **a** в упреждающем буфере совпадает с 5-ым **a** буфера поиска. Может показаться, что оба крайних **a** подходят с длиной совпадения 3, поэтому кодер выберет самый левый символ и создаст метку  $(28,3,«a»)$ . На самом деле кодер образует метку  $(3,4,«_»)$ . Страна из четырех символов **«alfa»** из упреждающего буфера совпадает с тремя символами **«alf»** буфера поиска и первым символом **«a»** упреждающего буфера. Причина этого заключается в том, что декодер может обращаться с такой меткой очень естественно безо всяких модификаций. Он начинает с позиции 3 буфера поиска и копирует следующие 4 символа, один за другим, расширяя буфер вправо. Первые 3 символа копируются из старого содержимого буфера, а четвертый является копией первого из этих новых трех. Следующий пример еще более убедителен (хотя он немного надуман):

... alf_eastman_easily_yells_A	AAAAAAA	AAAAN...
--------------------------------	---------	----------

Кодер создает метку  $(1,9,«A»)$  для совпадения первых девяти копий символа **«A»** в упреждающем буфере, включая десятый символ **A**. Вот почему длина совпадения, в принципе, может быть равна размеру упреждающего буфера минус 1.

Декодер метода LZ77 гораздо проще кодера (то есть, метод LZ77 является асимметричным методом сжатия). Он должен соорудить такой же буфер поиска, как и кодер. Декодер вводит метку, находит совпадение в своем буфере, записывает совпадающие символы и символ из третьего поля в буфер. Этот метод или его модификация хорошо работает, если файл сжимается один раз (или несколько раз), но часто разжимается. Часто используемый архив старых сжатых файлов может служить хорошим примером использования этого алгоритма.



На первый взгляд в описанном методе сжатия не делается никаких предположений относительно входных данных. В частности, не обращается внимание на частоты появления отдельных символов. Однако, немного подумав, видно, что из-за природы скользящего окна метод LZ77 всегда сравнивает упреждающий буфер с недавним содержимым буфера поиска, но не с данными, поступившими очень давно (и которые уже покинули буфер поиска). Следовательно, неявно предполагается, что похожие участки входного файла повторяются близко друг от друга. Данные, удовлетворяющие этому свойству будут хорошо сжиматься.

Базисный метод LZ77 улучшался исследователями и программистами многими способами в течение 80-х и 90-х годов прошлого века. Один возможный путь – это использование кодов переменной длины при кодировании полей смещения и длины в метке. Другой путь – увеличение размеров обоих буферов. Увеличение буфера поиска дает возможность искать больше совпадений, но ценой будет служить увеличение времени поиска. Очевидно, большой буфер поиска потребует более изощренной структуры данных для ускорения поиска (см. § 2.4.2). Третье улучшение относится к скользящему окну. Простейший подход заключается в перемещении всего текста влево после каждого совпадения. Более быстрый прием заключается в замене линейного окна на *циклическую очередь*, в которой скольжение окна делается переустановкой двух указателей (см. § 2.1.1). Еще одно улучшение состоит в добавлении одного бита (флага) в каждую метку, что исключает третье поле (см. § 2.2).

### 2.1.1. Циклическая очередь

Циклическая очередь является важной структурой данных. Физически это массив, но его индекс используется особым способом. Рис. 2.1 иллюстрирует простой пример. На нем показан массив из 16 байт с символами, из которых одни добавлены в «конец», а другие – удалены из «начала». Обе позиция конца и начала перемещаются, и два указателя **s** и **e** все время на них указывают. На рис. (a) имеется 8 символов **sid\_east**, а остаток буфера пуст. На рис. (b) все 16 символов заняты, а **e** указывает на конец буфера. На (c) первая буква **s** была удалена, а буква **l** в **easily** была вставлена. Заметьте, что указатель **e** теперь размещается слева от **s**. На рис. (d) две буквы **id** были удалены просто перемещением указателя **s**; сами символы все еще находятся в массиве, но они уже эффективно удалены из очереди. На рис. (e) два символа **u** – были добавлены в очередь и указатель **e** переместился. На рис. (f) указатели показывают, что буфер кон-

чается на `teas`, а начинается на `tman`. Добавление новых символов в циклическую очередь и перемещение указателей эквивалентно перемещению содержимого очереди. Итак, нет необходимости что-то двигать или перемещать.

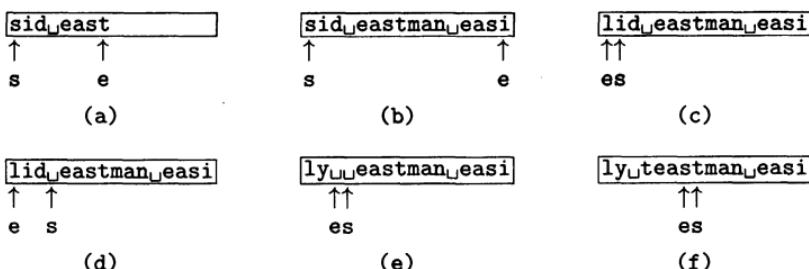


Рис. 2.1. Циклическая очередь.

## 2.2. LZSS

Эта версия LZ77 была разработана Сторером (Storer) и Сжимански (Szymanski) в 1982 [Storer 82]. Базовый алгоритм был улучшен по трем направлениям: (1) упреждающий буфер сохранялся в циклической очереди, (2) буфер поиска (словарь) хранился в виде дерева двоичного поиска и (3) метки имели два поля, а не три.

Двоичное дерево поиска – это двоичное дерево, в котором левое поддерево каждого узла  $A$  содержит узлы, меньшие чем  $A$ , а узлы правого поддерева все больше  $A$ . Поскольку узлы нашего двоичного дерева состоят из строк (или слов), прежде всего необходимо определиться, как эти строки следует сравнивать, что значит, одна строка «больше» другой. Это легко понять, представив себе строки в словаре, где они упорядочены по алфавиту. Ясно, что строка `rote` предшествует строке `said`, так как буква `r` стоит раньше буквы `s` (хотя `o` и следует после `a`). Мы будем считать, что строка `rote` меньше строки `said`. Такое упорядочение строк называется *лексикографическим*.

А как быть со строкой «`abc`»? Фактически все современные компьютеры используют код ASCII для представления символов (хотя все большее распространение получают коды Unicode, о которых говорится на стр. 90, а некоторые старые большие компьютеры IBM, Amdahl, Fujitsu, Siemens работают со старыми 8-битными кодами EBCDIC, разработанными в IBM). В кодах ASCII символ пробе-

ла предшествует символам букв, поэтому строка, начинающаяся с пробела будет считаться меньше любой строки, в начале которой стоит буква. В общем случае *сортировку последовательностей* в компьютере определяет последовательность символов, упорядоченных от меньшего к большему. На рис. 2.2 показаны два примера двоичных деревьев поиска.

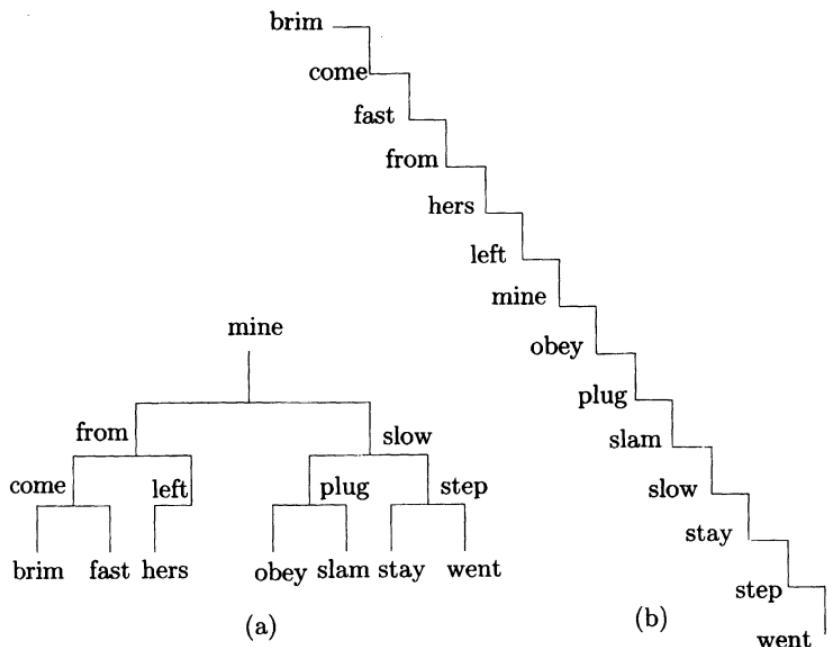


Рис. 2.2. Двоичные деревья поиска.

Обратите внимание на различие между (почти) сбалансированным деревом (а) и косым деревом (б). Эти деревья состоят из одного числа 14 узлов, но они устроены и ведут себя совершенно по-разному. В сбалансированном дереве любой узел можно достичь не более, чем в 4 шага, а в косом дереве для этого может потребоваться до 14 шагов. В любом случае максимальное число шагов потребуется для того, чтобы достичь узлы, удаленные от корня на полную высоту дерева. Для косого дерева (которое, на самом деле, эквивалентно присоединенному списку), высота дерева – это число его элементов  $n$ , а для сбалансированного дерева его высота равна  $\lceil \log_2 n \rceil$ , что существенно меньше. Более подробные сведения о двоичных деревьях поиска можно найти в любом учебнике по структурам данных.

**Unicode**

Новый международный стандартный коде Unicode был разработан и предложен для употребления международной организацией Unicode ([www.unicode.org](http://www.unicode.org)). В этом коде используются 16-битные последовательности для кодирования символов, что дает  $2^{16} = 64K = 65536$  различных символов. Unicode содержит все ASCII коды плюс коды для букв иностранных языков (включая такие сложные алфавиты, как корейский, китайский и японский), а кроме того многие математические символы и другие знаки. К настоящему времени около 39000 из 65536 кодов получили назначение, но еще остается много места для добавления новых символов в будущем.

Операционная система Microsoft Windows NT поддерживает коды Unicode.

*Единственное место, где успех приходит  
до работы – это в словаре*

— Видал Сассон

**Пример:** Покажем, как двоичное дерево способно ускорить поиск в словаре. Пусть входной файл содержит следующую последовательность: «`sid_eastman_clumsily_teases_sea_sick_seals`». Для простоты предположим, что окно состоит из 16-байтного буфера поиска и 5-байтного упреждающего буфера. После ввода первых  $16+5$  символов скользящее окно выглядит так:

<code>sid_eastman_clum</code>	<code>sily_</code>	<code>teases_sea_sick_seals</code> ,
-------------------------------	--------------------	--------------------------------------

причем подпоследовательность `teases_sea_sick_seals` ждет своей очереди на входе.

Кодер изучает буфер поиска, создавая двенадцать строк по пять символов (см. табл. 2.3) (их двенадцать, так как  $12 = 16 - 5 + 1$ ), которые помещены на двоичное дерево поиска вместе с их смещениями.

Первым символом в упреждающем буфере является `s`, поэтому кодер ищет на дереве строки, начинающиеся на `s`. Он находит две строки со смещениями 16 и 10, но первая из них, `sid_e`, имеет более длинное совпадение.

(Здесь необходимо отвлечься и обсудить случай, когда строка на дереве полностью совпадает с содержимым упреждающего буфера. Тогда кодер мог бы искать дальнейшие совпадения. В принципе, длина совпадения может быть  $L - 1$ .)

В нашем примере длина совпадения равна 2, поэтому кодер выдает метку (16,2). Теперь кодер должен переместить скользящее окно на две позиции вправо и перестроить дерево. Новое окно выглядит следующим образом:

si	d_eastman_clumsi	ly_te	ases_sea_sick_seals	.
----	------------------	-------	---------------------	---

С дерева необходимо удалить строки `sid_e` и `id_ea` и вставить новые строки `clums` и `lumsi`. Если бы случилось совпадение длины  $k$ , то дерево надо было бы перестроить путем удаления  $k$  строк и добавления  $k$  строк, но вот каких?

Немного подумав, становится ясно, что удаляться будут первые  $k$  строк буфера поиска *до* его сдвига, а добавляться будут последнее  $k$  строк этого буфера *после* сдвига. Простейшая процедура обновления дерева состоит в приготовлении строк из начала буфера, их поиска и удаления. Потом необходимо сдвинуть буфер на одну позицию вправо (или переместить данные на одну позицию влево), приготовить строку из последних 5 символов буфера поиска и добавить ее на дерево. Это следует повторить  $k$  раз. (Конец примера.)

---

<code>sid_e</code>	16
<code>id_ea</code>	15
<code>d_eas</code>	14
<code>_east</code>	13
<code>eastm</code>	12
<code>astma</code>	11
<code>stman</code>	10
<code>tman_</code>	09
<code>man_c</code>	08
<code>an_cl</code>	07
<code>n_clu</code>	06
<code>_clum</code>	05

---

Табл. 2.3. Строки по пять символов.

На дереве все время находится одинаковое число  $T$  узлов или строк, поскольку при его обновлении удаляется и добавляется одно и то же число строк. Число  $T$  равно: длина буфера поиска минус длина упреждающего буфера плюс 1 ( $T = S - L + 1$ ). Однако, форма дерева может существенно измениться. Поскольку узлы удаляются и добавляются, то форма дерева может меняться от абсолютно косой (худший случай для поиска) до сбалансированной, идеальной формы.

Третье улучшение метода LZ77 в алгоритме LZSS касалось создаваемых кодером меток. Метка LZSS состоит только из смещения и длины. Если не найдено совпадений, то кодер просто подает на выход несжатый код следующего символа вместо расточительной

метки из трех полей (0,0,...). Для различения меток и несжатых кодов используется флаговый бит.

На практике буфер поиска может состоять из нескольких сотен байт, поэтому поле смещения, обычно, состоит из 11–13 бит. Размер упреждающего буфера выбирается с таким условием, чтобы в сумме с буфером поиска длина составляла 16 бит (2 байта). Например, если буфер поиска имеет размер 2 КБ ( $= 2^{11}$ ), то упреждающий буфер состоит из 32 байт ( $= 2^5$ ). Длина поля смещения равна 11, а поле «длина» имеет 5 разрядов. При таком выборе размеров буферов кодер будет выдавать или 2-х байтовую метку или 1-байтовый несжатый код ASCII. А как насчет флага? Хороший практический совет состоит в накоплении восьми последовательных выходов (меток или ASCII кодов) в маленьком буфере, затем выдавать один байт из 8 флагов, за которым следуют восемь накопленных элементов по 2 или 1 байту.

### 2.2.1. Недостатки

Перед тем, как обсуждать алгоритм LZ78, остановимся на недостатках метода LZ77 и его вариантов. Было уже отмечено, что этот алгоритм основывается на предположении, что похожие образцы сжимаемых данных находятся близко друг от друга. Если содержимое файла не удовлетворяет этому условию, то он будет сжиматься плохо. Простой пример – это текст, в котором слово «есопому» встречается часто и равномерно распределено по всему тексту. Может случиться, что когда это слово попадает в упреждающий буфер, его предыдущая копия уже вышла из буфера просмотра. Более лучший алгоритм мог бы сохранять часто встречающиеся слова в словаре, а не сдвигал бы их все время.

Другой недостаток метода – это ограниченные размеры упреждающего буфера. Размер совпадающей строки лимитирован числом  $L - 1$ , но  $L$  приходится держать маленьким, так как процесс сравнения строк основан на сравнении индивидуальных символов. Если удвоить  $L$ , то степень сжатия могла бы возрасти, но одновременно с этим произойдет замедление процесса поиска совпадений. Размер  $S$  буфера поиска тоже ограничен. Большой буфер поиска мог бы тоже улучшить компрессию, но опять возрастет сложность поиска по дереву, даже двоичному. Увеличение буферов также означает удлинение меток, что сокращает фактор сжатия. Двухбайтовые метки сжимают последовательности из 2-х символов в два байта плюс один флаговый бит. Запись двух байтов кода ASCII в виде этого же кода плюс два флаговых бита дает весьма малую разницу в размере по

сравнению с записью метки. Это будет лучший выбор для кодера в смысле экономии времени, а плата – всего один лишний бит. Мы будем говорить, что в этом случае кодер имеет *2-х байтовую точку безызбыточности*. Для более длинных меток точка безызбыточности вырастает до 3 байтов.

## 2.3. LZ78

Метод LZ78 (иногда его называют LZ2, см. [Ziv 78]) не использует буфер поиска, упреждающий буфер и скользящее окно. Вместо этого имеется словарь встретившихся ранее строк. В начале этот словарь пуст (или почти пуст), и размер этого словаря ограничен только объемом доступной памяти. На выход кодера поступает последовательность меток, состоящих из двух полей. Первое поле – это указатель на строку в словаре, а второе – код символа. Метка не содержит длины строки, поскольку строка берется из словаря. Каждая метка соответствует последовательности во входном файле, и эта последовательность добавляется в словарь после того, как метка записана в выходной сжатый файл. Ничего из словаря не удаляется, что является одновременно и преимуществом над LZ77 (поскольку будущие строки могут совпадать даже с очень давними последовательностями) и недостатком (так как быстро растет объем словаря).

Словарь начинает строиться из пустой строки в позиции нуль. По мере поступления и кодирования символов, новые строки добавляются в позиции 1, 2 и т.д. Когда следующий символ  $x$  читается из входного файла, в словаре ищется строка из одного символа  $x$ . Если такой строки нет, то  $x$  добавляется в словарь, а на выход подается метка  $(0, x)$ . Эта метка означает строку «нуль  $x$ » (соединение нулевой строки и  $x$ ). Если вхождение символа  $x$  обнаружено (скажем, в позиции 37), то читается следующий символ  $y$ , и в словаре ищется вхождение двухсимвольной строки  $xy$ . Если такое не найдено, то в словарь записывается строка  $xy$ , а на выход подается метка  $(37, y)$ . Такая метка означает строку  $xy$ , так как позицию 37 в словаре занимает символ  $x$ . Процесс продолжается до конца входного файла.

В общем случае текущий символ читается и становится однобуквенной строкой. Затем кодер пытается найти ее в словаре. Если строка найдена, читается следующий символ и присоединяется к текущей строке, образуя двухбуквенную строку, которую кодер опять пытается найти в словаре. До тех пор пока такие строки находятся в словаре, происходит чтение новых символов и их присо-

единение к текущей строке. В некоторый момент такой строки в словаре не оказывается. Тогда кодер добавляет ее в словарь и строит метку, в первом поле которой стоит указатель на последнюю найденную в словаре строку, а во втором поле записан последний символ строки (на котором произошел обрыв успешных поисков). В табл. 2.4 показаны шаги при декодировании последовательности `«sir.sid.eastman.easily.teases.sea.sick.seals»`.

Словарь	Метка	Словарь	Метка
0 <code>null</code>			
1 <code>«s»</code>	<code>(0,«s»)</code>	14 <code>«у»</code>	<code>(0,«у»)</code>
2 <code>«i»</code>	<code>(0,«i»)</code>	15 <code>«_t»</code>	<code>(4,«t»)</code>
3 <code>«r»</code>	<code>(0,«r»)</code>	16 <code>«е»</code>	<code>(0,«е»)</code>
4 <code>«_»</code>	<code>(0,«_»)</code>	17 <code>«as»</code>	<code>(8,«s»)</code>
5 <code>«si»</code>	<code>(1,«i»)</code>	18 <code>«es»</code>	<code>(16,«s»)</code>
6 <code>«d»</code>	<code>(0,«d»)</code>	19 <code>«_s»</code>	<code>(4,«s»)</code>
7 <code>«_e»</code>	<code>(4,«e»)</code>	20 <code>«ea»</code>	<code>(4,«a»)</code>
8 <code>«a»</code>	<code>(0,«a»)</code>	21 <code>«_si»</code>	<code>(19,«i»)</code>
9 <code>«st»</code>	<code>(1,«t»)</code>	22 <code>«c»</code>	<code>(0,«c»)</code>
10 <code>«m»</code>	<code>(0,«m»)</code>	23 <code>«k»</code>	<code>(0,«k»)</code>
11 <code>«an»</code>	<code>(8,«n»)</code>	24 <code>«_se»</code>	<code>(19,«e»)</code>
12 <code>«_ea»</code>	<code>(7,«a»)</code>	25 <code>«al»</code>	<code>(8,«l»)</code>
13 <code>«sil»</code>	<code>(5,«l»)</code>	26 <code>«s(eof)»</code>	<code>(1,«(eof)»)</code>

Табл. 2.4. Шаги кодирования LZ78.

На каждом шаге строка, добавленная в словарь, совпадает с кодируемой строкой минус последний символ. В типичном процессе сжатия словарь начинается с коротких строк, но по мере продвижения по кодируемому тексту, все более и более длинные строки добавляются в словарь. Размер словаря может быть фиксированным или определяться размером доступной памяти каждый раз, когда запускается программа сжатия LZ78. Большой словарь позволяет делать глубокий поиск длинных совпадений, но ценой этого служит длина поля указателей (а, значит, и длина метки) и замедление процесса словарного поиска.

Хорошей структурой для организации словаря является дерево, но не двоичное. Дерево начинается нулевой строкой в корне. Все строки, начинающиеся с нулевой строки (строки, для которых указатель в метке равен нулю), добавляются к дереву как потомки корня. В нашем примере таковыми служат следующие строки: `s`, `i`, `r`, `_`, `d`, `a`, `m`, `у`, `е`, `с`, `k`. Все они становятся корнями поддеревьев, показанных на рис. 2.5. Например, все строки, начинающиеся с

символа **s** (четыре строки **si**, **sil**, **st** и **s(eof)**) образуют поддерево узла **s**.

В 8-битовом алфавите имеется всего 256 различных символов. В принципе, каждый узел дерева может иметь до 256 потомков. Следовательно, добавление новых узлов на дерево является динамическим процессом. Когда узел будет создаваться, у него еще нет потомков, поэтому необходимо зарезервировать некоторый объем памяти для них. Поскольку удалять узлы с дерева не придется, не придется и заниматься перераспределением памяти, что несколько упрощает манипулирование с ней.

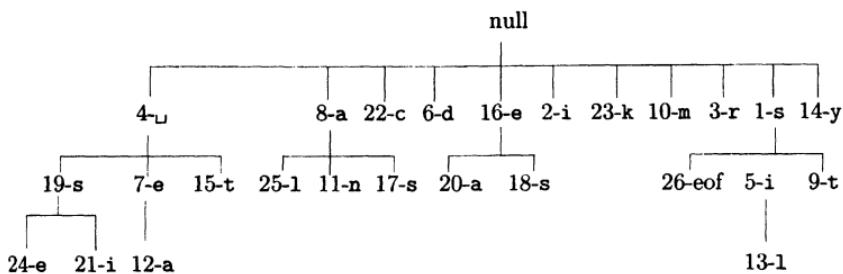


Рис. 2.5. Словарное дерево для LZ78.

На таком дереве очень легко искать строки и добавлять новые символы. Например, чтобы найти строку **sil**, программа ищет символ **s** в корне, затем его потомка **i** символа **s**, и так далее, спускаясь вниз по дереву. Вот еще некоторые примеры.

1. Когда символ **s** из **sid** появляется на входе при шаге 5, кодер находит узел «1-с» на дереве как потомка «null». Затем поступает символ **i**, но узел **s** не имеет такого потомка (у него в этот момент вовсе нет потомков). Тогда кодер добавляет узел «5-и» в виде потомка узла «1-с», что означает добавление на дерево строки **si**.
2. Когда на входе появляется пробел между **eastman** и **easily** на шаге 12, возникает похожая ситуация. Кодер обнаруживает узел «4-», получает символ **e**, находит «7-е», получает **a**, но у «7-е» нет потомка **a**, поэтому он добавляет узел «12-а», что эквивалентно добавлению строки «\_ea».

Возникающие деревья относятся к специальному типу деревьев, в которых образование новых веток и их дальнейшее ветвление зависит только от части поступающих новых данных, а не от всех данных. В случае LZ78, новая ветвь соответствует всего одному новому добавляемому символу.

Поскольку размер дерева ничем не ограничен, может возникнуть переполнение. На самом деле, это всегда происходит, за исключением тех случаев, когда входной файл имеет малый размер. Первоначальный LZ78 метод не устанавливает, что делать в этом случае. Он является, скорее, теоретическим методом, который удобно исследовать, но трудно использовать на практике. Ниже приводятся некоторые возможные решения этой проблемы.

1. Простейшее решение – это заморозить словарь, когда он переполнится. Новые узлы не добавляются, словарь становится статическим, но с его помощью можно по-прежнему кодировать последовательности.

2. Удалить все дерево при переполнении и начать строить новое. Это решение эффективно разбивает данные на блоки, и в каждом блоке используется свой собственный словарь. Если содержимое сжимаемых данных меняется от блока к блоку, то этот метод дает хорошие результаты, поскольку в словаре не хранятся строки, которые с малой вероятностью поступят в дальнейшем. Здесь опять, как и в LZ77 неявно предполагается, что похожие данные будут группироваться близко друг к другу.

3. Утилита `compress` операционной системы UNIX использует более сложное решение. Эта программа (еще называемая LZC) использует алгоритм LZW (см. § 2.4) с растущим словарем. Она начинает работать с маленьким словарем, состоящим всего из  $2^9 = 512$  строк, причем первые 256 уже заполнены. При использовании исходного словаря в сжатый файл записываются 9-битовые указатели. После заполнения этого словаря его размер удваивается до 1024 строк. С этого момента используются 10-битовые указатели. Этот процесс продолжается до тех пор, пока размер указателей не достигнет некоторого максимального значения, задаваемого пользователем (от 9 до 16 бит, причем 16 принято по умолчанию). Когда наибольший допустимый словарь полностью заполняется, программа продолжает работать без изменения словаря (который теперь становится статическим), но одновременно делается мониторинг коэффициента сжатия. Если этот коэффициент превышает некоторый предустановленный порог, то происходит сброс словаря, процесс начинается с исходного 512-строчного словаря. При таком подходе словарь никогда не устаревает.

4. Когда словарь заполнился, удалить из него некоторые самые старые строки, чтобы освободить место для новых. К сожалению, не существует хорошего алгоритма, который определял бы, какие строки удалять и в каком количестве.

Декодер LZ78 работает примерно так же, как и кодер, строя словарь и оперируя с ним. Поэтому он более сложен, чем декодер LZ77.

*Эти слова! Как невинно и беспомощно они выглядят в словаре. Но сколько могущества для добра или для зла они проявляют в руках тех, кто умеет объединять их.*

— Натаниэль Хаусорн

## 2.4. LZW

Это весьма популярный вариант алгоритма LZ78, который был разработан Терри Уэлчем (Terrry Welch) в 1984 ([Welch 84] и [Phillips 92]). Его главной особенностью является удаление второго поля из метки. Метка LZW состоит только из указателя на место в словаре. Для лучшего понимания алгоритма LZW мы временно забудем, что словарь является деревом и будем предполагать, что словарь — это просто массив, состоящий из строк разной длины. Метод LZW начинается инициализацией словаря всеми символами исходного алфавита. В общем случае 8-битного алфавита, первые 256 записей (отдельные символы с номерами от 0 до 255) заносятся в словарь до поступления сжимаемых данных. Поскольку словарь уже частично заполнен, первые поступившие символы всегда будут обнаружены в словаре, поэтому метка может состоять всего лишь из указателя, и нет надобности дополнительно хранить код символа, как в алгоритмах LZ77 и LZ78.

(Алгоритм LZW запатентован и для его использования необходима лицензия. Издание патентов для программного обеспечения обсуждается в [Salomon 00].)

Метод LZW накапливает поступающие на вход символы в строке  $I$ . После каждого нового символа, добавленного в строку  $I$ , кодер ищет  $I$  в словаре. Если строка обнаружена, то процесс удлинения  $I$  продолжается. В некоторый момент добавление нового символа  $x$  приводит к необнаружению строки  $Ix$  (символ  $x$  был добавлен к  $I$ ). Тогда кодер (1) записывает в выходной файл указатель в словаре на строку  $I$ , (2) сохраняет строку  $Ix$  (которая теперь будет называться *фразой*) в словаре на следующей допустимой позиции и (3) инициализирует (присваивает) строке  $I$  новое значение  $x$ . Чтобы проиллюстрировать процесс кодирования, мы еще раз воспользуемся последовательностью `«sir.sid.eastman.easily.teases.sea.sick.seals»`. Получаем следующие шаги.

I	В словаре?	Новая запись	Выход	I	В словаре?	Новая запись	Выход
s	да			y	да		
si	нет	256-si	115 (s)	y-	нет	274-y-	121 (y)
i	да			-	да		
ir	нет	257-ir	105 (i)	-t	нет	275-_t	32 (_)
r	да			t	да		
r-	нет	258-r-	114 (r)	te	нет	276-te	116 (t)
-	да			e	да		
_s	нет	259-_s	32 (_)	ea	да		
s	да			eas	нет	277-eas	263 (ea)
si	да			s	да		
sid	нет	260-sid	256 (si)	se	нет	278-se	115 (s)
d	да			e	да		
d-	нет	261-d-	100 (d)	es	нет	279-es	101 (e)
-	да			s	да		
_e	нет	262-_e	32 (_)	s-	нет	280-s-	115 (s)
e	да			-	да		
ea	нет	263-ea	101 (e)	_s	да		
a	да			_se	нет	281-_se	259 (_s)
as	нет	264-as	97 (a)	e	да		
s	да			ea	да		
st	нет	265-st	115 (s)	ea-	нет	282-ea-	263 (ea)
t	да			-	да		
tm	нет	266-tm	116 (t)	_s	да		
m	да			_si	нет	283-_si	259 (_s)
ma	нет	267-ma	109 (m)	i	да		
a	да			ic	нет	284-ic	105 (i)
an	нет	268-an	97 (a)	c	да		
n	да			ck	нет	285-ck	99 (c)
n-	нет	269-n-	110 (n)	k	да		
-	да			k-	нет	286-k-	107 (k)
_e	да			-	да		
_ea	нет	270-_ea	262 (_e)	_s	да		
a	да			_se	да		
as	да			_sea	нет	287-_sea	281 (_se)
asi	нет	271-asi	264 (as)	a	да		
i	да			al	нет	288-al	97 (a)
il	нет	272-il	105 (i)	l	да		
l	да			ls	нет	289-ls	108 (l)
ly	нет	273-ly	108 (l)	s	да		
				s, eof	нет		115 (s)

Табл. 2.6. Кодирование строки «sir.sid.eastman....».

0. Инициализируем словарь записями от 0 до 255 всеми 8-битовыми символами.

1. Первый входной символ **s** обнаруживается в словаре под номером 115 (это его код ASCII). Следующий входной символ **i**, но строки **si** нет в словаре. Кодер делает следующее: (1) он выдает на выход ссылку 115, (2) сохраняет **si** в следующей доступной позиции словаря (запись номер 256) и (3) инициализирует I строкой **i**.



2. На вход поступает символ *r*, но строки *ir* нет в словаре. Кодер (1) записывает на выход 105 (ASCII код *i*), (2) сохраняет в словаре *ir* (запись 257) и (3) инициализирует I строкой *r*.

0	NULL	110	<i>n</i>	262	<i>_e</i>	276	<i>te</i>
1	SOH	...		263	<i>ea</i>	277	<i>eas</i>
...		115	<i>s</i>	264	<i>as</i>	278	<i>se</i>
32	SP	116	<i>t</i>	265	<i>st</i>	279	<i>es</i>
...		...		266	<i>tm</i>	280	<i>s_</i>
97	<i>a</i>	121	<i>y</i>	267	<i>ma</i>	281	<i>_se</i>
98	<i>b</i>	...		268	<i>an</i>	282	<i>ea_</i>
99	<i>c</i>	255	255	269	<i>n_</i>	283	<i>_si</i>
100	<i>d</i>	256	<i>si</i>	270	<i>_ea</i>	284	<i>ic</i>
101	<i>e</i>	257	<i>ir</i>	271	<i>asi</i>	285	<i>ck</i>
...		258	<i>r_</i>	272	<i>il</i>	286	<i>k_</i>
107	<i>k</i>	259	<i>_s</i>	273	<i>ly</i>	287	<i>_sea</i>
108	<i>l</i>	260	<i>sid</i>	274	<i>y_</i>	288	<i>al</i>
109	<i>m</i>	261	<i>d_</i>	275	<i>_t</i>	289	<i>ls</i>

Табл. 2.7. Словарь LZW.

Табл. 2.6 суммирует все шаги процесса. В табл. 2.7 приведены некоторые исходные записи из словаря LZW плюс записи, добавленные в процессе кодирования данной строки. Полный выходной файл имеет следующий вид (входят только числа, но не символы в скобках):

115 (*s*), 105 (*i*), 114 (*r*), 32 (*\_*), 256 (*si*), 100 (*d*), 32 (*\_*), 101 (*e*), 97 (*a*), 115 (*s*), 116 (*t*), 109 (*m*), 97 (*a*), 110 (*n*), 262 (*\_e*), 264 (*as*), 105 (*i*), 108 (*l*), 121 (*y*), 32 (*\_*), 116 (*t*), 263 (*ea*), 115 (*s*), 101 (*e*), 115 (*s*), 259 (*\_s*), 263 (*ea*), 259 (*\_s*), 105 (*i*), 99 (*c*), 107 (*k*), 280 (*\_se*), 97 (*a*), 108 (*l*), 115 (*s*), *eof*.

На рис. 2.8 приведена запись этого алгоритма на псевдокоде. Через  $\lambda$  обозначается пустая строка, а через  $<<a, b>>$  – конкатенация (соединение) двух строк *a* и *b*.

Инструкция алгоритма «*добавить <<di, ch>> в словарь*» будет обсуждаться особо. Ясно, что на практике словарь может переполниться, поэтому эта инструкция должна также содержать тест на переполнение словаря и некоторые действия при обнаружении переполнения.

Поскольку первые 256 записей занесены в словарь в самом начале, указатели на словарь должны быть длиннее 8 бит. В простейшей реализации указатели занимают 16 бит, что допускает 64К записей в словаре ( $64K = 2^{16} = 65536$ ). Такой словарь, конечно, быстро переполнится. Такая же проблема возникает при реализации алгоритма LZ78, и любое решение, допустимое для LZ78, годится и для

LZW. Другое интересное свойство этого алгоритма заключается в том, что строки в словаре становятся длиннее на один символ на каждом шаге. Это означает, что требуется много времени для появления длинных строк в словаре, а значит и эффект сжатия будет проявляться медленно. Можно сказать, что LZW медленно приспособливается к входным данным.

```

for i:=0 to 255 do
    добавить i как 1-символьную строку в словарь;
    добавить λ в словарь;
    di:=словарный индекс λ;
repeat
    read(ch);
    if <<di,ch>> есть в словаре then
        di:=словарный индекс <<di,ch>>;
    else
        output(di);
        добавить <<di,ch>> в словарь;
        di:=словарный индекс ch;
    endif;
until конец входного файла;

```

Рис. 2.8. Алгоритм LZW.

**Пример:** Применим алгоритм LZW для кодирования строки символов «alf\_eats\_alfalfa». Последовательность шагов отображена в табл. 2.9. Кодер выдает на выход файл:

97 (a), 108 (l), 102 (f), 32 (\_), 101 (e), 97 (a), 116 (t), 115 (s), 32 (\_), 256 (al), 102 (f), 265 (alf), 97 (a),

а в словаре появляются следующие новые записи:

(256: al), (257: lf), (258: f\_), (259: \_e), (260: ea), (261: at), (262: ts), (263: s\_), (264: \_a), (265: alf), (266: fa), (267: alfa).

**Пример:** Проанализируем сжатие строки «aaaa...» алгоритмом LZW. Кодер заносит первую «a» в I, ищет и находит a в словаре. Добавляет в I следующую a, находит, что строки Ix (=aa) нет в словаре. Тогда он добавляет запись 256: aa в словарь и выводит метку 97 (a) в выходной файл. Переменная I инициализируется второй «a», вводится третья «a», Ix вновь равна aa, которая теперь имеется в словаре. I становится aa, появляется четвертая «a». Стока Ix

равна  $aaa$ , которых нет в словаре. Словарь пополняется этой строкой, а на выход идет метка 256 (aa). И инициализируется третьей «a», и т.д. и т.п. Дальнейший процесс вполне ясен.

В результате строки  $a$ ,  $aa$ ,  $aaa$ ,  $aaaa\dots$  добавляются в словарь в позиции 256, 257, 258, ..., а на выход подается последовательность

97 (a), 256 (aa), 257 (aaa), 258 (aaaa), ... .

Значит, выходной файл состоит из указателей на все более и более длинные последовательности символов  $a$ , и  $k$  указателей обозначают строку, длина которой равна  $1 + 2 + \dots + k = (k + k^2)/2$ .

I	В словаре?	Новая запись	Выход	I	В словаре?	Новая запись	Выход
a	да			s_	нет	263-s_	115 (s)
al	нет	256-al	97 (a)	-	да		
l	да			_a	нет	264-_a	32 (-)
lf	нет	257-lf	108 (1)	a	да		
f	да			al	да		
f_	нет	258-f_	102 (f)	alf	нет	265-alf	256 (al)
_	да			f	да		
_e	нет	259-_e	32 (w)	fa	нет	266-fa	102 (f)
e	да			a	да		
ea	нет	260-ea	101 (e)	al	да		
a	да			alf	да		
at	нет	261-at	97 (a)	alfa	нет	267-alfa	265 (alf)
t	да			a	да		
ts	нет	262-ts	116 (t)	a, eof	нет		97 (a)
s	да						

Табл. 2.9. Кодирование LZW для «alf\_eats\_alfalfa»

Предположим, что входной файл состоит из 1 миллиона символов  $a$ . Мы можем найти длину сжатого файла, решив квадратное уравнение  $(k + k^2)/2 = 1000000$  относительно неизвестной  $k$ . Решение будет  $k \approx 1414$ . Выходит, что файл длиной 8 миллионов бит будет сжат в 1414 указателей длины не менее 9 бит (а на самом деле, 16 бит). Фактор сжатия или  $8M/(1414 \times 9) \approx 628.6$  или  $8M/(1414 \times 16) \approx 353.6$ .

Результат потрясающий, но такие файлы попадаются весьма редко (заметим, что этот конкретный файл можно сжать, просто записав в выходной файл «1000000 a» без использования LZW).

#### 2.4.1. Декодирование LZW

Для того, чтобы понять как работает декодер метода LZW, прежде всего еще раз напомним основные три шага, которые выполняет



кодер каждый раз, делая очередную запись в выходной файл: (1) он заносит туда словарный указатель на строку  $I$ , (2) сохраняет строку  $Ix$  в следующей свободной позиции словаря и (3) инициализирует строку  $I$  символом  $x$ .

Декодер начинает с заполнения словаря первыми символами алфавита (их, обычно, 256). Затем он читает входной файл, который состоит из указателей в словаре, использует каждый указатель для того, чтобы восстановить несжатые символы из словаря и записать их в выходной файл. Кроме того, он строит словарь тем же методом, что и кодер (этот факт, обычно, отражается фразой: кодер и декодер работают *синхронно* или шаг в шаг).

На первом шаге декодирования, декодер вводит первый указатель и использует его для восстановления словарного элемента  $I$ . Это строка символов, и она записывается декодером в выходной файл. Далее следует записать в словарь строку  $Ix$ , однако символ  $x$  еще неизвестен; это будет первый символ следующей строки, извлеченной из словаря.

На каждом шаге декодирования после первого декодер вводит следующий указатель, извлекает следующую строку  $J$  из словаря, записывает ее в выходной файл, извлекает ее первый символ  $x$  и заносит строку  $Ix$  в словарь на свободную позицию (предварительно проверив, что строки  $Ix$  нет в словаре). Затем декодер перемещает  $J$  в  $I$ . Теперь он готов к следующему шагу декодирования.

В нашем примере «`sir.sid...`» первым символом входного файла декодера является указатель 115. Он соответствует строке `s`, которая извлекается из словаря, сохраняется в  $I$  и становится первой строкой, записанной в выходной (разжатый) файл. Следующий указатель – 105, поэтому в  $J$  заносится строка `i`, а содержимое  $J$  записывается в выходной файл. Первый символ строки  $J$  добавляется к переменной  $I$ , образуя строку `si`, которой нет в словаре, поэтому она добавляется в словарь на позиции 256. Содержимое переменной  $J$  переносится в переменную  $I$ ; теперь  $I$  равно `i`. Следующий указатель – 114, поэтому, из словаря извлекается строка `g`, заносится в  $J$  и пишется в выходной файл. Переменная  $I$  дополняется до значения `ig`; такой строки нет в словаре, поэтому она туда заносится под номером 257. Переменная  $J$  переписывается в  $I$ , теперь  $I$  равно `g`. На следующем шаге декодер читает указатель 32, записывает «`_`» в файл и сохраняет в словаре строку `g_`.

**Пример:** Стока «`alf_eats_alfalfa`» будет декодироваться с помощью сжатого файла из примера на стр. 100 следующим образом:

1. Читаем указатель 97. Из словаря заносим  $I=«a»$  и выводим в файл



- «a». Строку  $I_x$  надо записать в словарь, но  $x$  неизвестно.
2. Читаем указатель 108. Из словаря заносим  $J=«l»$  и выводим в файл «l». Сохраняем «al» в словаре под номером 256. Заносим  $I=«l»$ .
  3. Читаем указатель 102. Из словаря заносим  $J=«f»$  и выводим в файл «f». Сохраняем «lf» в словаре под номером 257. Заносим  $I=«f»$ .
  4. Читаем указатель 32. Из словаря заносим  $J=«_»$  и выводим в файл «\_». Сохраняем «f\_» в словаре под номером 258. Заносим  $I=«_»$ .
  5. Читаем указатель 101. Из словаря заносим  $J=«e»$  и выводим в файл «e». Сохраняем «\_e» в словаре под номером 259. Заносим  $I=«e»$ .
  6. Читаем указатель 97. Из словаря заносим  $J=«a»$  и выводим в файл «a». Сохраняем «ea» в словаре под номером 260. Заносим  $I=«a»$ .
  7. Читаем указатель 116. Из словаря заносим  $J=«t»$  и выводим в файл «t». Сохраняем «at» в словаре под номером 261. Заносим  $I=«t»$ .
  8. Читаем указатель 115. Из словаря заносим  $J=«s»$  и выводим в файл «s». Сохраняем «ts» в словаре под номером 262. Заносим  $I=«s»$ .
  9. Читаем указатель 32. Из словаря заносим  $J=«_»$  и выводим в файл «\_». Сохраняем «s\_» в словаре под номером 263. Заносим  $I=«_»$ .
  10. Читаем указатель 256. Из словаря заносим  $J=«al»$  и выводим в файл «al». Сохраняем «\_a» в словаре под номером 264. Заносим  $I=«al»$ .
  11. Читаем указатель 102. Из словаря заносим  $J=«f»$  и выводим в файл «f». Сохраняем «alf» в словаре под номером 265. Заносим  $I=«f»$ .
  12. Читаем указатель 265. Из словаря заносим  $J=«alf»$  и выводим в файл «alf». Сохраняем «fa» в словаре под номером 266. Заносим  $I=«alf»$ .
  13. Читаем указатель 97. Из словаря заносим  $J=«a»$  и выводим в файл «a». Сохраняем «alfa» в словаре под номером 267 (хотя она никогда не будет использована). Заносим  $I=«a»$ .
  14. Читаем eof. Конец.

**Пример:** Для алфавита из двух символов **a** и **b** покажем первые несколько шагов кодирования и декодирования последовательности «ababab...». Предполагается, что словарь инициализируется всего двумя строками (1: a) и (2: b). Кодер выдает на выход последовательность

1 (a), 2 (b), 3 (ab), 5 (aba), 4 (ba), 7 (bab), 6 (abab), 9 (ababa), 8 (baba),...

и добавляет в словарь новые строки: (3: ab), (4: ba), (5: aba), (6: abab), (7: bab), (8: baba), (9: ababa), (10: ababab), (11: babab). Процесс вполне регулярный, его легко проанализировать и предсказать, что будет заноситься на  $k$ -ом шаге в файл и словарь. Результат не стоит затраченных усилий.

### 2.4.2. Структура словаря LZW

До этого момента считалось, что словарем LZW служит массив из строк переменной длины. Чтобы понять, почему специальное дерево будет являться лучшей структурой для словаря, следует напомнить работу кодера. Он считывает символы и добавляет их в строку  $I$  до тех пор, пока  $I$  находится в словаре. В некоторый момент строки  $Ix$  в словаре не обнаруживается, и тогда строка  $Ix$  помещается в словарь. Значит, при добавлении новых строк в словарь поступает всего один новый символ  $x$ . Это предложение можно перефразировать еще так: для каждой словарной строки в словаре найдется «родительская» строка, которая короче ровно на один символ.

Поэтому для наших целей хорошим выбором будет структура дерева, которая уже использовалась в LZ78, при построении которого добавление новых строк  $Ix$  делается добавлением всего одного нового символа  $x$ . Основная проблема состоит в том, что каждый узел дерева LZW может иметь много потомков (дерево не двоичное, а  $q$ -ичное, где  $q$  – это объем алфавита). Представим себе узел буквы  $a$  с номером 97. В начале у него нет потомков, но если к дереву добавится строка  $ab$ , то у  $a$  появится один потомок. Если позже добавится новая строка, скажем,  $ae$ , то потомков станет два, и так далее. Значит, дерево следует сконструировать так, чтобы каждый узел в нем мог бы иметь любое число потомков, причем без резервирования дополнительной памяти для них заранее.

Один путь конструирования такой структуры данных – это построение дерева в виде массива узлов, в котором каждый узел является структурой из двух полей: символа и указателя на узел родителя. В самом узле нет указателей на его потомков. Перемещение вниз по дереву от родителя к потомку делается с помощью процесса хеширования (перемешивания или рандомизации), в котором указатели на узлы и символы потомков перемешиваются некоторой функцией для создания новых указателей.

Предположим, что строка  $abc$  уже была на входе, символ за символом, и ее сохранили в трех узлах с адресами 97, 266, 284. Пусть далее за ними следует символ  $d$ . Теперь кодер ищет строку  $abcd$ , а точнее, узел, содержащий символ  $d$ , чей родитель находится по адресу 284. Кодер хеширует адреса 284 (указатель на строку  $abc$ ) и 100 (ASCII код  $d$ ) и создает указатель на некоторый узел, скажем, 299. Потом кодер проверяет узел 299. Возможны три случая:

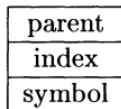
1. Этот узел не используется. Это означает строки  $abcd$  еще нет в словаре и его следует туда добавить. Кодер делает это путем сохранения родительского указателя и ASCII кода 100 в этом узле.

Получаем следующий результат:

Узел				
Адрес	:	97	266	284
Содержимое	:	(-: «а»)	(97: «б»)	(266: «с»)
Строки	:	«а»	«аб»	«абс»
				«abcd»

2. Узел содержит родительский указатель 284 и ASCII код символа d. Это означает, что строка abcd уже есть на дереве. Тогда кодер вводит следующий символ, скажем, e и ищет на дереве строку abcde.
3. В узле хранится что-то другое. Это означает, что хеширование другой пары указателя и кода ASCII дало адрес 299, и узел 299 уже содержит в себе информацию про другие строки. Такая ситуация называется *коллизией*, и ее можно разрешить несколькими путями. Простейший путь разрешения коллизии – это увеличивать адрес до 300, 301,... до тех пор, пока не будет найден пустой узел или узел с содержимым (284: «д»).

На практике узлы строятся состоящими из трех полей: указателя на родительский узел, указателя (или индекса), созданного в процессе хеширования, и кода (обычно, ASCII) символа, лежащего в этом узле. Второе поле необходимо для разрешения коллизий. Таким образом, узел имеет вид триплета



**Пример:** Проиллюстрируем эту структуру данных с помощью строки «ababab...» из примера на стр. 103). Словарем будет служить массив dict, элементами которого являются структуры с тремя полями parent, index, symbol. Мы будем обращаться к этим полям с помощью выражения вида dict[pointer].parent, где pointer – индекс в массиве. Словарь инициализируется двумя символами a и b. (Для простоты мы не будем использовать коды ASCII, но будем считать, что a и b имеют коды 1 и 2.) Первые несколько шагов кодера будут следующими:

*Шаг 0:* Отметить все позиции, начиная с третьей, как неиспользуемые.



*Шаг 1:* Первый входной символ **a** помещается в переменную **I**. Точнее, переменной **I** присваивается значение кода 1. Поскольку это первый символ входного файла, кодер его не ищет в словаре.

*Шаг 2:* Второй символ **b** помещается в переменную **J**, то есть, **J=2**. Кодер должен искать в словаре строку **ab**. Он выполняет операцию **pointer:=hash(I, J)**. Здесь **hash(x, y)** обозначает некоторую функцию двух аргументов **x** и **y**. Предположим, что результатом этой операции будет 5. Поле **dict[pointer].index** содержит «пусто», так как строки **ab** нет в словаре. Она туда заносится с помощью следующих действий

```
dict[pointer].parent:=I;
dict[pointer].index:=pointer;
dict[pointer].symbol:=J;
```

причем **pointer=5**. После чего **J** помещается в **I**, то есть **I=2**.

/	/	/	/	1	...
1	2	-	-	5	
a	b			b	

*Шаг 3:* Третий символ **a** поступает в **J**, то есть, **J=1**. Кодер должен искать в словаре строку **ba**. Выполняется **pointer:=hash(I, J)**. Пусть результат равен 8. Поле **dict[pointer].index** содержит «пусто», то есть, строки **ba** нет в словаре. Добавляем ее в словарь с помощью операций

```
dict[pointer].parent:=I;
dict[pointer].index:=pointer;
dict[pointer].symbol:=J;
```

причем **pointer=8**. После чего **J** помещается в **I**, то есть **I=1**.

/	/	/	/	1	/	/	2	/	...
1	2	-	-	5	-	-	8	-	
a	b			b			a		

*Шаг 4:* Четвертый символ **b** переносится в **J**, теперь **J=2**. Кодер должен искать в словаре строку **ab**. Выполняется **pointer:=hash(I, J)**. Мы знаем из шага 2, что результат – 5. Поле **dict[pointer].index** содержит 5, то есть строка **ab** имеется в словаре. Значение переменной **pointer** заносится в **I**, **I=5**.

*Шаг 5:* Пятый символ **a** попадает в **J**, теперь **J=1**. Кодер должен искать в словаре строку **aba**. Как обычно, выполняется оператор **pointer:=hash(I, J)**. Предположим, что результатом будет 8 (коллизия). Поле **dict[pointer].index** содержит 8, что хорошо, но поле

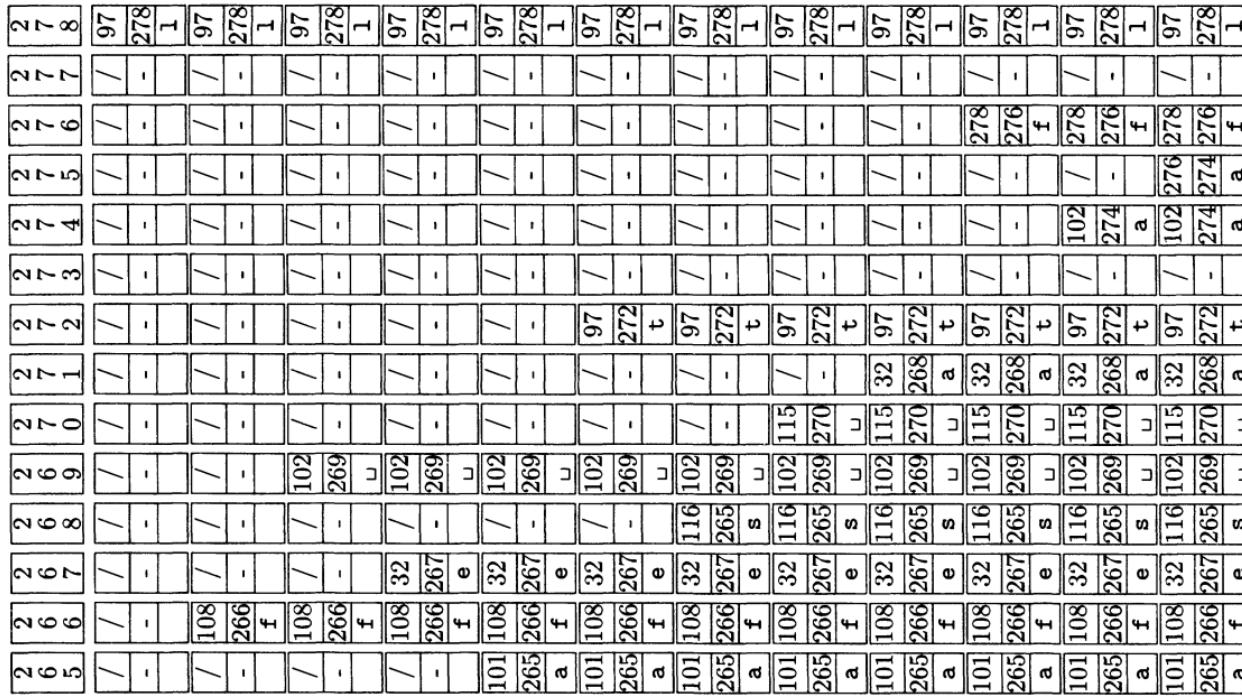
`dict[pointer].parent` содержит 2 вместо ожидавшего указателя 5. Произошла коллизия, это означает, что строки `aba` нет в словаре по адресу 8. Тогда `pointer` последовательно увеличивается на 1 до тех пор, пока не найдет узел, для которого `index=8` и `parent=5` или, который пуст. В первом случае `aba` имеется в словаре, и `pointer` записывается в I. Во втором случае `aba` нет в словаре, кодер сохраняет его в узле `pointer` и записывает в I содержимое J.

/	/	/	/	1	/	/	2	5	/	...
1	2	-	-	5	-	-	8	8	-	
a	b			b			a	a		

**Пример:** Сделаем процедуру хеширование для кодирования строки символов `«alf_eats_alfalfa»`. Сам процесс кодирования детально разобран в примере на стр. 100. Результаты хеширования выбраны произвольно. Они не порождены какой-либо реальной функцией хеширования. Все 12 построенных узлов этого дерева показаны на рис. 2.10.

1.  $\text{Hash}(1,97)=278$ . По адресу 278 заносится  $(97,278,1)$ .
2.  $\text{Hash}(f,108)=266$ . По адресу 266 заносится  $(108,266,f)$ .
3.  $\text{Hash}(-,102)=269$ . По адресу 269 заносится  $(102,269,-)$ .
4.  $\text{Hash}(e,32)=267$ . По адресу 267 заносится  $(32,267,e)$ .
5.  $\text{Hash}(a,101)=265$ . По адресу 265 заносится  $(101,265,a)$ .
6.  $\text{Hash}(t,97)=272$ . По адресу 272 заносится  $(97,272,t)$ .
7.  $\text{Hash}(s,116)=265$ . Коллизия! Спускаемся в следующую свободную позицию 268 и заносим  $(116,265,s)$ .
8.  $\text{Hash}(-,115)=270$ . По адресу 270 заносится  $(115,270,-)$ .
9.  $\text{Hash}(a,32)=268$ . Коллизия! Спускаемся в следующую свободную позицию 271 и заносим  $(32,268,a)$ .
10.  $\text{Hash}(1,97)=278$ . По адресу 278 в поле `index` записано 278, а поле `symbol` содержит 1. Значит, ничего не надо записывать или сохранять на дереве.
11.  $\text{Hash}(f,278)=276$ . По адресу 276 заносится  $(278,276,f)$ .
12.  $\text{Hash}(a,102)=274$ . По адресу 274 заносится  $(102,274,a)$ .
13.  $\text{Hash}(1,97)=278$ . По адресу 278 в поле `index` записано 278, а поле `symbol` содержит 1. Значит, ничего не надо делать.
14.  $\text{Hash}(f,278)=276$ . По адресу 276 в поле `index` записано 276, а поле `symbol` содержит `f`. Значит, ничего не надо делать.
15.  $\text{Hash}(a,276)=274$ . Коллизия! Спускаемся в следующую свободную позицию 275, и заносим  $(276,274,a)$ .

Читатель, который тщательно следил за нашими построениями до этого момента будет счастлив узнать, что декодер LZW использует словарь очень простым способом, не применяя хеширование.



Brz 210 Post zonza IZW za self costs 21 fol fca

Сначала декодер, как и кодер заполняет первые 256 позиций словаря кодами ASCII. Затем он читает указатели из входного файла и использует их для нахождения символов в словаре.

На первом шаге декодирования, декодер вводит первый указатель и использует его для обнаружения в словаре первой строки  $I$ . Этот символ записывается в выходной (разжатый) файл. Теперь необходимо записать в словарь строку  $Ix$ , но символ  $x$  еще не известен; им будет первый символ следующей строки извлекаемой из словаря.

На каждом шаге декодирования после первого декодер читает следующий указатель из файла, использует его для извлечения следующей строки  $J$  из словаря и записывает эту строку в выходной файл. Если указатель был равен, скажем, 8, то декодер изучает поле `dict[8].index`. Если это поле равно 8, то это правильный узел. В противном случае декодер изучает последующие элементы массива до тех пор, пока не найдет правильный узел.

После того, как правильный узел найден, его поле `parent` используется при проходе назад по дереву для извлечения всех символов строки в *обратном порядке*. Затем символы помещаются в  $J$  в правильном порядке (см. пункт 2 после следующего примера), декодер извлекает последний символ  $x$  из  $J$  и записывает в словарь строку  $Ix$  в следующую свободную позицию. (Строка  $I$  была найдена на предыдущем шаге, поэтому на дерево необходимо добавить только один узел с символом  $x$ .) После чего декодер перемещает  $J$  в  $I$ . Теперь он готов для следующего шага.

Для извлечения полной строки из дерева LZW декодер следует по указателям полей `parent`. Это эквивалентно продвижению вверх по дереву. Вот почему хеширование здесь не нужно.

**Пример:** В предыдущем примере описана процедура хеширования строки `«alf_eats_alfalfa»`. На последнем шаге в позицию 275 массива был записан узел (276,274,a), а в выходной сжатый файл был записан указатель 275. Когда этот файл читается декодером, указатель 275 является последним обработанным им элементом входного файла. Декодер находит символ  $a$  в поле `symbol` узла с адресом 275, а в поле `parent` читает указатель 276. Тогда декодер проверяет адрес 276 и находит в нем символ  $f$  и указатель на родительский узел 278. В позиции 278 находится символ  $l$  и указатель 97. Наконец, в позиции 97 декодер находит символ  $a$  и нулевой указатель. Значит, восстановленной строкой будет `alfa`. У декодера не возникает необходимости делать хеширование и применять поле `index`.

Нам осталось обсудить обращение строки. Возможны два варианта решения.

1. Использовать стек. Это часто применяемая структура в современных компьютерах. Стеком называется массив в памяти, к которому открыт доступ только с одного конца. В любой момент времени, элемент, который позже попал в стек, будет раньше других извлечен оттуда. Символы, которые извлекаются из словаря, помещаются в стек. Когда последний символ будет туда помещен, стек освобождается символ за символом, которые помещаются в переменную *J*. Стока оказывается перевернутой.

2. Извлекать символы из словаря и присоединять их к *J* справа налево. В итоге переменная *J* будет иметь правильный порядок следования символов. Переменная *J* должна иметь длину, достаточную для хранения самых длинных строк.

#### 2.4.3. LZW в практических приложениях

Опубликование в 1984 году алгоритма LZW произвело большое впечатление на всех специалистов по сжатию информации. За этим последовало большое количество программ и приложений с различными вариантами этого метода. Наиболее важные из них приведены в [Salomon 2000].

### 2.5. Заключение

Все представленные здесь различные методы словарного сжатия используют одни и те же общие принципы. Они читают файл символ за символом и добавляют фразы в словарь. Фразы являются отдельными символами и строками символов входного файла. Методы сжатия различаются только способом отбора фраз для сохранения в словаре. Когда строка входного файла совпадает с некоторой фразой в словаре, в сжатый файл записывается позиция этой фразы или метка. Если для хранения метки требуется меньше бит, чем для записи самой фразы, то наблюдается эффект сжатия.

В общем случае, словарные методы сжатия, если их применять грамотно, дают лучшие результаты, чем статистические методы компрессии, поэтому они активно используются во всевозможных компрессионных приложениях, или они являются одним из этапов многостадийного сжатия. Много других методов словарного сжатия можно найти в книге [Salomon 2000].

*У Рамси Макдональда имеется замечательный дар сжимать огромные потоки слов в короткие идеи.*  
— Уинстон Черчилль

## ГЛАВА 3

# СЖАТИЕ ИЗОБРАЖЕНИЙ

Современное поле сжатия информации весьма обширно, на нем взошло огромное количество различных методов компрессии всевозможных типов данных: текстов, изображений, видео и звука. Среди этого многообразия методов особое место занимает сжатие изображений, так как, во-первых, это первая область, где пользователи имеют дело с большим числом файлов, которые необходимо эффективно сжимать, а во-вторых, здесь мы впервые встречаем сжатие данных с частичной потерей информации.

В кодировке ASCII каждый символ занимает один байт. Типичная книга в четверть миллиона слов насчитывает примерно миллион символов и занимает около 1МВ. Однако, если к этой книге добавить хоть одну картинку для иллюстрации, то объем занимаемой памяти может удвоиться, так как файл, содержащий изображение, может занимать 1МВ и даже больше.

Файл изображения такой большой, потому что он представляет собой двумерный образ. Кроме того, современные дисплеи и другие средства представления изображений способны передавать огромное число цветов и оттенков, которые требуют много битов (обычно 24) для представления цвета каждого отдельного пикселя.

Сжатие текста всегда делается без потери информации. Можно представить себе документы, в которых часть информации не очень важна, и поэтому ее можно опустить без искажения смысла всего текста, однако, не существует алгоритмов, которые могли бы дать компьютеру возможность самостоятельно, без участия человека, определять, что в тексте важно, а что – нет. В отличие от сжатия текстов, для компрессии изображений такие алгоритмы существуют в большом количестве. Такой алгоритм может удалить большую часть информации из изображения, что приводит к замечательной компрессии. Проверкой качества сжатия изображения является визуальное сравнение оригинального изображения и изображения, полученного из его сжатого образа (с потерей части ин-

формации). Если испытатель не может отличить одно от другого, то сжатие с потерей допускается. В некоторых случаях человеческий глаз способен уловить разницу, но все равно сжатие считается удовлетворительным.

Обсуждение сжатия изображений в этой главе носит в основном общий теоретический характер. Здесь будут излагаться основные подходы к решению проблемы сжатия изображений. Из некоторых рассмотренных конкретных методов компрессии, выделяется наиболее важный из них – это метод JPEG (§ 3.7).

### 3.1. Введение

Современные компьютеры весьма интенсивно применяют графику. Операционные системы с интерфейсом оконного типа используют картинки, например, для отображения директорий или папок. Некоторые совершаемые системой действия, например загрузку и пересылку файлов, также отображаются графически. Многие программы и приложения предлагают пользователю графический интерфейс (GUI), который значительно упрощает работу пользователя и позволяет легко интерпретировать полученные результаты. Компьютерная графика используется во многих областях повседневной деятельности при переводе сложных массивов данных в графическое представление. Итак, графические изображения крайне важны, но они требуют так много места! Поскольку современные дисплеи передают множество цветов, каждый пиксель принятто интерпретировать в виде 24-битового числа, в котором компоненты красного, зеленого и голубого цветов занимают по 8 бит каждый. Такой 24-битовый пиксель может отображать  $2^{24} \approx 16.78$  миллионов цветов. Тогда изображение с разрешением  $512 \times 512$  пикселов будет занимать 786432 байта. А изображению размером  $1024 \times 1024$  пикселя потребуется 3145728 байт для его хранения. Мультфильмы и анимация, которые также весьма популярны в компьютерных приложениях, потребуют еще большего объема. Все это объясняет важность сжатия изображений. К счастью, изображения допускают частичную потерю информации при сжатии. В конце концов, изображения призваны для того, чтобы люди на них смотрели, поэтому та часть изображения, которая не воспринимается глазом, может быть опущена. Эта основная идея вот уже три десятилетия движет разработчиками методов сжатия, допускающих некоторую потерю данных.



В общем случае, информацию можно сжать, если в ней присутствует избыточность. В этой книге часто повторялось утверждение, что компрессия информации делается путем удаления или сокращения избыточности данных. При сжатии с потерями мы имеем дело с новой концепцией, с новой парадигмой, которая предполагает сжатие путем удаления *несущественной, нерелевантной информации*. Изображение можно сжать с потерей, удалив несущественную информацию, даже если в нем нет избыточности.

Отметим, что изображение без избыточности не обязательно является случайным. На стр. 18 обсуждались два типа избыточности, присущей графическим образом. Самой важной из них является корреляция (зависимость) пикселов. В редких случаях корреляция пикселов может быть слабой или вовсе отсутствовать, но изображение все еще не будет случайным и даже осмысленным.

Идея о допустимости потери графической информации станет более приятной, после того, как мы рассмотрим процесс образования цифровых изображений. Вот три примера: (1) реальное изображение может быть сканировано с фотографии или рисунка и оцифровано (переведено в матрицу пикселов), (2) образ может быть записан видеокамерой, которая сама порождает пиксели и сохраняет их в памяти, (3) картинка может быть нарисована на экране компьютера с помощью специальной компьютерной программы. Во всех трех случаях некоторая информация теряется при переводе в цифровую форму. Зритель готов позволить некоторую потерю информации при условии, что это будет терпимая и малозаметная потеря.

(Оцифровывание изображений состоит из двух этапов: выборки образцов и квантования. Выборка образцов, дискретизация, состоит в разбиении двумерного изображения на маленькие области, пиксели, а под квантованием подразумевается процесс присвоения каждому пикселу некоторого числа, обозначающего его цвет. Заметим, что оцифровывание звука состоит из тех же этапов, но только звуковой поток является одномерным.)

Приведем простой тест на качественное определение потери информации при сжатии изображения. Пусть данный образ  $A$  (1) сжат в  $B$ , (2) разжат в  $C$ , и (3) их разница обозначена  $D = C - A$ . Если  $A$  был сжат без потери информации и разжат подобающим образом, то  $C$  должен быть идентичен  $A$ , и образ  $D$  должен быть равномерно белым. Чем больше информации потеряно, тем дальше будет  $D$  от равномерно белого образа.

Так как же все-таки следует сжимать изображения? До этого момента мы рассматривали три подхода к задаче компрессии: это RLE, статистический метод и словарный метод. Неявно предполагалось, что эти методы применимы к данным любой природы, но из практических наблюдений мы заметили, что лучше всего эти методы работают при сжатии текстов. Поэтому новые методы сжатия должны учитывать три основных различия между текстами и графическими изображениями:

1. Текст одномерен, а изображение имеет размерность 2. Весь текст можно рассматривать как одну длинную строку символов. Каждая буква текста имеет двух соседей, слева и справа. Все соседи весьма слабо коррелированы между собой. Например, в этом абзаце букве «и» предшествуют буквы «н», «р», «л», «с», «п», а за ней следуют буквы «е», «в», «н», «м», «р». В других абзацах та же буква «и» может иметь других соседей. В изображении пиксел имеет четырех непосредственных соседей и восемь ближайших (за исключением пикселов, лежащих на границе, см. рис. 3.1, где восемь ближайших соседей пикселя «\*» показаны черным цветом), и между ними существует сильная корреляция.



Рис. 3.1. Четыре ближайших и восемь соседних пикселов.

2. Текст состоит из относительно небольшого числа символов алфавита. Обычно, это 128 кодов ASCII или 256 байтов длины по 8 бит каждый. Наоборот, каждый пиксел изображения представим 24 битами, поэтому может быть до  $2^{24} \approx 16.78$  миллионов различных пикселов. Значит, число элементарных «символов» в изображении может быть огромным.

3. Не известен алгоритм, который определял бы, какая часть текста является неважной или малозначимой, и ее можно удалить без ущерба для всего текста, но существуют методы, которые автоматически удаляют неважную информацию из графического образа. Этим достигается значительна степень компрессии.

Таким образом, методы сжатия текстовой информации становятся малоэффективными и неудовлетворительными при работе с изображениями. Поэтому в этой главе будут обсуждаться совершен-



но другие подходы к решению этой задачи. Они различны, но все они удаляют избыточность с использованием следующего принципа.

**Принцип сжатия изображений.** *Если случайно выбрать пиксель изображения, то с большой вероятностью ближайшие к нему пиксели будут иметь тот же или близкий цвет.*

Итак, сжатие изображений основывается на *сильной корреляции* соседних пикселов. Эта корреляция также называется *пространственной избыточностью*.

Вот простой пример того, что можно сделать с коррелированными пикселями. Следующая последовательность чисел отражает интенсивность 24 смежных пикселов в одной строке некоторого непрерывно тонового изображения:

12, 17, 14, 19, 21, 26, 23, 29, 41, 38, 31, 44, 46, 57, 53, 50, 60, 58, 55, 54, 52, 51, 56, 60.

Здесь только два пикселя совпадают. Их среднее значение равно 40.3. Вычитание каждого предыдущего символа даст последовательность

12, 5, -3, 5, 2, 5, -3, 6, 12, -3, -7, 13, 2, 11, -4, -3, 10, -2, -3, -1, -2, -1, 5, 4.

Эти последовательности пикселов проиллюстрированы графически на рис. 3.2, который показывает потенциал сжатия: (1) разность пикселов существенно меньше их абсолютных величин. Их среднее равно 2.58. (2) Они повторяются. Имеется только 15 различных значений. В принципе, каждый из них можно закодировать 4 битами. Они декоррелированы, то есть, разности соседних значений, в среднем, не уменьшаются. Это видно из последовательности вторых разностей:

12, -7, -8, 8, -3, 3, -8, 9, 6, -15, -4, 20, -11, 9, -15, -1, 13, -12, -1, 2, -1, -1, 6, 1.

Эти разности уже больше исходных величин.

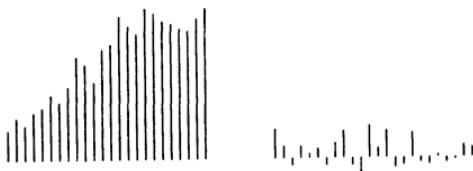


Рис. 3.2. Величины и разности 24 соседних пикселов.

Рис. 3.3 предлагает другую иллюстрацию «коррелированных величин». Матрица  $A$  размером  $32 \times 32$  заполнена случайными числами,

и ее значения на рисунке (а) изображены квадратиками различных серых оттенков. Случайная природа этих элементов очевидна. Затем эту матрицу обратили и результат – матрицу  $B$  – изобразили на рисунке (б). На этот раз этот массив квадратиков  $32 \times 32$  выглядит на глаз более структурированным. Прямое вычисление корреляции с помощью коэффициента Пирсона по формуле (3.1) показывает, что перекрестная корреляция верхних двух строк матрицы  $A$  является относительно малым числом 0.0412, в то время, как соответствующая величина, вычисленная для верхних строк матрицы  $B$  равна относительно большому числу  $-0.9831$ . Дело в том, что каждый элемент матрицы  $B$  зависит от *всех* элементов матрицы  $A$ .

$$R = \frac{n \sum x_i y_i - \sum x_i \sum y_i}{\sqrt{\left[ n \sum x_i^2 - (\sum x_i)^2 \right] \left[ n \sum y_i^2 - (\sum y_i)^2 \right]}}. \quad (3.1)$$

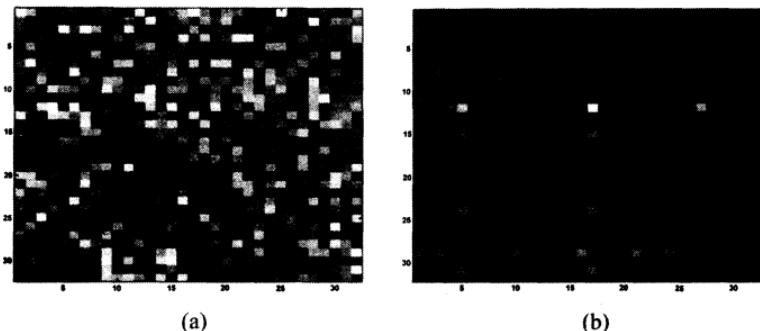


Рис. 3.3. Случайная матрица (а) и ее обратная матрица (б).

```
n=32; a=rand(n); imagesc(a); colormap(gray)
b=inv(a); imagesc(b)
Программа на Matlab для рис. 3.3.
```

**Пример:** Воспользуемся специальной программой для иллюстрации ковариационной матрицы для (1) матрицы с коррелированными элементами и (2) матрицы с декоррелированными элементами. На рис. 3.4 показаны две  $32 \times 32$  матрицы. Первая из них  $a$  является случайной (то есть, декоррелированной), а вторая матрица  $b$  является обратной для матрицы  $a$  (значит, она коррелирована). Их ковариационные матрицы также показаны. Очевидно, что матрица  $\text{cov}(a)$

близка к диагональной (недиагональные элементы равны нулю или близки к нулю), а матрица  $\text{cov}(b)$  далека от диагональной. Далее приведена программа для системы Matlab, которая рисует эти картинки.

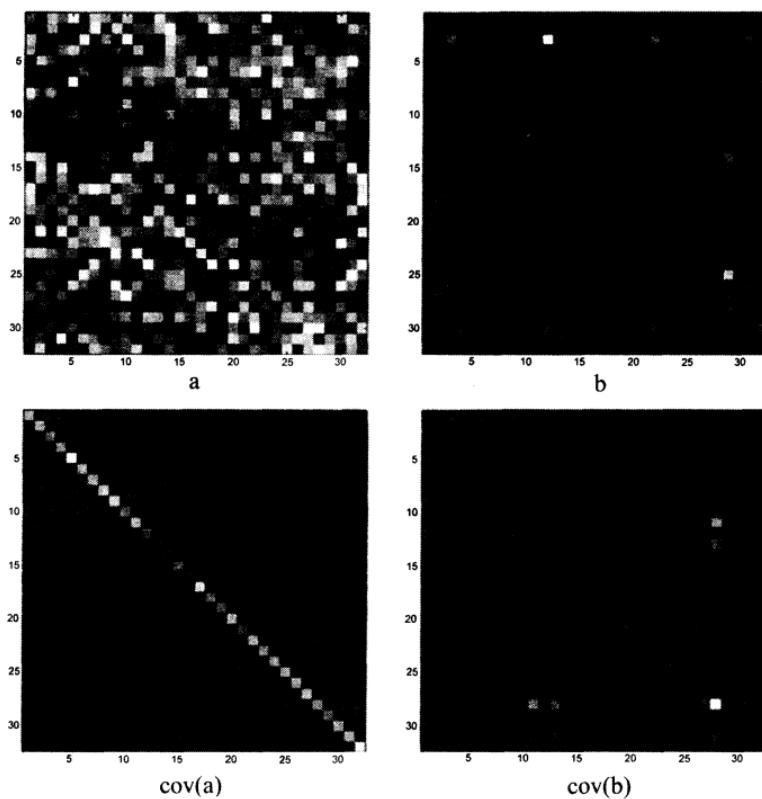


Рис. 3.4. Ковариация коррелированной и декоррелированной матриц.

Если понятие коррелированных величин стало более ясным, то можно легко ответить на вопрос: «Как проверить стали ли пиксели изображения после некоторого преобразования декоррелированными или нет?» Ответ будет таким: «Если матрица содержит декоррелированные значения, то ковариация любой ее строки с любым столбцом равна нулю». В результате ковариационная матрица  $M$  будет диагональной. Статистические понятия дисперсии, ковариации и корреляции обсуждаются в любом учебнике по статистике.

Принцип сжатия изображений имеет и другую сторону. Нам хорошо известно по опыту, что яркость близких пикселов тоже кор-



релирована. Два смежных пикселя могут иметь разные цвета. Один может быть близок к красному, а второй – к зеленому, однако, если первый был ярким, то его сосед, обычно, тоже будет ярким. Это свойство можно использовать для перевода представления пикселов RGB (red, green, blue) в виде трех других компонентов: яркости и двух хроматических компонентов, определяющих цвет. Одним таким форматом (или пространством цветов) служит YCbCr, где Y (компонент «светимости») отвечает за яркость пикселя, а Cb и Cr определяют его цвет. Этот формат будет обсуждаться в § 3.7.1, но его преимущество уже легко понять. Наш глаз чувствителен к маленьким изменениям яркости, но не цвета. Поэтому потеря информации в компонентах Cb и Cr сжимает образ, внося искажения, которые глаз не замечает. А искажение информации о компоненте Y, наоборот, является более приметной.

```
a=rand(32); b=inv(a);
figure(1); imagesc(a); colormap(gray); axis square
figure(2); imagesc(b); colormap(gray); axis square
figure(3); imagesc(cov(a)); colormap(gray); axis square
figure(4); imagesc(cov(b)); colormap(gray); axis square
```

Программа на Matlab для рис. 3.4.

### 3.2. Типы изображений

Цифровое изображение представляет собой прямоугольную таблицу точек, или элементов изображения, расположенных в  $m$  строках и  $n$  столбцах. Выражение  $m \times n$  называется *разрешением* изображения (хотя иногда этот термин используется для обозначения числа пикселей, приходящихся на единицу длины изображения). Точки изображения называются *пикселями* (за исключением случаев, когда изображение передается факсом или видео; в этих случаях точка называется *пелом*). Для целей сжатия графических образов удобно выделить следующие типы изображений:

1. *Двухуровневое* (или монохроматическое) изображение. В этом случае все пиксели могут иметь только два значения, которые обычно называют черным (двоичная единица, или основной цвет) и белым (двоичный нуль или цвет фона). Каждый пиксель такого изображения представлен одним битом, поэтому это самый простой тип изображения.
2. *Полутоновое* изображение. Каждый пиксель такого изображения может иметь  $2^n$  значений от 0 до  $2^n - 1$ , обозначающих одну из

$2^n$  градаций серого (или иного) цвета. Число  $n$  обычно сравнимо с размером байта, то есть, оно равно 4, 8, 12, 16, 24 или другое кратное 4 или 8. Множество самых значимых битов всех пикселов образуют самую значимую битовую плоскость или слой изображения. Итак, полуточковое изображение со шкалой из  $2^n$  уровней составлено из  $n$  битовых слоев.

3. *Цветное изображение*. Существует несколько методов задания цвета, но в каждом из них участвуют три параметра. Следовательно, цветной пиксель состоит из трех частей. Обычно, цветной пиксель состоит из трех байтов. Типичными цветовыми моделями являются RGB, HLS и CMYK. Детальное описание цветовых моделей выходит за рамки этой книги, но некоторое базовое рассмотрение вопроса будет приведено в § 3.7.1.

4. *Изображение с непрерывным тоном*. Этот тип изображений может иметь много похожих цветов (или полуточков). Когда соседние пиксели отличаются всего на единицу, глазу практически невозможно различить их цвета. В результате такие изображения могут содержать области, в которых цвет кажется глазу непрерывно меняющимся. В этом случае пиксель представляется или большим числом (в полуточковом случае) или тремя компонентами (в случае цветного образа). Изображения с непрерывным тоном являются природными или естественными (в отличие от рукотворных, искусственных); обычно они получаются при съемке на цифровую фотокамеру или при сканировании фотографий или рисунков.

5. *Дискретно-тоновое изображение* (оно еще называется синтетическим). Обычно, это изображение получается искусственным путем. В нем может быть всего несколько цветов или много цветов, но в нем нет шумов и пятен естественного изображения. Примерами таких изображений могут служить фотографии искусственных объектов, машин или механизмов, страницы текста, карты, рисунки или изображения на дисплее компьютера. (Не каждое искусственное изображение будет обязательно дискретно-тоновым. Сгенерированное компьютером изображение, которое должно выглядеть натуральным, будет иметь непрерывные тона, несмотря на свое искусственное происхождение.) Искусственные объекты, тексты, нарисованные линии имеют форму, хорошо определяемые границы. Они сильно контрастируют на фоне остальной части изображения (фона). Прилегающие пиксели дискретно-тонового образа часто бывают одиночными или сильно меняют свои значения. Такие изображения плохо сжимаются методами с потерей данных, поскольку искажение всего нескольких пикселов буквы делает ее неразборчивой, преобразует привычное начертание в совершенно неразличимое.

чимое. Методы сжатия изображений с непрерывными тонами плохо обращаются с четкими краями дискретно-тоновых образов, для которых следует разрабатывать особые методы компрессии. Отметим, что дискретно-тоновые изображения, обычно, несут в себе большую избыточность. Многие ее фрагменты повторяются много раз в разных местах изображения.

6. Изображения, *подобные мультильмам*. Это цветные изображения, в которых присутствуют большие области одного цвета. При этом соприкасающиеся области могут весьма различаться по своему цвету. Это свойство можно использовать для достижения лучшей компрессии.

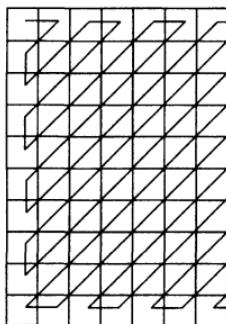
Интуитивно становится ясно, что каждому типу изображений присуща определенная избыточность, но все они избыточны по-разному. Поэтому трудно создать один метод, который одинаково хорошо сжимает любые типы изображений. Существуют отдельные методы для сжатия двухуровневых образов, непрерывно-тоновых и дискретно-тоновых изображений. Существуют также методы, которые пытаются разделить изображение на непрерывно-тоновую и дискретно-тоновую части и сжимать их по отдельности.

### 3.3. Подходы к сжатию изображений

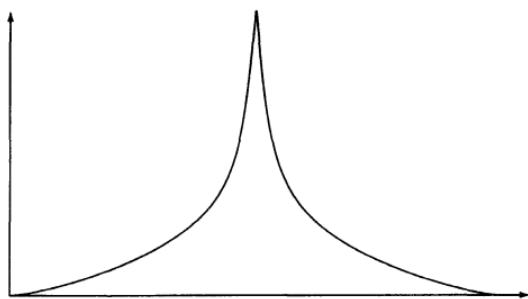
Методы сжатия изображений обычно разрабатываются для конкретного типа изображений. Здесь перечислены различные подходы к компрессии графических образов. При этом будут обсуждаться только общие принципы. Специфические методы описаны дальше в этой главе (см. также [Salomon 2000]).

**Подход 1.** Для сжатия двухуровневых изображений. Каждый пиксель такого образа представляется одним битом. Применение принципа сжатия образов к компрессии двухуровневых изображений означает, что непосредственные соседи пикселя  $P$  стремятся совпадать с  $P$ . Поэтому имеет смысл использовать методы кодирования длин серий (RLE) для сжатия таких изображений. Метод сжатия сканирует образ строка за строкой и вычисляет длины последовательных черных и белых пикселов. Длины кодируются кодами переменной длины и записываются в сжатый файл. Примером такого сжатия является факсимильная компрессии (см. § 1.6).

Еще раз подчеркнем, что это всего лишь общий принцип сжатия, конкретные методы могут сильно отличаться друг от друга. Например, метод может сканировать образ по строкам, а может зигзагообразно (см. рис. 3.5.а) или сканировать изображение область за областью с помощью заполняющих их кривых (см. [Salomon 2000]).



(a)



(b)

Рис. 3.5. (а) Сканирование зигзагом. (б) Распределение Лапласа.

**Подход 2.** Также метод для двухуровневых изображений. Опять используется принцип совпадения ближайших пикселов в расширенной форме: если текущий пиксель имеет цвет  $c$  (где  $c$  – белый или черный), то пиксели того же цвета, наблюдавшиеся в прошлом (а также те, которые появятся в будущем) будут иметь таких же ближайших соседей.

Этот подход рассматривает  $n$  последовательных соседей текущего пикселя и представляет их в виде  $n$ -битного числа. Это число называется *контекстом* пикселя. В принципе, может быть  $2^n$  различных контекстов, но в силу избыточности образа мы ожидаем, что контексты распределены неравномерно. Некоторые контексты встречаются чаще, а некоторые – реже.

Кодер вычисляет сколько раз встречался каждый контекст для пикселя цвета  $c$  и присваивает контекстам соответствующие вероятности. Если текущий пиксель имеет цвет  $c$  и его контекст имеет вероятность  $p$ , то кодер может использовать адаптивное арифметическое кодирование для кодирования пикселя с этой вероятностью. Такой подход использован в методе JBIG [Salomon 2000].

Перейдем теперь к полутооновым изображениям. Пиксель такого изображения представлен  $n$  битами и может иметь одно из  $2^n$  значений. Применяя принцип сжатия изображений, заключаем, что соседи пикселя  $P$  стремятся быть похожими на  $P$ , но не обязательно совпадают с ним. Поэтому метод RLE здесь не годится. Вместо этого рассмотрим две другие идеи.

**Подход 3.** Расслоить полутооновую картинку на  $n$  двухуровневых изображений и каждую сжать с помощью RLE и префиксных ко-

дов. Здесь кажется, что можно применить основной принцип сжатия изображений, но по отношению к каждому отдельному слову полуточнового изображения. Однако, это не так, и следующий пример проясняет ситуацию. Представим себе полуточновое изображение с  $n = 4$  (то есть, имеем дело с 4-битовыми пикселями, или с 16 градациями серого цвета). Его можно расслоить на 4 двухуровневых изображения. Если два смежных пикселя исходного образа имели величины 0000 и 0001, то они похожи по цвету. Однако два пикселя со значениями 0111 и 1000 также близки на полуточновом изображении (это, соответственно, числа 7 и 8), но они различаются во всех 4 слоях.

Эта проблема возникает, так как в двоичном представлении соседние числа могут отличаться во многих разрядах. Двоичные коды для 0 и 1 отличаются в одном разряде, коды для 1 и 2 отличаются в двух разрядах, а коды для 7 и 8 отличаются уже во всех четырех битах. Решением этой проблемы может служить разработка специальных последовательностей двоичных кодов, в которых последовательные коды с номерами  $i$  и  $i+1$  различались бы ровно на один бит. Примером таких кодов служат *рефлексные коды Грея*, описанные в § 3.3.1.

**Подход 4.** Использовать контекст пикселя  $P$ , который состоит из значений нескольких ближайших соседей. Выбираем несколько соседних пикселов, вычисляем среднее значение  $A$  их величин и делаем предсказание, что пиксель  $P$  будет равен  $A$ . Основной принцип сжатия изображений позволяет считать, что в большинстве случаев мы будем правы, во многих случаях будем почти правы и лишь в немногих случаях будем совершенно неправы. Можно сказать, что в предсказанной величине пикселя  $P$  содержится избыточная информация про  $P$ . Вычислим разность

$$\Delta = P - A$$

и присвоим некоторый (префиксный) код переменной длины величине  $\Delta$  так, чтобы малым величинам (которые ожидаются часто) соответствовали короткие коды, а большим величинам (которые ожидаются редко) назначались длинные коды. Если значение  $P$  лежит в интервале от 0 до  $m - 1$ , то значения  $\Delta$  попадают в интервал  $[-(m - 1), +(m - 1)]$ , для чего потребуется  $2(m - 1) + 1$  или  $2m - 1$  кодов.

Экспериментирование с большими числами показывает, что значения  $\Delta$  стремятся иметь распределение, близкое к распределению



Лапласа (см. рис. 3.5b), которое часто возникает в статистике. Тогда метод сжатия может использовать это распределение для присвоения вероятностей величинам  $\Delta$  и весьма эффективно применять арифметическое кодирование для  $\Delta$ . Этот подход лежит в основе метода сжатия MLP (см. [Salomon 2000]).

Контекст пикселя может состоять из одного или двух его соседей. Однако лучшие результаты получаются, если использовать несколько пикселов, причем каждому пикселу присваивается некоторый вес для вычисления взвешенного среднего: более дальним пикселям присваивается меньший вес. Другое важное рассмотрение касается декодирования. Для декодирования изображения необходимо уметь вычислять контекст до самого пикселя. Это означает, что контекст должен состоять из уже декодированных пикселов. Если изображение сканируется строка за строкой (сверху вниз и слева направо), то контекст может состоять только из пикселов, которые расположены выше и левее данного пикселя.

**Подход 5.** Сделать преобразование пикселов и кодировать преобразованные значения. Понятие преобразования образа, а также наиболее важные преобразования, используемые при компрессии изображений, будут разобраны в § 3.5. Глава 4 посвящена вейвлетным преобразованиям. Напомним, что компрессия достигается путем удаления или сокращения избыточности. Избыточность изображения образуется за счет корреляции между пикселями, поэтому преобразования, которые делают декорреляцию, одновременно удаляют избыточность. Квантуя преобразованные значения, можно получить эффективное сжатие с частичной потерей информации. Мы хотим преобразовывать величины в независимые, так как для независимых величин легче построить простую модель для их кодирования.

Обратимся теперь к кодированию цветных изображений. Пиксели таких изображений состоят из трех компонентов, например, красного, зеленого и голубого. Большинство цветных образов являются непрерывно-тоновыми или дискретно-тоновыми.

**Подход 6.** Идея этого метода заключается в разделении изображения на три полутоновых и в их независимом сжатии на основании одного из подходов 2, 3 или 4.

Принцип сжатия непрерывно-тоновых изображений гласит, что цвет соседних пикселов мало изменяется, хотя и не обязательно постоянен. Однако близость цветов еще не означает близость величин пикселов. Рассмотрим, например, цветной 12-битовый пиксель, в котором каждая компонента имеет 4 бита. Итак, 12 бит 1000|0100|0000 обозначают пиксель, цвет которого получается смешением 8 единиц

красного (около 50%, так как всего их 16), четырех единиц зеленого (около 25%) и совсем без голубого. Представим себе теперь два пикселя со значениями 0011|0101|0011 и 0010|0101|0011. Их цвета весьма близки, поскольку они отличаются только в одной красной компоненте, и там их различие состоит только в одном бите. Однако, если рассмотреть их как 12-битовые числа, то два числа 0011|0101|0011=851 и 0010|0101|0011=595 различаются весьма значительно.

Важная особенность этого подхода состоит в использовании представления цвета в виде яркости и цветности вместо обычного RGB. Эти понятия обсуждаются в § 3.7.1 и в книге [Salomon 2000]. Преимущество этого представления основано на том, что глаз чувствителен к маленьким изменениям яркости, но не цветности. Это позволяет допускать значительную потерю в компоненте цветности, но при этом совершать декодирование без значительного визуального ухудшения качества изображения.

**Подход 7.** Другие методы требуются для компрессии дискретно-тоновых изображений. Напомним, что такие изображения состоят из больших одноцветных областей, причем каждая область может появляться в нескольких местах образа. Хорошим примером служит содержимое экрана компьютера. Такой образ состоит из текста и пиктограмм (иконок). Каждая буква, каждая пиктограмма занимают некоторую область на экране, причем каждая из этих областей может находиться в разных местах экрана. Возможный способ сжатия такого изображение заключается в его сканировании и обнаружении таких повторяющихся областей. Если область  $B$  совпадает с ранее выделенной областью  $A$ , то можно сжать  $B$ , записав в файл указатель на область  $A$ . Примером такого метода служит алгоритм блоковой декомпозиции (FABD, [Salomon 2000]).

**Подход 8.** Разделение изображения на части (перекрывающиеся или нет) и сжатие их одна за другой. Предположим, что следующая требующая сжатия часть имеет номер 15. Постараемся сравнить ее с уже обработанными частями 1–14. Предположим, что ее можно выразить, например, с помощью комбинации частей 5 (растяжение) и 11 (поворот), тогда достаточно сохранить только несколько чисел, которые описывают требуемую комбинацию, а саму часть 15 можно отбросить. Если часть 15 не удается выразить в виде комбинации прежних частей, то ее придется сохранить в «сыром виде».

Другой подход является основой для различных *фрактальных* методов сжатия изображений. Здесь применяется основной принцип сжатия изображений, но вместо пикселов выступают части изобра-



жения. Этот принцип говорит о том, что «интересные» изображения (то есть те, которые будут сжиматься на практике) обладают некоторой степенью *самоподобия*. Часть изображения совпадает или близка всему изображению.

Методы сжатия изображений вовсе не ограничиваются перечисленными базисными подходами. В книге [Salomon 2000] обсуждаются методы, которые используют контекстные деревья, марковские цепи, вейвлеты и многое другое. Дополнительно следует упомянуть метод *прогрессивного сжатия образов* (см. § 3.6), который добавляет еще одно измерение сжатию изображений.

*Искусство – это техника передачи сообщений.  
Труднее всего передавать образы.*

— Клер Тур Олденбург

### 3.3.1. Коды Грэя

Методы сжатия изображений, разработанные для конкретных типов графических образов, иногда могут успешно применяться для компрессии других типов. Например, любой метод сжатия двухуровневых изображений годится для полутоновых образов, если предварительно расслойть полутоновой образ на несколько двухуровневых и каждый слой сжимать независимо. Представим себе изображение с 16 градациями серого цвета. Каждый пиксель определяется 4 битами, поэтому изображения разделяется на 4 двухуровневых. Однако при таком подходе может нарушиться основной принцип сжатия изображений. Вообразим себе два прилегающих 4-битных пикселя со значениями  $7 = 0111_2$  и  $8 = 1000_2$ . Эти пиксели имеют близкие значения, но при их разделении на 4 слоя, они будут различаться во всех 4 слоях! Происходит это из-за того, что в двоичном представлении числа 7 и 8 различаются во всех четырех разрядах. Для того, чтобы успешно применить здесь метод сжатия двухуровневых изображений необходимо, чтобы в двоичном представлении два последовательных числа имели двоичные коды, различающиеся только на один бит. Такое представление чисел существует и оно называется *рефлексными кодами Грэя* (RGC, reflected Gray code). Их легко построить, следуя предлагаемому рекурсивному правилу.

Начнем с двух однобитных кодов (0, 1). Построим два семейства двухбитных кодов, повторив (0, 1), а потом добавив слева (или справа), сначала 0, а потом 1 к исходным семействам. В результате получим (00, 01) и (10, 11). Теперь следует переставить (отразить)

коды второго семейства в обратном порядке и приписать к первому семейству. Тогда получатся 4 двухбитных кода RGC (00, 01, 11, 10), с помощью которых можно кодировать целые числа от 0 до 3, причем последовательные числа будут различаться ровно в одном двоичном разряде. Применяем эту процедуру еще раз и получаем два семейства кодов длины 3: (000, 001, 011, 010) и (110, 111, 101, 100). Объединяем их и получаем 3-битные коды RGC. Проделаем первые три шага для образования 4-битных кодов Грея:

добавить 0: (0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100),  
 добавить 1: (1000, 1001, 1011, 1010, 1110, 1111, 1101, 1100),  
 отразить: (1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000).

В табл. 3.6 показано, как меняются отдельные биты двоичных кодов, представляющих числа от 0 до 32. В этой таблице в нечетных столбцах приведены обычные двоичные коды целых чисел. Жирным шрифтом выделены биты числа  $i$ , которые отличаются от соответствующих битов числа  $i - 1$ . Видно, что бит самого младшего (нулевого) разряда (бит  $b_0$ ) меняется каждый раз, бит  $b_1$  меняется каждый второй раз, а в общем случае, бит  $b_k$  меняется у каждого  $2^k$ -го целого числа. В четных столбцах таблицы помещены все последовательные коды Грея. Под таблицей приведена рекурсивная функция Matlab, вычисляющая коды Грея.

43210	Gray	43210	Gray	43210	Gray	43210	Gray
00000	00000	<b>01000</b>	10010	<b>10000</b>	00011	<b>11000</b>	10001
00001	00100	01001	10110	10001	00111	11001	10101
00010	01100	<b>01010</b>	11110	<b>10010</b>	01111	<b>11010</b>	11101
00011	01000	01011	11010	10011	01011	11011	11001
00100	11000	<b>01100</b>	01010	<b>10100</b>	11011	<b>11100</b>	01001
00101	11100	01101	01110	10101	11111	11101	01101
00110	10100	<b>01110</b>	00110	<b>10110</b>	10111	<b>11110</b>	00101
00111	10000	01111	00010	10111	10011	11111	00001

```
function b=rgc(a,i)
[r,c]=size(a);
b=[zeros(r,1),a; ones(r,1),fliplr(a)];
if i>1, b=rgc(b,i-1); end;
```

Табл. 3.6. Первые 32 двоичных и RGC кода.

Коды RGC можно построить для целого  $n$ , и не прибегая к рекурсивной процедуре. Достаточно применить функцию «ИСКЛЮЧАЮЩЕЕ ИЛИ» к числу  $n$  и к его логическому сдвигу на один разряд вправо. На языке C это запишется следующими операторами:

$n^>(n>1)$ . Табл. 3.7 (подобная табл. 3.6) была порождена именно этим выражением.

Можно сделать следующий вывод. Слой, соответствующий самому старшему двоичному разряду (в обычном представлении) изображения, подчиняется принципу сжатия в большей, чем слой самого младшего разряда. Когда соседние пиксели имеют значения, которые различаются на единицу (равны  $p$  и  $p + 1$ ), шансы на то, что их менее значимые или более значимые разряды отличаются, одинаковы. Значит, любой метод компрессии, который будет по отдельности сжимать слои должен или учитывать эту особенность разрядности слоев, или использовать коды Грея для представления величин пикселов.

43210	Gray	43210	Gray	43210	Gray	43210	Gray
00000	00000	01000	01100	10000	11000	11000	10100
00001	00001	01001	01101	10001	11001	11001	10101
00010	00011	01010	01111	10010	11011	11010	10111
00011	00010	01011	01110	10011	11010	11011	10110
00100	00110	01100	01010	10100	11110	11100	10010
00101	00111	01101	01011	10101	11111	11101	10011
00110	00101	01110	01001	10110	11101	11110	10001
00111	00100	01111	01000	10111	11100	11111	10000

```
a=linspace(0,31,32); b=bitshift(a,-1);
b=bitxor(a,b); dec2bin(b)
```

Табл. 3.7. Первые 32 двоичных и RGC кода.

На рис. 3.10, 3.11 и 3.12 показаны восемь побитных слоев картинки «попугаи» в обычном двоичном представлении (слева) и в виде кодов Грея (справа). Слои, построенные с помощью программы Matlab, приведены на рис. 3.8. Они занумерованы числами от 8 (самый старший, самый значимый бит) до 1 (самый младший бит). Очевидно, что слой самого младшего бита не показывает никакую корреляцию между пикселями; они случайны или близки к случайным для обоих представлений, двоичного и RGC. Однако, слои от 2 до 5 демонстрируют большую корреляцию пикселов для кодов Грея. Слои с 6 по 8 выглядят по-разному для двоичных кодов и для кодов Грея, но кажутся сильно коррелированными в обоих случаях.

На рис. 3.13 изображены две версии первых 32 кодов Грея в их графическом представлении. На рисунке (б) показаны коды из табл. 3.6, а на рисунке (с) приведены коды из табл. 3.9. Несмотря на то, что оба семейства образуют коды Грея, в них по разному чередуются биты 0 и 1 в разных побитных слоях. На рисунке (б) самый значимый бит меняется 4 раза. Второй значимый бит меняется 8



раз, а биты трех младших разрядов меняются, соответственно, 16, 2 и 1 раз. Если использовать это множество кодов Грея для расслоения изображения, то средние слои, очевидно, обнаружат наименьшую корреляцию между соседними пикселями.

```
clear;
filename='parrots128'; dim=128;
fid=fopen(filename,'r');
img=fread(fid,[dim,dim])';
mask=1; % между 1 и 8
nimg=bitget(img,mask);
imagesc(nimg), colormap(gray)
Двоичный код
```

```
clear;
filename='parrots128'; dim=128;
fid=fopen(filename,'r');
img=fread(fid,[dim,dim])';
mask=1 % между 1 и 8
a=bitshift(img,-1);
b=bitxor(img,a);
nimg=bitget(b,mask);
imagesc(nimg), colormap(gray)
Код Грея
```

Рис. 3.8. Разделение изображения на слои (Matlab).

43210	Gray	43210	Gray	43210	Gray	43210	Gray
00000	00000	<b>01000</b>	01100	<b>10000</b>	11000	<b>11000</b>	10100
00001	00001	01001	01101	10001	11001	11001	10101
00010	00011	<b>01010</b>	01111	<b>10010</b>	11011	<b>11010</b>	10111
00011	00010	01011	01110	10011	11010	11011	10110
00100	00110	<b>01100</b>	01010	<b>10100</b>	11110	<b>11100</b>	10010
00101	00111	01101	01011	10101	11111	11101	10011
00110	00101	<b>01110</b>	01001	<b>10110</b>	11101	<b>11110</b>	10001
00111	00100	01111	01000	10111	11100	11111	10000

a=linspace(0,31,32); b=bitshift(a,-1);  
 b=bitxor(a,b); dec2bin(b)

Табл. 3.9. Первые 32 двоичных и RGC кода.

В противоположность этому, коды рисунка (с) будут демонстрировать большую корреляцию в слоях старших разрядов, поскольку там смена между 0 и 1 происходит, соответственно, 1, 2 и 4 раза (по старшинству разрядов). А младшие разряды покажут слабую корреляцию пикселов, которая стремится к нулю, как при случайном распределении пикселов. Для сравнения на рис. 3.13 (а) показаны обычные двоичные коды. Видно, что в этих кодах разряды меняются существенно чаще.

Даже поверхностный взгляд на коды Грея рис. 3.13 (с) обнаруживает в них некоторую регулярность. Более тщательное исследование выявляет две важные особенности этих кодов, которые можно использовать при их построении. Первая особенность заключается в периодической смене значений всех пяти разрядов. Легко написать

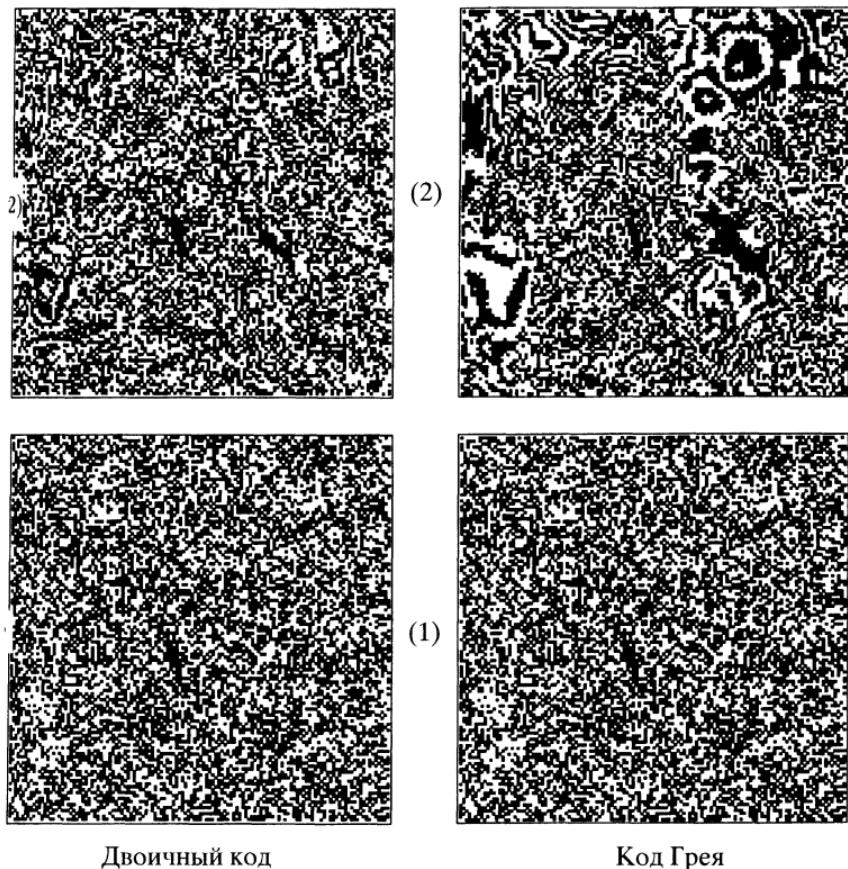


Рис. 3.10. (а) Слои 1 и 2 изображения «попугай».



(5)



(4)



(3)



Двоичный код

Код Грэя

Рис. 3.11. (а) Слои 3, 4 и 5 изображения «попугай».



(8)



(7)



(6)

Двоичный код



Код Грэя

Рис. 3.12. (а) Слои 6, 7 и 8 изображения «попугай».

программу, которая сначала установит все пять разрядов в 0, потом будет менять самый правый разряд у каждого второго кода, а остальные четыре разряда будет менять в середине периода своего правого соседнего разряда.

$b_4 b_3 b_2 b_1 b_0$		$b_4 b_3 b_2 b_1 b_0$		$b_4 b_3 b_2 b_1 b_0$	
0		0		0	
1		1		1	
2		2		2	
3		3		3	
4		4		4	
5		5		5	
6		6		6	
7		7		7	
8		8		8	
9		9		9	
10		10		10	
11		11		11	
12		12		12	
13		13		13	
14		14		14	
15		15		15	
16		16		16	
17		17		17	
18		18		18	
19		19		19	
20		20		20	
21		21		21	
22		22		22	
23		23		23	
24		24		24	
25		25		25	
26		26		26	
27		27		27	
28		28		28	
29		29		29	
30		30		30	
31		31		31	
1 3 7 15 31		4 8 16 2 1		1 2 4 8 16	

(a)

(b)

(c)

Рис. 3.13. Первые 32 двоичных и RGC кода.



Другая особенность кодов Грея состоит в том, что вторая половина таблицы является зеркальным отображением первой половины, если самый значимый бит установить в 1. Если первая половина таблицы уже вычислена, то вторая легко получается с помощью этого свойства.

На рис. 3.14 показаны круговые диаграммы, представляющие 4-х и 6-ти битные коды RGC (a), и обычные двоичные коды (b). Слои разрядов показаны в виде колец, причем слой самого значимого бита находится внутри. Видно, что максимальная угловая частота кода RGC в два раза меньше, чему у двоичного кода. Такое циклическое представление кодов Грея не нарушает их структуру, так как первый и последний код также отличаются ровно в одном бите.

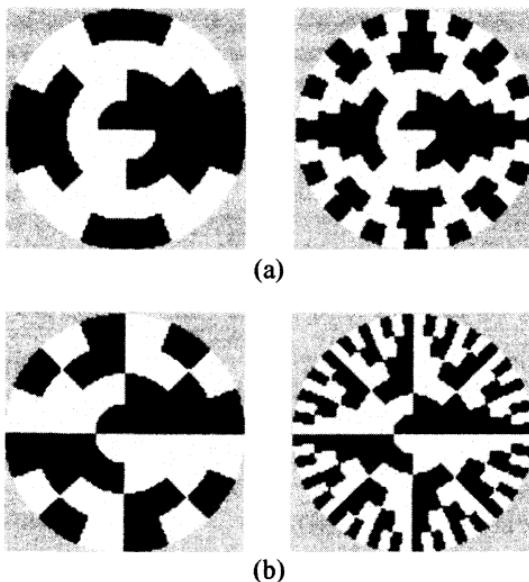


Рис. 3.14. Круговая диаграмма двоичных и RGC кодов.

К цветным изображениям также можно применять методы компрессии, разработанные для других типов изображений. Любой метод сжатия полутоновых образов позволяет сжимать цветные изображения. В цветном изображении каждый пиксель состоит из трех цветных компонент (например, RGB). Представим себе цветное изображение, в котором все компоненты состоят из одного байта. Пиксель описывается тремя байтами или 24 битами, но эти биты нельзя рассматривать как одно число. Три пикселя 118|206|12 и 117|206|12



различаются только на единицу в первом компоненте., поэтому они имеют близкие цвета. Если же их рассматривать в виде 24-битных чисел, то они будут сильно различаться, поскольку они разнятся в старшем бите. Любой метод компрессии, который будет основываться на таком различении пикселов, будет сильно неоптимальным. В этом смысле более подходящим является метод разделения изображения на цветные компоненты с их последующим независимым сжатием методом полутоновой компрессии.

Метод взвешенных контекстных деревьев изображений (см. [Solomon 2000 и Ekstrand 96]) является примером использования кодов RGC для сжатия образов.

### История кодов Грея

---

Эти коды названы в честь Франка Грея (Frank Gray), который запатентовал их использование в кодерах в 1953 году [Gray 53]. Однако эта работа была выполнена существенно раньше; он ее подал для патентования уже в 1947 году. Грей работал исследователем в лаборатории Белла. В течение 1930-х и 1940-х годов он получил несколько патентов в области телевидения. Если верить [Heath 72], то эти коды уже применялись Баудотом (J.M.E.Baudot) в телеграфии в 1870-х годах, но только после изобретения компьютеров эти коды стали широко известны.

Коды Баудота состояли из пяти бит на символ. С их помощью можно представить  $32 \times 2 - 2 = 62$  символа (каждый код имеет два смысла или значения, смысл обозначался LS и FS кодами). Они стали весьма популярными, и в 1950 они были приняты как стандарт N1 международного телеграфа. Они также использовались во многих компьютерах первого и второго поколения.

В августе 1972 в журнале *Scientific American* были опубликованы две интересные статьи на тему кодов Грея: одна про происхождение двоичных кодов [Heath 72], а другая – [Gardner 72] про некоторые развлекательные аспекты использования этих кодов.

---

### 3.3.2. Метрики ошибок

Разработчикам методов сжатия изображений с частичной потерей информации необходимы стандартные метрики для измерения расхождения восстановленных изображений и исходных изображений. Чем ближе восстановленный образ к исходному, тем больше должна быть эта метрика (ее удобно называть «метрикой сходства»). Эта метрика должна быть безразмерной и не слишком чувствительной к малым изменениям восстанавливаемого изображения. Общепринятой величиной, используемой для этих целей, служит *пиковое отношение сигнал/шум* (PSNR) (peak signal to noise ratio). Оно известно всем, кто работает в этой области, его легко вычислять, но оно имеет достаточно ограниченное, приближенное отношение к расхождениям, которые обнаруживаются органами зрения человека. Высокое

значение PSNR означает определенную схожесть реконструированного и исходного изображений, но оно не дает гарантию того, что зрителю понравится восстановленный образ.

Обозначим через  $P_i$  пиксели исходного изображения, а пиксели восстановленного изображения пусть будут  $Q_i$  (где  $1 \leq i \leq n$ ). Дадим сначала определение *среднеквадратической ошибке* (MSE, mean square error), которая равна

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (P_i - Q_i)^2. \quad (3.2)$$

Эта величина равна среднему квадратов ошибок (разностей пикселов) двух изображений. Число RMSE (корень среднеквадратической ошибки) определяется как квадратный корень числа MSE, а величина PSNR равна, по определению,

$$\text{PSNR} = 20 \log_{10} \frac{\max_i |P_i|}{\text{RMSE}}.$$

Абсолютная величина, обычно, бывает не нужна, поскольку пиксели редко бывают отрицательными. Для двухуровневых изображений числитель равен 1. Для полутоновых образов, пиксели которых состоят из 8 битов, числитель равен 255. Для изображений используется только компонента цветности.

Чем больше схожесть между образами, тем меньше величина RMSE, а, значит, больше PSNR. Число PSNR безразмерно, поскольку единицами измерения и числителя, и знаменателя служат величины пикселов. Тем не менее, из-за использования логарифмов говорится, что число PSNR измеряется в *дб* (дБ, см. § 6.1). Использование логарифмов сглаживает RMSE, делает эту величину менее чувствительной. Например, деление RMSE на 10 означает умножение PSNR на 2. Отметим, что PSNR не имеет абсолютного значения. Бессмысленно говорить, что если PSNR равно, скажем, 25, то это хорошо. Величины PSNR используются только для сравнения производительности различных методов сжатия и для изучения влияния разных параметров на производительность того или иного алгоритма. К примеру, комитет MPEG использует субъективный порог  $\text{PSNR} = 0.5$  дБ при включении кодовой оптимизации, поскольку считает, что улучшение на эту величину будет заметно глазу.

Обычно, величина PSNR варьируется в пределах от 20 до 40. Если значения пикселов находятся в интервале [0,255], то RMSE, равное 25.5, дает PSNR, равное 20, а при RMSE равном 2.55 величина

PSNR – 40. Значение RMSE равное нулю (совпадение изображений), дает для PSNR результат бесконечность (более точно, неопределенность). При RMSE равном 255 число PSNR равно 0, а если RMSE больше, чем 255, то PSNR будет отрицательным.

Читателю будет полезно ответить на следующий вопрос: если максимальное значение пикселя равно 255, может ли RMSE быть больше 255? Ответ будет «нет». Если величины пикселов принадлежат интервалу  $[0, 255]$ , то разность  $(P_i - Q_i)$  будет не более 255. В наихудшем случае она равна 255. Легко видеть, что тогда RMSE будет равно 255.

Некоторые авторы определяют PSNR следующей формулой

$$\text{PSNR} = 10 \log_{10} \frac{\max_i |P_i|^2}{\text{RMSE}}.$$

Для того, чтобы эти формулы давали одинаковые результаты, логарифм умножается на 10 а не на 20, поскольку  $\log_{10} A^2 = 2 \log_{10} A$ .

Характеристикой, близкой к этим величинам, является *отношение сигнал/шум* (SNR, signal to noise ratio). Оно определяется по формуле (в знаменателе стоит число RMSE исходного образа)

$$\text{SNR} = 20 \log_{10} \frac{\sqrt{\frac{1}{n} \sum_{i=1}^n P_i^2}}{\text{RMSE}}.$$

На рис. 3.15 приведена функция для системы Matlab, вычисляющая PSNR для двух изображений. Она вызывается как `PSNR(A, B)`, где `A` и `B` – два файла изображений. Изображения должны иметь одинаковое разрешение и значения пикселов должны лежать в интервале  $[0, 1]$ .

Другой родственной величиной для PSNR служит отношение сигнал/шум квантования (SQNR, signal to quantization noise ratio). Эта величина измеряет влияние эффекта квантования на качество сигнала. Она определяется выражением

$$\text{SQNR} = 10 \log_{10} \frac{\text{мощность сигнала}}{\text{ошибка квантования}},$$

где ошибка квантования равна разности между квантованным и исходным сигналами.

Другой подход к сравнению оригинального и восстановленного изображения состоит в построении разностного изображения и оценивании его качества визуальным наблюдением. Интуитивно разностное изображение равно  $D_i = P_i - Q_i$ , однако такой образ трудно оценить на глаз, так как значения пикселов  $D_i$  являются малыми

числами. Если нулевое значение соответствует белому цвету, то такой разностный образ будет почти невиден. Наоборот, если нуль соответствует черному цвету, то разность будет слишком темной для выработки точного суждения. Лучшие результаты можно получить, используя формулу

$$D_i = a(P_i - Q_i) + b,$$

где  $a$  – параметр амплитуды (обычно малое число, например, 2), а  $b$  – половина максимального значения пикселя (обычно, это 128). Параметр  $a$  нужен для увеличения малых разностей, а  $b$  сдвигает разностный образ из крайне белого в сторону черного в область комфорtnого для глаза серого цвета.

```
function PSNR(A,B)
if A==B
    error('Images are identical; PSNR is undefined')
end
max2_A=max(max(A)); max2_B=max(max(B));
min2_A=min(min(A)); min2_B=min(min(B));
if max2_A>1 | max2_B>1 | min2_A<0 | min2_B<0
    error('pixels must be in [0,1]')
end
differ=A-B;
decib=20*log10(1/(sqrt(mean(mean(differ.^2))))) ;
disp(sprintf('PSNR = +%5.2f dB',decib))
```

Рис. 3.15. Функция Matlab для вычисления PSNR.

## 3.4. Интуитивные методы

Легко придумать простые интуитивные методы сжатия изображений. Мы приведем их здесь только с иллюстративными целями. На практике, конечно, используются более изощренные (и эффективные) методы компрессии изображений.

### 3.4.1. Подвыборка

Подвыборка, возможно, является самым простым методом сжатия изображения. Простейший способ подвыборки – это просто отбросить некоторые пиксели. Кодер может, например, игнорировать каждую вторую строку и каждый второй столбец изображения и записывать оставшиеся пиксели в сжатый файл. Это составит 25%



от исходного. Декодер вводит сжатые данные и использует каждый пиксель для создания четырех одинаковых пикселов реконструированного изображения. Это, конечно, приводит к потере многих деталей изображения. Такой метод редко приводит к удовлетворительным результатам. Заметим, что для такого метода коэффициент сжатия известен заранее.

Более приемлемые результаты можно получить, если записывать в сжатый файл среднее каждого блока из 4 пикселов исходного изображения. При этом ни один пиксель не игнорируется, но метод по-прежнему крайне примитивен, поскольку хороший метод сжатия должен отбрасывать детали, незаметные глазу.

Лучшие результаты (но худшее сжатие) получаются, если цветное представление образа трансформируется из обычного (например, RGB) в представление с компонентами светимости и цветности. Кодер делает подвыборку в двух компонентах цветности, а компоненту яркости оставляет нетронутой. Если считать, что все компоненты используют одно и то же число бит, то две компоненты цветности занимают  $2/3$  размера исходного файла. Сделав подвыборку, сокращаем этот размер до  $25\%$  от  $2/3$ , то есть до  $1/6$ . Тогда размер сжатого файла будет складываться из  $1/3$  (для несжатой компоненты светимости) плюс  $1/6$  (для двух компонент цветности), что равно  $1/2$  исходного размера.

### 3.4.2. Квантование

Термин «квантование» при использовании в сжатии данных означает округление вещественных чисел до целых или преобразование целых чисел в меньшие целые. Существует два вида квантования, скалярное и векторное. Скалярное квантование является интуитивным методом, при котором не всегда теряется только малозначимая информация. При использовании второго метода можно добиться лучших результатов, поэтому мы его приводим ниже.

Изображение делится на равные блоки пикселов, которые называются *векторами*, а у кодера имеется список таких же блоков, называемый *кодовой книгой*. Каждый блок *B* изображения сравнивается со всеми блоками из кодовой книги и находится «ближайший» к *B* блок *C*. После чего в выходной файл записывается указатель на блок *C*. Если размер указателя меньше размера блока, то достигается сжатие. На рис. 3.16 показан пример такой деятельности.

Проблемы выбора кодовой книги и поиска в ней ближайшего



блока обсуждаются в [Salomon 2000] с применением к алгоритму, предложенному в [Linde et al. 80]. Отметим, что в методе векторного квантования коэффициент сжатия известен заранее.



Рис. 3.16. Интуитивное векторное квантование.

### 3.5. Преобразование изображений

Понятие преобразования широко применяется в математике. С его помощью решаются многие задачи в различных областях науки. Основная идея состоит в изменении математической величины (числа, вектора, функции или другого объекта) с целью придания ей другой формы, в которой она имеет, возможно, непривычный вид, но имеет полезные свойства. Преобразованная величина используется при решении задачи или при совершении некоторых вычислений, после чего к результату применяется обратное преобразование для возврата к исходной форме.

Простым иллюстративным примером могут служить арифметические операции над римскими числами. Древние римляне, по-видимому, знали, как оперировать с такими числами, однако, если требуется, скажем, перемножить два римских числа, то будет более удобно преобразовать их в современную (арабскую) форму, перемножить, а потом представить результат в виде римского числа. Вот простой пример:

$$\text{XCVI} \times \text{XII} \rightarrow 96 \times 12 = 1152 \rightarrow \text{MCLII}.$$

Изображение можно сжать, преобразуя его пиксели (которые коррелированы) в представление, где они будут декоррелированными. Произойдет сжатие, если новые величины будут, в среднем,

меньше исходных. Сжатие с потерей качества можно затем произвести с помощью квантования результата преобразования. Декодер читает сжатый файл и восстанавливает (точно или приближенно) исходные данные, применяя обратное преобразование. В этом параграфе рассматриваются ортогональные преобразования. В § 4.3 будет обсуждаться поддиапазонное преобразование.

Слово *декоррелированные* означает, что преобразованные величины являются статистически независимыми. В результате их можно кодировать независимо, что позволяет построить более простую статистическую модель. Образ можно сжать, если в нем имеется определенная избыточность. Избыточность проистекает от корреляции пикселов. Если перевести образ в представление, в котором пиксели декоррелированы, то одновременно произойдет удаление избыточности и образ будет полностью сжат.

Начнем с простого примера, в котором образ сканируется растровым способом (то есть, строка за строкой) и группируется в пары прилегающих пикселов. Поскольку пиксели коррелированы, два пикселя  $(x, y)$  в паре, обычно, имеют близкие значения. Рассмотрим теперь эти пары в виде точек на плоскости и отметим их на графике. Известно, что точки вида  $(x, x)$  лежат на прямой с наклоном  $45^\circ$ , уравнение которой имеет вид  $y = x$ , поэтому можно ожидать, что все точки будут сконцентрированы около этой прямой. Рис. 3.17a дает такой график для типичного изображения, где значения пикселов лежат в интервале  $[0, 255]$ . Большинство точек изображают «облако» около этой линии, и только некоторые точки лежат вдали от диагонали. Теперь мы преобразуем эту картинку с помощью поворота на угол  $45^\circ$  по часовой стрелке вокруг начала координат, так, чтобы диагональ легла на ось  $x$  (рис. 3.17b). Это делается с помощью простого преобразования

$$\begin{aligned} (x^*, y^*) &= (x, y) \begin{pmatrix} \cos 45^\circ & -\sin 45^\circ \\ \sin 45^\circ & \cos 45^\circ \end{pmatrix} = \\ &= (x, y) \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} = (x, y) \mathbf{R}, \end{aligned} \quad (3.3)$$

где матрица поворота  $\mathbf{R}$  является ортогональной (то есть, скалярное произведение строк на себя равно 1, а скалярное произведение векторов-строк друг на друга равно 0; то же самое верно и для векторов-столбцов). Обратным преобразованием служит поворот на  $45^\circ$  против часовой стрелки, который запишется в виде

$$(x, y) = (x^*, y^*) \mathbf{R}^{-1} = (x^*, y^*) \mathbf{R}^\top = (x^*, y^*) \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}. \quad (3.4)$$

(Обращение ортогональных матриц делается с помощью простого транспонирования.)

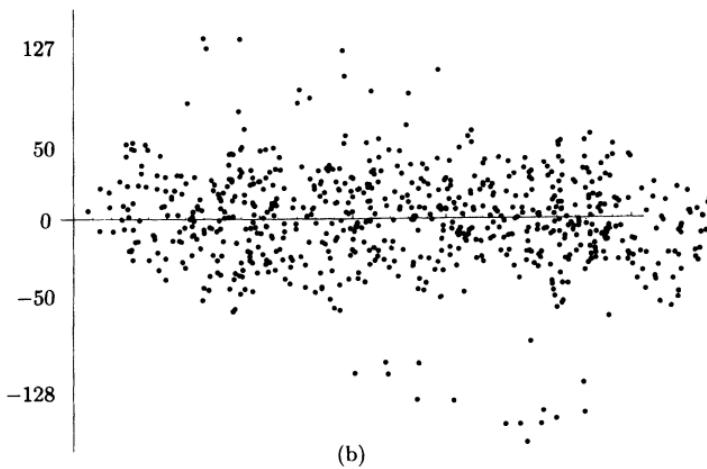
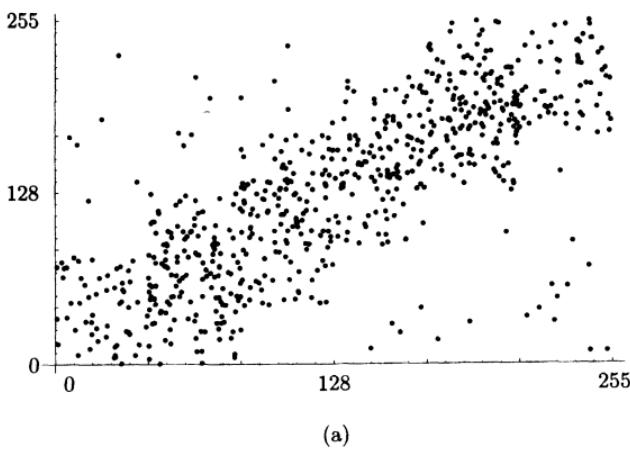


Рис. 3.17. Поворот облака точек.

Очевидно, что большинство точек преобразованного «облака» будут иметь координату  $y$ , близкую к нулю, а координата  $x$  изменится не слишком сильно. Рис. 3.18а,б изображают распределение координат  $x$  и  $y$  (то есть, пикселов с четными и нечетными номерами) типичного  $128 \times 128 \times 8$  полутонового изображения до вращения.



Ясно, что эти распределения отличаются не слишком. На рис. 3.18c,d показаны распределения координат после поворота. Распределение координат  $x$  почти не изменилось (увеличилась дисперсия), в то время как распределение координат  $y$  сконцентрировалось около нуля. Программа на Matlab, которая строит эти графики также приведена на рисунке. (На рис. 3.18d координата  $y$  сконцентрирована около 100, но это произошло из-за сдвига графика вправо, так как некоторые координаты были близки к  $-101$ , и их пришлось сдвинуть для попадания в массив Matlab, у которого индекс всегда начинается с единицы.)

Поскольку координаты точек известны до и после преобразования, легко вычислить уменьшение корреляции. Сумма  $\sum_i x_i y_i$  называется *перекрестной корреляцией* точек  $(x_i, y_i)$ .

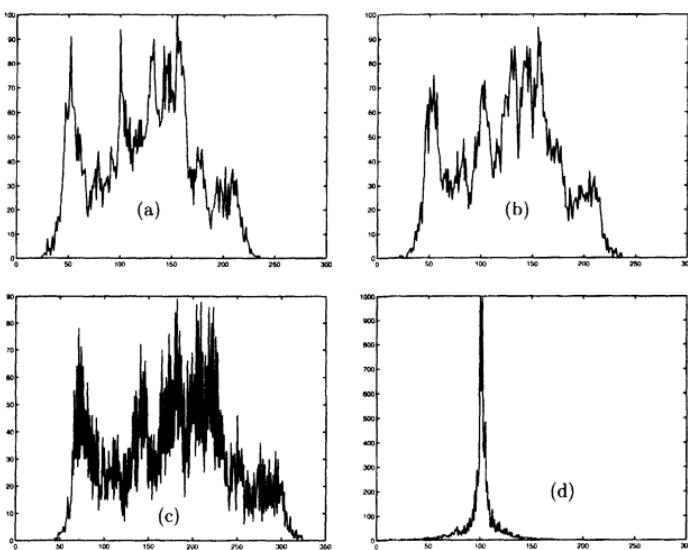
Следующий пример поясняет смысл этой величины. Повернем точки  $(5,5)$ ,  $(6,7)$ ,  $(12.1,13.2)$ ,  $(23,25)$  и  $(32,29)$  по часовой стрелке на  $45^\circ$  и вычислим перекрестную корреляцию до и после поворота. На рис. 3.19 приведена программы Matlab, вычисляющая координаты повернутых точек. Они равны

$$(7.071, 0), (9.19, 0.7071), (17.9, 0.78), (33.9, 1.41), (43.13, -2.12)$$

(заметьте, что координаты  $y$  стали малыми числами). Видно, что значение перекрестной корреляции уменьшилось с 1729.72 до поворота до  $-23.0846$  после поворота. Замечательное сокращение!

Теперь можно сжать образ, просто записав преобразованные координаты в выходной файл. Если допустима некоторая потеря информации, то можно сделать квантование всех пикселов, что даст малые значения пикселов. Можно также записывать в сжатый файл все нечетные пиксели (те, которые являются  $x$  координатами в парах), а за ними записать все четные пиксели. Эти две последовательности называются *векторами коэффициентов* преобразования. Вторая последовательность состоит из малых чисел, и, возможно, после ее квантования возникнут серии нулей, которые можно будет еще лучше сжать.

Легко показать, что полная дисперсия пикселов, которая определяется суммой  $\sum_i (x_i^2 + y_i^2)$ , не меняется при повороте, поскольку матрица этого преобразования является ортогональной. Однако дисперсия новых координат  $y$  стала малой, поэтому возросла дисперсия коэффициентов  $x$ . Дисперсию иногда называют *энергией* распределения пикселов. Поэтому мы можем сказать, что поворот концентрирует энергию в координатах  $x$ , с помощью чего достигается сжатие изображения.



```

filename='lena128'; dim=128;
xdist=zeros(256,1); ydist=zeros(256,1);
fid=fopen(filename,'r');
img=fread(fid,[dim,dim])';
for col=1:2:dim-1
    for row=1:dim
        x=img(row,col)+1; y=img(row,col+1)+1;
        xdist(x)=xdist(x)+1; ydist(y)=ydist(y)+1;
    end
end
figure(1), plot(xdist), colormap(gray) % распред. x&y
figure(2), plot(ydist), colormap(gray) % до поворота
xdist=zeros(325,1); % clear arrays
ydist=zeros(256,1);
for col=1:2:dim-1
    for row=1:dim
        x=round((img(row,col)+img(row,col+1))*0.7071);
        y=round((-img(row,col)+img(row,col+1))*0.7071)+101;
        xdist(x)=xdist(x)+1; ydist(y)=ydist(y)+1;
    end
end
figure(3), plot(xdist), colormap(gray) % распред. x&y
figure(4), plot(ydist), colormap(gray) % после поворота

```

Рис. 3.18. Распределение пикселов до и после поворота.

Концентрирование энергии в одной координате имеет и другое преимущество. Можно делать квантование этой координаты более



точным, чем квантование второй координаты. Такой способ квантования приводит к лучшему сжатию.

```
p = {{5,5},{6, 7},{12.1,13.2},{23,25},{32,29}};  
rot = {{0.7071,-0.7071},{0.7071,0.7071}};  
Sum[p[[i,1]]p[[i,2]], {i,5}]  
q=p.rot  
Sum[q[[i,1]]q[[i,2]], {i,5}]
```

Рис. 3.19. Поворот пяти точек.

Следующий простой пример иллюстрирует возможности этого ортогонального преобразования. Начнем с точки  $(4, 5)$  с близкими координатами. С помощью уравнения (3.3) эта точка переходит в  $(4, 5)\mathbf{R} = (9, 1)/\sqrt{2} \approx (6.36396, 0.7071)$ . Энергии этих точек равны:  $4^2 + 5^2 = 41 = (9^2 + 1^2)/2$ . Если удалить координату (4) исходной точки, то получится ошибка  $4^2/41 = 0.39$ . Однако, если удалить меньшую из двух координат преобразованной точки  $(0.7071)$ , то ошибка будет всего  $0.7071^2/41 = 0.012$ . Эту ошибку можно также вычислить, исходя из реконструированной точки. Применим обратное преобразование (уравнение (3.4)) к точке  $(9, 1)/\sqrt{2}$ . Получим, конечно, исходную точку  $(4, 5)$ . Сделав то же самое для точки  $(9, 0)/\sqrt{2}$ , получим после округления точку  $(4.5, 4.5)$ . Разность энергий исходной и реконструированной точек будет равна той же малой величине

$$\frac{[(4^2 + 5^2) - (4.5^2 + 4.5^2)]}{4^2 + 5^2} = \frac{41 - 40.5}{41} = 0.012.$$

Это простое преобразование легко обобщить на случай любого числа измерений. Вместо пар, можно выбирать тройки точек, триплеты. Каждый триплет становится точкой трехмерного пространства, а все точки образуют «облако» вокруг прямой, проходящей через начало координат под углом  $45^\circ$  к каждой координатной оси. Если эту прямую повернуть так, что она ляжет на ось  $x$ , то координаты  $y$  и  $z$  точек «облака» станут малыми числами. Такое преобразование совершается с помощью умножения каждой точки на некоторую матрицу размера  $3 \times 3$ , которая, конечно, является ортогональной. Вторая и третья компоненты координат преобразованных точек будут малыми числами, поэтому координаты всех точек следует разделить на три вектора коэффициентов. Для лучшего сжатия необходимо, чтобы квантование этих векторов коэффициентов делалось с разной степенью точности.

Этот метод легко распространить на более высокие размерности, с той лишь разницей, что получающиеся пространства уже нельзя будет представить зрительно. Однако, соответствующие матрицы преобразований легко выписываются. Единственно, что приходится учитывать, это то, что размерность не должна быть слишком большой, поскольку результат сжатия на основе поворота зависит от корреляции близких пикселов. Например, если объединять по 24 соседних пиксела в одну точку 24-мерного пространства, то полученные точки, вообще говоря, не будут лежать в малой окрестности «прямой под углом  $45^\circ$  к осям координат», поскольку не будет корреляции между пикселом и его дальним соседом. Поэтому после поворота, последние 23 координаты преобразованных точек уже не будут малыми. Наблюдения показывают, что корреляция пикселов сохраняется до размерности восемь, но редко дальше.

Метод JPEG, описанный в § 3.7, делит изображение на блоки пикселов размера  $8 \times 8$  и поворачивает каждый блок два раза с помощью уравнения (3.9), которое будет объяснено в § 3.5.3. Это двойное вращение дает множества, состоящие из 64 преобразованных величин, из которых первая, называемая «коэффициент DC», — большая, а все остальные 63 («коэффициенты AC») — обычно маленькие. Таким образом, это преобразование концентрирует энергию в первой компоненте из 64. Далее множество коэффициентов DC и 63 множества коэффициентов AC следует квантовать раздельно (метод JPEG делает это немного иначе, см. § 3.7.4).

### 3.5.1. Ортогональные преобразования

Преобразования, которые используются для сжатия изображений должны быть быстрыми, и, по возможности, легко реализуемыми на компьютере. Это прежде всего предполагает, что такие преобразования должны быть *линейными*. То есть, преобразованные величины  $c_i$  являются линейными комбинациями (суммами с некоторыми множителями или весами) исходных величин (пикселов)  $d_j$ , причем соответствующим множителем или весом служит некоторое число  $w_{ij}$  (коэффициент преобразования). Значит,  $c_i = \sum_j d_j w_{ij}$ , где  $i, j = 1, 2, \dots, n$ . Например, при  $n = 4$  это преобразование можно записать в матричной форме

$$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{pmatrix} \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{pmatrix},$$

которая в общем случае примет следующий вид:  $\mathbf{C} = \mathbf{W} \cdot \mathbf{D}$ . Каждый вектор-столбец матрицы  $\mathbf{W}$  называется «базисным вектором».

Важной задачей является определение коэффициентов преобразования  $w_{ij}$ . Основное требование заключается в том, чтобы после преобразования величина  $c_1$  была бы большой, а все остальные величины  $c_2, c_3, \dots$  стали бы малыми. Основное соотношение  $c_i = \sum_j d_j w_{ij}$  предполагает, что  $c_i$  будет большим, если веса  $w_{ij}$  будут усиливать соответствующие величины  $d_j$ . Это произойдет, например, если компоненты векторов  $w_{ij}$  и  $d_j$  имеют близкие значения и одинаковые знаки. Наоборот,  $c_i$  будет малым, если веса  $w_{ij}$  будут малыми, и половина из них будет иметь знак, противоположный знаку соответствующего числа  $d_j$ . Поэтому, если получаются большие  $c_i$ , то векторы  $w_{ij}$  имеют сходство с исходным вектором  $d_j$ , а малые  $c_i$  означают, что компоненты  $w_{ij}$  сильно отличаются от  $d_j$ . Следовательно, базисные векторы  $w_{ij}$  можно интерпретировать как инструмент для извлечения некоторых характерных признаков исходного вектора.

На практике веса  $w_{ij}$  не должны зависеть от исходных данных. В противном случае, их придется добавлять в сжатый файл для использования декодером. Это соображение, а также тот факт, что исходные данные являются пикселями, то есть, неотрицательными величинами, определяет способ выбора базисных векторов. Первый вектор, тот, который, порождает  $c_1$ , должен состоять из близких, возможно, совпадающих чисел. Он будет усиливать неотрицательные величины пикселов. А все остальные векторы базиса должны наполовину состоять из положительных чисел, а на другую половину – из отрицательных. После умножения на положительные величины и их сложения, результат будет малым числом. (Это особенно верно, когда исходные данные близки, а мы знаем, что соседние пиксели имеют, обычно, близкие величины.) Напомним, что базисные векторы представляют собой некоторый инструмент для извлечения особенностей из исходных данных. Поэтому хорошим выбором будут базисные векторы, которые сильно различаются друг от друга и, поэтому, могут извлекать разные особенности. Это приводит к мысли, что базисные векторы должны быть *взаимно ортогональными*. Если матрица преобразования  $\mathbf{W}$  состоит из ортогональных векторов, то преобразование называется *ортогональным*. Другое наблюдение, позволяющее правильно выбирать базисные векторы, состоит в том, что эти векторы должны иметь все большие частоты изменения знака, чтобы извлекать, так сказать, высокочастотные



характеристики сжимаемых данных при вычислении преобразованных величин.

Этим свойствам удовлетворяет следующая ортогональная матрица:

$$\mathbf{W} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{pmatrix}. \quad (3.5)$$

Первый базисный вектор (верхняя строка  $\mathbf{W}$ ) состоит из одних единиц, поэтому его частота равна нулю. Все остальные векторы имеют две  $+1$  и две  $-1$ , поэтому они дадут маленькие преобразованные величины, а их частоты (измеренные количеством смен знаков в строке) возрастают. Эта матрица подобна матрице преобразования Адамара–Уолша (см. уравнение (3.11)). Для примера, преобразуем начальный вектор  $(4, 6, 5, 2)$ :

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{pmatrix} \begin{pmatrix} 4 \\ 6 \\ 5 \\ 2 \end{pmatrix} = \begin{pmatrix} 17 \\ 3 \\ -5 \\ 1 \end{pmatrix}.$$

Результат вполне ободряющий, поскольку число  $c_1$  стало большим (по сравнению с исходными данными), а два других числа стали малыми. Вычислим энергию исходных и преобразованных данных. Начальная энергия равна  $4^2 + 6^2 + 5^2 + 2^2 = 81$ , а после преобразования энергия стала  $17^2 + 3^2 + (-5)^2 + 1^2 = 324$ , что в четыре раза больше. Энергию можно сохранить, если умножить матрицу преобразования  $\mathbf{W}$  на коэффициент  $1/2$ . Новое произведение  $\mathbf{W} \cdot (4, 6, 5, 2)^T$  будет равно  $(17/2, 3/2, -5/2, 1/2)$ . Итак, энергия сохраняется и концентрируется в первой компоненте, и она теперь составляет  $8.5^2/81 = 89\%$  от общей энергии исходных данных, в которых на долю первой компоненты приходилось всего 20%.

Другое преимущество матрицы  $\mathbf{W}$  состоит в том, что она же делает обратное преобразование. Исходные данные  $(4, 6, 5, 2)$  восстанавливаются с помощью произведения  $\mathbf{W} \cdot (17/2, 3/2, -5/2, 1/2)^T$ .

Теперь мы в состоянии оценить достоинства этого преобразования. Квантуем преобразованный вектор  $(8.5, 1.5, -2.5, 0.5)$  с помощью его округления до целого и получаем  $(9, 1, -3, 0)$ . Делаем обратное преобразование и получаем вектор  $(3.5, 6.5, 5.5, 2.5)$ . В аналогичном эксперименте мы просто удалили два наименьших числа и

получим  $(8.5, 0, -2.5, 0)$ , а потом сделаем обратное преобразование этого грубо квантованного вектора. Это приводит к восстановленным данным  $(3, 5.5, 5.5, 3)$ , которые также весьма близки к исходным. Итак, наш вывод: даже это простое и интуитивное преобразование является хорошим инструментом для «выжимания» избыточности из исходных данных. Более изощренные преобразования дают результаты, которые позволяют восстанавливать данные с высокой степенью схожести даже при весьма грубом квантовании.

*Одни художники отображают солнце в желтое пятно, а другие – желтое пятно в солнце.*

— Пабло Пикассо

### 3.5.2. Матричные преобразования

Рассмотрим двумерный массив данных, представленный в виде матрицы размером  $4 \times 4$

$$\mathbf{D} = \begin{pmatrix} 4 & 7 & 6 & 9 \\ 6 & 8 & 3 & 6 \\ 5 & 4 & 7 & 6 \\ 2 & 4 & 5 & 9 \end{pmatrix}.$$

(здесь в первом столбце стоит вектор из предыдущего примера). Применим наше простое преобразование к каждому столбцу матрицы  $\mathbf{D}$ . Результат равен

$$\mathbf{C}' = \mathbf{W} \cdot \mathbf{D} = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{pmatrix} \mathbf{D} = \begin{pmatrix} 8.5 & 11.5 & 10.5 & 15 \\ 1.5 & 3.5 & -1.5 & 0 \\ -2.5 & -0.5 & 0.5 & 3 \\ 0.5 & -0.5 & 2.5 & 0 \end{pmatrix}.$$

Каждый столбец матрицы  $\mathbf{C}'$  получен преобразованием столбцов матрицы  $\mathbf{D}$ . Заметим, что верхние элементы столбцов матрицы  $\mathbf{C}'$  являются доминирующими. Кроме того, все столбцы имеют ту же энергию, что и до преобразования. Будем считать матрицу  $\mathbf{C}'$  результатом первого этапа двухстадийного процесса преобразования матрицы  $\mathbf{D}$ . На втором этапе сделаем преобразование строк матрицы  $\mathbf{C}'$ . Для этого умножим матрицу  $\mathbf{C}'$  на транспонированную матрицу  $\mathbf{W}^T$ . Наша конкретная матрица  $\mathbf{W}$  является симметричной, поэтому можно записать:  $\mathbf{C} = \mathbf{C}' \mathbf{W}^T = \mathbf{W} \mathbf{D} \mathbf{W}^T = \mathbf{W} \mathbf{D} \mathbf{W}$  или

$$\begin{aligned}
 \mathbf{C} &= \begin{pmatrix} 8.5 & 11.5 & 10.5 & 15 \\ 1.5 & 3.5 & -1.5 & 0 \\ -2.5 & -0.5 & 0.5 & 3 \\ 0.5 & -0.5 & 2.5 & 0 \end{pmatrix} \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{pmatrix} = \\
 &= \begin{pmatrix} 22.75 & -2.75 & 0.75 & -3.75 \\ 1.75 & 3.25 & -0.25 & -1.75 \\ 0.25 & -3.25 & 0.25 & -2.25 \\ 1.25 & -1.25 & -0.75 & 1.75 \end{pmatrix}.
 \end{aligned}$$

Самый верхний левый элемент матрицы **C** доминирует. В нем сосредоточено 89% от общей энергии, равной 579, исходной матрицы **D**. Следовательно двухстадийное преобразование матричных данных сокращает корреляцию по обоим направлениям: по вертикали и по горизонтали.

Далее будут обсуждаться следующие преобразования:

1. Дискретное косинус-преобразование (DCT, discrete cosine transform, см. § 3.5.3 и § 3.7.2) является хорошо изученным и весьма эффективным преобразованием, которое применяется в таких методах компрессии, как JPEG и MPEG. Известные алгоритмы быстрого вычисления DCT делают этот метод особенно притягательным в конкретных приложениях.
2. Преобразование Кархунене–Лоэвэ (KLT, Karhunen–Loëve transform, § 3.5.8) является теоретически наилучшим с точки зрения концентрации энергии (или, что то же самое, удаления корреляции пикселов). К сожалению, его коэффициенты не фиксированы, а зависят от исходных данных. Вычисление этих коэффициентов (базиса преобразования) делается медленно, как и нахождение самих преобразованных величин. Поскольку преобразование зависит от исходных данных, приходится сохранять его коэффициенты в сжатом файле. По этим причинам, а также из-за того, что DCT дает примерно то же качество, но с большим выигрышем по быстродействию, метод KLT редко используется на практике.
3. Преобразование Уолша–Адамара (WHT, Walsh–Hadamard transform, § 3.5.6) быстро вычисляется (при этом используется только сложение и вычитание), но его характеристики, выраженные в терминах концентрации энергии, хуже, чем у DCT.
4. Преобразование Хаара [Stollnitz 96] является очень простым и быстрым. Оно является простейшим вейвлетным преобразованием, которое будет обсуждаться в § 3.5.7 и в главе 4.

### 3.5.3. Дискретное косинус-преобразование

Прежде всего мы рассмотрим одномерное (векторное) преобразование DCT (в приложениях используется *двумерное (матричное)* косинус-преобразование, но векторное DCT проще понять, и оно основано на тех же принципах). На рис. 3.20 показано восемь волн косинуса,  $w(f) = \cos(f\theta)$ , при  $0 \leq \theta \leq \pi$ , с частотами  $f = 0, 1, \dots, 7$ . На каждом графике отмечено восемь значений функции  $w(f)$  с абсциссами

$$\theta = \frac{\pi}{16}, \frac{3\pi}{16}, \frac{5\pi}{16}, \frac{7\pi}{16}, \frac{9\pi}{16}, \frac{11\pi}{16}, \frac{13\pi}{16}, \frac{15\pi}{16}, \quad (3.6)$$

которые формируют базисный вектор  $\mathbf{v}_f$ . В результате получится восемь векторов  $\mathbf{v}_f, f = 0, 1, \dots, 7$  (всего 64 числа), которые представлены в табл. 3.21. Они служат базисом одномерного косинус-преобразования. Отметим схожесть этой таблицы с матрицей  $\mathbf{W}$  из уравнения (3.5). В обоих случаях частота смены знаков возрастает по строкам.

```

dct[pw_]:=Plot[Cos[pw t], {t,0,Pi}, DisplayFunction->Identity,
  AspectRatio->Automatic];
dcdot[pw_]:=ListPlot[Table[{t,Cos[pw t]},{t,Pi/16,15Pi/16,Pi/8}],
  DisplayFunction->Identity]
Show[dct[0],dcdot[0], Prolog->AbsolutePointSize[4],
  DisplayFunction->$DisplayFunction]
...
Show[dct[7],dcdot[7], Prolog->AbsolutePointSize[4],
  DisplayFunction->$DisplayFunction]

```

Программа для вычисления рис. 3.20 (Mathematica).

Можно показать, что все векторы  $\mathbf{v}_i$  ортогональны между собой (из-за специального выбора восьми точек отсчета  $\theta$ ). То же самое можно обнаружить прямым вычислением с помощью подходящей математической программы. Значит, эти восемь векторов можно поместить в матрицу размером  $8 \times 8$  и рассмотреть соответствующее ей ортогональное преобразование – вращение в восьмимерном пространстве, которое называется одномерным дискретным косинус-преобразованием (DCT). Двумерное DCT можно также интерпретировать как двойное вращение. Это преобразование будет обсуждаться на стр. 155.

Одномерное DCT имеет также другую интерпретацию. Можно рассмотреть векторное пространство, базисом которого служат

векторы  $\mathbf{v}_i$ , и выразить любой вектор  $\mathbf{p}$  этого пространства в виде линейной комбинации векторов  $\mathbf{v}_i$ .

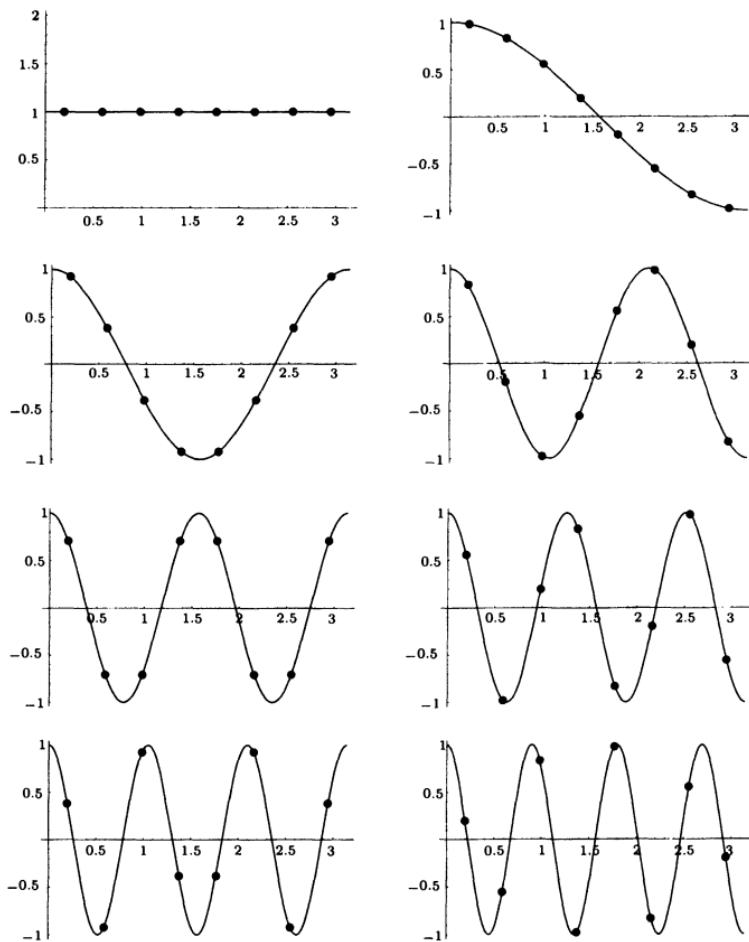


Рис. 3.20. Вычисление одномерного DCT.

Например, выберем 8 (коррелированных) чисел  $\mathbf{p} = (0.6, 0.5, 0.4, 0.5, 0.6, 0.5, 0.4, 0.55)$  в качестве тестовых данных. Выразим вектор  $\mathbf{p}$  в виде суммы  $\mathbf{p} = \sum_i w_i \mathbf{v}_i$  восьми векторов  $\mathbf{v}_i$ . Решив эту систему из 8 линейных уравнений, находим восемь весов

$$w_0 = 0.506, \quad w_1 = 0.0143, \quad w_2 = 0.0115, \quad w_3 = 0.0439, \\ w_4 = 0.0795, \quad w_5 = -0.0432, \quad w_6 = 0.00478, \quad w_7 = -0.0077.$$



Вес  $w_0$  не сильно отличается от элементов вектора  $\mathbf{p}$ , но остальные семь весов гораздо меньше. Это показывает, как DCT (или любое другое ортогональное преобразование) производит сжатие. Теперь можно просто записать эти восемь весов в сжатый файл, где они будут занимать меньше места, чем восемь компонентов исходного вектора  $\mathbf{p}$ . Квантование весов  $w_i$  может существенно повысить фактор сжатия, причем при весьма малой потере данных.

$\theta$	0.196	0.589	0.982	1.374	1.767	2.160	2.553	2.945
$\cos 0\theta$	1.	1.	1.	1.	1.	1.	1.	1.
$\cos 1\theta$	0.981	0.831	0.556	0.195	-0.195	-0.556	-0.831	-0.981
$\cos 2\theta$	0.924	0.383	-0.383	-0.924	-0.924	-0.383	0.383	0.924
$\cos 3\theta$	0.831	-0.195	-0.981	-0.556	0.556	0.981	0.195	-0.831
$\cos 4\theta$	0.707	-0.707	-0.707	0.707	0.707	-0.707	-0.707	0.707
$\cos 5\theta$	0.556	-0.981	0.195	0.831	-0.831	-0.195	0.981	-0.556
$\cos 6\theta$	0.383	-0.924	0.924	-0.383	-0.383	0.924	-0.924	0.383
$\cos 7\theta$	0.195	-0.556	0.831	-0.981	0.981	-0.831	0.556	-0.195

```
Table[N[t],{t,Pi/16,15Pi/16,Pi/8}]
dctp[pw_]:=Table[N[Cos[pw t]],{t,Pi/16,15Pi/16,Pi/8}]
dctp[0]
dctp[1]
...
dctp[7]
```

Табл. 3.21. Вычисление одномерного DCT.

Рис. 3.22 иллюстрирует эту линейную комбинацию графически. Все восемь векторов  $\mathbf{v}_i$  показаны в виде ряда из восьми маленьких серых квадратиков, причем значение  $+1$  представлено белым цветом, а значение  $-1$  окрашено в черный цвет. Каждый из восьми компонентов вектора  $\mathbf{p}$  выражен в виде взвешенной суммы восьми серых оттенков.

На практике одномерное DCT проще всего вычислять по формуле

$$G_f = \frac{1}{2} C_f \sum_{t=0}^7 p_t \cos \left( \frac{(2t+1)f\pi}{16} \right), \quad (3.7)$$

$$\text{где } C_f = \begin{cases} \frac{1}{\sqrt{2}}, & f = 0, \\ 1, & f > 0, \end{cases} \quad \text{при } f = 0, 1, \dots, 7.$$

Здесь исходными данными (пикселями, фрагментами звука или другими элементами) являются величины  $p_t$ , а им соответствующими

коэффициентами DCT служат числа  $G_f$ . Формула (3.7) очень проста, но процесс вычисления по ней медленный (в § 3.7.3 обсуждается ускоренный метод вычисления DCT). Декодер получает на входе коэффициенты DCT, делит их на восьмерки и применяет к ним *обратное* преобразование DCT (inverse DCT, IDCT) для восстановления исходных данных (тоже в виде групп по 8 элементов). Простейшая формула для вычисления IDCT имеет вид

$$p_t = \frac{1}{2} \sum_{j=0}^7 C_j G_j \cos \left( \frac{(2t+1)j\pi}{16} \right), \text{ при } t = 0, 1, \dots, 7. \quad (3.8)$$

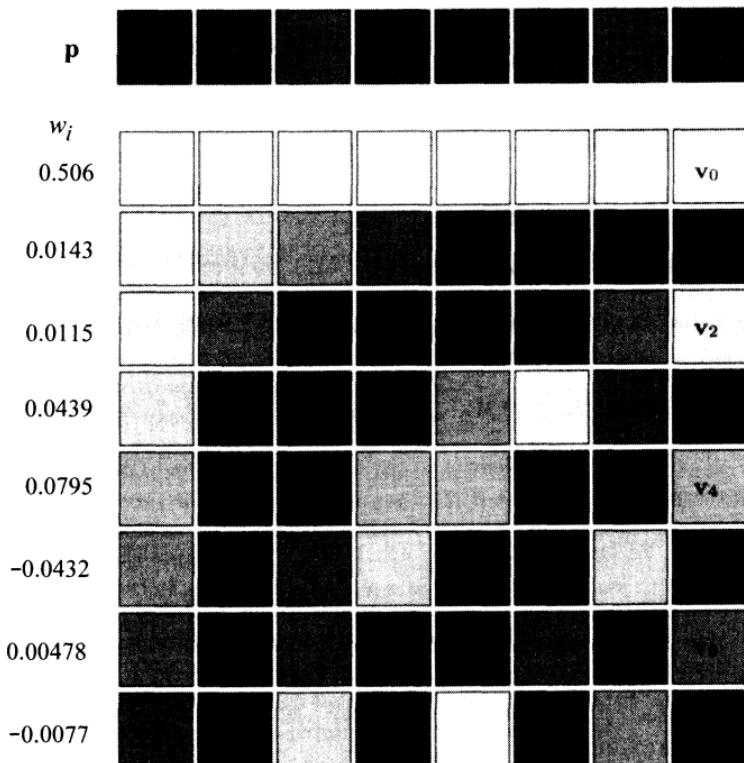


Рис. 3.22. Графическое представление одномерного DCT.

Следующий пример демонстрирует достоинства метода DCT. Рассмотрим множество, состоящее из 8 величин (исходных данных)  $\mathbf{p} = (12, 10, 8, 10, 12, 10, 8, 11)$ , применим к ним DCT и получим восемь коэффициентов

28.6375, 0.571202, 0.46194, 1.757, 3.18198, -1.72956, 0.191342, -0.308709.

Эти числа можно использовать для точного восстановления исходных данных (с маленькой ошибкой, вызванной ограничением на точность компьютерных вычислений). Наша цель, однако, улучшить сжатие с помощью подходящего квантования коэффициентов. Округляем (квантуем) их до 28.6, 0.6, 0.5, 1.8, 3.2, -1.8, 0.2, -0.3, применяем IDTC и получаем

12.0254, 10.0233, 7.96054, 9.93097, 12.0164, 9.99321, 7.94354, 10.9989.

Еще раз квантуем коэффициенты: 28, 1, 1, 2, 3, -2, 0, 0 и опять получаем с помощью IDCT следующий результат:

12.1883, 10.2315, 7.74931, 9.20863, 11.7876, 9.54549, 7.82865, 10.6557.

Наконец, квантуем коэффициенты до 28, 0, 0, 2, 3, -2, 0, 0 и получаем с помощью IDCT последовательность

11.236, 9.62443, 7.66286, 9.57302, 12.3471, 10.0146, 8.05304, 10.6842,

в которой наибольшая разность между исходным значением (12) и реконструированным (11.236) равна 0.764 (или 6.4% от 12). Программа вычисления для системы Mathematica приведена на рис. 3.23.

```
p={12.,10.,8.,10.,12.,10.,8.,11.};
c={.7071,1,1,1,1,1,1,1};
dct[i_]:=(c[[i+1]]/2)Sum[p[[t+1]]Cos[(2t+1)i Pi/16],{t,0,7}];
q=Table[dct[i],{i,0,7}] (* use precise DCT coefficients *)
q={28,0,0,2,3,-2,0,0}; (* or use quantized DCT coefficients *)
idct[t_]:=(1/2)Sum[c[[j+1]]q[[j+1]]Cos[(2t+1)j Pi/16],{j,0,7}];
ip=Table[idct[t],{t,0,7}]
```

Рис. 3.23. Эксперименты с одномерным DCT.

Эти простые примеры показывают достоинства метода DCT. Множество 28, 0, 0, 2, 3, -2, 0, 0 грубо квантованных коэффициентов DCT обладает четырьмя свойствами, которые делают его идеальным для сжатия, причем с замечательной декомпрессией при малой потери данных. Вот эти четыре свойства: (1) множество состоит только из целых чисел, (2) только четыре из них не равны нулю, (3) нулевые коэффициенты образуют серии, (4) среди ненулевых коэффициентов только первый имеет большую величину; остальные меньше исходных чисел. Эти свойства можно использовать при реализации схемы RLE, метода Хаффмана или любой другой техники (см. § 3.7.4 и 3.7.5) для дальнейшего сжатия этого множества.

**Пример:** Одномерное DCT (уравнение (3.7)) восьми коррелированных величин 11, 22, 33, 44, 55, 66, 77 и 88 произведет восемь коэффициентов 140, -71, 0, -7, 0, -2, 0, 0. После квантования получаем множество 140, -71, 0, 0, 0, 0, 0, 0 и применяем IDCT. В результате: 15, 20, 30, 43, 56, 69, 79 и 84. Эти числа весьма близки к исходным; наибольшее расхождение равно 4. На рис. 3.24 дана программа для этого примера.

```
Clear[Pixel, G, Gq, RecP];
Cr[i_]:=If[i==0, Sqrt[2]/2, 1];
DCT[x_]:={(1/2)Cr[i]Sum[Pixel[[x+1]]Cos[(2x+1)i Pi/16], {x,0,7,1}]};
IDCT[x_]:={(1/2)Sum[Cr[i]Gq[[i+1]]Cos[(2x+1)i Pi/16], {i,0,7,1}]};
Pixel={11,22,33,44,55,66,77,88};
G=Table[SetAccuracy[N[DCT[m]],0], {m,0,7}]
Gq={140.,-71.,0,0,0,0,0,0};
RecP=Table[SetAccuracy[N[IDCT[m]],0], {m,0,7}]
```

Рис. 3.24. Пример одномерного DCT (Mathematica).

Дойдя до этого места, воодушевленный читатель может воскликнуть: «Удивительно! Восемь исходных данных восстанавливаются всего с помощью двух чисел. Чудеса какие-то!» Однако, те, кто поняли свойства преобразований, могут дать простое объяснение. Восстановление данные происходит не только по двум числам 140 и -71, но также по их положению в последовательности из 8 коэффициентов. Кроме того, исходные величины восстанавливаются с высокой точностью благодаря присутствию избыточности.

Предельным случаем избыточных данных служит последовательность одинаковых величин. Они, конечно, имеют совершенную корреляцию, и мы чувствуем интуитивно, что одного числа будет достаточно для полного их восстановления. Реконструкция последовательности высоко коррелированных данных, такой, как 20, 31, 42, 53, ... потребует всего двух чисел. Ими могут быть начальное значение (20) и шаг (11) (разность этой арифметической прогрессии), но могут быть и другие числа. В общем случае, чем меньше коррелированы данные, тем больше чисел потребуется для их восстановления.

**Двумерное (матричное) DCT:** Из опыта хорошо известно, что пиксели изображения имеют корреляцию по двум направлениям, а не только по одному (пиксели коррелируют со своими соседями слева, справа, а также сверху и снизу). Поэтому методы сжатия изображений используют двумерное DCT, которое задается формулой

$$G_{ij} = \frac{1}{\sqrt{2n}} C_i C_j \sum_{x=0}^{n-1} \sum_{y=0}^{n-1} p_{xy} \cos\left(\frac{(2y+1)j\pi}{2n}\right) \cos\left(\frac{(2x+1)i\pi}{2n}\right), \quad (3.9)$$

при  $0 \leq i, j \leq n - 1$ . Изображение разбивается на блоки пикселов  $p_{xy}$  размера  $n \times n$  (в нашем примере  $n = 8$ ), и уравнения (3.9) используются для нахождения коэффициентов  $G_{ij}$  для каждого блока пикселов. Если допускается частичная потеря информации, то коэффициенты квантуются. Декодер восстанавливает сжатый блок данных (точно или приближенно), вычисляя обратное DCT (IDCT) по формуле

$$p_{xy} = \frac{1}{4} \sum_{i=0}^7 \sum_{j=0}^7 C_i C_j G_{ij} \cos\left(\frac{(2y+1)j\pi}{16}\right) \cos\left(\frac{(2x+1)i\pi}{16}\right), \quad (3.10)$$

$$\text{где } C_f = \begin{cases} \frac{1}{\sqrt{2}}, & f = 0, \\ 1, & f > 0. \end{cases}$$

Двумерное DCT можно интерпретировать двумя способами: с помощью вращения (на самом деле, композиции двух вращений), и с помощью базиса в  $n$ -мерном векторном пространстве. В первой интерпретации используется блок  $n \times n$  пикселов (рис. 3.25a, где элементы обозначены буквой «L»). Сначала рассматриваются строки этого блока как точки  $(p_{x,0}, p_{x,1}, \dots, p_{x,n-1})$  в  $n$ -мерном пространстве, которые поворачиваются в этом пространстве с помощью преобразования, задаваемого внутренней суммой

$$G1_{x,j} = C_j \sum_{y=0}^{n-1} p_{xy} \cos\left(\frac{(2y+1)j\pi}{2n}\right)$$

из уравнения (3.9). Результатом этого вращения служит блок  $G1_{x,j}$  из  $n \times n$  коэффициентов, в котором в строках доминируют первые элементы (обозначенные как «L» на рис. 3.25b), а все остальные элементы малы (они обозначены на этом рисунке как «S»). Внешняя сумма из уравнения (3.9) равна

$$G_{ij} = \frac{1}{\sqrt{2n}} C_i \sum_{x=0}^{n-1} p_{xy} G1_{x,j} \cos\left(\frac{(2x+1)i\pi}{2n}\right).$$



Здесь уже *столбцы* матрицы  $G1_{x,j}$  рассматриваются в качестве точек  $n$ -мерного векторного пространства, над которыми совершается преобразование вращения. В результате получается один большой коэффициент в верхнем левом углу блока (на рис.3.25с – это «L») и  $n^2 - 1$  маленьких коэффициентов в остальных местах («S» и «s» на рисунке). Эта интерпретация рассматривает двумерное DTC в виде двух разных вращений размерности  $n$ . Интересно отметить, что два вращения размерности  $n$  вычисляются быстрее, чем одно вращение размерности  $n^2$ , поскольку второе требует матрицу размера  $n^2 \times n^2$ .

L L L L L L L L	L S S S S S S S	L S S S S S S S
L L L L L L L L	L S S S S S S S	S s s s s s s s
L L L L L L L L	L S S S S S S S	S s s s s s s s
L L L L L L L L	L S S S S S S S	S s s s s s s s
L L L L L L L L	L S S S S S S S	S s s s s s s s
L L L L L L L L	L S S S S S S S	S s s s s s s s
L L L L L L L L	L S S S S S S S	S s s s s s s s
L L L L L L L L	L S S S S S S S	S s s s s s s s
L L L L L L L L	L S S S S S S S	S s s s s s s s
L L L L L L L L	L S S S S S S S	S s s s s s s s
L L L L L L L L	L S S S S S S S	S s s s s s s s
L L L L L L L L	L S S S S S S S	S s s s s s s s
L L L L L L L L	L S S S S S S S	S s s s s s s s
L L L L L L L L	L S S S S S S S	S s s s s s s s
L L L L L L L L	L S S S S S S S	S s s s s s s s
L L L L L L L L	L S S S S S S S	S s s s s s s s

(a)

(b)

(c)

Рис. 3.25. Двумерное DCT и двойное вращение.

Вторая интерпретация (при  $n = 8$ ) использует уравнение (3.9) для создания 64 блоков по  $8 \times 8$  величин в каждом. Все 64 блока рассматриваются в качестве базиса 64-мерного векторного пространства (это базисные изображения). Любой блок  $B$  из  $8 \times 8$  пикселов можно выразить как линейную комбинацию этих базисных изображений, и все 64 веса этой линейной комбинации образуют коэффициенты DCT блока  $B$ .

На рис. 3.26 показано графическое представление 64 базисных образов двумерного DCT при  $n = 8$ . Каждый элемент  $(i, j)$  на этом рисунке является блоком размера  $8 \times 8$ , который получен произведением  $\cos(i \cdot s) \cos(j \cdot t)$ , где  $s$  и  $t$  – меняются независимо в пределах, указанных в уравнении (3.6). Этот рисунок легко построить с помощью программы, приведенной на стр. 159. Там же приведена модифицированная программа из [Watson 94], требующая пакет GraphicsImage.m, который не очень распространен.

Используя подходящее программное обеспечение, легко выполнить вычисление DCT и отобразить результаты графически. На рис. 3.29а приведена случайная матрица  $8 \times 8$  из нулей и единиц. Эта матрица изображена на рис. 3.29б с помощью белых и черных квадратиков, обозначающих 1 и 0, соответственно. На рис. 3.29с

показаны численные значения весов, на которые следует умножить каждый из 64 коэффициентов DCT для того, чтобы восстановить исходную матрицу.

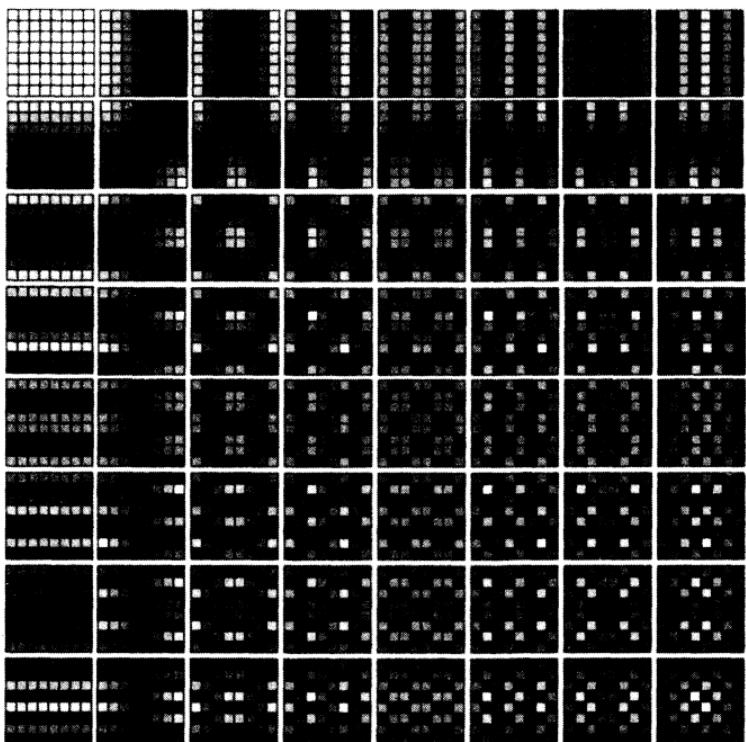


Рис. 3.26. 64 базисных изображения двумерного DCT.

На этом рисунке нуль показан нейтрально-серым цветом, положительные числа – светло-серым, а отрицательные – темным. На рис. 3.29d даны численные значения этих весов. Приведена также программа для построения всех этих графиков. На рис. 3.30 сделаны те же самые построения, но применительно к более регулярным исходным данным.

Теперь продемонстрируем достоинства двумерного DCT применительно к двум блокам чисел. Первый блок (табл. 3.27, слева) состоит из сильно коррелированных целых чисел в интервале [8,12], а второй (табл. 3.28, слева) образован случайными числами из того же интервала. Первый блок порождает один большой коэффициент DC, за которым следуют маленькие (включая 20 нулевых) коэффициен-

тов АС. А среди коэффициентов DCT второго, случайного, блока имеется всего один нуль.

```
dctp[fs_,ft_]:=Table[SetAccuracy[N[(1.-Cos[fs s]Cos[ft t])/2],3],{s,Pi/16,15Pi/16,Pi/8},{t,Pi/16,15Pi/16,Pi/8}]]//TableForm
dctp[0,0]
...
dctp[7,7]
```

Программа для рис. 3.26 (Mathematica)

```
Needs[»GraphicsImage«] (* Draws 2D DCT Coefficients *)
DCTMatrix=Table[If[k==0,Sqrt[1/8],Sqrt[1/4]Cos[Pi(2j+1)k/16]],{k,0,7},{j,0,7}]/.N;
DCTTensor=Array[Outer[Times,DCTMatrix[[#1]],DCTMatrix[[#2]]]&,{8,8}];
Show[GraphicsArray[Map[GraphicsImage[#, {-.25,.25}]&,DCTTensor,{2}]]]
```

Альтернативная программа для рис. 3.26

(Видно, почему пиксели в табл. 3.27 коррелированы. Все восемь чисел верхней строки таблицы близки друг к другу (расстояния между ними равны 2 или 3). Все остальные строки получаются циклическим сдвигом вправо предыдущей строки.)

12 10 8 10 12 10 8 11	81 0 0 0 0 0 0 0
11 12 10 8 10 12 10 8	0 1.57 0.61 1.90 0.38 -1.81 0.20 -0.32
8 11 12 10 8 10 12 10	0 -0.61 0.71 0.35 0 0.07 0 0.02
10 8 11 12 10 8 10 12	0 1.90 -0.35 4.76 0.77 -3.39 0.25 -0.54
12 10 8 11 12 10 8 10	0 -0.38 0 -0.77 8.00 0.51 0 0.07
10 12 10 8 11 12 10 8	0 -1.81 -0.07 -3.39 -0.51 1.57 0.56 0.25
8 10 12 10 8 11 12 10	0 -0.20 0 -0.25 0 -0.56 -0.71 0.29
10 8 10 12 10 8 11 12	0 -0.32 -0.02 -0.54 -0.07 0.25 -0.29 -0.90

Табл. 3.27. Двумерное DCT блока коррелированных величин.

8 10 9 11 11 9 9 12	79.12 0.98 0.64 -1.51 -0.62 -0.86 1.22 0.32
11 8 12 8 11 10 11 10	0.15 -1.64 -0.09 1.23 0.10 3.29 1.08 -2.97
9 11 9 10 12 9 9 8	-1.26 -0.29 -3.27 1.69 -0.51 1.13 1.52 1.33
9 12 10 8 8 9 8 9	-1.27 -0.25 -0.67 -0.15 1.63 -1.94 0.47 -1.30
12 8 9 9 12 10 8 11	-2.12 -0.67 -0.07 -0.79 0.13 -1.40 0.16 -0.15
8 11 10 12 9 12 12 10	-2.68 1.08 -1.99 -1.93 -1.77 -0.35 0 -0.80
10 10 12 10 12 10 10 12	1.20 2.10 -0.98 0.87 -1.55 -0.59 -0.98 2.76
12 9 11 11 9 8 8 12	-2.24 0.55 0.29 0.75 -2.40 -0.05 0.06 1.14

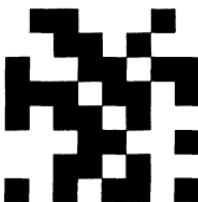
Табл. 3.28. Двумерное DCT блока случайных величин.

```

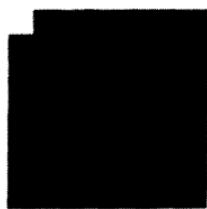
1 0 0 1 1 1 0 1
1 1 0 0 1 0 1 1
0 1 1 0 0 1 0 0
0 0 0 1 0 0 1 0
0 1 0 0 1 0 1 1
1 1 1 0 0 1 1 0
1 1 0 0 1 0 1 1
0 1 0 1 0 0 1 0

```

(a)



(b)



(c)

```

4.000 -0.133 0.637 0.272 -0.250 -0.181 -1.076 0.026
0.081 -0.178 -0.300 0.230 0.694 -0.309 0.875 -0.127
0.462 0.125 0.095 0.291 0.868 -0.070 0.021 -0.280
0.837 -0.194 0.455 0.583 0.588 -0.281 0.448 0.383
-0.500 -0.635 -0.749 -0.346 0.750 0.557 -0.502 -0.540
-0.167 0 -0.366 0.146 0.393 0.448 0.577 -0.268
-0.191 0.648 -0.729 -0.008 -1.171 0.306 1.155 -0.744
0.122 -0.200 0.038 -0.118 0.138 -1.154 0.134 0.148

```

(d)

```

DCTMatrix=Table[If[k==0,Sqrt[1/8],Sqrt[1/4]Cos[Pi(2j+1)k/16]],{k,0,7},{j,0,7}]/N;
DCTTensor=Array[Outer[Times,DCTMatrix[[#1]],DCTMatrix[[#2]]]&,{8,8}];
img={{1,0,0,1,1,1,0,1},{1,1,0,0,1,0,1,1},{0,1,1,0,0,1,0,0},{0,0,0,1,0,0,1,0},{0,1,0,0,1,0,1,1},{1,1,1,0,0,1,1,0},{1,1,0,0,1,0,1,1},{0,1,0,1,0,0,1,0}};
ShowImage[Reverse[img]];
dctcoeff=Array[(Plus@@Flatten[DCTTensor[[#1,#2]]img])&,{8,8}];
dctcoeff=SetAccuracy[dctcoeff,4]; dctcoeff=Chop[dctcoeff,.001];
MatrixForm[dctcoeff];
ShowImage[Reverse[dctcoeff]]

```

Рис. 3.29. Пример двумерного DCT (Mathematica).

Сжатие любого изображения с помощью DCT можно теперь сделать следующим образом.

1. Разделить его на  $k$  блоков пикселов размера  $n \times n$  (обычно  $8 \times 8$ ).
2. Применить DCT к каждому блоку  $B_i$ , то есть, представить каждый блок в виде линейной комбинации 64 базисных блоков рис. 3.26. Результатом станут блоки (мы будем их называть векторами)  $W^{(i)}$  из 64 весов  $w_j^i$ , где  $j = 0, 1, \dots, 63$ .
3. Все  $k$  векторов  $W^{(i)}$  ( $i = 1, 2, \dots, k$ ) разделить на 64 вектора коэффициентов  $C^{(j)}$  с  $k$  компонентами  $(w_j^{(1)}, w_j^{(2)}, \dots, w_j^{(k)})$ . Вектор первых компонентов  $C^{(0)}$  состоит из  $k$  коэффициентов DC.
4. Сделать квантование каждого вектора коэффициентов  $C^{(j)}$  неза-

висимо от других. Полученный квантованный вектор  $Q^{(j)}$  записать (после дополнительной компрессии по методу RLE, Хаффмана или иного метода) в сжатый файл.

0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1		
(a)	(b)	(c)
4.000 -0.721 0 -0.850 0 -1.273 0 -3.625 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		(d)

```

DCTMatrix=Table[If[k==0,Sqrt[1/8],Sqrt[1/4]Cos[Pi(2j+1)k/16]],
{k,0,7}, {j,0,7}] //N;
DCTTensor=Array[Outer[Times, DCTMatrix[[#1]],DCTMatrix[[#2]]]&,
{8,8}];
img={{0,1,0,1,0,1,0,1},{0,1,0,1,0,1,0,1},
{0,1,0,1,0,1,0,1},{0,1,0,1,0,1,0,1},{0,1,0,1,0,1,0,1},
{0,1,0,1,0,1,0,1},{0,1,0,1,0,1,0,1},{0,1,0,1,0,1,0,1}};
ShowImage[Reverse[img]]
dctcoeff=Array[(Plus @@ Flatten[DCTTensor[[#1,#2]] img])&,{8,8}];
dctcoeff=SetAccuracy[dctcoeff,4];
dctcoeff=Chop[dctcoeff,.001];
MatrixForm[dctcoeff]
ShowImage[Reverse[dctcoeff]]

```

Рис. 3.30. Пример двумерного DCT (Mathematica).

Декодер читает 64 квантованных вектора коэффициентов  $Q^{(j)}$ , использует их для построения  $k$  весовых векторов  $W^{(i)}$  и применяет IDCT к каждому весовому вектору для (приближенной) реконструкции 64 пикселов блока  $B_i$ . Отметим, что метод JPEG работает несколько иначе.

### 3.5.4. Пример

Этот пример демонстрирует разницу в производительности метода DCT при сжатии непрерывно тонового изображения и дискретно-тонового изображения. Мы исходим из сильно коррелированного образца, приведенного в табл. 3.31. Это будет идеализированная модель непрерывно тонового изображения, поскольку соседние пиксели отличаются на постоянную величину. Все 64 коэффициента DCT приведены в табл. 3.32. Видно, что имеется всего несколько доминирующих коэффициентов. В табл. 3.33 дан результат некоторого грубого квантования нашего образца. В этой таблице имеется всего четыре ненулевых коэффициента. Результат применения IDCT к этим коэффициентам представлен в табл. 3.34. Очевидно, что эти четыре коэффициента позволяют восстановить образец с высокой степенью точности.

В табл. 3.35–3.38 повторен тот же процесс применительно к Y-образному блоку данных, типичному для дискретно-тонового изображения. При этом использовалось достаточно легкое квантование (табл. 3.37). Оно состояло в округлении коэффициентов до ближайшего целого числа. Видно, что результат реконструкции этого образа (табл. 3.38) не столь хорош, как в предыдущем случае. Величины, которые были равны 10 стали лежать в интервале от 8.96 до 10.11, а те, что были нулями выросли до 0.86.

### 3.5.5. Дискретное синус-преобразование

Читатель в этом месте может задать логичный вопрос: «Почему косинус, а не синус?» Можно ли аналогичным образом использовать функцию синус для построения дискретного синус-преобразования? Существует DST (descrete sine transform) или нет? В этом коротком параграфе мы обсудим отличия синуса от косинуса, которые приводят к весьма неэффективному синус-преобразованию.

Функция  $f(x)$ , удовлетворяющая условию  $f(-x) = -f(x)$ , называется нечетной. Аналогично, если  $f(-x) = f(x)$ , то  $f(x)$  называется четной. Для любой нечетной функции  $f(0) = f(-0) = -f(0)$ , поэтому  $f(0)$  должно равняться 0. Большинство функций не являются ни четными, ни нечетными. Но основные тригонометрические функции  $\sin(x)$  и  $\cos(x)$  являются, соответственно, четной и нечетной. Из рис.3.39 видно, что эти функции различаются лишь по своей фазе (то есть, косинус получается из синуса сдвигом на  $\pi/2$ ), однако, этой разности достаточно для смены их четности. Когда (нечетная) функция синус сдвигается, она становится (четной) функцией косинус, которая имеет ту же форму.

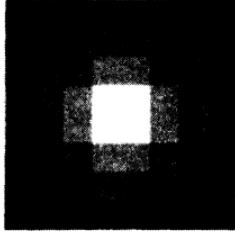
00	10	20	30	30	20	10	00	
10	20	30	40	40	30	20	10	
20	30	40	50	50	40	30	20	
30	40	50	60	60	50	40	30	
30	40	50	60	60	50	40	30	
20	30	40	50	50	40	30	20	
10	20	30	40	40	30	12	10	
00	10	20	30	30	20	10	00	

Табл. 3.31. Образец с высокой корреляцией.

239	1.19	-89.76	-0.28	1.00	-1.39	-5.03	-0.79
1.18	-1.39	0.64	0.32	-1.18	1.63	-1.54	0.92
-89.76	0.64	-0.29	-0.15	0.54	-0.75	0.71	-0.43
-0.28	0.32	-0.15	-0.08	0.28	-0.38	0.36	-0.22
1.00	-1.18	0.54	0.28	-1.00	1.39	-1.31	0.79
-1.39	1.63	-0.75	-0.38	1.39	-1.92	1.81	-1.09
-5.03	-1.54	0.71	0.36	-1.31	1.81	-1.71	1.03
-0.79	0.92	-0.43	-0.22	0.79	-1.09	1.03	-0.62

Табл. 3.32. DCT коэффициенты образца.

239	1	-90	0	0	0	0	0
0	0	0	0	0	0	0	0
-90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Табл 3.33: Грубое квантование с 4 ненулевыми коэффициентами.

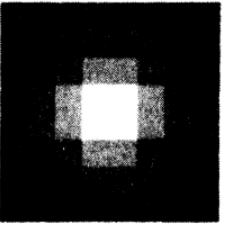
0.65	9.23	21.36	29.91	29.84	21.17	8.94	0.30	
9.26	17.85	29.97	38.52	38.45	29.78	17.55	8.91	
21.44	30.02	42.15	50.70	50.63	41.95	29.73	21.09	
30.05	38.63	50.76	59.31	59.24	50.56	38.34	29.70	
30.05	38.63	50.76	59.31	59.24	50.56	38.34	29.70	
21.44	30.02	42.15	50.70	50.63	41.95	29.73	21.09	
9.26	17.85	29.97	38.52	38.45	29.78	17.55	8.91	
0.65	9.23	21.36	29.91	29.84	21.17	8.94	0.30	

Табл. 3.34. Результат IDTC.

00	10	00	00	00	00	00	10
00	00	10	00	00	00	10	00
00	00	00	10	00	10	00	00
00	00	00	00	10	00	00	00
00	00	00	00	10	00	00	00
00	00	00	00	10	00	00	00
00	00	00	00	10	00	00	00
00	00	00	00	10	00	00	00

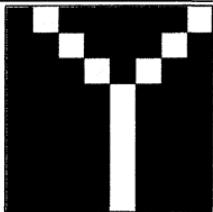


Табл. 3.35. Y-образный блок.

13.75	-3.11	-8.17	2.46	3.75	-6.86	-3.38	6.59
4.19	-0.29	6.86	-6.85	-7.13	4.48	1.69	-7.28
1.63	0.19	6.40	-4.81	-2.99	-1.11	-0.88	-0.94
-0.61	0.54	5.12	-2.31	1.30	-6.04	-2.78	3.05
-1.25	0.52	2.99	-0.20	3.75	-7.39	-2.59	1.16
-0.41	0.18	0.65	1.03	3.87	-5.19	-0.71	-4.76
0.68	-0.15	-0.88	1.28	2.59	-1.92	1.10	-9.05
0.83	-0.21	-0.99	0.82	1.13	-0.08	1.31	-7.21

Табл. 3.36. DCT коэффициенты блока.

13.75	-3	-8	2	3	-6	-3	6
4	-0	6	-6	-7	4	1	-7
1	0	6	-4	-2	-1	-0	-0
-0	0	5	-2	1	-6	-2	3
-1	0	2	-0	3	-7	-2	1
-0	0	0	1	3	-5	-0	-4
0	-0	-0	1	2	-1	1	-9
0	-0	-0	0	1	-0	1	-7

Табл. 3.37. Слабое квантование округлением до ближайшего целого.

-0.13	8.96	0.55	-0.27	0.27	0.86	0.15	9.22
0.32	0.22	9.10	0.40	0.84	-0.11	9.36	-0.14
0.00	0.62	-0.20	9.71	-1.30	8.57	0.28	-0.33
-0.58	0.44	0.78	0.71	10.11	1.14	0.44	-0.49
-0.39	0.67	0.07	0.38	8.82	0.09	0.28	0.41
0.34	0.11	0.26	0.18	8.93	0.41	0.47	0.37
0.09	-0.32	0.78	-0.20	9.78	0.05	-0.09	0.49
0.16	-0.83	0.09	0.12	9.15	-0.11	-0.08	0.01

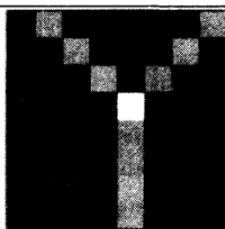


Табл. 3.38. Результат IDTC. Плохое качество.

Для того, чтобы понять разницу между DCT и DST, рассмотрим одномерный случай. Одномерное DCT, (см. уравнение (3.7)),

использует функцию  $\cos((2t + 1)f\pi/16)$  при  $f = 0, 1, \dots, 7$ . Для первого значения, равного  $f = 0$ , эта функция равна  $\cos(0) = 1$ . Этот член очень важен; он производит коэффициент DC, который соответствует среднему значению восьми преобразуемых величинам. По аналогии, DST основано на функции  $\sin((2t + 1)f\pi/16)$  которая равна  $\sin(0) = 0$  при  $f = 0$ , то есть, этот член не вносит никакого вклада в преобразование, то есть DST не имеет коэффициент DC.

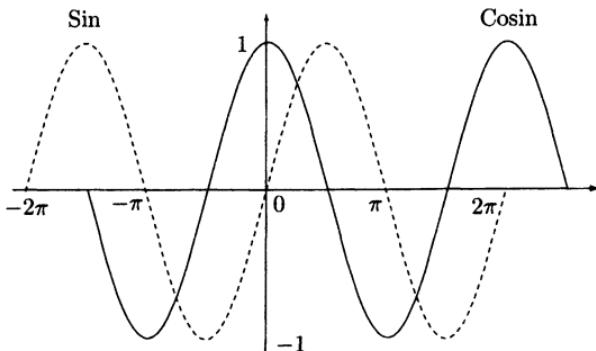


Рис. 3.39. Функции синус (нечетная) и косинус (четная).

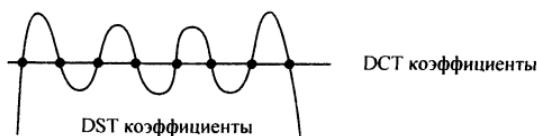
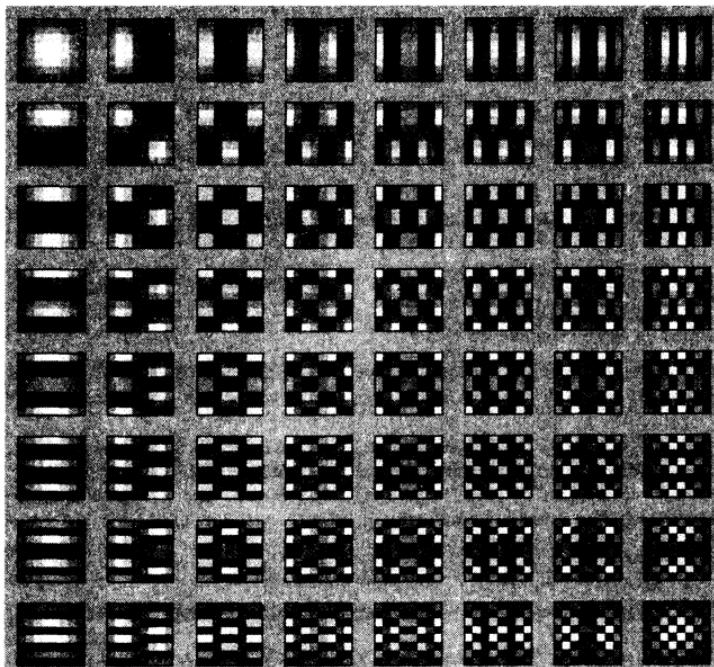


Рис. 3.40. DCT и DST данных из восьми тождественных значений.

Ущербность DST можно обнаружить, рассмотрев преобразование исходного образца из 8 одинаковых величин. Такие величины, безусловно, прекрасно коррелированы. Их графиком служит горизонтальная прямая. Применяя DCT, получаем один ненулевой DC, равный исходной величине. Преобразование IDCT также прекрасно восстановит данные (с незначительной потерей, обусловленной ограниченной точностью машинных вычислений). Если теперь применить DST к тем же данным, то в результате получится 7 ненулевых коэффициентов AC, сумма которых равна волнообразной функции, проходящей через все восемь исходных точек, но при этом осциллирует в промежутках между ними. Это поведение проиллю-

стрировано на рис. 3.40. Оно имеет три неприятных свойства. (1) Энергия исходных данных нигде не концентрируется. (2) Семь коэффициентов не являются декоррелированными (поскольку исходные данные полностью коррелированы). (3) Квантование семи коэффициентов может сильно уменьшить качество реконструированных данных после применения обратного DST.



```

N=8;
m=[1:N] '*ones(1,N); n=m';
% можно также использовать cos вместо sin
%A=sqrt(2/N)*cos(pi*(2*(n-1)+1).*(m-1)/(2*N));
A=sqrt(2/N)*sin(pi*(2*(n-1)+1).*(m-1)/(2*N));
A(1,:)=sqrt(1/N);
C=A';
for row=1:N
    for col=1:N
        B=C(:,row)*C(:,col).'; %тензорное произведение
        subplot(N,N,(row-1)*N+col)
        imagesc(B)
        drawnow
    end
end

```

Рис. 3.41. 64 базисных изображения двумерного DST.



**Пример:** Применим DST к последовательности из восьми одинаковых величин, равных 100. Получим последовательность коэффициентов (0, 256.3, 0, 90, 0, 60.1, 0, 51). С помощью этих коэффициентов обратное преобразование IDST может восстановить исходные данные, но видно, что коэффициенты АС ведут себя иначе, чем при использовании DCT. Они не становятся все меньше, и среди них нет серий из одних нулей. Применяя DST к восьми высоко коррелированным величинам (11, 22, 33, 44, 55, 66, 77, 88), получаем более плохое множество коэффициентов

$$0, 126.9, -57.5, 44.5, -31.1, 29.8, -23.8, 25.2.$$

Здесь совсем нет концентрации энергии.

Все эти аргументы и примеры вместе с тем фактом (обсуждаемым в [Ahmed et al.]) 74]), что DCT производит высоко декоррелированные коэффициенты, неоспоримо свидетельствуют в пользу метода DCT, для использования в алгоритмах сжатия данных.

На рис. 3.41 показаны 64 базисных изображения DST при  $n = 8$  и программа Matlab, порождающая их. (Ср. рис. 3.26.)

### 3.5.6. Преобразование Уолша–Адамара

Выше уже упоминалось (см. стр.149), что это преобразование мало эффективно для сжатия данных. Но оно очень быстро, так как его можно вычислять, применяя только сложение, вычитание и, иногда, сдвиг вправо (что эквивалентно делению на 2 двоичного представления величин).

Для заданного блока  $N \times N$  пикселов  $P_{xy}$  (здесь  $N$  должно быть степенью двойки,  $N = 2^n$ ), его двумерное прямое и обратное преобразования Уолша–Адамара (они обозначаются WHT и IWHT, соответственно) определяются с помощью следующих уравнений:

$$\begin{aligned} H(u, v) &= \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} P_{xy} g(x, y, u, v) = \\ &= \frac{1}{N} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} P_{xy} (-1)^{\sum_{i=0}^n [b_i(x)p_i(u) + b_i(y)p_i(v)]}, \end{aligned} \quad (3.11)$$

$$\begin{aligned} P_{xy} &= \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} H(u, v) h(x, y, u, v) = \\ &= \frac{1}{N} \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} H(u, v) (-1)^{\sum_{i=0}^n [b_i(x)p_i(u) + b_i(y)p_i(v)]}, \end{aligned} \quad (3.12)$$

где  $H(u, v)$  – это результат преобразования (то есть, коэффициенты WHT), величина  $b_i(u)$  равна биту  $i$  в двоичном представлении целого числа  $u$ , а  $p_i(u)$  определяется с помощью  $b_j(u)$  из следующих рекуррентных соотношений:

$$\begin{aligned} p_0(u) &= b_{n-1}(u), \\ p_1(u) &= b_{n-1}(u) + b_{n-2}(u), \\ p_2(u) &= b_{n-2}(u) + b_{n-3}(u), \\ &\dots \\ p_{n-1}(u) &= b_1(u) + b_0(u). \end{aligned} \quad (3.13)$$

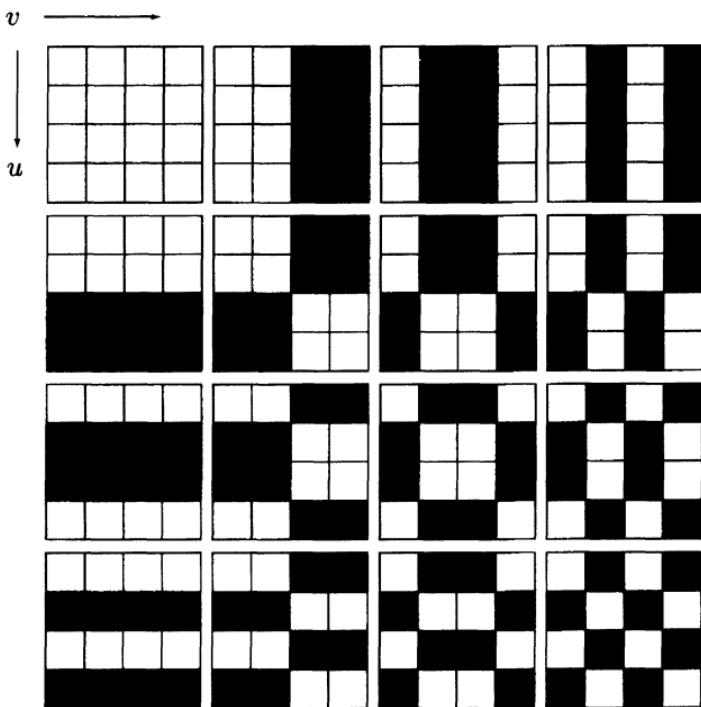
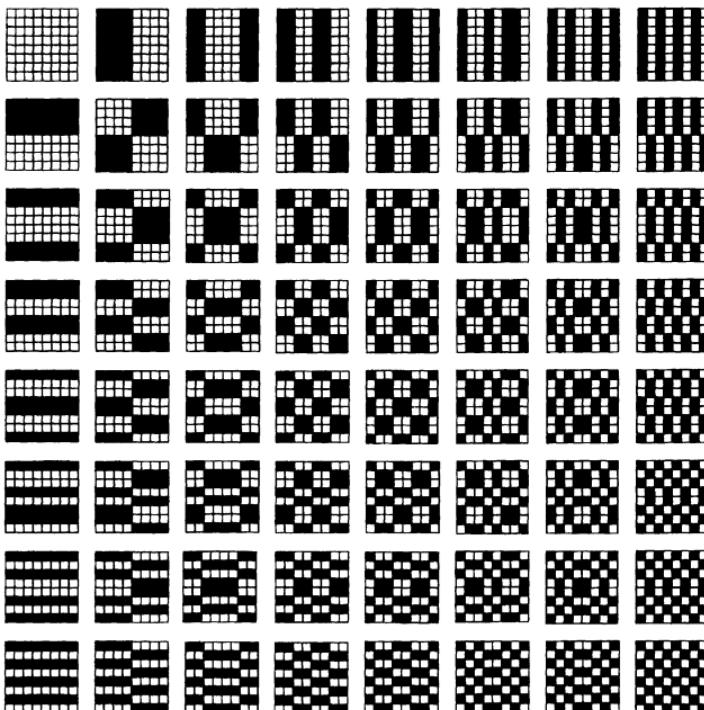


Рис. 3.42. Упорядоченные ядра WHT при  $N = 4$ .

(Напомним, что  $n$  определяется из соотношения  $N = 2^n$ ). Рассмотрим, например,  $u = 6 = 110_2$ . Нулевой, первый и второй биты равны соответственно, 0, 1, 1, поэтому  $b_0(6) = 0, b_1(6) = 1$  и  $b_2(6) = 1$ . Величины  $g(x, y, u, v)$  и  $h(x, y, u, v)$  называются **ядрами** (или **базисными изображениями**) WHT. Их матрицы совпадают. Элементами матриц служат числа +1 и -1, которые умножаются



на  $1/N$ . В результате, преобразование WHT состоит из умножения пикселов на  $+1$  или  $-1$ , сложения и деления суммы на  $N$ . Но поскольку  $N = 2^n$ , деление можно делать, сдвигая разряды чисел вправо на  $n$  позиций.



```

M=3; N=2^M; H=[1 1; 1 -1]/sqrt(2);
for m=1:(M-1) % рекурсия
    H=[H H; H -H]/sqrt(2);
end; A=H';
map=[1 5 7 3 4 8 6 2]; % 1:N
for n=1:N, B(:,n)=A(:,map(n)); end; A=B;
sc=1/(max(abs(A(:))).^2); % скалярный множитель
for row=1:N
    for col=1:N
        BI=A(:,row)*A(:,col).'; % тензорное произведение
        subplot(N,N,(row-1)*N+col)
        oe=round(BI*sc); % получаются значения -1, +1
        imagesc(oe), colormap([1 1 1; .5 .5 .5; 0 0 0])
        drawnow
    end
end

```

Рис. 3.43. Базисные изображения  $8 \times 8$  для WHT и программа Matlab.



Ядра WHT изображены в графической форме на рис. 3.42 при  $N = 4$ , где белый цвет означает  $+1$ , а черный  $-1$  (для наглядности множитель  $1/N$  опущен). Строки и столбцы в блоках занумерованы значениями  $u$  и  $v$  от 0 до 3, соответственно. Число смен знаков в строке или в столбце матрицы называется *частотностью* строки или столбца. Строки и столбцы на этом рисунке упорядочены по возрастанию частотности. Некоторые авторы предпочитают изображать ядра неупорядоченно, так, как это было определено Уолшем и Адамаром (см. [Gonzalez 92]).

Сжатие изображений с помощью WHT делается так же, как и для DCT с заменой уравнений (3.9) и (3.10) на (3.11) и (3.12).

**Пример:** На рис. 3.43 изображены 64 базисных изображения WHT и программа для их вычисления и построения на графике. Рассмотрен случай  $N = 8$ . Каждое базисное изображение – это таблица пикселов размера  $8 \times 8$ .

### 3.5.7. Преобразование Хаара

Преобразование Хаара [Stollnitz 92] используется на практике для отображения поддиапазона частот. Оно будет обсуждаться в главе 4. Однако, в силу простоты этого отображения, его также можно объяснить в терминах базисных изображений. Поэтому мы включили его рассмотрение и в этот параграф. Преобразование Хаара основывается на функциях Хаара  $h_k(x)$ , которые задаются при  $x \in [0, 1]$  и для  $k = 0, 1, \dots, N - 1$ , где  $N = 2^n$ .

Прежде, чем задать это преобразование, напомним, что любое целое число  $k$  можно представить в виде суммы  $k = 2^p + q - 1$ , где  $0 \leq p \leq n - 1$ ,  $q = 0$  или  $1$  при  $p = 0$ , и  $1 \leq q \leq 2^p$  при  $p \neq 0$ . Для  $N = 4 = 2^2$ , например, имеем следующие представления:  $0 = 2^0 + 0 - 1$ ,  $1 = 2^0 + 1 - 1$ ,  $2 = 2^1 + 1 - 1$  и  $3 = 2^1 + 2 - 1$ .

Базисные функции Хаара задаются формулами

$$h_0(x) \stackrel{\text{def}}{=} h_{00}(x) = \frac{1}{\sqrt{N}} \text{ при } 0 \leq x \leq 1, \quad (3.14)$$

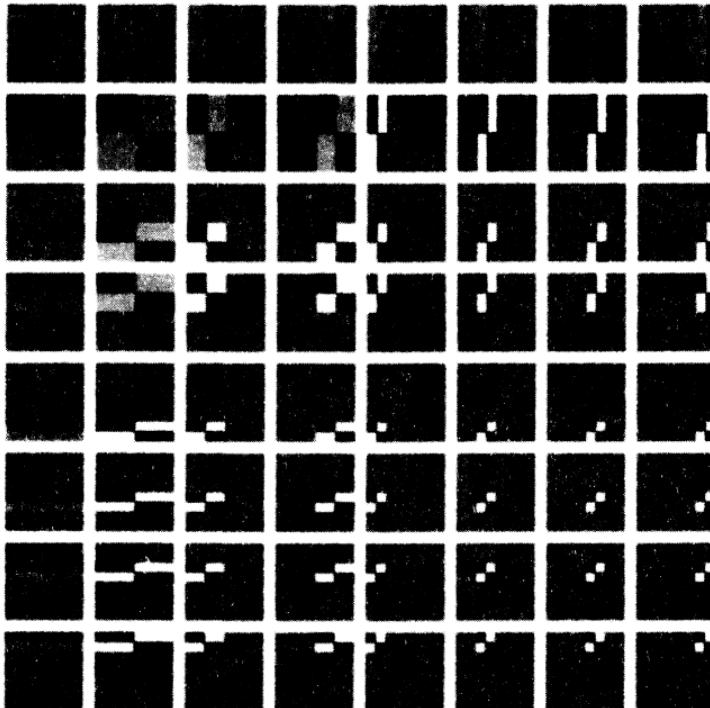
и

$$h_k(x) \stackrel{\text{def}}{=} h_{pq}(x) = \frac{1}{\sqrt{N}} \begin{cases} 2^{p/2}, & \frac{q-1}{2^p} \leq x \leq \frac{q-1/2}{2^p}, \\ -2^{p/2}, & \frac{q-1/2}{2^p} \leq x \leq \frac{q}{2^p}, \\ 0 & \text{для остальных } x \in [0, 1]. \end{cases} \quad (3.15)$$

Теперь можно построить  $N \times N$ -матрицу  $\mathbf{A}_N$  преобразования Хаара. Элемент с индексами  $i, j$  этой матрицы равен  $h_i(j)$ , где  $i =$

$= 0, 1, \dots, N - 1$  и  $j = 0/N, 1/N, \dots, (N - 1)/N$ . Например,

$$\mathbf{A}_2 = \begin{pmatrix} h_0(0/2) & h_0(0/2) \\ h_1(0/2) & h_1(1/2) \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad (3.16)$$



```
Needs[«GraphicsImage`"]
n=8;
h[k_,x_]:=Module[{p,q}, If[k==0, 1/Sqrt[n], (* h_0(x) *)
  p=0; While[2^p<=k, p++]; p--; q=k-2^p+1; (* if k>0, calc.p,q *)
  If[(q-1)/(2^p)<=x && x<(q-.5)/(2^p), 2^(p/2),
    If[(q-.5)/(2^p)<=x && x<q/(2^p), -2^(p/2), 0]]]];
HaarMatrix=Table[h[k,x], {k,0,7}, {x,0,7/n,1/n}] //N;
HaarTensor=Array[Outer[Times, HaarMatrix[[#1]],HaarMatrix[[#2]]]&,
{ n,n }];
Show[GraphicsArray[Map[GraphicsImage[#, {-2,2}]&, HaarTensor,{2}]]]
```

Рис. 3.44. Базисные изображения преобразования Хаара при  $N = 8$ .

(напомним, что  $p = 0$  и  $q = 1$  при  $i = 1$ ). На рис. 3.44 дана программа для вычисления этой матрицы для любого  $N$ , а также построены

базисные изображения при  $N = 8$  : Выпишем матрицы  $\mathbf{A}_4$  и  $\mathbf{A}_8$ .

$$\mathbf{A}_4 = \begin{pmatrix} h_0\left(\frac{0}{4}\right) & h_0\left(\frac{1}{4}\right) & h_0\left(\frac{2}{4}\right) & h_0\left(\frac{3}{4}\right) \\ h_1\left(\frac{0}{4}\right) & h_1\left(\frac{1}{4}\right) & h_1\left(\frac{2}{4}\right) & h_1\left(\frac{3}{4}\right) \\ h_2\left(\frac{0}{4}\right) & h_2\left(\frac{1}{4}\right) & h_2\left(\frac{2}{4}\right) & h_2\left(\frac{3}{4}\right) \\ h_3\left(\frac{0}{4}\right) & h_3\left(\frac{1}{4}\right) & h_3\left(\frac{2}{4}\right) & h_3\left(\frac{3}{4}\right) \end{pmatrix} = \frac{1}{\sqrt{4}} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ \sqrt{2} & -\sqrt{2} & 0 & 0 \\ 0 & 0 & \sqrt{2} & -\sqrt{2} \end{pmatrix},$$

$$\mathbf{A}_8 = \frac{1}{\sqrt{8}} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} \\ 2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & -2 \end{pmatrix}.$$

Для блока пикселов  $\mathbf{P}$  размера  $N \times N$ , где  $N = 2^n$ , его преобразование Хаара вычисляется по формуле  $\mathbf{A}_N \mathbf{P} \mathbf{A}_N$  (см. § 4.2.1).

### 3.5.8. Преобразование Кархунена–Лоэвэ

Преобразование Кархунена–Лоэвэ (его еще называют преобразованием Хотеллинга) имеет наилучшую эффективность в смысле концентрации энергии изображения, но по указанным выше причинам, оно имеет скорее теоретическое, нежели практическое значение. Данное изображение следует разделить на  $k$  блоков по  $n$  пикселов в каждом, обычно,  $n = 64$ , но допускаются и другие значения, а число  $k$  зависит от размера изображения. Рассматриваются векторы блоков, которые обозначаются  $\mathbf{b}^{(i)}$ , при  $i = 1, 2, \dots, k$ . Усредненный вектор равен  $\bar{\mathbf{b}} = (\sum_i \mathbf{b}^{(i)}) / k$ . Вводится новое семейство векторов  $\mathbf{v}^{(i)} = \mathbf{b}^{(i)} - \bar{\mathbf{b}}$ , для которого усредненный вектор  $(\sum_i \mathbf{v}^{(i)}) / k$  равен нулю. Матрицу преобразования (KLT) размера  $n \times n$ , которую мы будем строить, обозначим через  $\mathbf{A}$ . Результатом преобразования вектора  $\mathbf{v}^{(i)}$  будет весовой вектор  $\mathbf{w}^{(i)} = \mathbf{A} \mathbf{v}^{(i)}$ . Усреднение вектора  $\mathbf{w}^{(i)}$  также равно нулю. Построим матрицу  $\mathbf{V}$ , столбцами которой будут служить векторы  $\mathbf{v}^{(i)}$ . Рассмотрим также матрицу  $\mathbf{W}$  со столбцами  $\mathbf{w}^{(i)}$ :

$$\mathbf{V} = \left( \mathbf{v}^{(1)}, \mathbf{v}^{(2)}, \dots, \mathbf{v}^{(k)} \right), \quad \mathbf{W} = \left( \mathbf{w}^{(1)}, \mathbf{w}^{(2)}, \dots, \mathbf{w}^{(k)} \right).$$

Матрицы  $\mathbf{V}$  и  $\mathbf{W}$  имеют  $n$  строк и  $k$  столбцов. Из определения векторов  $\mathbf{w}^{(i)}$  заключаем, что  $\mathbf{W} = \mathbf{A} \cdot \mathbf{V}$ .

Все  $n$  векторов коэффициентов  $\mathbf{c}^{(j)}$  преобразования Кархунена–Лоэвэ определяются равенствами

$$\mathbf{c}^{(j)} = \left( w_j^{(1)}, w_j^{(2)}, \dots, w_j^{(k)} \right), \quad j = 1, 2, \dots, n.$$

Таким образом, вектор  $\mathbf{c}^{(j)}$  состоит из  $j$ -ых элементов весовых векторов  $\mathbf{w}^{(i)}$  при  $i = 1, 2, \dots, k$ .

Рассмотрим матрицу-произведение  $\mathbf{W} \cdot \mathbf{W}^T$ . Элемент строки  $a$  и столбца  $b$  этой матрицы равен сумме произведений

$$\left( \mathbf{W} \cdot \mathbf{W}^T \right)_{ab} = \sum_{i=1}^k w_a^{(i)} w_b^{(i)} = \sum_{i=1}^k c_i^{(a)} c_i^{(b)} = \mathbf{c}^{(a)} \cdot \mathbf{c}^{(b)}, \quad \text{для } a, b \in [1, n]. \quad (3.17)$$

Тот факт, что среднее каждого вектора  $\mathbf{w}^{(i)}$  равно нулю означает, что каждый диагональный элемент  $\left( \mathbf{W} \cdot \mathbf{W}^T \right)_{jj}$  матрицы-произведения является дисперсией (с множителем  $k$ )  $j$ -го элемента (или  $j$ -ой координаты) вектора  $\mathbf{w}^{(i)}$ . В самом деле, из уравнения (3.17) находим, что

$$\begin{aligned} \left( \mathbf{W} \cdot \mathbf{W}^T \right)_{jj} &= \sum_{i=1}^k w_j^{(i)} w_j^{(i)} = \sum_{i=1}^k \left( w_j^{(i)} - 0 \right)^2 = \\ &= \sum_{i=1}^k \left( c_i^{(j)} - 0 \right)^2 = k \cdot \text{var} \left( \mathbf{c}^{(j)} \right). \end{aligned}$$

Внедиагональные элементы матрицы  $(\mathbf{W} \cdot \mathbf{W}^T)$  являются ковариациями векторов  $\mathbf{w}^{(i)}$ , то есть, элемент  $\left( \mathbf{W} \cdot \mathbf{W}^T \right)_{ab}$  равен ковариации координат  $a$  и  $b$  векторов  $\mathbf{w}^{(i)}$ . Из уравнения (3.17) также видно, что эти величины равны скалярным произведениям  $\mathbf{c}^{(a)} \cdot \mathbf{c}^{(b)}$  векторов  $\mathbf{c}^{(a)}$  и  $\mathbf{c}^{(b)}$ . Одной из основных задач преобразования изображения является приведение его к декоррелированной форме координат векторов. Теория вероятности говорит о том, что две координаты являются декоррелированными, если их ковариация равна нулю (другая цель – это концентрация энергии, но эти две задачи тесно связаны). Значит, необходимо найти матрицу  $\mathbf{A}$ , такую, что произведение  $\mathbf{W} \cdot \mathbf{W}^T$  будет диагональной матрицей.

Из определения матрицы  $\mathbf{W}$  находим, что

$$\mathbf{W} \cdot \mathbf{W}^T = (\mathbf{A} \mathbf{V}) \cdot (\mathbf{A} \mathbf{V})^T = \mathbf{A} \left( \mathbf{V} \cdot \mathbf{V}^T \right) \mathbf{A}^T.$$

Матрица  $\mathbf{V} \cdot \mathbf{V}^T$  является симметрической, ее элементами служат ковариации координат векторов  $\mathbf{v}^{(i)}$ , то есть,

$$(\mathbf{V} \cdot \mathbf{V}^T)_{ab} = \sum_{i=1}^k v_a^{(i)} v_b^{(i)}, \text{ при } a, b \in [1, n].$$

Раз матрица  $\mathbf{V} \cdot \mathbf{V}^T$  – симметрическая, то ее собственные векторы ортогональны. Нормализуем их (то есть, сделаем их ортонормальными) и выберем их в качестве строк матрицы  $\mathbf{A}$ . Получим следующий результат:

$$\mathbf{W} \cdot \mathbf{W}^T = \mathbf{A} (\mathbf{V} \cdot \mathbf{V}^T) \mathbf{A}^T = \begin{pmatrix} \lambda_1 & 0 & 0 & \cdots & 0 \\ 0 & \lambda_2 & 0 & \cdots & 0 \\ 0 & 0 & \lambda_3 & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & \lambda_n \end{pmatrix}.$$

При таком выборе матрицы  $\mathbf{A}$  матрица  $\mathbf{W} \cdot \mathbf{W}^T$  будет диагональной, причем элементы диагонали являются собственными числами матрицы  $\mathbf{V} \cdot \mathbf{V}^T$ . Матрица  $\mathbf{A}$  служит матрицей преобразования Кархунена–Лоэвэ; ее строки являются базисными векторами KLT, а энергией (дисперсией) преобразованных векторов служат собственные числа  $\lambda_1, \lambda_2, \dots, \lambda_n$  матрицы  $\mathbf{V} \cdot \mathbf{V}^T$ .

Базисные векторы для KLT вычисляются с помощью пикселов исходного изображения, то есть, они зависят от исходных данных. В конкретном методе сжатия эти векторы следует записывать в сжатый файл для использования декодером. Кроме того не известен быстрый метод вычисления этих векторов. Все эти факты делают метод KLT сугубо теоретическим без реальных приложений.

Честность – вот лучший образ.  
—Том Уилсон

### 3.6. Прогрессирующее сжатие изображений

Большинство современных методов сжатия изображений являются *прогрессирующими* (поступательными) или они легко делаются такими. Это особенно важно, если сжатое изображение передается по каналам связи, а затем декодируется и наблюдается получателем в режиме реального времени (например, это делается браузерами

всемирной паутины). Когда такое изображение принимается и разжимается, декодер способен очень быстро показать всю картинку в формате с низким качеством, а затем постепенно улучшать качество по мере приема остальной части сжатого изображения и его декодирования. Пользователь, наблюдая на экране развитие образа, может распознать все особенности этого изображения после декодирования всего 5–10% от его размера.

Это можно сравнить со сжатием растрово-сканированных изображений. Когда изображение сканируется и сжимается по строкам сверху вниз, а потом декодируется *последовательно*, пользователь обычно не может многое узнать о нем, наблюдая на дисплее лишь 5–10% от всего изображения. Поскольку картинки будут рассматриваться людьми, прогрессирующее сжатие является более предпочтительным, даже если оно, в целом, работает медленнее и имеет меньшую эффективность, чем непрогрессирующее сжатие.

Для того, чтобы лучше понять прогрессирующее сжатие, представьте себе, что кодер сначала сжимает самую важную часть изображения и помещает ее в файл, затем сжимается менее важная информация и добавляется в сжатый файл и так далее. Это объясняет также, почему потерю информации при прогрессирующем сжатии можно контролировать естественным образом: достаточно остановить сжатие в некоторой точке. Пользователь может менять долю опущенной информации с помощью параметра, который сообщает кодеру, как долго следует продолжать процесс сжатия. Чем раньше произойдет эта остановка, тем выше будет степень сжатия и тем больше будет размер утерянной информации.

Другое преимущество прогрессирующего сжатия становится несомненным, когда сжатый файл должен разжиматься много раз и каждый раз с разным разрешением. Декодер может в каждом случае остановить процесс декодирования, когда изображение достигло разрешения, доступного данному устройству.

*Образование – это прогрессирующее обнаружение нашего невежества.*  
— Уил Дюрант

Мы уже упоминали прогрессирующее сжатие изображений в связи с алгоритмом JPEG (стр. 185). Этот алгоритм использует DCT для разделения образа на пространственные частоты и начинает сжатие с низкочастотных компонентов. Поэтому декодер может быстро показать соответствующую им грубую картинку. Высокочастотные компоненты содержат детали изображения.

Кроме того, полезно представлять себе прогрессирующее декодирование как процесс улучшения качества изображения во времени. Это можно сделать тремя путями:

1. Закодировать пространственные частоты изображения прогрессирующим образом. Наблюдатель, следящий за декодированием, увидит такое изображение изменяющимся от расплывшегося до резкого. Методы, которые работают подобным образом, имеют среднюю скорость кодирования и медленную скорость декодирования. Этот тип сжатия часто называется *прогрессирующим по соотношению сигнал/шум*, или *прогрессирующим по качеству изображения*.
2. Начать с черно-белого полуточнового изображения, а потом добавить цвет и тени. При декодировании зритель с самого начала увидит изображение со всеми деталями, которые будут улучшаться по мере добавления цветов и красок. Метод векторного квантования применяет этот принцип компрессии. Такие алгоритмы отличаются медленным кодированием и быстрым декодированием.



Рис. 3.45. Последовательное декодирование.

3. Кодировать изображение по слоям, где ранние слои состоят из нескольких пикселов большого разрешения. Наблюдатель заметит постепенную детализацию картинки во времени при ее декодировании. Такой метод добавляет все новые детали по мере разжатия файла. Этот путь прогрессирующего сжатия называется *пирамидальным* или *иерархическим кодированием*. Большинство методов сжатия используют этот принцип, поэтому в этом параграфе обсуждаются общие идеи осуществления пирамидального кодирования. Рис. 3.46 иллюстрирует три принципа прогрессирующего сжатия, перечисленные выше. Они контрастируют с рис. 3.45, на котором изображены этапы последовательного декодирования.

Предположим, что изображение состоит из  $2^n \times 2^n = 4^n$  пикселов. Простейший метод прогрессирующего сжатия, который приходит в голову, состоит в вычислении каждого пикселя слоя  $i - 1$  в виде среднего группы  $2 \times 2$  пикселов слоя  $i$ . Значит, слой  $n$  – это исходное изображение ( $4^n$  пикселов размера 1), слой  $n - 1$  состоит из  $2^{n-1} \times 2^{n-1} = 4^{n-1}$  пикселов размера  $2 \times 2$ , и так далее до слоя  $4^{n-n} = 1$  из одного большого пикселя размера  $4^n$ , который представляет все изображение. Если изображение не слишком велико, то все слои можно сохранить в памяти компьютера. Затем слои записываются в файл в обратном порядке, начиная со слоя 1. Единственный пиксель слоя 1 является «родителем» четырех пикселов слоя 2, каждый из которых порождает по 4 пикселя слоя 3 и так далее. Этот процесс породит прогрессирующий файл образа, но без сжатия, поскольку общее число пикселов этой пирамиды равно

$$4^0 + 4^1 + \dots + 4^{n-1} + 4^n = (4^{n+1} - 1) / 3 \approx 1.33 \times 4^n \approx 1.33 \times (2 \times 2)^n,$$

что на 33% больше размера исходного числа!

Простой метод приведения числа пикселов пирамиды к исходному числу  $4^n$  состоит в сохранении только трех пикселов слоя  $i$  из четырех и в вычислении четвертого с помощью трех его «братьев» и одного родительского пикселя этой группы из предыдущего слоя  $i - 1$ .

**Пример:** На рис. 3.47с дано изображение размера  $4 \times 4$ , которое является третьим слоем прогрессирующего сжатия. Второй слой показан на рис. 3.47б, где, например, пиксель 81.25 является средним значением четырех пикселов 90, 72, 140 и 23 из третьего слоя. Единственный пиксель первого слоя приведен на рис. 3.47а.

Сжатый файл будет содержать только числа

54.125, 32.5, 41.5, 61.25, 72, 23, 140, 33, 18, 21, 18, 32, 44, 70, 59, 16

(конечно, правильно закодированные) из которых легко восстановить недостающие пиксели. Например, отсутствующий пиксель 81.25 вычисляется из уравнения  $(x + 32.5 + 41.5 + 61.25)/4 = 54.125$ .

Проблема заключается в том, что среднее значение целых чисел может быть нецелым числом. Если мы хотим, чтобы пиксели оставались целыми, то придется или смириться с потерей точности, или сохранять все более и более длинные целые числа. Например, если исходные пиксели имеют 8 разрядов, то сложение четырех 8-ми битовых чисел дает 10-ти битовое число. Деление этого числа на 4 для вычисления среднего и округление до целого вновь приводит к

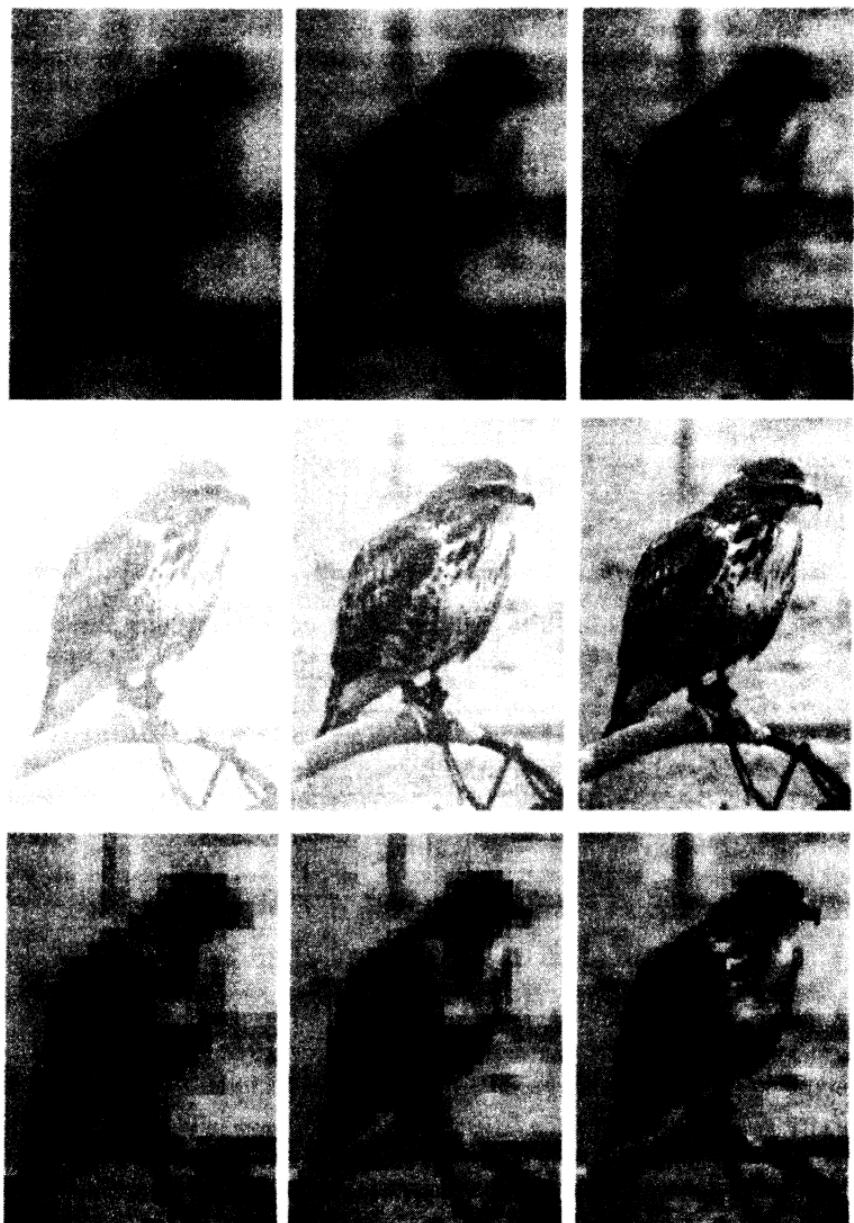


Рис. 3.46. Прогрессирующее декодирование.

8 разрядам, но с возможной потерей точности. Если потеря точности нежелательна, то можно представить наш пиксель второго слоя в виде 10-ти битовых чисел, а пиксель первого слоя – с помощью 12 разрядов. На рис. 3.47d,e,f показан результат округления, при котором происходит некоторая потеря информации. Содержимым сжатого файла будет последовательность

54, 33, 42, 61, 72, 23, 140, 33, 18, 21, 18, 32, 44, 70, 59, 16.

Первый отсутствующий пиксель 81 второго слоя можно вычислить из уравнения  $(x + 33 + 42 + 61)/4 = 54$ . Получится число 80 с небольшой ошибкой.

54.125	81.25	32.5	90	72	58	33
(a)	(b)	(c)	140	23	21	18
	61.25	41.5	100	70	72	18
			16	59	44	32
54	81	33	90	72	58	33
(d)	(e)	(f)	140	23	21	18
140	140	21	100	70	72	18
max	max	min	16	59	44	32
(g)	(h)	(i)	16	72		

Рис. 3.47. Прогрессирующее сжатие образа.

Лучшее решение заключается в использовании родителя группы при вычислении его четырех потомков. Это можно делать, вычисляя разность между родителем и его потомками и записывая эту разность (после подходящего кодирования) в слой  $i$  сжатого файла. Декодер восстанавливает разность и использует родителя из слоя  $i - 1$  для вычисления значений четырех пикселов. Для кодирования разностей можно применять метод Хаффмана или арифметическое кодирование. Если все слои вычислены и находятся в памяти, то

можно найти статистическое распределение этих разностей, которое можно использовать для достижения наилучшего статистического сжатия.

Если в памяти нет места для всех слоев, можно применить простую адаптивную модель. Она начинается присвоением счетчика 1 каждому значению разности (чтобы избегнуть проблему нулевой вероятности, см. [Salomon 2000]). После вычисления очередной разности, ей присваивается некоторая вероятность, и она кодируется, исходя из ее счетчика. После чего счетчик обновляется. Неплохо делать увеличение счетчика не единицу, а на большее число. Тогда исходные единичные значения счетчиков быстро станут незначимыми.

Некоторого улучшения можно достигнуть, если использовать родительский пиксель при вычислении трех его потомков, а затем этих трех потомков и родителя использовать при вычислении четвертого пикселя данной группы. Если четыре пикселя группы равны  $a$ ,  $b$ ,  $c$  и  $d$ , то их среднее равно  $v = (a + b + c + d) / 4$ . Это среднее становится частью слоя  $i - 1$ , а слой  $i$  может содержать лишь три разности  $k = a - b$ ,  $l = b - c$  и  $m = c - d$ . После того как декодер прочтет и декодирует эти три разности, он может вычислить все четыре пикселя группы с помощью значения  $v$  из предыдущего слоя. Вычисление  $v$  с помощью деления на 4 все еще означает потерю двух битов, но эту двухбитовую величину можно выделить до деления и кодировать ее отдельно вслед за тремя разностями.

Родительский пиксель не обязан быть равен среднему значению группы. Можно, например, выбрать в качестве родительского максимальный (или минимальный) пиксель группы. Преимущество такого метода заключается в том, что родительский пиксель равен одному из его потомков. Кодер просто закодирует три пикселя группы, а декодер декодирует три пикселя (или их разности) и дополнит группу четвертым родительским пикселом. При кодировании последовательных групп слоя, кодер должен выбирать поочередно максимальное или минимальное значение в качестве родителя, поскольку выбор только одного экстремума породит темнеющие или светлеющие слои. Рис. 3.47g,h,i показывает три слоя для этого случая. В сжатом файле будут записаны числа

140, (0), 21, 72, 16, (3), 90, 72, 23, (3), 58, 33, 18, (0), 18, 32, 44, (3), 100, 70, 59,

где число в скобках имеет длину в 2 бита. Оно говорит о том, в какой квадрант поместить родительский пиксель. Заметим, что квадранты

занумерованы следующим образом:

$$\begin{pmatrix} 0 & 1 \\ 3 & 2 \end{pmatrix}.$$

Выбор *медианы* группы производится несколько медленнее нахождения максимума или минимума, но улучшает динамику проявления слоев при прогрессирующей декомпрессии. По определению, медианой последовательности чисел  $(a_1, a_2, \dots, a_n)$  называется элемент  $a_i$ , такой, что половина (или почти половина) элементов последовательности меньше  $a_i$ , а другая половина – больше  $a_i$ . Если четыре элемента группы пикселов удовлетворяют неравенству  $a < b < c < d$ , то медианой будет служить число  $b$  или  $c$ . Основное достоинство выбора медианы для родительского пикселя заключается в том, что это позволит сгладить большие разности значений пикселов. В группе 1, 2, 3, 100 выбор в качестве родителя числа 2 или 3 является более характерным для этой группы, чем выбор среднего числа. Нахождение медианы требует двух сравнений, а для вычисления среднего понадобится деление на 4 (или правый сдвиг).

Если используется медиана при кодировании слоя  $i - 1$ , то оставшиеся три пикселя можно кодировать в слое  $i$  с помощью их разностей, за которыми следует 2-х битовый код, сообщающий, который из четырех был родительским. Другое небольшое преимущество использования медианы заключается в том, что когда декодер читает этот 2-х битовый код, то ему становится известно, сколько из трех пикселов больше медианы, а сколько – меньше. Например, если этот код говорит, что один пиксель меньше медианы, а два другие – больше, а прочитанный пиксель меньше медианы, то декодер точно знает, что оставшиеся пиксели будут больше медианы. Полученная информация изменяет распределение разностей пикселов, и это преимущество может быть реализовано с помощью трех таблиц счетчиков для определения вероятностей при кодировании разностей. Одна таблица используется при кодировании пикселов при условии, что декодер знает, что они больше медианы. Другая таблица применяется для кодирования, если известно, что пиксель меньше медианы. А третья таблица работает, если декодер не знает заранее их соотношение с медианой. Это обстоятельство улучшает сжатие на несколько процентов и показывает, как добавление новых признаков в метод сжатия дает общее уменьшение сжатого файла.

Некоторые методы прогрессирующего сжатия, используемые на практике, такие, как SPIHT и EZW, изложены в [Salomon 2000].

### 3.7. JPEG

JPEG является очень изощренным методом сжатия изображений с потерей и без потери информации. Он применим как к цветным, так и к полутоновым изображениям (но не к мультфильмам и анимациям). Этот алгоритм не очень хорошо сжимает двухуровневые черно-белые образы, но он прекрасно обрабатывает изображения с непрерывными тонами, в которых близкие пиксели обычно имеют схожие цвета. Важным достоинством метода JPEG является большое количество настраиваемых параметров, которые пользователь может выбирать по своему усмотрению, в частности, он может регулировать процент теряемой информации, а, значит, и коэффициент сжатия, в широком диапазоне. Обычно глаз не в состоянии заметить какого-либо ущерба даже при сжатии этим методом в 10 или 20 раз. Имеются две основные моды: с потерей (называемая также *базелиной*) и без потерь информации (которая не слишком эффективна и обычно дает фактор сжатия около 2). Большинство приложений прежде всего используют моду с потерей данных. Эта мода также содержит прогрессирующую и иерархическое кодирование.

JPEG является прежде всего методом сжатия. Его нельзя рассматривать в качестве полноценного стандарта представления изображений. Поэтому в нем не задаются такие специфические параметры изображения как геометрический размер пикселя, пространство цветов или чередование битовых строк.

JPEG был разработан как метод сжатия непрерывно-тоновых образов. Основные цели метода JPEG состоят в следующем:

1. Высокий коэффициент сжатия, особенно для изображений, качество которых расценивается как хорошее или отличное.
2. Большое число параметров, позволяющих искусенному пользователю экспериментировать с настройками метода и добиваться необходимого баланса сжатие/качество.
3. Хорошие результаты для любых типов непрерывно-тоновых изображений независимо от их разрешения, пространства цветов, размера пикселов или других свойств.
4. Достаточно изощренный метод сжатия, но не слишком сложный, позволяющий создавать соответствующие устройства и писать программы реализации метода для компьютеров большинства платформ.
5. Несколько мод операций: (а) Последовательная мода: все цветные компоненты сжимаются в виде простого сканирования слева направо и сверху вниз; (б) Прогрессирующая мода: изображение сжимается в виде нескольких блоков (называемых «сканами»), позволяю-

щими делать декомпрессию и видеть сначала грубые, а потом все более четкие детали изображения; (с) Мода без потерь информации (нужная на случай, если пользователь желает сохранить пиксели без изменений; при этом приходится расплачиваться низкой степенью сжатия); и (д) Иерархическая мода, когда изображение сжимается со множеством разрешений, позволяющая создавать блоки низкого разрешения, которые можно наблюдать перед блоками высокого разрешения.

Сокращение JPEG произведено от Joint Photographic Experts Group (объединенная группа по фотографии). Проект JPEG был инициирован совместно комитетом CCITT и ISO (the International Standard Organization, международная организация по стандартам). Он начался в июне 1987 года, а первый черновой алгоритм JPEG был разработан в 1991 году. Стандарт JPEG доказал свою эффективность и стал широко применяться для сжатия изображений, особенно во всемирной паутине.

Основные шаги сжатия метода JPEG, которые будут подробно описываться в следующих параграфах, состоят в следующем.

1. Цветное изображение преобразуется из RGB в представление светимость/цветность (§ 3.7.1; для полутонаовых черно-белых изображений этот шаг опускается). Глаз чувствителен к малым изменениям яркости пикселов, но не цветности, поэтому из компоненты цветности можно удалить значительную долю информации для достижения высокого сжатия без заметного визуального ухудшения качества образа. Этот шаг не является обязательным, но он очень важен, так как остальная часть алгоритма будет независимо работать с каждым цветным компонентом. Без преобразования пространства цветов из компонентов RGB нельзя удалить существенную часть информации, что не позволяет сделать сильное сжатие.

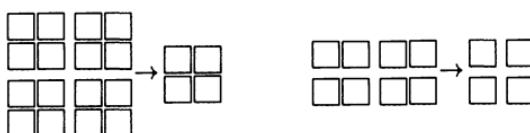


Рис. 3.48. 2h2v и 2h1v укрупнение пикселов.

2. Цветное изображение разбивается на крупные пиксели (этот шаг делается, если необходимо иерархическое сжатие; он всегда опускается для полутонаовых изображений). Эта операция не делается для компоненты яркости. Укрупнение пикселов (рис. 3.48) делается или в соотношении 2:1 по горизонтали и вертикали (так называемое

укрупнение 2h2v или «4:1:1» ) или в пропорциях 2:1 по горизонтали и 1:1 по вертикали (укрупнение 2h1v или «4:2:2» ). Поскольку это делается для двух компонентов из трех, 2h2v сокращает изображение до  $1/3 + (2/3) \times (1/4) = 1/2$  его размера, а 2h1v – до  $1/3 + (2/3) \times (1/2) = 2/3$  его размера. Компонента светимости не затрагивается, поэтому не происходит заметной потери качества изображения.

3. Пиксели каждой цветной компоненты собираются в блоки  $8 \times 8$ , которые называются *единицами данных*. Если число строк или столбцов изображения не кратно 8, то самая нижняя строка и самый правый столбец повторяются нужное число раз. Если мода с чередованием выключена, то кодер сначала работает со всеми единицами данных первой цветной компоненты, затем второй компоненты, а потом третьей компоненты. Если мода с чередованием включены, то кодер обрабатывает три самых верхних левых единицы данных трех компонентов (#1), затем три единицы данных (#2) справа от них и так далее.

4. Затем применяется *дискретное косинус-преобразование* (DCT, § 3.5.3) к каждой единице данных, в результате чего получаются блоки  $8 \times 8$  частот единиц данных (§ 3.7.2). Они содержат среднее значение пикселов единиц данных и следующие поправки для высоких частот. Все это приготавливает данные для основного шага выбрасывания части информации. Поскольку DCT использует трансцендентную функцию косинус, на этом шаге происходит незначительное изменение информации из-за ограниченности точности машинных вычислений. Это означает, что даже без основного шага потери данных (шаг 5 далее), происходит небольшое, крайне слабое искажение изображения.

5. Каждая из 64 компонент частот единиц данных делится на специальное число, называемое *коэффициентами квантования* (QC), которая округляется до целого (§ 3.7.4). Здесь информация невосполнимо теряется. Большие коэффициенты QC вызывают большую потерю, поэтому высокочастотные компоненты, обычно, имеют большие QC. Все 64 коэффициента QC являются изменяемыми параметрами, которые, в принципе, пользователь может регулировать самостоятельно. Однако большинство приложений используют таблицу QC, рекомендуемую стандартом JPEG для компонентов светимости и цветности (табл. 3.50).

6. Все 64 квантованных частотных коэффициента (теперь это целые числа) каждой единицы данных кодируются с помощью комбинации RLE и метода Хаффмана (§ 3.7.5). Вместо кодирования Хаффмана

может также применяться вариант арифметического кодирования, известный как кодер QM [Salomon 2000].

7. На последнем шаге добавляется заголовок из использованных параметров JPEG и результат выводится в сжатый файл. Сжатый файл может быть представлен в трех разных форматах: (1) формат *обмена*, когда файл содержит сжатый образ и все необходимые декодеру таблицы (в основном это таблицы квантования и коды Хаффмана), (2) *сокращенный* формат для сжатого изображения, где файл может не содержать всех таблиц, (3) *сокращенный* формат для таблиц, когда файл содержит только таблицы спецификаций без сжатых данных. Второй формат применяется, если при сжатии некоторые параметры и таблицы использовались по умолчанию, поэтому декодер их знает. Третий формат бывает полезен, если сжимается много однотипных изображений с помощью одних и тех же параметров. Если необходимо раскрыть эти изображения, то декодеру сначала направляется файл со спецификациями.

Декодер JPEG совершает обратные действия. (Значит, JPEG является симметричным методом сжатия.)

Прогрессирующая мода является опционной для JPEG. В этой моде высокочастотные коэффициенты DCT записываются в сжатый файл блоками, называемыми «сканами». Каждый прочитанный декодером скан дает возможность подправить и уточнить картинку. Идея заключается в том, что несколько первых сканов используются для быстрого показа изображения низкого качества. Далее происходит или декодирование следующих сканов, или отказ от дальнейшего декодирования. Плата за это заключается в том, что кодер должен хранить в буфере все коэффициенты всех единиц данных до того, как их послать в скан (так как они посылаются в скан в обратном порядке, а не в порядке их генерации, см. 176). Кроме того приходится делать весь процесс декодирования для каждого скана, что замедляет прогрессирующее декодирование.

На рис. 3.49а показан пример изображения с разрешением  $1024 \times 512$ . Это изображение разделено на  $128 \times 64 = 8192$  единиц данных, каждая из которых преобразована с помощью DCT в блок из 64 чисел по 8 бит. На рис 3.49б изображен параллелепипед, длина которого равна 8192 единицам данных, высота равна 64 коэффициентам DCT (коэффициент DC расположен наверху с номером 0), а ширина равна 8 битам каждого коэффициента.

После представления всех единиц данных в буфере памяти, кодер записывает их в сжатый файл одним из двух способов: с помо-

шью отбора спектра или методом последовательных приближений (рис. 3.49c,d). В обоих случаях первый скан состоит из коэффициентов DC.

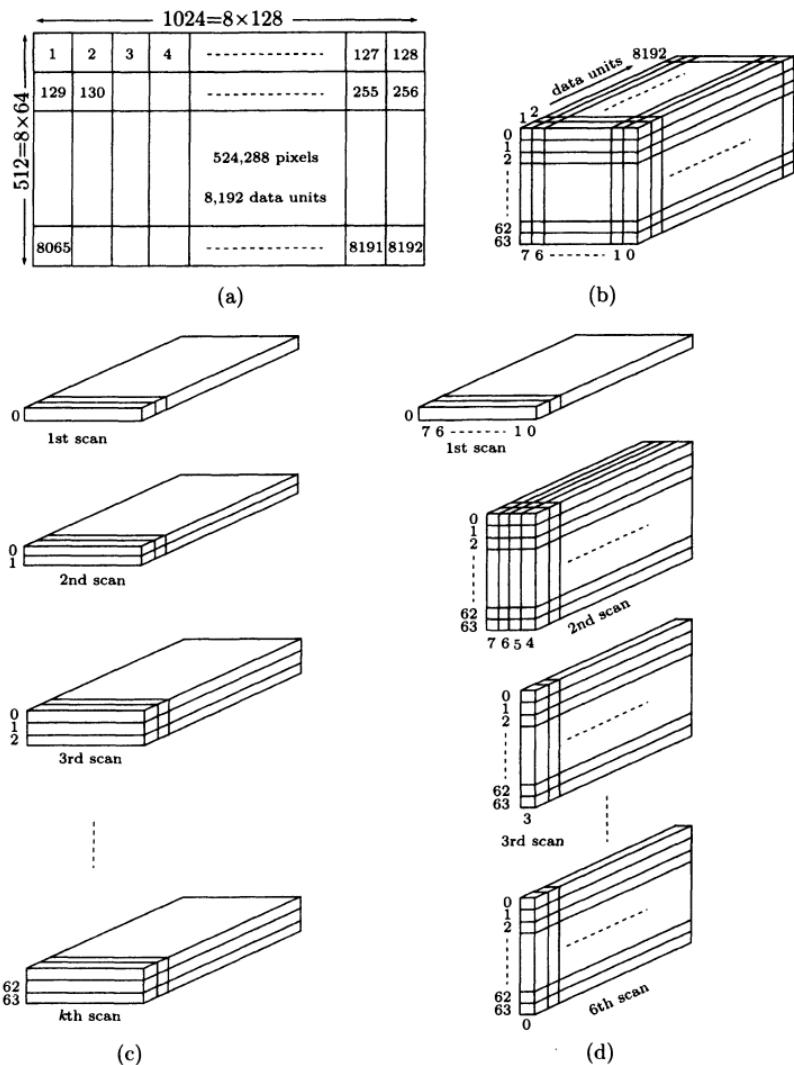


Рис. 3.49. Сканы в прогрессирующей mode JPEG.

Если делается отбор спектра, то каждый следующий скан состоит из нескольких последовательных коэффициентов (полосы) AC. При выборе метода последовательных приближений, второй скан



состоит из 4 самых значимых битов всех коэффициентов AC, а каждый следующий скан, имеющий номер от 3 до 6, добавляет по одному значащему биту (от третьего до нулевого).

В иерархической моде кодер сохраняет изображение в выходном файле несколько раз с разными разрешениями. Однако, каждая часть высокого разрешения использует части низкого разрешения, поэтому общее количество информации меньше, чем потребуется для раздельного хранения одинаковых изображений с разным разрешением. Каждая иерархическая часть может декодироваться прогрессирующим образом.

Иерархическая мода бывает полезной, если изображение с высоким разрешением требуется отобразить на устройстве визуализации с низким разрешением. Примером таких устройств могут служить устаревшие матричные принтеры, которые еще используются в работе.

Мода без потери данных в JPEG (§ 3.7.6) вычисляет «прогнозируемые» значения всех пикселов, берет разность между пикселом и его «прогнозом» для относительного кодирования. Кодирование производится как в шаге 5 методом Хаффмана или с помощью арифметического кодирования. Прогнозируемые величины вычисляются с использованием значений пикселов выше и слева от текущего (тех, которые уже закодированы). В следующих параграфах все шаги алгоритма будут разбираться более подробно.

*Если есть сомнения, то прогнозируйте сохранение текущей тенденции.*  
— Максим Меркин

### 3.7.1. Светимость

Главной международной организацией, занимающейся проблемами света и цвета, является Международный Комитет по Освещению (Commission Internationale de l'Éclairage, CIE). Эта организация отвечает за развитие стандартов и употребление терминов в этой области. Одним из первых достижений СИЕ явилось создание в 1931 году *хроматических диаграмм* (см. [Salomon 99]). Было показано, что для правильного отображения цвета достаточно трех компонент. Выражение некоторого цвета в виде триплета  $(x, y, z)$  похоже на обозначение точки в трехмерном пространстве, которое по аналогии называется *цветовым пространством*. Наиболее известным цветовым пространством является пространство RGB, в котором тремя

параметрами являются интенсивности красного, синего и зеленого в данном цвете. При отображении цвета на компьютере значения этих трех компонент выбираются в интервале от 0 до 255 (8 бит).

CIE дает определение цвету, как результат восприятия действия света видимого спектрального диапазона с длиной волны от 400 нм до 700 нм, попадающего на сетчатку глаза (нм, нанометр равен  $10^{-9}$  метра). Физическая мощность (или радиация) выражается в виде распределения мощности спектра (spectral power distribution, SPD), которое обычно состоит из 31 компоненты, причем каждая компонента представляет 10 нм видимой полосы.

CIE определяет яркость как атрибут визуального восприятия светящейся области глазом человека. Однако невозможно оценить количественно восприятие яркости мозгом человека, поэтому CIE определяет более практическую величину, которая называется *светимостью*. Она равна лучистой мощности, разделенной на функцию спектральной чувствительности, которое характеризует зрение. Световая отдача для стандартного наблюдателя определяется как положительная функция длины волны, которая имеет максимум, около 555 нм. Если проинтегрировать функцию распределения мощности спектра, деленную на функцию световой отдачи, то результатом будет светимость по CIE, которая обозначается Y. Светимость является весьма важной характеристикой в области обработки изображений и их сжатия.

Светимость пропорциональна мощности источника света. Она подобна интенсивности, но спектральный состав светимости соотносится с восприятием яркости глазом человека. Основываясь на результатах многочисленных экспериментов, светимость определяется как взвешенная сумма красного, зеленого и синего с весами 77/256, 150/256 и 29/256, соответственно.

Наш глаз очень чувствителен к малым изменениям светимости, поэтому удобно иметь цветовое пространство, в котором число Y является одним из трех компонентов. Простейший способ такого построения – это вычесть компоненту Y из красной и синей компонент RGB и использовать новое цветовое пространство Y, Cb = B – Y и Cr = R – Y. Последние две компоненты называются хроматическими (от греческого chroma – цвет, краска). Они выражают цвет в терминах присутствия или отсутствия синего (Cb) и красного (Cr) при данном значении светимости.

Различные числовые интервалы используются для выражения чисел Cb = B – Y и Cr = R – Y в разных приложениях. Пространство YPbPr оптимизировано для компонентов аналогового видео, а

пространство YCbCr лучше приспособлено для цифрового и студийного видео, а также для стандартов JPEG, JPEG 2000 и MPEG-1.

Цветовое пространство YCbCr было разработано как часть рекомендации ITU-R BT.601 (бывшая CCIR 601) при выработке всемирного стандарта для цифрового видео. Компонента Y имеет пределы от 16 до 235, а Cb и Cr изменяются от 16 до 240, причем 128 соответствует нулевым значениям. Существует также несколько форматов YCbCr для сэмплирования, такие как 4:4:4, 4:2:2, 4:1:1, и 4:2:0, которые также описаны в этих рекомендациях.

Связь между пространством RGB в интервале 16–235 и пространством YCbCr устанавливается в виде простых линейных соотношений. Это преобразование пространств можно записать в виде (заметьте, что синий цвет имеет малый вес)

$$\begin{aligned} Y &= (77/256)R + (150/256)G + (29/256)B, \\ Cb &= -(44/256)R - (87/256)G + (131/256)B + 128, \\ Cr &= (131/256)R - (110/256)G - (21/256)B + 128, \end{aligned}$$

а обратное преобразование равно

$$\begin{aligned} R &= Y + 1.371(Cr - 128), \\ G &= Y - 0.698(Cr - 128) - 0.336(Cb - 128), \\ B &= Y + 1.732(Cb - 128). \end{aligned}$$

Если перейти из пространства YCbCr в пространство RGB, то значения компонентов будут лежать в интервале 16–235 с возможным попаданием в области 0–15 и 236–255.

### 3.7.2. DCT

Дискретное косинус-преобразование (DCT) уже обсуждалось нами в § 3.5.3. Комитет JPEG остановил свой выбор именно на этом преобразовании из-за его хороших свойств, а также в силу того, что в нем не делается никаких ограничений на структуру сжимаемых данных. Кроме того, имеются возможности для ускорения DCT.

Стандарт JPEG применяет DCT не ко всему изображению, а к единицам данных (блоков) размера  $8 \times 8$  пикселов. Дело в том, что (1) применение DCT ко всему изображению использует большое число арифметических операций и поэтому делается медленно. Применение DCT к единицам данных вычисляется значительно быстрее. (2) Из опытов известно, что в непрерывно-тоновых изображениях корреляция пикселов сохраняется в малых областях. Пиксели такого изображения имеют величины (компоненты цвета или градацию

серого), близкие значениям окрестных пикселов, но дальние соседи уже не имеют корреляции. Следовательно, применение DCT ко всему изображению не произведет лучшее сжатие.

Однако следует отметить, что использование малых единиц данных имеет и обратную сторону. Применение DCT ко всему изображению (с последующим сокращением информации и декомпрессией) создает картину, более приятную для глаза. Совершение DCT над единицами данных делается существенно быстрее, но после компрессии и декомпрессии возможно появление блоковых артефактов (искусственных искажений), особенно на стыках блоков, поскольку разные блоки по разному взаимодействуют с этапом квантования, который следует за DCT.

Формулы DCT для JPEG совпадают с (3.9). Мы их повторяем для удобства изложения:

$$G_{ij} = \frac{1}{4} C_i C_j \sum_{x=0}^7 \sum_{y=0}^7 p_{xy} \cos \left( \frac{(2y+1)j\pi}{16} \right) \cos \left( \frac{(2x+1)i\pi}{16} \right), \quad (3.9)$$

где  $C_f = \begin{cases} \frac{1}{\sqrt{2}}, & f = 0, \\ 1, & f > 0, \end{cases}$  и  $0 \leq i, j \leq 7$ .

Преобразование DCT является основой сжатия с потерей информации в стандарте JPEG. Метод JPEG «выбрасывает» незначимую часть информации из изображения с помощью деления каждого из 64 коэффициентов DCT (особенно те, которые расположены в правой нижней части блока) на коэффициент квантования QC. В общем случае, каждый коэффициент DCT делится на особый коэффициент QC, но все 64 параметра QC могут изменяться по усмотрению пользователя (§ 3.7.4).

Декодер JPEG вычисляет обратное преобразование DCT (IDCT) с помощью уравнений (3.10), которые мы здесь повторяем

$$p_{xy} = \frac{1}{4} \sum_{i=0}^7 \sum_{j=0}^7 C_i C_j G_{ij} \cos \left( \frac{(2y+1)j\pi}{16} \right) \cos \left( \frac{(2x+1)i\pi}{16} \right), \quad (3.10)$$

где  $C_f = \begin{cases} \frac{1}{\sqrt{2}}, & f = 0; \\ 1, & f > 0. \end{cases}$

Здесь используются квантованные коэффициенты DCT, а в результате получаются значения пикселов  $p_{xy}$ . Говоря языком математики, преобразование DCT является взаимно однозначным линейным

отображением 64-мерного векторного пространства изображения в пространство частот той же размерности. IDCT является обратным отображением. Если бы были возможны вычисления с абсолютной точностью, то после применения DCT и IDCT (без квантования) результат совпал бы с исходным блоком. На практике, конечно, используется квантование, но если его делать аккуратно, то результатом этих преобразований будет блок, который очень близок исходному блоку изображения.

### 3.7.3. Практическое DCT

Уравнения (3.9) легко переводятся на любой язык программирования высокого уровня. Однако, имеется несколько возможностей для существенного ускорения вычисления этих величин. Эти формулы лежат в самом «сердце» метода JPEG, поэтому ускорение вычислений просто необходимо. Опишем несколько полезных приемов.

1. Независимо от размера изображения, используется только 32 значения функции косинус (см. следующий абзац). Их можно один раз вычислить и использовать много раз в операциях над единицами данных  $8 \times 8$ . Тогда вычисление выражения

$$p_{xy} \cos\left(\frac{(2y+1)j\pi}{16}\right) \cos\left(\frac{(2x+1)i\pi}{16}\right)$$

будет состоять всего из двух операций умножения. Двойная сумма (3.9) потребует  $64 \times 2 = 128$  умножений и 63 сложений.

(Аргументы функции косинус, используемые в DCT, имеют вид  $(2x+1)i\pi/16$ , где  $x$  и  $i$  – целые числа в интервале  $[0, 7]$ . Их можно записать в виде  $n\pi/16$ , где  $n$  – целое из интервала  $[0, 15 \times 7]$ . Поскольку косинус-функция периодическая, для нее  $\cos(32\pi/16) = \cos(0\pi/16)$ ,  $\cos(33\pi/16) = \cos(1\pi/16)$ , и так далее. В результате потребуется только 32 значения  $\cos(n\pi/16)$  при  $n = 0, 1, \dots, 31$ . Я признателен В.Сараванану (V.Saravanan), который указал мне на эту особенность DCT.)

2. Анализ двойной суммы (3.9) позволяет переписать ее в виде произведения матриц  $\mathbf{C}\mathbf{P}\mathbf{C}^T$ , где  $\mathbf{P}$  – исходная  $8 \times 8$  матрица пикселов, а матрица  $\mathbf{C}$  определяется формулами

$$C_{ij} = \begin{cases} \frac{1}{\sqrt{8}}, & i = 0; \\ \frac{1}{2} \cos\left(\frac{(2j+1)i\pi}{16}\right), & i > 0, \end{cases}$$

а  $\mathbf{C}^T$  – транспонированная матрица  $\mathbf{C}$ .

В итоге, вычисление одного элемента матрицы  $\mathbf{CP}$  требует восьми умножений и семи (ну пусть тоже восьми, для простоты) сложений. Умножение двух матриц  $\mathbf{C}$  и  $\mathbf{P}$  состоит из  $64 \times 8 = 8^3$  умножений и столько же сложений. Умножение  $\mathbf{CP}$  на  $\mathbf{C}^T$  потребует того же числа операций, значит одно DCT единицы данных состоит из  $2 \times 8^3$  операций умножения (и сложения). Если исходное изображение состоит из  $n \times n$  пикселов, причем  $n = 8q$ , где  $q$  – число единиц данных, то для вычисления DCT всех этих единиц понадобиться  $2q^28^3$  умножений (и столько же сложений). Для сравнения, вычисление одного DCT всего изображения потребует  $2n^3 = 2q^38^3 = (2q^28^3)q$  операций. С помощью разделения изображения на единицы данных мы сократили общее число умножений (и сложений) в  $q$  раз. К сожалению, число  $q$  не может быть слишком большим, поскольку это уменьшает размер единицы данных.

Напомним, что цветное изображение состоит из трех компонент (обычно это RGB, которое преобразуется в YCbCr или в YPbPr). Каждая компонента преобразуется отдельно, что дает общее число операций равное  $3 \cdot 2q^28^3 = 3072q^2$ . Для изображения с разрешением  $512 \times 512$  пикселов потребуется сделать  $3072 \times 64^2 = 12\,582\,912$  умножений (и сложений).

3. Другой путь ускорения DCT состоит в выполнении арифметических вычислений над числами, представленными в форме с фиксированной точкой, а не в форме с плавающей точкой. Для многих компьютеров операции над числами с фиксированной точкой делаются существенно быстрее операций с плавающей точкой (некоторые высокопроизводительные компьютеры, вроде CDC 6400, CDC 7600 и различные системы CRAY являются заметными исключениями).

Бессспорно, лучший алгоритмом для DCT описан в [Feig, Linz-er 90]. Для него требуется всего 54 умножения и 468 сложений и сдвигов. На сегодняшний день имеются различные специализированные микросхемы, которые выполняют эти процедуры очень эффективно. Интересующийся читатель может познакомиться в [Loeffler et al. 89] с быстрым алгоритмом одномерного DCT, использующим всего 11 умножений и 29 сложений.

### 3.7.4. Квантование

После вычисления всех коэффициентов DCT их необходимо проквантовать. На этом шаге происходит отбрасывание части информации (небольшие потери происходят и на предыдущем шаге из-за конечной точности вычислений на компьютере). Каждое число из матриц коэффициентов DCT делится на специальное число из «та-

блицы квантования», а результат округляется до ближайшего целого. Как уже отмечалось, необходимо иметь три такие таблицы для каждой цветовой компоненты. Стандарт JPEG допускает использование четырех таблиц, и пользователь может выбрать любую из этих таблиц для квантования компонентов цвета. Все 64 числа из таблицы квантования являются параметрами JPEG. В принципе, пользователь может поменять любой коэффициент для достижения большей степени сжатия. На практике весьма сложно экспериментировать с таким большим числом параметров, поэтому программное обеспечение JPEG использует два подхода:

1. Таблица квантования, принятая по умолчанию. Две такие таблицы, одна для компоненты светимости (и для градации серого цвета), а другая – для хроматических компонент, являются результатом продолжительного исследования со множеством экспериментов, проделанных комитетом JPEG. Они являются частью стандарта JPEG и воспроизведены в табл. 3.50. Видно, как коэффициенты QC таблиц растут при движении из левого верхнего угла в правый нижний угол. В этом отражается сокращение коэффициентов DCT, соответствующих высоким пространственным частотам.

2. Вычисляется простая таблица коэффициентов квантования, зависящая от параметра  $R$ , который задается пользователем. Простые выражения типа  $Q_{ij} = 1 + (i + j) \times R$  гарантируют убывание коэффициентов из левого верхнего угла в правый нижний.

16	11	10	16	24	40	51	61	17	18	24	47	99	99	99	99
12	12	14	19	26	58	60	55	18	21	26	66	99	99	99	99
14	13	16	24	40	57	69	56	24	26	56	99	99	99	99	99
14	17	22	29	51	87	80	62	47	66	99	99	99	99	99	99
18	22	37	56	68	109	103	77	99	99	99	99	99	99	99	99
24	35	55	64	81	104	113	92	99	99	99	99	99	99	99	99
49	64	78	87	103	121	120	101	99	99	99	99	99	99	99	99
72	92	95	98	112	100	103	99	99	99	99	99	99	99	99	99

Светимость

Цветность

Табл. 3.50. Рекомендуемые таблицы квантования.

Если квантование сделано правильно, то в блоке коэффициентов DCT останется всего несколько ненулевых коэффициентов, которые будут сконцентрированы в левом верхнем углу матрицы. Эти числа являются выходом алгоритма JPEG, но их следует еще сжать перед записью в выходной файл. В литературе по JPEG это сжатие называется «энтропийным кодированием», детали которого будут

разбираясь в § 3.7.5. Три технических приема используется при энтропийном кодировании для сжатия целочисленных матриц  $8 \times 8$ .

1. 64 числа выстраиваются одно за другим как при сканировании зигзагом (см. рис. 3.5а). В начале стоят ненулевые числа, за которыми обычно следует длинный хвост из одних нулей. В файл выводятся только ненулевые числа (после надлежащего кодирования) за которыми следует специальный код EOB (end-of-block, конец блока). Нет необходимости записывать весь хвост нулей (можно также сказать, что EOB кодирует длинную серию нулей).

**Пример:** В табл. 3.51 приведен список гипотетических коэффициентов DCT, из которых только 4 не равны нулю. При зигзагообразном упорядочении этих чисел получается последовательность коэффициентов: 1118, 2, 0,  $\underbrace{-2, 0, \dots, 0}_{13}$ ,  $\underbrace{-1, 0, \dots, 0}_{46}$ .

1118	2	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
-2	0	0	-1	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Табл. 3.51. Квантованные коэффициенты.

А как написать подпрограмму для считывания элементов матрицы по зигзагу? Простейший способ состоит в ручном прослеживании этого пути и в записи результата в массив структур *zz*, в котором каждая структура состоит из пары координат клеток, через которые проходит зигзагообразный путь (см. рис. 3.52).

Если компоненты структуры *zz* обозначить *zz.r* и *zz.c*, то путь по зигзагу можно совершить с помощью следующего цикла

```
for (i=0; i<64; i++){
    row:=zz[i].r; col:=zz[i].c
    ...data_unit[row][col]...}
```

- Ненулевые коэффициенты преобразования сжимаются по методу Хаффмана (см. § 3.7.5).
- Первое из этих чисел (коэффициент DC, см. стр. 145) обрабатывается отдельно от других чисел (коэффициентов AC).

(0, 0)	(0, 1)	(1, 0)	(2, 0)	(1, 1)	(0, 2)	(0, 3)	(1, 2)
(2, 1)	(3, 0)	(4, 0)	(3, 1)	(2, 2)	(1, 3)	(0, 4)	(0, 5)
(1, 4)	(2, 3)	(3, 2)	(4, 1)	(5, 0)	(6, 0)	(5, 1)	(4, 2)
(3, 3)	(2, 4)	(1, 5)	(0, 6)	(0, 7)	(1, 6)	(2, 5)	(3, 4)
(4, 3)	(5, 2)	(6, 1)	(7, 0)	(7, 1)	(6, 2)	(5, 3)	(4, 4)
(3, 5)	(2, 6)	(1, 7)	(2, 7)	(3, 6)	(4, 5)	(5, 4)	(6, 3)
(7, 2)	(7, 3)	(6, 4)	(5, 5)	(4, 6)	(3, 7)	(4, 7)	(5, 6)
(6, 5)	(7, 4)	(7, 5)	(6, 6)	(5, 7)	(6, 7)	(7, 6)	(7, 7)

Рис. 3.52. Координаты зигзагообразного пути.

### 3.7.5. Кодирование

Прежде всего обсудим пункт 3 из конца предыдущего параграфа. Каждая матрица  $8 \times 8$  квантованных коэффициентов DCT содержит коэффициент DC [в позиции (0,0) в левом верхнем углу], а также 63 коэффициента AC. Коэффициент DC равен среднему значению всех 64 пикселов исходной единицы. Наблюдения показывают, что при сжатии непрерывно-тоновых изображений, коэффициенты DC соседних единиц обычно являются коррелированными. Известно, что этот коэффициент равен сумме всех пикселов блока с некоторым общим множителем. Все это указывает на то, что коэффициенты DC близких блоков не должны сильно различаться. Поэтому JPEG записывает первый (закодированный) коэффициент DC, а затем кодирует разности коэффициентов DC последовательных блоков.

**Пример:** Если первые три единицы данных размера  $8 \times 8$  имеют квантованные коэффициенты DC равные 1118, 1114 и 1119, то JPEG записывает для первого блока число 1118 (закодированное по Хаффману, см. далее), за которым следует 63 закодированных коэффициента AC. Для второго блока на выходе будет стоять число  $1114 - 1118 = -4$  (также кодированное по Хаффману) впереди 63 (кодированных) коэффициента AC этого блока. Третьему блоку будет соответствовать кодированная запись  $1119 - 1114 = 5$  и следующие 63 коэффициента AC. Этот путь также позволяет меньше беспокоиться о проблемах, связанных с переполнением, так как разности обычно малы.

Кодирование разностей коэффициентов DC совершается с помощью табл. 3.53. В этой таблице записаны так называемые *унарные коды*, которые определяются следующим образом. Унарный код неотрицательного целого числа  $n$  состоит из строки в  $n$  единиц, за которыми следует один 0 или, наоборот,  $n$  нулей и одна 1. Длина унарного кода целого числа  $n$  равна  $n + 1$  бит.

Каждая строка табл. 3.53 начинается с ее номера (слева); в конце стоит унарный код строки, а между ними располагаются некоторые

числа. В каждой следующей строке записано больше чисел, чем в предыдущей, но они отличаются от чисел всех предыдущих строк. В строке  $i$  находятся числа из интервала  $[-(2^i - 1), +(2^i - 1)]$ , за вычетом чисел интервала  $[-(2^{i-1} - 1), +(2^{i-1} - 1)]$ . Длина строк растет очень быстро, поэтому такие данные не удобно представлять в виде простого двумерного массива. На самом деле, для их хранения не нужна никакая структура данных, поскольку подходящая программа легко определит позицию числа  $x$  в таблице, анализируя биты этого числа.

0:	0															0
1:	-1	1														10
2:	-3	-2	2	3												110
3:	-7	-6	-5	-4	4	5	6	7								1110
4:	-15	-14	...	-9	-8	8	9	10	...	15						11110
5:	-31	-30	-29	...	-17	-16	16	17	...	31						111110
6:	-63	-62	-61	...	-33	-32	32	33	...	63						1111110
7:	-127	-126	-125	...	-65	-64	64	65	...	127						11111110
14:	-16383	-16382	-16381	...	-8193	-8192	8192	8193	...	16383	1111111111111111					
15:	-32767	-32766	-32765	...	-16385	-16384	16384	16385	...	32767	1111111111111111					
16:	32768										1111111111111111					

Табл. 3.53. Кодирование разностей коэффициентов DC.

Первый коэффициент DC из нашего примера, который следует закодировать, равен 1118. Он находится в строке 11 и столбце 930 таблицы (столбцы занумерованы, начиная с нулевого). Тогда оно кодируется последовательностью 111111111110|01110100010 (унарный код строки 11, за которым следует двоичное представление числа 930 из 11 бит). Вторая разность коэффициентов DC, число  $-4$ , расположена в строке 3 и столбце 3; она кодируется в виде 1110|011 (унарный код строки 3 и число 3 в виде 3 бит). Третья разность 5 расположена в строке 3, столбец 5, поэтому ее кодом служит 1110|101.

Разберемся теперь с пунктом 2 предыдущего параграфа, когда надо кодировать 63 коэффициента AC. Это сжатие использует кодирование RLE в сочетании с методом Хаффмана или с арифметическим кодированием. Идея заключается в том, что в последовательности коэффициентов AC, как правило, имеется всего несколько ненулевых элементов, между которыми стоят серии нулей. Для каждого ненулевого числа  $x$  (1) кодер определяет число  $Z$  предшествующих ему нулей; (2) затем он находит число  $x$  в табл. 3.53 и готовит номер строки и столбца ( $R$  и  $C$ ); (3) пара  $(R, Z)$  (не  $(R, C)!$ ) используется для нахождения соответствующего числа по строке и столбцу в табл. 3.54; (4) наконец, полученный из этой

таблицы код Хаффмана присоединяется к  $C$  (где  $C$  записывается в виде R-битного числа); результатом этих действий служит код, выдаваемый на выход кодером JPEG для этого АС коэффициента  $x$  и всех предыдущих нулей. (Можно перевести дух.)

R	Z:	0:	1	...	15
0:		1010			11111111001(ZRL)
1:		00	1100	...	1111111111110101
2:		01	11011	...	1111111111110110
3:		100	1111001	...	1111111111110111
4:		1011	111110110	...	1111111111111000
5:		11010	11111110110	...	1111111111111001
⋮		⋮			

Табл. 3.54. Кодирование коэффициентов АС.

В табл. 3.54 приведен некоторый произвольный код Хаффмана, не тот, который предлагается комитетом JPEG. А стандарт JPEG рекомендует использовать для этих целей коды из табл. 3.55 и 3.56. При этом разрешается использовать до четырех таблиц Хаффмана, за исключением моды базелины, когда можно использовать только две таблицы. Читатель обнаружит код EOB в позиции (0,0) и код ZRL в позиции (0,15). Код EOB означает конец блока, а код ZRL замещает 15 последовательных нулей, когда их число превышает 15. Эти коды рекомендованы для компонентов светимости (табл. 3.55). Коды EOB и ZRL, рекомендованные для коэффициентов АС хроматических компонентов из табл. 3.56 равны 00 и 1111111010, соответственно.

**Пример:** Еще раз рассмотрим последовательность

$$1118, 2, 0, -2, \underbrace{0, \dots, 0}_{13}, -1, \underbrace{0, \dots, 0}_{46}.$$

Первый АС коэффициент  $x = 2$  не имеет перед собой нулей, то есть, для него  $Z = 0$ . Он находится в табл. 3.53 в строке 2 и столбце 2, поэтому, для него  $R = 2, C = 2$ . Код Хаффмана из табл. 3.54 в позиции  $(R, Z) = (2, 0)$  равен 01. Значит, окончательный код для  $x = 2$  будет 01|10. Следующий ненулевой коэффициент  $-2$  имеет один предшествующий нуль, то есть, для него  $Z = 1$ . Он стоит в табл. 3.53 в строке 2 и столбце 1, тогда  $R = 2, C = 1$ . Код Хаффмана из табл. 3.54 в позиции  $(R, Z) = (2, 1)$  равен 11011. В итоге кодом

числа  $-2$  будет служить последовательность  $11011|01$ . Последний ненулевой коэффициент АС равен  $-1$ , ему предшествует 13 нулей, и  $Z = 13$ . Сам коэффициент расположен в табл. 3.53 в строке 1 и столбце 0, то есть,  $R = 1, C = 0$ . Пусть в табл. 3.54 в позиции  $(R, Z) = (1, 13)$  находится код  $1110101$ . Тогда кодом для  $-1$  будет  $1110101|0$ .

Z	R				
	1 6	2 7	3 8	4 9	5 A
0 00	01 1111000	100 11111000	1011 111111110000010	11010 1111111110000011	
1 1100	11011 111111110000100	11110001 111111110000101	111110110 111111110000110	111111110110 111111110000111	111111110110 1111111100001000
2 11100	11111001 1111111100001010	1111110111 1111111100001011	111111110100 1111111100001100	111111110001001 111111110001101	
3 111010	111110111 1111111110010001	11111110101 1111111110010010	111111110001111 1111111110010100	1111111110001111 1111111110010101	
4 111011	1111111000 1111111110011001	111111110010110 1111111110011010	1111111110010111 1111111110011100	1111111110011001 1111111110011101	
5 1111010	11111110111 1111111110100001	111111110011110 1111111110100001	1111111110011111 1111111110100010	1111111110011111 1111111110100010	
6 1111011	111111110110 1111111110101001	1111111110100110 1111111110101010	11111111110100111 11111111110101010	11111111110101000 111111111101010101	
7 11111010	1111111110111 1111111111011001	11111111110101110 11111111110110010	111111111110101111 1111111111101100100	111111111110101111 1111111111101100101	
8 111111000	1111111111000000 11111111110111001	11111111110110110 11111111110111010	111111111110110111 111111111110111100	111111111110110111 111111111110111101	
9 11111110001	11111111111011110 111111111111000010	11111111111011111 111111111111000010	1111111111110000000 1111111111110000101	1111111111110000000 1111111111110000101	
A 1111111010	1111111111100011 1111111111100110	11111111111001000 11111111111001100	111111111111001001 111111111111001101	111111111111001001 111111111111001101	
B 11111111001	11111111111010000 111111111111010100	11111111111010001 111111111111010101	111111111111010010 111111111111010111	111111111111010010 111111111111010111	
C 11111111010	1111111111101001 11111111111101101	1111111111101010 11111111111101111	111111111111010111 111111111111100000	111111111111010111 111111111111100001	
D 111111111000	11111111111100010 11111111111110010	11111111111100011 11111111111110011	111111111111100100 111111111111100101	111111111111100100 111111111111100101	
E 111111111101000	11111111111110001 111111111111110001	111111111111110010 111111111111111001	1111111111111110110 11111111111111110011	1111111111111110111 11111111111111110100	
F 1111111111001	11111111111110101 111111111111111001	11111111111111010 111111111111111101	1111111111111111011 11111111111111111101	11111111111111110111 11111111111111111110	

Табл. 3.55. Рекомендованные коды Хаффмана для коэффициентов АС светимости.

Наконец, хвост из последних нулей кодируется как 1010 (EOB, конец блока). Значит, выходом для всех коэффициентов АС будет последовательность 011011011101110101010. Мы ранее установили, что кодом коэффициентом DC станет двоичная последовательность 111111111110|1110100010. Итак, окончательным кодом всего 64-пиксельного блока данных будет 46-битовое число

111111111110111010001001101101110111010101010.

Эти 46 бит кодируют одну цветную компоненту единицы данных. Предположим, что две остальные компоненты будут также закодированы 46-битовыми числами. Если каждый пиксель исходно состоял из 24 бит, то получим фактор сжатия равный  $64 \times 24 / (46 \times 3) = 11.13$ ; очень неплохой результат!

Z	R				
	1 6	2 7	3 8	4 9	5 A
0	01 111000	100 1111000	1010 11110100	11000 111110110	11001 11111110100
1	1011 111111110101	111001 111111110001000	11110110 111111110001001	111110101 111111110001010	11111110110 111111110001011
2	11010 111111110001100	11110111 111111110001101	1111110111 111111110001110	111111110110 111111110001111	11111111000010 111111110001000
3	11011 111111110010010	11111000 111111110010011	1111111000 111111110010100	111111110111 111111110010101	111111110010001 111111110010110
4	111010 111111110011010	11110110 111111110011011	111111110010111 111111110011000	111111110011000 111111110011101	111111110011001 111111110011110
5	111011 111111110100010	1111111001 111111110100011	111111110011111 111111110100100	111111110100000 111111110100101	111111110100001 111111110100110
6	1111001 111111110101010	111111110101011 111111110101100	111111110101111 111111110101101	111111110101000 111111110101101	111111110101001 111111110101110
7	1111010 111111110100101	11111111000 111111110100111	111111110101111 111111110101100	111111110101000 111111110101101	111111110101001 111111110101110
8	11111001 111111110101011	111111110101111 111111110101100	111111110110000 111111110110101	111111110110101 111111110110110	1111111101101010 111111110110111
9	111110111 11111111000100	111111111000000 111111111000101	111111111000001 111111111000110	111111111000010 111111111000111	111111111000011 111111111000100
A	111111000 11111111001101	111111111001001 111111111001110	111111111001010 111111111001111	111111111001011 111111111001100	111111111001100 111111111001001
B	111111001 111111110101010	111111111010010 111111111010111	111111111010011 111111111010100	111111111010100 111111111010101	1111111110101010 111111111010110
C	1111111010 11111111101111	111111111010101 111111111000000	111111111010110 111111111000010	111111111010111 111111111000011	111111111010110 111111111000100
D	11111111001 111111111101000	111111111100100 111111111101001	111111111100101 111111111101010	111111111100110 111111111101011	111111111100111 111111111101010
E	11111111100000 11111111110001	111111111101001 111111111101002	111111111101110 111111111101101	111111111101111 111111111101100	111111111101100 111111111101101
F	111111111100001 111111111101010	111111111101010 111111111101101	111111111110111 111111111110100	111111111111000 111111111111101	111111111111100 111111111111110

Табл. 3.56. Рекомендованные коды Хаффмана для коэффициентов АС хроматических компонент.

Те же таблицы (3.53 и 3.54) должны быть использованы декодером для восстановления блока данных. Они задаются по умолчанию кодеком JPEG, или могут быть специально построены для данного конкретного изображения на предварительном проходе. Однако сам JPEG не предусматривает алгоритма для построения таких таблиц. Конкретный кодек может самостоятельно сделать это.

Некоторые варианты JPEG предусматривают использование версии с арифметическим кодированием, которая называется кодированием QM. Это кодирование также устанавливается стандартом JPEG. Этот метод является адаптивным и для его работы не тре-

буются таблицы 3.53 и 3.54. Он приспособливает схему кодирования к статистическим свойствам конкретного изображения по мере его обработки. С помощью арифметического кодирования можно улучшить показатели компрессии метода Хаффмана на 5–10% для типичного непрерывно-тонового изображения. Однако этот метод имеет весьма сложную реализацию по сравнению с методом Хаффмана, поэтому он редко используется в приложениях.

### 3.7.6. Мода без потери данных

В этой моде метод JPEG использует комбинации разностей пикселов для уменьшения их значений перед тем, как они будут сжаты. Эти разности называются *прогнозами*. Величины некоторых близких пикселов вычитаются из данного пикселя для получения малого числа, которое будет сжиматься по методу Хаффмана или с помощью арифметического кодирования. На рис. 3.57а показан некоторый пиксель X и три соседних пикселя A, B и C. На рис. 3.57б даны восемь возможных линейных комбинаций (прогнозов) пикселя и его соседей. В моде без потерь пользователь может самостоятельно выбрать подходящий прогноз, а декодер вычтет эту комбинацию из пикселя X. Результатом, как правило, является малое число, для которого будет производиться энтропийное кодирование, очень близкое методу, использованному при кодировании коэффициентов DC в § 3.7.5.

Порядковый номер	Прогноз
0	нет прогноза
1	A
2	B
3	C
4	A+B-C
5	A+(B-C)/2
6	B+(A-C)/2
7	(A+B)/2

(a)

(b)

Табл. 3.57. Предсказание пикселов в моде без потерь.

Прогноз 0 используется только в иерархической моде JPEG. Прогнозы 1, 2 и 3 называются «одномерными», а прогнозы 4, 5, 6 и 7 – «двумерными».

Следует отметить, что мода без потерь не может быть очень эффективной. Ее фактор сжатия обычно находится около 2, и в этом



он значительно проигрывает другим методам сжатия изображений без потерь. По этой причине, многие популярные приложения, в которые встроен JPEG, не предусматривают возможность этой моды. Даже базелинна мода JPEG, если в ней задать в виде параметра минимальную потерю информации, работает недостаточно эффективно. В результате основные приложения не позволяют устанавливать этот параметр в минимальное значение. Достоинство метода JPEG прежде всего заключается в произведении сильно сжатых изображений, которые практически невозможно отличить от оригинала. Поняв это, ISO решило выпустить другой стандарт для сжатия без потерь непрерывно-тоновых изображений. Это хорошо известный метод JPEG-LS, который будет описан в § 3.8.

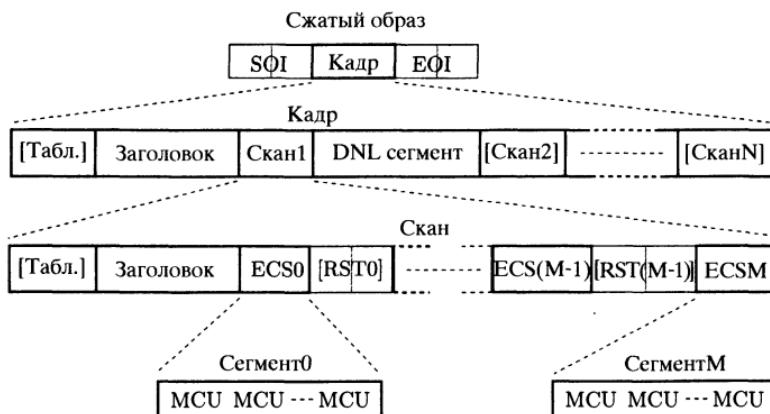


Рис. 3.58. Формат файла JPEG.

### 3.7.7. Сжатый файл

JPEG создает сжатый файл, в котором находятся все параметры, маркеры и, конечно, сжатые единицы данных изображения. Параметры состоят из слов длины 4 бита (объединяемых в пары), из одного байта или из двух байт. Маркеры необходимы для разделения файла на части. Маркеры имеют длину 2 байта. Первый байт равен 'FF'X, а второй – не ноль и не 'FF'X. Перед маркером может стоять несколько байтов с 'FF'X.

В табл. 3.59 перечислены все маркеры JPEG (первые четыре группы состоят из маркеров начала кадра). Сжатые единицы данных комбинируются в минимальные единицы данных (MCU, mini-

mal data unit), где MCU состоит или из одной единицы (мода без чередования) или из трех единиц данных всех цветных компонент (мода с чередованием).

На рис. 3.58 показаны все основные части выходного файла, сжатого по методу JPEG (части, заключенные в квадратные скобки, могут отсутствовать). Файл начинается с маркера SOI и кончается маркером EOI. Между этими маркерами сжатый образ делится на кадры. В иерархической моде может быть несколько кадров, а во всех других модах имеется только один кадр. В каждом кадре информация об изображении хранится в одном или нескольких сканах; у кадра также имеется заголовок, перед которым могут находиться таблицы (которые, в свою очередь, могут иметь маркеры). За первым сканом может следовать сегмент DNL (define number of lines, определение числа строк), который начинается маркером DNL. В нем записано число строк сжатого образа, содержащегося в кадре. Скан начинается с таблицы (которая может отсутствовать), за которой идет заголовок скана, после которого размещается несколько сегментов энтропийного кода (ECS, entropy-coded segment), которые разделяются маркерами рестарта RST (restart). Каждый ECS состоит из одного или нескольких MCU, где MCU – это или одна единица данных, или три такие единицы.

### 3.7.8. JFIF

Как уже отмечалось, JPEG является методом сжатия графических данных, а не графическим форматом. Поэтому в нем не определяются такие специфические параметры изображения, как геометрический размер пикселя, световое пространство или чередование битовых строк. Все это делается в формате JFIF.

JFIF (Jpeg File Interchange Format, формат обмена файлами стандарта JPEG) является графическим форматом данных, который обеспечивает обмен сжатыми файлами JPEG между компьютерами. Основные особенности этого формата заключаются в использовании цветового пространства YCbCr из трех цветовых компонент цветных изображений (или одна компонента для полутоновых изображений), а также использование маркера для обозначения параметров, отсутствующих в стандарте JPEG, а именно, разрешение изображения, геометрический размер пикселя и некоторые другие параметры, специфические для конкретных приложений.

Маркер JFIF (называемый еще APP0) начинается строкой символов JFIF(NUL). Затем записаны информация о пикселях и другие



Значение	Имя	Описание
Недифференциальное, кодирование Хаффмана		
FFC0	SOF <sub>0</sub>	Базелина DCT
FFC1	SOF <sub>1</sub>	Расширенное последовательное DCT
FFC2	SOF <sub>2</sub>	Прогрессирующее DCT
FFC3	SOF <sub>3</sub>	Без потери (последовательное)
Дифференциальное, кодирование Хаффмана		
FFC5	SOF <sub>5</sub>	Дифференциальное последовательное DCT
FFC6	SOF <sub>6</sub>	Дифференциальное прогрессирующее DCT
FFC7	SOF <sub>7</sub>	Дифференциальное без потери (последов.)
Недифференциальное, арифметическое кодирование		
FFC8	JPG	Зарезервировано для расширения
FFC9	SOF <sub>9</sub>	Расширенное последовательное DCT
FFCA	SOF <sub>10</sub>	Прогрессирующее DCT
FFCB	SOF <sub>11</sub>	Без потери (последов.)
Дифференциальное, арифметическое кодирование		
FFCD	SOF <sub>13</sub>	Дифференциальное последовательное DCT
FFCE	SOF <sub>14</sub>	Дифференциальное прогрессирующее DCT
FFCF	SOF <sub>15</sub>	Дифференциальное без потери (последов.)
Таблицы для метода Хаффмана		
FFC4	DHT	Задание таблиц для метода Хаффмана
Спецификации для арифметического кодирования		
FFCC	DAC	Задание условий арифм. кодирования
Начало нового интервала		
FFD0-FFD7	RST <sub>m</sub>	Рестарт по модулю 8 счетчика <i>m</i>
Другие маркеры		
FFD8	SOI	Начало образа
FFD9	EOI	Конец образа
FFDA	SOS	Начало скана
FFDB	DQT	Задание таблиц квантования
FFDC	DNL	Задание числа строк
FFDD	DRI	Задание интервала рестарта
FFDE	DHP	Задание иерархической прогрессии
FFDF	EXP	Расширенная компонента ссылки
FFE0-FFEF	APP <sub>n</sub>	Зарезервировано для сегментов приложений
FFF0-FFF4	JPG <sub>n</sub>	Зарезервировано для расширения JPEG
FFFE	COM	Комментарий
Зарезервированные маркеры		
FF01	TEM	Для временного использования
FF02-FFBF	RES	Зарезервированы

Табл. 3.59. Маркеры JPEG.

спецификации. Далее могут следовать дополнительные сегменты, описывающие расширения JFIF, в которых записывается платформенно ориентированная информация об изображении.

Каждое расширение начинается строкой JFXX (NUL). Далее следует 1 байт, идентифицирующий конкретное расширение. Расширение может содержать данные, используемые конкретными приложениями. Тогда они могут начинаться другими строками или специальными идентифицирующими маркерами, отличными от JFIF и JFXX.

Формат первого сегмента маркера APP0 состоит из следующих полей:

1. Маркер APP0 (4 байта): FFD8FFE0.
2. Длина (2 байта): общая длина маркера, включая 2 байта поля «длина», но исключая сам маркер APP0 (поле 1).
3. Идентификатор (5 байтов): 4A46494600<sub>16</sub>. Это строка JFIF (NUL), идентифицирующая маркер APP0.
4. Версия (2 байта). Пример: 0102<sub>16</sub> обозначает версию 1.02.
5. Единица измерения (1 байт) плотности по координатам X и Y. Число 0 означает отсутствие этой единицы, поля Xdensity и Ydensity обозначают геометрический размер пикселя. Число 1 обозначает, что величины Xdensity и Ydensity измеряются в точках на дюйм, а 2 – в точках на сантиметр.
6. Xdensity (2 байта), Ydensity (2 байта): плотность пикселов по горизонтали и по вертикали (обе должны быть ненулевые).
7. Xthumbnail (1 байт), Ythumbnail (1 байт): Размер крохотного пикселя по горизонтали и вертикали.
8. (RGB)*n* (3*n* байт) упакованные (24-битовые) величины RGB раскраски крохотного пикселя. *n* = Xthumbnail × Ythumbnail.

Синтаксис сегмента расширения маркера APP0 имеет следующий вид.

1. Маркер APP0.
2. Длина (2 байта): общая длина маркера, включая 2 байта поля «длина», но исключая сам маркер APP0 (поле 1).
3. Идентификатор (5 байтов): 4A46585800<sub>16</sub>. Это строка JFXX (NUL), идентифицирующая расширение.
4. Код расширения (1 байт): 10<sub>16</sub> означает, что пиксель закодирован JPEG, 11<sub>16</sub> – размер пикселя 1 байт/пиксель (монохроматический), 13<sub>16</sub> – размер пикселя 3 байт/пиксель (цветной).
5. Данные расширения (переменные): это поле зависит от конкретного приложения.

## 3.8. JPEG-LS

Метод сжатия JPEG-LS использует коды Голомба, поэтому мы дадим краткое описание этих мало известных кодов.

### 3.8.1. Коды Голомба

Код Голомба неотрицательного целого числа  $n$  [Golomb 66] может быть эффективным кодом Хаффмана. Этот код зависит от выбора некоторого параметра  $b$ . Прежде всего необходимо вычислить две величины  $q = \lfloor \frac{n-1}{b} \rfloor$ ,  $r = n - qb - 1$  (где выражение  $\lfloor x \rfloor$  обозначает округление  $x$ ), а затем построить код из двух частей; первая часть – это число  $q$ , закодированное с помощью унарного кода (см. стр. 195), а вторая – двоичное выражение для  $r$ , состоящее из  $\lfloor \log_2 b \rfloor$  бит (для малых остатков) или из  $\lceil \log_2 b \rceil$  бит (для больших). Если взять  $b = 3$ , то три возможных остатка 0, 1 и 2 будут кодироваться как 0, 10 и 11. Выбрав  $b = 5$ , получаем 5 остатков от 0 до 4, которые кодируются как 00, 01, 100, 101 и 110. В табл. 3.60 приведены некоторые коды Голомба при  $b = 3$  и  $b = 5$ .

$n$	1	2	3	4	5	6	7	8	9	10
$b = 3$	0 0	0 10	0 11	10 0	10 10	10 11	110 0	110 10	110 11	1110 0
$b = 5$	0 00	0 01	0 100	0 101	10 110	10 00	10 01	10 100	10 101	110 110

Табл. 3.60. Некоторые коды Голомба при  $b = 3$  и  $b = 5$ .

Предположим, что входной поток данных состоит из целых чисел, причем вероятность числа  $n$  равна  $P(n) = (1 - p)^{n-1}p$ . Здесь  $p$  – некоторый параметр,  $0 \leq p \leq 1$ . Можно показать, что коды Голомба будут оптимальными кодами для этого потока данных, если  $b$  выбрать из условия

$$(1 - p)^b + (1 - p)^{b+1} \leq 1 < (1 - p)^{b-1} + (1 - p)^b.$$

Имея такие данные на входе, легко породить наилучшие коды переменной длины, не прибегая к алгоритму Хаффмана.

### 3.8.2. Основы метода JPEG-LS

Мы уже отмечали в § 3.7.6, что мода без потерь метода JPEG весьма неэффективна, и часто ее даже не включают в конкретные приложения, использующие JPEG. В результате ISO в коопeração с IEC разработали новый стандарт для сжатия без потерь

(и почти без потерь) непрерывно-тоновых изображений. Этот метод официально известен как рекомендация ISO/IEC CD 14495, но его принято называть JPEG-LS. Здесь рассматриваются основные принципы этого метода, который не является расширением или модификацией метода JPEG. Это совершенно новый метод, простой и быстрый. Он не использует ни DCT, ни арифметическое кодирование. Применяется слабое квантование и только в mode почти без потерь. JPEG-LS основан на идеях, развитых в [Weinberger и др. 96] для метода компрессии LOCO-I. JPEG-LS (1) изучает несколько предыдущих соседей текущего пикселя, (2) рассматривает их как контекст этого пикселя, (3) использует контекст для прогнозирования пикселя и для выбора распределения вероятностей из нескольких имеющихся, и (4) применяет это распределение для кодирования ошибки прогноза с помощью специального кода Голомба. Имеется также серийная мода, когда длина серии одинаковых пикселов кодируется подходящим образом.

Пиксели контекста  $a, b, c, d$ , используемые для прогнозирования текущего пикселя  $x$ , показаны на рис. 3.61. Кодер изучает пиксели контекста и устанавливает, в какой mode кодировать данный пиксель  $x$ , в *серийной* или в *регулярной*. Если контекст указывает, что пиксели  $y$  и  $z$ , следующие за  $x$ , скорее всего будут совпадать, то выбирается серийная мода. В противном случае, используется регулярная мода. Если включена опция «почти без потерь», то выбор моды делается несколько иначе. Если контекст предполагает, что следующие пиксели будут почти совпадать (в соответствии с параметром допустимого отклонения NEAR), то декодер выбирает серийную моду. Если нет, то берется регулярная мода. Дальнейшее кодирование зависит от выбранной моды.

$c$	$b$	$d$		
$a$	$x$	$y$	$z$	

Табл. 3.61. Контекст для прогноза  $x$ .

В регулярной mode кодер использует величины пикселов  $a, b$  и  $c$  для вычисления прогноза пикселя  $x$ . Этот прогноз вычитается из  $x$ , в результате чего получается *ошибка прогноза*, которая обозначается через  $Errval$ . Затем ошибка прогноза корректируется некоторым членом, зависящим от контекста (корректировка делается с целью

компенсирования систематического отклонения прогноза), и потом она кодируется с помощью кодов Голомба. Код Голомба зависит от всех четырех пикселов контекста, а также от ошибок прогноза этих же самых пикселов (эта информация хранится в массивах  $A$  и  $N$ , которые будут использоваться в § 3.8.3). При компрессии почти без потерь ошибка прогноза еще дополнительно квантуется перед кодированием.

В серийной моде кодер начинает с пикселя  $x$  и находит в этой строке наибольшую длину серии пикселов, совпадающих с контекстным пикселям  $a$ . Кодер не расширяет эту серию за пределы текущей строки. Поскольку все символы серии совпадают с  $a$  (а этот пиксель известен декодеру), то достаточно закодировать длину серии, что делается с помощью массива  $J$  из 32 элементов (см. § 3.8.3). (При сжатии почти без потерь, кодер выбирает серию пикселов, близких к  $a$  с помощью параметра NEAR.)

Декодер мало отличается от кодера, поэтому JPEG-LS можно считать почти симметричным методом сжатия. Сжатый файл состоит из сегментов данных (содержащих коды Голомба и длины серий), сегментов маркеров (с информацией, необходимой декодеру) и просто маркеров (в качестве которых используются некоторые зарезервированные маркеры JPEG). Маркером является байт из одних единиц, за которым следует специальный код, сигнализирующий о начале нового сегмента. Если за маркером следует байт, у которого старший бит равен 1, то этот байт является началом сегмента маркеров. В противном случае, начинается сегмент данных.

### 3.8.3. Кодер

Обычно, JPEG-LS используется как метод сжатия без потери информации. В этом случае восстановленный файл изображения идентичен исходному файлу. В моде почти без потерь исходный и реконструированный образ могут отличаться. Будем обозначать реконструированный пиксель  $Rp$ , а исходный пиксель –  $p$ .

При кодировании верхней строки контекстные пиксели  $c, b$  и  $d$  отсутствуют, поэтому их значения считаются нулевыми. Если текущий пиксель находится в начале или конце строки, то пиксели  $a, c$  или  $d$  не определены. В этом случае для  $a$  и  $d$  используется реконструированное значение  $Rb$  пикселя  $b$  (или нуль для верхней строки), а для  $c$  используется реконструированное значение  $a$  при кодировании первого символа предыдущей строки. Все это означает, что кодер должен выполнить часть работы декодера, реконструируя некоторые пиксели.

Первый шаг при определении контекста заключается в вычислении значений градиентов

$$D1 = Rd - Rb, \quad D2 = Rb - Rc, \quad D3 = Rc - Ra.$$

Если все эти величины равны нулю (или в моде почти без потерь их абсолютные значения не превосходят порога NEAR), то кодер переходит в серийную моду и ищет наибольшую длину серии пикселов, совпадающих с  $Ra$ . На шаге 2 кодер сравнивает три градиента  $Di$  с некоторыми параметрами и вычисляет зонные числа  $Qi$  по некоторым правилам (эти правила здесь не обсуждаются). Каждое зонное число  $Qi$  может принимать одно из 9 целых значений в интервале  $[-4, 4]$ , поэтому имеется всего  $9 \times 9 \times 9 = 729$  троек зонных чисел. На третьем шаге используются абсолютные значения произведений зонных чисел  $Qi$  (всего разных троек будет 365, так как одна из 729 величин равна нулю) для вычисления числа  $Q$  из интервала  $[0, 364]$ . Детали этих вычислений не предписываются стандартом JPEG-LS, и кодер может делать это по любому правилу. Число  $Q$  становится контекстом текущего пикселя  $x$ . Используются индексные массивы  $A$  и  $N$  из рис. 3.65.

После установления контекста  $Q$ , кодер прогнозирует пиксель  $x$  за два шага. На первом шаге вычисляется прогноз  $Px$  с помощью *правила края*, как показано на рис. 3.62. На втором шаге производится корректировка прогноза (см. рис. 3.63) с помощью числа SIGN (зависящего от трех зонных чисел  $Qi$ ), корректирующих величин  $C(Q)$  (выводимых из систематических смещений, и здесь не обсуждаемых) и параметра MAXVAL.

```
if (Rc>=max(Ra,Rb)) Px=min(Ra,Rb)      if (SIGN=+1) Px=Px+C(Q)
else
  if (Rc<=min(Ra,Rb)) Px=max(Ra,Rb)      else Px=Px-C(Q)
  else Px=Ra+Rb-Rc
  endif;
endif;
endif;
```

Рис. 3.62. Обнаружение угла.

Рис. 3.63. Корректировка прогноза.

Чтобы понять правило края, рассмотрим случай  $b \leq a$ . При этом условии правило края выбирает  $b$  в качестве прогноза  $x$  во многих случаях, когда вертикальный край изображения находится непосредственно слева от  $x$ . Аналогично, пиксель  $a$  выбирается в качестве прогноза  $x$  во многих случаях, когда горизонтальный край находится непосредственно над  $x$ . Если край не обнаруживается, то правило

края вычисляет прогноз в виде  $a + b - c$ , что имеет простую геометрическую интерпретацию. Если каждый пиксель является точкой трехмерного пространства, то прогноз  $a + b - c$  помещает  $Px$  на ту же плоскость, что и точки  $a$ ,  $b$  и  $c$ .

После того, как прогноз  $Px$  найден, кодер вычисляет ошибку прогноза  $Errval$  в виде разности  $x - Px$ , но меняет знак, если величина  $SIGN$  отрицательная.

В моде почти без потерь ошибка квантуется, и кодер использует это реконструированное значение  $Rx$  пикселя  $x$  так же, как это будет делать декодер. Основной шаг квантования заключается в вычислении

$$Errval \leftarrow \frac{Errval + \text{NEAR}}{2 \times \text{NEAR} + 1}.$$

При этом используется параметр  $\text{NEAR}$ , однако имеются некоторые детали, которые здесь не приводятся. Основной шаг реконструкции состоит в нахождении

$$Rx \leftarrow Px + SIGN \times Errval \times (2 \times \text{NEAR} + 1).$$

Ошибка прогноза (после возможного квантования) претерпевает сокращение области (здесь эта процедура опущена). Теперь она готова для главного этапа кодирования.

Коды Голомба были введены в § 3.8.1, где основной параметр был обозначен через  $b$ . В JPEG-LS этот параметр обозначается  $m$ . Если число  $m$  уже выбрано, то код Голомба неотрицательного целого числа  $n$  состоит из двух частей: унарного кода целой части числа  $n/m$  и двоичного представления  $n \bmod m$ . Этот код является идеальным для целых чисел, имеющих геометрическое распределение (то есть, когда вероятность числа  $n$  равна  $(1 - r)r^n$ ,  $0 < r < 1$ ). Для каждого геометрического распределения найдется такое число  $m$ , что код Голомба, построенный по  $m$ , имеет наименьшую возможную среднюю длину. Простейший случай, когда  $m$  равно степени 2 ( $m = 2^k$ ), приводит к простым операциям кодирования/декодирования. Код числа  $n$  состоит в этом случае из  $k$  младших разрядов числа  $n$ , перед которыми стоит унарный код числа, составленного из остальных старших разрядов числа  $n$ . Этот специальный код Голомба обозначается через  $G(k)$ .

Для примера вычислим код  $G(2)$  числа  $n = 19 = 10011_2$ . Поскольку  $k = 2$ , то  $m = 4$ . Начнем с двух младших разрядов,  $11_2$ , числа  $n$ . Они равны 3, что то же самое, что  $n \bmod m$  ( $3 = 19 \bmod 4$ ). Оставшиеся старшие разряды,  $100_2$  дадут число 4, которое равно целой

части  $n/m$  ( $19/4 = 4.75$ ). Унарный код 4 равен 00001, значит код  $G(2)$  числа  $n = 19$  равен 00001|11.

На практике всегда имеется конечное число неотрицательных целых чисел. Обозначим наибольшее число через  $I$ . Наибольшая длина  $G(0)$  равна  $I + 1$ , а поскольку  $I$  может быть велико, желательно лимитировать размер кода Голомба. Это делается с помощью специального кода Голомба  $LG(k, glimit)$ , который зависит от двух параметров  $k$  и  $glimit$ . Сначала следует сформировать число  $q$  из самых старших разрядов числа  $n$ . Если  $q < glimit - \lceil \log I \rceil - 1$ , то код  $LG(k, glimit)$  совпадает с кодом  $LG(k)$ . В противном случае, приготавливается унарный код числа  $glimit - \lceil \log I \rceil - 1$  (то есть,  $glimit - \lceil \log I \rceil - 1$  нулей, за которыми стоит единственная 1). Это действует как код esc, после которого стоит двоичный код  $n - 1$  из  $\lceil \log I \rceil$  бит.

Ошибки прогнозов не обязательно являются положительными числами. Они равны некоторым разностям, которые могут быть нулевыми или отрицательными. Однако коды Голомба были построены для положительных чисел. Поэтому перед кодированием отрицательные значения ошибок следует отразить на множество неотрицательных чисел. Для этого используется отображение

$$Merrval = \begin{cases} 2Errval, & Errval \geq 0, \\ 2|Errval| - 1, & Errval < 0. \end{cases} \quad (3.18)$$

Это отображение перемежает отрицательные и положительные величины в виде последовательности

$$0, -1, +1, -2, +2, -3, \dots$$

В табл. 3.64 перечислены некоторые ошибки прогноза, отображенные значения и их коды  $LG(2, 32)$  при условии, что алфавит имеет размер 256 (то есть,  $I = 255$  и  $\lceil \log I \rceil = 8$ ).

Теперь необходимо обсудить выбор параметра  $k$  для кодов Голомба. Это делается адаптивно. Параметр  $k$  зависит от контекста, и его значение обновляется каждый раз, когда обнаруживается пиксель с этим контекстом. Вычисление  $k$  можно выразить простой строкой на языке C++

```
for (k=0; (N[Q]<<k)<A[Q]; k++) ,
```

где  $A$  и  $N$  – массивы индексов от 0 до 364. В этой формуле используется контекст  $Q$  в качестве индекса двух массивов. В начале  $k$  инициализируется нулем, а затем совершается цикл. На каждой

итерации цикла элемент массива  $N[Q]$  сдвигается влево на  $k$  разрядов и сравнивается с  $A[Q]$ . Если сдвинутое значение  $N[Q]$  больше или равно  $A[Q]$ , то выбирается текущее значение  $k$ . В противном случае,  $k$  увеличивается на 1, и тест повторяется.

Ошибка прогноза	Отображенное значение	Код
0		0 1 00
-1		1 1 01
1		2 1 10
-2		3 1 11
2		4 01 00
-3		5 01 01
3		6 01 10
-4		7 01 11
4		8 001 00
-5		9 001 01
5		10 001 10
-6		11 001 11
6		12 0001 00
-7		13 0001 01
7		14 0001 10
-8		15 0001 11
8		16 00001 00
-9		17 00001 01
9		18 00001 10
-10		19 00001 11
10		20 000001 00
-11		21 000001 01
11		22 000001 10
-12		23 000001 11
12		24 0000001 00
...		
50	100	000000000000 000000000001 01100011

Табл. 3.64. Ошибки прогнозов, отображения и коды  $LG(2, 32)$ .

После нахождения числа  $k$ , ошибка прогноза  $Errval$  преобразуется с помощью уравнения (3.18) в число  $M_{Errval}$ , которое кодируется с помощью кода  $LG(k, LIMIT)$ . Число  $LIMIT$  является параметром. Обновление массивов  $A$  и  $N$  (вместе со вспомогательным массивом  $B$ ) показано на рис. 3.65 (параметр RESET контролируется пользователем).

Кодирование в серийной моде делается иначе. Кодер выбирает эту моду, когда обнаруживает последовательные пиксели  $x$ , чьи значения  $Ix$  совпадают и равны восстановленной величине  $Ra$  контекстного пикселя  $a$ . Для опции почти без потерь пиксели в серии должны иметь значения  $Ix$ , которые удовлетворяют неравенству

$$|Ix - Ra| \leq \text{NEAR}.$$

Серия не должна выходить за пределы текущей строки. Длина серии кодируется (сам пиксель кодировать не нужно, поскольку он равен  $Ra$ ), и если конец серии находится раньше конца строки, то после ее закодированной длины будет сразу записан код следующего пикселя (который прерывает серию). Две основные задачи кодера в этой моде состоят (1) в отслеживании серии и кодировании ее длины; (2) в кодировании пикселя, прервавшего серию. Отслеживание серии показано на рис. 3.66. Кодирование серий приведено на рис. 3.67 (для сегментов серий длины  $rm$ ) и на рис. 3.68 (для сегментов серий длины меньше, чем  $rm$ ). Рассмотрим некоторые детали.

```

B[Q]=B[Q]+Errval*(2*NEAR+1);
A[Q]=A[Q]+abs(Errval);
if(N[Q]=RESET) then
  A[Q]=A[Q]>>1;
  B[Q]=B[Q]>>1;
  N[Q]=N[Q]>>1;
endif;
N[Q]=N[Q]+1;
RUNval=Ra;
RUNcnt=0;
while(abs(Ix-RUNval)<=NEAR)
  RUNcnt=RUNcnt+1;
  Rx=RUNval;
  if(EOLine=1) break
  else GetNextSample()
  endif;
endwhile;

```

Рис. 3.65. Обновление массивов  $A$ ,  $B$  и  $N$ . Рис. 3.66. Отслеживание серий.

Кодер использует таблицу  $J$ , состоящую из 32 записей, обозначаемых  $rk$ .  $J$  инициализируется величинами

0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 6, 6, 7, 7, 8, 9, 10, 11, 12, 13, 14, 15.

Для каждого значения  $rk$  обозначим  $rm = 2^{rk}$ . Числа  $rm$  (их всего 32) называются порядком кода. Первые 4 величины  $rk$  имеют  $rm = 2^0 = 1$ . Для второй четверки  $rm = 2^1 = 2$ , а для следующей четверки  $rm = 2^2 = 4$ . Для последнего числа  $rm = 2^{15} = 32768$ . Кодер выполняет процедуру, указанную на рис. 3.66, для нахождения длины серии, которая сохраняется в переменной  $\text{RUNlen}$ . Затем эта переменная кодируется разбиением на слагаемые, величины которых равны последовательным числам  $rm$ . Например, если

`RUNlen`=6, то его представляют в виде  $6 = 1 + 1 + 1 + 1 + 2$  с помощью первых пяти чисел  $rm$ . Оно кодируется с помощью 5 бит. Запись производится инструкцией `AppendToBitfile(1,1)` (рис. 3.67). Каждый раз, когда пишется 1, соответствующая величина  $rm$  вычитается из `RUNlen`. Если `RUNlen` было равно в начале 6, то она последовательно уменьшается до 5, 4, 3, 2 и 0.

```

if(EOLine=0) then
  AppendToBitfile(0,1);
  AppendToBitfile
    (RUNcnt,J[RUNindex]);
  if(RUNindex>0)
    RUNindex=RUNindex-1;
    endif;
  else if(RUNcnt>0)
    AppendToBitfile(1,1);
endwhile;

```

Рис. 3.67. Кодирование серий (I).

Рис. 3.68. Кодирование серий (II).

Может конечно случиться, что длина `RUNlen` серии не равна целой сумме чисел  $rm$ . Например, `RUNlen` = 7. В этом случае в качестве кода записывается пять битов 1, за которыми следует *префиксный* бит и остаток от `RUNlen` (в нашем примере это 1), который записывается в файл в виде числа из  $rk$  бит (текущее значение  $rk$  из нашего примера равно 2). Эта последняя операция выполняется вызовом процедуры `AppendToBitfile(RUNcnt, J[RUNindex])` на рис. 3.68. Префиксным битом служит 0, если серия прерывается в строке другим пикселом. Если серия идет до конца строки, то префиксный бит равен 1.

Вторая основная задача кодера, состоящая в кодировании пикселя прерывания серии, делается аналогично кодированию текущего пикселя и здесь не обсуждается.

*Нет ничего туже точного образа размытой концепции.*

— Ансель Адамс

## ГЛАВА 4

# ВЕЙВЛЕТНЫЕ МЕТОДЫ

Во введении уже отмечалось, что методы сжатия, основанные на свойствах вейвлетов, используют довольно глубокие математические результаты. Это обстоятельство бросает определенный вызов как автору, так и читателю. Целью этой главы является представление основ теории вейвлетных преобразований (с минимумом дополнительных математических сведений) и ее приложений к задачам сжатия данных. Глава начинается с изложения последовательности шагов, состоящих из вычисления средних (полусумм) и полуразностей, которые преобразовывают одномерный массив исходных данных к виду, удобному для сжатия. Затем этот метод обобщается на двумерные массивы данных, что позволяет применять эти результаты к сжатию оцифрованных изображений. Рассмотренная последовательность трансформаций массива данных является простейшим примером *поддиапазонного преобразования*. Будет показано, что она идентична преобразованию Хаара, определенному в § 3.5.7.

В § 4.2.1 устанавливается связь преобразования Хаара с умножением матриц. Это проложит путь к введению в § 4.4 понятия банка фильтров. В § 4.3 излагаются некоторые дополнительные математические результаты, знакомство с которыми можно опустить при первом чтении. В этом параграфе обсуждается операция дискретной свертки и ее применение к поддиапазонным преобразованиям. За этим материалом следует § 4.6, в котором излагается дискретное вейвлетное преобразование (DWT, discrete wavelet transform). Глава заканчивается описанием метода сжатия SPIHT, основанного на вейвлетном преобразовании.

Перед тем как углубиться в различные детали следует ответить на часто задаваемый вопрос: «А почему здесь используется именно термин «вейвлет» (wavelet – это слово можно перевести как «маленькая волна» или «всплеск»)?» Эта глава не содержит полного ответа на этот вопрос, но рис. 4.14 и 4.34 дают некоторое интуитивное объяснение.

## 4.1. Вычисление средних и полуразностей

Мы начнем с одномерного массива данных, состоящего из  $N$  элементов. В принципе, этими элементами могут быть соседние пиксели изображения или последовательные звуковые фрагменты. Для простоты предположим, что число  $N$  равняется степени двойки. (Это будет предполагаться на протяжении всей главы, но в этом нет ограничения общности. Если длина  $N$  имеет другие делители, то можно просто удлинить массив, добавив в конце нули или повторив последний элемент нужное число раз. После декомпрессии, добавленные элементы просто удаляются.) Примером будет служить массив чисел  $(1, 2, 3, 4, 5, 6, 7, 8)$ . Сначала вычислим четыре средние величины  $(1+2)/2 = 3/2$ ,  $(3+4)/2 = 7/2$ ,  $(5+6)/2 = 11/2$  и  $(7+8)/2 = 15/2$ . Ясно, что знания этих четырех полусумм не достаточно для восстановления всего массива, поэтому мы еще вычислим четыре полуразности  $(1-2)/2 = -1/2$ ,  $(3-4)/2 = -1/2$  и  $(5-6)/2 = -1/2$ , которые будем называть коэффициентами деталей. Мы будем равноправно использовать термины «полуразность» и «деталь». Средние числа можно представлять себе крупномасштабным разрешением исходного образа, а детали необходимы для восстановления мелких подробностей или поправок. Если исходные данные коррелированы, то крупномасштабное разрешение повторит исходный образ, а детали будут малыми.

Массив  $(3/2, 7/2, 11/2, 15/2, -1/2, -1/2, -1/2, -1/2)$ , состоящий из четырех полусумм и четырех полуразностей, можно использовать для восстановления исходного массива чисел. Новый массив также состоит из восьми чисел, но его последние четыре компоненты, полуразности, имеют тенденцию уменьшаться, что хорошо для сжатия. Воодушевленные этим обстоятельством, повторим нашу процедуру применительно к четырем первым (крупным) компонентам нашего нового массива. Они преобразуются в два средних и в две полуразности. Остальные четыре компонента оставим без изменений. Получим массив  $(10/4, 26/4, -4/4, -4/4, -1/2, -1/2, -1/2, -1/2)$ . Следующая и последняя итерация нашего процесса преобразует первые две компоненты этого массива в одно среднее (которое, на самом деле, равно среднему значению всех 8 элементов исходного массива) и одну полуразность. В итоге получим массив чисел  $(36/8, -16/8, -4/4, -4/4, -1/2, -1/2, -1/2, -1/2)$ , который называется *вейвлетным преобразованием Хаара* исходного массива данных.

Из-за взятия полуразностей вейвлетное преобразование приво-

дит к уменьшению значений исходных пикселов, поэтому их будет легче сжать с помощью квантования и кодирования длинами серий (RLE), методом Хаффмана, или, быть может, иным подходящим способом (см. [Salomon 2000]). Сжатие с потерей части информации достигается, как обычно, с помощью квантования или простого удаления наименьших полуразностей (заменой их на нули).

Перед тем как двигаться дальше, интересно (и полезно) оценить сложность этого преобразования, то есть, число арифметических операций как функцию размера данных. В нашем примере требуется  $8 + 4 + 2 = 14$  операций (сложений и вычитаний). Это число можно выразить как  $14 = 2(8 - 1)$ . В общем случае, пусть имеется  $N = 2^n$  элементов массива. На первой итерации потребуется  $2^n$  операций, на второй –  $2^{n-1}$  операций, и так далее до последней итерации, в которой будет  $2^{n-(n-1)} = 2^1$  операции. Значит, суммарное число операций равно

$$\sum_{i=1}^n 2^i = \left( \sum_{i=0}^n 2^i \right) - 1 = \frac{2^{n+1} - 1}{2 - 1} - 1 = 2^{n+1} - 2 = 2(2^n - 1) = 2(N - 1).$$

Таким образом, для совершения преобразования Хаара массива из  $N$  элементов потребуется совершить  $2(N - 1)$  арифметических операций, то есть, сложность преобразования имеет порядок  $\mathcal{O}(N)$ . Результат просто замечательный.

Удобно с каждой итерацией процесса связать величину, называемую ее *разрешением*, которая равна числу оставшихся средних в конце итерации. Разрешение после каждой из трех описанных выше итераций равно  $4 (= 2^2)$ ,  $2 (= 2^1)$  и  $1 (= 2^0)$ . В § 4.2.1 показано, что каждую компоненту вейвлетного преобразования следует нормализовать с помощью деления на квадратный корень из разрешения соответствующей итерации (это относится к ортонормированному преобразованию Хаара, которое также обсуждается в § 4.2.1). Итак, наш пример вейвлетного преобразования дает массив

$$\left( \frac{36/8}{\sqrt{2^0}}, \frac{-16/8}{\sqrt{2^0}}, \frac{-4/4}{\sqrt{2^1}}, \frac{-4/4}{\sqrt{2^1}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}} \right).$$

Можно показать, что при использовании нормализованного вейвлетного преобразования наилучшим выбором сжатия с потерей будет игнорирование наименьших полуразностей. При этом достигается наименьшая потеря информации.

Две процедуры на рис. 4.1 иллюстрируют, как вычисляется нормализованное вейвлетное преобразование массива из  $n$  компонентов



( $n$  равно степени 2). Обратное преобразование, которое восстанавливает исходный массив, показано в двух процедурах на рис. 4.2.

```

procedure NWTcalc(a:array of real, n:int);
  comment: n - размер массива (степень 2)
  a:=a/sqrt(n) comment: разделить весь массив
  j:=n;
  while j>=2 do
    NWTstep(a, j);
    j:=j/2;
  endwhile;
end;

procedure NWTstep(a:array of real, j:int);
  for i=1 to j/2 do
    b[i]:=(a[2i-1]+a[2i])/sqrt(2);
    b[j/2+i]:=(a[2i-1]-a[2i])/sqrt(2);
  endfor;
  a:=b; comment: переместить весь массив
end;

```

Рис. 4.1. Вычисление нормализованного вейвлетного преобразования (псевдокод).

```

procedure NWTRreconst(a:array of real, n:int);
  j:=2;
  while j<=n do
    NWTRstep(a, j);
    j:=2j;
  endwhile;
  a:=a*sqrt(n); comment: умножение всего массива
end;

procedure NWTRstep(a:array of real, j:int);
  for i=1 to j/2 do
    b[2i-1]:=(a[i]+a[j/2+i])/sqrt(2);
    b[2i]:=(a[i]-a[j/2+i])/sqrt(2);
  endfor;
  a:=b; comment: переместить весь массив
end;

```

Рис. 4.2. Обратное нормализованного вейвлетного преобразования (псевдокод).

Эти процедуры, на первый взгляд, отличаются от взятия средних и полуразностей, которые обсуждались выше, поскольку происходит деление на  $\sqrt{2}$ , а не на 2. Первая процедура начинается делением всего массива на  $\sqrt{n}$ , а вторая делает обратное. Окончательный

результат, тем не менее, совпадает с массивом, указанным выше. Начиная с массива  $(1, 2, 3, 4, 5, 6, 7, 8)$ , получаем после трех итераций процедуры `NWTcalc` следующие массивы

$$\begin{aligned} & \left( \frac{3}{\sqrt{2^4}}, \frac{7}{\sqrt{2^4}}, \frac{11}{\sqrt{2^4}}, \frac{15}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}} \right), \\ & \left( \frac{10}{\sqrt{2^5}}, \frac{26}{\sqrt{2^5}}, \frac{-4}{\sqrt{2^5}}, \frac{-4}{\sqrt{2^5}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}} \right), \\ & \left( \frac{36}{\sqrt{2^6}}, \frac{-16}{\sqrt{2^6}}, \frac{-4}{\sqrt{2^5}}, \frac{-4}{\sqrt{2^5}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}} \right), \\ & \left( \frac{36/8}{\sqrt{2^0}}, \frac{-16/8}{\sqrt{2^0}}, \frac{-4/4}{\sqrt{2^1}}, \frac{-4/4}{\sqrt{2^1}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}} \right). \end{aligned}$$

#### 4.1.1. Обобщение на двумерный случай

Одномерное вейвлетное преобразование Хаара легко переносится на двумерный случай. Это обобщение весьма важно, поскольку преобразование будет применяться к изображениям, которые имеют два измерения. Здесь снова производится вычисление средних и полуразностей. Существует много обобщений этого преобразования. Все они обсуждаются в [Salomon, 2000]. Здесь мы остановимся на двух подходах, которые называются *стандартное разложение* и *пирамидальное разложение*.

Стандартное разложение (рис. 4.3) начинается вычислением вейвлетных преобразований всех строк изображения. К каждой строке применяются все итерации процесса, до тех пора, пока самый левый элемент каждой строки не станет равен среднему значению чисел этой строки, а все остальные элементы будут равны взвешенным разностям. Получится образ, в первом столбце которого стоит среднее столбцов исходного образа. После этого стандартный алгоритм производит вейвлетное преобразование каждого столбца. В результате получится двумерный массив, в котором самый левый верхний угловой элемент равен среднему всего исходного массива. Остальные элементы верхней строки будут равны средним взвешенным разностям, ниже стоят разности средних, а все остальные пиксели преобразуются в соответствующие разности.

Пирамидальное разложение вычисляет вейвлетное преобразование, применяя итерации поочередно к строкам и столбцам. На первом шаге вычисляются полусуммы и полуразности для всех строк (только одна итерация, а не все вейвлетное преобразование). Это

действие производит средние в левой половине матрицы и полуразности – в правой половине. На втором шаге вычисляются полу суммы и полуразности для всех столбцов получившейся матрицы.

```

procedure StdCalc(a:array of real, n:int);
  comment: массив размера nn (n = степень 2)
  for r=1 to n do NWTcalc(row r of a, n);
  endfor;
  for c=n to 1 do comment: обратный цикл
    NWTcalc(col c of a, n);
  endfor;
end;

procedure StdReconst(a:array of real, n:int);
  for c=n to 1 do comment: обратный цикл
    NWTreconst(col c of a, n);
  endfor;
  for r=1 to n do
    NWTreconst(row r of a, n);
  endfor;
end;

```

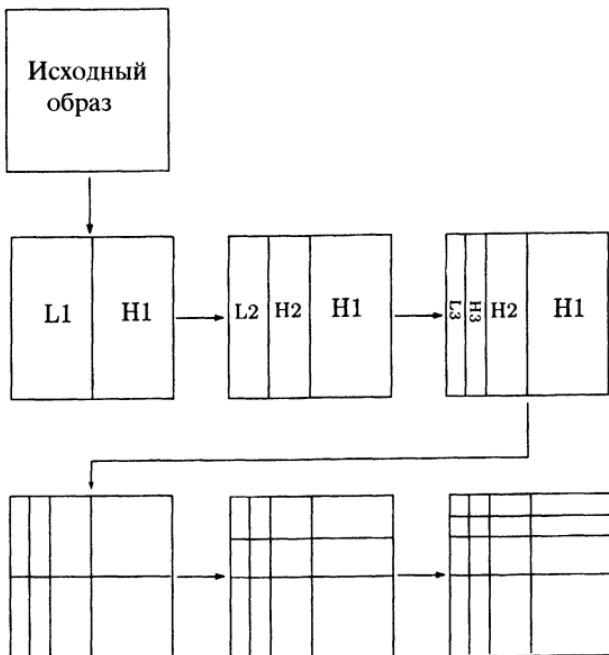


Рис. 4.3. Стандартное вейвлетное разложение.

В итоге в левом верхнем квадранте будут стоять средние четырех квадрантов исходного образа, а в остальных квадрантах будут находиться соответствующие полуразности. Шаги 3 и 4 оперируют со строками и столбцами, в результате чего средние величины будут сконцентрированы в левой верхней подматрице (одной шестнадцатой всей исходной таблицы). Эти пары шагов применяются к все более и более маленьким подматрицам, до тех пор пока в верхнем левом углу не будет стоять среднее всей исходной матрицы, а все остальные пиксели преобразуются в разности в соответствии с ходом алгоритма. Весь процесс показан на рис. 4.5.

Преобразования, описанные в § 3.5, являются ортогональными. Они преобразуют пиксели изображения во множество чисел, из которых некоторые числа будут большими, а остальные – маленькими. Вейвлетные преобразования, подобные преобразованию Хаара, работают иначе, они являются *поддиапазонными*. Они разбивают образ на подобласти, из которых одна область содержит большие числа (средние значения в случае преобразования Хаара), а другие области состоят из малых чисел (разностей в нашем случае). Однако эти области, называемые *поддиапазонами*, не просто являются семействами больших и малых чисел. Они отражают различные геометрические свойства трансформируемого образа. Чтобы пояснить эту особенность, изучим маленькое равномерное изображение, содержащее вертикальную и горизонтальную линию. На рис. 4.4а показан такой образ размера  $8 \times 8$ , в котором все пиксели равны 12 за исключением одной вертикальной строки с пикселями, равными 14, и одной горизонтальной строки, где пиксели равны 16.

12 12 12 12 14 12 12 12	12 12 13 12	0 0 2 0	12 12 13 12	0 0 2 0
12 12 12 12 14 12 12 12	12 12 13 12	0 0 2 0	12 12 13 12	0 0 2 0
12 12 12 12 14 12 12 12	12 12 13 12	0 0 2 0	14 14 14 14	0 0 0 0
12 12 12 12 14 12 12 12	12 12 13 12	0 0 2 0	12 12 13 12	0 0 2 0
12 12 12 12 14 12 12 12	12 12 13 12	0 0 2 0	0 0 0 0	0 0 0 0
16 16 16 16 14 16 16 16	16 16 15 16	0 0 2 0	0 0 0 0	0 0 0 0
12 12 12 12 14 12 12 12	12 12 13 12	0 0 2 0	4 4 2 4	0 0 4 0
12 12 12 12 14 12 12 12	12 12 13 12	0 0 2 0	0 0 0 0	0 0 0 0

(a)

(b)

(c)

Рис. 4.4. Образ  $8 \times 8$  и его поддиапазонное разложение.

На рис. 4.4b приведен результат применения одного шага преобразования Хаара ко всем строкам матрицы. Правая часть преобразованной матрицы (содержащая разности) в основном состоит



из нулей. В этом отражается равномерность образа. Однако след от вертикальной линии вполне заметен (подчеркнутые числа обозначают отрицательные разности).

```

procedure NStdCalc(a:array of real, n:int);
a:=a/ $\sqrt{n}$  comment: деление всего массива
j:=n;
while j $\geq 2$  do
  for r=1 to j do NWTstep(row r of a, j);
  endfor;
  for c=j to 1 do comment: обратный цикл
    NWTstep(col c of a, j);
  endfor; j:=j/2;
endwhile;
end;

procedure NStdReconst(a:array of real, n:int);
j:=2;
while j $\leq n$  do
  for c=j to 1 do comment: обратный цикл
    NWTRestep(col c of a, j);
  endfor;
  for r=1 to j do
    NWTRestep(row r of a, j);
  endfor; j:=2j;
endwhile
a:=a $\sqrt{n}$ ; comment: умножение всего массива
end;

```

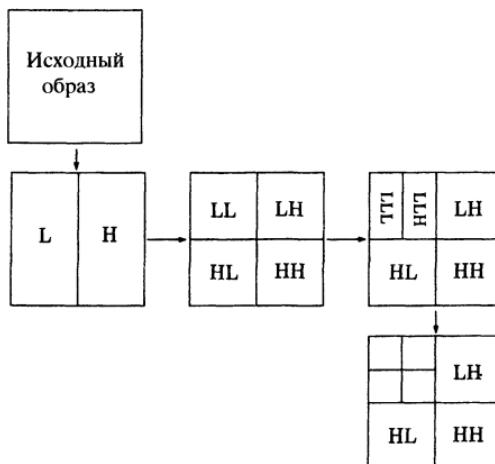


Рис. 4.5. Пирамидальное разложение образа.



На рис. 4.4с изображен результат применения того же преобразования к столбцам матрицы (b). Верхний правый поддиапазон содержит след от вертикальной линии, а в нижнем левом поддиапазоне отчетливо виден след от горизонтальной линии. Обозначим эти поддиапазоны HL и LH, соответственно (см. рис. 4.35, хотя имеется некоторое разнотечение в использовании обозначений разными авторами). Нижний правый поддиапазон обозначим HH, на котором отражаются диагональные особенности образа (в нашем случае отсутствующие). Самым интересным остается верхний левый поддиапазон, целиком состоящий из средних величин (он обозначается LL). Этот квадрант, являющийся уменьшенной копией исходного образа с пониженным качеством, содержит следы от обеих линий.

Рис. 4.6 иллюстрирует влияние диагональных особенностей образа на поддиапазон HH. На рис. 4.6а показана матрица равномерного образа с диагональной полосой чуть выше главной диагонали матрицы. На рис. 4.6б,с даны результаты двух первых шагов пирамидального преобразования. Видно, что преобразованные коэффициенты левого нижнего поддиапазона (HH) отражают диагональные особенности исходного образа, лежащие именно выше главной диагонали матрицы. Ясно также, что верхний левый поддиапазон (LL) является копией исходного изображения, но с более низким разрешением.

12 16 12 12 12 12 12 12 12	14 12 12 12 12   4 0 0 0	13 13 12 12   2 2 0 0
12 12 16 12 12 12 12 12 12	12 14 12 12   0 4 0 0	12 13 13 12   0 2 2 0
12 12 12 16 12 12 12 12 12	12 14 12 12   0 4 0 0	12 12 13 13   0 0 2 2
12 12 12 12 16 12 12 12 12	12 12 14 12   0 0 4 0	12 12 12 13   0 0 0 2
12 12 12 12 12 16 12 12 12	12 12 14 12   0 0 4 0	2 2 0 0   4 4 0 0
12 12 12 12 12 12 16 12	12 12 12 14   0 0 0 4	0 2 2 0   0 4 4 0
12 12 12 12 12 12 12 16	12 12 12 14   0 0 0 4	0 0 2 2   0 0 4 4
12 12 12 12 12 12 12 12	12 12 12 12   0 0 0 0	0 0 0 2   0 0 0 4

(a)

(b)

(c)

Рис. 4.6. Поддиапазонное разложение диагональной линии.

На рис. 4.35 изображены четыре уровня поддиапазонов, где первый уровень содержит подробные детали исходного изображения (который называется уровнем высокочастотных коэффициентов высокого разрешения), а верхний, четвертый уровень, содержит грубые детали изображения (низкочастотные коэффициенты низкого разрешения). Очевидно, коэффициенты четвертого уровня можно квантовать достаточно грубо без существенных потерь качества

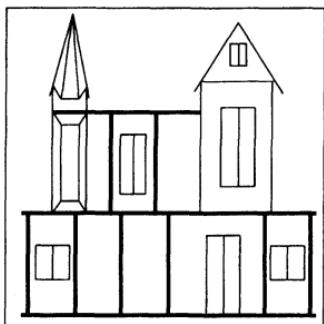


изображения, в то время как высокочастотные коэффициенты следует квантовать очень слабо или совсем не трогать. Структура поддиапазонов – вот базис любого метода сжатия, основанного на вейвлетных преобразованиях.

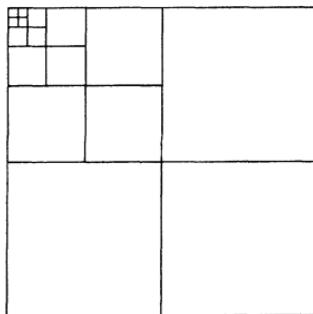
На рис. 4.7 показан типичный результат пирамидального вейвлетного преобразования. Исходное изображение дано на рис. 4.7а. На рис. 4.7с показана общая схема пирамидального разложения этого образа. Рисунок выбран состоящим в основном из горизонтальных, вертикальных и наклонных линий, чтобы были заметны особенности пирамидального преобразования. Четыре квадранта на рис. 4.7с передают уменьшенную копию этого изображения. Верхний левый поддиапазон, содержащий средние значения, подобен исходному образу, а три остальных квадранта (поддиапазона) показывают детали изображения. Верхний правый поддиапазон отражает вертикальные детали изображения, нижний левый – горизонтальные, а нижний правый содержит детали наклонных линий. На рис. 4.7б показана последовательность итераций этого преобразования. Все изображение трансформируется в последовательность поддиапазонов, отражающих особенности по горизонтали, вертикали и диагонали, а самый верхний левый квадратик, содержащий усредненное изображение, стягивается в один единственный пиксель.

Независимо от метода (стандартного или пирамидального) в результате преобразования получается одно большое среднее число в верхнем левом углу матрицы образа, а все остальные элементы матрицы являются малыми числами, разностями или средними разностей. Теперь этот массив чисел можно подвергнуть сжатию с помощью подходящей комбинации методов RLE, кодирования Хаффмана или других известных алгоритмов (см. [Salomon 2000]). Если допустима частичная потеря информации, то наименьшие разности можно дополнительно квантовать или просто обнулить. Этот шаг даст длинные серии нулей, к которым метод RLE можно применить с еще большей эффективностью.

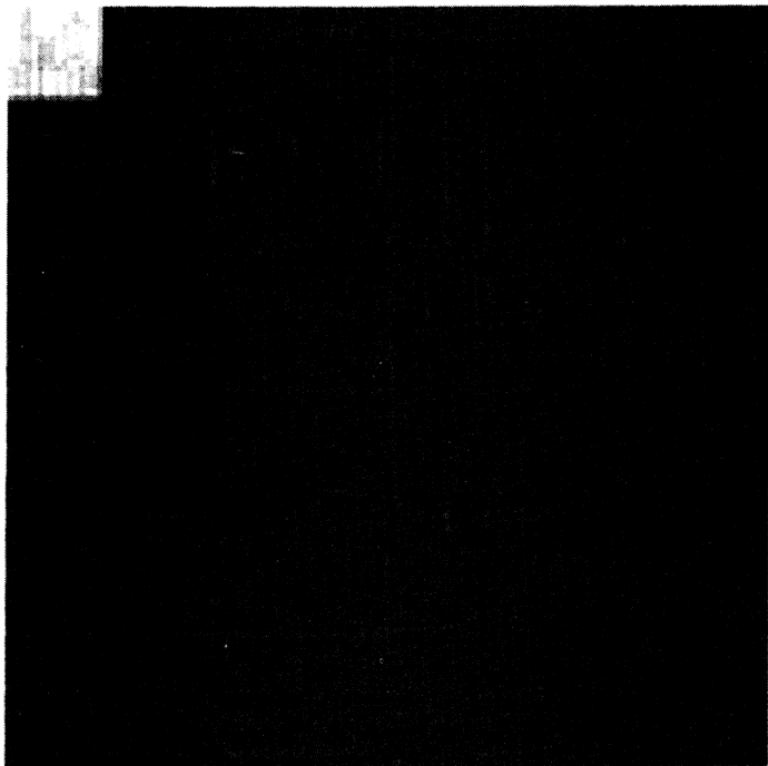
**Цветные изображения.** До этого момента предполагалось, что пиксели образа состоят из одиночных чисел (то есть, рассматривалось изображение из одной компоненты, представляющей различные оттенки одного цвета, обычно, серого). Любой метод сжатия таких изображений легко переносится на цветные образы, состоящие из трех компонентов. Для этого достаточно разделить образ на три подобраза и каждый независимо сжать. Если разрешается потеря информации, то имеет смысл сначала сделать преобразование исходного цветового пространства, которое обычно явля-



(a)



(c)



(b)

Рис. 4.7. Пример пирамидального разложения образа.



ется пространством RGB, в пространство YIQ. В новом цветовом представлении компонента Y – это *светимость*, а компоненты I и Q отвечают за *цветность* (см. [Salomon 99]). Преимущество этого представления состоит в том, что глаз человека имеет наибольшую чувствительность к изменениям светимости (Y), а наименьшую – к изменениям компоненты цветности Q. Поэтому метод с потерей данных должен отдельно сжимать компоненту Y почти без потерь, удалять часть информации из компоненты I, а из компоненты Q удалять еще больше данных. Тогда удастся добиться значительного сжатия изображения без заметных для глаза потерь качества и мелких деталей. В § 3.7.1 приведены более подробные сведения о цветовых пространствах и компонентах светимости и цветности.

Интересно отметить, что американский стандарт для передачи цветного телевизионного сигнала также учитывает преимущества представления YIQ. В общей полосе частот сигнала компонента Y занимает 4 MHz, на компоненту I приходится 1.5 MHz, а компоненте Q отведено всего 0.6 MHz.

#### 4.1.2. Свойства преобразования Хаара

Примеры из этого параграфа иллюстрируют некоторые важные свойства вейвлетного преобразования Хаара, а также общих вейвлетных преобразований. На рис. 4.8 показан высоко коррелированный образ размера  $8 \times 8$  и его преобразование Хаара. Даны числовые значение преобразованных коэффициентов и их графическое представление в виде квадратиков различных серых оттенков. Из-за высокой степени корреляции исходных пикселов, вейвлетные коэффициенты в основном малы по абсолютному значению и многие из них равны нулю.

**Замечание.** При первом взгляде на рис. 4.8 последнее утверждение кажется ложным. Приведенные коэффициенты вейвлетного преобразования достаточно велики по сравнению с исходными значениями пикселов. Нам известно, что верхний левый элемент матрицы коэффициентов преобразования Хаара должен быть равен среднему значению всех пикселов образа. Поскольку эти пиксели имеют примерно равномерное распределение на интервале [0,255], то это среднее должно быть около 128 (на самом деле, точное значение равно 131.375). А в приведенной таблице это число равно 1051 (что равно  $131.375 \times 8$ ). Причина заключается в том, что программа, выполнявшая эти вычисления использовала  $\sqrt{2}$  вместо 2 (см. функцию `individ(n)` на рис. 4.12).

При дискретном вейвлетном преобразовании большинство получающихся коэффициентов (разностей) отвечают за детали изображения. Детали нижнего уровня представляют мелкие особенности исходного образа. При перемещении на более высокие поддиапазонные уровни обнаруживаются более грубые детали данного изображения. Рис. 4.9а поясняет эту концепцию. Показано изображение, равномерно гладкое слева, у которого наблюдается некоторая «шероховатость» справа (то есть, соседние пиксели начинают различаться). На рисунке (б) дано графическое представление преобразования Хаара этого образа. Поддиапазоны низкого уровня (отвечающий точным деталям) имеют ненулевые коэффициенты справа, там, где наблюдается «шероховатость» изображения. Поддиапазоны высокого уровня (грубые детали) выглядят похоже, и их коэффициенты также имеют ненулевые коэффициенты слева, поскольку изображение не полностью белое слева.

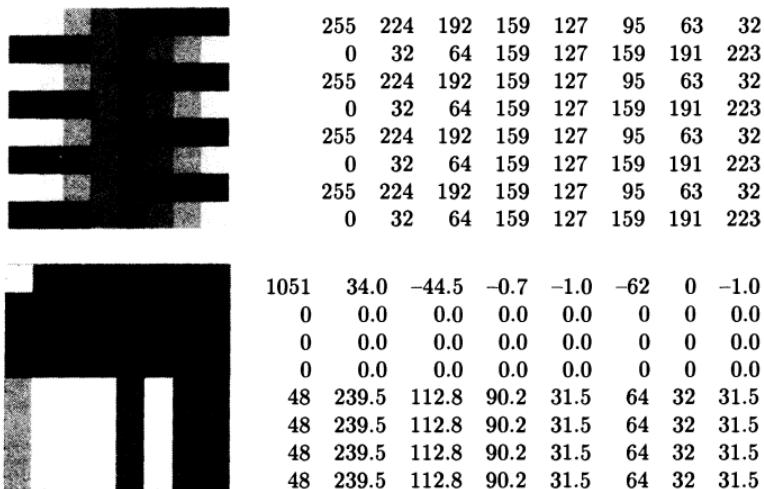


Рис. 4.8. Образ  $8 \times 8$ , реконструированный на рис. 4.11 и его преобразование Хаара.

Преобразование Хаара является простейшим вейвлетным преобразованием, однако уже на этом простом примере обнаруживаются замечательные свойства этих преобразований. Оказывается, что поддиапазоны низкого уровня состоят из несущественных особенностей изображения, поэтому их можно смело квантовать и даже отбрасывать. Это дает хорошее сжатие с частичной потерей информации, которая, однако не отразится на качестве восстановленно-

го образа. Реконструкция образа делается очень быстро при минимальной потери качества. На рис. 4.11 показаны три реконструкции исходного образа размера  $8 \times 8$  из рис. 4.8. Они получены с помощью, соответственно, 32, 13 и 5 вейвлетных коэффициентов.

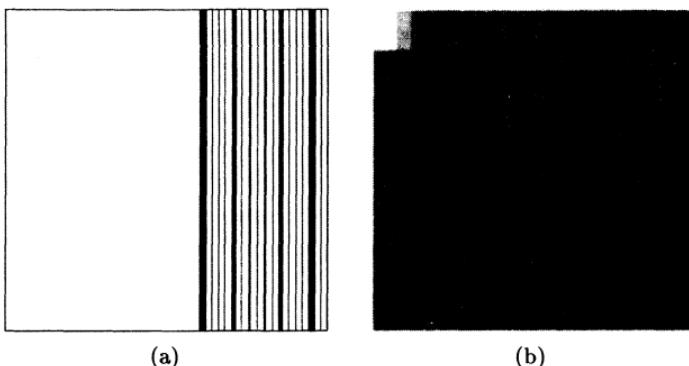


Рис. 4.9. (а) Образ  $128 \times 128$  пикселов с «шероховатостью» справа.  
 (б) Его преобразование.

Рис. 4.10 является аналогичным примером. В части (а) дан двухуровневый черно-белый образ, полностью восстановленный с помощью всего 4% коэффициентов (653 коэффициентов из  $128 \times 128$  показаны на рис. (б)).

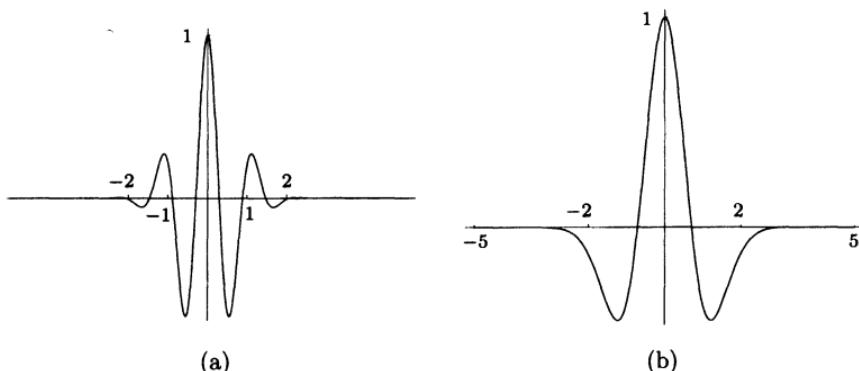
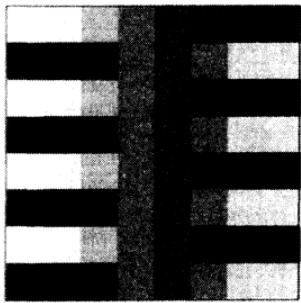
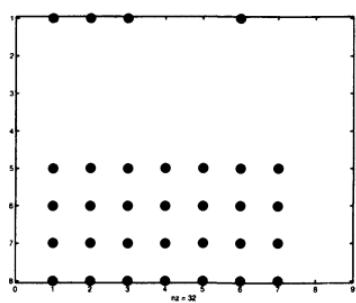


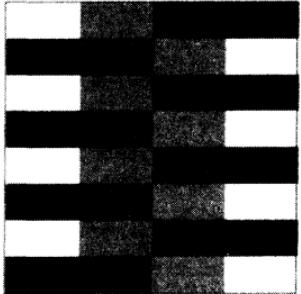
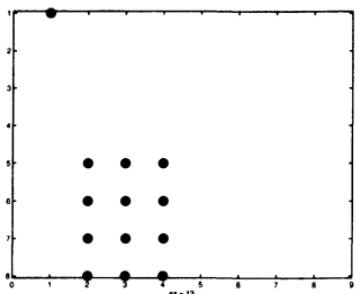
Рис. 4.10. Восстановление простого образа  $128 \times 128$  пикселов из 4% его коэффициентов.

Мой личный опыт подсказывает, что лучший способ понять вейвлетные преобразования – это как следует поэкспериментировать с изображениями с различными корреляциями и «шероховатостями»

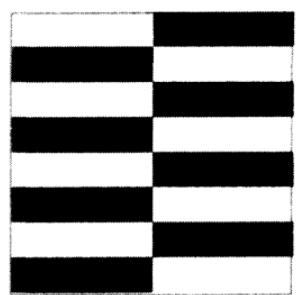
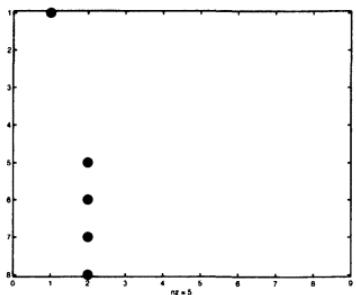
пикселов. Подходящее программное обеспечение позволит легко вводить изображения в компьютер и проверять различные особенности дискретных вейвлетных преобразований для разных параметров. В помощь читателю на рис. 4.12 приведена программа пакета Matlab, которая считывает файл с изображением, вычисляет его преобразование Хаара, отбрасывает заданный процент наименьших коэффициентов преобразования и делает обратное преобразование для восстановления изображения.



(a)



(b)



(c)

Рис. 4.11. Три реконструкции образа из  $8 \times 8$  пикселов.



Вейвлетное сжатие изображений пренебрегает частью коэффициентов, поэтому следует определить величину, называемую *коэффициентом прореживания*, равную доли отбрасываемых коэффициентов. Она определяется, как отношение числа ненулевых вейвлетных коэффициентов к числу коэффициентов, оставшихся после отбрасывания. Чем выше коэффициент прореживания, тем меньше вейвлетных коэффициентов было оставлено. Высокий коэффициент прореживания означает сильное сжатие, но с возможным ухудшением качества изображения. Коэффициент прореживания отдаленно напоминает фактор сжатия, который был определен во введении к этой книге.

В строке `filename='lena128'; dim=128;` записаны имя файла с изображением и его размер. Файл, использованный автором, был представлен в «сыром» виде. Он состоял из пикселов различных оттенков серого размера в 1 байт каждый. В файле не было никакого заголовка, не было даже разрешения образа (то есть, числа строк и столбцов). Тем не менее, Matlab читает любые файлы. Образ предполагался квадратным, а параметр `dim` должен быть степенью 2. Присваивание `thresh =` задает процент коэффициентов, которые следует удалить. Это позволяет легко экспериментировать с вейвлетным сжатием.

Файл `harmatt.m` содержит две функции, которые вычисляют преобразование Хаара в матричной форме (см. § 4.2.1).

(Техническое замечание: файлы Matlab с расширением `.m` могут содержать или последовательность команд, или функции, но не одновременно команды и функции. Однако допускается несколько функций в одном файле, при условии, что только самая верхняя функция будет вызываться извне этого файла. Все остальные функции должны вызываться только из этого файла. В нашем случае функция `harmatt(dim)` вызывает функцию `individ(n)`.)

**Пример:** Программа из рис. 4.12 используется для вычисления преобразования Хаара изображения «Lena». Затем это изображение реконструируется три раза с отбрасыванием все большего и большего числа коэффициентов деталей. На рис. 4.13 показаны результаты восстановления исходного изображения с помощью 3277, 1639 и 820 вейвлетных коэффициентов, соответственно. Несмотря на сильное прореживание коэффициентов, обнаруживается слабая потеря качества картинки. Полное число вейвлетных коэффициентов, конечно, равно разрешению образа, то есть,  $128 \times 128 = 16384$ .

```

clear; % главная программа
filename='lena128'; dim=128;
fid=fopen(filename,'r');
if fid==-1 disp('file not found')
else img=fread(fid,[dim,dim] ); fclose(fid);
end
thresh=0.0; % процент отбрасываемых коэффициентов
figure(1), imagesc(img), colormap(gray), axis off, axis square
w=harmatt(dim); % вычисление матрицы Хаара
timg=w*img*w'; % прямое преобразование Хаара
tsort=sort(abs(timg(:)));
tthresh=tsort(floor(max(thresh*dim*dim,1)));
cim=timg.*((abs(timg) > tthresh);
[i,j,s] =find(cim);
dimg=sparse(i,j,s,dim,dim);
% figure(2) показывает оставшиеся коэффициенты преобразования
% figure(2), spy(dimg), colormap(gray), axis square
figure(2), image(dimg), colormap(gray), axis square
cimg=full(w*sparse(dimg)*w); density = nnz(dimg);
disp(['num2str(100*thresh) '% отброшенных коэффициентов.'])
disp(['num2str(density) ' оставленных коэффициентов ' ...
num2str(dim) 'x' num2str(dim) '.'])
figure(3), imagesc(cimg), colormap(gray), axis off, axis square

```

Файл `harmatt.m` с двумя функциями

```

function x = harmatt(dim)
num=log2(dim);
p = sparse(eye(dim)); q = p;
i=1;
while i<=dim/2;
    q(1:2*i,1:2*i) = sparse(individ(2*i));
    p=p*q; i=2*i;
end
x=sparse(p);

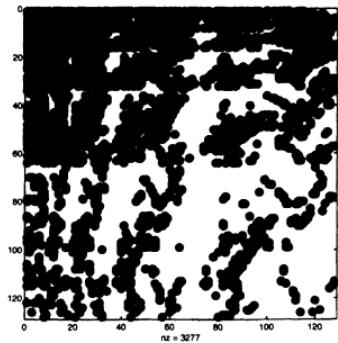
function f=individ(n)
x=[1, 1] /sqrt(2);
y=[1,-1] /sqrt(2);
while min(size(x)) < n/2
    x=[x, zeros(min(size(x)),max(size(x))); ...
    zeros(min(size(x)),max(size(x))), x];
end
while min(size(y)) < n/2
    y=[y, zeros(min(size(y)),max(size(y))); ...
    zeros(min(size(y)),max(size(y))), y];
end
f=[x;y];

```

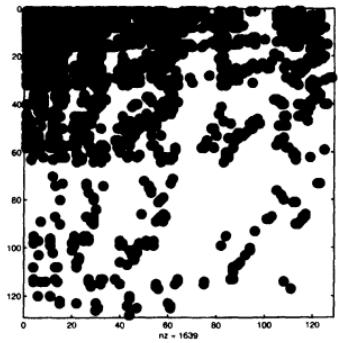
Рис. 4.12. Программа для вычисления преобразования Хаара (Matlab).



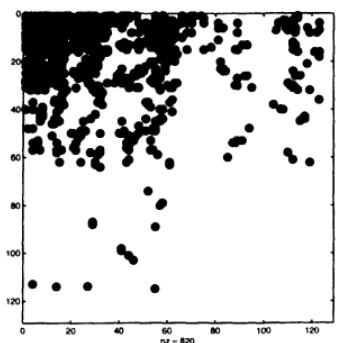
Использование только 820 коэффициентов соответствует отбрасыванию 95% наименьших из них (заметим однако, что часть коэффициентов сразу равнялось нулю, поэтому, реальная потеря данных будет меньше 95%).



(a)



(b)



(c)

Рис. 4.13. Три реконструкции образа «Lena» из  $128 \times 128$  пикселов.

## 4.2. Преобразование Хаара

Преобразование Хаара использует функцию шкалы  $\phi(t)$  и вейвлет  $\psi(t)$ , которые показаны на рис. 4.14а, для представления широкого класса функций. Это представление имеет вид бесконечной суммы

$$f(t) = \sum_{k=-\infty}^{\infty} c_k \phi(t-k) + \sum_{k=-\infty}^{\infty} \sum_{j=0}^{\infty} d_{j,k} \psi(2^j t - k),$$

где  $c_k$  и  $d_{j,k}$  – коэффициенты, которые необходимо определить.

Базисная функция шкалы  $\phi(t)$  является единичным импульсом

$$\phi(t) = \begin{cases} 1, & 0 \leq t < 1, \\ 0, & \text{иначе.} \end{cases}$$

Функция  $\phi(t-k)$  является копией функции  $\phi(t)$ , сдвинутой вправо на число  $k$ . Аналогично, функция  $\phi(2t-k)$  получается из функции  $\phi(t-k)$  сжатием аргумента в два раза (это еще можно назвать уменьшением масштаба). Сдвинутые функции используются для аппроксимации функции  $f(t)$  при различных моментах времени, а функции с разными масштабами нужны для аппроксимации функции  $f(t)$  при более высоком разрешении. На рис. 4.14б приведены графики функций  $\phi(2^j t - k)$  при  $j = 0, 1, 2, 3$  и при  $k = 0, 1, \dots, 7$ .

Базисный вейвлет Хаара  $\psi(t)$  является ступенчатой функцией

$$\psi(t) = \begin{cases} 1, & 0 \leq t < 0.5, \\ -1, & 0.5 \leq t < 1. \end{cases}$$

Из этого определения мы заключаем, что общий вейвлет  $\psi(2^j t - k)$  получается из  $\psi(t)$  сдвигом вправо на  $k$  единиц и сменой масштаба в  $2^j$  раз. Четыре вейвлета  $\psi(2^j t - k)$  при  $k = 0, 1, 2$  и  $3$  показаны на рис. 4.14с.

Обе функции  $\phi(2^j t - k)$  и  $\psi(2^j t - k)$  не равны нулю на интервале ширины  $1/2^j$ . Этот интервал называется *носителем* этих функций. Поскольку длина этого интервала стремится к нулю, когда  $j$  стремится к бесконечности, мы будем говорить, что функции имеют *компактный носитель*.

Проиллюстрируем основное преобразование с помощью простой ступенчатой функции

$$f(t) = \begin{cases} 5, & 0 \leq t < 0.5, \\ 3, & 0.5 \leq t < 1. \end{cases}$$

Легко видеть, что  $f(t) = 4\phi(t) + \psi(t)$ . Мы скажем, что исходные ступени (5, 3) были преобразованы в представление, имеющее среднее (низкое разрешение) 4 единицы и детали (высокое разрешение) 1 единица. Если воспользоваться матричным представлением, то это можно записать как  $(5, 3)\mathbf{A}_2 = (4, 1)$ , где  $\mathbf{A}_2$  – матрица преобразования Хаара порядка 2 (см. уравнение (3.16)).

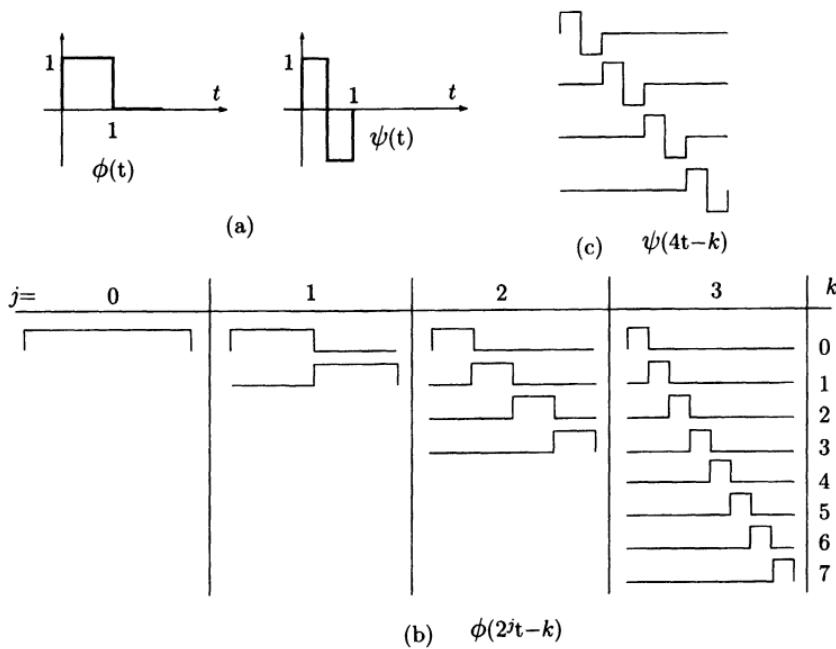


Рис. 4.14. Базисная шкала Хаара и вейвлетные функции.

#### 4.2.1. Матричная форма

В основе преобразования Хаара лежит вычисление средних и разностей. Оказывается, что эти операции можно легко выразить с помощью умножений соответствующих матриц (см. [Mulcahy 96] и [Mulcahy 97]). Для примера рассмотрим верхнюю строку простого изображения размера  $8 \times 8$  из рис. 4.8. Каждый, кто немного знаком с операциями над матрицами, легко построит матрицу, которая при умножении на некоторый вектор дает другой вектор, состоящий из четырех полусумм и четырех полуразностей элементов

этого вектора. Обозначим эту матрицу  $\mathbf{A}_1$ . Ее произведение на вектор рассматриваемого примера (верхняя строка матрицы на рис. 4.8) равно (239.5, 175.5, 111.0, 47.5, 15.5, 16.5, 16.0, 15.5). Это видно из уравнения (4.1). Аналогично, матрицы  $\mathbf{A}_2$  и  $\mathbf{A}_3$  производят, соответственно, второй и третий шаг преобразования. Его результат показан в формуле (4.2).

$$\mathbf{A}_1 = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & -\frac{1}{2} \end{pmatrix}, \quad A_1 \begin{pmatrix} 255 \\ 224 \\ 192 \\ 159 \\ 127 \\ 95 \\ 63 \\ 32 \end{pmatrix} = \begin{pmatrix} 239.5 \\ 175.5 \\ 111.0 \\ 47.5 \\ 15.5 \\ 16.5 \\ 16.0 \\ 15.5 \end{pmatrix}, \quad (4.1)$$

$$\mathbf{A}_2 = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \quad \mathbf{A}_3 = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

$$\mathbf{A}_2 \begin{pmatrix} 239.5 \\ 175.5 \\ 111.0 \\ 47.5 \\ 15.5 \\ 16.5 \\ 16.0 \\ 15.5 \end{pmatrix} = \begin{pmatrix} 207.5 \\ 79.25 \\ 32.0 \\ 31.75 \\ 15.5 \\ 16.5 \\ 16.0 \\ 15.5 \end{pmatrix}, \quad \mathbf{A}_2 \begin{pmatrix} 207.5 \\ 79.25 \\ 32.0 \\ 31.75 \\ 15.5 \\ 16.5 \\ 16.0 \\ 15.5 \end{pmatrix} = \begin{pmatrix} 143.375 \\ 64.125 \\ 32.0 \\ 31.75 \\ 15.5 \\ 16.5 \\ 16.0 \\ 15.5 \end{pmatrix}. \quad (4.2)$$

Вместо того, чтобы вычислять средние и разности строк, можно построить матрицы  $\mathbf{A}_1$ ,  $\mathbf{A}_2$  и  $\mathbf{A}_3$ , перемножить их, получить матрицу

$$\mathbf{W} = \mathbf{A}_1 \mathbf{A}_2 \mathbf{A}_3,$$

а затем применить ее к вектору  $\mathbf{I}$ :

$$W \begin{pmatrix} 255 \\ 224 \\ 192 \\ 159 \\ 127 \\ 95 \\ 63 \\ 32 \end{pmatrix} = \begin{pmatrix} \frac{1}{8} & \frac{1}{8} \\ \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} \\ \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & -\frac{1}{2} \end{pmatrix} \begin{pmatrix} 255 \\ 224 \\ 192 \\ 159 \\ 127 \\ 95 \\ 63 \\ 32 \end{pmatrix} = \begin{pmatrix} 143.38 \\ 64.125 \\ 32.0 \\ 31.75 \\ 15.5 \\ 16.5 \\ 16.0 \\ 15.5 \end{pmatrix}.$$

В этом заключается только половина работы. Для того, чтобы сделать полное преобразование, необходимо применить  $W$  к строкам произведения  $WI$ , или, что то же самое, умножим  $W$  на  $(WI)^T$ . Результат для удобства тоже транспонируем. Полное преобразование (см. строку `timg=w*img*w'` рис. 4.12) равно

$$I_{tr} = (W(WI)^T)^T = WIW^T.$$

Для обратного преобразования справедлива формула

$$W^{-1} (W^{-1} I_{tr}^T)^T = W^{-1} (I_{tr} (W^{-1})^T).$$

В этом месте становится важным нормализованное преобразование Хаара (упомянутое на стр. 216). Вместо вычисления средних (выражений  $(d_i + d_{i+1})/2$ ) и разностей (выражений  $(d_i - d_{i+1})/2$ ) лучше вычислять величины  $(d_i + d_{i+1})/\sqrt{2}$  и  $(d_i - d_{i+1})/\sqrt{2}$ . Это приводит к ортогональной матрице  $W$ , а хорошо известно, что обращение такой матрицы сводится к ее транспонированию. Следовательно, обратное преобразование запишется в простом виде  $W^T I_{tr} W$  (см. строку `cimg=full(w'*sparse(dim)*w` на рис. 4.12).

Между процедурами прямого и обратного преобразования некоторые коэффициенты могут быть квантованы или отброшены. Кроме того, для лучшего сжатия, матрицу  $I_{tr}$  можно кодировать по методу RLE и/или по методу Хаффмана.

Функция `individ(n)` на рис. 4.12 начинается с матрицы преобразования Хаара размера  $2 \times 2$  (заметим, что вместо знаменателя 2 взято число  $\sqrt{2}$ ), затем использует эту матрицу для построения необходимого числа матриц  $A_i$ . Функция `harmatt(dim)` формирует окончательную матрицу Хаара для изображения, состоящего из  $dim$  строк и  $dim$  столбцов.

**Пример:** Программа Matlab на рис. 4.15 вычисляет  $W$  в виде произведения трех матриц  $A_1, A_2$  и  $A_3$ , после чего делает преобразования изображения размера  $8 \times 8$  из рис. 4.8. Результатом становится матрица  $8 \times 8$ , состоящая из коэффициентов преобразования,

в которой верхний левый коэффициент 131.375 равен среднему всех 64 пикселов исходного изображения.

```

a1=[1/2 1/2 0 0 0 0 0 0; 0 0 1/2 1/2 0 0 0 0;
    0 0 0 1/2 1/2 0 0; 0 0 0 0 0 0 1/2 1/2;
    1/2 -1/2 0 0 0 0 0 0; 0 0 1/2 -1/2 0 0 0 0;
    0 0 0 1/2 -1/2 0 0; 0 0 0 0 0 0 1/2 -1/2];
% a1*[255; 224; 192; 159; 127; 95; 63; 32];
a2=[1/2 1/2 0 0 0 0 0 0; 0 0 1/2 1/2 0 0 0 0;
    1/2 -1/2 0 0 0 0 0 0; 0 0 1/2 -1/2 0 0 0 0;
    0 0 0 1 0 0 0; 0 0 0 0 0 1 0 0;
    0 0 0 0 0 1 0; 0 0 0 0 0 0 0 1];
a3=[1/2 1/2 0 0 0 0 0 0; 1/2 -1/2 0 0 0 0 0 0;
    0 0 1 0 0 0 0 0; 0 0 0 1 0 0 0 0;
    0 0 0 1 0 0 0; 0 0 0 0 0 1 0 0;
    0 0 0 0 0 1 0; 0 0 0 0 0 0 0 1];
w=a3*a2*a1;
dim=8;
fid=fopen('8x8','r');
img=fread(fid,[dim, dim])';
fclose(fid);
w*img*w % Результат преобразования

```

131.375	4.250	-7.875	-0.125	-0.25	-15.5	0	-0.25
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
12.000	59.875	39.875	31.875	15.75	32.0	16	15.75
12.000	59.875	39.875	31.875	15.75	32.0	16	15.75
12.000	59.875	39.875	31.875	15.75	32.0	16	15.75
12.000	59.875	39.875	31.875	15.75	32.0	16	15.75

Рис. 4.15. Программа и результат матричного вейвлетного преобразования  $WIW^T$ .

### 4.3. Поддиапазонные преобразования

Все преобразования, которые обсуждались в § 3.5, являются *ортогональными*, поскольку в их основе лежат ортогональные матрицы. Ортогональное преобразование можно также выразить с помощью *скалярного произведения* вектора данных (пикселов или звуковых фрагментов) и множества *базисных функций*. Результатом ортогонального преобразования служат преобразованные коэффициенты, которые можно сжимать с помощью RLE, кодирования Хаффмана

или иного метода. Сжатие с потерей осуществляется путем квантования части преобразованных коэффициентов, которое делается до процедуры сжатия.

Дискретное скалярное произведение двух векторов  $f_i$  и  $g_i$  задается формулой

$$\langle f, g \rangle = \sum_i f_i g_i.$$

В начале § 3.5 рассматривалось преобразование вида  $c_i = \sum_j d_j w_{ij}$ , где  $d_j$  – исходные данные, а  $w_{ij}$  – некоторые весовые коэффициенты.

С другой стороны, вейвлетные преобразования являются поддиапазонными преобразованиями. Их можно вычислять с помощью операции свертки исходных данных (будь то пиксели или звуковые фрагменты) и множества фильтров пропускания определенных частот. В результате поддиапазон охватывает некоторую часть полосы частот исходных данных.

Слово «свертка» означает совместное сворачивание величин или функций. Дискретная свертка двух векторов  $f_i$  и  $g_i$ , которая также является вектором, обозначается  $f \star g$ . Компоненты свертки вычисляются по формулам

$$(f \star g)_i = \sum_j f_j g_{i-j}. \quad (4.3)$$

(Операцию свертки можно также определить для функций, но для наших потребностей сжатия данных достаточно будет дискретной свертки). Заметим, что пределы суммирования в формуле (4.3) не указаны точно. Они зависят от размерности векторов  $f_i$  и  $g_i$ . Примерами могут служить формулы (4.9).

Далее в этом параграфе обсуждаются линейные системы. Здесь также объясняется, почему операция свертки задается таким странным способом. Этот математический материал можно пропустить при первом чтении.

Для начала рассмотрим интуитивное понятие *системы*. Система принимает на входе некоторый сигнал и в ответ генерирует некоторый сигнал на выходе. Входные и выходные сигналы могут быть одномерными (функциями времени), двумерными (пространственными функциями двух пространственных координат) или многомерными. Нас будет интересовать связь между входными и выходными сигналами. Мы будем рассматривать только *линейные системы*, поскольку они очень важны и легко устроены. *Линейная система* определяется следующим образом. Если входной сигнал  $x_1(t)$  порождает выходной сигнал  $y_1(t)$  (это будет обозначаться  $x_1(t) \rightarrow y_1(t)$ ),

и если  $x_2(t) \rightarrow y_2(t)$ , то  $x_1(t) + x_2(t) \rightarrow y_1(t) + y_2(t)$ . Если система не удовлетворяет этому свойству, то она является нелинейной.

Из этого определения в частности следует, что  $2x_1(t) = x_1(t) + x_1(t) \rightarrow y_1(t) + y_1(t) = 2y_1(t)$ , или в более общем виде:  $ax_1(t) \rightarrow ay_1(t)$  для любого вещественного числа  $a$ .

Некоторые линейные системы являются *трансляционно-инвариантными*. В такой системе, если  $x(t) \rightarrow y(t)$  то  $x(t - T) \rightarrow y(t - T)$ , то есть, сдвиг входного сигнала по времени на число  $T$  вызовет такой же сдвиг выходного сигнала. В связи с рассмотрением свертки, мы предполагаем, что обсуждаемая нами система является линейной и трансляционно-инвариантной. Это верно (с высокой точностью) для электрических цепей и для оптических систем, на базе которых строятся различные устройства для обработки и сжатия изображений и иных типов оцифрованных данных.

Полезно иметь некоторую общую формулу для представления линейных систем. Оказывается, что представление вида

$$y(t) = \int_{-\infty}^{+\infty} f(t, \tau) x(\tau) d\tau \quad (4.4)$$

является уже достаточно общим для этих целей. Другими словами, достаточно знать функцию  $f(t, \tau)$ , зависящую от двух параметров, чтобы уметь предсказывать выход системы  $y(t)$  по известному входу  $x(\tau)$ . Однако, нам хотелось бы иметь однопараметрическую функцию, и здесь нам поможет свойство трансляционной инвариантности. Если система, порождаемая уравнением (4.4), обладает этим свойством, то должно выполняться тождество

$$y(t - T) = \int_{-\infty}^{+\infty} f(t, \tau) x(\tau - T) d\tau,$$

при любых  $T$ . Сделаем замену переменной в обеих частях этого равенства, добавив число  $T$  к  $t$  и  $\tau$ , и получим

$$y(t) = \int_{-\infty}^{+\infty} f(t + T, \tau + T) x(\tau) d\tau. \quad (4.5)$$

Вычтем (4.4) из этого уравнения:

$$0 = \int_{-\infty}^{+\infty} [f(t + T, \tau + T) - f(t, \tau)] x(\tau) d\tau.$$

Это равенство должно выполняться для любого входного сигнала  $x(t)$ . Поэтому выражение в квадратных скобках должно равняться

нулю. Следовательно,  $f(t + T, \tau + T) \equiv f(t, \tau)$  при всех  $T$ . Значит, функция  $f(t, \tau)$  не изменится, если добавить любое число  $T$  к ее аргументам, то есть она остается постоянной, если разность аргументов – константа, а сама функция  $f$  зависит только от разности своих аргументов. Тогда ее можно записать в виде  $f(t, \tau) = g(t - \tau)$ , а уравнение (4.4) примет следующий простой вид:

$$y(t) = \int_{-\infty}^{+\infty} g(t - \tau) x(\tau) d\tau. \quad (4.6)$$

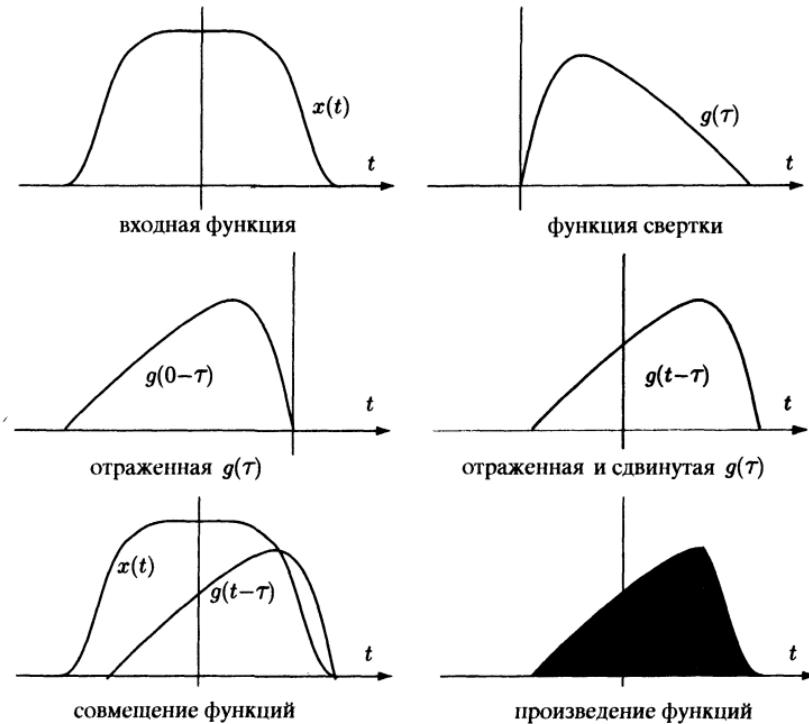


Рис. 4.16. Свертка функций  $x(t)$  и  $g(t)$ .

Это соотношение определяет *интегральную свертку*, важную операцию между  $x(t)$  и  $g(t)$ , которое связывает  $x(t)$  и  $y(t)$ . Эта операция обозначается  $y = g * x$ . Принято говорить, что линейная трансляционно-инвариантная система задается с помощью свертки (или конволюции) входного сигнала  $x$  и некоторой функции  $g(t)$ . Функция  $g(t)$ , которая является основной характеристикой систем-



мы, называется *импульсным откликом* системы. На рис. 4.16 приведено графическое представление свертки, где конечный результат (интеграл) равен площади серой области под кривой.

Свертка имеет несколько важных свойств. Она удовлетворяет свойствам коммутативности, ассоциативности и дистрибутивности по сложению, то есть, имеют место следующие тождества:

$$\begin{aligned} f \star g &= g \star f, \\ f \star (g \star h) &= (f \star g) \star h, \\ f \star (g + h) &= f \star g + f \star h. \end{aligned} \tag{4.7}$$

На практике непрерывные сигналы преобразуются в дискретные последовательности чисел, поэтому нам также понадобится *дискретная свертка*. Дискретная свертка двух числовых последовательностей  $f(i)$  и  $g(i)$  задается равенством

$$h(i) = f(i) \star g(i) = \sum_j f(j)g(i-j). \tag{4.8}$$

Если длины последовательностей  $f(i)$  и  $g(i)$  равны, соответственно,  $m$  и  $n$ , то  $h(i)$  имеет длину  $m + n - 1$ .

**Пример:** Даны две последовательности  $f = (f(0), f(1), \dots, f(5))$  (шесть элементов) и  $g = (g(0), g(1), \dots, g(4))$  (пять элементов). Уравнения (4.8) задают свертку  $h = f \star g$  из 10 элементов:

$$\begin{aligned} h(0) &= \sum_{j=0}^0 f(j)g(0-j) = f(0)g(0) \\ h(1) &= \sum_{j=0}^1 f(j)g(1-j) = f(0)g(1) + f(1)g(0) \\ h(2) &= \sum_{j=0}^2 f(j)g(2-j) = f(0)g(2) + f(1)g(1) + f(2)g(0) \\ h(3) &= \sum_{j=0}^3 f(j)g(3-j) = f(0)g(3) + f(1)g(2) + f(2)g(1) + f(3)g(0) \\ h(4) &= \sum_{j=0}^4 f(j)g(4-j) = f(0)g(4) + f(1)g(3) + f(2)g(2) + f(3)g(1) + \\ &\quad + f(4)g(0) \end{aligned}$$

$$\begin{aligned}
 h(5) &= \sum_{j=1}^5 f(j)g(5-j) = f(1)g(4) + f(2)g(3) + f(3)g(2) + f(4)g(1) + \\
 &\quad + f(5)g(0) \\
 h(6) &= \sum_{j=2}^5 f(j)g(6-j) = f(2)g(4) + f(3)g(3) + f(4)g(2) + f(5)g(1) \\
 h(7) &= \sum_{j=3}^5 f(j)g(7-j) = f(3)g(4) + f(4)g(3) + f(5)g(2) \\
 h(8) &= \sum_{j=4}^5 f(j)g(8-j) = f(4)g(4) + f(5)g(3) \\
 h(9) &= \sum_{j=5}^5 f(j)g(9-j) = f(5)g(4).
 \end{aligned} \tag{4.9}$$

Простейшим примером использования свертки является сглаживание или очищение сигналов от шума. Здесь становится ясным, как можно использовать свертку в фильтрах. Имея зашумленный сигнал  $f(t)$  (рис. 4.17), выберем прямоугольный импульс в качестве компоненты  $g(t)$  свертки:

$$g(t) = \begin{cases} 1, & -a/2 < t < a/2, \\ 1/2, & t = \pm a/2, \\ 0, & \text{иначе,} \end{cases}$$

где  $a$  – некоторое подходящее малое число (например,  $a = 1$ ). При вычислении свертки, импульс перемещается слева направо и умножается на  $f(t)$ . Результат произведения равен локальному среднему функции  $f(t)$  на интервале длины  $a$ . Это выглядит как отбрасывание высокочастотных флюктуаций из исходного сигнала  $f(t)$ .

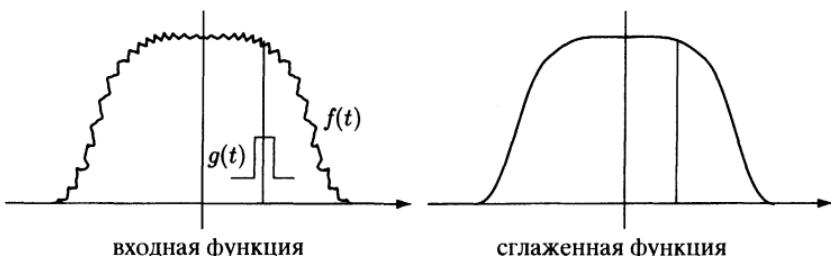


Рис. 4.17. Применение свертки при «очищении» функции от шума.

«О нет, — сказал Жорж, — это больше чем деньги.»

Он обхватил голову ладонями и постарался припомнить что-нибудь еще, кроме денег. Серое вещества в его голове было полно конвульсий и извилин, его барабанные перепонки были туго натянуты. Сквозь них проходили только звуки очень высокой частоты.

— Поль Скотт, «Клещи»

#### 4.4. Банк фильтров

Матричное определение преобразования Хаара будет использовано в этом параграфе для введения понятия *банка фильтров* [Strang, Nguen 96]. Будет показано, что преобразование Хаара можно интерпретировать как банк, состоящий из двух фильтров: один пропускает низкие частоты, а другой — высокие. Будет дано объяснение термину «фильтр», а также показано, как простая идея банка фильтров ложится в основу концепции поддиапазонных преобразований [Simoncelli и др. 90]. Конечно, преобразование Хаара является простейшим вейвлетным преобразованием, которое здесь употребляется для иллюстрации новых идей. Однако, его использование в качестве банка фильтров не очень эффективно. В конкретных приложениях используются более сложные множества фильтров, однако общая идея остается без изменений.

*Фильтром* называется линейный оператор, определяемый с помощью коэффициентов фильтра  $h(0), h(1), h(2), \dots$ . Этот оператор применяется к входному вектору  $x$ , в результате чего получается выходной вектор  $y$ :

$$y(n) = \sum_k h(k)x(n - k) = h \star x.$$

Заметим, что пределы суммирования зависят от выбора последовательностей  $x$  и  $h$ . Независимой переменной является время  $t$ , поэтому удобно считать, что входная и выходная последовательности заданы при всех временах  $t = \dots, -2, -1, 0, 1, 2, \dots$ . Итак, используется обозначения

$$x = (\dots, a, b, c, d, e, \dots),$$

причем центральное значение « $c$ » задает входной символ в нулевой момент времени [ $c = x(0)$ ], величины  $d$  и  $e$  определяют входные символы при  $t = 1$  и  $t = 2$ , а, кроме того,  $b = x(-1)$  и  $a = x(-2)$ . На практике входная последовательность всегда является конечной, поэтому считается, что бесконечный вектор  $x$  имеет лишь конечное число ненулевых компонентов.

Глубже вникнуть в работу фильтра можно с помощью простейшего входного сигнала  $x = (\dots, 0, 0, 1, 0, 0, \dots)$ . Эта последовательность равна нулю всюду кроме момента  $t = 0$ . Она называется единичным импульсом. Несмотря на то, что в сумме, задающей свертку, не заданы пределы суммирования, легко видеть, что при любом  $n$  имеется всего одно ненулевое слагаемое, то есть,  $y(n) = h(n)x(0) = h(n)$ . Будем говорить, что выходной сигнал  $y(n) = h(n)$  является *откликом* в момент времени  $t = n$  на единичный импульс  $x(0) = 1$ . Поскольку число коэффициентов фильтра  $h(i)$  конечно, то фильтр называется *конечным импульсным откликом* (FIR, finite impulse response).

На рис. 4.18 показана основная идея банка фильтров. Там изображен *банк анализа*, состоящий из двух фильтров: низкочастотного  $H_0$  и высокочастотного  $H_1$ . Низкочастотный фильтр использует свертку для удаления из сигнала высокочастотной составляющей. Он пропускает низкие частоты. А высокочастотный фильтр делает все наоборот. Он удаляет из сигнала высокие частоты. Вместе они разделяют входной сигнал на *поддиапазоны частот*.



Рис. 4.18. Банк фильтров из двух каналов.

Входной сигнал  $x$  может быть одномерным (вектором из вещественных компонентов, как это предполагается в этом параграфе) или двумерным, то есть, изображением. Элементы  $x(n)$  подаются на вход фильтров один за другим, и каждый фильтр вычисляет и выдает на выход один сигнальный отклик  $y(n)$ . Число откликов в два раза больше числа входных сигналов (так как рассматривается два фильтра). Разочаровывающий результат, поскольку мы хотим получить сжатие. Для исправления этого обстоятельства после прохождения через фильтр делается прореживание, при котором выбрасываются отклики с нечетными номерами. Эту операцию также называют *декимацией*. На рисунке она обозначается в квадратике в виде  $\downarrow 2$ . После применения декимации число элементов на выходе равно числу элементов на входе.

**Пример:** Легко построить банк фильтров, в котором низкочастотный фильтр вычисляет средние значения, а высокочастотный фильтр вычисляет полуразности, то есть, делается преобразование

Хаара входного сигнала. Коэффициенты низкочастотного фильтра равны  $h(0) = h(1) = 1/2$ , а коэффициенты высокочастотного фильтра равны  $h(0) = -1/2$  и  $h(1) = 1/2$ . Применяем эти фильтры к одномерной входной последовательности

$$(x(0), \dots, x(7)) = (255, 224, 192, 159, 127, 95, 63, 32)$$

и получаем последовательность средних чисел  $a(i)$

$$\begin{aligned} a(0) &= 0.5 \cdot x(0) + 0.5 \cdot x(-1) = 0.5 \cdot (255 + 0) = 127.5, \\ a(1) &= 0.5 \cdot x(1) + 0.5 \cdot x(0) = 0.5 \cdot (224 + 255) = 239.5, \\ a(2) &= 0.5 \cdot x(2) + 0.5 \cdot x(1) = 0.5 \cdot (192 + 224) = 208, \\ a(3) &= 0.5 \cdot x(3) + 0.5 \cdot x(2) = 0.5 \cdot (159 + 192) = 175.5, \\ a(4) &= 0.5 \cdot x(4) + 0.5 \cdot x(3) = 0.5 \cdot (127 + 159) = 143, \\ a(5) &= 0.5 \cdot x(5) + 0.5 \cdot x(4) = 0.5 \cdot (95 + 127) = 111, \\ a(6) &= 0.5 \cdot x(6) + 0.5 \cdot x(5) = 0.5 \cdot (63 + 95) = 79, \\ a(7) &= 0.5 \cdot x(7) + 0.5 \cdot x(6) = 0.5 \cdot (32 + 63) = 47.5, \\ a(8) &= 0.5 \cdot x(8) + 0.5 \cdot x(7) = 0.5 \cdot (0 + 32) = 16, \end{aligned}$$

и последовательность полуразностей  $d(i)$

$$\begin{aligned} d(0) &= -0.5 \cdot x(0) + 0.5 \cdot x(-1) = 0.5 \cdot (-255 + 0) = -127.5, \\ d(1) &= -0.5 \cdot x(1) + 0.5 \cdot x(0) = 0.5 \cdot (-224 + 255) = 15.5, \\ d(2) &= -0.5 \cdot x(2) + 0.5 \cdot x(1) = 0.5 \cdot (-192 + 224) = 16, \\ d(3) &= -0.5 \cdot x(3) + 0.5 \cdot x(2) = 0.5 \cdot (-159 + 192) = 16.5, \\ d(4) &= -0.5 \cdot x(4) + 0.5 \cdot x(3) = 0.5 \cdot (-127 + 159) = 16, \\ d(5) &= -0.5 \cdot x(5) + 0.5 \cdot x(4) = 0.5 \cdot (-95 + 127) = 16, \\ d(6) &= -0.5 \cdot x(6) + 0.5 \cdot x(5) = 0.5 \cdot (-63 + 95) = 16, \\ d(7) &= -0.5 \cdot x(7) + 0.5 \cdot x(6) = 0.5 \cdot (-32 + 63) = 15.5, \\ d(8) &= -0.5 \cdot x(8) + 0.5 \cdot x(7) = 0.5 \cdot (-0 + 32) = 16. \end{aligned}$$

После децимации (прореживания) первая последовательность сокращается до  $(239.5, 175.5, 111, 47.5)$ , а вторая приобретает следующий вид:  $(15.5, 16.5, 16, 15.5)$ . Затем эти две последовательности объединяются в одну:  $(239.5, 175.5, 111, 47.5, 15.5, 16.5, 16, 15.5)$ , совпадающую с последовательностью, которая получается из уравнения (4.1).

Восстановление исходной последовательности делается с помощью *банка синтеза*, который отличается от банка анализа. Фильтр синтеза низких частот использует два коэффициента фильтра 1 и 1

для вычисления четырех восстановленных значений  $y(0)$ ,  $y(2)$ ,  $y(4)$  и  $y(6)$ , а фильтр синтеза высоких частот с помощью коэффициентов  $-1$  и  $1$  восстанавливает значения  $y(1)$ ,  $y(3)$ ,  $y(5)$  и  $y(7)$ . Затем эти восемь значений перемежаются для получения окончательной реконструкции исходной последовательности.

$$\begin{aligned} y(0) &= a(1) + d(1) = (239.5 + 15.5) = 255, \\ y(2) &= a(3) + d(3) = (175.5 + 16.5) = 192, \\ y(4) &= a(5) + d(5) = (111 + 16) = 127, \\ y(6) &= a(7) + d(7) = (47.5 + 15.5) = 63, \\ \hline y(1) &= a(1) - d(1) = (239.5 - 15.5) = 224, \\ y(3) &= a(3) - d(3) = (175.5 - 16.5) = 159, \\ y(5) &= a(5) - d(5) = (111 - 16) = 95, \\ y(7) &= a(7) - d(7) = (47.5 - 15.5) = 32. \end{aligned}$$

Банки фильтров позволяют взглянуть на преобразование Хаара с общих позиций. Они же дают возможность построить другие, более изощренные вейвлетные преобразования. Эта техника обсуждается в § 4.5.

Использование банка фильтров имеет существенное преимущество по сравнению с одним фильтром, так как позволяет использовать свойства, недоступные одному фильтру. Прежде всего, это возможность восстанавливать исходный сигнал  $x$  из выходных сигналов  $H_0x$  и  $H_1x$  после их децимации (прореживания).

Децимация не является трансляционно инвариантной операцией. После ее применения выходная последовательность состоит из четных членов  $y(0), y(2), y(4), \dots$ , но если сделать задержку входа на один отсчет времени, то выходом будет служить последовательность  $y(-1), y(1), y(3), \dots$ , которая полностью отличается от подлинного выхода. Эти две последовательности являются двумя фазами выходного сигнала  $y$ .

Выходы банка анализа называются коэффициентами поддиапазона. Их можно квантовать (если допустима частичная потеря информации), а затем сжимать с помощью RLE, методом Хаффмана, арифметическим кодированием или любым другим методом. В конечном счете, они подаются на вход *банка синтеза*, где к ним сначала добавляются нули (на место отброшенных членов), после чего они пропускаются через обратные фильтры  $F_0$  и  $F_1$ , где из последовательностей формируется выходной вектор  $\hat{x}$ . Выход каждого фильтра анализа (после децимации) имеет вид

$$(\downarrow y) = (\dots, y(-4), y(-2), y(0), y(2), y(4), \dots).$$

Обратная процедура состоит в добавлении нулей на место отброшенных членов. Она преобразовывает этот вектор в форму

$$(\uparrow y) = (\dots, y(-4), 0, y(-2), 0, y(0), 0, y(2), 0, y(4), 0, \dots).$$

Децимация означает потерю данных. Обратная процедура не может компенсировать эту потерю, поскольку она лишь добавляет нули. Для того, чтобы достичнуть полного восстановления исходного сигнала  $x$ , необходимо конструировать фильтры так, чтобы они могли компенсировать эти потери. Первое свойство, которое часто применяется при проектировании фильтров – это свойство *ортогональности*. Банк анализа преобразования Хаара состоит из коэффициентов  $(1/2, 1/2)$  низкочастотного фильтра и из коэффициентов  $(-1/2, 1/2)$  высокочастотного фильтра. Скалярное произведение этих двух векторов  $(1/2, 1/2) \cdot (-1/2, 1/2) = 0$ . То есть, они ортогональны. Аналогично, банк синтеза состоит из двух ортогональных векторов коэффициентов фильтров  $(1, 1)$  и  $(-1, 1)$ .

На рис. 4.19 показано множество ортогональных фильтров размера 4. Эти фильтры ортогональны, поскольку скалярное произведение векторов их коэффициентов равно нулю:

$$(a, b, c, d) \cdot (d, -c, b, -a) = 0.$$

Заметьте, что фильтры  $H_0$  и  $F_0$  (как и фильтры  $H_1$  и  $F_1$ ) очень похожи. Конечно, еще необходимо выбрать подходящие значения для коэффициентов фильтров  $a, b, c$  и  $d$ . Детальное обсуждения этого вопроса выходит за рамки настоящей книги, но в § 4.5 иллюстрируются некоторые методы и правила, которые позволяют на практике определить коэффициенты фильтров. Примером служит фильтр Добеши (Daubechies) D4. Его коэффициенты приведены в уравнении (4.12).

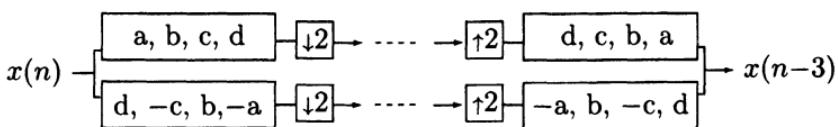


Рис. 4.19. Ортогональный банк фильтров с 4 коэффициентами.

Если с помощью этого фильтра вручную посчитать несколько примеров, то будет видно, что реконструированный сигнал иденти-

чен исходному входному сигналу, но отстает от него на три отсчета по времени.

Банк фильтров может быть *биортогональным*, менее ограничительным типом фильтров. На рис. 4.20 показан пример такого фильтра, который полностью восстанавливает исходный сигнал. Обратите внимание на схожесть фильтров  $H_0$  и  $F_1$ , а также фильтров  $H_1$  и  $F_0$ .

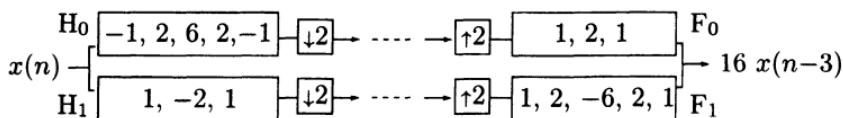


Рис. 4.20. Биортогональный банк фильтров с полным восстановлением.

Нам уже известно из § 4.2, что выходы фильтра низких частот  $H_0$  обычно пропускают через фильтры анализа несколько раз, при этом образуются все более короткие выходные сигналы. Эту рекурсивную процедуру можно изобразить в виде дерева (рис. 4.21). Поскольку каждый узел этого дерева вдвое сокращает число выходных символов своего предшественника, то это дерево называется *логарифмическим*. На рис. 4.21 показано, как масштабирующая функция  $\phi(t)$  и вейвлетная функция  $\psi(t)$  получаются в пределе из логарифмического дерева. Здесь обнаруживается связь дискретного вейвлетного преобразования (использующего банк фильтров) и континуального (непрерывного) вейвлетного преобразования (CWT, [Salomon 2000]).

Если «взбираться» вверх по логарифмическому дереву с уровня  $i$  на уровень  $i+1$ , то одновременно вычисляется новое среднее с помощью новой масштабирующей функции  $\phi(2^i t - k)$ , имеющей большую частоту, а также новые детали с помощью новой вейвлетной функции  $\psi(2^i t - k)$ :

$$\begin{array}{ccc} \text{сигнал на уровне } i \text{ (средние)} & \searrow & \\ + & & \\ \text{детали на уровне } i \text{ (разности)} & \nearrow & \text{сигнал на уровне } i+1. \end{array}$$

Каждый уровень дерева соответствует удвоению частоты (или разрешения) по отношению к предыдущему уровню, поэтому логарифмическое дерево еще называют деревом с *мультиразрешением*.

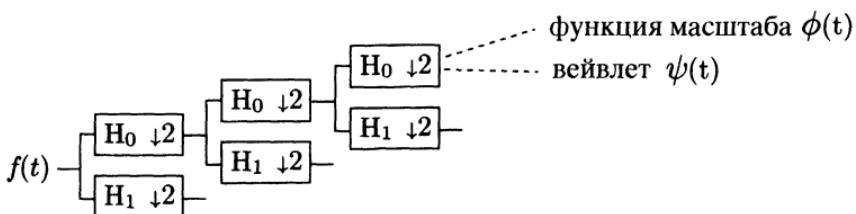


Рис. 4.21. Масштабирующая и вейвлетная функции как предел логарифмического дерева.

Тот, кто профессионально работает со звуком и музыкой, знает, что два тона, имеющие частоты  $\omega$  и  $2\omega$  звучат как один тон, но разной высоты. Частотный интервал между  $\omega$  и  $2\omega$  делится на 12 подинтервалов (называемых *хроматической гаммой*), однако в западной традиции отдается предпочтение 8 из 12 тонов этого разделения (*диатоническая гамма*, которая состоит из 7 нот, а 8 нота называется «октавой»). По этой причине основной частотный интервал, используемый в музыке, называется *октавой*. Следуя этой аналогии, можно сказать, что соседние уровни дерева мультиразрешения различаются на октаву частот.

**Заключение.** В этом параграфе рассматривались банки фильтров, которые будет полезно сравнить с преобразованием изображений из § 3.5. В обоих случаях обсуждаются преобразования, но это два разных типа преобразований. Каждое преобразование § 3.5 основано на некотором *ортогональном* базисе функций (или ортогональном базисе изображений), и они вычисляются с помощью взятия скалярного произведения входного сигнала со всеми базисными функциями. Результатом является множество коэффициентов преобразования, которые в дальнейшем сжимаются или без потери информации (с помощью метода RLE или иного энтропийного кодирования), или методом, допускающим потерю данных (тогда кодированию предшествует этап квантования преобразованных данных).

В этом параграфе рассматривались *поддиапазонные преобразования* – другой тип преобразований, вычисляемых с помощью *свертки* входного сигнала с частотными фильтрами из некоторого семейства с последующим прореживанием (децимацией) результатов. Каждая прореженная последовательность преобразованных коэффициентов является частью входного сигнала, частоты которого лежат в неко-

тором поддиапазоне. Реконструкция сигнала состоит в добавлении нулевых значений, после чего делаются обратные преобразования и их результаты складываются и подаются на выход.

Основное преимущество поддиапазонных преобразований состоит в том, что они выделяют различные частоты из входного сигнала, после чего можно точно контролировать часть сохраняемой (и отбрасываемой) информации из каждого поддиапазона частот. На практике такое преобразование разлагает образ на несколько поддиапазонов, отвечающих разным частотным областям данного образа, после чего каждый поддиапазон можно квантовать независимо от остальных поддиапазонов.

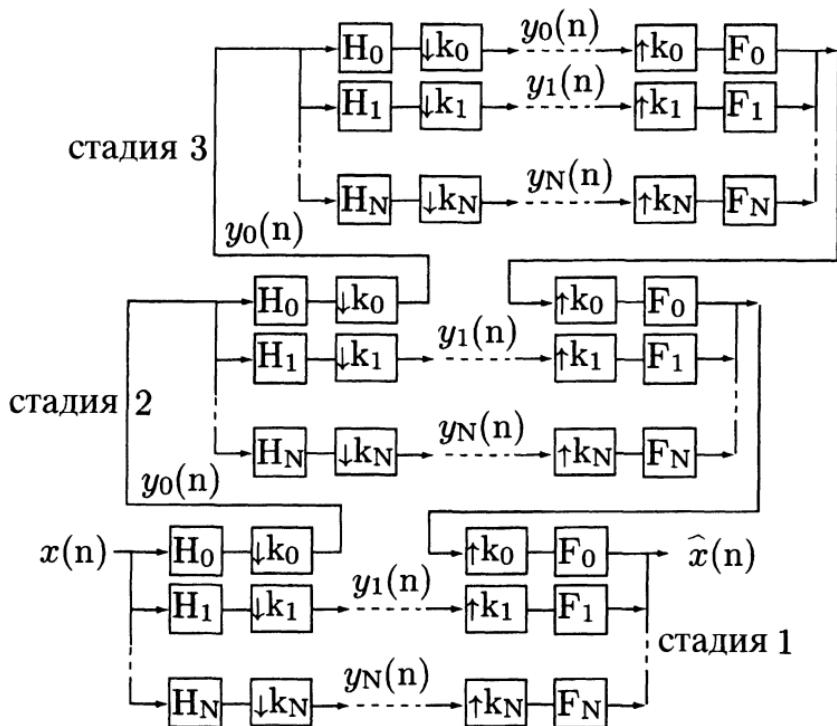


Рис. 4.22. Общий банк фильтров.

Главным недостатком таких преобразований является появление искусственных артефактов (то есть элементов, которых не было в

исходном образе, таких как наложения, затухания, «звоны») в реконструированном образе. По этой причине преобразование Хаара не является удовлетворительным, и основные исследования в этой области прежде всего направлены на поиски улучшенных фильтров.

На рис. 4.22 изображена общая схема банка фильтров, включающая  $N$  частотных фильтров и 3 стадии процесса преобразования. Обратите внимание, что выход частотного фильтра  $H_0$  каждой стадии направляется на вход следующей стадии для дальнейшего разделения, а комбинированный выход банка синтеза 3-ей и 2-ой стадий посыпается наверх обратного фильтра банка синтеза предшествующей стадии.

## 4.5. Нахождение коэффициентов фильтра

После рассмотрения общих операций над фильтрами возникает следующий вопрос: «Как находить коэффициенты фильтров?» Полный ответ на этот вопрос весьма непрост и выходит за рамки нашего изложения (см., например, [Akansu, Haddad 92]). В этом параграфе мы постараемся дать некоторое представление об основных правилах и методах вычисления коэффициентов различных банков фильтров.

Пусть имеется два прямых фильтра  $H_0, H_1$  и два обратных фильтра  $F_0, F_1$ , которые состоят из  $N$  отсчетов (число  $N$  предполагается четным). Обозначим их коэффициенты через

$$h_0 = (h_0(0), h_0(1), \dots, h_0(N-1)), \quad h_1 = (h_1(0), h_1(1), \dots, h_1(N-1)),$$

$$f_0 = (f_0(0), f_0(1), \dots, f_0(N-1)), \quad f_1 = (f_1(0), f_1(1), \dots, f_1(N-1)).$$

Четыре вектора  $h_0, h_1, f_0$  и  $f_1$  являются *импульсными откликами* четырех фильтров. Вот простейшие правила, позволяющие выбрать численные значения для этих векторов:

1. **Нормализация:** Вектор  $h_0$  имеет единичную длину.
2. **Ортогональность:** Для любого целого числа  $i$ , удовлетворяющего неравенству  $1 \leq i < N/2$ , вектор, состоящий из первых  $2i$  элементов  $h_0$ , должен быть ортогонален вектору, составленному из последних  $2i$  элементов того же вектора  $h_0$ .
3. **Вектор  $f_0$ :** Вектор  $f_0$  состоит из компонентов вектора  $h_0$ , записанных в обратном порядке.
4. **Вектор  $h_1$ :** Вектор  $h_1$  является копией  $f_0$ , но с обратными знаками у компонент на нечетных позициях (первой, третьей и т.д.). Формально



это можно выразить как покомпонентное умножение вектора  $h_1$  на вектор  $(-1, 1, -1, 1, \dots, -1, 1)$ .

5. Вектор  $f_1$  является копией  $h_0$ , но с обратными знаками у компонент на четных позициях (второй, четвертой и т.д.). То есть,  $f_1$  равен  $h_0$ , умноженному на  $(1, -1, 1, -1, \dots, 1, -1)$ .

Для фильтра с двумя отсчетами правило 1 означает, что

$$h_0^2(0) + h_0^2(1) = 1. \quad (4.10)$$

Правило 2 не применимо, так как  $N = 2$  и неравенство  $i < N/2$  означает, что  $i < 1$ . Правила 3–5 дают соотношения

$$f_0 = (h_0(1), h_0(0)), \quad h_1 = (-h_0(1), h_0(0)), \quad f_1 = (h_0(0), -h_0(1)).$$

Значит, все зависит от выбора чисел  $h_0(0)$  и  $h_0(1)$ . Однако, уравнения (4.10) не достаточно для их определения. Тем не менее видно, что значения  $h_0(0) = h_0(1) = 1/\sqrt{2}$  удовлетворяют этому условию.

Для фильтров из четырех отсчетов правила 1 и 2 означают, что

$$h_0^2(0) + h_0^2(1) + h_0^2(2) + h_0^2(3) = 1, \quad h_0(0)h_0(2) + h_0(1)h_0(3) = 0, \quad (4.11)$$

а правила 3–5 дают соотношения

$$\begin{aligned} f_0 &= (h_0(3), h_0(2), h_0(1), h_0(0)), \\ h_0 &= (-h_0(3), h_0(2), -h_0(1), h_0(0)), \\ f_1 &= (h_0(0), -h_0(1), h_0(2), -h_0(3)). \end{aligned}$$

Опять, уравнений (4.11) не достаточно для точного задания четырех неизвестных, поэтому необходимы дополнительные условия для вывода четырех коэффициентов (здесь помогает математическая интуиция). Эти коэффициенты приведены в (4.12), они образуют фильтр Добеши D4.

Для фильтра с восемью отсчетами, правила 1 и 2 приводят к уравнениям

$$\begin{aligned} h_0^2(0) + h_0^2(1) + h_0^2(2) + h_0^2(3) + h_0^2(4) + h_0^2(5) + h_0^2(6) + h_0^2(7) &= 1, \\ h_0(0)h_0(2) + h_0(1)h_0(3) + \\ + h_0(2)h_0(4) + h_0(3)h_0(5) + h_0(4)h_0(6) + h_0(5)h_0(7) &= 0, \\ h_0(0)h_0(4) + h_0(1)h_0(5) + h_0(2)h_0(6) + h_0(3)h_0(7) &= 0, \\ h_0(0)h_0(6) + h_0(1)h_0(7) &= 0, \end{aligned}$$

а правила 3–5 записываются в виде соотношений

$$\begin{aligned}f_0 &= (h_0(7), h_0(6), h_0(5), h_0(4), h_0(3), h_0(2), h_0(1), h_0(0)), \\h_1 &= (-h_0(7), h_0(6), -h_0(5), h_0(4), -h_0(3), h_0(2), -h_0(1), h_0(0)), \\f_1 &= (h_0(0), -h_0(1), h_0(2), -h_0(3), h_0(4), -h_0(5), h_0(6), -h_0(7)).\end{aligned}$$

Восемь допустимых коэффициентов приведены в табл. 4.23 (они образуют фильтр Добеши D8).

.230377813309	.714846570553	.630880767930	-.027983769417
-.187034811719	.030841381836	.032883011667	-.010597401785

Табл. 4.23. Коэффициенты фильтра Добеши с 8 отсчетами.

Для задания  $N$  коэффициентов для каждого из фильтров  $H_0$ ,  $H_1$ ,  $F_0$  и  $F_1$  необходимо знать коэффициенты с  $h_0(0)$  по  $h_0(N - 1)$ . Поэтому следует задать  $N$  уравнений для нахождения этих величин. Однако правила 1 и 2 дают только  $N/2$  уравнений. Значит, необходимо добавить новые условия для однозначного определения чисел с  $h_0(0)$  по  $h_0(N - 1)$ . Вот некоторые примеры таких условий:

*Фильтр пропускания низких частот:* Мы хотим, чтобы фильтр  $H_0$  пропускал только низкие частоты, поэтому имеет смысл потребовать, чтобы частотный отклик  $H_0(\omega)$  был равен нулю для самой высокой частоты  $\omega = \pi$ .

*Минимальный фильтр фазы:* Это условие означает, что нули комплексной функции  $H_0(z)$  должны лежать на единичной окружности комплексной плоскости или вне ее.

*Контролируемая колinearность:* Линейность фазового отклика можно контролировать с помощью нахождения минимума суммы

$$\sum_i (h_0(i) - h_0(N - 1 - i))^2.$$

Другие условия обсуждаются в [Akansu, Haddad 92].

## 4.6. Преобразование DWT

Информация, которая производится и анализируется в повседневной жизни, является дискретной. Она чаще поступает в виде чисел, а не в форме каких-то непрерывных функций. Поэтому на практике чаще применяются дискретные вейвлетные преобразования (DWT).

Конечно, непрерывные вейвлетные преобразования (CWT, см., например, [Lewalle 95] и [Rao, Bopardikar 98]) также интенсивно изучаются, поскольку это позволяет лучше понять действие DWT.

Преобразование DWT использует свертку, однако из опыта известно, что качество преобразований такого типа сильно зависит от двух вещей: от выбора масштабирующих множителей и временных сдвигов, а также от выбора вейвлета.

На практике преобразование DWT вычисляется с помощью масштабирующих множителей, которые равны отрицательным степеням двойки, и временных сдвигов, которые равны положительным степеням числа 2. На рис. 4.24 показана так называемая *двуухчленная решетка*, которая иллюстрирует такой выбор. Используемые вейвлеты порождают ортонормальные (или биортонормальные) базисы.

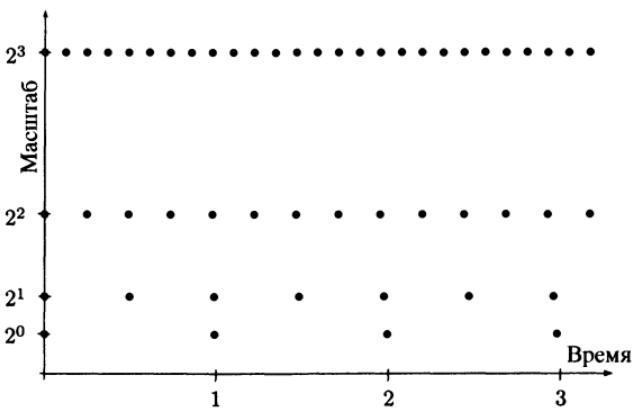


Рис. 4.24. Двуухчленная решетка. Связь масштаба со сдвигами по времени.

Основное направление исследования вейвлетов состоит в поисках семейств вейвлетов, которые образуют ортогональный базис. Среди этих вейвлетов предпочтение отдается вейвлетам с компактным носителем, поскольку они позволяют делать преобразования DWT с *конечным импульсным откликом* (FIR, finite impulse response).

Самый простой способ описания вейвлетных преобразований использует произведение матриц. Этот путь уже был продемонстрирован в § 4.2.1. Преобразование Хаара зависит от двух *коэффициентов фильтра*  $c_0$  и  $c_1$ , которые равны  $1/\sqrt{2} \approx 0.7071$ . Наименьшая матрица, которую можно построить в этом случае, равна

$$\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} / \sqrt{2}.$$

Эта матрица имеет размер  $2 \times 2$ . С ее помощью порождаются два коэффициента преобразования: среднее и разность. (Заметим, что эти среднее и разность не равны в точности полусумме и полуразности, поскольку вместо 2 используется знаменатель  $\sqrt{2}$ . Более точными терминами были бы, соответственно, выражения «*грубые детали*» и «*тонкие детали*»). В общем случае, DWT может использовать любое число фильтров, но все они вычисляются с помощью этого метода независимо от вида фильтров.

Сначала мы рассмотрим один из самых популярных вейвлетов, а именно вейвлет Добеши, который принято обозначать D4. Из этого обозначения видно, что он основан на четырех коэффициентах фильтра  $c_0, c_1, c_2$  и  $c_3$ , значения которых приведены в (4.12). Матрица  $W$  преобразования равна (ср. с матрицей  $A_1$  из (4.1))

$$W = \begin{pmatrix} c_0 & c_1 & c_2 & c_3 & 0 & 0 & \dots & 0 \\ c_3 & -c_2 & c_1 & -c_0 & 0 & 0 & \dots & 0 \\ 0 & 0 & c_0 & c_1 & c_2 & c_3 & \dots & 0 \\ 0 & 0 & c_3 & -c_2 & c_1 & -c_0 & \dots & 0 \\ \vdots & \vdots & & & & \ddots & & \\ 0 & 0 & \dots & 0 & c_0 & c_1 & c_2 & c_3 \\ 0 & 0 & \dots & 0 & c_3 & -c_2 & c_1 & -c_0 \\ c_2 & c_3 & 0 & \dots & 0 & 0 & c_0 & c_1 \\ c_1 & -c_0 & 0 & \dots & 0 & 0 & c_3 & -c_2 \end{pmatrix}.$$

Если эту матрицу применить к вектору-столбцу исходных данных  $(x_1, x_2, \dots, x_n)$ , то ее верхняя строка даст взвешенную сумму  $s_1 = c_0x_1 + c_1x_2 + c_2x_3 + c_3x_4$ . Третья строка матрицы определит сумму  $s_2 = c_0x_3 + c_1x_4 + c_2x_5 + c_3x_6$ , и все строки с нечетными номерами зададут аналогичные взвешенные суммы  $s_i$ . Такие суммы совпадают со свертками исходного вектора  $x_i$  и четырех коэффициентов фильтра. На языке вейвлетов все они называются *гладкими коэффициентами*, а вместе они именуются *сглаживающим фильтром*  $H$ .

Аналогично вторая строка матрицы  $W$  порождает величину  $d_1 = c_3x_1 - c_2x_2 + c_1x_3 - c_0x_4$ , а все остальные четные строки матрицы определяют подобные свертки. Каждое число  $d_i$  называется *детальным коэффициентом*, а все вместе они образуют фильтр  $G$ . Фильтр  $G$  не является сглаживающим. На самом деле, его коэффициенты подобраны так, чтобы фильтр  $G$  выдавал на выход маленькие числа,

когда данные  $x_i$  коррелированы. Все вместе,  $H$  и  $G$  называются *квадратурными зеркальными фильтрами* (QMF, quadrature mirror filters).

Таким образом вейвлетное преобразование любого изображения представляет собой прохождение исходного образа через фильтр QMF, который состоит из низкочастотного фильтра ( $H$ ) и высокочастотного фильтра ( $G$ ).

Если размер матрицы  $W$  равен  $n \times n$ , то она порождает  $n/2$  гладких коэффициентов  $s_i$  и  $n/2$  детальных коэффициентов  $d_i$ . Транспонированная матрица равна

$$W^T = \begin{pmatrix} c_0 & c_3 & 0 & 0 & \dots & c_2 & c_1 \\ c_1 & -c_2 & 0 & 0 & \dots & c_3 & -c_0 \\ c_2 & c_1 & c_0 & c_3 & \dots & 0 & 0 \\ c_3 & -c_0 & c_1 & -c_2 & \dots & 0 & 0 \\ & & & & \ddots & & \\ & & & & & c_2 & c_1 & c_0 & c_3 & 0 & 0 \\ & & & & & c_3 & -c_0 & c_1 & -c_2 & 0 & 0 \\ & & & & & & c_2 & c_1 & c_0 & c_3 & 0 & 0 \\ & & & & & & c_3 & -c_0 & c_1 & -c_2 & 0 & 0 \end{pmatrix}.$$

Можно показать, что матрица  $W$  будет ортогональной, если четыре порождающие ее коэффициента удовлетворяют соотношениям:  $c_0^2 + c_1^2 + c_2^2 + c_3^2 = 1$ ,  $c_2c_0 + c_3c_1 = 0$ . Еще два уравнения для вычисления коэффициентов фильтра имеют вид:  $c_3 - c_2 + c_1 - c_0 = 0$  и  $0c_3 - 1c_2 + 2c_1 - 3c_0 = 0$ . Они получаются из условия равенства нулю первых двух моментов последовательности  $(c_3, -c_2, c_1, -c_0)$ . Решением этих четырех уравнений служат следующие величины:

$$\begin{aligned} c_0 &= (1 + \sqrt{3}) / (4\sqrt{2}) \approx 0.48296, & c_1 &= (3 + \sqrt{3}) / (4\sqrt{2}) \approx 0.8365, \\ c_2 &= (3 - \sqrt{3}) / (4\sqrt{2}) \approx 0.2241, & c_3 &= (1 - \sqrt{3}) / (4\sqrt{2}) \approx 0.1294. \end{aligned} \tag{4.12}$$

Умножение на матрицу  $W$  очень просто и наглядно. Однако этот метод не практичен, так как  $W$  должна иметь такой же размер, что и исходное изображение, которое обычно велико. Если взглянуть на матрицу  $W$ , то видна ее регулярная структура, поэтому нет необходимости строить ее целиком. Достаточно хранить лишь ее верхнюю строку. На самом деле, достаточно иметь массив, состоящий из четырех коэффициентов фильтра.

```

function wc1=fwt1(dat,coarse,filter)
% Одномерное прямое дискретное вейвлетное преобразование
% dat - это вектор-строка размера  $2^n$ ,
% coarse - это самый грубый уровень преобразования
% (заметим, что coarse должен быть  $< n$ )
% используется ортогональный квадратурный фильтр,
% длина которого должна быть  $< 2^{(coarse+1)}$ 
n=length(dat); j=log2(n); wc1=zeros(1,n); beta=dat;
for i=j-1:-1:coarse
    alfa=HiPass(beta,filter);
    wc1((2^(i+1)): (2^(i+1)))=alfa;
    beta=LoPass(beta,filter) ;
end
wc1(1:(2^coarse))=beta;

function d=HiPass(dt,filter) % пропускает высокие частоты
d=iconv(mirror(filter),lshift(dt));
% iconv - функция свертки из пакета Matlab
n=length(d); d=d(1:2:(n-1));

function d=LoPass(dt,filter) % пропускает низкие частоты
d=aconv(filter,dt);
% aconv - это функция свертки из Matlab с обращенным
% во времени фильтром
n=length(d);
d=d(1:2:(n-1));

function sgn=mirrorfilt)
% возвращает коэффициенты фильтра с обратными знаками
sgn=-((-1).^(1:length(filt))).*filt;

```

Рис. 4.25. Одномерное прямое DWT (Matlab).

На рис. 4.25 показана программа Matlab, которая делает эти вычисления. Функция `fwt1(dat,coarse,filter)`, аргументом `dat` которой является исходный вектор из  $2^n$  элементов, а аргументом `filter` служит фильтр, вычисляет первые грубые уровни дискретного вейвлетного преобразования и записывает их в переменную `coarse`.

Простой тест для функции `fwt1`:

```

n=16; t=(1:n)./n;
dat=sin(2*pi*t)
filt=[0.4830 0.8365 0.2241 -0.1294];
wc=fwt1(dat,1,filt).

```

На рис. 4.26 приведена программа одномерного дискретного вейвлетного преобразования, которое вычисляется с помощью функции



`iwt1(wc,coarse,filter)`. Ниже приведен простой тест для ее проверки.

```

function dat=iwt1(wc,coarse,filter)
% Одномерное обратное дискретное вейвлетное преобразование
dat=wc(1:2^coarse);
n=length(wc); j=log2(n);
for i=coarse:j-1
    dat=ILoPass(dat,filter)+ ...
    IHiPass(wc((2^(i)+1):(2^(i+1))),filter);
end

function f=ILoPass(dt,filter)
f=iconv(filter,AltrntZro(dt));

function f=IHiPass(dt,filter)
f=aconv(mirror(filter),rshift(AltrntZro(dt)));

function sgn=mirror(filt)
% возвращает коэффициенты фильтра с обратными знаками
sgn=-((-1).^(1:length(filt))).*filt;

function f=AltrntZro(dt)
% возвращает вектор длины 2*n из нулей,
% размещенных между последовательными значениями
n =length(dt)*2; f =zeros(1,n);
f(1:2:(n-1))=dt;

```

Рис. 4.26. Одномерное обратное DWT (Matlab).

Простой тест для функции `iwt1`:

```

n=16; t=(1:n)./n;
dat=sin(2*pi*t)
filt=[0.4830 0.8365 0.2241 -0.1294];
wc=fwt1(dat,1,filt)
rec=iwt1(wc,1,filt)

```

Для читателей, которые потрудились разобраться с одномерными функциями `fwt1` и `iwt1` (рис. 4.25 и 4.26), мы приводим двумерные аналоги этих функций `fwt2` и `iwt2` (см. рис. 4.27 и 4.28) для которых также приведена простая тестовая программа.

Кроме семейства фильтров Добеши (между прочим, вейвлет Хаара порождает фильтр Добеши степени 2) существует множество других семейств вейвлетов, имеющие другие полезные свойства. Вот некоторые известные фильтры: фильтр Белкина, фильтр Койфмана, симметричный фильтр.

Семейство вейвлетов Добеши состоит из ортонормальных функций с компактным носителем, в котором каждая следующая функция имеет большую гладкость, чем предыдущая. В § 4.8 обсуждается вейвлет Добеши D4, а также его «строительный блок». Словосочетание «компактный носитель» означает, что эти функции равны нулю вне некоторого конечного отрезка числовой оси времени.

```

function wc=fwt2(dat,coarse,filter)
% Двумерное прямое вейвлетное преобразование
% dat - это матрица размера (2^n:2^n),
% «coarse» - самый грубый уровень преобразования
% (coarse должен быть <<n)
% длина фильтра <2^(coarse+1)
q=size(dat); n = q(1); j=log2(n);
if q(1) ~=q(2),
    disp('Неквадратное изображение!'),
end;
wc = dat; nc = n;
for i=j-1:-1:coarse,
    top = (nc/2+1):nc; bot = 1:(nc/2);
    for ic=1:nc,
        row = wc(ic,1:nc);
        wc(ic,bot)=LoPass(row,filter);
        wc(ic,top)=HiPass(row,filter);
    end
    for ir=1:nc, row = wc(1:nc,ir)';
        wc(top,ir)=HiPass(row,filter)';
        wc(bot,ir)=LoPass(row,filter)';
    end
    nc = nc/2;
end

function d=HiPass(dt,filter) % пропускает высокие частоты
d=iconv(mirror(filter),lshift(dt));
% iconv - свертка из Matlab
n=length(d); d=d(1:2:(n-1));

function d=LoPass(dt,filter) % пропускает низкие частоты
d=aconv(filter,dt);
% aconv - это функция свертки из Matlab с обращенным
% во времени фильтром
n=length(d);
d=d(1:2:(n-1));

function sgn=mirror(filt)
% возвращает коэффициенты фильтра с обратными знаками
sgn=-((-1).^(1:length(filt))).*filt;

```

Рис. 4.27. Двумерное прямое DWT (Matlab).

```

function dat=iwt2(wc,coarse,filter)
% Двумерное обратное вейвлетное преобразование
% n=length(wc); j=log2(n);
dat=wc;
nc=2^(coarse+1);
for i=coarse:j-1,
    top=(nc/2+1):nc; bot=1:(nc/2); all=1:nc;
    for ic=1:nc,
        dat(all,ic)=ILoPass(dat(bot,ic)',filter)';
        +IHiPass(dat(top,ic)',filter)';
    end % ic
    for ir=1:nc,
        dat(ir,all)=ILoPass(dat(ir,bot),filter)
        +IHiPass(dat(ir,top),filter);
    end % ir
    nc=2*nc;
end % i

function f=ILoPass(dt,filter)
f=iconv(filter,AltrntZro(dt));

function f=IHiPass(dt,filter)
f=aconv(mirror(filter),rshift(AltrntZro(dt)));
function sgn=mirror(filt)
% возвращает коэффициенты фильтра с обратными знаками
sgn=-((-1).^(1:length(filt))).*filt;
function f=AltrntZro(dt)
% возвращает вектор длины 2*n из нулей
% размещенных между последовательными значениями
n =length(dt)*2; f =zeros(1,n);
f(1:2:(n-1))=dt;

```

Рис. 4.28. Двумерное обратное DWT (Matlab).

Простой тест для функций **fwt2** и **iwt2**:

```

filename='house128'; dim=128;
fid=fopen(filename,'r');
if fid==-1 disp('file not found')
else img=fread(fid,[dim,dim]); fclose(fid);
end
filt=[0.4830 0.8365 0.2241 -0.1294];
fwim=fwt2(img,4,filt);
figure(1), imagesc(fwim), axis off, axis square
rec=iwt2(fwim,4,filt);
figure(2), imagesc(rec), axis off, axis square

```

Вейвлет Добеши D4 строится по четырем коэффициентам, приведенным в (4.12). Аналогично, вейвлет D6 имеет шесть коэффициентов. Их можно найти, решив систему из шести уравнений, три

из которых отражают свойство ортонормальности, а другие три получаются из условия равенства нулю первых трех моментов. Результат приведен в (4.13).

$$\begin{aligned}
 c_0 &= \frac{1 + \sqrt{10} + \sqrt{5 + 2\sqrt{10}}}{(16\sqrt{2})} \approx .3326, \\
 c_1 &= \frac{5 + \sqrt{10} + 3\sqrt{5 + 2\sqrt{10}}}{(16\sqrt{2})} \approx .8068, \\
 c_2 &= \frac{10 - 2\sqrt{10} + 2\sqrt{5 + 2\sqrt{10}}}{(16\sqrt{2})} \approx .4598, \\
 c_3 &= \frac{10 - 2\sqrt{10} - 2\sqrt{5 + 2\sqrt{10}}}{(16\sqrt{2})} \approx -.1350, \\
 c_4 &= \frac{5 + \sqrt{10} - 3\sqrt{5 + 2\sqrt{10}}}{(16\sqrt{2})} \approx -.0854, \\
 c_5 &= \frac{1 + \sqrt{10} - \sqrt{5 + 2\sqrt{10}}}{(16\sqrt{2})} \approx .0352.
 \end{aligned} \tag{4.13}$$

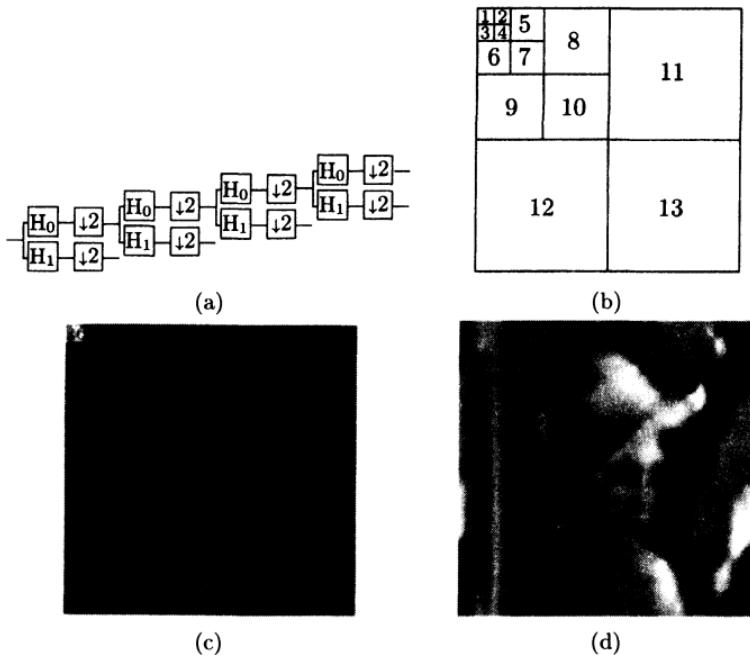
В каждой последовательности этого семейства число коэффициентов на два больше, чем в предыдущей, причем они являются более гладкими. Происхождение этих функций обсуждается в [Daubechies 88], [DeVore и др. 92] и [Vetterli, Kovacevic 95].

## 4.7. Примеры

Нам уже известно, что дискретное вейвлетное преобразование может восстанавливать изображения, если известно малое число коэффициентов преобразования. Первый пример этого параграфа иллюстрирует такое важное свойство, как способность реконструировать сильно огрубленные изображения, но без внесения артефактов, в которых обнулена значительная часть коэффициентов преобразования с помощью грубого квантования. Другие преобразования, особенно это касается DCT, способны вносить дополнительные артефакты в сжатый образ. Это свойство DWT делает его идеальным, например, в таких приложениях, как при сжатии отпечатков пальцев [Salomon 2000].

В этом примере используются функции `fwt2` и `iwt2` из рис. 4.27 и 4.28 для размытия или затуманивания изображения. Идея заключается в вычислении 4 шагов поддиапазонного преобразования образа (то есть, остановиться на 13 подуровнях), после чего приравнять большинство коэффициентов преобразования к нулю и грубо квантовать некоторые из оставшихся коэффициентов. Все это, конечно, означает потерю значительной доли информации и несовершенное восстановление данного изображения. При этом важно, что сжатый образ представляет собой *размытие* (затуманивание) исход-

ного изображения, а не огрубление его и внесение дополнительных артефактов.



```

clear, colormap(gray);
filename='lena128'; dim=128;
fid=fopen(filename,'r');
img=fread(fid,[dim,dim])';
filt=[0.23037,0.71484,0.63088,-0.02798, ...
-0.18703,0.03084,0.03288,-0.01059];
fwim=fwt2(img,3,filt);
figure(1), imagesc(fwim), axis square
    fwim(1:16,17:32)=fwim(1:16,17:32)/2;
    fwim(1:16,33:128)=0;
    fwim(17:32,1:32)=fwim(17:32,1:32)/2;
    fwim(17:32,33:128)=0;
    fwim(33:128,:)=0;
figure(2), colormap(gray), imagesc(fwim)
rec=iwt2(fwim,3,filt);
figure(3), colormap(gray), imagesc(rec)

```

Рис. 4.29. Размытие, как результат грубого квантования.

На рис. 4.29 показан результат размытия изображения «Лена». На рис. 4.29а и 4.29б изображены, соответственно, логарифмическое дерево мультиразрешения и структура поддиапазонов образа.

148	141	137	124	101	104	105	103	98	89	100	136
156	173	175	176	179	171	152	116	80	82	92	99
103	102	101	100	100	102	106	104	112	139	155	149
139	107	90	126	90	65	65	93	62	87	61	84
48	64	42	75	72	35	42	53	73	45	58	130
156	176	185	196	167	185	178	121	113	126	113	122
133	109	106	92	91	133	162	165	174	189	193	190
190	167	120	97	92	106	103	81	55	43	60	150
126	55	61	65	61	50	52	53	52	79	135	132
147	163	161	158	157	157	156	156	156	158	159	156
155	154	155	155	157	157	154	150				

(a)

117.95	-10.38	-5.99	-0.19	-11.64	12.6	-5.95	4.15
-2.57	6.61	-17.08	-0.50	7.88	-15.53	4.10	-10.80
-5.29	2.94	-0.63	5.42	-2.39	0.53	-5.96	2.67
-6.4	9.71	-5.43	0.56	-0.13	0.83	-0.02	1.17
-1.38	-2.68	1.92	3.14	-3.71	0.62	-0.02	-0.04
-1.41	-2.37	0.08	-1.62	-1.03	-3.50	2.52	2.81
-1.68	1.41	-1.79	1.11	3.55	-0.24	-7.44	0.28
-0.49	-2.56	1.98	-0.00	0.10	-0.17	0.42	0.65
0.35	-1.00	0.15	0.21	-1.30	0.31	0.21	0.45
0.85	-1.62	0.04	0.25	0	-0.10	0.23	-0.93
1.06	0.98	-2.43	0.35	-1.48	-1.72	-1.51	-1.54
-1.91	1.86	-0.67	1.95	-2.99	0.78	0.04	-1.55
2.42	-1.46	-0.64	1.47	0.23	-1.98	1.26	-0.32
0.42	0.95	-0.75	-1.02	1.01	-0.55	-3.45	3.31
-0.80	0.39	-0.11	-1.17	2.19	-0.25	0.25	-0.07
-0.03	-0.09	0.18	-0.02	0.02	0.06	0.08	0.19

(b)

117.95	-9.68	-16.44	1.31	-20.81	3.31	14.37	-29.44
-6.63	8.38	-20.56	39.38	10.44	-31.50	-14.25	1.13
7.75	22.13	4.25	-13.88	-24.37	21.50	24.00	9.25
0.13	11.38	-22.75	-28.88	0.38	-0.38	1.25	0.13
7.25	13.25	15.00	1.00	-1.75	11.00	-6.50	-25.75
-9.00	-5.00	6.50	35.00	4.75	3.50	21.50	-28.00
7.25	13.75	3.75	1.50	-6.50	-34.00	-10.75	-2.25
1.25	0.50	-0.50	-0.25	1.50	-0.25	-1.00	2.50
14.50	-9.00	3.50	28.50	4.00	-6.50	6.50	-4.50
-5.50	12.00	1.50	7.00	0.50	-21.00	-14.50	-1.50
-4.50	-7.50	-2.00	1.50	0	11.50	23.50	11.50
2.50	-7.00	1.50	11.00	13.00	6.00	-8.50	-45.00
12.00	35.50	-3.00	-2.00	2.00	5.50	-1.00	-0.50
0.50	-13.50	-28.00	1.50	-7.50	-8.00	1.00	1.50
0.50	0	0.50	0	0	-1.00	-0.50	1.50
0.50	0.50	-0.50	0	-1.00	0	1.50	2.00

(c)

Табл. 4.30. Преобразования Добеши и Хаара средней строки образа «Lena»

На рис. 4.29с приведен результат квантования. Коэффициенты преобразования поддиапазонов 5–7 были разделены на 2, а все коэффициенты поддиапазонов 8–13 были просто стерты. Прежде всего, большая часть изображения 4.29с выглядит равномерно черным (то есть, с нулевыми пикселями), однако более тщательное визуальное изучение обнаружит много ненулевых коэффициентов в поддиапазонах 5–7. Можно сказать, что размытое изображение на рис. 4.29д было получено при использовании всех коэффициентов поддиапазонов 1–4 (1/64 от общего числа коэффициентов преобразования) и половины коэффициентов поддиапазонов 5–7 (половины от 3/64, то есть, 3/128). Таким образом, изображение было восстановлено при использовании примерно  $5/128 \approx 0.039$  или 3.9% от общего числа коэффициентов преобразования. В этих вычислениях использовался вейвлет Добеши D8. Я призываю читателей поупражняться с этой программой и оценить достоинства этого и других фильтров.

```
<<WaveletTransform.m
data={148,141,137,124,101,104,105,103, 98, 89,100,136
...
,155,154,155,155,157,157,154,150};
forward = Wavelet[data, Daubechies[4]]
NumberForm[forward,{6,2}]
inverse = InverseWavelet[forward,Daubechies[4]]
data = inverse
(a) (Matematica)
```

```
% Преобразование Хаара (средние и разности)
data=[148 141 137 124 101 104 105 103 98 89 100 136 ...
...
155 154 155 155 157 157 154 150];
n=128; ln=7; %log_2 n=7
for k=1:ln,
  for i=1:n/2,
    i1=2*i; j=n/2+i;
    newdat(i)=(data(i1)+data(i1))/2;
    newdat(j)=(data(j1)-data(j))/2;
  end
  data=newdat; n=n/2;
end
round(100*data)/100
(b) (Matlab)
```

Рис. 4.31. Программы для вычисления табл. 4.30.

Второй пример демонстрирует преимущества фильтра Добеши D4 по сравнению с простейшим фильтром Хаара, основанном на

взятия средних и разностей, если сравнивать концентрацию энергии образов.

В табл. 4.30а приведены значения 128 пикселов, которые образуют строку 64 (среднюю линию) полутонового изображения «Lena» размера  $128 \times 128$ . В табл. 4.30б и 4.30с перечислены, соответственно, коэффициенты вейвлетного преобразования Добеши D4 и преобразования Хаара для этих данных. Первый коэффициент обоих преобразований одинаковый, но все остальные 127 коэффициентов, в среднем, меньше у преобразования D8. Это указывает на то, что фильтр D8 лучше концентрирует энергию образа. Среднее абсолютных величин этих 127 коэффициентов для D8 равно 2.1790, в то время как для фильтра Хаара эта величина равна 9.8446, то есть почти в 4.5 раза больше. Программы вычисления таблиц 4.30б и 4.30с даны на рис. 4.31. Отметим, что первая программа использует пакет WaveletTransform.m системы Mathematica, разработанный Алистаром Роу и Полем Эбботом, который размещен по адресу [Alistar, Abbott 01].

## 4.8. Вейвлеты Добеши

Многие полезные математические функции можно записать в явном виде. В этом смысле самой простой функцией, по-видимому, является многочлен. Однако имеется также большое число важных функций, которые задаются рекурсивно, то есть, через самих себя. Вы можете сказать, что если определять что-то через себя самого, то возникнет противоречие. Но этого легко избежать, если рекурсивное определение будет состоять из нескольких частей, причем одна из частей будет иметь явное выражение. Эта часть, обычно, содержит начальное задание определяемой функции. Простейшим примером может служить функция факториал. Ее можно задать явной формулой

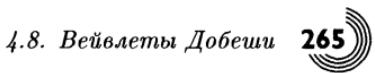
$$n! = n(n-1)(n-2) \cdots 3 \cdot 2 \cdot 1,$$

а можно определить с помощью двух равенств вида

$$1! = 1, \quad n! = n \cdot (n-1)!$$

Другим интересным примером служит функция  $e^x$  (или «exp»), которую можно также задать в виде рекурсивного соотношения

$$e^0 = 1, \quad \frac{de^x}{dx} = e^x.$$



Ингрид Добеши (Ingrid Daubechies) ввела вейвлет  $\psi$  и функцию шкалы (строительный блок)  $\varphi$  следующим образом. Одно требование состояло в том, чтобы функция шкалы  $\varphi$  имела компактный носитель. Она должна равняться нулю вне конечного отрезка. Добеши выбрала в качестве носителя отрезок  $[0, 3]$ . Она доказала, что эту функцию нельзя выразить через известные элементарные функции: многочлены, тригонометрические или степенные функции. Она также показала, что  $\varphi$  можно построить рекурсивно, с помощью некоторого начального задания и рекурсивного правила. Она выбрала следующие начальные значения:

$$\varphi(0) = 0, \varphi(1) = \frac{1 + \sqrt{3}}{2}, \varphi(2) = \frac{1 - \sqrt{3}}{2}, \varphi(3) = 0,$$

и задала рекурсивное соотношение

$$\begin{aligned} \varphi(r) &= \frac{1 + \sqrt{3}}{4} \varphi(2r) + \frac{3 + \sqrt{3}}{4} \varphi(2r - 1) + \frac{3 - \sqrt{3}}{4} \varphi(2r - 2) + \\ &+ \frac{1 - \sqrt{3}}{4} \varphi(2r - 3) = \\ &= h_0 \varphi(2r) + h_1 \varphi(2r - 1) + h_2 \varphi(2r - 2) + h_3 \varphi(2r - 3) = (4.14) \\ &= (h_0, h_1, h_2, h_3) \cdot (\varphi(2r), \varphi(2r - 1), \varphi(2r - 2), \varphi(2r - 3)). \end{aligned}$$

Отметим, что сумма начальных значений равна 1:

$$\varphi(0) + \varphi(1) + \varphi(2) + \varphi(3) = 0 + \frac{1 + \sqrt{3}}{2} + \frac{1 - \sqrt{3}}{2} + 0 = 1.$$

Дальнейшее вычисление значений функции  $\varphi$  совершается по шагам. На шаге 1 применяется условие компактного носителя, четыре начальные значения и рекурсивное соотношение (4.14) для вычисления значений  $\varphi(r)$  в трех новых точках  $r = 0.5, 1.5$  и  $2.5$ .

$$\begin{aligned} \varphi(1/2) &= h_0 \varphi(2/2) + h_1 \varphi(2/2 - 1) + h_2 \varphi(2/2 - 2) + h_3 \varphi(2/2 - 3) = \\ &= \frac{1 + \sqrt{3}}{4} \cdot \frac{1 + \sqrt{3}}{2} + h_1 \cdot 0 + h_2 \cdot 0 + h_3 \cdot 0 = \frac{2 + \sqrt{3}}{4}, \\ \varphi(3/2) &= h_0 \varphi(6/2) + h_1 \varphi(6/2 - 1) + h_2 \varphi(6/2 - 2) + h_3 \varphi(6/2 - 3) = \\ &= h_0 \cdot 0 + \frac{1 + \sqrt{3}}{4} \cdot \frac{1 - \sqrt{3}}{2} + \frac{1 - \sqrt{3}}{4} \cdot \frac{1 + \sqrt{3}}{2} + h_3 \cdot 0 = 0, \\ \varphi(5/2) &= h_0 \varphi(5) + h_1 \varphi(5 - 1) + h_2 \varphi(5 - 2) + h_3 \varphi(5 - 3) = \\ &= h_0 \cdot 0 + h_1 \cdot 0 + h_2 \cdot 0 + \frac{1 - \sqrt{3}}{4} \cdot \frac{1 - \sqrt{3}}{2} = \frac{2 - \sqrt{3}}{4}. \end{aligned}$$

Теперь значения функции  $\varphi(r)$  известны в четырех начальных точках 0, 1, 2, 3 и в трех промежуточных средних точках 0.5, 1.5 и 2.5, то есть, всего 7 значений. На шаге 2 можно вычислить еще 6 значений этой функции в точках  $1/4, 3/4, 5/4, 7/4, 9/4, 11/4$ . В результате получаются числа

$$\frac{5 + 3\sqrt{3}}{16}, \frac{9 + 5\sqrt{3}}{16}, \frac{1 + \sqrt{3}}{8}, \frac{1 - \sqrt{3}}{8}, \frac{9 - 5\sqrt{3}}{16}, \frac{5 - 3\sqrt{3}}{16}.$$

Итак, уже вычислены значения  $\varphi(r)$  в  $4+3+6 = 13$  точках (рис. 4.32).

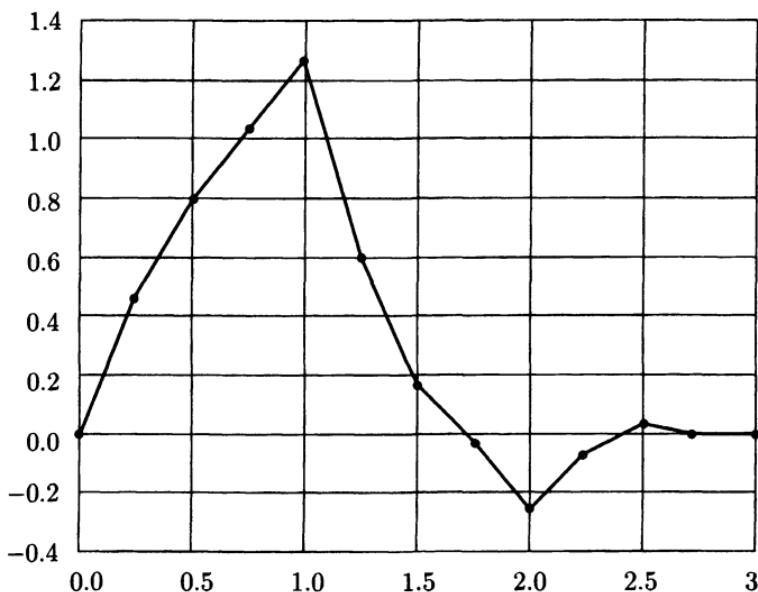


Рис. 4.32. Функция шкалы Добеши  $\varphi$  в 13 точках.

Шаг 3 даст еще 12 значений в точках посередине между 13 уже вычисленными точками. Получим всего  $12 + 13 = 25$  чисел. Следующие шаги дадут 24, 48, 96 и так далее значений. После шага  $n$  значения функции  $\varphi(r)$  будут уже известны в  $4+3+6+12+24+\dots+3 \cdot 2^n = 4 + 3(2^{n+1} - 1)$  точках. Например, после 9 шагов будет известно  $4 + 3(2^{10} - 1) = 3073$  значения (рис. 4.33).

Функция  $\varphi$  служит строительным блоком для построения вейвлета Добеши  $\psi$ , который задается также рекурсивно с помощью формулы

$$\begin{aligned}
 \psi(r) &= -\frac{1+\sqrt{3}}{4}\varphi(2r-1) + \frac{3+\sqrt{3}}{4}\varphi(2r) - \frac{3-\sqrt{3}}{4}\varphi(2r+1) + \\
 &+ \frac{1-\sqrt{3}}{4}\varphi(2r+2) = \\
 &= -h_0\varphi(2r-1) + h_1\varphi(2r) - h_2\varphi(2r+1) + h_3\varphi(2r+2).
 \end{aligned}$$

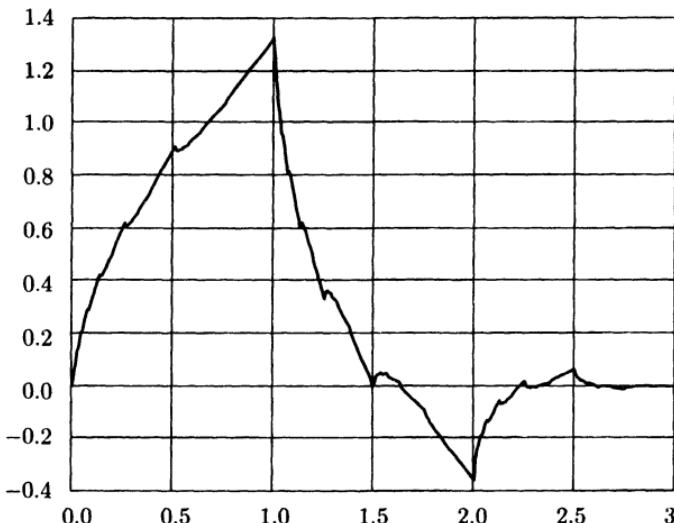


Рис. 4.33. Функция шкалы Добеши  $\varphi$  в 3073 точках.

Напомним, что функция  $\varphi(r)$  не равна нулю только на интервале  $[0, 3]$ . Поэтому носитель функции  $\psi(r)$  принадлежит отрезку  $[-1, 2]$ . Функцию  $\psi$  можно задать рекурсивно подобно функции  $\varphi$ . На рис. 4.34 показаны значения этого вейвлета в 3073 точках. Взгляд на рис. 4.33 и 4.34 также объясняет (хотя несколько поздно) причину выбора термина «вейвлет» (wavelet, маленькая волна, всплеск).

## 4.9. SPIHT

Алгоритм SPIHT представляет собой метод сжатия изображений. На одном из его этапов применяется вейвлетное преобразование, поэтому он рассматривается в этой главе. Кроме того, основная структура данных этого алгоритма, пространственно ориентированное дерево, использует тот факт, что различные поддиапазоны отражают разные геометрические особенности образа (это обстоятельство было отмечено на стр. 220).

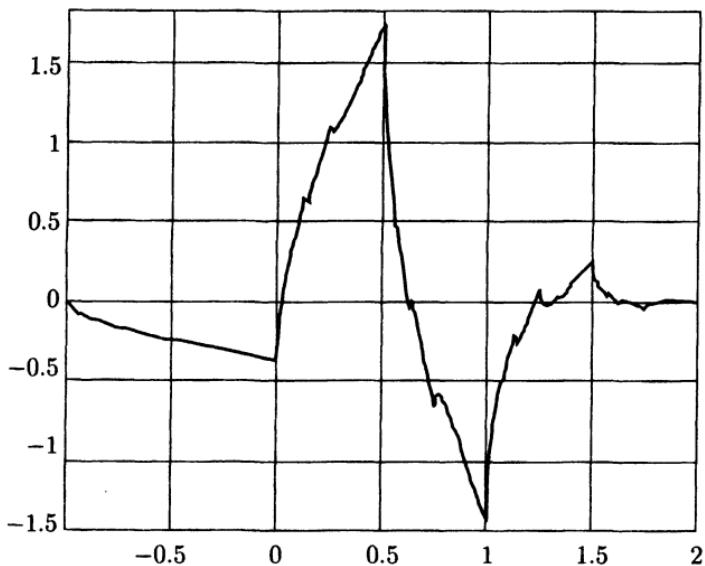


Рис. 4.34. Вейвлет Добеши  $\psi$  в 3073 точках.

В § 4.2 было показано, что преобразования Хаара можно применять к изображению несколько раз подряд. При этом образуются различные области (поддиапазоны), состоящие из средних и деталей данного образа. Преобразование Хаара является очень простым и наглядным, но для лучшего сжатия стоит использовать другие вейвлетные фильтры, которые лучше концентрируют энергию изображения. Представляется логичным, что различные вейвлетные фильтры будут давать разные коэффициенты сжатия для разных типов изображений. Однако исследователям пока не до конца ясно, как построить наилучший фильтр для каждого типа образов. Независимо от конкретно используемого фильтра, образ разлагается на поддиапазоны так, что нижние поддиапазоны соответствуют высоким частотам изображения, а верхние поддиапазоны отвечают низким частотам образа, в которых концентрируется основная часть энергии изображения (см. рис. 4.35). Поэтому можно ожидать, что коэффициенты деталей изображения уменьшаются при перемещении от высокого уровня к более низкому. Кроме того, имеется определенное пространственное подобие между поддиапазонами (рис. 4.7b). Части образа, такие как пики, занимают одну и ту же пространственную позицию во всех поддиапазонах. Эта особенность вейвлетного

разложения используется методом SPIHT, который расшифровывается как «Set partitioning in hierarchical trees» (разложение множества по иерархическим деревьям) (см. [Said, Pearlman 96]).

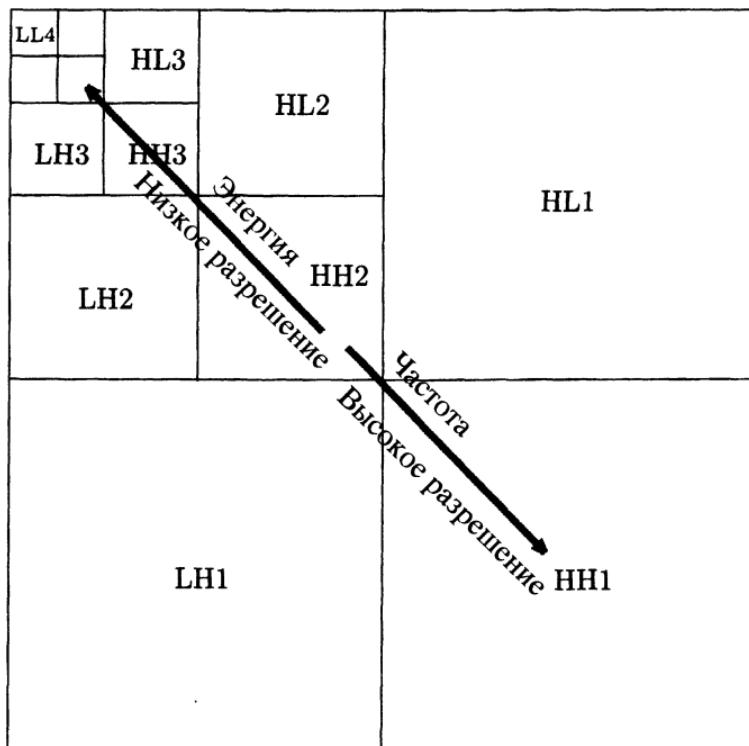


Рис. 4.35. Поддиапазоны и уровни вейвлетного разложения.

Метод SPIHT был разработан для оптимальной прогрессирующей передачи изображений, а также для их сжатия. Самая важная особенность этого алгоритма заключается в том, что на любом этапе декодирования качество отображаемой в этот момент картинки является наилучшим для введенного объема информации о данном образе.

Другая отличительная черта алгоритма SPIHT состоит в использовании им вложенного кодирования. Это свойство можно определить следующим образом: если кодер, использующий вложенное кодирование, производит два файла, большой объема  $M$  бит и маленький объема  $m$  бит, то меньший файл совпадает с первыми  $m$  битами большего файла.

Следующий пример удачно иллюстрирует это определение. Предположим, что три пользователя ожидают некоторое отправленное им сжатое изображение. При этом им требуется различное качество этого изображения. Первому из них требуется качество, обеспечиваемое 10 КБ образа, а второму и третьему пользователю необходимо качество, соответственно, в объеме 20 КБ и 50 КБ. Большинству методов сжатия изображения с частичной потерей данных потребуется сжимать исходный образ три раза с разным качеством, для того, чтобы сгенерировать три различных файла, требуемых объемов. А метод SPIHT произведет для этих целей всего один файл, и пользователям будут посланы три начальных фрагмента этого файла, длины которых соответственно равны 10 КБ, 20 КБ и 50 КБ.

Начнем с общего описания метода SPIHT. Обозначим пиксели исходного изображения  $\mathbf{p}$  через  $p_{i,j}$ . Любое множество фильтров  $\mathbf{T}$  может быть использовано для преобразования пикселов в вейвлетные коэффициенты (или коэффициенты преобразования)  $c_{i,j}$ . Эти коэффициенты образуют преобразованный образ  $\mathbf{c}$ . Само преобразование обозначается  $\mathbf{c} = \mathbf{T}(\mathbf{p})$ . При прогрессирующем методе передачи декодер начинает с того, что присваивает значение ноль реконструированному образу  $\hat{\mathbf{c}}$ . Затем он принимает (кодированные) преобразованные коэффициенты, декодирует их и использует для получения улучшенного образа  $\hat{\mathbf{c}}$ , который, в свою очередь, производит улучшенное изображение  $\hat{\mathbf{p}} = \mathbf{T}^{-1}(\hat{\mathbf{c}})$ .

Основная цель прогрессирующего метода состоит в скорейшей передаче самой важной части информации об изображении. Эта информация дает самое большое сокращение расхождения исходного и реконструированного образов. Для количественного измерения этого расхождения метод SPIHT использует среднеквадратическую ошибку MSE (mean squared error) (см. уравнение (3.2)),

$$D_{\text{mse}}(\mathbf{p} - \hat{\mathbf{p}}) = \frac{|\mathbf{p} - \hat{\mathbf{p}}|^2}{N} = \frac{1}{N} \sum_i \sum_j (p_{i,j} - \hat{p}_{i,j})^2,$$

где  $N$  – общее число пикселов. Принимая во внимание, что эта величина не меняется при переходе к преобразованным коэффициентам, можно записать равенства

$$D_{\text{mse}}(\mathbf{p} - \hat{\mathbf{p}}) = D_{\text{mse}}(\mathbf{c} - \hat{\mathbf{c}}) = \frac{|\mathbf{c} - \hat{\mathbf{c}}|^2}{N} = \frac{1}{N} \sum_i \sum_j (c_{i,j} - \hat{c}_{i,j})^2. \quad (4.15)$$

Уравнение (4.15) показывает, что MSE уменьшается на  $|c_{i,j}|^2/N$ , когда декодер получает коэффициент преобразования  $c_{i,j}$  (для про-

стоты мы предполагаем, что декодер получает точные значения коэффициентов преобразования, то есть не происходит потери точности из-за ограниченной разрядности компьютерной арифметики). Теперь становится ясно, что самые большие (по абсолютной величине) коэффициенты  $c_{i,j}$  несут в себе информацию, которая больше всего сокращает расхождение MSE, поэтому прогрессирующее кодирование должно посыпать эти коэффициенты в первую очередь. В этом заключается первый важный принцип SPIHT.

Другой принцип основан на наблюдении, что наиболее значимые биты двоичных представлений целых чисел стремятся быть единицами, когда эти числа приближаются к максимальным значениям. (Например, в 16-битном компьютере число +5 имеет представление 0|000...0101, а большое число +65382 запишется в виде 0|11111101100110. Это наводит на мысль, что старшие биты содержат наиболее значимую часть информации, и их также следует посыпать (или записывать в сжатый массив данных) в первую очередь.

Прогрессирующий метод передачи SPIHT использует эти два принципа. Он сортирует (упорядочивает) коэффициенты и сначала посыпает самые значимые биты этих коэффициентов. Для упрощения описания основ этого метода мы будем предполагать, что отсортированная информация непосредственно передается декодеру; в следующем параграфе обсуждается эффективный алгоритм для кодирования этой информации.

$k$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
ст.бит	знак	$s$														
	14	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	13	$a$	$b$	1	1	1	1	0	0	0	0	0	0	0	0	0
	12	$c$	$d$	$e$	$f$	$g$	$h$	1	1	1	0	0	0	0	0	0
	11	$i$	$j$	$k$	$l$	$m$	$m$	$o$	$p$	$q$	1	0	0	0	0	0
	:	:	:													:
мл.бит	0	$r$	$s$	$t$	$u$	$v$	$w$	$x$	$y$	...	...	...	...	...	...	$z$
$m(k) =$	$i, j$	2,3	3,4	3,2	4,4	1,2	3,1	3,3	4,2	4,2	4,1	...	...	...	...	4,3

Табл. 4.36. Коэффициенты преобразования, отсортированные по модулю.

Покажем теперь, как кодер метода SPIHT использует указанные принципы для прогрессирующей передачи (или записи в файл) вейвлетных коэффициентов, начиная с самой существенной информации. Предполагается, что вейвлетное преобразование уже применено к изображению (SPIHT является методом кодирования, и он

может работать с любым вейвлетным преобразованием) и коэффициенты  $c_{i,j}$  уже сохранены в памяти компьютера. Коэффициенты отсортированы по абсолютной величине, и они хранятся в массиве  $m$ , причем элемент  $m(k)$  содержит координаты  $(i, j)$  коэффициента  $c_{i,j}$  так, что  $|c_{m(k)}| \geq |c_{m(k+1)}|$  при всех  $k$ . В табл. 4.36 перечислены гипотетические значения 16 коэффициентов. Каждый коэффициент показан в виде 16-ти битного числа, причем самый старший бит (бит 15) содержит знак этого числа, а остальные 15 битов (с 14 по 0, сверху вниз) содержат модуль коэффициента. Первый коэффициент  $c_{m(1)} = c_{2,3}$  равен  $s1aci\dots r$  (где  $s, a, \dots$  – это биты). Второй коэффициент  $c_{m(2)} = c_{3,4}$  равен  $s1bdj\dots s$ , и так далее.

Упорядоченные данные, которые кодер должен передать, образуют последовательность  $m(k)$  вида

$(2, 3), (3, 4), (3, 2), (4, 4), (1, 2), (3, 1), (3, 3), (4, 2), \dots, (4, 3)$ .

Кроме того, необходимо передать 16 знаков и 16 коэффициентов в порядке самых значимых битов. Прямая передача должна состоять из 16 чисел

$ssssssssssssssss, \quad 1100000000000000, \quad ab11100000000000,$   
 $cdefgh1110000000, \quad ijklmnpqr1000000, \quad rstuvwxyz\dots z,$

что, конечно, слишком расточительно. Вместо этого кодер организует цикл, в котором на каждой итерации совершается *шаг сортировки* и *шаг поправки*. При первой итерации передается число  $l = 2$  (число коэффициентов  $c_{i,j}$  из нашего примера, которые удовлетворяют неравенству  $2^{14} \leq |c_{i,j}| < 2^{15}$ ), за которым следует пара координат  $(2,3)$  и  $(3,4)$ , а потом знаки первых двух коэффициентов. Это делается на первом проходе сортировки. Эта информация позволяет декодеру построить приближенную версию 16 коэффициентов следующим образом. Коэффициенты  $c_{2,3}$  и  $c_{3,4}$  строятся в виде 16-и битовых чисел  $s100\dots 0$ . Остальные 14 коэффициентов полагаются равными нулю. Таким образом, самые значимые биты самых больших коэффициентов передаются в первую очередь.

На следующем проходе кодер должен выполнить шаг поправки, но он пропускается при первой итерации цикла.

При второй итерации кодер выполняет оба шага. На шаге сортировки он передает число  $l = 4$  (равное числу коэффициентов  $c_{i,j}$ , удовлетворяющих неравенству  $2^{13} \leq |c_{i,j}| < 2^{14}$ ), за которым идут четыре пары координат  $(3,2), (4,4), (1,2)$  и  $(3,1)$ , а затем знаки этих четырех коэффициентов. На шаге поправки передаются два бита  $a$

и  $b$ . Это два тринадцатых бита двух коэффициентов, переданных при предыдущей итерации цикла.

Полученная на данный момент информация дает возможность декодеру уточнить 16 приближенных коэффициентов, построенных на предыдущей итерации. Первые 6 из них становятся равными

$$\begin{aligned} c_{2,3} &= s1a00\dots0, & c_{3,4} &= s1b00\dots0, & c_{3,2} &= s0100\dots0, \\ c_{4,4} &= s0100\dots0, & c_{1,2} &= s0100\dots0, & c_{3,1} &= s0100\dots0, \end{aligned}$$

а оставшиеся 10 коэффициентов не меняются, они по-прежнему равны 0.

На шаге сортировки при третьей итерации кодер посыпает  $l = 3$  (число коэффициентов  $c_{i,j}$ , удовлетворяющих  $2^{12} \leq |c_{i,j}| < 2^{13}$ ), потом три пары координат (3,3), (4,2) и (4,1), за которыми идут знаки этих трех коэффициентов. На шаге поправки передаются биты  $cdefgh$ , то есть двенадцатые биты шести коэффициентов, посланных на предыдущей итерации.

Полученная информация позволит декодеру еще лучше уточнить 16 приближенных коэффициентов. Первые 9 из них равны теперь

$$\begin{aligned} c_{2,3} &= s1ac0\dots0, & c_{3,4} &= s1bd0\dots0, & c_{3,2} &= s01e0\dots0, \\ c_{4,4} &= s01f0\dots0, & c_{1,2} &= s01g0\dots0, & c_{3,1} &= s01h0\dots0, \\ c_{3,3} &= s0010\dots0, & c_{4,2} &= s0010\dots0, & c_{4,1} &= s0010\dots0, \end{aligned}$$

а другие коэффициенты остаются нулевыми.

Теперь легко понять основные шаги кодера SPIHT. Они приводятся ниже.

**Шаг 1:** Для заданного сжимаемого изображения вычислить его вейвлетное преобразование, используя подходящие вейвлетные фильтры, разложить его на коэффициенты преобразования  $c_{i,j}$  и представить их в виде целых чисел фиксированной разрядности. (Здесь мы используем термины *пикセル* и *коэффициент* для обозначения одних и тех же объектов.) Предположим, что коэффициенты представлены в виде целых чисел со знаком, разрядность которых равна 16, причем самый левый бит является знаковым, а в остальных двоичных 15 разрядах записан модуль этого числа. (Отметим, что такое представление отличается от комплементарного представления чисел со знаком, которое традиционно применяется в компьютерах.) Значения этих чисел меняются в диапазоне от  $-(2^{15} - 1)$  до  $(2^{15} - 1)$ . Присвоим переменной  $n$  значение  $\lfloor \log_2 \max_{i,j} (c_{ij}) \rfloor$ . В нашем примере  $n = \lfloor \log_2 (2^{15} - 1) \rfloor = 14$ .

*Шаг 2:* Сортировка. Передать число  $l$  коэффициентов  $c_{i,j}$ , которые удовлетворяют неравенству  $2^n \leq |c_{i,j}| < 2^{n+1}$ . Затем передать  $l$  пар координат и  $l$  знаков этих коэффициентов.

*Шаг 3:* Поправка. Передать  $(n - 1)$ -ые самые старшие биты всех коэффициентов, удовлетворяющих неравенству  $|c_{i,j}| \geq 2^n$ . Эти коэффициенты были выбраны на шаге сортировки *предыдущей* итерации цикла (не этой итерации!).

*Шаг 4:* Итерация. Уменьшить  $n$  на 1. Если необходимо сделать еще одну итерацию, пойти на *Шаг 2*.

Обычно последняя итерация совершается при  $n = 0$ , но кодер может остановиться раньше. В этом случае наименее важная часть информации (некоторые менее значимые биты всех вейвлетных коэффициентов) не будет передаваться. В этом заключается естественное отбрасывание части информации в методе SPIHT. Это эквивалентно скалярному квантованию, но результат получается лучше, поскольку коэффициенты передаются в упорядоченной последовательности. В альтернативе кодер передает весь образ (то есть, все биты всех вейвлетных коэффициентов), а декодер может остановить процесс декодирования в любой момент, когда восстанавливаемое изображение достигло требуемого качества. Это качество или предопределется пользователем, или устанавливается декодером автоматически по затраченному времени.

#### 4.9.1. Алгоритм сортировки разделением множеств

Описанный выше алгоритм очень прост, так как в нем предполагалось, что коэффициенты были отсортированы (упорядочены) до начала цикла. В принципе, изображение может состоять из  $1K \times 1K$  пикселов или даже больше, в нем может быть более миллиона коэффициентов, и их сортировка может оказаться весьма медленной процедурой. Вместо сортировки коэффициентов алгоритм SPIHT использует тот факт, что сортировка делается с помощью сравнения в каждый момент времени двух элементов, а каждый результат сравнения – это просто ответ: да или нет. Поэтому, если кодер и декодер используют один и тот же алгоритм сортировки, то кодер может просто послать декодеру последовательность результатов сравнения да/нет, а декодер будет дублировать работу кодера. Это верно не только для сортировки, но и для любого алгоритма, основанного на сравнениях или на любом принципе ветвления.

Настоящий алгоритм, используемый методом SPIHT, основан на том, что нет необходимости сортировать *все* коэффициенты. Главной задачей этапа сортировки на каждой итерации является выявление коэффициентов, удовлетворяющих неравенству  $2^n \leq |c_{i,j}| < 2^{n+1}$ . Эта задача делится на две части. Для данной величины  $n$ , если коэффициент  $c_{i,j}$  удовлетворяет неравенству  $|c_{i,j}| \geq 2^n$ , то он называется *существенным*. В противном случае он называется *несущественным*. На первом шаге относительно немного коэффициентов будут существенными, но их число будет возрастать от итерации к итерации, так как число  $n$  станет убывать. Сортировка должна определить, какие существенные коэффициенты удовлетворяют второму неравенству  $|c_{i,j}| < 2^{n+1}$  и передать их координаты декодеру. В этом заключается важная часть алгоритма, используемого в SPIHT.

Кодер разделяет все коэффициенты на некоторое количество множеств  $T_k$  и выполняет тест на существенность

$$\max_{(i,j) \in T_k} |c_{i,j}| \geq 2^n?$$

для каждого множества  $T_k$ . Результатом может быть «нет» (все коэффициенты из  $T_k$  являются несущественными) или «да» (некоторые коэффициенты из  $T_k$  являются существенными, то есть, само  $T_k$  существенно). Этот результат передается декодеру. Если результат был «да», то  $T_k$  делится и кодером, и декодером с помощью общего правила на подмножества, к которым применяется тот же тест на существенность. Это деление продолжается до тех пор, пока все существенные множества не будут иметь размер 1 (то есть, каждое будет содержать ровно один коэффициент, который является существенным). С помощью этого метода производится выделение существенных коэффициентов на этапе сортировки при каждой итерации.

Тест на существенность множества  $T$  можно резюмировать в следующем виде:

$$S_n(T) = \begin{cases} 1, & \max_{(i,j) \in T} |c_{i,j}| \geq 2^n, \\ 0, & \text{иначе.} \end{cases} \quad (4.16)$$

Результатом этого теста служит бит  $S_n(T)$ , который следует передать декодеру. Поскольку результат каждого теста записывается в сжатый файл, то хорошо бы минимизировать число необходимых тестов. Для достижения этой цели множества должны создаваться и

разделяться так, чтобы ожидаемые существенные множества были большими, а несущественные множества содержали бы всего один элемент.

#### 4.9.2. Пространственно ориентированное дерево

Множества  $T_k$  создаются и разделяются с помощью специальной структуры данных, которая называется *пространственно ориентированным деревом*. Эта структура определяется с использованием пространственных соотношений между вейвлетными коэффициентами и различными уровнями пирамиды поддиапазонов. Многочисленные наблюдения показывают, что поддиапазоны каждого уровня пирамиды демонстрируют определенную пространственную симметрию (см. рис. 4.7b). Любые пространственные особенности изображения такие, как прямые края, равномерные области, остаются легко различимыми на всех уровнях.

Пространственно ориентированные деревья проиллюстрированы на рис. 4.37 для изображения размером  $16 \times 16$ . На этом рисунке показаны два уровня, уровень 1 (высокочастотный) и уровень 2 (низкочастотный). Каждый уровень разделен на 4 поддиапазона. Поддиапазон LL2 (низкой частоты) разделен на 4 группы из  $2 \times 2$  коэффициентов в каждой группе. На рис. 4.37a обозначена верхняя левая группа, а на рис. 4.37b выделена нижняя правая группа. В каждой группе (за исключением самой верхней левой) каждый из 4 ее коэффициентов становится корнем пространственно ориентированного дерева. Стрелками показан пример связи между различными уровнями деревьев. Жирные стрелки указывают, как каждая группа из  $4 \times 4$  коэффициентов уровня 2 становится родителем для четырех таких же групп из уровня 1. В общем случае, коэффициент с координатами  $(i, j)$  является родителем четырех коэффициентов с координатами  $(2i, 2j)$ ,  $(2i + 1, 2j)$ ,  $(2i, 2j + 1)$  и  $(2i + 1, 2j + 1)$ .

Корни пространственно ориентированных деревьев из нашего примера находятся в поддиапазоне LL2 (в общем случае они находятся в самом верхнем левом поддиапазоне LL, который может иметь любые размеры), причем каждый вейвлетный коэффициент уровня 1 за исключением серого коэффициента (а также исключая листья) служит корнем некоторого пространственно ориентированного поддерева. Листья всех этих деревьев расположены на уровне 1 пирамиды поддиапазонов.

В нашем примере поддиапазон  $LL2$  размера  $4 \times 4$  разделен на четыре группы по  $2 \times 2$  коэффициентов, и три коэффициента из этих четырех становятся корнями деревьев. Поэтому общее число деревьев в нашем примере равно 12. В общем случае число деревьев равно  $3/4$  от размера высшего поддиапазона  $LL$ .

Каждый из 12 корней поддиапазона  $LL2$  в нашем примере является родителем четырех потомков, которые расположены на том же уровне. Однако потомки этих потомков уже расположены на уровне 1. Это правило остается верным и в общем случае. Корни деревьев расположены в самом высоком уровне вместе со своими непосредственными потомками, а все следующие потомки любых четырех коэффициентов уровня  $k$  расположены в уровне  $k - 1$ .

Мы будем использовать термин *отпрыск* для четырех непосредственных потомков (детей) узла, а всех потомков данного узла будем называть *прямыми потомками*. Алгоритм сортировки разделением множеств использует следующие четыре множества координат.

1.  $\mathcal{O}(i, j)$  : Множество координат четырех отпрысков узла  $(i, j)$ . Если узел  $(i, j)$  является листом пространственно ориентированного дерева, то множество  $\mathcal{O}(i, j)$  пусто.

2.  $\mathcal{D}(i, j)$  : Множество координат всех прямых потомков узла  $(i, j)$ .

3.  $\mathcal{H}(i, j)$  : Множество координат корней всех пространственно ориентированных деревьев ( $3/4$  от числа вейвлетных коэффициентов в самом верхнем поддиапазоне  $LL$ ).

4.  $\mathcal{L}(i, j)$  : Разность множеств  $\mathcal{D}(i, j) - \mathcal{O}(i, j)$ . Это множество содержит всех потомков узла  $(i, j)$  за вычетом четырех его отпрысков.

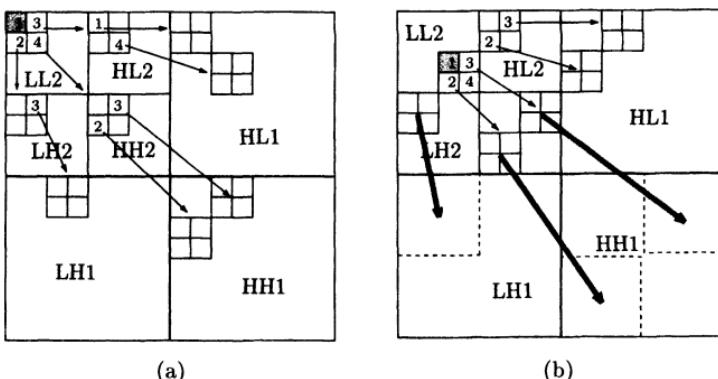


Рис. 4.37. Пространственно ориентированные деревья для SPIHT.



Пространственно ориентированные деревья используются для создания и разбиения множеств  $T_k$ . Это делается с помощью следующих правил.

1. Начальными множествами являются  $\{(i, j)\}$  и  $\mathcal{D}(i, j)$  при всех  $(i, j) \in \mathcal{H}$ . В нашем примере было 12 корней, значит, в начале будет 24 множества: 12 множеств, состоящих из одних корней и еще 12 множеств, состоящих из всех прямых потомков этих корней.
2. Если множество  $\mathcal{D}(i, j)$  является существенным, то его разбивают на  $\mathcal{L}(i, j)$  плюс еще четыре одноэлементных множества с четырьмя отпрысками  $(i, j)$ . Другими словами, если некоторый прямой потомок узла  $(i, j)$  является существенным, то его четыре отпрыска становятся четырьмя новыми множествами, а все его остальные прямые потомки образуют другое множество (тест на существенность находится в правиле 3).
3. Если множество  $\mathcal{L}(i, j)$  является существенным, то оно разбивается на четыре множества  $\mathcal{D}(k, l)$ , где  $(k, l)$  – это отпрыски узла  $(i, j)$ .

Мы объяснили, что такое пространственно ориентированные деревья, и описали правила разделения множеств. Теперь можно приступить к разбору алгоритма кодирования.

#### 4.9.3. Кодирование в алгоритме SPIHT

Прежде всего отметим, что кодер и декодер должны использовать единый тест при проверке множеств на существенность. Алгоритм кодирования использует три списка, которые называются: *список существенных пикселов* (LSP, list of significant pixels), *список несущественных пикселов* (LIP, list of insignificant pixels) и *список несущественных множеств* (LIS, list of insignificant sets). В эти списки заносятся координаты  $(i, j)$  так, что в списках LIP и LSP они представляют индивидуальные коэффициенты, а в списке LIS они представляют или множество  $\mathcal{D}(i, j)$  (запись типа A), или множество  $\mathcal{L}(i, j)$  (запись типа B).

Список LIP содержит координаты коэффициентов, которые были несущественными на предыдущей стадии сортировки. В текущей стадии они проверяются, и если множество является существенным, то они перемещаются в список LSP. Аналогично множества из LIS тестируются в последовательном порядке, и если обнаруживается, что множество стало существенным, то оно удаляется из LIS и разделяется. Новые подмножества, состоящие более чем из одного элемента, помещаются обратно в список LIS, где они позже будут подвергнуты тестированию, а одноэлементные подмножества проверя-

ются и добавляются в список LSP или LIP в зависимости от результатов теста. Стадия поправки передает  $n$ -ный самый старший бит записей из списка LSP.

На рис. 4.38 показаны детали алгоритма. На рис. 4.39 дана упрощенная версия этого алгоритма.

1. Инициализация: Присвоить переменной  $n$  значение  $\lfloor \log_2 \max_{i,j} (c_{ij}) \rfloor$  и передать  $n$ . Сделать список LSP пустым. Поместить в список LIP координаты всех корней  $(i, j) \in \mathcal{H}$ . Поместить в список LIS координаты корней  $(i, j) \in \mathcal{H}$ , у которых имеются прямые потомки.
2. Сортировка:
  - 2.1. для каждой записи  $(i, j)$  из списка LIP выполнить:
    - 2.1.1. выдать на выход  $S_n(i, j)$ ;
    - 2.1.2. если  $S_n(i, j) = 1$ , то переместить  $(i, j)$  в список LSP и выдать на выход знак  $c_{i,j}$ ;
  - 2.2. для каждой записи  $(i, j)$  из списка LIS выполнить:
    - 2.2.1. если запись типа  $A$ , то
      - выдать на выход  $S_n(\mathcal{D}(i, j))$ ;
      - если  $S_n(\mathcal{D}(i, j)) = 1$ , то
        - для каждого  $(k, l) \in \mathcal{O}(i, j)$  выполнить
          - выдать на выход  $S_n(k, l)$ ;
          - если  $S_n(k, l) = 1$ , то добавить  $(k, l)$  в список LSP, выдать на выход знак  $c_{k,l}$ ;
          - если  $S_n(k, l) = 0$ , то добавить  $(k, l)$  в список LIP;
        - если  $\mathcal{L}(i, j) \neq 0$ , переместить  $(i, j)$  в конец LIS в виде записи типа  $B$  и перейти к шагу 2.2.2.; иначе удалить  $(i, j)$  из списка LIS;
      - 2.2.2. если запись имеет тип  $B$ , то
        - выдать на выход  $S_n(\mathcal{L}(i, j))$ ;
        - если  $S_n(\mathcal{L}(i, j)) = 1$ , то
          - добавить каждую  $(k, l) \in \mathcal{O}(i, j)$  в LIS в виде записи типа  $A$ ;
          - удалить  $(i, j)$  из списка LIS;
3. Поправка: для каждой записи  $(i, j)$  из LSP, за исключением включенных в список при последней сортировке (с тем же самым  $n$ ), выдать на выход  $n$ -ый самый старший бит числа  $|c_{i,j}|$ ;
4. Цикл: уменьшить  $n$  на единицу и перейти к шагу 2, если необходимо.

Рис. 4.38. Алгоритм кодирования SPIHT.

Декодер работает по алгоритму рис. 4.38. Он всегда действует синхронно с кодером, и следующие замечания проясняют совершаемые действия.

1. Шаг 2.2 алгоритма выполняется для всех записей списка LIS. Однако шаг 2.2.1 добавляет некоторые записи в LIS (типа *B*), а шаг 2.2.2 добавляет другие записи в LIS (типа *A*). Важно понять, что эти действия применяются ко всем записям шага 2.2 в одной итерации.

2. Величина  $n$  уменьшается после каждой итерации, но это не обязательно должно продолжаться до нулевого значения. Цикл можно остановить после любой итерации, в результате произойдет сжатие с частичной потерей данных. Обычно пользователь сам определяет число совершаемых итераций, но он может также задавать допустимый порог отклонения сжатого изображения оригинала (в единицах MSE), а декодер сам определит необходимое число итераций, исходя из уравнений (4.15).

3. Кодер знает точные значения вейвлетных коэффициентов  $c_{i,j}$  и использует их для определения битов  $S_n$  (уравнение (4.16)), которые будут посыпаться в канал связи (или записываться в сжатый файл). Эти биты будут подаваться на вход декодера, который будет их использовать для вычисления значений  $c_{i,j}$ . Алгоритм, выполняемый декодером, в точности совпадает с алгоритмом рис. 4.38, но слово «выход» следует заменить на «вход».

4. Отсортированная информация, ранее обозначаемая  $m(k)$ , восстанавливается, когда координаты существенных коэффициентов добавляются в список LSP на шаге 2.1.2 и 2.2.1. Это означает, что вейвлетные коэффициенты с координатами из списка LSP, упорядочены в соответствии с условием

$$\lfloor \log_2 |c_{m(k)}| \rfloor \geq \lfloor \log_2 |c_{m(k+1)}| \rfloor$$

для всех значений  $k$ . Декодер восстанавливает этот порядок, так как все три списка (LIS, LIP и LSP) обновляются в той же последовательности, в которой это делает кодер (напомним, что декодер работает синхронно с кодером). Когда декодер вводит данные, эти три списка идентичны спискам кодер в тот момент, когда он выводит эти данные.

5. Кодер начинает работать с готовыми коэффициентами  $c_{i,j}$  вейвлетного преобразования образа; он никогда не «видел» настоящего изображения. А декодер должен показывать изображение и обновлять его после каждой итерации. При каждой итерации, когда координаты  $(i, j)$  коэффициента  $c_{i,j}$  помещаются в список LSP в

качестве записи, становится известно (и кодеру и декодеру), что  $2^n \leq |c_{i,j}| < 2^{n+1}$ .

1. Установить порог. Поместить в LIP коэффициенты всех корневых узлов. Поместить в LIS все деревья (присвоив им тип  $D$ ). Сделать список LSP пустым.
2. Сортировка: Проверить все коэффициенты из LSP на существенность:
  - 2.1. Если он существенный, то выдать на выход 1, затем выдать на выход бит знака и переместить этот коэффициент в LSP.
  - 2.2. Если он несущественный, то выдать на выход 0.
3. Проверить на существенность все деревья из LIS в соответствии с типом дерева:
  - 3.1. Для деревьев типа  $D$ :
    - 3.1.1. Если оно существенное, то выдать на выход 1 и закодировать его первых потомков:
      - 3.1.1.1. Если первый потомок существенный, то выдать на выход 1, затем выдать на выход бит знака и добавить его в список LSP.
      - 3.1.1.2. Если первый потомок несущественный, то выдать на выход 0 и добавить его в LIP.
      - 3.1.1.3. Если этот потомок имеет прямых потомков, переместить дерево в конец списка LIP, присвоив ему тип  $L$ ; в противном случае удалить его из LIS.
    - 3.1.2. Если оно несущественное, то выдать на выход 0.
  - 3.2. Для деревьев типа  $L$ :
    - 3.2.1. Если оно существенное, то выдать на выход 1, добавить всех первых потомков в конец списка LIS в виде записи с типом  $D$  и удалить родительское дерево из LIS.
    - 3.2.2. Если оно несущественное, то выдать на выход 0.
4. Цикл: уменьшить порог на единицу и перейти к шагу 2, если необходимо.

Рис. 4.39. Упрощенный алгоритм кодирования SPIHT.

В результате, лучшим приближенным значением  $\hat{c}_{i,j}$  этого коэффициента может служить середина между числами  $2^n$  и  $2^{n+1} = 2 \times 2^n$ . Тогда декодер устанавливает  $\hat{c}_{i,j} = \pm 1.5 \times 2^n$  (знак числа  $\hat{c}_{i,j}$  вводится декодером сразу после вставки). Во время этапа поправки, когда декодер вводит истинное значение  $n$ -го бита коэффициента  $c_{i,j}$ , он

исправляет значение  $1.5 \times 2^n$ , добавляя к нему  $2^{n-1}$ , если вводимый бит равен 1, или вычитая  $2^{n-1}$ , если этот бит равен 0. Таким образом, декодер способен улучшать показываемое зрителю изображение (уменьшать его расхождение с оригиналом) после прохождения *каждого* из этапов: сортировки и поправки.

Производительность алгоритма SPIHT можно повысить с помощью энтропийного кодирования выхода, но из опытов известно, что это вносит весьма незначительное улучшение, которое не покрывают временные расходы на его выполнение кодером и декодером. Оказывается, что распределение знаков и индивидуальных битов вейвлетных коэффициентов, производимых на каждой итерации, близко к равномерному, поэтому энтропийное кодирование не дает эффекта сжатия. С другой стороны, биты  $S_n(i, j)$  и  $S_n(\mathcal{D}(i, j))$  распределены неравномерно и это может дать выигрыш при таком кодировании.

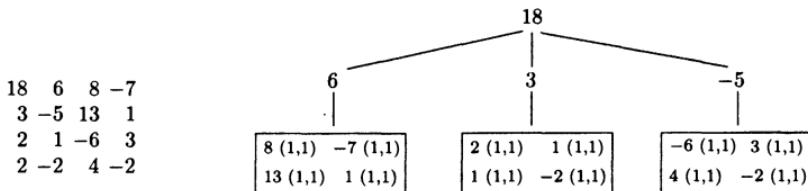


Рис. 4.40. Шестнадцать коэффициентов и пространственно ориентированное дерево.

#### 4.9.4. Пример

Предполагается, что изображение размера  $4 \times 4$  уже преобразовано и полученные 16 коэффициентов сохранены в памяти компьютера в виде целых чисел со знаком длины 6 бит (знаковый бит, за которым следует 5 битов модуля числа). Все они показаны на рис. 4.40 вместе с единственным пространственно ориентированным деревом. Алгоритм кодирования инициализирует список LIP одноэлементным множеством  $\{(1, 1)\}$ , список LIS множеством  $\{\mathcal{D}(1, 1)\}$ , а список LSP делает пустым. Наибольший коэффициент равен 18, поэтому переменная  $n$  равна  $\lfloor \log_2 18 \rfloor = 4$ . Приведем первые две итерации.

Сортировка 1:

$$2^n = 16.$$

Существен ли  $(1, 1)$ ? Да. Выход: 1.

$LSP = \{(1, 1)\}$ . Выход: бит знака: 0.

Существенно ли  $\mathcal{D}(1, 1)$ ? Нет. Выход: 0.



$LSP = \{(1, 1)\}$ ,  $LIP = \emptyset$ ,  $LIS = \{\mathcal{D}(1, 1)\}$ .

На выходе три бита.

Поправка: нет ничего на выходе (этот шаг работает с коэффициентами, отсортированными при итерации  $n - 1$ ).

Уменьшаем  $n$  до 3.

Сортировка 2:

$$2^n = 8.$$

Существенно ли  $\mathcal{D}(1, 1)$ ? Да. Выход: 1.

Существен ли  $(1, 2)$ ? Нет. Выход: 0.

Существен ли  $(2, 1)$ ? Нет. Выход: 0.

Существен ли  $(2, 2)$ ? Нет. Выход: 0.

$LIP = \{(1, 2), (2, 1), (2, 2)\}$ ,  $LIS = \{\mathcal{L}(1, 1)\}$ .

Существенно ли  $\mathcal{L}(1, 1)$ ? Да. Выход: 1.

$LIS = \{\mathcal{D}(1, 2), \mathcal{D}(2, 1), \mathcal{D}(2, 2)\}$ .

Существенно ли  $\mathcal{D}(1, 2)$ ? Да. Выход: 1.

Существен ли  $(1, 3)$ ? Да. Выход: 1.

$LSP = \{(1, 1), (1, 3)\}$ . Выход: бит знака: 1.

Существен ли  $(2, 3)$ ? Да. Выход: 1.

$LSP = \{(1, 1), (1, 3), (2, 3)\}$ . Выход: бит знака: 1.

Существен ли  $(1, 4)$ ? Нет. Выход: 0.

Существен ли  $(2, 4)$ ? Нет. Выход: 0.

$LIP = \{(1, 2), (2, 1), (2, 2), (1, 4), (2, 4)\}$ ,

$LIS = \{\mathcal{D}(2, 1), \mathcal{D}(2, 2)\}$ .

Существенно ли  $\mathcal{D}(2, 1)$ ? Нет. Выход: 0.

Существенно ли  $\mathcal{D}(2, 2)$ ? Нет. Выход: 0.

$LIP = \{(1, 2), (2, 1), (2, 2), (1, 4), (2, 4)\}$ ,

$LIS = \{\mathcal{D}(2, 1), \mathcal{D}(2, 2)\}$ ,

$LSP = \{(1, 1), (1, 3), (2, 3)\}$ .

Четырнадцать битов на выходе.

Поправка 2: после итерации 1, в списке LSP находится запись  $(1, 1)$ , чье значение  $18 = 10010_2$ .

Один бит на выходе.

Уменьшаем  $n$  до 2.

Сортировка 3:

$$2^2 = 4.$$

Существен ли  $(1, 2)$ ? Да. Выход: 1.

$LSP = \{(1, 1), (1, 3), (2, 3), (1, 2)\}$ . Выход: бит знака: 1.

Существен ли  $(2, 1)$ ? Нет. Выход: 0.

Существен ли  $(2, 2)$ ? Да. Выход: 1.

$LSP = \{(1, 1), (1, 3), (2, 3), (1, 2), (2, 2)\}$ . Выход: бит знака: 0.

Существен ли  $(1, 4)$ ? Да. Выход: 1.

$LSP = \{(1, 1), (1, 3), (2, 3), (1, 2), (2, 2), (1, 4)\}$ . Выход: бит знака: 1.

Существен ли  $(2, 4)$ ? Нет. Выход: 0.

LIP =  $\{(2, 1), (2, 4)\}$ .

Существенно ли  $\mathcal{D}(2, 1)$ ? Нет. Выход: 0.

Существенно ли  $\mathcal{D}(2, 2)$ ? Да. Выход: 1.

Существен ли  $(3, 3)$ ? Да. Выход: 1.

LSP =  $\{(1, 1), (1, 3), (2, 3), (1, 2), (2, 2), (1, 4), (3, 3)\}$ . Выход: бит знака: 0.

Существен ли  $(4, 3)$ ? Да. Выход: 1.

LSP =  $\{(1, 1), (1, 3), (2, 3), (1, 2), (2, 2), (1, 4), (3, 3), (4, 3)\}$ . Выход: бит знака: 1.

Существен ли  $(3, 4)$ ? Нет. Выход: 0.

LIP =  $\{(2, 1), (2, 4), (3, 4)\}$ .

Существен ли  $(4, 4)$ ? Нет. Выход: 0.

LIP =  $\{(2, 1), (2, 4), (3, 4), (4, 4)\}$ .

LIP =  $\{(2, 1), (2, 4), (3, 4), (4, 4)\}$ , LIS =  $\{\mathcal{D}(2, 1)\}$ ,

LSP =  $\{(1, 1), (1, 3), (2, 3), (1, 2), (2, 2), (1, 4), (3, 3), (4, 3)\}$ .

Шестнадцать битов на выходе.

Поправка 3: после итерации 2, в списке LSP записаны  $(1, 1)$ ,  $(1, 3)$  и  $(2, 3)$ , со значениями, соответственно,  $18 = 10010_2$ ,  $8 = 1000_2$  и  $13 = 1101_2$ .

Три бита на выходе.

После двух итераций общее число битов на выходе равно 37.

#### 4.9.5. QTCQ

Близким к методу SPIHT является алгоритм QTCQ (quadtree classification and trellis coding, классификация четвертичных деревьев и решетчатое кодирование) из работы [Banister, Fischer 99], который использует меньше списков, чем SPIHT, и явно формирует классы вейвлетных коэффициентов для дальнейшего квантования с помощью методов ACTCQ и TCQ из [Joshi, Crump, Fischer 93].

Этот метод основан на пространственно ориентированных деревьях, построенных для SPIHT. Этот тип деревьев является особым случаем четвертичных деревьев. Алгоритм кодирования является итеративным. На  $n$ -той итерации, если обнаружено, что некоторый элемент этого четвертичного дерева является существенным, то четырем верхним элементам дерева присваивается класс  $n$ . Одновременно эти элементы становятся корнями четырех новых четвертичных деревьев. Каждое из полученных деревьев проверяется на существенность, перемещаясь вниз по дереву пока не будут обнаружены все существенные элементы. Все вейвлетные коэффициенты, отнесенные к классу  $n$ , сохраняются в списке пикселов (LP, list of pixels).

В начале список LP заполнен всеми вейвлетными коэффициентами из низкочастотного поддиапазона LFS (lowest frequency subband). Тест на существенность совершается с помощью функции  $S_T(k)$ , которая определяется по формуле

$$S_T(k) = \begin{cases} 1, & \max_{(i,j) \in k} |C_{i,j}| \geq T, \\ 0, & \text{иначе,} \end{cases}$$

где  $T$  – это текущий порог существенности, а  $k$  – дерево вейвлетных коэффициентов. Алгоритм QTCQ, использующий этот тест, приведен на рис. 4.41.

Алгоритм декодирования QTCQ устроен похоже. Все строки с выводом данных надо заменить на ввод этих данных, а кодирование ACTCQ следует заменить на декодирование ACTCQ.

1. Инициализация:

Заполнить список LP всеми  $C_{i,j}$  из LFS,  
Заполнить список LIS всеми родительскими узлами,  
Выдать на выход  $n = \lfloor \log_2 (\max |C_{i,j}| / q) \rfloor$ .  
Задать порог  $T = q2^n$ , где  $q$  – множитель качества.

2. Сортировка:

Для каждого узла  $k$  из списка LIS выполнить

Выдать на выход  $S_T(k)$   
Если  $S_T(k) = 1$ , то  
    Для каждого отпрска  $k$  выполнить  
        Переместить коэффициенты в список LP  
        Добавить в список LIS в виде нового узла  
    Конец Для  
    Удалить  $k$  из LIS  
Конец Если

Конец Для

3. Квантование: Для каждого элемента из LP,  
Квантовать и кодировать его с помощью ACTCQ.  
(Использовать размер шага TCQ  $\Delta = |\alpha \cdot q|$ ).

4. Обновление: Удалить все элементы из LP. Присвоить  $T = T/2$ .  
Идти на шаг 2.

Рис. 4.41. Кодирование QTCQ (псевдокод).

Реализация метода QTCQ, приведенная в [Banister, Fischer 99], не предусматривает прогрессирующей передачи изображений, однако авторы утверждают, что такое свойство может быть добавлено в программу реализации.

*Что такое вейвлеты? Вейвлеты расширяют анализ Фурье. Как вычисляются вейвлеты? Быстрые преобразования делают это.*

*— Ив Нивергельт, «Вейвлеты делают это проще».*

## ГЛАВА 5

### СЖАТИЕ ВИДЕО

В середине сороковых годов прошлого века, сразу после окончания Второй мировой войны, группа молодых инженеров, среди которых были Ален Тьюринг, Джон Атанасов, Джон Мошли и Преспер Эккман, начала разрабатывать первые электронные компьютеры. Пионерам-изобретателям компьютеры представлялись быстрыми и надежными устройствами для совершения вычислений над числами. Однако очень скоро многие разработчики компьютеров осознали, что их можно применять не только в узковычислительных целях. Первые нечисловые приложения, разработанные в пятидесятые годы, обрабатывали тексты, потом пришла очередь изображений (шестидесятые годы), компьютерной анимации (семидесятые годы) и оцифрованного звука (восьмидесятые годы). В настоящее время компьютеры в основном используются для коммуникаций и развлечений, поэтому они выполняют всевозможные мультимедийные приложения, в которых приходится обрабатывать тексты, изображения, видео и звук. Все эти оцифрованные массивы приходится отображать, редактировать и передавать по линиям связи другим пользователям.

Любые типы компьютерных данных могут только существенно выиграть от применения эффективного сжатия. Однако, особенно полезной компрессия становится при работе с файлами, содержащими видео. Уже файл единичного изображения имеет достаточно большой объем. Что же говорить о видеофайле, который состоит из тысяч изображений? При обработке изображений, как мы знаем, часто применяется сжатие с потерями. При работе же с видео это становится просто необходимым. Сжатие изображений основывается на корреляции пикселов, а компрессия видео может использовать не только корреляцию близких пикселов каждого кадра, но и корреляцию между последовательными кадрами (см. §5.1).

В этой небольшой главе излагаются основные принципы и технические приемы, используемые при сжатии видеоданных. Мы не будем рассматривать здесь конкретные алгоритмы. Тем, кто хочет ближе познакомиться с методом сжатия MPEG или с другими алгоритмами, мы рекомендуем обратиться к книге [Salomon 2000].



## 5.1. Основные принципы

Сжатие видео основано на двух важных принципах. Первый – это пространственная избыточность, присущая каждому кадру видеоряда. А второй принцип основан на том факте, что большую часть времени каждый кадр похож на своего предшественника. Это называется *временная избыточность*. Таким образом, типичный метод сжатия видео начинает с кодирования первого кадра с помощью некоторого алгоритма компрессии изображения. Затем следует кодировать каждый последующий кадр, находя расхождение или разность между этим кадром и его предшественником и кодируя эту разность. Если новый кадр сильно отличается от предыдущего (это происходит, например, с первым кадром последовательности), то его можно кодировать независимым образом. В литературе по сжатию видеоданных, кадр, кодируемый с помощью своего предшественника называется *внутренним*. В противном случае он называется *внешним кадром*.

Сжатие видео, обычно, допускает частичную потерю информации. Кодирование кадра  $F_i$  с помощью его предшественника  $F_{i-1}$  вносит определенные искажения. Затем кодирование кадра  $F_{i+1}$  на основе кадра  $F_i$  добавляет еще большее искажение. Даже при использовании сжатия без потерь, это может привести к потере некоторых битов данных. То же может случиться при передаче файла или после долгого хранения ленты на полке. Если кадр  $F_i$  потерял некоторые биты, то все последующие кадры будут иметь искажения вплоть до следующего внешнего кадра. Это приводит к нежелательному накапливанию ошибок. Поэтому необходимо регулярно использовать внешние кадры при кодировании последовательного видеоряда, а не только в его начале. Внешние кадры обозначаются символом  $I$ , а внутренние кадры – символом  $P$  (от «предсказанный»).

Если эта идея принимается, то возможно дальнейшее обобщение данной концепции внутренних кадров. Такой кадр можно кодировать с использованием одного из его предшественников, а также с помощью некоторого последующего кадра. Очевидно, кодер не должен использовать информацию, недоступную декодеру, но сжатие видео имеет определенные особенности, поскольку оно вовлекает большие объемы данных. Для зрителя не важно, если кодер работает медленно. Важно, чтобы декодер работал быстро. Обычно сжатые видеоданные записываются на жесткий диск или на DVD для дальнейшего воспроизведения. Кодер может затратить часы на кодирование, а декодер должен работать со скоростью, не меньшей

чем стандартная скорость смены кадров (довольно большое число кадров должно появиться на экране за секунду времени для нормального восприятия). Поэтому типичный видеодекодер работает в параллельном режиме, то есть, одновременно в работе находится несколько кадров.

Имея это в виду, представим себе, что кодер кодирует кадр 2 с помощью кадров 1 и 3, а затем записывает сжатую информацию в выходной поток в последовательности 1, 3, 2. Декодер читает их в этом порядке, параллельно декодирует кадры 1 и 3, а потом декодирует кадр 2 на основе кадров 1 и 3. Конечно, эти кадры должны быть правильно помечены. Кадры, которые кодируются с применением прошлых и будущих кадров обозначаются буквой *B* (*bidirectional, двунаправленный*).

Предсказание кадров с помощью их предшественников имеет смысл, если движение объектов на картинке постепенно открывает фон изображения. Такие области могут быть лишь частично отображены на текущем кадре, а более подробная информация может быть получена из следующего кадра. Поэтому этот кадр является естественным кандидатом для предсказания этой области на текущем кадре.

Идея кадров типа *B* настолько полезна, что большинство кадров сжимается с помощью этого приема. Итак, мы имеем дело с кадрами трех типов: *I*, *B* и *P*. Кадр *I* кодируется независимо от всех остальных кадров. Кадр *P* кодируется на основе кадров *I* или *P*. Наконец, кодирование кадра типа *B* использует предыдущий кадр и следующий кадр типа *I* или *P*. На рис. 5.1а показан пример последовательности кадров в том порядке, в котором они генерируются кодером (и входом декодера). На рис. 5.1б отображена последовательность кадров, которая поступает на выход декодера и отображается на экране дисплея. Ясно, что кадр с номером 2 должен быть отображен ранее кадра 5. Следовательно, каждый кадр должен иметь две метки, а именно, время кодирования и время отображения.

Для начала рассмотрим два интуитивных метода сжатия видео.

**Прореживание:** Кодер выбирает кадры через одного и записывает их в сжатый поток. Это приводит к фактору сжатия 2. Декодер принимает кадры и дублирует их подряд два раза.

**Вычитание:** Кадр сравнивается со своим предшественником. Если разница между ними мала (всего в нескольких пикселях), то кодер кодирует только эти отличающиеся пиксели, то есть, записывает в выходной поток три числа для каждого из отличающихся

пикселов: его координаты и разность пикселов двух кадров. Если различие между кадрами велико, то в файл пишется текущий кадр в «сыром» виде. Вариант с частичной потерей для метода вычитания анализирует величину расхождения пикселов. Если разность между двумя значениями меньше некоторого порога, то пикселя не считаются разными.

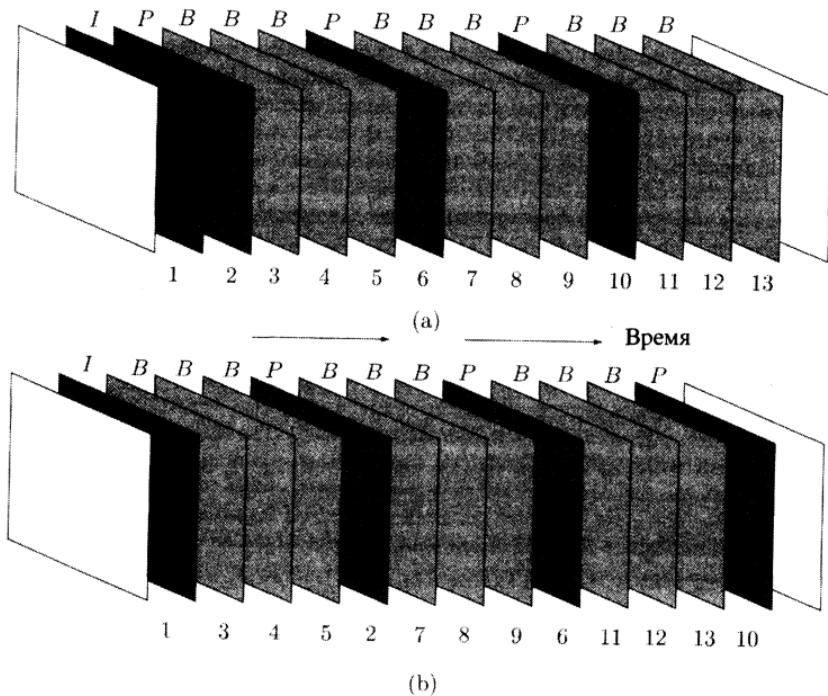


Рис. 5.1. (а) Порядок кодирования. (б) Порядок отображения.

**Вычитание по блокам:** Этот метод является развитием метода вычитаний. Изображение делится на блоки пикселов и каждый блок  $B$  сравнивается с соответствующим блоком  $P$  предыдущего кадра. Если число различающихся пикселов этих блоков не превосходит некоторую заданную величину, то блок  $B$  сжимается в виде записи координат отличных пикселов и значений их разностей. Преимущество такого подхода проявляется в том, что координаты в блоке являются малыми числами, и такие числа будут записаны всего один раз для всего блока. Недостаток такого решения состоит в том, что в противном случае приходится записывать в файл все

пиксели, в том числе и не меняющиеся. Этот метод весьма чувствителен к размеру блока.

**Компенсация движения:** Просмотр любого фильма наводит на мысль, что разница между последовательными кадрами мала из-за движения на сцене, перемещения камеры или в силу обеих причин. Это свойство можно использовать для улучшения сжатия. Если кодер обнаружил, что часть  $P$  предыдущего кадра, как одно целое, переместилась в новое положение на текущем кадре, то  $P$  можно сжать, записав следующие три компонента: предыдущее местоположение, новое местоположение и информацию о границе области  $P$ . Следующее обсуждение метода компенсации движения позаимствовано из [Manning 98].

В принципе, эта перемещающаяся часть может иметь любую форму. На практике мы ограничены блоками с равными сторонами (обычно это квадраты или прямоугольники). Кодер сканирует текущий кадр блок за блоком. Для каждого блока  $B$  на предыдущем кадре делается поиск тождественного блока  $C$  (если сжатие без потерь) или близкого блока (если сжатие с потерями). Если такой блок найден, то кодер записывает разность между его прошлым и настоящим положением. Эта разность имеет вид

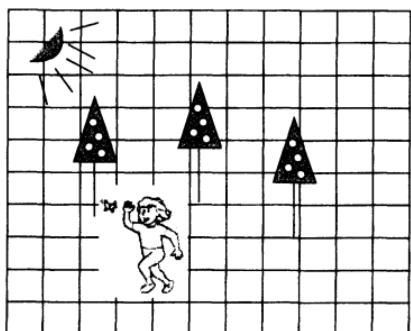
$$(C_x - B_x, C_y - B_y) = (\Delta x, \Delta y),$$

и она называется *вектором перемещения*. На рис. 5.2 показан простой пример, на котором солнце и деревья движутся вправо (из-за перемещения видеокамеры), а ребенок перемещается влево (это движение сцены).

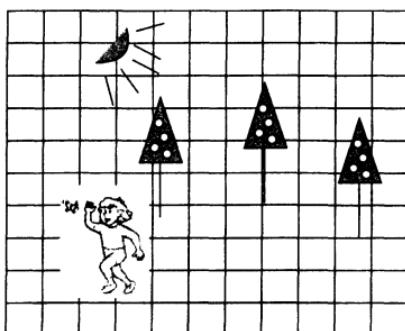
Компенсация движения будет эффективной, если объекты просто перемещаются по сцене, но не удаляются, приближаются или поворачиваются. Существенное изменение освещения сцены от кадра к кадру также снижает эффективность этого метода. Обычно метод компенсации движения используется для сжатия с потерями. В следующих абзацах детально обсуждаются основные нюансы этого метода.

**Сегментация кадров:** Текущий кадр разбивается на неперекрывающиеся равносторонние блоки. Блоки могут быть квадратными или прямоугольными. Во втором случае предполагается, что движение на экране происходит в основном по горизонтали, и тогда горизонтальные блоки уменьшают число векторов перемещения без ухудшения коэффициента сжатия. Размер блока очень важен, поскольку большие блоки уменьшают вероятность обнаружения со-

впадений, а маленькие блоки приводят к большому числу векторов перемещения. На практике применяются блоки, размеры которых являются степенями числа 2, например, 8 или 16; это упрощает программное обеспечение.



(a)



(b)

Рис. 5.2. Компенсация движения.

**Порог поиска:** Каждый блок  $B$  текущего кадра сначала сравнивается со своим двойником на предыдущем кадре. Если они совпадают или величина разности между ними меньше некоторого наперед заданного порога, то кодер считает, что данный блок не двигался.

**Поиск блока:** Эта процедура обычно занимает много времени, поэтому ее следует тщательно оптимизировать. Если  $B$  – текущий блок текущего кадра, то необходимо сделать поиск совпадающего или близкого к нему блока на предыдущем кадре. Обычно этот поиск делается в небольшой области (называемой *областью поиска*) вокруг  $B$ , которая задается с помощью параметров *максимального смещения*  $dx$  и  $dy$ . Эти параметры определяют максимальное допустимое расстояние по горизонтали и вертикали в пикселях между  $B$  и его копией на предыдущем кадре. Если блок  $B$  – это квадрат со стороной  $b$ , то область поиска состоит из  $(b + 2dx)(b + 2dy)$  пикселов (рис 5.3), по которым можно построить  $(2dx + 1)(2dy + 1)$  различных, частично перекрывающихся квадратов размера  $b \times b$ . Значит, число возможных блоков-кандидатов в этой области пропорционально  $dx dy$ .

**Измерение расхождения:** Это наиболее чувствительная часть кодера. Здесь необходимо выбрать наиболее близкий блок к исход-

ному блоку  $B$ . Эта процедура должна быть простой и быстрой, но, вместе с тем, надежной.

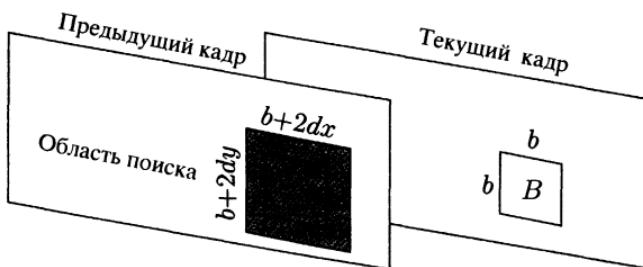


Рис. 5.3. Область поиска.

Средняя абсолютная разность (или средняя абсолютная ошибка) вычисляет среднее значение модуля разностей между пикселями  $B_{ij}$  блока  $B$  и пикселями  $C_{ij}$  блока-кандидата  $C$  по формуле

$$\frac{1}{b^2} \sum_{i=1}^b \sum_{j=1}^b |B_{ij} - C_{ij}|.$$

Для этого потребуется выполнить  $b^2$  операций вычитания, столько же операций взятия модуля числа и одной операции деления. Эта величина вычисляется для каждого из  $(2dx + 1)(2dy + 1)$  блоков-кандидатов для нахождения блока с наименьшим отклонением от блока  $B$  (обозначим его  $C_k$ ). Если это отклонение меньше заданного порога, то блок  $C_k$  выбирается в качестве похожей пары для блока  $B$ . В противном случае такой пары нет, и блок  $B$  необходимо кодировать без компенсации движения.

В этом месте можно задать законный вопрос: «Как может случиться, что некоторый блок текущего кадра не имеет подходящей похожей пары на предыдущем кадре?» Для ответа представим себе, что снимающая видеокамера движется слева направо. Новые объекты попадают в поле зрения камеры справа. Поэтому самый правый блок кадра может содержать объекты, которых не было на предыдущем кадре.

Другой мерой расхождения может служить «среднеквадратическое отклонение», в формуле которого вместо функции взятия модуля стоит возведение в квадрат разности пикселов:

$$\frac{1}{b^2} \sum_{i=1}^b \sum_{j=1}^b (B_{ij} - C_{ij})^2.$$

Функция ранжирования разностей пикселов PDC (pixel difference classificatin) определяет, сколько разностей  $|B_{ij} - C_{ij}|$  меньше, чем заданное число  $p$ .

Мера интегральной проекции вычисляется с помощью нахождения суммы строки блока  $B$ , из которой вычитается сумма соответствующей строки  $C$ . Модули результатов складываются для всех строк и к этому числу добавляется аналогичная величина, вычисленная для всех модулей разностей сумм столбцов:

$$\sum_{i=1}^b \left| \sum_{j=1}^b B_{ij} - \sum_{j=1}^b C_{ij} \right| + \sum_{j=1}^b \left| \sum_{i=1}^b B_{ij} - \sum_{i=1}^b C_{ij} \right|.$$

**Методы подоптимального поиска:** Такие методы делают поиск по части блоков среди всех  $(2dx + 1)(2dy + 1)$  кандидатов. Это ускоряет поиск подходящего схожего блока, но платой служит эффективность сжатия. Некоторые подобные методы будут обсуждаться в следующем параграфе.

**Корректировка вектора перемещения:** Если блок  $C$  выбран в качестве самого похожего на блок  $B$ , то для нахождения вектора перемещения следует найти разность между верхними левым углами блоков  $C$  и  $B$ . Независимо от выбора подходящего блока вектор перемещения может оказаться неверным из-за шума, локального минимума в кадре, или по причине недостатков самого алгоритма поиска. Тогда можно применить технику сглаживания вектора перемещения для достижения большей схожести блоков. Пространственная корреляция изображения предопределяет также корреляцию векторов перемещения. Если некоторый вектор нарушает это правило, то его следует подправить.

Этот шаг тоже достаточно продолжителен и может даже все испортить. Перемещение большинства объектов в кадре может быть медленным, гладким, но некоторые малые объекты могут двигаться быстро, скачкообразно. Подправленные векторы перемещения могут вступать в противоречие с векторами перемещения таких объектов и быть причиной ошибочно сжатых кадров.

**Кодирование векторов перемещения:** Большая часть текущего кадра (возможно, половина его) может быть преобразована в

векторы перемещения, поэтому кодирование этих векторов весьма актуально. Это кодирование должно быть без потерь. Два свойства векторов перемещения позволяют осуществить эффективное кодирование: (1) эти векторы коррелированы и (2) они имеют неравномерное распределение. При сканировании кадров блок за блоком оказывается, что прилегающие блоки обычно имеют близкие векторы перемещения. Кроме того векторы также не имеют любые направления. Они, как правило, направлены в одну, реже, в две стороны; значит, векторы распределены неравномерно.

Не существует единого общего метода кодирования, который был бы идеальным для всех случаев. Обычно применяется арифметическое кодирование, адаптивное кодирование Хаффмана, а также различные префиксные коды. Все они работают достаточно хорошо и с близкой эффективностью. Вот еще два возможных метода с хорошей степенью сжатия:

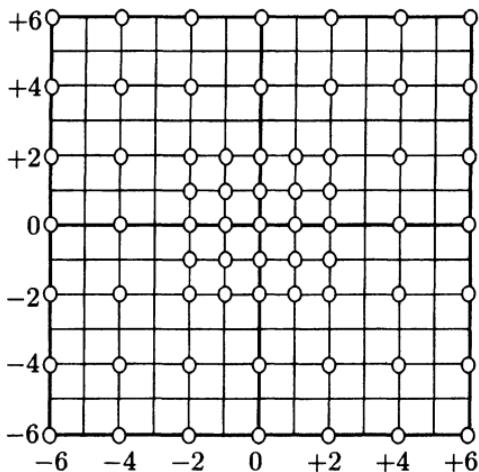
1. Спрогнозировать вектор перемещения на основе его предшественников в той же строке и том же столбце текущего кадра. Вычислить разность между вектором-прогнозом и истинным вектором и закодировать его по Хаффману. Этот метод весьма важен. Он используется в MPEG-1 и в других алгоритмах сжатия.
2. Сгруппировать векторы перемещения в блоки. Если все векторы в блоке идентичны, то блок кодируется одним вектором. Все другие блоки кодируются как в пункте 1. Каждый кодированный блок начинается соответствующим идентификатором типа.

**Кодирование ошибки предсказания:** Компенсация движения является сжатием с потерями, поскольку блок *B* обычно не совпадает с выбранным для него похожим блоком *C*. Сжатие можно улучшить, если кодировать разность между текущим несжатым кадром и его сжатым образом по блокам, причем делать это для достаточно сильно расходящихся блоков. Это часто делается с помощью кодирования преобразованных образов. Эта разность записывается в выходной файл после кода текущего кадра. Она будет использоваться декодером для улучшения отображения данного кадра.

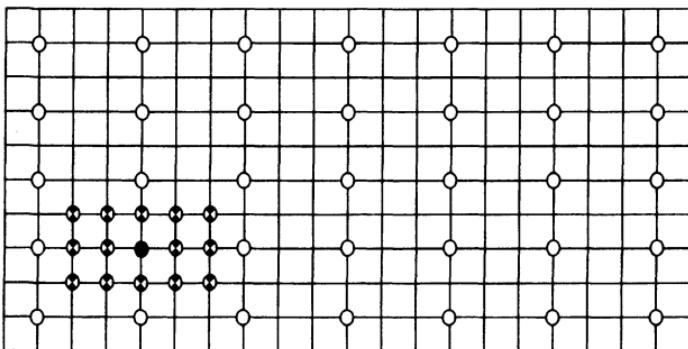
## 5.2. Методы подоптимального поиска

Сжатие видеоданных имеет много шагов и использует большие вычислительные ресурсы. Исследования в этой области направлены на оптимизацию и ускорение алгоритмов сжатия, особенно для шагов, использующих больше вычислений. Одним из таких шагов является

поиск похожего блока  $C$  из предыдущего кадра для блока  $B$  текущего кадра. Полный перебор всех возможных вариантов требует много времени, поэтому имеет смысл заняться поиском подоптимальных алгоритмов, которые делают поиск не среди всех блоков-кандидатов. Такие методы не всегда находят самый похожий блок, но они могут существенно ускорить весь процесс сжатия, за счет небольшого снижения эффективности компрессии.



(a)



○ — первый этап     ● — лучшие на первом этапе     ◊ — второй этап

(b)

Рис. 5.4. (a) Поиск с разбавленным расстоянием при  $dx = dy = 6$ .  
 (b) Локализованный поиск.

**Сигнатурные методы:** Эти методы совершают несколько шагов, сокращая число кандидатов на каждом шаге. На первом шаге все кандидаты испытываются с помощью простой и быстро вычисляемой меры расходимости, например, функцией ранжирования разностей пикселов. Только самые близкие блоки попадают на следующий шаг, где они оцениваются более сложными мерами расходимости или той же мерой, но с меньшим параметром. Сигнатурный метод может состоять из многих шагов с разными тестовыми мерами расходимости.

**Поиск с разбавленным расстоянием:** Из опыта известно, что быстро перемещающиеся объекты выглядят смазанными при воспроизведении на экране, даже если они имеют четкие очертания на каждом кадре. Это подсказывает возможный путь для отбрасывания части информации. Можно требовать хорошего совпадения для медленно перемещающихся объектов, и приблизительного совпадения для тех, что перемещаются быстро. В итоге получаем алгоритм, который оценивает для каждого блока  $B$  все ближайшие к нему блоки-кандидаты, а для более удаленных блоков он рассматривает все меньшую часть блоков. На рис. 5.4а показано, как такой метод мог бы работать для параметров максимального смещения  $dx = dy = 6$ . Общее число оцениваемых блоков  $C$  сокращается с  $(2dx + 1)(2dy + 1) = 13 \times 13 = 169$  до всего 65, то есть, до 39%!

**Локализованный поиск:** Этот метод основан на гипотезе, что если хорошее совпадение найдено, то еще лучшее совпадение, возможно, находится где-то рядом (напомним, что поиск делается среди сильно перекрывающихся блоков). Очевидный алгоритм поиска начинается с изучения разреженного семейства блоков. Затем наиболее подходящий блок из этого семейства используется в качестве центра для более тщательного поиска. На рис. 5.4б показаны два этапа поиска: первый этап рассматривает широко расположенные блоки и выбирает наиболее близкий из них, а второй тестирует каждый блок в окрестности этого хорошего блока.

**Монотонный поиск по квадрантам:** Это один из вариантов метода локализованного поиска. Сначала анализируется разреженное семейство блоков  $C$ . Для каждого блока из этого семейства вычисляется мера расходимости. Идея заключается в том, что величина этой меры увеличивается при удалении от оптимального блока, имеющего минимальное расхождение с блоком  $B$ . Это позволяет достаточно хорошо спрогнозировать область, в которой расположен оптимальный блок. На втором шаге изучается каждый блок из этой

области. На рис 5.5 показан пример поиска области размера  $2 \times 2$ . Величины расхождения показывают направление дальнейшего поиска оптимального блока.

Этот метод менее надежен, чем предыдущий, так как направление, указываемое множеством величин расхождений, может привести к другому локальному минимуму, а наилучший блок может располагаться в другом месте.

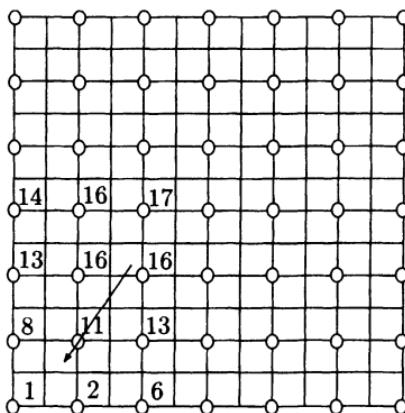


Рис. 5.5. Монотонный поиск по квадрантам.

**Зависимые алгоритмы:** Как уже выше упоминалось, движение в кадре может происходить в результате перемещения объектов съемки или самой видеокамеры. Если предположить, что объекты съемки больше, чем блок, то логично допустить, что векторы перемещения близких блоков будут коррелированы. Поэтому алгоритм поиска может оценить вектор перемещения блока  $B$  с помощью уже найденных векторов перемещения соседних к нему блоков. Затем алгоритм уточняет эту оценку тестированием соответствующих близких блоков-кандидатов  $C$ . Это соображение лежит в основе многих зависимых алгоритмов, которые могут быть пространственными или временными.

**Пространственная зависимость:** В алгоритме с пространственной зависимостью окрестность блока  $B$  текущего кадра используется для оценивания вектора перемещения этого блока. Конечно, имеется в виду окрестность, для блоков которой уже вычислены их векторы перемещения. Большинство блоков имеют восемь соседей, но использование всех этих блоков не обязательно приводит к

наилучшей стратегии (кроме того, не для всех этих соседей векторы перемещения будут уже вычислены). Если блоки сравниваются в растровом порядке, то имеет смысл использовать одного, двух или трех подходящих соседей, как показано на рис. 5.6a,b,c. Однако, в силу симметрии будет лучше использовать четырех соседей, показанных на рис. 5.6d,e. При этом можно применить метод, состоящий из трех проходов, который анализирует блоки, отмеченный на рис. 5.6f. На первом проходе сканируются черные блоки (четверть всех блоков на рисунке). Векторы перемещения для этих блоков вычисляются некоторым другим методом. На втором проходе оцениваются серые блоки (их тоже 25% от общего числа), с помощью векторов перемещения их угловых соседей. Наконец, белые блоки (их 50%) изучаются на третьем проходе. Для вычисления их векторов перемещения используются четыре соседа, прилегающие к их сторонам. Если векторы перемещений соседних блоков сильно различаются, то их не стоит использовать, а вектор перемещения блока  $B$  придется вычислять другим методом.

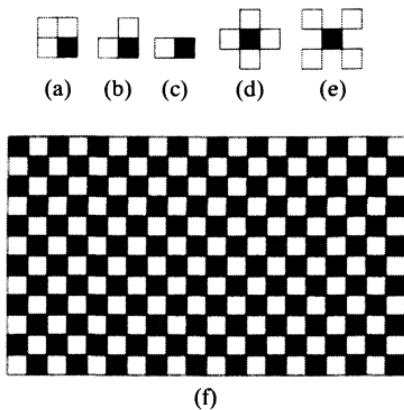


Рис. 5.6. Стратегии алгоритмов с пространственной зависимостью.

**Временная зависимость:** Вектор перемещения блока  $B$  в текущем кадре можно оценить с помощью вектора перемещения этого же блока на предыдущем кадре. Это имеет смысл, если движение в кадре предполагается плавным. После нахождения этой оценки, вектор перемещения можно уточнить другим подходящим методом.

**Вариант метода монотонного поиска по квадрантам:** Следующие подоптимальные алгоритмы используют основную идею ме-

тода монотонного поиска подходящего блока.

**Двумерный логарифмический поиск:** Этот многошаговый алгоритм сокращает область поиска на каждом шаге, пока она не сводится к одному блоку. Предположим, что текущий блок  $B$  локализован в точке  $(a, b)$  текущего кадра. Эта точка берется за центр поиска. Алгоритм зависит от параметра  $d$ , задающего удаление от центра поиска. Этот параметр контролируется пользователем. Квадратная область поиска состоит из  $(2d + 1)(2d + 1)$  блоков с центром в блоке  $B$ .

*Шаг 1:* Размер шага вычисляется по формуле

$$s = 2^{\lfloor \log_2 d \rfloor - 1},$$

и алгоритм сравнивает блок  $B$  с пятью блоками в точках с координатами  $(a, b)$ ,  $(a, b + s)$ ,  $(a, b - s)$ ,  $(a + s, b)$  и  $(a - s, b)$  на предыдущем кадре. Эти пять блоков образуют шаблон в форме знака +.

*Шаг 2:* Выбирается наилучший из этих пяти блоков. Обозначим его координаты через  $(x, y)$ . Если  $(x, y) = (a, b)$ , то  $s$  делится пополам (поэтому алгоритм называется логарифмическим). В противном случае шаг  $s$  не меняется, а центр поиска  $(a, b)$  перемещается в точку  $(x, y)$ .

*Шаг 3:* Если  $s = 1$ , то оцениваются девять блоков вокруг центра поиска  $(a, b)$ , и наилучший блок становится результатом работы алгоритма. В противном случае делается переход на Шаг 2.

Если нужный алгоритму блок выходит за пределы области поиска, то он игнорируется и не используется. Рис. 5.7 иллюстрирует случай, когда  $d = 8$ . Для простоты предполагается, что текущий блок  $B$  имеет координаты  $(0, 0)$ . Поиск ограничивается областью из  $17 \times 17$  блоков с центром в  $B$ . На шаге 1 вычисляется

$$s = 2^{\lfloor \log_2 8 \rfloor - 1} = 2^{3-1} = 4,$$

и изучаются пять блоков с координатами  $(0, 0)$ ,  $(4, 0)$ ,  $(-4, 0)$ ,  $(0, 4)$ ,  $(0, -4)$ . Предположим, что наилучшее значение имеет блок  $(0, 4)$ , который становится центром нового поиска. Теперь на втором шаге изучаются три блока (помеченные цифрой 2) с координатами  $(4, -4)$ ,  $(4, 4)$ ,  $(8, 0)$ .

Если лучшим среди них будет блок  $(4, 4)$ , то на следующем шаге будут исследоваться 2 блока с меткой 3 с координатами  $(8, 4)$  и  $(4, 8)$ , блок  $(4, 4)$  с меткой 2 и два блока  $(0, 4)$  и  $(4, 0)$ , имеющие метку 1.

Предположим, что на следующем шаге наилучшим выбором будет блок  $(4, 4)$ . Поскольку этот блок находится в центре знака +,

то после деления на 2 переменная  $s$  станет равна 4. На этом шаге исследуются блоки, помеченные цифрой 4 с центром в (4, 4). Предположим, что наилучшим блок имеет координаты (6, 4). Тогда исследуются два блока с меткой 5. Пусть опять наилучшим блоком служит (6, 4). Делим  $s$  на два и исследуем шесть блоков, помеченных цифрой 6. Диаграмма показывает, что окончательным оптимальным выбором алгоритма станет блок с координатами (7, 4).

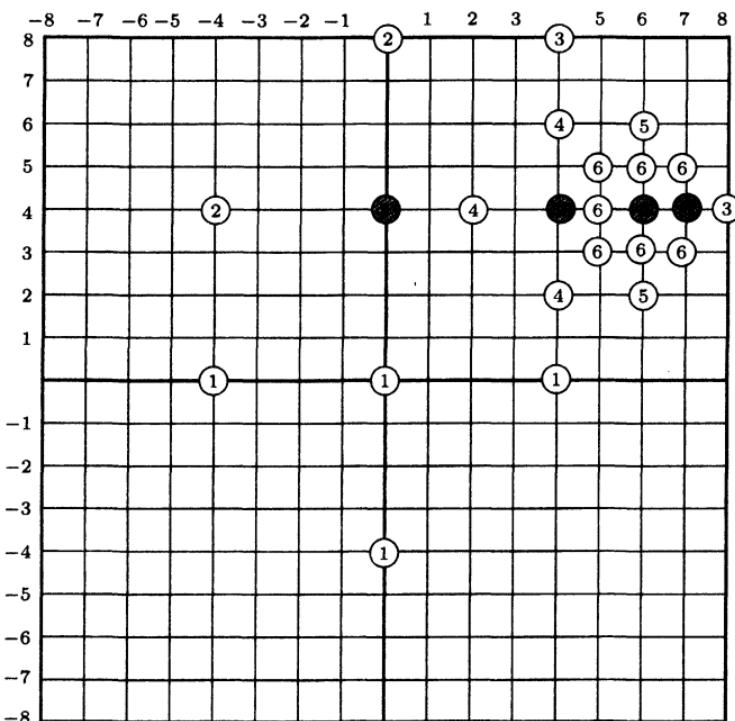


Рис. 5.7. Метод двумерного логарифмического поиска.

**Поиск за три шага:** Этот метод похож на процедуру двумерного логарифмического поиска. На каждом шаге тестируется восемь блоков вместо четырех вокруг центра поиска, после чего размер шага делится на два. Если в начале  $s = 4$ , то алгоритм завершится в три шага, что объясняет его название.

**Ортогональный поиск:** Это вариация двух алгоритмов, двумерного логарифмического поиска и поиска за три шага. На каждом шаге ортогонального алгоритма осуществляется вертикальный и



горизонтальный поиск. В начале размер шага  $s$  равен  $\lfloor (d + 1)/2 \rfloor$  и исследуется центральный блок, а также два его соседа по бокам на расстоянии  $s$ . Наилучший блок становится центром вертикального поиска, а двумя другими блоками-кандидатами становятся блоки сверху и снизу от центрального на расстоянии  $s$  от него. Лучший из них становится центром следующего поиска. Если размер шага  $s$  равен 1, то алгоритм обрывается и возвращаются координаты наилучшего блока, найденного на текущем шаге. В противном случае  $s$  делится на два, после чего совершается новый шаг, состоящий из горизонтального и вертикального поиска.

**Поиск по одному:** В этом методе снова имеется два шага, горизонтальный и вертикальный. На горизонтальном шаге исследуются все блоки области поиска, чьи координаты  $y$  имеют то же значение, что и у блока  $B$  (то есть лежат на одной горизонтали с этим блоком). Пусть некоторый блок  $H$  имеет наименьшее расхождение с  $B$ . Затем на вертикальном шаге анализируются все блоки, находящиеся на одной вертикальной оси с блоком  $H$ . Наилучший блок этой оси объявляется оптимальным и возвращается алгоритмом в качестве результата поиска. Модификация этого алгоритма повторяет это действие с последовательным сокращением области поиска.

**Перекрестный поиск:** Все этапы этого алгоритма, кроме последнего, исследуют пять блоков, находящихся по углам области в форме знака умножения  $\times$ . На каждом этапе размер шага поиска делится на два, пока он не станет равен 1. На последнем этапе в качестве области поиска используется область в форме знака  $+$  с центром в результате предыдущего этапа.

Этим методом мы завершили наш краткий обзор методов монотонного поиска по квадрантам.

**Методы иерархического поиска:** Иерархические методы основаны на преимуществе, которое обеспечивается тем, что близость блоков чувствительна к размеру блока. Иерархический поиск начинает с блоков больших размеров и использует их векторы перемещения как исходную точку поисков для блоков меньших размеров. Большие блоки с меньшей вероятностью могут привести к ошибочному локальному минимуму, одновременно с этим, малые блоки обычно производят лучшие векторы перемещения. Метод иерархического поиска имеет высокую вычислительную сложность, и ускорить его можно, сократив число выполняемых операций. Это делается несколькими способами. Вот некоторые из них:

1. На первом шаге, когда размер блока еще велик, выбрать приблизительно подходящие блоки. Соответствующие им векторы перемещения не будут наилучшими, но они будут использоваться лишь как отправные точки для дальнейших лучших векторов.
2. При исследовании больших блоков пропустить некоторые пиксели. Например, алгоритм может использовать только четверть пикселов больших блоков, половину пикселов меньших блоков и так далее.
3. Выбрать размеры блоков так, что блоки, используемые на шаге  $i$ , делятся на несколько (обычно четыре или девять) меньших блоков, используемых на следующем шаге. На этом пути каждый вектор перемещения, вычисленный на шаге  $i$ , будет служить приближенной оценкой для нескольких векторов перемещений меньших блоков шага  $i + 1$ .

**Методы многомерного пространственного поиска:** Эти методы более сложны. При поиске блока, близкого блоку  $B$ , они используют не только сдвиги данного блока  $B$ , но также его вращения, растяжения и сжатия.

Метод многомерного пространственного поиска может также найти блок  $C$ , который близок блоку  $B$ , но при других условиях освещения. Это бывает полезно, когда объекты в кадре пересекают участки с разной освещенностью. Все рассмотренные выше алгоритмы оценивали близость двух блоков друг другу с помощью сравнения величин светимости соответствующих пикселов. Два блока  $B$  и  $C$ , которые содержат одни и те же объекты, но с разной освещенностью, будут объявлены различными.

Если же метод многомерного пространственного поиска обнаружит блок  $C$ , который походит на блок  $B$ , но с другой светимостью, то он объявит его близким к  $B$  и добавит к сжатому кадру величину светимости. Эта величина (возможно, отрицательная) будет прибавлена декодером к пикселям декодированного кадра, чтобы придать им изначальную светимость.

Метод многомерного пространственного поиска может также сравнивать блок  $B$  с повернутыми копиями блоков-кандидатов  $C$ . Это полезно, если объекты видеоряда могут вращаться наряду с совершением поступательных перемещений. Более того, такой алгоритм может одновременно масштабировать блоки  $C$ , стараясь подобрать лучшее совпадение блоков. Например, если блок  $B$  состоит из  $8 \times 8$  пикселов, то алгоритм может попытаться сравнивать этот блок с блоками  $C$ , состоящими из  $12 \times 12$  пикселов, путем их сжатия до размеров  $8 \times 8$ .



Конечно, такие алгоритмы используют еще большие вычислительные мощности для совершения дополнительных операций и сравнений. Можно говорить, что это существенно увеличивает размерность *пространства поиска*, и этим оправдывается использование наименования *многомерное пространство поиска*. Однако, насколько известно автору, на практике пока не разработан метод многомерного пространственного поиска, который использует в полной мере масштабирование, вращение и изменение светимости.

*Video meliora proboque deterioriora sequor*

(Я вижу лучшее и одобряю его, но следую за худшим).

— Овидий «Метаморфозы», 7:20

# ГЛАВА 6

## СЖАТИЕ ЗВУКА

Файл с текстом занимает обычно мало места на диске компьютера. Типичная книга, содержащая около миллиона символов, в несжатом виде будет занимать объем порядка 1 МБ, т.к. каждому символу будет отведен один байт. Например, книга в 400 страниц, в среднем, по 45 строк из 60 букв на каждой странице будет содержать примерно  $60 \times 45 \times 400 = 1080000$  символов или байт.

В отличие от этого, хранение изображений требует гораздо больших объемов, которое придает иное звучание фразе «картина стоит тысяч слов ее описания». В зависимости от числа используемых цветов изображения, один пиксель требует от одного бита до трех байтов. Таким образом, картинка размером  $512 \times 512$  пикселов займет от 32 КБ до 768 КБ.

С появлением мощных и недорогих персональных компьютеров стали разрабатываться всевозможные мультимедийные приложения и программы, в которых используются тексты, изображения, анимированные фрагменты и звук. Всю эту разнородную цифровую информацию необходимо хранить в компьютере, отображать, редактировать и проигрывать. Для хранения звука места требуется меньше, чем для изображений и видео, но больше, чем для текста. Вот почему проблема сжатия аудио информации стала весьма актуальной в 1990 годах и привлекла пристальное внимание исследователей.

Эта глава начинается коротким введением о природе звука и методах его оцифровывания. Потом обсуждается строение органов слуха человека и особенности восприятия звука ухом и мозгом, которые позволяют выбрасывать при сжатии большую часть цифровой аудио информации без потери качества воспринимаемого звука. Затем обсуждаются два простых метода сжатия оцифрованного звука, а именно *подавление пауз* и *уплотнение*. В конце главы приводится описание популярного метода сжатия звука MP3, который является составной частью стандарта MPEG-1.



## 6.1. Звук

Для большинства из нас звук является привычным явлением, мы постоянно его слышим. Однако, если попытаться дать точное определение звуку, то быстро выясниться, что сделать это можно с двух различных точек зрения.

Интуитивное определение: звук, это ощущения, воспринимаемые нашим ухом и интерпретируемые мозгом определенным образом.

Научное определение: звук это колебание среды. Он распространяется в среде с помощью волн давления посредством колебания атомов и молекул.

Обычно, мы слышим звук, который распространяется в воздухе и колеблет наши барабанные перепонки. Однако звук может распространяться и во многих других средах. Морские животные способны издавать звуки в воде и откликаться на них. Если ударить молотком по концу металлического рельса, то в нем возникнут звуковые колебания, которые можно обнаружить на другом конце. Хорошие звуковыми изоляторами разрежены, а наилучшим изолятором служит вакуум, в котором отсутствуют частицы, способные колебаться и передавать возмущения.

Одновременно звук можно считать волной, даже если ее частота может все время меняться. Эта волна является продольной; в ней направление возмущения совпадает с направлением распространения волны. Наоборот, электромагнитные волны и волны в океане являются поперечными. Их колебания направлены перпендикулярно движению волны.

Как и любая волна звук имеет три важных атрибута, а именно, скорость, амплитуду и период. Частота волны не является независимым атрибутом, она равна числу периодов волны за единицу времени (одну секунду). Единицей частоты служит герц (Гц). Скорость звука зависит от свойств среды, в которой он распространяется, а также от температуры. В воздухе на уровне моря (при давлении в одну атмосферу) и при температуре 20° по Цельсию скорость звука равна 343.8 метров в секунду.

Человеческое ухо способно воспринимать звук в широком диапазоне частот, обычно, от 20 Гц до 22000 Гц, что зависит от возраста и состояния здоровья человека. Это, так называемый, *диапазон слышимых частот*. Некоторые животные, например, собаки и летучие мыши, могут слышать звук более высокой частоты (ультразвук). Простое вычисление дает периоды слышимых звуков. При частоте 22000 Гц период равен около 1.56 см., а при 20 Гц он равен 17.19 м.

Амплитуда звука также важна. Мы воспринимаем ее как громкость. Мы слышим звук, когда молекулы начинают ударять по барабанным перепонкам в ушах и оказывают на них определенное давление. Молекулы перемещаются вперед–назад на крошечное расстояние, которое соотносится с амплитудой, но не с периодом звука. Период звука может быть равен нескольким метрам, а молекулы при этом смещаются на миллионные доли сантиметра в своих колебаниях. Таким образом, устройство регистрации звуков должно иметь весьма чувствительную диафрагму, чтобы улавливать давление звуковой волны и переводить их в электромагнитные колебания, которые затем будут преобразовываться в цифровую форму.

Сложности с измерением интенсивности звука связаны с тем, что наше ухо чувствительно к весьма широкому диапазону уровней громкости (амплитуде) звука. Уровень грохота пушки и уровень комариного писка может различаться на 11–12 порядков. Если мы обозначим уровень наименьшего слышимого звука (порог слышимости) за 1, то уровень грохота пушки будет равен  $10^{11}$ ! Весьма затруднительно работать с таким широким размахом измеряемой величины, поэтому для измерения громкости звука используется *логарифмическая шкала*. Логарифм 1 равен 0, а десятичный логарифм  $10^{11}$  равен 11. Используя логарифмы, можно иметь дело с числами в интервале от 0 до 11. На самом деле, такой интервал маловат, поэтому его принято умножать на 10 или на 20, чтобы работать с числами от 0 до 110 или от 0 до 220. В этом заключается хорошо известный (но иногда вызывающий затруднения с пониманием) метод измерения с помощью *декибел*.

Единица измерения в 1 децибел (дБ) определяется как десятичный логарифм частного между двумя физическими величинами, для которых единицей измерения служит мощность (энергия в единицу времени). Этот логарифм следует умножить на 10 (Если не делать этого, то получится единица, называемая «бел», которая, впрочем, была давно отброшена в пользу единицы «декибел»). Итак, получаем

$$\text{уровень} = 10 \log_{10} \frac{P_1}{P_2} \text{ дБ},$$

где  $P_1$  и  $P_2$  – величины, измеренные в единицах мощности, то есть, ватт, джоуль/сек, грамм·см/сек или лошадиная сила. Это может быть мощность молекулы, электрическая мощность или еще что-то. При измерении громкости звука применяется единица акустической мощности. Поскольку громкий звук можно произвести с помощью малой энергии, то обычно используется единица микроватт ( $10^{-6}$ ).

Децибел – это логарифм частного двух величин. В числителе стоит мощность  $P_1$  звука, чей уровень громкости мы желаем измерить. В качестве знаменателя принято использовать мощность самого слабого различимого звука (порога слышимости). Из экспериментов было получено, что мощность порога слышимости составляет  $10^{-6}$  микроватт, то есть,  $10^{-12}$  ватт. Таким образом, стереоустройство, производящее 1 ватт акустической мощности, имеет уровень громкости

$$10 \log_{10} \frac{10^6}{10^{-6}} = 10 \log_{10}(10^{12}) = 10 \times 12 = 120 \text{ дБ}$$

(это где-то в районе порога болевого ощущения; см. рис. 6.1), а наушники, вырабатывающие  $3 \times 10^{-4}$  микроватт имеют уровень

$$10 \log_{10} \frac{3 \times 10^{-4}}{10^{-6}} = 10 \log_{10}(3 \times 10^2) = 10 \times (\log_{10} 3 + 2) \approx 24.77 \text{ дБ.}$$



Рис. 6.1. Шкала уровней звука в единицах дБ PSL.

В теории электричества существует простое соотношение между (электрической) мощностью  $P$  и давлением (напряжением)  $V$ . Электрическая мощность равна произведению электрического тока на напряжение  $P = I \cdot V$ . Ток, по закону Ома, пропорционален напряжению, то есть,  $I = V/R$ , где  $R$  – сопротивление. Следовательно, можно записать, что  $P = V^2/R$  и использовать давление (напряжение) при измерениях в децибелах.

На практике не всегда имеется доступ к источнику звука для измерения электрической мощности на выходе. Держа в руках измеритель децибелов звука, можно оказаться в сложном положении при измерении уровня шума вокруг себя. Измеритель децибелов определяет давление  $Pr$ , которое оказывают звуковые волны на его диафрагму. К счастью, акустическая мощность на единицу площади (обозначаемая  $P$ ) пропорциональна квадрату звукового давления  $Pr$ . Это имеет место в силу того, что мощность  $P$  равна произведению давления  $Pr$  и скорости звука  $v$ , а звук, в свою очередь, можно выразить как давление, деленное на особый импеданс (полное сопротивление) среды, через которую проходит данный звук. Поэтому громкость звука еще принято измерять в единицах дБ SPL (sound pressure level, уровень звукового давления) вместо мощности звука. По определению,

$$\text{уровень} = 10 \log_{10} \frac{P_1}{P_2} = 10 \log_{10} \frac{Pr_1^2}{Pr_2^2} = 20 \log_{10} \frac{Pr_1}{Pr_2} \text{ дБ SPL.}$$

Нулевой уровень, измеренный в единицах, дБ PSL соответствует величине 0.0002 дин/см<sup>2</sup>, где дина – это малая единица силы, равная примерно весу 0.0010197 грамм. Поскольку дина равна  $10^{-5}$  Н (ньютона), а сантиметр – это 0.01 метра, то нулевой уровень (порог слышимости) равен 0.00002 Н/м<sup>2</sup>. В табл. 6.2 приведены типичные значения дБ для обоих единиц мощности и SPL.

ватты	дБ	давление Н/м <sup>2</sup>	дБ SPL	источник
30000.0	165	2000.0	160	реактивный самолет
300.0	145	200.0	140	болевой порог
3.0	125	20.0	120	заводской шум
0.03	105	2.0	100	уличный транспорт
0.0003	85	0.2	80	бытовой прибор
0.000003	65	0.02	60	беседа
0.00000003	45	0.002	40	тихая комната
0.0000000003	25	0.0002	20	шепот
0.00000000001	0	0.00002	0	порог слышимости

Табл. 6.2. Уровни различных звуков в единицах мощности и давления.

Чувствительность уха человека к уровню звука зависит от его частоты. Из опытов известно, что люди более чувствительны к звукам высокой частоты (поэтому сирена воет высокими тонами). Можно слегка модифицировать систему дБ SPL, чтобы она сильнее зависела от высоких частот и слабее от низких. Такая система

называется стандартом dBA. Существуют также стандарты дБВ и дБС для измерения уровня шума. (В электротехнике применяются также стандарты dBm, dBm0 и dBm; см., например, [Shenio 95]).

Из-за применения функции логарифм величины, измеренные в децибелах, нельзя складывать. Если трубач заиграет после концерта на своей трубе, извлекая звуки, скажем в 70 дБ, а затем к нему присоединится второй музыкант, играя на тромbone с таким же уровнем звука, то (бедный) слушатель получит удвоение интенсивности звука, но этому будет соответствовать уровень лишь в 73 дБ, а не в 140 дБ. В самом деле, если

$$10 \log_{10} \left( \frac{P_1}{P_2} \right) = 70,$$

то

$$10 \log_{10} \left( \frac{2P_1}{P_2} \right) = 10 \left( \log_{10} 2 + \log_{10} \left( \frac{P_1}{P_2} \right) \right) = 10(0.3 + 7) = 73.$$

Удвоение интенсивности шума приводит к увеличению уровня на 3 единицы (при использовании единиц SPL это число следует удвоить).

Более подробные сведения о звуке, его свойствах и измерении можно почерпнуть из [Shenow 95].

## 6.2. Оцифрованный звук

Как уже отмечалось, любое изображение можно оцифровать разбив его на пиксели, а каждому пиксели приписать некоторое число. Точно также звук можно оцифровать, разбив его на фрагменты и присвоив им некоторые числовые значения. Если записывать звук через микрофон, то он переводится в электрический сигнал, напряжение которого непрерывно зависит от времени. На рис. 6.3 показан типичный пример записи звука, которая начинается в нуле и колеблется несколько раз. Это напряжение называется *аналоговым* представлением звука. Оцифровка звука делается с помощью измерения напряжения сигнала во многих точках оси времени, переводя каждого измерения в числовую форму и записи полученных чисел в файл. Этот процесс называется *сэмплированием* или отбором фрагментов. Звуковая волна сэмплируется, а сэмплы (звуковые фрагменты) становятся оцифрованным звуком. Устройство сэмплирования звука называется *аналого-цифровым преобразователем* (АЦП или, по-английски, ADC, analog-to-digital converter).



Разницу между звуковой волной и ее сэмплами можно сравнить с разницей между обычными часами с циферблатом, в которых стрелки вращаются непрерывно, и электронными часами, в которых показания дисплея сменяются скачками каждую секунду.

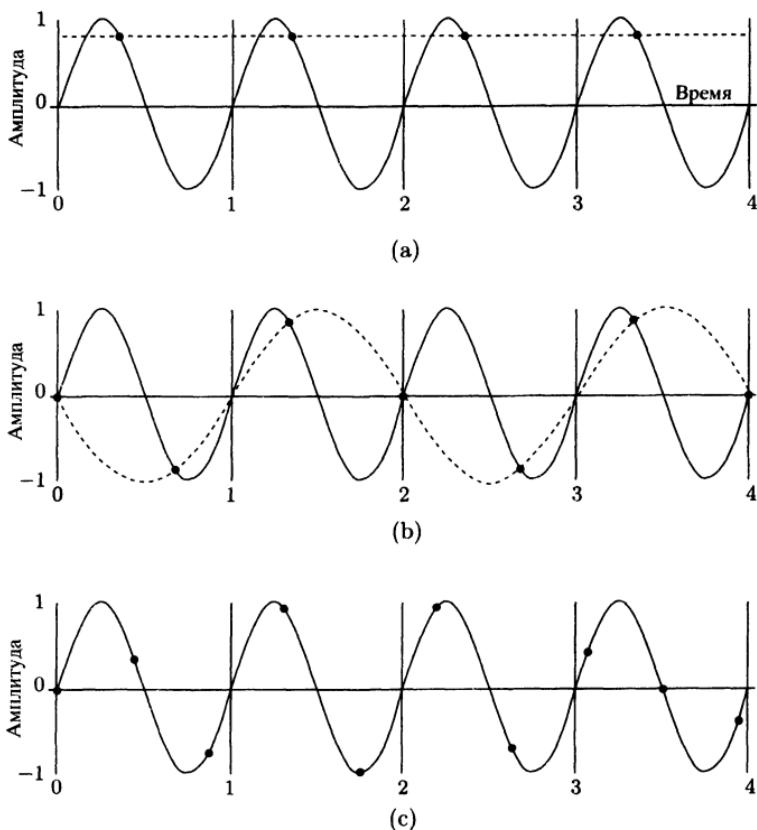


Рис. 6.3. Сэмплирование звуковой волны.

Поскольку звуковые сэмплы являются числами, их легко редактировать. Однако, основное назначение звуковых файлов состоит в их проигрывании и в прослушивании. Это делается с помощью перевода числовых сэмплов в электрическое напряжение, которое непрерывно подается на динамики. Устройство для выполнения этой процедуры называется цифро-аналоговым преобразователем (ЦАП или DAC, digital-to-analog converter). Очевидно, более высокая скорость сэмплирования дает лучшее представление звука, но это же



приводит к увеличению числа сэмплов (звуковых фрагментов) и к росту объема звукового файла. Следовательно, основная проблема сэмплирования состоит в определении оптимальной скорости отбора сэмплов.

Из рис. 6.3а видно, что может случиться, если скорость сэмплирования низка. Звуковая волна сэмплировалась четыре раза и все сэмплы оказались равными друг другу. Если проиграть эти сэмплы, то получится равномерный звук, похожий на жужжение. На рис. 6.3б показано семь сэмплов; они больше похожи на исходную волну. К сожалению, если использовать их для воспроизведения звука, то получится пунктирная волна. Их также недостаточно для точного отображения исходного звука.

Решение задачи сэмплирования состоит в отборе звуковых фрагментов со скоростью чуть выше скорости Найквиста, которая равна удвоенному максимуму частоты волн данного звука. Такое сэмплирование гарантирует весьма близкое восстановление звуковой волны. Это проиллюстрировано на рис. 6.3с, на котором приведены 10 отсчетов, взятых через равные интервалы времени на четырех периодах волны. Отметим, что сэмплы не обязательно отбираются от минимума до максимума волны; их можно брать в любых точках.

Диапазон слышимых частот лежит в интервале от 16–20 Гц до 20000–22000 Гц. Он зависит от возраста и других физических особенностей человека. Если необходимо оцифровывать звук с высокой точностью, то скорость сэмплирования должна быть выше скорости Найквиста, которая равна  $2 \times 22000 = 44000$  Гц. Поэтому высококачественный оцифрованный звук основан на скорости сэмплирования 44100 Гц. Скорость ниже этого значения приводит к искажениям, а большая скорость сэмплирования не даст улучшение реконструкции звука. Поэтому на практике можно использовать фильтры пропускания до 44100 Гц для эффективного удаления частот выше 22000 Гц.

Многие низкокачественные приложения сэмплируют звук на скорости 11000 Гц, а система телефонии, изначально разработанная для переговоров, но не для цифровых коммуникаций, сэмплирует звук с частотой всего в 8 кГц. Значит, любой звук с частотой выше 4 кГц будет искажаться при передаче по телефону. По этой же причине бывает трудно различить по телефону звуки «с» и «ф». Поэтому часто при сообщении по телефону имен, фамилий и адресов приходится диктовать первые буквы слов вроде «Михаил», «Ольга», «Семен», «Константин», «Виктория», «Анна».



Другая проблема сэмплирования заключается в размере звукового фрагмента, то есть, сэмпла. Каждый сэмпл – это число, но насколько большим может оно быть? Обычно на практике сэмплы состоят из 8 или 16 бит, но высококачественные звуковые карты допускают использование 32 бит. Предположим, что наибольшее напряжение звуковой волны равно 1 вольт, тогда при длине сэмпла в 8 бит можно будет различить напряжение с шагом в  $1/256 \approx 0.004$  вольт, то есть, 4 милливольт (мВ). Тихие звуки, генерирующие волны до 2 мВ, будут сэмплироваться в ноль и при воспроизведении не будут слышны. В отличие от этого, при 16-битных сэмплах возможно различить звуки, начиная с  $1/65536 \approx 15$  микровольт ( $\mu$ В). Можно считать размер сэмпла шагом квантования исходных аудиоданных. 8-битовое сэмплирование является более грубым, чем 16-битовое. В результате будет произведено лучшее сжатие, но более бедное воспроизведение (реконструированный звук будет иметь лишь 256 уровней).

Сэмплирование звука принято еще называть импульсной кодовой модуляцией (PCM, pulse code modulation). Все слышали про АМ и FM радио. Эти сокращения означают *amplitude modulation* (амплитудная модуляция) и *frequency modulation* (частотная модуляция). Они указывают на применяемый метод модуляции (т.е. вкладывания двоичной информации) в непрерывные волны. Термин импульсная модуляция обозначает технику перевода непрерывных волн в двоичный файл. Существует несколько методов импульсной модуляции, включающих импульсную амплитудную модуляцию (PAM, pulse amplitude modulation), импульсную позиционную модуляцию (PPM, pulse position modulation) и импульсную числовую модуляцию (PNM, pulse number modulation). Хорошим источником этой информации может служить [Pohlmann 85]. На практике, однако, наибольшую эффективность обнаруживает метод PCM перевода звуковых волн в числовую информацию. При оцифровывании стереозвука с помощью сэмплирования на 22000 кГц с 16-битными сэмплами в одну секунду генерируется 44000 сэмплов длины 16 бит, то есть, 704000 бит/сек или 88000 байт/сек.

### 6.3. Органы слуха человека

Как уже говорилось, человеческое ухо способно воспринять звуки с частотой от 20 до 22000 Гц, но его чувствительность не является одинаковой в этом интервале. Она зависит от частоты звука. Экс-

перименты указывают на то, что в тихой окружающей обстановке чувствительность уха максимальна при частотах от 2 до 4 кГц. На рис. 6.4а показан порог слышимости для тихого окружения.

Стоит отметить, что частотный диапазон человеческого голоса также весьма ограничен. Он располагается в интервале от 500 Гц до 2 кГц.

Существование порога слышимости дает основу для построения методов сжатия звука с потерями. Можно удалять все сэмплы, величина которых лежит ниже этого порога. Поскольку порог слышимости зависит от частоты, кодер должен знать спектр сжимаемого звука в каждый момент времени. Для этого нужно хранить несколько предыдущих входных сэмплов (обозначим это число  $n - 1$ ; оно или фиксировано, или задается пользователем). При вводе следующего сэмпла необходимо на первом шаге сделать преобразование  $n$  сэмплов в частотную область. Результатом служит вектор, состоящий из  $t$  числовых компонент, которые называются *сигналами*. Он определяет частотное разложение сигнала. Если сигнал для частоты  $f$  меньше порога слышимости этой частоты, то его следует отбросить.

Кроме того, для эффективного сжатия звука применяются еще два свойства органов слуха человека. Эти свойства называются *частотное маскирование и временное маскирование*.

Частотное маскирование (его еще называют *слуховое маскирование*) происходит тогда, когда нормально слышимый звук накрывается другим громким звуком с близкой частотой. Толстая стрелка на рис. 6.4б обозначает громкий источник звука с частотой 800 Гц. Этот звук приподнимает порог слышимости в своей окрестности (пунктирная линия). В результате звук, обозначенный тоненькой стрелкой в точке «х» и имеющий нормальную громкость выше своего порога чувствительности, становится неслышимым; он маскируется более громким звуком. Хороший метод сжатия звука должен использовать это свойство слуха и удалять сигналы, соответствующие звуку «х», поскольку они все равно не будут услышаны человеком. Это один возможный путь сжатия с потерями.

Частотное маскирование (область под пунктирной линией на рис. 6.4б) зависит от частоты сигнала. Оно варьируется от 100 Гц для низких слышимых частот до более чем 4 кГц высоких частот. Следовательно область слышимых частот можно разделить на несколько *критических полос*, которые обозначают падение чувствительности уха (не путать со снижением мощности разрешения) для

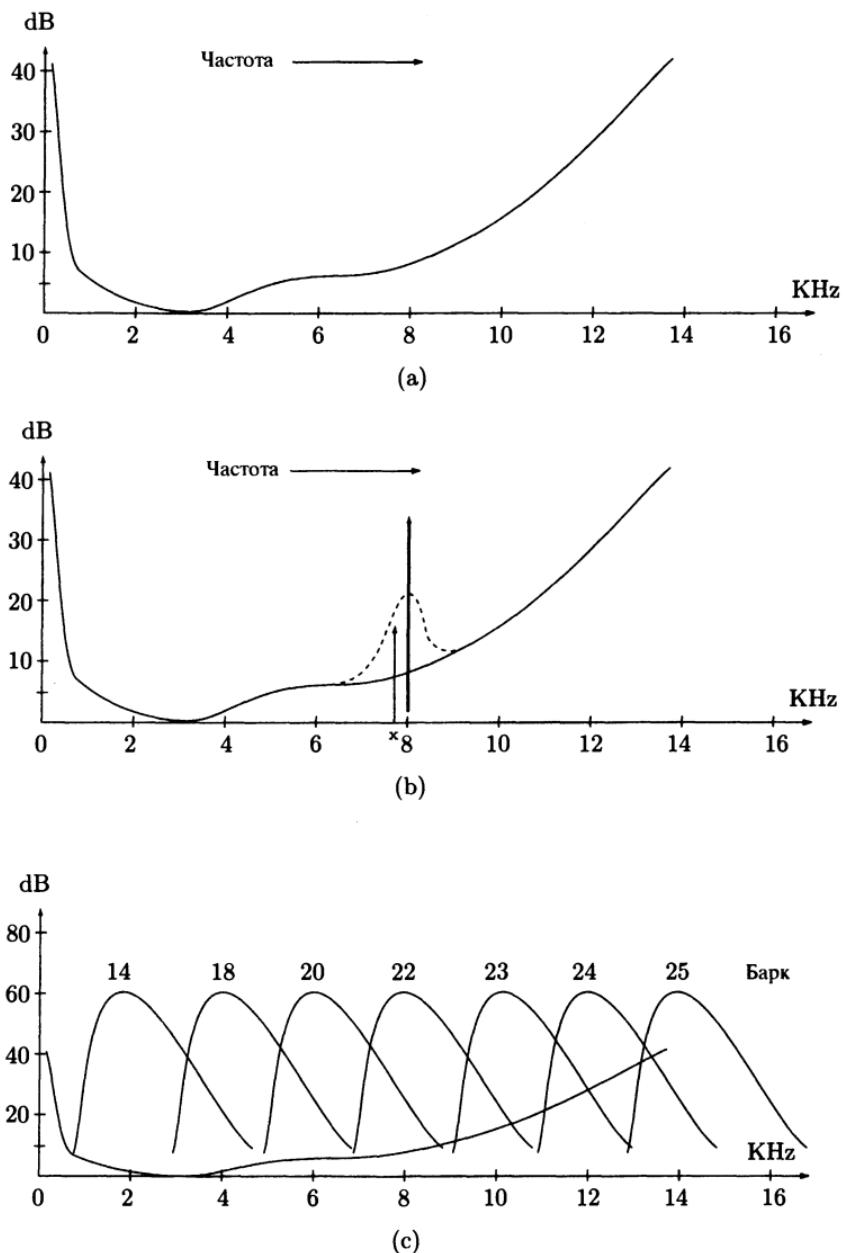


Рис. 6.4. Порог и маскирование звука.

более высоких частот. Можно считать критические полосы еще одной характеристикой звука, подобной его частоте. Однако, в отличие от частоты, которая абсолютна и не зависит от органов слуха, критические полосы определяются в соответствии со слуховым восприятием. В итоге они образуют некоторые меры восприятия частот. В табл. 6.5 перечислены 27 приближенных критических полос.

полоса	область	полоса	область	полоса	область
0	0–50	9	800–940	18	3280–3840
1	50–95	10	940–1125	19	3840–4690
2	95–140	11	1125–1265	20	4690–5440
3	140–235	12	1265–1500	21	5440–6375
4	235–330	13	1500–1735	22	6375–7690
5	330–420	14	1735–1970	23	7690–9375
6	420–560	15	1970–2340	24	9375–11625
7	560–660	16	2340–2720	25	11625–15375
8	660–800	17	2720–3280	26	15375–20250

Табл. 6.5. 27 приближенных критических полос.

Критические полосы можно описать следующим образом: из-за ограниченности слухового восприятия звуковых частот порог слышимости частоты  $f$  приподнимается соседним звуком, если звук находится в критической полосе  $f$ . Это свойство открывает путь для разработки практического алгоритма сжатия аудиоданных с потерями. Звук необходимо преобразовать в частотную область, а получившиеся величины (частотный спектр) следует разделить на подполосы, которые максимально приближают критические полосы. Если это сделано, то сигналы каждой из подполос нужно квантовать так, что шум квантования (разность между исходным звуковым сэмплом и его квантованными значениями) был неслышимым.

Еще один возможный взгляд на концепцию критической полосы состоит в том, что органы слуха человека можно представить себе как своего рода фильтр, который пропускает только частоты из некоторой области (полосы пропускания) от 20 до 20000 Гц. В качестве модели ухо–мозг мы рассматриваем некоторое семейство фильтров, каждый из которых имеет свою полосу пропускания. Эти полосы называются критическими. Они пересекаются и имеют разную ширину. Они достаточно узки (около 100 Гц) в низкочастотной области и расширяются (до 4–5 кГц) в области высоких частот.

Ширина критической полосы называется ее размером. Для измерения этой величины вводится новая единица «барк» («Bark» от

H.G.Barkhausen). Один барк равен ширине (в герцах) одной критической полосы. Эта единица определяется по формуле

$$1 \text{ барк} = \begin{cases} \frac{f}{100} & \text{для частот } f < 500 \text{ Гц}, \\ 9 + 4 \log \left( \frac{f}{1000} \right) & \text{для частот } f \geq 500 \text{ Гц.} \end{cases}$$

На рис. 6.4с показаны несколько критических полос с величиной барк от 14 до 25 единиц, которые помещены над кривой порогов слышимости.

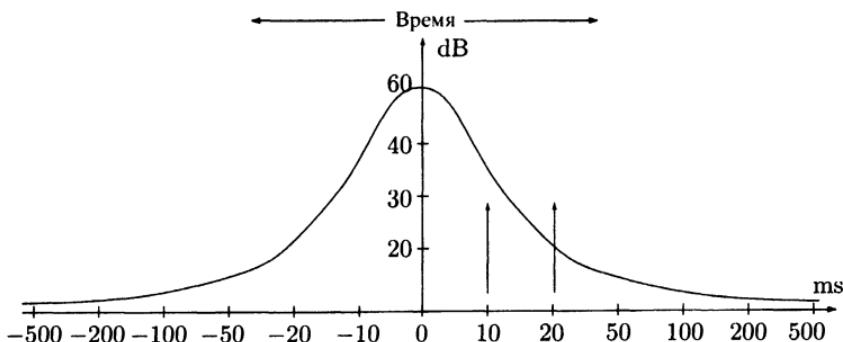


Рис. 6.6. Порог и маскирование звука.

Временное маскирование происходит, когда громкому звуку *A* частоты *f* по времени предшествует или за ним следует более слабый звук *B* близкой частоты. Если интервал времени между этими звуками не велик, то звук *B* будет не слышен. Рис. 6.6 иллюстрирует пример временного маскирования. Порог временного маскирования от громкого звука в момент времени 0 идет вверх сначала круто, а потом полого. Более слабый звук в 30 дБ не будет слышен, если он раздастся за 10 мсек до или после громкого звука, но будет различим, если временной интервал между ними будет больше 20 мсек.

#### 6.4. Общепризнанные методы

Общепризнанные методы сжатия данных, такие, как RLE, статистические и словарные методы, могут быть использованы для компрессии звуковых файлов без потерь, но результат существенно зависит от конкретных аудиоданных. Некоторые звуки будут хорошо сжиматься с помощью RLE, но плохо – статистическими алгоритмами. Другим звукам больше подходит статистическое сжатие, а при



словарном подходе, наоборот, может произойти расширение. Приведем краткую характеристику эффективности этих трех методов при сжатии звуковых файлов.

RLE хорошо работает со звуками, которые содержат длинные серии повторяющихся звуковых фрагментов – сэмплов. При 8-битном сэмплировании это может происходить довольно часто. Напомним, что разность электрического напряжения между двумя 8-битовыми сэмплами  $n$  и  $n + 1$  составляет около 4 мВ. Несколько секунд однородной музыки, в которой звуковая волна будет меняться менее чем на 4 мВ, породят последовательность из тысяч тождественных сэмплов. При 16-битном сэмплировании, очевидно, длинные повторы встречаются реже, и, следовательно, алгоритм RLE будет менее эффективен.

Статистические методы присваивают коды переменной длины звуковым сэмплам в соответствии с их частотностью. При 8-битном сэмплировании имеется всего 256 различных сэмплов, поэтому в большом звуковом файле сэмплу могут быть распределены равномерно. Такой файл не удастся хорошо сжать методом Хаффмана. При 16-битном сэмплировании допускается более 65000 звуковых фрагментов. В этом случае возможно, что некоторые сэмплы будут встречаться чаще, а другие – реже. При сильной асимметрии вероятностей хороших результатов можно добиться с помощью арифметического кодирования.

Методы, основанные на словарном подходе, предполагают, что некоторые фразы будут встречаться часто на протяжении всего файла. Это происходит в текстовом файле, в котором отдельные слова или их последовательности повторяются многократно. Звук, однако, является аналоговым сигналом и значения конкретных сгенерированных сэмплов в большой степени зависит от работы АЦП. Например, при 8-битном сэмплировании, волна в 8 мВ становится числовым сэмплом, равным 2, но близкая ей волна, скажем, в 7.6 мВ или 8.5 мВ может стать другим числом. По этой причине, речевые фрагменты, содержащие совпадающие фразы и звучащие для нас одинаково, могут слегка отличаться при их оцифровывании. Тогда они попадут в словарь в виде разных фраз, что не даст ожидаемого сжатия. Таким образом, словарные методы не очень подходят для сжатия звука.

Можно добиться лучших результатов при сжатии звука с потерей части аудиоинформации, развивая методы компрессии, которые учитывают особенности восприятия звука. Они удаляют ту часть данных, которая остается неслышимой для органов слуха. Это

похоже на сжатие изображений с отбрасыванием информации, незаметной для глаза. В обоих случаях мы исходим из того факта, что исходная информация (изображение или звук) является аналоговым, то есть, часть информации уже потеряна при квантовании и оцифровывании. Если допустить еще некоторую потерю, сделав это аккуратно, то это не повлияет на качество воспроизведения разожженного звука, который не будет сильно отличаться от оригинала. Мы кратко опишем два подхода, которые называются *подавлением пауз* и *уплотнением*.

Идея подавления пауз заключается в рассмотрении малых сэмплов, как если бы их не было (то есть, они равны нулю). Такое обнуление будет порождать серии нулей, поэтому метод подавления пауз, на самом деле, является вариантом RLE, приспособленным к сжатию звука. Этот метод основан на особенности звукового восприятия, которое состоит в терпимости уха человека к отбрасыванию еле слышных звуков. Аудиофайлы, содержащие длинные участки тихого звука будут лучше сжиматься методом подавления пауз, чем файлы, наполненные громкими звуками. Этот метод требует участие пользователя, который будет контролировать параметры, задающие порог громкости для сэмплов. При этом необходимы еще два параметра, они не обязательно контролируются пользователем. Один параметр служит для определение самых коротких последовательностей тихих сэмплов, обычно, это 2 или 3. А второй задает наименьшее число последовательных громких сэмплов, при появлении которых прекращается тишина или пауза. Например после 15 тихих сэмплов может последовать 2 громких, а затем 13 тихих, что будет определено как одна большая пауза длины 30, а аналогичная последовательность из 15, 3 и 12 сэмплов, станет двумя паузами с коротким звуком между ними.

Уплотнение основано на том свойстве, что ухо лучше различает изменения амплитуды тихих звуков, чем громких. Типичное АЦП звуковых карт компьютеров использует линейное преобразование при переводе напряжения в числовую форму. Если амплитуда  $a$  была конвертирована в число  $n$ , то амплитуда  $2a$  будет переведена в число  $2n$ . Метод сжатия на основе уплотнения сначала анализирует каждый сэмпл звукового файла и применяет к нему нелинейную функцию для сокращения числа бит, назначенных этому сэмплу. Например, при 16-битных сэмплах, кодер с уплотнением может применять следующую простую формулу

$$\text{образ} = 32767 \left( 2^{\frac{\text{сэмпл}}{65536}} - 1 \right) \quad (6.1)$$

для сокращения каждого сэмпла. Эта формула нелинейно отображает 16-битные сэмплы в 15-битные числа интервала [0,32767], причем маленькие (тихие) сэмплы меньше подвергаются искажению, чем большие (громкие). Табл. 6.7 иллюстрирует нелинейность этой функции. На ней показано 8 пар сэмплов, причем в каждой паре разность между сэмплами равна 100. Для первой пары разность между их образами равна 34, а разность между образами последней (громкой) пары равна 65. Преобразованные 15-битные числа могут быть приведены к исходным 16-битным сэмплам с помощью обратной формулы

$$\text{сэмпл} = 65536 \log_2 \left( 1 + \frac{\text{образ}}{32767} \right). \quad (6.2)$$

Сэмпл	Образ	Разность	Сэмпл	Образ	Разность
100 →	35		30000 →	12236	
200 →	69	34	30100 →	12283	47
1000 →	348		40000 →	17256	
1100 →	383	35	40100 →	17309	53
10000 →	3656		50000 →	22837	
10100 →	3694	38	50100 →	22896	59
20000 →	7719		60000 →	29040	
20100 →	7762	43	60100 →	29105	65

Табл. 6.7. Отображение 16-битных сэмплов в 15-битные числа.

Сокращение 16-битных сэмплов до 15-битных чисел не дает существенного сжатия. Лучшее сжатие получается, если в формулах (6.1) и (6.2) заменить число 32767 меньшим. Например, если взять число 127, то 16-битные сэмплы будут представлены 8-битными числами, то есть, коэффициент сжатия будет равен 0.5. Однако, декодирование будет менее аккуратным. Сэмпл 60100 будет отображен в число 113, а при декодировании по формуле (6.2) получится сэмпл 60172. А маленький 16-битный сэмпл 1000 будет отображен в 1.35, что после округления даст 1. При декодировании числа 1 получится 742, что сильно отличается от исходного сэмпла. Здесь коэффициент сжатия может быть параметром, непосредственно задаваемым пользователем. Это интересный пример метода сжатия, при котором коэффициент сжатия известен заранее.

На практике нет необходимости обращаться к уравнениям (6.1) и (6.2), поскольку результат отображения можно заранее приготовить в виде таблицы. Тогда и кодирование, и декодирование будут делаться быстро.

Уплотнение не ограничивается уравнениями (6.1) и (6.2). Более изощренные методы, такие как  $\mu$ -правило и А-правило, широко применяются на практике и входят во многие международные стандарты сжатия.

## 6.5. Сжатие звука в стандарте MPEG-1

Стандарт MPEG-1 сжатия видеофильмов состоит из двух основных частей: сжатия видео и сжатия звука. В этом параграфе обсуждаются принципы компрессии звука в MPEG-1, а именно, его третий слой, который широко известен по аббревиатуре MP3. Мы советуем читателям обязательно прочитать первую часть этой главы перед тем, как пытаться освоить этот материал.

Формальное имя стандарта MPEG-1 – *international standard for moving picture video compression IS 11172* (международный стандарт для сжатия движущихся изображений). Он состоит из 5 частей, среди которых часть 3 [ISO/IEC 93] определяет алгоритм сжатия звука. Как любой стандарт, выработанный ITU или ISO, документ, описывающий MPEG-1, имеет *нормативный* и *описательный* разделы. Нормативный раздел содержит спецификации стандарта. Он написан строгим языком для тех, кто будет создавать программные реализации метода для конкретных машинных платформ. Описательный раздел иллюстрирует выбранные концепции, объясняет причины выбора того или иного подхода, содержит необходимые базовые сведения.

Примером нормативного раздела являются таблицы с различными параметрами и с кодами Хаффмана, которые используются в стандарте MPEG. А примером описательного раздела служит алгоритм, задающий психоакустическую модель. MPEG не дает конкретного алгоритма, и кодер MPEG свободен в выборе метода реализации модели. В этом параграфе просто рассматриваются некоторые возможные альтернативы.

Аудиостандарт MPEG-1 описывает три метода сжатия, называемые слоями (*layer*), которые обозначаются римскими числами I, II и III. Все три слоя входят в стандарт MPEG-1, но здесь будет описан только слой III. При сжатии видеофильмов используется только один слой, который обозначается в заголовке сжатого файла. Любой из этих слоев можно независимо использовать для сжатия звука без видео. Функциональные модули младших слоев могут быть использованы старшими слоями, но более высокие слои использу-



ют дополнительные возможности для лучшего сжатия. Интересной особенностью слоев является их иерархическая структура, то есть, декодер слоя III может декодировать файлы сжатые слоями I и II.

Результатом разработки трех слоев было возрастание популярности слоя III. Кодер этого метода очень сложен, но он производит замечательную компрессию, это обстоятельство в сочетании с тем, что декодер существенно проще кодера, породило небывалый взрыв популярности звуковых файлов, которые называются MP3-файлами. Очень легко добыть декодер слоя III, с помощью которого можно прослушивать записи формата MP3, которые в огромном количестве находятся во всемирной паутине. Это был настоящий триумф аудиочасти проекта MPEG.

Аудиостандарт MPEG [ISO/IEC 93] начинается нормативным описанием формата сжатого файла для каждого из трех слоев. Затем следует нормативное описание декодера. Описание кодера (оно разное для всех слоев), а также двух психоакустических моделей содержится в описательном разделе; любой кодер, способный генерировать корректно сжатый файл, может считаться допустимым кодером MPEG. Имеется также несколько приложений, в которых обсуждаются смежные вопросы, например, защита от ошибок.

По контрасту с MPEG-видео, где имеется большое число информационных ресурсов, читателю доступно относительно малое число источников технической литературы по MPEG-аудио. Вместе со ссылками следующего абзаца можно порекомендовать MPEG консорциум [MPEG 2000]. На этом сайте имеется масса ссылок на другие ресурсы, которые время от времени обновляются. Другим источником информации может служить Ассоциация аудиоинженеров (Association of Audio Engineers, AES). Большинство идей и технических решений, использованных в аудиостандарте MPEG были опубликованы в трудах многих конференций этой организации. Однако эти материалы не являются свободно доступными и их можно получить только из AES.

Для дополнительной информации по трем слоям см. [Brandenburg, Stoll 94], [ISO/IEC 93], [Pan 95], [Rao, Hwang 96] и [Shlien 94].

При оцифровывании видеофильмов звуковая часть может состоять из двух звуковых дорожек (стереозвук), каждая из которых сэмплирована при 44.1 кГц с 16-битными звуковыми фрагментами. Это приводит к битовой скорости аудиоданных  $2 \times 44100 \times 16 = 1\,411\,200$  бит/сек, близкой к 1.5 Мбит/сек. Кроме скорости сэмплирования в 44.1 кГц предусмотрены скорости 32 кГц и 48 кГц. Важным свой-

ством MPEG аудио является возможность задания пользователем коэффициента сжатия. Стандарт позволяет получить битовую скорость сжатого звукового файла в диапазоне от 32 до 224 Кбит/сек на один аудиоканал (их обычно два для стереозвука). В зависимости от исходной частоты сэмплирования, эти битовые скорости означают фактор сжатия от 2.7 (низкий) до 24 (впечатляющий)! Причина жесткой заданности битовой скорости сжатого файла связана с необходимостью синхронизации звука и сжатого видеоряда.

В основе сжатия звука в MPEG лежит принцип квантования. Однако, квантуемые величины берутся не из звуковых сэмплов, а из чисел (называемых *сигналами*) которые выделяются из частотной области звука (это обсуждается в следующем абзаце). Тот факт, что коэффициент сжатия (или битовая скорость) известен кодеру означает, что кодер в каждый момент времени знает, сколько бит можно назначить квантуйемому сигналу. Следовательно важной частью кодера является (адаптивный) *алгоритм назначения битов*. Этот алгоритм использует известную битовую скорость и частотный спектр самых последних аудиосэмплов для определения размера квантованного сигнала так, чтобы шум квантования (разность между исходным сигналом и его квантованным образом) была неслышимой (т.е., она находится ниже порога *маскирования*, который обсуждался в § 6.3).

Психоакустические модели используют частоту сжимаемого звука, но входной файл содержит звуковые сэмплы, а не звуковые частоты. Эти частоты необходимо вычислить с помощью сэмплов. По этой причине первым шагом аудиокодера MPEG является дискретное преобразование Фурье, при котором 512 последовательных звуковых сэмплов преобразуется в частотную область. Поскольку количество частот может быть большим, их группируют в 32 подполосы одинаковой ширины. Для каждой подполосы вычисляется число, которое указывает на интенсивность звука в данной подполосе. Эти числа, называемые *сигналами*, затем квантуются. Грубость квантования на каждой подполосе определяется с помощью порога маскирования этой подполосы, а также с помощью числа оставшихся для кодирования битов. Порог маскирования для каждой подполосы вычисляется с помощью психоакустической модели.

MPEG использует две психоакустические модели для частотного и временного маскирования. Каждая модель описывает, как громкий звук маскирует другие звуки, которые близки к этому звуку по частоте или по времени. Модель разделяет область частот на

24 критические полосы и определяет, как эффекты маскирования проявляются в каждой из полос. Эффект маскирования, конечно, зависит от частот и амплитуд тонов. Когда звук разжимается и воспроизводится, пользователь (слушатель) может выбрать любую амплитуду звучания, поэтому психоакустическая модель должна быть разработана для наихудшего случая. Эффекты маскирования также зависят от природы источника сжимаемого звука. Источник может быть музыкальноподобным или шумоподобным. Две психоакустические модели основаны на результатах экспериментальной работе исследователей за многие годы.

Декодер должен быть быстрым, поскольку ему, возможно, предстоит декодировать видео и аудио в режиме реального времени. Поэтому он должен быть простым. Значит, у него нет времени использовать психоакустическую модель или алгоритм назначения битов. То есть, сжатый файл должен содержать исчерпывающую информацию, которую декодер будет использовать при деквантовании сигналов. Эта информация (размер квантованных сигналов) должна быть записана кодером в сжатый файл и она требует некоторое дополнительные расходы, которые будут удовлетворены за счет оставшихся битов.

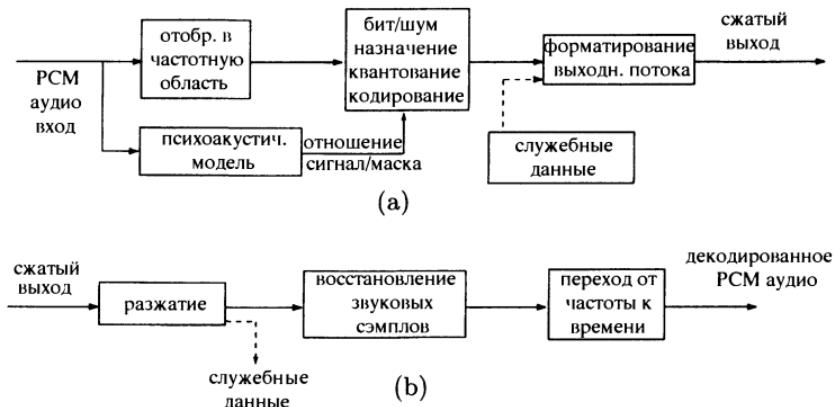


Рис. 6.8. Кодер звука MPEG (a) и его декодер (b).

На рис 6.8 приведена блок-схема основных компонентов кодера и декодера звука в MPEG. Вспомогательные данные определяются пользователем; обычно они связаны с конкретными приложениями. Эти данные не являются обязательными.

### 6.5.1. Кодирование частотной области

Первый шаг кодирования звуковых сэмплов заключается в преобразовании их в частотную область. Это делается с помощью банка многофазных фильтров, который отображает сэмплы в 32 частотные полосы равной ширины. Используемые фильтры обеспечивают быстрое преобразование с хорошим времененным и частотным разрешением. При этом разработчикам пришлось пойти на три компромисса.

Первый компромисс – это равенство ширины всех 32 частотных подполос. Это упрощает фильтры, но сильно контрастирует с особенностями слухового восприятия, которое зависит от частоты звука. В идеале было бы лучше разделить частоты на критические полосы, обсуждавшиеся в § 6.3. Эти полосы построены так, что воспринимаемая громкость данного звука и его слышимость в присутствии другого, маскирующего звука, является совместимой в пределах данной критической полосы, но различается между полосами. К сожалению, каждая из низкочастотных полос перекрывает несколько критических полос, в результате алгоритм назначения битов не может оптимизировать число присваиваемых битов квантованному сигналу в пределах этих подполос. Когда несколько критических полос накрываются подполосой X, алгоритм назначения битов выбирает критическую полосу с наименьшей шумовой маской и использует эту подполосу для нахождения числа присваиваемых битов квантованным сигналам из подполосы X.

Второй компромисс связан с обращением банка фильтров, который используется декодером. Исходное преобразование из временной области в частотную теряет часть информации (даже до квантования). Следовательно декодер получает данные, которые немного хуже; он делает обратное преобразование из частотной области во временную, и тем самым вносит еще большее искажение. Поэтому разработчики этих двух банков фильтров (для прямого и обратного преобразований) постарались минимизировать эту потерю.

Третий компромисс относится к конкретным фильтрам. Смежные фильтры должны идеально пропускать разные диапазоны частот. На практике они имеют существенное частотное перекрытие. Звук, состоящий из одного чистого тона, может попасть в два фильтра и породить сигналы (которые потом будут квантоваться) в две из 32 подполосы вместо одной.

Многофазный банк фильтров использует (помимо других промежуточных структур данных) буфер X для хранения 512 входных

сэмплов. Буфером служит очередь FIFO (first-in-first-out, первым вошел – первым вышел), которая всегда содержит не более 512 последних входных сэмплов. На рис. 6.9 показаны пять основных шагов алгоритма многофазного фильтрования.

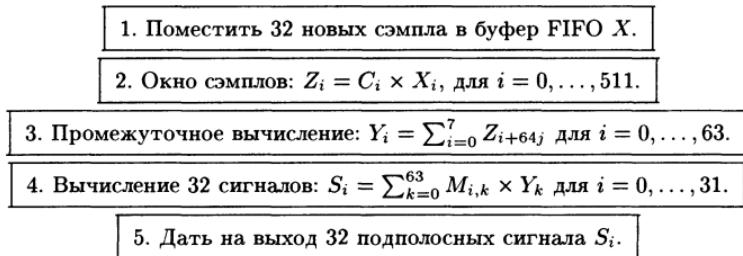


Рис. 6.9. Многофазный банк фильтров.

Алгоритм читает очередные 32 сэмпла из входного файла и заносит их в буфер, одновременно сдвигая его. Буфер всегда содержит 512 самых последних сэмплов. Сигналы  $S_t[i]$  для 32 подполос вычисляются по формуле

$$S_t[i] = \sum_{k=0}^{63} \sum_{j=0}^7 M_{i,k} (C[k + 64j] \times X[k + 64j]), \quad i = 0, \dots, 31. \quad (6.3)$$

Здесь  $S_t[i]$  обозначает сигнал подполосы  $i$  в момент времени  $t$ . Вектор  $C$  состоит из 512 коэффициентов окна анализа, которые жестко заданы стандартом.  $M$  обозначает матрицу анализа с компонентами

$$M_{i,k} = \cos \left( \frac{(2i + 1)(k - 16)\pi}{64} \right), \quad i = 0, \dots, 31; \quad k = 0, \dots, 63. \quad (6.4)$$

Отметим, что выражение в круглых скобках уравнения (6.3) не зависит от  $i$ , а числа  $M_{i,k}$  в (6.4) не зависят от  $j$ . (Эта матрица является модификацией матрицы хорошо известного преобразования DCT, поэтому она называется матрицей MDCT). Это особенность отражается в компромиссе, который позволяет уменьшить число арифметических операций. В самом деле, 32 сигнала  $S_t[i]$  вычисляются с помощью всего  $512 + 32 \times 64 = 2560$  умножений и  $64 \times 7 + 32 \times 63 = 2464$  сложений, что дает примерно 80 умножений и 80 сложений на один сигнал. Другой важный момент состоит в децимации (прореживании) сэмплов (см. § 4.4). Весь банк фильтров производит 32 выходных сигнала для 32 сэмплов. Поскольку

каждый из 32 фильтров порождает по 32 сигнала, их следует проредить, оставив только один сигнал на фильтр.

Рис. 6.10 графически иллюстрирует работу кодера и декодера при выполнении шага многофазного фильтрования. Часть (а) рисунка показывает буфер  $X$ , состоящий из 64 сегментов по 32 звуковых сэмпла в каждом. Буфер сдвигается на один сегмент вправо перед чтением следующих 32 новых сэмплов из входного файла, которые попадают в буфер слева. После умножения буфера  $X$  на коэффициенты окна  $C$ , результат помещается в вектор  $Z$ . Компоненты этого вектора делятся на сегменты по 64 числа в каждом, и эти сегменты складываются, образуя вектор  $Y$ . Вектор  $Y$  умножается на матрицу MDCT, и результат попадает в окончательный вектор из 32 компонент – сигналов подполосы.

Часть (б) рисунка показывает операции, совершаемые декодером. Группа из 32 сигналов подполосы умножается на матрицу IMDCT с компонентами  $N_{i,k}$ , и результат заносится в вектор  $V$ , состоящий из двух сегментов по 32 числа в каждом. Эти сегменты задвигаются в буфер FIFO  $V$  слева. Буфер  $V$  имеет ячейки для последних 16 векторов  $V$  (то есть, для  $16 \times 64$ , или 1024 чисел). Новый вектор  $U$  из 512 компонентов образуется из 32 альтернативных сегментов буфера  $V$ , как показано на рисунке. Затем вектор  $U$  умножается на 512 коэффициентов  $D_i$  окна синтеза (аналогично коэффициентам  $C_i$  окна анализа, используемого кодером) для вычисления вектора  $W$ . Этот вектор делится на 16 сегментов по 32 компоненты в каждом и все сегменты складываются. Результатом служат 32 реконструированных звуковых сэмпла. На рис. 6.11 приведена блок-схема, иллюстрирующая весь процесс вычислений. Компоненты матрицы синтеза IMDCT задаются формулой

$$N_{i,k} = \cos\left(\frac{(2k+1)(i+16)\pi}{64}\right), \quad i = 0, \dots, 63; \quad k = 0, \dots, 31.$$

Сигналы подполос, вычисленные на стадии фильтрования, затем собираются в кадры, содержащие по 1152 сигнала. После этого сигналы масштабируются и квантуются на основе психоакустической модели, используемой кодером, с применением алгоритма назначения битов. Квантованные величины, вместе с коэффициентами масштабирования и информацией о квантовании (число уровней квантования в каждой подполосе) записываются в сжатый файл (здесь также используется кодирование Хаффмана).

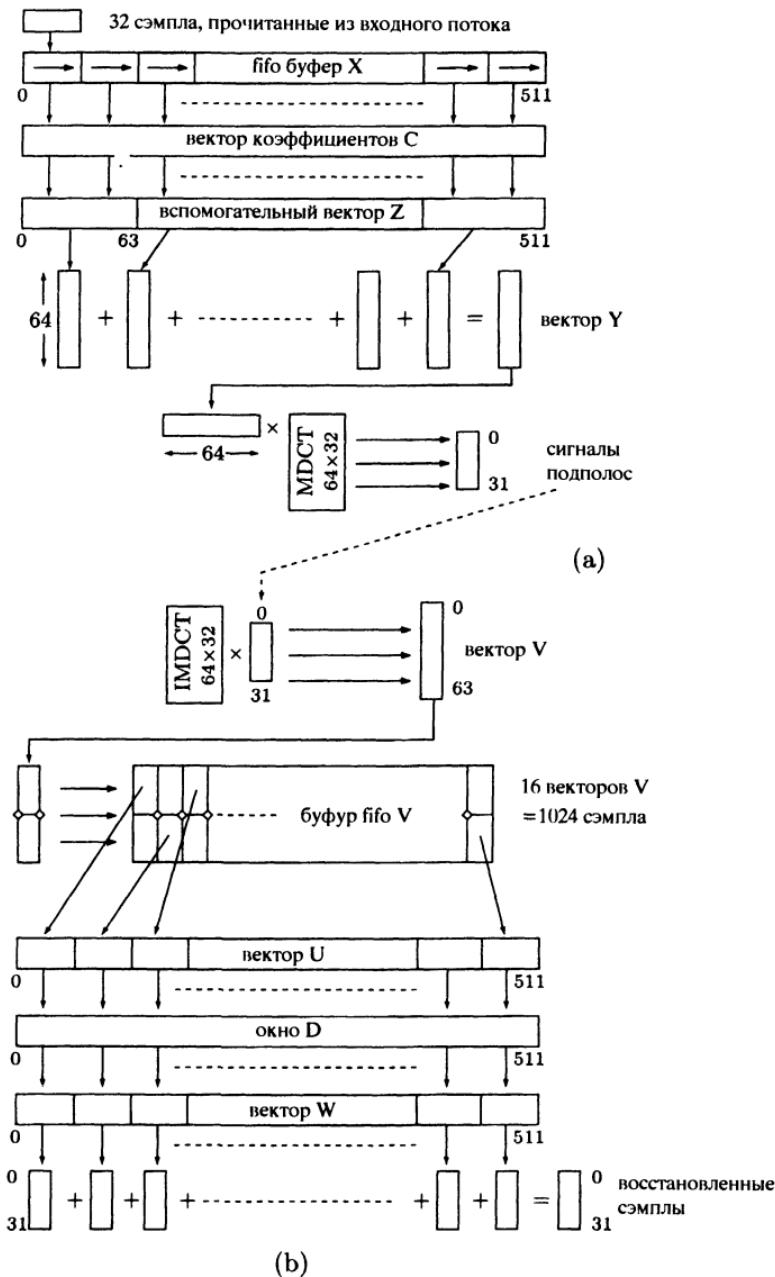


Рис. 6.10. Кодер звука MPEG (а) и его декодер (б).

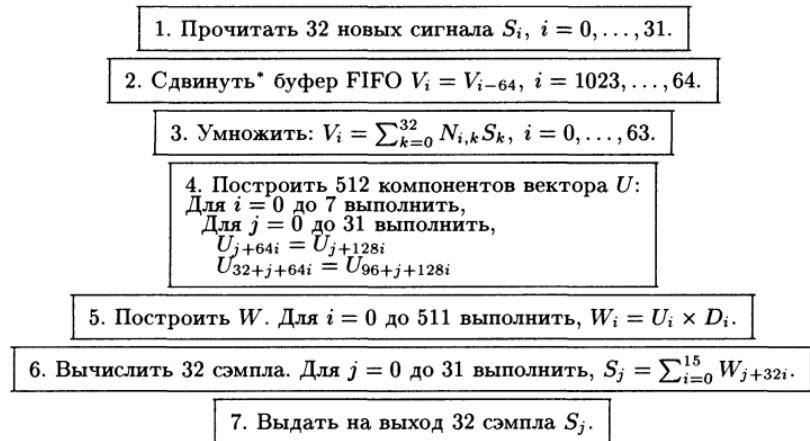


Рис. 6.11. Реконструкция аудиосэмплов.

### 6.5.2. Формат сжатых данных

Каждый кадр содержит 36 сигналов от каждой подполосы, а всего – 1152 сигнала. Сигналы кадра квантуются (этим достигается эффект компрессии) и записываются в сжатый файл вместе с другой информацией.

Каждый кадр, записанный в выходной файл, начинается заголовком из 32 бит, формат которого идентичен для всех трех слоев кодера. Заголовок имеет код синхронизации (12 битов единиц) и 20 битов параметров кодирования, которые перечислены ниже. Если применяется защита от ошибок, то за заголовком непосредственно следует 16-битное контрольное слово CRC. Затем располагаются квантованные сигналы, за которыми идет необязательный вспомогательный блок данных. Формат последних двух блоков зависит от номера слоя.

Код синхронизации нужен для того, чтобы декодер распознал заголовок кадра. Этот код состоит из 12 единиц, поэтому формат сжатого файла должен иметь такую структуру, чтобы избежать появление последовательности из 12 единиц в любом другом месте, кроме заголовков кадров.

Остальные 20 бит заголовка разделены на 12 полей следующим образом.

Поле 1. Идентификационный бит ID, значение которого равно 1 (указывает на использование MPEG). Значение 0 зарезервировано и пока не используется.



Поле 2. Два бита, обозначающие номер слоя. Значимые величины: 11 – слой I, 10 – слой II и 01 – слой III. Значение 00 зарезервировано.

Поле 3. Бит индексации использования защиты от ошибок. Нулевое значение означает, что к данным была добавлена избыточность для обнаружения возможных ошибок.

Поле 4. Четыре бита для обозначения битовой скорости. Нулевой индекс указывает на «фиксированную» скорость, когда кадр может содержать дополнительные вставки, зависящие от бита заполнения (поле 6).

Поле 5. Два бита для обозначения одну из трех возможных скоростей сэмплирования. Вот это три значения: 00 — 44.1 кГц, 01 — 48 кГц и 10 — 32 кГц. Значение 11 зарезервировано.

Поле 6. Один бит, указывающий на использование заполнения. Заполнение может добавить вставку (вставки здесь не обсуждаются) в сжатый файл после некоторого числа кадров для того, чтобы общий размер кадров был или равен, или чуть меньше суммы

$$\sum_{\substack{\text{последний кадр} \\ \text{первый кадр}}} \frac{\text{размер кадра} \times \text{битовая скорость}}{\text{частота сэмплирования}},$$

где размер кадра равен 384 сигнала для слоя I и 1152 сигнала для слоев II и III.

Следующий алгоритм может быть использован для выяснения необходимости вставки.

Для первого кадра:

`rest:=0;`

`padding:=No;`

Для каждого следующего кадра:

Если слой=I

То `dif:=(12×скорость) mod (частота сэмплир.)`

Иначе `dif:=(144×скорость) mod (частота сэмпл.)`;

`rest:=rest-dif;`

Если `rest < 0` То

`padding:=Yes;`

`rest:=rest+(частота сэмплирования);`

Иначе `padding:=No;`

Этот алгоритм имеет простую интерпретацию. Кадр делится на  $N$  или  $N + 1$  частей, где  $N$  зависит от слоя. Для слоя I число  $N$



задается выражением

$$N = 12 \times \frac{\text{битовая скорость}}{\text{частота сэмплирования}}.$$

А для слоев II и III используется следующая формула

$$N = 144 \times \frac{\text{битовая скорость}}{\text{частота сэмплирования}}.$$

Если это число не целое, то происходит его округление, и делаются вставки.

Поле 7. Один бит для частного использования кодером. Этот бит не будет использоваться ISO/IEC.

Поле 8. Двухбитное поле для указания стереозвука. Значение 00 – стереозвук, 01 – объединенное стерео (интенсивность–стерео или ms–стерео), 10 – двойной канал, 11 – одиночный канал.

Стереоинформация кодируется одним из 4 возможных мод: стерео, двойной канал, объединенное стерео и ms–стерео. В первых двух модах сэмплы от двух стереоканалов сжимаются независимо и записываются в выходной файл. Кодер не проверяет корреляцию этих двух каналов. Стерео мода используется для сжатия левого и правого стереоканалов. Мода двойной канал применяется для сжатия разные потоки звуковых сэмплов, например, параллельное вещание на двух языках. Объединенное стерео использует избыточность между левым и правым каналами, поскольку они часто идентичны, похожи или мало отличаются. Мода ms–стерео (сокращение «ms» означает «middle-side» – середина-край) является специальным случаем объединенного стерео, в котором кодируются два сигнала: среднее значение  $M_i$  и краевое значение  $S_i$  вместо левого и правого каналов  $L_i$  и  $R_i$ . Значения середина-край вычисляются по формулам

$$L_i = \frac{M_i + S_i}{\sqrt{2}} \text{ и } R_i = \frac{M_i - S_i}{\sqrt{2}}.$$

Поле 9. Двухбитовое поле расширения. Оно используется модой объединенное стерео. В слоях I и II эти биты указывают на то, какие подполосы используются для интенсивность–стерео. Все остальные подполосы кодируются в моде стерео. Имеются следующие четыре значения:

00 — подполосы 4–31 в моде интенсивность–стерео, нижний предел равен 4.

01 — подполосы 8–31 в моде интенсивность–стерео, нижний предел равен 8.



10 — подполосы 12–31 в моде интенсивность–стерео, нижний предел равен 12.

11 — подполосы 16–31 в моде интенсивность–стерео, нижний предел равен 16.

В слое III эти биты указывают на то, какой тип объединенного стерео использован при кодировании. Значения следующие:

00 — интенсивность–стерео выключено, ms-стерео выключено.

01 — интенсивность–стерео включено, ms-стерео выключено.

10 — интенсивность–стерео выключено, ms-стерео включено.

11 — интенсивность–стерео включено, ms-стерео включено.

Поле 10. Бит копирайта. Если сжатый файл защищен копирайтом, то этот бит равен 1.

Поле 11. Один бит для обозначения оригинала или копии. Значение 1 обозначает исходный сжатый файл.

Поле 12. Двухбитовое поле усиления. Указывает на тип использованного уменьшения значения. Значение 00 – никакое, 01 – 50/15 мксек, 10 – зарезервировано, 11 – уменьшение значения CCITT J.17.

### 6.5.3. Психоакустические модели

Психоакустические модели дают возможность кодеру определить порог допустимого шума квантования на каждой подполосе. Эта информация будет использоваться алгоритмом назначения битов, что в сочетании с количеством имеющихся битов задаст число уровней квантования для каждой подполосы. Стандарт MPEG устанавливает две психоакустические модели. Обе модели могут использоваться любым слоем, но только модель II выдает особую информацию для слоя III. На практике модель I используется только слоями I и II. Слой III может работать с любой моделью, но лучшие результаты получаются с моделью II.

Стандарт сжатия звука MPEG разрешает значительную свободу при реализации моделей. Изощренность этой реализации в конкретном кодере зависит от требуемой степени сжатия. В приложениях широкого потребления, в которых не требуется высокий фактор сжатия, психоакустическая модель может вовсе отсутствовать. В этом случае алгоритм назначения битов не использует соотношение SMR (signal to mask ratio, соотношение сигнал/маскирование).

Полное описание психоакустических моделей выходит за рамки этой книги. Его можно найти в разных материалах по аудиостандарту MPEG (см., например, [ISO/IEC 93], стр. 109–139). Основные шаги двух моделей состоят в следующем:

1. С помощью преобразования Фурье делается переход от исходных

звуковых сэмплов к их частотным коэффициентам. Это делается отдельно и не так, как в многофазным фильтрах, поскольку для моделей требуется более высокое разрешение для более аккуратного определения порогов маскирования.

2. Полученные частоты группируются по критическим полосам, но не по тем 32, что использовались в основной части кодера.

3. Спектральные значения критических полос разделяются на тональные (подобные синусоиде) и нетональные (шумоподобные) компоненты.

4. Перед определением порогов маскирования шумов для различных критических полос, модель применяет функцию маскирования к сигналам из разных критических полос. Эта функция находится эмпирически, то есть, из экспериментов.

5. Модель вычисляет пороги маскирования для каждой подполосы.

6. Значение SMR (signal to mask ratio, соотношение сигнал/маскирование) вычисляется для каждой подполосы. Оно равно частному от деления энергии сигнала подполосы на минимальный порог маскирования этой подполосы. Множество из 32 значений SMR, по одному на подполосу, образует выходные данные модели.

#### 6.5.4. Кодирование: слой III

Слой III использует гораздо более сложный алгоритм, чем первые два слоя. Это отражается в более высокой степени сжатия оцифрованного звука. Разница между этими слоями заметна уже на первом шаге алгоритма, который осуществляет фильтрование. Применяется один и тот же банк фильтров, но после него совершается модифицированное дискретное косинус-преобразование (MDCT). Это преобразование исправляет некоторые ошибки, внесенные многофазными фильтрами, а также подразделяет подполосы, чтобы сделать их более близкими к критическим полосам. Декодер слоя III должен применить обратное MDCT, поэтому его работа усложняется. Преобразование MDCT можно совершать либо над короткими блоками по 12 сэмплов (что дает 6 коэффициентов преобразования), либо над длинными блоками по 36 сэмплов (тогда образуется 18 коэффициентов). Независимо от выбора длины блока, последовательные преобразованные блоки имеют существенное перекрытие, как изображено на рис. 6.12. На этом рисунке блоки, показанные выше жирной линии производят лучший спектр для стационарных звуков (когда соседние сэмплы различаются не сильно), а короткие блоки более предпочтительны, когда звуковые сэмплы варьируются быстро.

Преобразование MDCT использует  $n$  входных сэмплов  $x_k$  (где



$n$  равно 36 или 12) и получает  $n/2$  (то есть 18 или 6) коэффициентов преобразования  $S_i$ . Преобразование и его обратное задаются формулами

$$S_i = \sum_{k=0}^{n-1} x_k \cos \left( \frac{\pi}{2n} \left[ 2k + 1 + \frac{n}{2} \right] (2i + 1) \right), \quad i = 0, 1, \dots, \frac{n}{2} - 1,$$

$$x_k = \sum_{i=0}^{n/2-1} S_i \cos \left( \frac{\pi}{2n} \left[ 2k + 1 + \frac{n}{2} \right] (2i + 1) \right), \quad k = 0, 1, \dots, n - 1.$$

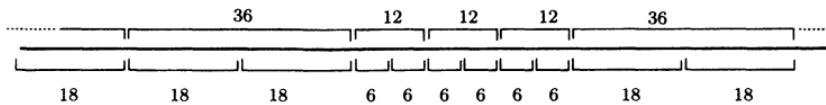


Рис. 6.12. Перекрытие окон MDCT.

Размер короткого блока составляет треть от длинного, поэтому их можно перемешивать. При построении кадра преобразование MDCT может применять или только длинные блоки, или только короткие (их будет в три раза больше), или длинные блоки для двух низкочастотных подполос, а в остальных 30 подполосах использовать короткие блоки. В этом компромиссе длинные блоки обеспечивают лучшее разрешение для низких частот, где это наиболее полезно, а короткие блоки лучше поддерживают временное разрешение для высоких частот.

Раз MDCT обеспечивает лучшее частотное разрешение, то одновременно в силу принципа неопределенности оно имеет более плохое временное разрешение [Salomon 2000]. На практике это выражается в том, что квантование коэффициентов MDCT порождает ошибки, которые распределены по времени, и кроме того образует искажения, которые проявляются в виде предшествующего эха.

Психоакустическая модель, применяемая слоем III, имеет дополнительные признаки для обнаружения пре-эха. В таких случаях слой III выполняет сложный алгоритм назначения битов, который занимает биты из доступного накопителя для того, чтобы временно увеличить число уровней квантования и тем самым побороть пре-эхо. Слой III также способен переключаться на короткие MDCT блоки, сокращая окно, если имеются подозрения в возможном появлении пре-эха.

(Психоакустическая модель слоя III вычисляет величину, которая называется «психоакустической энтропией» (РЕ) и кодер «подозревает» появление условий для пре-эха, если  $PE > 1800$ .)

Коэффициенты MDCT пропускаются через некоторый процесс удаления артефактов, которые вызваны перекрытием частот в 32 подполосах. Это называется *удалением паразитного сигнала*. Эта процедура применяется только к длинным блокам. MDCT использует 36 входных сэмплов для вычисления 18 коэффициентов, и удаление паразитного сигнала делается с помощью перекрестной операции между двумя множествами из 18 коэффициентов. Эта операция проиллюстрирована графически на рис. 6.13а, а соответствующий фрагмент программы на языке С показан на рис. 6.13б. Индекс  $i$  обозначает расстояние от последней строки предыдущего блока до первой строки текущего блока. Вычисляется 8 перекрестных операций с различными весовыми коэффициентами  $cs_i$  и  $ca_i$ , которые равны

$$cs_i = \frac{1}{\sqrt{1 + c_i^2}}, \quad ca_i = \frac{c_i}{\sqrt{1 + c_i^2}}, \quad i = 0, 1, \dots, 7.$$

Восемь величин  $c_i$  предписаны стандартом:  $-0.6, -0.535, -0.33, -0.185, -0.095, -0.041, -0.0142, -0.0037$ . На рис. 6.13с даны детали одной перекрестной операции, соответственно, для кодера и декодера.

Квантование в слое III является неравномерным. Квантователь сначала увеличивает мощность всех величин на  $3/4$ . Квантование делается по формуле

$$is(i) = \text{nint} \left[ \left( \frac{xr(i)}{\text{quant}} \right)^{3/4} - 0.0946 \right],$$

где  $xr(i)$  – абсолютная величина подполосы  $i$ , «quant» – размер шага квантования, «nint» – функция, вычисляющая ближайшее целое, а  $is(i)$  – это результат квантования. Как и в слоях I и II, квантование в слое III является серединным, то есть, значения около нуля квантуются в 0, и квантователь симметричен относительно нуля.

В слоях I и II каждая подполоса может иметь свой масштабный множитель. Слой III использует *полосы* масштабных множителей. Эти полосы перекрывают несколько MDCT коэффициентов, а их ширина близка к ширине критических полос. При этом работает особый алгоритм распределения шумов, который выбирает величины масштабных множителей.



Слой III использует коды Хаффмана для дальнейшего сжатия квантованных величин. Кодер генерирует по 18 MDCT коэффициентов на каждую подполосу. Далее происходит упорядочение получившихся 576 коэффициентов ( $= 18 \times 32$ ) в порядке возрастания частот (для коротких блоков имеется по три множества коэффициентов на каждую частоту). Напомним, что 576 коэффициентов соответствуют исходным 1152 звуковым сэмплам. Множество упорядоченных коэффициентов делится на три зоны, и каждая зона кодируется своими кодами Хаффмана. Это связано с тем, что в разных зонах коэффициенты имеют различные статистические распределения. Значения высоких частот имеют тенденцию к уменьшению и появлению нулевых серий, а низкочастотные коэффициенты, обычно, имеют большие значения. Таблицы кодов предписываются стандартом (см. 32 таблицы на стр.54–61 из [ISO/IEC 93]). Разделение квантованных величин на зоны также позволяет лучше контролировать распространение возможных ошибок.

Начав со значений наивысших частот, где встречается много нулей, кодер выбирает первую зону как последовательность нулей наивысших частот. Маловероятно, но возможно, отсутствие такой последовательности. Последовательность ограничивается четным числом нулей. Эту последовательность не нужно кодировать, так как ее значения легко определить, зная длину второй и третьей зоны. Но ее длина должна быть четной, поскольку другие две зоны кодируют свои значения группами четной длины.

Вторая зона состоит из последовательности, в которую входят лишь три значения:  $-1$ ,  $0$  и  $1$ . Эта зона называется «count1». Каждый код Хаффмана для этой зоны кодирует четыре последовательные величины, поэтому число различных кодов должно быть равно  $3^4 = 81$ . Конечно, длина этой зоны должна быть кратно 4.

Третья зона, которая состоит из «больших значений», содержит все остальные коэффициенты. Ее можно (опционно) разделить еще на три части, причем каждая может иметь свой собственный код Хаффмана. Каждый код Хаффмана кодирует два числа.

Наибольшая таблица кодов Хаффмана, предопределенная стандартом, содержит  $16 \times 16$  кодов. Более высокие значения кодируются механизмом esc-кодов.

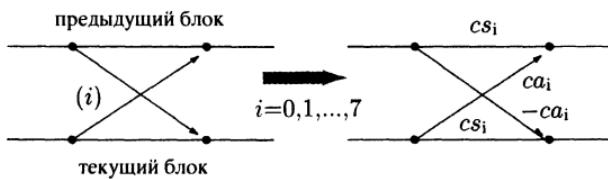
Кадр  $F$  слоя III организован следующим образом: он начинается обычным 32-битным заголовком, за которым следует (опциональный) код CRC длины 16 бит. Затем следует последовательность из 59 бит дополнительной информации. Далее располагается часть основных данных. За дополнительной информацией следует сегмент основных данных (дополнительная информация содержит, помимо прочего,



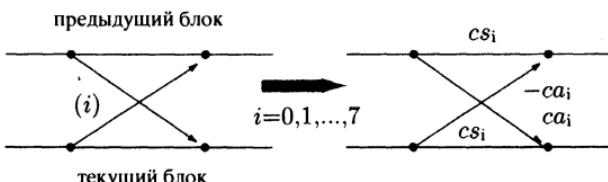
(a)

```
for(sb=1; sb<32; sb++)
  for(i=0; i<8; i++) {
    xar[18*sb-1-i]=xr[18*sb-1-i]cs[i]-xr[18*sb+i]ca[i]
    xar[18*sb+i]=xr[18*sb+i]cs[i]+xr[18*sb-1-i]ca[i]
  }
```

(b)



(c)



(d)

Рис. 6.13. Удаление паразитного сигнала в слое III.

длину сегмента данных), однако эти данные не обязательно будут данные именно кадра  $F$ ! Здесь могут находиться данные из других кадров, поскольку кодер использует накопитель битов.

Концепция накопителя битов очень полезна. Кодер может занять биты из накопителя, если ему необходимо увеличить число уровней квантования, из-за подозрения в появлении пре-эха. Кроме того, кодер может положить в накопитель биты, если ему нужно меньше бит, чем среднее их число, требуемое для кодирования кадра. Однако, занимать биты можно только при условии их возврата. Накопитель не может содержать отрицательное число битов.

Дополнительная информация о кадре включает в себя 9-битные указатели на начало основных данных кадра. Вся структура сегментов данных, а также указатели и накопитель битов показаны на рис. 6.14. На этом рисунке кадру 1 требовалось только половина его битов, поэтому вторая половина осталась в накопителе; в конечном счете эти биты были использованы кадром 2. Этому кадру необходимо немного дополнительного пространства из своего собственного сегмента, а остаток лежал в накопителе. Эти биты использовались кадрами 3 и 4. Кадру 3 вовсе не нужны его собственные биты, поэтому весь сегмент остался в накопителе и был использован кадром 4. Ему также потребовалось часть битов своего сегмента, а остаток перешел к кадру 5.



Рис. 6.14. Структура сжатого файла слоя III.

Назначение битов слоем III похоже на алгоритм слоев I и II, но имеет дополнительную сложность из-за распределения шумов. Кодер (см. рис. 6.15) распределяет биты, делает надлежащее квантование подполосных сигналов, кодирует их кодами Хаффмана и считает общее число битов, сгенерированных процессом. В этом состоит внутренний цикл назначения битов. Алгоритм распределения шумов (также называемый процедурой анализа-синтеза) становится внешним циклом, когда кодер вычисляет шум квантования (то есть, он

деквантует и восстанавливает сигналы подполос и вычисляет разность между исходным сигналом и его восстановленной копией).

Если обнаруживается, что некоторые перемасштабированные полосы имеют больший шум, чем позволяет психоакустическая модель, то кодер увеличивает число уровней квантования для этих полос и повторяет весь процесс. Процесс обрывается при выполнении следующих трех условий:

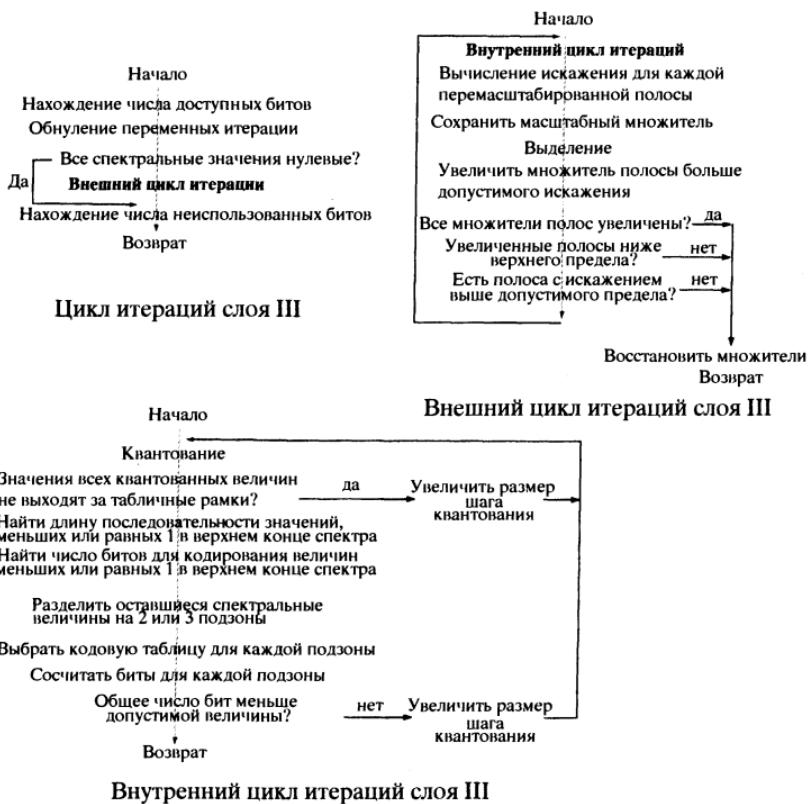


Рис. 6.15. Цикл итераций слоя III.

1. Все перемасштабированные полосы имеют допустимый шум, не превосходящий допустимый порог.
2. Следующая итерация потребует переквантования ВСЕХ перемасштабированных полос.
3. Следующей итерации потребуется больше битов, чем имеется в накопителе.



Кодер слоя III (MP3) очень сложен при программной реализации. А декодер, наоборот, достаточно прост, поскольку ему не приходится применять психоакустическую модель, не надо бороться с артефактами пре-эха и манипулировать с накопителями битов. Поэтому имеется огромное число доступных программ и приложений, которые способны проигрывать звуковые MP3-файлы на любых компьютерных plataформах.

*Было время, когда Боуман хорошо осознавал бессмысленность всего этого — сигнал тревоги немедленно раздастся, если что-то будет не так — тогда надо будет переключиться на звуковой выход. Он будет прислушиваться, наполовину загипнотизированный, к бесконечно медленным ударам сердец своих спящих коллег, вперив свой взгляд в скучные волны, которые синхронно текут по дисплеям.*

— Артур Кларк, «2001: Пространство Одиссея»

## Литература

- Ahmed N., Natarajan T., Rao R.K. (1974) "Discrete Cosine Transform," *IEEE Transactions on Computers* C-23:90–93.
- Akansu Ali, Haddad R. (1992) *Multiresolution Signal Decomposition*, San Diego, CA, Academic Press.
- Alistair A. (2001): <ftp://ftp.pd.uwa.edu.au/pub/Wavelets/>.
- Anderson K.L. и др. (1987) "Binary-Image-Manipulation Algorithm in the Image View Facility," *IBM Journal of Research and Development* 31(1):16–31, January.
- Banister B., Fischer T.R. (1999) "Quadtree Classification and TCQ Image Coding," in Storer, James A., and Martin Cohn (eds.) (1999) *DCC'99: Data Compression Conference*, Los Alamitos, CA, IEEE Computer Society Press, pp. 149–157.
- Blinn J. F. (1993) "What's the Deal with the DCT," *IEEE Computer Graphics and Applications*, pp. 78–83, July.
- Brandenburg, Heinz K., Stoll G. (1994) "ISO-MPEG-1 Audio: A Generic Standard for Coding of High-Quality Digital Audio," *Journal of the Audio Engineering Society*, 42(10):780–792, October.
- ccitt (2001):  
[URL src.doc.ic.ac.uk/computing/ccitt/ccitt-standards/1988/](http://src.doc.ic.ac.uk/computing/ccitt/ccitt-standards/1988/).
- Cleary J.G., Witten I.H. (1984) "Data Compression Using Adaptive Coding and Partial String Matching," *IEEE Transactions on Communications* COM-32(4):396–402, April.
- Daubechies I. (1988) "Orthonormal Bases of Compactly Supported Wavelets," *Communications on Pure and Applied Mathematics*, 41:909–996.
- DeVore R. и др. (1992) "Image Compression Through Wavelet Transform Coding," *IEEE Transactions on Information Theory* 38(2):719–746, March.
- Ekstrand N. (1996) "Lossless Compression of Gray Images via Context Tree Weighting," in Storer, James A. (ed.), *DCC '96: Data Compression*

Conference, Los Alamitos, CA, IEEE Computer Society Press, pp. 132–139, April.

Feig E.N., Linzer E. (1990) “Discrete Cosine Transform Algorithms for Image Data Compression,” in *Proceedings Electronic Imaging '90 East*, pp. 84–87, Boston, MA.

Funet (2001):

URL <ftp://nic.funet.fi/pub/graphics/misc/test-images/>.

Gardner M. (1972) “Mathematical Games,” *Scientific American*, **227** (2):106, August.

Golomb, S.W. (1966) “Run-Length Encodings,” *IEEE Transactions on Information Theory* IT-12(3):399–401.

Gonzalez R.C., Woods R.E. (1992) *Digital Image Processing*, Reading, MA, Addison-Wesley.

Grafica (1996): URL <http://www.sgi.com/grafica/huffman/>.

Gray F. (1953) “Pulse Code Communication,” United States Patent 2,632,058, March 17.

Heath F.G. (1972) “Origins of the Binary Code,” *Scientific American*, 227(2):76, August.

Huffman D. (1952) “A Method for the Construction of Minimum Redundancy Codes,” *Proceedings of the IRE* **40**(9): 1098–1101.

Hunter R., Robinson A.H. (1980) “International Digital Facsimile Coding Standards,” *Proceedings of the IEEE* **68**(7):854–867, July.

ISO/IEC (1993) International Standard IS 11172-3 “Information Technology—Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to about 1.5 Mbits/s—Part 3: Audio.”

Joshi R.L., Crump V.J., Fischer T.R. (1993) “Image Subband Coding Using Arithmetic and Trellis Coded Quantization,” *IEEE Transactions on Circuits and Systems Video Technology*, **5**(6):515–523, December.

Knuth, D.E. (1985) “Dynamic Huffman Coding,” *Journal of Algorithms* **6**:163–180.

Leiewer D.A., Hirschberg D.S. (1987) “Data compression,” *Computing Surveys* **19**, 3, 261–297. Reprinted in Japanese BIT Special issue in Computer Science, 16–195. Cm.

<http://www.ics.uci.edu/dan/pubs/DataCompression.html>.

Lewalle J. (1995) “Tutorial on Continuous Wavelet Analysis of Experimental Data.” Cm. <ftp.mame.syr.edu/pub/jlewalle/tutor.ps.Z>.

Linde Y., Buzo A., Gray R.M. (1980) "An Algorithm for Vector Quantization Design," *IEEE Transactions on Communications*, COM-28:84–95, January.

Loeffler C., Ligtenberg A., Moschytz G. (1989) "Practical Fast 1-D DOT Algorithms with 11 Multiplications," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP '89)*, pp. 988–991.

Manning (1998):

URL <http://www.newmediarepublic.com/dvideo/compression/>  
файл advOS.html.

Marking M. P. (1990) "Decoding Group 3 Images," *The C Users Journal*, pp. 45–54, June.

McConnell K. R. (1992) *FAX: Digital Facsimile Technology and Applications*, Norwood, MA, Artech House.

Moffat A. (1990) "Implementing the PPM Data Compression Scheme," *IEEE Transactions on Communications* COM-38(11):1917–1921, November.

Moffat A., Neal R., Witten I.H. (1998) "Arithmetic Coding Revisited," *ACM Transactions on Information Systems*, 16(3):256–294, July.

MPEG (2000): URL <http://www.mpeg.org/>.

Mulcahy C. (1996) "Plotting and Scheming with Wavelets," *Mathematics Magazine*, 69(5):323–343, December. См. также URL <http://www.spelman.edu/~colm/csam.ps>.

Mulcahy C. (1997) "Image Compression Using the Haar Wavelet Transform," *Spelman College Science and Mathematics Journal*, 1(1):22–31, April. См. также URL <http://www.spelman.edu/~colm/wav.ps>.

Ohio-state (2001):

URL <http://www.cis.ohio-state.edu/htbin/rfc/rfc804.html>.

Pan. D. Y. (1995) "A Tutorial on MPEG/Audio Compression," *IEEE Multimedia*, 2:60–74, Summer.

Pennebaker W.B., Mitchell J.L. (1992) *JPEG Still Image Data Compression Standard*, New York, Van Nostrand Reinhold.

Phillips D. (1992) "LZW Data Compression," *The Computer Application Journal*, Circuit Cellar Inc., 27:36–48, June/July.

Pohlmann K. (1985) *Principles of Digital Audio*, Indianapolis, IN, Howard Sams & Co.

- Rao K.R., Yip P. (1990) *Discrete Cosine Transform-Algorithms, Advantages, Applications*, London, Academic Press.
- Rao K. R., Hwang J.J. (1996) *Techniques and Standards for Image, Video, and Audio Coding*, Upper Saddle River, NJ, Prentice Hall.
- Rao, R. M., Bopardikar A.S. (1998) *Wavelet Transforms: Introduction to Theory and Applications*, Reading, MA, Addison-Wesley.
- Said A., Pearlman W.A. (1996) “A New Fast and Efficient Image Codec Based on Set Partitioning in Hierarchical Trees,” *IEEE Transactions on Circuits and Systems for Video Technology*, **6**(6):243–250, June.
- Salomon D. (1999) *Computer Graphics and Geometric Modeling*, New York, NY, Springer-Verlag.
- Salomon D. (2000) *Data Compression: The Complete Reference*, New York, NY, Springer-Verlag.
- Shenoi K. (1995) *Digital Signal Processing in Telecommunications*, Upper Saddle River, NJ, Prentice Hall.
- Shiien S. (1994) “Guide to MPEG-1 Audio Standard,” *IEEE Transactions on Broadcasting* **40**(4):206–218, December.
- Simoncelli E.P., Adelson E.H. (1990) “Subband Transforms,” in Woods, John, editor, *Subband Coding*, Boston, Kluwer Academic Press, 143–192.
- Stollnitz E.J., DeRose T.D., Salesin D.H. (1996) *Wavelets for Computer Graphics*, San Francisco, Morgan Kaufmann.
- Storer J.A., Szymanski T.G. (1982) “Data Compression via Textual Substitution,” *Journal of the ACM* **29**:928–951.
- Strang G., Truong Nguyen (1996) *Wavelets and Filter Banks*, Wellesley, MA, Wellesley-Cambridge Press.
- Vetterli M., Kovacevic J. (1995) *Wavelets and Subband Coding*, Englewood Cliffs, NJ, Prentice Hall.
- Vitter J. S. (1987) “Design and Analysis of Dynamic Huffman Codes,” *Journal of the ACM* **34**(4):825–845, October.
- Wallace G.K. (1991) “The JPEG Still Image Compression Standard,” *Communications of the ACM* **34**(4): 30–44, April.
- Watson A. (1994) “Image Compression Using the Discrete Cosine Transform,” *Mathematica Journal*, **4**(1):81–88.

Weinberger M. J., Seroussi G., Sapiro G. (1996) “LOCO-I: A Low Complexity, Context-Based, Lossless Image Compression Algorithm,” in *Proceedings of Data Compression Conference*, Storer J., editor, Los Alamitos, GA, IEEE Computer Society Press, pp. 140–149.

Welch T.A. (1984) “A Technique for High-Performance Data Compression,” *IEEE Computer* **17**(6):8–19, June.

Witten I.H., Neal R.M., Cleary J.G. (1987) “Arithmetic Coding for Data Compression,” *Communications of the ACM*, **30**(6):520–540.

Ziv J., Lempel A. (1977) “A Universal Algorithm for Sequential Data Compression,” *IEEE Transactions on Information Theory*, IT-23(3): 337–343.

Ziv J., Lempel A. (1978) “Compression of Individual Sequences via Variable-Rate Coding,” *IEEE Transactions on Information Theory* IT-24(5):530–536.

## Добавленная литература

Александров В.В., Горский Н.Д. *Представление и обработка изображений*. Л.: Наука, 1985.

Берлекэмп Э. *Алгебраическая теория кодирования*. М.: Мир, 1971.

Блейхут Р. *Теория и практика кодов, контролирующих ошибки*. М.: Мир, 1986.

Ватолин Д. *Алгоритмы сжатия изображений*. М.: Диалог-МГУ, 1999.

Ватолин Д., Ратушняк А., Смирнов М., Юкин В. *Методы сжатия данных. Устройство архиваторов, сжатие изображений и видео*. М.: Диалог-МИФИ, 2002.

Галлагер Р. *Теория информации и надежная связь*. М.: Советское радио, 1974.

Добеши И. *Десять лекций по вейвлетам*. М.: Ижевск, 2001.

Касами Т., Токура Н., Ивадари Е., Инагаки Я. *Теория кодирования*. М.: Мир, 1978.

Колесник В.Д., Мирончиков Е.Т. *Декодирование циклических кодов*. М.: Связь, 1968.

Кривошеев М.И. *Основы телевизионных измерений*. М.: Радио и связь, 1989.

Кричевский Р.Е. *Сжатие и поиск информации*. М.: Радио и связь, 1989.

Мак-Вильямс Ф., Слоэн Н.Дж. *Теория кодов, исправляющих ошибки*. М.: Связь, 1979.

Марков А.А. *Введение в теорию кодирования*. М.: Наука, 1982.

Питерсон У., Уэлдон Э. *Коды, исправляющие ошибки*. М.: Мир, 1976.

Фано Р.М. *Передача информации. Статистическая теория связи*. М.: Мир, 1965.

Хэмминг Р.В. *Теория кодирования и теория информации*. М.: Радио и связь, 1983.

Цифровое телевидение. Под ред. Кривошеева М.И. М.: Связь, 1980.

Чисар И., Кернер Я. *Теория информации: теоремы кодирования для дискретных систем без памяти*. М.: Мир, 1985.

Шенон К. *Работы по теории информации и кибернетике*. М.: ИЛ, 1963.

Яглом А.М., Яглом И.М. *Вероятность и информация*. М.: Наука, 1973.

Яншин В.В. *Анализ и обработка изображений (принципы и алгоритмы)*. М.: Машиностроение, 1995.

# Глоссарий

## **Адаптивное сжатие**

Метод сжатия, которые способен изменять (модифицировать) свои операции и/или параметры в зависимости от данных, поступивших из входного потока. Примером является адаптивный метод Хаффмана из § 1.5, а также словарные методы, описанные в главе 2.

## **Алфавит**

Множество всех различных символов входного потока данных. При сжатии текстов алфавитом обычно служит множество из 128 кодов ASCII. При компрессии изображений алфавитом является множество различных значений, которые могут принимать пиксели (2, 16, 256 значений или больше). (См. также Символ.)

## **Арифметическое кодирование**

Статистический метод сжатия (§ 1.7), который присваивает один (обычно, длинный) код всему входному потоку данных вместо назначения кодов индивидуальным символам алфавита. Метод считывает символы из входного потока один за другим и добавляет к коду новые биты после обработки новых символов. Арифметическое кодирование является медленной процедурой, но оно сжимает входной поток практически до его энтропии даже при сильной асимметрии вероятностей появления символов алфавита. (См. также Модель сжатия, Статистические методы.)

## **Барк**

Единица критической полосы. Названа в честь Г. Баркхозена (H. Barkhausen). Она используется в науке о звуковом восприятии. Шкала величин барк нелинейно соотносится с частотной шкалой. Она подобрана с учетом особенностей восприятия разных звуковых частот человеческим ухом.

## **Биграмм**

Пара последовательных символов.

## **Бит/Буква**

Бит на букву (pbc, bit-per-character). Мера эффективности сжатия текстовой информации. Служит также для измерения энтропии. (См. также Битовая скорость.)

## **Бит/Символ**

Бит на символ. Общая характеристика для измерения степени сжатия оцифрованных данных.

## **Битовая скорость**

В общем случае термин «Битовая скорость» (по-английски «bitrate») обозначает и bpb (bit-per-bit, бит на бит) и bpc (bit-per-character, бит на букву). Однако при сжатии звука в MPEG, этот термин используется для скорости, с которой сжатый поток считывается декодером. Эта скорость зависит от его источника или носителя (диск, канал связи, оперативная память компьютера и т.п.). Если битовая скорость аудиофайла MPEG определена, скажем, в 128 Кбит/с, то это означает, что кодер конвертирует каждую секунду исходного звукового потока в 128 Кбит сжатых данных, а декодер, соответственно, переводит каждые 128 Кбит сжатого файла в одну секунду звучания. Низкая битовая скорость означает малый размер сжатого файла. Вместе с тем, чем меньше битовая скорость, тем больше кодер должен сжать звуковую информацию, что приводит к значительной потере качества звука. Опытным путем установлено, что для качества звука, сравнимого с качеством записи на CD, битовая скорость должна находиться в интервале от 112 Кбит/с до 160 Кбит/с. (См. также Бит/Буква.)

## **Битовый слой**

Каждый пиксель цифрового изображения представлен несколькими битами. Множество, состоящее из значений битов с номером  $k$  всех пикселов изображения образует  $k$ -ый битовый слой изображения. Для примера, двухуровневое изображение состоит из одного битового слоя (См. также Двухуровневое изображение.)

## **Векторное квантование**

Это обобщение метода скалярного квантования. Оно применяется при компрессии изображений и звука. На практике векторное квантование осуществляется при сжатии оцифрованных аналоговых данных, таких как звуковые сэмплы, сканированные изображения (рисунки или фотографии). (См. также Скалярное квантование.)

**Двухуровневое изображение**

Изображение, в котором пиксели имеют всего два значения. Эти цвета, обычно, называются «черным» и «белым», «изображение» и «фон», 1 или 0. (См. также Битовый слой).

**Декодер**

Программа или алгоритм декодирования (разжатия) данных.

**Децибел**

Логарифмическая величина, которая используется для измерения параметров, имеющих очень широкий диапазон значений. Примером может служить интенсивность звука (амплитуда). Амплитуда звука может отличаться на 11–12 порядков. Вместо линейной меры, где могут понадобиться числа от 1 и до  $10^{11}$ , логарифмическая шкала использует интервал от 0 до 11.

**Дискретное косинус-преобразование (DCT)**

Вариант дискретного преобразования Фурье (DFT), которое генерирует вещественные числа. DCT (§§ 3.5.3, 3.7.2) преобразует множество из  $n$  чисел, которое рассматривается как вектор  $n$ -мерного пространства, в повернутый вектор, причем первая координата становится доминирующей. Преобразование DCT и обратное к нему, IDCT, используется в стандарте JPEG (см. § 3.7) для сжатия изображений с приемлемой потерей информации, которая происходит при отбрасывании высокочастотных компонентов образа и квантования низкочастотных составляющих.

**Дискретно-тоновое изображение**

Дискретно-тоновое изображение может быть двухуровневым, с градацией серого цвета или цветным. Такие изображения (за некоторыми исключениями) являются искусственно созданными, которые получаются при сканировании различных документов, или при захвате изображения дисплея компьютера. Как правило, величины пикселов таких изображений не меняются непрерывно или плавно при переходе от пикселя к его соседям. Обычно, пиксели имеют ограниченное множество различных значений. Значения соседних пикселов могут сильно различаться по интенсивности или цвету. (См. Непрерывно тоновые изображения.)

**Дискретное вейвлетное преобразование**

Дискретный вариант непрерывного вейвлетного преобразования. Вейвлет представлен с помощью нескольких коэффициентов фильтра,

а преобразование выполняется с помощью умножения на соответствующую матрицу вместо вычисления интеграла.

### **Изображение с градацией серого цвета**

Непрерывно тоновое изображение с оттенками одного цвета. (См. также Непрерывно тоновые изображения.)

### **Кодек**

Термин для совместного обозначения кодера и декодера.

### **Кодер**

Программа или алгоритм кодирования (сжатия) данных.

### **Кодирование длин повторов, RLE**

Общее название методов сжатия данных, при котором серии одинаковых символов заменяются одним кодом, или меткой, в которой записана длина этой серии. Обычно, метод RLE является одним из шагов многоэтапного алгоритма сжатия, в котором дополнительно совершается статистическое или словарное сжатие.

### **Кодирование Хаффмана**

Популярный метод сжатия данных (см. § 1.4). Присваивает «наилучшие» коды переменной длины множеству символов в зависимости от их вероятностей. Лежит в основе многих известных программ сжатия файлов на персональных компьютерах. Некоторые из них непосредственно применяют метод Хаффмана, а другие используют его как один из шагов многоступенчатого алгоритма компрессии. Метод Хаффмана чем-то похож на метод Шеннона–Фано. В общем случае он производит лучшие коды, и подобно методу Шеннона–Фано эти коды являются наилучшими, когда вероятности символов в точности равны отрицательным степеням числа 2. Основное различие между этими двумя методами состоит в том, что метод Шеннона–Фано строит коды сверху вниз (от самого левого к самому правому биту), а метод Хаффмана выстраивает свои коды с помощью дерева, направляясь вверх (то есть, код выписывается справа налево).

### **Коды**

Кодом называется символ, который подставляется на место другого символа. В компьютерных и телекоммуникационных приложениях коды всегда являются двоичными числами. Стандартом де-факто

выступает код ASCII. Многие новые приложения поддерживают кодовый стандарт UNICODE, а более старый стандарт EBCDIC все еще применяется в компьютерах IBM.

### Коды Грэя

Двоичные коды для представления натуральных чисел. Коды любых двух последовательных чисел различаются ровно на 1 бит. Эти коды используются при разделении изображений с градацией серого на битовые слои, каждый из которых становится двухуровневым изображением. (См. также Изображение с градацией серого цвета.)

### Коды переменной длины

Используются в статистических методах сжатия. Эти коды должны удовлетворять свойству префикса (§ 1.2) и они присваиваются входным символам в соответствии с их вероятностями. (См. также Свойство префикса, Статистические методы.)

### Конференция по сжатию данных (Data Compression Conference, DCC)

Конференция для исследователей и разработчиков в области сжатия данных. DCC происходит ежегодно в городе Сноуберд, штат Юта, США. Она происходит в марте и длится три дня.

### Корреляция

Статистическая мера линейной зависимости между двумя парными величинами. Эта величина меняется от  $-1$  (совершенная отрицательная зависимость), через  $0$  (отсутствие зависимости) и до  $+1$  (совершенная положительная зависимость).

### Коэффициент сжатия

Важная величина, которая постоянно используется для определения эффективности метода сжатия. Она равна частному

$$\text{Коэффициент сжатия} = \frac{\text{размер выходного файла}}{\text{размер входного файла}}.$$

Коэффициент 0.6 означает, что сжатые данные занимают 60% от исходного размера. Значения большие 1 говорят о том, что выходной файл больше входного (отрицательное сжатие).

Иногда величина равная  $100 \times (1 - \text{коэффициент сжатия})$  используется для определения качества сжатия. Значение, равное 60, означает, что выходной поток занимает 40% от объема исходного потока (то есть, при сжатии освободилось 60% объема). (См. также Фактор сжатия.)

## Коэффициент усиления сжатия

Это число определяется формулой

$$\text{Коэффициент усиления сжатия} = \frac{\text{контрольный размер}}{\text{сжатый размер}},$$

где контрольный размер – это либо размер входного потока, либо размер сжатого файла, произведенного некоторым эталонным методом сжатия.

## Методы LZ

Все словарные методы сжатия основаны на работах Я.Зива (J.Ziv) и А.Лемпэла (A.Lempel), опубликованных в 1977 и 1978 гг. С тех пор эти методы принято подразделять на методы LZ77 и LZ78. Эти замечательные идеи вдохновили многих исследователей, которые обобщали, улучшали и комбинировали их, например, с методом RLE или со статистическими алгоритмами. В результате появилось огромное число известных адаптивных методов сжатия текстов, изображений и звука. (См. Словарное сжатие, Сжатие скользящим окном).

## Методы LZSS

Это вариант метода LZ77 (см. § 2.2) был разработан Сторером (Storer) и Сжимански (Szymanski) в 1982 [Storer 82]. Базовый алгоритм был улучшен по трем направлениям: (1) буфер упреждения сохранялся в циклической очереди, (2) буфер поиска (словарь) хранился в виде дерева двоичного поиска и (3) метки имели два поля, а не три. (См. Методы LZ).

## Методы LZW

Это популярная версия алгоритма LZ78 (см. § 2.4) была разработана Терри Уэлчем (Terry Welch) в 1984. Его главной особенностью является удаление второго поля из метки. Метка LZW состоит только из указателя на место в словаре. В результате такая метка всегда кодирует строку из более чем одного символа.

## Модель сжатия LZW

Модель является методом предсказания (или прогнозирования вероятности) данных, которые необходимо сжать. Эта концепция весьма важна при статистическом сжатии данных. При использовании статистического сжатия модель необходимо построить до начала процесса сжатия. В простейшей модели происходит чтение всего входного потока и подсчет частоты появления каждого символа.

После этого входной поток считывается повторно, символ за символом, и производится сжатие с учетом информации о вероятностной модели. (См. также Статистические методы, Статистические модели.)

Важная особенность арифметического кодирования состоит в возможности отделения статистической модели (таблицы частот и вероятностей) от операций кодирования и декодирования. Например, легко закодировать первую половину потока данных с помощью одной модели, а вторую – с помощью другой модели.

### **Метод SPIHT**

Прогрессирующий метод кодирования, который эффективно кодирует изображения после применения вейвлетных фильтров. Этот метод является вложенным и имеет естественное сокращение информации. Его легко реализовывать, он работает быстро и дает прекрасные результаты при сжатии любых типов изображений. (См. также Методы LZW, Прогрессирующее сжатие изображений, Дискретное вейвлетное преобразование.)

### **Непрерывно-тоновое изображение**

Цифровое изображение с большим числом цветов, в котором соседние области окрашены в непрерывно меняющиеся цвета, то есть разность значений соседних пикселов мало и незаметно глазу. Примером может служить изображение с 256 градациями серого цвета. Если соседние пиксели такого изображения имеют соседние значения, то они будут восприниматься глазом, как непрерывно меняющиеся оттенки серого цвета. (См. также Двухуровневое изображение, Дискретно-тоновое изображение, Изображение с градацией серого цвета.)

### **Отсчеты**

Коэффициенты вейвлетного преобразования. (См. также Дискретное вейвлетное преобразование.)

### **Пел**

Наименьшая единица факсимильного изображения, точка. (См. также Пиксел.)

### **Пиксел**

Наименьшая единица цифрового изображения, точка. (См. также Пел.)



## Предсказание

Приписывание некоторых вероятностей символам алфавита.

## Преобразование изображений

Изображение можно сжать с помощью перевода его пикселов (которые бывают коррелированными) в другое представление, которое является *декоррелированным*. Сжатие достигается, если новые значения меньше, в среднем, чем исходные величины. Сжатие с потерями можно получить, если дополнительно квантовать преобразованные значения. Декодер получает преобразованные величины из сжатого потока и реконструирует (точно или приближенно) исходные данные с помощью применения обратного преобразования. (См. также Дискретное косинус-преобразование, Дискретное вейвлетное преобразование.)

## Прогрессирующее сжатие изображений

Метод компрессии изображений, при котором сжатый поток состоит из нескольких «слоев», причем каждый следующий слой несет все более детальную информацию об изображении. Декодер может быстро показать весь образ с грубым разрешением, а затем улучшать качество картинки по мере приема и декодирования следующих слоев. Зритель, наблюдающий на экране декодируемое изображение, сможет получить представление о нем после разжатия всего 5–10% образа. Улучшение качества изображения можно добиться с помощью (1) повышения резкости, (2) добавления цветов или (3) увеличения разрешения.

## Психоакустическая модель

Математическая модель маскирования звуков близких частот при восприятии органами слуха человека (ухо–мозг).

## Свойство префикса

Один из принципов построения кодов переменной длины. Если некоторая последовательность битов выбрана в качестве кода определенного символа, то никакой другой код не может начинаться с этой же последовательности битов (не должен быть префиксом другого кода). Например, если строка «1» назначена кодом символа  $a_1$ , то никакой другой код не может начинаться с 1 (т.е., все другие коды будут начинаться с 0). Если код «01» присвоен символу  $a_2$ , то в начале другого кода не должна стоять последовательность 01 (они могут начинаться с 00). (См. также Коды переменной длины, Статистические методы.)

## Сжатие без потерь

Метод сжатия, при котором выход декодера всегда тождественно совпадает с исходными данными, поступившими на вход кодера. (См. также Сжатие с потерями.)

## Сжатие видео

Сжатие видеоданных основано на двух принципах. Первый состоит в использовании пространственной избыточности, которая присутствует в каждом отдельном кадре. Второй базируется на том свойстве, что текущий кадр часто бывает близок или похож на соседние кадры. Это свойство называется временной избыточностью. Типичный алгоритм сжатия видеоряда начинает со сжатия первого кадра с применением некоторого эффективного метода сжатия изображений, после чего каждый следующий кадр будет представляться разностью с одним из своих предшественников, причем кодироваться будет только эта разность.

## Сжатие с потерями

Метод сжатия, при котором выход декодера отличается от исходных данных, сжатых кодером, но результат, тем не менее, устраивает пользователя. Такие методы применяются при сжатии изображений и звука, но они приемлемы при компрессии текстовой информации, в которой потеря хоть одной буквы может привести к двусмысленному и непонятному тексту (например, при сжатии исходных текстов компьютерных программ). (См. также Сжатие без потерь.)

## Сжатие скользящим окном

Метод LZ77 (§ 2.1) использует часть ранее полученного потока данных в качестве словаря. Кодер строит окно во входных данных, в которое задвигаются справа налево поступающие символы для кодирования. Этот метод основан на скользящем окне. (См. также Методы LZ.)

## Символ

Наименьшая единица в сжимаемых данных. Обычно, символом служит один байт, но им может быть один бит, или элемент из множества  $\{0, 1, 2\}$ , или еще что-то.



## Скалярное квантование

В словаре термин «квантование» определяется как «сокращение некоторой точной величины до ограниченного, дискретного множества значений». Если данные, которые необходимо сжать состоят из больших чисел, то при квантовании они переводятся в меньшие числа. Результатом является сжатие с потерей. Если необходимо сжать аналоговые данные (например, электрический сигнал переменного напряжения), то квантование означает оцифровывание. Такое квантование часто применяется в методах сжатия звука. (См. также Векторное квантование.)

## Словарное сжатие

Метод компрессии данных (см. главу 2), которые сохраняют некоторые образцы данных в специальной структуре, называемой «словарем» (обычно, это дерево). Если строка новых данных на входе тождественна некоторому образцу из словаря, то в выходной поток или файл записывается указатель на этот образец. (См. также Методы LZ.)

## Стандарт JFIF

Полное название этого метода (§ 3.7.8) – JPEG File Interchange Format (формат обмена файлами стандарта JPEG). Это формат графических файлов, который позволяет компьютерам обмениваться сжатыми изображениями стандарта JPEG. Главная особенность этого формата состоит в использовании трехбайтового цветового пространства YCbCr (или однобайтового для образов с градацией серого). Кроме того в файл добавляются некоторые атрибуты, которых нет в формате JPEG, а именно, разрешение изображения, геометрический размер пикселя и некоторые другие параметры, специфические для конкретных приложений.

## Стандарт JPEG

Весьма изощренный метод сжатия с потерями для компрессии цветных изображений (не анимации). Он отлично работает с непрерывно тоновыми изображениями, в которых соседние пиксели имеют близкие значения. Достоинство метода состоит в использовании большого числа легко настраиваемых параметров, которые дают пользователю возможность контролировать долю отбрасываемой информации (то есть, степень сжатия образа) в весьма широких пределах. Имеется две основные моды: с потерей информации (базовая мода) и без потери информации (которая дает коэффициент сжатия порядка 2:1 и выше). Большинство популярных приложений

поддерживают только базовую моду. Эта мода включает прогрессирующее кодирование и иерархическое кодирование.

Главная идея стандарта JPEG состоит в том, что изображения создаются для того, чтобы люди на них смотрели, поэтому при сжатии допустимо отбрасывание части информации изображения, которое не заметно или не воспринимается глазом человека.

Аббревиатура JPEG означает Joint Photographic Experts Group (объединенная группа по фотографии). Проект JPEG был инициирован совместно комитетом CCITT и организацией ISO в июле 1987 года. Этот стандарт был признан во всем мире. Он широко используется в представлении графических образов, особенно на страницах всемирной паутины. (См. также JPEG-LS, MPEG.)

### **Стандарт JPEG-LS**

Мода без потерь стандарта JPEG является неэффективной и поэтому ее редко реализуют в конкретных приложениях. Поэтому ISO приняло решение разработать новый стандарт для сжатия без потерь (или почти без потерь) непрерывно тоновых изображений. В результате появился известный стандарт JPEG-LS. Этот метод не является простым расширением или модификацией JPEG. Это новый метод, простой и быстрый. В нем не используется ни DCT, ни арифметическое кодирование, но зато применяется весьма ограниченное квантование (в моде с почти без потерями). JPEG-LS проверяет несколько предыдущих соседей текущего пикселя и использует их в качестве *контекста* этого пикселя. С помощью контекста делается прогноз текущего пикселя и выбирается некоторое распределение вероятностей из имеющегося семейства распределений. На основе выбранного распределения определяется код ошибки прогноза с помощью специального кода Голомба. Существует также мода кодирования длины повторов, в которой кодируются длины повторяющихся последовательностей одинаковых пикселов. (См. также JPEG.)

### **Статистическая модель**

См. Модель компрессии.

### **Статистические методы**

Эти методы (глава 1) выполняют присвоение символам из потока данных кодов переменной длины, причем более короткие коды назначаются символам или группам символов, которые чаще встречаются во входном потоке (имеют большую вероятность появления). (См. также Коды переменной длины, Свойство префикса, Кодирование Хаффмана, Арифметическое кодирование.)

## Теория информации

Математическая теория, которая придает точный количественный смысл понятию информации. Она определяет, как измерить информацию и ответить на вопрос: сколько информации содержится в том или ином массиве данных? Ответом выступает точное число. Теория информации была создана в 1948 в работах Клода Шеннона.

## Факсимильное сжатие

Передача типичной страницы между двумя факс-машинами по телефонным линиям связи без использования компрессии может занять от 10 до 11 минут. По этой причине комитет ITU разработал несколько стандартов сжатия факсимильных сообщений. Общепринятыми стандартами на сегодняшний день являются (см. § 1.6) T4 и T6, которые еще называются Group 3 и Group 4, соответственно.

## Фактор сжатия

Величина, обратная коэффициенту сжатия. Определяется по формуле

$$\text{Фактор сжатия} = \frac{\text{размер входного файла}}{\text{размер выходного файла}}.$$

Значение, большее 1, означает сжатие, а меньшее 1 – расширение. (См. также Коэффициент сжатия.)

## Энтропийное кодирование

Метод сжатия без потери информации, при котором данные сжимаются так, что среднее число битов на символ стремится к энтропии входного источника символов.

*Потребность – вот мать сжатия.  
— Эзоп (перефразированное)*

## Сообщество сжатия данных

Читатели, которые заинтересовались проблемами сжатия информации, могут присоединиться к «Сообществу сжатия данных». Можно также поучаствовать в Конференция по сжатию данных (Data Compression Conference), которая проходит ежегодно в Сноуберде, штат Юта, США. Она проводится в марте и длится три дня. Более подробную информацию можно обнаружить на сайте конференции <http://www.cs.brandeis.edu/dcc/index.html>. На этой странице также находятся сведения об организаторах конференции и ее географическом положении.

Помимо выступлений приглашенных докладчиков и технических сессий, на конференции проводятся «промежуточные встречи», на которых обсуждаются текущие научные проблемы и выставляются стенды участников конференции. После докладов происходит неформальное общение лекторов и слушателей в очень приятной и дружеской атмосфере.

Каждый год на конференции присуждается премия имени Капочелли за самую интересную студенческую работу в области сжатия данных.

Из программы конференции можно узнать «who's who» в сжатии данных, но две центральных фигуры – это профессор Джеймс Сторер (James Storer) и профессор Мартин Кон (Martin Cohn). Первый является председателем конференции, а второй – председателем программного комитета.

Публикуемые материалы конференции по традиции выходят под редакцией Сторера и Кона. Эти труды издаются в IEEE Computer Society (<http://www.computer.org/>).

Полная библиография (в формате  $\text{\TeX}$ ) статей, опубликованных на последних конференциях DCC находится по следующему адресу <http://www.cs.mu.oz.au/älistair/dccrefs.bib>.

## Список алгоритмов

	§§	Стр.
<b>Глава 1</b>		
Кодирование Хаффмана	1.4	30
Адаптивное кодирование Хаффмана	1.5	40
Модификация кодов Хаффмана	1.5.5	51
Факсимильная компрессия	1.6	52
Арифметическое кодирование	1.7	62
Адаптивное арифметическое кодирование	1.8	76
<b>Глава 2</b>	§§	Стр.
LZ77 (Скользящее окно)	2.1	84
LZSS	2.2	88
LZ78	2.3	93
LZW	2.4	97
<b>Глава 3</b>	§§	Стр.
Интуитивные методы	3.4	137
Преобразования изображений	3.5	139
Прогрессирующее сжатие изображений	3.6	174
JPEG	3.7	182
JPEG-LS	3.8	205
<b>Глава 4</b>	§§	Стр.
Преобразование Хаара	4.2	232
Поддиапазонные преобразования	4.3	236
Банки фильтров	4.4	242
Вывод коэффициентов фильтров	4.5	250
Дискретное вейвлетное преобразование	4.6	252
Вейвлеты Добеши	4.8	264
SPIHT	4.9	267
<b>Глава 5</b>	§§	Стр.
Подоптимальные методы поиска	5.2	294



<b>Глава 6</b>	<b>§§</b>	<b>Стр.</b>
Оцифрованный звук	6.2	309
Органы слуха человека	6.3	312
Дискретное вейвлетное преобразование	6.4	316
MPEG-1	6.5	320

*Перед тем, как сжать, надо хорошенько расширить.  
— Лао-Тсу, стих 36, Тао Тэ Чинг.*

## Предметный указатель

- адаптивное сжатие, 20, 29
- аэбука Брайля, 12
- алфавит, 23, 346
- амплитудная модуляция, 312
- аналоговые данные, 355
- арифметическое кодирование, 12, 62, 346, 352, 356
  - адаптивное, 76, 121
  - и JPEG, 185, 199
  - и звук, 317
  - и кодер QM, 185, 199
  - и сжатие видео, 294
- асимметричное сжатие, 21, 83, 86
- ассоциация аудиоинженеров, 321
- аудиосэмплирование, 14
- аудиофайл MP3, 304, 320, 321, 339
- банк фильтров, 214, 242
  - биортогональный, 247
  - декимация, 243
  - ортогональный, 246
- барт, 315, 346
- бел, 306
- биграмм, 25, 346
- биортогональный фильтр, 247
- бит/бит, 22
- бит/буква, 347
- бит/символ, 22, 347
- битовая скорость, 22, 347
- битовый бюджет, 22
- битовый слой, 347
  - и коды Грэя, 126
  - и корреляция пикселов, 128
- блоковая декомпозиция, 124
- блочные артефакты, 190, 249
- вейвлеты, 214, 233, 253, 267
  - Добеши, 246, 251, 252, 254, 258, 259, 263, 264, 267, 268
- Хаара, 232
- вектор перемещения, 290
- векторное квантование, 138, 139
- вероятностная модель, 23
- вероятность
  - символа, 25
  - строки, 26
- вложенное кодирование, 269
- герц, 305
- голосовой диапазон, 313
- двоичное сбалансированное дерево, 78
- двоичный поиск, 78, 80
- двухуровневое изображение, 36, 118, 120, 135, 348
- декодер, 20, 348
- декодирование Хаффмана, 38
- декомпрессор, 20
- декоррелированные пиксели, 115–117, 139, 140, 167
- декорреляция, 123, 140, 167
- дерево
  - Хаффмана, 38, 39
  - двоичного поиска, 88, 90, 351
- логарифмическое, 247
- полное, 78
- пространственно



- ориентированное, 267, 276, 282  
с мультиразрешением, 247  
сбалансированное, 78  
децибел, 135, 306, 348  
декимация, 243, 325  
динамический словарь, 81  
дискретное  
вейвлетное  
преобразование, 13, 214, 252, 253, 256, 259  
косинус-преобразование, 149, 150, 152, 155, 157  
преобразование Фурье, 322  
синус-преобразование, 162  
дисперсия, 117, 142, 174
- закон Ома, 307  
звуковой сэмпл, 309
- иерархическое кодирование, 176  
избыточность, 15, 17, 27, 113  
и сжатие данных, 15  
пространственная, 115  
изображение, 118  
двухуровневое, 36, 118, 120, 135, 348  
дискретно-тоновое, 119, 348  
непрерывно-тоновое, 119  
подобное мультфильму, 120  
полутоновое, 118  
с градацией серого цвета, 118, 122, 349  
цветное, 119  
цифровое, 118
- импульсная кодовая  
модуляция, 312
- кадр  
внешний, 287
- внутренний, 287  
дву направлений, 288  
квантование, 138, 144, 192, 322, 347, 355  
ковариация, 116  
кодек, 20, 349  
кодер, 20, 349  
энтропийный, 26
- кодирование  
иерархическое, 176  
кодирование Хаффмана, 12, 30, 31, 34, 37  
адаптивное, 20, 40, 346  
и сжатие видео, 294  
и JPEG, 184, 196  
и вейвлеты, 216  
и звук, 317  
полуадаптивное, 41  
кодирование длин серий, 36, 120
- кодовое переполнение, 47  
коды  
ASCII, 15, 23, 88, 114, 346  
EBCDIC, 88, 350  
esc, 42  
UNICODE, 90  
Баудота, 134  
Брайля, 12  
Голомба, 205  
Грея, 125–127, 134  
переменной длины, 26, 34, 52
- компактный носитель, 232, 253, 258  
компрессор, 20  
конечный автомат, 37  
концентрация энергии, 142–145, 147, 149, 172, 264, 268
- корреляция, 350  
корреляция пикселов, 113  
коэффициент

- DC, 145
- сжатия, 21, 350
  - в UNIX, 96
  - известный заранее, 138, 288, 319
  - усиления сжатия, 351
- коэффициенты
  - AC, 145
  - корреляции, 116
  - фильтров, 250
- лексикографический
  - порядок, 88
- линейные системы, 237
- медиана, 181
- метка
  - в LZ77, 84, 85
  - в LZ78, 93
  - в LZSS, 88
  - в LZW, 97
  - словарная, 81
- метод
  - JPEG-LS, 205, 207
  - LZ77, 84, 97, 351
  - LZ78, 93, 351
  - LZC, 96
  - LZH, 85
  - LZSS, 88, 351
  - LZW, 97, 101, 351
  - MPEG-1, 189, 294, 320
- метрики ошибок, 134
- модель
  - PPM, 80
  - адаптивная, 180
  - вероятностная, 77
  - марковская, 125
  - психоакустическая, 320, 322, 331
  - статистическая, 64
- накопленная частота, 77
- неадаптивное сжатие, 20
- непрерывно тоновое
  - изображение, 119, 352
- непрерывное вейвлетное
  - преобразование, 247, 253
- нечетная функция, 162
- обратное дискретное
  - косинус-преобразование, 153
- октава, 248
- органы
  - зрения человека, 188
  - слуха человека, 312
- ортогональные
  - базисы, 248, 253
  - матрицы, 235, 236
  - преобразования, 220, 236
  - фильтры, 246
- отношение сигнал/шум, 136
- оцифрованный звук, 304, 309
- пел, 53, 352
- перекрестная корреляция, 116
- переполнение счетчика, 46
- пиковое отношение
  - сигнал/шум, 134
- пиксель, 352
  - изображения (черный), 348
  - фона (белый), 348
- пирамидальное разложение, 222, 223
- поддиапазонные
  - преобразования, 214, 220, 236
- полутоновое изображение, 118
- пре-эхо, 333
- преобразование
  - Кархунена–Лоэвэ, 149, 172
  - Уолша–Адамара, 149
  - Хаара, 13, 149, 170, 215, 218,

- 220, 225, 229, 232
- изображений, 139, 227, 353
- преобразования
  - ортогональные, 220, 236
  - поддиапазонные, 214, 220, 236
- префикс, 28
- префиксные коды, 29
- принцип
  - неопределенности, 333
- протокол V.32, 43
- разложение
  - пирамидальное, 222, 223
  - стандартное, 218, 220
- разрешение изображения, 118
- распределение Лапласа, 123
- растровое сканирование, 140, 175, 298
- рефлексивные коды Грея, 125
- самоподобие
  - изображений, 125
  - свертка, 237, 240
  - светимость, 187, 188, 225
  - сжатие видео, 286, 354
  - вычитание, 288
  - вычитание по блокам, 289
  - компенсация движения, 290
  - мера отклонения, 292
  - многомерный поиск, 302
  - основы, 287
  - подоптимальный
    - поиск, 293
  - прореживание, 288
  - сегментация, 290
- сжатие данных, 20
  - адаптивное, 20, 29
  - асимметричное, 21, 83, 86
  - без потерь, 20, 354
  - двуихпроходное, 20, 29, 40, 63
- и избыточность, 15, 113
- и квантование, 138
- и нерелевантность, 113
- логическое, 81
- неадаптивное, 20
- полуадаптивное, 20, 29, 41
- с потерями, 20, 354
- симметричное, 21, 185, 207
- сжатие двухуровневых
  - изображений, 125
- сжатие звука, 304
  - и LZ, 83
  - и MP3, 21
  - и MPEG-1, 320
  - и временное
    - маскирование, 313
  - и подавление пауз, 304, 318
  - и уплотнение, 304, 318
  - и частотное
    - маскирование, 313
- сжатие изображений, 13, 22, 111
  - JPEG, 182
  - JPEG-LS, 205
  - QTCQ, 284
  - RLE, 114, 120, 154, 185, 196
  - SPIHT, 181, 267, 352
  - двуихуровневых, 120
  - и LZ, 83
  - и вложенное
    - кодирование, 269
  - и метрики ошибок, 134
  - интуитивное, 137
  - по самоподобию, 125
  - прогрессирующее, 174, 176
  - с потерями, 113
  - словарными методами, 114
  - фрактальное, 124
- сжатие отпечатков
  - пальцев, 260
- сжатие текста
  - и арифметическое

- кодирование, 62  
и кодирование  
Хаффмана, 30  
и методы LZ, 84  
синфазные коды, 43  
синхронизация кодера и  
декодера, 42  
скорость Найквиста, 311  
словарные методы  
сжатия, 81, 114  
случайные данные, 17  
собственные числа матрицы,  
174  
среднеквадратическая  
ошибка, 135  
среднеквадратическое  
отклонение, 292  
средняя абсолютная  
ошибка, 292  
стандарт  
JFIF, 202  
JPEG, 183, 192  
JPEG2000, 189  
MPEG-1, 189  
стандартное разложение,  
218, 220  
статистические методы, 25  
статический словарь, 81  
стек, 110  
структуры данных, 47, 51, 77,  
87, 94, 104, 105, 110, 194,  
276  
сэмплирование звука, 309  
теория информации, 357  
и избыточность, 27  
точка безызбыточности, 93  
укрупнение пикселов, 183  
унарные коды, 195  
устройство  
АЦП, 309  
ЦАП, 310  
факсимильное сжатие, 12, 52  
group 3, 54  
двумерное, 58  
одномерное, 53  
фактор сжатия, 22, 357  
фильтр  
нахождение  
коэффициентов, 250  
с конечным импульсным  
откликом, 243, 253  
с пропусканием высоких  
частот, 243  
с пропусканием низких  
частот, 243, 252  
хеширование, 104, 107  
хроматические диаграммы,  
187  
цветное изображение, 119  
цветность, 225  
цветовое пространство, 187  
CMYK, 119  
HLS, 119  
RGB, 119, 133, 187  
YCbCr, 188, 189, 202  
YPbPr, 188  
циклическая очередь, 87  
частотная  
модуляция, 312  
область, 315  
частотное маскирование, 313,  
323  
четная функция, 162  
энергия, 147  
энтропия, 25, 26, 75  
психоакустическая, 334  
эталонные документы, 29, 54

Заявки на книги присылайте по адресу:

125319 Москва, а/я 594

Издательство «Техносфера»

e-mail: [knigi@technosphera.ru](mailto:knigi@technosphera.ru)

[sales@technosphera.ru](mailto:sales@technosphera.ru)

факс: (095) 956 33 46

В заявке обязательно указывайте  
свой почтовый адрес!

Подробная информация о книгах на сайтах

<http://www.technosphera.ru>

**Д. Сэломон  
Сжатие данных, изображений и звука**

Компьютерная верстка – В. В. Чепыжков  
Дизайн книжных серий – С. Ю. Биричев  
Ответственный за выпуск – Л. Ф. Соловейчик

---

Формат 84x109 1/32. Печать офсетная.

Гарнитура Computer modern LaTeX.

Печ.л.23. Тираж 3000 экз. Зак. № 10607

Бумага офсет №1, плотность 65 г/м<sup>2</sup>.

---

Издательство «Техносфера»  
Москва, ул. Тверская, дом 10 строение 3

Диапозитивы изготовлены ООО «Европолиграфик»

Отпечатано в ППП «Типография «Наука»

Академиздатцентра «Наука» РАН,

121099 Москва, Шубинский пер., 6