Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Магнитогорский государственный технический университет им. Г.И. Носова»

И.И. Баранкова
М.В. Коновалов

ТЕОРИЯ ИНФОРМАЦИИ. КОДИРОВАНИЕ

*Утверждено Редакционно-издательским советом университета
в качестве учебного пособия*

Магнитогорск
2017

УДК 621.391

Рецензенты:

кандидат технических наук,
заместитель директора
ООО «ТЕХНОАП» ИНЖИНИРИНГ»
Д.В. Швидченко

кандидат технических наук,
доцент кафедры электроники и микроэлектроники,
ФГБОУ ВО «Магнитогорский государственный технический
университет им. Г.И. Носова»
Н.В. Швидченко

Баранкова И.И., Коновалов М.В.
Теория информации. Кодирование [Электронный ресурс] : учебное пособие / Инна Ильинична Баранкова, Максим Владимирович Коновалов ; ФГБОУ ВО «Магнитогорский государственный технический университет им. Г.И. Носова». – Электрон. текстовые дан. (1,94 Мб). – Магнитогорск : ФГБОУ ВО «МГТУ им. Г.И. Носова», 2017. – 1 электрон. опт. диск (CD-R). – Систем. требования : IBM PC, любой, более l GHz ; 512 Мб RAM ; 10 Мб HDD ; MS Windows XP и выше ; Adobe Reader 8.0 и выше ; CD/DVD-ROM дисковод ; мышь. – Загл. с титул. экрана.
ISBN 978-5-9967-1073-7

В пособии рассмотрены основные способы кодирования данных в информационных системах, способы эффективного кодирования данных, способы помехоустойчиво кодирования, способы статистического и адаптивного кодирования, рассмотрены популярные алгоритмы сжатия изображений и видеофайлов.

Пособие соответствует требованиям ФГОС ВО и образовательной программе по специальности 10.05.03 «Информационная безопасность автоматизированных систем», специализация «Обеспечение информационной безопасности распределенных информационных систем». Данное пособие полностью соответствует рабочей программе по дисциплине «Теория информации», а так же может использоваться в рамках изучения дисциплины: «Алгоритмы шифрования информации».

УДК 621.391

ISBN 978-5-9967-1073-7

Содержание

ВВЕДЕНИЕ

Одним из разделов теории информации является теория кодирования. Теория кодирования изучает способы отображения дискретных сообщений сигналами в виде определенных сочетаний символов.

Первые попытки передавать информацию без использования письменных сообщений предпринимались еще в древности (система сигнальных костров использовалась в Древнем Китае). Развитие средств связи в современном их понимании началось в 19 веке и продолжается до настоящего времени.

Примеры устройств для передачи информации:

- Оптический телеграф – семафор – впервые использовал Клод Шапп (1791);
- Электромагнитный телеграф П.Л. Шиллинга (1832);
- Азбука и телеграфный аппарат Морзе (1837);
- Международный флажковый код(1861);
- Беспроводной телеграф А.С. Попова (1895);
- Беспроводной телефон, телевидение (1935).

Теорию кодирования условно можно разделить на три зардела:

- Разработка принципов наиболее экономного представления информации (сжатие информации);
- Согласование параметров передаваемой информации с особенностями канала связи (разработка транспортных протоколов);
- Разработка приемов повышения надежности передачи информации (помехоустойчивое кодирование).

В 1948 г. Клод Шеннон публикует статью «Математическая теория связи». Идеи, предложенные Шенноном в статье, послужили толчком в развитии теории кодирования и послужили основой для последующих научных исследований в области обработки, передачи и хранения информации.

В общем случае под кодированием понимается переход от одного способа представления информации к другому, допускающий восстановление исходной информации [].

Алфавит, с помощью которого представлена информация до кодирования, называется первичным, а алфавит, на котором записано сообщение после кодирования называется вторичным.

Понятие «код» имеет два определения.

Код – правило, описывающее соответствие знаков или их сочетаний первичного алфавита знакам или их сочетаниям вторичного алфавита.

Код – набор знаков вторичного алфавита, используемый для представления знаков или их сочетаний первичного алфавита.

Кодирование – процесс перевода информации из первичного алфавита в код, записанный, из символов вторичного алфавита.

Декодирование – перевод последовательности кодов в соответствующий набор символов первичного алфавита.

Кодер – устройство, выполняющее операции кодирования.

Декодер – устройство, выполняющее декодирование.

В качестве примеров кодирования можно привести следующие:

- перевод текста с русского языка на английский язык (первичный алфавит – алфавит русского языка, вторичный алфавит – алфавит английского языка);

- ввод и сохранение текста на компьютере (первичный алфавит – алфавит русского языка, вторичный алфавит — последовательность «нулей» и «единиц»);
- Азбука Морзе (первичный алфавит – алфавит русского языка, вторичный алфавит – последовательность точек и тире).

ПРЕДСТАВЛЕНИЕ ДАННЫХ В ПАМЯТИ ЭВМ

Вся информация находящаяся в оперативной памяти и хранящаяся на жёстких дисках электронно-вычислительной машины (ЭВМ) представлена в двоичной системе исчисления.

Минимальный объем ячейки памяти ЭВМ составляет 8 бит (1 байт). У каждого ячейки есть свой адрес. Последовательность бит, которую ЭВМ может обрабатывать как единое целое, называют машинным словом. Длина машинного слова зависит от разрядности процессора и может быть равной 16, 32 битам и т. д.

Кодирование символов

Для кодирования символов в современных ЭВМ используется кодировка UTF-16 (Unicode Transformation Format), где 16‑длина кода в битах. UTF-16 содержит 65536 кодовых комбинаций. Указанного количества кодов достаточно для хранения символов всех существующих естественных языков, которые используются в настоящее время. Так же используется таблица кодов ASCII, которая содержит 256 символов. Кодовое слово в таблице ASCII имеет длину 8 бит. В таблице 1 представлена структура таблицы ASCII.

Таблица 1

Структура таблицы ASCII

Порядковый номер	Коды	Назначение символов
0-31	00000000-00011111	Управляющие символы
32-127	00100000-01111111	Стандартная часть таблицы
128-255	10000000-11111111	Альтернативная часть таблицы

Первые 32 символа таблицы используются для управления процессом вывода текста на экран или на печать и разметки текстового документа. Символы с 32 по 127 содержат строчные и прописные буквы латинского алфавита, десятичные цифры, знаки препинания и скобки. Блок символов с номерами 128-255, является расширением стандартной ASCII таблицы, и служит для размещения символов национальных алфавитов. Таким образом для каждой пары национальный язык-английский язык существует уникальная таблица ASCII кодов. С развитием цифровых средств передачи информации (глобальные сеть) указанный подход стал неудобным, так как требовалось хранить несколько ASCII таблиц для корректного отображения символов национальных алфавитов. После перехода на таблицу кодов UTF-16 указанная проблема была решена. UTF-16 и ASCII совместимы. Первые 127 символов UTF-16 повторяют коды таблицы ASCII.

Кодирование целых чисел

Для представления целых чисел в памяти ЭВМ используется дополнительный код. Ёмкость одного машинного слова зависит от количества бит, которое слово занимает в памяти ЭВМ. В языках программирования объем памяти который занимает одна переменная называется типом данных, который характеризует объем занимаемый переменной в оперативной памяти. Типы данных, используемые для хранения целых чисел, делят по количеству байт, которые занимает переменная данного типа в памяти. В таблице 2 представлены типы данных для хранения целых чисел используемые в языке программирования C#.

Переменные принадлежащие типам Byte и SByte имеют одинаковый размер, но имеют разный числовой диапазон. В типе Byte все 8 бит являются весовыми в типе SByte старший бит (7 бит) является знаковым. Если число положительное то в указанном бите записан «0», если число отрицательное то «1». Для записи целых неотрицательных чисел используется прямой код (типы Byte, UShort, UInt, ULong), для записи целых чисел со знаком используется дополнительный код.

Таблица 2

Типы данных для хранения целых чисел в C#

Наименование	Диапазон	Размер
Byte	0..255	1 байт
SByte	-127..128	1 байт
Short	-32768..32767	2 байта
UShort	0..65535	2 байта
Int	-2147483648..2147483647	4 байта
UInt	0..4294967295	4 байта
Long	-9223327036854775808…9223327036854775807	8 байта
ULong	0..18446744073709551615	8 байта

Прямой код числа совпадает с его двоичным представлением. Если требуется прямой код определенного типа, тогда к двоичному представлению числа слева добавляют такое количество нулей, которое требует тип данных. Число 25_{10} имеет двоичное представление 11001_2. Если требуется прямой код типа Byte, тогда двоичная запись числа 25_{10} примет следующий вид 00011001_2 (цифры в нижнем индексе обозначают основание системы исчисления, в которой записано число), если UShort, тогда 0000000000011001_2. Для более компактной записи используется шестнадцатеричный код. Число 25_{10} можно записать как 19_{16}

Дополнительный код положительного числа совпадает с прямым кодом. Для получения дополнительного кода отрицательного числа необходимо:

1. Перевести модуль числа в прямой код;
2. Инвертировать биты числа (преобразование в обратный код);
3. Прибавить к младшему разряду получившегося числа «1».

Пример. Перевести число -145_{10} в дополнительный код. Переменная, содержащая число имеет тип Short.

1. Получаем прямой код модуля числа $-145_{10}=0000\ 0000\ 1001\ 0001_2$;
2. Инвертируем число $\overline{0000000010010001_2}=1111\ 1111\ 0110\ 1110_2$;
3. Прибавить к младшему разряду «1»

   $1111\ 1111\ 0110\ 1110_2+0000\ 0000\ 0000\ 0001_2=1111\ 1111\ 0110\ 1111_2$ или $FF6F_{16}$

Правила перевода из дополнительного кода:

1. Если число положительное (знаковый разряд равен «0»), то выполняется перевод числа из двоичной в десятичную систему.
2. Если число отрицательное то:

   2.1. Биты числа записанного в дополнительном коде инвертируются;

   2.2. К полученному числу прибавляется «1».

Переведем число $1111\ 1111\ 0110\ 1111_2$ из дополнительного кода

Знаковый бит равен «1» значит число отрицательное. Необходимо инвертировать биты числа $1111\ 1111\ 0110\ 1111_2 \rightarrow 0000\ 000\ 1001\ 0000_2$. Прибавляем к полученному числу единицу и получаем модуль отрицательного числа:

$0000\ 000\ 1001\ 0000_2+0000\ 0000\ 0000\ 0001_2=0000\ 0000\ 1001\ 0001_2=145_{10}$

Еще одной формой представления чисел используемой в ЭВМ является двоично-десятичный код (ДДК). В ДДК для хранения одной десятичной цифры используется 4 бита.

Необходимо перевести число 235_{10} в ДДК.

Для выполнения указанной операции необходимо определить двоичное представление цифр числа 235_{10}:

- $2_{10} \rightarrow 0010_2$;
- $3_{10} \rightarrow 0100_2$;
- $5_{10} \rightarrow 0101_2$;

ДДК числа $235_{10} \rightarrow$ 0010 0100 0101.

Кодирование действительных чисел

Для представления действительных чисел (чисел с плавающей запятой) в двоичном коде используется стандарт IEEE 754. Указанный стандарт регламентирует:

- Представление нормализованных положительных и отрицательных действительных чисел;
- Представление денормализованных положительных и отрицательных действительных чисел;
- Представление нулевых чисел;
- Округление.

Стандарт определяет 4 формата представления чисел с плавающей точкой:

- С одинарной точностью (single-precision). Размер 32 бита.
- С двойной точностью (double-precision). Размер 64 бита.
- С одинарной расширенной точностью (single-extended precision). Размер более 43 бит.
- С двойной расширенной точностью (double-extended precision) Размер более 79 бит.

Согласно стандарту IEEE 754 действительные числа имеют следующую запись:

$$N = M \cdot exp_n^p,$$

где: N – действительное число; M – мантисса; p – порядок (число всегда целое), n – основание системы исчисления; exp_n^p– экспонента (n^p). Такое представление называют экспоненциальной записью числа.

Нормализованным считается действительное число, если мантисса находится в диапазоне (для десятичной системы исчисления):

$1 \leq M < 10$ – для десятичной системы исчисления;

$1 \leq M < 2$ – для двоичной системы исчисления.

Денормализованным считается действительное число, если мантисса находится в диапазоне:

$$0{,}1 \leq M < 1.$$

Нормализованный вид числа 345,453 имеет вид $3{,}45453 \cdot 10^3$, денормализованный вид того же числа $0{,}345453 \cdot 10^4$.

В двоичном коде десятичные действительные числа представляют в экспоненциальном нормализованном виде. Для получения двоичного числа представленного в экспоненциальном нормализованном виде применяется следующий алгоритм:

1. Необходимо получить прямой код целой и дробной части числа. Перевод десятичного числа в прямой двоичный код осуществляется по рекурсивному алгоритму:
   1.1. Перевод целой части десятичного числа;
   1.2. Дробная часть числа умножается на 2;
   1.3. В полученном произведении выделяется целая часть, которая принимается в качестве первого после запятой разряда числа.

1.4. Если дробная часть полученного произведения равна «0» или достигнута требуемая точность алгоритм завершается, в противном случае выполняется пункт 1.3.

2. Полученное число в двоичном коде, приводится к нормализованному виду.

Представим число 345,625 как двоичное число в экспоненциальном нормализованном виде.

$345_{10} \rightarrow 101011001_2$;

0,625·2=**1**,25

0,25·2=**0**,5

0,5·2=**1**

$345{,}625_{10} \rightarrow 101011001{,}101_2$

$3{,}45625exp_{10}^{2} \rightarrow 1{,}01011001101exp_{2}^{1000}$

В языке программирования C# используются 2 типа данных для операций над действительных чисел (Таблица 3).

Таблица 3

Действительные типы данных C#

Наименование	Диапазон	Размер
Float	$-3{,}402\,823 \cdot 10^{38} – 3{,}402\,823 \cdot 10^{38}$	4 байта
Double	$-1{,}797\,693\,134\,862\,32 \cdot 10^{308} – 1{,}797\,693\,134\,862\,32 \cdot 10^{308}$	8 байт

Типы Float и Double языка C# подчиняются стандарту IEEE 754

На рисунке 1 представлена структура формата одинарной точности стандарта IEEE 754 для хранения действительных чисел в памяти ЭВМ.

Рис. 1. Структура записи числа single-precision IEEE 754

S – бит знака, 1 бит; E – смещенная экспонента, 8 бит; M – остаток мантиссы, 23 бита

Знаковый бит S=0, если число положительное, и S=1, если число отрицательное.

Экспонента E может быть, как отрицательной так и положительной. Для определения знака экспоненты, чтобы не использовать знаковый бит, применяют смещение на величину 127_{10} ($0111\ 1111_2$).

Биты с 0 – 22 отводят для записи мантиссы. У нормализованной мантиссы в двоичном коде первый бит всегда равен «1», поэтому в отведенные биты записывают остаток от мантиссы.

Число $345{,}625_{10}$ в двоичном экспоненциальном нормализованном виде записывается как $1{,}01011001101exp_{2}^{1000}$. Представим указанное число в формате single-precision IEEE 754 (тип данных Float C#).

Число положительное – S=0;

Значение смещенной экспоненты $E=0111\ 1111_2+1000_2=1000\ 0111_2$;

Значение остатка мантиссы M=0101 1001 1010 0000 0000 0000 000

Двоичная запись числа $345{,}625_{10}$ в формате single-precision IEEE 754 имеет вид:

$0100\ 0111\ 1010\ 1100\ 1101\ 0000\ 0000\ 0000\ 0000_2$ или $47ACD0000_{16}$.

Преобразование двоичной записи формата single-precision IEEE 754 в действительное число D выполняется по указанному алгоритму в обратной последовательности или по формуле:

$$D = (-1)^S \cdot 2^{(E-127)} \cdot (1 + M / 2^{23}).$$

Преобразуем двоичную запись 0100 0111 1010 1100 1101 0000 0000 0000 0000 формата single-precision IEEE 754 в действительное число.

S=0;

E=1000 0111_2=135_{10};

M=0101 1001 1010 0000 0000 0000 000_2=0101 1001 101_2=2936832_{10} (при преобразовании номер старшего разряда мантиссы равен 22)

$D = (-1)^0 \cdot 2^{(135-127)} \cdot (1 + 2936832 / 2^{23}) = 345{,}625_{10}.$

Структура записи чисел с одинарной точностью и двойной точностью стандарта IEEE 754 одинакова. Различия заключаются в количестве бит отведенных для записи мантиссы и экспоненты. Для формата double-precision IEEE 754 для записи остатка мантиссы отводится 52 бита (0-51), для записи экспонаты 11 бит (52-62). Бит №63 является знаковым.

Формула для расчета экспоненты двоичного нормализованного числа с плавающей точкой в общем виде:

$$\exp_2 = E - (2^{(b-1)} - 1),$$

где: b – количество бит отведенное для записи смещенной экспоненты.

Десятичное действительное число с плавающей точкой, из двоичных записей формата IEEE 754 определяется по выражению

$$D = (-1)^S \cdot 2^{(E-2^{(b-1)}+1)} \cdot (1 + M / 2^n),$$

где: n – количество бит отведенное для записи остатка мантиссы.

Задания для самостоятельного выполнения

1. Выполнить преобразование десятичных чисел в ДДК.
2. Выполнить преобразование из ДДК в десятичные числа.
3. Представить текст как набор кодов таблицы ASCII.
4. Представить набор кодов таблицы ASCII как текст.
5. Представить десятичное число как переменную типа Byte.
6. Представить десятичное число как переменную типа SByte.
7. Представить десятичное число как переменную типа UShort
8. Представить десятичное число как переменную типа Short
9. Выполнить преобразование дополнительного двоичного кода в десятичный код.
10. Представить действительное число как переменную типа Float.
11. Представить двоичную запись double-precision IEEE 754 как действительное число.

Вариант 1

1) 844_{10}; 717_{10}; 939_{10};
2) $1000\ 0100\ 0011_2$; $1001\ 0110\ 0110_2$
3) EBVsFz2SBv
4) $72\ 44\ 33\ 48\ 38\ 79\ 4F\ 60_{16}$
5) 51_{10}; 243_{10}; 72_{10};
6) 40_{10}; -54_{10}; -48_{10};
7) 60290_{10}; 33422_{10};
8) 11081_{10}; -16586_{10};
9) 00110111101110102; 10011001110010012;
10) $-892{,}606_{10}$; $-689{,}451_{10}$
11)$4176D1B570000000_{16}$; $C146084300000000_{16}$

Вариант 2

1) 554_{10}; 970_{10}; 710_{10};
2) $0100\ 0001\ 0110_2$; $0111\ 1001\ 1000_2$
3) 4zIqkkoqyR
4) $6A\ 2E\ 4E\ 2D\ 2A\ 41\ 51\ 4B_{16}$
5) 204_{10}; 217_{10}; 13_{10};
6) 22_{10}; -57_{10}; -90_{10};
7) 52878_{10}; 35890_{10};
8) 13889_{10}; -15910_{10};
9) 00111110000100002; 10110101010011102;
10) $-644{,}833_{10}$; $-959{,}689_{10}$
11)$417A79C840000000_{16}$; $C139857A00000000_{16}$

Вариант 3

1) 397_{10}; 186_{10}; 631_{10};
2) $0000\ 0101\ 1001_2$; $0100\ 0111\ 0101_2$
3) s16GyQDCvw
4) $3F\ 35\ 51\ 64\ 3D\ 75\ 79\ 6E_{16}$
5) 50_{10}; 173_{10}; 7_{10};
6) 192_{10}; -114_{10}; -64_{10};
7) 49708_{10}; 12272_{10};
8) 29028_{10}; -20782_{10};
9) 01001010001101002; 10100000011011012;
10) $-815{,}871_{10}$; $-813{,}821_{10}$
11)$417557CBA0000000_{16}$; $C139CCE800000000_{16}$

Вариант 4

1) 253_{10}; 935_{10}; 188_{10};
2) $0101\ 0101\ 0100_2$; $0100\ 0011\ 0001_2$
3) w3sQF10iIH
4) $44\ 2A\ 58\ 2E\ 57\ 25\ 38\ 4E_{16}$
5) 248_{10}; 57_{10}; 228_{10};
6) 46_{10}; -124_{10}; -52_{10};
7) 30800_{10}; 23257_{10};
8) 31395_{10}; -22977_{10};
9) 01010100100010102; 11010000010010002;
10) $-714{,}228_{10}$; $-548{,}987_{10}$
11)$4169DC9E20000000_{16}$; $C13E4C7800000000_{16}$

Вариант 5

1) 499_{10}; 346_{10}; 885_{10};
2) $0010\ 1010\ 0001_2$; $0100\ 0100\ 0101_2$
3) 5eMsHKk15f
4) $3F\ 64\ 40\ 7C\ 3A\ 41\ 55\ 6E_{16}$
5) 228_{10}; 144_{10}; 249_{10};
6) 72_{10}; -91_{10}; -36_{10};
7) 32571_{10}; 13932_{10};
8) 31825_{10}; -27717_{10};
9) 01000010010010012; 10111011100100002;
10) $-780{,}52_{10}$; $-847{,}243_{10}$
11)$4168929D60000000_{16}$; $C1457A3780000000_{16}$

Вариант 6

1) 776_{10}; 346_{10}; 706_{10};
2) $0010\ 0101\ 0011_2$; $0110\ 0101\ 0010_2$
3) F9AkbUcfyJ4
4) $26\ 2A\ 42\ 26\ 6F\ 4A\ 7D\ 70_{16}$
5) 116_{10}; 248_{10}; 136_{10};
6) 44_{10}; -34_{10}; -6_{10};
7) 43043_{10}; 23620_{10};
8) 18316_{10}; -30747_{10};
9) 00111001001101002; 11010110100010112;
10) $-956{,}859_{10}$; $-917{,}409_{10}$
11)$41651AAC00000000_{16}$; $C13BC866000000000_{16}$

Вариант 7

1) 331_{10}; 181_{10}; 128_{10};
2) $1001\ 0111\ 0101_2$; $0100\ 0001\ 1000_2$
3) ZalUJPfZZw
4) $6E\ 50\ 52\ 3D\ 6D\ 23\ 5E\ 73_{16}$
5) 51_{10}; 182_{10}; 155_{10};
6) 242_{10}; -50_{10}; -73_{10};
7) 24011_{10}; 14344_{10};
8) 10698_{10}; -30220_{10};
9) 00111010110000002; 10110111000101002;
10) $-511{,}315_{10}$; $-771{,}68_{10}$
11)$417A860CE0000000_{16}$; $C13E00D300000000_{16}$

Вариант 8

1) 390_{10}; 811_{10}; 368_{10};
2) $0101\ 1000\ 0110_2$; $1000\ 0000\ 0010_2$
3) ze8HqgRN4l
4) $68\ 59\ 7D\ 42\ 3A\ 76\ 77\ 3E_{16}$
5) 108_{10}; 73_{10}; 81_{10};
6) 145_{10}; -82_{10}; -53_{10};
7) 43837_{10}; 33279_{10};
8) 15646_{10}; -31623_{10};
9) 01001010111111012; 11000111101110102;
10) $-958{,}082_{10}$; $-715{,}131_{10}$
11)$4174CDB390000000_{16}$; $C141051100000000_{16}$

Вариант 9

1) 312_{10}; 532_{10}; 329_{10};
2) $0001\ 0001\ 0011_2$; $0001\ 0000\ 0101_2$
3) Rmi6R4eR8g
4) $53\ 7D\ 7D\ 56\ 63\ 6A\ 43\ 31_{16}$
5) 51_{10}; 140_{10}; 45_{10};
6) 223_{10}; -74_{10}; -106_{10};
7) 51221_{10}; 16070_{10};
8) 13867_{10}; -13199_{10};
9) 01100100111001012; 10110110101111012;
10) $-838{,}974_{10}$; $-751{,}227_{10}$
11)$4176D24830000000_{16}$; $C13803FB00000000_{16}$

Вариант 10

1) 407_{10}; 140_{10}; 534_{10};
2) $0111\ 0101\ 1000_2$; $0001\ 0010\ 0111_2$
3) 1P8lFF8GNx
4) $45\ 7B\ 56\ 50\ 73\ 43\ 3F\ 45_{16}$
5) 253_{10}; 42_{10}; 122_{10};
6) 216_{10}; -108_{10}; -76_{10};
7) 46681_{10}; 15402_{10};
8) 24160_{10}; -23353_{10};
9) 00100111101000102; 10100100110011102;
10) $-756{,}869_{10}$; $-731{,}49_{10}$
11)$4174F81B90000000_{16}$; $C13CEC3D00000000_{16}$

Вариант 11

1) 285_{10}; 878_{10}; 630_{10};
2) $0101\ 0100\ 0001_2$; $0111\ 1001\ 0101_2$
3) hB20TMl5Gc
4) $4F\ 71\ 61\ 26\ 6D\ 5C\ 7A\ 62_{16}$
5) 190_{10}; 177_{10}; 110_{10};
6) 170_{10}; -54_{10}; -48_{10};
7) 18841_{10}; 47459_{10};
8) 18832_{10}; -16309_{10};
9) 00101101000010002; 10101011110101012;
10) $-676{,}736_{10}$; $-702{,}417_{10}$
11)$4168A5A5C0000000_{16}$; $C142AAB500000000_{16}$

Вариант 12

1) 779_{10}; 935_{10}; 398_{10};
2) $0001\ 1000\ 0100_2$; $0101\ 0101\ 0010_2$
3) E7JGMJzEqO
4) $72\ 3A\ 24\ 56\ 33\ 24\ 7F\ 36_{16}$
5) 134_{10}; 197_{10}; 233_{10};
6) 76_{10}; -78_{10}; -113_{10};
7) 37471_{10}; 34179_{10};
8) 29151_{10}; -30885_{10};
9) 00111100111011002; 11000101110001012;
10) $-634{,}866_{10}$; $-527{,}797_{10}$
11)$417B501EC0000000_{16}$; $C1302B2400000000_{16}$

Вариант 13

1) 589_{10}; 173_{10}; 671_{10};
2) $0011\ 0001\ 0110_2$; $0010\ 1001\ 0001_2$
3) pebKWJiZrQ
4) $2E\ 40\ 7C\ 4C\ 4A\ 45\ 2B\ 5A_{16}$
5) 219_{10}; 229_{10}; 98_{10};
6) 46_{10}; -67_{10}; -28_{10};
7) 58921_{10}; 34681_{10};
8) 21529_{10}; -12275_{10};
9) 01010011001011102; 10001101001000112;
10) $-621{,}923_{10}$; $-989{,}539_{10}$
11)$4172643BE0000000_{16}$; $C1341A8600000000_{16}$

Вариант 14

1) 469_{10}; 964_{10}; 203_{10};
2) $0100\ 0011\ 1001_2$; $1000\ 0011\ 0111_2$
3) RJisugjRkb
4) $2B\ 72\ 7E\ 5A\ 7C\ 7A\ 52\ 29_{16}$
5) 199_{10}; 194_{10}; 132_{10};
6) 191_{10}; -117_{10}; -12_{10};
7) 16831_{10}; 46527_{10};
8) 28655_{10}; -26819_{10};
9) 01010010011010112; 10001101010010102;
10) $-530{,}458_{10}$; $-695{,}146_{10}$
11)$41790B9160000000_{16}$; $C138207000000000_{16}$

Вариант 15

1) 931_{10}; 658_{10}; 413_{10};
2) $0011\ 0001\ 0000_2$; $0011\ 1000\ 0011_2$
3) WGZnSWhbz7
4) $20\ 43\ 67\ 36\ 68\ 77\ 76\ 5D_{16}$
5) 237_{10}; 8210; 212_{10};
6) 114_{10}; -84_{10}; -76_{10};
7) 26550_{10}; 50606_{10};
8) 12691_{10}; -31815_{10};
9) 01101100011111112; 10100111100111102;
10) $-682{,}498_{10}$; $-744{,}947_{10}$
11)4178410950000000_{16}; $C133460A00000000_{16}$

Вариант 16

1) 234_{10}; 532_{10}; 297_{10};
2) $0010\ 0101\ 0011_2$; $0011\ 0000\ 0101_2$
3) ng7AKhmibB
4) $36\ 5C\ 23\ 78\ 39\ 24\ 4A\ 2F_{16}$
5) 145_{10}; 37_{10}; 170_{10};
6) 171_{10}; -34_{10}; -39_{10};
7) 32543_{10}; 25941_{10};
8) 31684_{10}; -18459_{10};
9) 00110110000011112; 11000111011001102;
10) $-577{,}832_{10}$; $-737{,}23_{10}$
11)$417C777870000000_{16}$; $C13F630000000000_{16}$

Вариант 17

1) 994_{10}; 217_{10}; 126_{10};
2) $0010\ 1001\ 0110_2$; $1000\ 1001\ 0011_2$
3) yxT9WItkFF
4) $6A\ 72\ 5A\ 56\ 63\ 2A\ 5E\ 41_{16}$
5) 144_{10}; 252_{10}; 46_{10};
6) 111_{10}; -21_{10}; -34_{10};
7) 11223_{10}; 15772_{10};
8) 20786_{10}; -17546_{10};
9) 01011011100011012; 10101101000110012;
10) $-628{,}634_{10}$; $-814{,}376_{10}$
11)$41650F59A0000000_{16}$; $C137B47C00000000_{16}$

Вариант 18

1) 411_{10}; 593_{10}; 931_{10};
2) $0101\ 0001\ 0110_2$; $0100\ 1010\ 0001_2$
3) g4IN8H6vYb
4) $5A\ 43\ 36\ 60\ 24\ 43\ 5E\ 64_{16}$
5) 77_{10}; 75_{10}; 34_{10};
6) 187_{10}; -92_{10}; -101_{10};
7) 38759_{10}; 32124_{10};
8) 16619_{10}; -28122_{10};
9) 00101101011001012; 11010011101001112;
10) $-771{,}035_{10}$; $-578{,}151_{10}$
11)$4166560AC0000000_{16}$; $C1457ACD00000000_{16}$

Вариант 19

1) 584_{10}; 466_{10}; 863_{10};
2) $1001\ 0110\ 0011_2$; $0001\ 0101\ 0010_2$
3) hKQrfLZgFF
4) $28\ 7F\ 76\ 2E\ 52\ 79\ 41\ 51_{16}$
5) 247_{10}; 49_{10}; 62_{10};
6) 83_{10}; -49_{10}; -47_{10};
7) 48272_{10}; 25464_{10};
8) 19016_{10}; -22610_{10};
9) 01011001000110012; 10110011101001002;
10) $-969{,}273_{10}$; $-827{,}25_{10}$
11)$416D520D00000000_{16}$; $C135F51100000000_{16}$

Вариант 20

1) 844_{10}; 705_{10}; 750_{10};
2) $1010\ 0001\ 0000_2$; $0011\ 0101\ 1001_2$
3) jLHM6UigsA
4) $62\ 25\ 22\ 77\ 71\ 37\ 6A\ 52_{16}$
5) 195_{10}; 254_{10}; 184_{10};
6) 64_{10}; -118_{10}; -51_{10};
7) 29193_{10}; 61358_{10};
8) 30335_{10}; -13534_{10};
9) 00111101010011112; 11000111111110002;
10) $-753{,}044_{10}$; $-695{,}236_{10}$
11)$4174E2DE40000000_{16}$; $C140284280000000_{16}$

Вариант 21

1) 997_{10}; 406_{10}; 546_{10};
2) $0101\ 0100\ 1000_2$; $1000\ 0111\ 0111_2$
3) HEAfnfqKWw
4) $6A\ 6C\ 73\ 2B\ 59\ 20\ 4F\ 31_{16}$
5) 62_{10}; 179_{10}; 138_{10};
6) 114_{10}; -64_{10}; -29_{10};
7) 24710_{10}; 20983_{10};
8) 20608_{10}; -10092_{10};
9) 01100010100110002; 11010010010101002;
10) $-553{,}688_{10}$; $-891{,}998_{10}$
11)$41717A4290000000_{16}$; $C1402F6480000000_{16}$

Вариант 22

1) 472_{10}; 726_{10}; 261_{10};
2) $1010\ 0011\ 0101_2$; $0100\ 0111\ 0010_2$
3) SQcHVe6pyU
4) $48\ 4B\ 66\ 20\ 63\ 4D\ 56\ 30_{16}$
5) 182_{10}; 119_{10}; 140_{10};
6) 8_{10}; -126_{10}; -77_{10};
7) 42842_{10}; 43101_{10};
8) 16804_{10}; -17858_{10};
9) 01101100100010002; 10100101010100112;
10) $-729{,}82_{10}$; $-876{,}844_{10}$
11)$416CBEB2C0000000_{16}$; $C1337C0F00000000_{16}$

Вариант 23

1) 713_{10}; 898_{10}; 433_{10};
2) $0100\ 0101\ 1000_2$; $0101\ 0100\ 0101_2$
3) YKKBJqVTjV
4) $33\ 58\ 40\ 2C\ 4A\ 36\ 5D\ 4E_{16}$
5) 122_{10}; 214_{10}; 28_{10};
6) 204_{10}; -22_{10}; -99_{10};
7) 39606_{10}; 55743_{10};
8) 29935_{10}; -22589_{10};
9) 01110010111100112; 11010100101111102;
10) $-798{,}047_{10}$; $-916{,}365_{10}$
11)$417851A780000000_{16}$; $C141AF5580000000_{16}$

Вариант 24

1) 305_{10}; 657_{10}; 541_{10};
2) $0010\ 0110\ 0101_2$; $0111\ 1001\ 0100_2$
3) ohv00siE5E
4) $3F\ 32\ 60\ 66\ 76\ 35\ 29\ 3B_{16}$
5) 188_{10}; 19_{10}; 31_{10};
6) 39_{10}; -107_{10}; -16_{10};
7) 44717_{10}; 31031_{10};
8) 26630_{10}; -23475_{10};
9) 00100111100100102; 10011101000100012;
10) $-509{,}379_{10}$; $-605{,}184_{10}$
11)$41690C33E0000000_{16}$; $C14371CC00000000_{16}$

ИЗМЕРЕНИЕ КОЛИЧЕСТВА И ОБЪЕМА ИНФОРМАЦИИ

При измерении количества и объема информации используется единица измерения – бит.

Впервые термин бит (bit) использован К. Шенноном в статье «Математическая теория связи» (1948г.), как сокращение от binary digit.

Бит как единица измерения информации используется в математике, цифровой технике и теории информации.

В вычислительной технике в битах определяется емкость запоминающих устройств (ЗУ) и объем передаваемой по каналам связи информации. В теории информации в битах измеряется количество информации содержащееся в некотором объеме данных. В литературе подход к измерению информации используемый в вычислительной технике называют структурным или алфавитным. Подход используемый в теории информации называют вероятностным.

С развитием вычислительной техники и средств телекоммуникаций возрос объем хранимой и передаваемой по каналам связи объем информации. В 1956 году В. Бухгольцем при разработке суперкомпьютера IBM 7030 предложен термин «байт» для обозначения 6-ти битного массива данных. В дальнейшем термин укоренился но мог обозначать разное количество бит. Так в советской ЭВМ Минск-32 байт состоял из семи бит. Привычное соотношение 1 байт=8 бит стало стандартом в начале 70-х годов XX века.

Для обозначения объема информации кратного байту или биту используются приставки (Кбайт, Мбайт). Приставки различают на двоичные и десятичные.

При использовании двоичной приставки основанием является 2 в степени n, где n=10, 20, 30 и т.д. При использовании десятичной приставки основанием является степень 10. В таблице 4 представлены значения величин с десятичными и двоичными приставками.

Таблица 4

Соотношение величин

Приставка	Двоичная приставка	Десятичная приставка	Разность	Ошибка, %
Кило	2^{10}=1024	10^3=1000	24	2,34
Мега	2^{30}=1048576	10^6=1000000	48576	4,63
Гига	2^{40}=1073741824	10^9=1000000000	73741824	6,87
Тера	2^{50}=1099511627776	10^{12}=1000000000000	99511627776	9,05

При увеличении размерности существенно возрастает разность между величинами.

В настоящее время используются как двоичный так и десятичный подходы к интерпретации приставки.

Двоичный подход при обозначении количества информации применяется:

- В большинстве современных операционных систем для обозначения размера файлов и дискового пространства;
- Производителями оперативной и видеопамяти.

Десятичный подход при обозначении количества информации применяется:

- Производителями накопителей на жестких магнитных дисках;
- Производителями твердотельных накопителей (USB Flash, SSD).

Измерение объема информации

Текстовые данные

При кодировании данных, как правило, используется равномерный код. Равномерное кодирование это двоичное кодирование, при котором каждому знаку первичного алфавита ставиться в соответствие двоичное слово одинаковой длины (Таблица ASCII, UNICODE).

В общем случае при длина двоичного слова L определяется по выражению

$$L \geq \log_2 n,$$

где: n – количество знаков первичного алфавита.

Для кодирования символа текста записанного на русском языке с учетом трех знаков препинания (пробел, запятая, точка) необходимо 5,16 бита. Так как записать в ЗУ возможно только целое количество бит, результат округляют в большую сторону (6 бит).

Наибольшее распространение получило 8-битное кодирование (на кодирование одного символа отводится 8 бит=1 байт), позволяющее закодировать $N=2^8=256$ различных символов.

Самыми распространенными являются 8-битные таблицы кодировок ASCII, Windows-1251 (CP1251), КОИ-8, ISO и 16-битная таблица UNICODE.

Объем текстового сообщения I определяется по выражению

$$I = K \cdot i,$$

где: K – количество символов в тексте, i – количество байт на символ (например, если используется ASCII i=1).

В семействе операционных систем Windows для хранения текстовой информации наиболее часто используются контейнеры TXT и RTF (DOC и DOCX позволяют хранить в одном файле текстовую и графическую информацию).

Контейнер TXT не имеет заголовка и циклических блоков. Данные находящиеся в контейнере интерпретируются как байт массив содержащий значения ASCII символов.

Данный формат был определен фирмой Microsoft как стандартный формат для обмена текстовыми документами. В RTF для обмена документами используются только представимые символами коды из ASCII-, MAC- и PC-символьного набора. Кроме текста, файл в RT-формате в читаемой форме содержит команды управления. Документ состоит преимущественно из команд управления настройки программы чтения файлов в RTF-формате. Эти команды можно разделить на управляющие слова (control words) и управляющие символы (control symbols).

Управляющее слово представляет собой последовательность символов с разделителем в конце. Перед управляющим словом вводится обратная косая черта "\". В RT-формате для задания управляющей последовательности используются буквы от "A" до "Z" и от "a" до "z", а также цифры от "0" до "9". Национальные символы к управляющей информации не относятся.

В RT-формате существует возможность объединять отдельные последовательности в группы при помощи скобок. Такие группы создаются, например, при описании сносок, колонтитулов и т.п.

В RT-формате используются также некоторые символьные коды для управления печатью: «09H» – Табулятор , «0AH» – Символ CR, «0CH» – Символ LF. Символы CR и LF, расположенные внутри текста, будут пропущены. Microsoft использует эти символы для большей наглядности при представлении RTF-файла.

Изображения

Объем памяти занимаемый изображением зависит от размеров изображения (разрешения), глубины цвета, применения алгоритмов сжатия.

Глубина цвета это количество бит, используемых для кодирования цвета одной точки. От глубины цвета *k* зависит количество отображаемых цветов *N*:

$$N = 2^k .$$

Разрешение – количество точек (пикселей) изображения, приходящихся на единицу длины. Наиболее часто используемые экранные разрешения: 640x480, 800x600, 1024x768, 1280x1024 и т.д.

В общем случае объем памяти *I*, который занимает изображение в ЗУ, определяется по выражению

$$I = W \cdot H \cdot k ,$$

где: *W* – размер изображения в пикселях по горизонтали, H – размер изображения в пикселях по вертикали, k – глубина цвета в битах.

Все графические форматы делятся на два класса: растровые и векторные. Растровые графические файлы содержат описание каждой точки изображения. Они представляют собой прямоугольную матрицу (bitmap). Каждый элемент матрицы содержит значение цвета.

Файлы векторных форматов содержат математические формулы, описывающие координаты кривых. Например, прямая линия представлена координатами двух точек, а окружность — координатами центра и радиуса, поэтому достигается очень небольшой размер файла.

BMP (Windows Device Independent Bitmap) является один из старейших форматов. Основная область применения BMP — хранение файлов, использующихся внутри операционной системы (например, обоев для Рабочего стола или иконок программ). Файлы, сохраненные в BMP, могут содержать как 256 цветов, так и 16 700 000 оттенков. BMP - самый простой (если не сказать примитивный) графический формат.

Записанный в нем файл представляет собой массив данных, содержащий информацию о цвете каждого пиксела, т. е. изображение размером 1024x768 точек с глубиной цвета 24 бита (3 байта) будет занимать 1024x768x3= 2 359 296 байт (без учета служебной информации об объеме и имени файла, составляющей еще несколько сотен байт). Основной недостаток формата, ограничивающий его применение, — большой размер BMP-файлов.

Как и BMP, GIF (CompuServe Graphics Interchange Format) является одним из старейших форматов. В1977 г. израильские математики А. Лемпел и Я. Зив разработали новый класс алгоритмов сжатия без потерь, получивший название метод Лемпела—Зива. Одновременно с самим алгоритмом, именуемымLZ77 (по первым буквам фамилий ученых и последним цифрам года создания), свет увидела статья, где излагались общие идеи данного метода. Впоследствии появилась усовершенствованная версия продукта— LZ78. Многие специалисты стремились доработать его; наибольшее распространение получил вариант LZW, написанный Т. Уэлчем, сотрудником фирмы Unisys, в1983 г. Он отличался от своего прародителя более высоким быстродействием. В1985 г. Unisys запатентовала LZW. Этот алгоритм универсален и может использоваться для сжатия информации любого вида. Однако успешнее всего он справляется с графическими изображениями.

В1987 г. Компания CompuServe разработала на основе алгоритма LZW графический формат GIF (Graphic Interchange Format), позволивший эффективно сжимать изображения с глубиной цвета до 8 бит, а по тем временам 256 оттенков считалось огромным количеством. Он был разработан для передачи растровых изображений по сетям.

В1989 г. CompuServe выпустила усовершенствованную версию формата, названную GIF89a. В нее были добавлены две функции, которые и по сей день обеспечивают формату популярность в Интернете (GIF позиционируется прежде всего как сетевой формат). Во-первых, был добавлен альфа-канал, где может храниться маска прозрачности, во-вторых, GIF стал анимированным, т. е. в один файл можно поместить несколько изображений, которые будут сменять друг друга через заданный интервал времени. Таким образом, GIF начал под-

держивать прозрачность и анимацию. Файлы GIF содержат информацию в сжатом виде, что позволяет заметно уменьшить их размер, особенно если в них есть большие пространства, закрашенные одним цветом. Можно назначить один или несколько цветов прозрачными. Помимо этого GIF может хранить сразу несколько изображений, которые будут отображаться последовательно одно за другим. Объединив эти возможности, можно получить минимультфильмы, носящие название «анимированные GIF».

Еще одна полезная особенность формата— чересстрочная развертка. Во время загрузки изображения сначала показываются первая строка, пятая, десятая и т.д., а затем— вторая, шестая, одиннадцатая. Таким образом создается эффект постепенного проявления картинки на экране. Это позволяет увидеть и оценить изображение еще до завершения загрузки, что особенно выгодно при работе с файлами внушительных размеров.

Область применения GIF — это изображения с резкими цветовыми переходами и бизнес-графика(логотипы, кнопки, элементы оформления и т. п.). А вот для тех картинок, где важно постепенное изменение оттенков(например, для фотографий), данный формат подходит меньше всего. Первая причина таких ограничений— небольшая по современным меркам максимальная глубина цвета изображения. Для фото 8 бит явно недостаточно. Вторая— не всегда корректное преобразование файлов с плавными цветовыми переливами в палитру 256 цветов и менее. Третья причина ограниченности сферы применения GIF заключается в особенностях метода сжатия информации. Данные об изображении записываются построчно. В итоге все операции происходят с массивом строк высотой в один пиксел (каждая строка обрабатывается отдельно). Следствие этого— не только крайняя неэффективность сжатия фотографий и любых других изображений, содержащих мало однотонных областей, но и зависимость его результата от расположения объектов. Главный недостаток GIF заключается в том, что изображение может содержать256 цветов. Это слишком мало для большинства современных задач.

Форматы на основе LZW не справлялись с эффективной обработкой фотографий, и потому появилась идея сжатия с потерей качества. Суть его заключается в том, что часть малозаметных для глаза деталей изображения опускается, а восстановленный после сжатия цифровой массив не полностью соответствует оригиналу. Таким образом, можно добиться довольно большой степени сжатия данных— в10—20 раз вместо двукратного, производимого без потерь.

В 1991 г. группа Joint Photographic Experts Group, опирающаяся на более чем полувековой опыт исследований в области человеческого зрения, представила первую спецификацию формата JPEG. Через три года она была признана индустриальным стандартом кодирования неподвижных изображений, зарегистрированным как ISO/IEC 10918-1. Впоследствии JPEG лег в основу стандарта сжатия видео MPEG.

JPEG (Joint Photographic Experts Group) на сегодняшний день является одним из наиболее популярных форматов. Всеобщего признания он достиг благодаря одноименному алгоритму сжатия, который показал отличные результаты в соотношении размер/качество.

JPEG сжимает файлы весьма оригинальным образом: он ищет не одинаковые пиксели, а вычисляет разницу между соседними квадратами размером 9x9 пикселей.

Информация, не заслуживающая внимания, отбрасывается, а ряд полученных значений усредняется. В результате получают файл в десятки или сотни раз меньшего размера, чем BMP. Естественно, чем выше уровень компрессии, тем ниже качество.

С помощью формата JPEG лучше всего хранить фотографии и другие похожие на них изображения. Скриншоты, схемы и чертежи лучше хранить в формате TIFF, поскольку при сохранении в JPEG неизбежно появятся помехи и «шумы».

Процесс обработки графической информации алгоритмом JPEG:

1. Исходное изображение делится на блоки размером 16x16 пикселов. Дальнейшие операции выполняются над каждым из них по отдельности, что дает существенный выигрыш в скорости по сравнению с тем случаем, когда картинка обрабатывается как единый массив.

2. Переход к более подходящему для сжатия способу представления цветов. Привычная модель RGB переводится в YCbCr, где Y — сигнал яркости, а Cb и Cr — насыщенность синего и красного соответственно. Такой способ представления цветов будет предпочтительнее и с точки зрения восприятия изображения человеческим глазом. Как известно, зрительная информация воспринимается с помощью сенсоров двух типов: палочек и колбочек. Первые анализируют яркостную составляющую изображения, вторые— цвет. Палочек в20 раз больше, чем колбочек, и, следовательно, глаз более восприимчив к изменению яркости, чем цвета.

Если учесть эту особенность человеческого зрения, то из матриц значений насыщенности синего и красного следует отбрасывать все четные строки и столбцы. Таким образом теряется 75% информации о распределении цветов. Матрица отчетов о яркости не изменяется, а делится на четыре части, образуя блоки 8x8.

3. Выполняется дискретное косинусное преобразование (Discrete Cosine Transform, DCT), предложенное В. Ченом в1981 г. Оно сходно с преобразованием Фурье, только у DCT несколько ниже вероятность возникновения ошибки. Применение этого чисто математического приема объясняется тем, что в реальных изображениях соседние блоки достаточно похожи (коэффициент корреляции— 0,9—0,98). DCT преобразует информацию о цвете и яркости каждого пиксела в информацию о скорости изменения этих величин. Это преобразование является обратимым, и, значит, по новой матрице может быть восстановлена исходная с точностью до погрешности данного метода. DCT позволяет значительно сократить объем данных и размер получаемого файла.

В результате DCT графическое изображение описывается двухмерной функцией, показывающей скорости изменения яркости и цвета. Причем для большинства фотографий характерны плавные, мягкие переходы этих параметров на соседних участках. Как выявили исследования, для корректного восприятия таких изображений глазу важнее низкочастотные компоненты матрицы DCT (плавные переходы), а высокочастотные (резкая смена оттенков и яркости) уже не столь существенны. Поэтому последние кодируются с меньшей детальностью, а при превышении определенной пороговой величины, зависящей от выбранного качества сжатия, вообще принимаются равными нулю. Следовательно, при кодировании файлов с низким качеством резкие цветовые переходы смазываются, а изображение становится несколько мозаичным.

4. Кодирование полученных величин методом Хаффмана— последний этап. Оно заключается в присваивании часто повторяющимся элементам обрабатываемой информации самых коротких кодовых последовательностей, а редко встречающимся— более длинных. Вследствие этого размер получаемого файла несколько уменьшается.

PNG (Portable Network Graphics) обязан своим появлением на свет формату GIF, а точнее, его коммерческому статусу. Дело в том, что GIF основан на запатентованном алгоритме LZW, принадлежащем фирме Unisys, которая в первые годы существования GIF не предъявляла никаких претензий фирме CompuServe, разработавшей этот формат. Но стремительное развитие Интернета в 1993—1994 гг. внесло свои коррективы во взаимоотношения компаний. И из корыстных целейUnisys инициировала судебный процесс против CompuServe. Решение суда обязывало разработчиков ПО, в котором используется формат GIF, платить фирме Unisys лицензионные отчисления. Таким образом, GIF стал платным.

Днем рождения PNG можно считать 4 января1995 г., когда Т. Боутелл предложил в ходе конференций Usenet создать свободный формат, который был бы не хуже GIF. И уже через три недели после публикации идеи были разработаны четыре версии нового формата. Вначале он имел название PBF (Portable Bitmap Format), а нынешнее имя получил 23 янва-

ря1995 г. Уже в декабре того же года спецификация PNG версии 0.92 была рассмотрена консорциумом W3C, а с выходом 1 октября 1996 г. версии 1.0 PNG был рекомендован в качестве полноправного сетевого формата. К моменту создания PNG глубина цвета 8 бит, предоставляемая GIF, перестала удовлетворять современным требованиям. Среди предложений довести глубину цвета до 24, 48 и даже до 64 бит был выбран первый вариант. Как показало время, данное решение оказалось рациональным и отвечающим современным запросам. В заголовке файла формата PNG хранится описание всей палитры. Подстроившись таким образом под реальное количество цветов, можно получить достаточно компактный файл на выходе. Формат PNG имеет массу достоинств, выгодно отличающих его от конкурентов. Самое главное из них— предварительная фильтрация обрабатываемых данных, поскольку большинство алгоритмов сжатия рассчитаны на одну вполне определенную модель информации. Например, формат JPEG лучше подходит для изображений с плавными цветовыми переходами, а GIF — с большим количеством однотонных областей. И чем больше структура картинки отличается от эталона, тем ниже эффективность сжатия. Порой изменение всего нескольких бит приводит к тому, что алгоритм начинает работать очень хорошо или, наоборот, очень плохо. Тем временем формат PNG стабильнее воспринимает такие трансформации, и высокий коэффициент сжатия достигается практически для любого рода изображений путем преобразования информации в вид, наиболее приемлемый для обработки.

Аудиозаписи

Объем памяти занимаемый аудио файлом зависит от частоты дискретизации, разрядности аналого-цифрового преобразователя (АЦП), длительности записи и количества каналов.

Частота дискретизации – характеристика АЦП выполняющего запись звука. Характеризует количество измерений входного сигнала в единицу времени.

Разрядность АЦП определяет точность измерения входного сигнала.

В общем виде объем памяти занимаемый аудиофайлом в ЗУ определяется выражением

$$I = f \cdot k \cdot N \cdot m,$$

где: f – частота дискретизации (Гц), k – разрядность АЦП (бит), N – длительность аудиозаписи (с), m – количество каналов.

Существует два основных стандарта MP3 и WMA. Если стандарт WMA разрабатывается исключительно фирмой Microsoft, то кодек для сжатия цифрового звука в стандарте MP3 может создать любой программист. В условиях конкурентной борьбы за качество MP3-файлов на первое место вышел проект нескольких программистов — Lame.

Кодек Lame применяется с 1998 года, когда с развитием Интернета между пользователями начался активный обмен файлами с данными и возникла проблема передачи звука по сети Ethernet.

Многие фирмы стали разрабатывать программы для кодирования аудиосигнала с наименьшими потерями в качестве. Группа специалистов (MPEG), входящая в состав Международной организации стандартов (ISO), еще в конце 80-х годов разработала стандарт MPEG, на основе которого был создан современный стандарт сжатия звука MPEG 1.0 Audio Layer III — впоследствии он стал известен как MP3. Так как изначально утилиты для создания MP3-файлов распространялись за деньги, программисты стали искать способ бесплатно использовать алгоритмы работы кодека. Так среди многих других проектов появился кодек Lame, постепенно получивший огромную популярность. Основной особенностью Lame стало то, что разработчики сделали акцент на улучшении алгоритма специальной психоакустической модели. Принцип ее действия заключается в том, что из звукового файла удаляются те частоты, которые человеческое ухо воспринимать не в состоянии.

При кодировании цифрового звука (файлы с расширением WAV) основным параметром, влияющим на качество результата, является величина битрейта. Если использовать мак-

симальное значение этого параметра, то получают звук, наиболее соответствующий оригиналу. Примерно до 2000 года широко использовалось значение битрейта 128 Кбит/с, а затем, с возросшей пропускной способностью современных каналов связи, наиболее распространенным стал битрейт 192 Кбит/с.

Еще один параметр, который может существенно улучшить качество звука, — функция VBR, позволяющая кодировать информацию с переменным битрейтом в зависимости от характера сигнала. Например, если в музыке присутствуют различные высокочастотные звуки (качественно сжать которые труднее всего), кодек принимает решение использовать для них самый высокий битрейт 320 Кбит/с.

Кодек WMA был разработан фирмой Microsoft как стандарт хранения сжатой аудиоинформации в операционной системе Windows. Microsoft не стала пользоваться разработками организации MPEG, поэтому стандарт WMA является закрытым для использования другими разработчиками.

Microsoft стремится максимально использовать коммерческий потенциал WMA и поэтому защищает музыку в этом формате от нелегального копирования: WMA-файлы нельзя преобразовать в файлы с расширением WAV.

В последней, девятой, версии кодека создатели значительно переработали психоакустическую модель кодирования аудиоинформации, однако на максимальных значениях битрейта кодек от Microsoft до сих пор не может сравниться по качеству с кодеком Lame.

Измерение количества информации

Количество информации, содержащееся в одном элементе сообщения, равно средней энтропии этого сообщения. Если все элементы сообщения имеют одинаковую вероятность появления для расчета энтропии используют формулу Хартли

$$H = \log_2 N, \tag{1}$$

где: H – энтропия (бит); N – количество элементов в сообщении.

Если элементы сообщения имеют разную вероятность появления для расчета энтропии используют формулу Шеннона

$$H = -\sum_{i=1}^{N} p_i \log_2(p_i), \tag{2}$$

где: p_i – вероятность появления i-го элемента.

В качестве примера определим количество информации, связанное с появлением каждого символа в сообщениях, записанных на русском языке. Будем считать, что русский алфавит состоит из 33 букв и трех знаков препинания для разделения слов. По формуле Хартли (1)

$$H = \log_2 36 \approx 5.16 \text{ бита.}$$

Вероятность наблюдения букв русского языка (как и любого другого естественного языка) не одинакова. Гласные «а», «о», «е», «и» в тексте встречаются чаще чем согласные «щ» и «ж». В таблице 5 представлены значения вероятности наблюдения букв русского языка.

Таблица 5

Вероятность наблюдения букв русского языка

Символ	Вероятность	Символ	Вероятность	Символ	Вероятность
«_»	0,175	Л	0,035	Б	0,014
О	0,09	К	0,028	Г	0,012
Е	0,072	М	0,026	Ч	0,012
Ё	0,072	Д	0,025	Й	0,010
А	0,062	П	0,023	Х	0,009
И	0,062	У	0,021	Ж	0,007
Т	0,053	Я	0,018	Ю	0,006
Н	0,053	Ы	0,016	Ш	0,006
С	0,045	З	0,016	Ц	0,004
Р	0,040	Ь	0,014	Щ	0,003
В	0,038	Ъ	0,014	Э	0,003
				Ф	0,002

Значение энтропии рассчитанной по формуле Шеннона для букв русского языка с учетом вероятности их наблюдения равно I≈4.72 бита.

Избыточность информации

Есть два сообщения энтропия первого сообщения равна H_1, энтропия второго H_2. При этом $H_1 < H_2$. Количество информации содержащееся в сообщениях одинаково и равно

$$I = n_1 H_1 = n_2 H_2 ,$$

где: n_1 и n_2- длина первого и второго сообщения.

Эффективность кода определяется как

$$\kappa_э = \frac{n_2}{n_1} = \frac{H_1}{H_2} . \quad (3)$$

При передаче одинакового количества информации сообщение тем длиннее, чем меньше его энтропия.

Коэффициент сжатия, характеризует степень укорочения сообщения при переходе к кодированию состояний элементов, характеризующихся большей энтропией.

Доля излишних элементов оценивается коэффициентом избыточности:

$$\kappa_и = \frac{H_2 - H_1}{H_2} = 1 - \frac{H_1}{H_2} = 1 - \kappa_э . \quad (4)$$

Средняя длина кода определяется по выражению:

$$L = \sum p(i) \cdot l_i , \quad (5)$$

где: $p(i)$ – вероятность наблюдения i-го символа, l_i – длина i-го символа.

Если вероятность наблюдения символов одинакова то $L=l$.

Определим количество информации, коэффициент сжатия и коэффициент избыточности содержащееся скороговорке:

«Разнервничавшаяся Вавилонянка Варвара, разнервничала в Вавилоне, неразнервничавшегося вавилонянина Вавилу Вавилонейского»

Для расчета количества информации необходимо определить количество символов и их количество.

В тексте всего 120 символов. В таблице 6 представлен буквенный состав текста.

Таблица 6

Буквенный состав текста

Буква	Количество	Вероятность	Буква	Количество	Вероятность	Буква	Количество	Вероятность
Р	8	0,06666	А	18	0,15	З	3	0,025
Н	14	0,11667	Е	7	0,05833	В	18	0,15
И	9	0,075	Ч	3	0,025	Ш	2	0,01666
Я	5	0,04166	С	3	0,025	«_»	9	0,075
Л	6	0,05	О	7	0,05833	К	2	0,01666
«,»	2	0,01666	Г	2	0,01666	У	1	0,00833
Й	1	0,00833						

Энтропия сообщения по Хартли (1) равна

$$H = \log_2 N = \log_2 19 = 4.24 \text{ бита.}$$

Количество информации содержащееся в сообщении по Хартли равна

$$I = nH = 120 \cdot 4.24 = 509.75 \text{ бит.}$$

Энтропия сообщения по Шеннону (2) равна

$$H = -\sum_{i=1}^{N} p_i \log_2(p_i) = -(0.066 \cdot \log_2 0.066 + 0.15 \cdot \log_2 0.15 + ... + 0.0083 \cdot \log_2 0.0083) = 3.797 \text{ бит.}$$

Количество информации содержащееся в сообщении по Шеннону равна

$$I = nH = 120 \cdot 3.797 = 455.64 \text{ бита.}$$

Для расчета коэффициентов сжатия и избыточности примем значение рассчитанное по выражению 2 за H_1, а значение рассчитанное по выражению 1 за H_2.

$$\kappa_э = \frac{H_1}{H_2} = \frac{3,797}{4,24} = 0,89$$

$$\kappa_и = 1 - \kappa_c = 1 - 0,89 = 0,11$$

Избыточность играет положительную роль, т.к. благодаря ней сообщения защищены от помех. Это используют при помехоустойчивом кодировании.

Вполне нормальный на вид лазерный диск может содержать внутренние (процесс записи сопряжен с появлением различного рода ошибок) и внешние (наличие физических разрушений поверхности диска) дефекты. Однако даже при наличии физических разрушений поверхности лазерный диск может вполне нормально читаться за счет избыточности хранящихся на нем данных. Корректирующие коды C1, C2, Q - и P- уровней восстанавливают все известные приводы, и их корректирующая способность может достигать двух ошибок на каждый из уровней C1 и C2 и до 86 и 52 ошибок на уровни Q и P соответственно. Но затем, по мере разрастания дефектов, корректирующей способности кодов Рида—Соломона неожиданно перестает хватать, и диск без всяких видимых причин отказывает читаться, а то и вовсе не опознается приводом.

Избыточность устраняют построением оптимальных кодов, которые укорачивают сообщения по сравнению с равномерными кодами. Это используют при архивации данных. Действие средств архивации основано на использовании алгоритмов сжатия, имеющих достаточно длинную историю развития, начавшуюся задолго до появления первого компьюте-

ра еще в40-х гг. XX века. Группа ученых-математиков, работавших в области электротехники, заинтересовалась возможностью создания технологии хранения данных, обеспечивающей более экономное расходование пространства. Одним из них был Клод Элвуд Шеннон, основоположник современной теории информации. Из разработок того времени позже практическое применение нашли алгоритмы сжатия Хаффмана и Шеннона-Фано. А в1977 г. математики Якоб Зив и Абрахам Лемпел придумали новый алгоритм сжатия, который позже доработал Терри Велч. Большинство методов данного преобразования имеют сложную теоретическую математическую основу. Суть работы архиваторов: они находят в файлах избыточную информацию (повторяющиеся участки и пробелы), кодируют их, а затем при распаковке восстанавливают исходные файлы по особым отметкам. Основой для архивации послужили алгоритмы сжатия Я. Зива и А. Лемпела. Первым широкое признание получил архиватор Zip. Со временем завоевали популярность и другие программы: RAR, ARJ, ACE, TAR, LHA и т. д.В операционной системе Windows достаточно четко обозначились два лидера: WinZip и WinRAR, созданный российским программистом Евгением Рошалем. WinRAR активно вытесняет WinZip так как имеет: удобный и интуитивно понятный интерфейс; мощную и гибкую систему архивации файлов; высокую скорость работы; более плотно сжимает файлы. Обе утилиты обеспечивают совместимость с большим числом архивных форматов. Помимо них к довольно распространенным архиваторам можно причислить WinArj. Стоит назвать Cabinet Manager (поддерживает формат CAB, разработанный компанией Microsoft для хранения дистрибутивов своих программ) и WinAce (работает с файлами с расширением ace и некоторыми другими). Необходимо упомянуть программы-оболочки Norton Commander, Windows Commander или Far Manager. Они позволяют путем настройки файлов конфигурации подключать внешние DOS-архиваторы командной строки и организовывать прозрачное манипулирование архивами, представляя их на экране в виде обычных каталогов. Благодаря этому с помощью комбинаций функциональных клавиш можно легко просматривать содержимое архивов, извлекать файлы из них и создавать новые архивы. Хотя программы архивации, предназначенные для MS-DOS, умеют работать и под управлением большинства версий Windows, применять их в этой операционной системе нецелесообразно. Дело в том, что при обработке файлов DOS-архиваторами их имена урезаются до 8 символов, что далеко не всегда удобно, а в некоторых случаях даже противопоказано.

Программы-архиваторы можно разделить на три категории.

1. Программы, используемые для сжатия исполняемых файлов, причем все файлы, которые прошли сжатие, свободно запускаются, но изменение их содержимого, например русификация, возможны только после их разархивации.
2. Программы, используемые для сжатия мультимедийных файлов, причем можно после сжатия эти файлы свободно использовать, хотя, как правило, при сжатии изменяется их формат (внутренняя структура), а иногда и ассоциируемая с ними программа, что может привести к проблемам с запуском.
3. Программы, используемые для сжатия любых видов файлов и каталогов, причем в основном использование сжатых файлов возможно только после разархивации. Хотя имеются программы, которые "видят" некоторые типы архивов как самые обычные каталоги, но они имеют ряд неприятных нюансов, например, сильно нагружают центральный процессор, что исключает их использование на "слабых машинах".

Популярные архиваторы ARJ, PAK, PKZIP работают на основе алгоритма Лемпела-Зива. Эти архиваторы классифицируются как адаптивные словарные кодировщики, в которых текстовые строки заменяются указателями на идентичные им строки, встречающиеся ранее в тексте. Например, все слова какой-нибудь книги могут быть представлены в виде номеров страниц и номеров строк некоторого словаря. Важнейшей отличительной чертой этого алгоритма является использование грамматического разбора предшествующего текста с расположением его на фразы, которые записываются в словарь. Указатели позволяют сделать ссылки на любую фразу в окне установленного размера, предшествующего текущей

фразе. Если соответствие найдено, текущая фраза заменяется указателем на своего предыдущего двойника.

Задания для самостоятельного выполнения

1. Определить размер тестового файл, если известно количество символов в тексте и количество байтов используемое для кодирования одно символа.
2. Определить объем оперативной памяти требуемый для хранения изображения с заданными параметрами.
3. Определить объем оперативный памяти требуемый для хранения аудиофайла с заданными параметрами
4. Определить количество различных символов исходного текста, составить алфавит сообщения и определить длину кодовых слов. Составить таблицу кодирования символов кодом постоянной длины. Закодировать исходное сообщение полученным двоичным кодом и определить длину сообщения. Рассчитать эффективность полученного кода постоянной длины для заданного сообщения.

Вариант 1

1. 12, 158
2. 541, 937, 8
3. 3907, 8, 156, 2
4. Баран карабкался с карабином

Вариант 2

1. 7, 415
2. 594, 556, 3
3. 867, 8, 178, 5
4. Таракан попал в капкан

Вариант 3

1. 11, 124
2. 508, 792, 36
3. 9959, 32, 61, 3
4. Барабанщик бил в ящик

Вариант 4

1. 15, 469
2. 553, 532, 37
3. 12903, 64, 187, 5
4. Колокол из волоколамска

Вариант 5

1. 5, 379
2. 754, 859, 59
3. 10406, 16, 154, 3
4. Полотенце попало в болото

Вариант 6

1. 10, 441
2. 605, 856, 14
3. 5415, 64, 116, 3
4. Херес попал на перевязь

Вариант 7

1. 8, 357
2. 766, 511, 29
3. 10564, 64, 178, 1
4. Мел емеля мел в мельнице

Вариант 8

1. 7, 336
2. 793, 683, 17
3. 8911, 8, 214, 4
4. На лапу упала капля пакли

Вариант 9

1. 6, 260
2. 647, 925, 61
3. 13558, 64, 69, 4
4. Не пей пену у репейника

Вариант 10

1. 5, 485
2. 411, 592, 48
3. 12755, 16, 237, 5
4. Колесил сокол около околицы

Вариант 11

1. 12, 474
2. 535, 683, 27
3. 3255, 8, 188, 5
4. Как лом сам поломался пополам

Вариант 12

1. 6, 220
2. 780, 859, 60
3. 11260, 16, 64, 1
4. Интервьюер интервента интервьюировал

Вариант 13

1. 13, 349
2. 798, 761, 52
3. 12368, 32, 82, 1
4. Скороговорки, как караси на сковородке

Вариант 14

1. 13, 169
2. 729, 577, 39
3. 3081, 16, 185, 4
4. Невелик на ситиборде бодибилдера бицепс

Вариант 15

1. 16, 232
2. 672, 789, 59
3. 4604, 16, 208, 1
4. Львы только ленивым венки не вили

Вариант 16

1. 13, 479
2. 667, 659, 49
3. 1335, 32, 208, 5
4. В балкарии валокордин из болгарии

Вариант 17

1. 13, 104
2. 508, 803, 31
3. 3029, 32, 222, 1
4. Ребрендили, да не выребрендировали

Вариант 18

1. 15, 192
2. 577, 670, 61
3. 5260, 16, 172, 1
4. Скреативлен креатив не по-креативному

Вариант 19

1. 11, 427
2. 559, 669, 15
3. 2858, 32, 226, 5
4. Паласы заменили двумя полуполосами

Вариант 20

1. 15, 138
2. 591, 546, 22
3. 1962, 16, 230, 3
4. У рекламы ухватов — швах с охватом

Вариант 21

1. 6, 313
2. 710, 942, 33
3. 1124, 64, 76, 2
4. Ядро потребителей пиастров — пираты

Вариант 22

1. 15, 492
2. 496, 516, 10
3. 17471, 32, 89, 5
4. Карл у клары украл рекламу

Вариант 23

1. 11, 475
2. 662, 735, 15
3. 11196, 64, 79, 4
4. Выборка по уборщицам на роллс-ройсах

Вариант 24

1. 15, 116
2. 626, 986, 55
3. 17357, 64, 120, 5
4. Брейншторм: гам, гром, ор ртов

СЖАТИЕ ТЕКСТОВЫХ ДАННЫХ

Определения. Аббревиатуры и классификации методов сжатия

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

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

Блок — конечная последовательность цифровой информации.

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

Сжатием блока называется такое его описание, при котором создаваемый сжатый блок содержит меньше битов, чем исходный, но по нему возможно однозначное восстановление каждого бита исходного блока. Обратный процесс, восстановление по описанию, называется разжатием**.** Используют и такие пары терминов: компрессия/декомпрессия, кодирование/декодирование.

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

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

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

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

Конечная последовательность битов называется кодом, а количество битов в коде – длиной кода.

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

Например, в блоке из 14-и элементов «кинчотсихыннад» одно слово длиной 14 элементов, два слова длиной 13, и так далее, 13 слов длиной 2 и 14 слов длиной 1. Аналогично в блоке из семи битов «0100110» один код длиной 7 битов, два кода длиной 6, и так далее, семь кодов длиной 1.

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

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

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

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

Можно говорить, что источник без памяти порождает «элементы», а источник данных с памятью — «слова», поскольку во втором случае

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

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

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

Третья важнейшая применяемая при сжатии данных математическая модель — «аналоговый сигнал»:

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

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

Эффективность сжатия учитывает не только степень сжатия (отношение длины несжатых данных к длине соответствующих им сжатых данных), но и скорости сжатия и разжатия. Часто пользуются обратной к степени сжатия величиной – коэффициентом сжатия, определяемым как отношение длины сжатых данных к длине соответствующих им несжатых.

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

Названия и аббревиатуры методов

CM (Context Modeling) — Контекстное моделирование.

DMC (Dynamic Markov Compression) — Динамическое марковское сжатие (является частным случаем CM).

PPM (Prediction by Partial Match) — Предсказание по частичному совпадению (является частным случаем CM).

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

PBS (Parallel Blocks Sorting) — Сортировка параллельных блоков.

ST (Sort Transformation) — Частичное сортирующее преобразование (является частным случаем PBS).

BWT (Burrows-Wheeler Transform) — Преобразование Барроуза-Уилера (является частным случаем ST)

RLE (Run Length Encoding) — Кодирование длин повторов.

HUFF (Huffman Coding) — кодирование по методу Хаффмана.

SEM (Separate Exponents and Mantissas) — Разделение экспонент и мантисс (Представление целых чисел).

UNIC (Universal Coding) — Универсальное кодирование (является частным случаем SEM).

ARIC (Arithmetic Coding) — Арифметическое кодирование.

RC (Range Coding) — Интервальное кодирование (вариант арифметического).

DC (Distance Coding) — Кодирование расстояний.

IF (Inverted Frequences) — «Обратные частоты» (вариант DC).

MTF (Move To Front) — «Сдвиг к вершине», «Перемещение стопки книг».

ENUC (Enumerative Coding) — Нумерующее кодирование.

FT (Fourier Transform) — Преобразование Фурье.

DCT (Discrete Cosine Transform) — Дискретное Косинусное Преобразование, ДКП (является частным случаем FT).

DWT (Discrete Wavelet Transform) — Дискретное Вэйвлетное Преобразование, ДВП.

LPC (Linear Prediction Coding) — Линейно-Предсказывающее Кодирование, ЛПК (к нему относятся Дельта-кодирование, ADPCM, CELP и MELP).

SC (Subband Coding) — Субполосное кодирование.

VQ (Vector Quantization) — Векторное квантование.

Методы сжатия без потерь

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

Точная связь между вероятностями и кодами установлена в теореме Шеннона о кодировании источника, которая гласит, что элемент s_i, вероятность появления которого равняется $p(s_i)$, выгоднее всего представлять $-\log_2 p(s_i)$ битами. Если при кодировании размер кодов всегда в точности получается равным $-\log_2 p(s_i)$ битам, то в этом случае длина закодированной последовательности будет минимальной для всех возможных способов кодирования. Если распределение вероятностей $F=\{p(s_i)\}$ неизменно, и вероятности появления элементов независимы, то мы можем найти среднюю длину кодов как среднее взвешенное по выражению 2.

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

$$H = -\sum_k P_k \cdot H_k = -\sum_{k,i} P_k \cdot p_k(s_i) \log_2 p_k(s_i) \qquad (6)$$

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

Итак, если известно распределение вероятностей элементов, генерируемых источником, то возможно представить данные наиболее компактным образом, при этом средняя длина кодов может быть вычислена выражению (2).

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

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

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

Алгоритм Хаффмана

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

Для начала введем несколько определений.

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

$$A=a_{i_1}a_{i_2}\ldots a_{i_n}$$

называется словом в алфавите Ψ, а число n — длиной слова A. Длина слова обозначается как $l(A)$.

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

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

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

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

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

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

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

Пусть слово B имеет вид

$$B=B'B''$$

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

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

Если схема Σ обладает свойством префикса, то алфавитное кодирование будет взаимно однозначным.

Предположим, что задан алфавит $\Psi=\{a_1,..., a_r\}$ (r>1) и набор вероятностей $p_1, p_2, \ldots, p_r$ $\left(\sum_{i=1}^{r} p_i = 1\right)$ появления символов $a_1,..., a_r$. Пусть, далее, задан алфавит Ω, $\Omega=\{b_1, ..., b_q\}$ (q>1). Тогда можно построить целый ряд схем Σ алфавитного кодирования

$$a_1–B_1, a_2–B_2. a_r– B_r,$$

обладающих свойством взаимной однозначности.

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

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

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

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

$$l_* = \min_{\Sigma} l_{cp}^{\Sigma}$$

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

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

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

Алгоритм построения схемы Σ можно представить следующим образом:

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

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

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

Пример. Пусть алфавит источника содержит шесть элементов {А, Б, В, Г, Д, Е}, появляющихся с вероятностями р(А)=0,15, р(Б)=0,25, р(В)=0,1, р(Г)=0,13, р(Д)=0,25, р(Е)=0,12.

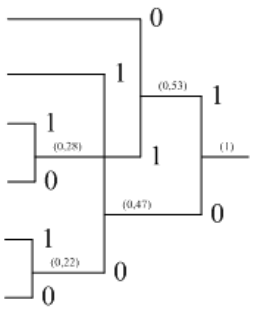
В таблице 7 представлен процесс построения схемы алгоритма Хаффмана.

Для кодирования 6-ти символов равномерным кодом необходимо 3 бита.

Средняя длина кода полученного по алгоритму Хаффмана 2,5 бита.

Таблица 7

Построение схемы по алгоритму Хаффмана

Элемент	Вероятность	Схема	Код
Б	0,25		10
Д	0,25		01
А	0,15		111
Г	0,13		110
Е	0,12		001
В	0,1		000

Канонический алгоритм Хаффмана требует помещения в файл со сжатыми данными таблицы соответствия кодируемых символов и кодирующих цепочек.

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

Характерной особенностью алгоритма является то что, он не увеличивает размера исходных данных в худшем случае (если не считать необходимости хранить таблицу перекодировки вместе с файлом).

Арифметическое кодирование

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

На рисунке 2 представлено сравнение оптимального кодирования и кодирования по методу Хаффмана. Видно, что в ситуации, когда относительные частоты не являются степенями двойки, сжатие становится менее эффективным (тратиться больше битов, чем это необходимо). Например, если у нас два символа *a* и *b* с вероятностями 253/256 и 3/256, то в идеале мы должны потратить на цепочку из 256 байт $-\log_2(253/256)\cdot 253-\log_2(3/256)\cdot 3 = 23.546$, т.е. 24 бита. При кодировании по Хаффману мы закодируем *a* и *b* как 0 и 1, и нам придется потратить $1\cdot 253+1\cdot 3=256$ битов, т.е. в 10 раз больше. Рассмотрим алгоритм, дающий результат, близкий к оптимальному.

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

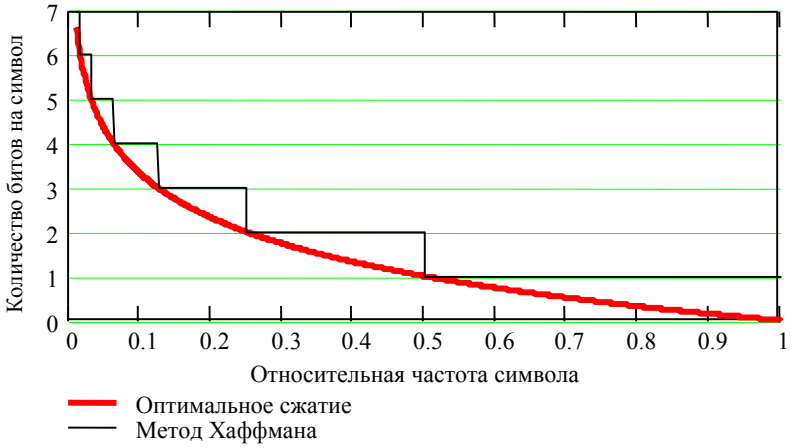


Рис. 2. Сравнение оптимального кодирования и кодирования по методу Хаффмана

В качестве примера используем текст "КОВ.КОРОВА". Определим вероятности появления каждого символа в тексте (в порядке убывания) и соответствующие этим символам диапазоны (таблица 8)

Таблица 8

Диапазоны символов текста

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

Пусть частоты символов известны в компрессоре и декомпрессоре. Кодирование заключается в уменьшении рабочего интервала. Для первого символа в качестве рабочего интервала берется [0, 1). Выбранный интервал разбивается на диапазоны в соответствии с заданными частотами символов (таблица 8). В качестве следующего рабочего интервала берется диапазон, соответствующий текущему кодируемому символу. Его длина пропорциональна вероятности появления этого символа в потоке. Далее считываем следующий символ. В качестве исходного берем рабочий интервал, полученный на предыдущем шаге, и опять разбиваем его в соответствии с таблицей диапазонов. Длина рабочего интервала уменьшается пропорционально вероятности текущего символа, а точка начала сдвигается вправо пропорционально началу диапазона для этого символа. Новый построенный диапазон берется в качестве рабочего, и т.д.

Исходный рабочий интервал: [0, 1).

Символ "К" [0.3; 0.5) получаем [0.3000; 0.5000).

Символ "О" [0.0; 0.3) получаем [0.3000; 0.3600).

Символ "В" [0.5; 0.7) получаем [0.3300; 0.3420).

Символ "." [0.9; 1.0) получаем [0,3408; 0.3420).

Процесс кодирования первых трех символов представлен на рисунке 3.

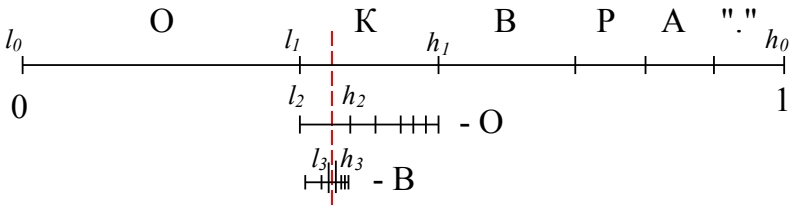


Рис. 3. Арифметическое кодирование

На рисунке 3 вертикальной чертой обозначено произвольное число, лежащее в полученном при работе интервале $[l_i, h_i)$. Для последовательности "КОВ.", состоящей из 3 символов, за такое число можно взять 0.341. Этого числа достаточно для восстановления исходной цепочки, если известна исходная таблица диапазонов и длина цепочки.

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

l_0=0; h_0=1; i=0;

while(not DataFile.EOF()){

c = DataFile.ReadSymbol(); i++;

$l_i = l_{i-1} + a[c]\cdot(h_{i-1} - l_{i-1})$;

$h_i = l_{i-1} + b[c]\cdot(h_{i-1} - l_{i-1})$;};

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

l_0=0; h_0=1; value=File.Code();

for(i=1; i<=File.DataLength(); i++){

for(для всех c_j){

$l_i = l_{i-1} + a[c_j]\cdot(h_{i-1} - l_{i-1})$;

$h_i = l_{i-1} + b[c_j]\cdot(h_{i-1} - l_{i-1})$;

if ((l_i <= value) && (value < h_i)) break;};

DataFile.WriteSymbol(c_j);};

Где value — прочитанное из потока число (дробь), а c — записываемые в выходной поток распаковываемые символы. При использовании алфавита из 256 символов c_j, внутренний цикл выполняется достаточно долго, однако его можно ускорить. Заметим, что поскольку $b[c_{j+1}]=a[c_j]$ (см. приведенную выше таблицу диапазонов), то l_i для c_{j+1} равно h_i для c_j, а последовательность h_i для c_j строго возрастает с ростом j. Т.е. количество операций во внутреннем цикле можно сократить вдвое, поскольку достаточно проверять только одну границу интервала. Также, если у нас мало символов, то, отсортировав их в порядке уменьшения вероятностей, мы сокращаем число итераций цикла и, таким образом, ускоряем работу декомпрессора. Первыми будут проверяться символы с наибольшей вероятностью, например в нашем примере мы с вероятностью 0,5 будем выходить из цикла уже на втором символе из шести. Если число символов велико, существуют другие эффективные методы ускорения поиска символов (например, бинарный поиск).

Приведенный выше алгоритм работоспособен, он будет работать медленно, по сравнению с алгоритмом, оперирующим двоичными дробями. Двоичная дробь задается как $0.A_1A_2A_{3...}A_i = A_1 \cdot 1/2 + A_2 \cdot 1/4 + A_3 \cdot 1/8 + \ldots A_i \cdot 1/2^i$. Таким образом, при сжатии необходимо дописывать в дробь дополнительные знаки до тех пор, пока получившееся число не попадет в интервал, соответствующий закодированной цепочке (рисунок 4). Получившееся число полностью задает закодированную цепочку при аналогичном алгоритме декодирования.

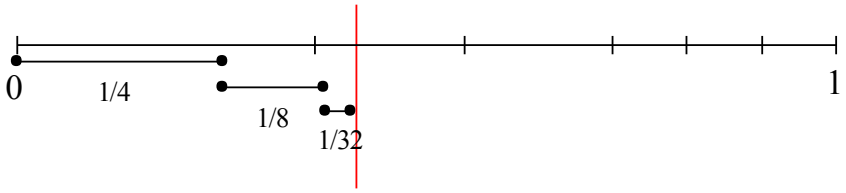


Рис. 4. Применение двоичных дробей

Особенностью арифметического кодирования является способность сильно сжимать отдельные длинные цепочки. Например, один бит "1" (двоичное число "0.1") для интервалов указанных в таблице 8 однозначно задает цепочку "BOOOOOOOOO…" произвольной длины (например, 1000000000 символов). Т.е. если наш файл заканчивается одинаковыми символами, например массивом нулей, то этот файл может быть сжат с весьма впечатляющей степенью сжатия. Очевидно, что длину исходного файла при этом следует передавать декомпрессору явным образом перед сжатыми данными, как это делалось в приведенных выше примерах.

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

Таблица 9

Диапазоны символов текста

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

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

```
l0=0; h0=65535; i=0; delitel= b[clast]; // delitel=10
First_qtr = (h0+1)/4;   // = 16384
Half = First_qtr*2;     // = 32768
Third_qtr = First_qtr*3;// = 49152
bits_to_follow =0;      // Сколько битов сбрасывать
while(not DataFile.EOF()) {
  c = DataFile.ReadSymbol();  // Читаем символ
  j = IndexForSymbol(c); i++; // Находим его индекс
  li = li-1 + b[j-1]*(hi-1 - li-1 + 1)/delitel;
  hi = li-1 + b[j  ]*(hi-1 - li-1 + 1)/delitel - 1;
  for(;;) {          // Обрабатываем варианты
          if(hi < Half)    // переполнения
                  BitsPlusFollow(0);
          else if(li >= Half) {
                  BitsPlusFollow(1);
                  li-= Half; hi-= Half;
          }
          else if((li >= Third_qtr)&&(hi < First_qtr)){
                  bits_to_follow++;
                  li-= First_qtr; hi-= First_qtr;
          } else break;
          li+=li; hi+= hi+1;}}
// Процедура переноса найденных битов в файл
void BitsPlusFollow(int bit)
{ CompressedFile.WriteBit(bit);
  for(; bits_to_follow > 0; bits_to_follow--)
          CompressedFile.WriteBit(!bit);}
```

Результат кодирования представлен в таблице 10

Таблица 10

Результат кодирования

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

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

```
l0=0; h0=65535; delitel= b[clast];
First_qtr = (h0+1)/4;     // = 16384
```

```
Half = First_qtr*2;       // = 32768
Third_qtr = First_qtr*3;          // = 49152
value=CompressedFile.Read16Bit();
for(i=1; i< CompressedFile.DataLength(); i++){
  freq=((value-l_{i-1}+1)*delitel-1)/(h_{i-1} - l_{i-1} + 1);
  for(j=1; b[j]<=freq; j++);   // Поиск символа
  l_i = l_{i-1} + b[j-1]*(h_{i-1} - l_{i-1} + 1)/delitel;
  h_i = l_{i-1} + b[j  ]*(h_{i-1} - l_{i-1} + 1)/delitel - 1;
  for(;;) {          // Обрабатываем варианты
          if(h_i < Half)    // переполнения
                  ; // Ничего
          else if(l_i >= Half) {
                  l_i-= Half; h_i-= Half; value-= Half;
          }
          else if((l_i >= Third_qtr)&&(h_i < First_qtr)){
                  l_i-= First_qtr; h_i-= First_qtr;
                  value-= First_qtr;
          } else break;
          l_i+=l_i; h_i+= h_i+1;
          value+=value+CompressedFile.ReadBit();}
  DataFile.WriteSymbol(c);};
```

Как видно, погрешность вычисления, компенсируется синхронным выполнением операции над l_i и h_i синхронно в компрессоре и декомпрессоре.

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

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

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

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

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

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

Словарные методы сжатия данных

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

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

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

Уменьшение размера возможно в первую очередь за счет того, что обычно в сжимаемых данных встречается лишь малая толика всех возможных строк длины n, поэтому для представления индекса фразы требуется, как правило, меньшее число битов, чем для представления исходной строки. Например, рассмотрим количество взаимно различных строк длины от 1 до 5 в тексте (таблица 11) на русском языке (роман Ф.М. Достоевского «Бесы», обычный неформатированный текст, размер около 1.3 Мбайт):

Далее, если есть заслуживающие доверия гипотезы о частоте использования тех или иных фраз, либо проводился какой-то частотный анализ обрабатываемых данных, возможно назначить более вероятным фразам коды меньшей длины. Например, для той же электронной версии романа «Бесы» статистика встречаемости строк длины 5 представлена в таблице 12

Размер (мощность) алфавита равен 136 символам, но используется только $2536/(136\cdot136)\cdot100\% \approx 13.7$ % от всех возможных двухсимвольных строк, и т.д.

Таблица 11

Количество взаимно различных строк

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

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

Таблица 12

Встречаемости строк

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

Обычно же просто предполагается, что короткие фразы используются чаще длинных. Поэтому в большинстве случаев индексы строятся таким образом, чтобы длина индекса короткой фразы была меньше длины индекса длинной фразы. Такой прием обычно способствует улучшению сжатия.

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

Классические алгоритмы Зива-Лемпела

Алгоритмы словарного сжатия Зива-Лемпела появились во второй половине 1970-х годов. Это были так называемые алгоритмы LZ77 и LZ78, разработанные совместно Зивом (Ziv) и Лемпелом (Lempel). В дальнейшем первоначальные схемы подвергались множественным изменениям, в результате чего в настоящее время имеются десятки достаточно самостоятельных алгоритмов и бессчетное количество модификаций.

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

Публикации Зива и Лемпела носили чисто теоретический характер, так как эти исследователи на самом деле занимались проблемой измерения «сложности» строки, и применение выработанных алгоритмов к сжатию данных явилось, скорее, лишь частным результатом. Потребовалось некоторое время, чтобы идея организации словаря, часто в переложении уже других людей, достигла разработчиков программного и аппаратного обеспечения. Поэтому практическое использование алгоритмов началось спустя пару лет.

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

Необходимо сказать несколько слов о наименованиях алгоритмов и методов. При обозначении семейства общепринятой является аббревиатура «LZ», но расшифровываться она должна как «Ziv-Lempel», поэтому и алгоритмы «Зива-Лемпела», а не «Лемпела-Зива». Со-

гласно общепринятому объяснению этого курьеза, Якоб Зив внес больший вклад в открытие соответствующих словарных схем и исследование их свойств и таким образом заслужил, чтобы первым стояла его фамилия, что мы и видим в заголовках статей. Но случайно была допущена ошибка, и прикрепилось сокращение «LZ» (буквы упорядочены в алфавитном порядке). Иногда, кстати, встречается и обозначение «ZL». В дальнейшем, если некий исследователь существенно изменял какой-то алгоритм, относимый к семейству LZ, то в названии полученной модификации к строчке «LZ» обычно дописывалась первая буква его фамилии, например: алгоритм LZB, автор Белл (Bell).

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

Алгоритм LZ77

Этот словарный алгоритм сжатия является самым старым среди методов LZ. Описание было опубликовано в 1977 году, но сам алгоритм разработан не позднее 1975 года.

Алгоритм LZ77 является «родоначальником» целого семейства словарных схем — так называемых алгоритмов со скользящим словарем, или скользящим окном. Действительно, в LZ77 в качестве словаря используется блок уже закодированной последовательности. Как правило, по мере выполнения обработки положение этого блока относительно начала последовательности постоянно меняется, словарь «скользит» по входному потоку данных.

Скользящее окно имеет длину N, т.е. в него помещается N символов, и состоит из 2 частей:

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

Пусть к текущему моменту времени мы уже закодировали t символов $s_1, s_2, \ldots, s_t$. Тогда словарем будут являться W предшествующих символов $s_{t-(W-1)}, s_{t-(W-1)+1}, \ldots, s_t$. Соответственно, в буфере находятся ожидающие кодирования символы $s_{t+1}, s_{t+2}, \ldots, s_{t+n}$. Очевидно, что если $W \geq t$, то словарем будет являться вся уже обработанная часть входной последовательности.

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

- смещения (offset) от начала буфера, i;
- длины соответствия, или совпадения (match length), j.
- Смещение и длина соответствия играют роль указателя (ссылки, однозначно определяющего фразу. Дополнительно в выходной поток записывается символ s, непосредственно следующий за совпавшей строкой буфера.

Таким образом, на каждом шаге кодер выдает описание трех объектов: смещения и длины соответствия, образующих код фразы, равной обработанной строке буфера, и одного символа s (литерала). Затем *окно* смещается на $j+1$ символов вправо и осуществляется пере-

ход к новому циклу кодирования. Величина сдвига объясняется тем, что мы реально закодировали именно *j*+1 символов: *j* с помощью указателя на фразу в словаре, и 1 с помощью тривиального копирования. Передача одного символа в явном виде позволяет разрешить проблему обработки еще ни разу не виденных символов, но существенно увеличивает размер сжатого блока.

Рассмотрим работу алгоритма LZ77 на примере сжатия строки «кот_ломом_колол_слона» длиной 21 символ. Пусть длина буфера равна 7 символам, а размер словаря больше длины сжимаемой строки (таблица 13). Условимся также, что:

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

Таблица 13

Пример алгоритма LZ77

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

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

Процесс кодирования можно описать следующим образом.

```
while ( ! DataFile.EOF() ){
/найдем максимальное совпадение, в match_pos получим смещение i, в match_len— длину j, в
unmatched_sym — первый несовпавший символ s_{t+1+j}; считаем также, что в функции
find_match учитывается ограничение на длину совпадения /
    find_match (&match_pos, &match_len, &unmatched_sym);
/запишем в файл сжатых данных описание найденной фразы, при этом длина битового
представления i задается константой OFFS_LN, длина представления j — константой
LEN_LN, размер символа s принимаем равным 8 битам/
    CompressedFile.WriteBits (match_pos, OFFS_LN);
```

```
CompressedFile.WriteBits (match_len, LEN_LN);
CompressedFile.WriteBits (unmatched_sym, 8);
for (i = 0; i <= match_len; i++){
/ прочтем очередной символ
c = DataFile.ReadSymbol();
/удалим из словаря одну самую старую фразу
DeletePhrase ();
/добавим в словарь одну фразу, начинающуюся с первого символа буфера
AddPhrase ();
/сдвинем окно на 1 позицию, добавим в конец буфера символ c/
MoveWindow(c);}}
CompressedFile.WriteBits (0, OFFS_LN);
```

Пример подтвердил, что способ формирования кодов в LZ77 неэффективен и позволяет сжимать только сравнительно длинные последовательности. До некоторой степени сжатие небольших файлов можно улучшить, используя коды переменной длины для смещения *i*. Действительно, даже если мы используем словарь в 32 кбайт, но закодировали еще только 3 кбайт, то смещение реально требует не 15, а 12 битов. Кроме того, происходит существенный проигрыш из-за использования кодов одинаковой длины при указании длин совпадения *j*. Например, для электронной версии романа «Бесы» частоты использования длин совпадения представлены в таблице 14.

Из таблицы 14 следует, что в целях минимизации закодированного представления для $j = 6$ следует использовать код наименьшей длины, так как эта длина совпадения встречается чаще всего.

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

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

Таблица 14

Частоты использования длин совпадения

j	Количество раз, когда максимальная длина совпадения была равна *j*
0	136
1	1593
2	4675
3	11165
4	20047
5	26939
6	28653
7	24725
8	19702
9	14767
10	10820
≥11	27903

Алгоритм декодирования может иметь следующий вид.

```
for (;;) {
// читаем смещение
match_pos = CompressedFile.ReadBits (OFFS_LN);
if (!match_pos)
// обнаружен признак конца файла, выходим из цикла
break;
// читаем длину совпадения
match_len = CompressedFile.ReadBits (LEN_LN);
for (i = 0; i < match_len; i++) {
/находим в словаре очередной символ совпавшей фразы
c = Dict (match_pos + i);
/сдвигаем словарь на 1 позицию, добавляем в его начало c
MoveDict (c)
/*записываем очередной раскодированный символ в выходной файл
DataFile.WriteSymbol (c);}
/*читаем несовпавший символ, добавляем его в словарь и записываем в выходной файл
c = CompressedFile.ReadBits (8);
MoveDict (c)
DataFile.WriteSymbol (c);}
```

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

Алгоритм LZSS

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

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

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

Закодируем строку «кот_ломом_колол_слона» (таблица 15) из предыдущего примера и сравним коэффициент сжатия для LZ77 и LZSS.

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

Таким образом, для кодирования строки по алгоритму LZSS потребовалось 17 шагов: 13 раз символы были переданы в явном виде, и 4 раза применялись указатели. Заметим, что при работе по алгоритму LZ77 потребовалось всего лишь 12 шагов. С другой стороны, если задаться теми же длинами для *i* и *j*, то размер закодированных по LZSS данных равен

13·(1+8) + 4·(1+5+3) = 153 битам. Это означает, что строка действительно была сжата, так как ее исходный размер 168 битов.

Таблица 15

Пример алгоритма LZSS

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

Рассмотрим алгоритм сжатия LZSS

```
const int
/порог для включения словарного кодирования
THRESHOLD = 1,
/размер представления смещения, в битах
OFFS_LN = 14,
/размер представления длины совпадения, в битах
LEN_LN = 4;
const int
WIN_SIZE = (1 << OFFS_LN), // размер окна
BUF_SIZE = (1 << LEN_LN) — 1; // размер буфера
/функция вычисления реального положения символа в окне
inline int MOD (int i) { return i & (WIN_SIZE-1); };
/собственно алгоритм сжатия
int buf_sz = BUF_SIZE;
 инициализация: заполнение буфера, поиск совпадения для первого шага
while ( buf_sz ) {
if ( match_len > BUF_SIZE) match_len = BUF_SIZE;
if ( match_len <= THRESHOLD ) {
```

/если длина совпадения меньше порога (1 в примере), то запишем в файл сжатых данных флаг и символ; pos определяет позицию начала буфера

```
CompressedFile.WriteBit (0);
CompressedFile.WriteBits (window [pos], 8);
```

```
// это понадобится при обновлении словаря
match_len = 1;
}else{
/иначе запишем флаг и указатель, состоящий из смещения и длины совпадения
CompressedFile.WriteBit (1);
CompressedFile.WriteBits (match_offs, OFFS_LN);
CompressedFile.WriteBits (match_len, LEN_LN);
}
for (int i = 0; i < match_len; i++) {
/удалим из словаря фразу, начинающуюся в позиции MOD (pos+buf_sz)
DeletePhrase ( MOD (pos+buf_sz) );
if ( (c = DataFile.ReadSymbol ()) == EOF)
// мы в конце файла, надо сократить буфер
buf_sz--;
else
/иначе надо добавить в конец буфера новый символ
window [MOD (pos+buf_sz)] = c;
pos = MOD (pos+1); /сдвиг окна на 1 символ
if (buf_sz)
```

/если в буфере еще что-то есть, то добавим в словарь новую фразу, начинающуюся в позиции pos; считаем, что в функции AddPhrase одновременно выполняется поиск максимального совпадения между буфером и фразами словаря

```
AddPhrase (pos, &match_offs, &match_len);}}
CompressedFile.WriteBit (1);
CompressedFile.WriteBits (0, OFFS_LN); // знак конца файла
```

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

Алгоритм декодирования может быть реализован следующим образом.

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

```
for (int i = 0; i < match_len; i++) {
//выдаем очередной совпавший символ с
c = window [MOD (match_pos+i)];
DataFile.WriteSymbol (c);
window [pos] = c;
pos = MOD (pos+1);}}}
```

Алгоритм LZ78

Алгоритм LZ78 был опубликован в 1978 году, и впоследствии стал «отцом» семейства словарных методов LZ78.

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

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

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

Закодируем строку «кот_ломом_колол_слона» длиной 21 символ (таблица 16). Для LZ78 буфер, в принципе, не требуется, поскольку достаточно легко так реализовать поиск совпадающей фразы максимальной длины, что последовательность незакодированных символов будет просматриваться только один раз. Поэтому буфер показан только с целью большей доходчивости примера. Фразу с номером 0 зарезервируем для обозначения конца сжатой строки, номером 1 будем задавать пустую фразу словаря.

Таблица 16

Алгоритм LZ78

Шаг	Добавляемая в словарь фраза		Буфер	Совпадающая фраза S	Закодированные данные	
	сама фраза	Номер фразы			n	s
1	к	2	кот_лом	-	1	‘к’
2	о	3	от_ломо	-	1	‘о’
3	т	4	т_ломом	-	1	‘т’
4	_	5	_ломом_	-	1	‘ _’
5	л	6	ломом_к	-	1	‘л’
6	ом	7	омом_ко	о	3	‘м’
7	ом_	8	ом_коло	ом	7	‘ _’
8	ко	9	колол_с	к	2	‘о’
9	ло	10	лол_сло	л	6	‘о’
10	л_	11	л_слона	л	6	‘ _’
11	с	12	слона	-	1	‘с’
12	лон	13	лона	ло	10	‘н’
13	а	14	а	-	1	‘а’

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

Ниже приведен пример реализации алгоритма сжатия LZ78.

```
n = 1;
while ( ! DataFile.EOF() ){
  s = DataFile.ReadSymbol; // читаем очередной символ
```

/пытаемся найти в словаре фразу, представляющую собой конкатенацию родительской фразы с номером n и символа s; функция возвращает номер искомой фразы в phrase_num; если же фразы нет, то phrase_num принимает значение 1, т.е. указывает на пустую фразу

```
FindPhrase (&phrase_num, n, s);
if (phrase_num != 1)
```

/такая фраза имеется в словаре, продолжим поиск совпадающей фразы максимальной длины

```
n = phrase_num;
else {
```

/такой фразы нет, запишем в выходной файл код INDEX_LN – это константа, определяющая длину битового представления номера n

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

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

```
for (;;){
  // читаем индекс родительской фразы
  n = CompressedFile.ReadBits (INDEX_LN);
  if (!n)
        break; // конец файла
  // читаем несовпавший символ s
s = CompressedFile.ReadBits (8);
```

/находим в словаре позицию начала фразы с индексом n и ее длину

```
  GetPhrase (&pos, &len, n)
```

/записываем фразу с индексом n в файл раскодированных данных

```
  for (i = 0; i < len; i++)
  DataFile.WriteSymbol (Dict[pos+i]);
// записываем в файл символ s
DataFile.WriteSymbol (s);
  AddPhrase (n, s); // добавляем новую фразу в словарь
}
```

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

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

Интересное свойство LZ78 заключается в том, что если исходные данные порождены источником с определенными характеристиками (он должен быть стационарным и эргодическим), то коэффициент сжатия приближается по мере кодирования к минимальному достижимому. Иначе говоря, количество битов, затрачиваемых на кодирование каждого символа, в среднем равно энтропии источника. Но, к сожалению, сходимость медленная, и на данных реальной длины алгоритм ведет себя не лучшим образом. Так, например, коэффициент сжатия текстов в зависимости от их размера обычно колеблется от 3.5 до 5 битов/символ. Кроме того, нередки ситуации, когда обрабатываемые данные порождены источником с ярко выраженной нестационарностью. Поэтому при оценке реального поведения алгоритма следует относиться с большой осторожностью к теоретическим выкладкам, обращая внимание на выполнение соответствующих условий.

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

Методы контекстного моделирования

Применение методов контекстного моделирования для сжатия данных опирается на парадигму сжатия с помощью «универсальных моделирования и кодирования» (universal modelling and coding), предложенную Риссаненом (Rissanen) и Лэнгдоном (Langdon) в 1981 году. В соответствии с данной идеей процесс сжатия состоит из двух самостоятельных частей:

- моделирование;
- кодирование.

Под моделированием понимается построение модели информационного источника, породившего сжимаемые данные, а под кодированием – отображение обрабатываемых данных в сжатую форму представления на основании результатов моделирования (рис. 5). «Кодировщик» создает выходной поток, являющийся компактной формой представления обрабатываемой последовательности, на основании информации, поставляемой ему «моделировщиком».

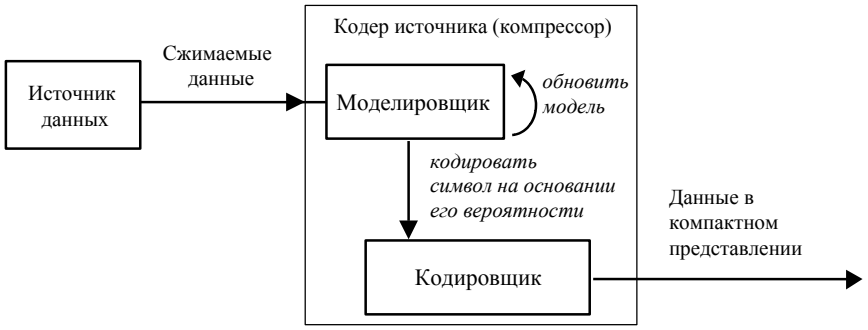


Рис. 5. Схема процесса сжатия данных в соответствии с концепцией универсальных моделирования и кодирования

Следует заметить, что понятие «кодирование» часто используют в широком смысле для обозначения всего процесса сжатия, т.е. включая моделирование в данном определении. Таким образом, необходимо различать понятия кодирования в широком смысле (весь процесс) и в узком (генерация потока кодов на основании информации модели). Понятие «статистическое кодирование» также используется, зачастую с сомнительной корректностью, для обо-

значения того или иного уровня кодирования. Во избежание путаницы ряд авторов применяет термин «энтропийное кодирование» для кодирования в узком смысле. Это наименование далеко от совершенства и встречает вполне обоснованную критику. Далее в этой главе процесс кодирования в широком смысле будем именовать «кодированием», а в узком смысле – «статистическим кодированием, или «собственно кодированием».

Из теоремы Шеннона о кодировании источника известно, что символ s_i, вероятность появления которого равняется $p(s_i)$, выгоднее всего представлять $-\log_2 p(s_i)$ битами, при этом средняя длина кодов может быть вычислена по приводившейся ранее формуле (1). Практически всегда истинная структура источника скрыта, поэтому необходимо строить модель источника, которая позволила бы в каждой позиции входной последовательности найти оценку $q(s_i)$ вероятности появления каждого символа s_i алфавита входной последовательности.

Оценка вероятностей символов при моделировании производится на основании известной статистики и, возможно, априорных предположений, поэтому часто говорят о задаче статистического моделирования. Можно сказать, что моделировщик предсказывает вероятность появления каждого символа в каждой позиции входной строки, отсюда еще одно наименование этого компонента – «предсказатель», или «предиктор» (от “predictor”). На этапе статистического кодирования выполняется замещение символа s_i с оценкой вероятности появления $q(s_i)$ кодом длиной $-\log_2 q(s_i)$ битов.

Рассмотрим пример. Предположим, что сжимается последовательность символов алфавита {‘0’,’1’}, порожденную источником без памяти, и вероятности генерации символов следующие: $p(\text{‘0’}) = 0.4$, $p(\text{‘1’}) = 0.6$. Пусть наша модель дает такие оценки вероятностей: $q(\text{‘0’}) = 0.35$, $q(\text{‘1’}) = 0.65$. Энтропия H источника равна

$$-p('0')\log_2 p('0') - p('1')\log_2 p('1') =$$
$$= -0.4\log_2 0.4 - 0.6\log_2 0.6 \approx 0.971 \textit{ бита.}$$

Если подходить формально, то «энтропия» модели получается равной

$$-q('0')\log_2 q('0') - q('1')\log_2 q('1') =$$
$$= -0.35\log_2 0.35 - 0.65\log_2 0.65 \approx 0.934 \textit{ бита.}$$

Казалось бы, что модель обеспечивает лучшее сжатие, чем это позволяет формула Шеннона. Но истинные вероятности появления символов не изменились! Если исходить из вероятностей p, то ‘0’ следует кодировать $-\log_2 0.4 \approx 1.322$ бита, а для ‘1’ нужно отводить $-\log_2 0.6 \approx 0.737$ бита. Для оценок вероятностей q мы имеем $-\log_2 0.35 \approx 1.515$ бита и $-\log_2 0.65 \approx 0.621$ бита соответственно. При каждом кодировании на основании информации модели в случае ‘0’ мы будем терять $1.515 - 1.322 = 0.193$ бита, а в случае ‘1’ выигрывать $0.737 - 0.621 = 0.116$ бита. С учетом вероятностей появления символов средний проигрыш при каждом кодировании составит $0.4 \cdot 0.193 - 0.6 \cdot 0.116 = 0.008$ бита.

Чем точнее оценка вероятностей появления символов, тем больше коды соответствуют оптимальным, тем лучше сжатие.

Правильность декодирования обеспечивается использованием точно такой же модели, что была применена при кодировании. Следовательно, при моделировании для сжатия данных нельзя пользоваться информацией, которая неизвестна декодеру.

Осознание двойственной природы процесса сжатия позволяет осуществлять декомпозицию задач компрессии данных со сложной структурой и нетривиальными взаимозависимостями, обеспечивать определенную самостоятельность процедур, решающих частные проблемы, сосредотачивать больше внимания на деталях реализации конкретного элемента.

Задача статистического кодирования была в целом успешно решена к началу 1980-х годов. Арифметический кодер позволяет сгенерировать сжатую последовательность, длина которой обычно всего лишь на десятые доли процента превышает теоретическую длину, рассчитанную с помощью формулы (1). Более того, применение современной модификации

арифметического кодера – интервального кодера – позволяет осуществлять собственно кодирование очень быстро. Скорость статистического кодирования составляет миллионы символов в секунду на современных ПК.

В свете вышесказанного, повышение точности моделей является, фактически, единственным способом существенного улучшения сжатия.

Классификация стратегий моделирования

Перед рассмотрением контекстных методов моделирования следует сказать о классификации стратегий моделирования источника данных по способу построения и обновления модели. Выделяют четыре варианта моделирования:

- статическое;
- полуадаптивное;
- адаптивное (динамическое);
- блочно-адаптивное

При статическом моделировании для любых обрабатываемых данных используется одна и та же модель. Иначе говоря, не производится адаптация модели к особенностям сжимаемых данных. Описание заранее построенной модели хранится в структурах данных кодера и декодера; таким образом достигается однозначность кодирования, с одной стороны, и отсутствие необходимости в явной передачи модели, с другой. Недостаток подхода также очевиден: мы можем получать плохое сжатие и даже увеличивать размер представления, если обрабатываемые данные не соответствуют выбранной модели. Поэтому такая стратегия используется только в специализированных приложениях, когда тип сжимаемых данных неизменен и заранее известен.

Полуадаптивное сжатие является развитием стратегии статического моделирования. В этом случае для сжатия заданной последовательности выбирается или строится модель на основании анализа именно обрабатываемых данных. Понятно, что кодер должен передавать декодеру не только закодированные данные, но и описание использованной модели. Если модель выбирается из заранее созданных и известных как кодеру, так и декодеру, то это просто порядковый номер модели. Иначе, если модель была настроена или построена при кодировании, то необходимо передавать либо значения параметров настройки, либо модель полностью. В общем случае полуадаптивный подход дает лучшее сжатие, чем статический, т.к. обеспечивает приспособление к природе обрабатываемых данных, уменьшая вероятность значительной разницы между предсказаниями модели и реальным поведением потока данных.

Адаптивное моделирование является естественной противоположностью статической стратегии. По мере кодирования модель изменяется по заданному алгоритму после сжатия каждого символа. Однозначность декодирования достигается тем, что, во-первых, изначально кодер и декодер имеют идентичную и обычно очень простую модель и, во-вторых, модификация модели при сжатии и разжатии осуществляется одинаковым образом. Опыт использования моделей различных типов показывает, что адаптивное моделирование является не только элегантной техникой, но и обеспечивает, по крайней мере, не худшее сжатие, чем полуадаптивное моделирование. Понятно, что если стоит задача создания «универсального» компрессора для сжатия данных несходных типов, то адаптивный подход является естественным выбором разработчика.

Блочно-адаптивное моделирование можно рассматривать как частный случай адаптивной стратегии (или наоборот, что сути дела не меняет). В зависимости от конкретного алгоритма обновления модели, оценки вероятностей символов, метода статистического кодирования и самих данных изменение модели после обработки каждого символа может быть сопряжено со следующими проблемами:

- потеря устойчивости (робастности) оценок, если данные «зашумлены», или имеются значительные локальные изменения статистических взаимосвязей между символами обраба-

тываемого потока; иначе говоря, чересчур быстрая, «агрессивная» адаптация модели может приводить к ухудшению точности оценок;

- большие вычислительные расходы на обновление модели (как пример — в случае адаптивного кодирования по алгоритму Хаффмана);
- большие расходы памяти для хранения структур данных, обеспечивающих быструю модификацию модели.

Поэтому обновление модели может выполняться после обработки целого блока символов, в общем случае переменной длины. Для обеспечения правильности разжатия декодер должен выполнять такую же последовательность действий по обновлению модели, что и кодер, либо кодеру необходимо передавать вместе со сжатыми данными инструкции по модификации модели. Последний вариант достаточно часто используется при блочно-адаптивном моделировании для ускорения процесса декодирования в ущерб коэффициенту сжатия.

Контекстное моделирование

Необходимо решить задачу оценки вероятностей появления символов в каждой позиции обрабатываемой последовательности. Для того чтобы декомпрессия прошла без потерь, необходимо использовать только ту информацию, которая в полной мере известна как кодеру, так и декодеру. Обычно это означает, что оценка вероятности очередного символа должна зависеть только от свойств уже обработанного блока данных.

Наиболее простой способ оценки реализуется с помощью полуадаптивного моделирования и заключается в предварительном подсчете безусловной частоты появления символов в сжимаемом блоке. Полученное распределение вероятностей используется для статистического кодирования всех символов блока. Если, например, такую модель применить для сжатия текста на русском языке, то в среднем на кодирование каждого символа будет потрачено примерно 4.5 бита. Это значение является средней длиной кодов для модели, базирующейся на использовании безусловного распределения вероятностей букв в тексте. Заметим, что уже в этом простом случае достигается степень сжатия 1.5 по отношению к тривиальному кодированию, когда всем символам назначаются коды одинаковой длины. Действительно, размер алфавита русского текста превышает 64, но меньше 128 знаков (строчные и заглавные буквы, знаки препинания, пробел), что требует 7-битовых кодов.

Анализ распространенных типов данных (текстов на естественных языках), выявляет сильную зависимость вероятности появления символов от непосредственно им предшествующих. Иначе говоря, большая часть данных, с которыми мы сталкиваемся, порождается источниками с памятью. Допустим, нам известно, что сжимаемый блок является текстом на русском языке. Если, например, строка из трех только что обработанных символов равна “_цы” (подчеркиванием здесь и далее обозначается пробел), то текущий символ скорее всего входит в следующую группу: ‘г’ («цыган»), ‘к’ («цыкать»), ‘п’ («цыпочки»), ‘ц’ («цыц»). Или, в случае анализа сразу нескольких слов, если предыдущая строка равна “Вставай,_проклятьем_заклейменный,”, то продолжением явно будет “весь_мир_”. Следовательно, учет зависимости частоты появления символа (в общем случае — блока символов) от предыдущих должен давать более точные оценки и, в конечном счете, лучшее сжатие. Действительно, в случае посимвольного кодирования при использовании информации об одном непосредственно предшествующем символе достигается средняя длина кодов в 3.6 бита для русских текстов, при учете двух последних – уже порядка 3.2 бита. В первом случае моделируются условные распределения вероятностей символов, зависящие от значения строки из одного непосредственно предшествующего символа, во втором – зависящие от строки из двух предшествующих символов.

Улучшение сжатия при учете предыдущих элементов (пикселов, сэмплов, отсчетов, чисел) отмечается и при обработке данных других распространенных типов: объектных файлов, изображений, аудиозаписей, таблиц чисел.

Под контекстным моделированием понимается оценка вероятности появления символа (элемента, пиксела, сэмпла, отсчета и даже набора качественно разных объектов) в зависимости от непосредственно ему предшествующих, или контекста.

Заметим, что в быту понятие «контекст» обычно используется в глобальном значении – как совокупность символов (элементов), окружающих текущий обрабатываемый. Это контекст в широком смысле. Выделяют также «левосторонние» и «правосторонние» контексты, т.е. последовательности символов, непосредственно примыкающие к текущему символу слева и справа соответственно. Здесь и далее под контекстом будем понимать именно классический левосторонний: так, например, для последнего символа ‘о’ последовательности “…молоко…” контекстом является “…молок”.

Если длина контекста ограничена, то такой подход называется контекстным моделированием ограниченного порядка (finite-context modeling), при этом под порядком понимается максимальная длина используемых контекстов N. Например, при моделировании порядка 3 для последнего символа ‘о’ в последовательности “…молоко…” контекстом максимальной длины 3 является строка “лок”. При сжатии этого символа под «текущими контекстами» могут пониматься “лок”, “ок”, “к”, а также пустая строка “”. Все эти контексты длины от N до 0 назовем активными контекстами в том смысле, что при оценке символа может быть использована накопленная для них статистика.

Далее вместо «контекст длины o, $o \le N$» мы будем обычно говорить «контекст порядка o».

В силу объективных причин — ограниченность вычислительных ресурсов — техника контекстного моделирования именно ограниченного порядка получила наибольшее развитие и распространение, поэтому далее под контекстным моделированием будем понимать именно ее. Дальнейшее изложение также учитывает специфику того, что контекстное моделирование практически всегда применяется как адаптивное.

Оценки вероятностей при контекстном моделировании строятся на основании обычных счетчиков частот, связанных с текущим контекстом. Если мы обработали строку “абсабвбабс”, то для контекста “аб” счетчик символа ‘с’ равен двум (говорят, что символ ‘с’ *появился в контексте* “аб” два раза), символа ‘в’ — единице. На основании этой статистики можно утверждать, что вероятность появления ‘с’ после “аб” равна 2/3, а вероятность появления ‘в’– 1/3, т.е. оценки формируются на основе уже просмотренной части потока.

В общем случае для каждого контекста конечной длины $o \le N$, встречаемого в обрабатываемой последовательности, создается контекстная модель (КМ). Любая КМ включает в себя счетчики всех символов, встреченных в соответствующем ей контексте, т.е. сразу после строки контекста. После каждого появления какого-то символа s в рассматриваемом контексте производится увеличение значения счетчика символа s в соответствующей контексту КМ. Обычно счетчики инициализируются нулями. На практике счетчики обычно создаются по мере появления в заданном контексте новых символов, т.е. счетчиков ни разу не виденных в заданном контексте символов просто не существует.

Под порядком КМ будем понимать длину соответствующего ей контекста. Если порядок КМ равен o, то будем обозначать такую КМ как «КМ(o)».

Кроме обычных КМ, часто используют контекстную модель минус первого порядка КМ(–1), присваивающую одинаковую вероятность всем символам алфавита сжимаемого потока.

Понятно, что для нулевого и минус первого порядка контекстная модель одна, а КМ большего порядка может быть несколько, вплоть до q^N, где q – размер алфавита обрабатываемой последовательности. КМ(0) и КМ(–1) всегда активны.

Часто говорят о «родительских» и «дочерних» контекстах. Для контекста “к” дочерними являются “ок” и “лк”, поскольку они образованы сцеплением (конкатенацией) одного символа и контекста “к”. Аналогично, для контекста “лок” родительским является контекст

“ок”, а контекстами-предками — “ок”, “к”,. Очевидно, что «пустой» контекст “” является предком для всех. Аналогичные термины применяются для КМ, соответствующих контекстам.

Совокупность КМ образует модель источника данных. Под порядком модели понимается максимальный порядок используемых КМ.

Виды контекстного моделирования

Пример обработки строки “абсабвбабс” иллюстрирует сразу две проблемы контекстного моделирования:

- как выбирать подходящий контекст (или контексты) среди активных с целью получения более точной оценки, ведь текущий символ может лучше предсказываться не контекстом второго порядка “аб”, а контекстом первого порядка “б”;
- как оценивать вероятность символов, имеющих нулевую частоту (например, ‘г’).

Выше приведены цифры, в соответствии с которыми при увеличении длины используемого контекста сжатие данных улучшается. К сожалению, при кодировании блоков типичной длины — единицы мегабайтов и меньше — это справедливо только для небольших порядков модели, т.к. статистика для длинных контекстов медленно накапливается. При этом также следует учитывать, что большинство реальных данных характеризуется неоднородностью, нестабильностью силы и вида статистических взаимосвязей, поэтому «старая» статистика контекстно-зависимых частот появления символов малополезна или даже вредна. Поэтому модели, строящие оценку только на основании информации КМ максимального порядка N, обеспечивают сравнительно низкую точность предсказания. Кроме того, хранение модели большого порядка требует много памяти.

Если в модели используются для оценки только КМ(N), то иногда такой подход называют «чистым» (pure) контекстным моделированием порядка N. Из-за вышеуказанного недостатка «чистые» модели представляют обычно только научный интерес.

Действительно, реально используемые файлы обычно имеют сравнительно небольшой размер, поэтому для улучшения их сжатия необходимо учитывать оценки вероятностей, получаемые на основании статистики контекстов разных длин. Техника объединения оценок вероятностей, соответствующих отдельным активным контекстам, в одну оценку называется смешиванием (blending). Известно несколько способов выполнения смешивания.

Рассмотрим модель произвольного порядка N. Если $q(s_i|o)$ есть вероятность, присваиваемая в активной КМ(o) символу s_i алфавита сжимаемого потока, то смешанная вероятность $q(s_i)$ вычисляется в общем случае как

$$q(s_i) = \sum_{o=-1}^{N} w(o) q(s_i \mid o),$$

где: $w(o)$ – вес оценки КМ(o).

Оценка $q(s_i|o)$ обычно определяется через частоту символа s_i по тривиальной формуле

$$q(s_i \mid o) = \frac{f(s_i \mid o)}{f(o)},$$

где: $f(s_i|o)$ – частота появления символа s_i в соответствующем контексте порядка o; $f(o)$– общая частота появления соответствующего контекста порядка o в обработанной последовательности.

Заметим, что правильнее было бы писать не, скажем, $f(s_i|o)$, а $f(s_i|C_{j(o)})$, т.е. «частота появления символа s_i в КМ порядка o с номером $j(o)$», поскольку контекстных моделей порядка o может быть огромное количество. Но при сжатии каждого текущего символа мы рассматриваем только *одну* КМ для каждого порядка, т.к. контекст определяется *непосредственно* примыкающей слева к символу строкой определенной длины. Иначе говоря, для каждого символа мы имеем набор из N+1 активных контекстов длины от N до 0, каждому из которых

однозначно соответствует только одна КМ, если она вообще есть. Поэтому здесь и далее используется сокращенная запись.

Если вес $w(-1) > 0$, то это гарантирует успешность кодирования любого символа входного потока, т.к. наличие КМ(–1) позволяет всегда получать ненулевую оценку вероятности и, соответственно, код конечной длины.

Различают модели с полным смешиванием (fully blended), когда предсказание определяется статистикой КМ всех используемых порядков, и с частичным смешиванием (partially blended) – в противном случае.

Рассмотрим процесс оценки отмеченного символа ‘Л’, встретившегося в блоке “молочное_моЛоко”. Считаем, что модель работает на уровне символов.

Пусть мы используем контекстное моделирование порядка 2 и делаем полное смешивание оценок распределений вероятностей в КМ второго, первого и нулевого порядков с весами 0.6, 0.3 и 0.1. Считаем, что в начале кодирования в КМ(0) создаются счетчики для всех символов алфавита {‘м’, ‘о’, ‘л’, ‘ч’, ‘н’, ‘е’, ‘_’, ‘к’} и инициализируются единицей; счетчик символа после его обработки увеличивается на 1.

Для текущего символа ‘л’ имеются контексты “мо”, “о” и пустой (нулевого порядка). К данному моменту для них накоплена статистика, показанная в таблице 17.

Таблица 17

Накопленная статистика

Символы		‘м’	‘о’	‘л’	‘ч’	‘н’	‘е’	‘_’	‘к’
КМ порядка 0 (контекст “”)	Частоты	3	5	2	2	2	2	2	1
	Накопленные частоты	3	8	10	12	14	16	18	19
КМ порядка 1 (контекст “о”)	Частоты	-	-	1	1	-	1	-	-
	Накопленные частоты	-	-	1	2	-	3	-	-
КМ порядка 2 (“мо”)	Частоты	-	-	1	-	-	-	-	-
	Накопленные частоты	-	-	1	-	-	-	-	-

Тогда оценка вероятности для символа ‘л’ будет равна

$$q('л') = 0.1 \cdot \frac{2}{19} + 0.3 \cdot \frac{1}{3} + 0.6 \cdot \frac{1}{1} = 0{,}71.$$

В общем случае, для однозначного кодирования символа ‘л’ такую оценку необходимо проделать для всех символов алфавита. Действительно, с одной стороны, декодер не знает, чему равен текущий символ, с другой стороны, оценка вероятности не гарантирует уникальность кода, а лишь задает его длину. Поэтому статистическое кодирование выполняется на основании накопленной частоты. Например, если кодировать на основании статистики только нулевого порядка, то существует взаимно однозначное соответствие между накопленными частотами из диапазона (8,10] и символом ‘л’, что не имеет места в случае просто частоты (частоту 2 имеют еще 4 символа). Понятно, что аналогичные свойства остаются в силе и в случае оценок, получаемых частичным смешиванием.

Очевидно, что успех применения смешивания зависит от способа выбора весов $w(o)$. Простой путь состоит в использовании заданного набора фиксированных весов КМ разных порядков при каждой оценке. Естественно, альтернативой является адаптация весов по мере кодирования. Приспособление может заключаться в придании все большей значимости КМ все больших порядков или, скажем, попытке выбрать наилучшие веса на основании определенных статистических характеристик последнего обработанного блока данных. Но так ис-

торически сложилось, что реальное развитие получили методы неявного взвешивания. Это объясняется в первую очередь их меньшей вычислительной сложностью.

Техника неявного взвешивания связана с введением вспомогательного символа ухода (escape). Символ ухода является квазисимволом и не должен принадлежать алфавиту сжимаемой последовательности. Фактически, он используется для передачи декодеру указаний кодера. Идея заключается в том, что если используемая КМ не позволяет оценить текущий символ (его счетчик равен 0 в этой КМ), то на выход посылается закодированный символ ухода и производится попытка оценить текущий символ в другой КМ, которой соответствует контекст иной длины. Обычно попытка оценки начинается с КМ наибольшего порядка N, затем в определенной последовательности осуществляется переход к контекстным моделям меньших порядков.

Естественно, статистическое кодирование символа ухода выполняется на основании его вероятности, так называемой вероятности ухода. Очевидно, что символы ухода порождаются не источником данных, а моделью. Следовательно, их вероятность может зависеть от характеристик сжимаемых данных, свойств КМ, с которой производится уход, свойств КМ, на которую происходит уход, и т.д. Как можно оценить эту вероятность, имея в виду, что конечный критерий качества — улучшение сжатия? Вероятность ухода — это вероятность появления в контексте нового символа. Тогда, фактически, необходимо оценить правдоподобность наступления ни разу не происходившего события. Теоретического фундамента для решения этой проблемы, видимо, не существует, но за время развития техник контекстного моделирования было предложено несколько подходов, хорошо работающих в большинстве реальных ситуаций. Кроме того, эксперименты показывают, что модели с неявным взвешиванием устойчивы относительно используемого метода оценки вероятности ухода, т.е. выбор какого-то способа вычисления этой величины не влияет на коэффициент сжатия кардинальным образом.

Алгоритмы PPM

Техника контекстного моделирования Prediction by Partial Matching (предсказание по частичному совпадению), предложенная в 1984 году Клири (Cleary) и Уиттеном (Witten), является одним из самых известных подходов к сжатию качественных данных и уж точно самым популярным среди контекстных методов. Значимость подхода обусловлена и тем фактом, что алгоритмы, причисляемые к PPM, неизменно обеспечивают в среднем наилучшее сжатие при кодировании данных различных типов и служат стандартом, «точкой отсчета» при сравнении универсальных алгоритмов сжатия.

Перед собственно рассмотрением алгоритмов PPM необходимо сделать замечание о корректности используемой терминологии. На протяжении примерно 10 лет – с середины 1980-х годов до середины 1990-х – под PPM понималась группа методов с вполне определенными характеристиками. В последние годы, вероятно из-за резкого увеличения числа всевозможных гибридных схем и активного практического использования статистических моделей для сжатия, произошло смешение понятий, и термин «PPM» часто используется для обозначения контекстных методов вообще.

Ниже будет описан некий обобщенный алгоритм PPM, а затем особенности конкретных распространенных схем.

Как и в случае многих других контекстных методов, для каждого контекста, встречаемого в обрабатываемой последовательности, создается своя контекстная модель (КМ). При этом под контекстом понимается последовательность элементов одного типа – символов, пикселов, чисел, но не набор разнородных объектов. Далее вместо слова «элемент» мы будем использовать «символ». Каждая КМ включает в себя счетчики всех символов, встреченных в соответствующем контексте.

PPM относится к адаптивным методам моделирования. Исходно кодеру и декодеру поставлена в соответствие начальная модель источника данных. Будем считать, что она состоит

из КМ(–1), присваивающей одинаковую вероятность всем символам алфавита входной последовательности. После обработки текущего символа кодер и декодер изменяют свои модели одинаковым образом, в частности, наращивая величину оценки вероятности рассматриваемого символа. Следующий символ кодируется (декодируется) на основании новой, измененной модели, после чего модель снова модифицируется и т.д. На каждом шаге обеспечивается идентичность модели кодера и декодера за счет применения одинакового механизма ее обновления.

В PPM используется неявное взвешивание оценок. Попытка оценки символа начинается с КМ(N), где N является параметром алгоритма и называется порядком PPM-модели. В случае нулевой частоты символа в КМ текущего порядка осуществляется переход к КМ меньшего порядка за счет использования механизма уходов (escape strategy), рассмотренного в предыдущем пункте.

Фактически, вероятность ухода – это суммарная вероятность всех символов алфавита входного потока, еще ни разу не появлявшихся в контексте. Любая КМ должна давать отличную от нуля оценку вероятности ухода. Исключения из этого правила возможны только в тех случаях, когда значения всех счетчиков КМ для всех символов алфавита отличны от нуля, то есть любой символ может быть оценен в рассматриваемом контексте. Оценка вероятности ухода традиционно является одной из основных проблем алгоритмов с неявным взвешиванием, и она будет специально рассмотрена ниже в пункте «Оценка вероятности ухода».

Способ моделирования источника с помощью классических алгоритмов PPM опирается на следующие предположения о природе источника:

1. источник является Марковским с порядком N, т.е. вероятность генерации символа зависит от N предыдущих символов и только от них;
2. источник имеет такую дополнительную особенность, что чем ближе располагается один из символов контекста к текущему символу, тем больше корреляция между ними.

Таким образом, механизм уходов первоначально рассматривался лишь как вспомогательный прием, позволяющий решить проблему кодирования символов, ни разу не встречавшихся в контексте порядка N. В идеале, достигаемом после обработки достаточно длинного блока, никакого обращения к КМ порядка меньше N происходить не должно. Иначе говоря, причисление классических алгоритмов PPM к методам, производящим взвешивание, пусть и неявным образом, является не вполне корректным.

При сжатии очередного символа выполняются следующие действия.

Если символ s обрабатывается с использованием PPM-модели порядка N, то, как мы уже отмечали, в первую очередь рассматривается КМ(N). Если она оценивает вероятность s числом, не равным нулю, то сама и используется для кодирования s. Иначе выдается сигнал в виде символа ухода, и на основе меньшей по порядку КМ(N–1) производится очередная попытка оценить вероятность s. Кодирование происходит через уход к КМ меньших порядков до тех пор, пока s не будет оценен. КМ(–1) гарантирует, что это в конце концов произойдет. Таким образом, каждый символ кодируется серией кодов символа ухода, за которой следует код самого символа. Из этого следует, что вероятность ухода также можно рассматривать как вероятность перехода к контекстной модели меньшего порядка.

Если в процессе оценки обнаруживается, что текущий рассматриваемый контекст встречается в первый раз, то для него создается КМ.

При оценке вероятности символа в КМ порядка $o<N$ можно исключить из рассмотрения все символы, которые содержатся в КМ(o+1), поскольку ни один из них точно не является символом s. Для этого в текущей КМ(o) нужно замаскировать, т.е. временно установить в ноль, значения счетчиков всех символов, имеющихся в КМ(o+1). Такая техника называется методом исключения (exclusion).

После собственно кодирования символа обычно осуществляется обновление статистики всех КМ, использованных при оценке его вероятности, за исключением статической КМ(–1). Такой подход называется методом исключения при обновлении. Простейшим способом модификации является инкремент счетчиков символа в этих КМ. Подробнее о стратегиях обновления будет сказано в пункте «Обновление счетчиков символов».

Рассмотрим подробнее работу алгоритма PPM с помощью примера.

Имеется последовательность символов "абвавабввбббв" алфавита {‘а’, ‘б’, ‘в’, ‘г’}, которая уже была закодирована.

Пусть счетчик символа ухода равен 1 для всех КМ, при обновлении модели счетчики символов увеличиваются на 1 во всех активных КМ, применяется метод исключения, и максимальная длина контекста равна 3, т.е. *N*=3.

Первоначально модель состоит из КМ(–1), в которой счетчики всех четырех символов алфавита имеют значение 1. Состояние модели обработки последовательности "абвавабввбббв" представлено на рисунке 6, где прямоугольниками обозначены контекстные модели, при этом для каждой КМ указан курсивом контекст, а также встречавшиеся в контексте символы и их частоты.

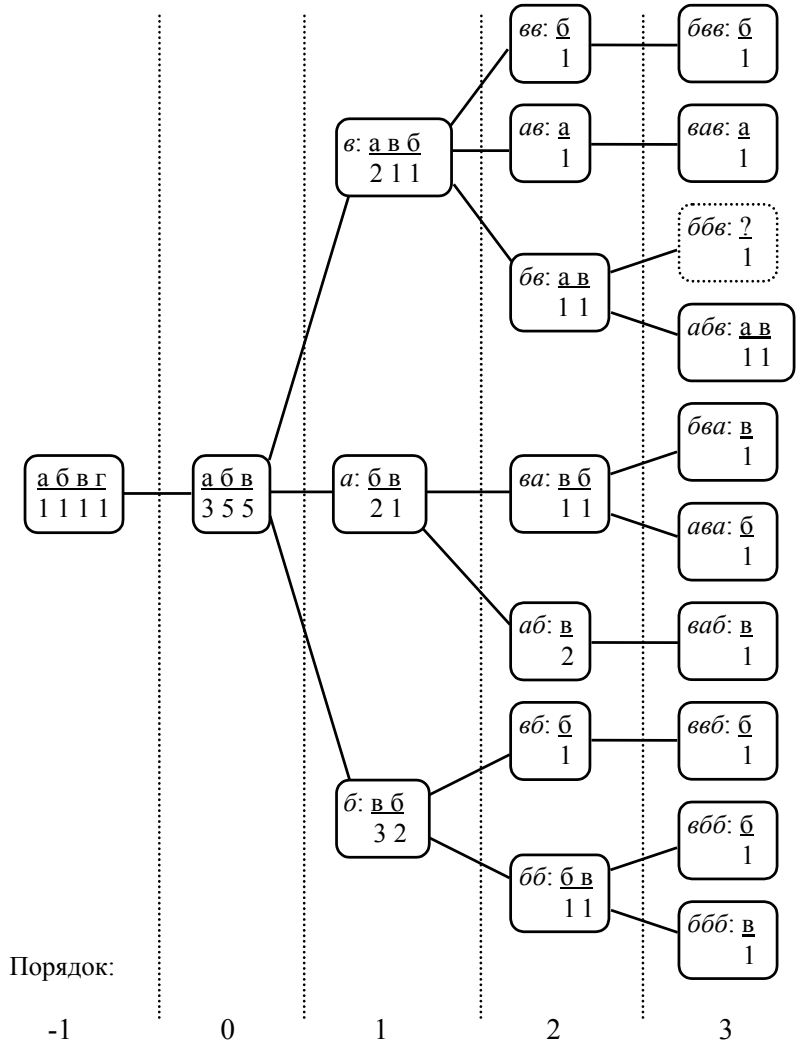


Рис. 6. Пример работы алгоритма PPM

Пусть текущий символ равен ‘г’, т.е. ‘?’ = ‘г’, тогда процесс его кодирования будет выглядеть следующим образом.

Сначала рассматривается контекст 3-го порядка “ббв”. Ранее он не встречался, поэтому кодер, ничего не послав на выход, переходит к анализу статистики для контекста 2-го поряд-

ка. В этом контексте (“бв”) встречались символ ‘а’ и символ ‘в’, счетчики которых в соответствующей КМ равны 1 каждый, поэтому символ ухода кодируется с вероятностью 1/(2+1), где в знаменателе число 2 – это наблюдавшаяся частота появления контекста “бв”, 1 – это значение счетчика символа ухода. В контексте 1-го порядка “в” дважды встречался символ ‘а’, который исключается (маскируется), один раз также исключаемый ‘в’ и один раз ‘б’, поэтому оценка вероятности ухода будет равна 1/(1+1). В КМ(0) символ ‘г’ также оценить нельзя, причем все имеющиеся в этой КМ символы ‘а’, ’б’, ‘в’ исключаются, так как уже встречались нам в КМ более высокого порядка. Поэтому вероятность ухода получается равной 1. Цикл оценивания завершается на уровне КМ(–1), где ‘г’ к этому времени остается единственным до сих пор не попадавшимся символом, поэтому он получает вероятность 1 и кодируется посредством 0 битов. Таким образом, при использовании хорошего статистического кодировщика для представления ‘г’ потребуется в целом примерно 2.6 бита.

Перед обработкой следующего символа создается КМ для строки “ббв” и производится модификация счетчиков символа ‘г’ в созданной и во всех просмотренных КМ. В данном случае требуется изменение КМ всех порядков от 0 до N.

В таблице 18 представлены оценки вероятностей, которые должны были быть использованы при кодировании символов алфавита {‘а’, ‘б’, ‘в’, ‘г’} в текущей позиции.

Таблица 18

Оценки вероятностей

Символ s	Последовательность оценок для КМ каждого порядка от 3 до –1					Общая оценка вероятности q(s)	Представление требует битов
	3	2	1	0	–1		
	“ббв”	“бв”	“в”	“”			
‘а’	-	$\frac{1}{2+1}$	-	-	-	$\frac{1}{3}$	1.6
‘б’	-	$\frac{1}{2+1}$	$\frac{1}{1+1}$	-	-	$\frac{1}{6}$	2.6
‘в’	-	$\frac{1}{2+1}$	-	-	-	$\frac{1}{3}$	1.6
‘г’	-	$\frac{1}{2+1}$	$\frac{1}{1+1}$	1	1	$\frac{1}{6}$	2.6

Алгоритм декодирования абсолютно симметричен алгоритму кодирования. После декодирования символа в текущей КМ проверяется, не является ли он символом ухода, если это так, то выполняется переход к КМ порядком ниже. Иначе считается, что исходный символ восстановлен, он записывается в декодированный поток и осуществляется переход к следующему шагу. Содержание процедур обновления счетчиков, создания новых контекстных моделей, прочих вспомогательных действий и последовательность их применения должны быть строго одинаковыми при кодировании и декодировании. Иначе возможна рассинхронизация копий модели кодера и декодера, что рано или поздно приведет к ошибочному декодированию какого-то символа. Начиная с этой позиции, вся оставшаяся часть сжатой последовательности будет разжата неправильно.

Разница между кодами символов, оценки, вероятности которых одинаковы, достигается за счет того, что PPM-предсказатель передает кодировщику так называемые накопленные частоты (или накопленные вероятности) оцениваемого символа и его соседей или кодовые пространства символов (таблица 19).

Хороший кодировщик должен отобразить символ s с оценкой вероятности $q(s)$ в код длины $\log_2 q(s)$, что и обеспечит сжатие всей обрабатываемой последовательности в целом.

Таблица 19

Накопленные частоты

Символ	Частота	Оценка вероятности	Накопленная вероятность (оценка)	Кодовое пространство
‘а’	1	$\frac{1}{3}$	$\frac{1}{3}$	[0 … 0.33)
‘б’	0	-	-	-
‘в’	1	$\frac{1}{3}$	$\frac{2}{3}$	[0.33 … 0.66)
‘г’	0	-	-	-
Уход	1	$\frac{1}{3}$	1	[0.66 … 1)

В обобщенном виде алгоритм кодирования можно записать так.

/инициализация контекста длины *N* (в смысле строки предыдущих символов), эта строка должна содержать *N* предыдущих символов, определяя набор активных контекстов длины $o \le N$/

```
context = "";
while ( ! DataFile.EOF() ){
c = DataFile.ReadSymbol(); // текущий символ
order = N;// текущий порядок КМ
success = 0;// успешность оценки в текущей КМ
do{
// найдем КМ для контекста текущей длины
CM = ContextModel.FindModel (context, order);
```

/попробуем найти текущий символ *c* в этой КМ, в CumFreq получим его накопленную частоту (или накопленную частоту символа ухода), в counter – ссылку на счетчик символа; флаг success указывает на отсутствие ухода/

```
success = CM.EvaluateSymbol (c, &CumFreq, counter);
```

/запомним в стеке КМ и указатель на счетчик для последующего обновления модели/

```
Stack.Push (CM, counter);
// закодируем c или символ ухода
StatCoder.Encode (CM, CumFreq, counter);
order--;
}while ( ! success );
```

/обновим модель: добавим КМ в случае необходимости, изменим значения счетчиков и т.д./

```
UpdateModel (Stack);
// обновим контекст: сдвинем влево, справа добавим c
MoveContext (c);
```

Рассмотрим основные моменты реализации компрессора PPM для простейшего случая с порядком модели $N = 1$ без исключения символов. Будем также исходить из того, что статистическое кодирование выполняется арифметическим кодером.

При контекстном моделировании 1-го порядка нам не требуются сложные структуры данных, обеспечивающие эффективное хранение и доступ к информации отдельных КМ. Можно просто хранить описания КМ в одномерном массиве, размер которого равен количеству символов в алфавите входной последовательности, и находить нужную КМ, используя

символ ее контекста как индекс. Мы используем байт-ориентированное моделирование, поэтому размер массива для контекстных моделей порядка 1 будет равен 256. Чтобы не плодить лишних сущностей, мы, во-первых, откажемся от КМ(–1) за счет соответствующей инициализации КМ(0), и, во-вторых, будем хранить КМ(0) в том же массиве, что и КМ(1). Считаем, что КМ(0) соответствует индекс 256.

В структуру контекстной модели ContextModel включим массив счетчиков count для всех возможных 256 символов. Для символа ухода введем в структуру КМ специальный счетчик esc, а также добавим поле TotFr, в котором будет содержаться сумма значений счетчиков всех обычных символов. Использование поля TotFr не обязательно, но позволит ускорить обработку данных.

С учетом сказанного структуры данных компрессора будут такими.

```
struct ContextModel{
        int     esc,
                        TotFr;
        int     count[256];
};
ContextModel cm[257];
```

Если размер типа int равен 4 байтам, то нам потребуется не менее 257 кбайт памяти для хранения модели.

Опишем стек, в котором будут храниться указатели на требующие модификации КМ, а также указатель стека SP и контекст context.

```
ContextModel *stack[2];
        int     SP,
context [1]; //контекст вырождается в 1 симвов
```

Больше никаких глобальных переменных и структур данных нам не нужно.

Инициализацию модели будем выполнять в общей для кодера и декодера функции init_model.

```
void init_model (void){
/*Так как cm является глобальной переменной, то значения всех полей равны 0. Нам требуется только распределить кодовое пространство в КМ(0) так, чтобы все символы, включая символ ухода, всегда бы имели ненулевые оценки. Пусть также символы будут равновероятными/
        for ( int j = 0; j < 256; j++ )
          cm[256].count[j] = 1  ;
          cm[256].TotFr = 256;
/Явно запишем, что в начале моделирования мы считаем контекст равным 0. Число не имеет значения, лишь бы кодер и декодер точно следовали принятым соглашениям. Обратите на это внимание/
        context [0] = 0;
SP = 0;
        }
```

Собственно кодер реализуем функцией encode. Эта функция управляет последовательностью действий при сжатии данных, вызывая вспомогательные процедуры в требуемом по-

рядке, а также находит нужную КМ. Оценка текущего символа производится в функции encode_sym, которая передает результаты своей работы арифметическому кодеру.

```
int encode_sym (ContextModel *CM, int c){
  // КМ потребует инкремента счетчиков, запомним ее
  stack [SP++] = CM;

  if (CM->count[c]){
  /счетчик сжимаемого символа не равен нулю, тогда его можно оценить в текущей КМ; найдем накопленную частоту предыдущего в массиве count символа/
        int CumFreqUnder = 0;
        for (int i = 0; i < c; i++)
              CumFreqUnder += CM->count[i];
  /передадим описание кодового пространства, занимаемого символом c, арифметическому кодеру/
        AC.encode (CumFreqUnder, CM->count[c],
                        CM->TotFr + CM->esc);
        return 1;       // возвращаемся в encode с победой
  }else{
  /нужно уходить на КМ(0);если текущий контекст 1-го порядка встретился первый раз, то заранее известно, что его КМ пуста (все счетчики равны нулю), и кодировать уход не только не имеет смысла, но и нельзя, т.к. TotFr+esc = 0/
        if (CM->esc)
              AC.encode (CM->TotFr, CM->esc, CM->TotFr + CM->esc)
;
        return 0; // закодировать символ не удалось
  }}
void encode (void){
  int     c,      // текущий символ
              success;        // успешность кодирования символа в КМ
  init_model ();
  AC.StartEncode ();          // проинициализируем арифм. кодер
  while (( c = DataFile.ReadSymbol() ) != EOF) {
        // попробуем закодировать в КМ(1)
        success = encode_sym (&cm[context[0]], c);
        if (!success)
/уходим на КМ(0), где любой символ получит ненулевую оценку и будет закодирован/
              encode_sym (&cm[256], c);
        update_model (c);
        context [0] = c; // сдвинем контекст
  }
  // закодируем знак конца файла символом ухода с КМ(0)
  AC.encode (cm[context[0]].TotFr, cm[context[0]].esc,
        cm[context[0]].TotFr + cm[context[0]].esc);
  AC.encode (cm[256].TotFr, cm[256].esc,
```

```
cm[256].TotFr + cm[256].esc);
  // завершим работу арифметического кодера
  AC.FinishEncode();}
```

Реализация декодера выглядит аналогично. Внимания заслуживает разве что только процедура поиска символа по описанию его кодового пространства. Метод get_freq арифметического кодера возвращает число *x*, лежащее в диапазоне [CumFreqUnder, CumFreqUnder+CM->count[i]), т.е. CumFreqUnder ≤ x < CumFreqUnder+CM->count[i]. Поэтому искомым символом является *i*, для которого выполнится это условие.

```
int decode_sym (ContextModel *CM, int *c){
  stack [SP++] = CM;
  if (!CM->esc) return 0;
  int cum_freq = AC.get_freq (CM->TotFr + CM->esc);
  if (cum_freq < CM->TotFr){
/символ был закодирован в этой КМ; найдем символ и его точное кодовое пространство/
        int CumFreqUnder = 0;
        int i = 0;
        for (;;){
              if ( (CumFreqUnder + CM->count[i]) <= cum_freq)
                    CumFreqUnder += CM->count[i];
              else break;
              i++;
        }
/обновим состояние арифметического кодера на основании точной накопленной частоты символа/
        AC.decode_update (CumFreqUnder, CM->count[i],
                                                CM->TotFr + CM->esc);
        *c = i;
        return 1;
  }else{
/обновим состояние арифметического кодера на основании точной накопленной частоты символа, оказавшегося символом ухода/
        AC.decode_update (CM->TotFr, CM->esc,
                    CM->TotFr + CM->esc);
        return 0;
  }
}
void decode (void){
  int     c,
    success;
  init_model ();
  AC.StartDecode ();
  for (;;){
```

```
            success = decode_sym (&cm[context[0]], &c);
            if (!success){
                  success = decode_sym (&cm[256], &c);
                  if (!success) break; //признак конца файла
            }
            update_model (c);
            context [0] = c;
            DataFile.WriteSymbol (c);
      }
}
```

Оценка вероятности ухода

На долю символов ухода обычно приходится порядка 30% и более от всех оценок, вычисляемых моделировщиком PPM. Это определило пристальное внимание к проблеме оценки вероятности символов с нулевой частотой. Львиная доля публикаций, посвященных PPM, прямо касаются оценки вероятности ухода (ОВУ).

Можно выделить два подхода к решению проблемы ОВУ: априорные методы, основанные на предположениях о природе сжимаемых данных, и адаптивные методы, которые пытаются приспособить оценку к данным. Понятно, что первые призваны обеспечить хороший коэффициент сжатия при обработке типичных данных в сочетании с высокой скоростью вычислений, а вторые ориентированы на обеспечение максимально возможной степени сжатия.

Введем обозначения:

C – общее число просмотров контекста, т.е. сколько раз он встретился в обработанном блоке данных;

S – количество разных символов в контексте;

$S^{(i)}$ – количество таких разных символов, что они встречались в контексте ровно i раз;

$E^{(x)}$ – значение ОВУ по методу x.

Изобретатели алгоритма PPM предложили два метода ОВУ: так называемые метод A и метод B. Использующие их алгоритмы PPM были названы PPMA и PPMB соответственно.

В дальнейшем было описано еще 5 априорных подходов к ОВУ: методы C, D, P, X и XC. По аналогии с PPMA и PPMB, алгоритмы PPM, применяющие методы C и D, получили названия PPMC и PPMD соответственно.

Идея методов и их сравнение представлены в таблицах 20 и 21.

Таблица 20

Оценки вероятности ухода

Метод	$E^{(x)} =$	Метод	$E^{(x)} =$
A	$\frac{1}{C+1}$	P	$\frac{S^{(1)}}{C} - \frac{S^{(2)}}{C^2} + \frac{S^{(3)}}{C^3} - \ldots$
B	$\frac{S - S^{(1)}}{C}$	X	$\frac{S^{(1)}}{C}$
C	$\frac{S}{C+S}$	XC	$\begin{cases} \frac{S^{(1)}}{C}, & при \quad 0 < S^{(1)} < C \\ E^{(C)}, & в\ противном\ случае \end{cases}$
D	$\frac{S}{2C}$		

При реализации метода B воздерживаются от оценки символов до тех пор, пока они не появятся в текущем контексте более одного раза. Это достигается за счет вычитания единицы из счетчиков. Методы P, X, XC базируются на предположении о том, что вероятность появления в обрабатываемых данных символа s_i подчиняется закону Пуассона с параметром λ_i.

Так, например, при сжатии текстов методы XC, D, P, X показывают весьма близкие результаты, и многое зависит от порядка модели и используемых для сравнения файлов. В большинстве случаев существенным является только отставание точности ОВУ по способам A и B от других методов.

Таблица 21

Сравнительный анализ алгоритмов

Тип файлов	Точность предсказания						
	Лучше			→			хуже
Тексты	XC	D	P	X	C	B	A
Двоичные файлы	C	X	P	XC	D	B	A

Чтобы улучшить оценку вероятности ухода, необходимо иметь такую модель оценки, которая бы адаптировалась к обрабатываемым данным. Подобный адаптивный механизм получил название Secondary Escape Estimation (SEE), т.е. «дополнительной оценки ухода», или «вторичной оценки ухода». Метод заключается в тривиальном вычислении вероятности ухода из текущей КМ через частоту появления новых символов (или, что то же, символов ухода) в контекстных моделях со схожими характеристиками:

$$E^{(SEE)}(i) = \frac{f_i(esc)}{n_i},$$

где: $f_i(esc)$– число наблюдавшихся уходов из контекстных моделей типа i; n_i – число просмотров контекстных моделей типа i.

Вразумительные обоснования выбора этих характеристик и критериев «схожести» при отсутствии априорных знаний о характере сжимаемой последовательности дать сложно, поэтому известные алгоритмы адаптивной оценки базируются на эмпирическом анализе типовых данных.

Одна из самых ранних попыток реализации SEE известна как метод Z, а использующая его разновидность алгоритма PPM – PPMZ. Для точности описания этой техники SEE объект «контекст» ниже будет также именоваться «PPM-контекстом».

Для нахождения ОВУ строятся так называемые контексты ухода (escape contexts) КУ, формируемые из четырех полей. В полях КУ содержится информация о значениях следующих величин: последние четыре символа PPM-контекста, порядок PPM-контекста, количество уходов и количество успешных оценок в соответствующей КМ. Нескольким КМ может соответствовать один КУ.

Информация о фактическом количестве уходов и успешных кодирований во всех контекстных моделях, имеющих общий КУ, запоминается в счетчиках контекстной модели уходов КМУ, построенной для данного КУ. Эта информация определяет ОВУ для текущей КМ. ОВУ находится путем взвешивания оценок, которые дают три КМУ (КМУ порядка 2, 1 и 0), отвечающие характеристикам текущей КМ.

КУ порядка 2 наиболее точно соответствует текущей КМ, контексты ухода порядком ниже формируются главным образом путем выбрасывания части информации из полей КУ порядка 2. Компоненты КУ порядка 2 определяются в соответствии с таблицей 21.

В состав КУ всех порядков входят поля 1, 2, 3. Для КУ порядка 1 поле 4 состоит из 8 битов и строится из шестых и пятых битов последних четырех обработанных символов. У КУ порядка 0 четвертое поле отсутствует. Очевидно, что алгоритм построения поля 4 для КУ порядков 1 и 2 призван улучшить предсказание ухода для текстов на английском языке в кодировке ASCII. Аналогичный прием, хотя и в не столь явном виде, используется в адаптивных методах ОВУ SEE-d1 и SEE-d2, рассмотренных ниже.

В состав КУ всех порядков входят поля 1, 2, 3. Для КУ порядка 1 поле 4 состоит из 8 битов и строится из шестых и пятых битов последних четырех обработанных символов. У КУ порядка 0 четвертое поле отсутствует. Очевидно, что алгоритм построения поля 4 для КУ порядков 1 и 2 призван улучшить предсказание ухода для текстов на английском языке в кодировке ASCII. Аналогичный прием, хотя и в не столь явном виде, используется в адаптивных методах ОВУ SEE-d1 и SEE-d2, рассмотренных ниже.

При взвешивании статистики КМУ(n) используются следующие веса w_n

$$\frac{1}{w_n} = e \cdot \log_2\left(\frac{1}{e}\right) + (1-e) \cdot \log_2\left(\frac{1}{1-e}\right),$$

Таблица 22

Контексты ухода

№ поля	Размер	Способ формирования значения поля	
		Параметр и его значения	Значение поля
1	2	Порядок КМ	порядок КМ / 2 (с округлением до младшего);
2	2	Количество уходов из КМ	
		1	0
		2	1
		3	2
		> 3	3
3	3	Количество успешных оценок в КМ	
		0	0
		1	1
		2	2
		3, 4	3
		5, 6	4
		7, 8, 9	5
		10, 11, 12	6
		> 12	7
4	9	X_1 = семь младших битов последнего (только что обработанного) символа PPM-контекста; X_2 = шестой и пятый биты предпоследнего символа; т.е., если расписать байт как совокупность 8 битов xXXxxxxx, то это биты XX	$((X_2\&0x60) << 2) \mid X_1$

где: e – ОВУ, которую дает данная взвешиваемая КМУ(n); формируется из фактического количества уходов и успешных кодирований в контекстных моделях, соответствующих этой КМУ, или, иначе, определяется наблюдавшейся частотой ухода из таких КМ.

Окончательная оценка:

$$E^{(Z)} = \frac{\sum_{n=0}^{2} e_n w_n}{\sum_{n=0}^{2} w_n}.$$

После ОВУ выполняется поиск текущего символа среди имеющихся в КМ. По результатам поиска (символ найден или нет) обновляются счетчики соответствующих трех КМУ порядка 0, 1 и 2.

АЛГОРИТМЫ СЖАТИЯ ИЗОБРАЖЕНИЙ БЕЗ ПОТЕРЬ

Алгоритмы сжатия изображений — бурно развивающаяся область машинной графики. Основной объект приложения усилий в ней — изображения — своеобразный тип данных, характеризуемый тремя особенностями:

1. Изображения (как и видео) *обычно требует для хранения гораздо большего объема памяти, чем текст*. Так, скромная, не очень качественная иллюстрация на обложке книги размером 500x800 точек, занимает 1.2 Мб – столько же, сколько художественная книга из 400 страниц (60 знаков в строке, 42 строки на странице). В качестве примера можно рассмотреть также, сколько тысяч страниц текста мы сможем поместить на CD-ROM, и как мало там

поместится несжатых фотографий высокого качества. Эта особенность изображений **определяет актуальность алгоритмов архивации** графики.

2. Второй особенностью изображений является то, что человеческое зрение при анализе изображения оперирует контурами, общим переходом цветов и сравнительно нечувствительно к малым изменениям в изображении. Таким образом, возможно создать эффективные алгоритмы архивации изображений, в которых декомпрессированное изображение не будет совпадать с оригиналом, однако человек этого не заметит. Данная особенность человеческого зрения позволила создать специальные алгоритмы сжатия, ориентированные только на изображения. Эти алгоритмы позволяют сжимать изображения с высокой степенью сжатия и незначительными с точки зрения человека потерями.

3. Легко заметить, что изображение, в отличие, например, от текста, обладает избыточностью в 2-х измерениях. Т.е. как правило, соседние точки, как по горизонтали, так и по вертикали, в изображении близки по цвету. Кроме того, мы можем использовать подобие между цветовыми плоскостями R, G и B в наших алгоритмах, что дает возможность создать еще более эффективные алгоритмы. Таким образом, при создании алгоритма компрессии графики используются особенности структуры изображения.

Всего на данный момент известно минимум три семейства алгоритмов, которые разработаны исключительно для сжатия изображений, и применяемые в них методы практически невозможно применить к архивации еще каких-либо видов данных.

Для того, чтобы говорить об алгоритмах сжатия изображений, мы должны определиться с несколькими важными вопросами:

- Какие критерии мы можем предложить для сравнения различных алгоритмов?
- Какие классы изображений существуют?
- Какие классы приложений, использующих алгоритмы компрессии графики, существуют, и какие требования они предъявляют к алгоритмам?

Классы изображений

Статические растровые изображения представляют собой двумерный массив чисел. Элементы этого массива называют пикселами (от английского «pixel»— «picture element»). Все изображения можно подразделить на две группы – с палитрой и без нее. У изображений с палитрой в пикселе хранится число – индекс в некотором одномерном векторе цветов, называемом палитрой. Чаще всего встречаются палитры из 16 и 256 цветов.

Изображения без палитры бывают в какой-либо системе цветопредставления и в градациях серого (grayscale). Для последних значение каждого пиксела интерпретируется как яркость соответствующей точки. Чаще всего встречаются изображения с 2, 16 и 256 уровнями серого. Одна из интересных практических задач заключается в приведении цветного или черно-белого изображения к двум градациям яркости, например, для печати на лазерном принтере. При использовании некой системы цветопредставления каждый пиксел представляет собой запись (структуру), полями которой являются компоненты цвета. Самой распространенной является система RGB, в которой цвет передается значениями интенсивности красной (R), зеленой (G) и синей (B) компонент. Существуют и другие системы цветопредставления, такие, как CMYK, CIE XYZccir60-1, YVU, YCrCb и т.п.

Для того, чтобы корректнее оценивать степень сжатия, нужно ввести понятие класса изображений. Под классом будет пониматься совокупность изображений, применение к которым алгоритма архивации дает качественно одинаковые результаты. Например, для одного класса алгоритм дает очень высокую степень сжатия, для другого – почти не сжимает, для третьего – увеличивает файл в размере. Например, все алгоритмы сжатия без потерь в худшем случае увеличивают файл.

Дадим неформальное определение наиболее распространенных классов изображений:

Класс 1. Изображения с небольшим количеством цветов (4-16) и большими областями, заполненными одним цветом. Плавные переходы цветов отсутствуют. Примеры: деловая графика – гистограммы, диаграммы, графики и т.п.

Класс 2. Изображения с плавными переходами цветов, построенные на компьютере. Примеры: графика презентаций, эскизные модели в САПР, изображения, построенные по методу Гуро.

Класс 3. Фотореалистичные изображения. Пример: отсканированные фотографии.

Класс 4. Фотореалистичные изображения с наложением деловой графики. Пример: реклама.

Развивая данную классификацию, в качестве отдельных классов могут быть предложены некачественно отсканированные в 256 градаций серого цвета страницы книг или растровые изображения топографических карт. (Заметим, что этот класс не тождественен классу 4). Формально являясь 8- или 24-битными, они несут не растровую, а чисто векторную информацию. Отдельные классы могут образовывать и совсем специфичные изображения: рентгеновские снимки или фотографии в профиль и фас из электронного досье.

Достаточно сложной и интересной задачей является поиск наилучшего алгоритма для конкретного класса изображений.

Нет смысла говорить о том, что какой-то алгоритм сжатия лучше другого, если мы не обозначили классы изображений, на которых сравниваются наши алгоритмы.

Классы приложений

Рассмотрим следующую простую классификацию приложений, использующих алгоритмы компрессии:

Класс 1. Характеризуются высокими требованиями ко времени архивации и разархивации. Нередко требуется просмотр уменьшенной копии изображения и поиск в базе данных изображений.

Примеры*:* Издательские системы в широком смысле этого слова, причем как готовящие качественные публикации (журналы) с заведомо высоким качеством изображений и использованием алгоритмов архивации без потерь, так и готовящие газеты, и WWW-сервера где есть возможность оперировать изображениями меньшего качества и использовать алгоритмы сжатия с потерями. В подобных системах приходится иметь дело с полноцветными изображениями самого разного размера (от 640x480, до 3000x2000) и с большими двуцветными изображениями. Поскольку иллюстрации занимают львиную долю от общего объема материала в документе, проблема хранения стоит очень остро. Проблемы также создает большая разнородность иллюстраций (приходится использовать универсальные алгоритмы). Единственное, что можно сказать заранее, это то, что будут преобладать фотореалистичные изображения и деловая графика.

Класс 2. Характеризуется высокими требованиями к степени архивации и времени разархивации. Время архивации роли не играет. Иногда подобные приложения также требуют от алгоритма компрессии легкости масштабирования изображения под конкретное разрешение монитора у пользователя.

Пример*:* Справочники и энциклопедии на CD-ROM. С появлением большого количества компьютеров, оснащенных этим приводом (в США уровень в 50% машин достигнут еще в 1995), достаточно быстро сформировался рынок программ, выпускаемых на лазерных дисках. Несмотря на то, что емкость одного диска довольно велика (примерно 650 Мб), ее, как правило, не хватает. При создании энциклопедий и игр большую часть диска занимают статические изображения и видео. Таким образом, для этого класса приложений актуальность приобретают существенно асимметричные по времени алгоритмы (*симметричность по времени* – отношение времени компрессии ко времени декомпрессии).

Класс 3. Характеризуется очень высокими требованиями к степени архивации. Приложение клиента получает от сервера информацию по сети.

Пример*:* Новая быстро развивающаяся система «Всемирная информационная паутина» – WWW. В этой гипертекстовой системе достаточно активно используются иллюстрации. При оформлении информационных или рекламных страниц хочется сделать их более яркими и красочными, что естественно сказывается на размере изображений. Больше всего при этом страдают пользователи, подключенные к сети с помощью медленных каналов связи. Если страница WWW перенасыщена графикой, то ожидание ее полного появления на экране может затянуться. Поскольку при этом нагрузка на процессор мала, то здесь могут найти применение эффективно сжимающие сложные алгоритмы со сравнительно большим временем разархивации. Кроме того, мы можем видоизменить алгоритм и формат данных так, чтобы просматривать огрубленное изображение файла до его полного получения.

Можно привести множество более узких классов приложений. Так, свое применение машинная графика находит и в различных информационных системах. Например, уже становится привычным исследовать ультразвуковые и рентгеновские снимки не на бумаге, а на экране монитора. Постепенно в электронный вид переводят и истории болезней. Понятно, что хранить эти материалы логичнее в единой картотеке. При этом без использования специальных алгоритмов большую часть архивов займут фотографии. Поэтому при создании эффективных алгоритмов решения этой задачи нужно учесть специфику рентгеновских снимков – преобладание размытых участков.

В геоинформационных системах – при хранении аэрофотоснимков местности – специфическими проблемами являются большой размер изображения и необходимость выборки лишь части изображения по требованию. Кроме того, может потребоваться масштабирование. Это неизбежно накладывает свои ограничения на алгоритм компрессии.

В электронных картотеках и досье различных служб для изображений характерно подобие между фотографиями в профиль и подобие между фотографиями в фас, которое также необходимо учитывать при создании алгоритма архивации. Подобие между фотографиями наблюдается и в любых других специализированных справочниках. В качестве примера можно привести энциклопедии птиц или цветов.

Нет смысла говорить о том, что какой-то конкретный алгоритм компрессии лучше другого, если мы не обозначили **класс приложений**, относительно которого мы эти алгоритмы собираемся сравнивать.

Требования приложений к алгоритмам компрессии

В предыдущем разделе мы определили, какие приложения являются потребителями алгоритмов архивации изображений. Однако заметим, что приложение определяет характер использования изображений (либо большое количество изображений хранится и используется, либо изображения скачиваются по сети, либо изображения велики по размерам, и нам необходима возможность получения лишь части). Характер использования изображений задает степень важности следующих ниже противоречивых требований к алгоритму:

1. Высокая степень компрессии. Заметим, что далеко не для всех приложений актуальна высокая степень компрессии. Кроме того, некоторые алгоритмы дают лучшее соотношение качества к размеру файла при высоких степенях компрессии, однако проигрывают другим алгоритмам при низких степенях.

2. Высокое качество изображений**.** Выполнение этого требования напрямую противоречит выполнению предыдущего.

3. Высокая скорость компрессии. Это требование для некоторых алгоритмов с потерей информации является взаимоисключающим с первыми двумя. Интуитивно понятно, что чем больше времени мы будем анализировать изображение, пытаясь получить наивысшую степень компрессии, тем лучше будет результат. И, соответственно, чем меньше мы времени

потратим на компрессию (анализ), тем ниже будет качество изображения и больше его размер.

4. Высокая скорость декомпрессии. Достаточно универсальное требование, актуальное для многих приложений. Однако можно привести примеры приложений, где время декомпрессии далеко не критично.

5. Масштабирование изображений. Данное требование подразумевает легкость изменения размеров изображения до размеров окна активного приложения. Дело в том, что одни алгоритмы позволяют легко масштабировать изображение прямо во время декомпрессии, в то время как другие не только не позволяют легко масштабировать, но и увеличивают вероятность появления неприятных артефактов после применения стандартных алгоритмов масштабирования к декомпрессированному изображению. Например, можно привести пример «плохого» изображения для алгоритма JPEG – это изображение с достаточно мелким регулярным рисунком (пиджак в мелкую клетку). Характер вносимых алгоритмом JPEG искажений таков, что уменьшение или увеличение изображения может дать неприятные эффекты.

6. Возможность показать огрубленное изображение (низкого разрешения)**,** использовав только начало файла. Данная возможность актуальна для различного рода сетевых приложений, где перекачивание изображений может занять достаточно большое время, и желательно, получив начало файла, корректно показать preview. Заметим, что примитивная реализация указанного требования путем записывания в начало изображения его уменьшенной копии заметно ухудшит степень компрессии.

7. Устойчивость к ошибкам**.** Данное требование означает локальность нарушений в изображении при порче или потере фрагмента передаваемого файла. Данная возможность используется при широковещании (broadcasting – передача по многим адресам) изображений по сети, то есть в тех случаях, когда невозможно использовать протокол передачи, повторно запрашивающий данные у сервера при ошибках. Например, если передается видеоряд, то было бы неправильно использовать алгоритм, у которого сбой приводил бы к прекращению правильного показа всех последующих кадров. Данное требование противоречит высокой степени архивации, поскольку интуитивно понятно, что мы должны вводить в поток избыточную информацию. Однако для разных алгоритмов объем этой избыточной информации может существенно отличаться.

8. Учет специфики изображения. Более высокая степень архивации для класса изображений, которые статистически чаще будут применяться в нашем приложении. В предыдущих разделах это требование уже обсуждалось.

9. Редактируемость. Под редактируемостью понимается минимальная степень ухудшения качества изображения при его повторном сохранении после редактирования. Многие алгоритмы с потерей информации могут существенно испортить изображение за несколько итераций редактирования.

10. Небольшая стоимость аппаратной реализации. Эффективность программной реализации. Данные требования к алгоритму реально предъявляют не только производители игровых приставок, но и производители многих информационных систем. Так, декомпрессор фрактального алгоритма очень эффективно и коротко реализуется с использованием технологии MMX и распараллеливания вычислений, а сжатие по стандарту CCITT Group 3 легко реализуется аппаратно.

Очевидно, что для конкретной задачи нам будут очень важны одни требования и менее важны (и даже абсолютно безразличны) другие.

На практике для каждой задачи мы можем сформулировать набор приоритетов из требований, изложенных выше, который и определит наиболее подходящий в наших условиях алгоритм (либо набор алгоритмов) для ее решения.

Критерии сравнения алгоритмов

Заметим, что характеристики алгоритма относительно некоторых требований приложений, сформулированные выше, зависят от конкретных условий, в которые будет поставлен алгоритм. Так, степень компрессии зависит от того, на каком классе изображений алгоритм тестируется. Аналогично, скорость компрессии нередко зависит от того, на какой платформе реализован алгоритм. Преимущество одному алгоритму перед другим может дать, например, возможность использования в вычислениях алгоритма технологий нижнего уровня, типа MMX, а это возможно далеко не для всех алгоритмов. Так, JPEG существенно выигрывает от применения технологии MMX, а LZW нет. Кроме того, нам придется учитывать, что некоторые алгоритмы распараллеливаются легко, а некоторые нет.

Таким образом, невозможно составить универсальное сравнительное описание известных алгоритмов. Это можно сделать только для типовых классов приложений при условии использования типовых алгоритмов на типовых платформах. Однако такие данные необычайно быстро устаревают.

Так, например, еще в 1994, интерес к показу огрубленного изображения, используя только начало файла (требование 6), был чисто абстрактным. Реально эта возможность практически нигде не требовалась и класс приложений, использующих данную технологию, был крайне невелик. С взрывным распространением Internet, который характеризуется передачей изображений по сравнительно медленным каналам связи, использование Interlaced GIF (алгоритм LZW) и Progressive JPEG (вариант алгоритма JPEG), реализующих эту возможность, резко возросло. То, что новый алгоритм (например, wavelet) поддерживает такую возможность, существеннейший плюс для него сегодня.

В то же время мы можем рассмотреть такое редкое на сегодня требование, как устойчивость к ошибкам. Можно предположить, что в скором времени (через 5-10 лет) с распространением широковещания в сети Internet для его обеспечения будут использоваться именно алгоритмы, устойчивые к ошибкам, даже не рассматриваемые в сегодняшних статьях и обзорах.

Со всеми сделанными выше оговорками, выделим несколько наиболее важных для нас критериев сравнения алгоритмов компрессии, которые и будем использовать в дальнейшем. Как легко заметить, мы будем обсуждать меньше критериев, чем было сформулировано выше. Это позволит избежать лишних деталей при кратком изложении данного материала.

1. Худшая, средняя и лучшая степень сжатия. То есть доля, на которую возрастет изображение, если исходные данные будут наихудшими; некая среднестатистическая степень для того класса изображений, *на который ориентирован алгоритм;* и, наконец, лучшая степень. Последняя необходима лишь теоретически, поскольку показывает степень сжатия наилучшего (как правило, абсолютно черного) изображения, иногда фиксированного размера.

2. Класс изображений, на который ориентирован алгоритм. Иногда указано также, почему на других классах изображений получаются худшие результаты.

3. Симметричность. Отношение характеристики алгоритма кодирования к аналогичной характеристике при декодировании. Характеризует ресурсоемкость процессов кодирования и декодирования. Для нас наиболее важной является симметричность по времени: отношение времени кодирования ко времени декодирования. Иногда нам потребуется симметричность по памяти.

4. Есть ли потери качества? И если есть, то за счет чего изменяется степень сжатия? Дело в том, что у большинства алгоритмов сжатия с потерей информации существует возможность изменения степени сжатия.

5. Характерные особенности алгоритма и изображений, к которым его применяют. Здесь могут указываться наиболее важные для алгоритма свойства, которые могут стать определяющими при его выборе.

Используя данные критерии, приступим к рассмотрению алгоритмов архивации изображений.

Прежде, чем непосредственно начать разговор об алгоритмах, хотелось бы сделать оговорку. Один и тот же алгоритм часто можно реализовать разными способами. Многие известные алгоритмы, такие как RLE, LZW или JPEG, имеют десятки различающихся реализаций. Кроме того, у алгоритмов бывает несколько явных параметров, варьируя которые, можно изменять характеристики процессов архивации и разархивации. (См. примеры в разделе о форматах). При конкретной реализации эти параметры фиксируются, исходя из наиболее вероятных характеристик входных изображений, требований на экономию памяти, требований на время архивации и т.д. Поэтому у алгоритмов одного семейства лучшая и худшая степени сжатия могут отличаться, но качественно картина не изменится.

Алгоритмы изображений архивации без потерь

Алгоритм RLE

Данный алгоритм необычайно прост в реализации. Групповое кодирование – от английского Run Length Encoding (RLE) – один из самых старых и самых простых алгоритмов архивации графики. Изображение в нем вытягивается в цепочку байт по строкам растра. Само сжатие в RLE происходит за счет того, что в исходном изображении встречаются цепочки одинаковых байт. Замена их на пары <счетчик повторений, значение> уменьшает избыточность данных.

Алгоритм декомпрессии при этом выглядит так:

```
Initialization(...);
do {
  byte = ImageFile.ReadNextByte();
  if(является счетчиком(byte)) {
          counter = Low6bits(byte)+1;
          value = ImageFile.ReadNextByte();
          for(i=1 to counter)
                  DecompressedFile.WriteByte(value)
  }
  else {
  DecompressedFile.WriteByte(byte)
} while(!ImageFile.EOF());
```

В данном алгоритме признаком счетчика (counter) служат единицы в двух верхних битах считанного файла. Соответственно оставшиеся 6 бит расходуются на счетчик, который может принимать значения от 1 до 64. Строку из 64 повторяющихся байтов мы превращаем в два байта, т.е. сожмем в 32 раза.

Алгоритм рассчитан на деловую графику – изображения с большими областями повторяющегося цвета. Ситуация, когда файл увеличивается, для этого простого алгоритма не так уж редка. Ее можно легко получить, применяя групповое кодирование к обработанным цветным фотографиям. Для того, чтобы увеличить изображение в два раза, его надо применить к изображению, в котором значения всех пикселов больше 11000000_2 и подряд попарно не повторяются.

Второй вариант этого алгоритма имеет большую максимальную степень сжатия и меньше увеличивает в размерах исходный файл.

Алгоритм декомпрессии для него выглядит так:

```
Initialization(...);
do {
```

```
byte = ImageFile.ReadNextByte();
counter = Low7bits(byte)+1;
if(если признак повтора(byte)) {
        value = ImageFile.ReadNextByte();
        for (i=1 to counter)
                CompressedFile.WriteByte(value)
}
else {
        for(i=1 to counter){
                value = ImageFile.ReadNextByte();
                CompressedFile.WriteByte(value)
        }
} while(!ImageFile.EOF());
```

Признаком повтора в данном алгоритме является единица в старшем разряде соответствующего байта. Как можно легко подсчитать, в лучшем случае этот алгоритм сжимает файл в 64 раза (а не в 32 раза, как в предыдущем варианте), в худшем увеличивает на 1/128. Средние показатели степени компрессии данного алгоритма находятся на уровне показателей первого варианта. Похожие схемы компрессии использованы в качестве одного из алгоритмов, поддерживаемых форматом TIFF.

Алгоритм LZ

Существует довольно большое семейство LZ-подобных алгоритмов, различающихся, например, методом поиска повторяющихся цепочек. Один из достаточно простых вариантов этого алгоритма, например, предполагает, что во входном потоке идет либо пара <счетчик, смещение относительно текущей позиции>, либо просто <счетчик> «пропускаемых» байт и сами значения байтов (как во втором варианте алгоритма RLE). При разархивации для пары <счетчик, смещение> копируются <счетчик> байт из выходного массива, полученного в результате разархивации, на <смещение> байт раньше, а <счетчик> (т.е. число равное счетчику) значений «пропускаемых» байт просто копируются в выходной массив из входного потока. Данный алгоритм является несимметричным по времени, поскольку требует полного перебора буфера при поиске одинаковых подстрок. В результате нам сложно задать большой буфер из-за резкого возрастания времени компрессии. Однако потенциально построение алгоритма, в котором на <счетчик> и на <смещение> будет выделено по 2 байта (старший бит старшего байта счетчика – признак повтора строки / копирования потока), даст нам возможность сжимать все повторяющиеся подстроки размером до 32Кб в буфере размером 64Кб.

При этом мы получим увеличение размера файла в худшем случае на 32770/32768 (в двух байтах записано, что нужно переписать в выходной поток следующие 2^{15} байт), что совсем неплохо. Максимальная степень сжатия составит в пределе 8192 раза. В пределе, поскольку максимальное сжатие мы получаем, превращая 32Кб буфера в 4 байта, а буфер такого размера мы накопим не сразу. Однако, минимальная подстрока, для которой нам выгодно проводить сжатие, должна состоять в общем случае минимум из 5 байт, что и определяет малую ценность данного алгоритма. К достоинствам LZ можно отнести чрезвычайную простоту алгоритма декомпрессии.

Алгоритм LZW

Рассматриваемый нами ниже вариант алгоритма будет использовать дерево для представления и хранения цепочек. Очевидно, что это достаточно сильное ограничение на вид цепочек, и далеко не все одинаковые подцепочки в нашем изображении будут использованы при сжатии. Однако в предлагаемом алгоритме выгодно сжимать даже цепочки, состоящие из 2 байт.

Процесс сжатия выглядит достаточно просто. Мы считываем последовательно символы входного потока и проверяем, есть ли в созданной нами таблице строк такая строка. Если строка есть, то мы считываем следующий символ, а если строки нет, то мы заносим в поток код для предыдущей найденной строки, заносим строку в таблицу и начинаем поиск снова.

Функция InitTable() очищает таблицу и помещает в нее все строки единичной длины.

```
InitTable();
CompressedFile.WriteCode(ClearCode);
CurStr=пустая строка;
while(не ImageFile.EOF()){    //Пока не конец файла
  C=ImageFile.ReadNextByte();
  if(CurStr+C есть в таблице)
        CurStr=CurStr+C;//Приклеить символ к строке
  else {
        code=CodeForString(CurStr);//code-не байт!
        CompressedFile.WriteCode(code);
        AddStringToTable (CurStr+C);
        CurStr=C;     // Строка из одного символа
  }
}
code=CodeForString(CurStr);
CompressedFile.WriteCode(code);
CompressedFile.WriteCode(CodeEndOfInformation);
```

Как говорилось выше, функция InitTable() инициализирует таблицу строк так, чтобы она содержала все возможные строки, состоящие из одного символа. Например, если мы сжимаем байтовые данные, то таких строк в таблице будет 256 («0», «1», ... , «255»). Для кода очистки (ClearCode) и кода конца информации (CodeEndOfInformation) зарезервированы значения 256 и 257. В рассматриваемом варианте алгоритма используется 12-битный код, и, соответственно, под коды для строк нам остаются значения от 258 до 4095. Добавляемые строки записываются в таблицу последовательно, при этом индекс строки в таблице становится ее кодом.

Функция ReadNextByte() читает символ из файла. Функция WriteCode() записывает код (не равный по размеру байту) в выходной файл. Функция AddStringToTable() добавляет новую строку в таблицу, приписывая ей код. Кроме того, в данной функции происходит обработка ситуации переполнения таблицы. В этом случае в поток записывается код предыдущей найденной строки и код очистки, после чего таблица очищается функцией InitTable(). Функция CodeForString() находит строку в таблице и выдает код этой строки.

Пусть мы сжимаем последовательность 45, 55, 55, 151, 55, 55, 55. Тогда, согласно изложенному выше алгоритму, мы поместим в выходной поток сначала код очистки <256>, потом добавим к изначально пустой строке «45» и проверим, есть ли строка «45» в таблице. Поскольку мы при инициализации занесли в таблицу все строки из одного символа, то строка «45» есть в таблице. Далее мы читаем следующий символ 55 из входного потока и проверяем, есть ли строка «45, 55» в таблице. Такой строки в таблице пока нет. Мы заносим в таблицу строку «45, 55» (с первым свободным кодом 258) и записываем в поток код <45>. Можно коротко представить архивацию так:

«45» – есть в таблице;

«45, 55» – нет. Добавляем в таблицу <258>«45, 55». В поток: <45>;

«55, 55» – нет. В таблицу: <259>«55, 55». В поток: <55>;

«55, 151» – нет. В таблицу: <260>«55, 151». В поток: <55>;

«151, 55» – нет. В таблицу: <261>«151, 55». В поток: <151>;

«55, 55» – есть в таблице;

«55, 55, 55» – нет. В таблицу: «55, 55, 55» <262>. В поток: <259>;

Последовательность кодов для данного примера, попадающих в выходной поток: <256>, <45>, <55>, <55>, <151>, <259>.

Особенность LZW заключается в том, что для декомпрессии нам не надо сохранять таблицу строк в файл для распаковки. Алгоритм построен таким образом, что мы в состоянии восстановить таблицу строк, пользуясь только потоком кодов.

Мы знаем, что для каждого кода надо добавлять в таблицу строку, состоящую из уже присутствующей там строки и символа, с которого начинается следующая строка в потоке.

Алгоритм декомпрессии, осуществляющий эту операцию, выглядит следующим образом:

```
code=File.ReadCode();
while(code != CodeEndOfInformation){
 if(code = ClearCode) {
      InitTable();
      code=File.ReadCode();
         if(code = CodeEndOfInformation)
                {закончить работу};
      ImageFile.WriteString(StrFromTable(code));
      old_code=code;
 }
 else {
      if(InTable(code)) {
          ImageFile.WriteString(FromTable(code));
          AddStringToTable(StrFromTable(old_code)+
                FirstChar(StrFromTable(code)));
          old_code=code;
      }
      else {
          OutString= StrFromTable(old_code)+
            FirstChar(StrFromTable(old_code));
          ImageFile.WriteString(OutString);
          AddStringToTable(OutString);
          old_code=code;
      }
 }
}
```

Здесь функция ReadCode() читает очередной код из декомпрессируемого файла. Функция InitTable() выполняет те же действия, что и при компрессии, т.е. очищает таблицу и заносит в нее все строки из одного символа. Функция FirstChar() выдает нам первый символ строки. Функция StrFromTable() выдает строку из таблицы по коду. Функция

AddStringToTable() добавляет новую строку в таблицу (присваивая ей первый свободный код). Функция WriteString() записывает строку в файл.

Подсчитаем лучшую и худшую степень сжатия для данного алгоритма. Лучшее сжатие, очевидно, будет получено для цепочки одинаковых байт большой длины (т.е. для 8-битного изображения, все точки которого имеют, для определенности, цвет 0). При этом в 258 строку таблицы мы запишем строку «0, 0», в 259 — «0, 0, 0», ... в 4095 — строку из 3839 (=4095-256) нулей. При этом в поток попадет (проверьте по алгоритму!) 3840 кодов, включая код очистки. Следовательно, посчитав сумму арифметической прогрессии от 2 до 3839 (т.е. длину сжатой цепочки) и поделив ее на 3840*12/8 (в поток записываются 12-битные коды), мы получим лучшую степень сжатия.

Худшее сжатие будет получено, если мы ни разу не встретим подстроку, которая уже есть в таблице (в ней не должно встретиться ни одной одинаковой пары символов).

В случае, если мы постоянно будем встречать новую подстроку, мы запишем в выходной поток 3840 кодов, которым будет соответствовать строка из 3838 символов. Без учета замечания 1 это составит увеличение файла почти в 1.5 раза.

Алгоритм Хаффмана с фиксированной таблицей CCITT Group 3

Близкая модификация алгоритма Хаффмана используется при сжатии черно-белых изображений (один бит на пиксел). Полное название данного алгоритма CCITT Group 3. Это означает, что данный алгоритм был предложен третьей группой по стандартизации Международного Консультационного Комитета по Телеграфии и Телефонии (Consultative Committee International Telegraph and Telephone). Последовательности подряд идущих черных и белых точек в нем заменяются числом, равным их количеству. А этот ряд, уже в свою очередь, сжимается по Хаффману с фиксированной таблицей.

Набор идущих подряд точек изображения одного цвета называется *серией*. Длина этого набора точек называется длиной серии.

В таблице 21, приведенной ниже, заданы два вида кодов:

Коды завершения серий – заданы с 0 до 63 с шагом 1.

Составные (дополнительные) коды – заданы с 64 до 2560 с шагом 64.

Каждая строка изображения сжимается независимо. Мы считаем, что в нашем изображении существенно преобладает белый цвет, и все строки изображения начинаются с белой точки. Если строка начинается с черной точки, то мы считаем, что строка начинается белой серией с длиной 0. Например, последовательность длин серий 0, 3, 556, 10, ... означает, что в этой строке изображения идут сначала 3 черных точки, затем 556 белых, затем 10 черных и т.д.

На практике в тех случаях, когда в изображении преобладает черный цвет, мы инвертируем изображение перед компрессией и записываем информацию об этом в заголовок файла.

Алгоритм компрессии выглядит так:

```
for(по всем строкам изображения) {
  Преобразуем строку в набор длин серий;
  for(по всем сериям) {
        if(серия белая) {
              L= длина серии;
              while(L > 2623) { // 2623=2560+63
                    L=L-2560;
                    ЗаписатьБелыйКодДля(2560);
              }
              if(L > 63) {
```

```
                L2=МаксимальныйСостКодМеньшеL(L);
                    L=L-L2;
                    ЗаписатьБелыйКодДля(L2);
                }
                ЗаписатьБелыйКодДля(L); //Это всегда код завершения
            }
            else {
            /Код аналогичный белой серии, с той разницей, что записываются  черные коды/
          }
        } // Окончание строки изображения
        }
```

Поскольку черные и белые серии чередуются, то реально код для белой и код для черной серии будут работать попеременно.

В терминах регулярных выражений мы получим для каждой строки нашего изображения (достаточно длинной, начинающейся с белой точки) выходной битовый поток вида:

((<Б-2560>)*[<Б-сст.>]<Б-зв.>(<Ч-2560>)*[<Ч-сст.>]<Ч-зв.>)$^{+}$[(<Б-2560>)*[<Б-сст.>]<Б-зв.>],

где ()*– повтор 0 или более раз, ()$^{+}$.– повтор 1 или более раз, [] – включение 1 или 0 раз.

Для приведенного ранее примера: 0, 3, 556, 10... алгоритм сформирует следующий код: <Б-0><Ч-3><Б-512><Б-44><Ч-10>, или, согласно таблице, 00110101**10**0110010*10101101***0000100** (разные коды в потоке выделены для удобства). Этот код обладает свойством префиксных кодов и легко может быть свернут обратно в последовательность длин серий. Легко подсчитать, что для приведенной строки в 569 бит мы получили код длиной в 33 бита, т.е. степень сжатия составляет примерно 17 раз.

Заметим, что единственное «сложное» выражение в нашем алгоритме: L2=МаксимальныйДопКодМеньшеL(L) – на практике работает очень просто: L2=(L>>6)*64, где >> – побитовый сдвиг L влево на 6 битов (можно сделать то же самое за одну побитовую операцию & – логическое И).

Таблица 23

Коды завершения серий

Длина серии	Код белой подстроки	Код черной подстроки	Длина серии	Код белой подстроки	Код черной подстроки
0	00110101	0000110111	32	00011011	000001101010
1	00111	010	33	00010010	000001101011
2	0111	11	34	00010011	000011010010
3	1000	10	35	00010100	000011010011
4	1011	011	36	00010101	000011010100
5	1100	0011	37	00010110	000011010101
6	1110	0010	38	00010111	000011010110
7	1111	00011	39	00101000	000011010111
8	10011	000101	40	00101001	000001101100
9	10100	000100	41	00101010	000001101101
10	00111	0000100	42	00101011	000011011010
11	01000	0000101	43	00101100	000011011011
12	001000	0000111	44	00101101	000001010100
13	000011	00000100	45	00000100	000001010101
14	110100	00000111	46	00000101	000001010110
15	110101	000011000	47	00001010	000001010111
16	101010	0000010111	48	00001011	000001100100
17	101011	0000011000	49	01010010	000001100101
18	0100111	0000001000	50	01010011	000001010010
19	0001100	00001100111	51	01010100	000001010011
20	0001000	00001101000	52	01010101	000000100100
21	0010111	00001101100	53	00100100	000000110111
22	0000011	00000110111	54	00100101	000000111000
23	0000100	00000101000	55	01011000	000000100111
24	0101000	00000010111	56	01011001	000000101000
25	0101011	00000011000	57	01011010	000001011000
26	0010011	000011001010	58	01011011	000001011001
27	0100100	000011001011	59	01001010	000000101011
28	0011000	000011001100	60	01001011	000000101100
29	00000010	000011001101	61	00110010	000001011010
30	00000011	000001101000	62	00110011	000001100110
31	00011010	000001101001	63	00110100	000001100111

JBIG

Алгоритм разработан группой экспертов ISO (Joint Bi-level Experts Group) специально для сжатия однобитных черно-белых изображений. Например, факсов или отсканированных документов. В принципе, может применяться и к 2-х, и к 4-х битовым картинкам. При этом алгоритм разбивает их на отдельные битовые плоскости. JBIG позволяет управлять такими параметрами, как порядок разбиения изображения на битовые плоскости, ширина полос в изображении, уровни масштабирования. Последняя возможность позволяет легко ориентироваться в базе больших по размерам изображений, просматривая сначала их уменьшенные копии. Настраивая эти параметры, можно использовать описанный выше эффект «огрубленного изображения» при получении изображения по сети или по любому другому каналу, пропускная способность которого мала по сравнению с возможностями процессора. Распа-

ковываться изображение на экране будет постепенно, как бы медленно «проявляясь». При этом человек начинает анализировать картинку задолго до конца процесса разархивации.

Алгоритм построен на базе Q-кодировщика, патентом на который владеет IBM. Q-кодер, так же как и алгоритм Хаффмана, использует для чаще появляющихся символов короткие цепочки, а для реже появляющихся — длинные. Однако, в отличие от него, в алгоритме используются и последовательности символов.

Lossless JPEG

Этот алгоритм разработан группой экспертов в области фотографии (Joint Photographic Expert Group). В отличие от JBIG, Lossless JPEGориентирован на полноцветные 24-битные или 8-битные в градациях серого изображения без палитры. Он представляет собой специальную реализацию JPEG без потерь. Степени сжатия: 20, 2, 1. Lossless JPEG рекомендуется применять в тех приложениях, где необходимо побитовое соответствие исходного и декомпрессированного изображений.

АЛГОРИТМЫ СЖАТИЯ ИЗОБРАЖЕНИЙ С ПОТЕРЯМИ

Первыми для архивации изображений стали применяться привычные алгоритмы. Те, что использовались и используются в системах резервного копирования, при создании дистрибутивов и т.п. Эти алгоритмы архивировали информацию без изменений. Однако основной тенденцией в последнее время стало использование новых классов изображений. Старые алгоритмы перестали удовлетворять требованиям, предъявляемым к архивации. Многие изображения практически не сжимались, хотя «на взгляд» обладали явной избыточностью. Это привело к созданию нового типа алгоритмов – сжимающих с потерей информации. Как правило, степень сжатия и, следовательно, степень потерь качества в них можно задавать. При этом достигается компромисс между размером и качеством изображений.

Одна из серьезных проблем машинной графики заключается в том, что до сих пор не найден адекватный критерий оценки потерь качества изображения. А теряется оно постоянно –при оцифровке, при переводе в ограниченную палитру цветов, при переводе в другую систему цветопредставления для печати, и, что для нас особенно важно, при архивации с потерями. Можно привести пример простого критерия: среднеквадратичное отклонение значений пикселов (L_2 мера, или root mean square – RMS):

$$d(x,y)=\sqrt{\frac{\sum_{i=1,j=1}^{n,n}\left(x_{ij}-y_{ij}\right)^2}{n^2}}$$

По нему изображение будет сильно испорчено при понижении яркости всего на 5% (глаз этого не заметит – у разных мониторов настройка яркости варьируется гораздо сильнее). В то же время изображения со «снегом» – резким изменением цвета отдельных точек, слабыми полосами или «муаром» будут признаны «почти не изменившимися». Свои неприятные стороны есть и у других критериев.

Рассмотрим, например, максимальное отклонение:

$$d(x,y)=\max_{i,j}\left|x_{ij}-y_{ij}\right|$$

Эта мера, как можно догадаться, крайне чувствительна к биению отдельных пикселов. Т.е. во всем изображении может существенно измениться только значение одного пиксела (что практически незаметно для глаза), однако согласно этой мере изображение будет сильно испорчено.

Мера, которую сейчас используют на практике, называется мерой отношения сигнала к шуму (peak-to-peak signal-to-noise ratio – PSNR).

$$d(x,y) = 10 \cdot \log_{10} \frac{255^2 \cdot n^2}{\sum_{i=1,j=1}^{n,n} \left(x_{ij} - y_{ij}\right)^2}$$

Данная мера, по сути, аналогична среднеквадратичному отклонению, однако пользоваться ей несколько удобнее за счет логарифмического масштаба шкалы. Ей присущи те же недостатки, что и среднеквадратичному отклонению.

Лучше всего потери качества изображений оценивают наши глаза. Отличной считается архивация, при которой невозможно на глаз различить первоначальное и разархивированное изображения. Хорошей — когда сказать, какое из изображений подвергалось архивации, можно только сравнивая две находящихся рядом картинки. При дальнейшем увеличении степени сжатия, как правило, становятся заметны побочные эффекты, характерные для данного алгоритма. На практике, даже при отличном сохранении качества, в изображение могут быть внесены регулярные специфические изменения. Поэтому алгоритмы архивации с потерями не рекомендуется использовать при сжатии изображений, которые в дальнейшем собираются либо печатать с высоким качеством, либо обрабатывать программами распознавания образов. Неприятные эффекты с такими изображениями, как мы уже говорили, могут возникнуть даже при простом масштабировании изображения.

Алгоритм JPEG

JPEG – один из самых новых и достаточно мощных алгоритмов. Практически он является стандартом де-факто для полноцветных изображений. Оперирует алгоритм областями 8x8, на которых яркость и цвет меняются сравнительно плавно. Вследствие этого, при разложении матрицы такой области в двойной ряд по косинусам (см. формулы ниже) значимыми оказываются только первые коэффициенты. Таким образом, сжатие в JPEG осуществляется за счет плавности изменения цветов в изображении.

Алгоритм разработан группой экспертов в области фотографии специально для сжатия 24-битных изображений. JPEG –Joint Photographic Expert Group – подразделение в рамках ISO – Международной организации по стандартизации. Название алгоритма читается ['jei'peg]. В целом алгоритм основан на дискретном косинусоидальном преобразовании (ДКП), применяемом к матрице изображения для получения некоторой новой матрицы коэффициентов. Для получения исходного изображения применяется обратное преобразование.

ДКП раскладывает изображение по амплитудам некоторых частот. Таким образом, при преобразовании мы получаем матрицу, в которой многие коэффициенты либо близки, либо равны нулю. Кроме того, благодаря несовершенству человеческого зрения, можно аппроксимировать коэффициенты более грубо без заметной потери качества изображения.

Для этого используется квантование коэффициентов (quantization). В самом простом случае – это арифметический побитовый сдвиг вправо. При этом преобразовании теряется часть информации, но может достигаться большая степень сжатия.

Итак, рассмотрим алгоритм подробнее. Пусть мы сжимаем 24-битное изображение.

Шаг 1.

Переводим изображение из цветового пространства RGB, с компонентами, отвечающими за красную (Red), зеленую (Green) и синюю (Blue) составляющие цвета точки, в цветовое пространство YCrCb (иногда называют YUV).

В нем Y– яркостная составляющая, а Cr, Cb – компоненты, отвечающие за цвет (хроматический красный и хроматический синий). За счет того, что человеческий глаз менее чувствителен к цвету, чем к яркости, появляется возможность архивировать массивы для Cr и Cb компонент с большими потерями и, соответственно, большими степенями сжатия. Подобное преобразование уже давно используется в телевидении. На сигналы, отвечающие за цвет, там выделяется более узкая полоса частот.

Упрощенно перевод из цветового пространства RGB в цветовое пространство YCrCb можно представить с помощью матрицы перехода:

$$\begin{vmatrix} Y \\ Cb \\ Cr \end{vmatrix} = \begin{vmatrix} 0.299 & 0.587 & 0.114 \\ 0.5 & -0.4187 & -0.0813 \\ 0.1687 & -0.3313 & 0.5 \end{vmatrix} * \begin{vmatrix} R \\ G \\ B \end{vmatrix} + \begin{vmatrix} 0 \\ 128 \\ 128 \end{vmatrix}$$

Обратное преобразование осуществляется умножением вектора YUV на обратную матрицу.

$$\begin{vmatrix} R \\ G \\ B \end{vmatrix} = \begin{vmatrix} 1 & 0 & 1.402 \\ 1 & -0.34414 & -0.71414 \\ 1 & 1.772 & 0 \end{vmatrix} * \left(\begin{vmatrix} Y \\ Cb \\ Cr \end{vmatrix} - \begin{vmatrix} 0 \\ 128 \\ 128 \end{vmatrix} \right).$$

Шаг 2.

Разбиваем исходное изображение на матрицы 8x8. Формируем из каждой три рабочие матрицы ДКП – по 8 бит отдельно для каждой компоненты. При больших степенях сжатия этот шаг может выполняться чуть сложнее. Изображение делится по компоненте Y – как и в первом случае, а для компонент Cr и Cb матрицы набираются через строчку и через столбец. Т.е. из исходной матрицы размером 16x16 получается только одна рабочая матрица ДКП. При этом, как нетрудно заметить, мы теряем 3/4 полезной информации о цветовых составляющих изображения и получаем сразу сжатие в два раза. Мы можем поступать так благодаря работе в пространстве YCrCb. На результирующем RGB изображении, как показала практика, это сказывается несильно.

Шаг 3.

В упрощенном виде ДКП при n=8 можно представить так:

$$Y[u,v] = \frac{1}{4} \sum_{i=0}^{7} \sum_{j=0}^{7} C(i,u) \times C(j,v) \times y[i,j],$$

где: $C(i,u) = A(u) \times \cos\left(\frac{(2 \times i + 1) \times u \times \pi}{2 \cdot n} \right)$, $A(u) = \begin{cases} \frac{1}{\sqrt{2}}, & \text{for u} \equiv 0 \\ 1, & \text{for u} \neq 0 \end{cases}$

Применяем ДКП к каждой рабочей матрице. При этом мы получаем матрицу, в которой коэффициенты в левом верхнем углу соответствуют низкочастотной составляющей изображения, а в правом нижнем — высокочастотной. Понятие частоты следует из рассмотрения изображения как двумерного сигнала (аналогично рассмотрению звука как сигнала). Плавное изменение цвета соответствует низкочастотной составляющей, а резкие скачки — низкочастотной.

Шаг 4.

Производим квантование. В принципе, это просто деление рабочей матрицы на матрицу квантования поэлементно. Для каждой компоненты (Y, U и V), в общем случае, задается своя матрица квантования q[u,v] (далее МК).

$$Yq[u,v] = \text{IntegerRound} \left(\frac{Y[u,v]}{q[u,v]} \right)$$

На этом шаге осуществляется управление степенью сжатия, и происходят самые большие потери. Понятно, что, задавая МК с большими коэффициентами, мы получим больше нулей и, следовательно, большую степень сжатия.

В стандарт JPEG включены рекомендованные МК, построенные опытным путем. Матрицы для большей или меньшей степени сжатия получают путем умножения исходной матрицы на некоторое число gamma.

С квантованием связаны и специфические эффекты алгоритма. При больших значениях коэффициента gamma потери в низких частотах могут быть настолько велики, что изображение распадется на квадраты 8x8. Потери в высоких частотах могут проявиться в так называемом «эффекте Гиббса», когда вокруг контуров с резким переходом цвета образуется своеобразный «нимб».

Шаг 5.

Переводим матрицу 8x8 в 64-элементный вектор при помощи «зигзаг» – сканирования, т.е. берем элементы с индексами (0,0), (0,1), (1,0), (2,0) (рисунок 6).

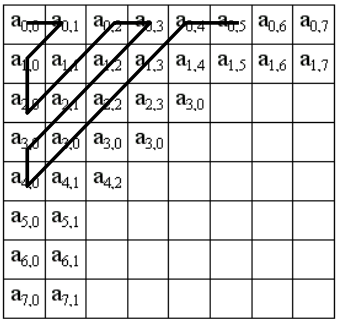


Рис. 6. Зигзаг–сканирование

Таким образом, в начале вектора мы получаем коэффициенты матрицы, соответствующие низким частотам, а в конце – высоким.

Шаг 6.

Свертываем вектор с помощью алгоритма группового кодирования. При этом получаем пары типа (пропустить, число), где «пропустить» является счетчиком пропускаемых нулей, а «число» — значение, которое необходимо поставить в следующую ячейку. Так, вектор 42 3 0 0 0 -2 0 0 0 0 1 . будет свернут в пары (0,42) (0,3) (3,-2) (4,1) .

Шаг 7.

Свертываем получившиеся пары кодированием по Хаффману с фиксированной таблицей.

Процесс восстановления изображения в этом алгоритме полностью симметричен (рисунок 7). Метод позволяет сжимать некоторые изображения в 10-15 раз без серьезных потерь.

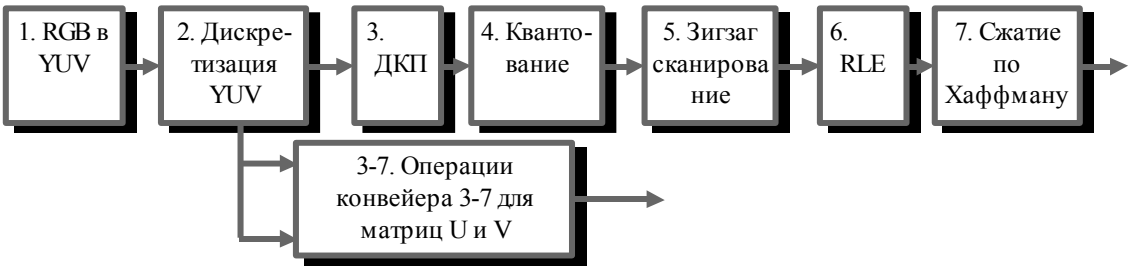


Рис. 7. Конвейер операций, используемый в алгоритме JPEG

Существенными положительными сторонами алгоритма является то, что:

- Задается степень сжатия.
- Выходное цветное изображение может иметь 24 бита на точку.
- Отрицательными сторонами алгоритма является то, что:
- При повышении степени сжатия изображение распадается на отдельные квадраты (8x8). Это связано с тем, что происходят большие потери в низких частотах при квантовании, и восстановить исходные данные становится невозможно.
- Проявляется эффект Гиббса – ореолы по границам резких переходов цветов.

Как уже говорилось, стандартизован JPEG относительно недавно – в 1991 году. Но уже тогда существовали алгоритмы, сжимающие сильнее при меньших потерях качества. Дело в том, что действия разработчиков стандарта были ограничены мощностью существовавшей на тот момент техники. То есть даже на персональном компьютере алгоритм должен был работать меньше минуты на среднем изображении, а его аппаратная реализация должна быть относительно простой и дешевой. Алгоритм должен был быть симметричным (время разархивации примерно равно времени архивации).

Выполнение последнего требования сделало возможным появление таких устройств, как цифровые фотоаппараты, снимающие 24-битовые фотографии на 8-256 Мб флэш карту. Потом эта карта вставляется в разъем на вашем ноутбуке и соответствующая программа позволяет считать изображения. Не правда ли, если бы алгоритм был несимметричен, было бы неприятно долго ждать, пока аппарат «перезарядится» – сожмет изображение.

Не очень приятным свойством JPEG является также то, что нередко горизонтальные и вертикальные полосы на дисплее абсолютно не видны и могут проявиться только при печати в виде муарового узора. Он возникает при наложении наклонного растра печати на горизонтальные и вертикальные полосы изображения. Из-за этих сюрпризов JPEG не рекомендуется активно использовать в полиграфии, задавая высокие коэффициенты матрицы квантования. Однако при архивации изображений, предназначенных для просмотра человеком, он на данный момент незаменим.

Широкое применение JPEG долгое время сдерживалось, пожалуй, лишь тем, что он оперирует 24-битными изображениями. Поэтому для того, чтобы с приемлемым качеством посмотреть картинку на обычном мониторе в 256-цветной палитре, требовалось применение соответствующих алгоритмов и, следовательно, определенное время. В приложениях, ориентированных на придирчивого пользователя, таких, например, как игры, подобные задержки неприемлемы. Кроме того, если имеющиеся у вас изображения, допустим, в 8-битном формате GIF перевести в 24-битный JPEG, а потом обратно в GIF для просмотра, то потеря качества произойдет дважды при обоих преобразованиях. Тем не менее, выигрыш в размерах архивов зачастую настолько велик (в 3-20 раз), а потери качества настолько малы, что хранение изображений в JPEG оказывается очень эффективным.

Несколько слов необходимо сказать о модификациях этого алгоритма. Хотя JPEG и является стандартом ISO, формат его файлов не был зафиксирован. Пользуясь этим, производители создают свои, несовместимые между собой форматы, и, следовательно, могут изменить алгоритм. Так, внутренние таблицы алгоритма, рекомендованные ISO, заменяются ими на свои собственные. Кроме того, легкая неразбериха присутствует при задании степени потерь. Например, при тестировании выясняется, что «отличное» качество, «100%» и «10 баллов» дают существенно различающиеся картинки. При этом, кстати, «100%» качества не означают сжатие без потерь. Встречаются также варианты JPEG для специфических приложений.

Как стандарт ISO JPEG начинает все шире использоваться при обмене изображениями в компьютерных сетях. Поддерживается алгоритм JPEG в форматах Quick Time, PostScript Level 2, Tiff 6.0 и, на данный момент, занимает видное место в системах мультимедиа.

Фрактальный алгоритм

Фрактальная архивация основана на том, что мы представляем изображение в более компактной форме – с помощью коэффициентов системы итерируемых функций (Iterated Function System –IFS). Прежде, чем рассматривать сам процесс архивации, разберем, как IFS строит изображение, т.е. процесс декомпрессии.

Строго говоря, IFS представляет собой набор трехмерных аффинных преобразований, в нашем случае переводящих одно изображение в другое. Преобразованию подвергаются точки в трехмерном пространстве (x_координата, y_координата, яркость).

Наиболее наглядно этот процесс продемонстрировал Барнсли в своей книге «Fractal Image Compression». Там введено понятие Фотокопировальной Машины (рисунок 8), состоящей из экрана, на котором изображена исходная картинка, и системы линз, проецирующих изображение на другой экран:

- Линзы могут проецировать часть изображения **произвольной формы** в любое другое место нового изображения.
- Области, **в которые** проецируются изображения, **не пересекаются**.
- Линза может **менять яркость и уменьшать контрастность**.
- Линза может **зеркально отражать и поворачивать** свой фрагмент изображения.
- Линза **должна масштабировать** (причем только уменьшая) свой фрагмент изображения.

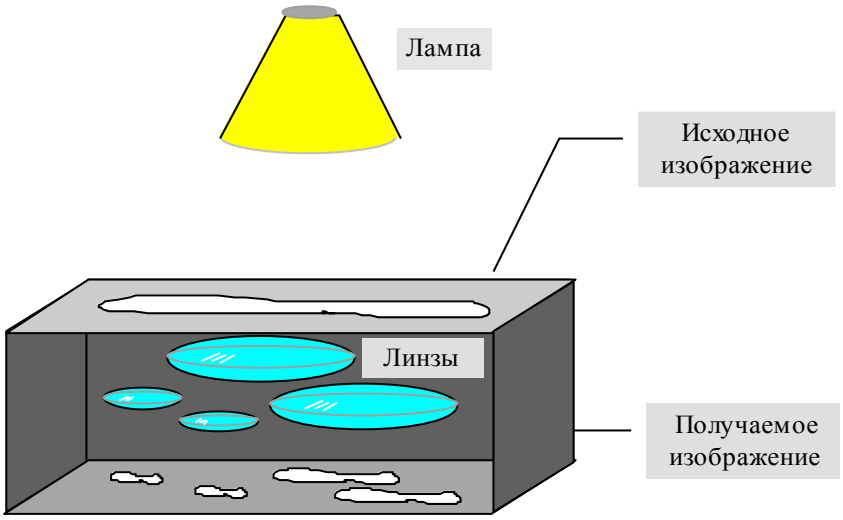


Рис. 8. Машина Барнсли

Расставляя линзы и меняя их характеристики, мы можем управлять получаемым изображением. Одна итерация работы Машины заключается в том, что по исходному изображению с помощью проектирования строится новое, после чего новое берется в качестве исходного. Утверждается, что в процессе итераций мы получим изображение, которое перестанет изменяться. Оно будет зависеть только от расположения и характеристик линз, и не будет зависеть от исходной картинки. Это изображение называется «неподвижной точкой» или аттрактором данной IFS. Соответствующая теория гарантирует наличие ровно одной неподвижной точки для каждой IFS.

Поскольку отображение линз является сжимающим, каждая линза в явном виде задает самоподобные области в нашем изображении. Благодаря самоподобию мы получаем сложную структуру изображения при любом увеличении. Таким образом, интуитивно понятно, что система итерируемых функций задает фрактал (нестрого — самоподобный математический объект).

Наиболее известны два изображения, полученных с помощью IFS: «треугольник Серпинского» (рисунок 9) и «папоротник Барнсли» (рисунок 10).

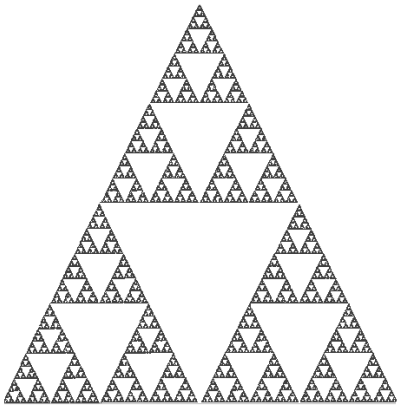


Рис. 9. Треугольник Серпинского.

«Треугольник Серпинского» задается тремя, а «папоротник Барнсли» четырьмя аффинными преобразованиями (или, в нашей терминологии, «линзами»). Каждое преобразование кодируется буквально считанными байтами, в то время как изображение, построенное с их помощью, может занимать и несколько мегабайт.

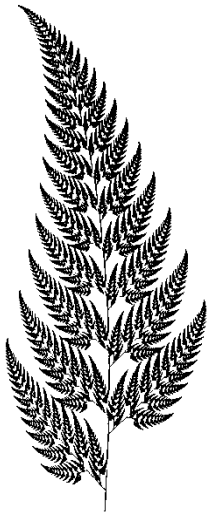


Рис. 10. Папоротник Барнсли

Из вышесказанного становится понятно, как работает архиватор, и почему ему требуется так много времени. Фактически, фрактальная компрессия – это поиск самоподобных областей в изображении и определение для них параметров аффинных преобразований.

В худшем случае, если не будет применяться оптимизирующий алгоритм, потребуется перебор и сравнение всех возможных фрагментов изображения разного размера. Даже для небольших изображений при учете дискретности мы получим астрономическое число перебираемых вариантов. Причем, даже резкое сужение классов преобразований, например, за счет масштабирования только в определенное количество раз, не дает заметного выигрыша во времени. Кроме того, при этом теряется качество изображения. Подавляющее большинство исследований в области фрактальной компрессии сейчас направлены на уменьшение времени архивации, необходимого для получения качественного изображения.

Далее приводятся основные определения и теоремы, на которых базируется фрактальная компрессия.

Преобразование $w: R^2 \to R^2$, представимое в виде

$$w(\bar{r}) = w\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \bullet \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}$$

где: *a, b, c, d, e, f* действительные числа и $(x \quad y) \in R^2$ называется двумерным аффинным преобразованием

Преобразование $w: R^3 \to R^3$, представимое в виде

$$w(\bar{r}) = w\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a & b & t \\ c & d & u \\ r & s & p \end{pmatrix} \bullet \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} e \\ f \\ q \end{pmatrix}$$

где: a, b, c, d, e, f, p, q, r, s, t, u действительные числа и $(x \quad y \quad z) \in R^3$ называется трехмерным аффинным преобразованием.

Пусть $f: \mathrm{X} \to \mathrm{X}$ – преобразование в пространстве X. Точка $x_f \in \mathrm{X}$ такая, что $f(x_f) = x_f$ называется неподвижной точкой (аттрактором) преобразования.

Преобразование $f: \mathrm{X} \to \mathrm{X}$ в метрическом пространстве (X, d) называется сжимающим, если существует число s: $0 \le s < 1$, такое, что

$$d(f(x), f(y)) \le s \cdot d(x, y) \qquad \forall x, y \in \mathrm{X}$$

Формально возможно использовать любое сжимающее отображение при фрактальной компрессии, но реально используются лишь трехмерные аффинные преобразования с достаточно сильными ограничениями на коэффициенты.

Теорема «О сжимающем преобразовании»

Пусть $f: \mathrm{X} \to \mathrm{X}$ — сжимающее преобразование в полном метрическом пространстве (X, *d*). Тогда существует в точности одна неподвижная точка $x_f \in \mathrm{X}$ этого преобразования, и для любой точки $x \in \mathrm{X}$ последовательность $\{f^n(x): \quad n = 0,1,2... \quad\}$ сходится к x_f.

Более общая формулировка этой теоремы гарантирует нам сходимость.

Изображением называется функция S, определенная на единичном квадрате и принимающая значения от 0 до 1 или $S(x, y) \in [0...1] \qquad \forall x, y \in [0...1]$

Пусть трехмерное аффинное преобразование $w_i: R^3 \to R^3$, записано в виде

$$w_i(\bar{r}) = w_i\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & p \end{pmatrix} \bullet \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} e \\ f \\ q \end{pmatrix}$$

и определено на компактном подмножестве D_i декартова квадрата [0..1]x[0..1] (используем особым видом матрицы преобразования, чтобы уменьшить размерность области определения с R^3 до R^2). Тогда оно переведет часть поверхности *S* в область R_i, расположенную со сдвигом *(e,f)* и поворотом, заданным матрицей

$$\begin{pmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

При этом, если интерпретировать значения функции $S(x, y) \in [0...1]$ как яркость соответствующих точек, она уменьшится в *p* раз (преобразование обязано быть сжимающим) и изменится на сдвиг *q*.

Конечная совокупность *W* сжимающих трехмерных аффинных преобразований w_i, определенных на областях D_i, таких, что $w_i(D_i) = R_i$ и $R_i \cap R_j = \phi \quad \forall i \ne j$, называется системой итерируемых функций (IFS).

Системе итерируемых функций однозначно сопоставляется неподвижная точка — изображение. Таким образом, процесс компрессии заключается в поиске коэффициентов системы, а процесс декомпрессии – в проведении итераций системы до стабилизации полученного изображения (неподвижной точки IFS). На практике бывает достаточно 7-16 итераций. Области R_i в дальнейшем будут именоваться ранговыми, а области D_i – доменными.

Построение алгоритма

Как уже стало очевидным из изложенного выше, основной задачей при компрессии фрактальным алгоритмом является нахождение соответствующих аффинных преобразований. В самом общем случае мы можем переводить любые по размеру и форме области изображения, однако в этом случае получается астрономическое число перебираемых вариантов разных фрагментов, которое невозможно обработать на текущий момент даже на суперкомпьютере.

В учебном варианте алгоритма, изложенном далее, сделаны следующие ограничения на области:

Все области являются квадратами со сторонами, параллельными сторонам изображения. Это ограничение достаточно жесткое. Фактически мы собираемся аппроксимировать все многообразие геометрических фигур лишь квадратами.

При переводе доменной области в ранговую уменьшение размеров производится ровно в два раза. Это существенно упрощает как компрессор, так и декомпрессор, т.к. задача масштабирования небольших областей является нетривиальной.

Все доменные блоки – квадраты и имеют фиксированный размер. Изображение равномерной сеткой разбивается на набор доменных блоков.

Доменные области берутся «через точку» и по X, и по Y, что сразу уменьшает перебор в 4 раза.

При переводе доменной области в ранговую поворот куба возможен только на 00, 900, 1800 или 2700. Также допускается зеркальное отражение. Общее число возможных преобразований (считая пустое) – 8.

Масштабирование (сжатие) по вертикали (яркости) осуществляется *в* фиксированное число раз – в 0,75.

Эти ограничения позволяют:

- Построить алгоритм, для которого требуется сравнительно малое число операций даже на достаточно больших изображениях.
- Очень компактно представить данные для записи в файл. Нам требуется на каждое аффинное преобразование в IFS:
    - два числа для того, чтобы задать смещение доменного блока. Если мы ограничим входные изображения размером 512x512, то достаточно будет по 8 бит на каждое число.
    - три бита для того, чтобы задать преобразование симметрии при переводе доменного блока в ранговый.
    - 7-9 бит для того, чтобы задать сдвиг по яркости при переводе.

Информацию о размере блоков можно хранить в заголовке файла. Таким образом, мы затратили менее 4 байт на одно аффинное преобразование. В зависимости от того, каков размер блока, можно высчитать, сколько блоков будет в изображении. Таким образом, мы можем получить оценку степени компрессии.

Например, для файла в градациях серого 256 цветов 512x512 пикселов при размере блока 8 пикселов аффинных преобразований будет 4096 (512/8·512/8). На каждое потребуется 3.5 байта. Следовательно, если исходный файл занимал 262144 (512·512) байт (без учета заголовка), то файл с коэффициентами будет занимать 14336 байт. Степень сжатия — 18 раз.

При этом мы не учитываем, что файл с коэффициентами тоже может обладать избыточностью и архивироваться методом архивации без потерь, например LZW.

Отрицательные стороны предложенных ограничений:

1. Поскольку все области являются квадратами, невозможно воспользоваться подобием объектов, по форме далеких от квадратов (которые встречаются в реальных изображениях достаточно часто.)
2. Аналогично мы не сможем воспользоваться подобием объектов в изображении, коэффициент подобия между которыми сильно отличается от 2.
3. Алгоритм не сможет воспользоваться подобием объектов в изображении, угол между которыми не кратен 90^0.

Такова плата за скорость компрессии и за простоту упаковки коэффициентов в файл.

Сам алгоритм упаковки сводится к перебору всех доменных блоков и подбору для каждого соответствующего ему рангового блока. Ниже приводится схема этого алгоритма.

```
for (all range blocks) {
  min_distance = MaximumDistance;
  Rij = image->CopyBlock(i,j);
  for (all domain blocks) { // С поворотами и отр.
        current=Координаты тек. преобразования;
        D=image->CopyBlock(current);
        current_distance = Rij.L2distance(D);
        if(current_distance < min_distance) {
              // Если коэффициенты best хуже:
              min_distance = current_distance;
              best = current;
        }
  } // Next range block
  Save_Coefficients_to_file(best);
} // Next domain block
```

Как видно из приведенного алгоритма, для каждого рангового блока делаем его проверку со всеми возможными доменными блоками (в том числе с прошедшими преобразование симметрии), находим вариант с наименьшей мерой L_2 (наименьшим среднеквадратичным отклонением) и сохраняем коэффициенты этого преобразования в файл. Коэффициенты — это (1) координаты найденного блока, (2) число от 0 до 7, характеризующее преобразование симметрии (поворот, отражение блока), и (3) сдвиг по яркости для этой пары блоков. Сдвиг по яркости вычисляется как:

$$q = \left\lfloor \sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij} - \sum_{i=1}^{n}\sum_{j=1}^{n} r_{ij} \right\rfloor / n^2 ,$$

где r_{ij} — значения пикселов рангового блока (R), а d_{ij} — значения пикселов доменного блока (D). При этом мера считается как:

$$d(R,D) = \sum_{i=1}^{n}\sum_{j=1}^{n} (0.75 r_{ij} + q - d_{ij})^2 .$$

Мы не вычисляем квадратного корня из L_2 меры и не делим ее на n, поскольку данные преобразования монотонны и не помешают нам найти экстремум, однако мы сможем выполнять на две операции меньше для каждого блока.

Посчитаем количество операций, необходимых нам для сжатия изображения в градациях серого 256 цветов 512x512 пикселов при размере блока 8 пикселов:

Часть программы	Число операций
for (all domain blocks)	4096 (=512/8·512/8)
for (all range blocks) + symmetry transformation	492032 (=(512/2-8)* (512/2-8)*8)
Вычисление q и $d(R,D)$	> 3*64 операций «+»
	> 2*64 операций «·»
Итог:	> 3* 128.983.236.608 операций «+»
	> 2* 128.983.236.608 операций «·»

Таким образом, нам удалось уменьшить число операций алгоритма компрессии до вполне вычисляемых (пусть и за несколько часов) величин.

Декомпрессия алгоритма фрактального сжатия чрезвычайно проста. Необходимо провести несколько итераций трехмерных аффинных преобразований, коэффициенты которых были получены на этапе компрессии.

В качестве начального может быть взято абсолютно любое изображение (например, абсолютно черное), поскольку соответствующий математический аппарат гарантирует нам сходимость последовательности изображений, получаемых в ходе итераций IFS, к неподвижному изображению (близкому к исходному). Обычно для этого достаточно 16 итераций.

```
Прочитаем из файла коэффициенты всех блоков;
Создадим черное изображение нужного размера;
Until(изображение не станет неподвижным){
  For(every range (R)){
        D=image->CopyBlock(D_coord_for_R);
        For(every pixel(i,j) in the block{
              Rij = 0.75Dij + oR;
        } //Next pixel
  } //Next block
}//Until end
```

Поскольку мы записывали коэффициенты для блоков R_{ij} (которые, как мы оговорили, в нашем частном случае являются квадратами одинакового размера) *последовательно*, то получается, что мы последовательно заполняем изображение по квадратам сетки разбиения использованием аффинного преобразования.

Как можно подсчитать, количество операций на один пиксел изображения в градациях серого при восстановлении необычайно мало (N операций сложения «+» и N операций умножения «·», где N — количество итераций, т.е. 7-16). Благодаря этому, декомпрессия изображений для фрактального алгоритма проходит быстрее декомпрессии, например, для алгоритма JPEG. В простой реализации JPEG на точку приходится 64 операции сложения «+» и 64 операции умножения «·». При реализации быстрого ДКП можно получить, 7 сложений и 5 умножений на точку, но это без учета шагов RLE, квантования и кодирования по Хаффману. При этом для фрактального алгоритма умножение происходит на рациональное число, одно для каждого блока. Это означает, что мы можем, во-первых, использовать целочисленную рациональную арифметику, которая быстрее арифметики с плавающей точкой. Во-вторых, можно использовать умножение вектора на число — более простую и быструю операцию, часто закладываемую в архитектуру процессора (процессоры SGI, Intel MMX,

векторные операции Athlon и т.д.). Для полноцветного изображения ситуация качественно не изменяется, поскольку перевод в другое цветовое пространство используют оба алгоритма.

При кратком изложении упрощенного варианта алгоритма были пропущены многие важные вопросы. Например, что делать, если алгоритм не может подобрать для какого-либо фрагмента изображения подобный ему? Достаточно очевидное решение — разбить этот фрагмент на более мелкие, и попытаться поискать для них. В то же время понятно, что эту процедуру нельзя повторять до бесконечности, иначе количество необходимых преобразований станет так велико, что алгоритм перестанет быть алгоритмом компрессии. Следовательно, мы допускаем потери в какой-то части изображения.

Для фрактального алгоритма компрессии, как и для других алгоритмов сжатия с потерями, очень важны механизмы, с помощью которых можно будет регулировать степень сжатия и степень потерь. К настоящему времени разработан достаточно большой набор таких методов. Во-первых, можно ограничить количество аффинных преобразований, заведомо обеспечив степень сжатия не ниже фиксированной величины. Во-вторых, можно потребовать, чтобы в ситуации, когда разница между обрабатываемым фрагментом и наилучшим его приближением будет выше определенного порогового значения, этот фрагмент дробился обязательно (для него обязательно заводится несколько «линз»). В-третьих, можно запретить дробить фрагменты размером меньше, допустим, четырех точек. Изменяя пороговые значения и приоритет этих условий, мы будем очень гибко управлять коэффициентом компрессии изображения в диапазоне от побитового соответствия до любой степени сжатия. Заметим, что эта гибкость будет гораздо выше, чем у ближайшего «конкурента» – алгоритма JPEG.

Рекурсивный (волновой) алгоритм

Английское название рекурсивного сжатия – wavelet. На русский язык оно переводится как волновое сжатие, как сжатие с использованием всплесков, а в последнее время и калькой вэйвлет-сжатие. Этот вид архивации известен довольно давно и напрямую исходит из идеи использования когерентности областей. Ориентирован алгоритм на цветные и черно-белые изображения с плавными переходами. Идеален для картинок типа рентгеновских снимков. Степень сжатия задается и варьируется в пределах 5-100. При попытке задать больший коэффициент на резких границах, особенно проходящих по диагонали, проявляется «лестничный эффект» – ступеньки разной яркости размером в несколько пикселов.

Идея алгоритма заключается в том, что мы сохраняем в файл разницу — число между средними значениями соседних блоков в изображении, которая обычно принимает значения, близкие к 0.

Так два числа a_{2i} и a_{2i+1} всегда можно представить в виде $b^1_i=(a_{2i}+a_{2i+1})/2$ и $b^2_i=(a_{2i}-a_{2i+1})/2$. Аналогично последовательность a_i может быть попарно переведена в последовательность $b^{1,2}_i$.

Разберем конкретный пример: пусть мы сжимаем строку из 8 значений яркости пикселов (a_i): (220, 211, 212, 218, 217, 214, 210, 202). Мы получим следующие последовательности b^1_i, и b^2_i: (215.5, 215, 215.5, 206) и (4.5, -3, 1.5, 4). Заметим, что значения b^2_i достаточно близки к 0. Повторим операцию, рассматривая b^1_i как a_i. Данное действие выполняется как бы рекурсивно, откуда и название алгоритма. Мы получим из (215.5, 215, 215.5, 206): (215.25, 210.75) (0.25, 4.75). Полученные коэффициенты, округлив до целых и сжав, например, с помощью алгоритма Хаффмана с фиксированными таблицами, мы можем поместить в файл.

Заметим, что мы применяли наше преобразование к цепочке только два раза. Реально мы можем позволить себе применение wavelet – преобразования 4-6 раз. Более того, дополнительное сжатие можно получить, используя таблицы алгоритма Хаффмана с неравномерным шагом (т.е. нам придется сохранять код Хаффмана для ближайшего в таблице значения). Эти приемы позволяют достичь заметных степеней сжатия.

Алгоритм для двумерных данных реализуется аналогично. Если у нас есть квадрат из 4 точек с яркостями $a_{2i,2j}$, $a_{2i+1,2j}$, $a_{2i,2j+1}$, и $a_{2i+1,2j+1}$, то

$$b^1_{i,j} = (a_{2i,2j} + a_{2i+1,2j} + a_{2i,2j+1} + a_{2i+1,2j+1})/4$$
$$b^2_{i,j} = (a_{2i,2j} + a_{2i+1,2j} - a_{2i,2j+1} - a_{2i+1,2j+1})/4$$
$$b^3_{i,j} = (a_{2i,2j} - a_{2i+1,2j} + a_{2i,2j+1} - a_{2i+1,2j+1})/4$$
$$b^4_{i,j} = (a_{2i,2j} - a_{2i+1,2j} - a_{2i,2j+1} + a_{2i+1,2j+1})/4$$

Используя эти формулы, мы для изображения 512x512 пикселов получим после первого преобразования 4 матрицы размером 256x256 элементов (рисунок 11).

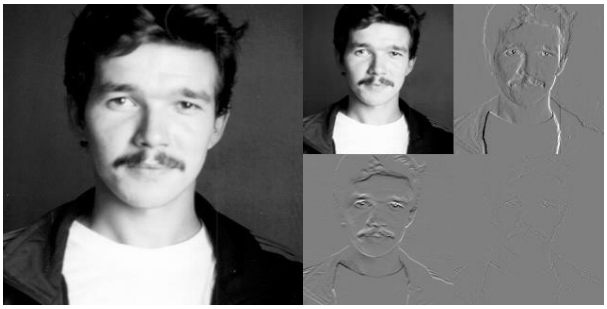


Рис. 11. Рекурсивный алгоритм

В первой, как легко догадаться, будет храниться уменьшенная копия изображения. Во второй — усредненные разности пар значений пикселов по горизонтали. В третьей — усредненные разности пар значений пикселов по вертикали. В четвертой — усредненные разности значений пикселов по диагонали. По аналогии с двумерным случаем мы можем повторить наше преобразование и получить вместо первой матрицы 4 матрицы размером 128x128. Повторив наше преобразование в третий раз, мы получим в итоге: 4 матрицы 64x64, 3 матрицы 128x128 и 3 матрицы 256x256. На практике при записи в файл, значениями, получаемыми в последней строке ($b^4_{i,j}$), обычно пренебрегают (сразу получая выигрыш примерно на треть размера файла — 1- 1/4 - 1/16 - 1/64...).

К достоинствам этого алгоритма можно отнести то, что он очень легко позволяет реализовать возможность постепенного «проявления» изображения при передаче изображения по сети. Кроме того, поскольку в начале изображения мы фактически храним его уменьшенную копию, упрощается показ «огрубленного» изображения по заголовку.

В отличие от JPEG и фрактального алгоритма данный метод не оперирует блоками, например, 8x8 пикселов. Точнее, мы оперируем блоками 2x2, 4x4, 8x8 и т.д. Однако за счет того, что коэффициенты для этих блоков мы сохраняем независимо, мы можем достаточно легко избежать дробления изображения на «мозаичные» квадраты.

Алгоритм JPEG 2000

Алгоритм JPEG-2000 разработан той же группой экспертов в области фотографии, что и JPEG. Формирование JPEG как международного стандарта было закончено в 1992 году. В 1997 стало ясно, что необходим новый, более гибкий и мощный стандарт, который и был доработан к зиме 2000 года. Основные отличия алгоритма в JPEG 2000 от алгоритма в JPEG заключаются в следующем:

1. Лучшее качество изображения при сильной степени сжатия. Или, что то же самое, большая степень сжатия при том же качестве для высоких степеней сжатия. Фактически это означает заметное уменьшение размеров графики «Web-качества», используемой большинством сайтов.
2. Поддержка кодирования отдельных областей с лучшим качеством. Известно, что отдельные области изображения критичны для восприятия человеком (например, глаза на фотографии), в то время как качеством других можно пожертвовать (например, задний

план). При «ручной» оптимизации увеличение степени сжатия проводится до тех пор, пока не будет потеряно качество в какой-то важной части изображения. Сейчас появляется возможность задать качество в критичных областях, сжав остальные области сильнее, т.е. мы получаем еще большую окончательную степень сжатия при субъективно равном качестве изображения.

3. Основной алгоритм сжатия заменен на wavelet. Помимо указанного повышения степени сжатия это позволило избавиться от 8-пиксельной блочности, возникающей при повышении степени сжатия. Кроме того, плавное проявление изображения теперь изначально заложено в стандарт (Progressive JPEG, активно применяемый в Интернет, появился много позднее JPEG).
4. Для повышения степени сжатия в алгоритме используется арифметическое сжатие. Изначально в стандарте JPEG также было заложено арифметическое сжатие, однако позднее оно было заменено менее эффективным сжатием по Хаффману, поскольку арифметическое сжатие было защищено патентами. Сейчас срок действия основного патента истек, и появилась возможность улучшить алгоритм.
5. Поддержка сжатия без потерь. Помимо привычного сжатия с потерями новый JPEG теперь будет поддерживать и сжатие без потерь. Таким образом, становится возможным использование JPEG для сжатия медицинских изображений, в полиграфии, при сохранении текста под распознавание OCR системами и т.д.
6. Поддержка сжатия однобитных (2-цветных) изображений. Для сохранения однобитных изображений (рисунки тушью, отсканированный текст и т.п.) ранее повсеместно рекомендовался формат GIF, поскольку сжатие с использованием ДКП весьма неэффективно к изображениям с резкими переходами цветов. В JPEG при сжатии 1-битная картинка приводилась к 8-битной, т.е. увеличивалась в 8 раз, после чего делалась попытка сжимать, нередко менее чем в 8 раз. Сейчас можно рекомендовать JPEG 2000 как универсальный алгоритм.
7. На уровне формата поддерживается прозрачность. Плавно накладывать фон при создании WWW страниц теперь можно будет не только в GIF, но и в JPEG 2000. Кроме того, поддерживается не только 1 бит прозрачности (пиксел прозрачен/непрозрачен), а отдельный канал, что позволит задавать плавный переход от непрозрачного изображения к прозрачному фону.

Кроме того, на уровне формата поддерживаются включение в изображение информации о копирайте, поддержка устойчивости к битовым ошибкам при передаче и широковещании, можно запрашивать для декомпрессии или обработки внешние средства (plug-ins), можно включать в изображение его описание, информацию для поиска и т.д.

Базовая схема JPEG-2000 очень похожа на базовую схему JPEG. Отличия заключаются в следующем:

1. Вместо дискретного косинусного преобразования (DCT) используется дискретное вэйвлет-преобразование (DWT).
2. Вместо кодирования по Хаффману используется арифметическое сжатие.
3. В алгоритм изначально заложено управление качеством областей изображения.
4. Не используется явно дискретизация компонент U и V после преобразования цветовых пространств, поскольку при DWT можно достичь того же результата, но более аккуратно.

Рис. 12.Конвейер операций, используемый в алгоритме JPEG-2000.

Шаг 1. В JPEG-2000 предусмотрен сдвиг яркости (DC level shift) каждой компоненты (RGB) изображения перед преобразованием в YUV. Это делается для выравнивания динамического диапазона (приближения к 0 гистограммы частот), что приводит к увеличению степени сжатия. Формулу преобразования можно записать как:

$$I'(x,y) = I(x,y) - 2^{ST-1}$$

Значение степени ST для как каждой компоненты R, G и B свое (определяется при сжатии компрессором). При восстановлении изображения выполняется обратное преобразование:

$$I'(x,y) = I(x,y) + 2^{ST-1}$$

Шаг 2. Переводим изображение из цветового пространства RGB, с компонентами, отвечающими за красную (Red), зеленую (Green) и синюю (Blue) составляющие цвета точки, в цветовое пространство YUV. Этот шаг аналогичен JPEG (см. матрицы преобразования в описании JPEG), за тем исключением, что кроме преобразования с потерями предусмотрено также и преобразование без потерь. Его матрица выглядит так:

$$\begin{vmatrix} Y \\ U \\ V \end{vmatrix} = \begin{pmatrix} \left\lfloor \frac{R + 2G + B}{4} \right\rfloor \\ R - G \\ B - G \end{pmatrix}$$

Обратное преобразование осуществляется с помощью обратной матрицы:

$$\begin{vmatrix} R \\ G \\ B \end{vmatrix} = \begin{vmatrix} U + G \\ Y - \left\lfloor \frac{U + V}{4} \right\rfloor \\ V + G \end{vmatrix}$$

Шаг 3.Дискретное wavelet преобразование (DWT) также может быть двух видов — для случая сжатия с потерями и для сжатия без потерь. Его коэффициенты задаются таблицами.

Само преобразование в одномерном случае представляет собой скалярное произведение коэффициентов фильтра на строку преобразуемых значений (в нашем случае — на строку изображения). При этом четные выходящие значения формируются с помощью низкочастотного преобразования, а нечетные с помощью высокочастотного:

$$y_{output}(2n) = \sum_{j=0}^{N-1} x_{input}(j) \cdot h_H(j - 2n)$$

$$y_{output}(2n+1) = \sum_{j=0}^{N-1} x_{input}(j) \cdot h_L(j - 2n - 1)$$

Поскольку большинство $h_L(i)$, кроме окрестности i=0, равны 0, то можно переписать приведенные формулы с меньшим количеством операций. Для простоты рассмотрим случай сжатия без потерь.

$$y_{out}(2n) = \frac{-x_{in}(2n-1) + 2 \cdot x_{in}(2n) + 6 \cdot x_{in}(2n+1) + 2 \cdot x_{in}(2n+2) - x_{in}(2n+3)}{8}$$

$$y_{out}(2n+1) = -\frac{x_{in}(2n)}{2} + x_{in}(2n+1) - \frac{x_{in}(2n+2)}{2}$$

Легко показать, что данную запись можно эквивалентно переписать, уменьшив еще втрое количество операций умножения и деления (однако теперь необходимо будет подсчитать сначала все нечетные y). Добавим также операции округления до ближайшего целого, не превышающего заданное число a, обозначаемые как $\lfloor a \rfloor$:

$$y_{out}(2n+1) = x_{in}(2n+1) - \left\lfloor \frac{x_{in}(2n) + x_{in}(2n+2)}{2} \right\rfloor$$

$$y_{out}(2n) = x_{in}(2n) + \left\lfloor \frac{y_{out}(2n-1) + y_{out}(2n+1) + 2}{4} \right\rfloor$$

Рассмотрим на примере, как работает данное преобразование. Для того, чтобы преобразование можно было применять к крайним пикселам изображения, оно симметрично достраивается в обе стороны на несколько пикселов, как показано на рисунке ниже. В худшем случае (сжатие с потерями) нам необходимо достроить изображение на 4 пиксела (рисунок 13).

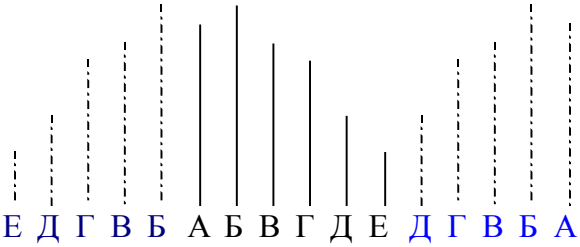


Рис. 12. Симметричное расширение изображения (яркости АБ…Е) по строке вправо и влево

Пусть мы преобразуем строку из 10 пикселов. Расширим ее значения вправо и влево и применим DWT преобразование:

n	-2	-1	0	1	2		4	5	6	7	8	9	10	11
x_{in}	3	2	1	2	3	7	10	15	12	9	10	5	10	9
y_{out}		0	1	0	3	1	11	4	13	-2	8	-5		

Получившаяся строка 1, 0, 3, 1, 11, 4, 13, -2, 8, -5 и является цепочкой, однозначно задающей исходные данные. Совершив аналогичные преобразования с коэффициентами для распаковки, приведенными выше в таблице, получим необходимые формулы:

$$x_{out}(2n) = y_{out}(2n) - \left\lfloor \frac{y_{out}(2n-1) + y_{out}(2n+1) + 2}{4} \right\rfloor$$

$$x_{out}(2n+1) = y_{out}(2n+1) + \left\lfloor \frac{x_{out}(2n) + x_{out}(2n+2)}{2} \right\rfloor$$

Легко проверить (используя преобразование упаковки), что значения на концах строк в y_{out} также симметричны относительно n=0 и 9. Воспользовавшись этим свойством, расширим нашу строку вправо и влево и применим обратное преобразование:

n	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11
y_{out}		0	1	0	3	1	11	4	13	-2	8	-5	8	-2
x_{out}			1	2	3	7	10	15	12	9	10	5	10	

Как видим, мы получили исходную цепочку (x_{in}= x_{out}).

Далее к строке применяется чересстрочное преобразование, суть которого заключается в том, что все четные коэффициенты переписываются в начало строки, а все нечетные — в конец. В результате этого преобразования в начале строки формируется «уменьшенная копия» всей строки (низкочастотная составляющая), а в конце строки — информация о колебаниях значений промежуточных пикселов (высокочастотная составляющая).

y_{out}	1		3	1	11	4	13	-2	8	-5
y'_{out}	1	3	11	13	8	0	1	4	-2	-5

Это преобразование применяется сначала ко всем строкам изображения, а затем ко всем столбцам изображения. В результате изображение делится на 4 квадранта (примеры смотрите в описании рекурсивного сжатия). В первом квадранте будет сформирована уменьшенная копия изображения, а в остальных трех — высокочастотная информация. По-

сле чего преобразование повторно применяется уже только к первому квадранту изображения по тем же правилам (преобразование второго уровня).

Для корректного сохранения результатов под данные 2 и 3 квадрантов выделяется на один бит больше, а под данные 4-го квадранта — на 2 бита больше. Т.е. если исходные данные были 8-битные, то на 2 и 3 квадранты нужно 9 бит, а на 4-й — 10, независимо от уровня применения DWT. При записи коэффициентов в файл можно использовать иерархическую структуру DWT, помещая коэффициенты преобразований с большего уровня в начало файла. Это позволяет получить «изображение для предварительного просмотра», прочитав небольшой участок данных из начала файла, а не распаковывая весь файл, как это приходилось делать при сжатии изображения целиком. Иерархичность преобразования может также использоваться для плавного улучшения качества изображения при передаче его по сети.

Шаг 4.Так же, как и в алгоритме JPEG, после DWT применяется квантование. Коэффициенты квадрантов делятся на заранее заданное число. При увеличении этого числа снижается динамический диапазон коэффициентов, они становятся ближе к 0, и мы получаем б**о**льшую степень сжатия. Варьируя эти числа для разных уровней преобразования, для разных цветовых компонент и для разных квадрантов, мы очень гибко управляем степенью потерь в изображении. Рассчитанные в компрессоре оптимальные коэффициенты квантования передаются в декомпрессор для однозначной распаковки.

Шаг 5. Для сжатия получающихся массивов данных в JPEG 2000 используется вариант арифметического сжатия, называемый MQ-кодер, прообраз которого (QM-кодер) рассматривался еще в стандарте JPEG, но реально не использовался из-за патентных ограничений. Подробнее об алгоритме арифметического сжатия читайте в соответствующей главе раздела.

Основная задача, которую мы решаем — повышение степени сжатия изображений. Когда практически достигнут предел сжатия изображения в целом и различные методы дают очень небольшой выигрыш, мы можем существенно (в разы) увеличить степень сжатия за счет изменения качества разных участков изображения.

Проблемой этого подхода является то, что необходимо каким-то образом получать расположение наиболее важных для человека участков изображения. Например, таким участком на фотографии человека является лицо, а на лице — глаза. Если при сжатии портрета с большими потерями будут размыты предметы, находящиеся на заднем плане — это будет несущественно. Однако, если будет размыто лицо или глаза — экспертная оценка степени потерь будет хуже.

Работы по автоматическому выделению таких областей активно ведутся. В частности, созданы алгоритмы автоматического выделения лиц на изображениях. Продолжаются исследования методов выделения наиболее значимых (при анализе изображения мозгом человека) контуров и т.д. Однако очевидно, что универсальный алгоритм в ближайшее время создан не будет, поскольку для этого требуется построить полную схему восприятия изображений мозгом человека.

На сегодня вполне реально применение полуавтоматических систем, в которых качество областей изображения будет задаваться интерактивно. Данный подход уменьшает количество возможных областей применения модифицированного алгоритма, но позволяет достичь большей степени сжатия.

Такой подход логично применять, если:

1. Для приложения должна быть критична (максимальна) степень сжатия, причем настолько, что возможен индивидуальный подход к каждому изображению.
2. Изображение сжимается один раз, а разжимается множество раз.

В качестве примеров приложений, удовлетворяющим этим ограничениям, можно привести практически все мультимедийные продукты на CD-ROM. И для CD-ROM энциклопедий, и для игр важно записать на диск как можно больше информации, а графика, как прави-

ло, занимает до 70% всего объема диска. При этом технология производства дисков позволяет сжимать каждое изображение индивидуально, максимально повышая степень сжатия.

Интересным примером являются WWW-сервера. Для них тоже, как правило, выполняются оба изложенных выше условия. При этом совершенно не обязательно индивидуально подходить к каждому изображению, поскольку по статистике 10% изображений будут запрашиваться 90% раз. Т.е. для крупных справочных или игровых серверов появляется возможность уменьшать время загрузки изображений и степень загруженности каналов связи адаптивно.

В JPEG-2000 используется однобитное изображение-маска, задающее повышение качества в данной области изображения. Поскольку за качество областей у нас отвечают коэффициенты DWT преобразования во 2, 3 и 4 квадрантах, то маска преобразуется таким образом, чтобы указывать на все коэффициенты, соответствующие областям повышения качества.

Эти области обрабатываются далее другими алгоритмами (с меньшими потерями), что и позволяет достичь искомого баланса по общему качеству и степени сжатия.

АЛГОРИТМЫ СЖАТИЯ ВИДЕО

Основной сложностью при работе с видео являются большие объемы дискового пространства, необходимого для хранения даже небольших фрагментов. Причем даже применение современных алгоритмов сжатия не изменяет ситуацию кардинально. При записи на один компакт-диск «в бытовом качестве», на него можно поместить несколько тысяч фотографий, примерно 10 часов музыки и всего полчаса видео. Видео «телевизионного» формата 720x576 пикселов 25 кадров в секунду в системе RGB требует потока данных примерно в 240 Мбит/сек (т.е. 1.8 Гб в минуту). При этом традиционные алгоритмы сжатия изображений, ориентированные на отдельные кадры, не спасают ситуации, поскольку даже при уменьшении потока в 10 раз он составляет достаточно большие величины.

В результате подавляющее большинство сегодняшних алгоритмов сжатия видео являются алгоритмами с потерей данных. При сжатии используется несколько типов избыточности:

1.Когерентность областей изображения — малое изменение цвета изображения в соседних пикселах (свойство, которое эксплуатируют все алгоритмы сжатия изображений с потерями).

2.Избыточность в цветовых плоскостях — используется большая важность яркости изображения для восприятия.

3.Подобие между кадрами — использование того факта, что на скорости 25 кадров в секунду, как правило, соседние кадры изменяются незначительно.

Первые два пункта знакомы вам по алгоритмам сжатия графики. Использование подобия между кадрами в самом простом и наиболее часто используемом случае означает кодирование не самого нового кадра, а его разности с предыдущим кадром. Для видео типа «говорящая голова» (передача новостей, видеотелефоны), большая часть кадра остается неизменной, и даже такой простой метод позволяет значительно уменьшить поток данных. Более сложный метод заключается в нахождении для каждого блока в сжимаемом кадре наименее отличающегося от него блока в кадре, используемом в качестве базового. Далее кодируется разница между этими блоками. Этот метод существенно более ресурсоемкий.

Определимся с основными понятиями, которые используются при сжатии видео. Видеопоток характеризуется разрешением, частотой кадров и системой представления цветов. Из телевизионных стандартов пришли разрешения в 720x576 и 640x480, и частоты в 25 (стандарты PAL или SECAM) и 30 (стандарт NTSC) кадров в секунду. Для низких разрешений существуют специальные названия CIF — Common Interchange Format, равный 352x288 и QCIF— Quartered Common Interchange Format, равный 176x144. Поскольку CIF и QCIF

ориентированы на крайне небольшие потоки, то с ними работают на частотах от 5 до 30 кадров в секунду.

В 1988 году в рамках Международной Организации по Стандартизации (ISO) начала работу группа MPEG (Moving Pictures Experts Group) – группа экспертов в области цифрового видео (ISO-IEC/JTC1/SC2/WG11/MPEG). Группа работала в направлениях, которые можно условно назвать MPEG-Video– сжатие видеосигнала в поток со скоростью до 1,5 Мбит/сек, MPEG-Audio — сжатие звука до 64, 128 или 192 Кбит/сек на канал и MPEG-System – синхронизация видео и аудио потоков. Нас в основном будут интересовать достижения MPEG-Video, хотя очевидно, что многие решения в этом направлении принимались с учетом требований синхронизации.

Как алгоритм, MPEG имеет несколько предшественников. Это, прежде всего, универсальный алгоритм JPEG. Его универсальность означает, что JPEG показывает неплохие результаты на широком классе изображений.

Если быть более точным, то стандарт MPEG, как и другие стандарты на сжатие, описывает лишь выходной битовый поток, неявно задавая алгоритмы кодирования и декодирования. При этом их реализация перекладывается на программистов-разработчиков. Такой подход открывает широкие горизонты для тех, кто желает оптимально реализовать алгоритм для конкретного вычислительного устройства (контроллера, ПК, распределенной вычислительной системы), операционной системы, видеокарты и т.п. При специализированных реализациях могут быть учтены весьма специфические требования на время работы, расход памяти и качество получаемых изображений. Алгоритмы сжатия видео весьма гибки и зачастую для разных подходов к реализации, можно получить существенную разницу по *качеству* видео, при одной и той же степени сжатия. Более того – для одного и того же сжатого файла с помощью разных алгоритмов декодирования можно получить существенно различающиеся по визуальному качеству фильмы. Зачастую «простая» реализация стандарта дает дергающий видеоряд с хорошо заметными блоками, в то время как программы известных производителей проигрывают этот же файл вполне плавно и без бросающейся в глаза блочности. Эти нюансы необходимо хорошо себе представлять, когда речь заходит о сравнении разных различных стандартов.

В сентябре 1990 был представлен предварительный стандарт кодирования MPEG-1. В январе 1992 работа над MPEG-1 была завершена, и начата работа над MPEG-2, в задачу которого входило описание потока данных со скоростью от 3 до 10 Мбит/сек. Практически в то же время была начата работа над MPEG-3, который был предназначен для описания потоков 20-40 Мбит/сек. Однако вскоре выяснилось, что алгоритмические решения для MPEG-2 и MPEG-3 принципиально близки и можно безболезненно расширить рамки MPEG-2 до потоков в 40 Мбит/сек. В результате работа над MPEG-3 была прекращена. MPEG-2 был окончательно доработан к 1995 году.

В 1991 группой экспертов по видеотелефонам (EGVT) при Международном консультативный комитет по телефонии и телеграфии (CCITT) предложен стандарт видеотелефонов px64 Kbits. Запись px64 означает, что алгоритм ориентирован на параллельную передачу оцифрованного видеоизображения по p каналам с пропускной способностью 64 Кбита/сек. Таким образом, захватывая несколько телефонных линий, можно получать изображение вполне приемлемого качества. Одним из главных ограничений при создании алгоритма являлось время задержки, которое должно было составлять не более 150 мс. Кроме того, уровень помех в телефонных каналах достаточно высок, и это, естественно, нашло отражение в алгоритме. Можно считать, что px64 Kbits – предшественник MPEG-а для потоков данных менее 1,5 Мбит/сек и специфического класса видео.

В группе при CMTT (совместный комитет при CCITT и CCIR – International Consultative Committee on b Roadcasting) работы были направлены на передачу оцифрованного видео по выделенным каналам с высокой пропускной способностью и радиолиниям. Соответству-

ющие стандарты H21 и H22 ориентированы на 34 и 45 Мбит/сек, и сигнал передается с очень высоким качеством.

MPEG-4 изначально был задуман как стандарт для работы со сверхнизкими потоками. Однако в процессе довольно долгой подготовки стандарт претерпел совершенно революционные изменения и сейчас собственно сжатие с низким потоком входит в него как одна составная часть, причем достаточно небольшая по размеру. Например, сам формат сегодня включает в себя такие вещи, как синтез речи, рендеринг изображений и описания параметров визуализации лица на стороне программы просмотра.

Разработка MPEG-7 была начата в 1996. Собственно к алгоритмам сжатия видео этот стандарт имеет еще меньшее отношение, чем MPEG-4, поскольку его основная задача заключается в описании контента и управлении им.

Параллельно все это время существовали форматы Motion-JPEG и недавно появившийся Motion-JPEG2000, предназначенные в основном для удобства обработки сжатого видео.

Алгоритм MPEG

Технология сжатия видео в MPEG распадается на две части: уменьшение избыточности видеоинформации во временном измерении, основанное на том, что соседние кадры, как правило, отличаются не сильно, и сжатие отдельных изображений.

Для того чтобы удовлетворить противоречивым требованиям и увеличить гибкость алгоритма, рассматривается четыре типа кадров:

1.I-кадры – кадры сжатые независимо от других кадров (I-Intra pictures);

2.P-кадры – сжатые с использованием ссылки на одно изображение (P-Predicted);

3.B-кадры – сжатые с использованием ссылки на два изображения (B-Bidirection);

4.DC-кадры – независимо сжатые с большой потерей качества (используются только при быстром поиске).

I-кадры обеспечивают возможность произвольного доступа к любому кадру, являясь своеобразными входными точками в поток данных для декодера. P-кадры используют при архивации ссылку на один I- или P-кадр, повышая тем самым степень сжатия фильма в целом. B-кадры, используя ссылки на два кадра, находящихся впереди и позади, обеспечивают наивысшую степень сжатия. Сами в качестве ссылки использоваться не могут. Последовательность кадров в фильме может быть, например, такой: IBBPBBPBBPBBIBBPBB.

Частота I-кадров выбирается в зависимости от требований на время произвольного доступа и надежности потока при передаче через канал с ошибками. Соотношение P- и B-кадров подбирается, исходя из требований к величине компрессии и ограничений декодера. Как правило, декодирование B-кадров требует больше вычислительных мощностей, однако позволяет повысить степень сжатия. Именно варьирование частоты кадров разных типов обеспечивает алгоритму необходимую гибкость и возможность расширения. Понятно, что для того, чтобы распаковать B-кадр, мы должны уже распаковать те кадры, на которые он ссылается. Поэтому для последовательности IBBPBBPBBPBBIBBPBB кадры в фильме будут записаны так: 0**312645..., где цифры – номера кадров, а звездочкам соответствуют либо B-кадры с номерами -1 и -2, если мы находимся в середине потока, либо пустые кадры (ничего), если мы в начале фильма. Подобный формат обладает достаточно большой гибкостью и способен удовлетворять самым различным наборам требований.

Одним из основных понятий при сжатии нескольких изображений является понятие макроблока. При сжатии кадр из цветового пространства RGB переводится в цветовое пространство YUV. Каждая из плоскостей сжимаемого изображения (Y, U, V) разделяется на блоки 8x8, с которыми работает ДКП. Причем плоскости U и V, соответствующие компоненте цветности берутся с разрешением в два раза меньшим (по вертикали и горизонтали), чем исходное изображение. Таким образом, мы сразу получаем сжатие в два раза, пользуясь тем, что глаз человека хуже различает цвет отдельной точки изображения, чем ее яркость (по-

дробнее об этих преобразованиях смотрите в описании алгоритма JPEG). Блоки 8x8 группируются в макроблоки. Макроблок – это группа из четырех соседних блоков в плоскости яркостной компоненты Y (матрица пикселов 16x16 элементов) и два соответствующих им по расположению блока из плоскостей цветности U и V. Таким образом, кадр разбивается на независимые единицы, несущие полную информацию о части изображения. При этом размер изображения должен быть кратен 16.

Отдельные макроблоки сжимаются независимо, т.е. в B-кадрах мы можем сжать макроблок конкретный как I-блок, P-блок со ссылкой на предыдущий кадр, P-блок со ссылкой на последующий кадр и, наконец, как B-блок.

Алгоритм сжатия отдельных кадров в MPEG похож на соответствующий алгоритм для статических изображений – JPEG. Если говорить коротко, то сам алгоритм сжатия представляет собой конвейер преобразований. Это дискретное косинусное преобразование исходной матрицы 8x8, квантование матрицы и вытягивание ее в вектор v11,v12,v21,v31,v22,...,v88 (зигзаг-сканирование), сжатие вектора групповым кодированием и, наконец, сжатие по алгоритму Хаффмана.

В целом весь конвейер преобразований можно представить так:

1) Подготовка макроблоков. Для каждого макроблока определяется, каким образом он будет сжат. В I-кадрах все макроблоки сжимаются независимо. В P-кадрах блок либо сжимается независимо, либо представляет собой разность с одном из макроблоков в предыдущем опорном кадре, на который ссылается P-кадр.
2) Перевод макроблока в цветовое пространство YUV. Получение нужного количества матриц 8x8.
3) Для P-блоков и B-блоков производится вычисление разности с соответствующим макроблоком в опорном кадре.
4) ДКП
5) Квантование.
6) Зигзаг-сканирование.
7) Групповое кодирование.
8) Кодирование Хаффмана.

При декодировании весь конвейер повторяется для обратных преобразований, начиная с конца.

Алгоритм Motion-JPEG

Motion-JPEG является наиболее простым алгоритмом сжатия видео. В нем каждый кадр сжимается независимо алгоритмом JPEG. Этот прием дает высокую скорость доступа к произвольным кадрам, как в прямом, так и в обратном порядке следования. Соответственно легко реализуются плавные «перемотки» в обоих направлениях, аудио-визуальная синхронизация и, что самое главное – редактирование. Типичные операции JPEG сейчас поддерживаются на аппаратном уровне большинством видеокарт и данный формат позволяет легко оперировать большими объемами данных при монтаже фильмов. Независимое сжатие отдельных кадров позволяет накладывать различные эффекты, не опасаясь, что взаимное влияние соседних кадров внесет дополнительные искажения в фильм.

Алгоритм H.261

Стандарт H.261специфицирует кодирование и декодирование видеопотока для передачи по каналу p*64 Кбит, где p=1..30. В качестве канала может выступать, например, несколько телефонных линий.

Входной формат изображения — разрешения CIF или QCIF в формате YUV (CCIR 601) частота кадров от 30 fps и ниже. Используется уменьшение разрешения в 2 раза для компонент цветности.

В выходной поток записываются два типа кадров: INTRA — сжатые независимо (соответствуют I-кадрам) и INTER - сжатые со ссылкой на предыдущий кадр (соответствуют P-кадрам). В передаваемом кадре не обязательно присутствуют все макроблоки изображения, если блок изменился незначительно передавать его обычно нет смысла. Сжатие в INTRA кадрах осуществляется по схеме сжатия отдельного изображения. В INTER кадрах производится аналогичное сжатие разности каждого передаваемого макроблока с «наиболее похожим» макроблоком из предыдущего кадра (компенсация движения). Для сглаживания артефактов ДКП предусмотрена возможность применения размытия внутри каждого блока 8x8 пикселей. Стандарт требует, чтобы INTRA кадры встречались в потоке не реже чем через каждые 132 INTER кадра (чтобы не накапливалась погрешность кодирования и была возможность восстановиться в случае ошибки в потоке).

Степень сжатия зависит в основном от метода нахождения «похожих» макроблоков в предыдущем кадре, алгоритма решения передавать ли конкретный макроблок, выбора способа кодирования каждого макроблока (INTER/INTRA) и выбора коэффициентов квантования результатов ДКП. Ни один из перечисленных вопросы стандартом не регламентируются, оставляя свободу для построения собственных оптимальных алгоритмов.

Алгоритм MPEG-2

Как уже говорилось, MPEG-2 занимается сжатием оцифрованного видео при потоке данных от 3 до 10 Мбит/сек. Многое в нем заимствовано из формата CCIR-601. CCIR-601 представляет собой стандарт цифрового видео с размером передаваемого изображения 720x486 при 60 полукадрах в секунду. Строки изображения передаются с чередованием, и два полукадра составляют кадр. Этот прием нередко применяют для уменьшения мерцания. Хроматические каналы (U и V в YUV) передаются размером 360x243 60 раз в секунду и также чередуются уже между собой. Подобное деление называется 4:2:2. Перевод из CCIR-601 в MPEG-I прост: надо поделить в 2 раза яркостную компоненту по горизонтали, поделить поток в 2 раза во временном измерении (убрав чередование), добавить вторую хроматическую компоненту и выкинуть «лишние» строки, чтобы размер по вертикали делился на 16. Мы получим поток YUV кадров размером 352x240 с частотой 30 кадров в секунду. Здесь все просто.

Проблемы начинаются, когда появляется возможность увеличить поток данных и довести качество изображения до CCIR-601. Это не такая простая задача, как кажется. Проблема состоит в чередовании полукадров во входном формате. Тривиальное решение — работать с кадрами 720x486 при 30 кадрах в секунду, как с обычным видео. Этот путь приводит к неприятным эффектам при быстром движении объектов на экране. Между двумя исходными полукадрами 720x243 сдвиг становится заметным, а т.к. наш кадр формируется из исходных полукадров через строку, то при сжатии происходит размывание движущегося объекта. Виновно в этом эффекте ДКП, и как-то исправить ситуацию, не уменьшив степени сжатия видео, или не потеряв в визуальном качестве нельзя. Достаточно распространенным является применение «деинтерлейсинга» (от английского deinterlacing — удаление чередования строк). Эта операция позволяет удалить чередование, смещая четные строки в одном направлении, а нечетные в другом, пропорционально относительному движению объекта в данной области экрана. В результате мы получаем визуально более качественное изображение, но несколько более длительную предобработку перед сжатием.

Другим решением является архивация четных и нечетных кадров в потоке CCIR-601 независимо. При этом мы, конечно, избавимся от артефактов, возникающих при быстром движении объектов, но существенно уменьшим степень сжатия, т.к. не будем использовать важнейшей вещи –избыточности между соседними кадрами, которая очень велика.

Алгоритм MPEG-4

MPEG-4 кардинально отличается от принимаемых ранее стандартов. В состав декодера MPEG-4 как составная часть входит блок визуализации трехмерных объектов (Animation

Framework eXtension — AFX — то, что в просторечии называют данными для трехмерного движка). Те, кто кодировал видео, знают, сколько проблем доставляют титры и вообще любые накладываемые поверх фильма объекты (логотипы, заставки и т.п.). Если хорошо выглядит основной план — будут подпорчены накладываемые объекты, если хорошо смотрятся они — будет низкой общая степень сжатия. В MPEG-4 предлагается решить проблему кардинально. Накладываемые объекты рассчитываются отдельно и накладываются потом. Кроме того, можно использовать видеопоток даже как текстуру, накладываемую на поверхности рассчитываемых объектов. Такая гибкая работа с трехмерными объектами позволяет существенно поднять степень сжатия при заметно лучшем качестве изображения. Более того — никто не мешает делать видеоролики вообще без живого видео, а состоящие только из рассчитанных (синтетических) объектов. Размер их описания будет в разы меньше, чем размер аналогичных фильмов, сжатых просто как поток кадров. Кстати, отдельно в стандарте предусмотрена работа со «спрайтами» — статическими изображениями, накладываемыми на кадр. При этом размер спрайта может быть как совсем маленький (логотип канала в уголке экрана), так и превышать размер кадра и «прокручиваться» (т.е. в качестве «спрайта» может быть задан фон, а небольшие видео-объекты, например, голову диктора, будут на него накладывать). Это дает значительную гибкость при создании MPEG-4 фильмов и позволяет заметно уменьшить объем кодируемой информации.

Объектно-ориентированная работа с потоком данных. Теперь работа с потоком данных становится объектно-ориентированной. При этом данные могут быть живым видео, звуковыми данными, синтетическими объектами и т.д. Из них создаются сцены, этими сценами можно управлять. Для простых смертных при этом мало что изменится, однако для программистов объектная среда означает кардинальное упрощение работы с возникающими сложными структурами.

Помещение в поток двоичного кода “C++ подобного” языка BIFS. С помощью BIFS в поток добавляются описания объектов, классов объектов и сцен. Также на нем можно менять координаты, размеры, свойства, поведение и реакцию объектов на действия пользователя. В свое время Flash был назван революцией 2D графики в Интернете. Аналогичный прорыв в области видео совершает MPEG-4.

Активная зрительская позиция. Как было замечено выше, BIFS позволяет задавать реакцию объектов сцены на действия пользователя. Потенциально возможно удаление, добавление или перемещение объектов, ввод команд с клавиатуры. Событийная модель заимствована из развивавшегося уже долгое время языка моделирования виртуальной реальности VRML. Для тех, кто играл в написанные на VRML игры, очевидно, что в MPEG-4 будет совершенно реально создавать «квест»-подобные (и не только) игры. Широчайший простор открывается для создания обучающих и развлекательных программ. Представляете, скачиваете из Интернета один файл, который сразу в себе содержит все, что необходимо для небольшого курса лекций, причем вы можете прослушать его, видя говорящую голову преподавателя, или отключив его, можете увеличить фрагменты («спрайты») с материалами. А в конце — пройти короткий тест на понимание предмета. Кстати — в стандарте предусмотрено обработка команд на стороне сервера, т.е. программа-просмотрщик может отослать данные на сервер и получить оттуда оценку. Отличие от предыдущих стандартов революционное.

Синтезатор лиц и фигур. В стандарт заложен интерфейс к модулю синтеза лиц и фигур. Например, в файле сохраняются ключевые данные о профиле лица и текстуры лица, а при записи фильма сохраняются только коэффициенты изменения формы. Для передач типа новостей, этот прием позволяет в десятки раз сократить размер файла при замечательном качестве.

Синтезатор звуков и речи. Помимо синтеза лиц в стандарт MPEG-4 также заложены алгоритмы синтеза звуков, и даже речи.

Улучшенные алгоритмы сжатия видео. В стандарте предусмотрены блоки, отвечающие за потоки 4.8-65Кбит/с с прогрессивной разверткой и большие потоки с поддержкой черес-

строчной развертки. Для передачи по ненадежным каналам возможно использование помехоустойчивых методов кодирования (за счет незначительного увеличения объема передаваемых данных резко снижается вероятность искажения изображения). При передаче видео с одновременным просмотром заложена возможность огрубить изображение, если декодер из-за ограничений канала связи не успевает получить всю информацию. Всего в стандарт заложено 3 уровня детализации. Эта возможность позволит легко адаптировать алгоритм для трансляций видео по сети.

Поддержка профилей на уровне стандарта. Понятно, что реализация всех возможностей стандарта превращает декодер в весьма сложную и большую конструкцию. При этом далеко не для всех приложений необходимы какие-то сложные специфические функции (например, синтез речи). Создатели стандарта поступили просто: они оговорили наборы профилей, каждый из которых включает в себя набор обязательных функций. Если в фильме записано, что ему для проигрывания необходим такой-то профиль и декодер этот профиль поддерживает, то стандарт гарантирует, что фильм будет проигран правильно.

Выше кратко перечислены некоторые отличия MPEG-4 от предыдущих стандартов. Надо отметить, что на момент создания стандарта острой потребности в описанных выше вещах еще не было. Иначе говоря, мы имеем дело с хорошо продуманной работой по формированию стандарта, которая была закончена к тому времени, как в нем возникла первая необходимость.

Создателями MPEG-4 учтен опыт предшественников (в частности VRML), когда слишком раннее появление стандарта и отсутствие в нем механизма профилей серьезно подорвало его массовое применение.

КОДЫ ДЛЯ ОБНАРУЖЕНИЯ ОШИБОК

Коды, которые обеспечивают возможность обнаружения и исправления ошибки, называют помехоустойчивыми.

Эти коды используют для:

1) исправления ошибок – корректирующие коды;

2) обнаружения ошибок.

Корректирующие коды основаны на введении избыточности. У подавляющего большинства помехоустойчивых кодов помехоустойчивость обеспечивается их алгебраической структурой. Поэтому их называют алгебраическими кодами.

Алгебраические коды подразделяются на два класса:

1) блоковые;

2) непрерывные.

В случае блоковых кодов процедура кодирования заключается в сопоставлении каждой букве сообщения (или последовательности из k символов, соответствующей этой букве) блока из n символов.

В операциях по преобразованию принимают участие только указанные k символов, и выходная последовательность не зависит от других символов в передаваемом сообщении. Блоковый код называют равномерным, если n остается постоянным для всех букв сообщения. Различают разделимые и неразделимые блоковые коды.

При кодировании разделимыми кодами выходные последовательности состоят из символов, роль которых может быть отчетливо разграничена. Это информационные символы, совпадающие с символами последовательности, поступающей на вход кодера канала, и избыточные (проверочные) символы, вводимые в исходную последовательность кодером канала и служащие для обнаружения и исправления ошибок.

При кодировании неразделимыми кодами разделить символы входной последовательности на информационные и проверочные невозможно. Непрерывными (древовидными)

называют такие коды, в которых введение избыточных символов в кодируемую последовательность информационных символов осуществляется непрерывно, без разделения ее на независимые блоки. Непрерывные коды также могут быть разделимыми и неразделимыми.

Способность кода обнаруживать и исправлять ошибки обусловлена наличием в нем избыточных *символов.*

На вход кодирующего устройства поступает последовательность из k информационных двоичных символов. На выходе ей соответствует последовательность из n двоичных символов, причем n>k.

Всего может быть 2^k различных входных и 2^n различных выходных последовательностей. Из общего числа 2^n выходных последовательностей только 2^k последовательностей соответствуют входным. Их называют разрешенными кодовыми комбинациями.

Остальные 2^n-2^k возможных выходных последовательностей для передачи не используются. Их называют запрещенными кодовыми комбинациями.

Искажения информации в процессе передачи сводятся к тому, что некоторые из передаваемых символов заменяются другими – неверными.

Так как каждая из 2^k разрешенных комбинаций в результате действия помех может трансформироваться в любую другую, то всегда имеется 2^k*2^n возможных случаев передачи. В это число входят:

1) 2^k случаев безошибочной передачи;

2) $2^k(2^k-1)$ случаев перехода в другие разрешенные комбинации, что соответствует необнаруженным ошибкам;

3) $2^k(2^n - 2^k)$ случаев перехода в неразрешенные комбинации, которые могут быть обнаружены.

Следовательно, часть обнаруживаемых ошибочных кодовых комбинаций от общего числа возможных случаев передачи составляет

$$\frac{2^k(2^n-2^k)}{2^n \cdot 2^k} = 1 - \frac{2^k}{2^n}.$$

Определить обнаруживающую способность кода, каждая комбинация которого содержит всего один избыточный символ (n=k+1).

Решение:

1. Общее число выходных последовательностей составляет 2^{k+1}, т.е. вдвое больше общего числа кодируемых входных последовательностей.

2. За подмножество разрешенных кодовых комбинаций можно принять, например, подмножество 2^k комбинаций, содержащих четное число единиц (или нулей).

3. При кодировании к каждой последовательности из k информационных символов добавляют один символ (0 или 1), такой, чтобы число единиц в кодовой комбинации было четным. Исполнение любого нечетного числа символов переводит разрешенную кодовую комбинацию в подмножество запрещенных комбинаций, что обнаруживается на приемной стороне по нечетности числа единиц. Часть опознанных ошибок составляет

$$1 - \frac{2^k}{2^{k-1}} = \frac{1}{2}$$

Любой метод декодирования можно рассматривать как правило разбиения всего множества запрещенных кодовых комбинаций на 2^k пересекающихся подмножеств M_i, каждая из которых ставится в соответствие одной из разрешенных комбинаций. При получении запрещенной комбинации, принадлежащей подмножеству M_i, принимают решение, что передавалась запрещенная комбинация A_i. Ошибка будет исправлена в тех случаях, когда полученная комбинация действительно образовалась из A_i, т.е. 2^n-2^k случаях.

Всего случаев перехода в неразрешенные комбинации $2^k(2^n - 2^k)$. Таким образом, при наличии избыточности любой код способен исправлять ошибки.

Отношение числа исправляемых кодом ошибочных кодовых комбинаций к числу обнаруживаемых ошибочных комбинаций равно

$$\frac{2^n - 2^k}{2^k(2^n - 2^k)} = \frac{1}{2^k}$$

Способ разбиения на подмножества зависит от того, какие ошибки должны направляться конкретным кодом.

Большинство разработанных кодов предназначено для корректирования взаимно независимых ошибок определенной кратности и пачек (пакетов) ошибок.

Взаимно независимыми ошибками называют такие искажения в передаваемой последовательности символов, при которых вероятность появления любой комбинации искаженных символов зависит только от числа искаженных символов r и вероятности искажения обычного символа p.

При взаимно независимых ошибках вероятность искажения любых r символов в nразрядной кодовой комбинации:

$$p_r = c_n^r p^r (1-\mathrm{p})^{n-r}$$

где p – вероятность искажения одного символа; r – число искаженных символов; n – число двоичных символов на входе кодирующего устройства; c_n^r – число ошибок порядка r.

Если учесть, что p<<1, то в этом случае наиболее вероятны ошибки низшей кратности. Их следует обнаруживать и исправлять в первую очередь.

Контроль четности/нечетности

Простейшим способом обнаружения ошибок является контроль по четности. Обычно контролируется передача блока данных (M бит). Этому блоку ставится в соответствие кодовое слово длиной N бит, причем N>M. Избыточность кода характеризуется величиной 1-M/N. Вероятность обнаружения ошибки определяется отношением M/N (чем меньше это отношение, тем выше вероятность обнаружения ошибки, но и выше избыточность).

В простейшем случае кодовое слово на 1 бит больше блока данных. При таком контроле кодовое слово, передаваемое от источника к приемнику, должно содержать соответственно четное/нечетное число единиц.

Источник формирует передаваемый блок информации так, чтобы условие выполнялось. Приемник проверяет выполнение условия. Если четность/нечетность нарушена, то считается, что произошла ошибка при передаче информации. Приемник сообщает об этом источнику, и источник заново посылает этот блок информации. Передаваемый блок информации состоит из информационных битов и одного контрольного:

Передаваемый блок информации

Информационные биты	Контрольный бит
$X_1, X_2, \ldots, X_n$	K

При контроле на четность в бит K записывается сумма по модулю 2 информационных разрядов. При контроле на нечетность записывается инверсное значение этой суммы.

Рассмотрим пример такого контроля. Пусть от источника к приемнику надо переслать 8 бит информации с контролем на четность.

Тогда пересылаемый блок информации должен содержать 9 бит.

Пример. Переслать код 01110101.

Побитное сложение по модулю для этого кода дает значение, равное 1. Тогда значение контрольного бита должно равняться 1, чтобы передаваемый код содержал четное число единиц.

ПЕРЕДАТЧИК									ПРИЕМНИК										комментарий
информационные биты								контрольный бит	принятые биты									сумма по mod 2	
0	1	1	1	0	1	0	1	1	0	1	1	1	0	1	0	1	1	0	нет ошибки
									0	1	1	1	**1**	1	0	1	1	1	есть ошибка

Код Джонсона

Последовательность чисел в этом коде моделируется односторонним последовательным заполнением его разрядов вначале единицами, а затем нулями. Код Джонсона легко формируется с помощью регистров сдвига и легко дешифруется (таблица 23).

Код «1 из m»

Для случая *m*=8 представлен в таблице 23. Этот код характерен тем, что в любой кодовой комбинации присутствует только одна единица, что позволяет легко находить ошибки в случае искажения кода, и не требуется его дешифрация. Данный код, как и код Джонсона, является избыточным, требующим для своего изображения больше разрядов, чем соответствующие неизбыточные коды.

Таблица 23

Помехоустойчивые коды

S	Код Джонсона	Код «1 из 8»
0	0 0 0 0	0 0 0 0 0 0 0 1
1	0 0 0 1	0 0 0 0 0 0 1 0
2	0 0 1 1	0 0 0 0 0 1 0 0
3	0 1 1 1	0 0 0 0 1 0 0 0
4	1 1 1 1	0 0 0 1 0 0 0 0
5	1 1 1 0	0 0 1 0 0 0 0 0
6	1 1 0 0	0 1 0 0 0 0 0 0
7	1 0 0 0	1 0 0 0 0 0 0 0

Код Грея

При использовании обыкновенного двоичного кода изменение числа на 1 может привести к изменению двоичного числа сразу в нескольких битах, что может вызвать значительные ошибки считывания.

Погрешности считывания можно устранить, если воспользоваться невесовыми кодами, представляющими собой последовательности *n*-разрядных двоичных чисел, в которых каждые два соседних числа отличаются одно от другого только в одном разряде. У таких кодов много названий – коды Грея, рефлексные, отраженные, циклические, прогрессивные коды.

Частный случай, получивший наибольшее распространение, – «код Грея». Главная особенность кода Грея, обеспечившая ему широкое практическое применение, состоит в простоте его построения.

Пусть a – двоичное n-разрядное число обычной (весовой) системы счисления, b – соответствующее число в коде Грея. Тогда правило, по которому можно найти код Грея по заданному числу a, представится формулой вида:

$$b_i = a_i \oplus a_{i+1},$$

где $\oplus$ – знак сложения по модулю 2; i – порядковый номер разряда в числе a; i = 1, 2, 3, …, n; счет начинается с младшего разряда. Чтобы по этому правилу найти код Грея, достаточно поразрядно сложить по модулю 2 число a с самим собой, но сдвинутым вправо на один разряд с потерей цифры младшего разряда и записью нуля в старшем разряде:

$$a = a_n a_{n-1} a_{n-2} \ \dots \ a_2 a_1$$
$$0 \ \ a_n \ \ a_{n-1} a_{n-2} \dots a_2,$$
$$b = b_n b_{n-1} b_{n-2} \ \dots \ b_2 b_1$$

где $b_1 = a_1 \oplus a_2; b_2 = a_2 \oplus a_3; \dots b_{n-1} = a_{n-1} \oplus a_n; b_n = a_n$.

Например, при n = 4 последовательность кодов Грея имеет вид: 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000.

Код Грея является невесовым в отличие от обычной двоичной системы счисления. Это значит, что образующие его двоичные числа надо рассматривать только как упорядоченные наборы нулей и единиц без присвоения им весов. Например, двоичному весовому числу 10011 (в десятичной системе – это 19) соответствует код Грея 11010, и если его считать весовым, то получится число 26 (в десятичной системе). В связи с этим каждому невесовому коду обычно присваивается та или иная величина либо с применением правила, как в случае кода Грея, либо при помощи таблицы.

Коды Грея используются для кодирования номера дорожек в жестких дисках.

Код Хэмминга

Исправлять ошибку при приеме информации труднее, чем ее обнаруживать. Исправление ошибки предполагает два совмещенных процесса: обнаружение факта, что есть ошибка, и определение ее места. После решения этих двух задач, исправление тривиально – надо инвертировать значение ошибочного бита. В наземных каналах связи, где вероятность ошибки невелика, обычно используется только метод обнаружения ошибки и повторной пересылки блока информации, содержавшего ошибку. Для спутниковых каналов с типичными для них большими задержками коррекция ошибки становится необходимой. Здесь используется код Хэмминга.

Код Хэмминга представляет собой блочный код, который позволяет выявить и исправить ошибочно переданный бит в пределах переданного блока. Блочными называются коды, в которых информационный поток символов разбивается на отрезки, и каждый из них преобразуется в определенную последовательность (блок) кодовых символов. В блочных кодах кодирование при передаче (формирование проверочных элементов) и декодирование при приеме (обнаружение и исправление ошибок) выполняются в пределах каждой кодовой комбинации (блока) в отдельности по соответствующим алгоритмам.

Обычно код Хэмминга характеризуется двумя целыми числами, например, код (11,7), используемый при передаче семибитных ASCII-кодов. Такая запись говорит, что при передаче семибитного кода используется дополнительно четыре контрольных бита (7+4=11). При этом предполагается, что может иметь место ошибка в одном бите и что ошибка в двух или более битах существенно менее вероятна. С учетом этого исправление ошибки осуществляется с определенной вероятностью.

При рассмотрении кода Хэмминга требуется знать, что такое кодовое расстояние. Кодовое расстояние между двумя двоичными кодами одинаковой длины определяется количеством битов, в которых эти коды отличаются.

Пример 1. Кодовое расстояние между «кодом 1» и «кодом 2» равно 1.

КОД 1 0 1 0

КОД 2 0 1 1

Пример 2. Кодовое расстояние между «кодом 1» и «кодом 2» равно 2.

КОД 1 0 0 1

КОД 2 1 1 1

Правило вычисления кодового расстояния — комбинации кодов суммируются по модулю 2, после чего подсчитывается количество единиц в полученной сумме.

Например,

КОД 1	1	0	0	1	0
КОД 2	1	0	1	1	1
КОД1⊕КОД2	0	0	1	0	1

Кодовое расстояние равно 2, т.к. число единиц в сумме – 2.

Можно обнаружить ошибку только, если между используемыми кодовыми комбинациями есть необходимое для этого кодовое расстояние, т.е. между соседними используемыми кодовыми комбинациями есть другие комбинации. Эти кодовые комбинации не используются для передачи информации, и их получение на приеме свидетельствует об ошибке передачи информации в канале связи. Для заданного кода минимальное кодовое расстояние – это минимальное из всех расстояний всех пар кодовых слов.

В обычном равномерном непомехоустойчивом коде число разрядов n в кодовых комбинациях определяется числом сообщений и основанием кода. Коды, у которых все кодовые комбинации разрешены к передаче, называются простыми или равнодоступными и являются полностью безызбыточными. Безызбыточные первичные коды обладают большой «чувствительностью» к помехам. Внесение избыточности при использовании помехоустойчивых кодов обязательно связано с увеличением числа разрядов (длины) кодовой комбинации.

В этом случае все множество $N=2^n$ комбинаций можно разбить на два подмножества: подмножество разрешенных комбинаций, т.е. обладающих определенными признаками, и подмножество запрещенных комбинаций, этими признаками не обладающих. Помехоустойчивый код отличается от обычного тем, что в канал передаются не все кодовые комбинации N, которые можно сформировать из имеющегося числа разрядов n, а только их часть N_K, которая составляет подмножество разрешенных комбинаций.

Если при приеме выясняется, что кодовая комбинация принадлежит к запрещенным, то это свидетельствует о наличии ошибки, т.е. таким образом решается задача обнаружения ошибок. При этом принятая комбинация не декодируется (не принимается решение о приеме сообщения). Помехоустойчивые коды называют корректирующими кодами. Корректирующие свойства избыточных кодов зависят от правила их построения, определяющего структуру кода, и параметров кода.

Например, при n=3 можно составить 8 кодов (000, 001, 010, 011, 100, 101, 110, 111).

Пусть все восемь кодовых комбинаций являются разрешенными (допустимыми), тогда кодовое расстояние между соседними комбинациями равно 1. Т.к. в данном случае отсутствует какой-либо признак, позволяющий судить о появлении ошибки в кодовой комбинации, то такой код не является помехозащищенным. Допустим, что лишь четыре из восьми кодовых комбинаций считаются разрешенными, например это комбинации 001, 010, 100 и 111. Тогда кодовое расстояние равно 2, причем искажение символа в одном из разрядов приводит к получению запрещенной кодовой комбинации (000, 011, 101 или 110), что легко выявляется при проверке. Полученный таким образом двоичный код называют кодом с обнаружением одиночной ошибки. Для кодового расстояния равного 3 в качестве разрешенных кодовых комбинаций можно принять, например, 010 и 101. При этом обеспечивается возможность не только обнаружения, но и исправления одиночной ошибки. Действительно, получение запрещенной кодовой комбинации 110 указывает на наличие ошибки, для исправления которой необходимо перейти к ближайшей из разрешенных кодовых комбинаций (в

данном случае – 010). Данный код позволяет обнаруживать и двойные ошибки, так как при одновременном искажении символов в двух разрядах кодовой комбинации последняя также попадает в число запрещенных.

Первые работы по корректирующим кодам принадлежат Хэммингу, который ввел понятие минимального кодового расстояния и предложил код, позволяющий однозначно указать ту позицию в кодовой комбинации, где произошла ошибка. К «*M*» (*message*) информационным битам в коде Хэмминга добавляется «*C*» (*control*) проверочных битов для определения местоположения ошибочного бита.

Таким образом, код Хэмминга состоит из информационных и контрольных битов. Информационные биты будем обозначать через *M*, контрольные биты – через *C*:

$$M=\{M_1,M_2,\ldots,M_n\};$$

$$C=\{C_1, C_2,\ldots C_k\};$$

$N=M+C$ – общее количество передаваемых битов в канал связи (передаваемый блок).

Сущность кода Хэмминга заключается в том, что местоположение контрольных и информационных битов определяется по правилу, которое устанавливает зависимость, позволяющую в случае ошибки определить ее местоположение, т.е. номер бита.

Основные положения для получения кода Хэмминга состоят в следующем:

1. Число, составленное из контрольных разрядов, должно указывать номер бита, в котором произошла ошибка, или 0, если нет ошибки. Исходя из этого, если произошла ошибка, проверочное *C*-битовое число – должно быть в диапазоне от 1 до (2^C-1). Из этого следует, что $2^C-1\geq N$; тогда $2^C\geq N+1$; умножим левую и правую части на 2^M, получим $2^M\times 2^C\geq 2^M\times(N+1)$, откуда $2^{(M+C)}\geq 2^M\times(N+1)$, тогда $2^N\geq 2^M\times(N+1)$; делим левую и правую части на $(N+1)$, получаем условие

$$\frac{2^N}{N+1}\geq 2^M.$$

С использованием этого условия и определяется минимально необходимое число контрольных битов для передачи заданного количества информационных.

2. Контрольные биты в коде Хэмминга занимают позиции с номерами, равными 2^n, где $n=0,1,2,\ldots$ и т.д. Причем позиции кода номеруются справа налево, начиная с 1. Информационные биты располагаются в остающихся позициях справа налево по возрастанию весов разрядов.

3. Значение каждого контрольного бита получается сложением по модулю 2 тех информационных битов, для которых в номере кодовой позиции информационных битов, записанном через контрольные разряды, значение бита равно 1. Эта формулировка будет более понятна, если посмотреть ее использование по табл. 1 или 2.

Вначале построим код Хэмминга для передачи четырехбитового кода.

Условие 1. Это условие будет выполняться при $N=7$:

$$\left(\frac{27}{7+1}=\frac{128}{8}=16\right)\geq(2^4=16).$$

Таким образом, для четырех информационных битов ($M=4$) требуются три контрольных бита ($C=3$).

Условие 2. Определяем местоположение информационных и контрольных битов в коде.

Положение информационных и контрольных битов	M4	M_3	M_2	C_3	M_1	C_2	C_1
Номер позиции кода	7	6	5	4	3	2	1

Условие 3. Определяем логические выражения для вычисления контрольных битов (таблица 24).

Таблица 24

Вычисление контрольных битов для кода (7,4)

ПОЗИЦИЯ КОДА	КОНТРОЛЬНЫЕ БИТЫ		
	C_3	C_2	C_1
3 бит (M_1)	0	1	1
5 бит (M_2)	1	0	1
6 бит (M_3)	1	1	0
7 бит (M_4)	1	1	1
СУММИРУЕМЫЕ ПО МОДУЛЮ 2 БИТЫ	M_2 M_3 M_4	M_1 M_3 M_4	M_1 M_2 M_4

Таким образом, передатчик вычисляет значения контрольных разрядов по следующим логическим выражениям:

$$C_1=M_1\oplus M_2\oplus M_4;$$
$$C_2=M_1\oplus M_3\oplus M_4;$$
$$C_3=M_2\oplus M_3\oplus M_4.$$

Приемник проверяет правильность принятого кода, вычисляя следующие выражения:

$$C_{11}=C_1\oplus M_1\oplus M_2\oplus M_4;$$
$$C_{12}=C_2\oplus M_1\oplus M_3\oplus M_4;$$
$$C_{13}=C_3\oplus M_2\oplus M_3\oplus M_4.$$

Построим код Хэмминга для передачи семибитового кода.

Условие 1. Это условие будет выполняться при N=11:

$$\left(\frac{2^{11}}{11+1}=\frac{2048}{12}\approx 170\right)\geq (2^7=128).$$

Таким образом, для семи информационных битов (M=7), требуются четыре контрольных бита (C=4).

Условие 2. Определяем местоположение информационных и контрольных битов в коде.

Положение информационных и контрольных битов	M_7	M_6	M_5	C_4	M_4	M_3	M_2	C_3	M_1	C_2	C_1
Номер позиции кода	11	10	9	8	7	6	5	4	3	2	1

Условие 3. Определяем логические выражения для вычисления контрольных битов (таблица 25).

Таблица 25

Вычисление контрольных битов для кода (11,4)

ПОЗИЦИЯ КОДА	КОНТРОЛЬНЫЕ БИТЫ			
	C_4	C_3	C_2	C_1
3 бит (M_1)	0	0	1	1
5 бит (M_2)	0	1	0	1
6 бит (M_3)	0	1	1	0
7 бит (M_4)	0	1	1	1
9 бит (M_5)	1	0	0	1
10 бит (M_6)	1	0	1	0
11 бит (M_7)	1	0	1	1
СУММИРУЕМЫЕ ПО МОДУЛЮ 2 БИТЫ	M_5 M_6 M_7	M_2 M_3 M_4	M_1 M_3 M_4 M_6 M_7	M_1 M_2 M_4 M_5 M_7

Таким образом, передатчик вычисляет значения контрольных разрядов по следующим логическим выражениям:

$$C_1=M_1\oplus M_2\oplus M_4\oplus M_5\oplus M_7;$$
$$C_2=M_1\oplus M_3\oplus M_4\oplus M_6\oplus M_7;$$
$$C_3=M_2\oplus M_3\oplus M_4;$$
$$C_4=M_5\oplus M_6\oplus M_7.$$

Приемник проверяет правильность передачи кода, вычисляя следующие выражения:

$$C_{11}=C_1\oplus M_1\oplus M_2\oplus M_4\oplus M_5\oplus M_7;$$
$$C_{12}=C_2\oplus M_1\oplus M_3\oplus M_4\oplus M_6\oplus M_7;$$
$$C_{13}=C_3\oplus M_2\oplus M_3\oplus M_4;$$
$$C_{14}=C_4\oplus M_5\oplus M_6\oplus M_7.$$

Рассмотрим примеры передачи информации с использованием кода Хэмминга для случаев, когда ошибки при передаче нет, и когда есть.

Начнем с кода (7,4).

Пример. В канал связи нужно передать следующий блок информации: 1101_2.

Передатчик формирует код Хэмминга:

M_4	M_3	M_2	C_3	M_1	C_2	C_1
1	1	0	*	1	*	*

$$C_1=M_1\oplus M_2\oplus M_4=1\oplus 0\oplus 1=0;$$
$$C_2=M_1\oplus M_3\oplus M_4=1\oplus 1\oplus 1=1;$$
$$C_3=M_2\oplus M_3\oplus M_4=0\oplus 1\oplus 1=0,$$

тогда код, передаваемый в канал, будет:

M_4	M_3	M_2	C_3	M_1	C_2	C_1
1	1	0	0	1	1	0

Рассмотрим случай, когда не было ошибки при передаче кода.

Тогда точно такой же код принял приемник. Теперь приемник проверяет правильность передачи кода, вычисляя следующие выражения:

$$C_{11}=C_1\oplus M_1\oplus M_2\oplus M_4=0\oplus 1\oplus 0\oplus 1=0;$$
$$C_{12}=C_2\oplus M_1\oplus M_3\oplus M_4=1\oplus 1\oplus 1\oplus 1=0;$$
$$C_{13}=C_3\oplus M_2\oplus M_3\oplus M_4=0\oplus 0\oplus 1\oplus 1=0.$$

Поскольку значение, составленное из контрольных битов, равно нулю, ошибки нет, и код передается от приемника далее.

Рассмотрим случай, когда была ошибка при передаче кода.

Пусть ошибка произошла в третьем бите, т.е. принят код:

M_4	M_3	M_2	C_3	M_1	C_2	C_1
4	3	2	3	1	2	1
1	1	0	0	**0**	1	0

Теперь приемник проверяет правильность передачи кода, вычисляя следующие выражения:

$$C_{11}=C_1\oplus M_1\oplus M_2\oplus M_4=0\oplus 0\oplus 0\oplus 1=1;$$
$$C_{12}=C_2\oplus M_1\oplus M_3\oplus M_4=1\oplus 0\oplus 1\oplus 1=1;$$
$$C_{13}=C_3\oplus M_2\oplus M_3\oplus M_4=0\oplus 0\oplus 1\oplus 1=0.$$

Поскольку, значение, составленное из контрольных разрядов, равно 011_2, ошибка в бите № 3. Приемник меняет значение этого бита на противоположное.

Теперь рассмотрим код (11,7).

Пример. В канал связи нужно передать следующий блок информации: 1101010_2.

Передатчик формирует код Хэмминга:

M_7	M_6	M_5	C_4	M_4	M_3	M_2	C_3	M_1	C_2	C_1
1	1	0	*	1	0	1	*	0	*	*

$$C_1=M_1\oplus M_2\oplus M_4\oplus M_5\oplus M_7=0\oplus 1\oplus 1\oplus 0\oplus 1=1;$$
$$C_2=M_1\oplus M_3\oplus M_4\oplus M_6\oplus M_7=0\oplus 0\oplus 1\oplus 1\oplus 1=1;$$
$$C_3=M_2\oplus M_3\oplus M_4=1\oplus 0\oplus 1=0;$$
$$C_4=M_5\oplus M_6\oplus M_7=0\oplus 1\oplus 1=0,$$

тогда код, передаваемый в канал, будет:

M_7	M_6	M_5	C_4	M_4	M_3	M_2	C_3	M_1	C_2	C_1
1	1	0	0	1	0	1	0	0	1	1

Рассмотрим случай, когда не было ошибки при передаче кода.

Тогда точно такой же код принял приемник. Теперь приемник проверяет правильность передачи кода, вычисляя следующие выражения:

$$C_{11}=C_1\oplus M_1\oplus M_2\oplus M_4\oplus M_5\oplus M_7=1\oplus 0\oplus 1\oplus 1\oplus 0\oplus 1=0;$$
$$C_{12}=C_2\oplus M_1\oplus M_3\oplus M_4\oplus M_6\oplus M_7=1\oplus 0\oplus 0\oplus 1\oplus 1\oplus 1=0;$$
$$C_{13}=C_3\oplus M_2\oplus M_3\oplus M_4=0\oplus 1\oplus 0\oplus 1=0;$$
$$C_{14}=C_4\oplus M_5\oplus M_6\oplus M_7=0\oplus 0\oplus 1\oplus 1=0.$$

Поскольку значение, составленное из контрольных битов, равно нулю, ошибки нет, и код передается от приемника далее.

Рассмотрим случай, когда была ошибка при передаче кода.

Пусть ошибка произошла во втором бите, т.е. принят код:

M_7	M_6	M_5	C_4	M_4	M_3	M_2	C_3	M_1	C_2	C_1
1	1	0	0	1	0	1	0	0	**0**	1

Теперь приемник проверяет правильность передачи кода, вычисляя следующие выражения:

$$C_{11}=C_1\oplus M_1\oplus M_2\oplus M_4\oplus M_5\oplus M_7=1\oplus 0\oplus 1\oplus 1\oplus 0\oplus 1=0;$$
$$C_{12}=C_2\oplus M_1\oplus M_3\oplus M_4\oplus M_6\oplus M_7=0\oplus 0\oplus 0\oplus 1\oplus 1\oplus 1=1;$$
$$C_{13}=C_3\oplus M_2\oplus M_3\oplus M_4=0\oplus 1\oplus 0\oplus 1=0;$$
$$C_{14}=C_4\oplus M_5\oplus M_6\oplus M_7=0\oplus 0\oplus 1\oplus 1=0.$$

Поскольку значение, составленное из контрольных битов, равно 0010_2, ошибка во втором бите. Приемник меняет значение этого бита на противоположное.

Код Хэмминга является равномерно защищенным, исправляет ошибки как в информационных, так и в контрольных битах. С его помощью возможно исправление одиночной и обнаружение двойной ошибки. Для того чтобы обнаружить двойную ошибку, в код Хэмминга вводят дополнительный контрольный бит E_0. Значение E_0 вычисляется как сумма по модулю 2 всех разрядов кода Хэмминга, т.е. он дополняет код Хэмминга по четности. На стороне приемника, кроме получения контрольных разрядов, которые определяют позицию с ошибкой, вычисляется контрольный разряд E_{10}:

$E_{10}=E_0\oplus$<сложенные по модулю 2 все остальные биты принятого кода>.

Если $E_{10}=0$, то ошибок нет. Если значение, полученное из контрольных разрядов, не равно 0 и $E_{10}=1$, то имеет место одиночная ошибка, позиция которой определяется значением контрольных разрядов. Если значение $C\neq 0$ и $E_{10}=0$, то имеет место двойная ошибка.

Рассмотрим пример для кода, исправляющего одиночную ошибку и обнаруживающего двойную.

Для кода (11,7) формат блока информации:

M_7	M_6	M_5	C_4	M_4	M_3	M_2	C_3	M_1	C_2	C_1	E_0

Пример. В канал связи нужно передать следующий блок информации: 1101010_2.

Передатчик формирует код Хэмминга:

M_7	M_6	M_5	C_4	M_4	M_3	M_2	C_3	M_1	C_2	C_1	E_0
1	1	0	*	1	0	1	*	0	*	*	*

$$C_1=M_1\oplus M_2\oplus M_4\oplus M_5\oplus M_7=0\oplus 1\oplus 1\oplus 0\oplus 1=1;$$
$$C_2=M_1\oplus M_3\oplus M_4\oplus M_6\oplus M_7=0\oplus 0\oplus 1\oplus 1\oplus 1=1;$$
$$C_3=M_2\oplus M_3\oplus M_4=1\oplus 0\oplus 1=0;$$
$$C_4=M_5\oplus M_6\oplus M_7=0\oplus 1\oplus 1=0,$$
$$E_0=C_1\oplus C_2\oplus M_1\oplus C_3\oplus M_2\oplus M_3\oplus M_4\oplus C_4\oplus M_5\oplus M_6\oplus M_7=1\oplus 1\oplus 0\oplus 0\oplus 1\oplus 0\oplus\oplus 1\oplus 0\oplus 0\oplus 1\oplus 1=0,$$

тогда код, передаваемый в канал, будет:

M_7	M_6	M_5	C_4	M_4	M_3	M_2	C_3	M_1	C_2	C_1	E_0
1	1	0	0	1	0	1	0	0	1	1	0

Рассмотрим случай, когда не было ошибки при передаче кода.

Тогда точно такой же код принял приемник. Теперь приемник проверяет правильность передачи кода, вычисляя следующие выражения:

$$C_{11}=C_1\oplus M_1\oplus M_2\oplus M_4\oplus M_5\oplus M_7=1\oplus 0\oplus 1\oplus 1\oplus 0\oplus 1=0;$$
$$C_{12}=C_2\oplus M_1\oplus M_3\oplus M_4\oplus M_6\oplus M_7=1\oplus 0\oplus 0\oplus 1\oplus 1\oplus 1=0;$$
$$C_{13}=C_3\oplus M_2\oplus M_3\oplus M_4=0\oplus 1\oplus 0\oplus 1=0;$$
$$C_{14}=C_4\oplus M_5\oplus M_6\oplus M_7=0\oplus 0\oplus 1\oplus 1=0,$$
$$E_{10}= E_0\oplus C_1\oplus C_2\oplus M_1\oplus C_3\oplus M_2\oplus M_3\oplus M_4\oplus C_4\oplus M_5\oplus M_6\oplus M_7=$$
$$=0\oplus 1\oplus 1\oplus 0\oplus 0\oplus 1\oplus 0\oplus 1\oplus 0\oplus 0\oplus 1\oplus 1=0.$$

Поскольку значение, составленное из контрольных битов, равно 0, и $E_{10}=0$, ошибки нет, и код передается от приемника далее.

Рассмотрим случай, когда была одиночная ошибка при передаче кода.

Пусть ошибка произошла во втором бите (не считая E_0, т.е. в C_2) и принят код:

M_7	M_6	M_5	C_4	M_4	M_3	M_2	C_3	M_1	C_2	C_1	E_0
1	1	0	0	1	0	1	0	0	**0**	1	0

Теперь приемник проверяет правильность передачи кода, вычисляя следующие выражения:

$$C_{11}=C_1\oplus M_1\oplus M_2\oplus M_4\oplus M_5\oplus M_7=1\oplus 0\oplus 1\oplus 1\oplus 0\oplus 1=0;$$
$$C_{12}=C_2\oplus M_1\oplus M_3\oplus M_4\oplus M_6\oplus M_7=0\oplus 0\oplus 0\oplus 1\oplus 1\oplus 1=1;$$
$$C_{13}=C_3\oplus M_2\oplus M_3\oplus M_4=0\oplus 1\oplus 0\oplus 1=0;$$
$$C_{14}=C_4\oplus M_5\oplus M_6\oplus M_7=0\oplus 0\oplus 1\oplus 1=0,$$
$$E_{10}= E_0\oplus C_1\oplus C_2\oplus M_1\oplus C_3\oplus M_2\oplus M_3\oplus M_4\oplus C_4\oplus M_5\oplus M_6\oplus M_7=$$
$$=0\oplus 1\oplus 0\oplus 0\oplus 0\oplus 1\oplus 0\oplus 1\oplus 0\oplus 0\oplus 1\oplus 1=1.$$

Поскольку значение $E_{10}=1$ и значение, составленное из контрольных разрядов, равно 0010_2, то ошибка во втором бите. Приемник меняет значение этого бита на противоположное.

Рассмотрим случай, когда была двойная ошибка при передаче кода.

Пусть ошибка произошла во втором (т.е. в C_2) и в пятом битах (т.е. M_2) и принят код:

M_7	M_6	M_5	C_4	M_4	M_3	M_2	C_3	M_1	C_2	C_1	E_0
1	1	0	0	1	0	**0**	0	0	**0**	1	0

Теперь приемник проверяет правильность передачи кода, вычисляя следующие выражения:

$$C_{11}=C_1\oplus M_1\oplus M_2\oplus M_4\oplus M_5\oplus M_7=1\oplus 0\oplus 0\oplus 1\oplus 0\oplus 1=1;$$
$$C_{12}=C_2\oplus M_1\oplus M_3\oplus M_4\oplus M_6\oplus M_7=0\oplus 0\oplus 0\oplus 1\oplus 1\oplus 1=1;$$
$$C_{13}=C_3\oplus M_2\oplus M_3\oplus M_4=0\oplus 0\oplus 0\oplus 1=1;$$
$$C_{14}=C_4\oplus M_5\oplus M_6\oplus M_7=0\oplus 0\oplus 1\oplus 1=0,$$
$$E_{10}= E_0\oplus C_1\oplus C_2\oplus M_1\oplus C_3\oplus M_2\oplus M_3\oplus M_4\oplus C_4\oplus M_5\oplus M_6\oplus M_7=$$
$$=0\oplus 1\oplus 0\oplus 0\oplus 0\oplus 0\oplus 0\oplus 1\oplus 0\oplus 0\oplus 1\oplus 1=0.$$

Поскольку значение $E_{10}=0$ и значение, составленное из контрольных разрядов, равно $0111_2\neq 0$, то произошла двойная ошибка. Приемник не передает код далее, а запрашивает от источника повторную передачу этого блока информации.

Код Рида-Соломона

В кодах Рида-Соломона сообщение представляется в виде набора символов некоторого алфавита. Собственно говоря, в качестве алфавита используется поле Галуа. То есть если необходимо закодировать сообщение, представленное двоичным кодом, то он разбивается (в случае, если используется поле Галуа из 16 элементов) на группы по 4 бита и дальше работаем с каждой группой как с числом из этого поля Галуа.

При построении кода Рида-Соломона задаётся пара чисел N, K, где N – общее количество символов, а K – «полезное» количество символов, остальные N-K символов представляют собой избыточный код, предназначенный для восстановления ошибок.

Такой код Галуа будет иметь так называемое «расстояние Хэмминга» D = N – K +1. Расстояние Хэмминга является параметром кода и определяется как минимальное число различий между двумя различными кодовыми словами. В соответствии с теорией кодирования, код, имеющий расстояние Хемминга D = 2t+1, позволяет восстанавливать t ошибок. Таким образом, если в наше кодовое слово случайно внести t = (N-K)/2 ошибок (т.е. просто произвольно заменить значения t символов любыми значениями), то окажется возможным обна-

ружить и исправить эти ошибки. Сообщения при кодировании Рида-Соломона представляются полиномами. Исходное сообщение представляется как коэффициенты полинома p(x) степени K-1, имеющего K коэффициентов. Важную роль играет порождающий многочлен Рида-Соломона, g(x), который строится следующим образом:

$$g(\mathrm{x}) = \prod_{i=1}^{D-1}(x + a^i),$$

где a – это примитивный член. Нетрудно заметить (учитывая, что операция сложения равносильна операции вычитания), что a^1 , a^2 .. a^{D-1} - являются корнями этого многочлена.

Например, построим порождающий многочлен кода Рида-Соломона с N = 15, K = 9:

$$g(\mathrm{x}) = (\mathrm{x}+2)(\mathrm{x}+2^2)(\mathrm{x}+2^3)(\mathrm{x}+2^4)(\mathrm{x}+2^5)(\mathrm{x}+2^6) = x^6 + 7x^5 + 9x^4 + 3x^3 + 12x^2 + 10^x + 12 .$$

(Возведения в степень и умножения выполнены по правилам поля Галуа)

Этот код будет способен исправлять до трёх ошибок: (15 – 9) / 2 = 3.

Кодирование Рида-Соломона выполняется достаточно просто. Вообще говоря, существует две разновидности кодирования: систематический и несистематический код. В несистематическом коде закодированное сообщение не содержит в явном виде исходного сообщения: закодированное сообщение получается как произведение исходного на порождающий многочлен g(x):

$$C(\mathrm{x}) = \mathrm{p}(\mathrm{x}) \cdot \mathrm{g}(\mathrm{x})$$

Систематический код строится по-другому:

Сначала полином сдвигается на K коэффициентов влево

$$p'(\mathrm{x}) = \mathrm{p}(\mathrm{x}) \cdot \mathrm{x}^{(\mathrm{N}-\mathrm{k})},$$

далее вычисляется его остаток от деления на порождающий полином и прибавляется к p'(x):

$$C(\mathrm{x}) = \mathrm{p}'(\mathrm{x}) + \mathrm{p}'(\mathrm{x}) \cdot \mathrm{mod}\, g(\mathrm{x}) .$$

Другими словами, сообщение «сдвигается» на N-K символов - так, что его полином имеет такие коэффициенты:

m8, m7, m6, m5, m4, m3, m2, m1, m0, 0,0,0,0,0,0 (m0.. m8 - символы сообщения).

Далее этот полином делится с остатком на порождающий полином g(x), в результате чего в остатке получается полином степени (N-K-1) с N-k коэффициентами. (Поскольку полином g(x) имеет степень N-k, что следует из принципа его построения). Этот полином прибавляется к исходному полиному, сдвинутому на N-K символов (т.е. коэффициенты остатка как раз занимают место нулей).

Для систематического кода очевидно, что K старших коэффициентов полученного кода C(x) содержат исходное сообщение. Это удобно при декодировании, поэтому в дальнейшем будем рассматривать именно систематический вариант.

Закодированное сообщение C(x) обладает очень важным свойством: оно без остатка делится на порождающий многочлен g(x).

Для несистематического кодирования этот факт очевиден: ведь C(x) является произведением g(x) на p(x); для систематического следует рассуждать так:

Пусть r(x) – остаток от деления p'(x) на g(x). Тогда, u(x) - некий полином, в данном случае абсолютно неважно, какой.

Итак, r(x) = p'(x) mod g(x). Тогда C(x) = p'(x) + p'(x) mod g(x) = g(x)*u(x) + r(x) + r(x).

Вспомним, что в арифметике поля Галуа сложения является одновременно и вычитанием – тогда r(x)+r(x) = 0. Следовательно, C(x) = g(x)* u(x), т.е. делится на g(x) без остатка. Таким образом, C(x) mod g(x) = 0.

В случае, если закодированное сообщение будет изменено, то это равенство окажется нарушенным. Факт искажения можно рассматривать как прибавление к C(x) некоторого полинома ошибки E(x).

C'(x) = C(x) + E(x), тогда C'(x) mod g(x) = E(x) mod g(x) = e(x) <> 0.

Рассмотрим кодирование информации. Пусть наше сообщение такое:

7, 5, 10, 0, 9, 1, 1, 1, 9

Полином имеет следующий вид: $p(x) = 9x^8 +1x^7 +1x^6 +1x^5 +9x^4 +0x^3 +10x^2 +5x+7$.

Умножая на x^6 , получаем: $9x^{14}+x^{13}+x^{12}+x^{11}+9x^{10}+10x^8 +5x^7 +7x^6$.

Делим на g(x): $9x^{14}+x^{13}+x^{12}+x^{11}+9x^{10}+0x^9+10x^8+5x^7+7x^6/x^6+7x^5 +9x^4+3x^3 +12x^2+10x+12$.

(деление выполняется как и деление обычных полиномов, однако с использованием правил операций в полях Галуа) и получаем в остатке: $13x^5 +6x^4 +14x^3 +15x^2 +15x+3$.

Получается полином закодированного сообщения

$C(x)= 9x^{14}+x^{13}+x^{12}+x^{11}+9x^{10}+10x^8+5x^7+7x^6+13x^5+6x^4+14x^3+15x^2+15x+3$.

Полученное закодированное сообщение 3, 15, 15, 14, 6, 13 7, 5, 10, 0, 9, 1, 1, 1, 9.

Декодирование кодов Рида-Соломона значительно сложнее кодирования. Очевидно, что первым шагом необходимо выполнить деление полинома на порождающий полином g(x). Если остаток равен нулю, то сообщение неискажено и декодирование (для систематического кода) тривиально: следует просто выделить из сообщения коэффициенты с N-k+1 до N-1 – это и будет основное сообщение.

В случае же присутствия ошибки (т.е. e(x) = C(x) mod g(x) <>0), то придётся выполнить следующие действия.

Декодирование основано на построении многочлена синдрома ошибки S(x) и отыскании соответствующего ему многочлена локаторов L(x).

Локаторы ошибок – это элементы поля Галуа, степень которых совпадает с позицией ошибки. Так, если искажён коэффициент при x^4 , то локатор этой ошибки равен a^4, если искажён коэффициент при x^7 то локатор ошибки будет равен a^7 и т.п. (a – примитивный член, т.е. в нашем случае a=2).

Многочлен локаторов L(x) – это многочлен, корни которого обратны локаторам ошибок. Таким образом, многочлен L(x) должен иметь вид

$$L(x) = (1+xX_1)(1+xX_2)\ldots(1+xX_i),$$

где X_1, X_2, X_i – локаторы ошибок. $(1+xX_i$ обращается в ноль при $x=X_i^{-1} : X_iX_i^{-1}= 1, 1+1 =0)$.

Ясно, что если этот многочлен будет найден, то мы легко сможем определить локаторы ошибок – для этого потребуется только определить его корни, что легко сделать обычным перебором.

Для определения этого полинома сначала получают вспомогательный полином S(x), так называемый синдром ошибки. Коэффициенты синдрома ошибки получаются подстановкой степеней примитивного члена в остаток многочлен e(x) = C(x) mod g(x), или в сам многочлен сообщения C(x).

$$S_i = e(a^{i+1}) \text{ или } S_i = C(a^{i+1})$$

Нетрудно убедиться, что если бы сообщение не было искажено, то все коэффициенты S_i оказались бы равны нулю: ведь неискажённое сообщение C(x) кратно порождающему многочлену g(x), для которого числа a^1 , a^2 … a^{N-K} являются корнями.

Между L(x) и S(x) существует соотношение $L(x)*S(x) = W(x) \bmod x^{N-k}$.

W(x) – называется многочленом ошибок. Степень многочлена W(x) не может превышать u-1, где u – количество ошибок. А максимальное количество ошибок, которые может исправить код Рида-Соломона, это (N-K)/2.

С учётом этого обстоятельства, а также учитывая, что свободный член L(x) L_0=1 (ведь $L(x) = (1+xX_1)(1+xX_2)\ldots(1+xX_i)$) можно составить систему линейных уравнений.

$W(x) =$

L_0S_0

$(L_0S_1+L_1S_0)X+$

$(L_0S_2+L_1S_1+L_2S_0)X^2+$

$(L_0S_3+L_1S_2+L_2S_1+L_3S_0)X^3+$

$(L_0S_{N-K-1}+L_1S_{N-K-2}\ldots. L_{(N-K)/2}S_{(N-K)/2-1})X^{N-K-1}$

$L_{(N-K)/2}S_{N-K-1}X^{3/2(N-K)-1}$

$L_0 = 1$. Пусть $t = (N-k)/2$. Коэффициенты при степенях от 0 до t – 1 не равны нулю, при старших степенях должны быть нулевыми.

$$L_0S_t + L_1S_{t-1} + L_2S_{t-2} + L_3S_{t-3}.. L_tS_0 = 0$$
$$L_0S_{t+1} + L_1S_t + L_2S_{t-1} + L_3S_{t-2}.. L_tS_1 = 0$$
$$L_0S_{t+2} + L_1S_{t+1} + L_2S_t + L_3S_{t-1}.. L_tS_2 = 0$$

Коэффициент L0 известен, остальные необходимо найти, следовательно требуется составить t уравнений.

$$L_1S_{t-1} + L_2S_{t-2} + L_3S_{t-3}.. L_tS_0 = S_t$$
$$L_1S_t + L_2S_{t-1} + L_3S_{t-2}.. L_tS_1 = S_{t+1}$$
$$L_1S_{t+1} + L_2S_t + L_3S_{t-1}.. L_tS_2 = S_{t+2}$$
$$L_1S_{2t-2}+L_2S_{2t-2}+L3S_{2t-3}\ldots +L_tS_t = S_{2t-1}$$

В матричном виде

$$M = \begin{Vmatrix} S_{t-1} & S_{t-2} & S_{t-3} & .... & S_0 \\ S_t & S_{t-1} & S_{t-2} & .... & S_1 \\ S_{t+1} & S_t & S_{t-1} & .... & S_2 \\ .... & .... & .... & .... & .... \\ S_{2t-2} & S_{2t-3} & S_{2t-4} & .... & S_t \end{Vmatrix} \quad V = \begin{Vmatrix} S_t \\ S_{t+1} \\ S_{t+2} \\ ..... \\ S_{2t+1} \end{Vmatrix} \quad L = \begin{Vmatrix} L_1 \\ L_2 \\ L_3 \\ .... \\ L_t \end{Vmatrix}$$

$$ML = V \rightarrow L = M^{-1}V$$

Таким образом, вычисление полинома локаторов сводится к построению матрицы M, нахождению обратной ей и умножению на вектор V.

Обратная матрица получается так же, как и в обычной математике, например Жордановым методом.

Возможно, что матрица M окажется линейно-зависимой. Это означает, что ошибок меньше чем t, в этом случае следует повторить построение матрицы для t, уменьшенного на 1.

Найти полином L(x) можно и другими методами, например, можно применить алгоритм Евклида поиска НОД или применить метод Берлекампа-Месси, который является наиболее эффективным.

После того, как полином L(x) найден, следует найти его корни – они будут обратны к локаторам ошибок.

Вычисляется W(x) = L(x)*S(x), коэффициенты старшие чем N-k должны быть обнулены.

Далее следует вычислить производную L(x). Производная вычисляется следующим образом – для чётных степеней производная равна нулю, для нечётных - степени уменьшенной на 1.Далее вычисляются значения ошибок по формуле $Y_i = W(X_i^{-1})/L'(X_i^{-1})$.

Шаги декодирования.

1. Вычислить $e(x) = C(x) \bmod g(x)$.

2. Если $e(x) = 0$ то выделить $p(x)$ из $C(x)$.

3. Иначе, вычислить полином синдрома $Si = e(a^{i+1})$

4. Построить матрицу M и вычислить L(x)

5. Вычислить L'(x). $L'_i = L_{i+1}$ для чётных i и 0 для нечётных.

6. Вычислить $W(x) = S(x)*L(x)$

7. Получить корни L(x) – локаторы ошибок

8. Получить значения ошибок $Y_i = W(X_{i-1})/L'(X_{i-1})$

9. Сформировать многочлен ошибок E(X) на основе локаторов и значений ошибок и скорректировать $C(x) = C(x) + E(x)$.

«Испортим» многочлен, полученный ранее

$C(x)= 9x^{14}+x^{13}+x^{12}+x^{11}+9x^{10}+10x^8 +5x^7 +7x^6 +13x^5 +6x^4 +14x^3 +15x^2 +15x+3.$

Путём добавления к нему полинома ошибки $E(x) = 2x^{13}+3x^{11}+7x^8$.

Получаем

$C'(x)= 9x^{14}+3x^{13}+x^{12}+2x^{11}+9x^{10}+13x^8 +5x^7 +7x^6 +13x^5 +6x^4 +14x^3 +15x^2 +15x+3.$

То есть, полученное сообщение имеет вид: 3, 15, 15, 14, 6, **13** 7, 5, 13, 0, 9, **2**, 1, **3**, 9.

Делим C'(x) на g(x): получаем полином ошибки e(x): $6x^5 +0x^4 +15x^3 +3x^2 +10x+13$.

Полином синдрома ошибки $S(x) = 9x^5 + 3x^4 +2x^3 +15x^2 +15x$.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Дмитриев В.И. Прикладная теория информации: Учебник для вузов. М.: Высшая школа, 1989. 320 с.

2. Венцель Е.С., Овчаров Л.А. Теория случайных процессов и ее инженерные приложения: Учебное пособие для вузов 2-е изд. М.: Высшая школа, 2000. 383 с.

3. Лидовский В.В. Теория информации: Учебное пособие. М.: Компания Спутник+, 2004. 111 с.

4. Ватолин Д., Ратушняк А., Смирнов М. Методы сжатия данных. Устройства архиваторов, сжатие изображений и видео. М.:ДИАЛОГ-МИФИ, 2003. 381 с.

Учебное текстовое электронное издание

Баранкова Инна Ильинична
Коновалов Максим Владимирович

ТЕОРИЯ ИНФОРМАЦИИ. КОДИРОВАНИЕ

Учебное пособие

1,94 Мб
1 электрон. опт. диск

г. Магнитогорск, 2017 год
ФГБОУ ВО «МГТУ им. Г.И. Носова»
Адрес: 455000, Россия, Челябинская область, г. Магнитогорск,
пр. Ленина 38

ФГБОУ ВО «Магнитогорский государственный
технический университет им. Г.И. Носова»
Кафедра информатики и информационной безопасности
Центр электронных образовательных ресурсов и
дистанционных образовательных технологий
e-mail: ceor_dot@mail.ru