Способы сжатия информации

Введение в теорию сжатия

Методы сжатия данных имеют достаточно длинную историю развития, которая началась задолго до появления первого компьютера. В этой статье будет произведена попытка дать краткий обзор основных теорий, концепций идей и их реализаций, не претендующий, однако, на абсолютную полноту. Более подробные сведения можно найти, например, в Кричевский Р.Е. [1989], Рябко Б.Я. [1980], Witten I.H. [1987], Rissanen J. [1981], Huffman D.A.[1952], Gallager R.G. [1978], Knuth D.E. [1985], Vitter J.S. [1986] и др.

Сжатие информации — проблема, имеющая достаточно давнюю историю, гораздо более давнюю, нежели история развития вычислительной техники, которая (история) обычно шла параллельно с историей развития проблемы кодирования и шифровки информации. Все алгоритмы сжатия оперируют входным потоком информации, минимальной единицей которой является бит, а максимальной - несколько бит, байт или несколько байт. Целью процесса сжатия, как правило, есть получение более компактного выходного потока информационных единиц из некоторого изначально некомпактного входного потока при помощи некоторого их преобразования. Основными техническими характеристиками процессов сжатия и результатов их работы являются:

- степень сжатия (compress rating) или отношение (ratio) объемов исходного и результирующего потоков
- скорость сжатия - время, затрачиваемое на сжатие некоторого объема информации входного потока, до получения из него эквивалентного выходного потока

- качество сжатия - величина, показывающая на сколько сильно упакован выходной поток, при помощи применения к нему повторного сжатия по этому же или иному алгоритму

Существует несколько различных подходов к проблеме сжатия информации. Одни имеют весьма сложную теоретическую математическую базу, другие основаны на свойствах информационного потока и алгоритмически достаточно просты. Любой способ подход и алгоритм, реализующий сжатие или компрессию данных, предназначен для снижения объема выходного потока информации в битах при помощи ее обратимого или необратимого преобразования. Поэтому, прежде всего, по критерию, связанному с характером или форматом данных, все способы сжатия можно разделить на две категории: обратимое и необратимое сжатие.

Под необратимым сжатием подразумевают такое преобразование входного потока данных, при котором выходной поток, основанный на определенном формате информации, представляет, с некоторой точки зрения, достаточно похожий по внешним характеристикам на входной поток объект, однако отличается от него объемом. Степень сходства входного и выходного потоков определяется степенью соответствия некоторых свойств объекта (т.е. сжатой и несжатой информации, в соответствии с некоторым определенным форматом данных), представляемого данным потоком информации. Такие подходы и алгоритмы используются для сжатия, например, данных растровых графических файлов с низкой степенью повторяемости байтов в потоке. При таком подходе используется свойство структуры формата графического файла и возможность представить графическую картинку приблизительно схожую по качеству отображения (для восприятия человеческим глазом) несколькими (а точнее n) способами. Поэтому, кроме степени или величины сжатия, в таких алгоритмах возникает понятие качества, т.к. исходное изображение в процессе

сжатия изменяется, то под качеством можно понимать степень соответствия исходного и результирующего изображения, оцениваемая субъективно, исходя из формата информации. Для графических файлов такое соответствие определяется визуально, хотя имеются и соответствующие интеллектуальные алгоритмы и программы. Необратимое сжатие невозможно применять в областях, в которых необходимо иметь точное соответствие информационной структуры входного и выходного потоков. Данный подход реализован в популярных форматах представления видео и фото информации, известных как JPEG и JFIF алгоритмы и JPG и JIF форматы файлов.

Обратимое сжатие всегда приводит к снижению объема выходного потока информации без изменения его информативности, т.е. - без потери информационной структуры. Более того, из выходного потока, при помощи восстанавливающего или декомпрессирующего алгоритма, можно получить входной, а процесс восстановления называется декомпрессией или распаковкой, и только после процесса распаковки данные пригодны для обработки в соответствии с их внутренним форматом.

В обратимых алгоритмах кодирование как процесс можно рассматривать со статистической точки зрения, что еще более полезно, не только для построения алгоритмов сжатия, но и для оценки их эффективности. Для всех обратимых алгоритмов существует понятие стоимости кодирования. Под стоимостью кодирования понимается средняя длина кодового слова в битах. Избыточность кодирования равна разности между стоимостью и энтропией кодирования, а хороший алгоритм сжатия всегда должен минимизировать избыточность (напомним, что под энтропией информации понимают меру ее неупорядоченности.). Фундаментальная теорема Шеннона о кодировании информации говорит о том, что "стоимость кодирования всегда не меньше энтропии источника, хотя может быть сколь угодно близка к ней". Поэтому, для

любого алгоритма, всегда имеется некоторый предел степени сжатия, определяемый энтропией входного потока.

Перейдем теперь непосредственно к алгоритмическим особенностям обратимых алгоритмов и рассмотрим важнейшие теоретические подходы к сжатию данных, связанные с реализацией кодирующих систем и способы сжатия информации.

Групповое кодирование (RLE).

Наиболее известный простой подход и алгоритм сжатия информации обратимым путем - это кодирование серий последовательностей (Run Length Encoding - RLE). Суть методов данного подхода состоит в замене цепочек или серий повторяющихся байтов или их последовательностей на один кодирующий байт и счетчик числа их повторений. Проблема всех аналогичных методов заключается лишь в определении способа, при помощи которого распаковывающий алгоритм мог бы отличить в результирующем потоке байтов кодированную серию от других — некодированных последовательностей байтов. Решение проблемы достигается обычно простановкой меток в начале кодированных цепочек. Такими метками могут быть, например, характерные значения битов в первом байте кодированной серии, значения первого байта кодированной серии и т.п. Данные методы, как правило, достаточно эффективны для сжатия растровых графических изображений (BMP, PCX, TIF, GIF:), т.к. последние содержат достаточно много длинных серий повторяющихся последовательностей байтов. Недостатком метода RLE является достаточно низкая степень сжатия или стоимость кодирования файлов с малым числом серий и, что еще хуже - с малым числом повторяющихся байтов в сериях.

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 двоичного 11000000 и подряд попарно не повторяются.

Данный алгоритм реализован в формате PCX.

Второй вариант алгоритма имеет больший максимальный коэффициент архивации 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)
}
CompressedFile.WriteByte(byte)
} 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, а также в формате TGA.

Арифметическое кодирование

Арифметическое кодирование является методом, позволяющим упаковывать символы входного алфавита без потерь при условии, что известно распределение частот этих символов и является наиболее оптимальным, т.к. достигается теоретическая граница степени сжатия.

Предполагаемая требуемая последовательность символов, при сжатии методом арифметического кодирования, рассматривается как некоторая двоичная дробь из интервала [0, 1). Результат сжатия представляется, как последовательность двоичных цифр из записи этой дроби. Идея метода состоит в следующем: исходный текст рассматривается как запись этой дроби, где каждый входной символ является "цифрой" с весом, пропорциональным вероятности его появления. Этим объясняется интервал, соответствующий минимальной и максимальной вероятностям появления символа в потоке. Поясним работу метода на примере:

Пусть алфавит состоит из двух символов: a и b с вероятностями соответственно 3/4 и 1/4. К ак уже говорилось выше, кодирование Хаффмана не может упаковывать слова в данном алфавите, т.к. не справляется без сегментации с двухсимвольным алфавитом.

Рассмотрим наш интервал вероятностей [0, 1). Разобьем его на части, длина которых пропорциональна вероятностям символов. В нашем случае это [0, 3/4) и [3/4,1). Суть алгоритма в следующем: каждому слову во входном алфавите соответствует некоторый подинтервал из интервала [0, 1) а пустому слову соответствует весь интервал [0, 1). После получения каждого следующего символа интервал уменьшается с выбором той его части, которая соответствует новому символу. Кодом цепочки является интервал, выделенный после обработки всех ее символов, точнее, двоичная запись любой точки из этого интервала, а длина полученного интервала пропорциональна вероятности появления кодируемой цепочки.

Применим данный алгоритм для цепочки "aaba":

Шаг	Просмотренная цепочка	Интервал
0	нет	[0, 1) = [0, 1)
1	a	[0, 3/4) = [0, 0.11)
2	aa	[0, 9/16) = [0, 0.1001)
3	aab	[27/64, 36/64) = [0.011011, 0.100100)
3	aaba	[108/256, 135/256) = [0.01101100, 0.10000111)

В качестве кода можно взять любое число из интервала, полученного на шаге 4, например, 0.1.

Алгоритм декодирования работает синхронно с кодирующим: начав с интервала [0, 1), он последовательно определяет символы входной цепочки. В частности, в нашем случае он вначале разделит (пропорционально частотам символов) интервал [0, 1) на [0, 0.11) и [0.11, 1). Поскольку число 0.0111 (код

цепочки "aaba") находится в первом из них, можно получить первый символ: "a". Затем делим первый подынтервал [0, 0.11) на [0, 0.1001) и [0.1001, 0.1100) (пропорционально частотам символов). Опять выбираем первый, так как 0 " 0.0111 " 0.1001. Продолжая этот процесс, мы однозначно декодируем все четыре символа. Для того, чтобы декодирующий алгоритм мог определить конец цепочки, мы можем либо передавать ее длину отдельно, либо добавить к алфавиту дополнительный уникальный символ - "конец цепочки".

При разработке этого метода возникают две проблемы: во-первых, необходима арифметика с плавающей точкой, теоретически, неограниченной точности, и, во-вторых, - результат кодирования становится известен лишь при окончании входного потока. Однако, дальнейшие исследования показывают [Rubin], что можно практически без потерь обойтись целочисленной арифметикой небольшой точности (16-32 разряда), а также добиться инкрементальной работы алгоритма: цифры кода могут выдаваться последовательно по мере чтения входного потока при ограничении числа символов входной цепочки каким-либо разумным числом.

LZW - кодирование

Двухступенчатое кодирование. Алгоритм Лемпеля-Зива

Все рассмотренные выше методы и модели кодирования предполагали в качестве входных данных цепочки символов (тексты) в некотором конечном алфавите. При этом оставался открытым вопрос о связи этого входного алфавита кодирующего алгоритма с данными, подлежащими упаковке (обычно также представленными в виде цепочек в алфавите (при байтовой организации обычно состоящем из 256 различных символов - значений байт).

В простейшем случае для кодирования в качестве входного алфавита можно использовать именно эти символы (байты) входного потока. Именно так работает метод squashing программы PKPAK (использовано статическое кодирование Хаффмана и двухпроходный алгоритм). Степень сжатия при этом относительно невелика - для текстовых файлов порядка 50%. Гораздо большей степени сжатия можно добиться при выделении из входного потока повторяющихся цепочек - блоков и кодирования ссылок на эти цепочки с построением хеш-таблиц от первого до n-го уровня.

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.

Метод, о котором и пойдет речь, принадлежит Лемпелю и Зиву, и обычно называется LZ-compression. Суть его состоит в следующем: упаковщик постоянно хранит некоторое количество последних обработанных символов в буфере. По мере обработки входного потока вновь поступившие символы попадают в конец буфера, сдвигая предшествующие символы и вытесняя самые старые. Размеры этого буфера, называемого также скользящим словарем (sliding dictionary), варьируются в разных реализациях кодирующих систем. Экспериментальным путем установлено, что программа LHarc использует 4-килобайтный буфер, LHA и PKZIP - 8-ми, а ARJ - 16-килобайтный.

Затем, после построения хеш таблиц алгоритм выделяет (путем поиска в словаре) самую длинную начальную подстроку входного потока, совпадающую с одной из подстрок в словаре, и выдает на выход пару (length, distance), где length - длина найденной в словаре подстроки, а distance — расстояние от нее до входной подстроки (то есть фактически индекс подстроки в буфере, вычтенный из его размера). В случае, если такая подстрока не найдена, в выходной поток просто копируется очередной символ входного потока.

В первоначальной версии алгоритма предлагалось использовать простейший поиск по всему словарю. Время сжатия при такой реализации было пропорционально произведению длины входного потока на размер буфера, что совсем непригодно для практического использования. Однако, в дальнейшем, было предложено использовать двоичное дерево и хеширование для быстрого поиска в словаре, что позволило на порядок поднять скорость работы алгоритма.

Таким образом, алгоритм Лемпеля-Зива преобразует один поток исходных символов в два параллельных потока длин и индексов в таблице (length + distance). Очевидно, что эти потоки являются потоками символов с двумя новыми алфавитами, и к ним можно применить один из упоминавшихся выше методов (RLE, кодирование Хаффмана или арифметическое кодирование).

Так мы приходим к схеме двухступенчатого кодирования - наиболее эффективной из практически используемых в настоящее время. При реализации этого метода необходимо добиться согласованного вывода обоих потоков в один файл. Эта проблема обычно решается путем поочередной записи кодов символов из обоих потоков.

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. Максимальный коэффициент 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 Лемпеля-Зива-Велча (Lempel-Ziv-Welch - LZW)

Данный алгоритм отличают высокая скорость работы как при упаковке, так и при распаковке, достаточно скромные требования к памяти и простая аппаратная реализация. Недостаток - низкая степень сжатия по сравнению со схемой двухступенчатого кодирования. Предположим, что у нас имеется словарь, хранящий строки текста и содержащий порядка от 2-х до 8-ми тысяч пронумерованных гнезд. Запишем в первые 256 гнезд строки, состоящие из одного символа, номер которого равен номеру гнезда. Алгоритм просматривает входной поток, разбивая его на подстроки и добавляя новые гнезда в конец словаря. Прочитаем несколько символов в строку s и найдем в словаре строку t - самый длинный префикс s. Пусть он найден в гнезде с номером n. Выведем

число n в выходной поток, переместим указатель входного потока на length(t) символов вперед и добавим в словарь новое гнездо, содержащее строку t+c, где c - очередной символ на входе (сразу по-сле t). Алгоритм преобразует поток символов на входе в поток индексов ячеек словаря на выходе. При размере словаря в 4096 гнезд можно передавать 12 бит на каждый индекс. Каждая распознанная цепочка добавляет в словарь одно гнездо. При переполнении словаря упаковщик может либо прекратить его заполнение, либо очистить (полностью или частично).

При практической реализации этого алгоритма следует учесть, что любое гнездо словаря, кроме самых первых, содержащих односимвольные цепочки, хранит копию некоторого другого гнезда, к которой в конец приписан один символ. Вследствие этого можно обойтись простой списочной структурой с одной связью.

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);
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.

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.

Фрактальное сжатие

Упрощённо ***фрактал*** - это структура, состоящая из подобных форм и рисунков разных масштабов и размеров. Этот термин впервые ввёл Бенуа Мальдеброт для описания повторяющихся рисунков, которые он нашёл во многих различных структурах. Он обнаружил, что фракталы можно описать математически и создавать при помощи очень простых алгоритмов.

Фрактальная архивация основана на том, что мы представляем изображение в более компактной форме - с помощью коэффициентов системы итерируемых функций (Iterated Function System - далее по тексту как IFS). Прежде чем

рассматривать сам процесс архивации, разберем, как IFS строит изображение, т.е. процесс декомпрессии.

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

Наиболее наглядно этот процесс продемонстрировал Барнсли в своей книге "Fractal Image Compression". Там введено понятие Фотокопировальной Машины, 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) уменьшать свой фрагмент изображения.

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" и "папоротник Барнсли". "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.

Большинство алгоритмов фрактального сжатия запатентованы и закрыты для свободного использования в программах.

BMP - формат

Формат файла BMP (сокращенно от BitMaP) - это "родной" формат растровой графики для Windows, поскольку он наиболее близко соответствует внутреннему формату Windows, в котором эта система хранит свои растровые массивы. Для имени файла, представленного в BMP-формате, чаще всего используется расширение BMP, хотя некоторые файлы имеют расширение RLE, означающее run length encoding (кодирование длины серий). Расширение RLE имени файла обычно указывает на то, что произведено сжатие растровой информации файла одним из двух способов сжатия RLE, которые допустимы для файлов BMP-формата.

В файлах BMP информация о цвете каждого пикселя кодируется 1, 4, 8, 16 или 24 бит (бит/пиксель). Числом бит/пиксель, называемым также глубиной представления цвета, определяется максимальное число цветов в изображении. Изображение при глубине 1 бит/пиксель может иметь всего два цвета, а при глубине 24 бит/пиксель - более 16 млн. различных цветов.

На приведенной схеме показана структура типичного BMP-файла, содержащего 256-цветное изображение (с глубиной 8 бит/пиксель). Файл разбит на четыре основные раздела: заголовок файла растровой графики, информационный заголовок растрового массива, таблица цветов и собственно данные растрового массива. Заголовок файла растровой графики содержит информацию о файле, в том числе адрес, с которого начинается область данных растрового массива. В информационном заголовке растрового массива содержатся сведения об изображении, хранящемся в файле, например, его высоте и ширине в пикселях. В таблице цветов представлены значения основных цветов RGB (красный, зеленый, синий) для используемых в изображении цветов. Программы, считывающие и отображающие BMP-файлы, в случае использования видеоадаптеров, которые не позволяют отображать более 256 цветов, для точной цветопередачи могут программно устанавливать такие значения RGB в цветовых палитрах адаптеров.

Формат собственно данных растрового массива в файле BMP зависит от числа бит, используемых для кодирования данных о цвете каждого пикселя. При 256-цветном изображении каждый пиксель в той части файла, где содержатся собственно данные растрового массива, описывается одним байтом (8 бит). Это описание пикселя не представляет значений цветов RGB, а служит указателем для входа в таблицу цветов файла. Таким образом, если в качестве первого значения цвета RGB в таблице цветов файла BMP хранится R/G/B=255/0/0, то значению пикселя 0 в растровом массиве будет поставлен в соответствие ярко-

красный цвет. Значения пикселей хранятся в порядке их расположения слева направо, начиная (как правило) с нижней строки изображения. Таким образом, в 256-цветном BMP-файле первый байт данных растрового массива представляет собой индекс для цвета пикселя, находящегося в нижнем левом углу изображения; второй байт представляет индекс для цвета соседнего справа пиксела и т. д. Если число байт в каждой строке нечетно, то к каждой строке добавляется дополнительный байт, чтобы выровнять данные растрового массива по 16-бит границам.

Не все файлы BMP имеют структуру, подобную показанной на схеме. Например, файлы BMP с глубиной 16 и 24 бит/пиксель не имеют таблиц цветов; в этих файлах значения пикселей растрового массива непосредственно характеризуют значения цветов RGB. Также могут различаться внутренние форматы хранения отдельных разделов файла. Например, информация растрового массива в некоторых 16 и 256-цветных BMP-файлах может сжиматься посредством алгоритма RLE, который заменяет последовательности идентичных пикселей изображения на лексемы, определяющие число пикселей в последовательности и их цвет. В Windows допускается работа с BMP-файлами стиля OS/2, в которых используются различные форматы информационного заголовка растрового массива и таблицы цветов.

Структура файла BMP
Заголовок файла растровой графики (14 байт)
Сигнатура файла BMP (2 байт)
Размер файла (4 байт)
Не используется (2 байт)
Не используется (2 байт)
Местонахождение данных растрового массива (4 байт)

Информационный заголовок растрового массива (40 байт)
Длина этого заголовка (4 байт)
Ширина изображения (4 байт)
Высота изображения (4 байт)
Число цветовых плоскостей (2 байт)
Бит/пиксел (2 байт)
Метод сжатия (4 байт)
Длина растрового массива (4 байт)
Горизонтальное разрешение (4 байт)
Вертикальное разрешение (4 байт)
Число цветов изображения (4 байт)
Число основных цветов (4 байт)
Таблица цветов (длина изменяется от 8 до 1024 байт)
Собственно данные растрового массива (длина переменная)

Сжатие JPEG

Аббревиатура JPEG происходит от названия комитета по стандартам Joint Photographic Experts Group (Объединённая группа экспертов по фотографии). Фактически JPEG не является одним алгоритмом сжатия, это целый набор методов сжатия.

Данный Алгоритм сжатия используются и предназначен для сжатия картинок то есть растровых изображений(растровое изображение - это когда каждый

пиксель на картинки задаётся числом которое представляет собой номер цвета в текущей палитре) растровое изображение хранится в формате BMP. Поэтому любая картинка представляет собой массив пикселей в Паскале это выглядит так

Pixels: Array [1..256,1..256] of Byte

Цвет каждого пикселя кодируется Байтом(байт это восемь бит- а бит это либо ноль либо один поэтому один байт это 256 в десятичной системе) Так вот у нас есть массив который содержит полную информацию о картинке картинка размером 256*256*256 - третье измерение это цвет. Размер этой картинки, 65536 байт- ровно 64 Кбайта.

В алгоритме JPEG исходное изображение представляется двумерной матрицей размера *N*N*, элементами которой являются цвет или яркость пикселя.

Этапы работы алгоритма JPEG

- Дискретное косинус преобразование
- Этап Кодирования
- Этап Вторичного Сжатия

Процесс сжатия изображения JPEG достаточно сложен и часто для достижения приемлемой производительности требует специальной аппаратуры. Вначале изображение разбивается на квадратные блоки со стороной размером 8 пикселей. Затем производится сжатие каждого блока отдельно за три шага. На первом шаге с помощью формулы дискретного косинусоидального преобразования фуры (DCT) производится преобразование блока 8x8 с информацией о пикселях в матрицу 8x8 амплитудных значений, отражающих различные частоты (скорости изменения цвета) в изображении. На втором шаге значения матрицы амплитуд делятся на значения матрицы квантования,

которая смещена так, чтобы отфильтровать амплитуды, незначительно влияющие на общий вид изображения. На третьем и последнем шаге квантованная матрица амплитуд сжимается с использованием алгоритма сжатия без потерь.

Поскольку в квантованной матрице отсутствует значительная доля высокочастотной информации, имеющейся в исходной матрице, первая часто сжимается до половины своего первоначального размера или даже еще больше. Реальные фотографические изображения часто совсем невозможно сжать с помощью методов сжатия без потерь, поэтому 50%-ное сжатие следует признать достаточно хорошим. С другой стороны, применяя методы сжатия без потерь, можно сжимать некоторые изображения на 90%. Такие изображения плохо подходят для сжатия методом JPEG.

При сжатии методом JPEG потери информации происходят на втором шаге процесса. Чем больше значения в матрице квантования, тем больше отбрасывается информации из изображения и тем более плотно сжимается изображение. Компромисс состоит в том, что более высокие значения квантования приводят к худшему качеству изображения. При формировании изображения JPEG пользователь устанавливает показатель качества, величине которого "управляет" значениями матрицы квантования. Оптимальные показатели качества, обеспечивающие лучший баланс между коэффициентом сжатия и качеством изображения, различны для разных изображений и обычно могут быть найдены только методом проб и ошибок.

Файлы GIF

Большинство ведущих специалистов-графиков, имеющих дело с алгоритмом LZW, сталкиваются с аналогичными юридическими проблемами при использовании популярного межплатформенного формата файлов растровой

графики GIF (Graphics Interchange Format - формат обмена графическими данными, произносится "джиф"), разработанного компанией CompuServe. Обычно для имени файлов GIF используется расширение GIF, и тысячи таких файлов можно получить в CompuServe.

Структура файла GIF зависит от версии GIF-спецификации, которой соответствует файл. В настоящее время используются две версии, GIF87a и GIF89a. Первая из них проще. Независимо от номера версии, файл GIF начинается с 13-байт заголовка, содержащего сигнатуру, которая идентифицирует этот файл в качестве GIF-файла, номер версии GIF и другую информацию. Если файл хранит всего одно изображение, вслед за заголовком обычно располагается общая таблица цветов, определяющая цвета изображения. Если в файле хранится несколько изображений (формат GIF, аналогично TIFF, позволяет в одном файле кодировать два и больше изображений), то вместо общей таблицы цветов каждое изображение сопровождается локальной таблицей цветов.

В файле GIF87a вслед за заголовком и общей таблицей цветов размещается изображение, которое может быть первым из нескольких располагаемых подряд изображений. Каждое изображение состоит из 10-байт описателя изображения, расположенной вслед за ним локальной таблицы цветов и битов растрового массива. Для повышения эффективности использования памяти данные растрового массива сжимаются с помощью алгоритма LZW.

Файлы GIF89a имеют аналогичную структуру, но они могут содержать факультативные блоки расширения с дополнительной информацией о каждом изображении. В спецификации GIF89a определены четыре типа блоков расширения. Это блоки расширения для управления графикой, которые описывают, как изображение должно выводиться на экран (например, накладывается ли оно на предыдущее изображение подобно диапозитиву или

просто заменяет его); блоки расширения с обычным текстом, содержащие текст, отображаемый вместе с графикой; блоки расширения для комментария, содержащие комментарии в коде ASCII; и блоки расширения прикладных программ, в которых хранится информация, принадлежащая только создавшей этот файл программе. Блоки расширения могут находиться практически в любом месте файла после общей таблицы цветов.

Основные достоинства GIF заключаются в широком распространении этого формата и его компактности. Но ему присущи два достаточно серьезных недостатка. Один из них состоит в том, что в изображениях, хранящихся в виде GIF-файла, не может быть использовано более 256 цветов. Второй, возможно, еще более серьезный, заключается в том, что разработчики программ, использующие в них форматы GIF, должны иметь лицензионное соглашение с CompuServe и вносить плату за каждый экземпляр программы; такая ценовая политика была принята CompuServe после того, как Unisys объявила, что начнет добиваться соблюдения своих прав собственности и потребовала от тех, кто пользуется алгоритмом сжатия LZW, вносить лицензионные платежи. Возникшее в результате этого запутанное юридическое положение тормозит внедрение программистами в свои графические программы средств для работы с файлами GIF.

Вейвлет-сжатие

Английское название рекурсивного сжатия - wavelet. На русский оно также переводится как волновое сжатие, и как сжатие с использованием всплесков. 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. Коэффициент сжатия задается и варьируется в

пределах 5-100 раз. При попытке задать больший коэффициент, на резких границах, особенно проходящих по диагонали, проявляется "лестничный эффект" - ступеньки разной яркости, размером в несколько пикселей.

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 a2i и a2i+1 всегда можно представить в виде b1i=(a2i+a2i+1)/2 и b2i=(a2i-a2i+1)/2. Аналогично последовательность ai может быть попарно переведена в последовательность b1,2i.

Разберем конкретный пример: пусть мы сжимаем строку из 8 значений яркости пикселей (ai): (220, 211, 212, 218, 217, 214, 210, 202). Мы получим следующие последовательности b1i, и b2i: (215.5, 215, 215.5, 206) и (4.5, -3, 0.5, 4). Заметим, что значения b2i достаточно близки к 0. Повторим операцию, рассматривая b1i как ai. 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 коэффициентов сжатия.

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, что упрощает показ "огрубленного" изображения по заголовку.

В отличие от JPEG и фрактального алгоритма данный метод не оперирует блоками, например, 8x8 пикселей. 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.

Сжатие звука MP3

Цифровой звук, если это не музыка, которую можно закодировать в виде MIDI, столь же неудобен для сжатия, как и картинка. Звуковой сигнал редко обладает избыточностью, т.е. имеет повторяющиеся участки (в основном из-за шумов). А значит, плохо сжимается с использованием алгоритмов компрессии без потерь, аналогичных LZW или методу Хаффмана.

В 1940 г. Харви Флетчер, выдающийся американский физик, отец стереозвука, привлёк для исследований человеческого слуха большое число испытуемых. Он проанализировал зависимость абсолютного порога слышимости от частоты сигнала, т.е. при какой амплитуде звук определённой частоты не слышен для человека. В построенной на основе опытов кривой максимальные значения находятся, как и ожидалось, на границах диапазона слышимости (около 20 Гц и ближе к 20 кГц), а минимум - приблизительно 5 кГц. Но главное , на что он обратил внимание, - это способность слуха адаптироваться к появлению новых звуков, что выражается в повышении порога слышимости. Иначе говоря, одни звуки способны делать неслышимыми другие, что называют маскированием одного звука другим.

Последнее свойство слуха при компрессии позволяет после громкого звукового сигнала некоторое непродолжительное время вообще не воспроизводить, а значит и не сохранять никакого звука. Например, громкий щелчок продолжительностью в 0,1 с может замаскировать последующие за ним звуки на 0,5 с, которые не надо сохранять. Говорят, что коэффициент компрессии в этом примере достигает $6:1=\frac{0{,}5+0{,}1}{0{,}1}$, а описанную процедуру сжатия обычно называют *маскированием во временной области*.

При *маскировании в частотной области* синусоидальный сигнал маскирует более тихие, близкие по частоте сигналы, в том числе и синусоидальные сигналы много меньшей амплитуды. Удобно использовать разбиение спектра на полосы различной ширины, основываясь на особенностях слуха человека. Обычно выделяют 27 так называемых критических полос (critical band): 0-я от 50 до 95 Гц, 1-я от 95 до 140 Гц, …, 26-я от 20250 Гц и выше.

Для выполнения алгоритма сжатия исходный сигнал разбивается на кадры, которые подвергаются частотному анализу. Алгоритм сжатия выглядит примерно так:

1. При помощи специальных алгоритмов (ими могут быть быстрое преобразование Фурье или аналогичные), сигналы разделяются на 32 равные полосы спектра, при этом в одну получившуюся полосу могут попасть сразу несколько критических полос.
2. Используя так называемую психоакустическую модель (в которую, как правило, и входит частотное маскирование), определяют уровень маскирования полосы соседними.

3. Уровень в полосе, не превышающий вычисленный порог, считается равным нулю и не сохраняется. Наоборот, немаскированный уровень записывается в выходные данные.

В дальнейшем на каждый ненулевой уровень выделяется некоторое число битов, достаточное для его примерного представления. Так, в той части спектра, где человеческое ухо имеет наименьший порог слышимости, информация кодируется шестнадцатью битами, а на краях, там, где ухо менее чувствительно к искажениям, шестью и менее битами. К полученному потоку битов можно, например, применить алгоритм сжатия Хаффмана.

Различаются три версии алгоритма описанного MPEG-сжатием звука. В каждой версии данные разделяются на кадры, т.е. отдельный кадр состоит из 32 полос по 12 значений в каждой.

В MPEG layer1 (дословно "слой 1") в частотном фильтре используются один кадр и алгоритмы, основанные на дискретном косинусе - преобразовании (DCT). Психоакустическая модель задействует только частотное маскирование. Алгоритм позволяет упаковывать при соотношении 1:4 с потоком 384 Кбит/с.

MPEG layer2 использует три кадра в частотном фильтре(предыдущий, текущий и последующий) общий объём 32 полосы по 12 значений в 3 кадрах. Модель использует и временное маскирование. Упаковывает с соотношением от 1:6 до 1:8.

MPEG layer3, он же MP 3, использует частотный фильтр с разной величиной полос, психоакустическая модель включает временное маскирование сигнала. Имеет малые потери качества при высоком уровне компрессии: от 1:10 до 1:12.

Источники

1. Энциклопедия для детей/ Информатика. - М.: Аванта+, 2003.
2. Д.С.Ватолин. Алгоритмы сжатия изображений - М.: МГУ, 1999.
3. http://graphics.malchuk.ru/algorithm.html
4. http://www.ffke-campus.mipt.ru/~sany/sanyjpeg/main.html
5. http://arctest.narod.ru/descript/arithm.htm
6. http://book.itep.ru/2/26/indexs.htm