Алгоритм N 3 JPEG

Основные соотношения

JPEG - один из новых и достаточно мощных алгоритмов. Оперирует алгоритм областями 8x8, на которых яркость меняется сравнительно плавно. Вследствие этого при разложении матрицы такой, области в двойной ряд по косинусам значимыми оказываются только первые коэффициенты. Алгоритм разработан группой экспертов в области фотографии. JPEG - Joint Photographic Expert Group- подразделение в рамках ISO - Международной организации по стандартизации.

Метод позволяет сжимать некоторые изображения в 10-15 раз без серьезных потерь.

Существенными положительными сторонами алгоритма является то, что:

• задается степень сжатия (степень сжатия: 2-200 (задается пользователем));

• выходное цветное изображение может иметь 24 бита на точку. Класс изображений: полноцветные, 4 битовые изображения или изображения в градациях серого без резких переходов цветов.

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 горизонтальных и вертикальных переходов цветов.

В алгоритме JPEG используется ДКП, которое раскладывает изображение по амплитудам некоторых частот. Таким образом, при преобразовании получается матрица, 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. В самом простом случае - это арифметический побитовый сдвиг вправо. При этом преобразовании теряется часть информации, но может достигаться большая степень сжатия. Порядок работы прямого и обратного алгоритмов приведены на рисунке 1 и рисунке 2.

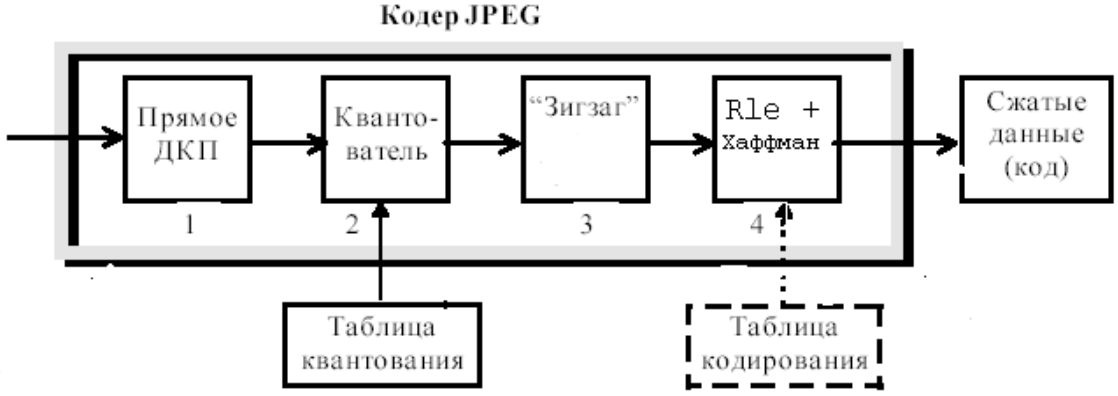


Рис. 1. Схема сжатия изображений по JPEG

Рис. 2. Схема восстановления изображений по JPEG

Для преобразования используются следующие формулы:

Формула прямого дискретного косинусного преобразования:

$$ДКП(i,j) = \frac{1}{\sqrt{2N}} C(i)C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} pixel(x,y) \cos\left[\frac{(2x+1)i\pi}{2N}\right] \cos\left[\frac{(2y+1)j\pi}{2N}\right]$$

$$C(x) = \begin{cases} \frac{1}{\sqrt{2}}, x = 0 \\ 1, x > 0 \end{cases}$$

Формула обратного дискретного косинусного преобразования.

$$ДКП(i,j) = \frac{1}{\sqrt{2N}} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} C(i)C(j) \; ДКП(i,j) \cos\left[\frac{(2x+1)i\pi}{2N}\right] \cos\left[\frac{(2y+1)j\pi}{2N}\right]$$

$$C(x) = \begin{cases} \frac{1}{\sqrt{2}}, x = 0 \\ 1, x > 0 \end{cases}$$

В получившейся матрице коэффициентов низкочастотные компоненты расположены ближе к левому верхнему углу, а высокочастотные - справа и внизу. Это важно потому, что большинство графических образов на экране компьютера состоит из низкочастотной информации. Высокочастотные компоненты не так важны для передачи изображения. Таким образом, дискретное косинусное преобразование позволяет определить, какую часть информации можно безболезненно выбросить, не внося серьезных искажений в картинку.

Время, необходимое для вычисления каждого элемента матрицы дискретного косинусного преобразования, сильно зависит от ее размера. Так как используются два вложенных цикла,

время вычислений составляет O (NxN). Одной из особенностей является то, что практически невозможно выполнить дискретное косинусное преобразование для всего изображения сразу. Поэтому изображение разбивается на блоки размером 8x8 точек. 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

$$C(i,u)=A(u)\times\cos\left(\frac{(2\times i+1)\times u\times\pi}{2\cdot n}\right)$$

$$A(u)=\begin{cases}\frac{1}{\sqrt{2}}, for & u=0\\ 1, for & u\neq 0\end{cases}$$

Значительно более эффективный вариант вычисления коэффициентов дискретного косинусного преобразования реализован через перемножение матриц.

При таком подходе формула дискретного косинусного преобразования может быть записана в следующем виде:

Прямое:

*ДКП = C * (Точки * Ст) (1)*

Обратное:

*Точки = Ст *(ДКП * C) (2)*

Где:

- ДКП - дискретное косинусное преобразование;

- C - матрица косинусного преобразования размером 8x8, элементы которой определяются по формуле

C(i,j) =| 1/sqrt(8) , при i=0;
| sqrt(2/8)*cos((2j+1)*i*3.1415, при i>0;

- *Точки* - матрица размером 8x8, состоящая из пикселов изображения;

- Ст - транспонированная матрица.

При перемножении матриц "цена" вычисления одного элемента результирующей матрицы составляют 8 умножений и 8 сложений, при вычислении матрицы дискретного косинусного преобразования - 2x8 соответственно. По сравнению с O(8x8) это заметное повышение производительности. Так как дискретное косинусное преобразование является разновидностью преобразования Фурье, то все методы ускорения преобразования Фурье могут быть применены и в данном случае.

Для оптимизации дискретного косинусного преобразования каждый элемент матрицы C умножается на константу sqrt(8).

$$C1\ (i,j) = C\ (i,j) * sqrt(8).$$

Таким образом матрица C1приобретает вид:

$$C1(i,j) = \begin{cases} 1, & \text{при } i=0; \\ sqrt(2)*\cos((2j+1)*i*3.1415), & \text{при } i>0; \end{cases}$$

Формулы (1) и (2) упрощаются:

$$ДКП = (C1 * (Точки * C1т))/8$$

$$Точки = (C1т * (ДКП * C1))/8$$

- C1т - транспонированная матрица C1.

Затем рассматривается матрица *C1* и с помощью тригонометрических преобразований, процедур разложения матриц на множители приводится к более простому виду.

Дискретное косинусное преобразование представляет собой преобразование информации без потерь и не осуществляет никакого сжатия. Дискретное косинусное преобразование подготавливает информацию для этапа сжатия с потерями или округления.

Квантование

Стандарт JPEG реализует эту процедуру через матрицу округления. Для каждого элемента матрицы дискретного косинусного преобразования существует соответствующий элемент матрицы округления. Результирующая матрица получается делением каждого элемента матрицы дискретного косинусного преобразования на соответствующий элемент матрицы округления и последующим округлением результата до ближайшего целого числа. Как правило, значения элементов матрицы округления растут по направлению слева направо и сверху вниз.

Выбор матрицы округления. На этом шаге осуществляется управление степенью сжатия и происходят самые большие потери. Понятно, что, задавая матрицы квантования с большими коэффициентами, получается больше нулей и, следовательно, большая степень сжатия. От выбора матрицы округления зависит баланс между степенью сжатия изображения и его качеством после восстановления. Стандарт JPEG позволяет использовать любую матрицу округления, однако ISO разработала набор матриц округления.

Один из вариантов получения матрицы округления: при помощи очень простого алгоритма. Для того чтобы определить шаг роста значений в матрице округления, задается одно значение в диапазоне [1, 25], называемое фактором качества QualityFactor. Затем матрица заполняется следующим образом:

```
for (i = 0; i < N; i++)
```

for (j = 0; j < N; j++)

Matrix[i][j] = 1 + (1 + i + j) * QualityFactor;

Фактор качества задает интервал между соседними уровнями матрицы округления, расположенными на ее диагоналях. Пример, полученной таким образом матрицы округления, представлен на рис. 3

3	5	7	9	11	13	15	17
5	7	11	13	15	17	19	21
7	11	13	15	17	19	21	23
9	11	13	15	17	19	21	23
11	13	15	17	19	21	23	25
13	15	17	19	21	23	25	27
15	17	19	21	23	25	27	29
17	19	21	23	25	27	29	31

Рис. 3. Матрица округления с фактором качества, равным 2.

Второй из вариантов получения матрицы округления:

Это Матрица округления ***basic_table*** из стандарта JPEG:

16, 11, 10, 16, 24, 40, 51, 61,

12, 12, 14, 19, 26, 58, 60, 55,

14, 13, 16, 24, 40, 57, 69, 56,

14, 17, 22, 29, 51, 87, 80, 62,

18, 22, 37, 56, 68, 109, 103, 77,

24, 35, 55, 64, 81, 104, 113, 92,

49, 64, 78, 87, 103, 121, 120, 101,

72, 92, 95, 98, 112, 100, 103, 99

В стандарте коэффициент масштабирования QualityFactor рассчитывается таким образом:

```
if (QualityFactor <= 0) QualityFactor = 1;
if (QualityFactor > 100) QualityFactor = 100;
if (QualityFactor < 50)
  QualityFactor = 5000 / QualityFactor;
else
  QualityFactor = 200 - QualityFactor *2;
```

Далее вычисляется матрица квантования:

```
for (i = 0; i < 64; i++) {
```

```
    temp = ((long) basic_table[i] * QualityFactor + 50L) / 100;
    if (temp <= 0L) temp = 1;
    if (temp > 255)
      temp = 255;
    quantval[i] =  temp << 3;
  }
```

Проводится квантование матрицы *ДКП:*

```
for (i = 0; i < 8; i++)
for (j = 0; j < 8; j++)
     ДКП (i, j) = ДКП (i, j) / quantval[i*8+j];
```

Операция округления является единственной фазой работы JPEG, где происходит потеря информации.

Выбор соответствующей таблицы квантования является "высоким искусством". Большинство существующих компрессоров используют таблицу, разработанную Комитетом JPEG ISO.

"Зигзаг"-сканирование

Матрица 8x8 переводится в 64-элементный вектор при помощи "зигзаг"-сканирования, т.е. берутся элементы с индексами (0,0), (0,1), (1,0), (2,0)..., как показано на рисунке 4

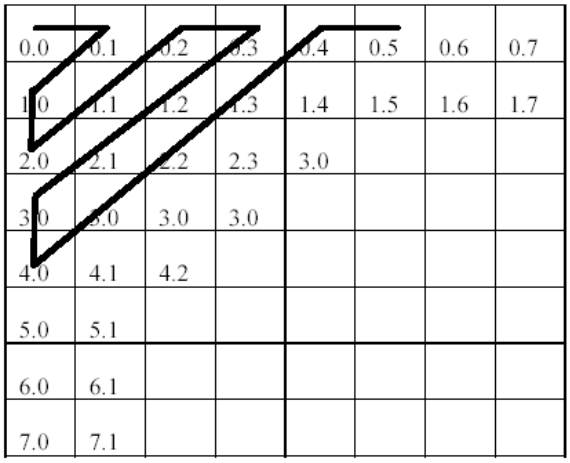


Рисунок 4. «Зигзаг»-сканирование в 64-элементном векторе

Таким образом, в начале вектора получаются коэффициенты матрицы, соответствующие низким частотам, а в конце - высоким. Полученный вектор свертывается с помощью алгоритма группового кодирования (RLE).

Алгоритм группового кодирования (RLE)

Алгоритм RLE (Run Length Encoding) заключается в следующем: любой последовательности повторяющихся входных символов ставится в соответствие набор из трех выходных символов: первый — байт префикса, говорящий о том, что встретилась входная повторяющаяся последовательность, второй — байт, определяющий длину входной последовательности, третий — сам входной символ, т.е. сжатый поток состоит из троек (prefix, length, symbol). Например, пусть имеется (шестнадцатеричный) текст вида:

05 05 05 05 05 05 01 01 03 03 03 03 03 03 05 03 FF FF FF FF,

состоящий из 20 байт. Выберается в качестве префикса байт FF. Тогда на выходе алгоритма получается последовательность:

FF 06 05 FF 02 01 FF 06 03 FF 01 05 FF 01 03 FF 04 FF,

длина которой равна 18 байт, то есть достигнуто некоторое сжатие. Однако, нетрудно заметить, что при кодировании некоторых символов размер выходного кода возрастает (например, вместо 01 01 — FF 02 01). Очевидно, одиночные или дважды (трижды) повторяющиеся символы кодировать не имеет смысла — их надо записывать в явном виде. Получим новую последовательность:

FF 06 05 01 01 FF 06 03 05 03 FF 04 FAA

длиной 13 байт. Достигнутая степень сжатия: 13/20 = 65%. Нетрудно заметить, что префикс может совпасть с одним из входных символов. В этом случае входной символ может быть заменен своим «префиксным» представлением, например: FF то же самое, что и FF 01 FF (три байта вместо одного). Поэтому, от правильного выбора префикса зависит качество самого алгоритма сжатия, так как, если бы в исходной последовательность встречались одиночные байты FF, размер выходных данных мог бы превысить размер входных. В общем случае в качестве префикса следует выбирать самый редкий символ входной последовательности. Можно еще повысить степень сжатия, если объединить префикс и длину в одном байте. Пусть префикс — число от F0 до FF, где вторая цифра определяет длину от 0 до 15. В этом случае выходной код будет двухбайтным, но при этом сужается диапазон представления длины с 255 до 15 байтов. Для вышеуказанного примера выходной код будет иметь следующий вид:

F6 05 F2 01 F6 03 05 03 F4 FF

Длина — 10 байт, степень сжатия — 50%. Таким образом, имеется несколько возможностей применения алгоритма RLE для кодирования выходных данных после зигзагообразного упорядочивания. В простейшем случае результат RLE — пары типа <пропустить, число>, где "пропустить" является счетчиком пропускаемых нулей, а "число" - значение, которое необходимо поставить в следующую ячейку. Т.е., вектор 42 3000-2 0000 1 ... будет свернут в пары (0,42) (0,3) (3,-2) (4,1).... Получившиеся пары кодируются по Хаффману.

Кодирование по Хаффману

В основе алгоритма кодирования Хаффмана лежит довольно простой принцип: символы заменяются кодовыми последовательностями различной длины. Чем чаще используется символ, тем короче должна быть кодовая последовательность. Именно поэтому алгоритм Хаффмана называется также кодированием символами переменной длины (Variable-Lenth Coding). Код переменной длины позволяет записывать наиболее часто встречающиеся символы короткими кодовыми последовательностями, а редко встречающиеся – более длинными. Например, для английского текста символам E,T и A можно поставить в соответствие 3-битовые последовательности, а J, Z и Q – 8-битовые. В одних алгоритмах реализации алгоритма Хаффмана используются готовые кодовые таблицы, в других – кодовая таблица строится только на основе статистического анализа имеющейся информации. Кодирование по Хаффману гарантирует возможность полного последующего декодирования.

Алгоритм основан на том факте, что некоторые символы из стандартного 256-символьного набора в произвольном тексте могут встречаться чаще среднего периода повтора, а другие, соответственно, – реже. Следовательно, если для записи распространенных символов использовать короткие последовательности бит, длиной меньше 8, а для записи редких символов – длинные, то суммарный объем файла уменьшится.

Алгоритм Хаффмана с фиксированной таблицей CCITT GROUP 3

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

В таблицах, приведенных ниже (табл.1.1 и 1.2), заданы два вида кодов:

коды совершения серий – заданы с 2 до 63 с шагом 1;

составные (дополнительные) коды - заданы с 64 до 2560 с шагом 64.

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

На практике в тех случаях, когда в изображении преобладает черный цвет, изображение инвертируется перед компрессией и записывается информация 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;
                   ЗаписьБелыйКодДля(L2);
             }
             ЗаписьБелыйКодДля(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 для каждой строки нашего изображения (достаточно длинной, начинающейся белой точки) выходной битовый поток получается вида:

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

где $(\ )^{*}$ - повтор 0 или более раз; $(\ )^{+}$ - повтор 1 или более раз; $[\ ]$ - включение 1 или 0 раз.

Для проведенного ранее примера: 0, 3, 556, 10 …алгоритм сформирует следующий код $< Б-0>< Ч-3>< Б-512>< Б-44>< Ч-10>$, или, согласно таблице, 00110101100110010100101101000010 0 (разные коды в потоке выделены для удобства). Этот

код обладает свойствами префиксных кодов и легко может быть свернут обратно в последовательность длин серий, Легко подсчитать, для приведенной строки в 569 бит получается код длиной в 33 бита, т.е. степень сжатия составляет примерно 17 раз.
Единственное «сложное» выражение в алгоритме это:

L2=МаксимальныйДопКодМеньшеL(L) – на практике работает очень просто $L2 = (L >> 6)^* 64$, где >> - битовый сдвиг L влево на 6 бит (можно сделать то же самое за одну побитовую операцию & - логическое И).

Таблицы 1.1 и 1.2 построены с помощью классического алгоритма Хаффмана (отдельно для длин черных и белых серий). Значение вероятностей появления для конкретных длин серий были получены путем анализа большего количества факсимильных изображений.

Таблица 1.1. Коды завершения

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

Таблица 1.2 Составные коды

Длина серии	Код белой подстроки	Код черной подстроки	Длина серии	Код белой подстроки	Код черной подстроки
64	11011	0000001111	1344	011011010	0000001010011
128	10010	000011001000	1408	011011011	0000001010100
192	01011	000011001001	1472	010011000	0000001010101
256	0110111	000001011011	1536	010011001	0000001011010
320	00110110	000000110011	1600	010011010	0000001011011
384	00110111	000000110100	1664	011000	0000001100100
448	01100100	000000110101	1728	010011011	0000001100101
512	01100101	0000001101100	1792	00000001000	Совп.с белой
576	01101000	0000001101101	1856	00000001100	-//-
640	01100111	0000001001010	1920	00000001101	-//-
704	011001100	0000001001011	1984	000000010010	-//-
768	011001101	0000001001100	2048	000000010011	-//-
832	011010010	0000001001101	2112	000000010100	-//-
896	011010011	0000001110010	2176	000000010101	-//-
960	011010100	0000001110011	2240	000000010110	-//-
1024	011010101	0000001110100	2304	000000011100	-//-
1088	011010110	0000001110101	2368	000000011101	-//-
1152	011010111	0000001110110	2432	000000011110	-//-
1216	011011000	0000001110111	2496	000000011111	-//-
1280	011011001	0000001010010	2560		-//-

Характеристики алгоритма CCITT Group3

Степень сжатия: лучшая стремится в пределе к 213.(3), средняя 2, в худшем случае увеличивает файл в 5 раз.

Класс изображений: двуцветные черно-белые изображения, в которых преобладает большие пространства, заполненные белым цветом (рас.1.1. и 1.2.).

Симметричность: близка к единице.

Характерные особенности: данный алгоритм чрезвычайно прост в реализации, быстр и может быть легко реализован аппаратно.

Сжатие по алгоритму Хаффмана

Сжимая файл по алгоритму Хаффмана, первое, что нужно сделать - это необходимо прочитать файл полностью и подсчитать сколько раз встречается каждый символ из расширенного набора ASCII. После подсчета частоты вхождения каждого символа, необходимо просмотреть таблицу кодов ASCII и сформировать мнимую компоновку между кодами по убыванию. То есть, не меняя местонахождение каждого символа из таблицы в памяти, отсортировать таблицу ссылок на них по убыванию. Каждая ссылку из последней таблицы - "узел". В дальнейшем (в дереве) будут размещаться указатели, которые будут указывать на этот "узел".

Имеется файл длиной в 100 байт, имеющий 6 различных символов в себе. Подсчитано вхождение каждого из символов в файл и получено следующее:

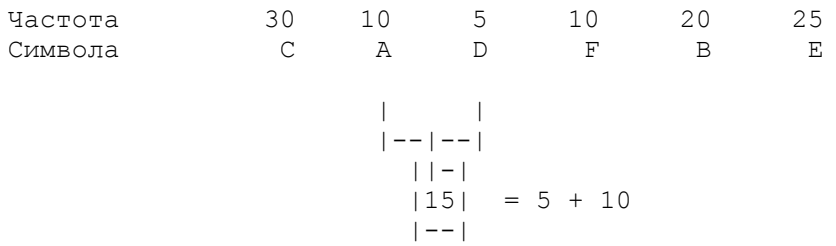
символ	A	B	C	D	E	F
число вхождений	10	20	30	5	25	10

Эти числа - частота вхождения для каждого символа. Таблица размещается, как ниже.

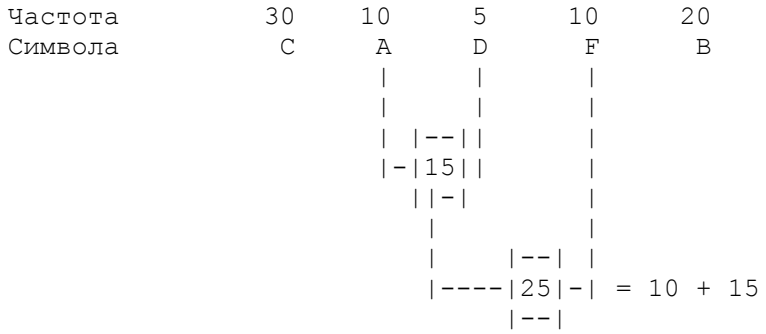
символ	C	E	B	F	A	D
число вхождений	30	25	20	10	10	5

Из последней таблицы выбираются символы с наименьшей частотой. В данном случае это D (5) и какой либо символ из F или A (10), можно взять любой из них, например A.

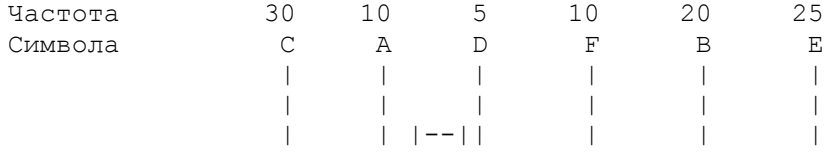
Из "узлов" D и A создается новый "узел", частота вхождения, для которого будет равна сумме частот D и A:

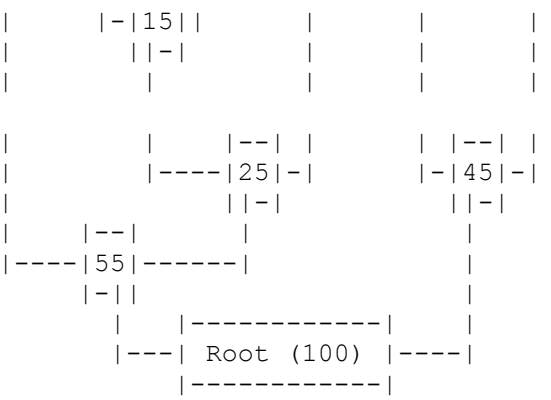


Номер в рамке - сумма частот символов D и A. Теперь снова ищутся два символа с самыми низкими частотами вхождения, исключая из просмотра D и A и рассматривая вместо них новый "узел" с суммарной частотой вхождения. Самая низкая частота теперь у F и нового "узла". Снова делается операция слияния узлов:



Рассматривается таблица снова для следующих двух символов (B и E). Продолжается этот процесс, пока все "дерево" не сформируется, т.е. пока все не сведется к одному узлу.





Теперь, когда дерево создано, можно кодировать файл. Нужно всегда начинать из корня (Root). Кодируя первый символ (лист дерева C), прослеживаются все повороты ветвей вверх по дереву. И если делается левый поворот, то запоминается 0-й бит, и аналогично 1-й бит для правого поворота. Так для C, нужно идти влево к 55 (запоминается 0), затем снова влево (0) к самому символу. Код Хаффмана для символа C - 00. Для следующего символа (A) получается - лево, право, лево, лево, что выливается в последовательность 0100. Выполнив выше сказанное для всех символов, получается

```
C = 00   ( 2 бита )
A = 0100 ( 4 бита )
D = 0101 ( 4 бита )
F = 011  ( 3 бита )
B = 10   ( 2 бита )
E = 11   ( 2 бита )
```

Каждый символ изначально представлялся 8-ю битами (один байт), и так как уменьшилось число битов необходимых для представления каждого символа, то, следовательно, уменьшился размер выходного файла. Сжатие складывается следующим образом:

Частота	первоначально	уплотненные биты	уменьшено на
C 30	30 x 8 = 240	30 x 2 = 60	180
A 10	10 x 8 = 80	10 x 3 = 30	50
D 5	5 x 8 = 40	5 x 4 = 20	20
F 10	10 x 8 = 80	10 x 4 = 40	40
B 20	20 x 8 = 160	20 x 2 = 40	120
E 25	25 x 8 = 200	25 x 2 = 50	150

```
Первоначальный размер файла : 100 байт - 800 бит;
      Размер сжатого файла :  30 байт - 240 бит;
240 - 30% из 800 , так что мы сжали этот файл на 70%.
```

Все это довольно хорошо, но неприятность находится в том факте, что для восстановления первоначального файла, нужно иметь декодирующее дерево, так как деревья будут различны для разных файлов. Следовательно, необходимо сохранять дерево вместе с файлом. Это превращается в итоге в увеличение размеров выходного файла. В данной методике сжатия и каждом узле находятся 4 байта указателя, по этому, полная таблица для 256 байт будет приблизительно 1 Кбайт длиной.

Таблица в данном примере имеет 5 узлов плюс 6 вершин (где и находятся символы), всего 11. 4 байта 11 раз - 44. Если добавить после небольшое количество байтов для сохранения места узла и некоторую другую статистику - таблица будет приблизительно 50 байтов длины.

Добавив к 30 байтам сжатой информации, 50 байтов таблицы получается, что общая длина архивного файла вырастет до 80 байт. Учитывая, что первоначальная длина файла в рассматриваемом примере была 100 байт - получается 20% сжатие информации.

Алгоритм JPG

1. Вводится коэффициент качества. Рассчитывается матрица квантования.
2. Вводится матрица изображения. Из каждого элемента матрицы вычитается 128.
3. Матрица разбивается на блоки размерностью 8*8.
4. Для каждого блока производятся следующие вычисления:

4.1.

Temp = C1 * Точки;

dataptr = Temp * C1т/8;

Производится квантование.
Производится "зигзаг"-сканирование.

5. Выполняется кодирование RLE+Хаффман.

Один из вариантов вычисления (пункты 4.1, 4.2 алгоритма):

```
#ifdef SHORTxSHORT_32
  #define MULTIPLY16C16(var,const)  (((INT16) (var)) * ((INT16) (const)))
#endif
#ifdef SHORTxLCONST_32
  #define MULTIPLY16C16(var,const)  (((INT16) (var)) * ((long) (const)))
#endif
#ifndef MULTIPLY16C16
  #define MULTIPLY16C16(var,const)  ((var) * (const))
#endif
#ifdef SHORTxSHORT_32
  #define MULTIPLY16V16(var1,var2)  (((INT16) (var1)) * ((INT16) (var2)))
#endif
#ifndef MULTIPLY16V16
   #define MULTIPLY16V16(var1,var2)  ((var1) * (var2))
#endif
#ifdef SHORTxSHORT_32
  #define MULTIPLY16V16(var1,var2)  (((long) (var1)) * ((long) (var2)))
#endif
#define MULTIPLY(var,const)  MULTIPLY16C16(var,const)
#define DIVIDE_BY(a,b)     a /= b
#define DCTSIZE ((int))  8)
#define FIX_0_298631336  ((long)  2446)
#define FIX_0_390180644  ((long)  3196)
#define FIX_0_541196100  ((long)  4433)
#define FIX_0_765366865  ((long)  6270)
#define FIX_0_899976223  ((long)  7373)
#define FIX_1_175875602  ((long)  9633)
#define FIX_1_501321110  ((long)  12299)
#define FIX_1_847759065  ((long)  15137)
#define FIX_1_961570560  ((long)  16069)
```

```
#define FIX_2_053119869  ((long)  16819)
#define FIX_2_562915447  ((long)  20995)
#define FIX_3_072711026  ((long)  25172)
#define CONST_BITS  13
#define PASS1_BITS  2
#ifdef RIGHT_SHIFT_IS_UNSIGNED
  #define SHIFT_TEMPS     longshift_temp;
  #define RIGHT_SHIFT(x,shft)  \
        ((shift_temp = (x)) < 0 ? \
         (shift_temp >> (shft)) | ((~((long) 0)) << (32-(shft))) : \
         (shift_temp >> (shft)))
#else
  #define SHIFT_TEMPS
  #define RIGHT_SHIFT(x,shft)     ((x) >> (shft))
#endif
#define ONE  ((long) 1)
#define DESCALE(x,n)  RIGHT_SHIFT((x) + (ONE << ((n)-1)), n)

 long tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
 long tmp10, tmp11, tmp12, tmp13;
 long z1, z2, z3, z4, z5;
 int *dataptr;
 int ctr;

 int DCTSIZE = 8;
 dataptr = workspaceptr;
 for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
   tmp0 = dataptr[0] + dataptr[7];
   tmp7 = dataptr[0] - dataptr[7];
   tmp1 = dataptr[1] + dataptr[6];
   tmp6 = dataptr[1] - dataptr[6];
   tmp2 = dataptr[2] + dataptr[5];
   tmp5 = dataptr[2] - dataptr[5];
   tmp3 = dataptr[3] + dataptr[4];
   tmp4 = dataptr[3] - dataptr[4];
   tmp10 = tmp0 + tmp3;
   tmp13 = tmp0 - tmp3;
   tmp11 = tmp1 + tmp2;
   tmp12 = tmp1 - tmp2;
   dataptr[0] = (int) ((tmp10 + tmp11) << PASS1_BITS);
   dataptr[4] = (int) ((tmp10 - tmp11) << PASS1_BITS);
   z1 = MULTIPLY(tmp12 + tmp13, FIX_0_541196100);
   double temp = FIX_0_541196100;
    temp =  CONST_BITS;
    temp =  PASS1_BITS;
//     temp =    BITS_IN_JSAMPLE;
   temp =  temp;
   dataptr[2] = (int) DESCALE(z1 + MULTIPLY(tmp13, FIX_0_765366865),
                              CONST_BITS-PASS1_BITS);
   dataptr[6] = (int) DESCALE(z1 + MULTIPLY(tmp12, - FIX_1_847759065),
                              CONST_BITS-PASS1_BITS);
   z1 = tmp4 + tmp7;
   z2 = tmp5 + tmp6;
   z3 = tmp4 + tmp6;
   z4 = tmp5 + tmp7;
```

```
  z5 = MULTIPLY(z3 + z4, FIX_1_175875602);
  tmp4 = MULTIPLY(tmp4, FIX_0_298631336);
  tmp5 = MULTIPLY(tmp5, FIX_2_053119869);
  tmp6 = MULTIPLY(tmp6, FIX_3_072711026);
  tmp7 = MULTIPLY(tmp7, FIX_1_501321110);
  z1 = MULTIPLY(z1, - FIX_0_899976223);
  z2 = MULTIPLY(z2, - FIX_2_562915447);
  z3 = MULTIPLY(z3, - FIX_1_961570560);
  z4 = MULTIPLY(z4, - FIX_0_390180644);
  z3 += z5;
  z4 += z5;
  dataptr[7] = (int) DESCALE(tmp4 + z1 + z3, CONST_BITS-PASS1_BITS);
  dataptr[5] = (int) DESCALE(tmp5 + z2 + z4, CONST_BITS-PASS1_BITS);
  dataptr[3] = (int) DESCALE(tmp6 + z2 + z3, CONST_BITS-PASS1_BITS);
  dataptr[1] = (int) DESCALE(tmp7 + z1 + z4, CONST_BITS-PASS1_BITS);
  dataptr += DCTSIZE;
}
dataptr = workspaceptr;
for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
  tmp0 = dataptr[DCTSIZE*0] + dataptr[DCTSIZE*7];
  tmp7 = dataptr[DCTSIZE*0] - dataptr[DCTSIZE*7];
  tmp1 = dataptr[DCTSIZE*1] + dataptr[DCTSIZE*6];
  tmp6 = dataptr[DCTSIZE*1] - dataptr[DCTSIZE*6];
  tmp2 = dataptr[DCTSIZE*2] + dataptr[DCTSIZE*5];
  tmp5 = dataptr[DCTSIZE*2] - dataptr[DCTSIZE*5];
  tmp3 = dataptr[DCTSIZE*3] + dataptr[DCTSIZE*4];
  tmp4 = dataptr[DCTSIZE*3] - dataptr[DCTSIZE*4];
  tmp10 = tmp0 + tmp3;
  tmp13 = tmp0 - tmp3;
  tmp11 = tmp1 + tmp2;
  tmp12 = tmp1 - tmp2;
  dataptr[DCTSIZE*0] = (int) DESCALE(tmp10 + tmp11, PASS1_BITS);
  dataptr[DCTSIZE*4] = (int) DESCALE(tmp10 - tmp11, PASS1_BITS);
  z1 = MULTIPLY(tmp12 + tmp13, FIX_0_541196100);
  dataptr[DCTSIZE*2] = (int) DESCALE(z1 + MULTIPLY(tmp13, FIX_0_765366865),
                                     CONST_BITS+PASS1_BITS);
  dataptr[DCTSIZE*6] = (int) DESCALE(z1 + MULTIPLY(tmp12, - FIX_1_847759065),
                                     CONST_BITS+PASS1_BITS);
  z1 = tmp4 + tmp7;
  z2 = tmp5 + tmp6;
  z3 = tmp4 + tmp6;
  z4 = tmp5 + tmp7;
  z5 = MULTIPLY(z3 + z4, FIX_1_175875602);
  tmp4 = MULTIPLY(tmp4, FIX_0_298631336);
  tmp5 = MULTIPLY(tmp5, FIX_2_053119869);
  tmp6 = MULTIPLY(tmp6, FIX_3_072711026);
  tmp7 = MULTIPLY(tmp7, FIX_1_501321110);
  z1 = MULTIPLY(z1, - FIX_0_899976223);
  z2 = MULTIPLY(z2, - FIX_2_562915447);
  z3 = MULTIPLY(z3, - FIX_1_961570560);
  z4 = MULTIPLY(z4, - FIX_0_390180644);
  z3 += z5;
  z4 += z5;
  dataptr[DCTSIZE*7] = (int) DESCALE(tmp4 + z1 + z3,
                                     CONST_BITS+PASS1_BITS);
```

```
  dataptr[DCTSIZE*5] = (int) DESCALE(tmp5 + z2 + z4,
                                     CONST_BITS+PASS1_BITS);
  dataptr[DCTSIZE*3] = (int) DESCALE(tmp6 + z2 + z3,
                                     CONST_BITS+PASS1_BITS);
  dataptr[DCTSIZE*1] = (int) DESCALE(tmp7 + z1 + z4,
                                     CONST_BITS+PASS1_BITS);
  dataptr++;
 }

int temp, qval, DCTSIZE2=64;
      int i;
      BYTE output_ptr[64];// = coef_blocks[bi];
     for (i = 0; i < DCTSIZE2; i++) {
        qval = table.quantval [i];
        temp = workspaceptr[i];
        if (temp < 0) {
          temp = -temp;
          temp += qval>>1;
          DIVIDE_BY(temp, qval);
          temp = -temp;
        } else {
          temp += qval>>1;
          DIVIDE_BY(temp, qval);
        }
        output_ptr[i] =  temp;
     }
```

Приложение

Структура входных и выходных данных модулей для упаковки и распаковки матриц

```
struct  TImageSubRect { /////////////// riXXXXXXXX ////////////////// Y ^
////////////////////////////////// in VMemory ////////////////////   |      Lx
/////////////////////////////////////////////////////////////////////   |   +++++++++
 BYTE    *riAv; //  (A_i)Адрес  подблока изображения /////////////////   |   +++++++++ Ly
   long   riDx; //  (D_x)Смещение по X подблока ЦИ в пикселах           |Dy +++++++++
   long   riDy; //  (D_y)Смещение по Y подблока ЦИ в пикселах           |   Dx
   long   riCx; //  (L_x)Ширина подблока изображения в пикселах         +--------------> X
   long   riCy; //  (L_y)Высота подблока изображения в пикселах == в байтах
   long   riWx; //  (B_x)Ширина подблока ЦИ в байтах (длина строки подблока),
   short  riBp; //  (B_p)Число  составляющих пиксела  ImSamples == { [1]..3, 4, ...}
   short  riEs; //  (E_b)Размер составляющей Sample в байтах    == { [1], 2, 4, ...}
        BYTE riSwapBR;// (B_p==3) -> Inverse colors: SwapValue(b,r)== {  0 | 1       }
        BYTE  ribNoBMP;//  НЕ  ИМЕЕТ  ЗНАЧЕНИЯ:  НАЛИЧИЕ  riWx  уже  достаточно.|>>
bSwapBytesOfWord
   // ^для PNG и др.: Исп.при(riEs==2) для определения обратного к INTEL расп. байт в WORD
        BYTE ribLCD;  // {0|1}1: aSImage представляет адрес L-подблока, за ним следует R...
   union {BYTE riMask;  // =1..15: Маска каналов, пока используется только в  WPEL
        BYTE riBits;  // Число БИТ в одной составляющей пиксела= 8..12,...
   };
```