JPEG - Idea and Practice/The Huffman coding

The main difference between our procedure in *part one* and the real JPEG procedure is that in *part one* we used a method of coding which is easy to understand and use, but which was not very efficient, partly because it was based on frequencies that were more determined by our desire for a simple coding procedure than by reality. JPEG uses the more efficient Huffman coding and frequencies that either are determined by the actual picture or by the average values for a number of typical pictures. Furthermore, we used the same coding procedure for all the numbers, whereas JPEG uses different coding for the DC and the AC numbers and also different coding for the Y component and for the two colour components - this implying that the coding can demand tables of more than 450 numbers.

We will here choose Huffman tables based on typical frequencies, rather than on the frequencies measured by a pre-scanning of the actual picture. Therefore we only need to know how the Huffman encoding and decoding is to be performed once we have the necessary tables: we do not need to know how these tables are constructed on the basis of frequency. We will, however, show the procedure for the construction of the Huffman tables. It is a rather simple procedure, and the reader might want to make a program that measures frequencies and constructs the Huffman tables from the actual picture (we will show the programs in *Appendix 2*).

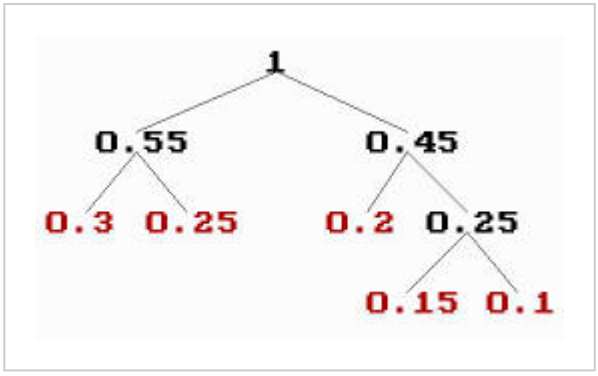
Assume that we have some values a1, a2, ..., an, which are attached to frequencies and which are to be equipped with code words so that the most frequently used values get the shortest codes. This can be done by constructing a so-called *Huffman tree* with the values as *leaves* with attached frequencies. Usually a Huffman tree can be constructed in several ways giving different code lengths. JPEG chooses the following:

We order the values according to decreasing frequency. For the two last values we add their frequencies, remove the two values and insert a *node* at the place among the remaining values where this frequency belongs (so that the frequencies are still decreasing - note that if the new frequency occurs among the others, the insertion can be made in more than one way). This is repeated until there is only one node left, and this will have frequency 1. We have for each operation removed two things: either two values, or two nodes, or a value and a node. We construct the Huffman tree by placing the values (leaves) at the bottom and successively connecting with lines the pairs of removed things with the node that has replaced them.

If, for example, the values are the numbers 0, 1, 2, 3 and 4, and their frequencies are 0.3, 0.25, 0.2, 0.15 and 0.1 (having sum 1), respectively, the removal procedure could look like this:

a1	0	0.3	0.3	0.45	0.55	1
a2	1	0.25	0.25	0.3	0.45	
a3	2	0.2	0.25	0.25		
a4	3	0.15	0.2			
a5	4	0.1				

And the Huffman tree could look like this:



Huffman tree

The length of the Huffman code assigned to a value is the number of lines from the value to the last node (the top node with frequency 1). Once we know the lengths (of the codes) assigned to the values, we can form the code words, and this can be done in different ways:

By using the Huffman tree, we can code for instance by writing 1 when we go to the right and 0 when we go to the left when we progress from the value towards the top node:

0	00
1	10
2	01
3	011
4	111

But we can also code without the Huffman tree, what is essential is the code lengths for the values. For instance, we can code so that the sequence of code words (identified by numbers via their binary digit expressions) is increasing: forming consecutive numbers when the code length is unaltered and adjoining zeros when the code length increases:

0	00
1	01
2	10
3	110
4	111

It is this last way of forming codes that is used by JPEG, because it is fast to decode.

In JPEG a code word must not consist of only 1's. We can avoid this by adding provisionally an extra value whose frequency is half (for instance) of the frequency of the last and least value (and finally remove a code from the codes of the largest length).

Furthermore, the length of a code word must not exceed 16. Therefore, if the Huffman tree leads to code lengths of more than 16 bits, the longest codes must successively be shortened. In our case, where we have imported the coding, we do not need to care about this problem, but we will briefly describe it: the longest code length is assigned to an even numbers of values. Therefore we can shorten the longest length by one bit (the last) and assign this code to one of the values, if we can find another (shorter) code to the other value. Assuming that the last (longest) codes with fewer bits have j bits, we can remove the last of these codes (of length j) and extend it by a 0 and a 1, respectively, so that we get two new codes of length j+1 which can replace the two removed codes.

The Huffman coding is performed from the (Huffman) values (occurring in the picture) and the code length assigned to each value (determined by its frequency). Therefore our point of departure is two lists of bytes: the first, called BITS, goes from 1 to 16, and tells us, for each of these numbers, the number of codes of this code length. The second, called HUFFVAL, reels off, for each code length having a non-zero number of codes, the values to be coded with codes of this length (and as many values as there are codes of this length). The values in HUFFVAL are called the Huffman values, and they are ordered according to increasing code length (within a given code length the ordering is arbitrary).

In our program we use these lists for the DC numbers of the Y component:

BITS

0 1 5 1 1 1 1 1 1 0 0 0 0 0 0 0

HUFFVAL

0
1 2 3 4 5
6
7
8
9
10
11

and these lists for the DC numbers of the two colour components:

BITS

0 3 1 1 1 1 1 1 1 1 1 0 0 0 0 0

HUFFVAL

0 1 2
3
4
5
6
7
8
9
10
11

The last lists tell that there are: 0 codes of length 1, 3 codes of length 2 (coding the Huffman values 0, 1 and 2), 1 code of length 3 (coding the Huffman value 3), etc.

Most of the numbers to be coded are AC numbers, and they are coded in another way than the DC numbers. Moreover, the values range a larger interval. As we import the Huffman coding, we must use lists containing all the possible values.

In our program we use these lists for the AC numbers of the Y component:

BITS

0 2 1 3 3 2 4 3 5 5 4 4 0 0 1 125

HUFFVAL

1 2
3
0 4 17
5 18 33
49 65
6 19 81 97
7 34 113
20 50 129 145 161
8 35 66 177 193
21 82 209 240
36 51 98 114
130
9 10 22 23 24 25 26 37 38 39 40 41 42 52 53 54 55 56 57 58 67 68 69 70 71 72 73
74 83 84 85 86 87 88 89 90 99 100 101 102 103 104 105 106 115 116 117 118 119
120 121 122 131 132 133 134 135 136 137 138 146 147 148 149 150 151 152 153
154 162 163 164 165 166 167 168 169 170 178 179 180 181 182 183 184 185 186
194 195 196 197 198 199 200 201 202 210 211 212 213 214 215 216 217 218 225
226 227 228 229 230 231 232 233 234 241 242 243 244 245 246 247 248 249 250

and these lists for the AC numbers of the two colour components:

BITS

0 2 1 2 4 4 3 4 7 5 4 4 0 1 2 119

HUFFVAL

0 1
2
3 17
4 5 33 49
6 18 65 81
7 97 113
19 34 50 129
8 20 66 145 161 177 193
9 35 51 82 240
21 98 114 209
10 22 36 52
225
37 241
23 24 25 26 38 39 40 41 42 53 54 55 56 57 58 67 68 69 70 71 72 73 74 83 84 85 86
87 88 89 90 99 100 101 102 103 104 105 106 115 116 117 118 119 120 121 122
130 131 132 133 134 135 136 137 138 146 147 148 149 150 151 152 153 154 162
163 164 165 166 167 168 169 170 178 179 180 181 182 183 184 185 186 194 195
196 197 198 199 200 201 202 210 211 212 213 214 215 216 217 218 226 227 228
229 230 231 232 233 234 242 243 244 245 246 247 248 249 250

If we call the number of Huffman values nhv, we have an array HUFFVAL[k] from k = 1 to nhv arranging the Huffman values in their enumerated order. From the list BITS[i] we form an array HUFFSIZE[k] from k = 1 to nhv of the code lengths i for which the number BITS[i] is non-zero, each i repeated BITS[i] times, so that the array HUFFSIZE[k] is parallel to HUFFVAL[k]. And we now

construct an array HUFFCODE[k] from k = 1 to nhv stating the Huffman code assigned to HUFFVAL[k]. We identify a code with the integer having the bits of the code as binary digit expression (e.g. 110 = 6), being aware that as the code can start with one or more zeros, the digit expression must start with zeros in order to get the right length (e.g. 011 = 3).

The code words are generated in this way: assume that we have formed all the codes of length <= n, and that the last formed code is the number c. Now assume that the next code length is n+i, then the next code is $c = 2^i \cdot (c + 1)$ (the code got by joining i zeros to c + 1), and the following codes are the consecutive numbers (c+1, c+2, ...), so many as there are codes of (the new) length n = n+i. At the start c is set to 0. Code number k, HUFFCODE[k], is the code assigned to the Huffman value HUFFVAL[k].

The encoding

For the encoding we reorder the lists (arrays) HUFFSIZE and HUFFCODE so that they become functions of the *Huffman values* (instead of functions of the order number), forming arrays EHUFSI[val] and EHUFCO[val]:

```
if val = HUFFVAL[k] then

    EHUFSI[val] = HUFFSIZE[k] and
    EHUFCO[val] = HUFFCODE[k]
```

Note that EHUFCO[val] is an array: EHUFCO[val][j] is the j-th bit of the code.

If we let the function *size*(n) (n integer) state the number of digits in the binary digit expression of n, and let *digit*(n) be the digit expression itself (so that *digit*(n) is an array of bits from 1 to *size*(n)), the procedures for the construction of HUFFSIZE[k], HUFFCODE[k], EHUFSI[val] and EHUFCO[val] (and which are to be applied for each Huffman table) could look like the following:

```
k = 1
i = 1
j = 1

1

    if j <= bits[i] then

        begin

            huffsize[k] = i
            k = k + 1
            j = j + 1
            goto 1

        end

    i = i + 1
    j = 1
    if i <= 16 then

        goto 1

    nhv = k - 1
    k = 1
```

```
    c = 0
    i = huffsize[k]

2

    huffcode[k] = c
    c = c + 1
    if k = nhv then

        goto 4

    k = k + 1
    if huffsize[k] = i then

        goto 2

3

    c = 2 * c
    i = i + 1
    if huffsize[k] = i then

        goto 2

    else

        goto 3

4

    k = 1

5

    val = huffval[k]
    e = huffsize[k]
    ehufsi[val] = e
    l = size(huffcode[k])
    dig = digit(huffcode[k])
    if l < e then

        for j = 1 to e - l do

            ehufco[val, j] = 0

    for j = 1 to l do

        ehufco[val, e - l + j] = dig[j]

    k = k + 1
    if k <= nhv then

        goto 5
```

For the lists above for the DC numbers for the Y component, nhv = 12, HUFFSIZE[k] is the sequence 2 3 3 3 3 3 4 5 6 7 8 9, and HUFFCODE[k] is the sequence 00, 010, 011, 100, 101, 110, 1110, 11100, 111000, 1110000, 11100000, 111000000. And for the functions EHUFSI[val] and EHUFCO[val], we have: EHUFSI[0] = 2, EHUFSI[1] = 3, EHUFSI[2] = 3, etc., and EHUFCO[0] = 00, EHUFCO[1] = 010, EHUFCO[2] = 011, etc.

In the encoding we must for non-negative integer n know how many digits are in the binary expression of n. The function *size*(n) states this number, and it is extended to negative n by letting -n have the same size as n. It is given by *size*(0) = 0 and *size*(n) = trunc(ln(abs(n)) / ln(2)+0.000001) + 1 for n <> 0:

n	size
0	0
1	1
2, 3	2
4 ... 7	3
8 ... 15	4
16 ... 31	5
32 ... 63	6
64 .. 127	7
128 .. 255	8
256 .. 511	9
512 .. 1023	10
1024 .. 2047	11

etc.

The integer the binary digit expression of which follows a Huffman code, can be negative, and (as explained in *part one*) we do not need an extra bit to indicate this: the digit expression will always begins with 1 and we can write 0 instead of the 1. At the decoding of the sequence, the start with 0 will then show that the number is negative, and 1 followed by the rest of the digits will be the binary expression of the numerical value. However, in order to indicate that the number is negative, JPEG has chosen to replace *all* the digits by their opposite bit (forming the *complement* of the number). Therefore, if the digit expression begins with 0, has val digits and corresponds to the (non-negative) integer n, then the negative integer is $-(2^{val} - 1 - n)$ (in T.81 it is said that if the sequence of digits begins with 0 and if the number of digits is T, then we get the numerical value by adding $2^T + 1$ to the number, but this is not correct, the number of course is obtained by subtracting it from $2^T - 1 = 11...1$ (T figures 1)).

The program for the function, *digit*(n) (n <> 0), giving the binary digit expression for the integer n, when n is positive, and the complement to the digit expression, when n is negative, could look like this:

```
j = size(n)
if n < 0 then

    n = round(exp(j * ln(2))) - 1 - abs(n)

if j = 1 then

    digit[1] = n

else

    begin
```

```
        j = j - 1
        q = round(exp(j * ln(2)))
        i = 0
        while i <= j do

            begin

                i = i + 1
                l = n div q
                n = n - l * q
                q = q div 2
                digit[i] = l

            end

    end
```

The DC numbers: For a DC number (the first number of the 64-array) it is not the number itself, but the difference DIFF between the number and the preceding DC number which is to be coded, and it is not DIFF itself, but the number val of bits needed to express it: val = *size*(DIFF). The code is then EHUFCO[val] and after this comes the val binary digits of DIFF: *digit*(DIFF)[j], j = 1, ..., val.

The AC numbers: The 63 AC numbers (of the 64-array) are encoded in another way than the DC number. Here the size of the actual number (not a difference) is coded, and since there are usually many zeros in an AC array, the number of these in an uninterrupted row is *combined* with the size of the following non-zero AC number. If there are m zeros before the non-zero AC number n and if the size of n is k, we combine these two numbers (being half bytes) to the byte val = m*16 + k, and it is this byte that is Huffman coded. This presupposes, however, that m and k really are half bytes (that is, <= 15). k is always <= 11, but there can be more than 15 zeros in a row, therefore, when a row of zeros has reached 15 and is followed by another zero, we must code these 16 zeros separately: the byte to be coded is val = 15*16 + 0 = 240 (called ZRL). If the last of the 63 AC numbers is zeros, this is indicated by writing the Huffman code assigned to val = 0*16 + 0 = 0 (called EOB, End-Of-Block). After the Huffman code has been written, the k binary digits of the non-zero AC number are written in the same way as for the DC (or rather the DIFF) numbers. Frequencies and code lengths are assigned to all the (Huffman) values val = m*16 + k that are constructed in this way (or at least those values occurring in the picture). The number of Huffman values (to be coded) can at most be the number of possible zeros (0, 1, ..., 15, that is, 16) times the number of possible sizes of the non-zero AC numbers (namely 10), and in addition to this product (160), the two extra values 240 and 0. In total 162 Huffman values. As we here have chosen to import Huffman tables based on tests of a number of casual pictures, our AC Huffman tables most contain 162 values.

The decoding

For the decoding (when the file is read)(instead of the arrays EHUFSI[val] and EHUFCO[val]) we must have constructed beforehand three arrays from k = 1 to 16: the minimum (first) code of length number k, MINCODE[k], the maximum (last) code of length number k, MAXCODE[k], and the *number* of MINCODE[k] in the sequence of the codes (and Huffman values), VALPTR[k] (value pointer):

```
    j = 0
    k = 0
```

```
        k = k + 1
        if k > 16 then

                goto fin

        if bits[k] = 0 then

                begin

                        maxcode[k] = -1
                        goto 0

                end

        j = j + 1
        valptr[k] = j
        mincode[k] = huffcode[j]
        j = j + bits[k] - 1
        maxcode[k] = huffcode[j]
        goto 0
```

fin

Note that when there are no codes of code length k, MAXCODE[k] = -1, and MINCODE[k] and VALPTR[k] are not defined.

Decoding then goes on as following: In the stream of bits, the first thing to do is to collect as many together that they form a code: we must determine where to stop. We start with k = 0, c = 0 and MAXCODE[0] = -1 (so that c > MAXCODE[0]), and for each read bit we join this to c and increase k by 1, until c <= MAXCODE[k]. Since we identify codes with numbers, the joining means that we set c = 2*c + bit for each new bit (called bit). The code then is c, and we shall find the Huffman value val assigned to c, and this is the Huffman value having the number k = VALPTR[k] + c - MINCODE[k], so that val = HUFFVAL[k]:

```
        k = 0
        c = 0
        while c > maxcode[k] do

                begin

                        nbit
                        c = 2 * c + bit
                        k = k + 1

                end

        val = huffval[valptr[k] + c - mincode[k]]
```

Here *nbit* is the procedure described later, which reads the next bit.
