Chapter 6

Image Compression

Data Compression reduces the amount of data that represents a certain information. Specific characteristics of digital images makes compression more efficient. According to [1] there are three data redundancies that can be identified in digital images: coding, interpixel and psychovisual. Video compression may use all these redundancies in each frame and a forth one called temporal redundancy, which uses the similarity between frames from the sequence.

6.1 Introduction to compression

Coding makes use of the distribution (probabilities with which each value appears) in the image. One can change the code to minimise the total length of the data representing the image (see example in tables 6.1 and 6.2).

Some images have many values repeated in neighbour pixels, as in the case where whole regions are homogeneous. Inter-pixel redundancy can be partially eliminated if this neighbourhood information is made available.

The human perception of brightness is limited, making some information contained in images redundant. This particular way of compression uses psychovisual redundancies. Although the information lost after compression might not be essential to the human visual processing, the image loses information that could be important for further processing. Therefore, for certain types of image processing it is more convenient to work with uncompressed images to ensure accuracy.

Compression ratio is defined as:

$$C_r = \frac{d_1}{d_2} \tag{6.1}$$

where: d_1 and d_2 are the amounts of data in the original and compressed images respectively.

6.2 Loss-less compression

Loss-less compression uses methods with which it is possible to recover the original image from the compressed one. The information about each pixel is perfectly preserved. These methods have a limitation regarding the compression ratio predicted by the Information Theory.

6.2.1 Shannon's Information Theory and Entropy

Shannon was concerned with the transmission of information and with its fundamental properties. He came with the concept of *information entropy*, based on the thermodynamics idea of

entropy. Information entropy is given by the following equation:

$$H = -\sum_{i=0}^{N} p_i \, log(p_i) \tag{6.2}$$

Where: p_i is the probability of a symbol to appear in a stream.

What does entropy mean? If we take the *log* base 2 we can measure the entropy based on binary. Let's assume that we have a system with 8 symbols. We need 3 bits to represent each one of the 8 symbols (000, 001, 010, 011, 100, 101, 110, and 111). If a stream has equal probabilities for all i symbols, it means that the number of occurrences is the same for all of the symbols (probability = $\frac{1}{8}$). In this case, the entropy is:

$$H = -\sum_{i=0}^{N} p_i \, log_2(p_i) = -8 \, \frac{1}{8} \, log_2(\frac{1}{8}) = 3$$

So the highest possible entropy H occurs when all the probabilities are the same. For binary entropy typically the maximum entropy is equals the number of bits in the system.

What has to happen for the entropy to be somewhat smaller than the maximum? When symbols appear in different quantities in the stream, then entropy is lower. For example, suppose that in the 3 bits system we have symbols A, B, C, D, E, F, G and H. The probabilities are: A(0.4) B(0.3) C(0.1) D(0.1) E(0.06) F(0.04) G(0) and H(0). Notice that G and H never occur. The entropy of the stream is:

$$H = -(0.4 \, log_2(0.4) + 0.3 \, log_2(0.3) + 2 \; 0.1 \, log_2(0.1) + 0.06 \, log_2(0.06) + 0.04 \, log_2(0.04)) = 2.1435$$

In another example, let's assume that the only symbols that show in the stream are A and B in equal quantities (probabilities are 0.5). All other symbols have probabilities equals zero. The entropy would is:

$$H = -2 \, (0.5) \, log_2(0.5) = 1$$

The entropy measure in binaries would mean (approximately) the average number of bits that would take to represent the same stream using a *variable-length code*. In other words, instead of using a fixed number of bits to represent a symbol, we can use 1 bit to represent the symbol with the highest probability, 2 bits to represent the next highest probability and so on and so forth. In the first example where all symbols had the same probability, entropy was 3, indicating that this is the minimum number of bits to represent the stream. In the case of the 6 symbols stream, the entropy was 2.1435. It means that we could shrink the representation of the same stream from 3 to 2.1435 bits per symbol. In the second example where only A and B shows up, we only need 1 bit per symbol (e.g., A=0 and B=1 as no other symbol shows up).

This begs a question: how can we recover the information if we no longer have a clue about how many bits represent which symbol? Enters Huffman coding.

6.2.2 Huffman coding

Huffman published his method in 1952. This method has been popular for information compression and has been shown to yield a variable-length code that is close to optimum (although it is not optimum compared to other more complicated coding methods such as *arithmetic coding*).

The idea behind Huffman's method is to create a tree that can uniquely translate a combination of bits into a symbol and vice-versa. This allows to compress a sequence without any ambiguity to where a symbol bit sequence ends and where another sequence starts.

Algorithm 3 shows one version of the method using a heap to build a Huffman tree. The first step is to create an ordered list of probabilities for each symbol. Let's say each pixel value in an image is a symbol. In a digital image some pixel values might not appear or appear very rarely, so the heap might contain something less than 256 symbols. The symbols are then ordered by probability (in reverse, as the heap is a min-heap or a priority queue). The two symbols with the smallest probability form a non-leaf node of the tree, with the key as a sum of the children's probabilities. The non-leaf node is inserted back in the heap. This process is repeated until the heap has only one node: it becomes the root of the Huffman tree.

The second step is to code the set. This can be achieved by traversing the tree to find each leaf, and making a note of the bits: 0 if we follow the left pointer, 1 if we follow the right pointer. [1]

Algorithm 3 ***Huffman Coding***

Require: Heap (minHeap, or a priority queue) P
Require: Tree node T_{huf} and N tree nodes T_i, temporary tree nodes T_a, T_b and T
1: Create a leaf T_i for each symbol (using the probability as a key)
2: Insert all T_i in the heap P
3: **while** there is more than one node in the Heap P **do**
4: delete the two nodes from P (the two with lowest probability) and copy to T_a and T_b
5: create a **new** tree node T
6: make the probability of T equals to the sum of the probabilities of T_a + T_b
7: make T_a and T_b children (smallest on left) of the new node
8: insert the new tree node back into the heap P
9: **end while**
10: the remaining node in P (with probability 1) is the root of tree T_{huf}.
11: Traverse T_{huf}, marking 0 if going left and 1 if going right.
12: When reaching a leaf, output the code.

The Huffman tree is not unique, and depending on how the heap is inserted and deleted, we may have different trees. However, the codes obtained with different trees built with algorithm 3 are equivalent in terms of the bits per symbol ratio. A simple example can be seen in figure 6.1, where the three different trees yield different encoding, but are equivalent. Note that one of the trees is better balanced.

Figure 6.1 shows three different trees. The difference between the trees was caused by decisions made when building them: if there is a draw in the probabilities for certain symbols (it is used as a key in the heap), than the nodes may appear in different sides of the parent node in the tree. However, the coding is equivalent in terms of the final entropy and in the bits-per-symbol ratio.

The compression ratio can be calculated considering how many bits would be needed to represent the image using this new code. For the tree on the left and for the tree on the middle of figure 6.1, the compression ratio is:

$$C_r = \frac{3}{(0.4 * 1 + 0.3 * 2 + 0.1 * 3 + 0.1 * 4 + 0.06 * 5 + 0.04 * 5)} = \frac{3}{2.2} = 1.36$$

For the tree on the right of figure 6.1, the compression ratio is:

$$C_r = \frac{3}{(0.4 * 1 + 0.3 * 2 + 0.1 * 4 + 0.1 * 4 + 0.06 * 4 + 0.04 * 4)} = \frac{3}{2.2} = 1.36$$

[1] 0 to the left and 1 to the right is an arbitrary convention, we can invert that and have a different code of same entropy.

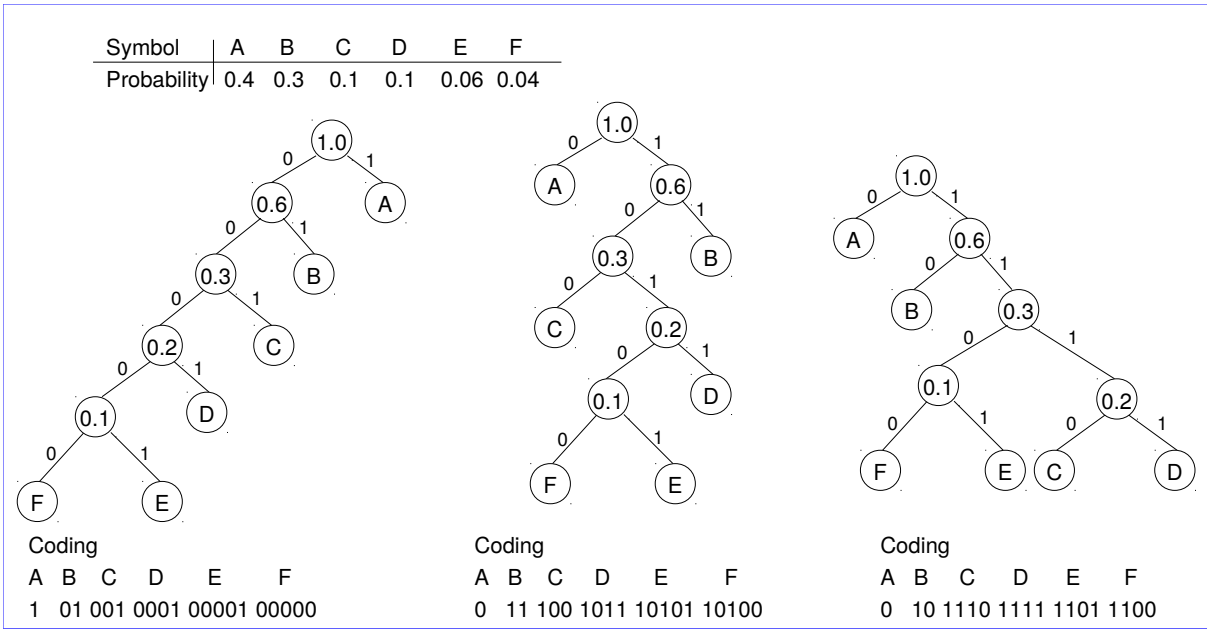

Figure 6.1: Huffman Example.

Note that the three trees result in exactly the same compression ratio. To fully understand the mechanism of the coding, let's use a fixed size string to encode. Suppose that the string is: ABCDEFABCDEFEDDDCCCABABABABABABABABABABABABAAAAA.

If you count the number of each symbol, you will find the following occurrences: A 20, B 15, C 5, D 5, E 3 and F 2. A histogram will result in the same probabilities as the examples above. Using the coding on the third tree in figure 6.1, we can encode the string as:

01011101111110111000101110111111011100110111111111111 etc (the remaining part is left as an exercise)

To decode a compressed string back to the original string, we can use the bits to find to which direction we should go from the root until we reach a leaf. When we reach a leaf, we translate the symbol and go back to the root and restart the process of traversing the tree. Let us use part of the string above:

0101110111111011100

The first bit is 0. Starting from the root, we reach node A. So the first symbol is A. The next bit is 1. Starting from the root, we go to the right and reach a non-leaf node. We carry on with the next bit, which is 0. This takes us to the left side and reach a leaf node with B. We translate 10 as B. The rest of the decoding is left as an exercise.

0 10 1110 1111 1101 1100
A 10 1110 1111 1101 1100
A B 1110 1111 1101 1100
A B C 1111 1101 1100
A B C D 1101 1100
A B C D E 1100
A B C D E F

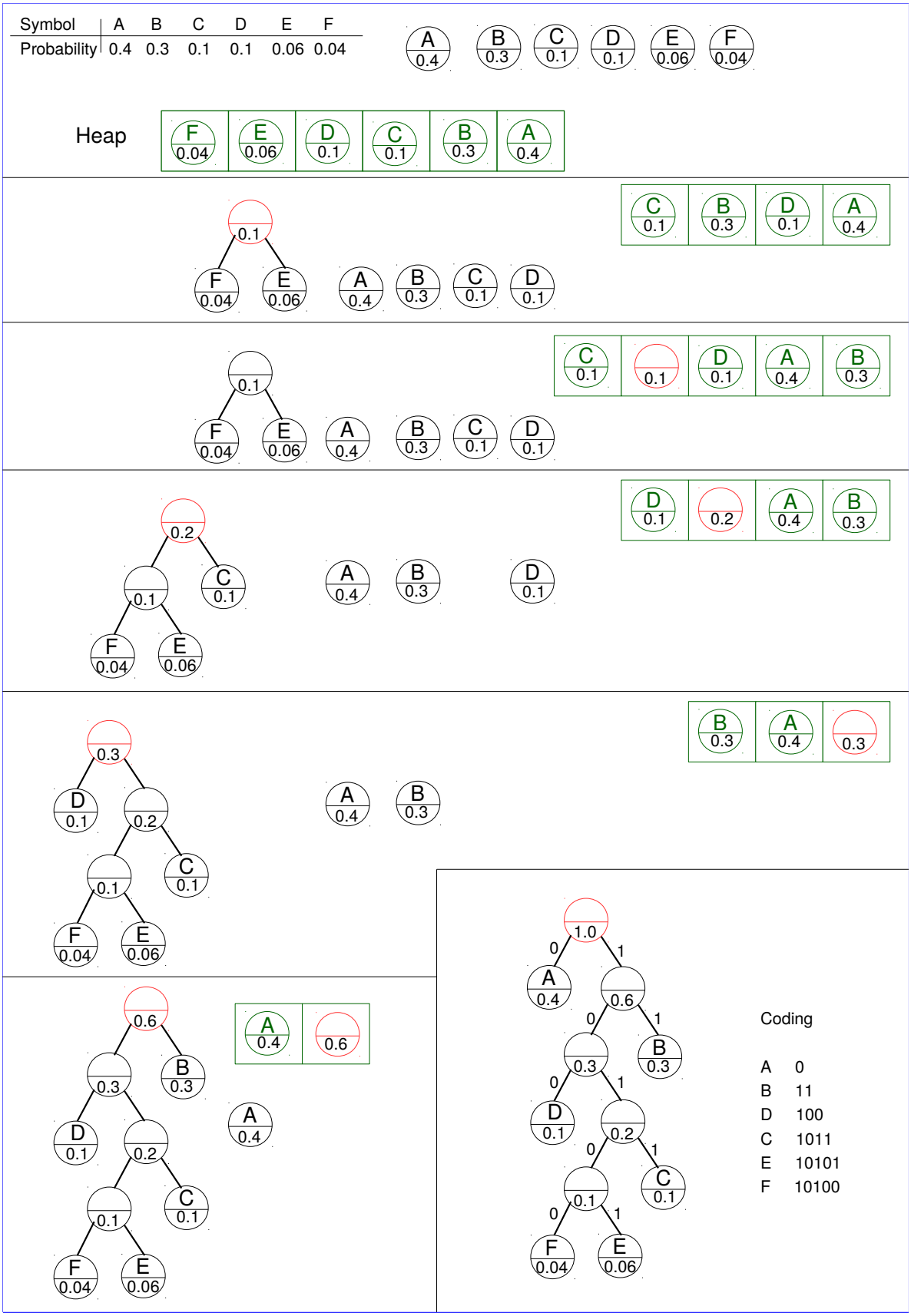

Figure 6.2: Building a Huffman Tree using algorithm 3

6.2.3 LZW

LZW (Lempel-Ziv-Welch) is a popular method that was incorporated in many file formats (for example GIF and TIF). This method was under patent until 2003 and for that reason it was not longer used in many open source applications and libraries (specially GIF). It uses the inter-pixel redundancies such as repetitions of certain value sequences.

The LZW algorithm starts with a dictionary that has all the pixel values. For example, for grey-scale images, the dictionary ranges from 0 to 255. The size of the dictionary is always larger than the original set. For the case of grey-scale images, one could start the process with a 9 bit dictionary, so all values between 256 and 511 are free.

Algorithm 4 is used for compression. The symbol $\oplus$ means that two strings are being appended. One of the challenges for this simple algorithm is to choose the correct size for the dictionary. If the dictionary is too big, than there are more bits per character than necessary. If it is too small and the image is large with little redundancy, than the dictionary will run out of entries before the compression is completed.

Algorithm 4 ***LZW compression***

Require: The input is a grey-scale *image* and the output is a *file*.
Require: Auxiliary structures: a string S, a character C and a vector $T[N]$ (the dictionary).
1: Initialise a table with a single value strings (from 0 to 255).
2: Initialise S = pixel[0]
3: **while** there are pixels in *image* **do**
4: C = get pixel[i]
5: **if** $S \oplus C$ is in the dictionary T **then**
6: $S = S \oplus C$
7: **else**
8: output the code for S
9: add $S \oplus C$ to the dictionary T
10: $S = C$
11: **end if**
12: **end while**
13: output the code for S

Let us use the image example found in [1]. The original sequence for a 16 pixels image is:

39 39 126 126 39 39 126 126 39 39 126 126 39 39 126 126

Using algorithm 4, the sequence obtained (table 6.1) is:

39 39 126 126 256 258 260 259 257 126

The compression ratio can be calculated based in the number of bits necessary for each image. For example, suppose that the image is using 8 bis per pixel. The dictionary has 9 bits. The compression is given by the total number of entries multiplied by 8 bits, then divided by the number of entries of the output of algorithm 4 multiplied by 9 bits:

$$C_r = \frac{16 * 8}{10 * 9} = \frac{128}{90} = 1.42$$

To decompress the image, algorithm 5 should be used. Table 6.2 shows how it works.

These algorithms represent the simplest version of the LZW method. Note that for certain sequences it may be that a dictionary entry is needed before it is created. If the entry does not

Table 6.1: LZW compression.

S	C	Output	Code	Dictionary String
39				
39	39	39	256	39-39
39	126	39	257	39-126
126	126	126	258	126-126
126	39	126	259	126-39
39	39			
39-39	126	256	260	39-39-126
126	126			
126-126	39	258	261	126-126-39
39	39			
39-39	126			
39-39-126	126	260	262	39-39-126-126
126	39			
126-39	39	259	263	126-39-39
39	126			
39-126	126	257	264	39-126-126
126		126		

Table 6.2: LZW decompression.

N	O	C	S	Dictionary String	Code
	39		39		
39	39	39	39	39-39	256
126	39	126	126	39-126	257
126	126	126	126	126-126	258
256	126	39	39-39	126-39	259
258	256	126	126-126	39-39-126	260
260	258	39	39-39-126	126-126-39	261
259	260	126	126-39	39-39-126-126	262
257	259	39	39-126	126-39-39	263
126	257	126	126	39-126-126	264

Algorithm 5 ***LZW decompression***

Require: The input is $file$ and the output is a grey-scale $image$
Require: Auxiliary structures: variable N, strings S,O, a character C and a vector $T[N]$.
1: Initialise a dictionary with single value strings (say from 0 to 255).
2: Read O = word[0] of $file$
3: S = get translation of O
4: output S to $image$
5: **while** there are words in $file$ **do**
6: Read N = word[i]
7: **if** N is in the dictionary T **then**
8: S = get translation of N
9: **else**
10: $S = O \oplus$ first character in O
11: **end if**
12: output S to $image$
13: C = first character in S
14: add $(O \oplus C)$ to the dictionary T
15: $O = S$ (or translation of N)
16: **end while**

yet exist in the dictionary, it can be computed by appending the previous output O with its own first character (statement 10 of algorithm 5).

6.3 Lossy compression

Some information contained in a digital image is not necessarily essential. A limited loss of information can lead to high compression ratios with a subtle change in the quality of the image

6.3.1 quantisation

The staircase quantisation is commonly used to achieve compression with a limited loss. This uses the principle that human beings do not perceive all the possible values for each pixel and therefore they can be clustered together. The new values obtained by some statistical approach can be coded. For example the Lloyd-Max quantisers use the minimisation of the mean-square errors in such a way that the staircase is not homogeneous (so more values are quantised in some steps than in others).

6.3.2 Discrete Cosine Transform (DCT)

The equation for a *Discrete Cosine Transform* (DCT) for a 2D square sized image (NxN) $i(x, y)$ is:

$$C(u,v) = \alpha(u)\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} i(x,y).cos\left[\frac{(2x+1)u\pi}{2N}\right].cos\left[\frac{(2y+1)v\pi}{2N}\right] \tag{6.3}$$

Where:
$\alpha(0) = \sqrt{1/N}$
$\alpha(u) = \sqrt{2/N} for u = 1, 2, ..., N-1$

To recover the image given the coefficients $C(u, v)$, the following equation is used:

$$i(x, y) = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} \alpha(u)\alpha(v)C(u, v).cos\left[\frac{(2x+1)u\pi}{2N}\right].cos\left[\frac{(2y+1)v\pi}{2N}\right] \quad (6.4)$$

The coefficients $C(u, v)$ are computed to form a 2D matrix of the same size as the image. The coefficients can be positive or negative depending on the combination of cosines. In order to save runtime, a floating point NxN matrix can be pre-calculated, so one only needs to multiply the elements of this matrix by the pixel values for that block. This still requires N^2 multiplications per block though it avoids the re-computation of the whole equation 6.3. For that reason, transforms like DCT are often represented by an image showing the basis functions (see figure 6.3 for examples with 4x4 and 8x8 basis functions.). Of course these images just roughly represent what the coefficients are because to show it we have to round up values to integers and choose a corresponding multiplier for them (e.g., the smallest negative coefficient equals zero, and the largest coefficient equals 255).

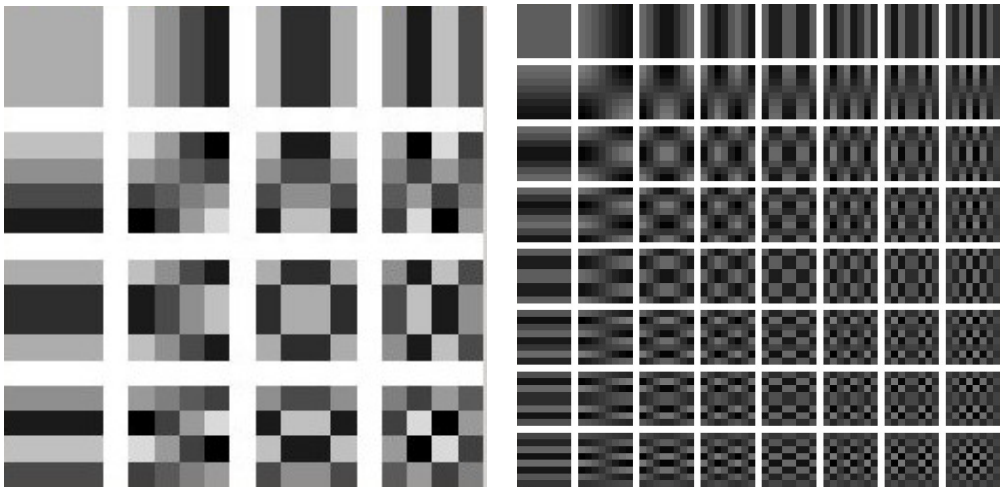

Figure 6.3: Basis functions for a DCT transform. a) 4x4 transform b) 8x8 transform.

After the cosine transform is applied to an image, there may be many elements that are zero or very close to zero. In figure 6.4 one can see an example of a small block that is 8x8 pixels. Even after rounding the coefficients up to the next integer, the inverse cosine transform (equation 6.4) recovers the image without any error. More interestingly, only 8 of the coefficients are different than zero and therefore the image could be represented using only these 8. The compression ratio in this example is good due to the strong redundancy seen in each row of the image. What would be the compression ratio for this block? We can write the coefficients in many different ways. Let us assume that each coefficient is represented by a `short int` (a signed short integer can store numbers up to -32768 to 32767). In this case, the compressed string would have 16 bits * 8 = 128 bits. The original block has 8 bits * 64 = 512 bits, so the compression ratio is 4.

In figure 6.5, most of the coefficients are different than zero. Note the trend that the largest coefficient is at the top left corner, and it tends to decrease as we go to the right or to the bottom of the block. This is one reason why the so called zigzag order is used, to produce the longest runs of zeros.

Next there is a code example for an oversimplified DCT compression and decompression. For clarity, the code uses arrays rather than images. Note that the image (IMAGE array) is the

Image 8x8							
1	127	127	120	120	123	124	23
1	127	127	120	120	123	124	23
1	127	127	120	120	123	124	23
1	127	127	120	120	123	124	23
1	127	127	120	120	123	124	23
1	127	127	120	120	123	124	23
1	127	127	120	120	123	124	23
1	127	127	120	120	123	124	23

DCT result							
765	−24	−281	−32	−237	−20	−118	−3
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 6.4: DCT example.

same as in figure 6.4. One can experiment by eliminating some coefficients to see the effects on the recovered image.

Listing 6.1: DCT code example

```
#include "stdio.h"
#include "math.h"
#define PI 3.14159265358979323846264338327950288419716939937
#define N 8

main(){
  float C[N][N]= {0,0,0,...   //coefficients
  float I[N][N]= {0,0,0,...   //image after recovery
  int R[N][N]= {0,0,0...   //quantized coefficients
  int DECOMPIMAGE[N][N]={0,0,0,...    //decompressed image
  int IMAGE[N][N]=    //image
  {1,127,127,120,120,123,124,23,
  1,127,127,120,120,123,124,23,
  1,127,127,120,120,123,124,23,
  1,127,127,120,120,123,124,23,
  1,127,127,120,120,123,124,23,
  1,127,127,120,120,123,124,23,
  1,127,127,120,120,123,124,23,
  1,127,127,120,120,123,124,23};
  float alphaU0=sqrt(1.0/N);
  float alphaU=sqrt(2.0/N);
  float alphaa=0,alphab=0;
  for (int a=0;a<N;a++){
    for (int b=0;b<N;b++){
      if (a==0) alphaa=alphaU0;
      else alphaa=alphaU;
```

Image 8x8

58	45	29	27	24	19	17	20
62	52	42	41	38	30	22	18
48	47	49	44	40	36	31	25
59	78	49	32	28	31	31	31
98	138	116	78	39	24	25	27
115	160	143	97	48	27	24	21
99	137	127	84	42	25	24	20
74	95	82	67	40	25	25	19

DCT result

421	203	11	45	−30	−14	−14	−7
−108	−93	10	49	27	6	8	2
−42	−20	−6	16	17	9	3	3
56	69	7	−25	−10	−5	−2	−2
−33	−21	17	8	3	−4	−5	−3
−16	−14	8	2	−4	−2	1	1
0	−5	−6	−1	2	3	1	1
8	5	−6	−9	0	3	3	2

Figure 6.5: DCT example from [2].

```
27        if (b==0) alphab=alphaU0;
28        else alphab=alphaU;
29        for(int x=0;x<N;x++){
30          for(int y=0;y<N;y++){
31            C[a][b]=(alphaa*alphab*IMAGE[x][y]*cos(((2*x+1)*a*PI)
                  /(2*N))*cos(((2*y+1)*b*PI)/(2*N))) +C[a][b];
32          }
33        }
34    //rounding up values for R(u,v)
35        R[a][b]=(int)round(C[a][b]);
36      }
37    }
38    for (int a=0;a<N;a++){
39      for (int b=0;b<N;b++){
40        printf("%1.5f    ",C[a][b]);
41        if (((b+1)%N)==0) printf("\n");
42      }
43    }
44    //recovering the image from the coefficients
45    for (int x=0;x<N;x++){
46      for (int y=0;y<N;y++){
47        for(int a=0;a<N;a++){
48          for(int b=0;b<N;b++){
49          if (a==0) alphaa=alphaU0;
50          else alphaa=alphaU;
51          if (b==0) alphab=alphaU0;
52          else alphab=alphaU;
53            I[x][y]=(alphaa*alphab*R[a][b]*cos(((2*x+1)*a*PI)/(2*N))
                  *cos(((2*y+1)*b*PI)/(2*N))) +I[x][y];
54          }
55        }
```

```
56          DECOMPIMAGE[x][y]=(int) round(I[x][y]);
57        }
58      }
59
60      for (int a=0;a<N;a++){
61        for (int b=0;b<N;b++){
62          printf("%d   ",DECOMPIMAGE[a][b]);
63          if (((b+1)%N)==0) printf("\n");
64        }
65      }
66  }
```

6.3.3 JPEG

JPEG (Joint Photographic Experts Group) is a standard compression method widely used today. The lossy mode compression method loses information and adds noises to the compressed image. However, for most purposes related to storage and to presentation of images to users, such problems can be disregarded. However, for the lossy modes if the compression ratios are too high then the quality can drop dramatically. There are 4 modes for JPEG, two of them are lossless compression [3].

- Sequential DCT-based compression
- Progressive DCT-based compression
- Sequential lossless predictive compression
- Hierarchical lossy or lossless compression

Here we will present only the first one.

Sequential DCT-based compression

This mode consists of:

- forward DCT transform
- quantiser
- entropy encoder

The original image is shifted to an interval where the origin is in the middle of the range. For example, if using 256 values the image would be shifted to [-128,127] interval. The image is then divided in 8 by 8 pixel sub-windows and each of these sub-windows is transformed into the frequency domain using the DCT transform and obtaining 64 DCT coefficients. These coefficients are quantised. The last step can use simple Huffman coding or other method to achieve greater compression ratios.

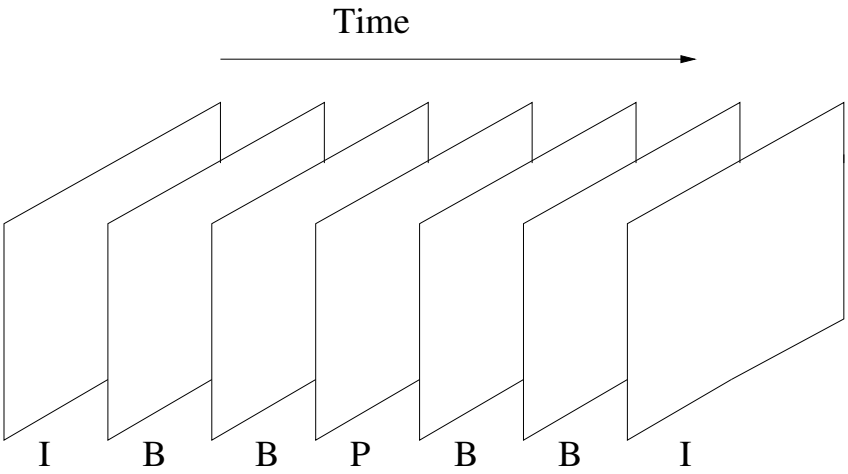

Figure 6.6: MPEG frame types.

6.4 Video compression

Video compression uses an additional cue, which is inter-frame similarities. Let us imagine a video sequence where there is a character moving but the camera is static. If the background is also static, most of the pixels that are related to the background are redundant. It would suffice to send the background once.

The MPEG (Motion Picture Expert Group) standards are commonly used. They are MPEG-1, MPEG-2, MPEG-4. There are other MPEG being discussed, for example MPEG-7 and MPEG-21.

In MPEG compression one can define three frame types: intraframes I, predicted frames P and interpolated frames B (B stands for bi-predictive). I frames are independent from the others. These frames are normally actual images compressed using either JPEG or a similar method. P frames are built based on predictive code in relation to a previous I frame. Finally, B frames are compressed with the highest compression ratio and they use interpolation between either two I frames or between an I and a P frames. Figure 6.6 shows a hypothetical example. The frames may have to be sent in a different order, so decompressing works more efficiently. Suppose that figure 6.6 is used. The two I frames should be sent first, then the P and B frames. The P and B frames can only be recovered if the two I frames are available.

The additional compression ratio for B frames is obtained using for example *block matching* algorithms. In fact the method is not specified in the standards, so any *motion estimation* technique would be valid. More on motion estimation will be explored in chapter 7.

6.5 Reading

Read chapter 8 of Gonzales & Woods.

6.6 Exercises

1. Create the Huffman tree for the following set of symbols:

Symbol	Probability
A	0.1
B	0.08
C	0.02
D	0.35
E	0.25
F	0.05
G	0.04
H	0.01
I	0.099
J	0.001

   After building the tree, write the Huffman coding for the symbols.

   What is the compression ratio?

2. Write a simple version of Huffman code that is able to compress a grey-scale image in OpenCV. How does your compression rate fares against jpg? (to check that, you can export a jpg image in GIMP, after transforming it into grey-scale.)

3. Using the DCT transform above, write a program to compute the cosine transform coefficients for an 8x8 block. Compress it using Huffman code.

4. Using the DCT transform above, write a program to compute the cosine transform coefficients for an 16x16 block. Compress it using Huffman code. Any difference in the compression rate?

5. Modify the program to compute the inverse cosine transform. Analyse the errors in the resulting images by simple rounding (to integers) and quantisation of the coefficients compared to the use of floating point numbers to represent the coefficients.

6. Manually use the LZW algorithms to compress the following sequence: 25-72-25-72-25-72-25-72-25-72-25-72-25-72-25-72. Can you decompress it, even though some of the dictionary entries are not computed before translating N for some cases?

Bibliography

[1] R. C. Gonzalez and R. E. Woods, *Digital Image Processing.* Prentice Hall, 2002.

[2] J. Miano, *Compressed Image File Formats.* Addison-Wesley, 1999.

[3] M. Sonka, V. Hlavac, and R. Boyle, *Image Processing, Analysis and Machine Vision.* PWS Publishing, 1999.