Answers for the exercises in Chapter 6

Andre L. C. Barczak

2015

1 Exercise 1

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.)

Answer: Firstly we need to compute Huffman's codes for a certain number of entries. For example, we can create a HuffmanCodes[] array for 256 entries. For 6 entries, the codes would look like this:

1. 1
2. 00
3. 011
4. 0100
5. 01011
6. 010100

Although it is possible to avoid the last bit on the last entry (it could be 01010 instead), we simplify the code to allow for all of the 256 entries.

Listing 1: Huffman Codes computation

```
void print_Huffman_Codes(){
  unsigned int bits[N][N];//level, bit
  int b=0;
  int level=0;
  bits[0][0]=1;
  int i=0; int j=0;
  for(i=1;i<N;i++){
    level++;
    b=0;
    for(j=0;j<=level;j++){
      bits[i][j]=b;
      if(b==1)b=0;
      else b=1;
    }
```

```
      if(bits[i][j-1]==1)bits[i][j-1]=0;
      else bits[i][j-1]=1;
    }
    level=-1;
    for(int i=0;i<N;i++){
      level++;
      stringstream tempss;
      for(int j=0;j<=level;j++){
        tempss << bits[i][j];
      }
      HuffmanCodes[i]=tempss.str();
    }
}
```

After computing the Huffman codes, we need a histogram of the file for a certain number of bits, say 8. In this case we have at most 256 symbols to encode, using the codes above.

Listing 2: Structures for Compression and Decompression

```
#define N 256 //1 byte
unsigned long int accumulator[N];//the histogram
unsigned long int indexsorting[N];//the order of the probabilities by index
string HuffmanCodes[N];//This is the dictionary, it needs to be saved in the
    compressed file, so we know how to decode it
stringstream compressedbits;
int number_of_entries=0;//this is the number of entries with accumulator != 0
```

The following compress function is listed below:

Listing 3: Huffman Compression

```
void Compress(char * filename, char * filenameOut){
  ifstream inputfile;
  inputfile.open(filename,ifstream::binary|ios::in);     // open file
  if(!inputfile.good()){printf("Can't open the file \n");exit(0);}
  char c;
  unsigned char unsignec;
  int numberofbytes=0;
  while(inputfile.read((char *)(&c), sizeof(c))){
    unsignec=(unsigned char)c;
    //use the unsigned char to Encode (choose a Huffman's code) based on the
        histogram
    if(!inputfile.eof()){
      compressedbits << Search_Huffmans_Code(unsignec);
      numberofbytes++;
    }
  }
  string printable_compressedbits=compressedbits.str();
```

```
  cout << "Number of bits: " << printable_compressedbits.size() << " Number of bytes: " << printable_compressedbits.size()/8.0 << endl;
  cout << "Compression ratio: " << numberofbytes/(printable_compressedbits.size()/8.0) << endl;
  inputfile.close();
  //now save the bits into an output file, with the extra information about the table and number of bits
   ofstream outputfile;
   outputfile.open(filenameOut, ofstream::binary|ios::out);
   if(!outputfile.good()){printf("Can't open the output file \n");exit(0);}
   unsigned char putbyte=0;
   putbyte=(unsigned char)number_of_entries-1;
   printf("Number of entries %d \n", number_of_entries);//write byte with number of entries
   outputfile.write((char *)&putbyte,sizeof(putbyte));
   //write entries (indexsorting)
   for(int a=0;a<N;a++){
     if(accumulator[a]!=0) {
       putbyte=(unsigned char)indexsorting[a];
       outputfile.write((char *)&putbyte,sizeof(putbyte));
     }
     else break;
   }
   //write the number of bits (using 4 bytes for that)
   unsigned int number_of_bits=printable_compressedbits.size();
   outputfile.write((char *)&number_of_bits,sizeof(number_of_bits));
   //Finally, write the bits from compressedbits
   string sputdata = compressedbits.str();
   int counter=7;
   putbyte=0;
   for(int i=0;i<sputdata.size();i++){
     unsigned char mask=1;
     mask=mask<<counter;
     counter--;
     if(sputdata[i]=='1') putbyte = putbyte | mask;
     if(counter==-1) {
       outputfile.write((char *)&putbyte,sizeof(putbyte));
       counter=7;
       putbyte=0;
     }
   }
   //last byte, possibly with fewer than 8 bits of data
   if(counter!=7) outputfile.write((char *)&putbyte,sizeof(putbyte));
   outputfile.close();
}
```

And the decompression is:

Listing 4: Huffman Decompression

```
void Decompress(char * filename, char * filenameOut){
 ifstream inputfile;
 inputfile.open(filename,ifstream::binary|ios::in);      // open file
 if(!inputfile.good()){printf("Can't open the file \n");exit(0);}
 char c;
 unsigned char unsignec;
 //get number of entries in the table
 inputfile.read((char *)(&c), sizeof(c));
 unsignec= (unsigned char)c;
 unsigned int number_of_entries=unsignec+1;
 printf("Number of entries %d \n",number_of_entries);
 //write the table
 for (int a=0;a<number_of_entries;a++){
   inputfile.read((char *)(&c), sizeof(c));
   unsignec=(unsigned char)c;
   indexsorting[a]=unsignec;
 }
 //read the number of bits in the file
 int number_of_bits=0;
 inputfile.read((char *)(&number_of_bits), sizeof(number_of_bits));
 printf("Number of bits %d \n", number_of_bits);
 int bitcounter=0;//control the number of bits
 compressedbits.str("");//clean up stringstream to receive data
 while(inputfile.read((char *)(&c), sizeof(c))){
   unsignec=(unsigned char)c;
   for(int a=7;a>=0;a--){
     unsigned char mask=1;
     mask = mask << a;
     if( (c & mask)!=0) compressedbits << "1";
     else compressedbits << "0";
     bitcounter++;
     if(bitcounter>number_of_bits) break;
   }
 }
 //search for HuffmanCodes until it matches a character
 string decode = compressedbits.str();
 string findchar="";
 stringstream temps;
 stringstream decodedss;
 string decodedstring;
 int a=0;
 cout << endl << "Starting decompression " << endl;
 for(int i=0;i<decode.size();i++){
   temps << decode[i];
   findchar=temps.str();
```

```
    for(a=0;a<N;a++){
      if(findchar.compare(HuffmanCodes[a])==0){
        findchar="";
        temps.str("");
        break;
      }
    }
    if(a < number_of_entries){//if a>=number_of_entries, didn't find a match
      decodedss<<(unsigned char)indexsorting[a];
    }
  }
  decodedstring=decodedss.str();
  inputfile.close();
  //now right the output file
  ofstream outputfile;
  outputfile.open(filenameOut, ofstream::binary|ios::out);
  if(!outputfile.good()){printf("Can't open the output file \n");exit(0);}
  unsigned char putbyte=0;
  for(int i=0;i<decodedstring.size();i++){
    putbyte=decodedstring[i];
    outputfile.write((char *)&putbyte,sizeof(putbyte));
  }
  outputfile.close();
}
```

For example, a file containing:

AAAAAAAAAABBBBBBBBBBAAAAAAAAAABBBBBCCCCCDDDDDEEEFF

could be compressed with the following bit sequence:

1 1 1 1 1 1 1 1 1 1 00 00 00 00 00 00 00 00 00 00 1 1 1 1 1 1 1 1 1 1 00 00 00 00 00

0100 0100 0100 0100 0100 011 011 011 011 011 01011 01011 01011 010100 010100

Note that this sequence is unambiguous, starting from bit 1 the only possible match for '1' is A. Later when finding the first '0', there could be many codes, but finding the next '0' unambiguously matches 'B' and so on.

2 Exercise 2

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

Answer:

Listing 5: DCT 8x8

```
//A very simple cosine transform for blocks of images
//This code is O(N^2), there are more efficient ways to compute DCTs
//OpenCV has faster DCT transform implementations
#include <cstdlib>
#include <iostream>
#include <sstream>
```

```
#include <fstream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
using namespace std;

#define PI 3.14159265358979323846264338327950288419716939937510582097494459230
#define N 8
main(){
  //coefficients
  float C[N][N]= {0,0,0...
  //image recovery
  float I[N][N]= {0,0,0...
  //image, repeated values are represented by fewer coefficients
  int IMAGE[N][N]=
  {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};
  //quantized coefficients
  int R[N][N]= {0,0,0...
  //decompressed image
  int DECOMPIMAGE[N][N]= {0,0,0...
  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;
      if (b==0) alphab=alphaU0;
      else alphab=alphaU;
      for(int x=0;x<N;x++){
        for(int y=0;y<N;y++){
          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];
        }
      }
      //rounding up values for R(u,v)
      R[a][b]=(int)round(C[a][b]);
    }
  }
```

```
    for (int a=0;a<N;a++){
        for (int b=0;b<N;b++){
            printf("%1.5f        ",C[a][b]);
            if (((b+1)%N)==0) printf("\n");
        }
    }
    //get only a few coefficients //recover image from coefficients
    for (int x=0;x<N;x++){
        for (int y=0;y<N;y++){
            for(int a=0;a<N;a++){
                for(int b=0;b<N;b++){
                    if (a==0) alphaa=alphaU0;
                    else alphaa=alphaU;
                    if (b==0) alphab=alphaU0;
                    else alphab=alphaU;
                    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];
                }
            }
            DECOMPIMAGE[x][y]=(int) round(I[x][y]);
        }
    }

    for (int a=0;a<N;a++){
        for (int b=0;b<N;b++){
            printf("%d  ",DECOMPIMAGE[a][b]);
            if (((b+1)%N)==0) printf("\n");
        }
    }
    //save to a file as binary (4 bytes per coefficient)
    ofstream outputfile;
    outputfile.open("Cosine8x8.bin", ofstream::binary|ios::out);
    if(!outputfile.good()){printf("Can't open the output file \n");return 0;}
    for (int a=0;a<N;a++){
        for (int b=0;b<N;b++){
            if( C[a][b] > 0.001 || C[a][b]< -0.001){
                outputfile.write((char *)&C[a][b],sizeof(C[a][b]));
                printf("%1.5f        ",C[a][b]);
            }
        }
    }
    outputfile.close();
}
```

In the 8x8 block example there are redundancies by repeating columns. This will generate the following coefficients:

764.99994	-23.84446	-281.67648	-32.24512	-237.00000	-20.34282	-118.20490	-3.72337
0.00004	-0.00000	-0.00002	0.00000	-0.00000	-0.00000	0.00000	-0.00000
-0.00001	-0.00000	0.00000	-0.00000	0.00000	0.00000	0.00000	-0.00000
0.00000	0.00000	0.00001	-0.00000	-0.00001	0.00000	-0.00001	0.00000
-0.00000	-0.00000	0.00000	-0.00000	-0.00000	0.00000	0.00000	-0.00000
0.00001	0.00000	0.00000	0.00000	0.00000	-0.00000	-0.00000	0.00000
0.00000	0.00000	-0.00000	0.00000	-0.00000	-0.00000	0.00000	-0.00000
-0.00001	0.00000	0.00000	-0.00000	-0.00000	0.00000	0.00000	0.00000

From which we can deduce that only 8 are important (larger than $|0.001|$). Therefore, saving the coefficients using floats (4 bytes) only takes 32 bytes, a compression ratio of 2. However, trying to use Huffman's code directly on this binary file does not compress it, rather it expands it. To gain more compression we would have to encode the coefficients (in a way to have more redundancies) before using Huffman.

3 Exercise 3

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?

Answer: The same code above, with $N = 16$ and the appropriate changes in the image. Lets suppose that we want 4 repeated blocks of 8x8 as in question 2. If we used the 8x8 DCT, the compression would have been:

$$C_r = \frac{64\,bytes * 4}{32\,bytes * 4} = 2$$

Using 16x16 blocks however, yields the following 12 coefficients smaller than $|0.1|$:

1530.00000 -22.76264 -26.64937 -563.35254 -24.54036 -49.10177
-474.00000 2.34227 -47.52755 -236.40994 25.22849 -31.64850

So for the same 16x16 image, the compression ratio would be:

$$C_r = \frac{64\,bytes * 4}{12\,bytes} = 21.33$$

A massive gain in performance. It is important to understand that this is achieved because the 4 blocks of 8x8 are identical, so the redundancies are large. If comparing 4 different blocks of 8x8, the gain would not have been the same.

4 Exercise 4

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.

Answer: The code in Exercise 2 already does the inverse cosine transform. We can modify the code to quantise the coefficients and try to recover the pixels, and see the results. Using the same 8x8 block used above, suppose that we round up the coefficients to integers:

765 -24 -282 -32 -237 -20 -118 -4

What do we recover? The same original 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

Lets dismiss the smaller components such as -4 and -20. Now we would recover:

3	123	128	122	118	122	128	21
3	123	128	122	118	122	128	21
3	123	128	122	118	122	128	21
3	123	128	122	118	122	128	21
3	123	128	122	118	122	128	21
3	123	128	122	118	122	128	21
3	123	128	122	118	122	128	21

This is close enough, but not the same. This is an example of a lossy compression, as it would only take $4 * 6$ bytes to store the same block of 8x8 pixels.