Multimedia Technologies

Project Report

1. Members

序号	学号	专业班级	姓名	性别	分工

2. Project Introduction

2-1. Topic Selection

I'm not so confident about my programming ability. Besides, it might be hard for me to finish a difficult task individually. So I decided to proceed the images. It comes to my mind that image compression may be a good idea. That's why I chose 24-bit BMP image compression as my topic.

2-2. Work Description

To accomplish the compression of images. I chose to done it in both lossy and lossless ways.

1. Lossy Compression:

   In this aspect, my work is much easier. To compress the image data. I wanted to convert every 24-bit RGB data into 8-bit. And because human eyes are less sensitive to blue colors, so I reserved 3 bit for R, 3 bit for G, and 2 bit for B. Then I've done a simple uniform quantification.

2. Lossless Compression:

   To do a lossless compression. The basic way is Huffman Coding. First sort the 8-bit image data by frequency. Then combine the two with least frequency recursively until there is only one tree. Then convert their data into 01 string. By this way, we can convert every byte into different lengths of strings.

2-3. Development Environment

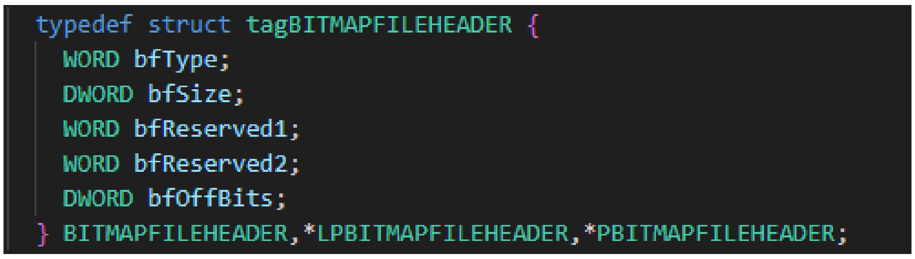
Microsoft Visual Studio.

Windows 11.

3. Technical Details

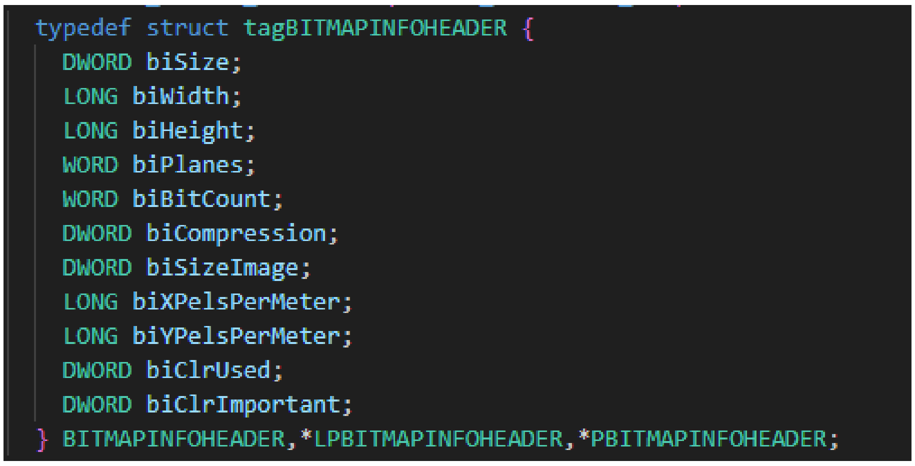
3-1. Theoretical Basis

(1) BMP Headers:



```
typedef struct tagBITMAPFILEHEADER {
    WORD bfType;
    DWORD bfSize;
    WORD bfReserved1;
    WORD bfReserved2;
    DWORD bfOffBits;
} BITMAPFILEHEADER,*LPBITMAPFILEHEADER,*PBITMAPFILEHEADER;
```

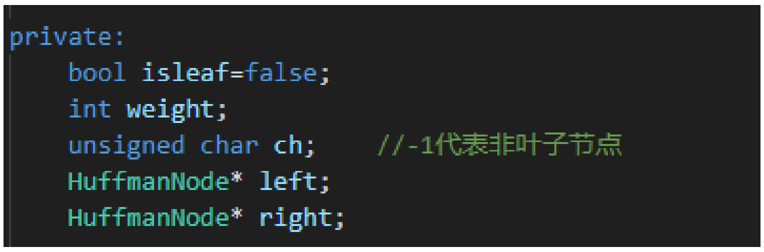
This is the BMP File Header defined in the standard library of C++. **bfType** shows the image type. When reading the image, we should first make sure that it equals **0x4D42**, otherwise it's invalid.



```
typedef struct tagBITMAPINFOHEADER {
    DWORD biSize;
    LONG biWidth;
    LONG biHeight;
    WORD biPlanes;
    WORD biBitCount;
    DWORD biCompression;
    DWORD biSizeImage;
    LONG biXPelsPerMeter;
    LONG biYPelsPerMeter;
    DWORD biClrUsed;
    DWORD biClrImportant;
} BITMAPINFOHEADER,*LPBITMAPINFOHEADER,*PBITMAPINFOHEADER;
```

This is the BMP Information Header. It shows the size, width, height, bitcount etc. information. **biSizeImage** shows the number of pixels, which we will use to proceed the data later.

(2) Huffman Code



```
private:
    bool isleaf=false;
    int weight;
    unsigned char ch;    //-1代表非叶子节点
    HuffmanNode* left;
    HuffmanNode* right;
```

To generate the huffman code, we've got to define the huffman tree. Above is the class variables that I've defined for a huffman tree node. As we've known, with all the merging operations, the node that really stands for a specific value from 0 to 255 is on the leaves. **isleaf** is the variable that indicates whether a node is a leaf. **weight** is the total weight of the subtree rooted at that node. **ch** is the value of a leafnode, from 0 to 255.

We traverse the whole tree, add 0 to the string when going left, add 1 on the other side. Then we can get the huffman code for each color value.

$$H(X) = -\sum_{x} P(x) \log_2 P(x)$$

Though huffman code do compress the original data. It still has some distance to the limit of the entropy. That's because huffman code is a kind of integer coding, which is not precise enough. However, it's undoubtedly a good way for lossless compression.

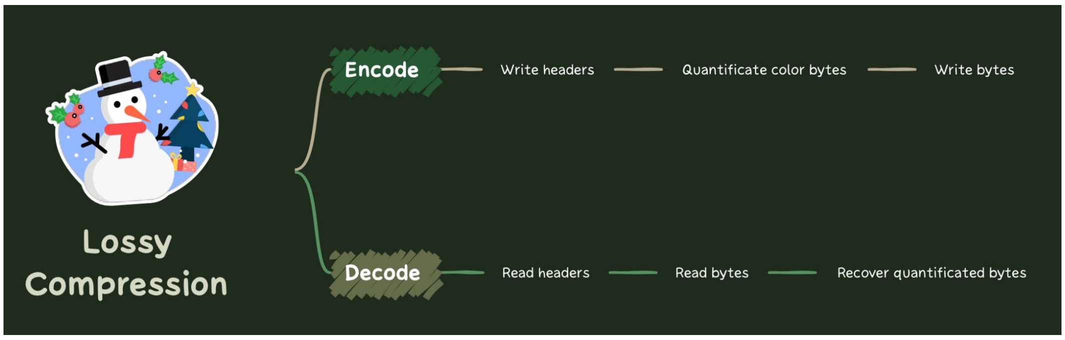
(3) Compression:

For lossless compression, we convert each byte of the image data into the huffman code, then store it in forms of bytes.

For lossy compression, we quantificates every 3 bytes into 1 byte. Recover it when decoding.

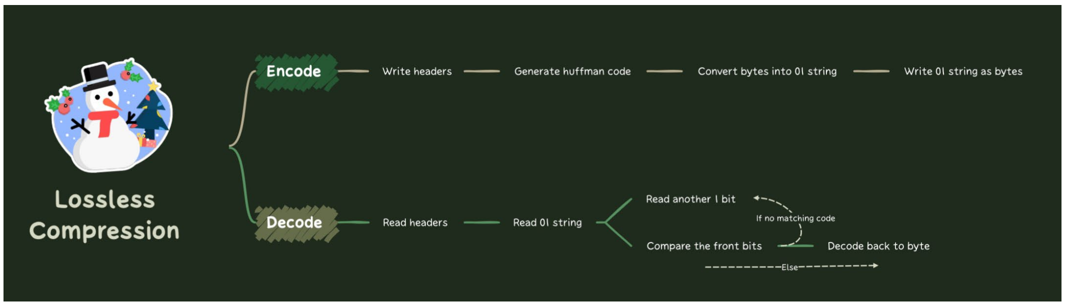
3-2. Algorithms

(1) Lossy Compression:



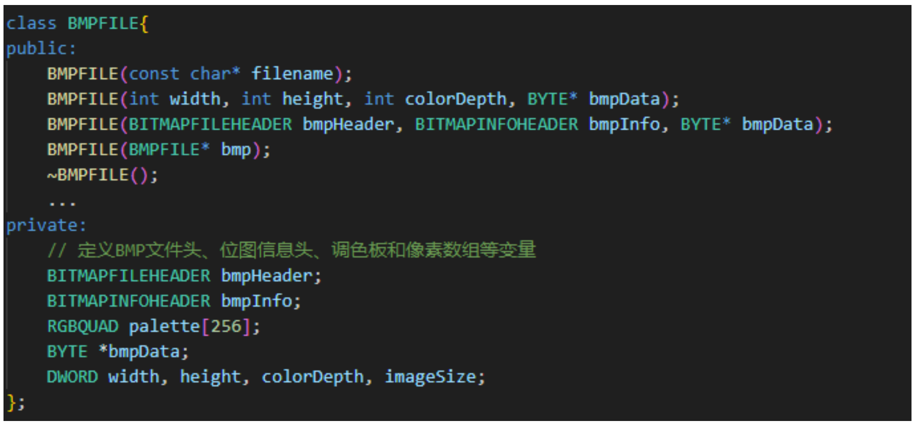
When encoding, we first write the headers into a file, then quantificate the color information into a single byte, and write it into the file. When decoding, we first read the headers to get the image size, then read the remaining bytes, and convert it back to a 3-byte color information.

(2) Lossless Compression:



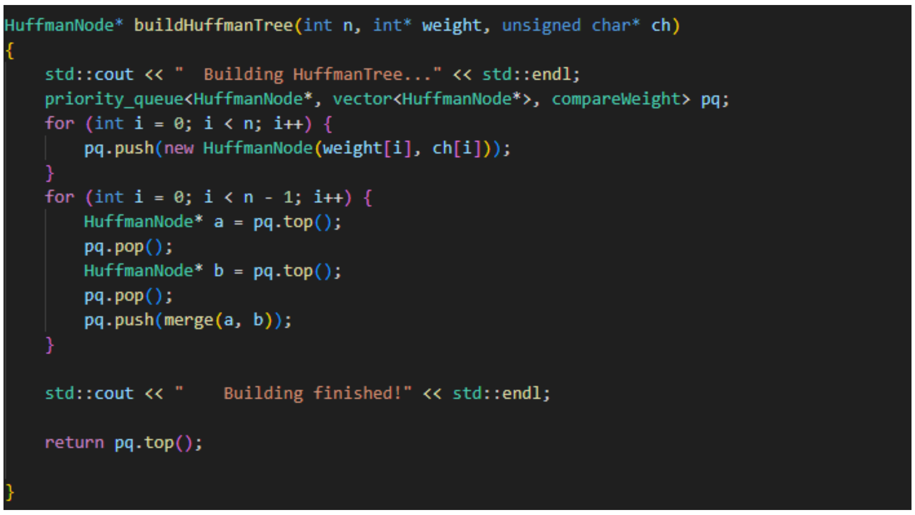
When encoding, we first write the headers, then generate the huffman code. After that, we convert the bytes into huffman code continuously, the write it into the file as bytes. (We can't write the 01 string directly because it's much bigger!) When decoding, we read the headers to obtain crutial information, then read the bytes 1-bit by 1-bit until we find the first several bits matches a huffman code. In this way we can get the original data with no errors.

3-3. Technical Details



```
class BMPFILE{
public:
    BMPFILE(const char* filename);
    BMPFILE(int width, int height, int colorDepth, BYTE* bmpData);
    BMPFILE(BITMAPFILEHEADER bmpHeader, BITMAPINFOHEADER bmpInfo, BYTE* bmpData);
    BMPFILE(BMPFILE* bmp);
    ~BMPFILE();
    ...
private:
    // 定义BMP文件头、位图信息头、调色板和像素数组等变量
    BITMAPFILEHEADER bmpHeader;
    BITMAPINFOHEADER bmpInfo;
    RGBQUAD palette[256];
    BYTE *bmpData;
    DWORD width, height, colorDepth, imageSize;
};
```

This is the BMP image class I defined in the project. I write 4 construction functions. The first is to read from a existing file. The second is to construct a bmp image by some basic information. The third is to construct by given headers and data. The fourth is a copy construction function.



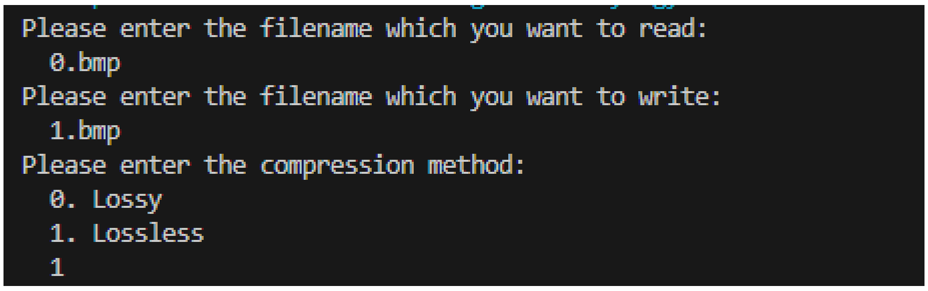
```
HuffmanNode* buildHuffmanTree(int n, int* weight, unsigned char* ch)
{
    std::cout << "  Building HuffmanTree..." << std::endl;
    priority_queue<HuffmanNode*, vector<HuffmanNode*>, compareWeight> pq;
    for (int i = 0; i < n; i++) {
        pq.push(new HuffmanNode(weight[i], ch[i]));
    }
    for (int i = 0; i < n - 1; i++) {
        HuffmanNode* a = pq.top();
        pq.pop();
        HuffmanNode* b = pq.top();
        pq.pop();
        pq.push(merge(a, b));
    }

    std::cout << "    Building finished!" << std::endl;

    return pq.top();

}
```

To build a huffman tree, I used a priority_queue because it's self-sorted. Every time I can only take out 2 trees at the top, and merge them.

In order to keep the report tidy, more details will not be shown here, you can check the source files for more information.

4. Experiment Results

(1) System Interface:



```
Please enter the filename which you want to read:
  0.bmp
Please enter the filename which you want to write:
  1.bmp
Please enter the compression method:
  0. Lossy
  1. Lossless
  1
```

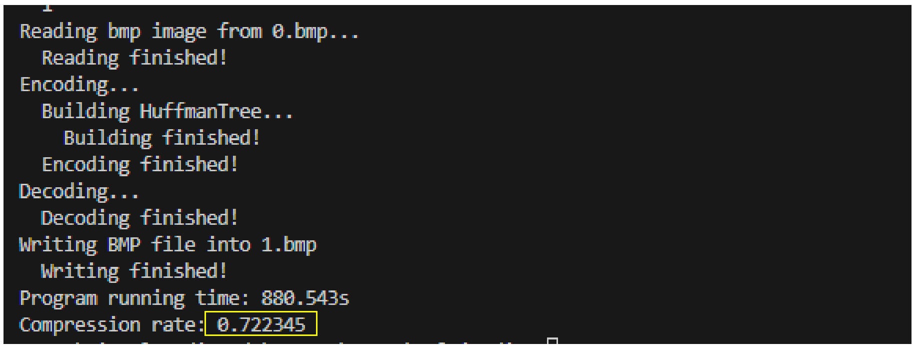
When starting the program, you will need to input the **source file name**, the **target file name**, and the **compression method**. Then it will starting compressing.

(2) Operating Instructions:

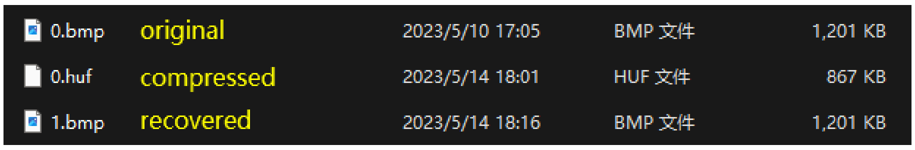
i. put a 24-bit ***bmp image*** in the directory of source files.
ii. ***compile*** and ***run*** main.cpp
iii. input the ***source filename***, ***target filename*** and ***compression method***.
iv. ***wait*** for the results. The compressed file will be stored as ****.huf*** or ****.los*** (depends on which method you've chosen.)

(3) Running Results:

1) Lossless Compression:



```
Reading bmp image from 0.bmp...
  Reading finished!
Encoding...
  Building HuffmanTree...
    Building finished!
  Encoding finished!
Decoding...
  Decoding finished!
Writing BMP file into 1.bmp
  Writing finished!
Program running time: 880.543s
Compression rate: 0.722345
```

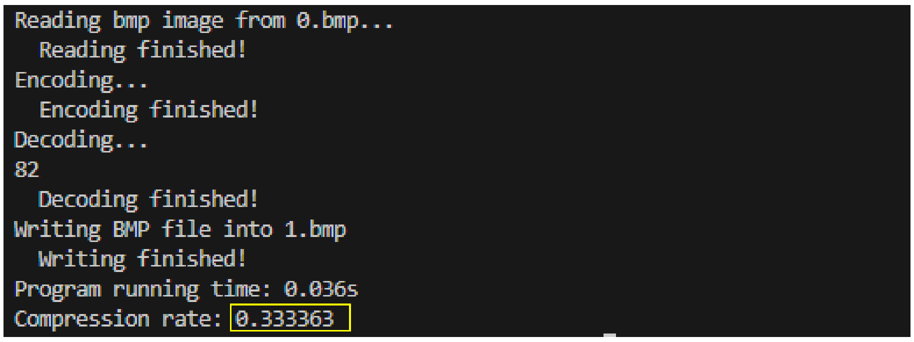


0.bmp	original	2023/5/10 17:05	BMP 文件	1,201 KB
0.huf	compressed	2023/5/14 18:01	HUF 文件	867 KB
1.bmp	recovered	2023/5/14 18:16	BMP 文件	1,201 KB

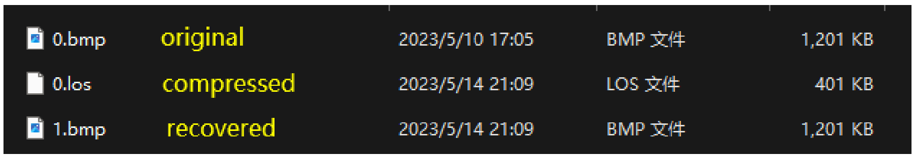
original

compressed

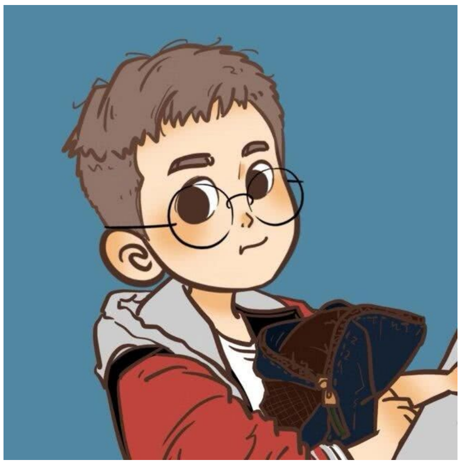
2) Lossy Compression:

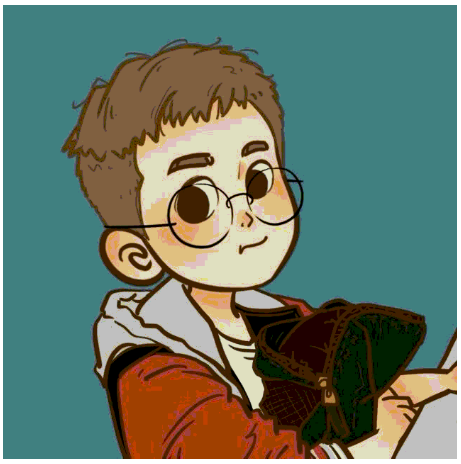


```
Reading bmp image from 0.bmp...
  Reading finished!
Encoding...
  Encoding finished!
Decoding...
82
  Decoding finished!
Writing BMP file into 1.bmp
  Writing finished!
Program running time: 0.036s
Compression rate: 0.333363
```



0.bmp	original	2023/5/10 17:05	BMP 文件	1,201 KB
0.los	compressed	2023/5/14 21:09	LOS 文件	401 KB
1.bmp	recovered	2023/5/14 21:09	BMP 文件	1,201 KB


original


compressed

(4) Summary:

It’s really a pity that I didn’t learn python. It will be much more convenient if I use the functions given by python. There are many functions to deal with images in python, such as matrix transformation. But in C++, I have to deal with the original data bytes, which means I have to know clearly how image data are stored. Anyway with debugging for many times, I eventually made it.

To accomplish such a ‘easy’ function. I wrote a lot of extra functions to assure that every step gives a correct result. Such as functions to only reserve the R,G or B color portion, functions to print the Huffman tree, functions to output the corresponding code of the original byte, etc. Anyway, the whole process is just tricky. But thanks to god, the result didn’t fail.

Anyway, I have to tell you that, my program might be kind of unpleasant because it takes too long to finish the decoding of lossless compression. For example, it takes me nearly 20 mins to decode a 640*640 24-bit bmp image. So if you wants to run this program, you may have to prepare a smaller image, or just be mentally prepared for the long progress.

Seeing the result. It has a long way to go. The compression rate is so ordinary. And for the lossy compression, the recovered image is so lossy. If I’ve done the differential before, and do some DCT transformation after, the result may seem more fantastic. However, if I was given more time, I could have made it better.

Thanks for Mr.Xiao. I’ve learnt so many new things in this course. And it’s actually my first time to finish such a project individually, which makes me more efficient in debugging and version managing. And I’ve uploaded the project onto github. This is my website: FelixChristian011226/Multimedia (github.com).

Thank you!

References:

Fundamentals of Multimedia,Ze-Nian Li, Mark S. Drew.

位图(bmp)文件格式分析: https://blog.csdn.net/aidem_brown/article/details/80500637.