COMP7503 Multimedia Technologies Programming Assignment

Hanke Wang
hanke@hku.hk

Jiaxing Zeng
everstar@hku.hk

Contents

Quantize565 & Dequantize565

Quantize565

Dequantize565

Compress & Decompress

Brief Introduction

In the compression & decompression part, we use **Huffman Coding** as our compression algorithm.

- There are 4 steps to compress the image:
    1. A *Huffman Tree* will be built according to the **appearance frequency** of pixels bits.
    2. The *Huffman Tree* will be converted to a **Huffman Coding Mapping Table**.
    3. All the pixels of the input image will be encoded on the basis of the **mapping table**.
    4. The mapping table and the encoded string will be written to variable **compressedData**.
- There are 4 steps to decompress the image:
    1. Recover the **Huffman Coding Mapping Table** from the encoded string (variable **compressedData**).
    2. Rebuild the *Huffman Tree* from the **mapping table**.
    3. Scan the encoded string bit by bit to recover the pixels according the *Huffman Tree*.
    4. Recovered pixels will be written to variable **uncompressedData** .

Compress

1. Variable Initialization

In the beginning, we create a new `unsigned char[]` with size 1024. Because we won't know the compressed size until we encoded everything, so we simply create a suitable size and then enlarge it when we need.

```
// Initialize variable compressedData
int size_compressed_data = 1024;
unsigned char *compressedData = new unsigned char[size_compressed_data];
memset(compressedData, 0, sizeof(unsigned char) * size_compressed_data);
```

Each time we write data to **compressedData**, we will check whether the size is enough. Otherwise we will create a new `unsigned char*` with **110%** size of the previous size and copy the previous data to it.

The reason why we choose **110%** is that if we use **200%** instead, the size may be too large at the end which may left too much leisure by waste memory.

```
/* Automatically adjust size of unsigned char* */
void writeData(unsigned char* &target, int& size, int pos, unsigned char data) {
        if (pos + 1 > size) {
                int adjustedSize = int(size * 1.1);
                unsigned char* base = new unsigned char[adjustedSize];
                memcpy(base, target, size);
                delete[] target;
                size = adjustedSize;
                target = base;
        }
        target[pos] = data;
}
```

Next, we will construct a mapping table to store the *Huffman Tree*. We allocate `unsigned char[]` with size **256 * 32**.

```
/* Huffman Code Mapping Table */
unsigned char* huffmanTree = new unsigned char[256 * 32];
memset(huffmanTree, 0, sizeof(unsigned char) * 256 * 32);
```

We will encoding the 'R' or 'G' or 'B' of the pixel which is stored using 8 bit previously, which means it can have maximum 256 kinds of situation.

Considering the worst situation that the *Huffman Tree* is extremely unbalanced(Figure below). There will be **256** leaf nodes of *Huffman code* when 0 ~ 255 all appear in the pixels RGB bits. Consequently the depth of the tree will be maximum **255**, which means we have to use at least **255** bits to store the *Huffman Code*.

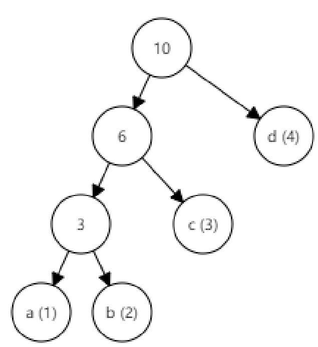

Figure 1: Unbalanced Huffman Tree

To manipulate it simply, we complement it to **256** bits so we decide to use `unsigned char[32]` to store each *Huffman Code*. In order to store 256 kinds of *Huffman Code*, we allocate a `unsigned char[256 * 32]`.

In the meantime, we will also store the length of each *Huffman Code*.

```
/* Encoding length of Huffman Code */
int *encoding_len = new int[256];
memset(encoding_len, 0, sizeof(int) * 256);
```

Considering the following situation:

```
32 : 01
48 : 001
```

The *Huffman Code* of 32 is "01" while 48 is "001". The value of them are equal but the number of bits are different, thus we need to record the length of each *Huffman Code*.

2. Huffman Tree Building

TODO: Introduce how to build the Huffman Tree.

3. Encoding Pixel

After Building the *Huffman Tree* and converting it to the *Mapping Table*, we can know the **maximum length** of all *Huffman Code*.

Thus, we can no longer use `unsigned char[32]` to store a *Huffman Code*. We will use the complemented maximum length of *Huffman Code* as the size.

```
/**
 * Calculate the length that complement binary bit
 * examples:
 *       5 ->  8
 *       9 -> 16
 *      17 -> 32
 */
int calcComplementLength(int len) {
        int n = 0;
        while (++n <= 32) {
                if (len < n * 8) return n * 8;
        }
        return 256;
}
```

So we can adjust the size of the *Huffman Code* to smallest when we encoding the pixels.

```
/* Suppose the maximum size of Huffman Code is 13
 * We will only need 16 bit to store it.
 * So we only need to allocate "unsigned char[2]" for each Huffman Code.
 */
int encoding_len_max = 13;
encoding_len_max = calcComplement(encoding_len_max); // 16
int byte_encoding_len_max = encoding_len_max / 8; // 2
unsigned char* encoded = unsigned char[byte_encoding_len_max];
```

Next, we will transform the pixel **RGB** to an **encoded sequence** according to the *Huffman Code Mapping Table*.

```
// Create Encoded Sequence
unsigned char *encodedSequence = new unsigned char[cDataSize * byte_encoding_max];
memset(encodedSequence, 0, sizeof(unsigned char) * cDataSize * byte_encoding_max);
// Length of each Encoded bits
unsigned int *encodedLength = new unsigned int[cDataSize * byte_encoding_max];
memset(encodedLength, 0, sizeof(unsigned int) * cDataSize * byte_encoding_max);
```

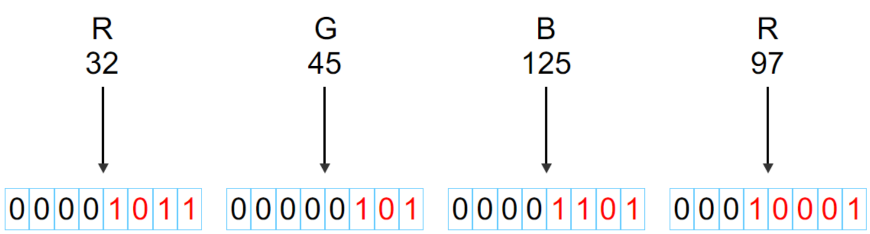

Figure 2: Sample Encoded Sequence

```
// pmy => primary, means one of the three-primary-color, like one of the (R, G, B)
for (int pmy = 0; pmy < cDataSize; ++pmy) {
    // Empty chars
    memset(encoded, 0, sizeof(unsigned char) * byte_encoding_max);
    // Get mapped Huffman Code
    for (int i = 0; i < 32; ++i) {
        huffmanCode[i] = huffmanTree[(pInput[pmy] * 32) + i] & 0xFF;
    }


    // Get mapped Huffman Code Length
    int len = encoding_len[pInput[pmy]];
    // Copy Huffman Code to Encoded Sequence for certain bits, from back to front
    for (int i = 31, j = byte_encoding_max; len > 0 && i >= 0 && j >= 0; --i) {
        unsigned int l = len > 8 ? 8 : len;
        len -= l;
```

```
        encoded[--j] = huffmanCode[i] & ((1 << l) - 1);
        // Save the encoded bits
        encodedSequence[pmy * byte_encoding_max + j] = encoded[j];
        // Save the length of encoded bits
        encodedLength[pmy * byte_encoding_max + j] = l;
    }
}
```

4. Write Data

In the beginning, we need to write the Huffman Code Mapping Table to **CompressedData**.

```
/* Package the Huffman Table into @var{compressedData} */

unsigned int tableSize = 0;
for (unsigned int i = 0; i < 256; ++i) {
    if (encoding_len[i] != 0) { // Ignore the pixel that not appear
        ++tableSize;             // calculate the table size
        // Put the key of Huffman Table
        writeData(compressedData, size_compressed_data, ++pos_data, (unsigned char)i);
        // Put the value of Huffman Table
        for (int j = 32 - byte_encoding_max; j < 32; ++j) {
            writeData(
                compressedData,
                size_compressed_data,
                ++pos_data,
                huffmanTree[i * 32 + j] & 0xFF

            );
        }
        // Put the encoding len of each Huffman code
        writeData(
            compressedData,
            size_compressed_data,
            ++pos_data,
            (unsigned char)encoding_len[i]
        );
    }
}
```

In addition, we will store the **size** of *Huffman Code Mapping Table* and the **maximum size** of *Huffman Code* to the first two byte of **compressedData**.

```
/**
 * Store the size of Huffman Table
```

```
 * &
 * bytes of max encoding len of Huffman Table
 * at the head @var{compressedData}
 */
compressedData[0] = (unsigned char)tableSize;
compressedData[1] = (unsigned char)byte_encoding_max;
```

Finally, we will write the encoded sequence to **compressedData**.

Due to the that we can only manipulate a char(8 bit) at a time, we need to split the bits to into groups which each consist of 8 bits.

To achieve that, we use an unsigned char as a **buffer**. Then, we write the encoded sequence to the **buffer** bit by bit. Each time the **buffer** is **fully filled**, we will write the buffer to the **compressedData**.

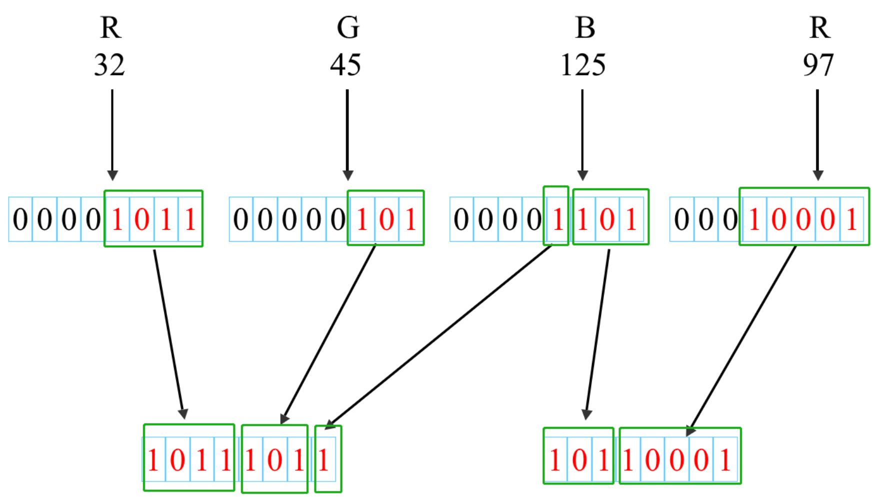

Figure 3: Writing Encoded Sequence to CompressedData

After all above operations completed, we modify the **cDataSize** and return the **compressedData**.

Decompress

1. Recover Huffman Tree Mapping Table

Firstly, we will recover the **size** of *Huffman Table* and the **maximum size** of *Huffman Code* from **compressedData**.

```
// Huffman Tree Table Size
int treeSize = compressedData[++pos_data] & 0xFF;
/*
 * Since The table size will range from 1 to 256
 * We use 0 to denote 256
 */
if (treeSize == 0) treeSize = 256;
// Maximun Length of Encoding
int byte_encoding_max = compressedData[++pos_data] & 0xFF;
```

Then, we recover the *Huffman Tree Mapping Table*.

```
unsigned char* huffmanTree = new unsigned char[256 * 32];
memset(huffmanTree, 0, sizeof(unsigned char) * 256 * 32);
int *encoding_len = new int[256];
memset(encoding_len, 0, sizeof(unsigned int) * 256);
for (int i = 0; i < treeSize && pos_data < cDataSize; ++i) {
    // Recover the key of Huffman Table
    unsigned int key = compressedData[++pos_data] & 0xFF;
    // Recover the value of Huffman Table
    for (int j = 32 - byte_encoding_max; j < 32; ++j) {
        huffmanTree[key * 32 + j] = (compressedData[++pos_data] & 0xFF);
    }
    // Recover the encoding length of Huffman Table
    encoding_len[key] = (int)(compressedData[++pos_data] & 0xFF);
}
```

2. Rebuild Huffman Tree

TODO: Introductoin of rebuilding Huffman Tree

3. Scan encoded string and recover pixels

Due to the that we can only manipulate a char(8 bit) at a time, we use an unsigned char as a buffer to read 8 bit at a time.

Then we traverse the *Huffman Tree* node by node according the data bit by bit.

```
unsigned char buf = compressedData[++pos_data] & 0xFF;
unsigned int pos_buf = 0;
while (pos_data < cDataSize) {
    treeNode* node = rootNode;
    while (1) {
        // Buffer end, read new data
        if (pos_buf > 7) {
            buf = compressedData[++pos_data] & 0xFF;
```

```
            pos_buf = 0;
        }
        if (((buf >> (7 - pos_buf)) & 0b1) == 1) {
            if (node->leftChild != nullptr) {
                node = node->leftChild;
            } else {
                // Find a value
                break;
            }
        }
        else {
            if (node->rightChild != nullptr) {
                node = node->rightChild;
            } else {
                // Find a value
                break;
            }
        }
        ++pos_buf;
    }
    // Write to uncompressedData
    uncompressedData[pos_uncompressed_data++] = node->key & 0xFF;
}
```

Each time we find a value in the tree, we write the value to **uncompresedData** simultaneously.

Finally, we finished decompressing the image.

Experiment

Procedure

Beach

Figure 4: Beach.jpg

Type	Compression Ratio
Before Quantized	1.04
After Quantized	1.71

Red Panda

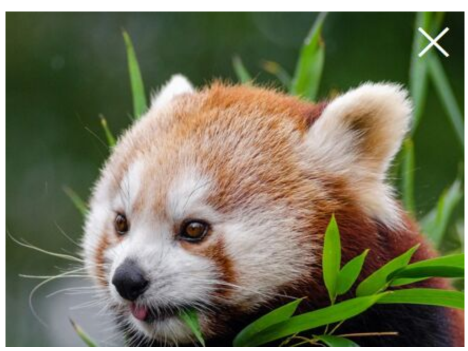

Figure 5: Red Panda.jpg

Type	Compression Ratio
Before Quantized	1.02
After Quantized	1.65

sunset

Figure 6:

Type	Compression Ratio
Before Quantized	1.03
After Quantized	1.67

Tuxinu

Figure 7: Tuxinu.png

Type	Compression Ratio
Before Quantized	1.52
After Quantized	2.25

Hydrogen (Custom Image)

Figure 8: Hydrogen.png

Type	Compression Ratio
Before Quantized	5.65
After Quantized	5.65

Explaination

According to the result above, we can see there is an improvement of the **Compression Ratio** in images *Beach*, *Red Panda*, *sunset*, *Tuxinu*.

The reason why the **Compression Ratio** improves is that we did a **5-5-5 quantization** on the image. This is compressing the color space because it will map similar color to same color. This will reduce the number of distinct numerical value of three-primary colors of RGB and increase the appearance frequency of certain kinds of three-primary colors of RGB.

Since we use **Huffman Coding** algorithm to compress image, the encoded length is depending on the appearance frequency of the three-primary colors of RGB of all pixels, as well as the number of distinct numerical value of three-primary colors of RGB.

After **5-5-5 quantization**, the number of distinct numerical value of three-primary colors of RGB reduced, while the appearance frequency of certain kinds of three-primary colors of RGB increased. This will obviously improve the **Compression Ratio**.

Thus, we can explain that the **Compression Ratio** of the last image (*Hydrogen.png*) won't improve because the image has too few colors since **5-5-5 quantization** neither reduced colors nor increase the appearance frequency of colors.