一，目标

将lena图片使用jpeg算法进行压缩并解压缩，验证算法正确性。（最终成果共1700行左右代码）

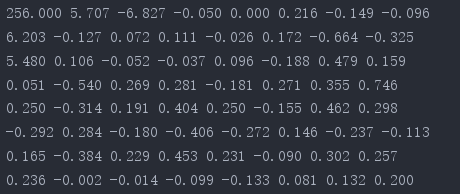
二，算法流程和原理简介

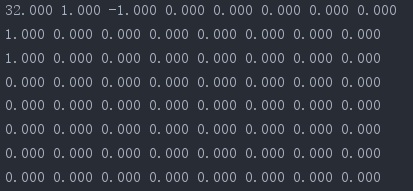
1. Jpeg是一个有损压缩算法，有损步骤主要在量化步中的取整缓解，而压缩的空间则来自于基于熵编码的霍夫曼编码(Huffman Coding)系统和取整后大量的0被游程编码(RLE)高效替代或丢弃。
2. 总的来看，jpeg算法流程如下：图片分8x8块读取(彩色RGB需转YCbCr但本次实验为简化后续对JFIF文件格式二进制级别的低层探索，采用灰度图输入) -> 对每个8x8patch做离散余弦变换 -> 量化 -> Zig Zag展开 -> 长程编码和霍夫曼编码 -> 按照JFIF ISO标准写入比特流(不借助外部库直接写入二进制得到win能打开的jpg文件)
3. 解压缩流程则是将上述过程反向：按照JFIF格式读取比特流，并从文件头中恢复该jpg文件使用的量化表，霍夫曼编码表，采样率等基础信息 -> 使用以上信息进行霍夫曼反编码 -> 反Zig-Zag重建8x8 Patch -> 反量化 -> Inverse DCT得到原始图片信息
4. 值得一提的是，为了提升算法速度，本次实现中我们测试了Naïve DCT和一种高效的快速DCT，并基于快速DCT的一维算法，先行后列构建了2D的图片快速DCT



图1 8x8 Patch得到的DCT系数频谱(白色为高值，不过8x8相对于输入512x512显得依然很小，因此频谱图中依然可以近似看出原图模样)

1. 量化步骤：此步骤其实是使用一个预先定义的数指矩阵对DCT系数矩阵进行缩放，并进行取整，从而将一些高频系数降低为0，达到图片压缩的目的。此处基于的考虑是，人类对于图片中的高频部分也即边缘和剧烈的色彩变动并不是很敏感，因此对彩色图片我们可以降低Cb/Cr通道的采样率，保持照度也即Y通道的采样率(灰度)，进一步，使用量化矩阵将高频系数直接置零，并且在感官上并不会影响图片观感





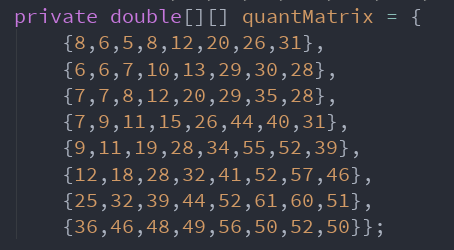


图2 使用量化矩阵对一个patch的DCT系数做量化后的结果: 可以发现大量高频系数通过取整变为了0，从而减少了大量容量

1. Zig-Zag展开：最终存储到文件中的数据不可能保证矩阵的形式，不利于压缩。因此人们按照从低频系数到高频的顺序来展开8x8系数矩阵，也即Zig-Zag顺序。如下图，(来自wiki)

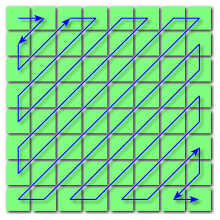


图3 Zig-Zag展开

代码实现中这一部分可以理解为逐对角行扫描，并且在满足角点条件后改变一下系数变化规则即可。



图4 核心原理部分

1. RLE和Huffman编码：在得到一串串长度为64的量化系数数组之后，我们需要对这串数字编码，从而得到最终的结果。假设后63个系数输入如下(暂时不讨论第一个系数，后文详解)，

57,45,0,0,0,0,23,0,-30,-16,0,0,1,0,0,0, 0 , 0 ,0 , 0,..,0

经过 RLE 压缩后就是

(0,57) ; (0,45) ; (4,23) ; (1,-30) ; (0,-16) ; (2,1) ; EOB

EOB 是一个结束标记, 表示后面都是 0 了。实际上, 我们用 (0,0) 表示 EOB。另外需要注意的是, 由于后面 huffman 编码的要求, 每组数字前一个表示 0 的数量的必须是 4 bit, 就是说, 只能是 0~15, 所以, 如果有这么一组数字:

57, 十八个0, 3, 0, 0, 0, 0, 2, 三十三个0, 895, EOB

我们实际这样编码:

(0,57) ; (15,0) (4,3) ; (4,2) ; (15,0) (15,0) (4,895) , (0,0)

注意 (15,0) 表示了 15个连续的 0。最后在每个块结束后加一个EOB表示当前8x8已经结束。(以上例子来自[网络](http://www.opennet.ru/docs/formats/jpeg.txt))

根据以上结果，进一步进行Huffman编码，实际的Huffman编码算法大家必然都清楚，但是为了统一标准，ISO有一套专门的标准Huffman表格，并且宣称这个编码适合大部分图片并能达到很好的平均压缩水平。这里的Huffman编码分为DC/AC系数，也即直流和交流，这两个术语源自于余弦变换中首个系数实为t=0时的分量，也即是一个常量，代表了图片的均值，具体推导可以参考图像处理一书。由于第一个系数是常量是不变的，而后的系数都有一个变化的余弦分量，因此第一个系数叫直流分量(DC)，其后的63个系数都叫交流分量(AC)。

每一个数字的编码分为两部分，基码和数值编码。基码部分通过查表即可获得，DC系数还需要减去上一个量化后的DC系数(JPEG委员会认为临近patch的直流系数强相关，如此可以进一步增大压缩率)，然后得到DC/AC类别(AC还需要一个前面的0系数个数作为游程来查表)，即可获得基码。然后这条基码的类别数值其实就是二进制的数字表示下这个数据的长度(注意负数需要对绝对值转二进制后取反但不加一，并非标准的2’s Complement表示)

以下例子来自[网络](http://www.opennet.ru/docs/formats/jpeg.txt)

还是来看前面的例子:

(0,57) ; (0,45) ; (4,23) ; (1,-30) ; (0,-8) ; (2,1) ; (0,0)

只处理每对数右边的那个:

57 是第 6 组的, 实际保存值为 111001 , 所以被编码为 (6,111001)

45 , 同样的操作, 编码为 (6,101101)

23 -> (5,10111)

-30 -> (5,00001)

-8 -> (4,0111)

1 -> (1,1)

(组,对应的二进制)

前面的那串数字就变成了:

(0,6), 111001 ;

(0,6), 101101 ;

(4,5), 10111;

(1,5), 00001;

(0,4) , 0111 ;

(2,1), 1 ;

(0,0)

括号里的数值正好合成一个字节。后面被编码的数字表示范围是 -32767..32767.

合成的字节里, 高 4 位是前续 0 的个数, 低 4 位描述了后面数字的位数。注意括号后面数字的编码，正值就直接变二进制，负值取反就够了。继续刚才的例子, 如果 06 的 huffman 编码为 111000 ( 06 对应 111000 为查表所得。jpeg 文件里保存了压缩时所产生的 huffman 表, 将 0~255 这 256 个 8 bits 定长数字, 对应成 1~16 bits 的不定长数值. 出现频率高的数字小于 8bits, 频率低的大于8bits,从而使整个的数据长度降低, jpeg 实际使用的是范式 Huffman 编码(Canonical Huffman Code)

69 = (4,5) --- 1111111110011001 ( 注: 69=4\*16+5=0x45 )

21 = (1,5) --- 11111110110

4 = (0,4) --- 1011

33 = (2,1) --- 11011

0 = EOB = (0,0) --- 1010

那么最后对于前面的例子表示的 63 个系数 (记得我们将第一个跳过了吗?) 按位流

写入 JPG 文件中就是这样的:

111000 111001

111000 101101

1111111110011001 10111

11111110110 00001

1011 0111

11011 1

1010

对于前面掠过的DC系数，每次编码的是差值，也即Diff = DC(i) - DC(i-1)。所以恢复的时候这一块的 DC(i) 就是: DC(i) = DC(i-1) + Diff。JPG 从 0 开始对 DC 编码, 所以 DC(0)=0. 然后再将当前 Diff 值加在上一个值上得到当前值。下面再来看看上面那个例子: (记住我们保存的 DC 是和上一块 DC 的差值 Diff)例如上面例子中, Diff 是 -511, 就编码成

(9, 000000000)

如果 9 的 Huffman 编码是 1111110 (在 JPG 文件中, 一般有两个 Huffman 表, 一

个是 DC 用, 一个是 AC 用) 那么在 JPG 文件中, DC 的 2 进制表示为

1111110 000000000

它将放在 63 个 AC 的前面, 上面上个例子的最终 BIT 流如下:

1111110 000000000 111000 111001 111000 101101 1111111110011001 10111

11111110110 00001 1011 0111 11011 1 1010

三，代码实现和实验结果

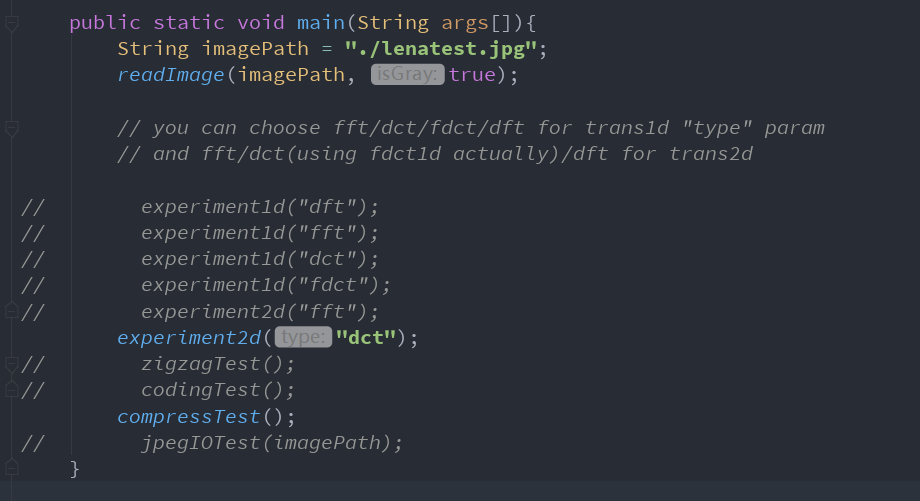
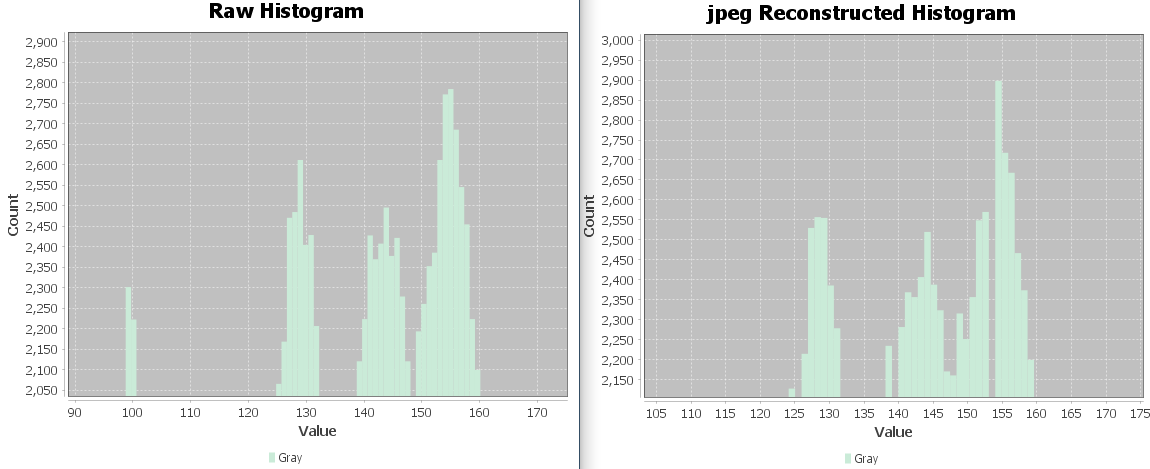


图5 函数入口，可以选择各类实验进行测试，最终图片的压缩测试在compressTest中，探索实验在jpegIOTest中



图6 (左到右)原图，DCT重构，JPEG(DCT+量化)重构：可以发现观感上并无明显差别



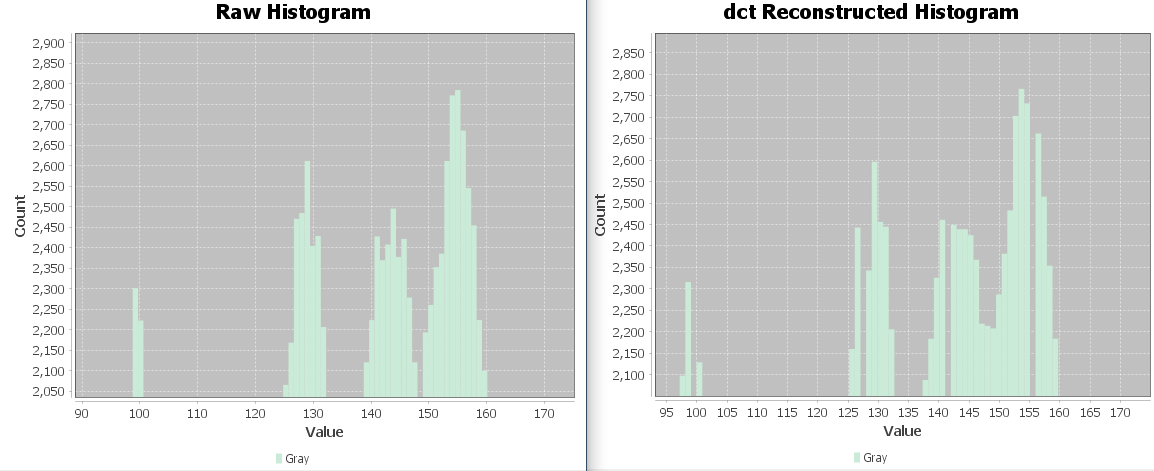


图7 灰度分布：对比图片的灰度分布细节，可以发现jpeg确实相比较DCT又损失了一部份精度

总的来说，JPEG压缩后图片细节的保留程度取决于量化矩阵的数值大小，本实验中采用了标准矩阵和java imageIO库中使用的矩阵(实为标准阵/2)，以上分布图来自更高质量的量化矩阵(/2版本)，因此重构后的图片中8x8的矩阵小方格不是很明显。但根据作者个人经验，尤其是深度学习的小目标检测/分割任务中，png输入和jpg输入对模型最终性能有较大影响，8x8的小边缘对模型其实有着很大的干扰并且也丢失了很多小目标的边缘信息(不再满足8x8小patch中像素分布平滑的假设，同理文字扫描图片jpg格式会有斑点)

四，额外的探索和发现

最后，也是最有趣的部分，来自于我尝试对照JPEG标准，从二进制数据中读取jpg，并在不依赖任何外部图片库的条件下，读写jpg文件。

以下是一个简单的8x8patch的jpg文件16进制数据(small.jpg)，

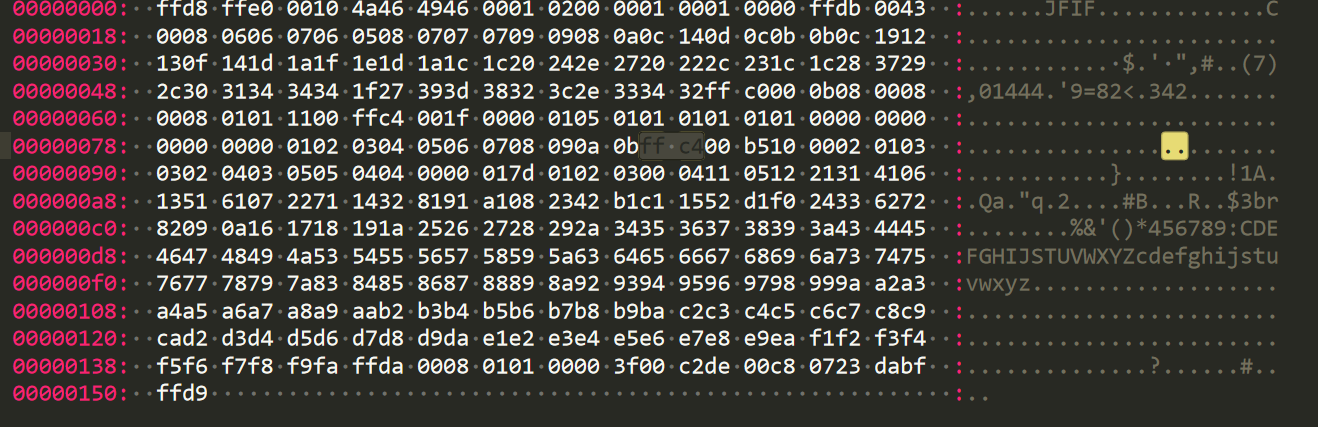


图8 16进制jpg图片数据

JPG文件是一种分段文件格式，每段都记录了压缩图片所用到的各种信息，并且有自己的标识等规定，细节如下，更多资料请查阅随附链接

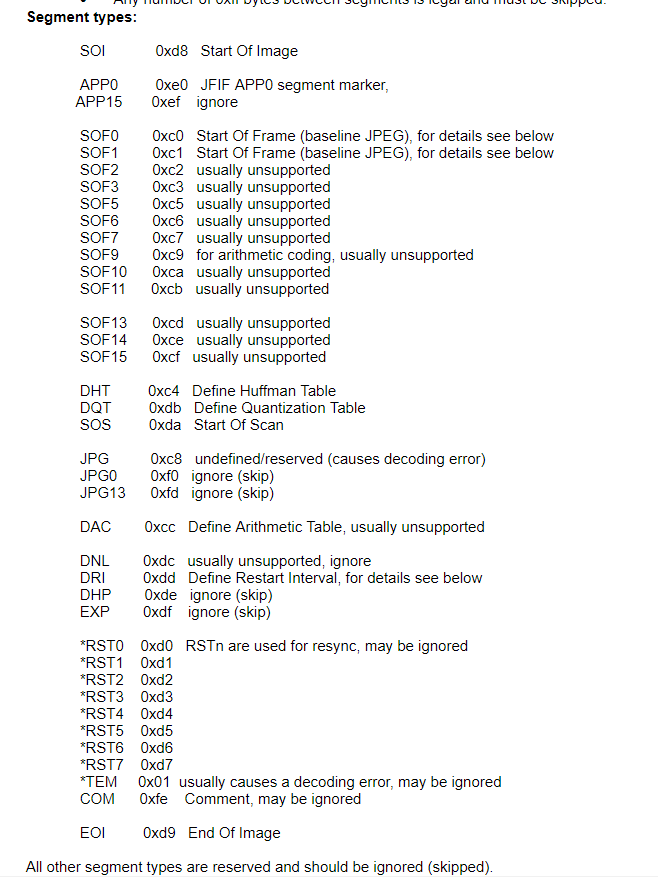


图9 jpg文件段格式

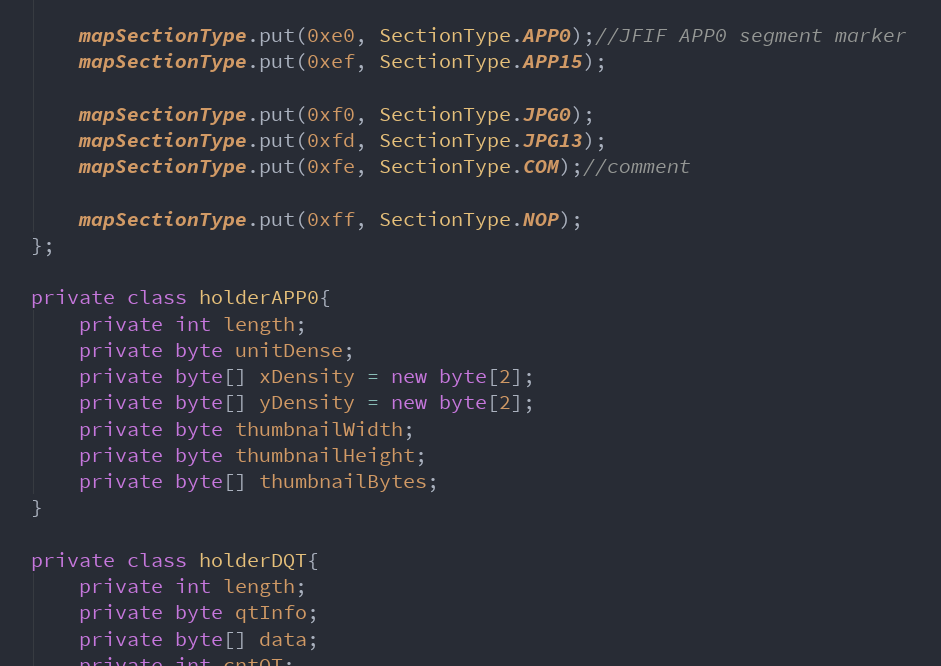




图10 jpegIO类代码节选

这段探索历程非常底层，也很痛苦，再一次让我认识到抽象是一个多么伟大的思想。但是在从jpg文件头中读取到对应的DHT/DQT段信息之后，我发现无法使用jpg自带的这些量化表和huffman表重建图片，同时经过多方核对验证，我读取和恢复文件头信息的过程都是正确的，并且对于同一个8x8patch我的算法压缩结果和jpg数据也不一样。此时我突然想到使用图像处理书本背后的huffman表格进行解码(此前就注意到网络上很多公开资料使用的所谓jpeg标准huffman编码不同于书背后附录中的编码，很奇怪)，为此我手工进行了解码，发现竟然可以成功解码。因此我不得不怀疑Java ImageIO库和windows系统中对jpg文件的解释都并非按照jpg头中的信息解码，而是为了速度起见，有一套默认的约定算法进行编码解码，并且相关参数并不透明，更多问题细节，详见随附文档。

最后附上整个作业过程中使用到的多个链接，感谢前人的探索结果，让我有了今日的更进一步。

<http://www.opennet.ru/docs/formats/jpeg.txt> 上古jpeg文件格式详解

<http://vip.sugovica.hu/Sardi/kepnezo/JPEG%20File%20Layout%20and%20Format.htm> 简明jpeg段格式总结

[https://www.globalspec.com/reference/39556/203279/appendix-b-huffman-tables-for-the-dc-and-ac-coefficients-of-the-jpeg-baseline-encoder](https://www.globalspec.com/reference/39556/203279/appendix-b-huffman-tables-for-the-dc-and-ac-coefficients-of-the-jpeg-baseline-encoder%20注意对比书本中AC系数0/6)  注意对比书本中AC系数0/6的编码和此处的编码（根据jpg头信息手绘huffman树得到的编码也与这些网上公开资料相同，和书本附录不同）

<https://digitalexploration.wordpress.com/2009/07/29/jpeg-huffman-tables/> huffman树和表格的互相转化

<https://www.cnblogs.com/reddusty/p/4850406.html>

<https://blog.csdn.net/play_fun_tech/article/details/18132611>

<https://www.nayuki.io/page/how-to-implement-the-discrete-fourier-transform> 简洁明了的fft实现(算法细节可通过图像处理书本自学，讲的很好)

<https://unix4lyfe.org/dct-1d/> 为数不多写对了DCT归一化系数的博客。。。