

1

****

软件技术基础大作业

基于Huffman编码与RLE编码实现bmp文件的无损压缩与解压

|  |  |
| --- | --- |
| 学院名称 | 航空科学与工程学院 |
| 学号 | SY1905120 |
| 学生姓名 | 王睿 |

**2019**年**12**月

摘要

数据压缩技术的主要目的是通过特定的编码方式将信息数据量以压缩形式进行储存和传输。图像压缩是数据压缩中应用最为广泛的方面之一。用于表示一幅数字图像的数据量通常是很大的，从而给其储存和传输都带来许多问题，单纯增加储存器容量以及提高信道带宽并不现实，因此解决储存和传输问题需要依靠图像编码进行图像压缩。图像压缩分为有损压缩和无损压缩两类，Huffman编码和RLE编码都是用于无损压缩的效率较高的图像编码方式之一。本文基于Huffman编码和RLE编码设计了用于bmp图像文件的压缩与解压算法，对算法原理和优劣势进行了剖析，利用若干图像测试了算法的压缩率，并分析了进一步改进措施。

关键词： 图像压缩，Huffman编码，RLE编码

目 录

[1 引言 1](#_Toc27143892)

[1.1 研究背景与研究目的 1](#_Toc27143893)

[1.2 无损压缩算法 1](#_Toc27143894)

[1.2.1 基于概率统计模型的压缩算法 1](#_Toc27143895)

[1.2.2 基于字典模型的压缩算法 2](#_Toc27143896)

[1.3 有损压缩算法 3](#_Toc27143897)

[1.4 本文内容与章节安排 3](#_Toc27143898)

[2 算法设计 4](#_Toc27143899)

[2.1 编码原理 4](#_Toc27143900)

[2.1.1 Huffman编码原理 4](#_Toc27143901)

[2.1.2 RLE编码原理 5](#_Toc27143902)

[2.2 算法设计 5](#_Toc27143903)

[2.2.1 RLE编码与解码 5](#_Toc27143904)

[2.2.2 Huffman节点类 6](#_Toc27143905)

[2.2.3 Huffman树构建 7](#_Toc27143906)

[2.2.4 文件压缩与解压 7](#_Toc27143907)

[3 压缩测试 10](#_Toc27143908)

[4 结论与分析 11](#_Toc27143909)

[4.1 总结 11](#_Toc27143910)

[4.2 算法改进 11](#_Toc27143911)

[参考文献 12](#_Toc27143912)

[附录 13](#_Toc27143913)

[附录：源代码 13](#_Toc27143914)

## 引言

* 1. 研究背景与研究目的

随着信息技术的不断发展，海量数据的传输与储存成为了不可忽视的问题。数据压缩便是解决这一问题的重要方法。数据压缩随着人类进入信息时代, 人类将更多的依靠电脑，但获取的大部分数据信息却都具有数据的海量性。在这种情况下, 信息的传输与存储就成为首要问题。数据压缩就成为解决这一瓶颈问题的重要方法。数据压缩起源于20世纪40年代的信息论，其主要目的是通过数据压缩手段将信息数据量以压缩的形式储存和传输。数据压缩的理论极限是信息熵。众所周知，数据文件在计算机中是以计算机符号表示的，每个计算机符号有一定的出现概率。通过一定的模型与编码进行组合计算，分析符号的出现概率并进行重编码构成了压缩技术的主要思路。根据不同的编码对原始文件数据产生不同损失的效果，可以把压缩技术分为有损压缩和无损压缩两大类。无损压缩是可逆的，即从压缩后的数据可以完全恢复出原来的图像，没有任何信息损失；而有损压缩是不可逆的，从压缩后的数据无法完全恢复原来的图像，信息有一定的损失。通常情况下有损压缩的压缩效率比无损压缩要高。

* 1. 无损压缩算法

无损压缩算法根据模型的不同可以分为基于统计概率模型的压缩算法和基于字典模型的压缩算法。

* + 1. 基于概率统计模型的压缩算法

基于概率统计模型的压缩算法分为静态统计模型方式和自适应模型方式。静态统计模型方式需要预先扫描文件中所有的字符，以便统计出每个字符出现的频率，但由于扫描文件需要花费大量的时间，并需要保存一份概率表，从而导致压缩效率的下降。而对于自适应统计模型方式，则在开始时假定每个字符出现的概率相等，但随着字符的不断输入和编码，统计并记录字符出现的概率，并将此概率应用于后续字符的编码。自适应统计模型方式不需要保存概率表。基于概率统计模型的压缩算法最具代表性的编码方式是Huffman编码和算术编码。

（1）Huffman编码：Huffman编码用于TIFF，JPEG等图像格式中，该编码是Huffman为压缩文本文件而建立的，其压缩原理是先统计需要编码的字符的出现频率，然后将短码赋予出现频率高的字符，将长码赋予出现频率低的字符。Huffman编码简单有效，应用广泛。但产生Huffman编码需要对原始数据扫描两遍，第一遍扫描要统计出原始数据中每个值出现的频率，第二次扫描是建立Huffman树并进行编码。由于需要建立二叉树并遍历二叉树生成编码，因此数据的压缩和还原速度都比较慢。另外，由于Huffman编码时所有的位都集中在一起而不考虑字节分界，译码程序判断码结束的唯一方法是达到二叉树的一个分支。因此Huffman编码对于位的增减反应敏感，如果增加或减少位，译码程序将无法正确翻译出后面的数据。

（2）算术编码：将编码的一条消息或字符串表示成0和1之间的一段间隔，即对一串符号直接编码成[0,1]区间上的一个浮点小数，用小数表示二进制位，并由此接近无损压缩的熵极限。算术编码可以较为方便的使用静态统计模型方式和自适应模型方式。使用静态统计模型方式能够更好地接近无损压缩的熵极限，但无法适应信息的多样性，同时必须在压缩前对所有字符分布进行统计耗费大量时间。而自适应模型方式能够弥补这一缺陷，因此对于算术编码通常使用自适应模型方式，根据当前接受的数据不断更改概率模型以达到更好的压缩效率，当信源信号概率较为接近时采用算术编码。

* + 1. 基于字典模型的压缩算法

基于字典模型的压缩算法是将已经编码过的信息作为原字典，如果需要编码过的信息曾经出现过，就输出该字符串出现位置及长度，否则就输出一个新的字符串。字典压缩模型通常使用自适应的方式。如果使用静态字典模型方式，存在适应性不强以及需维护信息量较大的字典的缺点，从而影响了最终的压缩效果。基于字典模型的压缩算法主要有游程编码（RLE）和LZW编码两种方式。

（1）RLE编码：RLE编码通常称为游程编码。这种编码方式适用于PCX, TIFF等图像格式中，其编码原理较为简单。在图像压缩中，游程编码将一行中颜色值相同的相邻像素用一个计数值和颜色值代替。当图像中存在很多块颜色相同的大面积区域，则RLE编码产生的压缩率是很高的。

（2）LZW编码：LZW编码用于GIF, TIFF等图像文件中。LZW编码不需要在编码前构造码表，而是在压缩过程中逐步建立字典。其基本思想是将每一个字节的值都要与下一个字节值配对成一个字符对，并为每一个字符设置一个代码。当同样的一个字符对再度出现时就用代号代替这一字符对，然后再用这个代号和下一个字符配对。在配对过程中，必须建立三个表格，即字首表、字符表、代号表。所有字符对和代号都分别存入这三个表格中。LZW不仅可以与RLE压缩算法一样对连续出现的相同字符进行压缩，而且还可以对经常出现的由不同字符组成的字符串进行压缩。但如果原始图像数据值中带有随机变化的噪音图像，则很难利用LZW算法来压缩。

* 1. 有损压缩算法

有损压缩主要是利用人类视觉对图像中的某些频率成分不敏感的特性，允许在不影响整个图像的视觉效果的前提条件下，在压缩过程中损失一定的信息以获取较高的压缩比。有损压缩虽然不能完全恢复原始数据，但损失的数据对于理解原始图像信息影响不大。并由此换取较大的压缩比。因此，有损压缩大部分应用于影音、图像和视频数据的压缩，利用图像数据的冗余性来减少图像的数据量，在对于原图像进行抽样量化和编码过程去掉冗余信息后，再对其进行存储、传输，最后恢复成不影响视觉效果的图像。其关键技术是抽样量化和编码方法。

离散余弦变换（DCT）是常用的有损压缩算法之一。DCT属于变换编码，处理过程中将预先已经分成小块的原始图像进行DCT变换，其高频部分包含了锐利的边缘信息，而低频部分包含了图像的主要信息，通过量化步骤有选择性的消除或较为粗糙地量化高频部分。需要注意的是，压缩不是在变换步骤取得的，而是在量化时取得的，并且是有损压缩不可恢复。与离散傅里叶变换（DFT）和沃尔什-哈德玛变换（WHT）相比，DCT具有更强的信息集中能力，近年来在有损压缩中得到了较多的应用。

* 1. 本文内容与章节安排

本文结合Huffman编码和RLE编码方式对bmp图像文件进行压缩和解压。第一章为引言部分，介绍了有损压缩与无损压缩的概念及常见的压缩算法；第二章为算法设计，解释了算法与代码逻辑；第三章为压缩测试，采用若干bmp图像文件对算法进行压缩和解压测试，给出其压缩率；第四章为结论与分析，同时给出算法进一步改进的方向。

## 算法设计

* 1. 编码原理
     1. Huffman编码原理

在计算机数据处理中，Huffman编码使用变长度编码表对原符号进行编码，其中变长度编码表是通过一种评估来源符号出现几率的方法得到的，出现几率搞得字母使用较短的编码，反之，出现几率低的则使用较长的编码，这便使得编码后的字符串的平均长度、期望值降低，从而达到无损压缩数据的目的。Huffman编码的基本步骤为：

（1）根据给定的n个权值构成的n棵二叉树的集合F，其中每棵二叉树中只有一个带权根节点，其左右子树均为空。

（2）在F中选取两棵根节点权值最小的树作为左右子树构造一棵新的二叉树，且将新的二叉树根节点权值置为左右子树上根节点权值之和。

（3）在F中删除这两棵子树，同时将新得到的二叉树加入到F中。

（4）重复2和3步骤，直到F仅含有一棵树为止，这棵树便是Huffman树。

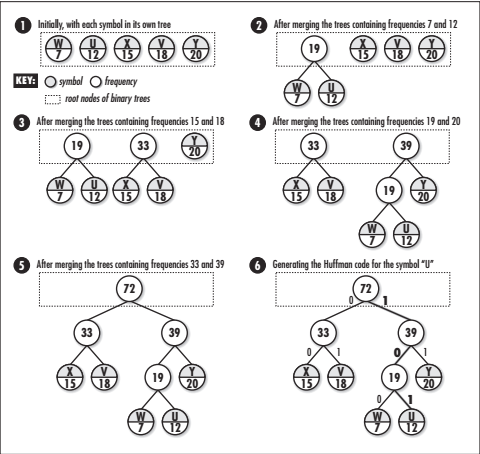


图2.1 Huffman编码原理

* + 1. RLE编码原理

游程编码（RLE）是一种比较简单的压缩算法，其基本思想是将重复且连续出现多次的字符使用（连续出现次数，某个字符）来描述。例如一个字符串AAAAABBBBCCC，使用游程编码可以将其描述为5A4B3C。其中，5A表示这个地方有5个连续的A，同理4B表示有4个连续的B，3C表示有3个连续的C，其它情况以此类推。原字符串需要12个字符才能描述，而使用游程编码压缩之后只需要6个字符就可以表示，还原回去的时候只需要将字符重复n次即可。

由于游程编码的基本思想是将重复且连续出现的字符进行压缩，使用更简短的方式来描述，这种方式是基于柯氏复杂度的。例如有三个字符串长度都是100，其中第一个是100个A，第二个是99个A和一个B，第三个是100个完全随机的字符，用尽可能短的语言来描述原字符串，描述第一个字符串时可以说“这是100个A”，描述第二个字符串可以说“这是99个A，然后是一个B”，但是描述第三个字符串时如果说“这是100个随机字符”，则无法根据描述还原字符串内容。这个描述信息的长度就称之为柯氏复杂度，在这个例子中第一个柯氏复杂度最小，第二第三依次次之。

假如采用定长1个字节来描述连续出现次数，并且一个字符占用1个字节，那么描述（连续出现次数，某个字符）需要的空间是2个字节，只要这个连续出现次数大于2就能够节省空间，比如AAA占用3个字节，编码为(3,A)占用两个字节，能够节省一个字节的空间，可以看出连续出现的次数越多压缩效果越好，节省的空间越大，对一个字符编码能够节省的空间等于=连续出现次数-2。

* 1. 算法设计
     1. RLE编码与解码

将游程编码与解码的代码写作如下：

*'''*

*1. 游程编码压缩*

*'''*

defrle*(*inputfile*,*outputfile*):*

*# 1.打开读入文件和写入文件*

*f*=open*(inputfile, 'rb')*

*w*=open*(outputfile, 'wb')*

*# 2.遍历并计数*

*last*=*None*

*count*=*0*

*t*=*f.read(1)*

while*t:*

if*last*is*None:*

*last*=*t*

*count*=*1*

else*:*

if*t*==*last*and*count*<*255:*

*count*+=*1*

else*:*

*w.write(int.to\_bytes(count, 1,*byteorder=*'big'))*

*w.write(last)*

*last*=*t*

*count*=*1*

*t*=*f.read(1)*

*# 3.将计数和字符写入压缩文件中*

*w.write(int.to\_bytes(count, 1,*byteorder=*'big'))*

*w.write(last)*

*f.close()*

*w.close()*

*'''*

*2. 游程编码解压*

*'''*

defderle*(*inputfile*,*outputfile*):*

*w*=open*(outputfile, 'wb')*

*f*=open*(inputfile, 'rb')*

*count*=*f.read(1)*

*byte*=*f.read(1)*

while*count*and*byte:*

*w.write(int.from\_bytes(count,*byteorder=*'big')*\**byte)*

*count*=*f.read(1)*

*byte*=*f.read(1)*

*w.close()*

*f.close()*

* + 1. Huffman节点类

定义Huffman节点类，包含节点权值与父子节点的信息。同时定义了子节点的编码方式，左子节点编码为0，右子节点编码为1。定义类的代码写作如下：

*'''*

*3. 定义哈夫曼树的节点类HuffNode*

*'''*

class*HuffNode(object):*

*# 1.初始化*

def\_\_init\_\_*(*self*,*value=*None,*left=*None,*right=*None,*father=*None):*

self*.value*=*value*

self*.left*=*left*

self*.right*=*right*

self*.father*=*father*

*# 2.父节点构建，并关联子节点*

defbuildfather*(*left*,*right*):*

*n*=*HuffNode(*value=*left.value*+*right.value,*left=*left,*right=*right)*

*left.father*=*n*

*right.father*=*n*

return*n*

*# 3.左右节点编码*

defencodenode*(*n*):*

if*n.father*==*None:*

returnb*''*

if*n.father.left*==*n:*

return*HuffNode.encodenode(n.father)*+b*'0'*

else*:*

return*HuffNode.encodenode(n.father)*+b*'1'*

* + 1. Huffman树构建

按照前文中提到的树的构建方式构建Huffman树，代码写作如下：

*'''*

*4. 哈夫曼树的构建*

*'''*

defbuildtree*(*l*):*

*# 1.若树唯一则返回节点为哈夫曼树*

iflen*(l)*==*1:*

return*l*

*# 2.若树不唯一则按规则进行二叉哈夫曼树构建,排序-构建-删除-插入-排序*

*sorts*=sorted*(l,*key=lambdax*: x.value,*reverse=*False)*

*n*=*HuffNode.buildfather(sorts[0], sorts[1])*

*sorts.pop(0)*

*sorts.pop(0)*

*sorts.append(n)*

*# 3.不断循环，直到仅存有一棵哈夫曼树*

return*buildtree(sorts)*

* + 1. 文件压缩与解压

经过游程编码与Huffman编码后的数据应写入到压缩文件中，其中写入数据包括文件名、文件大小、字节宽度、字符出现频率和编码信息等。文件压缩的主要思路如下：

（1）读入待压缩文件，利用游程编码进行第一次压缩并储存压缩文件

（2）在Huffman编码中进行第一次扫描，遍历文件的所有字符，统计每个出现的频率并将单个字符构建为单一节点的Huffman节点类

（3）建立Huffman树并对其进行编码

（4）数据写入。以二进制的方式进行数据写入，写入代码如下：

*# 5.信息写入*

*o.write((name[*len*(name)*-*1]*+*'\n').encode(*encoding=*"utf-8"))*

*o.write(int.to\_bytes(*len*(ec\_dict), 2,*byteorder=*'big'))*

*o.write(int.to\_bytes(writewidth, 1,*byteorder=*'big'))*

for*x*in*ec\_dict.keys():*

*o.write(x)*

*o.write(int.to\_bytes(count\_dict[x], writewidth,*byteorder=*'big'))*

*# 6.数据压缩写入*

for*i*inrange*(int(count)):*

for*x*in*ec\_dict[buff[i]]:*

*raw*=*raw*<<*1*

if*x*==*49:*

*raw*=*raw*|*1*

if*raw.bit\_length()*==*9:*

*raw*=*raw*&*(*~*(1*<<*8))*

*o.write(int.to\_bytes(raw, 1,*byteorder=*'big'))*

*o.flush()*

*raw*=0b*1*

*tem*=*int(i*/len*(buff)*\**100)*

if*tem*>*last:*

print*("压缩进度：", tem, '%')*

*last*=*tem*

*i*=*i*+*1*

（5）处理最后不足8位的01串，进行字节对齐。字节对齐的代码写作如下：

*# 8.处理尾部数据*

if*raw.bit\_length()*>*1:*

*raw*=*raw*<<*(8*-*(raw.bit\_length()*-*1))*

*raw*=*raw*&*(*~*(1*<<*raw.bit\_length()*-*1))*

*o.write(int.to\_bytes(raw,1,*byteorder=*'big'))*

文件的解压与文件压缩是逆向过程。通过重构Huffman树和编码表还原源文件数据。详细代码可见附录。

## 压缩测试

采用若干张bmp文件对进行压缩与解压测试，压缩效果如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 源文件名 | 源文件大小 | 压缩文件大小 | 压缩率 | 解压缩文件大小 |
| test1.bmp |  |  |  |  |
| test2.bmp |  |  |  |  |
| test3.bmp |  |  |  |  |
| test4.bmp |  |  |  |  |
| test5.bmp |  |  |  |  |
| test6.bmp |  |  |  |  |
| test7.bmp |  |  |  |  |
| test8.bmp |  |  |  |  |
| test9.bmp |  |  |  |  |
| test10.bmp |  |  |  |  |

## 结论与分析

* 1. 总结
  2. 算法改进

## 参考文献

[1] 于耕. 航空应急救援[M]. 航空工业出版社, 2009.

## 附录

附录：源代码