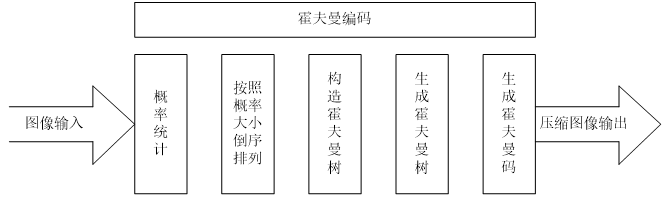
# 实验六 数字图像压缩与解压实验

## 实验目的

1. 理解霍夫曼编码和解码的基本步骤。
2. 掌握霍夫曼编码和解码的操作，编程实现。

## 实验原理

1. **霍夫曼编码**
2. 霍夫曼在1952年提出了一种构造最佳码的方法，称之为霍夫曼编码（Huffman）。霍夫曼编码是一种无损的统计编码方法，利用信息符号概率分布特性的改变字长进行编码。霍夫曼编码适用于多远独立信源，对于多元独立信源来说它是最佳码。
3. 霍夫曼编码系统主要分为压缩对象输入、概率统计、构造Huffman树、生成Huffman树、压缩编码环节组成，如下图所示霍夫曼编解码系统构成。



1. **霍夫曼编码的步骤如下:**
2. 将信源符号按出现概率从大到小排成一列,然后把最末两个符号的概率相加,合成一个概率。
3. 把这个符号的概率与其余符号的概率按从大到小排列,然后再把最末两个符号的概率加起来，合成一个概率。
4. 重复上述做法，直到最后剩下两个概率为止。
5. 从最后一步剩下的两个概率开始逐步反向进行编码。每步只需对两个分支各赋予一个二进制码，如对概率大的赋予码1，对概率小的赋予码O。

## 实验内容与要求

基于霍夫曼编码和解码实现图像的压缩和解压缩，计算图像压缩比。

## 实验的具体实现

请同学们完成上述实验并完成实验报告（代码+实验结果）。

# -\*- encoding: utf-8 -\*-

'''

@File    :   main.py

@Time    :   2022/05/20 20:37:32

@Author  :   HuangRongQuan

@Version :   1.0

@Contact :   huang\_rongquan@outlook.com

'''

import os

import six

class HuffNode(object):

    """\_定义一个HuffNode虚类,包含两个虚方法\_

    Args:

        object (\_type\_): \_description\_

    """

    # 获取节点的权重函数

    def get\_weight(self):

        raise NotImplementedError(

            "抽象节点类未定义'get\_weight'")

    # 获取此节点是否是叶节点的函数

    def isleaf(self):

        raise NotImplementedError(

            "抽象节点类未定义 'isleaf'")

class LeafNode(HuffNode):

    """\_树叶节点类\_

    Args:

        HuffNode (\_type\_): \_description\_

    """

    # 初始化树节点,需要初始化的对象参数有节点值value及其出现的频率frequency

    def \_\_init\_\_(self, value=0, frequency=0,):

        super(LeafNode, self).\_\_init\_\_()

        # 节点的值

        self.value = value

        self.weight = frequency

    # 基类的方法,返回True,代表是叶节点

    def isleaf(self):

        return True

    # 基类的方法,返回对象属性 weight,表示对象的权重

    def get\_weight(self):

        return self.weight

    # 获取叶子节点的字符的值

    def get\_value(self):

        return self.value

class IntlNode(HuffNode):

    """\_中间节点类\_

    Args:

        HuffNode (\_type\_): \_description\_

    """

    # 初始化中间节点,需要初始化的对象参数有:左孩子、右孩子、节点权重

    def \_\_init\_\_(self, left\_child=None, right\_child=None):

        super(IntlNode, self).\_\_init\_\_()

        # 节点的值

        self.weight = left\_child.get\_weight() + right\_child.get\_weight()

        # 节点的左右子节点

        self.left\_child = left\_child

        self.right\_child = right\_child

    # 基类的方法,返回False,代表是中间节点

    def isleaf(self):

        return False

    # 基类的方法,返回对象属性 weight,表示对象的权重

    def get\_weight(self):

        return self.weight

    # 获取左孩子

    def get\_left(self):

        return self.left\_child

    # 获取右孩子

    def get\_right(self):

        return self.right\_child

class HuffTree(object):

    """\_霍夫曼编码树\_

    Args:

        object (\_type\_): \_description\_

    """

    # 初始化霍夫曼编码树,需要初始化的对象参数有:根节点标记、节点的值、节点权重、左孩子、右孩子

    def \_\_init\_\_(self, flag, value =0, frequency=0, left\_tree=None, right\_tree=None):

        super(HuffTree, self).\_\_init\_\_()

        if flag == 0:

            self.root = LeafNode(value, frequency)

        else:

            self.root = IntlNode(left\_tree.get\_root(), right\_tree.get\_root())

    # 获取霍夫曼编码树的根节点

    def get\_root(self):

        return self.root

    # 获取霍夫曼编码树的根节点的权重

    def get\_weight(self):

        return self.root.get\_weight()

    # 采用递归的方法遍历霍夫曼编码树,并以此方式将得到每个字符对应的霍夫曼编码保存在字典char\_frequency中

    def traverse\_huffman\_tree(self, root, code, char\_frequency):

        if root.isleaf():

            char\_frequency[root.get\_value()] = code

            print(("it = %c  and  frequency = %d  code = %s")%(chr(root.get\_value()),root.get\_weight(), code))

            return None

        else:

            self.traverse\_huffman\_tree(root.get\_left(), code+'0', char\_frequency)

            self.traverse\_huffman\_tree(root.get\_right(), code+'1', char\_frequency)

def Create\_HuffmanTree(list\_hufftrees):

    """\_构造huffman树\_

    Args:

        list\_hufftrees (\_type\_): \_description\_

    Returns:

        \_type\_: \_description\_

    """

    while len(list\_hufftrees) >1 :

        # 按照信源符号出现概率(weight)对霍夫曼树进行从小到大的排序

        list\_hufftrees.sort(key=lambda x: x.get\_weight())

        # 跳出weight的最小两个huffman编码树

        temp1 = list\_hufftrees[0]

        temp2 = list\_hufftrees[1]

        list\_hufftrees = list\_hufftrees[2:]

        #构造新的霍夫曼树

        new\_huffman\_tree = HuffTree(1, 0, 0, temp1, temp2)

        #将霍夫曼树存放入到数组中

        list\_hufftrees.append(new\_huffman\_tree)

    # 数组中最后剩下来的那棵树,就是构造的Huffman编码树

    return list\_hufftrees[0]

def compress(input\_file\_name:str, output\_file\_name:str):

    """\_压缩文件\_

    Args:

        input\_file\_name (str): \_被压缩的文件的地址\_

        output\_file\_name (str): \_压缩文件的存放地址\_

    """

    #以二进制的方式打开文件

    f = open(input\_file\_name,'rb')

    filedata = f.read()

    # 获取文件的字节总数

    filesize = f.tell()

    # 统计byte的取值[0-255]的每个值出现的频率,并保存在字典char\_frequency中

    char\_frequency = {}

    for x in range(filesize):

        temp = filedata[x]

        if temp in char\_frequency.keys():

            char\_frequency[temp] = char\_frequency[temp] + 1

        else:

            char\_frequency[temp] = 1

    # 输出统计结果

    for temp in char\_frequency.keys():

        print(temp,' : ',char\_frequency[temp])

    # 开始构造原始的霍夫曼编码树数组,用于构造Huffman编码树

    list\_hufftrees = []

    for x in char\_frequency.keys():

        # 使用霍夫曼编码树的构造函数,定义一棵只包含一个叶节点的霍夫曼编码树

        temp = HuffTree(0, x, char\_frequency[x], None, None)

        # 将其添加到数组 list\_hufftrees 当中

        list\_hufftrees.append(temp)

    # 保存叶节点的个数

    length = len(char\_frequency.keys())

    output = open(output\_file\_name, 'wb')

    # 一个int型的数有四个字节,所以将其分成四个字节写入到输出文件当中

    a4 = length&255

    length = length>>8

    a3 = length&255

    length = length>>8

    a2 = length&255

    length = length>>8

    a1 = length&255

    output.write(six.int2byte(a1))

    output.write(six.int2byte(a2))

    output.write(six.int2byte(a3))

    output.write(six.int2byte(a4))

    # 遍历字典char\_frequency

    for x in char\_frequency.keys():

        output.write(six.int2byte(x))

        temp = char\_frequency[x]

        # 同样出现的次数是int型,分成四个字节写入到压缩文件当中

        a4 = temp&255

        temp = temp>>8

        a3 = temp&255

        temp = temp>>8

        a2 = temp&255

        temp = temp>>8

        a1 = temp&255

        output.write(six.int2byte(a1))

        output.write(six.int2byte(a2))

        output.write(six.int2byte(a3))

        output.write(six.int2byte(a4))

    # 构造霍夫曼编码树,并且获取到每个字符对应的编码

    temp = Create\_HuffmanTree(list\_hufftrees)

    temp.traverse\_huffman\_tree(temp.get\_root(),'',char\_frequency)

    # 开始对文件进行压缩操作

    code = ''

    for i in range(filesize):

        key = filedata[i]

        code = code + char\_frequency[key]

        out = 0

        while len(code)>8:

            for x in range(8):

                out = out<<1

                if code[x] == '1':

                    out = out|1

            code = code[8:]

            output.write(six.int2byte(out))

            out = 0

    # 处理剩下来的不满8位的code

    output.write(six.int2byte(len(code)))

    out = 0

    for i in range(len(code)):

        out = out<<1

        if code[i]=='1':

            out = out|1

    for i in range(8-len(code)):

        out = out<<1

    # 把最后一位给写入到文件当中

    output.write(six.int2byte(out))

    # 关闭输出文件,文件压缩完毕

    output.close()

def decompress(input\_file\_name:str, output\_file\_name:str):

    """\_解压缩文件\_

    Args:

        input\_file\_name (str): \_压缩文件的地址\_

        output\_file\_name (str): \_解压缩文件的存放地址\_

    """

    # 读取文件

    f = open(input\_file\_name,'rb')

    filedata = f.read()

    # 获取文件的字节总数

    filesize = f.tell()

    #读取压缩文件中保存的树的叶节点的个数

    #一次读取4个字节,代表一个int型的值

    a1 = filedata[0]

    a2 = filedata[1]

    a3 = filedata[2]

    a4 = filedata[3]

    j = 0

    j = j|a1

    j = j<<8

    j = j|a2

    j = j<<8

    j = j|a3

    j = j<<8

    j = j|a4

    leaf\_node\_size = j

    # 读取压缩文件中保存的相信的原文件中[0-255]出现的频率的信息

    # 构造一个字典char\_frequency,重建霍夫曼编码树

    char\_frequency = {}

    for i in range(leaf\_node\_size):

        c = filedata[4+i\*5+0]

        # 同样的,出现的频率是int型的,读区四个字节来读取到正确的数值

        a1 = filedata[4+i\*5+1]

        a2 = filedata[4+i\*5+2]

        a3 = filedata[4+i\*5+3]

        a4 = filedata[4+i\*5+4]

        j = 0

        j = j|a1

        j = j<<8

        j = j|a2

        j = j<<8

        j = j|a3

        j = j<<8

        j = j|a4

        print(c, j)

        char\_frequency[c] = j

    # 重建霍夫曼编码树,与压缩文件中建立霍夫曼编码树的方法一致

    list\_hufftrees = []

    for x in char\_frequency.keys():

        temp = HuffTree(0, x, char\_frequency[x], None, None)

        list\_hufftrees.append(temp)

    temp = Create\_HuffmanTree(list\_hufftrees)

    temp.traverse\_huffman\_tree(temp.get\_root(),'',char\_frequency)

    # 使用重建的huffman编码树,对压缩文件进行解压缩

    output = open(output\_file\_name, 'wb')

    code = ''

    currnode = temp.get\_root()

    for x in range(leaf\_node\_size\*5+4,filesize):

        c = filedata[x]

        for i in range(8):

            if c&128:

                code = code +'1'

            else:

                code = code + '0'

            c = c<<1

        while len(code) > 24:

            if currnode.isleaf():

                tem\_byte = six.int2byte(currnode.get\_value())

                output.write(tem\_byte)

                currnode = temp.get\_root()

            if code[0] == '1':

                currnode = currnode.get\_right()

            else:

                currnode = currnode.get\_left()

            code = code[1:]

    # 处理最后的24位

    sub\_code = code[-16:-8]

    last\_length = 0

    for i in range(8):

        last\_length = last\_length<<1

        if sub\_code[i] == '1':

            last\_length = last\_length|1

    code = code[:-16] + code[-8:-8 + last\_length]

    while len(code) > 0:

            if currnode.isleaf():

                tem\_byte = six.int2byte(currnode.get\_value())

                output.write(tem\_byte)

                currnode = temp.get\_root()

            if code[0] == '1':

                currnode = currnode.get\_right()

            else:

                currnode = currnode.get\_left()

            code = code[1:]

    if currnode.isleaf():

        tem\_byte = six.int2byte(currnode.get\_value())

        output.write(tem\_byte)

        currnode = temp.get\_root()

    # 关闭文件,解压缩完毕

    output.close()

if \_\_name\_\_ == '\_\_main\_\_':

    compress("./lion.jpg","./lion")

    original\_size=os.path.getsize("./lion.jpg")

    Compressed\_size=os.path.getsize("./lion")

    decompress("./lion","./lion.png")

    print("图像压缩比:",'%.2f'%Compressed\_size/original\_size)

