# Huffman 编码实验报告

## 一、功能实现

### 1. 代码解释

实验采用 python 语言。主要分为 Huffman 树、文件读写以及编码译码三部分。具体解释主要在代码注释里。

（1）Huffman 树

首先定义 Huffman 结点类，如图 1 所示。

```python
class HuffmanNode(object):
    def __init__(self, value, key=None, symbol='', left_child=None, right_child=None):
        self.left_child = left_child
        self.right_child = right_child
        self.value = value
        self.key = key
        assert symbol == ''
        self.symbol = symbol

    def __eq__(self, other):
        return self.value == other.value

    def __gt__(self, other):
        return self.value > other.value

    def __lt__(self, other):
        return self.value < other.value
```

图 1 Huffman 结点类

然后进行 Huffman 树的创建，如图 2 所示。调用 queue 库，借助优先级队列实现对符号频率的排序。每一次选频率最小的两个值，作为二叉树的两个叶子节点，将和作为它们的父节点，这两个叶子节点不再参与比较，新的父节点参与比较

```python
def createTree(hist_dict: dict) -> HuffmanNode:
    # 借助优先级队列实现频率排序，取出和插入元素很方便
    q = PriorityQueue()
    # 根据频率字典构造哈夫曼节点并放入队列中
    for k, v in hist_dict.items():
        # 这里放入的都是之后哈夫曼树的叶子节点，key都是各自的元素
        q.put(HuffmanNode(value=v, key=k))
    # 判断条件，直到队列中只剩下一个根节点
    while q.qsize() > 1:
        # 取出两个最小的哈夫曼节点，队列中这两个节点就不在了
        l_freq, r_freq = q.get(), q.get()
        # 增加他们的父节点，父节点值为这两个哈夫曼节点的和，但是没有key值；左子节点是较小的，右子节点是较大的
        node = HuffmanNode(value=l_freq.value + r_freq.value, left_child=l_freq, right_child=r_freq)
        # 把构造的父节点放在队列中，继续排序和取放、构造其他的节点
        q.put(node)
    # 队列中只剩下根节点了，返回根节点
    return q.get()
```

图 2 创建 Huffman 树

最后是在编码过程中访问树并不断记录编码。左孩子记 0，右孩子记 1。

```
def walkTree(root_node: HuffmanNode, symbol=''):
    # 为了不增加变量复制的成本，直接使用一个dict类型的全局变量保存每个元素对应的哈夫曼编码
    global Huffman_encode_dict
    # 判断节点是不是HuffmanNode，因为叶子节点的子节点是None
    if isinstance(root_node, HuffmanNode):
        # 编码操作，改变每个子树的根节点的哈夫曼编码，根据遍历过程是逐渐增加编码长度到完整的
        root_node.symbol += symbol
        # 判断是否走到了叶子节点，叶子节点的key!=None
        if root_node.key != None:
            # 记录叶子节点的编码到全局的dict中
            Huffman_encode_dict[root_node.key] = root_node.symbol
        # 访问左子树，左子树在此根节点基础上赋值'0'
        walkTree(root_node.left_child, symbol=root_node.symbol + '0')
        # 访问右子树，右子树在此根节点基础上赋值'1'
        walkTree(root_node.right_child, symbol=root_node.symbol + '1')
    return
```

图 3 填充 Huffman 树

（2）文件读写

文件读写都采用二进制字节流形式。根据实验要求，写文件（即生成压缩文件）时将 Huffman 字典也拆开一并写入。采用 struct.pack 打包不同类型数据。

```
def writeBinFile(file_encode: str, huffman_file_name: str):
    # 以huf后缀标志该程序编码文件
    with open(huffman_file_name + '.huf', 'wb') as f:
        # 存编码字符串长度,这里只用了4字节存，编码长度不可超过2^32
        file_encode_len = len(file_encode)
        file_encode_len_bin = struct.pack('i', file_encode_len)
        f.write(file_encode_len_bin)
        # 用4字节存字典长度
        dict_len = len(Huffman_encode_dict)
        dict_len_bin = struct.pack('i', dict_len)
        f.write(dict_len_bin)
        # 写入字典
        k = Huffman_encode_dict.keys()
        v = Huffman_encode_dict.values()
        l = []  # 存编码长度
        int_v = []  # 存十进制编码
        for val in v:
            l.append(len(val))
            int_v.append(int(val, 2))
```

```
# 存键
for key in k:
    key_bin = struct.pack('B', key)
    f.write(key_bin)
# 存编码长度
for lenth in l:
    len_bin = struct.pack('B', lenth)
    f.write(len_bin)
# 存编码，用4字节打包编码，防止不够
for i in int_v:
    i_bin = struct.pack('i', i)
    f.write(i_bin)

# 每8个bit组成一个byte。反正存十进制，这里最后不用补位。
for i in range(0, len(file_encode), 8):
    # 把这一个字节的数据根据二进制翻译为十进制的数字
    file_encode_dec = int(file_encode[i:i + 8], 2)
    # 把这一个字节的十进制数据打包为一个unsigned char，大端（可省略）
    file_encode_bin = struct.pack('>B', file_encode_dec)
    # 写入这一个字节数据
    f.write(file_encode_bin)
```

图 4 写二进制流文件

如图 5，该函数用于读取编码文件中的二进制数据。根据写文件时打包格式与顺序解包，注意单个字节型解出来是十进制数，转二进制要注意补 0 到 8 位。最后一字节要去掉高位多余 0，因为写的时候也没补最后一字节。

```
def readBinFile(huffman_file_path: str):
    code_bin_str = ""
    # 读二进制数据
    with open(huffman_file_path, 'rb') as f:
        # 读取编码长度、字典长度
        file_encode_len = struct.unpack('i', f.read(4))[0]
        dict_len = struct.unpack('i', f.read(4))[0]
        # 跳到编码内容
        f.seek(8 + 6 * dict_len, 0)
        content = f.read()
        # 从二进制数据解包到十进制数据，所有数据组成的是tuple
        code_dec_tuple = struct.unpack('>' + 'B' * len(content), content)
        for code_dec in code_dec_tuple:
            # 通过bin把解压的十进制数据翻译为二进制的字符串，并填充为8位，否则会丢失高位的0
            code_bin_str += bin(code_dec)[2:].zfill(8)
        # 计算读取的编码字符串与原始编码字符串长度的差
        len_diff = len(code_bin_str) - file_encode_len
        # 在读取的编码字符串最后8位去掉高位的多余的0，因为写进去的时候也没补
        code_bin_str = code_bin_str[:-8] + code_bin_str[-(8 - len_diff):]
    return code_bin_str
```

图 5 读二进制流文件

（3）编码译码

编码时将原文件每个字符在编码字典中查找，获得编码字符串。

```
def encodeFile(src_file: bytes, encode_dict: dict):
    file_encode = ""
    for i in src_file:
        file_encode += encode_dict[i]
    return file_encode
```

图 6 编码原文件

译码稍麻烦，先将编码字典颠倒过来，获得键为 01 编码，值为字符的译码字典。按 01 串长度对比输入的二进制串看是否相同。由于 Huffman 是前缀码，只要一段编码与译码字典中某键长度相同且相等，则可确定该段编码为该键对应符号编码而来。返回字节流方便再写入译码文件。

```
def decodeFile(file_encode: str, encode_dict: dict):
    file_src_val = []
    decode_dict = {}
    # 构造一个key-value互换的字典
    for k, v in encode_dict.items():
        decode_dict[v] = k
    # s用来记录当前字符串的访问位置，相当于一个指针
    s = 0
    # 只要没有访问到最后
    while len(file_encode) > s:
        # 遍历字典中每一个键code
        for k in decode_dict.keys():
            if k == file_encode[s:s + len(k)]:
                file_src_val.append(decode_dict[k])
                # 指针移动k个单位
                s += len(k)
                # 如果已经找到了相应的编码了，就可以找下一个了
                break
    return bytearray(file_src_val)
```

图 7 译码二进制数据

真正实现完全编码功能的函数为 encode 和 decode，集成上述函数功能，且涉及用户交互和鲁棒性处理。encode 包括频率字典生成，构造 Huffman 树，遍历 Huffman 树，

原文件编码，写二进制流文件以及一些结果计算展示。译码函数 decode 需要读取字典各部分信息并恢复，恢复二进制流数据，译码二进制数据并恢复原文件。

**2. 运行测试截图**

本次实验实现了使用 Huffman 码对任意文件进行编码译码。下面是对 testpic.bmp 进行编码译码的测试截图。本次实验所有原文件都在 testfiles 文件夹中。

编码结果可在“中间文件”文件夹中查看 testpic_encode.huf。

```
请输入模式：（1 编码；0 译码） 1
请输入待处理文件路径：H:\THINGS\study\专业课\信息论与编码\实验\Huffman\testfiles\testpic.bmp
请输入编码后文件名：testpic_encode
哈夫曼编码字典： {79: '0000000', 137: '00000010', 198: '00000011', 125: '0000010', 159: '0000011', 
平均编码长度为：7.952
编码效率为：0.996429
原文件大小 3067.6376953125 KB  压缩后大小(带字典) 3050.7662353515625 KB  压缩率 0.5499821568472063 %
```

图 8 编码测试

译码结果可在“结果文件”文件夹中查看 testpic_decode.bmp。

```
请输入模式：（1 编码；0 译码） 0
请输入待处理文件路径：H:\THINGS\study\专业课\信息论与编码\实验\Huffman\testpic_encode.huf
请输入译码后文件名（包括后缀）：testpic_decode.bmp
哈夫曼字典： {79: '0000000', 137: '00000010', 198: '00000011', 125: '0000010', 159: '0000011',
译码完成，请查看testpic_decode.bmp
```

图 9 译码测试

如图 10，编码前、译码后文件的 SHA256 值均为

9c7d41230dd8d84996cdd33a62ef76e2be845394923f18dc64a1f22727ee1e39

```
C:\Users\Administrator>certutil -hashfile H:\THINGS\study\专业课\信息论与编码\实验\Huffman\testfiles\testpic.bmp SHA256
SHA256 的 H:\THINGS\study\专业课\信息论与编码\实验\Huffman\testfiles\testpic.bmp 哈希:
9c7d41230dd8d84996cdd33a62ef76e2be845394923f18dc64a1f22727ee1e39
CertUtil: -hashfile 命令成功完成。

C:\Users\Administrator>certutil -hashfile H:\THINGS\study\专业课\信息论与编码\实验\Huffman\testpic_decode.bmp SHA256
SHA256 的 H:\THINGS\study\专业课\信息论与编码\实验\Huffman\testpic_decode.bmp 哈希:
9c7d41230dd8d84996cdd33a62ef76e2be845394923f18dc64a1f22727ee1e39
CertUtil: -hashfile 命令成功完成。
```

图 10 SHA256 值相同

（另外，这张图本来是 jpg 格式，我直接将后缀改成了 bmp，由结果来看改这个后缀没什么用处。）

## 二、代码风格与用户交互

代码风格上，本代码有清楚的变量命名、恰当的注释和合理的结构设计。变量命名均采用下划线分隔单词，可顾名思义。定义函数时均已写明相关参数类型，便于查看修改。重要功能语句均有注释解释，而且注释了一些调试用 print 代码，需要的话可取消注释查看中间结果。先定义类，再定义相关函数，按 Huffman 树、二进制流文件读写、

编码译码顺序，结构清晰明了。

如图 8、9 所示，本代码设计了 CLI 终端交互。可以输入 1 或 0 选择编码或译码模式。用户需要输入待处理的文件路径，可自定义文件名。译码模式时用户输入文件名时需要加上后缀便于恢复原文件。编码和译码时都会显示 Huffman 编码字典。编码时还会显示平均编码长度、编码效率、原文件大小、压缩后大小和压缩率。交互代码如图 11 所示。

```
if __name__ == '__main__':
    Huffman_encode_dict = {}
    m = int(input('请输入模式：（1 编码；0 译码） '))
    path = input('请输入待处理文件路径：')
    if os.path.getsize(path):
        if m == 1:
            file_name = input('请输入编码后文件名：')
            encode(path, file_name)
        else:
            file_name = input('请输入译码后文件名（包括后缀）：')
            decode(path, file_name)
            print('译码完成，请查看' + file_name)
    else:
        print('输入文件为空，请重新输入路径')
```

图 11 交互部分代码

## 三、代码鲁棒性与安全性

本程序考虑了一些边界情况和错误。

空文件无需编译码。先通过 os.path.getsize()获取文件大小，为空则不进行编码译码，直接返回错误提示，如图 12 所示。

```
请输入模式：（1 编码；0 译码） 1
请输入待处理文件路径：H:\THINGS\study\专业课\信息论与编码\实验\Huffman\testfiles\empty_test.txt
输入文件为空，请重新输入路径
```

图 12 空文件测试

若文件仅含有一种字节，则 Huffman 字典编码为空，无法编码译码。若频率字典长度为 1，说明文件只有一种字节。如图 13 所示，此时直接令该字节编码为‘0’，将文件中所有字节都变成‘0’，不进行正常 Huffman 编码译码操作。

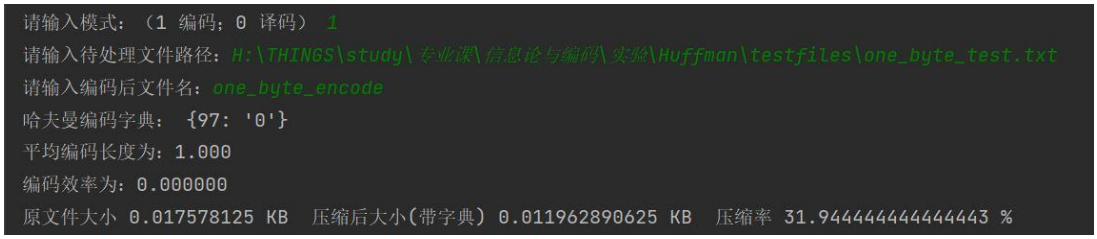


图 13 单字节测试

本程序不能解译非本程序编码的文件。为标识本程序产生的编码文件，一种方法是在写文件时一起写入标识，但这样做会使文件变大，压缩效率变低。因此本程序采取特定后缀的方式标识编码文件。编码时只要输入文件名而不要后缀，是因为本程序会自动使编码文件后缀为.huf。如图 14 所示，译码时传入文件路径最后后缀不是.huf，则可确定文件不能被本程序译码，返回错误提示。


图 14 文件标识测试

## 四、实践问题

### 1. 重复性的文件结构

编码 testfile1 结果如图 15 所示。由图可见原文件大小为 64KB，压缩后反而变成了 65.5KB。原来的文件结构就是所有符号等概，编码完平均码长还是 8，和原来没有变化，对于该文件 Huffman 编码不能起到压缩作用。由于还要往编码文件里写入字典等信息，反而使编码后文件更大了。


图 15 testfile1 编码结果

分别以 2、4、8、13、128 字节为单位对 testfile2 编码，结果如图 16-20 所示。

```
请输入模式：（1 编码；0 译码） 1
请输入以多少字节为一组进行编/译码：2
请输入待处理文件路径：H:\THINGS\study\专业课\信息论与编码\实验\Huffman\testfiles\testfile2
请输入编码后文件名：2
哈夫曼编码字典： {b'\x00\x00': '0', b'\xff\xff': '1'}
编码结果：0000000011111111000000001111111100000000111111110000000011111111000000001111
平均编码长度为：1.000
编码效率为：1.000000
原文件大小 8.0 KB  压缩后大小(带字典) 0.5234375 KB  压缩率 93.45703125 %
```

图 16　2 字节编码结果

```
请输入模式：（1 编码；0 译码） 1
请输入以多少字节为一组进行编/译码：4
请输入待处理文件路径：H:\THINGS\study\专业课\信息论与编码\实验\Huffman\testfiles\testfile2
请输入编码后文件名：4
哈夫曼编码字典： {b'\x00\x00\x00\x00': '0', b'\xff\xff\xff\xff': '1'}
编码结果：00001111000011110000111100001111000011110000111100001111000011110000111100001111000
平均编码长度为：1.000
编码效率为：1.000000
原文件大小 8.0 KB  压缩后大小(带字典) 0.2734375 KB  压缩率 96.58203125 %
```

图 17　4 字节编码结果

```
请输入模式：（1 编码；0 译码） 1
请输入以多少字节为一组进行编/译码：8
请输入待处理文件路径：H:\THINGS\study\专业课\信息论与编码\实验\Huffman\testfiles\testfile2
请输入编码后文件名：8
哈夫曼编码字典： {b'\x00\x00\x00\x00\x00\x00\x00\x00': '0', b'\xff\xff\xff\xff\xff\xff\xff\xff': '1'}
编码结果：0011001100110011001100110011001100110011001100110011001100110011001100110011001100110011
平均编码长度为：1.000
编码效率为：1.000000
原文件大小 8.0 KB  压缩后大小(带字典) 0.1484375 KB  压缩率 98.14453125 %
```

图 18　8 字节编码结果

```
请输入模式：（1 编码；0 译码） 1
请输入以多少字节为一组进行编/译码：13
请输入待处理文件路径：H:\THINGS\study\专业课\信息论与编码\实验\Huffman\testfiles\testfile2
请输入编码后文件名：13
哈夫曼编码字典： {b'\x00\x00\x00\x00\x00\xff\xff\xff\xff\xff\xff\xff\xff': '00000', b'\x00\x00\
编码结果：0111000011011110001010001111010001111001111100011101011011010001111110110101111011110
平均编码长度为：4.499
编码效率为：0.999929
原文件大小 8.0 KB  压缩后大小(带字典) 0.5106201171875 KB  压缩率 93.61724853515625 %
```

图 19　13 字节编码结果


图 20　128 字节编码结果

这里压缩率的定义是编码前后文件大小变化除以原文件大小。由以上结果可见，以多字节为单位字符对 testfile2 编码可大大提高文件压缩率，这是因为 testfile2 有高度连续重复结构，多字节将被编码成更小的字符串。进一步，当我们以 2 的幂次个字节为单位时，编码效率能达到 100%，即平均码长和熵相等。非 2 的幂次时，如 13 字节反而压缩率比 8 字节低，是 testfile2 以 32 为周期重复内容的特殊结构导致的。

以 2 字节为单位已经有很高的压缩率了，经试验 2、4、8、16 字节压缩率依次上升。16 字节最大，压缩率达 98.9%。但这也不意味着单位字符越大越好，当以 128 字节为单位时，压缩率大幅降低，连 2 字节都不如了。这是因为我们需要往编码文件里存字典等信息，单位字符越大，Huffman 字典越大，占用了很多空间。经试验，32 字节再继续增大 2 的幂次，压缩率将保持 87%，因为之后都是只有 1 种字符的情况。这启示我们单位字符的大小选取最好是原文件重复内容周期的最大因数。

**2. 不同格式的压缩**

（1）图片压缩

bmp 格式压缩文件结果如图 21 所示。


图 21　bmp 压缩结果

同一张图 jpeg 格式压缩结果如图 22 所示。

```
请输入模式：（1 编码；0 译码） 1
请输入待处理文件路径：H:\THINGS\study\专业课\信息论与编码\实验\Huffman\testfiles\jpeg111.jpg
请输入编码后文件名：jpeg222
哈夫曼编码字典： {161: '00000000', 173: '00000001', 148: '00000010', 156: '00000011', 22: '00000100',
平均编码长度为：7.614
编码效率为：0.995641
原文件大小 51.244140625 KB  压缩后大小(带字典) 50.27880859375 KB  压缩率 1.8837900674619767 %
```

图 22　jpeg 压缩结果

由结果可见 bmp 格式压缩率很高，而 jpg 格式压缩率极低。压缩前 bmp 文件就比 jpg 文件大，因为 bmp 是未压缩色域的格式，jpg 在保存时已经压缩过了，部分像素值会有较小波动。测试文件有大量白色，因此 bmp 压缩时白色对应字符编码长度就会最短，从而大大压缩文件。而 jpg 由于本身的失真，即使同样是白色可能对应字符也不同，破坏了字符重复结构，因此压缩率很低。

（2）不同格式文件压缩

```
请输入模式：（1 编码；0 译码） 1
请输入待处理文件路径：H:\THINGS\others\software\Youdao\Dict\YoudaoDict.exe
请输入编码后文件名：exe
哈夫曼编码字典： {32: '000000', 119: '000001000', 231: '0000010010', 45: '0000010011', 208: '00000101',
平均编码长度为：6.584
编码效率为：0.993803
原文件大小 9706.5078125 KB  压缩后大小(带字典) 7990.4349365234375 KB  压缩率 17.67961154645764 %
```

图 24　exe 压缩结果

```
请输入模式：（1 编码；0 译码） 1
请输入待处理文件路径：H:\THINGS\study\专业课\信息论与编码\实验\Huffman\testfiles\png_test.PNG
请输入编码后文件名：png_encode
哈夫曼编码字典： {140: '0000000', 241: '0000001', 5: '00000100', 10: '00000101', 227: '0000011', 252: '0000100', 33: '00001010', 165: '00001011', 242:
平均编码长度为：7.997
编码效率为：0.996777
原文件大小 71.947265625 KB  压缩后大小(带字典) 73.4246826171875 KB  压缩率 -2.0534720525558603 %
```

图 24　png 压缩结果

```
请输入模式：（1 编码；0 译码） 1
请输入待处理文件路径：H:\THINGS\study\专业课\信息论与编码\实验\Huffman\testfiles\English_test.txt
请输入编码后文件名：English_encode
哈夫曼编码字典： {114: '000', 33: '001', 108: '01', 111: '100', 101: '1010', 100: '1011', 32: '110', 104: '1110', 119: '1111'}
平均编码长度为：3.077
编码效率为：0.983771
原文件大小 0.0126953125 KB  压缩后大小(带字典) 0.0615234375 KB  压缩率 -384.6153846153846 %
```

图 25　英文 txt 压缩结果

```
请输入模式：（1 编码；0 译码） 1
请输入待处理文件路径：H:\THINGS\study\专业课\信息论与编码\实验\Huffman\testfiles\Chinese_test.txt
请输入编码后文件名：Chinese_encode
哈夫曼编码字典： {152: '0000', 130: '0001', 133: '0010', 142: '0011', 141: '0100', 136: '0101', 128: '0110', 227: '0111', 230: '100', 137: '101', 156:
平均编码长度为：3.722
编码效率为：0.989608
原文件大小 0.017578125 KB  压缩后大小(带字典) 0.0941162109375 KB  压缩率 -435.4166666666667 %
```

图 26　中文 txt 压缩结果

```
请输入模式：（1 编码；0 译码） 1
请输入待处理文件路径：H:\THINGS\study\专业课\信息论与编码\实验\Huffman\信息论与编码 - 信源编码实验指导(2).pdf
请输入编码后文件名：pdf_encode
哈夫曼编码字典： {8: '00000000', 132: '00000001', 128: '00000010', 5: '00000011', 9: '00000100', 4: '00000101', 130: '00000110', 38: '00000111', 17: '0
平均编码长度为：7.995
编码效率为：0.998855
原文件大小 573.1064453125 KB  压缩后大小(带字典) 574.2578125 KB  压缩率 -0.20089936117750895 %
```

图 27　pdf 压缩结果

```
请输入模式：（1 编码；0 译码） 1
请输入待处理文件路径：H:\THINGS\study\专业课\信息论与编码\实验\Huffman\testfiles\lab1_data.rar
请输入编码后文件名：rar
哈夫曼编码字典： {99: '0000000', 75: '0000001', 130: '0000010', 171: '0000011', 194: '0000100', 88: '0000101',
平均编码长度为：7.745
编码效率为：0.996436
原文件大小 1.5478515625 KB  压缩后大小(带字典) 2.98876953125 KB  压缩率 -93.09148264984226 %
```

图 28　rar 压缩结果

### 3. 黑洞

对上一问中的 bmp 文件压缩 10 次，结果如图 29 所示。

| | | | |
|---|---|---|---|
| 1.huf | 2022/4/30 17:14 | HUF 文件 | 495 KB |
| 2.huf | 2022/4/30 17:14 | HUF 文件 | 309 KB |
| 3.huf | 2022/4/30 17:15 | HUF 文件 | 279 KB |
| 4.huf | 2022/4/30 17:15 | HUF 文件 | 277 KB |
| 5.huf | 2022/4/30 17:16 | HUF 文件 | 279 KB |
| 6.huf | 2022/4/30 17:16 | HUF 文件 | 280 KB |
| 7.huf | 2022/4/30 17:16 | HUF 文件 | 282 KB |
| 8.huf | 2022/4/30 17:17 | HUF 文件 | 283 KB |
| 9.huf | 2022/4/30 17:17 | HUF 文件 | 285 KB |
| 10.huf | 2022/4/30 17:17 | HUF 文件 | 286 KB |
| bmp111.bmp | 2022/4/30 16:49 | BMP 图片文件 | 1,843 KB |

图 29　bmp 压缩 10 次结果

```
请输入模式：（1 编码；0 译码） 1
请输入待处理文件路径：H:\THINGS\study\专业课\信息论与编码\实验\Huffman\9.huf
请输入编码后文件名：10
哈夫曼编码字典： {1: '00000000', 2: '00000001', 3: '00000010', 4: '00000011', 5: '00000100',
平均编码长度为：8.000
编码效率为：0.998654
原文件大小 284.228515625 KB  压缩后大小(带字典) 285.736328125 KB  压缩率 -0.5304930424325782 %
```

图 30　bmp 压缩第 10 次结果

由图可见，压缩到第 4 次时文件最小，之后开始慢慢增大，文件并不会越压缩越小。这是因为信源熵是平均码长的下界，第四次及以后编码效率都接近 1，已经不能再继续压缩下去了。至于后来越压越大，是因为每次编码还要存字典，之后每次基本都是-0.5%的压缩率。