

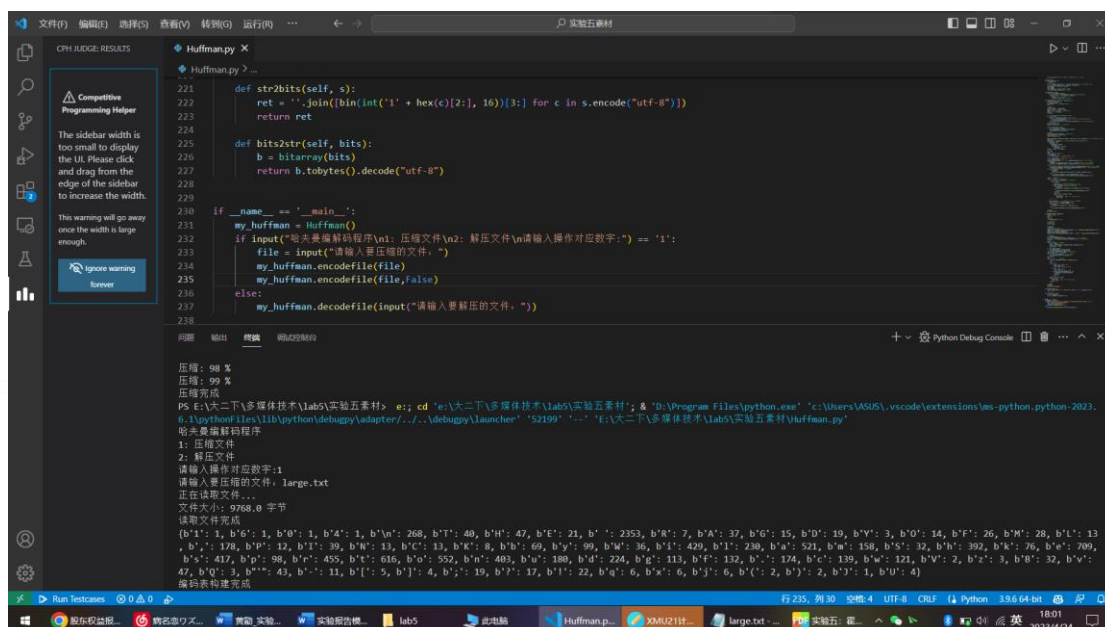
《多媒体技术》实验报告 5

黄勳 22920212204392

1. 运行程序截图和简要说明

编写 Python 程序，实现利用霍夫曼编码对文本文件的压缩。压缩后的二进制代码可以字符串的形式输出并保存。

运行：



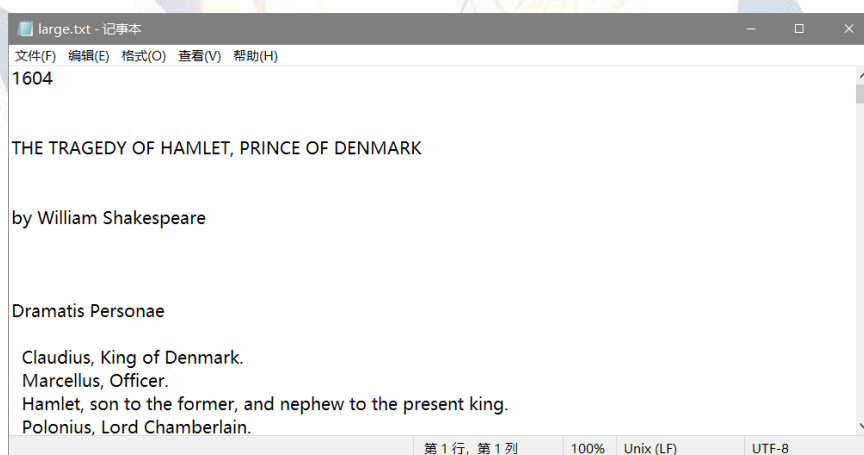
```
def str2bits(self, s):
    ret = ''
    for c in s.encode("utf-8"):
        ret += bin(int('1' + hex(c)[2:], 16))[3:]
    return ret

def bits2str(self, bits):
    b = bytearray(bits)
    return b.decode("utf-8")

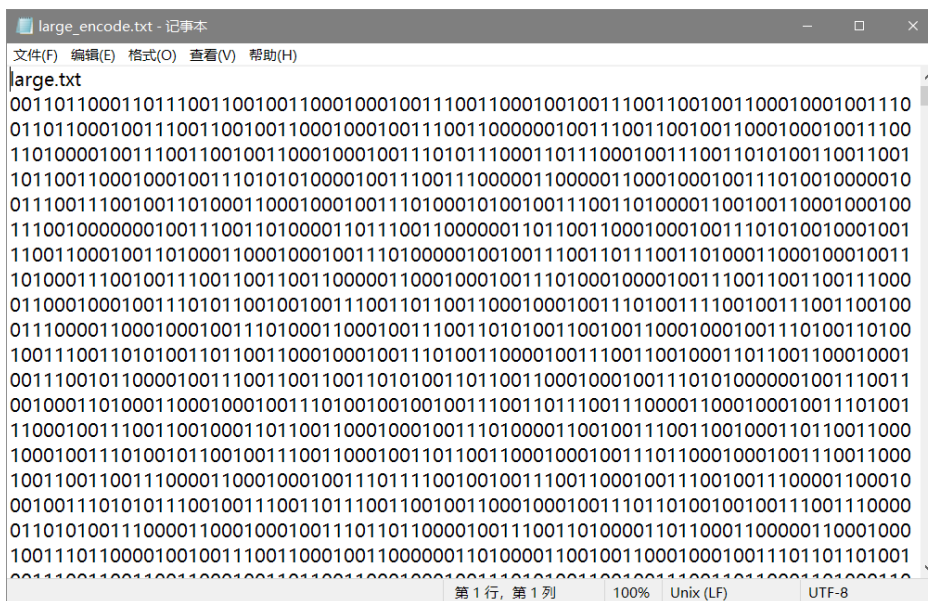
if __name__ == '__main__':
    my_huffman = Huffman()
    if input("哈夫曼编码程序\n1: 压缩文件\n2: 解压文件\n请输入操作对应数字:") == '1':
        file = input("请输入要压缩的文件:")
        my_huffman.encodefile(file)
    else:
        my_huffman.decodefile(input("请输入要解压的文件:"))
```

压缩: 98 %
解压: 99 %
压缩完成
PS E:\大二下\多媒体技术\lab5\实验五素材> .\ Huffman.py
哈夫曼编码程序
1: 压缩文件
2: 解压文件
请输入操作对应数字:1
请输入要压缩的文件: large.txt
正在读取文件...
文件大小: 97688.0 字节
读取文件完成
[b'1': 1, b'6': 1, b'0': 1, b'4': 1, b'v': 268, b't': 40, b'h': 47, b'e': 21, b' ': 2353, b'r': 7, b'a': 37, b'g': 35, b'd': 19, b'y': 3, b'o': 14, b'f': 26, b'm': 28, b'l': 13, b',': 178, b'p': 12, b'i': 39, b'n': 13, b'c': 13, b'k': 8, b'b': 69, b'j': 99, b'u': 36, b'1': 429, b'2': 230, b'3': 521, b'4': 158, b'5': 32, b'6': 392, b'7': 76, b'8': 709, b'9': 417, b'p': 98, b'e': 455, b't': 616, b'o': 552, b'n': 483, b'u': 180, b'd': 224, b'g': 113, b'f': 132, b' ': 174, b'c': 139, b'm': 121, b'v': 2, b'z': 3, b'8': 32, b'v': 47, b'2': 3, b'3': 43, b'1': 11, b'4': 5, b'j': 4, b'5': 19, b'7': 17, b'1': 22, b'g': 6, b'k': 6, b'j': 6, b'1': 2, b'3': 2, b'3': 1, b'0': 4]

1) 压缩前：



2) 压缩后文件：



3) 压缩后文件结果分析:

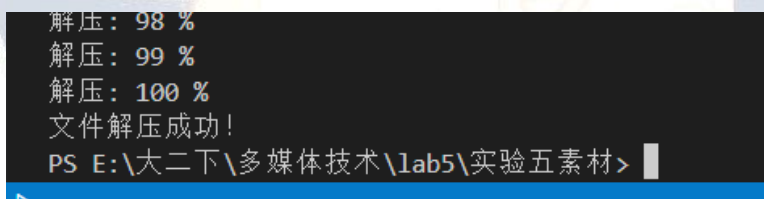
large.txt	2014/2/25 3:24	文本文档	10 KB
large_encode.huffman	2023/4/24 18:00	HUFFMAN 文件	6 KB
large_encode.txt	2023/4/24 18:00	文本文档	114 KB

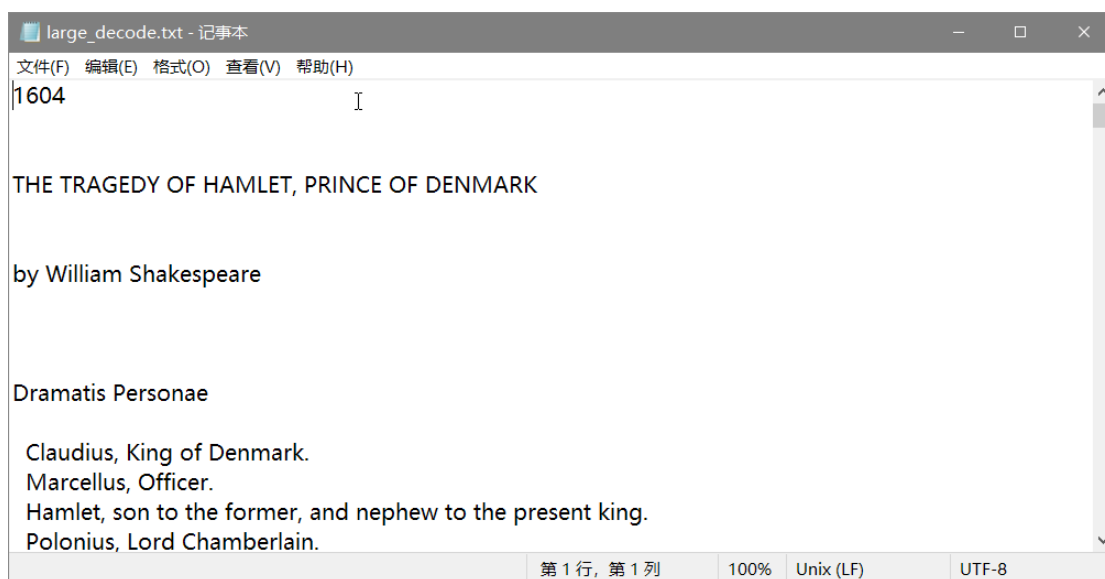
Large.txt 文件为源文件

.huffman 文件为二进制输出的文件, 可以看出实际的压缩率十分可观。

.encode.txt 文件是以字符串输出的二进制文件

4) 压缩后的数据还以二进制的形式输出了文件, 并添加了解压功能, 可以恢复原来的文件。

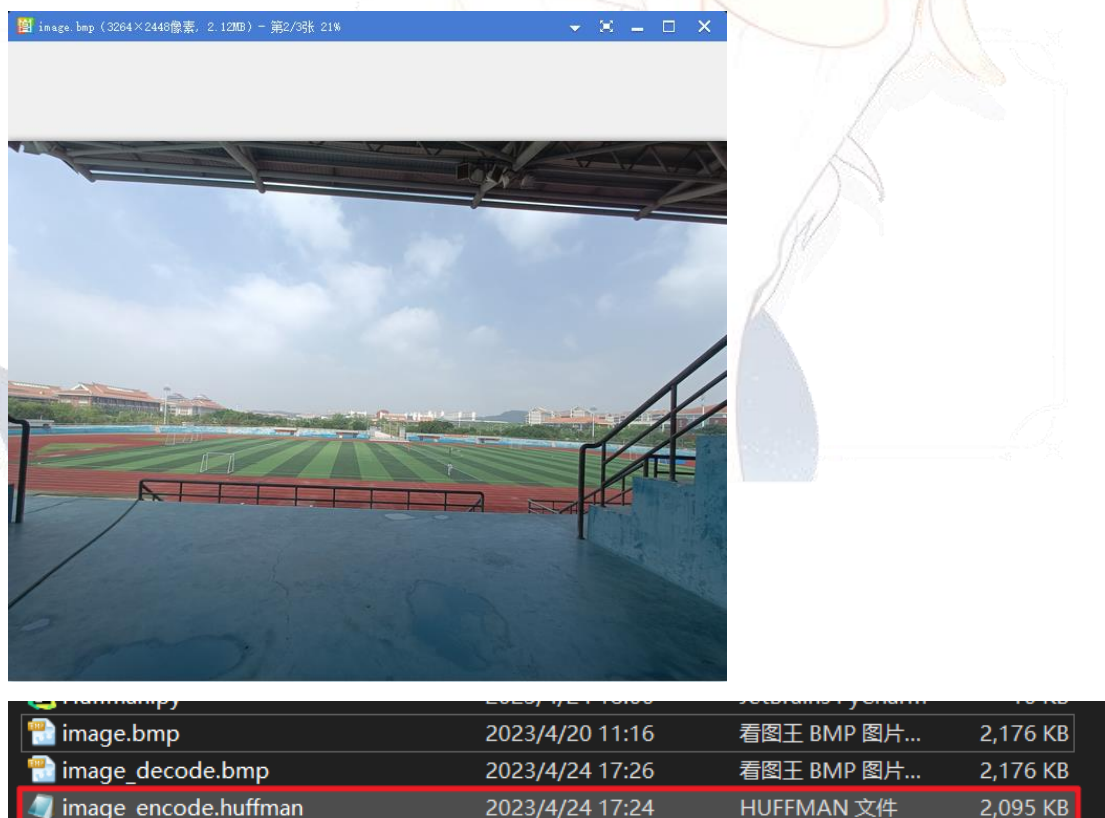




Decode 文件即恢复的源文件，大小与原来一致。

5) 还可以压缩其他的文件

这里以图片为例：



由于图片已经采用了一部分压缩算法，所以最后文件大小缩减并没有很多。

2.主要代码展示和分析

1) 定义节点

```
9  # 定义哈夫曼树的节点类
10 class node(object):
11
12     def __init__(self, value=None, left=None, right=None, father=None):
13         self.value = value # 权值
14         self.left = left # 左子节点
15         self.right = right # 右子节点
16         self.father = father # 父节点
17
18     def build_father(left, right):
19         n = node(value=left.value + right.value, left=left, right=right) # 构建父节点
20         left.father = right.father = n # 子节点指向父节点
21         return n
22
23     def encode(n):
24         if n.father == None:
25             return b''
26         if n.father.left == n: # 递归编码, 左子节点编码为0, 右子节点编码为1
27             return node.encode(n.father) + b'0'
28         else:
29             return node.encode(n.father) + b'1'
```

2) 哈夫曼树构建

```
41 # 哈夫曼树构建
42 def build_tree(self, l):
43     if len(l) == 1:
44         return l
45     sorts = sorted(l, key=lambda x: x.value, reverse=False) # 对所有根节点进行排序
46     n = node.build_father(sorts[0], sorts[1]) # 构建父节点
47     sorts.pop(0) # 删除已经构建过的节点
48     sorts.pop(0)
49     sorts.append(n) # 将新构建的节点加入列表
50     return self.build_tree(sorts) # 递归构建
```

3) 编码并输出

```
59 def encodefile(self, inputfile, flag_bytes=True):
60     print("正在读取文件...")
61     f = open(inputfile, "rb")
62     bytes_width = 1 # 每次读取的字节宽度
63     i = 0
64
65     f.seek(0, 2)
66     count = f.tell() / bytes_width
67     print("文件大小:", count, "字节")
68     nodes = [] # 结点列表, 用于构建哈夫曼树
69     buff = [b''] * int(count)
70     f.seek(0)
71
72     # 计算字符频率, 并将单个字符构建成为单一节点
73     while i < count:
74         buff[i] = f.read(bytes_width) # 读取一个字节
75         if self.count_dict.get(buff[i], -1) == -1: # 如果字典中没有该字符, 则添加
76             self.count_dict[buff[i]] = 0
77         self.count_dict[buff[i]] = self.count_dict[buff[i]] + 1 # 字典中该字符的权值加1
78         i = i + 1
79     print("读取文件完成")
80     print(self.count_dict) # 输出权值字典
```

```

82     # 将单个字符构建成一节点
83     for x in self.count_dict.keys():
84         self.node_dict[x] = node(self.count_dict[x])
85         nodes.append(self.node_dict[x])
86
87     f.close()
88     tree = self.build_tree(nodes) # 哈夫曼树构建
89     self.encode(False) # 构建编码表
90     print("编码表构建完成")
91
92     head = sorted(self.count_dict.items(), key=lambda x: x[1], reverse=True) # 对所有根节点进行排序
93     bit_width = 1

```

```

111     o.write((name[len(name) - 1] + '\n').encode(encoding="utf-8")) # 写出原文件名
112     if flag_bytes:
113         o.write(int.to_bytes(len(self.ec_dict), 2, byteorder='big')) # 写出结点数量
114         o.write(int.to_bytes(bit_width, 1, byteorder='big')) # 写出编码表字节宽度
115     else:
116         o.write(self.str2bits(str(len(self.ec_dict))).encode(encoding="utf-8")) # 写出结点数量
117         o.write(self.str2bits(str(bit_width)).encode(encoding="utf-8")) # 写出编码表字节宽度
118     for x in self.ec_dict.keys(): # 编码文件头
119         if flag_bytes:
120             o.write(x)
121             o.write(int.to_bytes(self.count_dict[x], bit_width, byteorder='big'))
122         else:
123             o.write(self.str2bits(str(x)).encode(encoding="utf-8"))
124             o.write(self.str2bits(str(self.count_dict[x])).encode(encoding="utf-8"))
125
126     print('开始压缩数据...')
127     while i < count: # 开始压缩数据
128         for x in self.ec_dict[buff[i]]: # 读取编码表
129             raw = raw << 1
130             if x == 49:
131                 raw = raw | 1 # 将编码表中的字符转换为二进制
132             if raw.bit_length() == 9: # 如果已经读取了一个字节
133                 raw = raw & ~(1 << 8) # 去掉最高位的1

```

```

135         o.write(int.to_bytes(raw, 1, byteorder='big')) # 写入文件
136     else:
137         o.write(self.str2bits(str(raw)).encode(encoding="utf-8"))
138     o.flush()
139     raw = 0b1
140     tem = int(i / len(buff) * 100)
141     if tem > last:
142         print("压缩:", tem, '%') # 输出压缩进度
143         last = tem
144     i = i + 1
145
146     if raw.bit_length() > 1: # 处理文件尾部不足一个字节的数
147         raw = raw << (8 - (raw.bit_length() - 1))
148         raw = raw & ~(1 << raw.bit_length() - 1)
149         if flag_bytes:
150             o.write(int.to_bytes(raw, 1, byteorder='big'))
151         else:
152             o.write(self.str2bits(str(raw)).encode(encoding="utf-8"))
153
154     o.close()
155     print("压缩完成")

```

4) 解压数据

```
157     def decodefile(self, inputfile):
158         print("开始解压...")
159         count = 0
160         raw = 0
161         last = 0
162         f = open(inputfile, 'rb')
163
164         # 获取文件大小
165         f.seek(0, 2)
166         eof = f.tell()
167
168         # 获取文件名
169         f.seek(0)
170         name = inputfile.split('/')
171         outputfile = inputfile.replace(name[len(name) - 1], f.readline().decode(encoding="utf-8"))
172         outputfile = outputfile.replace('\n', '')
173         temp = os.path.splitext(outputfile)
174         outputfile = os.path.splitext(outputfile)[0] + "_decode" + os.path.splitext(outputfile)[1]
175
176         # 读取文件头
177         o = open(outputfile, 'wb')
178         count = int.from_bytes(f.read(2), byteorder='big') # 取出结点数量
179         bit_width = int.from_bytes(f.read(1), byteorder='big') # 取出编码表字宽
180         i = 0
181         de_dict = {}
```

```
182         while i < count: # 解析文件头
183             key = f.read(1)
184             value = int.from_bytes(f.read(bit_width), byteorder='big')
185             de_dict[key] = value
186             i = i + 1
187         for x in de_dict.keys():
188             self.node_dict[x] = node(de_dict[x])
189             self.nodes.append(self.node_dict[x])
190         tree = self.build_tree(self.nodes) # 重建哈夫曼树
191         self.encode(False) # 建立编码表
192         for x in self.ec_dict.keys(): # 反向字典构建
193             self.inverse_dict[self.ec_dict[x]] = x
194         i = f.tell()
195         data = b''
196         while i < eof: # 开始解压数据
197             raw = int.from_bytes(f.read(1), byteorder='big')
198             i = i + 1
199             j = 8
200             while j > 0:
201                 if (raw >> (j - 1)) & 1 == 1:
202                     data = data + b'1'
203                     raw = raw & (~(1 << (j - 1)))
204                 else:
205                     data = data + b'0'
206                     raw = raw & (~(1 << (j - 1)))
207                 if self.inverse_dict.get(data, 0) != 0:
208                     o.write(self.inverse_dict[data])
```

```
207         if self.inverse_dict.get(data, 0) != 0:
208             o.write(self.inverse_dict[data])
209             o.flush()
210             data = b''
211         j = j - 1
212         tem = int(i / eof * 100)
213         if tem > last:
214             print("解压:", tem, '%') # 输出解压进度
215             last = tem
216         raw = 0
217
218     f.close()
219     o.close()
220     print("文件解压成功！")
221
```

3.其他

这次试验我实际编写了哈夫曼算法的编解码程序,通过实践体会了编码与解码的实际运行过程,收获颇丰。

