第六次编程实验

PB16150288 黄志鹏

实验目的

1. 实现一个对于图片的Huffman 编解码器
2. 有训练数据统计训练编解码器， 得到压缩率
3. 使用编解码器对测试图像进行编码，存储，和解码重构

实验原理

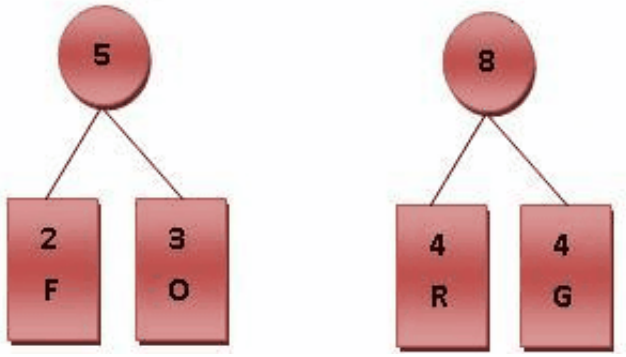
1. 霍夫曼编码

（一）进行霍夫曼编码前，我们先创建一个霍夫曼树。

1.将每个英文字母依照出现频率由小排到大，最小在左，如Fig.1。
2.每个字母都代表一个终端节点（叶节点），比较F.O.R.G.E.T六个字母中每个字母的出现频率，将最小的两个字母频率相加合成一个新的节点。如Fig.2所示，发现F与O的频率最小，故相加2+3=5。
3.比较5.R.G.E.T，发现R与G的频率最小，故相加4+4=8。
4.比较5.8.E.T，发现5与E的频率最小，故相加5+5=10。
5.比较8.10.T，发现8与T的频率最小，故相加8+7=15。
6.最后剩10.15，没有可以比较的对象，相加10+15=25。

Symbol	F	O	R	G	E	T
Frequency	2	3	4	4	5	7

最后产生的树状图就是霍夫曼树，参考*Fig.2*。



（二）进行编码

```
1.给霍夫曼树的所有左链接'0'与右链接'1'。
2.从树根至树叶依序记录所有字母的编码，如Fig.3。
```

Symbol	F	O	R	G	E	T
Frequency	2	3	4	4	5	7
CODE	000	001	100	101	01	11

2. 我实现的霍夫曼编码器

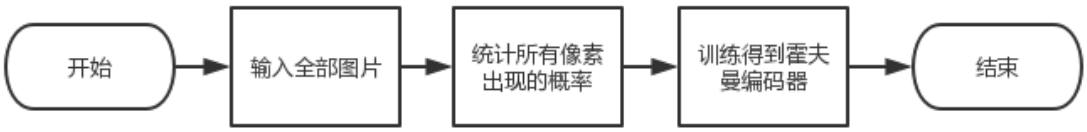
```
def to_dic(T: Tree.Node, dic: dict, path: list):
    if T.left == None and T.right == None:
        dic[T.char] = path.copy()
        return
    if T.left != None:
        path.append(0)
        to_dic(T.left, dic, path)
        path.pop(-1)
    if T.right != None:
```

```
        path.append(1)
        to_dic(T.right, dic, path)
        path.pop(-1)


class Coder:
    def __init__(self, train_str: str, probs: list):
        if len(train_str) != len(probs):
            print("输入长度没有对应")
            return None
        if is_repeat(train_str):
            print("不能有重复的字符")
            return None
        train_data = [x for x in train_str]
        data = [Tree.Node(train_str[i], probs[i]) for i in range(len(probs))]
        while len(data) > 1:
            data.sort(key=lambda x: x.prob, reverse=True)
            parent = Tree.Node('', data[-1].prob + data[-2].prob)
            parent.right = data.pop(-1)
            parent.left = data.pop(-1)
            data.append(parent)
        self.tree = data[0]
        self.dic = {}
        to_dic(self.tree, self.dic, [])


    def encode(self, read_filename: str, write_filename: str):
        img = Image.open('../data/test/' + read_filename)
        img = img.convert('L')

        img = np.array(img)
        size = img.shape
        img = img.flatten()
        assert(len(img) == size[1] * size[0])
        test_str = ''.join(chr(x) for x in img)



        result = []
        for i, char in enumerate(test_str):
            result += self.dic[char]
        size_x = [int(x) for x in '{0:010b}'.format(size[0])]
        size_y = [int(y) for y in '{0:010b}'.format(size[1])]
        padding_len = math.ceil((len(result) + 3 + 10 * 2) / 8) * 8 -
(len(result) + 3 + 10 * 2)
        result = [int(x) for x in '{0:03b}'.format(padding_len)] \
            + size_x + size_y + result + [0] * padding_len

        # convert bit list into bytearray and write it into file
```

```
        result_str = ''.join([chr(int(''.join(str(x)
                    for x in result[i * 8: i * 8 + 8]), 2)) for i in
range(len(result) // 8)])
        f = open('../data/compress/' + write_filename, mode='wb')
        f.write(str.encode(result_str))
        f.close()



    def decode(self, read_filename, write_img_filename):
        # read bytearray from file and convert bytearray into bit list
        f = open('../data/compress/' + read_filename, mode='rb')
        data_str = f.read().decode()
        f.close()
        data = []
        for c in data_str:
            data += [int(x) for x in '{0:08b}'.format(ord(c))]


        padding_len = int(''.join([str(x) for x in data[: 3]]), 2)
        size_x = data[3: 3 + 10]
        size_y = data[3 + 10: 3 + 2 * 10]
        data = data[3 + 2 * 10: len(data) - padding_len].copy()
        result = []
        T = self.tree
        for i, bit in enumerate(data):
            if bit == 0:
                T = T.left
            else:
                T = T.right
            if T.left == None and T.right == None:
                result.append(T.char)
                T = self.tree

        # TODO: convert to img and save to file
        size_x = int(''.join([str(x) for x in size_x]), 2)
        size_y = int(''.join([str(x) for x in size_y]), 2)
        result = [ord(x) for x in result]
        result = np.array(result)
        assert(len(result) == size_x * size_y)
        result = result.reshape([size_x, size_y])
        result = np.uint8(result)
        img = Image.fromarray(result)

        img.save('../data/reconstruction/' + write_img_filename)
```

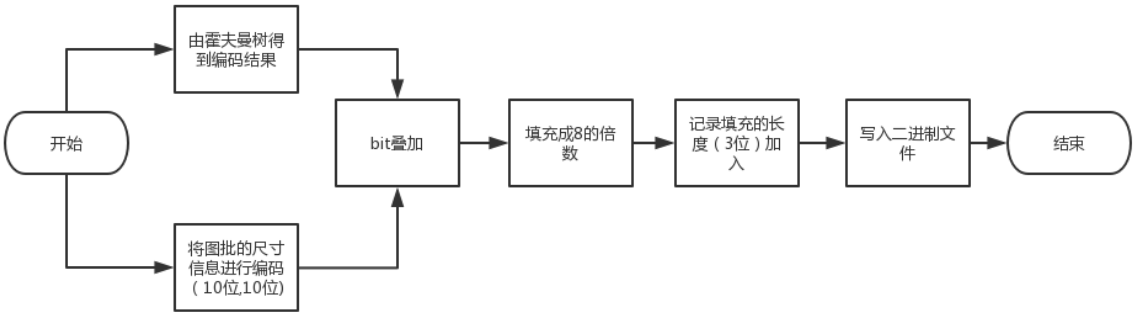
实验过程

1. 训练过程



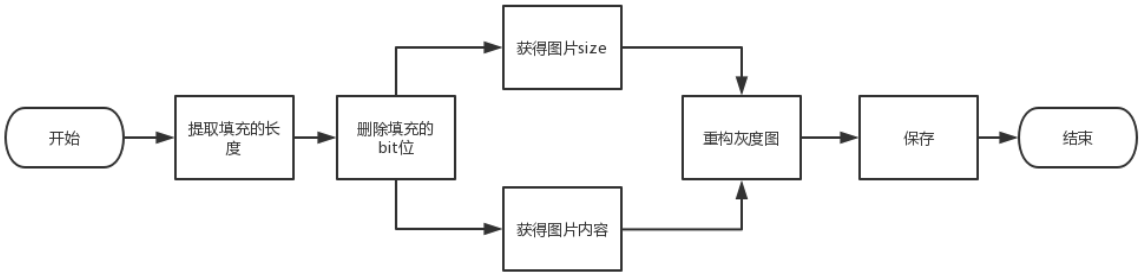

```
def train() -> "Coder, rate(int)":
    path = '../data/train/'
    files = os.listdir(path)
    stat = {}
    for f in files:
        img = Image.open(path + f)
        img = img.convert('L')
        img = np.array(img)
        col, raw = img.shape
        for x in range(col):
            for y in range(raw):
                c = chr(int(img[x, y]))
                if c in stat:
                    stat[c] += 1
                else:
                    stat[c] = 1
    # 归一化
    sum_stat = sum(stat.values())
    for c in stat.keys():
        stat[c] = stat[c] / sum_stat

    train_str = ''.join(stat.keys())
    train_probs = list(stat.values())

    coder = Coder.Coder(train_str, train_probs)
    compress_len = sum([len(coder.dic[c]) * stat[c] for c in train_str])
    true_len = 8 * len(train_str)
    rate = compress_len / true_len

    return coder, rate
```

2. 编码过程




```
    def encode(self, read_filename: str, write_filename: str):
        img = Image.open('../data/test/' + read_filename)
        img = img.convert('L')

        img = np.array(img)
        size = img.shape
        img = img.flatten()
        assert(len(img) == size[1] * size[0])
        test_str = ''.join(chr(x) for x in img)



        result = []
        for i, char in enumerate(test_str):
            result += self.dic[char]
        size_x = [int(x) for x in '{0:010b}'.format(size[0])]
        size_y = [int(y) for y in '{0:010b}'.format(size[1])]
        padding_len = math.ceil((len(result) + 3 + 10 * 2) / 8) * 8 -
(len(result) + 3 + 10 * 2)
        result = [int(x) for x in '{0:03b}'.format(padding_len)] \
            + size_x + size_y + result + [0] * padding_len

        # convert bit list into bytearray and write it into file
        result_str = ''.join([chr(int(''.join(str(x)
                    for x in result[i * 8: i * 8 + 8]), 2)) for i in
range(len(result) // 8)])
        f = open('../data/compress/' + write_filename, mode='wb')
        f.write(str.encode(result_str))
        f.close()
```

3. 解码过程

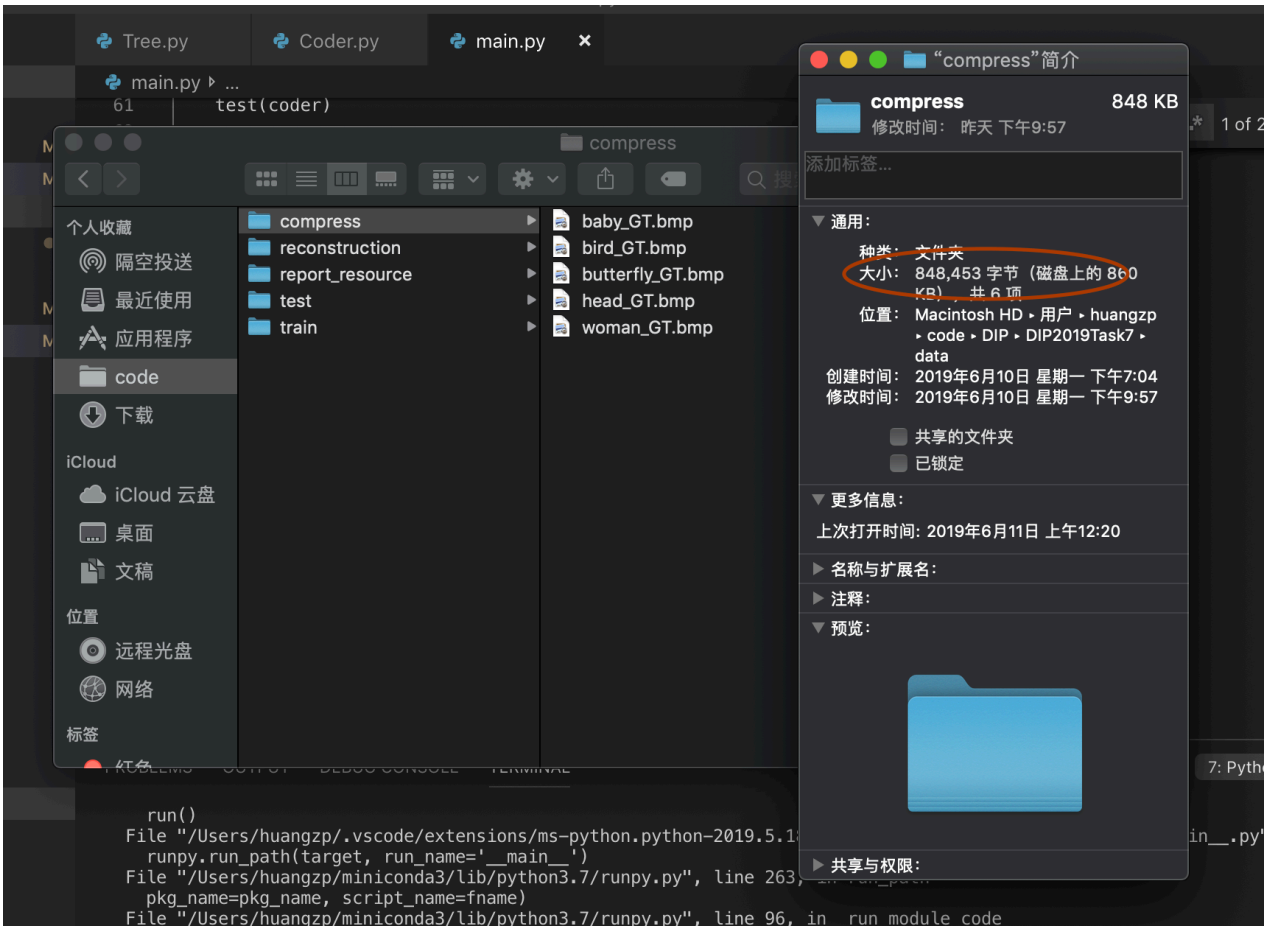


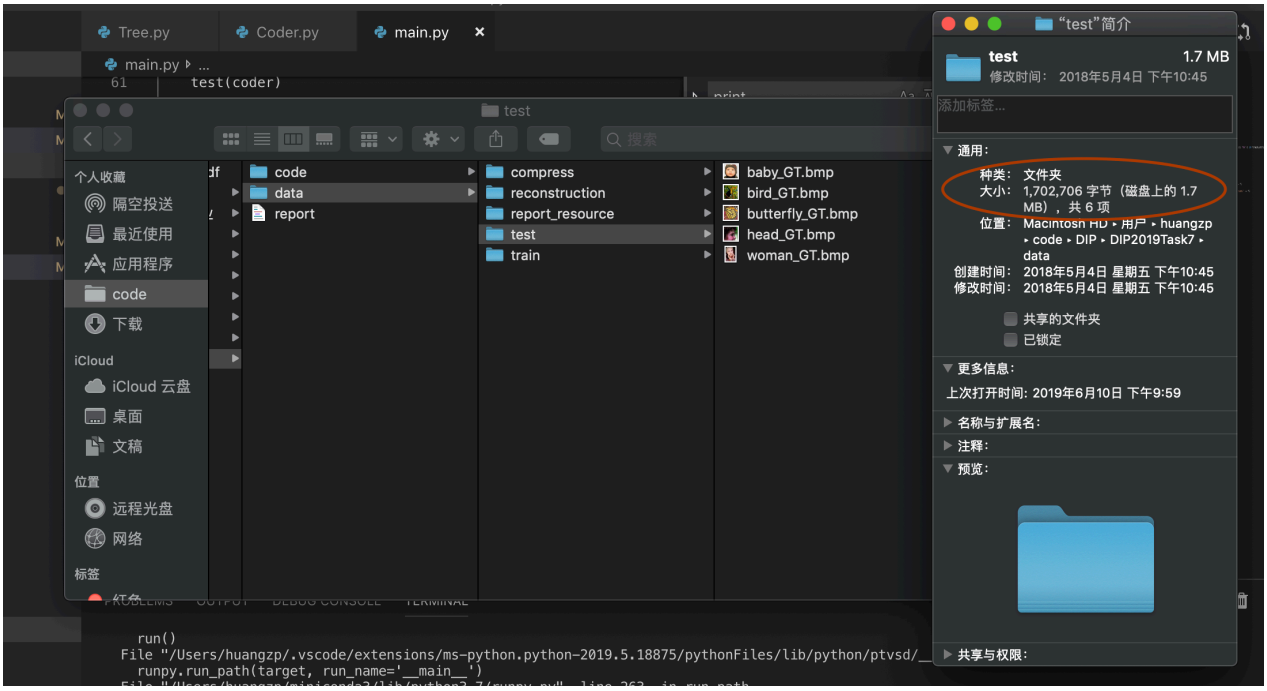

```
def decode(self, read_filename, write_img_filename):
        # read bytearray from file and convert bytearray into bit list
        f = open('../data/compress/' + read_filename, mode='rb')
        data_str = f.read().decode()
        f.close()
        data = []
        for c in data_str:
            data += [int(x) for x in '{0:08b}'.format(ord(c))]


        padding_len = int(''.join([str(x) for x in data[: 3]]), 2)
        size_x = data[3: 3 + 10]
        size_y = data[3 + 10: 3 + 2 * 10]
        data = data[3 + 2 * 10: len(data) - padding_len].copy()
        result = []
        T = self.tree
        for i, bit in enumerate(data):
            if bit == 0:
                T = T.left
            else:
                T = T.right
            if T.left == None and T.right == None:
                result.append(T.char)
                T = self.tree

        # TODO: convert to img and save to file
        size_x = int(''.join([str(x) for x in size_x]), 2)
        size_y = int(''.join([str(x) for x in size_y]), 2)
        result = [ord(x) for x in result]
        result = np.array(result)
        assert(len(result) == size_x * size_y)
        result = result.reshape([size_x, size_y])
        result = np.uint8(result)
        img = Image.fromarray(result)

        img.save('../data/reconstruction/' + write_img_filename)
```

实验结果

1. 压缩率=0.00389319725701988 73

2. 测试压缩情况





从文件大小可以清楚地看出压缩的效果

3. 解码重构

结果分析

1. 从压缩的效果和重构的效果可以看出，霍夫曼编码是效果非常好的压缩编码。
2. 编写代码过程中， 中间的压缩二进制文件， 我开始的时候使用的是文本文件读写存储，导致之后重构的图片存在失真。分析其中的原因是，使用文本文件存储因为一些特殊字符被忽略而丢失数据，在resize 的时候便出现失真。

实验所得

1. 二进制数据不要使用纯文本文件进行存储
2. python 各种数据结构之间的相互转换 如 int bitarray bypearray str 等等。