实验六实验报告

胡延伸 PB22050983

---

1. 实验内容

本实验的目标是实现 **Huffman 编码算法**，并通过此算法对文本文件中的字符进行 01 编码，理解其核心思想，同时计算 Huffman 编码相对于定长编码的压缩率。

具体任务包括：

1. 编写程序实现 Huffman 编码算法，完成对字符的 01 编码。
2. 对输入字符串或文件内容进行编码，并输出编码后的 01 序列。
3. 比较定长编码和 Huffman 编码的总长度，计算压缩率。
4. 将字符的出现频率及其编码以表格形式输出到文件 `table.txt`，格式如下：

```
字符    出现频率    编码
E       4           0
```

---

2. 算法设计思路

2.1 Huffman 编码核心思想

Huffman 编码是一种基于字符出现频率的前缀编码算法，具有以下特点：

1. **最优性**：Huffman 编码能够最小化编码后的平均长度。
2. **前缀码**：编码不会出现某个字符的编码是其他字符编码的前缀，确保解码的唯一性。
3. **构造过程**：
   - 根据字符出现频率构造一个优先队列，频率小的节点排在前面。
   - 每次从队列中取出两个频率最小的节点，合并成一个新节点，并将新节点加入队列。
   - 重复上述步骤，直到队列中只剩一个节点，形成 Huffman 树。

- 从根节点遍历 Huffman 树，为每个字符分配编码：左子树为 `0`，右子树为 `1`。

2.2 实现步骤

1. **统计频率**：统计文件中每个字符的出现频率，构造初始节点集合。
2. **构造 Huffman 树**：使用频率最小优先原则，通过合并节点的方式构造 Huffman 树。
3. **生成编码**：从根节点遍历 Huffman 树，生成每个字符对应的编码。
4. **计算压缩率**：
   - **定长编码长度**：假设字符集的大小为 `n`，则每个字符需 `ceil(log2(n))` 位编码，总长度为 `字符数 * 每字符编码长度`。
   - **Huffman 编码长度**：每个字符的编码长度乘以其出现频率，总和即为 Huffman 编码总长度。
   - 压缩率 = Huffman 编码长度 / 定长编码长度。
5. **输出编码表**：生成字符、频率和编码的表格，保存至文件 `table.txt`。

---

3. 源码 + 注释

以下为实验代码：

```
import math  # 用于计算定长编码的位数

# 定义 Huffman 节点类
class Node:
    def __init__(self, freq, char=None):
        self.freq = freq  # 节点频率
        self.char = char  # 节点字符
        self.code = None  # 节点编码
        self.left = None  # 左子节点
        self.right = None  # 右子节点
        self.parent = None  # 父节点

# 从队列中提取频率最小的节点
def EXTRACT_MIN(Q):
    return Q.pop(0)

# 将节点插入队列，并保持队列按频率升序排列
```

```python
def INSERT(Q, x):
    i = 0
    while i < len(Q) and Q[i].freq < x.freq:
        i += 1
    Q.insert(i, x)

# 构造 Huffman 树
def HUFFMAN(C):
    Q = [c for c in C]  # 初始化优先队列
    while len(Q) > 1:  # 合并直到只剩一个根节点
        z = Node(0)  # 新建一个节点
        z.left = x = EXTRACT_MIN(Q)  # 取出频率最小的两个节点
        z.right = y = EXTRACT_MIN(Q)
        z.freq = x.freq + y.freq  # 新节点频率为两子节点频率之和
        x.parent = y.parent = z  # 设置父节点
        INSERT(Q, z)  # 将新节点插回队列
    return Q[0]  # 返回根节点

# 生成 Huffman 编码
def HUFFMANCODING(C):
    root = HUFFMAN(C)  # 构造 Huffman 树
    for x in C:  # 遍历所有叶子节点
        x.code = ''  # 初始化编码
        y = x
        while y.parent:  # 从叶子节点回溯到根节点
            if y.parent.left == y:
                x.code = '0' + x.code  # 左子树为 0
            else:
                x.code = '1' + x.code  # 右子树为 1
            y = y.parent
    return C

# 主程序
with open('orignal.txt', 'r') as f:
    text = f.read()  # 读取文件内容
    freq = {}
    for c in text:  # 统计字符频率
        if c != '\n' and c != ' ':  # 忽略空格和换行符
            freq[c] = freq.get(c, 0) + 1

# 生成节点集合
C = [Node(freq[c], c) for c in freq]
```

```
    C.sort(key=lambda x: x.freq)  # 按频率升序排序

    # 计算定长编码总长度
    fixed_length = math.ceil(math.log2(len(C))) * sum([c.freq for c in C])

    # 求出 Huffman 编码
    C = HUFFMANCODING(C)

    # 计算 Huffman 编码总长度
    huffman_length = sum([c.freq * len(c.code) for c in C])

    # 计算压缩效率
    compression_ratio = huffman_length / fixed_length
    print('定长编码总长度: ', fixed_length)
    print('Huffman编码总长度: ', huffman_length)
    print('压缩效率: {:.2%}'.format(compression_ratio))

    # 输出编码表到 table.txt 文件
    to_write = []
    to_write.append("字符\t频率\t编码\n")
    for c in C:
        to_write.append(f"{c.char}\t{c.freq}\t{c.code}\n")

    with open('table.txt', 'w') as f:
        f.write(''.join(to_write))
```

---

4. 算法测试结果

输入文件 `orignal.txt` ，控制台输出如下:

```
定长编码总长度:  968100
Huffman编码总长度:  620836
压缩效率:  0.6412932548290465
```

输出文件 `table.txt` ,见附件。

5. 总结

1. **实验结果**：通过 Huffman 编码算法实现了对输入文件的编码，并成功计算了定长编码和 Huffman 编码的压缩率。
   - 对输入字符串 `AABBBEEEEGZ`，定长编码需要 33 位，而 Huffman 编码只需 24 位。
   - 压缩率为 **72.73%**，证明 Huffman 编码在字符频率分布不均时具有显著的压缩效果。
2. **算法优点**：Huffman 编码能够根据字符频率分配更短的编码，减少冗余，提高存储和传输效率。
3. **改进方向**：当前程序无法处理动态输入（如实时生成 Huffman 树），未来可增加实时编码功能，并支持更大规模的字符集。