

# Lab 6: Huffman 编码问题

PB21020718 曾健斌

## 一、实验内容

编程实现 Huffman 编码问题,并理解其核心思想

- 对字符串进行 01 编码,输出编码后的 01 序列, 并比较其相对于定长编码的压缩率
- 对文件 `original.txt` 中所有的大小写字母、数字(0-9)以及标点符号（即除空格 换行符之外的所有字符）按照 Huffman 编码方式编码为 01 序列, 将编码表输出至 `table.txt` 文件, 并在控制台打印压缩率

## 二、算法实现

首先定义结构体

```
1 struct Node {
2     char data;
3     int freq;
4     Node* left;
5     Node* right;
6 };
7
8 struct Compare {
9     bool operator()(Node* a, Node* b) { // 重载()用于优先队列的比较
10         return a->freq > b->freq;
11     }
12 };
```

`Node` 为 Huffman 树的节点, 由于本实现使用 STL 库中的 `priority_queue` 实现, 故定义了包含重载运算符 `()` 的结构体 `Compare`

以下为 `Huffman` 树的类声明

```
1 class Huffman {
2 public:
3     string text;
4     unordered_map<char, int> freqm; // 统计各字符频次
5     unordered_map<char, string> codem; // 记录各字符编码
6     unordered_map<string, char> decodem; // 记录各编码对应字符
7     Node* root;
8     string encoded;
9     string decoded;
10    int bitlength; // 定长编码所需每字符码长
11    float compressionRatio; // 压缩率
12
13    Huffman(string text) :
14        text(text), root(nullptr), compressionRatio(-1), bitlength(8) {}; // 初始令bitlength为 8 (ascii)
15    ~Huffman() {};
```

```

16     void buildHuffmanTree();
17     void buildCodeTable(Node* root, string code);
18     void encode();
19     void decode();
20     void writeEncoded(string filename);
21     void writeDecoded(string filename);
22     void printCodeTable(int base = 1);
23     void getCompressionRatio();
24 };

```

本实现使用了 STL 中的 unordered\_map 来记录键值对

以下为类方法的具体实现：

统计字符频次，并使用一个优先队列构建 Huffman 树

```

1  void
2  Huffman::buildHuffmanTree()
3  {
4      priority_queue<Node*, vector<Node*>, Compare> pq;
5      unordered_map<char, int> freqmtmp;
6
7      for (char c : text) {
8          // Only need a-z, A-Z, 0-9, and punctuation(all characters without
          space, tab, newline) for this project
9          if (c == ' ' || c == '\t' || c == '\n') { // 不考虑这些字符
10             continue;
11         }
12         freqmtmp[c]++;
13     }
14     for (auto it : freqmtmp) { // 每个字符 push 进入优先队列 pq
15         Node* node = new Node;
16         node->data = it.first;
17         node->freq = it.second;
18         node->left = nullptr;
19         node->right = nullptr;
20         pq.push(node);
21     }
22
23     while (pq.size() > 1) { // 利用优先队列的有序特性建立 Huffman 树
24         Node* left = pq.top();
25         pq.pop();
26         Node* right = pq.top();
27         pq.pop();
28         Node* parent = new Node;
29         parent->data = '\0';
30         parent->freq = left->freq + right->freq;
31         parent->left = left;
32         parent->right = right;
33         pq.push(parent);
34     }
35     freqm = freqmtmp;
36     root = pq.top();
37 }

```

```

1 void
2 Huffman::buildCodeTable(Node* root, string code)
3 {
4     if (root == nullptr) {        // 递归临界条件
5         return;
6     }
7     if (root->data != '\0') {      // 遇到字符则记录编码
8         codem[root->data] = code;
9         decodem[code] = root->data;
10    }
11    buildCodeTable(root->left, code + "0");
12    buildCodeTable(root->right, code + "1");
13 }

```

```
1 void
2 Huffman::getCompressionRatio()
3 {
4     bitlength = ceil(log2(codem.size()));    // 计算定长编码长度
5     int originalsize = 0;
6     int compressedsize = 0;
7     for (auto it : codem) {
8         originalsize += freqm[it.first] * bitlength;
9         compressedsize += freqm[it.first] * size(it.second);
10    }
11    compressionRatio = (float) compressedsize / originalsize * 100;
12 }
```

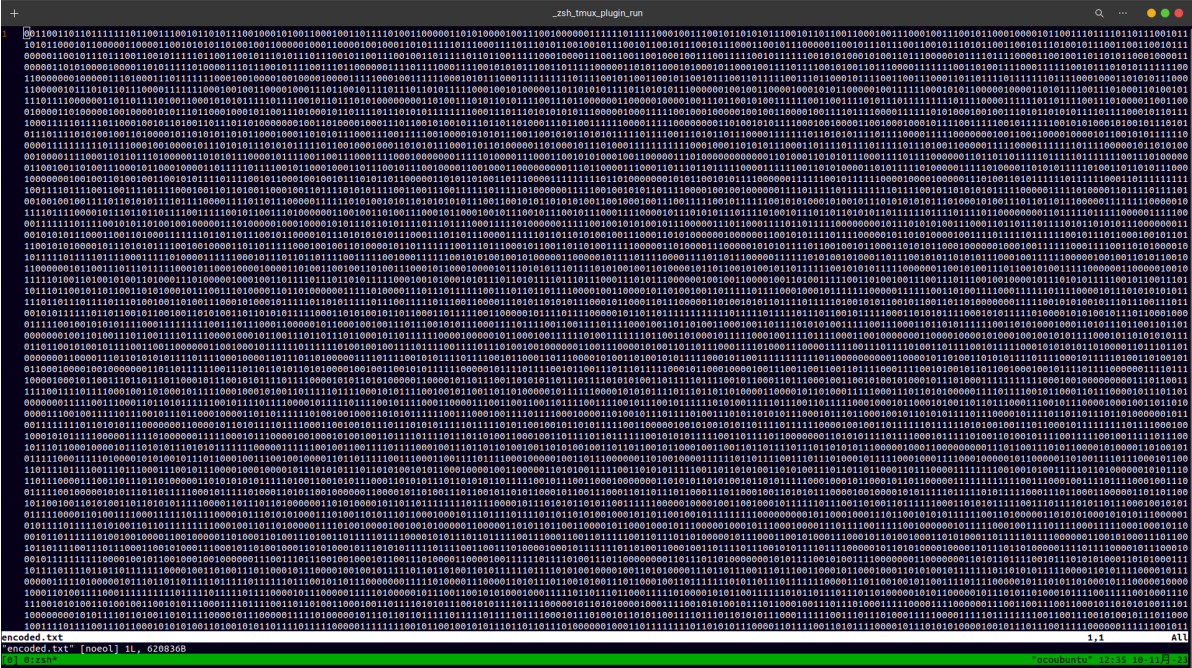
```

      1  freq: 4      code: 001100101110110
      2  freq: 1      code: 0011001011101010
      3  freq: 3494   code: 00100
      4  freq: 430   code: 00110011
      5  freq: 1      code: 00110010111100101
      6  freq: 7643   code: 0100
      7  freq: 1      code: 0011001011100100
      8  freq: 5      code: 00110010111000
      9  freq: 6281   code: 0000
     10  freq: 18    code: 1011100101100
     11  freq: 1938  code: 001101
     12  freq: 2761  code: 1110000
     13  freq: 2878  code: 00010
     14  freq: 188   code: 001100000
     15  freq: 3518  code: 00101
     16  freq: 3     code: 001100101110100
     17  freq: 9253  code: 1010
     18  freq: 50    code: 00110000011
     19  freq: 95    code: 00110000010
     20  freq: 2     code: 001100101111011
     21  freq: 2149  code: 101100
     22  freq: 49    code: 001100000110
     23  freq: 391   code: 00110001
     24  freq: 2     code: 0011001011101011
     25  freq: 985   code: 0011101
     26  freq: 197   code: 001100100
     27  freq: 112   code: 0011001010
     28  freq: 3     code: 001100101110011
     29  freq: 3428  code: 00011
     30  freq: 56    code: 00110010110
     31  freq: 1192  code: 1011101
     32  freq: 233   code: 001110000
     33  freq: 2029  code: 001111
     34  freq: 120   code: 00111000010
     35  freq: 120   code: 00111100011
     36  freq: 245   code: 001110010
     37  freq: 261   code: 0011110011
     38  freq: 7685  code: 0101
     39  freq: 7886  code: 0110
     40  freq: 8491  code: 0111
     41  freq: 292   code: 101110011
     42  freq: 17214  code: 100
     43  freq: 2173  code: 101101
     44  freq: 274   code: 101110000
     45  freq: 280    code: 101110001
     46  freq: 141   code: 1011100100
     47  freq: 67    code: 10111001010
     48  freq: 18    code: 10111001010101
     49  freq: 18    code: 1011100101110
     50  freq: 2663  code: 101111
     51  freq: 21    code: 1011100101111
     52  freq: 10834 code: 1101
     53  freq: 312   code: 111001000
Compression ratio: 64.1293%
12:37 Zeng@ocubuntu /home/zeng/Documents/alg-lab/lab6/source/cpp_source

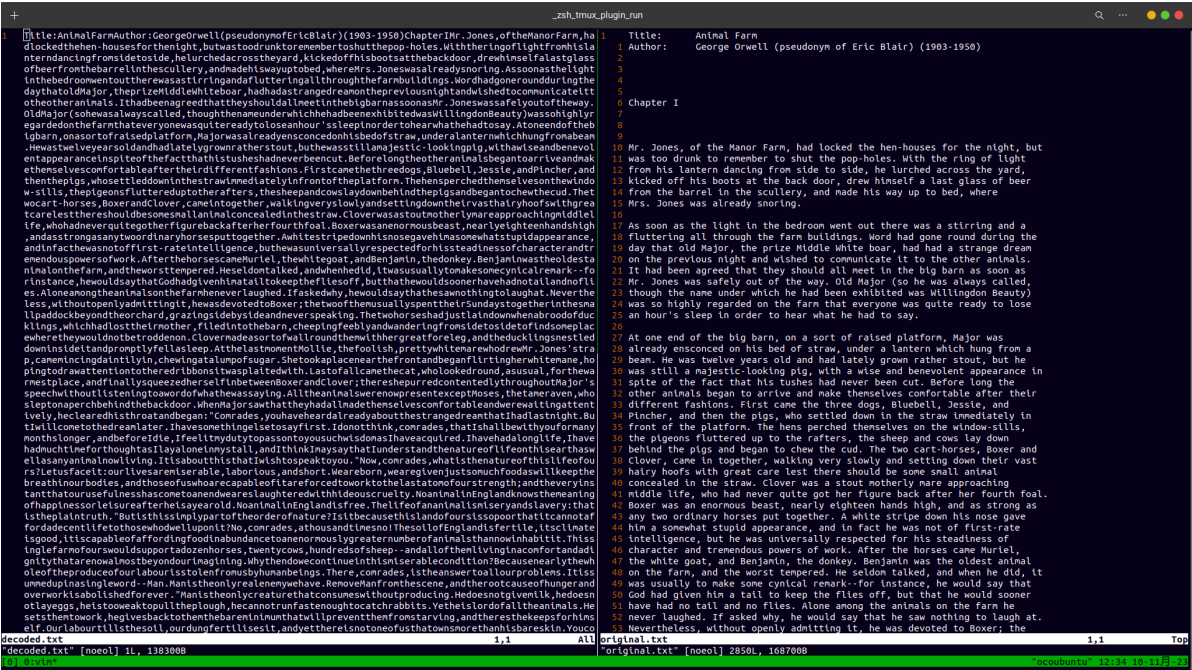
```

输出各字符对应出现频次以及对应的 Huffman 码

同时输出压缩率，得到64.1293%



输出编码结果



解码后与原文本进行对比

## 四、实验总结

本次实验旨在实现 Huffman 编码，总体较为简单

- 切身体会到了 Huffman 码作为一种贪心得到的前缀码的最优性
- priority\_queue 和 unordered\_map 的使用很有趣