Experiment—$Huffman$ Code

PB18010496 杨乐园

Introduction

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。

例如对于字符串"$AABBBEEEEGZ$",如果使用定长编码，‘A’,‘B’,‘E’,‘G’,‘Z’字符各需要3位01串编码，编码后的字符长度为$3*11=33$位，如果使用$Huffman$编码，可编码为下图，编码后的字符长度为$2*3+3*2+4*1+4+4=24$，从而压缩率为$24/33=72.73\%$。

对文件$data.txt$的字符串按照$Huffman$编码方式编码为01序列，并输出到$encode.txt$文件，控制台打印压缩率。

Purpose

实验目的：熟悉并掌握贪心法的算法设计思想，学习优先队列以及基于二叉树的相关操作，并理解有关$Huffman$编码问题的核心思想。

Data structure

$Huffman$节点数据结构为为：

$$
\begin{aligned}
huffman\_node=\{&\\
&c:char;\\
&weight:int\\
&huffman\_code[]:char;\\
&left:huffman\_code*;\\
&right:huffman\_code*;\\
\}&
\end{aligned}
$$

Idea

首先$Huffman$编码的正确性由其贪心选择性质与最优子结构的正确性保证。其次，由于其编码是为了压缩编码后的输出长度，所以我们在统计词频后，将所出现的字符按从小到大排列。进而，自底向上的构造出对应的最优编码二叉树T。设待编码的字符串由n个字符集合C组成，从而我们从第n个叶子结点开始构造，执行$n-1$次"合并"操作，即基于较小值得合并操作，逐节点向上构造出最终的二叉树。

综合以上构造方法，我们可以看到，所有的叶子结点即为字符串中出现的字符，我们可以采取层次遍历，当想左行进编码补0，相反向右行进编码补1，进而当遍历到某个叶子节点后所形成的编码即为其对应的$Huffman$
编码。

Algorithm

首先，我们读取文件中的字符串，并通过系统的$map<char,int>$封装统计每个字符的词频：

```
//读取文件并解析词频，word记录每个字符的词频，code为了之后记录每个字符的Huffman编码
int read_file(FILE* fn, map<char, int>& word, map<char, string>& code)
{
```

```
    if (fn == NULL) return 0;
    char line[MAX_LINE];
    while (fgets(line, MAX_LINE, fn))
    {//解析，统计词频
        char* p = line;
        while (*p != '\0' && *p != '\n') {
            map<char, int>::iterator it = word.find(*p);
            if (it == word.end()) // 不存在，插入
                code.insert(make_pair(*p, "\0")), word.insert(make_pair(*p, 1));
            else
                it->second++;
            p++;
        }
    }
    return 0;
}
```

其次，调用系统内置的$sort$函数，将字符按词频从小到大排列，并逐节点构造$Huffman$树：

```
//按weight升序排列
bool sort_by_weight(huffman_node* a, huffman_node *b) { return a->weight < b-
>weight;}

//构造Huffman树
int huffman_tree_create(huffman_node*& root, map<char, int>& word) {
    char line[MAX_LINE];
    vector<huffman_node*> huffman_tree_node;
    map<char, int>::iterator it_t;
    for (it_t = word.begin(); it_t != word.end(); it_t++) // 为每一个节点申请空间
    {
        huffman_node* node = (huffman_node*)malloc(sizeof(huffman_node));
        node->c = it_t->first;
        node->weight = it_t->second;
        node->huffman_code[0] = '\0';
        node->left = NULL;
        node->right = NULL;
        huffman_tree_node.push_back(node);
    }
    while (huffman_tree_node.size() > 0) //从叶节点开始构建Huffman树
    {
        // 按照weight升序排序
        sort(huffman_tree_node.begin(), huffman_tree_node.end(),
sort_by_weight);
        if (huffman_tree_node.size() == 1) // 只有一个根结点
        {
            root = huffman_tree_node[0];
            huffman_tree_node.erase(huffman_tree_node.begin());
        }
        else
        {   // 取出前两个
            huffman_node* node_1 = huffman_tree_node[0];
            huffman_node* node_2 = huffman_tree_node[1];
            // 删除
            huffman_tree_node.erase(huffman_tree_node.begin());
            huffman_tree_node.erase(huffman_tree_node.begin());
            // 生成新的节点
            huffman_node* node = (huffman_node*)malloc(sizeof(huffman_node));
```

```
            node->c = '.', node->left = NULL, node->right = NULL;
            node->huffman_code[0] = '\0';
            node->weight = node_1->weight + node_2->weight;
            (node_1->weight < node_2->weight) ? (node->left = node_1, node->right = node_2) : (node->left = node_2, node->right = node_1);
            huffman_tree_node.push_back(node);
        }
    }
    return 0;
}
```

进一步，采取层次遍历给出不同字符的$Huffman$编码：

```
//实现Huffman编码
int get_huffman_code(huffman_node*& node)
{
    if (node == NULL) return 1;
    // 利用层次遍历，构造每一个节点
    huffman_node* p = node;
    queue<huffman_node*> q;
    q.push(p);
    while (q.size() > 0) {
        p = q.front();
        q.pop();
        if (p->left != NULL)
        {
            q.push(p->left);
            strcpy((p->left)->huffman_code, p->huffman_code);//, size(p->huffman_code)
            char* ptr = (p->left)->huffman_code;
            while (*ptr != '\0') ptr++;
            *ptr = '0';
            ptr++;
            *ptr = '\0';
        }
        if (p->right != NULL)
        {
            q.push(p->right);
            strcpy((p->right)->huffman_code, p->huffman_code);
            char* ptr = (p->right)->huffman_code;
            while (*ptr != '\0') ptr++;
            *ptr = '1';
            ptr++;
            *ptr = '\0';
        }
    }
    return 0;
}
```

最后，我们打印出相应的$Huffman$编码，并计算对应的压缩率：

```
//打印叶子结点信息，并计算对应长度
void print_leaf(huffman_node* node, map<char, int>& word, int& huffman_size, map<char, string>& code)
{
    if (node != NULL)
```

```
    {
        string temp;
        print_leaf(node->left, word, huffman_size, code);
        if (node->left == NULL && node->right == NULL)
        {
            temp = node->huffman_code;
            map<char, int>::iterator it = word.find(node->c);
            map<char, string>::iterator its = code.find(node->c);
            its->second = temp;
            huffman_size += size(temp) * it->second;
        }
        print_leaf(node->right, word, huffman_size, code);
    }
}

//计算压缩率
void compression_ratio(map<char, int> word, int huffman_size)
{
    //计算定长编码的长度
    int k = 0, n = word.size() - 1;
    while (n)
    {
        ++k;
        n >>= 1;
    }
    //计算字符数
    n = 0;
    map<char, int>::iterator m1_Iter;
    for (m1_Iter = word.begin(); m1_Iter != word.end(); m1_Iter++)
        n += m1_Iter->second;
    cout << "Huffman编码所需字符长度为：" << huffman_size << endl;
    cout << "定长编码所需字符长度为：" << k * n << endl;
    cout << "压缩率为：" << 1.0 * huffman_size / (k * n) * 100 << "%" << endl;

}
```

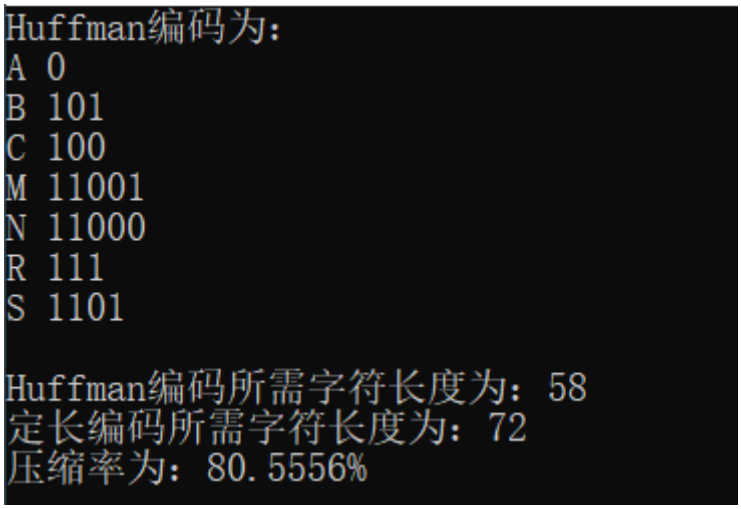
Results

通过运行程序与测试数据，我们有如下输出结果：

原字符串为：$AASMABBAAARRAABCAACCRRSN$。

对应$Huffman$编码为：
00110111001010110100011111100101100001001001111111110111000。

相关压缩率以及其他信息如下：



我们可以看到，输出结果正确。

Code

具体完整代码，参看附件文件$Huffman$。