同济大学

离散数学说明文档

--最优2元树在通信编码中的应用

姓　　名：胡正华

学　　号：2353741

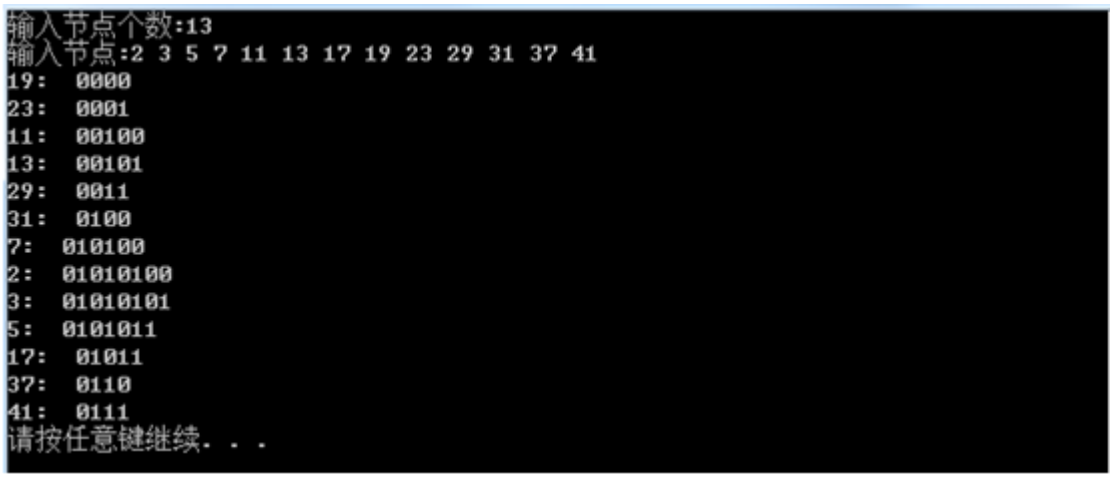
任课教师：李　冰

1. 题目简介

1.1 题目要求

根据用户给出的通信编码使用频率，建立哈夫曼树，并求每个节点的对应二元编码。

1.2 题目样例



2. 解题思路

本实验的目的是通过 Huffman 编码方法，根据通信符号的使用频率生成对应的前缀码，从而实现数据的无损压缩。实验的解题思路如下：

2.1 问题分析

Huffman 编码是一种广泛用于数据压缩的算法，其核心思想是使用较短的编码表示频率较高的符号，较长的编码表示频率较低的符号，从而有效减少编码后的总长度。

实现过程中，需要基于给定符号频率构造 Huffman 树，并通过遍历生成符号对应的前缀码。

2.2 算法设计

Huffman 编码的实现主要分为以下几个步骤：

1. **初始化叶子节点**

根据输入的通信符号及其频率，生成对应的叶子节点，每个节点存储符号的频率值。

这些叶子节点可以看作是初始状态的森林，每个节点是一棵只有一个节点的树。

2. **构造 Huffman 树**

利用贪心策略，每次从森林中选择频率最小的两棵树进行合并，生成一棵新的树，其根节点的频率值为两棵子树频率之和。

合并后的新树重新加入森林，并对森林按频率升序排序。

重复上述操作，直到森林中仅剩一棵树，此时得到的即为 Huffman 树。

3. **生成前缀码**

对生成的 Huffman 树进行前序遍历，每次向左子树添加一个“0”，向右子树添加一个“1”，从而为每个叶子节点生成唯一的前缀码。

前缀码具有前缀无关性，即任何一个编码都不是其他编码的前缀，从而保证了解码的唯一性。

2.3 算法实现

2.3.1 数据结构设计：

使用一个 TreeNode 结构体存储树节点的信息，包括频率值（权值）、左子节点指针和右子节点指针。

使用动态数组或 vector 存储节点列表，便于对森林进行排序和操作。

2.3.2 关键操作：

排序：在每次合并树之前，对节点列表按频率值从小到大排序，确保每次选择频率最小的两棵树。

合并：将两棵树合并为一棵新树，频率值为两棵子树的频率之和。

遍历：通过递归方式生成前缀码，分别为左子树追加“0”、右子树追加“1”。

3. 代码实现

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
using namespace std;

// 定义树节点结构体
struct TreeNode {
    int num;                // 节点权值
    TreeNode* Lnode;        // 左子树
    TreeNode* Rnode;        // 右子树
    TreeNode(int val) : num(val), Lnode(nullptr),
Rnode(nullptr) {}
};

// 全局变量
vector<TreeNode*> nodes;    // 保存所有节点
string prefixCode;          // 保存前缀码

// 初始化叶子节点
void initNodes(const vector<int>& frequencies) {
    for (int freq : frequencies) {
        nodes.push_back(new TreeNode(freq));
    }
}

// 排序节点数组（升序）
void sortNodes() {
    sort(nodes.begin(), nodes.end(), [](TreeNode* a,
TreeNode* b) { return a->num < b->num;
        });
}
```

```cpp
// 构造Huffman树
TreeNode* constructTree() {
    while (nodes.size() > 1) {
        // 取出权值最小的两个节点
        TreeNode* left = nodes[0];
        TreeNode* right = nodes[1];
        nodes.erase(nodes.begin());
        nodes.erase(nodes.begin());

        // 合并成新节点
        TreeNode* parent = new TreeNode(left->num + right->num);
        parent->Lnode = left;
        parent->Rnode = right;

        // 插入新节点并重新排序
        nodes.push_back(parent);
        sortNodes();
    }
    return nodes[0]; // 返回根节点
}

// 前序遍历输出前缀码
void preorder(TreeNode* node, string code) {
    if (node != nullptr) {
        if (node->Lnode == nullptr && node->Rnode == nullptr) { // 叶子节点
            cout << node->num << ": " << code << endl;
        }
        preorder(node->Lnode, code + "0");
        preorder(node->Rnode, code + "1");
    }
}
```

```cpp
int main() {
    int n;
    cout << "输入节点个数: ";
    cin >> n;

    vector<int> frequencies(n);
    cout << "输入节点: ";
    for (int i = 0; i < n; i++) {
        cin >> frequencies[i];
    }

    initNodes(frequencies);     // 初始化叶子节点
    sortNodes();                // 对节点数组进行初始排序
    TreeNode* root = constructTree(); // 构造Huffman树
    preorder(root, "");         // 前序遍历并输出前缀码
    return 0;
}
```

4. 心得体会

通过本次实验，我深入学习并实践了 Huffman编码算法 的原理和实现方法，充分体会到数据压缩算法在实际应用中的重要性。

1. 贪心策略的强大
   Huffman 编码是一种典型的贪心算法，它的核心思想是每次选择权值最小的两棵子树合并，从而逐步构建出最优的 Huffman 树。实验过程中，我感受到贪心策略的高效性，虽然每次只关注局部最优解，但最终却能保证全局最优的结果。这让我更加认识到贪心算法在解决最优化问题中的广泛应用。
2. 树的构造与遍历的实践
   本实验重点在于 Huffman 树的构造和遍历。从初始化叶子节点到合并子树，再到前序遍历生成前缀码，每一步都让我对树这一数据结构的理解更加深刻。特别是在递归实现前序遍历时，我深刻体会到递归的简洁与强大，同时也注意到递归过程中需要谨慎处理终止条件，避免程序出现无限递归的问题。