哈尔滨工业大学计算学部

实验报告

课程名称：数据结构与算法

课程类型：专业基础（必修）

实验项目：树形结构及其应用

实验题目：哈夫曼编码与译码方法

实验日期：2023年10月12日

班级：2237102

学号：2022111654

姓名：李宸

|  |  |  |
| --- | --- | --- |
| 设计成绩 | 报告成绩 | 指导老师 |
|  |  | 李秀坤 |

**一、实验目的**

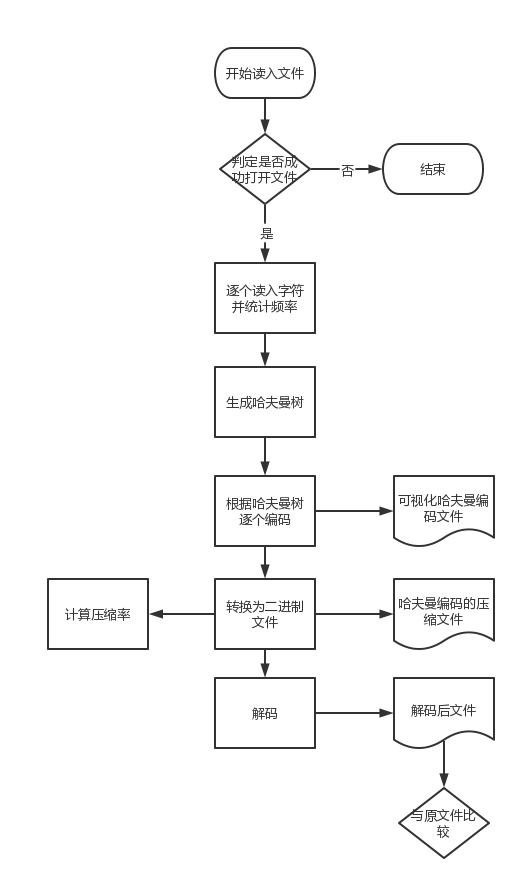
本实验主要要求从文件中正确读取，统计字符及出现频率，并根据频率利用哈夫曼树进行哈夫曼编码，以达到压缩文件的效果。并根据压缩后的文件计算压缩率，并解码与原文件进行比较。

**二、实验要求及实验环境**

1. **从文件中读入任意一篇英文文本文件，分别统计英文文本文件中各字符（包括标点符号和空格）的使用频率；**
2. **根据已统计的字符使用频率构造哈夫曼编码树，并给出每个字符的哈夫曼编码（字符集的哈夫曼编码表）；**
3. **将文本文件利用哈夫曼树进行编码，存储成压缩文件（哈夫曼编码文件）；**
4. **计算哈夫曼编码文件的压缩率；**
5. **将哈夫曼编码文件译码为文本文件，并与原文件进行比较。**

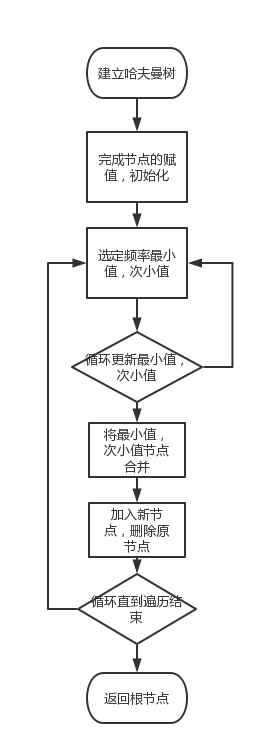
**三、设计思想**（本程序中的用到的所有数据类型的定义，各程序模块之间的调用关系、核心算法的流程图或主要步骤）

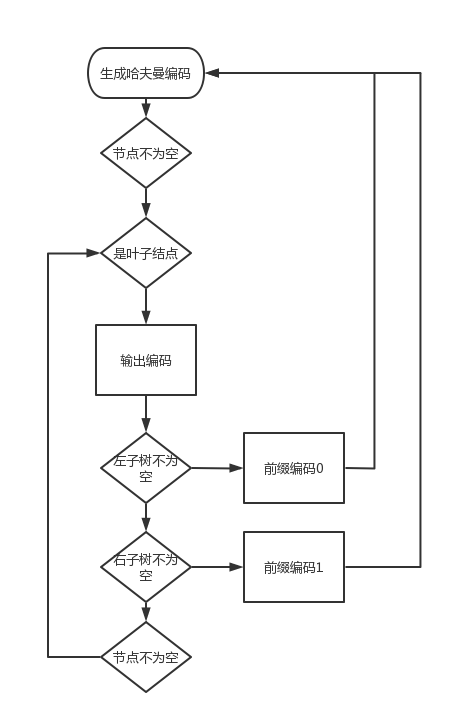
1．逻辑设计

整体流程图如下

核心算法步骤：

（1）建立哈夫曼树

（2）进行哈夫曼编码



1. 物理设计

函数表如下：

|  |  |  |
| --- | --- | --- |
| 函数名 | 类型 | 功能 |
| isInCharacterSet | int | 判断是否在字符集中出现过 |
| findCharacterIndex | int | 用于查找字符在字符频率数组中的位置 |
| buildHuffmanTree | TreeNode \* | 用于构建哈夫曼树 |
| generateHuffmanCodes | void | 生成字符的哈夫曼编码 |
| main | int | 主函数 |

变量表如下：

|  |  |  |
| --- | --- | --- |
| 变量 | 数据类型 | 功能 |
| inputFile | FILE\* | 指向输入文件的指针 |
| outputFile | FILE\* | 指向输出文件的指针 |
| charsetSize | int | 存储字符数组的长度 |
| prefix | char\* | 存储字符的哈夫曼编码前缀 |
| currentChar | char | 用于读取当前字符 |
| binaryString | char\* | 存储二进制编码的字符串，加上 '\0' 结尾 |
| originalSize | long | 原文件大小 |
| compressedSize | long | 压缩后文件大小 |
| compressionRatio | double | 计算压缩率 |

本实验还设计了三个结构体：

1. CharFrequency结构体：用于存储字符和其频率

包含char charater，用于存储字符

int frequency，用于存储频率

1. TreeNode结构体：表示哈夫曼树的节点

包含char character;用于存储字符

int frequency;用于存储频率

char code[MAX\_SIZE];哈夫曼编码

struct TreeNode \*left;左子树

struct TreeNode \*right;右子树

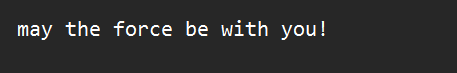
1. HuffmanCode结构体：存储字符及对应的哈夫曼编码

包含 char character;存储字符

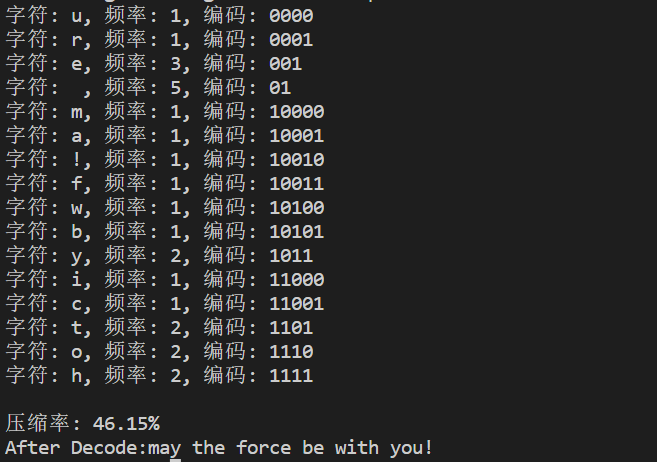
char code[MAX\_SIZE];存储字符对应的哈夫曼编码

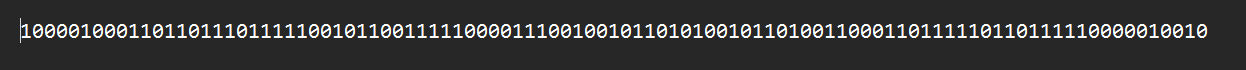
1. **测试结果**

测试数据：（input.txt）

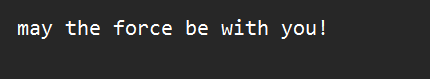


测试结果：(终端输出)

可视化编码：(output.txt)



解码结果：（output\_decoded.txt）



**五、经验体会与不足**

通过本次实验，我充分掌握了哈夫曼编码的过程，完成了压缩文件的任务。在此基础上可以优化取最小值，次小值算法；并可尝试按单词压缩。

**六、附录：源代码（带注释）**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_SIZE 256

int id=0;//用于表示HuffmanCode数组的下标

// 结构体用于存储字符和其频率

typedef struct {

char character;

int frequency;

} CharFrequency;

// 结构体表示哈夫曼树的节点

typedef struct TreeNode {

char character;

int frequency;

char code[MAX\_SIZE];

struct TreeNode \*left;

struct TreeNode \*right;

} TreeNode;

//存储字符及对应的哈夫曼编码

typedef struct HuffmanCode {

char character;

char code[MAX\_SIZE];

}HuffmanCode;

//判断是否在字符集中出现过

int isInCharacterSet(char character, CharFrequency charset[], int charsetSize) {

for (int i = 0; i < charsetSize; i++) {

if (charset[i].character == character) {

return 1;

}

}

return 0;

}

// 用于查找字符在字符频率数组中的位置

int findCharacterIndex(CharFrequency \*freqArray, int size, char character) {

for (int i = 0; i < size; i++) {

if (freqArray[i].character == character) {

return i;

}

}

return -1;

}

// 构建哈夫曼树

TreeNode \*buildHuffmanTree(CharFrequency \*freqArray, int size) {

// 创建节点数组，每个字符对应一个节点

TreeNode \*nodes[size];

for (int i = 0; i < size; i++) {

nodes[i] = (TreeNode \*)malloc(sizeof(TreeNode));

nodes[i]->character = freqArray[i].character;

nodes[i]->frequency = freqArray[i].frequency;

nodes[i]->left = NULL;

nodes[i]->right = NULL;

}

// 构建哈夫曼树

while (size > 1) {

// 找到两个频率最低的节点

int min1 = 0, min2 = 1;

if (nodes[min1]->frequency > nodes[min2]->frequency) {

int temp = min1;

min1 = min2;

min2 = temp;

}

for (int i = 2; i < size; i++) {

if (nodes[i]->frequency < nodes[min1]->frequency) {

min2 = min1;

min1 = i;

} else if (nodes[i]->frequency < nodes[min2]->frequency) {

min2 = i;

}

}

// 创建一个新节点，合并两个最小频率的节点

TreeNode \*newNode = (TreeNode \*)malloc(sizeof(TreeNode));

newNode->character = '\0';

newNode->frequency = nodes[min1]->frequency + nodes[min2]->frequency;

newNode->left = nodes[min1];

newNode->right = nodes[min2];

// 将新节点插入节点数组中

nodes[min1] = newNode;

nodes[min2] = nodes[size - 1];

size--;

}

return nodes[0]; // 返回根节点

}

// 函数用于生成字符的哈夫曼编码

void generateHuffmanCodes(TreeNode \*root, char \*prefix, int depth,HuffmanCode huffmanCodes[]) {

if (root == NULL) {

return;

}

if (root->character != '\0') {

prefix[depth] = '\0';

printf("字符: %c, 频率: %d, 编码: %s\n", root->character, root->frequency, prefix);

huffmanCodes[id].character=root->character;

strcpy(huffmanCodes[id].code,prefix);

id++;

}

if (root->left != NULL){

prefix[depth] = '0';

generateHuffmanCodes(root->left, prefix, depth +1, huffmanCodes);

}

if (root->right != NULL){

prefix[depth] = '1';

generateHuffmanCodes(root->right, prefix, depth +1, huffmanCodes);

}

}

int main() {

// 打开待处理的文本文件

FILE \*inputFile = fopen("input.txt", "r");

if (inputFile == NULL) {

printf("无法打开文件\n");

return 1;

}

// 初始化字符集和字符频率统计数组

CharFrequency charset[MAX\_SIZE]; // 使用ASCII字符集

int charsetSize = 0;

// 逐字符读取文件并统计频率

char c;

while ((c = fgetc(inputFile)) != EOF) {

if (!isInCharacterSet(c, charset, charsetSize)) {

charset[charsetSize].character = c;

charset[charsetSize].frequency = 1;

charsetSize++;

} else {

// 统计字符的频率

for (int i = 0; i < charsetSize; i++) {

if (charset[i].character == c) {

charset[i].frequency++;

break;

}

}

}

}

// 关闭文件

fclose(inputFile);

//建立哈夫曼编码数组，存储字符和对应的编码

HuffmanCode huffmanCodes[charsetSize];

// 构建哈夫曼树

TreeNode \*huffmanTree = buildHuffmanTree(charset, charsetSize);

char prefix[charsetSize]; // 存储字符的哈夫曼编码前缀

generateHuffmanCodes(huffmanTree, prefix, 0, huffmanCodes);

// 关闭输入文件

fclose(inputFile);

// 打开输出文件

FILE \*outputFile = fopen("output.txt", "w");

if (outputFile == NULL) {

printf("无法创建输出文件\n");

return 1;

}

// 重新打开待处理的文本文件

inputFile = fopen("input.txt", "r");

char currentChar;

// 逐字符读取原始文件并写入哈夫曼编码文件

while ((currentChar = fgetc(inputFile)) != EOF) {

for (int i = 0; i < charsetSize; i++) {

if (huffmanCodes[i].character == currentChar) {

fprintf(outputFile,"%s",huffmanCodes[i].code);

}

}

}

// 关闭文件

fclose(inputFile);

fclose(outputFile);

//转换二进制字符

FILE \*textFile = fopen("output.txt", "r");

FILE \*binaryFile = fopen("output.bin", "wb");

if (textFile == NULL || binaryFile == NULL) {

perror("无法打开文件");

return 1;

}

int currentChar2;

char binaryString[9]; // 存储二进制编码的字符串，加上 '\0' 结尾

int count = 0;

while ((currentChar2 = fgetc(textFile)) != EOF) {

if (currentChar2 == '0' || currentChar2 == '1') {

binaryString[count] = currentChar2;

count++;

// 如果已经读取了8个字符，将其转换并写入二进制文件

if (count == 8) {

binaryString[count] = '\0';

unsigned char byte = (unsigned char)strtol(binaryString, NULL, 2);

fwrite(&byte, 1, 1, binaryFile);

count = 0;

}

}

}

// 关闭文件

fclose(textFile);

fclose(binaryFile);

// 计算压缩率

inputFile = fopen("input.txt", "r");

outputFile = fopen("output.bin", "rb");

fseek(inputFile, 0L, SEEK\_END);

fseek(outputFile, 0L, SEEK\_END);

long originalSize = ftell(inputFile);

long compressedSize = ftell(outputFile);

double compressionRatio = (double)compressedSize / originalSize;

printf("\n压缩率: %.2f%%\n", compressionRatio \* 100);

// 关闭文件

fclose(inputFile);

fclose(outputFile);

// 解码部分

inputFile = fopen("output.txt", "r");

outputFile = fopen("output\_decoded.txt", "w");

TreeNode \*currentNode = huffmanTree;

char currentChar3;

printf("After Decode:");

while ((currentChar3 = fgetc(inputFile)) != EOF) {

if (currentChar3 == '0') {

currentNode = currentNode->left;

} else if (currentChar3 == '1') {

currentNode = currentNode->right;

}

if (currentNode->left == NULL && currentNode->right == NULL) {

printf("%c",currentNode->character);

fprintf(outputFile, "%c", currentNode->character);

currentNode = huffmanTree; // 重置为根节点

}

}

// 关闭文件

fclose(inputFile);

fclose(outputFile);

return 0;

}