

实验报告

实验题目

Huffman 编码压缩/解压器

问题描述

在合适的情况下，利用 Huffman 编码对文件进行压缩可以减少其占用空间，同时在使用到文件的时候也可以根据压缩文件中所提供的信息来将其还原为原文件。本次实验中，我们将实现一个基于 Huffman 编码的文件压缩/解压缩工具。

基本要求

基于 Huffman 编码实现一个压缩器和解压缩器（其中 Huffman 编码以字节作为统计和编码的基本符号单元），使其可以对任意的文件进行压缩和解压缩操作。针对编译生成的程序，要求压缩和解压缩部分可以分别独立运行。具体要求为：

1. 每次运行程序时，用户可以指定只压缩/只解压缩指定路径的文件。实现的时候不限制与用户的交互方式，可供参考的方式包括但不限于
 - 根据命令行参数指定功能（压缩/解压缩）和输入/输出文件路径
 - GUI 界面
 - 运行程序后由用户交互输入指定功能和路径

2. **CAUTION!**不被允许的交互方式：通过修改源代码指定功能和文件路径

3. 压缩时不需要指定解压文件的目标路径，解压时不需要指定压缩前原文件的路径，压缩后的文件可以换到另一个位置再做解压缩

设计思路

压缩部分

1. 先读取一次文件，统计各种字符数量
2. 构建 Huffman 树和 Huffman 编码
3. 再读取一次文件，并且将文件字符翻译成 Huffman 编码，达到 1 字节的时候即写入，并且将最后一位写入压缩的文件。

解压部分

1. 先读取一次文件，得到文件头和各种字符的出现次数
2. 利用上面得到的信息构建出 Huffman 树和 Huffman 编码
3. 再读取一次文件，从文件中一位位的读取信息遍历 Huffman 树，得到 Huffman 编码写入文件，达到文件长时就结束。

关键代码

part0: 各种结构的声明

```
struct filehead {
    //存放关键信息
    unsigned char flag[3];           //压缩二进制文件头部标志store huff
    int alphaVariety;               //字符种类
    unsigned char lastValidBit; //最后一个字节的有效位数
    long filelenth;
    unsigned char unused[4];        //待用空间
};
struct huffman
{
    int parent, lchild, rchild;
    int count; //存字符出现的频数
    unsigned char data; //存这个字符
    char *code; //存huffman码
    int visit; //看是否被访问
};
struct alphafreq
{
    unsigned char ch;
    long freq;
};
```

filehead是文件的头部数据。huffman是Huffman树的基本结点，在初始化的时候会把parent, lchild, rchild设为-1，表示到了尽头。

part1: Huffman树的构建

此为生成Huffman树的过程

由于有n个叶子结点的Huffman树有 $2n-1$ 个结点，总长度为 $2n-1$

首先从Huffman表中搜寻出两个最小的结点调用select函数时会将这个最小的结点的visited设为true

然后将min1和min2拼接到第j个结点。直到 $j=2n-1$ 。

```
/*alphatype是整个文件中的文件类型，huffmantree已经申请好内存并初始化完，totallenth是
Huffman树的总长度
*/
void createhuffmantree(huffman* huffmantree, int alphatypes, int totallenth) {
    int min1, min2;
    for (int j = alphatypes; j < totallenth; j++) {
        select(huffmantree, j, min1); //select用于搜寻huffmantree中未被访问的，最小的结
        点，
        select(huffmantree, j, min2);
        //实现结点间的连接
        huffmantree[min1].parent = huffmantree[min2].parent = j;
        huffmantree[j].lchild = min1;
        huffmantree[j].rchild = min2;
        huffmantree[j].count = huffmantree[min1].count +
        huffmantree[min2].count;
    }
}
```

part2:huffman编码的构建

此为构建Huffman编码的过程

思路是从孩子向根遍历，如果是左孩子，则为'0'，如果是右孩子，则为'1'；

这样一步步向上知道到达根(利用parent=-1控制)

然后将整串Huffman编码复制到对应的code上面

```
void createhuffmancode(huffman* huffmantree, int alphasizes) {
    char temp[256];
    for (int i = 0; i < 256; i++) {
        temp[i] = '\0';
    }
    int cur, index, next;
    //从孩子到父母去遍历
    for (int i = 0; i < alphasizes; i++) {
        index = 255;
        for (cur = i, next = huffmantree[cur].parent; next != -1; cur = next,
            next = huffmantree[next].parent)
        {
            if (cur == huffmantree[next].lchild) {
                temp[index--] = '0';
            }
            else {
                temp[index--] = '1';
            }
        }
        strcpy_s(huffmantree[i].code, alphasizes, &temp[index+1]); //把整个huffman编
        码弄到对应的叶子结点
    }
}
```

part3:压缩文件

利用得到的Huffman编码，然后遍历整个文件，逐步将其翻译成二进制文件。

```
void huffmancompress(char* source, char* des) {
    FILE* fin, * fzip;
    alphafreq* freqtable;
    int alphavariety;
    freqtable = countalpha(source, alphavariety);
    huffman* sourcehuf;
    int* hufindex = new int[256];
    for (int i = 0; i < 256; i++) {
        hufindex[i] = 0;
    }
    //构造huffman树，huffman表，huffman编码，并且实现打印
    sourcehuf = sethuffmantable(freqtable, alphavariety, hufindex);
    createhuffmantree(sourcehuf, alphavariety, 2 * alphavariety - 1);
    createhuffmancode(sourcehuf, alphavariety);
    showmyhuff(sourcehuf, alphavariety);

    //文件头相关参数的设定，解压的时候用于识别
    filehead info = {'h', 'u', 'f'};
}
```

```

info.alphaVariety = alphavariety;
info.lastValidBit = getlastbit(sourcehuf, alphavariety);
info.filelength = 0;
for (int i = 0; i < alphavariety; i++) {
    info.filelength += sourcehuf[i].count * sizeof(unsigned char);
}

fin = fopen(source, "rb");
fzip = fopen(des, "wb");

//给文件头部写入元数据
fwrite(&info, sizeof(filehead), 1, fzip);
//给元数据后写字符种类和频度，解压缩时需要用这些生成一模一样新的哈夫曼树
fwrite(freqtable, sizeof(alphafreq), alphavariety, fzip);

//开始过数据
unsigned char codebuf;
int flag = 0; //flag=8 表明已经达到1个字节 写入文件
char* hufcode;
int value = 0;

codebuf = fgetc(fin);
while (!feof(fin)) {
    hufcode = sourcehuf[hufindex[codebuf]].code;
    for (int i = 0; hufcode[i]; i++) {
        if (hufcode[i] == '1') { //如果这一位huffman码是'1'，则value左移1位，并且将
//最左边设为1
            value <<= 1;
            value |= 1;
        }
        else {
            value <<= 1; //如果这一位huffman码是'0'，则value左移1位。
        }
        flag++;
        //达到1个字节就写到文件
        if (flag >= 8) {
            fwrite(&value, sizeof(unsigned char), 1, fzip);
            flag = 0;
        }
    }
    codebuf = fgetc(fin);
}
//如果最后还有数据，则将其移动到最前端，并且写入文件
if (flag) {
    value <<= (8 - info.lastValidBit);
    fwrite(&value, sizeof(unsigned char), 1, fzip);
}

fclose(fin);
fclose(fzip);
delete sourcehuf;
}

```

part4 解压文件

4.1读取文件头

解压时需要先读取文件头部的一些文件

读取的filehead用于判断是否为规定的格式,字符个数, 文件长度, 最后一位的有效位数等信息。

读取的alphafreq*用于和字符总数共同构建Huffman树

```
filehead readfilehead(char* source) {
    FILE* toread;
    toread = fopen(source, "rb");
    filehead info;
    fread(&info, sizeof(filehead), 1, toread);
    fclose(toread);
    return info;
}

alphafreq* readalphafreq(char* source, int &alphavariety, filehead base) {
    FILE* toread;
    toread = fopen(source, "rb");
    alphavariety = base.alphavariety;
    alphafreq* result = new alphafreq[alphavariety];
    for (int i = 0; i < alphavariety; i++) {
        result[i].ch = 0;
        result[i].freq = 0;
    }
    fseek(toread, sizeof(filehead), SEEK_SET);
    fread(result, sizeof(alphafreq), alphavariety, toread);
    fclose(toread);
    return result;
}
```

4.2进行Huffman编码的解压

```
void huf_decode(char* source, char* des) {
    filehead info;
    //读取文件头的一些信息
    info = readfilehead(source);

    //检测文件是否正确
    char* origin = "huf";
    for (int i = 0; i < 3; i++) {
        if (origin[i] != info.flag[i]) {
            cout << "错误的文件";
            exit(0);
        }
    }

    //读取字符种类数
```

```

    alphafreq* alphainfo;
    int alphavariety = 0;
    alphavariety = info.alphavariety;
    alphainfo = readalphafreq(source, alphavariety, info);

    //构建Huffman树和编码
    int hufindex[256] = { 0 }; int lastbit;
    huffman* datahuf;
    datahuf = sethuffmantable(alphainfo, alphavariety, hufindex);
    createhuffmantree(datahuf, alphavariety, 2 * alphavariety - 1);
    createhuffmancode(datahuf, alphavariety);
    showmyhuff(datahuf, alphavariety);

    //开始解码初始化
    int root = 2 * alphavariety - 2; //start point
    bool finish = false;
    FILE* fzip, * forigin;
    fzip = fopen(source, "rb");
    forigin = fopen(des, "wb");

    long filesize, currentplace;
    fseek(fzip, 0, SEEK_END);
    filesize = ftell(fzip);
    fseek(fzip, sizeof(filehead) + alphavariety * sizeof(alphafreq), SEEK_SET);
    currentplace = ftell(fzip);

    //解码开始
    unsigned char value;
    unsigned char outValue;
    int index = 0;
    int writtehlenth = 0;
    fread(&value, sizeof(unsigned char), 1, fzip);

    while (!finish) {
        if (datahuf[root].lchild == -1 && datahuf[root].rchild == -1) { //到达叶子
            //结点以后写入文件
            outValue = datahuf[root].data;
            fwrite(&outValue, sizeof(unsigned char), 1, forigin);
            root = 2 * alphavariety - 2;
            writtehlenth++;
            //文件长度到达时，则结束循环
            if (writtehlenth == info.filelenth) {
                break;
            }
        }
        //find the hufcode
        //从根部逐渐向下寻找结点
        if (value & (1 << (7 - index))) {
            root = datahuf[root].rchild;
        }
        else {
            root = datahuf[root].lchild;
        }
        if (++index >= 8) {

```

```

        index = 0;
        value = 0;
        fread(&value, sizeof(unsigned char), 1, fzip);
        currentplace = ftell(fzip);
    }

}

fclose(fzip);
fclose(forigin);
delete datahuf;

}

```

调试分析

时空复杂度分析

压缩时，**假设文件长度为 n** ，那么读取了两次，假设每一次读取时间为 $K(n)$ ，时间复杂度为 $2K(n)$ 。**如果文件字符种类为 m** ，那么构建Huffman树的时间复杂度为 $O(m^2)$ 。在将字符翻译成Huffman编码时每一个字符时间为 $O(1)$ (使用哈希表搜索)。将每一个huffman编码翻译为二进制时时间为 $O(\log m)$ ，因此，总的时间复杂度为 $O(2K(n)+m^2+n\log m)$ 。空间复杂度一般可以忽略。

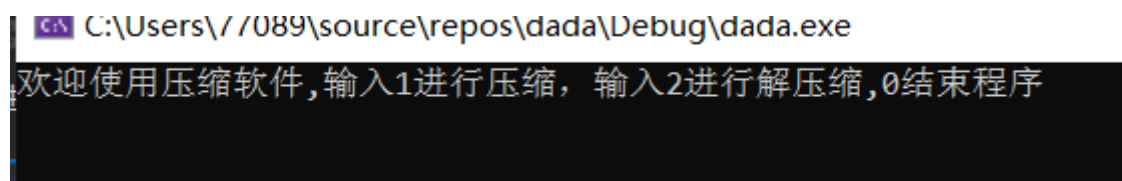
解压时，**假设文件长度为 n** ，只读取了一次，假设每一次读取时间为 $K(n)$ ，时间复杂度为 $K(n)$ 。**如果文件字符种类为 m** ，那么构建Huffman树的时间复杂度为 $O(m^2)$ 。将每一个字符还原回来时间为 $O(\log m)$ ，因此，总的时间复杂度为 $O(K(n)+m^2+n\log m)$ 。空间复杂度一般可以忽略。

问题分析







1. 运行的时候，明明解压的时空复杂度要小于压缩，但是运行的时候时间却远远大于压缩的。
分析：压缩的时候基本都是位运算，基本没有实现内存的移动，而解压的时候涉及到了大量的内存改变，导致花的时间远远变大。这里充分体现了位运算的优势。
2. 一开始把filehead的alphavariety设为unsigned char，在字符为256时会出现问题
分析：unsigned char在256时会自动溢出为0，导致错误，将alphavariety设为int即可解决。

代码测试

1. 压缩pdf，并对其解压



开始界面

 1.pdf	2021/6/5 8:04	Microsoft Edge ...	3,278 KB
 fib	2021/12/14 13:09	文件	1 KB
 fibnew	2021/12/14 15:08	文件	1 KB
 temp	2021/12/14 13:07	文件	3,278 KB
 temp.pdf	2021/12/14 13:08	Microsoft Edge ...	3,278 KB
 temp2.pdf	2021/12/14 16:32	Microsoft Edge ...	3,278 KB

测试文件列表








对1.pdf压缩

```
输入要压缩的文件路径+名字:D:\test\1.pdf
输入要压缩出来的文件路径+名字:D:\test\1compress
```

得到的Huffman树









12390	-1	-1	278	00101010
12657	-1	-1	354	11000011
12483	-1	-1	300	01010110
12435	-1	-1	288	00111111
12591	-1	-1	340	10100110
12306	-1	-1	262	00001011
12590	-1	-1	339	10100100
12492	-1	-1	302	01011011
12328	-1	-1	266	00010011
12355	-1	-1	270	00011011
12525	-1	-1	314	01110011
12510	-1	-1	308	01100111
12642	-1	-1	352	10111110
12493	-1	-1	303	01011101
12482	-1	-1	298	01010010
12383	-1	-1	275	00100101
12623	-1	-1	349	10111000
12590	-1	-1	339	10100101
12603	-1	-1	343	10101100
12571	-1	-1	334	10011010
12420	-1	-1	286	00111011
12628	-1	-1	350	10111010
12452	-1	-1	291	01000101
12655	-1	-1	353	11000001
12287	-1	-1	259	00000101
12533	-1	-1	316	01110111
12575	-1	-1	336	10011110

1compress大小和1.pdf大小相等，压缩的效果不好

 1.pdf	2021/6/5 8:04	Microsoft Edge ...	3,278 KB
 1compress	2021/12/14 17:38	文件	3,278 KB
 fib	2021/12/14 13:09	文件	1 KB
 fibnew	2021/12/14 15:08	文件	1 KB
 temp	2021/12/14 13:07	文件	3,278 KB
 temp.pdf	2021/12/14 13:08	Microsoft Edge ...	3,278 KB
 temp2.pdf	2021/12/14 16:32	Microsoft Edge ...	3,278 KB

对其解压

```
C:\Users\77089\source\repos\dada\Debug\dada.exe
输入要解压的文件路径+名字:D:\test\1compress
输入要解压出来的文件路径+名字:D:\test\1compress.pdf
```

 1.pdf	2021/6/5 8:04	Microsoft Edge ...	3,278 KB
 1compress	2021/12/14 17:38	文件	3,278 KB
 1compress.pdf	2021/12/14 17:39	Microsoft Edge ...	3,278 KB
 fib	2021/12/14 13:09	文件	1 KB
 fibnew	2021/12/14 15:08	文件	1 KB
 temp	2021/12/14 13:07	文件	3,278 KB
 temp.pdf	2021/12/14 13:08	Microsoft Edge ...	3,278 KB
 temp2.pdf	2021/12/14 16:32	Microsoft Edge ...	3,278 KB

國立屏東大學文化創意產業學系

碩士論文

從傳奇小說到劇場動畫：

以跨文本觀點探究《空之境界》之敘事結構

From Legendary Novel to Animated Film Adaptation:
A Trans-textual Analysis on the Narrative Structure of “Kara no Kyokai”



可以正常打开

2.txt文件

C:\Users\77089\source\repos\dada\Debug\dada.exe

输入要压缩的文件路径+名字:D:\test\fib.txt
输入要压缩出来的文件路径+名字:D:\test\fibcompress

名称	修改日期	类型	大小
1.pdf	2021/6/5 8:04	Microsoft Edge ...	3,278 KB
1compress	2021/12/14 17:38	文件	3,278 KB
1compress.pdf	2021/12/14 17:39	Microsoft Edge ...	3,278 KB
fib	2021/12/14 13:09	文件	1 KB
fib.txt	2021/12/14 17:43	文本文档	213 KB
fibnew	2021/12/14 15:08	文件	1 KB
temp	2021/12/14 13:07	文件	3,278 KB
temp.pdf	2021/12/14 13:08	Microsoft Edge ...	3,278 KB
temp2.pdf	2021/12/14 16:32	Microsoft Edge ...	3,278 KB

压缩完

fib.txt	2021/12/14 17:43	文本文档	213 KB
fibcompress	2021/12/14 17:44	文件	123 KB

大小小了将近一半

现在解压

```
输入要解压的文件路径+名字:D:\test\fibcompress
输入要解压出来的文件路径+名字:D:\test\fibdecode.txt
```

fib.txt	2021/12/14 17:43	文本文档	213 KB
fibcompress	2021/12/14 17:44	文件	123 KB
fibdecode.txt	2021/12/14 17:45	文本文档	213 KB

大小不变

文件也可正常打开

3..png文件

```
输入要压缩的文件路径+名字:D:\test\test.png
输入要压缩出来的文件路径+名字:D:\test\testconpress
```

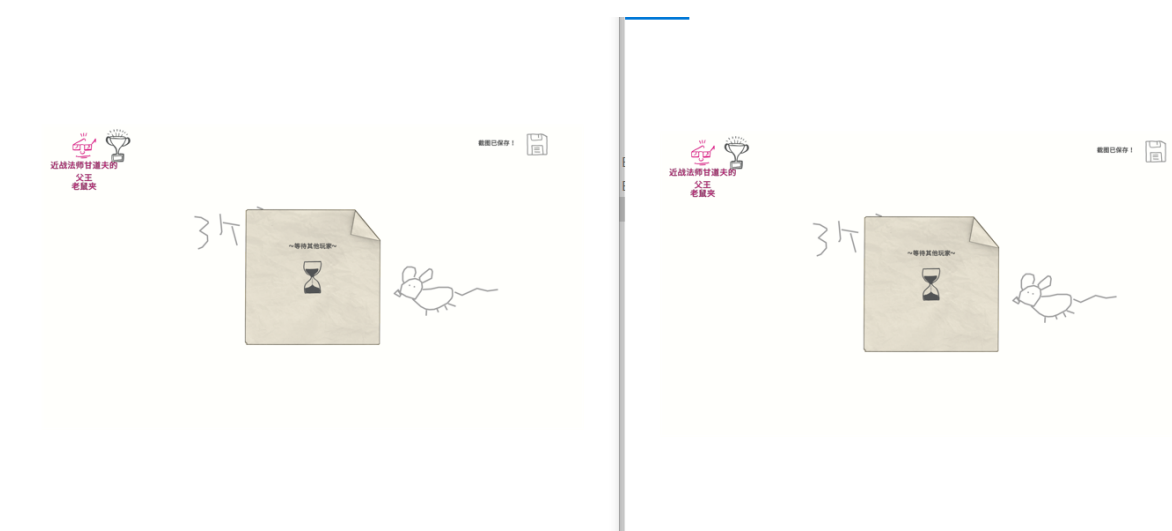
压缩完

test.png	2021/7/30 16:48	PNG 文件	396 KB
testconpress	2021/12/14 17:47	文件	397 KB

小了1kb

现在解压

```
C:\Users\77089\source\repos\dada\Debug\dada.exe
输入要解压的文件路径+名字:D:\test\testconpress
输入要解压出来的文件路径+名字:D:\test\test2.png
```



完全一样

4.MP3文件

输入要压缩的文件路径+名字:D:\test\test.mp3				
输入要压缩出来的文件路径+名字:D:\test\mp3compress				
mp3compress	2021/12/14 17:50	文件	9,524 KB	
temp.pdf	2021/12/14 13:08	Microsoft Edge ...	3,278 KB	
temp2.pdf	2021/12/14 16:32	Microsoft Edge ...	3,278 KB	
test.mp3	2021/12/13 21:40	MP3 文件	9,565 KB	

略微变小

现在解压

mp3compress	2021/12/14 17:50	文件	9,524 KB	
mp3decode.mp3	2021/12/14 17:51	MP3 文件	9,565 KB	
temp.pdf	2021/12/14 13:08	Microsoft Edge ...	3,278 KB	
temp2.pdf	2021/12/14 16:32	Microsoft Edge ...	3,278 KB	
test.mp3	2021/12/13 21:40	MP3 文件	9,565 KB	
test.png	2021/7/30 16:48	PNG 文件	396 KB	

完全一样

5.rar文件

test.rar	2021/12/14 17:52	WinRAR 压缩文件	749 KB	
test2.png	2021/12/14 17:48	PNG 文件	396 KB	
testcompress	2021/12/14 17:47	文件	397 KB	

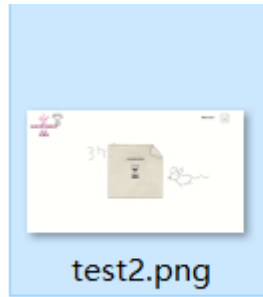
名称	大小	压缩后大小	类型	修改时间	CRC32
..			文件夹		
test2.png	405,007	382,911	PNG 文件	2021/12/14 1...	57004B1F
test.png	405,007	382,911	PNG 文件	2021/7/30 16:...	57004B1F

文件结构一致

解压后也正常



test.png



test2.png

实验总结

在这次实验中，我对Huffman树和Huffman编码有了更深入的了解，自己完成了一个较大的项目，不仅如此，我还对当时学c语言的时候没弄懂的文件操作和位运算有了一定的了解，提高了自己的写代码水平。

附录

实验报告-吕凯盛-PB2008150.pdf

huffman.cpp

huf.exe