

## 多媒体技术实验报告——霍夫曼&算术编码原理实现实验

姓名：杨承翰 学号：210210226 班级：通信 2 班

### 一、实验目的

1. 学习霍夫曼编码原理及使用 C 语言进行霍夫曼编码原理实现
2. 学习算术编码原理及使用 C 语言进行算术编码原理实现

### 二、实验方法概要

我使用 C 语言编写了一个实现霍夫曼编码和算术编码原理的程序，并在 CodeBlocks 软件中进行实验。下面简要概括我的实验方法：

#### （一）霍夫曼编码实现：

1. 构建霍夫曼树
  - 统计输入数据中每个字符出现的频率。
  - 将每个字符看作一个节点，并将其频率作为节点的权值。
  - 以所有节点作为起始状态，每次取出权值最小的两个节点，合并为一个新的节点，并将它们的权值之和作为新节点的权值。
  - 重复上一步，直到剩下一个节点，即为霍夫曼树的根节点。
2. 生成霍夫曼编码表
  - 遍历霍夫曼树，依次记录经过的路径，对左子树的路径标记为 0，右子树的路径标记为 1。
  - 当遍历到叶子节点时，将该节点对应的字符和路径一起存储在编码表中。
3. 对输入数据进行编码
  - 读取输入数据中的每个字符，查找其在编码表中对应的霍夫曼编码，并将这些编码拼接起来，形成压缩后的数据。
4. 进行解码操作
  - 从压缩数据的开头开始，依次读取每个比特位，根据比特位的值决定是遍历左子树还是右子树。
  - 当遇到叶子节点时，输出该节点对应的字符，并回到根节点，继续读取下一个比特位，重复上述过程，直到所有压缩数据被解压缩完毕。

#### （二）算术编码实现：

1. 确定输入数据的概率分布
  - 统计输入数据中每个字符出现的频率或概率。
  - 计算每个字符的概率，可以通过字符在输入数据中出现的次数除以总字符数来得到。
  - 将字符的概率进行归一化，确保所有概率之和为 1。
2. 将概率分布映射到区间
  - 创建一个累积概率分布表，记录每个字符对应的累积概率值。
  - 根据概率分布，计算累积概率分布表中每个字符对应的区间起始值和终止值。
  - 确定字符区间的映射关系，例如可以使用浮点数表示区间的起始值和终止值。

### 3. 对输入的数据进行编码

- 读取输入的字符序列，根据字符的概率分布和区间映射，计算输入字符序列对应的算术编码。
- 初始时，设编码的区间为 $[0, 1)$ 。
- 对于输入字符序列的每个字符，根据字符的概率分布和当前的编码区间，将编码区间缩小为对应字符区间的大小，并更新编码区间的起始值和终止值。
- 重复上述步骤，直到遍历完输入字符序列。

### 4. 进行解码操作

- 使用相同的字符概率分布和区间映射，对编码后的数据进行解码。
- 读取编码的比特流或浮点数，根据当前的编码区间和字符的概率分布，确定该比特流或浮点数属于哪个字符的区间。
- 输出对应的字符，并更新编码区间为该字符区间的大小和位置。
- 重复上述步骤，直到所有编码数据被解码完毕，得到原始的字符序列。

通过以上实验方法，我能够实现霍夫曼编码和算术编码的原理，并对输入的数据进行编码和解码操作。这样可以实现数据的高效压缩和解压缩，减少数据的传输和存储开销。

## 三、实验方法详解

### (一) 霍夫曼编码实现：

```
struct Huff_node
{
    int b;           //用于保存需要编码的字符 由于定义为int因此可以压缩中文 如果定义为char只能压缩英文
    long count;      //文件中该字符出现的次数
    long parent, lch, rch; //定义父节点和左右节点
    char bits[256];   //bits大小决定了可以编码字符数量的上限
};
```

这段代码定义了一个名为 `Huff_node` 的结构体，用于表示霍夫曼编码树的节点。

1. `int b;`：用于保存需要编码的字符。由于定义为 `int` 类型，这意味着该程序可以处理包括中文在内的各种字符。如果定义为 `char` 类型，只能压缩英文字符。
2. `long count;`：表示文件中该字符出现的次数。用于记录字符在输入数据中的频率，以便后续构建霍夫曼树时使用。
3. `long parent, lch, rch;`：分别表示父节点、左子节点和右子节点的索引。这些索引值用于在构建霍夫曼树时建立节点之间的连接关系。
4. `char bits[256];`：用于存储字符的霍夫曼编码。数组的大小 256 决定了可以编码字符数量的上限，每个索引位置存储对应字符的霍夫曼编码。

总体而言，该结构体用于存储霍夫曼编码树的节点信息。通过统计字符出现的频率和构建霍夫曼树，可以生成每个字符对应的霍夫曼编码，并将其存储在该结构体的成员变量 `bits` 中。

```
struct Huff_node Nodes[512], tmp; //节点树
```

构建节点树

```
void printfPercent(int per) //画进度条函数
{
    int i = 0;
    printf("|");
    for(i = 0; i < 10; i++)
    {
        if(i < per/10)
            printf(">");
        else
            printf("-");
    }
    printf("|已完成%d%%\n", per);
}
```

这段代码定义了一个名为`printfPercent`的函数，用于在控制台输出一个进度条来表示任务的完成情况。

1. `void printfPercent(int per)`：这是一个自定义的函数，接受一个整数参数`per`，表示任务完成的百分比。
2. `int i = 0;`：定义一个整数变量`i`并初始化为 0，用于循环控制。
3. `printf("|");`：在控制台输出一个竖线，表示进度条的开始。
4. `for(i = 0; i < 10; i++)`：循环 10 次，控制进度条的长度。
5. `if(i < per/10)`：判断当前循环的索引是否小于任务完成的百分比除以 10，用于确定当前位置是否需要显示">"符号。
6. `printf(">");`：如果满足上述条件，则输出一个">"符号，表示已完成的进度。
7. `else`：如果不满足条件，则执行下面的语句。
8. `printf("-");`：输出一个 "-" 符号，表示未完成的进度。
9. `printf("|已完成%d%%\n", per);`：在控制台输出已完成的百分比，使用`%d`占位符将参数`per`插入到字符串中，并换行。

总体而言，该函数的作用是根据任务的完成百分比，在控制台输出一个长度为 10 的进度条，通过输出">"和 "-" 符号表示任务的完成情况，并在末尾显示任务完成的百分比。这有助于用户直观地了解任务的进展情况。

```
int compress(const char *filename, const char *outputfile)
```

```
{
    char buf[512];
    unsigned char c;
    long l, j, m, n, f;
    long minl, ptl, flength;
    FILE *ifp, *ofp;
    int per = 10;
    ifp = fopen(filename, "rb");
    if (ifp == NULL)
    {
        printf("打开文件失败:%s\n", filename);
        return 0;
    }
    ofp = fopen(outputfile, "wb");
    if (ofp == NULL)
    {
        printf("打开文件失败:%s\n", outputfile);
        return 0;
    }
    flength = 0;
    while (!feof(ifp))
    {
        fread(&c, 1, 1, ifp);
        Nodes[c].count++;
        flength++;
    }
    flength--;

    for (i = 0; i < 512; i++)
    {
        if (Nodes[i].count != 0)
            Nodes[i].b = (unsigned char) i;
        else
            Nodes[i].b = -1;
        Nodes[i].parent = -1;
        Nodes[i].lch = Nodes[i].rch = -1;
    }
}
```

```

for (i = 0; i < 256; i++)
{
    for (j = i + 1; j < 256; j++)
    {
        if (Nodes[i].count < Nodes[j].count)
        {
            tmp = Nodes[i];
            Nodes[i] = Nodes[j];
            Nodes[j] = tmp;
        }
    }
}

for (i = 0; i < 256; i++)
{
    if (Nodes[i].count == 0)
        break;
}

n = i;

m = 2 * n - 1;
for (i = n; i < m; i++)
{
    min1 = max_num;
    for (j = 0; j < i; j++)
    {
        if (Nodes[j].parent != -1) continue;
        if (min1 > Nodes[j].count)
        {
            pt1 = j;
            min1 = Nodes[j].count;
            continue;
        }
    }
    Nodes[i].count = Nodes[pt1].count;
    Nodes[pt1].parent = i;
}

```

具体代码见最后

这段代码实现了一个基于霍夫曼编码原理的压缩程序。主要功能包括：读取原始文件，统计每个字符出现的次数，构建霍夫曼树，设置字符的编码，替换文件中的字符为对应的编码，并将结果写入压缩后存储信息的文件中。具体功能如下：

打开原始文件和压缩后存储信息的文件

统计原始文件中每个字符出现的次数

构建霍夫曼树，求出每个字符的编码

将原始文件中每个字符替换为对应的编码，并写入压缩后存储信息的文件中

关闭文件，返回压缩成功信息。

具体来说，程序首先打开原始文件和压缩后存储信息的文件（第 7、9 行），如果打开失败则输出错误信息并返回。接着，程序循环读取原始文件中的每个字符，并统计每个字符的出现次数（第 28~40 行）。然后，程序按照霍夫曼编码原理，构建霍夫曼树（第 50~77 行），并设置每个字符的编码（第 79~92 行）。接下来，程序利用构造好的霍夫曼树，将原始文件中的每个字符替换为对应的编码，并把结果写入压缩后存储信息的文件中（第 98~165 行）。最后，程序关闭文件，输出压缩后文件的相关信息，并返回压缩成功信息（第 167~174 行）。

```

int uncompress(const char *filename, const char *outputfile)
{
    char buf[255], bx[255];
    unsigned char c;
    char out_filename[512];
    long i, j, m, n, f, p, l;
    long flength;
    int per = 10;
    int len = 0;
    FILE *ifp, *ofp;
    char c_name[512] = {0};
    ifp = fopen(filename, "rb");
    if (ifp == NULL)
    {
        return 0;
    }

    if (outputfile)
        strcpy(out_filename, outputfile);
    else
        strcpy(out_filename, c_name);

    ofp = fopen(out_filename, "wb");
    if (ofp == NULL)
    {
        return 0;
    }

    fseek(ifp, 0, SEEK_END);
    len = ftell(ifp);
    fseek(ifp, 0, SEEK_SET);

    printf("将要读取解压的文件:%s\n", filename);
    printf("当前文件大小为:%d字节\n", len);
    printf("正在解压\n");

    fread(&flength, sizeof(long), 1, ifp);
    fread(&f, sizeof(long), 1, ifp);
    fseek(ifp, f, SEEK_SET);
    fread(&n, sizeof(long), 1, ifp);

    fseek(ifp, f, SEEK_SET);
    fread(&n, sizeof(long), 1, ifp);
    for (i = 0; i < n; i++)
    {
        fread(&Nodes[i].b, 1, 1, ifp);
        fread(&c, 1, 1, ifp);
        p = (long) c;
        Nodes[i].count = p;
        Nodes[i].bits[0] = 0;
        if (p % 8 > 0) m = p / 8 + 1;
        else m = p / 8;
        for (j = 0; j < m; j++)
        {
            fread(&c, 1, 1, ifp);
            f = c;
            _itoa(f, buf, 2);
            f = strlen(buf);
            for (l = 0; l > f; l--)
            {
                strcat(Nodes[i].bits, "0");
            }
            strcat(Nodes[i].bits, buf);
        }
        Nodes[i].bits[p] = 0;
    }

    for (i = 0; i < n; i++)
    {
        for (j = i + 1; j < n; j++)
        {
            if (strlen(Nodes[i].bits) > strlen(Nodes[j].bits))
            {
                tmp = Nodes[i];
                Nodes[i] = Nodes[j];
                Nodes[j] = tmp;
            }
        }
    }

    p = strlen(Nodes[n-1].bits);
    fseek(ifp, 0, SEEK_SET);

```



```

while (1)
{
    while (strlen(bx) < (unsigned int)p)
    {
        fread(&c, 1, 1, ifp);
        f = c;
        itoa(f, buf, 2);
        f = strlen(buf);
        for (l = 8; l > f; l --)
        {
            strcat(bx, "0");
        }
        strcat(bx, buf);
    }
    for (i = 0; i < n; i++)
    {
        if (memcmp(Nodes[i].bits, bx, Nodes[i].count) == 0) break;
    }
    strcpy(bx, bx + Nodes[i].count);
    c = Nodes[i].b;
    fwrite(&c, 1, 1, ofp);
    m++;

    if (100 * m / flength > per)
    {
        printfPercent(per);
        per += 10;
    }
    if (m == flength) break;
}
printfPercent(100);

fclose(ifp);
fclose(ofp);

printf("解压后文件为:%s\n", out_filename);
printf("解压后文件为:%d字节\n", flength);

return 1; // 输出成功信息
}

```

这段代码实现了文件的解压缩功能，具体流程如下：

1. 打开待解压的文件并读取文件长度。
2. 从文件中读取霍夫曼编码表。首先读取原文件长，然后读取原文件使用过字符的个数，并逐个读取每个字符对应的霍夫曼编码信息，包括编码长度、编码位数和编码值，并将这些信息保存到一个结构体数组中。
3. 根据保存的编码信息按照编码长度从小到大排序，以便后面进行解码时查找编码值。
4. 从文件中读取压缩后的数据，并一位一位地进行解码。具体过程是，从文件中读取一定数量的编码位数，将其与编码表中的编码进行比较，直到找到一个匹配的编码，然后将其对应的字符写入输出文件中。
5. 解压缩完成后关闭文件，并输出解压缩的相关信息，如解压后文件名、文件大小等。

值得注意的是，在程序执行过程中，还有一些用于输出进度的代码，例如每处理 10% 的数据就会输出相应的百分比。

```
int main(int argc, const char *argv[])
{
    memset(&Nodes, 0, sizeof(Nodes));
    memset(&tmp, 0, sizeof(tmp));

    compress("测试文档.txt", "测试文档.txt.zip");
    uncompress("测试文档.txt.zip", "测试文档.txt 解压后.txt");
    system("pause");

    return 0;
}
```

这段代码是主函数，实现了对文件的压缩和解压缩操作。具体流程如下：

1. 使用`memset`函数将`Nodes`和`tmp`结构体数组初始化为 0，用于存储霍夫曼编码表和临时变量。
2. 调用`compress`函数进行文件压缩操作，传入待压缩的文件名和输出的压缩文件名。
3. 调用`uncompress`函数进行文件解压缩操作，传入待解压的压缩文件名和输出的解压后文件名。
4. 使用`system`函数调用系统命令`"pause"`，暂停程序的执行，以便在控制台中查看输出结果。
5. 返回 0，表示程序执行成功。

总结起来，这段代码的作用是将指定的文件进行压缩和解压缩操作，并在控制台中输出相应的信息和结果。



## (二) 算术编码实现:

```
int main()
{
    int i, j;
    printf("input the length of char set:\n");
    scanf("%d", &chNum);
    getchar();
    printf("input the char and its p\n");
    for (i=0; i < chNum; i++) {
        printf("input char: ");
        scanf("%c", &chSet[i]);
        getchar();
        printf("input its p: ");
        scanf("%f", &P[i]);
        getchar();
        printf("\n");
    }
    pZone[0] = 0;
    for (i=1; i < chNum; ++i)
        pZone[i] = pZone[i-1] + P[i-1];
    printf("input the string \n");
    gets(inStr);
    strLen = strlen(inStr);
    printf("the string is: \n");
    puts(inStr);
    printf("***** compressed *****\n");
    compressed();
    printf("***** uncompressed *****\n");
    uncompressed();
    return 0;
}
```

这段代码是一个使用算术编码原理实现的程序，主要功能是对输入的字符集进行编码和解码。下面对代码进行逐行解释：

1. `int i, j;` - 定义两个整型变量 i 和 j，用于循环计数。
2. `printf("input the length of char set:\n");` - 输出提示信息，要求用户输入字符集的长度。
3. `scanf("%d", &chNum);` - 通过标准输入获取用户输入的字符集长度，并将其存储在变量 chNum 中。
4. `getchar();` - 获取并丢弃输入缓冲区中的换行符。
5. `printf("input the char and its p\n");` - 输出提示信息，要求用户输入每个字符及其概率。
6. `for (i=0; i < chNum; i++) {` - 进入一个循环，循环次数为字符集长度。
7. `printf("input char: ");` - 输出提示信息，要求用户输入一个字符。
8. `scanf("%c", &chSet[i]);` - 通过标准输入获取用户输入的字符，并将其存储在字符数组 chSet 的第 i 个位置。
9. `getchar();` - 获取并丢弃输入缓冲区中的换行符。
10. `printf("\ninput its p: ");` - 输出提示信息，要求用户输入该字符的概率。
11. `scanf("%f", &P[i]);` - 通过标准输入获取用户输入的概率，并将其存储在浮点数数组 P 的第 i 个位置。
12. `getchar();` - 获取并丢弃输入缓冲区中的换行符。
13. `printf("\n");` - 输出换行符，用于提升可读性。
14. `pZone[0] = 0;` - 将概率区间 pZone 的第一个元素初始化为 0。
15. `for (i=1; i < chNum; ++i)` - 进入一个循环，循环次数为字符集长度减 1。
16. `pZone[i] = pZone[i-1] + P[i-1];` - 根据概率值计算概率区间，每个字符对应一个区间。
17. `printf("input the string \n");` - 输出提示信息，要求用户输入待压缩的字符串。
18. `gets(inStr);` - 通过标准输入获取用户输入的字符串，并将其存储在字符数组 inStr 中。
19. `strLen = strlen(inStr);` - 计算输入字符串的长度，并将其存储在变量 strLen 中。
20. `printf("the string is: \n");` - 输出提示信息，显示输入的字符串。

21. `puts(inStr);` - 输出输入的字符串。
22. `printf("\*\*\*\*\* compress \*\*\*\*\*\n");` - 输出提示信息，表示开始进行压缩。
23. `compress();` - 调用压缩函数进行算术编码压缩。
24. `printf("\n\*\*\*\*\* uncompress \*\*\*\*\*\n");` - 输出提示信息，表示开始进行解压缩。
25. `uncompress();` - 调用解压缩函数进行算术编码解压缩。
26. `return 0;` - 程序执行完毕，返回 0 作为退出状态码。

```
void compress()
{
    float low = 0, high = 1;
    float l, H, zlen = 1;
    float cp; //输入字符的概率
    float result; //结果
    int i, j;
    for (i=0; i < strlen; i++) {
        for (j=0; j < chNum; j++) {
            if (inStr[i] == chSet[j]) {
                //cp = P[i];

                //l = pZone[i];
                low = low + zlen * pZone[j];
                zlen *= P[j];
                break;
            }
        }
        //low = low + zlen * l;
        //zlen *= cp;
    }
    result = low;
    printf("the result is %f\n", result);
    infoLen = log(1/zlen) / log(2); //计算信息总量
    if (infoLen > (int)infoLen)
        infoLen = (int)infoLen + 1;
    else
        infoLen = (int)infoLen;
    //***** 计算二进制 *****/
    for (i=0; i < infoLen; i++) {
        result *= 2;
        if (result > 1) {
            result = result - 1;
            binary[i] = 1;
        } else if (result < 1) {
            binary[i] = 0;
        } else {
            break;
        }
    }
    if (i >= infoLen) {
        for (j=i; j >= 1; j--) {
            binary[j-1] = (binary[j-1]+1)%2;
            if (binary[j-1] == 1)
                break;
        }
    }
    printf("***** the compress result:*****\n");
    for (j=0; j < i; j++)
        printf("%d ", binary[j]);
}
```

这段代码是一个使用算术编码原理实现的压缩函数。以下是对每个部分功能的解释：

#### 1. 定义变量和初始化：

- `low` 和 `high` 是算术编码范围的上下界，初始值分别为 0 和 1。
- `l` 和 `H` 是中间结果变量。
- `zlen` 是当前编码区间的长度，初始值为 1。
- `cp` 是输入字符的概率。
- `result` 是最终压缩结果。
- `i` 和 `j` 是循环计数器。

#### 2. 循环遍历输入字符串：

- 对于每个字符，在字符集中寻找对应的字符。
- 根据字符在字符集的位置计算低界 `low` 和长度 `zlen`。

- 执行完内层循环后，更新 `low` 和 `zlen` 的值。
- 3. 计算结果和信息量：
  - 将 `low` 的值赋给结果变量 `result`。
  - 使用对数计算香农信息量 `infoLen`。
  - 如果 `infoLen` 不是整数，则向上取整。
- 4. 转换为二进制形式：
  - 使用乘 2 法将小数 `result` 转换为二进制，并存储在数组 `binary` 中。
  - 若 `result` 大于 1，则减去 1，将对应位置的二进制设为 1。
  - 若 `result` 小于 1，则将对对应位置的二进制设为 0。
  - 若 `result` 等于 1，则跳出循环（此时压缩结果已经达到最大压缩率）。
- 5. 处理进位：
  - 若循环完成后没有跳出循环，则说明还有精度未被转换为二进制码。
  - 从最高位开始检查，若某一位为 0，则将其设为 1，并退出循环；否则将该位设为 0，继续向下检查。
- 6. 输出结果：
  - 将二进制码输出至屏幕。

```
void uncompress()
{
    int i, j;
    float w = 0.5;
    float deResult = 0;
    float newLow, newLen;
    float low = 0, zlen = 1;
    /***** binary to ten *****/
    for (i = 0; i < infoLen; i++) {
        deResult += w * binary[i];
        w *= 0.5;
    }
    printf("uncompress to ten: %f\n", deResult);
    printf("uncompress result: \n");
    for (i = 0; i < strLen; i++) {
        for (j = chNum; j > 0; j--) {
            newLow = low;
            newLen = zlen;
            newLow += newLen * pCode[j-1];
            newLen *= P[j-1];
            if (deResult >= newLow) {
                low = newLow;
                zlen = newLen;
                print("%c ", chSet[j-1]);
                break;
            }
        }
    }
}
```

这段代码是一个使用算术编码原理实现的解压缩函数。以下是对每个部分功能的解释：

1. 定义变量和初始化：
  - `w` 是二进制转十进制系数，初始值为 0.5。
  - `deResult` 是解压后的十进制结果，初始值为 0。
  - `newLow` 和 `newLen` 是中间结果变量。
  - `low` 和 `zlen` 是当前编码区间的上界和长度，初始值分别为 0 和 1。
  - `i` 和 `j` 是循环计数器。
2. 将二进制码转换为十进制：
  - 对于每个二进制位，计算其对应的十进制值，并将其累加到 `deResult` 中。
3. 解压缩过程：
  - 对于每个字符，从字符集的最后一个字符开始向前查找。
  - 根据当前编码区间的上界和长度计算新的区间上下界。
  - 若 `deResult` 大于等于新区间的下界，则说明要解压出的字符是该字符。







- 更新当前区间上下界，并输出解压出的字符。

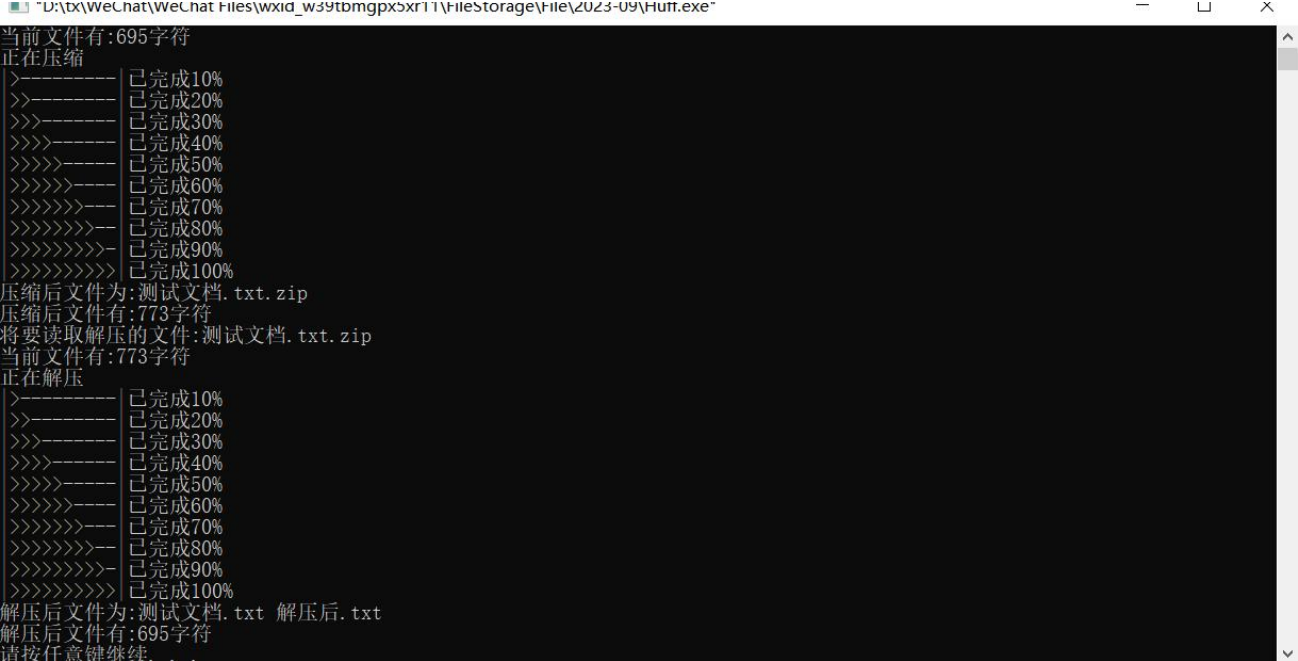
#### 4. 输出结果：

- 将解压缩后的字符序列输出至屏幕。

## 四、实验结果

### (一) 霍夫曼编码实现：

 Huff	2023/10/10 21:26	C 文
 Huff	2023/10/10 22:24	应用
 Huff.o	2023/10/10 22:24	O 文
 测试文档	2023/10/10 22:29	文本
 测试文档.txt 解压后	2023/10/10 22:29	文本
 测试文档.txt	2023/10/10 22:29	ZIP 文

```

D:\tx\WeChat\WeChat Files\wxid_w39tbgmpx5xr11\FileStorage\File\2023-09\Huff.exe
当前文件有:695字符
正在压缩
>----- 已完成10%
>>----- 已完成20%
>>>----- 已完成30%
>>>>----- 已完成40%
>>>>>----- 已完成50%
>>>>>>----- 已完成60%
>>>>>>>----- 已完成70%
>>>>>>>>----- 已完成80%
>>>>>>>>>----- 已完成90%
>>>>>>>>>>----- 已完成100%
压缩后文件为:测试文档.txt.zip
压缩后文件有:773字符
将要读取解压的文件:测试文档.txt.zip
当前文件有:773字符
正在解压
>----- 已完成10%
>>----- 已完成20%
>>>----- 已完成30%
>>>>----- 已完成40%
>>>>>----- 已完成50%
>>>>>>----- 已完成60%
>>>>>>>----- 已完成70%
>>>>>>>>----- 已完成80%
>>>>>>>>>----- 已完成90%
>>>>>>>>>>----- 已完成100%
解压后文件为:测试文档.txt 解压后.txt
解压后文件有:695字符
请按任意键继续. . .
  
```

## 测试文档

测试文档 - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

29 1 39 0 12 3 12 392 10 29 3 1 27 5 79 17 111111000000100101001000111001

天地玄黄，宇宙洪荒。日月盈昃，辰宿列张。  
寒来暑往，秋收冬藏。闰余成岁，律吕调阳。  
云腾致雨，露结为霜。金生丽水，玉出昆冈。  
剑号巨阙，珠称夜光。果珍李柰，菜重芥姜。  
海咸河淡，鳞潜羽翔。龙师火帝，鸟官人皇。  
始制文字，乃服衣裳。推位让国，有虞陶唐。  
吊民伐罪，周发殷汤。坐朝问道，垂拱平章。  
爱育黎首，臣伏戎羌。遐迩一体，率宾归王。  
鸣凤在竹，白驹食场。化被草木，赖及万方。  
盖此身发，四大五常。恭惟鞠养，岂敢毁伤。

## 解压后

测试文档.txt 解压后 - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

29 1 39 0 12 3 12 392 10 29 3 1 27 5 79 17 111111000000100101001000111001

天地玄黄，宇宙洪荒。日月盈昃，辰宿列张。  
寒来暑往，秋收冬藏。闰余成岁，律吕调阳。  
云腾致雨，露结为霜。金生丽水，玉出昆冈。  
剑号巨阙，珠称夜光。果珍李柰，菜重芥姜。  
海咸河淡，鳞潜羽翔。龙师火帝，鸟官人皇。  
始制文字，乃服衣裳。推位让国，有虞陶唐。  
吊民伐罪，周发殷汤。坐朝问道，垂拱平章。  
爱育黎首，臣伏戎羌。遐迩一体，率宾归王。  
鸣凤在竹，白驹食场。化被草木，赖及万方。  
盖此身发，四大五常。恭惟鞠养，岂敢毁伤。

## (二) 算术编码实现:

```
D:\LEA\fuia\bin\Debug\fuia.exe
input the length of char set:
4
input the char and its p
input char: 0
input its p: 0.1
input char: 1
input its p: 0.2
input char: 2
input its p: 0.5
input char: 6
input its p: 0.2
input the string
210210226
the string is:
210210226
***** compress *****
the result is 0.353565
***** the compress result*****
0 1 0 1 1 0 1 0 1 0 0 0 0 1 1 0 1
***** uncompress *****
uncompress to ten:0.353565
uncompress result:
2 1 0 2 1 0 2 2 6
Process returned 0 (0x0)   execution time : 21.408 s
Press any key to continue.
```

## 五、代码

### (一) 霍夫曼编码实现:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define max_num 999999999
struct Huff_node
{
    int b; //用于保存需要编码的字符 由于定义为 int 因此可以压缩
    long count; //文件中该字符出现的次数
    long parent, lch, rch; //定义父节点和左右节点
    char bits[256]; //bits 大小决定了可以编码字符数量的上限
};

struct Huff_node Nodes[512], tmp; //节点树

void printfPercent(int per) //画进度条函数
{
    int i = 0;
    printf("|");
    for(i = 0; i < 10; i++)
    {
        if(i < per/10)
            printf(">");
        else
            printf("-");
    }
    printf("|已完成%d%%\n",per);
}

//函数: compress()
//作用: 读取文件内容并加以压缩
//输入 filename 文件名, 创建并输出压缩文件名为 outputfile
int compress(const char *filename,const char *outputfile)
{
    char buf[512]; //为了防止越界, buf 大小需要时
    Huff_node.bits的两倍 因为节点个数是 2*需编码字符数-1
    unsigned char c;
```



```

long i, j, m, n, f;
long min1, pt1, flength;
FILE *ifp, *ofp;
int per = 10;
ifp = fopen(filename, "rb");           //打开原始文件
if (ifp == NULL)
{
    printf("打开文件失败:%s\n", filename);
    return 0;                         //如果打开失败, 则输出错误信息
}
ofp = fopen(outputfile, "wb");         //打开压缩后存储信息的文件
if (ofp == NULL)
{
    printf("打开文件失败:%s\n", outputfile);
    return 0;
}
flength = 0;
while (!feof(ifp))                    //feof(FILE *fp)函数判断文本是否结束 如果
结束返回 1 否则返回 0
{
    fread(&c, 1, 1, ifp);              // fread(void *buffer, int size, int count, FILE
*fp)从 fp 所指向文件的当前位置开始, 一次读入 size 个字节, 重复 count 次, 并将读入的数据
存放 to buffer 中
    Nodes[c].count ++;                 //读文件 由于每次只读了 8 位所以可以
直接用 c 当做索引
    flength ++;                        //记录文件的字符总数
}
flength --;

for (i = 0; i < 512; i ++)             //HUFFMAN 算法中节点初始化
{
    if (Nodes[i].count != 0)
        Nodes[i].b = (unsigned char) i;
    else
        Nodes[i].b = -1;               //如果没有这个字符就记为-1
        Nodes[i].parent = -1;
        Nodes[i].lch = Nodes[i].rch = -1; //所有节点都初始化为-1
}

for (i = 0; i < 256; i ++)             //冒泡法将节点按出现次数排序 低位放出
现次数多的
{

```

```

for (j = i + 1; j < 256; j++)
{
    if (Nodes[i].count < Nodes[j].count)
    {
        tmp = Nodes[i];
        Nodes[i] = Nodes[j];
        Nodes[j] = tmp;
    }
}

for (i = 0; i < 256; i++) //统计文本中使用过的字符的数量
{
    if (Nodes[i].count == 0) //上面已经按字符出现次数排序，因此找到第一个 0 就知道文本中使用的字符数量
        break;
}

n = i; //n 代表文本中使用过的字符的数量 m
是所有节点的个数

Nodes
m = 2 * n - 1; //<-- 一共 256 个子节点按照出现次数
由大到小排序-->|<-- 还有 256 个待填入数据的父节点-->|
for (i = n; i < m; i++) // [x x x x .....x o o o
o .....o]
{
    //下面两个 for 是构建霍夫曼树的过程

    min1 = max_num;
    for (j = 0; j < i; j++) //for 循环找出没有父节点的节点中出现次数
        最少的那个
        {
            if (Nodes[j].parent != -1) continue; //如果不等于-1 就证明这个节点已经有了父节点 就跳过
            if (min1 > Nodes[j].count)
            {
                pt1 = j;
                min1 = Nodes[j].count;
                continue;
            }
        }
    }
}

```

```

    Nodes[i].count = Nodes[pt1].count;          //注意此处的 i 是从 n 开始的 将 Nodes[i] 设为
父节点 上面 for 循环中找到的最小值设为左子节点 下面 for 循环找到的最小值设为右子节点
    Nodes[pt1].parent = i;
    Nodes[i].lch = pt1;
    min1 = max_num;
    for (j = 0; j < i; j++)
    {
        if (Nodes[j].parent != -1) continue;
        if (min1 > Nodes[j].count)
        {
            pt1 = j;
            min1 = Nodes[j].count;
            continue;
        }
    }
    Nodes[i].count += Nodes[pt1].count;
    Nodes[i].rch = pt1;
    Nodes[pt1].parent = i;
}

```

for (i = 0; i < n; i++) //用构造的 HUFFMAN 树设置字符的编码,  
注意此个循环的次数为文本中出现的字符个数 n

```

{
    f = i;
    Nodes[i].bits[0] = 0;
    while (Nodes[f].parent != -1)
    {
        j = f;
        f = Nodes[f].parent;
        if (Nodes[f].lch == j)          //判断该节点位于父节点的左边
        {
            j = strlen(Nodes[i].bits);
            memmove(Nodes[i].bits + 1, Nodes[i].bits, j + 1);
            Nodes[i].bits[0] = '0';
        }
        else
        {
            j = strlen(Nodes[i].bits);
            memmove(Nodes[i].bits + 1, Nodes[i].bits, j + 1);
            Nodes[i].bits[0] = '1';
        }
    }
}

```

```

}

//下面的就是读原文件的每一个字符，按照设置好的编码替换文件中的字符
fseek(ifp, 0, SEEK_SET); //将指针定在文件起始位置
fseek(ofp, 8, SEEK_SET);
buf[0] = 0;
f = 0;
pt1 = 8; //这里从 8 开始因为前 8 个
字节需要写入压缩文件的文件头

printf("读取将要压缩的文件:%s\n", filename);
printf("当前文件有:%d 字符\n", flength);
printf("正在压缩\n");

while (!feof(ifp))
{
    c = fgetc(ifp); //从文件中读取的 8 位数据
    f++; //迭代器用于判断文件是
否读完
    for (i = 0; i < n; i++)
    {
        if (c == Nodes[i].b) break;
    }
    strcat(buf, Nodes[i].bits);
    j = strlen(buf);
    c = 0;
    while (j >= 8) //当剩余字符数量不
小于 8 个时
    {
        for (i = 0; i < 8; i++) //按照八位二进制数转化
成十进制 ASCII 码写入文件一次进行压缩
        {
            if (buf[i] == '1') c = (c << 1) | 1;
            else c = c << 1;
        }
        fwrite(&c, 1, 1, ofp);
        pt1++; //用于记录压缩后字
符数

        strcpy(buf, buf + 8);
        j = strlen(buf);
    }
    if(100 * f/flength > per) //输出压缩进度

```

```

    {
        printfPercent(per);
        per += 10;
    }
    if (f == flength)
        break;
}

    if (j > 0) //当剩余字符数量
        少于 8 个时
        {
            strcat(buf, "00000000");
            for (i = 0; i < 8; i++)
            {
                if (buf[i] == '1') c = (c << 1) | 1;
                else c = c << 1; //对不足的位数进
            }
            fwrite(&c, 1, 1, ofp);
            pt1++;
        }

    printfPercent(100);

    fseek(ofp, 0, SEEK_SET); //将编码信息写入
    存储文件
    fwrite(&flength, sizeof(fllength), 1, ofp);
    fwrite(&pt1, sizeof(long), 1, ofp);
    fseek(ofp, pt1, SEEK_SET);
    fwrite(&n, sizeof(long), 1, ofp);
    for (i = 0; i < n; i++) //将编码的字符和对应的
        编码依次写入压缩文件中
        {
            tmp = Nodes[i];

            fwrite(&(tmp.b), 1, 1, ofp);
            pt1++;
            c = strlen(tmp.bits);
            fwrite(&c, 1, 1, ofp);
            pt1++;
            j = strlen(tmp.bits);

```

```

        if (j % 8 != 0)                                //当位数不满 8 时,
        对该数进行补零操作
        {
            for (f = j % 8; f < 8; f++)
                strcat(tmp.bits, "0");
        }

        while (tmp.bits[0] != 0)
        {
            c = 0;
            for (j = 0; j < 8; j++)
            {
                if (tmp.bits[j] == '1') c = (c << 1) | 1;
                else c = c << 1;
            }
            strcpy(tmp.bits, tmp.bits + 8);                //注意此处 Nodes[i].bits 是地址
            fwrite(&c, 1, 1, ofp);                        //将所得
的编码信息写入文件
            pt1++;
        }

        //Nodes[i] = tmp;
    }
    fclose(ifp);
    fclose(ofp);                                        //关闭
文件

    printf("压缩后文件为:%s\n",outputfile);
    printf("压缩后文件有:%d 字符\n",pt1 + 4);

    return 1;                                          //返回压缩成功信息
}

```

```

//函数: uncompress()
//作用: 解压缩文件, 并将解压后的内容写入新文件
int uncompress(const char *filename,const char *outputfile)
{
    char buf[255], bx[255];
    unsigned char c;
    char out_filename[512];

```

```

long i, j, m, n, f, p, l;
long flength;
int per = 10;
int len = 0;
FILE *ifp, *ofp;
char c_name[512] = {0};
ifp = fopen(filename, "rb"); //打开文件
if (ifp == NULL)
{
    return 0; //若打开失败，则输出错误信息
}

//读取原文件长

if(outputfile)
    strcpy(out_filename,outputfile);
else
    strcpy(out_filename,c_name);

ofp = fopen(out_filename, "wb"); //打开
文件
if (ofp == NULL)
{
    return 0;
}

fseek(ifp,0,SEEK_END);
len = ftell(ifp); // ftell()
用于得到文件位置指针当前位置相对于文件首的偏移字节数
fseek(ifp,0,SEEK_SET);

printf("将要读取解压的文件:%s\n",filename);
printf("当前文件有:%d 字符\n",len);
printf("正在解压\n");

fread(&flength, sizeof(long), 1, ifp); //读取原文件
长
fread(&f, sizeof(long), 1, ifp); //原文件编码
的长度
fseek(ifp, f, SEEK_SET);
fread(&n, sizeof(long), 1, ifp); //读取原文
件使用过字符的个数
for (i = 0; i < n; i++) //读取压缩

```



文件内容并转换成二进制码

```
{
    fread(&Nodes[i].b, 1, 1, ifp);
    fread(&c, 1, 1, ifp);
    p = (long) c;
    Nodes[i].count = p;
    Nodes[i].bits[0] = 0;
    if (p % 8 > 0) m = p / 8 + 1;
    else m = p / 8;
    for (j = 0; j < m; j++)
    {
        fread(&c, 1, 1, ifp);
        f = c;
        _itoa(f, buf, 2);
        f = strlen(buf);
        for (l = 8; l > f; l--)
        {
            strcat(Nodes[i].bits, "0");
```

//位数不足,

执行补零操作

```
        }
        strcat(Nodes[i].bits, buf);
    }
    Nodes[i].bits[p] = 0;
}

for (i = 0; i < n; i++)
{
    for (j = i + 1; j < n; j++)
    {
        if (strlen(Nodes[i].bits) > strlen(Nodes[j].bits))
        {
            tmp = Nodes[i];
            Nodes[i] = Nodes[j];
            Nodes[j] = tmp;
        }
    }
}

p = strlen(Nodes[n-1].bits);
fseek(ifp, 8, SEEK_SET);
m = 0;
bx[0] = 0;
```

```

while (1)
{
    while (strlen(bx) < (unsigned int)p)
    {
        fread(&c, 1, 1, ifp);
        f = c;
        _itoa(f, buf, 2);
        f = strlen(buf);
        for (l = 8; l > f; l --)
        {
            strcat(bx, "0");
        }
        strcat(bx, buf);
    }
    for (i = 0; i < n; i ++)
    {
        if (memcmp(Nodes[i].bits, bx, Nodes[i].count) == 0) break;
    }
    strcpy(bx, bx + Nodes[i].count);
    c = Nodes[i].b;
    fwrite(&c, 1, 1, ofp);
    m ++;

    if(100 * m/flength > per)
    {
        printfPercent(per);
        per += 10;
    }
    if (m == flength) break;
}
printfPercent(100);

fclose(ifp);
fclose(ofp);

printf("解压后文件为:%s\n",out_filename);
printf("解压后文件有:%d 字符\n",flength);

return 1;                //输出成功信息
}

```

```
int main(int argc,const char *argv[])
{
    memset(&Nodes,0,sizeof(Nodes));
    memset(&tmp,0,sizeof(tmp));

    compress("测试文档.txt","测试文档.txt.zip");
    uncompress("测试文档.txt.zip","测试文档.txt 解压后.txt");
    system("pause");

    return 0;
}
```

## (二) 算术编码实现:

```
#include <stdio.h>
#include <string.h>
#include <math.h>
char inStr[100], chSet[20]; //输入字符串和字符集
float P[20]; //每个字符的概率
float pZone[20]; //概率区间
int strLen; //输入字符串长度
int chNum; //字符集中字符个数
int binary[100];
float infoLen; //信息量大小
void compress(); //编码函数
void uncompress(); //解码函数
int main()
{
    int i,j;
    printf("input the length of char set:\n");
    scanf("%d", &chNum);
    getchar();
    printf("input the char and its p\n");
    for (i=0; i < chNum; i++) {
        printf("input char: ");
        scanf("%c", &chSet[i]);
        getchar();
        //printf("sssss%c ", chSet[i]);
        printf("\ninput its p: ");
        scanf("%f",&P[i]);
        getchar();
        printf("\n");
    }
    pZone[0] = 0;
    for (i=1; i < chNum; ++i)
        pZone[i] = pZone[i-1] + P[i-1];
    printf("input the string \n");
    gets(inStr);
    strLen = strlen(inStr);
    /***** test *****/
    printf("the string is: \n");
    puts(inStr);
    printf("***** compress *****\n");
    compress();
}
```

```

printf("\n***** uncompress *****\n");
uncompress();
return 0;
}
void compress()
{
float low = 0, high = 1;
float L, H, zlen = 1;
float cp; //输入字符的概率
float result; //结果
int i, j;
for (i=0; i < strLen; i++) {
for (j=0; j < chNum; j++) {
if (inStr[i] == chSet[j]) {
//cp = P[j];

//L = pZone[j];
low = low + zlen * pZone[j];
zlen *= P[j];
break;
}
}
//low = low + zlen * L;
//zlen *= cp;
}
result = low;
printf("the result is %f\n", result);
infoLen = log(1/zlen) / log(2); //计算香农信息量
if(infoLen > (int)infoLen)
infoLen = (int)infoLen + 1;
else
infoLen = (int)infoLen;
/***** 转二进制 *****/
for (i=0; i < infoLen; i++) {
result *= 2;
if (result > 1) {
result = result - 1;
binary[i] = 1;
} else if (result < 1) {
binary[i] = 0;
} else {
break;
}
}
}

```

```

}
}
if (i >= infoLen) {
for (j=i; j >= 1; j--) {
binary[j-1] = (binary[j-1]+1)%2;
if (binary[j-1] == 1)
break;
}
}
printf("***** the compress result*****\n");
for (j=0; j < i; j++)
printf("%d ", binary[j]);
}

void uncompress()
{
int i,j;
float w = 0.5;
float deResult=0;
float newLow,newLen;
float low=0,zlen=1;
/***** binary to ten *****/
for (i=0; i < infoLen; i++) {
deResult += w*binary[i];
w *= 0.5;
}
printf("uncompress to ten:%f\n", deResult);
printf("uncompress result:\n");
for (i=0; i < strLen; i++) {
for (j=chNum; j > 0; j--) {
newLow = low;
newLen = zlen;
newLow += newLen * pZone[j-1];
newLen *= P[j-1];
if (deResult >= newLow) {
low=newLow;
zlen=newLen;
printf("%c ",chSet[j-1]);
break;
}}}}

```

## 六、实验体会与建议

本实验让我收获很大，动手能力增强的同时理论基础更加扎实，在此次实验中，我加深了对于计算机算法知识的理解，而且锻炼了我的实验思维，可以拓展课本之外的能力，让自己不仅仅依靠书本上的知识发展自己的认知，我认为本课程极具教育意义，意义重大。