南开大学

Nankai University

南　开　大　学

网 络 空 间 安 全 学 院

算法导论大作业

基于 Huffman 编码和动态规划的图像压缩及对比论证

聂志强 20121307

年级：2020 级

专业：信息安全

指导教师：苏明

2022 年 6 月 6 日

摘要

随着图像采集设备（例如摄像头等）分辨率、精度越来越高，会存在由于文件内存占用过大，导致传输效率低下，网络延迟等，因此该大作业解决的应用问题为图像压缩，探究基于哈夫曼编码和动态规划压缩算法及对比论证分析。

数字化图像是 n×n 的像素阵列，假定每个像素有一个 0 ~ 255 的像素值，即存储一个像素需 8 位。为了减少存储空间，均为采用变长模式，即不同的像素用不同位数来存储。最终使用较少的位数对所有像素点进行存储，从而节约存储空间。

基于 Huffman 编码算法对于压缩效果的影响依赖于图像像素的复杂性以及图像大小，具有一定局限性（该论断将在 [四] 综合分析与探究中给出），而动态规划算法对于图像压缩的效果依赖于图像像素大小以及连续性（该定义与推论也将在 [四] 综合分析与探究中给出），恰好与 Huffman 算法形成互补，因此同时探究基于 Huffman 和动态规划的图像压缩算法具有一定必要性。

本文构建了基于 Huffman 编码及动态规划两种图像无损压缩算法，通过多个测试样例探究权值梯度、图像大小等多个因素对压缩效果的影响，最终得出两种算法最佳适用环境。

关键字：动态规划　Huffman 编码　图像压缩

目录

一、 问题描述

图像压缩编码就是在满足一定保真度和图像质量的前提下，对图像数据进行变换、编码和压缩，去除多余的数据以减少表示数字图像时需要的数据量，便于图像的存储和传输。即以较少的数据量有损或无损地表示原来的像素矩阵的技术，也称图像编码 [2]。

随着图像、声音在信息传输中占的比例越来越大，传输速率和实时性要求的提高，对数据进行压缩处理以节省存储空间和传输信道带宽非常必要，如果每一个像素点都用上界 8 位来编码，则造成很多空间的浪费，我利用 Huffman 编码和动态规划算法使得不同的像素用不同位数来存储，进而实现图像压缩的目的。

功能要求：

1. 针对一幅任意格式的图片文件，对图片文件进行压缩和解压缩。
2. 探究这两种算法最佳的适用场景，使得图片压缩效果最大化

二、 基于 Huffman 编码图像压缩

从出现概率角度考虑，某个值出现的概率越大，编码长度越短

(一) 算法概述 [1]

1. 读取原文件

   读取图像文件，每个像素包含 RGB 三个色彩通道，每个通道占 1 个字节，这是编码的单元。对读取到的每个像素的色彩通道数据进行权重统计。

2. 生成 Huffman 树

   根据权重统计构建 Huffman 编码树。

3. 生成 Huffman 编码

   遍历 Huffman 树，记录 256 个叶子节点的路径，生成 Huffman 编码；

4. 压缩编码

   使用编码表对原来的每个通道的色彩通道数据进行重新编码，获得压缩后的文件数据；

5. 保存文件

   将编码输出到自定义的编码文件中。

(二) 数据结构的设计

1. 二叉树的存储结构。使用结构体存储节点，使用数组存储树的节点，使用静态二叉链表方式存储二叉树。

```
struct HuffNode
{
    unsigned char ch;  //字节符
    int weight;  //字节出现频度
    int parent; //父节点
    int lchild;  //左孩子
```

```
    int rchild;  //右孩子
    char bits[256]; // 哈夫曼编码
};

// Huffman树
typedef char** HuffmanCode;
```

2. Huffman 编码存储结构定义一个二重指针:

```
typedef char** HuffmanCode;
```

3. 压缩文件的算法的数据结构:

为正确解压文件，要保存原文件中 256 种字节重复的次数，即权值。定义一个文件头，保存相关的信息:

```
struct FILESTRUCT
{
    int sum;    //权值
    int fileid;
};
```

(三) 核心算法设计

1. 构造 Huffman 树

```
int CreateHuffmanTree(HuffNode *huf_tree, int n)
{
    int i;
    int s1, s2;
    for (i = n; i < 2 * n - 1; ++i)
    {
        select(huf_tree, i, &s1, &s2);          // 选择最小的两个结点
        huf_tree[s1].parent = i;              //原点双亲为i
        huf_tree[s2].parent = i;
        huf_tree[i].lchild = s1;                //新结点左子树是最小的s1
        huf_tree[i].rchild = s2;                //新结点右子树是最小s2
        huf_tree[i].weight = huf_tree[s1].weight + huf_tree[s2].weight;////新
            结点的权值
    }
    return OK;
}
//选择parent为零且为最小的两个数，序号分别为s1,s2
void select(HuffNode *huf_tree, int n, int *s1, int *s2)
{
    // 找最小结点
    unsigned int i;
    unsigned long min = LONG_MAX;
    for (i = 0; i < n; i++)
        if (huf_tree[i].parent == -1 && huf_tree[i].weight < min)
```

```
        {
            min = huf_tree[i].weight;
            *s1 = i;//记录下标
        }
    huf_tree[*s1].parent = 1;    // 标记此结点已被选中
    // 找次小结点
    min = LONG_MAX;
    for (i = 0; i < n; i++)
    {
        if (huf_tree[i].parent == -1 && huf_tree[i].weight < min)
        {
            min = huf_tree[i].weight;
            *s2 = i;
        }
    }
}
```

2. 生成 Huffman 编码

从每一个叶节点开始向上循环直至父节点不存在，若处于父节点左边则编码 0，右边则编码 1，最终形成 01 串

```
for (i = 0; i < n; ++i)
{
    index = 256 - 1;     // 编码临时空间初始化

    // 从叶子向根求编码
    for (cur = i, next = huf_tree[i].parent; next != -1; next = huf_tree[next
        ].parent)
    {
        if (huf_tree[next].lchild == cur)
            code_tmp[--index] = '0';     // 左 '0'
        else
            code_tmp[--index] = '1';     // 右 '1'
        cur = next;
    }
    strcpy(huf_tree[i].bits, &code_tmp[index]);  // 正向保存编码到树结点相应
        域 index是第一个
}
```

3. 压缩编码及写文件

遍历图片每一个像素点，根据生成 Huffman 编码对每一个像素值重新编码并保存至待存文件，同时存入文件头部，总长度，字符及对应长度等信息以便解码

```
//存文件头部
fwrite((char *)&tou, sizeof(char), z + 1, outFile);
//存总长度
fwrite(&flength, sizeof(long), 1, outFile);
```

```
//存字符的种类（256）
fwrite(&n, sizeof(int), 1, outFile);
new_huf = new_huf + (z + 1)*sizeof(char) + sizeof(long) + sizeof(int);
//存每个编号对应的字符,权重
for (int i = 0; i < n; i++) {
    fwrite(&huf_tree[i].ch, sizeof(unsigned char), 1, outFile);
    fwrite(&huf_tree[i].weight, sizeof(long), 1, outFile);
    new_huf = new_huf + sizeof(unsigned char) + sizeof(long);
}
while (fread(&temp, sizeof(unsigned char), 1, inFile))//文件不为空
{
    for (int i = 0; i < n; i++)//找对应字符
    {
        if (temp == huf_tree[i].ch)
        {       //过滤掉非0非1的编码
            for (int f = 0; huf_tree[i].bits[f] == '0' || huf_tree[i].bits[f]
                == '1'; f++)
                strncat(buffer, &huf_tree[i].bits[f], 1);
        }
    }
    while (strlen(buffer) >= 8)//8位为1个字节，缓存流大于等于8个bits进入循环
    {
        new_huf++;
        temp = 0;
        for (int i = 0; i < 8; i++)//每8个bits循环一次
        {
            temp = temp << 1;//左移1
            if (buffer[i] == '1')//如果是为1，就按位为1
            {
                temp = temp | 0x01;//在不影响其他位的情况下，最后位置1
            }
        }
        fwrite(&temp, sizeof(unsigned char), 1, outFile);//写入文件
        strcpy(buffer, buffer + 8);//将写入文件的bits删除
    }
}
int m = strlen(buffer);//将剩余不足为8的bits的个数给1
if (m) {
    new_huf++;
    temp = 0;
    for (int i = 0; i < m; i++)
    {
        temp = temp << 1;//移动1
        if (buffer[i] == '1')//如果是为1，就按位为1
            temp = temp | 0x01;
    }
    temp <<= 8 - m;// // 将编码字段从尾部移到字节的高位
    fwrite(&temp, sizeof(unsigned char), 1, outFile);//写入最后一个
```

```
52 }
```

（四） 复杂度分析

1. **时间复杂度：**

由于在算法 CreateHuffmanTree 中 i 和 j 内外两层循环次数不超过 256 和 512, 故对构建 HuffmanTree 可在 O(1) 时间内完成。通过哈夫曼树对 256 个不同像素的颜色值进行编码 (HuffmanCoding 函数)，一共两层循环均为定值（分别为 256 和 log(256)），所以时间复杂度仍为 O(1)。对图片中每一个像素点的颜色值按照编码表进行压缩，时间复杂度为 O(n)

综上，基于 Huffman 编码的图像压缩算法时间复杂度为 O(n)

2. **空间复杂度：**

构建 Huffman 树时最多 511 个节点，因此存储所有树节点空间复杂度为 O(1)，遍历图像所有像素点统计不同像素值的频率并存储在数组中，空间复杂度 O(n)，综上，基于 Huffman 编码图像压缩算法空间复杂度为 O(n)

（五） 测试结果

1. 输入“1”选择“压缩功能”并输入压缩图像名称”text1.png”与压缩后文件”test1.txt”

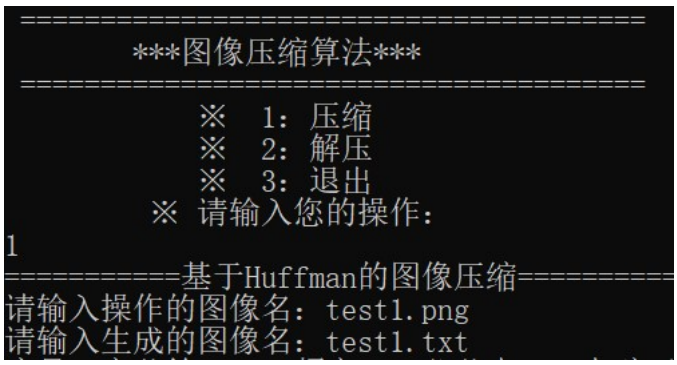
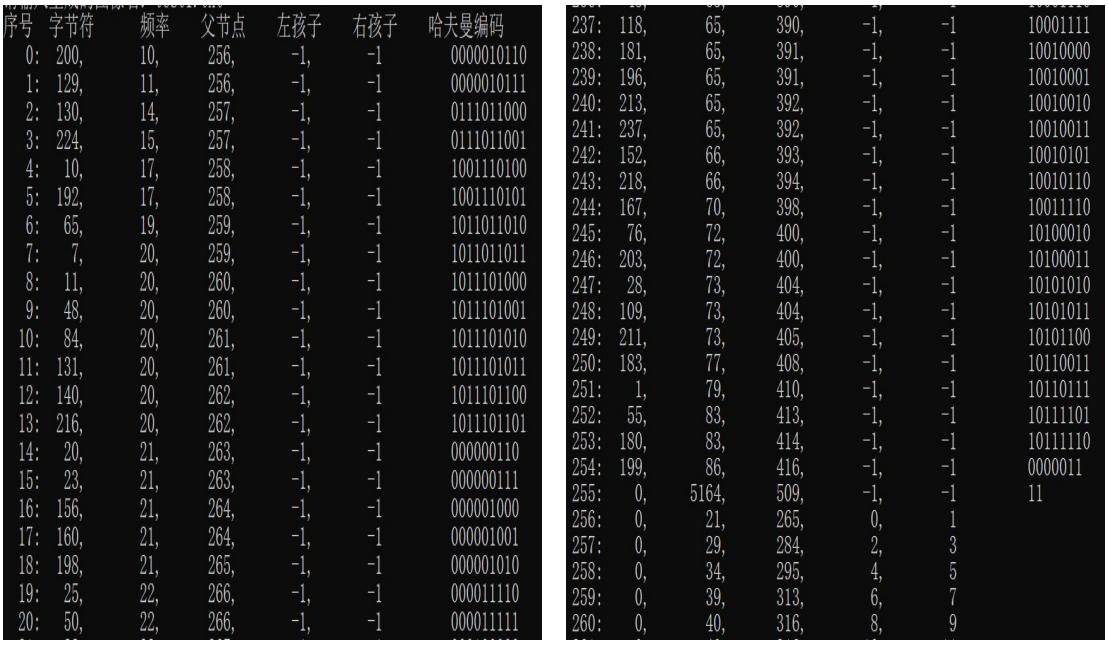


图 1

2. Huffman 编码后各个节点信息（部分），前 255 个序号对应的就是有效像素值及其对应的叶子节点信息和 Huffman 编码，后面的节点均为 Huffman 树中间节点，为了体现父子关系因此均输出出来，仅对叶子节点编码



序号	字节符	频率	父节点	左孩子	右孩子	哈夫曼编码
0:	200,	10,	256,	-1,	-1	0000010110
1:	129,	11,	256,	-1,	-1	0000010111
2:	130,	14,	257,	-1,	-1	0111011000
3:	224,	15,	257,	-1,	-1	0111011001
4:	10,	17,	258,	-1,	-1	1001110100
5:	192,	17,	258,	-1,	-1	1001110101
6:	65,	19,	259,	-1,	-1	1011011010
7:	7,	20,	259,	-1,	-1	1011011011
8:	11,	20,	260,	-1,	-1	1011101000
9:	48,	20,	260,	-1,	-1	1011101001
10:	84,	20,	261,	-1,	-1	1011101010
11:	131,	20,	261,	-1,	-1	1011101011
12:	140,	20,	262,	-1,	-1	1011101100
13:	216,	20,	262,	-1,	-1	1011101101
14:	20,	21,	263,	-1,	-1	000000110
15:	23,	21,	263,	-1,	-1	000000111
16:	156,	21,	264,	-1,	-1	000001000
17:	160,	21,	264,	-1,	-1	000001001
18:	198,	21,	265,	-1,	-1	000001010
19:	25,	22,	266,	-1,	-1	000011110
20:	50,	22,	266,	-1,	-1	000011111
237:	118,	65,	390,	-1,	-1	10001111
238:	181,	65,	391,	-1,	-1	10010000
239:	196,	65,	391,	-1,	-1	10010001
240:	213,	65,	392,	-1,	-1	10010010
241:	237,	65,	392,	-1,	-1	10010011
242:	152,	66,	393,	-1,	-1	10010101
243:	218,	66,	394,	-1,	-1	10010110
244:	167,	70,	398,	-1,	-1	10011110
245:	76,	72,	400,	-1,	-1	10100010
246:	203,	72,	400,	-1,	-1	10100011
247:	28,	73,	404,	-1,	-1	10101010
248:	109,	73,	404,	-1,	-1	10101011
249:	211,	73,	405,	-1,	-1	10101100
250:	183,	77,	408,	-1,	-1	10110011
251:	1,	79,	410,	-1,	-1	10110111
252:	55,	83,	413,	-1,	-1	10111101
253:	180,	83,	414,	-1,	-1	10111110
254:	199,	86,	416,	-1,	-1	0000011
255:	0,	5164,	509,	-1,	-1	11
256:	0,	21,	265,	0,	1	
257:	0,	29,	284,	2,	3	
258:	0,	34,	295,	4,	5	
259:	0,	39,	313,	6,	7	
260:	0,	40,	316,	8,	9	

图 2

3. 成功压缩后，图像大小由 16181 字节减小到 14077 字节，压缩率为 1.1495

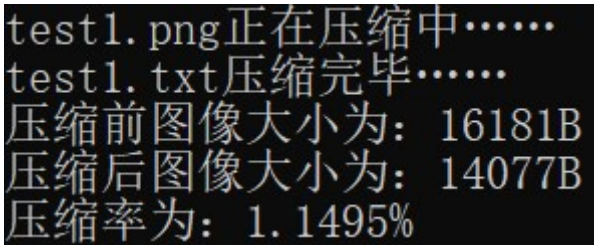


图 3

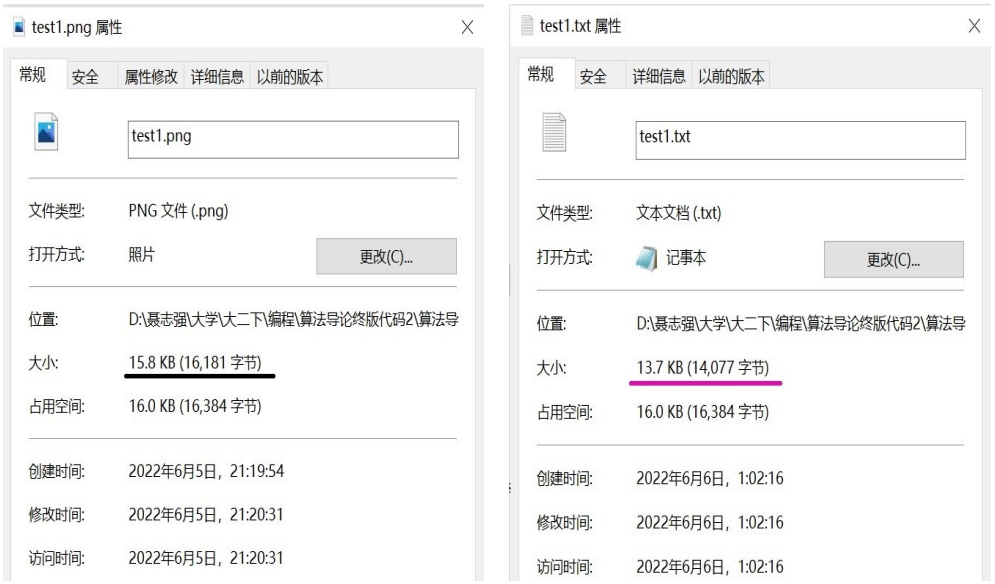


图 4

4. 执行解压缩操作，将”test1.txt” 压缩文件解压为”test1_new.png” 图像 (由于本文探讨的是图像压缩算法，解压代码及相关分析 [3] 均在提交的源代码中给出，因此为了避免冗余该报告中没有具体列出解压缩相关代码)

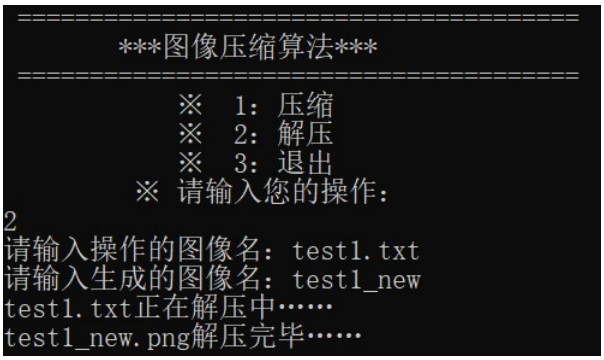


图 5

图 6

5. 图 7 所示测试图通过本次所设计的基于哈夫曼编码的图像压缩系统进行压缩后，得到压缩图。对测试图原图和压缩图进行比较后发现，压缩图与原图文件大小基本一致，且压缩图与原图相比，压缩图的外观形状、大小等特性与原图相同，可视清晰度也与原图相同。

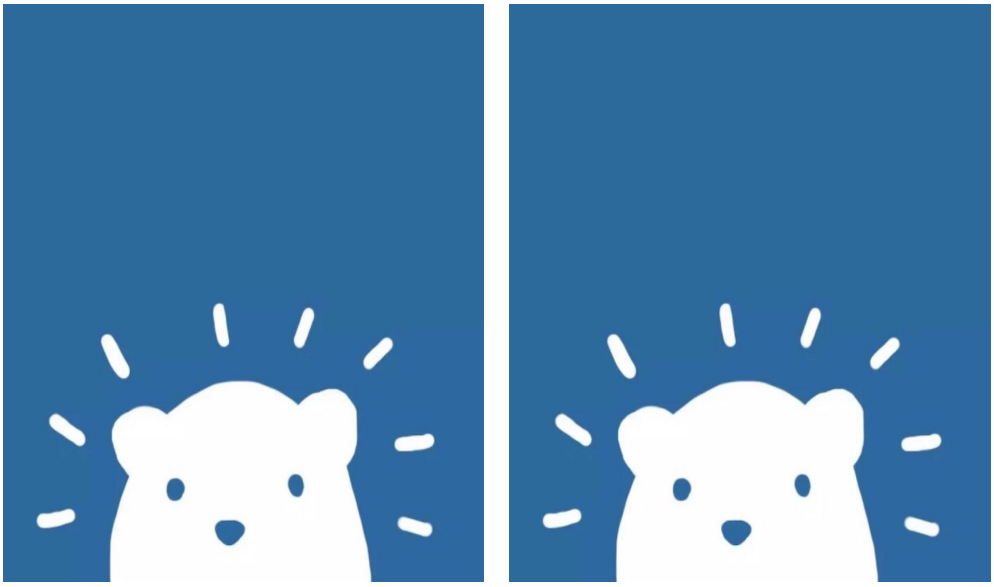


图 7

三、 图像压缩算法优化：动态规划算法

基于 Huffman 编码算法对于压缩效果的影响依赖于图像像素的复杂性以及图像大小，因此具有一定局限性（该论断将在 [四] 综合分析与探究中给出），而动态规划算法对于图像压缩的效果依赖于图像像素大小以及连续性（该定义与推论也将在 [四] 综合分析与探究中给出），恰好与 Huffman 算法形成互补，因此探究基于动态规划的图像压缩算法具有一定必要性，其核心在于将线性化的数据根据动态规划进行分段使总位数最小。

(一) 算法概述

在计算机中，常用像素点的灰度值序列 $\{P_1\ P_2\ \cdots P_n{}^2\}$ 表示图像。其中整数 $p_i, 1 <= i <= n$，表示像素点 i 的灰度值。通常灰度值的范围是 0 255。因此最多需要 8 位表示一个像素。压缩的原理就是把序列 $\{P_1\ P_2\ \cdots P_n{}^2\}$ 进行分段，每段内各个像素位数相同，分段的过程就是要找出断点，让一段里面的像素的最大灰度值比较小，那么这一段像素 (本来需要 8 位) 就可以用较少的位 (比如 7 位) 来表示，从而减少存储空间。利用动态规划算法找到一个最优分段，使得存储空间最少 [6] [7]

具体步骤：

1. **图像线性化：**将 $n\times n$ 维图像转换为 $1\times n^2$ 向量，将矩阵式的数字变成线性：$\{P_1\ P_2\ \cdots P_n{}^2\}$
2. **分段：** 将像素分为连续 m 段 s1，s2，$\cdots$ sm，使每段中像素存储位数相同，每段最多包含 256 个像素点。
3. **创建三个表：**

   b 代表 bits, l 代表 length

   b[i]：第 i 个像素段的每个像素点位数

   l[i]：第 i 个像素段里面像素点个数

   s[i]：像素序列 $p_1, p_2, \ldots\ldots p_n$ 的最优分段所需的存储位数

考虑时间复杂度，如果限制 $l[i] <= 255$，则需要 8 位来表示 $l[i]$。而 $b[i] <= 8$，需要 3 位表示 $b[i]$。所以每段所需的存储空间为 $l[i] * b[i] + 11$ 位。假设将原图像分成 m 段，那么需要 $\sum_{i=1}^{m} l[i] * b[i] + 11m$ 位的存储空间。图像压缩问题就是要确定像素序列 $\{p_1, p_2, \ldots\ldots p_n\}$ 的最优分段，使得依此分段所需的存储空间最小。

(二) 递推关系

1. 最优子结构

设 $l[i], b[i], 1 <= i <= m$ 是的一 $\{P_1\ P_2\ \cdots P_n{}^2\}$ 个最优分段，则 $l[1], b[1]$ 是 $\{p_1, \ldots\ldots, p_{l[1]}\}$ 的一个最优分段，且 $l[i], b[i], 2 <= i <= m$ 是 $pl[1] + 1, \ldots\ldots, pn$ 的一个最优分段。即图像压缩问题满足最优子结构性质。

2. 元素 p[i] 的分段情况

1. 假设 p[i] 自成一段，则 s[i]=s[i-1]+ 最后一个像素点的代价；
2. 假设最后两个像素点为一段，则 s[i]=s[i-2]+ 最后两个像素点的代价；
3. 假设最后 i 个像素点为一段，则 s[i]=s[0]+ 最后 i 个像素点的代价.

3. 递推公式

由最优子结构以及元素 p[i] 分段情况可知，递推公式为

$$s[n] = min_{l \le k \le min\{n,256\}}\{s[n-k] + k \times bmax(n-k+1, n)\} + 11$$

其中

$$bmax(i, j) = \lceil log(max_{i \le k \le j}\{p_k\} + 1) \rceil$$

(三) 核心代码

1. 计算存储像素 p_i 所需的位数

```
int length(int i)
{
    int k=1;
    i = i/2;
    while(i>0)
    {
        k++;
        i=i/2;
    }
    return k;
}
```

2. 动态规划核心代码

数组 $l[i], b[i]$ 记录了最优分段所需的信息：b[i]：第 i 个像素段的每个像素点位 l[i]：第 i 个像素段里面像素点个数

最优分段的最后一段的段长度和像素位数分别存储在 $l[n]$ $b[n]$ 中，其前一段的段长度和像素位数存储于 $l[n-l[n]]$ $b[n-l[n]]$ 中

```
void Compress(int n,int p[],int s[],int l[],int b[])
{//n个像素，存在数组p[]中，
    int Lmax = 256,header = 11;
    s[0] = 0;
    for(int i=1; i<=n; i++)
    {
        b[i] = length(p[i]);//计算像素点p需要的存储位数
        int bmax = b[i];
        s[i] = s[i-1] + bmax;
        l[i] = 1;

        for(int j=2; j<=i && j<=Lmax;j++)//i=1时，不进入此循环
        {
            if(bmax<b[i-j+1])
            {
                bmax = b[i-j+1];
            }
            if(s[i]>s[i-j]+j*bmax)
            {
```

```
                s[i] = s[i-j] + j*bmax;
                l[i] = j;
            }
        }
        s[i] += header;
    }
}
```

3. 递归输出分段结果

```
void Traceback(int n,int& i,int s[],int l[])
{
    if(n==0)
        return;
    Traceback(n-l[n],i,s,l);//p1,p2,p3,...,p(n-l[n])像素序列，最后一段有l[n]
        个像素
    s[i++]=n-l[n];//重新为s[]数组赋值，用来存储分段位置，最终i为共分了多少段
}
```

（四） 复杂度分析 [5]

1. 空间复杂度

算法 Compress 只需 O(n) 空间

2. 时间复杂度

由于在算法 Compress 中 j 的循环次数不超过 256, 故对每一个确定的 i 可在 O(1) 时间内完成。因此整个算法的时间复杂度为 O(n)。

（五） 测试结果

1. 测试样例 1

像素序列 4，6，5，7，129，138，1

a. 理论推导：

$$s[0] = 0 \tag{1}$$

$$s[1] = 0 + 1 \times 3 + 11 = 14 \tag{2}$$

$$s[2] = min \begin{cases} s[0] + 2 \times max(3,3) + 11 \\ s[1] + 1 \times max(3) + 11 \end{cases} = min(17, 28) = 17 \tag{3}$$

$$s[3] = min \begin{cases} s[0] + 3 \times max(3,3,3) + 11 \\ s[1] + 2 \times max(3,3) + 11 \\ s[2] + 1 \times max(3) + 11 \end{cases} = min(20, 31, 31) = 20 \tag{4}$$

$$s[4] = min\begin{cases} s[0] + 4 \times max(3,3,3,3) + 11 \\ s[1] + 3 \times max(3,3,3) + 11 \\ s[2] + 2 \times max(3,3) + 11 \\ s[3] + 1 \times max(3) + 11 \end{cases} = min(23,34,34,34) = 23 \tag{5}$$

$$s[5] = min\begin{cases} s[0] + 5 \times max(3,3,3,3,8) + 11 \\ s[1] + 4 \times max(3,3,3,8) + 11 \\ s[2] + 3 \times max(3,3,8) + 11 \\ s[3] + 2 \times max(3,8) + 11 \\ s[4] + 1 \times max(8) + 11 \end{cases} = min(51,57,52,47,42) = 42 \tag{6}$$

$$s[6] = min\begin{cases} s[0] + 6 \times max(3,3,3,3,8,8) + 11 \\ s[1] + 5 \times max(3,3,3,8,8) + 11 \\ s[2] + 4 \times max(3,3,8,8) + 11 \\ s[3] + 3 \times max(3,8,8) + 11 \\ s[4] + 2 \times max(8,8) + 11 \\ s[5] + 1 \times max(8) + 11 \end{cases} = min(59,65,60,55,50,61) = 50 \tag{7}$$

$$s[7] = min\begin{cases} s[0] + 6 \times max(3,3,3,3,8,8,1) + 11 \\ s[1] + 6 \times max(3,3,3,3,8,8) + 11 \\ s[2] + 5 \times max(3,3,3,8,8) + 11 \\ s[3] + 4 \times max(3,3,8,8) + 11 \\ s[4] + 3 \times max(3,8,8) + 11 \\ s[5] + 2 \times max(8,8) + 11 \\ s[6] + 1 \times max(8) + 11 \end{cases} = min(67,73,68,63,58,62) = 58 \tag{8}$$

b. 程序结果：

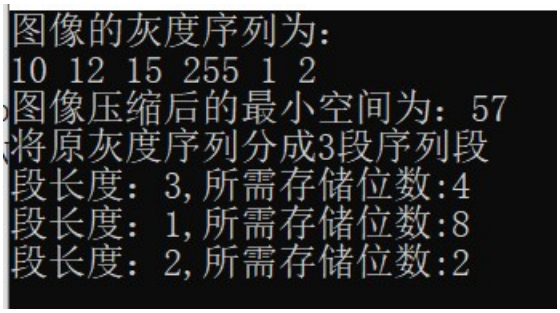


图 8

2. **测试样例** 2

a. 为了与 Huffman 编码形成对照，仍采用 test1.png 图像进行测试成功压缩后，图像大小由 16181 字节减小到 12140 字节，压缩率为 1.3329

```
==========基于动态规划的图像压缩==========
压缩前图像大小为：16181B
图像压缩后的最小空间为：12140
压缩率为：1.3329%
```

图 9

b. 输出部分具体分段内容

```
将原灰度序列分成329段序列段
段长度：9,所需存储位数:7
段长度：5,所需存储位数:1
段长度：9,所需存储位数:7
段长度：16,所需存储位数:4
段长度：11,所需存储位数:5
段长度：30,所需存储位数:6
段长度：12,所需存储位数:7
段长度：7,所需存储位数:4
段长度：58,所需存储位数:6
段长度：23,所需存储位数:5
段长度：6,所需存储位数:2
段长度：13,所需存储位数:1
段长度：6,所需存储位数:3
段长度：6,所需存储位数:5
段长度：12,所需存储位数:3
段长度：10,所需存储位数:3
```

图 10

c. 图 11 所示测试图通过本次所设计的基于动态规划的图像压缩算法进行压缩后，得到压缩图。对测试图原图和压缩图进行比较后发现，压缩图与原图文件大小基本一致，且压缩图与原图相比，压缩图的外观形状、大小等特性与原图相同，可视清晰度也与原图相同。

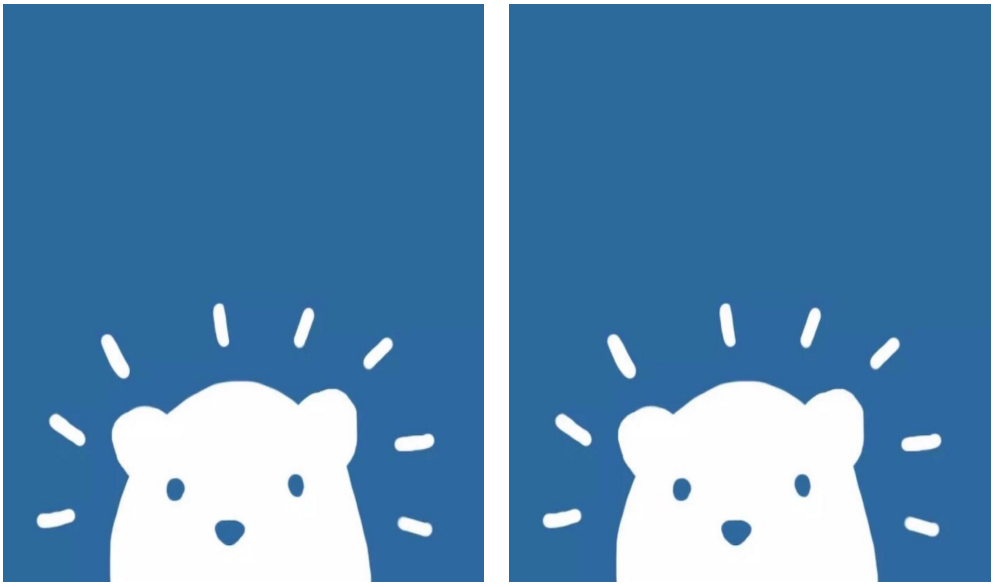


图 11

四、 综合分析和结论

(一) Huffman 编码压缩

1. 探究权值梯度对压缩效果影响

a. 色彩简单的图像，主要颜色对应的像素值频率会远大于其他像素值，根据 Huffman 树构建原理可知该像素值对应的霍夫曼编码为短码，此时能够使短码最大限度利用，图像压缩效果较好

(1). 对如下图像进行压缩

图 12

(2). 下图为 Huffman 编码生成的权值编码映射表，可以观察到 256 个像素中像素值为 162、81、138、40、20、0 是其他像素值频率的 3-80 倍，形成明显频率差，这几个像素值对应的霍夫曼编码为 4 位，使短码能够充分利用

序号	字节符	频率	父节点	左孩子	右孩子	霍夫曼编码
0:	76,	31,	256,	-1,	-1	11100111110
1:	96,	31,	256,	-1,	-1	11100111111
2:	132,	32,	257,	-1,	-1	11110101110
3:	16,	37,	257,	-1,	-1	11110101111
4:	66,	37,	258,	-1,	-1	0001111110
5:	129,	37,	258,	-1,	-1	0001111111
6:	41,	39,	259,	-1,	-1	0101010000
7:	108,	39,	259,	-1,	-1	0101010001
8:	38,	40,	260,	-1,	-1	0101110100
9:	18,	41,	260,	-1,	-1	0101110101
10:	48,	44,	261,	-1,	-1	0111010100
11:	157,	46,	261,	-1,	-1	0111010101
12:	32,	47,	262,	-1,	-1	0111110000
13:	72,	47,	262,	-1,	-1	0111110001
14:	74,	47,	263,	-1,	-1	0111110010
15:	152,	47,	263,	-1,	-1	0111110011
16:	6,	48,	264,	-1,	-1	0111111110
17:	77,	48,	264,	-1,	-1	0111111111
18:	84,	48,	265,	-1,	-1	1010001010
19:	36,	49,	265,	-1,	-1	1010001011
20:	137,	50,	266,	-1,	-1	1010011010
21:	46,	51,	266,	-1,	-1	1010011011
22:	56,	51,	267,	-1,	-1	1010011110
23:	166,	51,	267,	-1,	-1	1010011111
24:	9,	52,	268,	-1,	-1	1011100000
25:	148,	52,	268,	-1,	-1	1011100001

230:	175,	213,	418,	-1,	-1	10111100
231:	143,	214,	418,	-1,	-1	10111101
232:	191,	229,	424,	-1,	-1	11001101
233:	248,	229,	425,	-1,	-1	11001110
234:	95,	236,	427,	-1,	-1	11100010
235:	252,	249,	430,	-1,	-1	11101000
236:	253,	277,	437,	-1,	-1	11110110
237:	63,	322,	445,	-1,	-1	0101100
238:	127,	366,	450,	-1,	-1	0111011
239:	254,	385,	453,	-1,	-1	1010000
240:	128,	604,	469,	-1,	-1	001100
241:	10,	606,	470,	-1,	-1	001110
242:	80,	631,	471,	-1,	-1	010100
243:	1,	683,	473,	-1,	-1	011001
244:	255,	708,	474,	-1,	-1	011011
245:	64,	811,	478,	-1,	-1	101010
246:	2,	814,	478,	-1,	-1	101011
247:	5,	836,	479,	-1,	-1	101101
248:	160,	861,	481,	-1,	-1	110000
249:	69,	2213,	495,	-1,	-1	11111
250:	162,	2226,	496,	-1,	-1	0000
251:	81,	2389,	497,	-1,	-1	0010
252:	138,	2492,	498,	-1,	-1	0100
253:	40,	2962,	500,	-1,	-1	1000
254:	20,	2999,	500,	-1,	-1	1001
255:	0,	3852,	502,	-1,	-1	1101

图 13

(3). 原图像大小为 48394B，通过 Huffman 编码压缩后文件大小为 41345B，压缩率为 1.1705%，达到了图像压缩的目的

```
flag.jpg正在压缩中……
flag.txt压缩完毕……
压缩前图像大小为：48394B
压缩后图像大小为：41345B
压缩率为：1.1705%
```

图 14

b. 色彩丰富的图像，像素种类多，各个不同像素值频率之间差距不大，此时会使各个频率霍夫曼编码长度普遍较长，不能使频率高的像素值映射编码有效变短，反而会使映射编码可能超多原来的最大 8 位，压缩效果不明显。

(1). 对如下图像进行压缩

图 15

(2). 下图为 Huffman 编码生成的权值编码映射表，可以观察到各个像素值频率差距不大且像素值种类多（256 个），各个像素值对应的霍夫曼编码在 7-9 位，没起到了有效压缩的目的

序号	字节符	频率	父节点	左孩子	右孩子	霍夫曼编码
0:	44,	48,	256,	-1,	-1	111010100
1:	190,	48,	256,	-1,	-1	111010101
2:	73,	49,	257,	-1,	-1	111101100
3:	250,	51,	257,	-1,	-1	111101101
4:	72,	55,	258,	-1,	-1	111111000
5:	151,	55,	258,	-1,	-1	111111001
6:	36,	56,	259,	-1,	-1	111111010
7:	144,	57,	259,	-1,	-1	111111011
8:	244,	59,	260,	-1,	-1	111111100
9:	47,	60,	260,	-1,	-1	111111101
10:	125,	60,	261,	-1,	-1	111111110
11:	30,	61,	261,	-1,	-1	111111111
12:	66,	61,	262,	-1,	-1	00000000
13:	100,	61,	262,	-1,	-1	00000001
14:	191,	61,	263,	-1,	-1	00000010
15:	208,	61,	263,	-1,	-1	00000011
16:	221,	61,	264,	-1,	-1	00000100
17:	95,	62,	264,	-1,	-1	00000101
18:	133,	62,	265,	-1,	-1	00000110
19:	229,	62,	265,	-1,	-1	00000111
20:	14,	63,	266,	-1,	-1	00001000
21:	246,	63,	266,	-1,	-1	00001001
22:	9,	64,	267,	-1,	-1	00001010
23:	123,	64,	267,	-1,	-1	00001011
24:	157,	64,	268,	-1,	-1	00001100
25:	200,	64,	268,	-1,	-1	00001101
26:	21,	65,	269,	-1,	-1	00001110
27:	26,	65,	269,	-1,	-1	00001111
28:	156,	65,	270,	-1,	-1	00010010
29:	18,	66,	270,	-1,	-1	00010011
30:	193,	66,	271,	-1,	-1	00010100
31:	202,	66,	271,	-1,	-1	00010101
32:	135,	67,	272,	-1,	-1	00010110
33:	15,	68,	272,	-1,	-1	00010111

222:	53,	94,	367,	-1,	-1	11011100
223:	61,	94,	367,	-1,	-1	11011101
224:	70,	94,	368,	-1,	-1	11011110
225:	85,	94,	368,	-1,	-1	11011111
226:	162,	94,	369,	-1,	-1	11100000
227:	166,	94,	369,	-1,	-1	11100001
228:	8,	95,	370,	-1,	-1	11100010
229:	5,	96,	370,	-1,	-1	11100011
230:	17,	96,	371,	-1,	-1	11100100
231:	92,	96,	371,	-1,	-1	11100101
232:	94,	96,	372,	-1,	-1	11100110
233:	171,	96,	372,	-1,	-1	11100111
234:	196,	96,	373,	-1,	-1	11101000
235:	214,	96,	373,	-1,	-1	11101001
236:	113,	97,	374,	-1,	-1	11101011
237:	140,	97,	375,	-1,	-1	11101100
238:	16,	98,	375,	-1,	-1	11101101
239:	13,	99,	376,	-1,	-1	11101110
240:	46,	99,	376,	-1,	-1	11101111
241:	55,	99,	377,	-1,	-1	11110000
242:	57,	99,	377,	-1,	-1	11110001
243:	177,	99,	378,	-1,	-1	11110010
244:	122,	100,	378,	-1,	-1	11110011
245:	218,	100,	379,	-1,	-1	11110100
246:	243,	100,	379,	-1,	-1	11110101
247:	3,	101,	380,	-1,	-1	11110111
248:	215,	102,	381,	-1,	-1	11111000
249:	121,	106,	381,	-1,	-1	11111001
250:	6,	110,	382,	-1,	-1	11111010
251:	170,	110,	382,	-1,	-1	11111011
252:	2,	131,	389,	-1,	-1	0001000
253:	10,	153,	407,	-1,	-1	0101110
254:	1,	172,	426,	-1,	-1	1010101
255:	0,	280,	452,	-1,	-1	001000

图 16

(3). 原图像大小为 20800B，通过 Huffman 编码压缩后文件大小为 22049B，压缩率为 0.9434%，没有达到压缩的目的

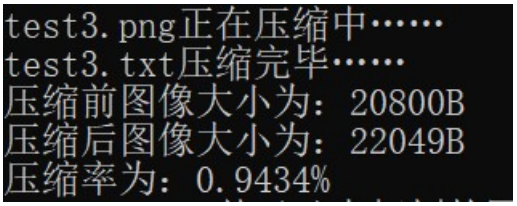


图 17

2. **探究图像大小对压缩效果影响**

在 Huffman 编码图像压缩算法中需要用额外的位保存和传输 Huffman 树，从而“浪费”掉一些存储位，把本已减少的数据量又增加了一些。如果文件比较大，这一点多余的数据所占比例很小，但如果压缩的文件本来就很小的话，增加的多余存储数据会明显影响压缩效果

针对同一张图片两个不同尺寸进行压缩

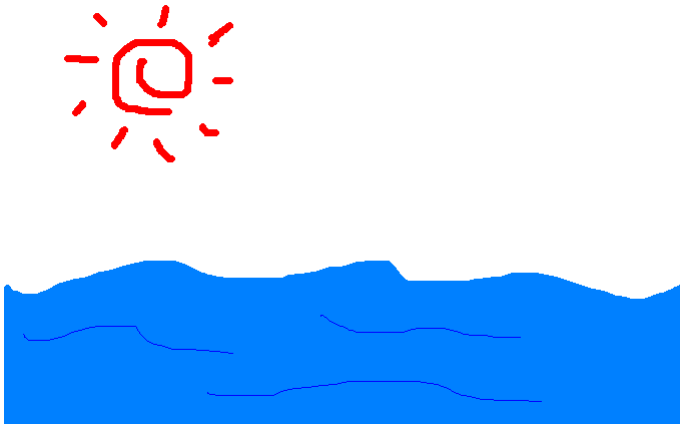


图 18

a. **针对大尺寸原始图像的图像进行基于 Huffman 编码图像压缩**

(1). 原图像大小为 884262B，通过 Huffman 编码压缩后文件大小为 148738，其中存储的 Huffman 树信息及文件头所占空间与图像总大小比值为 0.000659，对于整个存储空间影响不大，压缩率为 5.9451%，压缩效果显著

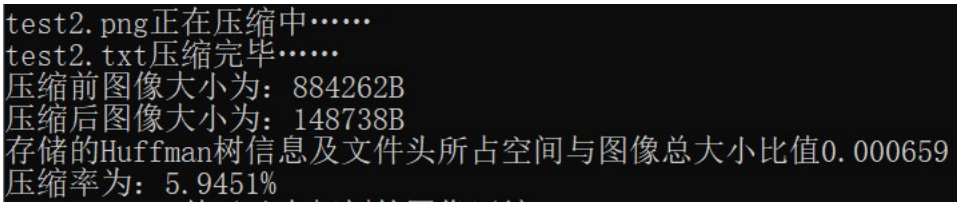


图 19

b. **针对小尺寸原始图像的图像进行基于 Huffman 编码图像压缩**

(1). 原图像大小 19133B，通过 Huffman 编码压缩后文件大小为 6880B，其中存储的 Huffman 树信息及文件头所占空间与图像总大小比值为 0.135610，对于整个存储空间有影响，压

缩率由 5.9451% 降为 2.7810%，压缩效果不如大尺寸图像压缩显著

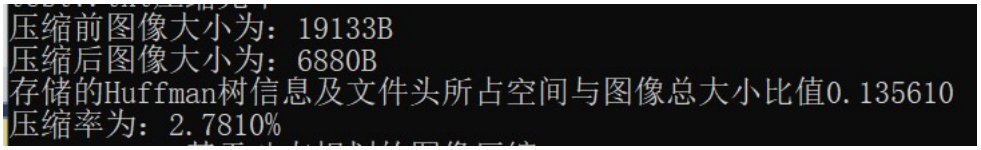


图 20

(二) 动态规划压缩

1. 探究像素值大小及连续性对压缩效果影响

由于动态规划图像压缩算法核心思想是将一系列相邻且位数相同或近似的像素值放在一个段里表示，所以动态规划的压缩效果与像素值实际大小有关，同时也和连续的像素值位数近似度有关。

说明：连续性主要是指相同或浮动不超过 1 的像素值大小连续出现的位数概率，构建连续性模型，如果连续性概率小于 0.3，则连续性等级为 1，如果连续性概率大于 0.3 小于 0.6，则连续性等级为 2，连续性概率大于 0.6 小于 1 时，连续性等级为 3。像素值大小即该像素点对应的像素值真实大小。

a. 针对像素值大小普遍较小且连续性高的图像进行基于动态规划算法压缩 (1). 对如下图像进行压缩

图 21

(2). 根据统计，该图像像素值小于 100 的像素点比率为 0.4390，大于等于 100 小于 200 像素点的比率为 0.3563，大于等于 200 小于 255 像素点的比率为 0.2048，根据各个区间比率可以判定该图像像素值大小普遍较小；该图像连续性概率为 0.7811，根据连续性的定义该图像连续性为 3 级，具有很好的连续性。

该图像原始大小为 40495B，压缩后大小为 37022B，压缩率为 1.0938%，达到了压缩的效果

```
==========基于动态规划的图像压缩==========
压缩前图像大小为：40495B
图像压缩后的最小空间为：37022B
压缩率为：1.0938%
图像全部像素点的个数=40495
像素值小于100像素点的个数=17776
像素值小于200大于等于100像素点的个数=14427
像素值小于255大于等于200像素点的个数8292
像素值小于100像素点的概率=0.4390
像素值小于200大于等于100像素点的概率=0.3563
像素值小于255大于等于200像素点的概率=0.2048
连续性等级=3
```

图 22

b. 针对像素值大小普遍较大且连续性低的图像进行基于动态规划算法压缩

(1). 对如下图像进行压缩

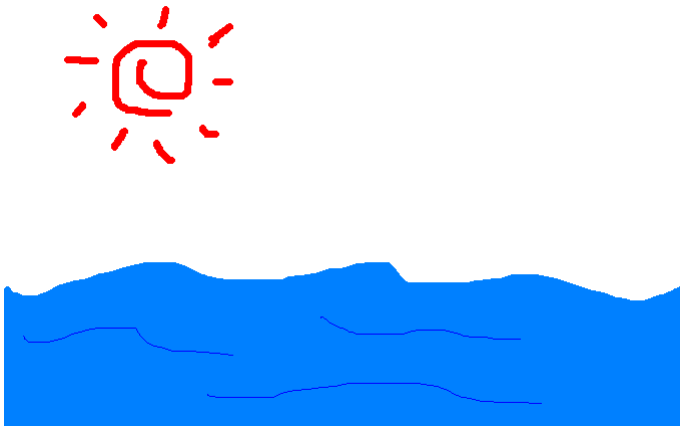


图 23

(2). 根据统计，该图像像素值小于 100 的像素点比率为 0.1226，大于等于 100 小于 200 像素点的比率为 0.1110，大于等于 200 小于 255 像素点的比率为 0.7664，根据各个区间比率可以判定该图像像素值大小普遍较大；该图像连续性概率为 0.2145，连续性等级为 1 级，通过输出该图像具体像素值可以观察到有一系列的 [0,128,255]RGB 像素值连续排列，导致该图像连续性较差。该图像原始大小为 884262B，压缩后大小为 888344B，压缩率为 0.9954%，没有达到了压缩的效果

```
==========基于动态规划的图像压缩==========
压缩前图像大小为：884262B
图像压缩后的最小空间为：888344B
压缩率为：0.9954%
图像全部像素点的个数=884262
像素值小于100像素点的个数=108404
像素值小于200大于等于100像素点的个数=98186
像素值小于255大于等于200像素点的个数677672
像素值小于100像素点的概率=0.1226
像素值小于200大于等于100像素点的概率=0.1110
像素值小于255大于等于200像素点的概率=0.7664
连续性等级=1
```

图 24

(三) 探究总结

1. 在基于 Huffman 编码图像压缩算法中，考虑的是不同像素值出现频率，目标是使短码得到充分利用。经过上述实验推断，该算法图像压缩效果与图像每个像素值本身大小没有关系，与像素值之间梯度（频率差）以及图像尺寸有关，当像素值之间梯度（频率差）大时，能够使频率很大的像素值映射到短码，图像尺寸大时能够使多保存的 Huffman 结构影响尽可能小。

2. 在基于动态规划压缩算法中，通过以上探究，影响压缩效果的是每一个像素值的实际大小以及连续性，因此通过我构建的像素值大小及连续性评估模型得出该图像两个指标的等级，该压缩算法适合应用于像素值大小普遍较小且连续性较好的图像。

3. 本文选用测试图对其核心的图片压缩功能进行了测试，测试图通过本次所设计系统压缩功能的压缩后，所得到的压缩图与原图文件视觉效果一致且具有无损压缩特性，在压缩的过程中，并不会存在数据的损坏与丢失。

五、 调试说明

1. 在读取图片文件统计 0-255 个字符的权值的过程中，一开始采用了 C+ 的 ifstream fin(“图像名”) 文件流，然后通过 while(fin»ch) cout«ch; 测试输出文件字符码，就出现了无限循环，一直连续不断地输出 6 位十六进制的数。当时认为是文件流读取方式的原因，加了 iOS::binary 来控制采用二进制形式，还是没有解决。改用 C 语言的 FILE *fp = fope( ) 终于可以正常读取输出文件字符码。

2. 文件编码压缩 ‘Encode()‘ 函数会产生编码后的一个缓冲区 ‘char *pBuffer；‘写文件函数会使用它直接写磁盘文件。调试过程中并没发现任何问题，就是不能成功地写后缀为.huf 的文件。在相关函数中设置断点，观察缓冲区的情况，且编写屏幕输出缓冲区数据的程序段，发现缓冲区是空的。通过在 Encode 函数中以及 WriteFile 函数中做同样的跟踪调试，发现在 Encode 函数中建立的缓冲区数据并没有带出来，通过分析发现是缓冲区空间构建位置的问题，即不能直接用这个变量做 ‘Encode()‘ 函数形参，而应该采用指针或者引用类型做函数形参，这样可以通过直接访问 pBuffer 的内存地址改变缓冲区内容。

3. 编译时没有错误，通过 pHC[i] = (char*)malloc((cdlen + 1) * sizeof(char)); 获取内存空间，然后 pHC[i] = cd; 将字符串数组 cd 的内容复制到 pHC[i] 中时出现了内存错误。编译后无错误，运行时出现错误一般是指针使用是内存分配出现错误。打开调试界面查看 CPU 状态，看到汇编代码，但是汇编功底不是很深，还是没有找到底层原因。只好换了一种实现方式：strcpy(pHC[i], cd[start]); 问题得以解决。

4. 编译时总是提示 fopen，strcpy，strcat 等函数存在不安全问题，一方面通过 “打开项目 » 属性 » C/C++ » 预处理器 » 预处理器定义 “内添加 _CRT_SECURE_NO_WARNIN ，另一方面还在文件开头添加：pragma warning(disable:4996) 后，成功解决该问题

六、 改进与优化

1. 在实际应用中要考虑信道占用和传输时间，上述基于 Huffman 编码图像压缩算法需要先建树再编码压缩串行执行，而且解码时也需要先重新建树，会降低效率。因此可以对上述 Huffman 编码压缩进行优化，可以采取动态 Huffman 编码采取了一边扫描数据一边编码

（也就是边建树）的方式，扫描完成后编码也完成。在扫描的过程中就可以同时传输数据。这就解决了实时传翰的问题。同时，接收端一边接收数据一边建树，与发送方建树方式完全一致，这期间解码也在进行，接收完成解码也就完成。[4]

参考文献

[1] 吕姣霖 and 徐艳. 哈夫曼编码在图像压缩中的应用与分析. 数字通信世界, 2021.

[2] 张霞. 图像压缩方法分类及其评价. 泰山学院学报, 40(3):81–83, 1 2018.

[3] 王兆丽, 肖春霞, 王梅娟, 蒋园园, and 董会. 哈夫曼编码在图像无损压缩中的应用. 计算机工程与科学, 41(S01):3, 2019.

[4] 王继林. 动态 huffman 编码在图像压缩中的应用. 电脑编程技巧与维护, 000(009):56–61, 2007.

[5] 舒畅, 王大为, and 李龙腾. 动态规划算法实现数字图像压缩的研究. 计算机与数字工程, 36(004):134–136, 2008.

[6] 饶兴. 基于 huffman 编码的图像压缩解压研究. 电脑知识与技术, 007(004):887–889, 2011.

[7] 马子睿. 基于动态规划法的图像压缩算法. 宁夏师范学院学报, 28(3):3, 2007.