大学生研究计划结题报告

——天文数据压缩算法的研究

第一部分：有关情况说明

1.1、课题名称：天文数据压缩算法的研究

1.2、课题指导师：董小波老师、彭波师兄

1.3、学生：薛兆江 PB12025015

1.4、课题的来源与确定

天文观测领域获得的图像数据量大，为解决其与有限的网络带宽之间的矛盾，对数据进行有效的压缩是必要的。而传统的压缩软件并未针对天文数据特有的特点对其进行处理，压缩效果不是特别理想。本课题将对这种算法进行研究。
通常，注重细节的天文图像尺寸都是相当巨大的，但由于Internet网络的特点是网络带宽随网络负载动态变化，不能保证有稳定的带宽，并且根据网络拥塞状况会出现丢包情况而丢失了数据，从而会中断传输，不能保证信息快速、无损、安全的传递。因此，在无法改变网络条件的情况下，就需要对传送的内容进行处理。对静态图像进行编码和无损压缩，可以在信息量不变的同时降低网络传输量，减少传输的时阐。并且由于天文图像是一种专用格式，其珍贵精确、长时间持续获取的特点，就要求算法能够实时无损高效快速地将数据进行压缩。如果使用通用的压缩方法压缩效果不理想，因此就要对天文图像进行分析，找出一种适合它的压缩方法。本课题将对这种算法进行研究。

1.5、课题研究的目的和意义

通过研究这种算法，熟悉研究课题的过程，了解与微电子方向相关的知识。了解各种压缩算法的特点，提高用C语言的能力，以及了解和学习硬件描述性语言。锻炼查找和阅读文献的能力。

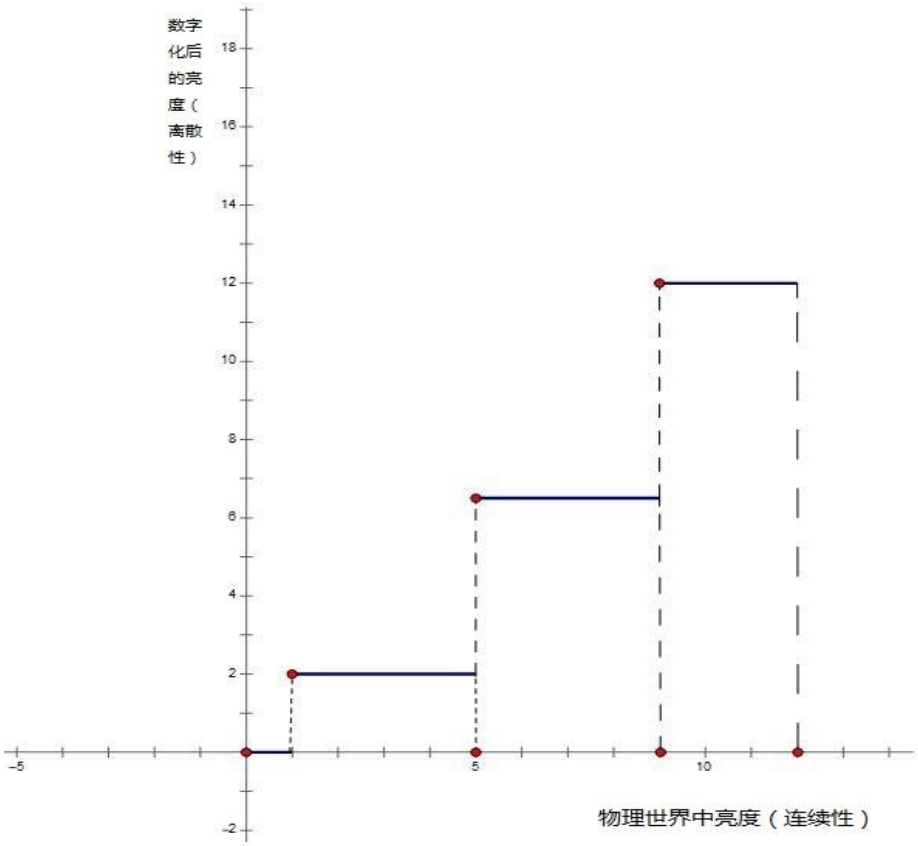
第二部分：课题研究

2.1、图像数字化

计算机只能处理数字而不是图形，那么就需要将图形数字化。将物理图像划分为图像元素(像素)，将其亮度数字化，对于黑白图像只需要一个亮度值，而对于彩色图像是红绿蓝三个亮度值。这样可见像素的两个属性：位置，灰度。

2.1.1、数字化过程

1、扫描，对图像内容给定位置的寻址。
2、采样，通过映射变换描写信号的方式亮度变为电压、电流。
3、量化，利用模数转换器，将其转换为计算机可处理的数字信号，这一步将减少信号精度，必须保证有足够的取样频率和量化精度。

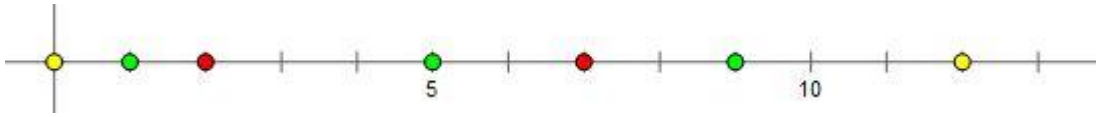


2.1.2、量化

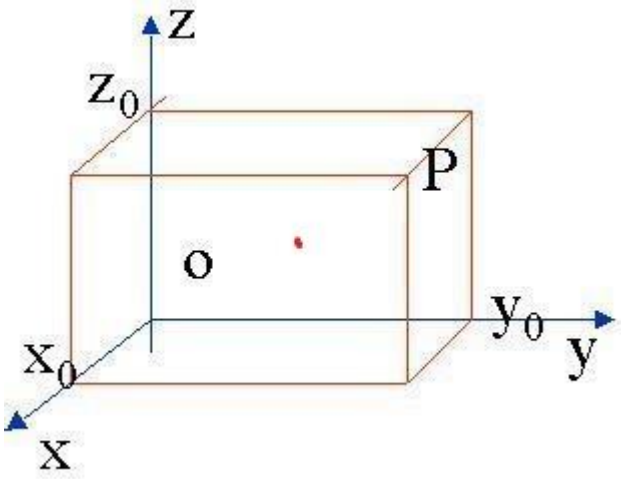
标量量化，概括为两步

- 1、分割成有限的子区间

• 2、选取区间代表值



矢量量化，先把每K个样值分为一组，K维空间中的一个矢量，然后同样的步骤进行量化。



2.2、数据编码种类

冗余信息和不相关信息：冗余信息是指由于数据结构、存储等方面设计的不合理，而造成的信息重复。冗余度表现为空间（帧内）和时间（帧间）相关性。信号统计上的冗余度来源于被编码信号概率密度分布的不均匀。

另：信道编码，新加入人工控制的冗余度，进行误码防护，广泛使用了具有一定纠错能力的信道编码技术，如奇偶校验码、行列监督码等编码技术

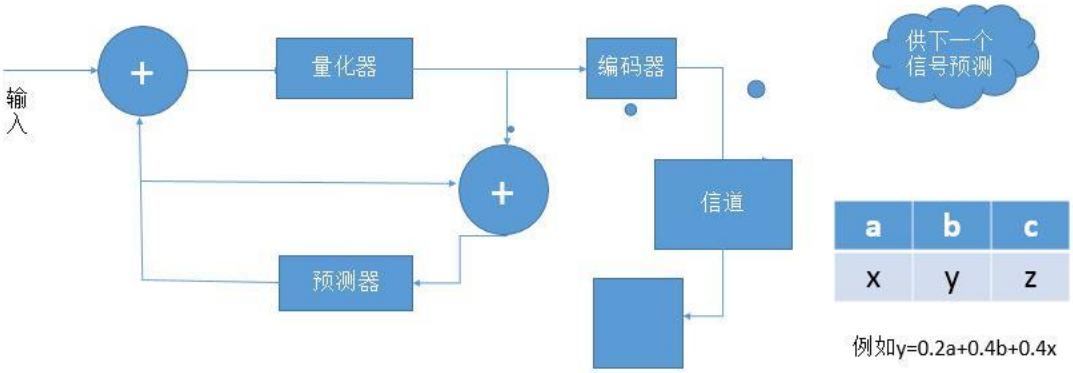
另：格雷码(Gray code),又叫循环二进制码或反射二进制码 。数字系统中只能识别0和1，各种数据要转换为二进制代码才能进行处理，格雷码是一种无权码，采用绝对编码方式，典型的格雷码是一种具有反射特性和循环特性的单步自补码，它的循环、单步特性消除了随机取数时出现重大误差的可能，它的反射、自补特性使得求反非常方便。格雷码属于可靠性编码，是一种错误最小化的编码方式，因为，自然二进制码可以直接由数/模转换器转换成模拟信号，但某些情况，例如从十进制的 3 转换成 4 时二进制码的每一位都要变，使数字电路产生很大的尖峰电流脉冲。而格雷码则没有这一缺点，它是一种数字排序系统，其中的所有相邻整数在它们的数字表示中只有一个数字不同。它在任意两个相邻的数之间转换时，只有一个数位发生变化。它大大地减少了由一个状态到下一个状态时逻辑的混淆。

2.2.1、预测编码

预测编码是根据离散信号之间存在着一定关联性的特点，利用前面一个或多个信号通过一定的算法预测下一个信号的属性，然后对实际值和预测值的差（预测误差）进行编码。如果预测比较准确，误差就会很小。预测误差的概率密度近似是一围绕 0 附近的尖锐的拉普拉斯分布，熵小，方差小，在同等精度要求的条件下，就可以用比较少的比特进行编码，达到

压缩数据的目的。缺点是对信道误码的敏感性，有累积效应，会产生差错扩散。

- 利用像素周围的已知像素来预测该像素



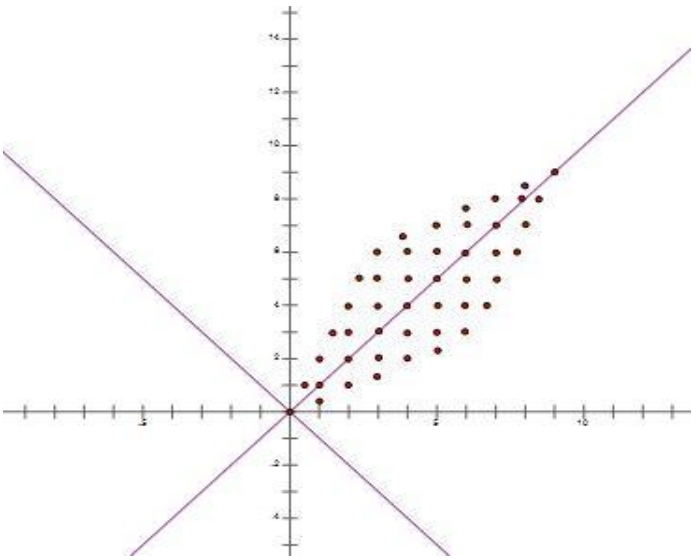
对于视频图像，当景物不发生剧烈运动，不切换场景时，可以利用帧间相关性，静止区利用前帧预测，运动区帧内预测，帧间运动补偿预测，实现时间轴冗余度压缩。

2.2.2、变换编码

变换编码是从频域的角度减小图像信号的空间相关性，它在降低数码率等方面取得了和预测编码相近的效果。

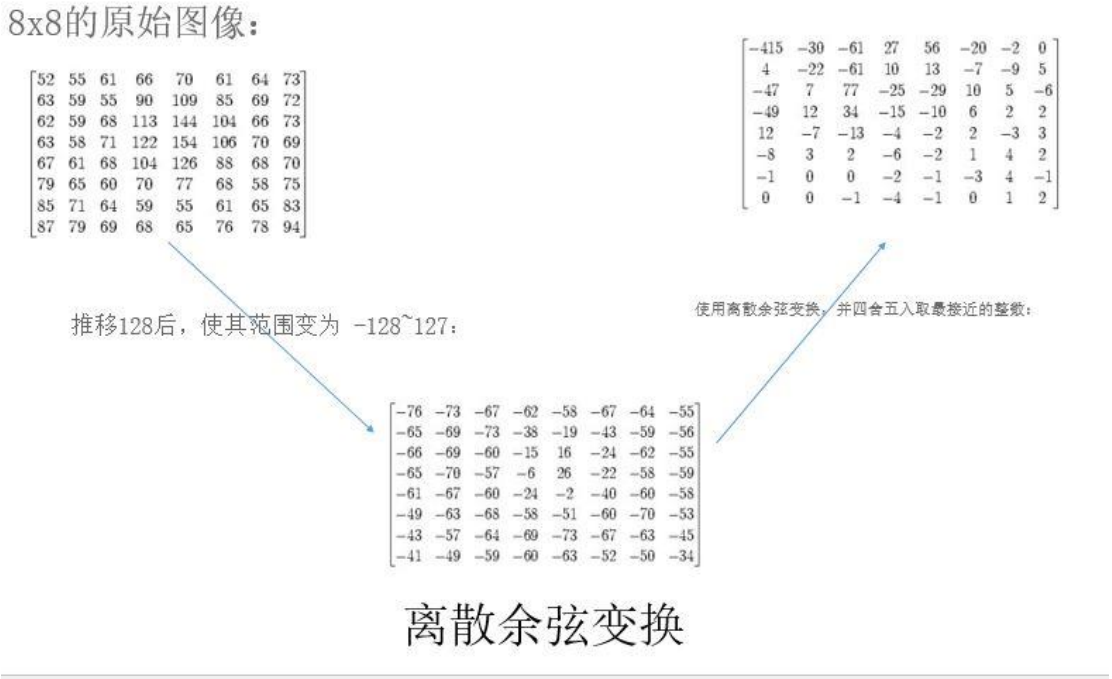
变换编码不是直接对空域图像信号进行编码，而是首先将空域图像信号映射变换到另一个正交矢量空间（变换域或频域），产生一批变换系数，然后对这些变换系数进行编码处理。变换编码是一种间接编码方法，其中关键问题是在时域或空域描述时，数据之间相关性大，数据冗余度大，经过变换在变换域中描述，如果所选的正交向量空间的基向量与图像本身的特征向量很接近，数据相关性大大减少，数据冗余量减少，参数独立，数据量少，这样再进行量化，编码就能得到较大的压缩比。典型的准最佳变换有 DCT（离散余弦变换）、DFT(离散傅里叶变换)、WHT（Walsh Hadama 变换）、HrT(Haar 变换)等。其中，最常用的是离散余弦变换。

下图中所示的图像，相关性比较强，像块出现在 45 度线上的概率大，如果将坐标系逆时针旋转 45 度，后，两坐标值近似统计独立，但是信息量并没有减少。



傅里叶变换：傅里叶变换是一种分析信号的方法，它可分析信号的成分，也可用这些成分合成信号。人的主管视觉对高频成分不如对低频敏感，反变换回去又是原信号。

$$f(t)=\frac{a_0}{2}+\sum_{n=1}^{\infty}\left[a_n\cos\left(\frac{2\pi nt}{T}\right)+b_n\sin\left(\frac{2\pi nt}{T}\right)\right]$$



分块单独变换编码的思想：由于计算量、存储量、相关半径的局限，可以将数据进行分块，进行分别变换编码处理。

2.2.3 统计编码

统计编码是根据消息出现概率的分布特性而进行的压缩编码,它有别于预测编码和变换编码。这种编码的宗旨在于,在消息和码字之间找到明确的一一对应关系,以便在恢复时能准确无误地再现出来，使平均码长或码率压低到最低限度。包括哈夫曼编码，算术编码，比特平面编码，游程编码，LZW 编码（字典），无损预测编码等。

2.2.3.1、哈夫曼（Huffman）编码

基本原理：按信源符号出现的概率大小进行排序，出现概率大的分配短码，出现概率小的则分配长码。（定长码采用相同的码长对数据进行编码，如 ASCII 码是定长码，其码长为 1 字节。）在变长码中，对于出现概率在的信息符号编以短字长的码，对于出现概率小的信息符号以长字长的码，如果码字长度严格按照符号概率的大小的相反顺序排列，则平均码字长度一定小于按任何其他符号顺序排列方式得到的码字长度。

2.2.3.2、算术编码（Arithmetic Coding）

算术编码方法也是利用信源概率分布特性、能够趋近熵极限的编码的方法。算术编码不

按符号编码，即不是用一个特定的码字与输入符号之间建立一一对应的关系，而是从整个符号序列出发，采用递推形式进行连续编码，用一个单独的浮点数来表示一串输入符号。算术编码是将被编码的信息表示成实数 0 和 1 之间的一个间隔。信息越长，编码表示它的间隙就越小，表示这一间隙所须二进位就越多，大概率符号出现的概率越大对应于区间愈宽，可用长度较短的码字表示；小概率符号出现概率越小层间愈窄，需要较长码字表示。它的编码方法比 Huffman 编码方式要复杂，但它不需要传送像 Huffman 编码中的 Huffman 码表，同时算术编码还有自适应的优点，所以算术编码是实现高效压缩数据中很有前途的编码方法。

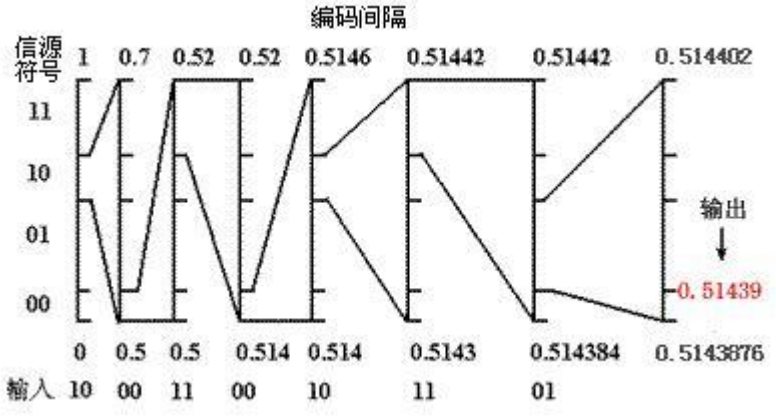
特点：方法比较复杂，具有自适应能力（随着编码符号流中 01 出现的概率的变化将自适应的改变）。在信源符号概率接近时，算术编码比 Huffman 编码效率要高。

举例：假设信源符号为{00, 01, 10, 11}，这些符号的概率分别为{ 0.1, 0.4, 0.2, 0.3 }，根据这些概率可把间隔[0,1)分成 4 个子间隔：[0, 0.1), [0.1, 0.5), [0.5, 0.7), [0.7, 1)，其中[x,y)表示半开放间隔，即包含 x 不包含 y。上面的信息可综合在下表中。

表 信源符号，概率和初始编码间隔

符号	00	01	10	11
概率	0.1	0.4	0.2	0.3
初始编码间隔	[0,0.1)	[0.1,0.5)	[0.5,0.7)	[0.7,1)

如果二进制消息序列的输入为：10 00 11 00 10 11 01。编码时首先输入的符号是 10，找到它的编码范围是[0.5, 0.7)。由于消息中第二个符号 00 的编码范围是[0, 0.1)，因此它的间隔就取[0.5, 0.7)的第一个十分之一作为新间隔[0.5, 0.52)。依此类推，编码第 3 个符号 11 时取新间隔为[0.514, 0.52)，编码第 4 个符号 00 时，取新间隔为[0.514, 0.5146)，… 。消息的编码输出可以是最后一个间隔中的任意数。整个编码过程如下图示：



表：编码过程

步骤	输入符号	编码间隔	编码判决
1	10	[0.5, 0.7)	符号的间隔范围[0.5, 0.7)
2	00	[0.5, 0.52)	[0.5, 0.7)间隔的第一个 1/10
3	11	[0.514, 0.52)	[0.5, 0.52)间隔的最后三个 1/10
4	00	[0.514, 0.5146)	[0.514, 0.52)间隔的第一个 1/10

5	10	[0.5143, 0.51442)	[0.514, 0.5146)间隔的第六个 1/10 开始的两个 1/10
6	11	[0.514384, 0.51442)	[0.5143, 0.51442)间隔的最后三个 1/10
7	01	[0.5143836, 0.514402)	[0.514384, 0.51442)间隔的从第二个 1/10 开始的四个 1/10
8	从[0.5143876, 0.514402 中选择一个数作为输出：0.51439		

表：译码过程

步骤	间隔	译码符号	译码判决
1	[0.5, 0.7)	10	0.51439 在间隔 [0, 1) 第六个 1/10
2	[0.5, 0.52)	00	0.51439 在间隔 [0.5, 0.7)的第一个 1/10
3	[0.514, 0.52)	11	0.51439 在间隔[0.5, 0.52)的第八个 1/10
4	[0.514, 0.5146)	00	0.51439 在间隔[0.514, 0.52)的第一个 1/10
5	[0.5143, 0.51442)	10	0.51439 在间隔[0.514, 0.5146)的第七个 1/10
6	[0.514384, 0.51442)	11	0.51439 在间隔[0.5143, 0.51442)的第八个 1/10
7	[0.5143876, 0.514402)	01	0.51439 在间隔[0.5143876, 0.514402)的第二个 1/10
8	译码的消息：10 00 11 00 10 11 01		

译码器的译码过程应无限制地运行下去。在译码器中需要添加一个专门的终止符，当译码器看到终止符时就停止译码。

在算术编码中需要注意的几个问题：

1、由于实际的计算机的精度不可能无限长，运算中出现溢出是一个明显的问题，但多数机器都有 16 位、32 位或者 64 位的精度，因此这个问题可使用比例缩放方法解决。

2、算术编码器对整个消息只产生一个码字，这个码字是在间隔[0,1)中的一个实数，因此译码器在接受到表示这个实数的所有位之前不能进行译码。

3、算术编码也是一种对错误很敏感的编码方法，如果有一位发生错误就会导致整个消息译错。

算术编码可以是静态的或者自适应的。在静态算术编码中，信源符号的概率是固定的。在自适应算术编码中，信源符号的概率根据编码时符号出现的频繁程度动态地进行修改，在编码期间估算信源符号概率的过程叫做建模。需要开发自适应算术编码的原因是因为事先知道精确的信源概率是很难的，而且是不切实际的。当压缩消息时，不能期待一个算术编码器获得最大的效率，所能做的最有效的方法是在编码过程中估算概率。因此动态建模就成为确定编码器压缩效率的关键。

2.2.3.3、游程编码

基本原理是：用一个符号值或串长代替具有相同值的连续符号（连续符号构成了一段连续的“行程”。行程编码因此而得名），使符号长度少于原始数据的长度。只在各行或者各列数据的代码发生变化时，一次记录该代码及相同代码重复的个数，从而实现数据的压缩。

举例来说，一组资料串"AAAABBBCCDEEEE"，由 4 个 A、3 个 B、2 个 C、1 个 D、4 个 E 组成，经过变动长度编码法可将资料压缩为 4A3B2C1D4E(由 14 个单位转成 10 个单位)。

简言之，其优点在于将重复性高的资料量压缩成小单位；然而，其缺点在于若该资料出现频率不高，可能导致压缩结果资料量比原始资料大，例如：原始资料"ABCDE"，压缩结果为"1A1B1C1D1E"(由 5 个单位转成 10 个单位)。

一维游程编码只利用了同一扫描行上各像素的相关性，如果进一步利用相邻的多个扫描行之间的相关性，则有可能达到更高的压缩比，称为二维游程编码。

2.2.3.4、LZW 编码

LZW 是通过建立一个字符串表，用较短的代码来表示较长的字符串来实现压缩。 应该注意到的是，我们这里的编译表不是事先创建好的，而是根据原始文件数据动态创建的，解码时还要从已编码的数据中还原出原来的编译表。

建立一个包含所有待编码信源符号的码书，当编码器顺序地检查输入的像素时，将新的序列放在下一个位置上。LZW 压缩算法和其它一些压缩技术的不同之处在于它是动态地标记数据流中出现的重复串。它把在压缩过程中遇到的字符串记录在串表中，在下次又碰到这一字符串的时侯，就用一个代码来表示它，通过用短代码表示相对较长的字符串来压缩数据量。

2.3、课题着重学习的算法

无损数据压缩按照采用模型的不同,可分为统计模型与字典模型这两类技术。选取两种较有代表性以及比较适合天文方面应用的 Huffman 编码和 LZW 编码，进行学习。

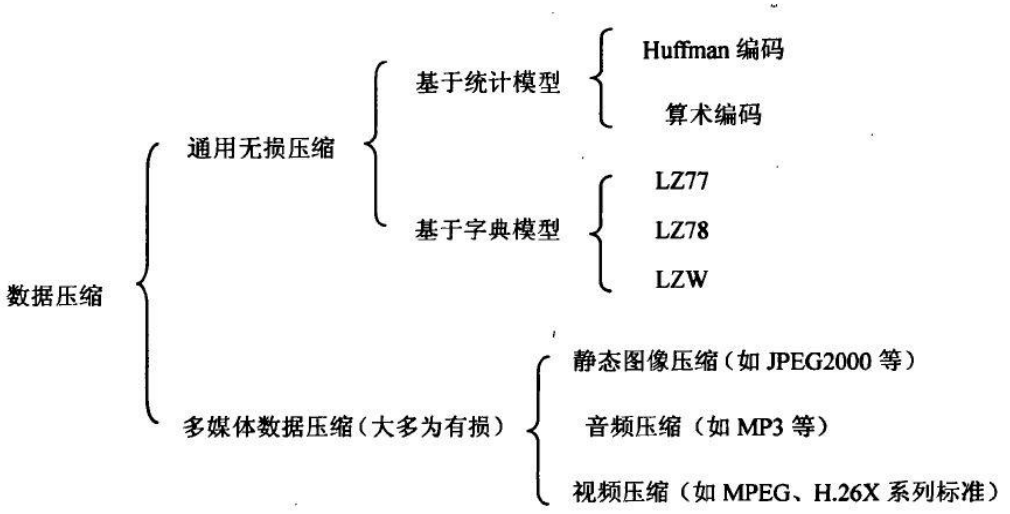


图 2.1 数据压缩技术分类

2.3.1、哈夫曼（Huffman）编码

2.3.1.1、算法简介

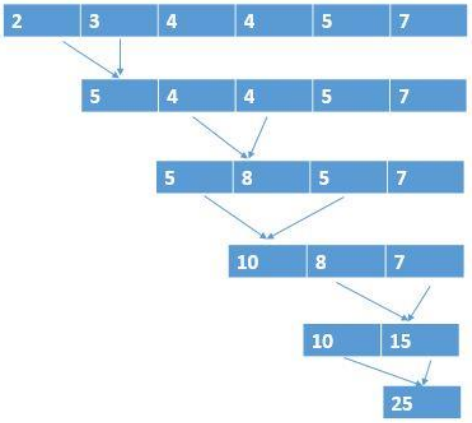
哈夫曼编码是一种非前缀码，按信源符号出现的概率大小进行排序，出现概率大的分配短码，出现概率小的则分配长码。效率高，但是计算复杂。哈夫曼树又称最优二叉树，是一

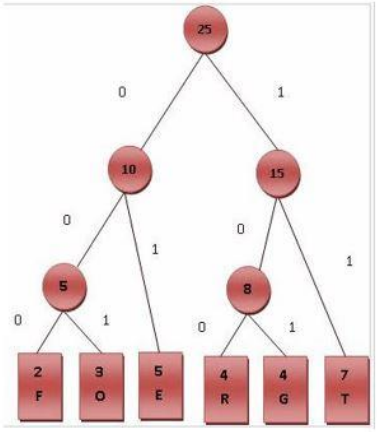
种带权路径长度最短的二叉树。所谓树的带权路径长度，就是树中所有的叶结点的权值乘上其到根结点的路径长度（若根结点为 0 层，叶结点到根结点的路径长度为叶结点的层数）。

构造哈夫曼树非常简单，将所有的节点放到一个队列中，用一个节点替换两个频率最低的节点，新节点的频率就是这两个节点的频率之和。这样，新节点就是两个被替换节点的父节点了。如此循环，直到队列中只剩一个节点（树根）。

下图中给出一个简单的建立哈夫曼树的过程。

Symbol	F	O	R	G	E	T
Frequency	2	3	4	4	5	7
CODE	000	001	100	101	01	11





算法的关键在于统计出各符号出现的频率，以及正确建立哈夫曼树。由于冒泡排序实现比较简单，并且不需要对数组进行全部排序就可以选出出现频率较高的几个字符，因此代码中选取冒泡排序。

2.3.1.2、Huffman 算法 C++源代码

```
/*****************************************************************************
*****************************************************************************/
//先run一下关掉，在cmd中，拖入exe文件，空格，拖入要压缩的文件，回车
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
using namespace std;

FILE *in;
FILE *out;

typedef struct char_rate{
  unsigned char c; //字母
  unsigned int r;//频率
}char_rate;
//#define node_num(n) (( 1 <<( n ))-1)
//int level=4;//哈夫曼树的层次
void Statistics(char_rate a[]);//统计各种字符出现的频率
void BubbleSort(char_rate a[]);//对频率进行排序
void Compress(char_rate b[]);//再次扫描，压缩
void output (unsigned int code,int bitsize);//将不定长码转为8bit定长输出
```

```c
int main(int argc,char* argv[]){
  char filename[255];
  char_rate a[256];
  int i;//初始化数组时用

  if(argc<2){    //0,1
    printf("usage:the command format is:lzw_compression <filename>!");
    return(1);
  }
  if((in = fopen(argv[1], "rb")) == NULL){
    printf ("Cannot open input file - %s/n", argv[1]);
    exit (1);
  }

  strcpy(filename, argv[1]);
 //加上后缀.huf,表明是压缩文件
  strncat(filename, ".huf", 4);
  if((out = fopen(filename, "wb")) == NULL){
    printf("Cannot open output file - %s/n", filename);
    fclose(in);
    exit(1);
  }

  for(i=0;i<256;i++){
      a[i].c=i;
      a[i].r=0;
  }
  Statistics(a);
  fclose (in);
  fclose (out);
  return 0;
}

void Statistics(char_rate a[]){
    unsigned char ch;
  //  int ch;
    int n1=0;
    while(  ((ch = fgetc(in))!= EOF ) , (!feof(in))  ){//这次虽然能正确复制 0xFF，但却不能判断文件结束。
    //事实上，在 c为unsigned char时，c != -1 是永远成立的，一个高质量的编译器，比如gcc会在编译时指出这一点。
        a[ch].r++;
        n1++;
        if(feof(in)){
           break;
        }
    }
    rewind(in);
//  fclose(in);
    BubbleSort(a);

    int i;
    char_rate b[4];
    for(i=0;i<4;i++){
        b[i]=a[i];
        printf("字符%c频率%d\n",b[i].c,b[i].r);
    }

    printf("字符总频率%d \n",n1 );
    Compress(b);
}
```

```
void BubbleSort(char_rate a[]){
    char_rate temp;
    for(int i =0 ; i<4; i++) {
        for(int j = 255; j > i; j--) {
           if(a[j].r > a[j-1].r) {
              temp = a[j];
              a[j] = a[j-1] ;
              a[j-1] = temp;
           }
        }
    }
}

void Compress(char_rate b[]){
    unsigned char ch;
    int bitsize;
    unsigned int code;
    while(  ((ch = fgetc(in))!= EOF ) && (!feof(in))  ){
        if(ch==b[0].c){
            code=0b0;bitsize=1;
            output(code,bitsize);
        }
        else
            if(ch==b[1].c){
                code=0b10;bitsize=2;
                output (code,bitsize);
            }
            else
                if(ch==b[2].c){
                    code=0b110;bitsize=3;
                    output (code,bitsize);
                }
                else
                    if(ch==b[3].c){
                        code=0b1110;bitsize=4;
                        output (code,bitsize);
                    }
                    else{
                        code=0b1111;
                        code<<=8;
                        code|=ch;
                        bitsize=12;
                        output(code,bitsize);
                    }
    }
}

void output (unsigned int code,int bitsize){
  static int count = 0;
  static unsigned long buffer = 0L; //buffer 为定义的存储字节的缓冲区
  buffer |= (unsigned long) code << (32 - bitsize - count);//a|=b等价于a=a|b;
  count += bitsize;
  while (count >= 8){   //如果缓冲区大于8则输出里面的前8位
    fputc(buffer >> 24,out);
    buffer <<= 8;
    count -= 8;
  }
}
```

2.3.1.3、结果分析

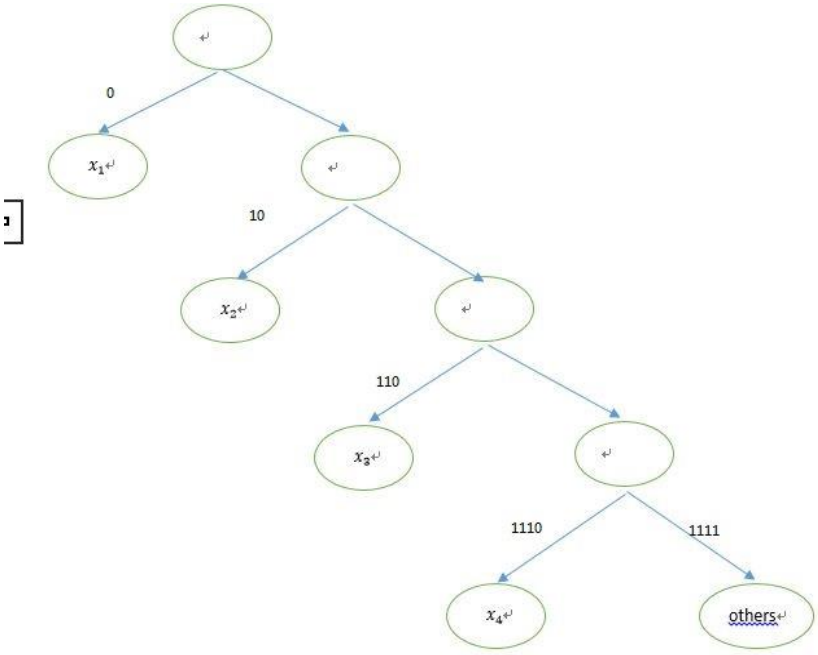
```
C:\Users\encounter>D:\CodeBlocks\Project\Compression\Huffman\bin\Debug\Huffman.e
xe "E:\U2\天文图像\M71_DV436_Sat 26_1_0000.fits"
字符频率4141594
字符频率271415
字符频率266770
字符频率265831
字符总频率8395200
```

```
C:\Users\encounter>D:\CodeBlocks\Project\Compression\Huffman\bin\Debug\Huffman.e
xe "E:\U2\天文图像\M71_DV436_Sat 45_0_0000.fits"
字符频率4184618
字符频率436171
字符频率424566
字符频率412926
字符总频率8395200
```

可以看到出现频率在高的字符几乎占到了总字符的50%，这就意味着，剩下的字符出现的频率相差无几，并且由于建立哈夫曼树硬件上实现比较困难，这样建立哈夫曼树的效果就显得没有必要。

如果将出现频率最高的字符用一个字节代替，而剩余的字节在原基础上扩充一位，这样压缩或每个字符的平均字节数约为$1 \times 50\% + (8+1) \times 50\% = 4.5$。压缩比约为$4.5/8 = 56.25\%$。

出于对代码可复用的考虑，选取出现频率相对较高的四个字符进行编码。下面是对这种方法的分析。



对于这种形状的哈夫曼树，满足$x_1 \geq x_2 \geq x_3 \geq x_4 \geq 1/256$，

要想达到压缩的效果，须有:$x_1 + 2x_2 + 3x_3 + 4x_4$ +(4+8)others ≤8

对等式左边进行操作，注意到$x_1 + x_2 + x_3 + x_4 = 1 - others$

$x_1 + 2x_2 + 3x_3 + 4x_4$ +(4+8)others

$$= x_1 + x_2 + x_3 + x_4 + x_2 + x_3 + x_4 + x_3 + x_4 + x_4 + 12others$$
$$\leq \left(1 + \frac{3}{4} + \frac{2}{4} + \frac{1}{4}\right)(1 - \text{others}) + 12\text{others} = \frac{19}{2}\text{others} + \frac{5}{2} \leq 8$$

即 others≤11/19，$x_1 + x_2 + x_3 + x_4 \geq 8/19$就肯定能得到压缩效果。

如果达不到压缩的效果，须有:$x_1 + 2x_2 + 3x_3 + 4x_4$ +(4+8)others ≥8
对等式左边进行操作，
有$x_1 + 2x_2 + 3x_3 + 4x_4$ +(4+8)others

$$\geq 1 - \text{others} + \frac{6}{256} + 12others = 11\text{others} + \frac{262}{256}$$

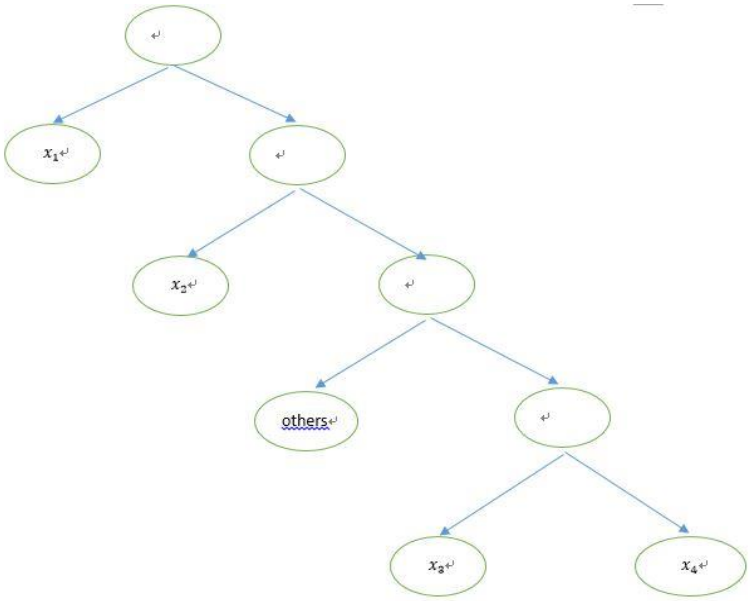
若$11\text{others} + \frac{262}{256} \geq 8$，

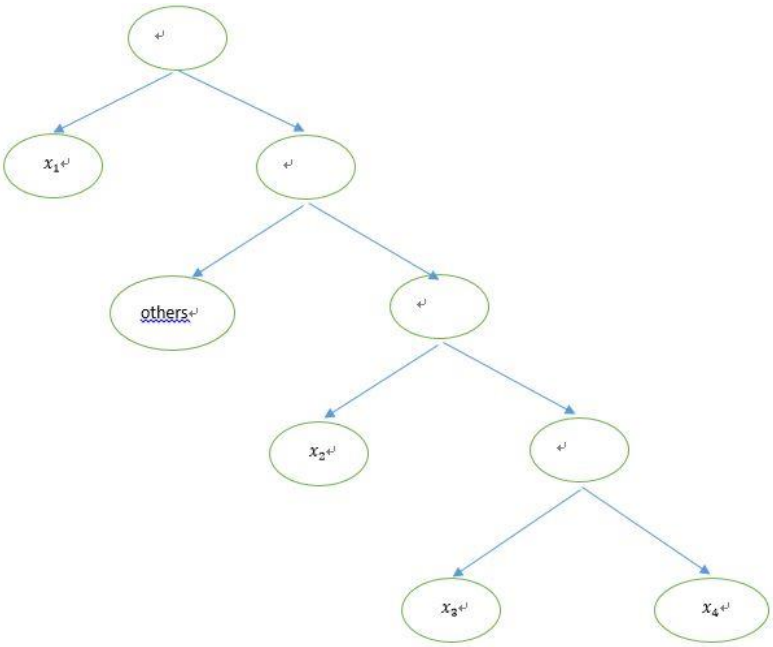
只要 others≥893/1408，或者说$x_1 + x_2 + x_3 + x_4$ ≤515/1408≈ 0.366就肯定不能得到压缩效果。
无疑图片的数据是可以满足要求的。

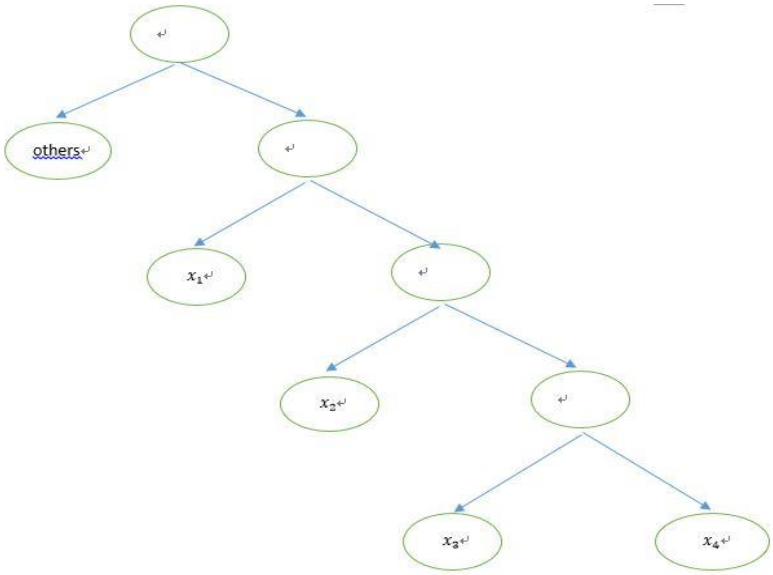
M71_DV436_Sat 26_1_0000.fits	2014/9/10 17:05	FITS 文件	8,199 KB
M71_DV436_Sat 26_1_0000.fits.huf	2014/9/29 19:25	HUF 文件	5,853 KB
M71_DV436_Sat 45_0_0000.fits	2014/9/10 17:05	FITS 文件	8,199 KB
M71_DV436_Sat 45_0_0000.fits.huf	2014/9/29 20:28	HUF 文件	5,277 KB

压缩率分别约为 70%和 65%。

其实对于这种形状的数，字符处于不同的位置，压缩效果可能会更好。比如：







$x_1 + 2x_2 + 3x_3 + 4x_4$ +(4+8)others

相差$\text{others} - x_3$

$x_1 + 2x_2 + 4x_3 + 4x_4$ +(3+8)others

$\text{others} - x_2$

$x_1 + 3x_2 + 4x_3 + 4x_4$ +(2+8)others

$\text{others} - x_1$

$2x_1 + 3x_2 + 4x_3 + 4x_4$ +(1+8)others

欲使前一种算法优于后一种，分别要求$\text{others} \le x_3$，$\text{others} \le x_2$，$\text{others} \le x_1$。说明在这种形状的树的前提下，最后一种方式效果最好。压缩率约为$(414 \times 2 + 27 \times (3 + 4 + 4) + (840 - 414 - 27 \times 3) \times 9) \div 840 \div 8 \approx 63\%$。当然最好的方法还是将出现频率最高的字符用一个字节代替，而剩余的字节在原基础上扩充一位，这种方法压缩率高，方法简单，解压方便，但是对误码敏感。

2.3.2、LZW 编码

统计模型所来用的方式是先将要被编码的数据整体所采用的符号做个统计,然后将重复性大的符号重新以短的码来编排, 而重复性愈小的符号则以愈长的码来编排。

大多数通用的压缩方案还是基于统计的压缩,既出现频率高的字符给予较短的编码来表示,考虑的都是单个字符的概率,而现实中,信源的不同字符之间是存在相关性的,如果考虑这个相关性,则压缩比还可以有显著的提高。

字典模型的编码原理则是以较长的字符串或经常出现的字毋组合构成字典中的数据项,并用相应较短的数字或符号作为代码表示。当从源数据流中读入的数据能与字典中的数据项相匹配,则输出其时应的代码。

这两种方法的焦点不同又使得它们可以组合起来使用,即在字典建模的基础上再辅之以后续的统计编码,以形成压缩性能更高的复合算法。

2.3.2.1、算法简介

字典压缩算法是 1977 年两位以色列科学家,1977 年发表题为“顺序数据压缩的一个通用算法”的论文描述的算法被后人称为 LZ77 算法。1978 年,二人又发表了该论文的续篇“通过可变比率编码的独立序列的压缩”,描述了后来被命名为 LZ78 的压缩算法。

1984 年,T. h. Welch 发表了名为“高性能数据压缩技术”的论文,描述了他在 Sperry 研究中心(该研究中心后来并入了 Unisys 公司)的研究成果,这是 LZ78 算法的一个变种,也就是后来非常有名的 LZW 算法。LZW 继承了 LZ77 和 LZ78 压缩效果好、速度快的优点,而且在算法描述上更容易被人们接受(有的研究者认为是由于 Welch 的论文比 Ziv 和 Lempel 的更容易理解),实现也比较简单。

LZ 系列算法有实现简单,通用性好,速度快,压缩效果好等特性,可以满足不同类型图像对无损压缩的需要的特点,因此使用 LZ 系列算法是数据压缩编码的理想工具。LZ 基于以下思想:待编码的符号串可能包含在已编码的信息结构中,从而呈现出数据冗余。

LZW 压缩算法是 LZ78 算法的一个变种,继承了 LZ77 和 LZ78 压缩效果好、速度快的优点,而且在算法描述上更容易被人们接受,实现也比较简单,是一种比较新颖的压缩方法,其基本原理在于:任何具有某种可预见性的数据,都可以通过用某种标记来表示这种可预见性而使数据变短。

LZW 算法基于字典,压缩有 3 个重要的对象:数据流(Char Stream)、编码流(Code Stream)和编译表(String Table)。在压缩编码时,数据流是输入数据,编码流是输出编码(经过压缩运算的数据);在解码时,编码流则是输入,数据流是输出。编译表是在编码和解码时都需要用借助的字典对象,是整个算法的核心。LZW 压缩算法和其它一些压缩技术的不同之处在于它是动态地标记数据流中出现的重复串。它把在压缩过程中遇到的字符串记录在串表中,在下次又碰到这一字符串的时侯,就用一个代码来表示它,通过用短代码表示相对较长的字符串来压缩数据量。

LZW 的编码词典就象一张转换表,用来存放字符串项,每个表项被分配一个代码,转换表实际上是把 8 位 ASC II 字符集进行了扩充,增加的符号用来表示在文本或图像中出现的可变长长度 ASC II 字符串。扩充后的代码可用 9—12 位甚至更多位来表示,如果用 12 位,那么就可以表示 2 的 12 次方即 4096 个表项,其中 256 个表项用来存放已定义的单个字符,剩下的 3840 个用来存放短语字符串。LZW 编码器通过管理这个词典完成输入与输出之间的转换,即输入字符流后用 n(例如 12 位)位来表示代码流,代码表示单个字符或多个字符组成

的短语字符串。

2.3.2.2、LZW 算法的流程图

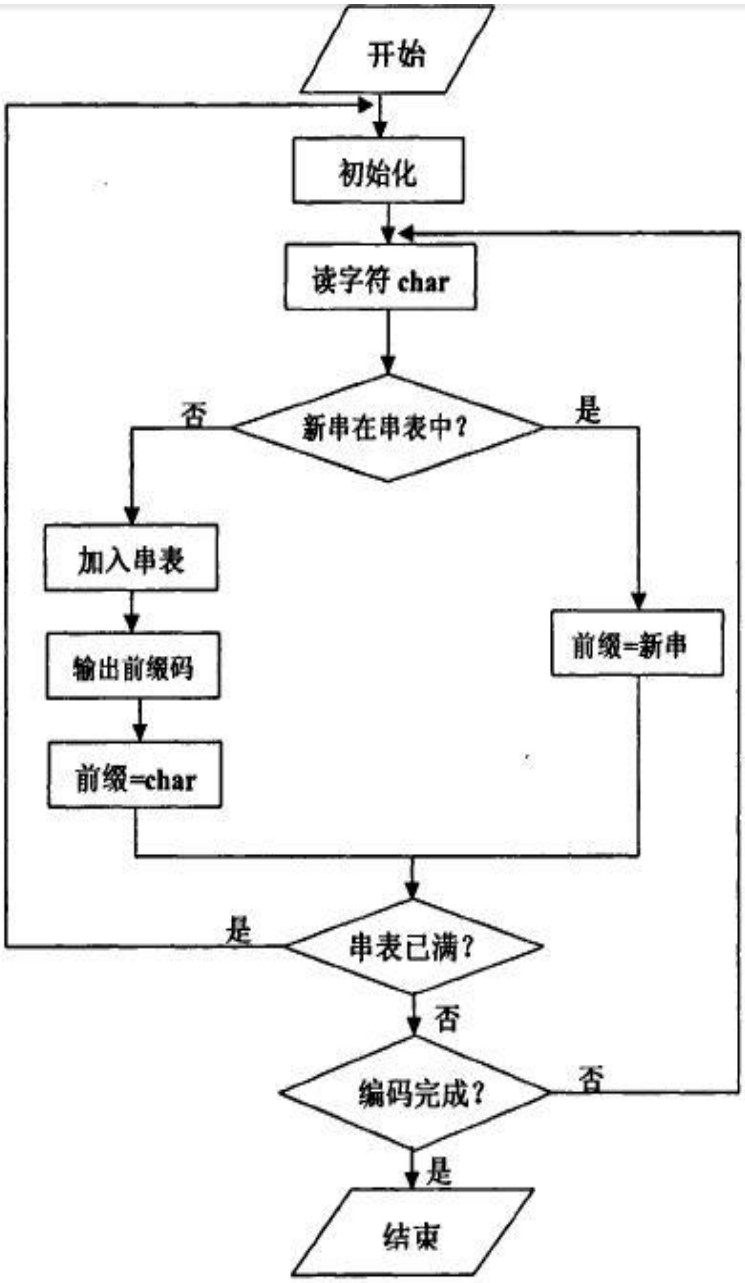


图 1 LZW 压缩算法流程图

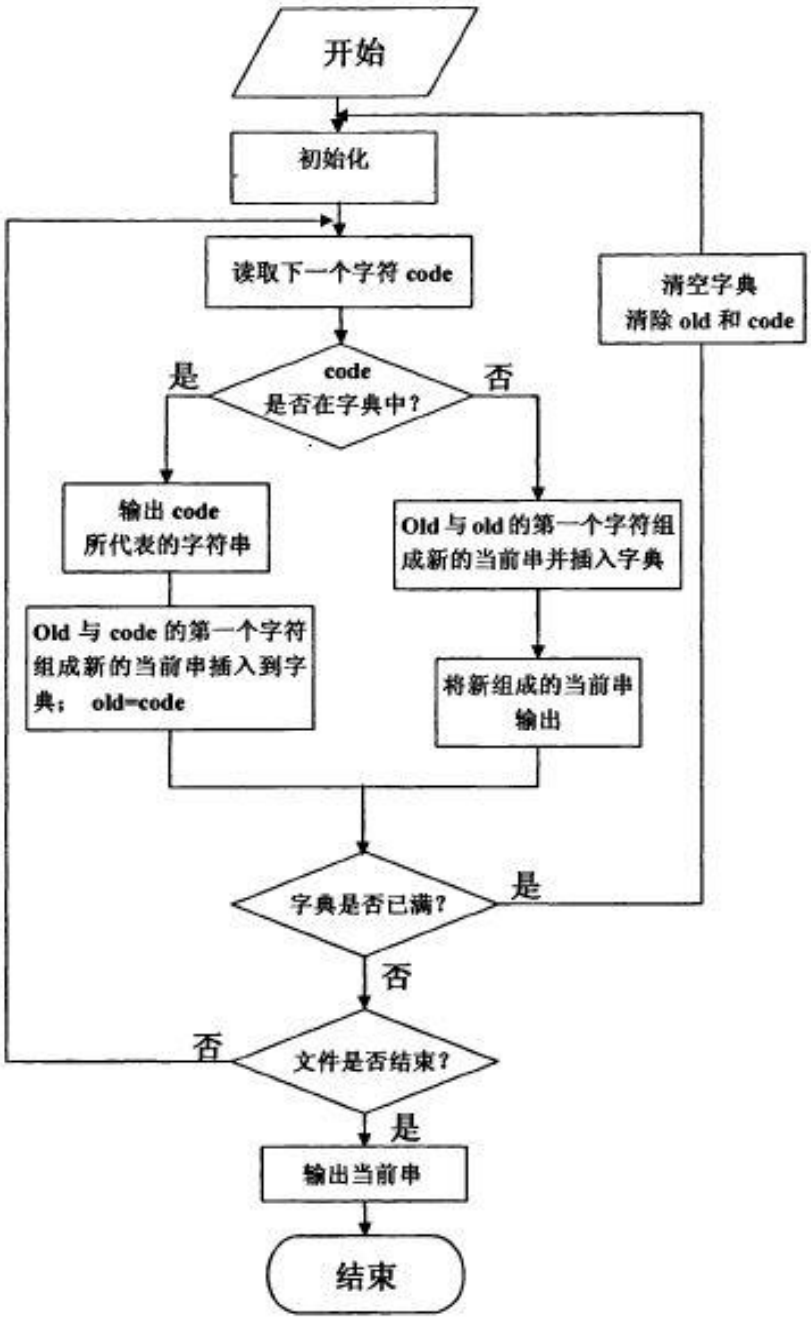


图3 LZW算法解码的流程图

2.3.2.3、LZW 算法示例

- 被压缩原始数据：xxyyyz
- 执行压缩后得到的输出编码用二进制表示为：0001111000，0001111000，0001111001，0100000010，0001111001
- 在压缩过程中存储在字典中的字符串：
- 2 个字符：xx，xy，yy
- 3 个字符：yyz

LZW编码举例，XXYYYZ				
当前被识别序列	被编码的灰度值	编码输出	存入字典内容	存入字典位置
	x			
x	x	0001111000 (x)	xx	0100000000
x	y	0001111000 (x)	xy	0100000001
y	y	0001111001 (y)	yy	0100000010
y	y			
yy	z	0100000010(yy)	yyz	0100000011
z		0001111001 (z)		

- 解压缩过程如下：

第1个	第2个	第3个	第4个	第5个
0001111000	0001111000	0001111001	0100000010	0001111001

- 1)获得第一个编码0001111000。前缀是00，所以在第一个字典中(dic0)；后面8位为01111000，是字符X的ASCII码，输出x

步骤	输出	dic0	dic1
1	x	ASCII码	

- 2)获得第二个编码，也是0001111000。同样输出字符x，同时把字串xx加入到第二个字典中(dicl)。到此为止输出为xx

步骤	输出	dic0	dic1
1	x	ASCII码	
2	x		xx

- 3)获得第三个编码，0001111001。前缀是00，所以也在第一个字典(dic0)中；后面8位为01111001，所以输出y；同时把字串xy加入到第二个字典(dicl)中。到此为止，输出为xxy，字典中的已存储的字符串为：xx，xy

步骤	输出	dic0	dic1
1	x	ASCII码	
2	x		xx
3	y		xy

4)获得第四个编码，0100000010。前缀为01，要在第二个字典dic1中以地址00000010(当前code的后八位)查找，地址为2，但此时第二个字典中地址为2的位置还没有写入字符串。

这是LZW算法解压时会碰到的一个特殊情况，需要通过下面的方法加以解决。把前一个输出的字串加上前一个字串的第一个字符，作为当前要输出的字串，所以输出yy。字典的更新也是一样，把前一个输出的字串加上前输出的字串的第一个字符，加入到相应长度的字典中。到此为止输出为xxyyy，字典中的字符串为：xx，xy，yy

步骤	输出	dic0	dic1
1	x	ASCII码	
2	x		xx
3	y		xy
4	yy		yy

- 5)获得第五个编码，0001111010。前缀为00，输出字符z；同时用yyz(前一个输出字串加上当前输出字串的第一个字符)更新第三个字典。到此为止的输出：xxyyyz，字典中一共存储的字符串：2个字符：xx，xy，yy。3个字符：yyz

步骤	输出	dic0	dic1
1	x	ASCII码	
2	x		xx
3	y		xy
4	yy		yy
5	z		yyz

- 6)解压结束

2.3.2.4、LZW 算法 C++源代码

算法的关键在于建立字典，建立字典的关键在于查找字典。

0	ASCII码
1	ASCII码
⋮	⋮
255	ASCII码
256	clear
257	end
258	
259	
4094	
4095	

如左边所示的字典形状，如果碰到什么字符串，就依次地存入字典，这样码与字符串之间的相关性低，就给建立字典过程中查找字典带来时间上的开销，于是算法采用哈希表的方式建立字典，利用这种相关性，以节省时间。

散列表（Hash table，也叫哈希表），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

其自适应的特点，使得在编码过程中清空字典以及解码过程中不需要压缩过程中建立的字典。

```
/*************************************************************************
*************************************************************************/
//先run一下关掉，在cmd中，拖入exe文件，空格，拖入要压缩的文件，回车
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
using namespace std;
#define hashsize       4096
#define clear           256        /*清除标志位*/
#define teminate        257
#define not_used        -1
#define MAXVAL(n) (( 1 <<( n )) -1)
#define max_bits         12
FILE *in;
FILE *out;
int bitsize = 9;
int maxcode;

//字典数据结构
typedef struct prex_cha{
    int value;                  //值
  unsigned int prefix;          //字符串
  unsigned int character;       //追加的字母
}Hash_Table;

Hash_Table Hash_table[hashsize];//全局变量

initial_hashtable();//字典初始化
output (unsigned int code);//把数据改写成定长8bit流输出
int find_match(int prefix,unsigned int character);//查找字典，看是否已经存入
void lzwcompression ();//压缩
```

```
int main(int argc,char* argv[]){
  char filename[255];
  if(argc < 2){//0,1
    printf("usage:the command format is:lzw_compression <filename>!");
    return(1);
  }
  if((in = fopen(argv[1], "rb")) == NULL){
    printf ("Cannot open input file - %s/n", argv[1]);
    exit (1);
  }
  strcpy(filename, argv[1]);
 //加上后缀.lzw,表明是压缩文件
  strncat(filename, ".lzw", 4);
  if((out = fopen(filename, "wb")) == NULL){
    printf("Cannot open output file - %s/n", filename);
    fclose(in);
    exit(1);
  }
  maxcode = MAXVAL(bitsize);
  lzwcompression();
  fclose (in);
  fclose (out);
  return 0;
}
```

```
void initial_hashtable(){//字典初始化
    int i;
  for (i=0; i<hashsize; i++){
    Hash_table[i].value = not_used;
  }
}
//把数据改写成bit流输出
void output (unsigned int code){
  static int count = 0;
  static unsigned long buffer = 0L; //buffer 为定义的存储字节的缓冲区
  buffer |= (unsigned long) code << (32 - bitsize - count);
  count += bitsize;
  while (count >= 8){    //如果缓冲区大于8则输出里面的前8位
    fputc(buffer >> 24,out);
    buffer <<= 8;
    count -= 8;
  }
}

int find_match(int prefix,unsigned int character){
  int index;
  index = prefix % 4096;
  while (1){
    if(Hash_table[index].value == not_used){
        return (index);
    }
    if(Hash_table[index].prefix == prefix && Hash_table[index].character == character){
        return (index);
    }
    index = index + 1;
    if(index >= 4096){
      index = index - 4096;
    }
  }
}

void lzwcompression (){
  unsigned int prefix;
  unsigned int character;
  unsigned int index;
  unsigned int next_code = 258;            //当前字典的标号
  initial_hashtable ();
  prefix = fgetc (in);

  while ((character = fgetc(in)) !=(unsigned) EOF){
    index = find_match (prefix,character);
    if(Hash_table[index].value != not_used){//能够找到
      prefix = Hash_table[index].value;
    }
    else{
      if(next_code <= maxcode){//不能找到，是新的字符串，则添加到表中
        Hash_table[index].value = next_code++;
        Hash_table[index].character =  character;
        Hash_table[index].prefix = prefix;
      }
      output(prefix);
      prefix = character; //把后缀给前缀，准备下次输入

      //特殊标志，当位数必须增加时
      if(next_code > maxcode){
        if(bitsize < 12){
          maxcode = MAXVAL(++bitsize);
        }
        else {//达到4096时候，必须清除哈希表，重新开始
          output (256);        //输出清除标志到文件中
          initial_hashtable();
          next_code = 258;
          bitsize = 9;
          maxcode = MAXVAL(bitsize);//maxcode 变为511
```

```
        }
      }
    }//if一else结束
  }//while结束
  output(prefix);       //输出最后一个
  if(next_code == maxcode){
    //如果在最后的哈息表刚好刚好是maxcode，则也必须把位数增加一位
    ++bitsize;
  }
  output(257);    //输出结束标志
  output(0);          //解码时要用到
  output(0);
  output(0);
}
```

2.3.2.5、结果分析

压缩率分别约为 45%，37%。

2.3.2.6、LZW 算法优点

- 字典压缩算法利用简单代码代替符号串，使得编码本身的影响被极大的削弱，而将焦点集中到模型的建立上来。通过对输入码流进行分析，能够自适应地生成一个包含输入流中不重复的字符串表，然后将每一个字串加入字典，映射为一个独立的码字输出，因此充分利用了相邻数据的相关性。数字图像数据是高度相关的，存在大量的局部相同或相似性数据，所以采用该算法在通用数据压缩领域，尤其是数字图像方面，可以取得相当好的压缩率。
- LZW 采用了一种先进的串表压缩，将每个第一次出现的串放在一个串表中，并用一个数字来表示该串，在压缩文件中只存贮数字而不存贮串，从而使得图象文件的压缩效率能够得到较大的提高。重要的是字典信息是完全包含在压缩后的数据中的，解压程序可以动态的从压缩过的数据中构造出来，LZW 压缩算法不管是在压缩还是在解压缩的过程中都能正确的建立这个串表，当压缩结束后，字典中的内容不需要保存和传输，在解压的时候可以生成与压缩时一模一样的字典，非常的方便。压缩或解压缩完成后这个串表又被丢弃。
- LZW 算法逻辑简单，实现速度快，擅长于压缩重复出现的字符串。LZW 编码只需扫描一次数据，无需有关输入数据统计量的先验信息而运算时间正比于消息的长度。而统计编码，需要有关输入数据统计量的先验信息。LZW 算法相对与其他算法，更有利于用硬件实现。
- LZW 算法与其他算法相比具有自适应的特点，即可以根据压缩内容的不同动态建立不同的字典，以减少冗余度，提高压缩比，实时性好；并且解压时这个字典不需要与压缩代码同时传送，而是在解压过程中逐步建立与压缩时完全相同的字典，从而完整、准确地恢复被压缩内容，字典对于存储器的容量要求也不高，一般为几 K。

因此，压缩、解压速度快，所占存储空间较小，是一种解码速度与压缩性能综合指标相当好的一种压缩算法。

2.3.2.7、LZW 算法缺点

- 算法生成的串表要用额外存储器，在面积的限制下，存储 memory 的容量受到一定限制。
- 需要压缩的数据流中存在大量简单字符，需要设置额外存储，增加了字典开销。
- 算法受到字典容量、数据复杂度和计算速度等因素的限制，串表的项数比较有限。
- 算法通常是在字典满时将其清空、重建，由于字典建立初期的字符串表很少，需要压缩的数据与字典中字串的匹配率很低，导致字典重建初期压缩效果很差。需要改进字典替换策略。
- 对较大的文件进行压缩编码时，频繁的查找字典、磁盘读写访问会降低数据编码的速度。
- 一般信源所具有的局部平稳性随缓存容量加大。当数据流相对杂乱无章时，该算法压缩效果比较一般。

2.3.3、综合分析

M71_DV436_Sat 26_1_0000.7z	2014/9/26 16:31	7Z 文件	2,685 KB
M71_DV436_Sat 26_1_0000.fits	2014/9/10 17:05	FITS 文件	8,199 KB
M71_DV436_Sat 26_1_0000.fits.huf	2014/9/29 19:25	HUF 文件	5,853 KB
M71_DV436_Sat 26_1_0000.fits.lzw	2014/9/26 16:35	LZW 文件	3,764 KB
M71_DV436_Sat 26_1_0000.rar	2014/9/29 22:24	WinRAR 压缩文件	5,436 KB

M71_DV436_Sat 45_0_0000.7z	2014/9/29 22:23	7Z 文件	2,288 KB
M71_DV436_Sat 45_0_0000.fits	2014/9/10 17:05	FITS 文件	8,199 KB
M71_DV436_Sat 45_0_0000.fits.huf	2014/9/29 20:28	HUF 文件	5,277 KB
M71_DV436_Sat 45_0_0000.fits.lzw	2014/9/26 16:37	LZW 文件	3,056 KB
M71_DV436_Sat 45_0_0000.rar	2014/9/29 22:23	WinRAR 压缩文件	2,331 KB

可以看出，LZW 编码压缩效果比 Huffman 编码效果要好，但是都不如压缩软件的效果。并且由于为了提高单个算法的效率 LZW 编码和 Huffman 编码均采用的变长码，使得两种算法结合使用的效果反而不如单个使用。

第三部分：总结

3.1、收获

了解了物理世界的信息是怎样称为数字图像的，编码概念，几种不同的编码方式。

Huffman 编码和 LZW 编码，分别针对单个字符的概率，不同字符之间的相关性，实现压缩，这种思想令我思考。

LZW 编码实现过程中用到了 Hash table，把关键码值映射到表中一个位置来访问记录，

以加快查找的速度。通过使用这种方法，对这种方法有了更深的体会。

学习 Verilog 语言，体会到真实的硬件实现与程序设计语言之间的差别。

在实验室的组会中，学习了一些计算机、图形图像相关的知识。锻炼表达自己的能力。

3.2、不足

对算法研究深度不够，只建有宽泛的概念，也并没有找出什么优化方法。

对于压缩文件没有写出头文件。

压缩算法可以软件实现也可以硬件实现，尽管用软件压缩方法可以较好的实现数据压缩的目的，但软件压缩解压有个致命的弱点就是过多地消耗宝贵地 CPU 资源、速度慢。特别是对于大批量数据压缩解压时，其速度和 CPU 资源占用率都是令人难以忍受的。虽然现在 CPU 处理速度不断提高达到了几个 G，但是还是不能满足某些系统快速地实时地压缩解压。所以软件压缩一般应用在对时间要求不高的场合，而对运行速度有特殊要求的情况下，对数据进行实时压缩一般都要用硬件实现。

而由于自己对 Verilog 的时序，状态机，FIFO、储存等概念理解不好，并没用达到用硬件实现的目的。

3.4、参考文献

1、张春田、苏育挺、张静.数字图像压缩编码. 北京：清华大学出版社，2006.

2、马鸿泰.国家天文台静、动态图像编码与传输的研究. 北京邮电大学硕士学位论文.

3. 郑翠芳. 几种常用无损数据压缩算法研究. 四川绵阳:中国工程物理研究院计算机应用研究所

4、高建国.LZW 无损压缩算法的研究及改进. 北京工业大学硕士学位论文.

5、黄静. LZW 压缩算法 VC 实现、改进及其应用研究. 中南大学硕士学位论文.

6、王智.LZW 字典压缩改进算法研究及 FPGA 硬件实现. 南京师范大学硕士学位论文.

7、李雷定.基于 FPGA 的数据实时无损压缩系统设计. 中北大学硕士学位论文.

8、刘洪庆. 基于 LZW 算法的数据无损压缩硬件实现. 浙江大学硕士学位论文.

9、栗志、周卫红. 数字图像压缩方法在天文上的应用. 昆明：中国科学院云南天文台.

10、朱贵富.天文图像无损压缩算法研究与实现. 昆明理工大学硕士学位论文.

3.4、致谢

感谢彭波师兄在课题研究方面的指导。

感谢金西老师初步提供的研究方向。

感谢实验室的师兄师姐在生活上的照顾，在学习上的指点，在组会交流时从你们这里学到很多。

感谢董小波老师，以及各位看这篇报告的老师百忙之中能抽出时间来。

感谢参考文献中的各位作者。

感谢家庭、学校、国家对我的培养。