第 3 章　数据无损压缩

在数据存储和传输系统中，增加冗余数据可提高数据的可靠性，而消除或减少冗余数据可降低对存储容量和传输带宽的要求。本章的核心内容是介绍几种消除或减少冗余数据的数据无损压缩技术，包括统计编码、RLE 编码和词典编码[1][2]。

3.1　数 据 冗 余

数据可被压缩的依据是数据本身存在冗余。所有无损压缩算法的共同点都是利用数据本身的冗余性，其差别主要体现在压缩比上，压缩比越高表示冗余数据消除得越多，压缩比的上限值由数据集的熵限定。

3.1.1　冗余概念

在数据压缩技术中，我们会经常遇到“冗余(redundancy)”这个术语，在不同场合下，“冗余”有不同的含义，现从下面三个方面加以说明。

(1) 人为冗余：①在信息处理系统中，使用两台计算机做同样的工作是提高系统可靠性的一种措施。在这样的系统中，一台计算机在工作，而另一台计算机处于等待状态。如果正在工作的机器出现故障，则由处于等待状态的机器马上接替，我们就说这样的系统是冗余的系统，备用设备称为冗余设备。②在数据存储和传输中，为了检测和恢复在数据存储或数据传输过程中出现的错误，根据使用的算法的要求，在数据存储或数据传输之前把额外的数据添加到用户数据中，这个额外的数据就是冗余数据。

从这两个例子中可以看到，冗余设备和冗余数据都是人为添加的，目的是为了提高系统的可靠性和保证数据的正确性。由此可见，冗余并非多余，冗余是人为的。

(2) 视听冗余：由于人的视觉系统和听觉系统的局限性，在图像数据和声音数据中，有些数据确实是多余的，使用算法将其去掉后并不会丢失实质性的信息或含义，对理解数据表达的信息几乎没有影响。这种冗余称为视听冗余。例如，在 BT.601 数字化标准中，使用 4∶2∶2 的子采样对视像进行数字化，就是单纯地利用了人的视觉特性，去除冗余的数据。

(3) 数据冗余：不考虑数据来源时，单纯数据集中也可能存在多余的数据，去掉这些多余数据并不会丢失任何信息，这种冗余称为数据冗余，而且还可定量表达。

在介绍数据冗余量的定量表达之前，先介绍信息论①中的几个基本术语。

---

① 信息论(information theory，IT)是 1948 年创建的数学理论的分支学科，研究信息的编码、传输和存储。该术语源于 Claude Shannon(香农)发表的“A Mathematical Theory of Communication”论文题目，提议用二进制数据对信息进行编码。它最初只应用于通信工程领域，后来扩展到包括计算在内的其他领域，如信息的存储和检索等。在通信方面，主要研究数据量、传输速率、信道容量、传输正确率等问题。

3.1.2 决策量

在有限数目的互斥事件集合中，决策量(decision content)是事件数的对数值，在数学上表示为公式 3-1：

$$H_0 = \log(n) \tag{3-1}$$

其中，n 是事件数。

对数的底数决定决策量的单位。决策量可使用的单位包括：(1)Sh (Shannon)：用于以 2 为底的对数；(2)Nat (natural unit)：用于以 e 为底的对数；(3)Hart (hartley)：用于以 10 为底的对数。这三种单位之间的转换关系如下：

$$\begin{aligned}
1\ \text{Sh} &= 0.693\ \text{Nat} = 0.301\ \text{Hart}\\
1\ \text{Nat} &= 1.433\ \text{Sh} = 0.434\ \text{Hart}\\
1\ \text{Hart} &= 3.322\ \text{Sh} = 2.303\ \text{Nat}
\end{aligned}$$

3.1.3 信息量

信息量(information content)是具有确定概率事件的信息的定量度量，在数学上定义为公式 3-2：

$$I(x) = \log_2[1/p(x)] = -\log_2 p(x) \tag{3-2}$$

$p(x)$是事件 x 出现的概率。

【例 3.1】 假设 $X=\{a,b,c\}$是由三个事件构成的集合，$p(a)=0.5$，$p(b)=0.25$，$p(c)=0.25$ 分别是事件 a、b 和 c 出现的概率，这些事件的信息量分别为：

$$I(a) = \log_2[1/(0.50)]\text{Sh} = 1\ \text{Sh}$$
$$I(b) = \log_2[1/(0.25)]\text{Sh} = 2\ \text{Sh}$$
$$I(c) = \log_2[1/(0.25)]\text{Sh} = 2\ \text{Sh}$$

对一个等概率事件的集合，每个事件的信息量等于该集合的决策量。

3.1.4 信息的熵

在信息论中，熵(entropy)是指消息中的信息量的度量。在数据压缩技术中，熵是指非冗余的且不压缩的数据量的度量，单位为位(bit)。

按照香农(Shannon)的理论，在有限的互斥和联合穷举事件的集合中，熵(entropy)定义为事件的信息量的平均值，也称事件的平均信息量(mean information content)，表示为公式 3-3：

$$H(X) = \sum_{i=1}^{n} h(x_i) = \sum_{i=1}^{n} p(x_i)I(x_i) = -\sum_{i=1}^{n} p(x_i)\log_2 p(x_i) \tag{3-3}$$

其中：

(1) $X=\{x_1,x_2,\cdots,x_n\}$ 是事件 $x_i(i=1,2,\cdots,n)$ 的集合，并满足 $\sum_{i=1}^{n} p(x_i)=1$；

(2) $I(x_i)=-\log_2 p(x_i)$表示某个事件 x_i 的信息量，其中 $p(x_i)$为事件 x_i 出现的概率，$0<p(x_i)\leqslant 1$；$h(x_i)=-p(x_i)\log_2 p(x_i)$表示事件 x_i 的熵。

【例 3.2】 $X=\{a,b,c\}$是由三个符号构成的集合，符号 a、b 和 c 出现的概率分别为 $p(a)=0.5$，$p(b)=0.25$，$p(c)=0.25$，那么符号 a、b 和 c 的熵分别等于 0.5、0.5、0.5，这个集合的熵为：

$$H(X) = p(a)I(a) + p(b)I(b) + p(c)I(c) = 1.5(\text{Sh})$$

3.1.5 数据冗余量

在信息论中，数据的冗余量(R)定义为决策量(H_0)超过熵(H)的量，数学上表示为公式 3-4：

$$R = H_0 - H \tag{3-4}$$

【例 3.3】 令$\{a,b,c\}$为三个事件构成的数据集，它们出现的概率分别为 $p(a)=0.5$，$P(b)=0.25$，$p(c)=0.25$，这个数据集的冗余量则为：

$$R = H_0 - H = \log_2 3 - \left[-\sum_{i=1}^{n} p(x_i)\log_2 p(x_i)\right] = 1.58 - 1.50 = 0.08(\text{Sh})$$

3.2 统 计 编 码

统计编码是给已知统计信息的符号分配代码的数据无损压缩方法。香农-范诺编码和霍夫曼编码的原理相同，都是根据符号集中各个符号出现的频繁程度来编码，出现次数越多的符号，给它分配的代码位数就越少；算术编码使用 0 和 1 之间的实数的间隔长度代表概率大小，概率越大间隔越长，编码效率可接近于熵。

3.2.1 香农-范诺编码

在香农的源编码理论中，熵的大小表示非冗余的不可压缩的信息量。在计算熵时，如果对数的底数用 2，熵的单位就用“香农(Sh)”，也称“位(bit)”。例如，事件 x_i 的熵 $h(x_i)=p(x_i)\log_2(1/p(x_i))$，表示编码符号 x_i 所需要的位数。“位”是 1948 年 Shannon 首次使用的术语。下面通过一个具体例子说明如何对概率已知的数据进行编码。

【例 3.4】 有一幅由 40 个像素组成的灰度图像，灰度共有 5 级，分别用符号 A、B、C、D 和 E 表示。40 个像素中出现灰度 A 的像素数有 15 个，出现灰度 B 的像素数有 7 个，出现灰度 C 的像素数有 7 个，其余情况见表 3-1。(1)计算该图像可能获得的压缩比的理论值；(2)对 5 个符号进行编码；(3)计算该图像可能获得的压缩比的实际值。

表 3-1 符号在图像中出现的数目

符　　号	A	B	C	D	E
出现的次数	15	7	7	6	5
出现的概率	$\frac{15}{40}$	$\frac{7}{40}$	$\frac{7}{40}$	$\frac{6}{40}$	$\frac{5}{40}$

(1) 压缩比的理论值：按照常规的编码方法，表示 5 个符号最少需要 3 位，如用 000 表示 A，001 表示 B，…，100 表示 E，其余 3 个代码 (101，110，111)不用。这就意味每个像素用 3 位，编码这幅图像总共需要 120 位。按照香农理论，这幅图像的熵为：

$$\begin{aligned} H(X) &= -\sum_{i=1}^{n} p(x_i)\log_2 p(x_i) \\ &= -p(A)\log_2(p(A)) - p(B)\log_2(p(B)) - \cdots - p(E)\log_2(p(E)) \\ &= (15/40)\log_2(40/15) + (7/40)\log_2(40/7) + \cdots + (5/40)\log_2(40/5) \approx 2.196 \end{aligned}$$

这个数值表明，每个符号不需要用 3 位代码表示，用 2.196 位就可以，40 个像素只需 87.84 位，因此理论上的压缩比为 120∶87.84≈1.37∶1，实际上就是 3∶2.196≈1.37。

(2) 符号编码：对每个符号进行编码时采用“从上到下”的方法。首先按照符号出现的频度或概率排序，如 A、B、C、D 和 E，见表 3-2。然后使用递归方法分成两个部分，每一部分具有近似相同的次数，如图 3-1 所示。

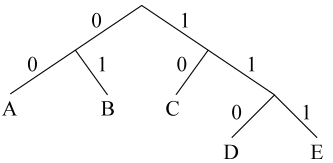

图 3-1　香农-范诺算法编码举例

表 3-2　Shannon-Fano 算法举例

符号	出现的次数($p(x_i)$)	$\log_2(1/p(x_i))$	分配的代码	需要的位数
A	15 (0.375)	1.4150	00	30
B	7 (0.175)	2.5145	01	14
C	7 (0.175)	2.5145	10	14
D	6 (0.150)	2.7369	110	18
E	5 (0.125)	3.0000	111	15

(3) 压缩比的实际值：按照这种方法进行编码需要的总位数为 30＋14＋14＋18＋15＝91，实际的压缩比为 120∶91≈1.32∶1。

最早阐述和实现这种编码方法的人是 Shannon(1948 年)和 Fano(1949 年)，因此将这种“从上到下”的熵编码方法称为香农-范诺(Shannon-Fano)编码法。

3.2.2　霍夫曼编码

霍夫曼(D. A. Huffman)在 1952 年提出了“从下到上”的熵编码方法，因此被称为霍夫曼编码(Huffman coding)，现在已广泛用在各种信息编码标准中。

下面仍然用一个具体例子来说明霍夫曼编码的编码步骤和性能。

【例 3.5】 有一幅 39 个像素组成的灰度图像，灰度共有 5 级，分别用符号 A、B、C、D 和 E 表示，每个符号在图像中出现的次数见表 3-3。(1)计算该图像可能获得的压缩比的理论值；(2)对 5 个符号进行编码；(3)计算该图像可能获得的压缩比的实际值。

表 3-3　霍夫曼编码举例

符号	出现的次数	$\log_2(1/p(x_i))$	分配的代码	需要的位数
A	15(0.3846)	1.38	0	15
B	7(0.1795)	2.48	100	21
C	6(0.1538)	2.70	101	18
D	6(0.1538)	2.70	110	18
E	5(0.1282)	2.96	111	15

(1) 压缩比的理论值：按照常规编码方法，5 个符号至少要用 3 位组成的代码表示。按照香农理论，这幅图像的熵也就是每个符号的平均长度为

$$H(S)=(15/39)\times\log_2(39/15)+(7/39)\times\log_2(39/7)+\cdots+(5/39)\times\log_2(39/5)$$
$$\approx 2.1859$$

因此，理论上可获得的压缩比为 3∶2.1859≈1.37。

(2) 符号编码：按如下步骤进行。

步骤 1：初始化，根据符号概率的大小，按由大到小的顺序对符号进行排序，如表 3-3 和图 3-2 所示。

步骤 2：把概率最小的两个符号组成一个节点，如图 3-2 中的 D 和 E 组成节点 P_1。

步骤 3：重复步骤 2，得到节点 P_2、P_3 和 P_4，这样就形成了一棵“树”。树上的 P_4 称为根节点，其余的节点称为“叶节点”。在这个编码步骤中特别要注意：“重复步骤 2”时要“把概率最小的两个符号组成一个节点”。

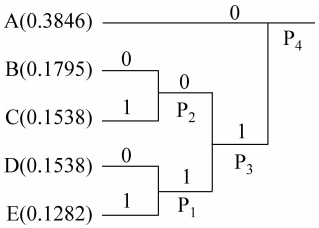

图 3-2　霍夫曼编码方法

步骤 4：从根节点 P_4 开始到相应于每个符号的“树叶”，按从上到下的顺序选择两种方法之一进行标注：①在上枝叶上标“0”(对应概率大的节点)，在下枝叶上标“1”(对应概率小的节点)；②在上枝叶上标“1”(对应概率大的节点)，在下枝叶上标“0”(对应概率小的节点)。这两种标法的最后结果仅仅是分配的代码不同，代码的平均长度是相同的。

步骤 5：从根节点 P_4 开始顺着树枝到每个叶子分别写出每个符号的代码，见表 3-3。

(3) 压缩比的实际值：编码 39 个像素需要 39×3=117(位)，实际使用的总位数为 15+21+18+18+15=87(位)，实际的压缩比为 117∶87≈1.34。

采用霍夫曼编码方法给每个符号分配的代码的长度不是固定的，在编码时也不需要在生成的码流中附加同步代码，原因是在解码时可按霍夫曼码本身的特性加以区分。例如，码流中的第 1 位为 0，那么第一个符号肯定是 A，因为表示其他符号的代码没有一个是以 0 开始的，因此下一位就表示下一个符号代码的第 1 位。同样，如果出现 110，那么它就代表符号 D。这就意味着，编码时需要生成解释各种代码意义的码表，在解码时就可根据这张码表进行译码。

采用霍夫曼编码时有两个问题值得注意：①霍夫曼码没有错误保护功能。在存储或传输过程中，如果码流中没有出现错误，解码时就能一个接一个地正确译出代码。如果码流中出现错误，哪怕只有一位出错，解码时不但这个代码会被译错，更糟糕的是还会导致后面的代码也会译错，这种现象称为错误传播(error propagation)。计算机对这种错误也无能为力，说不出错在哪里，更谈不上去纠正它。②霍夫曼码是可变长度码，因此很难随意查找或调用压缩文件中的内容，然后再译码，这就需要在编码时加以考虑。尽管如此，霍夫曼编码还是得到广泛应用。

与香农-范诺编码相比，这两种方法产生的代码都是自含同步的代码，在编码之后的码流中都不需要另外添加标记符号，即在译码时分割符号的特殊代码。此外，霍夫曼编码方法的编码效率比香农-范诺编码效率高一些。请读者自行验证。

3.2.3　算术编码

算术编码(arithmetic coding)是给已知统计信息的符号分配代码的数据无损压缩技术。它的基本思想是，用 0 和 1 之间的一个数值范围表示输入流中的一个字符，而不是给输入流中的每个字符分别指定一个码字，实质上是为整个输入字符流分配一个“码字”，因此它的编码效

率可接近于熵。下面用两个具体例子来说明算术编码的编码步骤和性能。

【例 3.6】 使用算术编码对二进制消息序列 10 00 11 00 10 11 01 … 进行编码。假设信源符号为{00，01，10，11}，它们的概率分别为{0.1，0.4，0.2，0.3}。

根据信源符号的概率把间隔[0,1)分成如表 3-4 所示的 4 个子间隔：[0,0.1)，[0.1，0.5)，[0.5,0.7)，[0.7,1)。其中的[x,y)表示半开放间隔，即包含 x 不包含 y，x 称为低边界或左边界，y 称为高边界或右边界。

表 3-4　例 3.6 的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)

编码时输入第 1 个符号是 10，找到它的编码范围是[0.5，0.7]。由于消息中第 2 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。整个编码过程如图 3-3 所示，这个例子的编码和译码的全过程分别表示在表 3-5 和表 3-6 中。

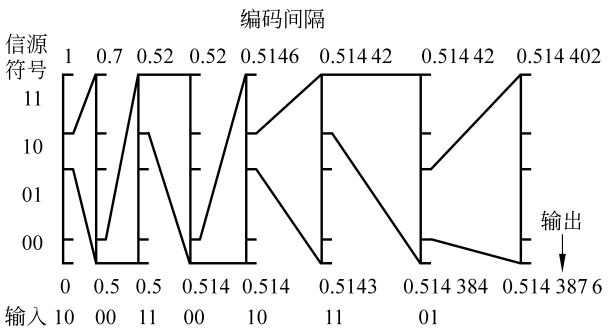

图 3-3　例 3.6 的算术编码过程

表 3-5　例 3.6 的编码过程

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	[0.5143，0.514 42]	[0.514，0.5146]间隔的第五个 1/10 开始，2 个 1/10
6	11	[0.514 384，0.514 42]	[0.5143，0.514 42]间隔的最后三个 1/10
7	01	[0.514 387 6，0.514 402]	[0.514 384，0.514 42]间隔的 4 个 1/10，从第一个 1/10 开始
8	从[0.514 387 6，0.514 402]中选择一个数(如 0.514 387 6)作为输出		

表 3-6　例 3.6 的译码过程

步骤	间　　隔	译码符号	译码判决
1	[0.5, 0.7]	10	0.514 39 在间隔 [0.5, 0.7)
2	[0.5, 0.52]	00	0.514 39 在间隔 [0.5, 0.7)的第 1 个 1/10
3	[0.514, 0.52]	11	0.514 39 在间隔[0.5, 0.52)的第 7 个 1/10
4	[0.514, 0.5146]	00	0.514 39 在间隔[0.514, 0.52)的第 1 个 1/10
5	[0.5143, 0.51442]	10	0.514 39 在间隔[0.514, 0.5146)的第 5 个 1/10
6	[0.514 384, 0.514 42]	11	0.514 39 在间隔[0.5143, 0.514 42)的第 7 个 1/10
7	[0.514 39, 0.514 394 8]	01	0.514 39 在间隔[0.514 39, 0.514 394 8]的第 1 个 1/10
8	译码的消息：10 00 11 00 10 11 01		

执行算术编码的编码算法可用如下伪代码(即描述算法或程序的非正式符号)表示：

```
------------------------------------------------------------
Low=0.0; high=1.0;
while not EOF do
    range=high-low; read(c);
    high=low+range×high_range(c);
    low=low+range×low_range(c);
enddo
output(low);
------------------------------------------------------------
```

【例 3.7】 使用算术编码方法对输入序列 x_n：a_2,a_1,a_3,…进行编码。该序列由 4 个符号组成,它们的概率和初始间隔见表 3-7。

表 3-7　例 3.7 的信源符号概率和初始编码间隔

信源符号 a_i	a_1	a_2	a_3	a_4
概率 $p(a_i)$	$p_1=0.5$	$p_2=0.25$	$p_3=0.125$	$p_4=0.125$
初始编码间隔	[0, 0.5]	[0.5, 0.75)	[0.75, 0.875)	[0.875, 1)

输入序列为 x_n：a_2,a_1,a_3,…,使用算术编码的过程如图 3-4 所示,现说明如下：

(1) 输入第 1 个符号 a_2 时,初始间隔为[0.5, 0.75],左右边界的二进制数分别为 $L=0.5=0.1$(B),$R=0.75=0.11$(B)。

(2) 输入第 2 个符号 a_1 时,它的间隔为[0.5, 0.625),左右边界的二进制数分别为 $L=0.5=0.1$(B),$R=0.101$(B)。

(3) 输入第 3 个符号 a_3 时,它的间隔为[0.593 75,0.609 375],左右边界的二进制数分别为：$L=0.593\ 75=0.10011$(B),$R=0.609\ 375=0.100111$(B)。

…

在编码器的输出端发送的符号是：10011…。

在译码时,算术译码器接收到的第 1 位是“1”,它的间隔范围就限制在[0.5, 1),而在这个

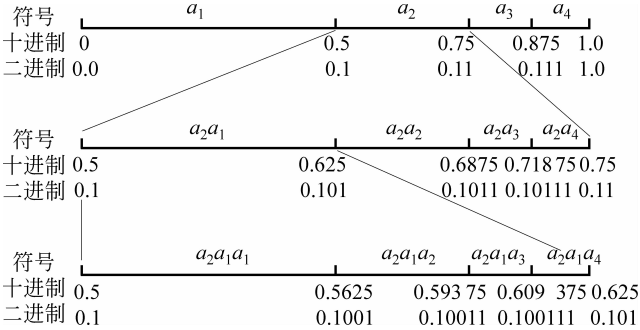

图 3-4　例 3.7 的算术编码过程

范围里的符号有可能是 a_2、a_3 或 a_4，说明第 1 位没有包含足够的译码信息。在接收第 2 位之后就变成“10”，它落在[0.5，0.75)的间隔里，由于这两位表示的符号都指向 a_2 的间隔，因此就可断定第一个符号是 a_2。译码的整个过程见表 3-8。

表 3-8　例 3.7 的译码过程

接收的数字	间隔	译码输出
1	[0.5，1)	—
0	[0.5，0.75)	a_2
0	[0.5，0.609 375)	a_1
1	[0.5625，0.609 375)	—
1	[0.593 75，0.609 375)	a_3
…	…	…

算术编码和霍夫曼编码相比，有如下几个异同点：(1)算术编码的编码效率更高些；(2)它们都是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；(3)它们的信源概率都是固定的，而且要事先统计确定；(4)都有相应的“自适应编码”。由于事先知道精确的信源概率是很难的，或者是不切实际的，因此要在编码过程中，根据符号出现的频繁程度不断修改信源符号的概率，估算信源符号概率的过程叫作建模(modeling)。采用这种技术开发的编码分别称为“自适应霍夫曼编码”和“自适应算术编码”。

3.3　RLE 编码

行程长度编码(run-length encoding，RLE)是数据无损压缩技术。它利用连续数据单元有相同数值这一特点对数据进行压缩。在编码时，对相同的数值只编码一次，同时计算相同数值连续重复的次数，称为“行程程度”。

【例 3.8】　假定有一幅灰度图像，第 n 行的像素值如图 3-5 所示。用 RLE 编码方法得到的代码为：80315084180。代码中用黑体表示的数字是行程长度，黑体字后面的数字代表像素的颜色值。例如，黑

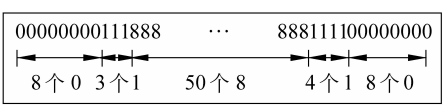

图 3-5　RLE 编码概念

体字 50 代表有连续 50 个像素具有相同的颜色值，它的颜色值是 8。

对比 RLE 编码前后的代码数可以发现，在编码前要用 73 个代码表示这一行的数据，而编码后只需要用 11 个代码表示原来的 73 个代码，压缩前后的数据量之比约为 7∶1。这说明 RLE 确实是一种压缩技术，而且这种编码技术相当直观，也非常经济。RLE 所能获得的压缩比大小主要是取决于数据本身的特点。

译码时按照与编码时采用的相同规则进行，还原后得到的数据与压缩前的数据完全相同。因此，RLE 是无损压缩技术。

RLE 编码尤其适用于计算机生成的图像，对减少图像文件的存储空间非常有效。然而，RLE 对颜色丰富的自然图像就显得力不从心，因为在同一行上具有相同颜色的连续像素往往很少，而连续几行都具有相同颜色值的连续行数就更少。如果仍然使用 RLE 编码方法，不仅不能压缩图像数据，反而可能使原来的图像数据变得更大。但这并不是说 RLE 编码方法不适用于自然图像的压缩，相反，在自然图像的压缩中还真少不了 RLE。在 JPEG 和 MPEG 等标准中，RLE 用来对图像数据经过变换和量化后的系数进行编码。

3.4 词典编码

词典编码是用词在词典中表示位置的号码代替词本身的无损数据压缩方法。词典编码利用数据本身包含重复代码的特性生成编码词典，然后用编码词典中表示该词所在位置的号码代替重复代码。采用静态词典编码技术时，编码器需要事先构造词典，解码器要事先知道词典。采用动态辞典编码技术时，编码器将从被压缩的文本中自动导出词典，解码器解码时一边解码一边构造解码词典。词典编码适用于编码数据的统计特性事先不知道或不可能知道的场合，如文本文件和电视图像就具有这种特性。

3.4.1 词典编码的思想

词典编码(dictionary encoding)的根据是数据本身包含有重复代码，如文本文件和光栅图像就具有这种特性。词典编码法的种类很多，归纳起来大致有两类。

第一类算法的想法是，企图查找正在压缩的字符序列是否在以前输入的数据中出现过，然后用已经出现过的字符串替代重复的部分，它的输出仅仅是指向早期出现过的字符串的“指针”。这种编码概念如图 3-6 所示。

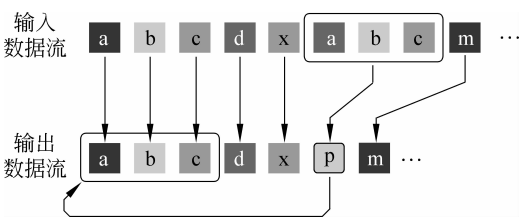

图 3-6 第一类词典编码概念

这里所指的“词典”是指用以前处理过的数据来表示编码过程中遇到的重复部分。这类编码中的所有算法都是以 Abraham Lempel 和 Jakob Ziv 在 1977 年开发和发表的 LZ77 算法为

基础的，例如 1982 年由 Storer 和 Szymanski 改进的称为 LZSS 算法就是属于这种情况。

第二类算法的想法是，企图从输入的数据中创建一个“短语词典(dictionary of the phrases)”，这种短语不一定是像“严谨勤奋求实创新”和“国泰民安是坐稳总统宝座的根本”这类具有具体含义的短语，它可以是任意字符的组合。在编码数据过程中，当遇到已经在词典中出现的“短语”时，编码器就输出这个词典中的短语的“索引号”，而不是短语本身。这个概念如图 3-7 所示。

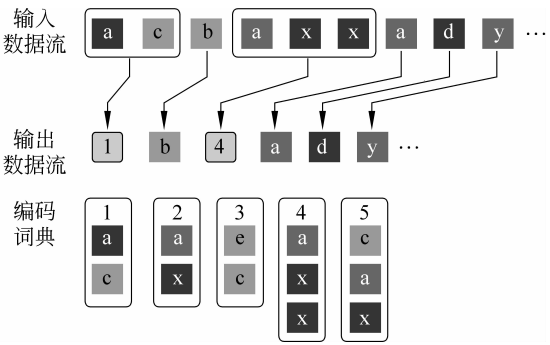

图 3-7　第二类词典编码概念

J. Ziv 和 A. Lempel 在 1978 年首次发表了介绍这种编码方法的文章。在他们的研究基础上，Terry A. Weltch 在 1984 年发表了改进这种编码算法的文章[3]，因此把这种编码方法称为 LZW(Lempel-Ziv Walch)压缩算法，并首先成功地应用在高速硬盘控制器上。

3.4.2　LZ77 算法

为了更好地说明 LZ77 算法的原理，首先介绍算法中用到的几个术语：

- 输入流(input stream)：要被压缩的字符序列。
- 字符(character)：输入流中的基本数据单元。
- 编码位置(coding position)：输入流中当前要编码的字符位置，指前向缓冲存储器中的开始字符。
- 前向缓存(Lookahead buffer)：存放从编码位置到输入流结束的字符序列。
- 窗口(window)：包含 W 个字符的窗口，字符是从编码位置开始往后数也就是最后处理的字符数。
- 指针(pointer)：指向窗口中的匹配串并包含长度的指针。

LZ77 编码算法的核心是查找从前向缓冲存储器开始的最长的匹配串。编码算法的具体执行步骤如下：

步骤 1：把编码位置设置到输入流的开始位置。

步骤 2：查找窗口中最长的匹配串。

步骤 3：以“(Pointer, Length) Characters”的格式输出。其中 Pointer 是指向窗口中匹配串的指针，Length 表示匹配字符的长度，Characters 是前向缓存中不匹配的第 1 个字符。

步骤 4：如果前向缓冲存储器不是空的，则把编码位置和窗口向前移(Length＋1)个字符，然后返回到步骤 2 。

【例 3.9】 待编码的输入流见表 3-9，LZ77 的编码过程见表 3-10。现作如下说明：

(1)“步骤”栏表示编码步骤,每个“步骤”都有输出。

(2)“位置”栏表示编码位置,输入流中的第 1 个字符为编码位置 1。

(3)“匹配串”栏表示窗口中找到的最长的匹配串。

(4)“字符”栏表示匹配之后在前向缓冲存储器中的第 1 个字符。

(5)“输出”栏以“(Back_chars, Chars_length) Explicit_character”格式输出。其中,(Back_chars, Chars_length)是指向匹配串的指针,告诉译码器“在这个窗口中向后退 Back_chars 个字符,然后拷贝 Chars_length 个字符到输出”,Explicit_character 是真实字符。例如,输出“(5, 2) C”告诉译码器回退 5 个字符,然后拷贝两个字符“AB”。

表 3-9 待编码的输入流

位置	1	2	3	4	5	6	7	8	9
字符	A	A	B	C	B	B	A	B	C

表 3-10 编码过程

步骤	位置	匹配串	字符	输出
1	1	--	A	(0,0) A
2	2	A	B	(1,1) B
3	4	--	C	(0,0) C
4	5	B	B	(2,1) B
5	7	A B	C	(5,2) C
…	…	…	…	…

3.4.3 LZSS 算法

LZ77 通过输出真实字符解决了在窗口中出现没有匹配串的问题,但这种解决方案包含有冗余信息。冗余信息表现在两个方面:一是编码器输出可能包含空指针;二是编码器可能输出额外字符,就是可能包含在下一个匹配串中的字符。LZSS 算法以比较有效的方法解决了这个问题,它的思想是如果匹配串的长度比指针本身的长度长就输出指针,否则就输出真实字符。由于输出数据流中包含有指针和字符本身,为了区分它们就需要有额外的标志位,即 ID 位。

LZSS 编码算法的具体执行步骤如下:

步骤 1:把编码位置置于输入流的开始位置。

步骤 2:在前向缓冲存储器中查找与窗口中最长的匹配串:

① Pointer:=匹配串指针。

② Length:=匹配串长度。

步骤 3:判断匹配串长度 Length 是否大于等于最小匹配串长度(Length≥MIN_LENGTH)。

是:输出指针,然后把编码位置向前移动 Length 个字符。

否:输出前向缓冲存储器中的第 1 个字符,然后把编码位置向前移 1 个字符。

步骤 4：如果前向缓冲存储器不是空的，就返回到步骤 2。

【例 3.10】 待编码字符串如表 3-11 所示，编码过程见表 3-12。现说明如下：

(1)“步骤”栏表示编码步骤。

(2)“位置”栏表示编码位置，输入流中的第 1 个字符为编码位置 1。

(3)“匹配”栏表示窗口中找到的最长的匹配串。

(4)“字符”栏表示匹配之后在前向缓冲存储器中的第 1 个字符。

(5)“输出”栏的输出为：

① 如果匹配串本身的长度 Length≥MIN_LENGTH，输出指向匹配串的指针，格式为(Back_chars，Chars_length)。该指针告诉译码器“在这个窗口中向后退 Back_chars 个字符然后拷贝 Chars_length 个字符到输出”。

② 如果匹配串本身的长度 Length ≤ MIN_LENGTH，则输出真实的匹配串。

表 3-11　输入流

位置	1	2	3	4	5	6	7	8	9	10	11
字符	A	A	B	B	C	B	B	A	A	B	C

表 3-12　编码过程(MIN_LENGTH＝2)

步骤	位置	匹配串	输出
1	1	--	A
2	2	A	A
3	3	--	B
4	4	B	B
5	5	--	C
6	6	B B	(3,2)
7	8	A A B	(7,3)
8	11	C	C

在相同的计算机环境下，LZSS 算法比 LZ77 可获得比较高的压缩比，而译码同样简单。这也就是为什么这种算法成为开发新算法的基础，许多后来开发的文档压缩程序都使用了 LZSS 的思想，如 PKZip、ARJ、LHArc 和 ZOO 等，其差别仅仅是指针的长短和窗口的大小有所不同。

LZSS 同样可以和熵编码联合使用，如 ARJ 就与霍夫曼编码联用，而 PKZip 则与 Shannon-Fano 联用，它的后续版本也采用霍夫曼编码。

3.4.4　LZ78 算法

在介绍 LZ78 算法之前，首先说明在算法中用到的几个术语和符号：

- 字符流(Charstream)：要被编码的数据序列。
- 字符(Character)：字符流中的基本数据单元。
- 前缀(Prefix)：在一个字符之前的字符序列。

- 缀-符串(String)：前缀＋字符。
- 码字(Code word)：码字流中的基本数据单元，代表词典中的一串字符。
- 码字流(Codestream)：码字和字符组成的序列，是编码器的输出。
- 词典(Dictionary)：缀-符串表。按词典中的索引号对每条缀-符串指定一个码字。
- 当前前缀(Current prefix)：在编码时用，指当前正在处理的前缀，用符号 P 表示。
- 当前字符(Current character)：在编码时用，指当前前缀后的字符，用符号 C 表示。
- 当前码字(Current code word)：在译码时用，指当前处理的码字，用 W 表示当前码字，String. W 表示当前码字的缀-符串。

1. 编码算法

LZ78 的编码思想是不断地从字符流中提取新的缀-符串，通俗地理解为新“词条”，然后用“代号”也就是码字表示这个“词条”。这样一来，对字符流的编码就变成了用码字去替换字符流，生成码字流，从而达到压缩数据的目的。

在编码开始时词典是空的，不包含任何缀-符串。在这种情况下编码器就输出一个表示空字符串的特殊码字(例如“0”)和字符流中的第一个字符 C，并把这个字符 C 添加到词典中作为由一个字符组成的缀-符串。在编码过程中，如果出现类似的情况，也照此办理。在词典中已经包含某些缀-符串的情况下，如果“当前前缀 P＋当前字符 C”已经在词典中，就用字符 C 来扩展这个前缀，这样的扩展操作一直重复到获得一个在词典中没有的缀-符串为止。此时就输出表示当前前缀 P 的码字和字符 C，并把 P＋C 添加到词典中作为前缀，然后开始处理字符流中的下一个前缀。

LZ78 编码器的输出是码字-字符(W，C)对，每次输出一对到码字流中，与码字 W 相对应的缀-符串用字符 C 进行扩展生成新的缀-符串，然后添加到词典中。LZ78 编码的具体算法如下：

步骤 1：在开始时，词典和当前前缀 P 都是空的。

步骤 2：当前字符 C：＝字符流中的下一个字符。

步骤 3：判断 P＋C 是否在词典中：

(1) 如果“是”：用 C 扩展 P，让 P：＝P＋C；

(2) 如果“否”：

① 输出与当前前缀 P 相对应的码字和当前字符 C。

② 把字符串 P＋C 添加到词典中。

③ 令 P：＝空值。

(3) 判断字符流中是否还有字符需要编码：

① 如果“是”：返回到步骤 2。

② 如果“否”：

- 若 P 不是空的，输出相应于当前前缀 P 的码字。
- 结束编码。

2. 译码算法

在译码开始时译码词典是空的，它将在译码过程中从码字流中重构。每当从码字流中读入一对码字-字符(W，C)对时，码字就参考已经在词典中的缀-符串，然后把当前码字的缀-符串 string. W 和字符 C 输出到字符流，而把(string. W＋C)添加到词典中。在译码结束之后，

重构的词典与编码时生成的词典完全相同。LZ78 译码的具体算法如下：

步骤 1：在开始时词典是空的。

步骤 2：当前码字 W：＝码字流中的下一个码字。

步骤 3：当前字符 C:＝紧随码字之后的字符。

步骤 4：把当前码字的缀-符串(string. W)输出到字符流(Charstream)后输出字符 C。

步骤 5：把 string. W＋C 添加到词典中。

步骤 6：判断码字流中是否还有码字要译。

(1) 如果“是”,就返回到步骤 2。

(2) 如果“否”,则结束。

【例 3.11】 待编码字符串如表 3-13 所示,编码过程见表 3-14。现说明如下：

(1)“步骤”栏表示编码步骤。

(2)“位置”栏表示在输入数据中的当前位置。

(3)“词典”栏表示添加到词典中的缀-符串,缀-符串的索引等于“步骤”序号。

(4)“输出”栏以(当前码字 W，当前字符 C)简化为(W，C)的形式输出。

表 3-13　编码字符串

位置	1	2	3	4	5	6	7	8	9
字符	A	B	B	C	B	C	A	B	A

表 3-14　编码过程

步骤	位置	词典	输出
1	1	A	(0,A)
2	2	B	(0,B)
3	3	B C	(2,C)
4	5	B C A	(3,A)
5	8	B A	(2,A)

与 LZ77 相比,LZ78 的最大优点是在每个编码步骤中减少了缀-符串比较的次数,而压缩率基本相同。

3.4.5　LZW 算法

在 LZW 算法中使用的术语与 LZ78 使用的相同,仅增加了一个术语：前缀根(Root),它是由单个字符串组成的缀-符串。

在编码原理上,LZW 与 LZ78 相比有如下差别：

(1) LZW 只输出代表词典中的缀-符串的码字。这就意味在开始时词典不能是空的,它必须包含可能在字符流中出现的所有单个字符,即前缀根。

(2) 由于所有可能出现的单个字符都事先包含在词典中,每个编码步骤开始时都使用一字符前缀(one-character prefix),因此在词典中搜索的第 1 个缀-符串有两个字符。

(3) 新前缀开始的字符是先前缀-符串(C)的最后一个字符,这样在重构词典时就不需要

在码字流中加入额外的字符。

1. 编码算法

LZW 编码[3~5]是围绕称为词典的转换表来完成的。这张转换表用来存放称为前缀(Prefix)的字符序列,并且为每个表项分配一个码字(Code word),或者叫作序号,如表 3-15 所示。这张转换表实际上是把 8 位 ASCII 字符集进行扩充,增加的符号用来表示在文本或图像中出现的可变长度 ASCII 字符串。扩充后的代码可用 9 位、10 位、11 位、12 位甚至更多的位来表示。Welch 的论文中用了 12 位,12 位可以有 4096 个不同的 12 位代码,这就是说,转换表有 4096 个表项,其中 256 个表项用来存放已定义的字符,剩下 3840 个表项用来存放前缀(Prefix)。

表 3-15 LZW 词典

码字(Code word)	前缀(Prefix)	码字(Code word)	前缀(Prefix)
1		255	
…	…	…	…
193	A	1305	abcdefxyF01234
194	B	…	…
…	…		

LZW 编码器(软件编码器或硬件编码器)就是通过管理这个词典完成输入与输出之间的转换。LZW 编码器的输入是字符流(Charstream),字符流可以是用 8 位 ASCII 字符组成的字符串,而输出是用 n 位(例如 $n=12$ 位)表示的码字流(Codestream),码字代表单个字符或多个字符组成的字符串。

LZW 编码器使用了一种很实用的分析(parsing)算法,称为贪婪分析算法(greedy parsing algorithm)。在贪婪分析算法中,每一次分析都要串行地检查来自字符流的字符串,从中分解出已经识别的最长的字符串,也就是已经在词典中出现的最长的前缀。用已知的前缀加上下一个输入字符 C,也就是当前字符(Current character),作为该前缀的扩展字符,形成新的扩展字符串——缀-符串:Prefix.C。这个新的缀-符串是否要加到词典中,还要看词典中是否存有和它相同的缀-符串。如果有,那么这个缀-符串就变成前缀(Prefix),继续输入新的字符,否则就把这个缀-符串写到词典中生成一个新的前缀,并给一个代码。

LZW 编码算法的具体执行步骤如下:

步骤 1:开始时的词典包含所有可能的根(Root),而当前前缀 P 是空的。

步骤 2:当前字符(C):=字符流中的下一个字符。

步骤 3:判断缀-符串 P+C 是否在词典中:

(1) 如果"是",P:=P+C// (用 C 扩展 P)。

(2) 如果"否",则:

① 把代表当前前缀 P 的码字输出到码字流。

② 把缀-符串 P+C 添加到词典。

③ 令 P:=C //(现在的 P 仅包含一个字符 C)。

步骤 4:判断码字流中是否还有码字要译:

(1) 如果"是",就返回到步骤 2。

(2) 如果“否”,则:

① 把代表当前前缀 P 的码字输出到码字流。

② 结束。

LZW 编码算法可用伪码表示。开始时假设编码词典包含若干个已经定义的单个码字。例如,256 个字符的码字,用伪码可以表示成:

```
------------------------------------------------
Dictionary[j] ← all n single-character, j=1, 2, …,n
j ← n+ 1
Prefix ← read first Character in Charstream
while((C ← next Character)!=NULL)
    Begin
        If Prefix.C is in Dictionary
            Prefix ← Prefix.C
        else
            Codestream ← cW for Prefix
            Dictionary[j] ← Prefix.C
            j ← n+ 1
            Prefix ← C
        end
Codestream ← cW for Prefix
------------------------------------------------
```

2. 译码算法

LZW 译码算法中还用到另外两个术语:(1)当前码字(Current code word):指当前正在处理的码字,用 cW 表示,用 string. cW 表示当前缀-符串;(2)先前码字(Previous code word):指先于当前码字的码字,用 pW 表示,用 string. pW 表示先前缀-符串。

LZW 译码算法开始时,译码词典与编码词典相同,它包含所有可能的前缀根(roots)。LZW 算法在译码过程中会记住先前码字(pW),从码字流中读当前码字(cW)之后输出当前缀-符串 string. cW,然后把用 string. cW 的第一个字符扩展的先前缀-符串 string. pW 添加到词典中。

LZW 译码算法的具体执行步骤如下:

步骤 1:在开始译码时词典包含所有可能的前缀根(Root)。

步骤 2:当前码字 cW:=码字流中的第一个码字。

步骤 3:输出当前缀-符串 string. cW 到字符流。

步骤 4:先前码字 pW:=当前码字 cW。

步骤 5:当前码字 cW:=码字流中的下一个码字。

步骤 6:判断当前缀-符串 string. cW 是否在词典中:

(1) 如果“是”,则:

① 当前缀-符串 string. cW 输出到字符流。

② 当前前缀 P:=先前缀-符串 string. pW。

③ 当前字符 C：＝当前前缀-符串 string.cW 的第一个字符。

④ 把缀-符串 P＋C 添加到词典。

(2) 如果“否”,则：

① 当前前缀 P：＝先前缀-符串 string.pW。

② 当前字符 C：＝当前缀-符串 string.pW 的第一个字符。

③ 输出缀-符串 P＋C 到字符流,然后把它添加到词典中。

步骤 7：判断码字流中是否还有码字要译：

(1) 如果“是”,就返回到步骤 4。

(2) 如果“否”，结束。

LZW 译码算法可用伪码表示如下：

```
-----------------------------------------------------------------
Dictionary[j] ← all n single-character, j=1, 2, …,n
j ← n+ 1
cW ← first code from Codestream
Charstream ← Dictionary[cW]
pW ← cW
While((cW ← next Code word)!=NULL)
    Begin
        If cW is in Dictionary
        Charstream ← Dictionary[cW]
        Prefix ← Dictionary[pW]
        cW ← first Character of Dictionary[cW]
        Dictionary[j] ← Prefix.cW
        j ← n+ 1
        pW ← cW
    else
        Prefix ← Dictionary[pW]
        cW ← first Character of Prefix
        Charstream ← Prefix.cW
        Dictionary[j] ← Prefix.C
        pW ← cW
        j ← n+ 1
    end
end
-----------------------------------------------------------------
```

【例 3.12】 待编码字符串如表 3-16 所示,编码过程见表 3-17。现说明如下：

(1) “步骤”栏表示编码步骤。

(2) “位置”栏表示在输入数据中的当前位置。

(3) “词典”栏表示添加到词典中的缀-符串,它的索引在括号中。

(4) “输出”栏表示码字输出。

表 3-18 解释了译码过程。每个译码步骤译码器读一个码字,输出相应的缀-符串,并把它添加到词典中。例如,在步骤 4 中,先前码字(2)存储在先前码字(pW)中,当前码字(cW)是

(4),当前缀-符串 string. cW 是输出("A B"),先前缀-符串 string. pW ("B")是用当前缀-符串 string. cW ("A")的第一个字符,其结果("B A") 添加到词典中,它的索引号是(6)。

表 3-16　待编码的字符串

位置	1	2	3	4	5	6	7	8	9
字符	A	B	B	A	B	A	B	A	C

表 3-17　LZW 的编码过程

步　骤	位　置	词　典		输　出
		(1)	A	
		(2)	B	
		(3)	C	
1	1	(4)	A B	(1)
2	2	(5)	B B	(2)
3	3	(6)	B A	(2)
4	4	(7)	A B A	(4)
5	6	(8)	A B A C	(7)
6	--	--	--	(3)

表 3-18　LZW 的译码过程

步　骤	输入代码	词　典		输　出
		(1)	A	
		(2)	B	
		(3)	C	
1	(1)	--	--	A
2	(2)	(4)	A B	B
3	(2)	(5)	B B	B
4	(4)	(6)	B A	A B
5	(7)	(7)	A B A	A B A
6	(3)	(8)	A B A C	C

LZW 算法得到普遍采用,它的速度比使用 LZ77 算法的速度快,因为它不需要执行那么多的缀-符串的比较操作。对 LZW 算法进一步的改进是增加可变的码字长度,以及在词典中删除老的缀-符串。

LZW 算法取得了专利,专利权的所有者是美国的一个大型计算机公司——Unisys(优利系统公司),除了商业软件生产公司之外,可以免费使用 LZW 算法。

练习与思考题

3.1　熵(entropy)是什么?

3.2　熵编码(entropy coding)是什么?

3.3　假设{a,b,c}是由3个事件组成的集合,计算该集合的决策量。(分别用Sh、Nat和Hart作单位。)

3.4　现有一幅用256级灰度表示的图像,如果每级灰度出现的概率均为$p(x_i)=1/256$,$i=0,\cdots,255$,计算这幅图像数据的熵。

3.5　现有8个待编码的符号$m_0,\cdots,m_7$,它们的概率如练习表3-1所示,计算这些符号的霍夫曼码并填入表中。

练习表 3-1

待编码符号	概率	分配的代码	代码长度(位)
m_0	0.40		
m_1	0.20		
m_2	0.15		
m_3	0.10		
m_4	0.07		
m_5	0.04		
m_6	0.03		
m_7	0.01		

3.6　现有5个待编码的符号,它们的概率见练习表3-2。计算该符号集的:(1)熵;(2)霍夫曼码;(3)平均码长。

练习表 3-2

符号	a_1	a_2	a_3	a_4	a_5
概率	0.2	0.4	0.2	0.1	0.1

3.7　使用算术编码生成字符串games的代码。字符g、a、m、e、s的概率见练习表3-3。

练习表 3-3

符号	g	a	m	e	s
概率	0.4	0.2	0.2	0.1	0.1

3.8　字符流的输入如练习表3-4所示,使用LZW算法计算输出的码字流。如果对本章介绍的LZW算法不打算改进,请核对计算的输出码字流为:

(1)(2)(4)(3)(5)(8)(1)(10)(11)…

练习表 3-4

输入位置	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	…
输入字符流	a	b	a	b	c	b	a	b	a	b	a	a	a	a	a	a	a	…
输出码字																		

3.9 LZ78 算法和 LZ77 算法的差别在哪里？

3.10 LZSS 算法和 LZ77 算法的核心思想是什么？它们之间有什么差别？

3.11 LZW 算法和 LZ78 算法的核心思想是什么？它们之间有什么差别？

3.12 你是否同意“某个事件的信息量就是某个事件的熵”的看法。

参考文献和站点

[1] David Salomon. Data Compression. Third Edition. Springer-Verlag, 2004.

[2] Timothy C. Bell, John G. Cleary, Ian H. Witten. Text Compression. Prentice-Hall, Inc. ,1990.

[3] Ziv J. , Lempel A.. A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory, May 1977.

[4] Terry A. Welch. A Technique for High-Performance Data Compression. Computer, June 1984.

[5] Nelson, M. R. LZW Data Compression. Dr. Dobb's Journal, October 1989. http://marknelson. us/1989/10/01/lzw-data-compression/.

[6] R. Hunter, A. H. Robison. International Digital Facsimile Coding Standards. Proceedings of the IEEE, Vol. 68, No. 7, July, 1980, 854～867.