四种压缩算法原理介绍

四种压缩算法：RLE、霍夫曼、Rice、Lempel-Ziv (LZ77)

RLE

RLE 又叫 Run Length Encoding，是一个针对无损压缩的非常简单的算法。它用重复字节和重复的次数来简单描述来代替重复的字节。尽管简单并且对于通常的压缩非常低效，但它有的时候却非常有用（例如，JPEG 就使用它）。

原理

下图显示了一个如何使用 RLE 算法来对一个数据流编码的例子，其中出现六次的符号'93'已经用 3 个字节来代替：一个标记字节（'0'在本例中）重复的次数（'6'）和符号本身（'93'）。

RLE 解码器遇到符号'0'的时候，它表明后面的两个字节决定了需要输出哪个符号以及输出多少次。

实现

RLE 可以使用很多不同的方法。基本压缩库中详细实现的方式是非常有效的一个。一个特殊的标记字节用来指示重复节的开始，而不是对于重复非重复节都 coding run。

因此非重复节可以有任意长度而不被控制字节打断，除非指定的标记字节出现在非重复节（顶多以两个字节来编码）的稀有情况下。为了最优化效率，标记字节应该是输入流中最少出现的符号（或许就不存在）。

重复 runs 能够在 32768 字节的时候运转。少于 129 字节的要求 3 个字节编码（标记+次数+符号），而大雨 128 字节要求四个字节（标记+次数的高 4 位|0×80+次数的低 4 位）。这是通常所有采用的压缩的做法，并且也是相比较三个字节固定编码（允许使用 3 个字节来编码 256 个字节）而言非常少见的有损压缩率的方法。

在这种模式下，最坏的压缩结果是：

输出大小=257/256*输入大小+1

霍夫曼

霍夫曼编码是无损压缩当中最好的方法。它使用预先二进制描述来替换每个符号，长度由特殊符号出现的频率决定。常见的符号需要很少的位来表示，而不常见的符号需要很多为来表示。

霍夫曼算法在改变任何符号二进制编码引起少量密集表现方面是最佳的。然而，它并不处理符号的顺序和重复或序号的序列。

原理

简短的说，这个问题的解决方案是为了查找每个符号的通用程度，我们建立一个未压缩数据的柱状图；通过递归拆分这个柱状图为两部分来创建一个二叉树，每个递归的一半应该和另一半具有同样的权（权是∑NK =1 符号数 k, N 是分之中符号的数量，符号数 k 是符号 k 出现的次数）

这棵树有两个目的：

1. 编码器使用这棵树来找到每个符号最优的表示方法
2. 解码器使用这棵树唯一的标识在压缩流中每个编码的开始和结束，其通过在读压缩数据位的时候自顶向底的遍历树，选择基于数据流中的每个独立位的分支，一旦一个到达叶子节点，解码器知道一个完整的编码已经读出来了。

实现

霍夫曼编码器可以在基本压缩库中找到，其是非常直接的实现。

这个实现的基本缺陷是：

1. 慢位流实现
2. 相当慢的解码（比编码慢）

3．最大的树深度是 32（编码器在任何超过 32 位大小的时候退出）。如果我不是搞错的话，这是不可能的，除非输出的数据大于 232 字节。

另一方面，这个实现有几个优点：

1. 霍夫曼树以一个紧密的形式每个符号要求 12 位（对于 8 位的符号）的方式存储，这意味着最大的头为 384。
2. 编码相当容易理解

霍夫曼编码在数据有噪音的情况（不是有规律的，例如 RLE）下非常好，这中情况下大多数基于字典方式的编码器都有问题。

Rice

对于由大 word（例如：16 或 32 位）组成的数据和教低的数据值，Rice 编码能够获得较好的压缩比。音频和高动态变化的图像都是这种类型的数据，它们被某种预言预处理过（例如 delta 相邻的采样）。

尽管哈夫曼编码处理这种数据是最优的，却由于几个原因而不适合处理这种数据（例如：32 位大小要求 16GB 的柱状图缓冲区来进行哈夫曼树编码）。因此一个比较动态的方式更适合由大 word 组成的数据。

原理

Rice 编码背后的基本思想是尽可能的用较少的位来存储多个字（正像使用霍夫曼编码一样）。实际上，有人可能想到 Rice 是静态的霍夫曼编码（例如，编码不是由实际数据内容的统计信息决定，而是由小的值比高的值常见的假定决定）。

编码非常简单：将值 X 用 X 个‘1’位之后跟一个 0 位来表示。

实现

在基本压缩库针对 Rice 做了许多优化：

1．每个字最没有意义的位被存储为 k 和最有意义的 N-k 位用 Rice 编码。K 作为先前流中少许采样的位平均数。这是通常最好使用 Rice 编码的方法，隐藏噪音且对于动态变化的范围并不导致非常长的 Rice 编码。

2．如果 rice 编码比固定的开端长，T，一个可选的编码：输出 T 个‘1’位，紧跟（log2(X-T)）个‘1’和一个‘0’位，接着是 X-T（最没有意义的(log2(X-T))-1 位）。这对于大值来说都是比较高效的代码并且阻止可笑的长 Rice 编码（最坏的情况，对于一个 32 位 word 单个 Rice 编码可能变成 232 位或 512MB）。

3. 最坏的情况，输出。

Lempel-Ziv (LZ77)

Lempel-Ziv 压缩模式有许多不同的变量。基本压缩库有清晰的 LZ77 算法的实现（Lempel-Ziv，1977），执行的很好，源代码也非常容易理解。LZ 编码器能用来通用目标的压缩，特别对于文本执行的很好。它也在 RLE 和哈夫曼编码器（RLE，LZ，哈夫曼）中使用来大多数情况下获得更多的压缩。

原理

在 LZ 压缩算法的背后是使用 RLE 算法用先前出现的相同字节序列的引用来替代。

简单的讲，LZ 算法被认为是字符串匹配的算法。例如：在一段文本中某字符串经常出现，并且可以通过前面文本中出现的字符串指针来表示。当然这个想法的前提是指针应该比字符串本身要短。

例如，在上一段短语“字符串”经常出现，可以将除第一个字符串之外的所有用第一个字符串引用来表示从而节省一些空间。

一个字符串引用通过下面的方式来表示：

1. 唯一的标记
2. 偏移数量
3. 字符串长度

由编码的模式决定引用是一个固定的或变动的长度。后面的情况经常是首选，因为它允许编码器用引用的大小来交换字符串的大小（例如，如果字符串相当长，增加引用的长度可能是值得的）。

实现

使用 LZ77 的一个问题是由于算法需要字符串匹配，对于每个输入流的单个字节，每个流中此字节前面的哪个字节都必须被作为字符串的开始从而尽可能的进行字符串匹配，这意味着算法非常慢。

另一个问题是为了最优化压缩而调整字符串引用的表示形式并不容易。例如，必须决定是否所有的引用和非压缩字节应该在压缩流中的字节边界发生。

基本压缩库使用一个清晰的实现来保证所有的符号和引用是字节对齐的，因此牺牲了压缩比率，并且字符串匹配程序并不是最优化的（没有缓存、历史缓冲区或提高速度的小技巧），这意味着程序非常慢。

另一方面，解压缩程序非常简单。

一个提高 LZ77 速度的试验已经进行了，这个试验中使用数组索引来加速字符串匹配的过程。然而，它还是比通常的压缩程序慢。