第33卷 第5期 计算机研究与发展 Vol. 33 No. 5
1996年 5月 COMPUTER RESEARCH & DEVELOPMENT May 1996

329-340

关于 Lempel-Ziv 77 压缩算法及其实现的研究

王忠效 姜 丹
(中国科学技术大学管理学院 合肥 230026)
(中国科学院管理干部学院 北京 101408)

TP301.6

摘 要 本文在研究著名的LZ77压缩算法的基础上，讨论了对这一算法的种种改进。新的算法同样适用于任何类型的数据文件，而且无论是压缩速度还是压缩效率均好于LZ77算法。我们的算法所用到的工作缓冲区是一个循环缓冲区，不再包括一个输入符号超前缓冲区；结果，匹配过程是边接收输入边进行，无需等待一组输入数据填满超前缓冲区才开始，同时，最大匹配长度也不再受超前缓冲区大小的限制，而且，避免了大量的平移工作缓冲区的操作。另外，还涉及一些其他方面的改进，主要包括改等长压缩码为变长码和引入匹配位置滑动表技术等。本文详细讨论了各种改进及其对算法性能的影响。

关键词 数据压缩，LZ77压缩算法，LZ78压缩算法，循环缓冲区，匹配位置滑动表。

RESEARCH ON LEMPEL-ZIV 77 ALGORITHM AND ITS IMPLEMENTATION

Wang Zhongxiao and Jiang Dan
School of Management, University of Science & Technology of China, Hefei 230026
Beijing Institute of Management, Chinese Academy of Sciences, Beijing 101408

Abstract Based on the analysis of LZ77 algorithm, this paper discusses its major improvements. A new algorithm adopts a circular history list, leaving the look-ahead buffer of LZ77 discarded. That makes it possible to let a matched string be as long as the size of the history list and results in the reduction of code length. Another main improvement is that the new algorithm takes variable-length code instead of LZ77's fixed-length one. Besides, a new data structure called matching position list is introduced, which contributes much to the high efficiency of the encoder. How and why the improvements affect the algorithm's performance are shown in detail.

文稿收到日期：1994-10-20。王忠效，1963年生，副教授，1983年于江西大学毕业，获理学学士学位，1986年于华中理工大学获工学硕士学位，现主要从事数据压缩、汉语文本压缩的算法研究及软件开发工作。姜 丹，1941年生，教授，1964年毕业于中国科学技术大学，现从事信息科学、管理科学的教学与研究工作。

Keywords Data compression, LZ77 algorithm, LZ78 algorithm, circular buffer, matching position list.

1 Lempel-Ziv 压缩算法的基本原理

Lempel-Ziv 压缩算法是由两位以色列信息论专家 Jacob Ziv 和 Abraham Lempel 在 1977 年提出的。可以讲，它是基于字符串匹配(或词典编码)的第一个具有实用价值的数据压缩算法。今天，该算法在数据压缩领域已经占居主导地位，各种流行的压缩软硬件均以之为基础，有关压缩理论与技术的书籍也都不能不开辟专门的章节予以介绍。

Lempel-Ziv 算法建立在下述认识的基础上：待编码的数据符号串可能包含在已经编码的信息结构中，因而整个数据源在待编码的符号串上呈现出冗余，也就是讲，根据已编码的数据我们完全可以构造出后续与之相同的数据。这种思想的确很朴素，而且无论是压缩效率还是速度都相当出色。然而，直到 80 年代初，研究人员和工程技术人员非常看好著名的 Huffman 编码，大都将兴趣放到如何实现和改进 Huffman 编码上。Lempel-Ziv 压缩算法的诞生是数据压缩领域的重大转折，尽管近 20 年来信息论刊物上仍不时发表有关 Huffman 编码的论文，但关于 Lempel-Ziv 算法的文献显然丰富得多，而且大家所知道的，目前引起围绕压缩技术专利之争的算法正是 Lempel-Ziv 算法及其衍生物。

Lempel-Ziv 压缩算法的实现主要分 LZ77 和 LZ78 两类。本质上讲，二者是共同的，只是因为对于待编码的信息究竟取决于最近有限的上下文还是依赖于整个已被编码的上下文取不同的认识，所采用的压缩码和编码技术才不同。从理论上讲，上下文环境越大，待编码的信息结构在上下文中呈现冗余的概率越大。因此，依赖于整个已编码上下文的 LZ78 比只依赖于最近有限上下文的 LZ77 从压缩比率上讲，一般肯定要好。然而，无论在自然语言文本、程序设计语言源程序、图象视频数据，还是在二进制可执行的机器代码中，近邻原则始终悄无声息地发生作用。所谓近邻原则，是指一个现象对它的最近的上下文的依赖或制约的程度，要高于对它的较远的上下文的依赖或制约的程度。这也可以翻译为：一个现象在它的最近的上下文找到相同结构的可能性，要大于在它的较远的上下文中找到相同结构的可能性。譬如，在计算机高级语言源程序中，局部量仅在其很有限的作用域内存在，如果该局部量的标识符名在整个程序中是独一无二的，显然，该标识符的第二次、第三次……出现只可能在所处的最近有限的上下文中重复，而第一次出现找遍全部上文也不存在重复。因此，对于 Lempel-Ziv 压缩算法而言，取无限大的上下文还是取最近适当大小的上下文，压缩效率一般没有太明显的差异。然而，从技术实现上讲，依赖于有限上下文的算法效率要比依赖于无限上下文高得多。我们认为，正是基于这种认识，尽管 LZ77 比 LZ78 更早问世，其价值依然不减，仍旧受到工程技术人员的亲睐，譬如大家非常熟悉的压缩工具 LHArc 即以之为基础。本文仅局限讨论 LZ77 压缩算法(简称 LZ77)。

2 LZ77 压缩算法及其压缩码结构

根据上述思想，需要解决的问题变成在已编码的历史数据(保存于历史表)中寻求与待编码的数据相同的符号串，然后，在压缩码中仅存储在历史表中获得成功匹配的数据段的起点位置及其长度。对于存在大量重复内容的数据，由于重复部分的编码比较原始数据自身的码长要短，因此，可望获得较好的压缩效果。比如，由于英语文本中频繁出现□the□、□an□、□that□、□which

□、ation□、□not□(为可视起见，□代表空格字符)等符号串，而汉语中不存在这类或其他类似的高频出现的语言现象，如果单从字符串匹配的角度看，英语文本的可压缩性必然要比汉语大(参见表 2)。

那么，历史表是怎样的数据结构呢？这里，我们仅讨论通常以字节为单位进行的数据压缩。LZ77 将其工作缓冲区 buffer[0..n-1]划分为两部分：历史表 buffer[0..p-1]和输入超前缓冲区 buffer[p..n-1]，如图 1 所示。

建立在这个数据结构的基础上，编码过程概述如下：

(1) 初始化历史表，并从输入中读取 q 个字节的数据装入超前缓冲区中；

(2) 在历史表中寻找与超前缓冲区中字符串的最大匹配(设 addr 记录匹配串在历史表中的起始位置，len 表示匹配串的长度)；

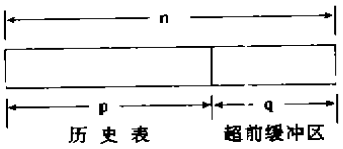

图 1　LZ77 工作缓冲区

(3) 按照某种约定将 addr、len 及匹配串的最后一个字符三者一起形成一个压缩代码送入输出位流中；

(4) 将 buffer 中的数据左移 len 个元素，并从输入中截取 len 个字节的数据填补到超前缓冲区的尾部；

(5) 重复 2 直至无输入数据。

译码是编码的逆过程，可以想象，此时可以不使用超前缓冲区。译码主要是遵循编码时建立的约定从压缩码中正确解释上面的 addr 和 len 等，然后，从历史表中 addr 处开始顺序向输出数据队列中拷贝 len 个字符。显然，历史表必须与编码过程同步更新，也就是讲，需要频繁左移历史表并且还需要将送入输出队列中的那段数据拷贝一份到历史表的尾部。

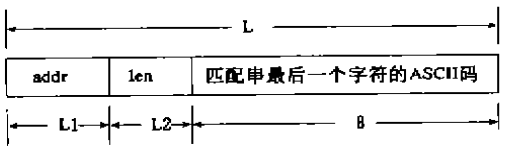

图 2　LZ77 压缩码结构

再看看 LZ77 的压缩码结构。LZ77 采用等长一致的压缩码描述匹配串以及不包含在匹配串中的孤立的数据。由上述算法的第 3 步可知，其压缩码结构如图 2 所示。

可见，压缩码位宽 $L=L1+L2+8$。譬如，如果 $p=4096$，$q=16$，则需要 $L1=12$ 个二进制位描述 addr，需要 $L2=4$ 个二进制位表示 len，此时，$L=24$ 比特。

3　LZ77 压缩算法的缺点

实验表明：当 $p=2$K 或 $p=4$K 字节、$q=16$ 或更多个字节时，压缩效果通常较好。一般地讲，最大匹配的概率随 p 的增长而增长，因此，p 越大，压缩效果越好。然而，对于大的工作缓冲区来讲，频繁地左移操作势必增加大量额外的时间开销。在最坏的情况下，匹配不成功(即 len=0)，为匹配进行的比较次数最多为 p，而对 buffer 必须进行 $n-1$ 个元素的左移操作。设每个比较和左移操作均在单位时间内完成，此时，左移操作甚至比匹配和编码本身更费时。而在最好的情况下，超前缓冲区的字符串得到完全匹配，所需的左移操作最少，仍还要求进行 $n-q=p$ 次。另一方面，最大匹配长度受超前缓冲区大小的限制，而实际的 $q \ll p$，此时，当待编码数据序列在历史表中存在长度远大于 q 的匹配时，必须将待编码数据分割成多个子段分别按图 2 所示的结构码独立编码，显然既影响编码质量又影响编码速度。其次，每次左移之后需要等待从输入数据中截取 *len* 个

字节填补超前缓冲区，而不是从输入数据流中截取一个字节即马上投入寻找匹配和编码这一中心工作。当压缩算法应用于数据通信时，LZ77 对时间的分配显然不很合理，尤其当超前缓冲区较大而信道中通过的信号量不均匀时，编码器只能是或者长时间空闲或者非常忙碌。

另外，LZ77 采用等长码，其突出的缺点在于：为了编码输入流中不包含在匹配串中的孤立数据项，不惜在每个压缩码中包括 8 位 ASCII 码。这对于孤立数据项而言，其"压缩码"比原码扩展了 $L1-L2$ 位；对于匹配串来讲，加上串的最后一个字符的 ASCII 码纯属人为造成的冗余——因为由 addr 和 len 两项元素足可以唯一描述被编码的数据对象。可见，LZ77 压缩码自身包含了不小的冗余。

那么，怎样减除压缩码自身的冗余？有没有更合理的数据结构及与之相适应的算法能够减少维护工作缓冲区的操作、同时又能克服其他 LZ77 的不足呢？

4 对 LZ77 算法的改进

4.1 循环缓冲区

用滑动表来命名 LZ77 算法的超前缓冲区的确再形象不过。为了求得最大匹配，必须在历史表上进行一系列模式匹配。每一匹配过程总是要首先滑向超前缓冲区的第一个元素位置，并从此处开始与历史表上的元素进行比较。由此可见，超前缓冲区的作用仅在于保存一个输入数据序列，以免输入数据"稍纵即逝"，从而无法寻求最大匹配。

其实，根本没有必要设置专门的超前缓冲区。它的角色是多余的，完全可以由历史表自身来承担。一次成功匹配之后，整个待编码的输入数据序列已经被包含在历史表中了。图 3 能很好地解释历史表的这一工作原理，即使图中两个匹配段重叠，历史表仍能正确地工作。

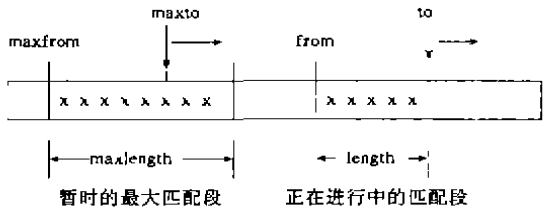

图 3 历史表兼作超前缓冲区

设在 maxfrom 处获得长度为 maxlength 的暂时的最大匹配。继续在历史表的下一个位置 from 处寻求与从 maxfrom 处开始、长度为 maxlength 的历史段匹配。如果成功匹配的字符数 length 达到 maxlength，则从输入数据流中截取下一个字符继续往后匹配，最大匹配很可能就发生在历史表的 from 处……。这样解决问题的意外好处是：自然地解决了 LZ77 的两点不足，即能够边接收输入边寻找最大匹配，而无需等待从信道中传来足够的信号；同时，最大匹配长度不受超前缓冲区大小的限制，最大匹配长度甚至可以等于历史表的尺寸。

既然超前缓冲区的作用完全可为历史表兼担，工作缓冲区就可以仅仅由一张历史表构成。那么，又如何克服 LZ77 的另一主要缺点——频繁地平移尺寸较大的工作缓冲区呢？

我们将工作缓冲区设计为循环队列，因此，表的任一位置均可以成为历史表的起止位置。这样一来，向历史表增加一个元素，仅仅只需要移动起止位置指针并将新的元素填在原来的起始位置（见图 4）。可见，更新历史表所需进行的操作的总量与历史表的长度无关，而不像 LZ77 一样与之成正比。

为了加快在历史表上寻求最大匹配的速度，参照 Knuth-Morris-Pratt(KMP)模式匹配算法，我们引入了另一个与历史表平行的匹配位置滑动表 nextpos，它与 history 元素成简单的 1-1 对应关系：nextpos[i]=字符 history[i]在历史表中下一次出现的位置。而为了方便地使用和更新匹配

位置滑动表，又引入了另外两个表，即 first[0..255]和 last[0..255]。其中 first[ch]记录字符 ch 在历史表中首次出现的位置，last[ch]则存放字符 ch 在历史表中最后一次出现的位置。如果等待编码的第一个符号为 ch，则匹配过程总是从历史表上 first[ch]处开始，然后继续在 nextpos[first[ch]]、nextpos[nextpos[first[ch]]]……等历史表上的位置寻求更长的匹配，最终求得最长匹配串在历史表中的偏移地址及其长度。

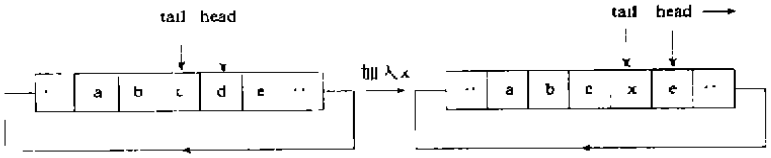

图 4　新的工作缓冲区 -·- 循环历史表 history

4.2　压缩码结构

为了便于讨论，下文姑且称改进了的算法为新算法。新算法的压缩码被定义为动态的变长码，其结构如图 5。

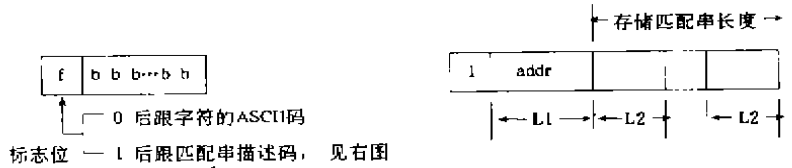

图 5　压缩码及匹配串描述码结构

当历史表长 4K 时，$L1=12$，即需要用 12 个二进制位存储匹配串首字符在历史表中的偏移地址 addr。因为按照我们的算法匹配串长度的变化区间较大，最大甚至达到历史表的尺寸，显然不能按其最大值设定存储匹配串长度信息的位宽。实际上，在压缩码中存放的串长值 x＝实际匹配串长度－最小匹配长度。x 被分段存储，方法如下：用 $L2$ 个连续的二进制位组成一段，第一段存放 x 与 y（$L2$ 个二进制位所能表示的最大无符号数）二者中的最小者，并令 x 等于原 $x-y$；如果 $x\geqslant 0$，即第一段的值为最大值 y，复用后继第二段存储剩下的长度 x；如果第二段的值仍为 y，再用第三段表示新余下的长度 x，等等。这里，所谓最小匹配长度，指匹配的字符数只有达到该值时，采用匹配串描述码才能达到压缩数据的目的。显然，当匹配串长度为 1 时，采用匹配串描述码不但不能缩短码长，反而会扩大码长。因此，最小匹配长度必大于 1。最小匹配长度究竟取值多少，取决于 $L1$ 和 $L2$。比如，当历史表长等于 4K 字节、用 4 个二进制位为一段存储匹配串长度时，最小匹配长度等于 2。

无论采用等长码还是变长码，都必须遵循一个基本原则，即必须确保被压缩的数据能够得到正确、唯一的译码。由上述压缩码结构的描述，尽管我们采用变长码通常比等长码的操作要复杂些，但是它的结构和取值特征保证了这一原则不被破坏。

4.3　两种压缩码的比较

现在，我们来比较两种压缩码方案，看究竟哪一种更有效，也就是采用哪一种结构码产生的压缩码的总码长更短。

首先，我们讨论至少存在一次成功匹配时的情况。显然，此时的平均匹配长度 $L>0$，平均匹配成功率 $P>0$。又记输入数据总字节数为 N。另外，为简单起见，还进一步假定历史表长 4K，地址均用 4 个二进制位为一段表示(可表示 16 个不同的无符号数，其中最大数等于 15)。采用新算法时，匹配串长度需用额外一个存放 0 值的二进制位位段表示的平均概率$\leqslant 2/16=0.125$(每 16 个连续的整数中最多只有二个为 15 的倍数)。不难理解：

(1) 成功的匹配次数为 $P(N/L)=PN/L$；

(2) 包含在成功的匹配串中的字符总数为$(PN/L)\times L=PN$；

(3) 孤立的、不包含在成功的匹配串中的字符总数为 $N-PN=(1-P)N$；

(4) 采用新码时，串长需要额外存放 0 值的位段的匹配数为 $0.125PN/L$。

于是(以下码长以比特为单位)，

$$LZ77\text{ 的总码长}=24(PN/L+(1-P)N) \tag{1}$$

由于新算法允许较大的匹配长度，在变长码中无论匹配串长度多大，其地址总是占用 12 比特，再考虑到区分普通 ASCII 码和匹配串描述码的标志位，可知

$$\begin{aligned}\text{新算法总码长}&<9(1-P)N+17(1-0.125)PN/L+21\times 0.125PN/L\\&=17.5PN/L+9(1-P)N\end{aligned} \tag{2}$$

由(1)、(2)有：

$$\text{LZ77 的总码长}-\text{新算法总码长}>6.5PN/L+15(1-P)N>0 \tag{3}$$

其中，$0<P\leqslant 1$，$L>0$。

现在，再让我们看看不存在任何成功匹配时的情况。根据两个算法的压缩码结构，显然有：

$$\text{LZ77 总码长}-\text{新算法总码长}=24N-9N=15N>0 \tag{4}$$

上述计算方法和结果可以推广到历史表取不同长度、存储匹配串长度信息的位段取不同宽度时的情况。由(3)、(4)两式可知，LZ77 与新算法相比，平均匹配长度 L 越小，压缩效率越低。另外，匹配成功的平均概率 P 越大，LZ77 与新算法的压缩效率越接近；否则越差。但无论如何，LZ77 的平均压缩效率恒低于新算法。实验结果与这里的结论相符。实际上，如果匹配不成功，LZ77 不仅没达到压缩的目的，反而付出非常大的代价将输入字符由原来的 8 位扩展到“压缩码”的 24 位。显然，如果匹配成功的概率越小，被扩展的输入数据就越多，从而严重影响到最终总体的压缩效率。

5 编译码算法描述

见图 6、图 7。

6 实验分析及与 LZ77 的对比

6.1 在压缩效率与压缩速度之间寻求平衡

如果不采取一定策略制止贪婪匹配，编码速度将因为频繁的匹配过程而过于缓慢。禁止贪婪匹配，意味着在没有找到实际存在的最大匹配之前，因为仅仅发现了一个适当长度(记 FAIR_MATCH_LEN)的匹配串即确认匹配过程已经获得成功、而不再继续寻求更大的匹配串，这类似于人类行为中所谓的见好就收。显然，采取这一策略可能需要在压缩效率方面做出牺牲。不过，通过对 FAIR_MATCH_LEN 取不同数值进行实验，总能找到一个理想的值，使得在压缩效率与

压缩速度之间达成某种平衡。由表 3 可知，当 FAIR_MATCH_LEN 取值 6～10 时，通常较为理想。

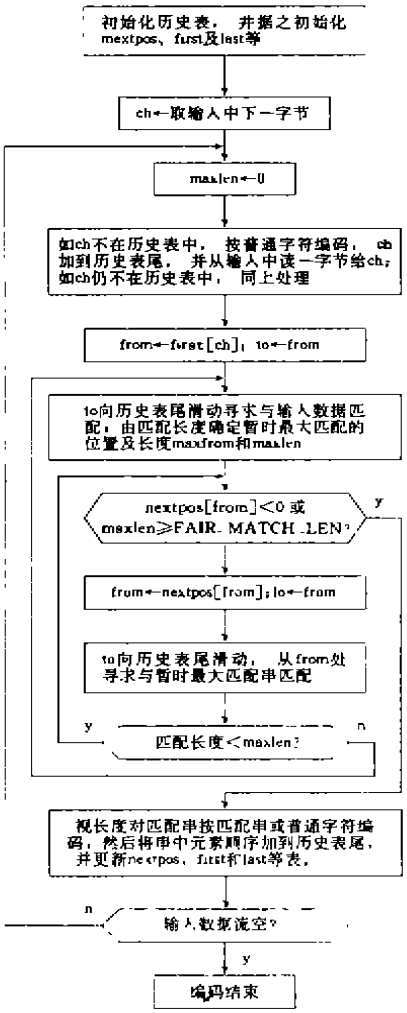

图 6　压缩算法基本流程图

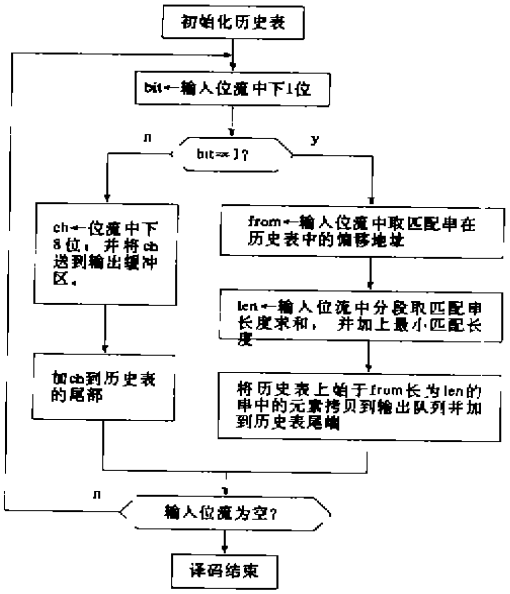

图 7　译码算法基本流程图

6.2　采用匹配位置滑动表技术能够显著改进时间性能

引入匹配位置滑动表技术对于提高压缩算法的性能是否很有必要呢？前面已经提到，它仅仅只围绕一个目的，这就是加快编码速度。采用这一技术与否，不但与压缩比无关，而且与译码过程无关。以下，我们通过对比实验来考察引入这一技术的真实意义。

对比实验比较容易进行。仅改变采用匹配位置滑动表技术的源程序 STRING.C 中寻找最大匹配串这一函数，同时删除与更新匹配位置滑动表有关的操作，即得到不采用匹配位置滑动表的程序 STRING0.C。这两个 C 程序绝大部分是完全相同的，即使寻找最大匹配的子程序有些差异，也仅仅表现在寻找每一个可能的匹配位置的少数几行语句上，总的匹配算法仍完全相同。为了使比较尽可能客观，我们选择同一编译器和相同的编译选项。这里，选择 BorlandC++3.1，并按下列步骤建立命令文件 STRING.COM 和 STRING0.COM：

(1) bcc -e *.exe -mt *.c

(2) exe2bin *.exe *.com

其中，bcc.exe 是 Borland C++的命令行编译器，exe2bin.exe 是 MS-DOS 系统盘上的实用程序。

我们对表 3 中的文件分别用 STRING.COM 和 STRING0.COM 进行对比实验，得到表 1。

表 1 两种算法的时间和压缩比对比

文件名	字节数	压缩时间(单位:秒)		压缩比
		string0.com	string.com	
STRING.COM	11330	41.03	1.92	1.27
STRING0.COM	11074	40.43	1.86	1.26
COMMAND.COM	47845	134.95	8.90	1.39
BCC.EXE	672150	1366.00	116.06	1.90
TC.EXE	290249	703.05	49.49	1.53
PWB.EXE	555874	1275.81	116.22	1.67
CHINESE1.TXT	433336	807.02	83.16	1.41
CHINESE2.TXT	314819	545.41	53.99	1.53
CHINESE3.TXT	88778	138.96	10.66	2.09
CHINESE4.TXT	51942	105.24	8.29	1.50
CHINESE5.TXT	238528	449.46	43.83	1.41
CHINESE6.TXT	38621	74.70	5.83	1.61
CHINESE7.TXT	12975	33.23	2.09	1.43
CHINESE8.TXT	28712	50.37	4.40	1.71
ENGLISH1.TXT	22721	35.54	8.02	2.01
ENGLISH2.TXT	18730	28.23	6.21	2.16
ENGLISH3.TXT	26575	39.54	8.40	2.14
ENGLISH4.TXT	18106	28.83	6.32	2.09
STRING.C	20508	41.53	4.39	2.21
LARGE.DBF	476354	285.83	65.36	5.49
SMALL.DBF	39710	24.34	5.28	4.18
TARTAN.BMP	32886	6.64	2.58	6.46
NONSENSE.TXT	124417	1.87	2.19	29.39

注：(1) 历史表长 4096 字节，匹配串长度用 4 个二进制位为一段描述；

(2) FAIR_MATCH_LEN=8。

为什么两个算法的时间开销有时存在显著差异、有时却很接近呢？一般地讲，随着数据分布随机性的增大，匹配位置滑动表技术能够更有效地加速匹配过程，因此，采用这一技术将明显缩短时间。表中，NONSENSE.TXT 完全由仅包含空格(或某一特定字符，如"a")的正文行构成。可以想象，在编码过程经历了一个短暂的阶段之后，由于在历史表的开始位置总能获得成功的匹配，且匹配串的长度大于或等于 FAIR_MATCH_LEN，因此，无论哪种算法均能在第一次尝试匹配时获得成功。但由于采用匹配位置滑动表技术需要更新匹配位置滑动表及相关数据，其运行

速度要稍比不采用该技术慢。这是表 1 中采用滑动表技术比之不采用该技术稍慢的唯一一个例外，而且正如文件名所示，它是一个没有实际意义的数据文件。表中，*.BMP 文件系 MS Windows 的图形文件。大家知道，二值图象存在大量的"空白"块。这使得两种算法在编码空白块时都能够迅速找到长度达到或超过 FAIR_MATCH_LEN 的成功匹配，因此，两种算法在编码时间上没有太大的差异。至于数据库文件 *.DBF，由于字段宽度总是被定义得足够大，因此，等长记录的数据库文件中免不了存在大量连续出现的空格符，同样道理，两种算法间的时间差异还是相对地要小些。不过，由于数据库文件总是经过精心设计，其冗余度应保证尽可能小，记录之间的关系远比二值图象数据之间松散，所以，这里的时间差异要比压缩 *.BMP 大。再看现代英语文本 ENGLISH?.TXT 和现代汉语文本 CHINESE?.TXT。由于英语文本中大量"片段"(例如，□at□the□和 ization□等，这里□代表空格符)频繁出现，而且其长度又往往达到甚至超过 FAIR_MATCH_LEN，相比之下，汉语文本中却几乎不存在类似的语言现象，故在英语文本中寻找成功匹配要比在汉语文本中来得迅速。所以，不采用匹配位置滑动表技术压缩 CHINESE?.TXT 比 ENGLISH?.TXT 通常要慢。相反，采用这一技术压缩汉语文本会更快。这是因为汉语系大字符集，即使将汉字拆作单字节编码压缩，其字符总数以国家标准《信息交换用汉字编码字符集基本集 GB2312-80》汉字内部交换码的位号计算有 94 个，比英语的 52 个(其中 26 个大写字母使用频率相对很低)也近多出一倍；再考虑到汉语文本中词汇之间无空格符，不难理解，编码汉语文本时匹配位置滑动表上的平均滑动速度要比英文快很多。最后，对于可执行文件和命令文件而言，文件内容的随机性增大，因此，两种算法表现出显著的时间差异，采用匹配位置滑动表技术能大大加快编码过程。

另一方面，可以看出，采用匹配位置滑动表技术，压缩时间与文本长度几乎很规则地成正比；而不采用匹配位置滑动表技术，压缩时间在很大程度上取决于文本中的数据分布情况，即对数据非常敏感。

的确，从表 1 看，匹配位置滑动表技术的意义十分明显。然而，要在程序中增加匹配位置滑动表 nextpos[0..HISTORY_SIZE-1]，而且，为了方便地更新、维护它还必须增加线性表 first[0..255]和 last[0..255]。这显然比不采用匹配位置滑动表技术需要额外 2×(HISTORY_SIZE+256+256)=2(HISTORY_SIZE+512)字节的内存空间。考虑到要在压缩率与压缩速度之间达成某种平衡，循环历史表不能太大，4096 字节是比较合适的选择，即 HISTORY_SIZE=4096。此时，所需要的额外空间为 9K。可以讲，这并不构成现代计算机系统的任何负担。因此，基于字符串匹配的压缩算法应该充分考虑匹配位置滑动表技术。

6.3 对比 LZ77

由第四节的讨论可知，新算法的压缩效率恒比 LZ77 为佳。表 2 是一组关于这两个算法的压缩比的实验数据。表中，LZ77′对 LZ77 的压缩码进行了一种合理的精简。上面谈到，LZ77 压缩码中记录匹配串最后一个字符纯属多余。为了避免这种压缩码自身的冗余，且能保证压缩码等长一致，LZ77′约定：(1) 工作缓冲区至少 256 个字节大小；(2) 当编码孤立数据时，因为具有匹配串长为 0 这一区别性特征，因此用结构码中描述成功匹配串在工作缓冲区上的偏移地址的码段记录孤立数据的 ASCII 码。于是，LZ77′的压缩码仅保留 LZ77 压缩码的前两个码段。表中的数据清楚地表明：LZ77′的压缩率较 LZ77 有显著提高。即使如此，新算法的平均压缩效率仍然比 LZ77′高得多。

特别值得注意的是：对于英语文本，LZ77′的压缩效率要略比新算法高；但对于计算机可执行的二进制文件，尤其是汉语文本，新算法的压缩效率远为可观。的确，任何一个有价值的压缩算

法都有其局限性，压缩效率的高低取决于具有不同特征的数据。但就算法对于不同类型数据的敏感程度，总可以综合评价一个算法的优劣。根据普遍的认识，如果单纯一种算法的平均压缩比能够达到 1.50，这一算法可称得上优秀。我们的新算法就是这种算法。

表 2 两个算法的压缩效率对比

文件名	字节数	压缩比		
		LZ77	LZ77′	新算法
STRING.COM	11330	0.66	0.99	1.27
STRING0.COM	11074	0.66	0.99	1.26
COMMAND.COM	47845	0.73	1.10	1.39
BCC.EXE	672150	1.05	1.58	1.90
TC.EXE	290249	0.83	1.25	1.53
PWB.EXE	555874	0.91	1.36	1.67
CHINESE1.TXT	433336	0.85	1.28	1.41
CHINESE2.TXT	314819	0.91	1.37	1.53
CHINESE3.TXT	88778	1.22	1.83	2.09
CHINESE4.TXT	51942	0.91	1.37	1.50
CHINESE5.TXT	238528	0.86	1.29	1.41
CHINESE6.TXT	38621	0.97	1.46	1.61
CHINESE7.TXT	12975	0.85	1.28	1.43
CHINESE8.TXT	28712	1.07	1.60	1.71
ENGLISH1.TXT	22721	1.37	2.05	2.01
ENGLISH2.TXT	18730	1.44	2.17	2.16
ENGLISH3.TXT	26575	1.42	2.13	2.14
ENGLISH4.TXT	18106	1.41	2.11	2.09
STRING.C	20508	1.26	1.90	2.21
LARGE.DBF	476354	2.76	4.14	5.49
SMALL.DBF	39710	2.74	4.11	4.18
TARTAN.BMP	32886	4.56	6.84	6.46
NONSENSE.TXT	124417	4.99	7.49	29.39

注：同表 1。

本文不准备对两个算法的时间性能进行实验比较。这主要因为 LZ77 的工作缓冲区系非循环队列，平移操作可以用高效、简单的专门机器指令完成(如 IBM 宏汇编指令 MOVSB)，因此，用高级语言还是直接用机器语言实现算法在运行效率方面可能存在很大的差别。此时，如果采用 C 程序设计语言实现 LZ77，算法的运行效率未必能得到准确地反映，而如果采用机器语言实现，则要求新算法也应采用同一语言书写，否则，实验的相对性能结果很难说具有可比性。然而，类似 MOVSB 的指令其时钟周期与待移动的数据对象的长度、也即此处的工作缓冲区的尺寸成正比。因此，可以断言，当工作缓冲区达到一定长度时，新算法的时间性能也将比 LZ77 更好。

关于译码的时间性能，新算法与 LZ77 之间没有明显差别，用以译码的时间均比编码短得多。

由于新算法描述长配匹串时更经济，用以解析包含在压缩码中的偏移地址所开销的时间更短。因此，其时间性能总的讲要比 LZ77 好。

7 结束语

值得注意的是，通过对程序进行微调可以改进压缩比及压缩速度。首先，可以对历史表进行专门的初始化设计。譬如，从英语语料中精心选取高频词汇和高频片语用以初始化历史表，则对于适当规模的英语文本，其压缩率会有明显提高。但是，它对于大的英语文本、别的语文文本或其他类型文件不会有太大的作用。然而，压缩程序应具有良好的实用性，它应该以最快的自适应速度适应被压缩数据的模式。考虑到历史表表长取 4K 时压缩速度尚能接受，在这么大的空间足以装入各类常见数据的最高频出现的数据模式。因此，实际投入使用的程序最好从多种常用语言的文本中选取高频出现的单词、片语，或者从可执行的二进制代码程序中抽取高频出现的指令或其组合等等，将这些模式拼接起来初始化历史表，并根据历史表初始化匹配位置滑动表等。其次，还可以对历史表表长、存储匹配串长度信息的二进制位位段宽度等程序参数取不同值、不同组合进行实验，在实验数据的基础上选择一组比较理想的经验数据代换到程序中，从而保证在压缩速度和压缩比率两方面均达到较理想的程度。

总之，新算法的压缩效率要比 LZ77 好得多，加入配匹位置滑动表技术后，新算法的时间性能获得数量级上的提高。我们认为，在实际工程应用中，LZ77 应让位给新算法。

表 3　FAIR_MATCH_LEN＝4,6,8…时的压缩比与压缩时间

文件名	字节数	4		6		8		10	
		压缩比	时间	压缩比	时间	压缩比	时间	压缩比	时间
STRING.COM	11330	1.23	1.98	1.26	1.82	1.27	1.92	1.27	1.92
STRING0.COM	11074	1.24	1.81	1.26	1.81	1.26	1.86	1.26	1.87
COMMAND.COM	47845	1.31	9.40	1.38	9.01	1.39	8.90	1.39	9.39
BCC.EXE	672150	1.79	101.78	1.86	110.62	1.90	116.06	1.91	112.05
TC.EXE	290249	1.47	50.75	1.51	47.84	1.53	49.49	1.53	53.72
PWB.EXE	555874	1.62	107.49	1.66	114.13	1.67	116.22	1.68	112.21
CHINESE1.TXT	433336	1.38	77.50	1.41	81.95	1.41	83.16	1.42	84.53
CHINESE2.TXT	314819	1.50	54.15	1.52	53.94	1.53	53.99	1.54	57.67
CHINESE3.TXT	88778	1.95	10.05	2.05	9.94	2.09	10.66	2.11	11.75
CHINESE4.TXT	51942	1.44	8.13	1.49	8.13	1.50	8.29	1.51	9.06
CHINESE5.TXT	238528	1.38	44.54	1.41	43.66	1.41	43.83	1.42	47.23
CHINESE6.TXT	38621	1.53	5.60	1.59	5.66	1.61	5.83	1.62	6.32
CHINESE7.TXT	12975	1.38	2.04	1.42	2.03	1.43	2.09	1.43	2.19
CHINESE8.TXT	28712	1.61	4.11	1.69	4.23	1.71	4.40	1.72	4.67
ENGLISH1.TXT	22721	1.89	6.48	1.99	7.69	2.01	8.02	2.03	8.02
ENGLISH2.TXT	18730	2.00	5.16	2.12	6.15	2.16	6.21	2.18	6.26
ENGLISH3.TXT	26575	1.99	6.87	2.10	8.02	2.14	8.40	2.16	8.35
ENGLISH4.TXT	18106	1.92	5.16	2.05	6.15	2.09	6.32	2.10	6.26
STRING.C	20508	2.03	3.46	2.16	4.07	2.21	4.39	2.24	4.56
LARGE.DBF	476354	5.13	39.66	5.41	46.46	5.49	65.36	5.53	69.10
SMALL.DBF	39710	3.66	4.61	4.09	4.95	4.18	5.28	4.36	5.98
TARTAN.BMP	32886	4.94	1.76	5.70	2.09	6.46	2.58	6.99	3.02
NONSENSE.TXT	124417	29.39	2.14	29.39	2.20	29.39	2.19	29.41	2.19

（接上表）

文 件 名	字节数	12		14		16		18	
		压缩比	时间	压缩比	时间	压缩比	时间	压缩比	时间
STRING. COM	11330	1.27	1.93	1.27	1.92	1.27	1.92	1.27	1.76
STRING0. COM	11074	1.26	1.87	1.26	1.87	1.26	1.86	1.26	1.81
COMMAND. COM	47845	1.40	9.12	1.40	9.61	1.40	9.07	1.40	9.34
BCC. EXE	672150	1.92	119.30	1.92	113.86	1.92	119.02	1.92	118.53
TC. EXE	290249	1.53	53.55	1.53	53.93	1.53	55.59	1.54	52.13
PWB. EXE	555874	1.68	117.38	1.68	113.04	1.68	117.87	1.68	117.87
CHINESE1. TXT	433336	1.42	84.20	1.42	85.58	1.42	84.42	1.42	84.92
CHINESE2. TXT	314819	1.54	54.60	1.54	58.27	1.54	55.15	1.54	56.85
CHINESE3. TXT	88778	2.12	11.36	2.12	12.25	2.13	11.75	2.13	12.36
CHINESE4. TXT	51942	1.51	8.73	1.51	9.61	1.51	9.28	1.51	9.56
CHINESE5. TXT	238528	1.42	44.71	1.42	48.01	1.42	44.99	1.42	46.02
CHINESE6. TXT	38621	1.62	6.04	1.62	6.54	1.62	6.10	1.62	6.37
CHINESE7. TXT	12975	1.44	2.08	1.44	2.25	1.44	2.25	1.44	2.31
CHINESE8. TXT	28712	1.72	4.45	1.72	4.78	1.72	4.50	1.72	4.61
ENGLISH1. TXT	22721	2.03	8.24	2.03	8.18	2.03	8.40	2.03	8.57
ENGLISH2. TXT	18730	2.19	6.43	2.19	6.37	2.19	6.48	2.19	6.59
ENGLISH3. TXT	26575	2.17	8.62	2.17	8.51	2.17	8.73	2.17	8.79
ENGLISH4. TXT	18106	2.11	6.60	2.11	6.48	2.11	6.65	2.11	6.70
STRING. C	20508	2.25	5.00	2.27	4.67	2.27	4.94	2.27	4.94
LARGE. DBF	476354	5.57	57.62	5.58	56.41	5.60	62.01	5.68	87.39
SMALL. DBF	39710	4.53	6.27	4.60	6.65	4.64	6.64	4.65	6.70
TARTAN. BMP	32886	7.38	3.85	7.63	4.12	8.03	5.33	8.14	7.14
NONSENSE. TXT	124417	29.41	2.20	29.41	2.20	29.41	2.20	29.41	2.20

注：(1) 本文所讨论的实验均是在 Smart 386DX/33 上(未安装数字协处理器)完成的；

(2) 历史表长度为 4096 字节，匹配串长度用 4 个二进制位为一段描述；

(3) 压缩比＝文件压缩前的长度/文件压缩后的长度；

(4) CHINESE?. TXT 系政府电子部门有关行业发展的综合报告(中文)，ENGLISH?. TXT 取自计算机基础知识教科书(英文)。其余文件的说明，见表 1 下面的文字。

参 考 文 献

[1] Ziv J, Lempel A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 1977, 23(3).

[2] Held G, Marshall T R. *Data Compression*. John Wiley & Sons, 1991.

[3] Williams R N. *Adaptive Data Compression*. Kluwer Academic Publishers, 1991.

[4] Storer J A. *Data Compression-Methods and Theory*. Roclville MD: Computer Science Press, 1988.

[5] Bell T C *et al*. *Text Compression*. Prentice Hall, 1990.

[6] 陈 俭等. 关于 ZL 数据压缩算法性能的实验研究. 计算机应用, 1992, 12(5).

[7] 徐秉铮等. 中文文本压缩的 LZW 算法. 华南理工大学学报(自然科学版), 1989, 17(3).

[8] 贺前华等. 中文文本压缩的自适应算法. 中文信息学报, 1993, 7(3).

[9] 王忠敦. 基于字符串匹配的通用数据压缩算法. 计算机应用, 1995,(1).

[10] 姜 丹. 信息理论与编码. 合肥：中国科学技术大学出版社, 1992.