Software Engineering and Applications 软件工程与应用, 2014, 3, 50-56
Published Online June 2014 in Hans. http://www.hanspub.org/journal/sea
http://dx.doi.org/10.12677/sea.2014.33007

Hans汉斯

Modified Algorithms of the LZ77 Data Compression

Jianjun Huang, Zhenglu Jiang

Sun Yat-Sen University, Guangzhou
Email: mcsjzl@mail.sysu.edu.cn

Received: Apr. 11th, 2014; revised: May 10th, 2014; accepted: May 20th, 2014

Abstract

Every day much information is transmitted in such forms as images, sounds and texts. These data of images and sounds are particularly large and need an efficient compression method. The LZ77 algorithm is one of the most effective compression algorithms. In this paper, two new algorithms are given in order to improve the performance of the original LZ77 algorithm. One is to check whether it can get the longest match before matching, and the other is to save the longest common prefix of the adjacent strings in the list to improve the efficiency. It is proved that the two new algorithms achieve the expectation after a comparative analysis with other versions of the LZ compression algorithms.

Keywords

Lossless Compression, Codeword Encoding, LZ77 Compression Algorithm, Hash Table

改进的LZ77数据压缩算法

黄健骏，姜正禄

中山大学，广州
Email: mcsjzl@mail.sysu.edu.cn

收稿日期：2014年4月11日；修回日期：2014年5月10日；录用日期：2014年5月20日

摘　要

每天都有大量的信息，而这些信息以诸如图像、声音和文本的形式来传递。其中图像和声音的数据量特别地大，需要高效的压缩方法。LZ77算法是有效的压缩算法之一。本文针对LZ77算法提出两种新的算法，来提高压缩算法的性能。算法一在查找匹配前先检验是否可能得到最长匹配，而算法二则是保存链表中相邻字符串的最长公共前缀来提高效率。与其他版本的LZ系列压缩算法进行对比分析后发现，改进后的这个新方案达到预期效果。

关键词

无损压缩，字典编码，LZ77压缩算法，散列表

1. 引言

现今越来越多的研究者意识到数据压缩的重要性，这主要是因为当今如此大量的数据不仅占据了大量的存储空间，而且占据了大量的传输带宽，使得计算机的数据处理和通信干线信道的带宽都受到了极大的压力。因此，无论从存储角度还是从传输角度，对数据压缩进行研究都有着相当重大的意义。目前有许多较成熟的数据压缩技术，而 LZ77 算法是有效的压缩算法之一。LZ77 算法是由 Jacob Ziv 和 Abraham Lempel [1]在 1977 年给出的，其设计思想是采用字典查找方法，它是用滑动窗口来构作字典的。

后来，Jacob Ziv 和 Abraham Lempel 两人在 1978 年提出了另一种 LZ 算法[2]，这种算法被称为 LZ78 算法。LZ78 取消了 LZ77 算法的滑动窗口，从字符流中不断提取新的缀－符串保存到字典中，并在压缩过程中把字符串替换成相应的缀－符串。

1982 年 James Storer 和 Thomas Szymanski [3]对 LZ77 算法进行了改进，后来这个算法被称为 LZSS 算法。LZSS 算法通过二叉树匹配查找代替 LZ77 算法的顺序匹配查找，并通过一个额外的 ID 位来区分输出的是指针还是真实字符。在相同的条件下 LZSS 算法大大地提高了压缩的时间效率，同时可获得更小的压缩率。

1984 年，Welch 研究出一种高性能数据压缩技术，这是 LZ78 算法的一个变种，也就是后来非常有名的 LZW 算法[4]。LZW 算法的实现比较简单，同时它改善了 LZ78 算法的自适应性，字典一开始保存 256 个索引，每一条索引对应一个字符，在压缩时生成新的索引保存到字典中，并输出索引代替相应的字符串。

1991 年 Ross Williams 提出的 LZRW 算法[5]是 LZ77 算法的另一改进，它的主要思想是通过构建散列表来提高算法的效率。使用散列表来进行匹配的压缩速度是极快的，把每一个字符串的开头若干个字节作为散列表的关键码值，通过散列函数找到开头若干个字节与之匹配的字符串的位置，之后再做顺序的匹配。另外，LZRW 算法没有改进 LZ77 算法的压缩比。

以上介绍的算法统称 LZ 系列算法，这些算法以及它们的各种变体几乎垄断了整个通用数据压缩领域，我们熟悉的 WinZIP、Gzip 等压缩工具都采用了 LZ 算法来进行压缩。其它有关 LZ 算法的研究可以参考文献[6]-[8]。

本文主要研究如何提高 LZ77 算法的时间效率，提出了两种新的 LZHJ 算法。这两种新算法都是通过链表法实现的散列表来进行匹配查找，其中 LZHJ 算法一在查找匹配前先检验是否可能得到最长匹配，而 LZHJ 算法二则是保存链表中相邻字符串的最长公共前缀来提高效率。通过实验对比新的算法达到预

期的效果。

2. LZHJ 算法

LZ77 压缩算法的性能瓶颈在于查找最长匹配长度上，由于 LZ77 算法在滑动窗口中顺序查找匹配，所以当滑动窗口过大时，算法的时间效率就变得极低。为此，我们提出两种新的算法，统称为 LZHJ 算法。

LZHJ 算法都是由 LZ77 算法[1]改进而来的，它跟 LZSS[3]算法一样包括一个滑动窗口和一个输入缓存器，而且如果最大匹配长度大于 1，就输出一个由 1、匹配位置和最大匹配长度组成的三元组，否则输出一个二元组，它由 0 和一个源字符组成。

另外，LZHJ 算法采用了 LZRW 算法[5]中的散列表来提高算法的时间效率。这里对每个长度为 2 的字符串建立索引，散列函数的参数是字符串本身的前 2 个字符值。每个索引点之后是该字符串在滑动窗口中出现的所有位置，并且使用单链表来存储这些位置。在此基础上，LZHJ 算法改进了原来查找最长匹配的方式，这也是本文的主要工作。

以字典查找为设计思想的 LZ77 算法及其改进出来的 LZSS 算法和 LZRW 算法都使用滑动窗口中出现过的数据来替换当前数据，而 LZ78 算法和 LZW 算法都通过预先扫描输入缓存中的数据并与其字典中的数据进行匹配来实现压缩。对于大型的数据而言，LZ78 算法往往能得到较小的压缩率。作为其实用版本的 LZW 算法，它设计逻辑简单，有利于简单高效快速的实现。可是 LZ78 算法和 LZW 算法的自适应速度比较慢，而 LZ77 算法、LZSS 算法和 LZRW 算法则有较好的自适应性。一个长度为 10 的字符串出现 10 次，LZ78 算法才能完整拷贝它；LZW 算法需要这个字符串出现 9 次才能完全拷贝它；而 LZ77 算法、LZSS 算法、LZRW 算法和后面给出的新的 LZHJ 算法都只需在这个字符串第 2 次出现时就能对其进行完整拷贝。

一般地说，在相同条件下，LZHJ 算法运行所花的时间比 LZ77、LZSS 和 LZRW 这三种算法所花的都少，而且其压缩率比 LZ78 和 LZW 这两种算法的都要小。下面介绍 LZHJ 算法查找匹配的方式。

2.1. LZHJ 算法一

在介绍 LZHJ 算法一之前，先介绍 LZ77 需要修改的地方，为此需要引进一些记号。对于前向缓冲中的字符串，记为 $X_1, X_2, \cdots, X_N$，对于当前匹配字符串记为 $Y_1, Y_2, \cdots, Y_K$，其中 Y_K 为滑动压缩窗口中的最后一个字符。当前最大匹配长度记为 m，则有 $N > m$，而且 $X_1 = Y_1$，$X_2 = Y_2$。对 $X_1, X_2, \cdots, X_N$ 和 $Y_1, Y_2, \cdots, Y_K$ 进行有顺序的对比，得到的匹配长度记为 *Length*，则有

$$Length = \max\left\{i \mid X_i = Y_i, 2 \le i \le \min(N, K)\right\}。$$

在 LZ77 算法中，如果有 $Length > m$ 成立，就把最大匹配长度的值改为 *Length*，并把匹配位置改为当前字符串在滑动窗口中的位置。另外，如果下一匹配位置存在，则跳到下一匹配位置进行匹配；如果在对比 $X_1, X_2, \cdots, X_N$ 和 $Y_1, Y_2, \cdots, Y_K$ 之后得到的匹配长度为当前最大匹配长度，即 $Length = m$，那么这种有顺序的对比总共进行 $m-2$ 次；事实上，有顺序的对比从第三个字符开始，而第一和第二个字符无需对比，这是因为有 $X_1 = Y_1$ 和 $X_2 = Y_2$。所以 LZ77 算法得到的匹配长度并没有比已有的最大匹配长度大，这样的比较显得意义不大。这便是 LZ77 问题所在。

因此，在 LZ77 算法的基础上，只摒弃了 LZ77 算法中顺序查找最长匹配的方法，提出如下新的匹配方案，称之为 LZHJ 算法一。

在 LZHJ 算法一中，对 $X_1, X_2, \cdots, X_N$ 和 $Y_1, Y_2, \cdots, Y_K$ 进行对比之前，会先判断条件 $X_{m+1} = Y_{m+1}$ 是否成

立。如果这个条件不成立，就有 $Length \le m$，这样一来，这个新算法就可以直接跳过 $X_1, X_2, \cdots, X_N$ 与 $Y_1, Y_2, \cdots, Y_K$ 之间的对比，跳到下一个匹配位置中去寻找最大匹配长度；否则就对 $X_1, X_2, \cdots, X_N$ 和 $Y_1, Y_2, \cdots, Y_K$ 进行有顺序的对比，得到它们的匹配长度。与这样一个有顺序的对比作比较，这样一个判断条件可以更快地实现最长匹配长度的查找。

事实上，由于文件的上下文之间一般具有相关性，所以一般可以把未压缩的字符串看作是具有有限记忆的性质的马尔科夫链。这就意味着当前位置出现给定字符的概率要取决于当前位置之前若干个字符串。由于字符 $X_{m+1}(Y_{m+1})$ 与 $X_1(Y_1)$ 相距较远，所以还可以把它们看作是相对独立的。同时由于信源的遍历性质

$$H(X_3|X_1, X_2) \le H(X_3) = H(X_{m+1}) = H(X_{m+1}|X_1, X_2)$$

其中 $H(X)$ 表示香农信息熵，所以可以看出在已知 X_1 和 X_2 的情况下，X_{m+1} 的不确定度要比 X_3 的不确定度大。因此，在 $X_1 = Y_1$ 和 $X_1 = Y_1$ 成立的情况下，$X_3 = Y_3$ 的几率也相对较高。如此一来，当有顺序地对比两个字符串时，即使得到的当前匹配长度小于已有的最大匹配长度，也需要进行好几回的对比之后才能得到第一对互不相同的字符，记为 (X_i, Y_i)，且 $X_i \ne Y_i$；另一方面，由于字符 $X_{m+1}(Y_{m+1})$ 与 $X_1(Y_1)$ 相距较远，所以可以把它们看作是相对独立的，故而 $X_{m+1} = Y_{m+1}$ 的几率相对来说一般比较小，即有

$$P\{X_{m+1} = Y_{m+1} | X_1 = Y_1, X_2 = Y_2\} \le P\{X_3 = Y_3 | X_1 = Y_1, X_2 = Y_2\}。$$

因此，在匹配长度小于已有的最大匹配长度的情况下，对 X_{m+1} 和 Y_{m+1} 进行对比，LZHJ 算法一可以更快得到不匹配的结果，减少与不必要的字符串进行比较，从而可以更快地得到最大匹配长度和它对应的匹配位置。

LZSS 算法是在二叉树中查找最长匹配的，而 LZHJ 算法一使用散列表的方式来组织已压缩的数据，并且在查找匹配前先检验是否可能得到最长匹配。在相同条件下，与 LZSS 算法相比，LZHJ 算法一的时间效率较高。

2.2. LZHJ 算法二

LZHJ 算法二也是在 LZ77 算法的基础上提出的新的技巧。在 LZ77 算法中，使用散列表来进行字符串匹配，这时每次压缩完成一段字符串以后，都要对被压缩的字符串进行处理，如果散列函数的参数是字符串本身的前 2 个字符，就要把其中任意双字符的位置信息加入到相应的链表中，以便后续的匹配查找。在压缩的过程中，某些字符串会反复地出现，从而导致查找最长匹配的效率降低。下面给出的 LZHJ 算法二仅仅修改 LZ77 算法中这段处理，以此提高查找最长匹配的效率。

沿用上一节的记号，前向缓冲窗口中的字符串为 $X_1, X_2, \cdots, X_N$，当前匹配字符串为 $Y_1, Y_2, \cdots, Y_K$，当前匹配长度为 $Length$，当前最长匹配长度为 m。

当 LZHJ 算法二将一个双字符加入到链表的时候，它把这个双字符为首的字符串与其在链表中的下一字符串进行对比，并记录链表间相邻字符串的最长公共前缀的长度，记之为 $same\_length$。还有，在 LZHJ 算法二中，如果散列表中还有下一匹配位置存在，在与下一个匹配位置中的字符串进行对比之前，先对 $Length$ 与当前匹配位置的 $same\_length$ 作一次比较，从而减少不必要的匹配查找。这样一次的比较会有三种可能，分别作如下处理：

1) 若 $Length > same\_length$，则可知字符串 $X_1, X_2, \cdots, X_N$ 与下一匹配位置的字符串匹配所得到的长度必为 $same\_length$，就把 $same\_length$ 赋值给 $Length$；

2) 若 $Length < same\_length$，则字符串 $X_1, X_2, \cdots, X_N$ 与下一匹配位置的字符串匹配得到的长度为 $Length$；

3) 若 $Length = same\_length$，则 $X_1, X_2, ..., X_N$ 与下一匹配位置的字符串的前 $Length$ 个字符都相同，这样一来，从它的第 $Length+1$ 个字符开始进行对比。如果得到的匹配长度 $Length > m$，就把 m 的值改为 $Length$，并把匹配位置改为当前字符串在滑动窗口中的位置。

无论是 LZHJ 算法一，还是 LZHJ 算法二，如果还有下一个匹配位置存在，就用各自相同的方法进行匹配；否则就令 $m = \max(Length, m)$，如果最大匹配长度 m 大于 1，就把当前匹配位置记录下来，并输出一个三元组，它是由 1、匹配位置和最大匹配长度组成的。

3. 实验数据的对比与分析

实验时采用现有的 LZ77、LZSS 和 LZW 压缩算法以及新的 LZHJ 算法对不同的文本文件进行压缩，并比较这些算法的所耗时间和压缩率。表 1 和图 1 给出 LZ77、LZSS 和新算法的所耗时间的对比，其中 LZHJ_1 和 LZHJ_2 分别表示 LZHJ 算法一和算法二，LZ77_Hash 表示使用散列表实现的 LZ77 算法，它是基于 LZRW 算法实现的。LZSS 算法的滑动窗口大小为 4 KB，压缩对象是四个大小不同的文档文件。

在图 1 中，两个改进方案相对于原有的算法在时间效率上均有明显的改善，其中方案一的运行时间约为原有算法运行时间的 45%，而方案二的运行时间约为原有算法运行时间的 60%。另外，图 1 还表明采用二叉树的 LZSS 算法比其它三种使用散列表的算法更耗时，这意味着使用散列表可以使得程序运行时间更短。

Table 1. Comparison of run time
表 1. 运行时间的对比

公共数据集 / 所耗时间 / 压缩算法	Test_1.txt (4363KB)	Test_2.txt (959KB)	Test_3.txt (142KB)	Test_4.txt (35KB)
LZ77_Hash	8188 ms	2359 ms	312 ms	47 ms
LZHJ_1	3610 ms	1031 ms	141 ms	16 ms
LZHJ_2	5093 ms	1453 ms	188 ms	31 ms
LZSS	44015 ms	11375 ms	1828 ms	406 ms

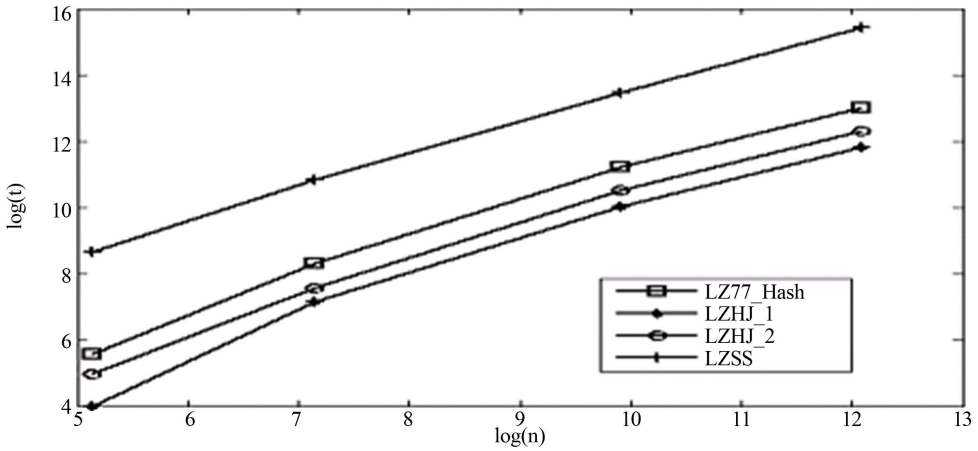

Figure 1. Run time
图 1. 运行时间

Table 2. Comparison of compression ratios
表 2. 压缩率的对比

压缩后的字节数 / 公共数据集 / 压缩算法	Test_1.txt (4363KB)	Test_2.txt (959KB)	Test_3.txt (142KB)	Test_4.txt (35KB)
LZ77_Hash	1560KB	427KB	71KB	17KB
LZHJ_1	1560KB	427KB	71KB	17KB
LZHJ_2	1560KB	427KB	71KB	17KB
LZW_12	2097KB	516KB	74KB	18KB
LZW_16	1617KB	364KB	72KB	22KB

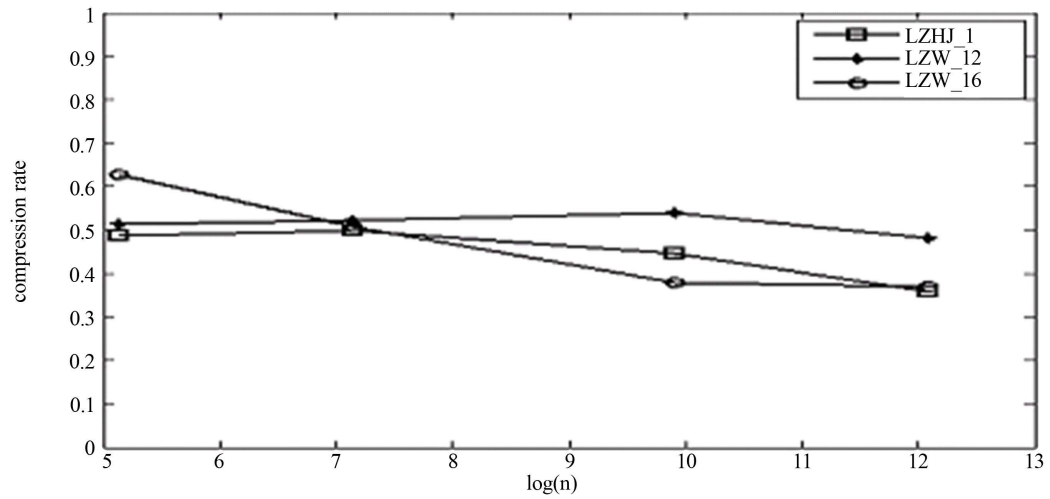

Figure 2. Compression ratios
图 2. 压缩率

表 2 和图 2 给出了 LZW 算法和 LZHJ 算法的压缩率的对比。其中 LZW_12 是 12 位的字典，LZW_16 是 16 位的字典。LZHJ_1、LZHJ_2 和 LZ77_Hash 的压缩率是一样的。

从图 2 可以看出，如果文件较小，LZ77_Hash 的压缩率一般就会比 LZW 算法的压缩率低；如果文件较大，LZW 的压缩率就可能会比 LZ77_Hash 的压缩率低；而且 LZHJ 算法的压缩率总体上要小于 LZW 算法的压缩率。

基金项目

该项目得到国家自然科学基金资助。项目号：11171356。

参考文献 (References)

[1] Ziv, J. and Lempel, A. (1977) A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, **23**, 337-343.

[2] Ziv, J. and Lempel, A. (1978) Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, **24**, 530-536.

[3] Storer, J. and Szymanski, T. (1982) Data compression via textual substitution. *Journal of the Association for Compu-*

ting Machinery, **11**, 928-951.

[4] Welch, T.A. (1984) A Technique for High-Performance Data Compression. *Computer*, **6**, 8-19.

[5] Williams, R.N. (1991) An extremely fast Ziv-Lempel data compression algorithm. *Data Compression Conference* 1991 (DCC'91), 8-11 April 1991, Snowbird, 362-371.

[6] Bender, P.E. (1991) New asymptotic bounds and improvements on the Lempel-Ziv data compression algorithm. *IEEE Transactions on Information Theory*, **5**, 721-729.

[7] Ziv, J. and Wyner, A.D. (1994) The sliding-window Lempel-Ziv algorithm is asymptotically optimal. *Proceedings of the IEEE*, **6**, 872-877.

[8] Kosaraju, R.S. and Manzini, G. (1997) Some entropic bounds for Lempel-Ziv algorithms. *Proceedings of DCC*, **3**, 446.