文件压缩作业报告

信息科学技术学院 史舒扬 1300012959

May 17, 2014

Contents

1 简述

通过对 lz77 压缩算法的学习，结合了一些编码技巧，添加了一些对本身 lz77 算法的改进，并尝试结合其它压缩的算法，笔者做了一个 myzip 的文件压缩和解压缩的小程序。

2 算法简述

2.1 算法基础：lz77

此处不再赘述，详情请看论文与课件。

2.2 窗口上的改动

原来的算法采用的是滑动窗口，每次动态维护字典的元素。这里我采用了静态窗口，并设置窗口大小最大为 64KB，即，每 64KB 的过程中动态添加字典，处理完 64KB 之后把字典清零重新计算。简单来说，这样的好处大概有：

- 时间上有一定的节省（节省了从字典数据结构删除的时间）；
- 方便从中间解压文件（如果要继续改进成多线程的压缩，这种写法具有可观的优势）；
- 不需要维护删除之后就不需要重复添加，这样可以使得 off 值相对保持较小，结合后文的编码方式可以适当改进。

2.3 匹配长度的限制

显然，匹配长度很小，例如只有 1、2 的时候，采用 off, len 这些方式来压缩是很不经济的，将使文件变大很多。为了避免这种效应，结合后文要提到的编码方式，这里设置了最小匹配长度限制 $MinMatchLength$。

另一方面，虽然匹配长度很大的时候，期望能有很大的压缩量，但是这样做有以下问题：

1. 数据结构有较大的维护代价，不仅有空间上的（虽然现在的硬件配置使得空间基本不成问题），还有时间上的（插入删除的时候会变得很慢），即，有较大弊端；
2. 实际处理压缩的时候并不太会出现这种情况，事实上除非刻意，简直没有那么好的运气碰上这样的情况，即，并无太大益处。

基于以上考虑，决定设置最大匹配长度 $MaxMatchLength$。

测试中采用的值分别为：

- $MinMatchLength = 3$
- $MaxMatchLength = 30$

2.4 编码上的改进

2.4.1 格式

原来的 lz77 算法使用的是三元组 $(off,\ len,\ c)$，其中 off 表示匹配串相对窗口的位移，len 表示匹配的长度，c 表示匹配串之后的下一个字符。但是由于实际过程中有大量的是单字符无匹配的情况（或者匹配之后的三元组编码反而加长了很多），所以考虑适当改变编码格式。具体如下。

每次先输出一个标志位 flag。分 flag 为 0 和为 1 两种情况：

flag = 0 表示匹配长度小，选择输出单字符，后跟八位单字符 ch。

flag = 1 表示有相对合理的匹配长度。下面输出二元组 $(len,\ off)$，意义与原先相同。

通过这样的方式，大大减少了原来不必要的浪费，该改进从效果来说还是比较明显的。

2.4.2 len 的编码方式

由前文可以看到，len 是属于 $MinMatchLength$ 和 $MaxMatchLength$ 之间的数，直接使用 5 bits 有所浪费，因为绝大部分匹配的长度比较小。考虑使用 Gamma 编码 (Elias Gamma Code)，即

> 对于任意的自然数 x，它的二进制需要 $floor(log(x)) + 1$ bits 来表示。在其二进制表示的前面加上 $floor(log(x))$ 个 0，即 Elias Gamma Code。
> 例如：13d = 1011b
> 所以，EGC(13d) = 000 1011b

经过测试确实对编码有所改进。

2.4.3 off 的编码方式

由于 off 的取值范围是 0 - 65535，但是由于普遍较大，且分布不均匀，因此不适合采用 Gamma Code，直接用 $Upper(log(WindowSize))$ 位存储。其中 $Upper(n)$ 是向上取整函数。

2.5 字典的数据结构

考虑到哈希容易出现大量冲突，这里字典选择使用平衡二叉树（为了实现方便，并且考虑到树的大小有限，这里使用 Treap）存储字符串，不进行删除。

3 Huffman 压缩算法

3.1 基本内容

将文件按照 8 位分隔为 unsigned char，构建 Huffman 树，建立 Huffman 编码进行压缩。

4 压缩实验（算法的单独使用以及组合）

4.1 文本文件

对文本文件的压缩，应该算是效果比较明显的，随机选了一个 299,284 字节的开源 C 代码（含大量注释），单独用改进过的 lz77 和 Huffman 压缩分别压缩到 240,647 字节和 171,540 字节，有 20% 至 40% 左右的压缩率。

分析原因，大概是因为：

1. 文本文件集中在可见字符区，利于 Huffman 压缩；
2. 文本文件容易出现重复段，利于 lz77 压缩。

组合的效果如下：

1. 在 huffman 的基础上进行 lz77 压缩该文件，得到的大小为 156,076 字节，压缩率 48%，用时 0.125 s。
2. 在 lz77 的基础上进行 huffman 压缩该文件，得到的大小为 224,030 字节，压缩率 25%，用时 0.125 s。

由于该文件不大，压缩和解压缩的过程都很快，结合使用有一定的改善作用。

4.2 EXE 可执行文件

选择了 1,815,412 字节的可执行文件（test.exe）进行压缩，表现如下：

1. 改进的 lz77: 得到 1,711,563 字节，压缩 6%。
2. huffman: 得到 1,307,960 字节，压缩 28%。
3. 先 lz77 后 huffman: 得到 1,323,278 字节，压缩率 27%，用时 1.254 s。
4. 先 huffman 后 lz77: 得到 1,213,587 字节，压缩 33%，用时 0.780 s。

4.3 MP3 音频文件

选择了 5,802,217 字节的 MP3 音频（InTransit.mp3）进行压缩，表现如下：

1. 改进的 lz77: 得到 6,456,255 字节，实际过程中可以不压缩直接复制。
2. huffman: 得到 5,785,702 字节，略有压缩。
3. 先 lz77 后 huffman: 得到 6,376,462 字节，实际过程中可以不压缩直接复制，用时 4.902 s。
4. 先 huffman 后 lz77: 得到 6,480,937 字节，实际过程中可以不压缩直接复制，用时 4.890 s。

由于音频本来就是经过编码和压缩的，所以对其进行压缩的效果不太理想也是符合预期的。

4.4 PDF 文件

选择了 1,553,447 字节大小的 PDF 文稿（the_data_compression_book_nd_edition.pdf，一本与数据压缩有关的书）进行压缩，表现如下：

1. 改进的 lz77: 得到 1,672,208 字节，基本不变。
2. huffman: 得到 1,539,839 字节，略有压缩。
3. 先 lz77 后 huffman: 得到 1,666,191 字节，基本不变，用时 1.206 s。
4. 先 huffman 后 lz77: 得到 1,683,334 字节，基本不变，用时 1.266 s。

5 实验结果小结

实验发现，相对 lz77，用 huffman 速度较快（不用维护字典），而且在单独使用的时候也有较好的表现。但是由于其算法要存储字典，使得有一定的起步大小（每个文件开头要存储下各个字符的 huffman 码）。

两算法结合会对之前的单算法使用用改进效果，两种顺序中略优一点的是先 huffman 后 lz77。由于时间的关系，程序写的是 lz77 之后再 huffman。

基于上述实验，考虑速度和效果，在实验后将 lz77 的宏中

$$MaxMatchLength = 30$$

改为

$$MaxMatchLength = 20$$

以上实验都只是针对个别文件的结果，不能完全代表结果。

6 文件及目录处理

在压缩文件的时候逐次存储，开头是文件名和大小；文件夹递归处理。

具体实现方法参见具体代码。

7 相关代码解释

具体代码见附件。主要有以下文件：

lz77.h lz77 压缩相关函数、常量以及位处理函数的声明；

lz77.c lz77 压缩相关的函数以及位处理函数的定义（函数体）；

huffman.h Huffman 压缩相关的函数、常量的声明（位处理的调用 lz77.h）；

huffman.c Huffman 压缩相关的函数的定义（函数体）；

main.c 主函数体，包含（多文件压缩相关的）文件操作，以及一些与功能相关的宏（编译为压缩程序还是解压缩程序）定义。

编译使用 GCC，命令为：

$$gcc\ -o\ main\ main.c\ lz77.c\ huffman.c$$

其中 main 可替换为 myzip 或者 myunzip。
文件 main.c 中的宏 ZIP_OR_UNZIP 控制功能，当其为 1 的时候编译为压缩程序 myzip，为 0 的时候编译为解压缩程序 myunzip。