Multimedia Technologies

基于LZW算法的中文文本压缩和优化

1.Members

序号	学号	专业班级	姓名	性别	分工
1	3190104534	计科1903	庞懿非	男	资料研究、提出问题、寻找已有的解决方案；代码复现与优化、结果分析、报告撰写，以及问题总结思考。

2.Project Introduction

（1）选题

文本压缩有很多可以选择的编码方式，比如以Huffman编码为代表的变长编码（**熵编码**的重要部分）、**算数编码**、**词典编码**等，每种编码方式又对应着许多不同的实现算法。同时，可以压缩的文本其实也有很多不同的类型，我们最常见的就是英文文本、中文文本、中英文混合文本以及结构化语言文本这4种类型。对文本压缩进行深入全面透彻的研究是非常困难复杂的。我在查询资料的时候发现，任何一种常用的算法都不是基于中文文本提出的（大多数基于英文），后来的研究也没有很多针对中文文本的算法优化，这可能是由于使用这些算法也可以对中文文本进行压缩，同时为了使得算法的通用性和普适性更强，统一使用了这些基于英文语法结构提出的算法。随着计算机底层硬件的提升，压缩文本的小幅提升已经不是那么重要；更多的多媒体设备出现，使得使用一个统一的工具对视频图片等大量数据进行压缩成为主流趋势。

即使如此，由于中国人对汉字特有的情感以及对现有压缩效果的不满，我仍然认为对中文文本进行压缩算法的优化和适配是极其重要的。以**LZW**算法为例，这一在LZ78算法下延伸出的词典编码算法被认为有着良好的压缩性能。但是，很明显我们可以发现，二者对中文和英文文本的压缩比差异还是比较大的，对英文文本的压缩比通常能突破2，但是中文文本平均压缩比只能达到1.6左右，这在大量数据的压缩下会有非常显著的差距。同时，根据论文的分析，使用Cover方法估算汉字信息熵 [1]实际值在 4.1 bit 左右，这意味着我们有非常大的压缩空间。

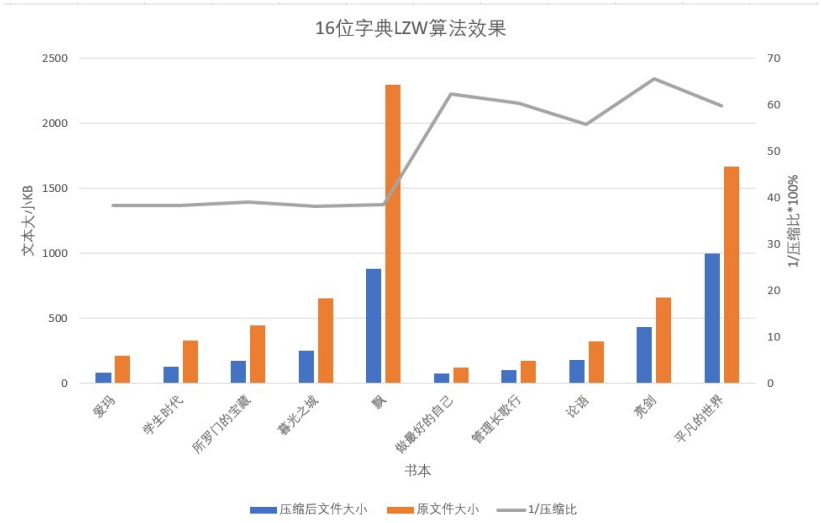

Fig 1. 16位LZW算法压缩效果示意图（蓝色和橙色分别代表压缩前后的文本大小，折线为1/压缩比*100%）

在查找到的论文 [2]的理论支撑下，我决定通过实践的方式，复现论文中提供的**LZW中文压缩优化**的方法并且提出我的认识与思考。

（2）工作简介

在基于LZW算法中文文本压缩的压缩的过程中，我们主要需要做3件工作：

- LZW英文文本压缩算法的实现和优化
  - LZW算法实现以及**比特流输出**
  - LZW扩展自适应编码输出
  - LZW对中文算法进行**字节流处理**
- LZW中文文本适配
  - 选择不同的中文编码字符集加入字典
  - **处理不在字典中的字符的编解码问题**
  - 寻找最好的算法，进行输出再编码进一步压缩
- 将改进后的算法进行性能分析，并与流行压缩工具进行对比

（3）开发环境即系统运行要求

我的程序是基于**Java**完成的，使用了jdk 16.0.2版本，在Eclipse下完成，运行时请保存完成的工程结构，其中文本路径使用在工程中的相对路径，运行时在主文件中调节字典大小参数、算法参数和文本路径参数，可完成不同版本的压缩和解压缩的工作（具体文件路径依赖关系请参考工程README文档），没有使用其他的开发包和开源库。

- OS: Windows10 （其他操作系统均可，需要有JVM）
- IDE: Eclipse
- 文件编码格式：输入txt文档使用ANSI 编码下的GBK字符集（默认），如果是UTF-8请另存为ANSI编码进行格式转换

3.Technical Details

（1）理论基础阐述

① LZW算法（Lempel-Ziv-Welch Encoding）

LZW算法是字典编码中的重要算法，是LZ78[3]系列的一个重要的改进，取得了更好的效果。

在LZW算法中，字典由**码字（*Code word*）**以及**前缀（*prefix*）**组成。在压缩编码开始之前，字典就已经填充了相应的前缀，码字顺次增加，不重复。按照文本顺序读取字符，直到新的字符串不在字典中，就输出前缀部分的码字，将新的字符添加在字典中，并且将前缀变成新的字符。整个程序不断地进行循环从而在输出全部的码字的同时不断丰富自己的字典内容。

码字(*Code word*)	前缀(*Prefix*)
1	
…	…

Fig 2. 字典示意图

LZW算法的解码过程与编码类似，需要预先填充字典，并且二者要在程序中构造相同的字典才可以正确编解码。之后对压缩后的码流进行分析，提取出全部的码字，并且相应地将字符串添加到字典中。因为在编码过程中每次码字输出都伴随着新的字符的加入，所以在解码端为了构建相同的字典，在每个码字分解出之后也需要添加相应的串到字典中。

简易的概念流程如下所示：

```
/* compress */
read new_char and add new_char to prefix until prefix + new_char not in the
dictionary
dic.put(prefix)
output(code of prefix)
prefix = new_char
```

```
/* decompress */
analyze code_word from bit stream
prefix = dic.get(code_word)
dic.put(last_word + prefix[0])
last_word = prefix
```

具体的算法部分在后续中会进行讲解分析。

② 字符集合编码概念基础

在读写文件的时候，我们经常注意到，如果不对文件的编码进行指定，很可能会出现乱码。对编码以及字符集的概念进行理解和使用非常重要。这一点在我的工程中也非常重要。

- **ASCII字符集**

  ASCII[4]（American Standard Code for Information Interchange，美国信息互换标准代码）是基于罗马字母表的一套电脑编码系统，它主要用于显示现代英语和其他西欧语言。它是**现今最通用的单字节编码系统**，并等同于国际标准**ISO 646**。

  ASCII字符集实际上只对值为0~127的符号进行编码使用。其中，第0～32号及第127号(共34个)是控制字符或通讯专用字符；第**33～126**号(共**94**个)是字符，48～57号为0～9十个阿拉伯数字；65～90号为26个大写英文字母；97～122号为26个小写英文字母；其余为一些标点符号、运算符号等。

  在使用上，ASCII字符集的最高位通常用来做奇偶检验，并不代表实际的值。

- **ANSI 编码**

  ANSI 编码是一个非常重要的编码格式，下含很多字符集，包括GB2312字符集、GBK字符集、Big5字符集、GB18030字符集等等。在ANSI 编码下通过设定可以使用任意一种字符集。

  - **GB2312字符集**

    GB2312又称为GB2312-80字符集，由中国国家标准总局发布。它所收录的汉字已经覆盖99.75%的使用频率，基本满足了汉字的计算机处理需要。在工程中我们也使用了GB2312中的一些字符作为字典的基础码集.

    GB2312字符集使用**分区处理**的方式（区位码）[5]，其中：

    - 01-09区为特殊符号。
    - 16-55区为一级汉字，按拼音排序。
    - 56-87区为二级汉字，按部首/笔画排序。
    - 10-15区及88-94区则未有编码。

    这些字符（2字节）在GB2312中编码的数值也可以由区位号得出，“高位字节”使用了0xA1-0xF7(把01-87区的区号加上0xA0)，“低位字节”使用了0xA1-0xFE(把01-94加上 0xA0)。具体的字符和区位的对应关系可以查看 GB2312编码表。

- **GBK字符集**

  GBK字符集是GB2312的扩展(K)，GBK1.0收录了21886个符号，它分为汉字区和图形符号区，汉字区包括21003个字符。GBK字符集主要扩展了繁体中文字的支持。

  我的Java工程使用GBK字符集作为基本编码字符集，在具体的实现中也使用GBK字符集去表示在GB2312中不存在的字符从而保证信息的完整性。（不存在的值无法读取，生成String会使得字节值丢失）

- **Unicode字符集**

  Unicode字符集编码[4]是（Universal Multiple-Octet Coded Character Set） 通用多八位编码字符集的简称，支持世界上超过650种语言的国际字符集。

  由于每种语言都制定了自己的字符集，导致最后存在的各种字符集实在太多，在国际交流中要经常转换字符集非常不便。因此，产生了Unicode字符集，它固定使用16 bits(两个字节)来表示一个字符，共可以表示65536个字符 。
  标准的 Unicode 称为UTF-16(UTF:UCS Transformation Format)。后来为了双字节的Unicode能够在现存的处理单字节的系统上正确传输，出现了UTF-8，在UTF-8中，每个汉字用3个字节编码。

- **Windows怎样识别txt文件的编码方式**

  在完成工程中，我发现只通过比特流输出，Windows系统就可以判断我的文件编码方式。在工程中我读取ANSI编码的文件（因为要使用GB2312字符集），在比特流输出的时候txt文档也会自动识别是ANSI编码。如果读取UTF-8编码的文件写入比特流则不会识别成ANSI编码。

  一般我们认为，软件一般采用三种方式来决定文本的字符集和编码[4]:

  1） 检测文件头标识;

  2）提示用户选择;

  3）根据一定的规则猜测 。

  Unicode编码的文件一般都有特殊的文件头标识；ANSI编码的文件没有文件头，可以通过一定的规则进行猜测。

  还有其他的编码类型这里就不进行展开了。

（2）算法描述分析

① LZW原始算法（Lempel-Ziv-Welch Encoding）

LZW原始的算法在教学中已经讲解，我的实现过程完全使用字节流，这里简单展示伪代码。

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

```
/* decompress */
BEGIN
    s = NIL;
    while not EOF
    {
        k = next input code;
        entry = dictionary entry for k;
        if (entry == NULL & s != NULL)
            entry = s + s[0];
        output entry;
        if (s != NIL)
            add string s + entry[0] to dictionary with a new code;
        s = entry;
    }
END
```

我在实现传统LZW算法的时候进行了简单的改进，采用了**自适应的扩位编码**输出码字。通俗上来讲，就是编码输出的长度自适应于当前字典的长度，编码输出的长度随字典长度增大而增大。相对于传统的定长编码，这样显然可以在一定程度上减少输出编码的长度。后续的压缩比分析也是基于这个方法进行的。

同时，字典的大小可以使用参数进行调整，字典大小是指程序输出代码的最大长度。

② 比特流输出

由于Java没有比特流处理的机制，所以只能通过字节处理来实现比特流处理的效果。在字节处理中，使用int来存储要输出的8 bit数值，同时使用一个int表示已经存入数字的比特数。大致实现如下：

```
int buffer = 0;
int n = 0;
while(input this_bit)
begin
    n+=1;
    buffer = (buffer << 1 ) | this_bit;
    if(n == 8)
        output buffer; n = 0;
end
```

比特流输出在编码端和解码端都非常重要，在编码端需要将码字以比特的方式输出；同样，在解码端，由于汉字通常是2个字节组成，在字节流处理中需要将2个字节按照比特分别写入，单独写字节可能会在字符串转换字节的过程中因为编码问题而失效（比如可能出现某字节不识别，即不在编码范围从而变成`0x3F（问号?）`，因为128-255的字节在GBK中是非法的，在字符串转化过程中经常会出现问题）。

③ 码字分析

在上述解压的描述中，码字分析作为第一步。在不需要对字典进行重建的情况下，码字分析可以独立于解压步骤。也就是说，可以一次性分析出所有码字的值。基本伪代码如下所示：

```
while(not EOF)
begin
    length_code = dic.getoutlength()
    read length_code bits from the file
    array.add(code word)
    dic.size++
end
```

这里关键的部分是`getoutlength`得到下一个码字的码长，通常我们可以通过字典的大小来判断，这一点不论是在原始的LZW算法、自适应扩展的LZW算法、还是码值再编码的算法中都可以实现，只需要进行一些if else判断即可，逻辑相对清晰，和编码部分的输出码字长度相对应。

④读取中文字符处理的LZW算法

读取中文字符，并且按照字符处理，就需要在基础码集中添加对应的中文符号。我选择了GB2312字符集中的符号进行添加。

在这里，有一个至关重要的问题：**无论我们选择哪些符号到字典中作为基础码集，总有一些基础码集以外的符号出现，这些符号我们应该如何去编码以及解码**：

- 将这个符号添加到字典中继续传输

  使用这种方法，我们就遇到了一个难以避免的问题，解码端如何知道这一点？这可能还需要存储一些字典内容到压缩文件中去，抛开复杂度不谈，将这些低频词加入到字典中本身就不是一件高效的事情，所以这个方法暂且不予适用。

- 将字符拆分成字节进行编解码

  这一点是可行的，因为我在预先填充的时候填入了所有0-255字节值的符号，所以按照字节编码是可以的。但是同时，我们不仅要字符编码，还要字节编码，在算法上我们就需要一些特别的设计，在改进算法中，我设计的相对要复杂一些，概念伪代码如下：

```
/* compress */
Dictionary pre_fill
read next prefix as char
while(EOF)
Begin
    read C from file if tmp's bytes are all read
    if C is not in Dictionary
        byte []tmp = C.getBytes()
        read C from tmp until all bytes are read (not from file)
    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
```

压缩过程中使用如上的方法，就可以使得所有字符（或者字符的字节值）正确压缩，为之后的解压奠定了基础。

在解压端，解析码值的部分完全不受影响，只有输出的部分需要改变，因为原先所有的字符`char`都是1字节的方式进入到输出端，现在变成了1字节或者2字节。在Java中，`char`是unicode编码，一般用两个字节表示汉字。经过尝试，我们发现，使用`char`传输单字节的数据时，高位字节的值为0，这一点对于工程来说至关重要，因为我们需要按照比特流输出（尤其是有些字符拆分成了比特进行表示），对char的内容的判断及输出方式的调整都尤为关键。

```
get char from prefix
if char & 0xff00 == 0
    binary.output(char & 0xff)
else
    binary.output((char & 0xff00) >> 8 )
    binary.output(char & 0xff)
```

⑤ 基础码集的选择

中文基础码集在GB2312字符集上进行选择，我选择了4种模式（在主文件 *LZW_main.java* 中进行了标注）：

- LZW_CH_2

  使用了GB2312中的所有字符（包括汉字字符以及字符区）。

- LZW_CH_3

  使用了GB2312中的一级汉字。

- LZW_CH_4

  使用了GB2312中的一级汉字和全部字符区。

- LZW_CH_5

  使用了GB2312中的一级汉字以及字符区的1区3区。

后续在此基础上对各种算法进行了压缩比分析和进一步优化。

⑥ 码值再编码

在前边的实验中，我们发现字典大小变大可能导致压缩比下降，这往往是由于每个码值输出的长度增加，而短码值输出频率较高。针对这种现象，我们对码值再次编码，使得码值输出有更高的自适应性，短码值输出更短。基本原理很简单，如下所示：

```
/* encode */
if dic.length < 14
    output.length = dic.length
else if dic.length = 14
    if code.length <= 12
        output.length = 13, first_bit = 0;
    else
        output.length = 15, first_bit = 1;
else if dic.length < 18
    if code.length <= dic.length - 3
        output.length = dic.length - 2, first_bit = 0;
    else
        output.length = dic.length + 1, first_bit = 1;
else if dic.length < 20
    if code.length <= dic.length - 4
        output.length = dic.length - 3, first_bit = 0;
    else
        output.length = dic.length + 1, first_bit = 1;
```

解码端按照相同的思路，从结果倒推可以得到码长并且解析出码值，这一步再编码对压缩比又进行了小小的提升，这也是我工程中最后一个优化部分。

（3）程序技术细节

① 使用的库中的内容

在工程中，我基本没有使用开源库。相对重要的就是`HashMap`的数据结构用于充当字典进行快速查找以及`byte String char`的转换。实际上尽管Java在创建、使用这些基础类比如`String`的时候非常方便；但是在处理比特流的数据、甚至是字节流的数据都是相对复杂的，远远不如C简单便捷。

```
byte [] bytes = new String("...").getbytes();
byte s = (byte)(char_1 & 0xff) ;
char_1 = String.charAt();
String.getChars(0, String.length(), chars_array, 0);
new String (bytes, "GBK")
new String (chars_array, "GBK")
```

```
HashMap<String, String> map = new HashMap<>();
map.put("zhang", "31"); //用于向map中添加元素
map.get("String");  //用于得到value
map.containsKey("String")   //用于查看map中是否有改key值
map.size(); //查看map中不同key值的个数
```

② 我的重要函数以及函数依赖关系

所有类以及重要方法绘制成ER图如下所示：

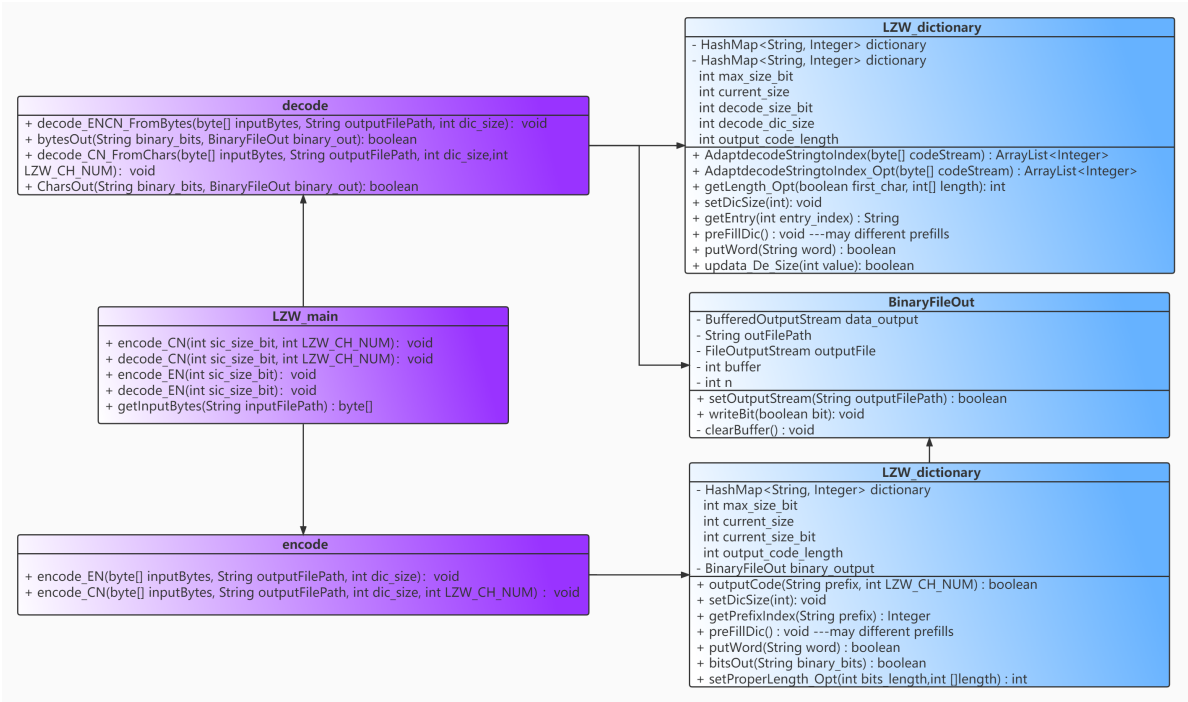

Fig 3. 主要类和功能ER图

我将按照编码和解码分成两部分介绍，解析重点函数功能。在源代码中我已经进行了比较详细的注释。

编码：

- **LZW_main**类的主函数指定参数，获取输入的字节流，调用相应的编码函数

方法	功能
`getInputBytes()`	从指定文件读取字节流信息
`encode_EN()`	使用原始LZW算法进行字节流压缩
`encode_CN()`	使用中文文本改进的LZW算法进行字符流压缩

- **encode**类两个主要方法，分别被上述两种不同的LZW算法调用

方法	功能
`encode_EN()`	使用原始LZW算法，每个字节进行读取，然后执行LZW算法的基本流程，被LZW_main的`encode_EN()`方法调用。
`decode_CN()`	使用中文文本改进的LZW算法，识别汉字字符并且使用加入了部分GB2312字符的字典进行字符流编码，对于不在字典中的汉字字符仍然使用字节流进行编码。

- **LZW_dictionary**类作为编码字典，其中存储了字典变量以及对字典进行设置、添加、修改等的方法。

变量	含义
`HashMap<String, Integer> dictionary`	按照*"prefix，code word"*的方式存储字典信息
`int max_size_bit`	字典最大编码长度
`int current_size`	字典当前存储"prefix"数量
`int current_size_bit`	字典当前使用的最大编码长度
`BinaryFileOut binary_output`	二进制输出类，在之后介绍

方法	功能
`bitsOut()` `bitsOut_Opt()`	实现二进制流输出，调用`binary_output`的方法；Opt版本在码字再编码时使用
`setProperLength_Opt()`	在码字再编码时获取算法规定的码字输出长度
`outputCode()`	根据参数调用`bitsOut()` `bitsOut_Opt()`，将一个前缀（*prefix*）的对应码字（*code word*）输出
`putWord()`	在字典未满的情况下添加新的*"prefix"*到字典中
`preFillDic()`	预先填充字典方法，这类方法有几个具体的填充对象，分别是：0-255字节值、GB2312的一级汉字、二级汉字、字符区以及1区和3区字符区。预填充的时候根据具体的参数选择填充的内容。

解码：

- **LZW_main**类的主函数指定参数，获取压缩文件输入的字节流，调用相应的解码函数

方法	功能
`decode_EN()`	使用原始LZW算法字节流解压缩
`decode_CN()`	解压缩过程基本与上述方法类似，额外使用了基于中文字符的输出方法

- **decode**类4个主要方法，分别被上述两种不同的LZW算法调用

方法	功能
`decode_ENCN_FromBytes()`	使用原始LZW算法，分析出*"code word"*，然后对每个破译出的*"prefix"*调用`bytesOut()`字节输出。
`bytesOut()`	将字符串分解成`char`输出末8位
`decode_CN_FromChars()`	功能上与`decode_ENCN_FromBytes()`几乎相同，额外需要填充指定的中文字符，在输出时调用`CharsOut()`进行字符或者字节输出
`CharsOut()`	默认输出的字符串由两字节的`char`组成并且输出两字节的数据，如果判断出该`char`是单字节则只输出末八位

- **LZW_de_dictionary**类作为编码字典，其中存储了字典变量以及对字典进行设置、添加、修改等的方法，与**LZW_dictionary**类似，所以这里只介绍他们的不同点。二者最大的不同在于解码字典独立地进行码字提取，一次性获取所有码字。

变量	含义
`HashMap<Integer,String> de_dictionary`	按照*"code word，prefix"*的方式存储字典信息
`HashMap<String, Integer> dictionary`	按照*"prefix，code word"*的方式存储字典信息。算法本身是不需要使用这个字典的，但是在添加中文字符的时候为了防止有些串重复导致某个码字浪费从而引发错误，所以使用了这个字典来保证相同字符只能在首次被添加
`int decode_dic_size`	字典在码字提取的过程中字典假想大小
`int decode_size_bit`	字典在码字提取的过程中字典假想大小的编码长度

方法	功能
`AdaptdecodeStringtoIndex()` `AdaptdecodeStringtoIndex_Opt()`	这个方法是解码字典的核心，用于码字分析提取，根据当前码字的输出大小提取；并且每次提取一个码字对假象字典大小增加1（这一点在算法上是一致的）。其Opt版本是使用下面的方法获取码字的长度，与编码时码字再编码相对应
`getLength_Opt()`	在解码端获得编码时码字再编码时的码字长度

二进制输出：

- **BinaryFileOut**类用于二进制输出，具体实现方式如下：

变量	含义
`BufferedOutputStream data_output`	输出流，在文件输出流的基础上建立
`int buffer`	存储这个字节的具体值
`int n`	存储这个字节值中已有多少位数据写入

方法	功能
`writeBit()`	写当前bit到`buffer`中，`n`自增1，如果`n==8`，调用下面的方法
`clearBuffer()`	将`buffer`的内容写到输出流并且将`buffer`和`n`归零

4.Experiment Results

（1）传统自适应扩位编码LZW算法对英文和中文数据的压缩比分析

这里使用和原论文[2]相同的中英文小说进行压缩，同样也是用了**压缩率**（1/压缩比）与原论文进行对照，使用**压缩比**更直观的表示压缩效果。

算法名称为LZW1_字典最大码字长度，如下依次为12、14、16、19。

Table 1. 字典大小对英文文本压缩的影响

英文小说名称/压缩效果	爱玛	学生时代	飘	所罗门王的宝藏	暮光之城
小说文本大小（字节）	287,034	589,824	2,376,837	462857	4019328
LZW1_12 压缩率	45.49%	48.91%	47.78%	48.00%	54.35%
LZW1_12 压缩比	2.20	2.04	2.09	2.08	1.84
LZW1_14 压缩率	39.76%	44.05%	41.89%	42.26%	50.34%
LZW1_14 压缩比	2.52	2.27	2.39	2.37	1.99
LZW1_16 压缩率	38.12%	40.30%	37.93%	38.71%	47.06%
LZW1_16 压缩比	2.62	2.48	2.64	2.58	2.13
LZW1_19 压缩率	38.12%	40.07%	35.69%	38.97%	35.12%
LZW1_19 压缩比	2.62	2.50	2.80	2.57	2.85

Table 2. 字典大小对中文文本压缩的影响

中文小说名称/压缩效果	做最好的自己	管理长歌行	论语	亮剑	平凡的世界
小说文本大小（字节）	126285	181828	50777	739737	1617890
LZW1_12 压缩率	74.33%	73.56%	60.36%	75.57%	79.46%
LZW1_12 压缩比	1.35	1.36	1.66	1.32	1.26
LZW1_14 压缩率	65.29%	64.89%	56.48%	67.98%	67.75%
LZW1_14 压缩比	1.53	1.54	1.77	1.47	1.48
LZW1_16 压缩率	62.95%	59.56%	56.69%	61.86%	61.39%
LZW1_16 压缩比	1.59	1.68	1.76	1.62	1.63
LZW1_19 压缩率	62.15%	59.56%	56.69%	59.47%	57.23%
LZW1_19 压缩比	1.61	1.68	1.76	1.68	1.75

结合表中的数据，对压缩比绘制折线图比较，如下所示：

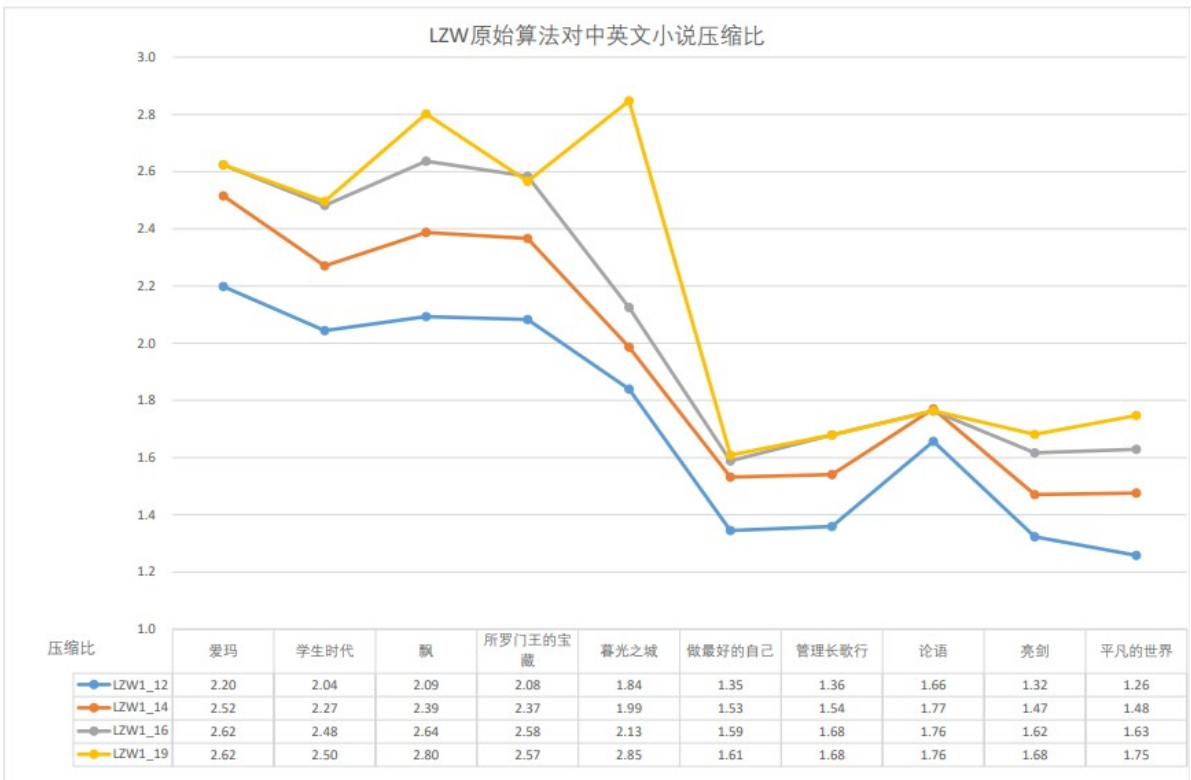

Fig 4. LZW原始算法对中英文小说压缩比折线图

很明显，我们可以看出，设定字典的编码长度增加，压缩比一般会增加；这也是因为字典越大，字典中的长字符串也会越多，字典对文本的自适应性就越强，其压缩效果也会越好，即可以用一个码字表示更多的字符。同时，我们也发现，中文文本压缩比要明显低于英文文本，并且浮动在1.6左右，这样的压缩效果显然是我们不能满足的。这一点的原因主要是，LZW算法是基于英语语系发明的，压缩过程中利用了英文中的短语、符号以及语法结构，与英文有较高的适配度；然而，中文文本不具备英语语系的性质，所以压缩效果要相对差一些。

（2）中文文本下改进的LZW算法对中文数据的压缩比分析

在LZW算法基础上，我们使用改进LZW算法使得其可以无差错的处理中文字符；并且添加了相应的GB2312字符集中的汉字和字符，分别形成了LZW_CH_2、LZW_CH_3、LZW_CH_4、LZW_CH_5一共4个算法，其选择的基础码集在第三章的基础码集部分选择中已经介绍。同样，为了参考原论文[2]中相关数据，我们仍然使用压缩率和压缩比展示这些算法的压缩性能。由于添加中文字符会使得字典大于在4000-7000个基础词汇，所以我们统一使用了19位的字典进行编解码。

Table 3. 改进的LZW算法对中文文本的压缩效果

小说名/压缩效果	做最好的自己	管理长歌行	围城	巴黎圣母院	亮剑	安徒生童话	三国演义	平均
小说原大小（字节）	126285	181828	431595	511750	739737	1774668	1199820	709383
LZW_CH2 压缩率	53.20%	50.96%	54.18%	52.62%	51.84%	45.88%	52.52%	51.60%
LZW_CH2 压缩比	1.88	1.96	1.85	1.90	1.93	2.18	1.90	1.94
LZW_CH3 压缩率	53.30%	51.56%	55.24%	53.57%	52.65%	46.65%	54.06%	52.43%
LZW_CH3 压缩比	1.88	1.94	1.81	1.87	1.90	2.14	1.85	1.91
LZW_CH4压缩率	52.43%	50.52%	54.11%	52.57%	51.75%	45.86%	52.68%	51.42%
LZW_CH4 压缩比	1.91	1.98	1.85	1.90	1.93	2.18	1.90	1.94
LZW_CH5 压缩率	52.30%	50.42%	54.07%	52.53%	51.71%	45.85%	52.66%	51.36%
LZW_CH5 压缩比	1.91	1.98	1.85	1.90	1.93	2.18	1.90	1.95

在上述表格中，可以很明显的看到，不论是什么文本，LZW_CH5的压缩效果总要略胜一筹。其实，从添加的字符集也可以看出，一级汉字具有非常高的频率，其次是1区和3区的符号区，这两个区几乎包含了常见的所有汉字符号，包括逗号、句号、引号、书名号等等，其他的字符区符号出现的频率非常低；同时，二级汉字数量接近3000个，但是出现频率较低，添加到基础码集中效率比较低；所以LZW_CH5具有良好的压缩比。

（3）基于LZW_CH5算法进行码字再编码算法LZW_CH6

基于LZW_CH5的良好性质，我们使用码字再编码的方法进一步增大压缩比，具体原理在第三章码值再编码部分已经介绍过。

Table 4. LZW_CH6算法对比LZW_CH5算法对中文文本的压缩效果

小说名/压缩效果	做最好的自己	管理长歌行	围城	巴黎圣母院	亮剑	安徒生童话	三国演义	平均
小说原大小（字节）	126285	181828	431595	511750	739737	1774668	1199820	709383
LZW_CH5 压缩比	1.91	1.98	1.85	1.90	1.93	2.18	1.90	1.95
LZW_CH6 压缩比	1.93	2.02	1.92	1.97	2.01	2.24	1.97	2.01

我们看到，LZW_CH6获得了更佳的压缩比。通过码字再编码，我们获得了更好的压缩性质。

不仅如此，相较于使用19位字典自适应扩位编码的LZW_19算法压缩比只能达到1.6-1.7，LZW_CH6的压缩比已经可以达到2左右，这也证明我们的LZW算法改进是卓有成效的。

当然，我们此时也使用Zip工具和WinRAR工具进行了压缩，与LZW_CH6进行对比，如下所示：

Table 5. LZW_CH6算法与Zip和WinRAR对中文文本的压缩效果

小说名/压缩效果	做最好的自己	管理长歌行	围城	巴黎圣母院	亮剑	安徒生童话	三国演义	平均
小说原大小（字节）	126285	181828	431595	511750	739737	1774668	1199820	709383
LZW_CH6	1.93	2.02	1.92	1.97	2.01	2.24	1.97	2.01
ZIP	1.95	2.00	1.80	1.83	1.84	1.99	1.83	1.89
WinRAR	1.98	2.06	1.86	1.95	2.00	2.22	2.00	2.01

可以看到，我改进的LZW_CH6算法在压缩比性能上与常用的压缩工具性能不相上下，甚至要略好。尽管这些压缩工具使用的范围远远不限于文本压缩，但是我仍然认为，只使用单一的压缩算法LZW进行修改，达到压缩比为2仍然是一件非常有价值的事情。

（4）总结与说明

① 我的创新点与效果

我实现的LZW中文文本下的改进算法主要的方法论都是在**《中文文本压缩的 LZW 算法》**[2]论文中提到的。但是，该论文中只提到了基本的算法思想，即填充汉字字符集按照汉字字符流编解码；并没有提出实际实现中如何处理复杂多变的文本的方法。因为**在英文文本处理中，基础字符最多只有0-255一共256个字符，并不存在特殊情况，任意一个字符一定都属于这个范围。**这个假设的成立对于LZW算法来说是非常必要的，没有异常情况的出现让英文字符流，即字节流，变得非常简单明了。但是，在中文文本下的LZW算法中，这一假设并不能满足。汉字的数量非常多，我们不可能要将全部的汉字字符添加到基础码集中。所以，我们就要处理这些"不认识"的字符。这一问题在论文中并未提到，在我初次看论文的时候就觉得很困惑。在实现的过程中，**我认为可以将这些不在基础码集中的字符切分成字节再进行处理**，这样就可以很好的解决前面提到的问题，因为我只要提前把字节值0-255存入字典，任何的字符都可以被拆分成字节传输。但是同样，这样的拆分在编码转换的时候很容易出问题，因为128-255的字节值不在GBK编码中，在使用字节值转换字符串之后，这个单字节的值就会丢失变成0x3F（即英文?），处理过程也是相当复杂。

在经历了相当多各种各样的错误之后，我总结了一些寻找可能存在的问题的方法：

- 检查压缩前和解压后文件的大小是否完全相同，这个是最基础的要求，也是最简单的寻找方法。
- 检查文本中是否含有英文"?"，一般中文小说不会出现这个符号，但是编码错误或异常时会生成这个符号。在整体框架正确的情况下，具体输出中的差异可能导致程序出现异常。很多情况下即使两个文件大小完全相等，但是有一些符号的编码改变了就会导致整段的错误。
- 检查源代码中每个字节数组转字符串以及字符串转字节数组、`char`类型值要经过转换才可以输出，因为`char`是unicode编码，这一点在编码标准处也提到了。
- 对一些不在字符集中的符号进行debug测试来判断编解码的正确性。

在调试和实现的过程中，我使用了VSCode的插件**hexdump**，帮助我查看文件的二进制流，在实现算法的过程中作用巨大。

从结果出发，我使用了论文中的算法思路，并且使用相同的小说测试样例做测试（因为原论文代码和数据都无从获得，所以在样例文本大小上存在一些偏差），获得了明显更好的效果，平均上有0.05-0.1的压缩比的提升，我认为是我与原论文作者在具体实现上的差异导致，这也更加证明我的实现方法有着更好的压缩比。

② 反思与改进

在对大文本文件压缩的时候，LZW_CH6算法的压缩和解压时间都会相对增长，我虽然没有对压缩解压缩的时间进行分析，但还是记录了一些运行时间数据。在小文本（比如1000KB以内）的压缩和解压缩过程几种算法相差不大，LZW_CH5略快；LZW_CH6在大文本的压缩和解压中相对上会使用更长的时间，后续可能需要在算法复杂度上进一步优化从而达到更好的效果。

尽管改进后的LZW_CH6压缩比在2左右，但是从理论上来讲，这个压缩比仍然有上升的空间。我认为，对于压缩后的二进制码流文件，仍然可以使用字节流的文本压缩，得到更小的压缩比。这一点在实现上并不难做，但是我尝试后发现再次使用LZW算法压缩反而会使得压缩后的文件比原先的文件还大。在语法结构丢失之后，LZW算法几乎失效。我们也可以使用其他的算法，比如Huffman编码对编码后的文档再编码进一步压缩，很有可能进一步提升压缩比。

③ 个人声明

本工程完全由我个人实现，没有参考任何现有代码，算法思想来源于论文**《中文文本压缩的 LZW 算法》**[2]，但并没有完全实现其中所有的算法，对应的算法标号也不完全相同，没有借鉴原作者的有关实现。在中文文本压缩领域的论文并不多，关注LZW算法的中文文本改进算法在国际期刊上屈指可数，在国内这篇文章提出的思想相对上最有价值，对我非常有启发。

参考文献

[1] 徐秉铮 吴立忠 Victor K. Wei，中文文本压缩的算法， 华南理工大学学报，1989,17（3）

[2] 陈庆辉 陈小松 韩德良，中文文本压缩的LZW算法，计算机工程与应用，2014，50（3）：112-116

[3] LZ78算法的原理和实现

[4] 常用的编码简介及windows下.txt编码识别方法

[5] GB2312字符集百度百科

[6] 王忠效，汉语文本压缩研究及其应用，中文信息学报，1996,11（3）

[7] 中文字符集编码查询