

《信息论与编码》大作业

# Huffman 编码实现文件压缩 及压缩性能的比较与讨论

小组成员学号：

517021910722      517021910654

517021911065      517021910754

2019 年 11 月 17 日

## 摘要

我们使用 *python* 语言实现了一个编解码软件，其具有 *huffman* 编码与解码、*LZ* 编码与解码的功能，并且具有较友好的 UI 界面。在本报告中，我们主要使用 *Huffman* 编码来进行对文件的压缩实验，其编码算法和解码算法时间复杂度都为  $O(n)$ ，对提供的两个文件都能正确编码与译码，其中对提供的 *.txt* 文件的压缩比能达到 1.115，对提供的 *.docx* 文件的压缩比为 1.004。我们分析了 *Huffman* 编码对两种文件压缩性能差异的原因，并使用其他的若干原文件进行深入实验来验证了我们的结论。另外，我们使用 *Huffman* 编码对 *.bmp*、*.png* 等图片格式的原文件进行压缩实验并对结果进行了较为深入的分析。之后我们还使用 *LZ* 编码来与 *Huffman* 编码进行对比实验，依据实验结果对两种编码的性能进行了分析。

关键词： Huffman 编码    压缩性能    文件格式    LZ 编码

# 1 编解码基本原理

将文件作为信源，对文件进行压缩是通过信源编码实现的。由于信源符号之间存在分布不均匀的相关性，使得信源存在冗余度，信源编码的主要任务就是减少冗余，提高编码效率。对于计算机中存储的文件，其大多是按二进制形式存储的，即，它们已经对原文件信息进行了二进制编码，并把信息按二进制编码存储。但由于这些文件的自身特性，如考虑通用性或易操作性而采用定长编码或整字节编码，导致其并未完全按照信源符号概率分布进行编码，编码效率较低，进而使存储时有较大冗余。如图 1。

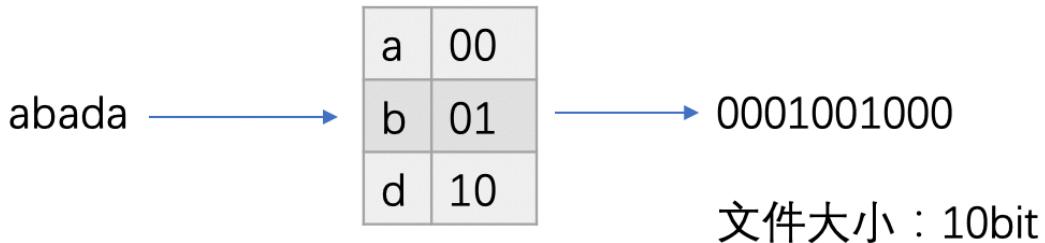


图 1: 定长编码示例

而所谓压缩文件，就是为原文件的信息寻找一种新的信源编码方法，如图 2，提高编码效率，减小冗余，以更短的平均码长存储原文件信息，即原文件信息在无失真或限失真情况下以更小的资源需求进行存储或传输，这样便实现了压缩，虽然这可能使文件失去通用性和易操作性。实现压缩编码的基本途径有两个：使序列中的各个符号尽可能地相互独立，即解除相关性，如预测编码和变换编码；使编码中各个符号出现的概率尽可能地相等，即概率均匀化，如 Huffman 编码、香农编码。

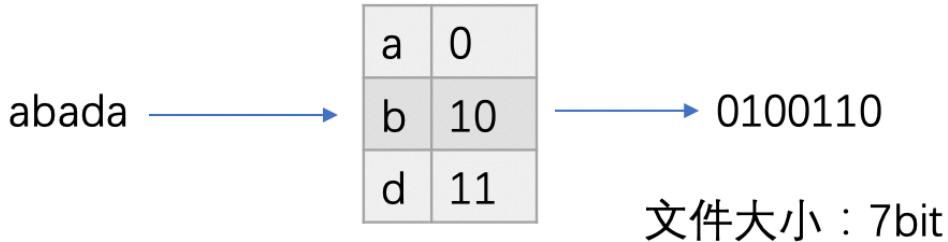


图 2: 变长编码示例

而解码是将以编码形式存储的文件还原回原文件，其过程是编码过程的逆过程。如图 3。对于分组码就是按码表一一对应进行信源符号复原，对于非分组码就是按原编码算法的逆过程进行信源符号复原。

为了能无失真还原文件，我们使用了无失真的 *Huffman* 编码进行文件压缩。以下将简要介绍 *Huffman* 编解码原理。

## 1.1 Huffman 编码

*Huffman* 编码是一种变长分组编码，完全依照各字符出现的概率来构造码字，是一种统计匹配编码。其基本原理是基于二叉树的思想，所有可能的输入符号在 *Huffman* 树上对应一个节点，节点的位置就是该符号的 *Huffman* 编码。为了构造唯一可译码，这些节点都是 *Huffman* 树上的叶子结点。

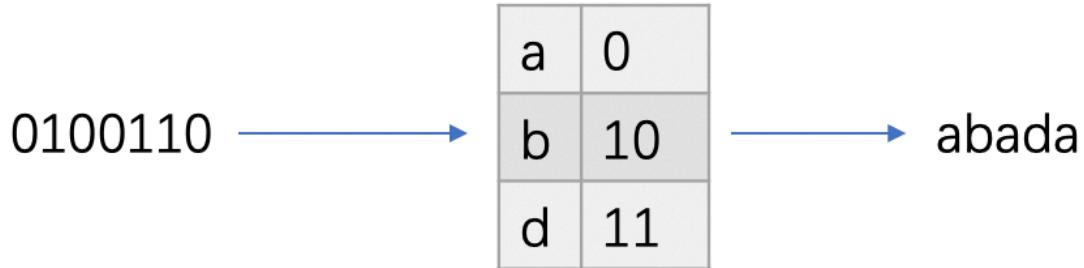


图 3: 译码示例

为了构造 *Huffman* 树，首先须知道所有信源符号的概率。具体编码方法如下：

1. 将信源消息符号按其出现的概率大小依次排列。
2. 取两个概率最小的符号分别配以 0 和 1 两个码元，并将这两个概率相加作为一个新符号的概率，与未分配二进制符号的字母一起重新排队。
3. 对重排后的两个最小符号重复步骤（2）的过程。
4. 不断继续上述过程，直到最后两个符号配以 0 和 1 为止。
5. 从树根开始，向下返回得到各个信源符号所对应的码元序列，即相应的码字。如图 4。

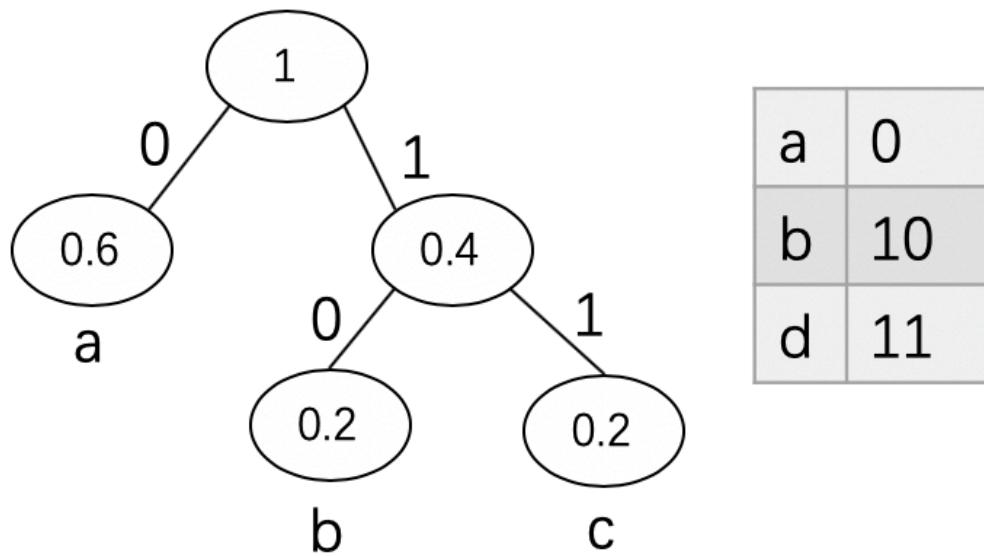


图 4: *Huffman* 码树构建示例

对编码过程的理解和解释：构造 *Huffman* 树的过程就是不断二分码元序列概率，使其逼近信源符号概率分布的过程。*Huffman* 树完成后，概率较大的符号对应的编码路径较短，对应的编码码字码长较短，即有较小自信息量；概率较小的符号对应的编码路径较长，对应的编码码字较长，即有较大自信息量。根据香农辅

助定理，编码时拟合的信源符号概率分布越接近真实信源符号概率分布，编码后的平均码长越小，即冗余度越小，压缩效果越好。

## 1.2 Huffman 编码的译码

Huffman 编码的译码方法，就是按之前编码的码表一一对应把二进制序列还原回去即可，由于 *Huffman* 编码是即时码，所以译码的结果是唯一的。

# 2 算法实现思路

## 2.1 Huffman 编码实现思路

*Huffman* 压缩编码实现流程如图 5 所示。

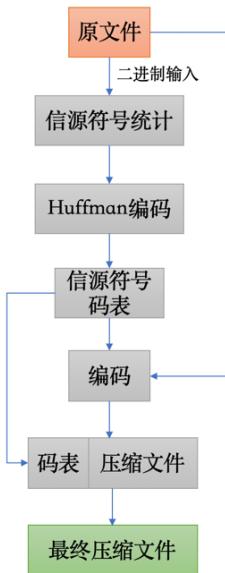


图 5: *Huffman* 编码实现思路

由于原文件在计算机中都是以二进制存储的，且计算机的最小寻址单位是字节，所以我们把八位二进制，也就是一个字节作为一个信源符号，这样任何文件都最多包含 256 种信源符号。在信源符号统计阶段，我们创建了结点类来保存并代表信源符号，并对信源符号的概率进行统计与保存。在 *Huffman* 编码阶段，我们依照 1.1 节 *Huffman* 编码的原理为信源符号构建了 *Huffman* 树，并在之后使用 *Huffman* 树对文件进行编码。在生成最终压缩文件时，为了方便解压缩，我们把码表和压缩文件组合在了一起，并试图用最少的空间来保存码表来减少保存码表对压缩效果的影响。另外在争取空间最小时，我们也试图压缩算法的时间复杂度。经计算，该算法时间需求由信源符号统计阶段和文件编码（压缩）阶段决定，最大时间复杂度为  $O(n)$ ， $n$  为原文件字节数。以下我们将单独说明相对重要的信源符号统计阶段，*Huffman* 编码阶段，文件编码（压缩）阶段，与生成最终文件阶段，并给出各阶段的时间复杂度。

需要说明的是，以下是我们理想中的算法流程，整体上也是按此思路去实现。但由于时间和技术原因，一些细节上并未完全按算法去实施，在各种因素之间取了折衷，所以最后的空间与时间效率可能会有些出入，但都只是常数倍的误差，对最终结果影响不大。

### 2.1.1 信源符号统计

信源符号统计阶段的算法框图如图 6。

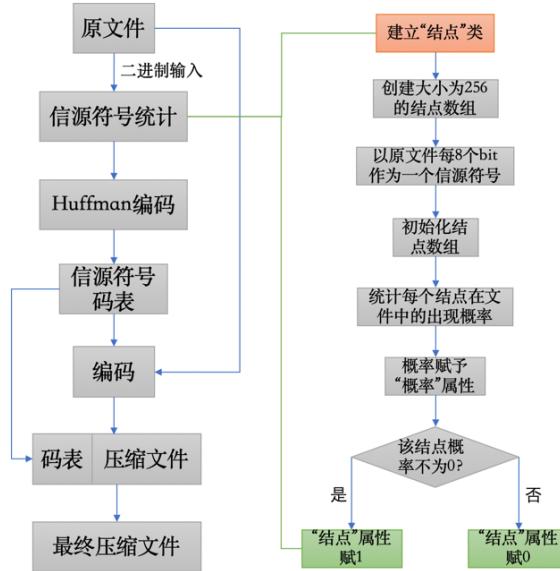


图 6: 信源符号统计流程

正如本节开头所述，我们把每一个字节作为一个信源符号，这样任何文件都最多包含 256 种信源符号。所以在统计之前，我们先建立了一个含 256 个结点的数组，每个结点代表一种信源符号。每个结点所代表的字节（信源符号）的二进制数值等于结点在数组中的序号，如数组中序号为 0 的结点代表字节（信源符号）“00000000”。之后遍历原文件的二进制编码，其每个字节都转化为数值，将其作为数组下标寻址，增加相应结点的“概率”属性。遍历完原文件后就完成了概率统计。此时有些结点的概率可能仍为 0，这是因为原文件中未出现这个字节。令“概率”属性为 0，也就是未在原文件中出现过的信源符号的“结点”属性为“0”，代表其在本次编码中不作为有效结点；而“概率”属性不为 0 的信源符号的“结点”属性赋为“1”，代表该结点在本次编码中为有效结点。将信源符号统计这一步设置得相对复杂是为了在保存码表时所要求的空间更小。且实际上本阶段的最大时间复杂度为  $O(n)$ ， $n$  为原文件字节数，时间大部分仍耗费在对原文件的遍历统计上。

### 2.1.2 Huffman 编码

Huffman 编码阶段的算法框图如图 7。

本阶段使用“结点”属性为“1”的结点，即本次编码的有效结点来构建 *Huffman* 树。首先把有效结点组成根结点数组，即该数组中保存的都是当前各树的根结点。依照“概率”属性进行排序后，每次对数组末尾的两个根结点进行操作。由于算法框图中的描述已经非常详细，且该过程与 1.1 节 *Huffman* 编码原理中所描述的过程相同，所以这里就不加赘述。需要说明的是，结点的“码字”属性保存结点路径的 01 字符串，“码长”属性保存结点路径的长度。本阶段的最大时间复杂度为  $O(K)$ ，即为常数时间复杂度。

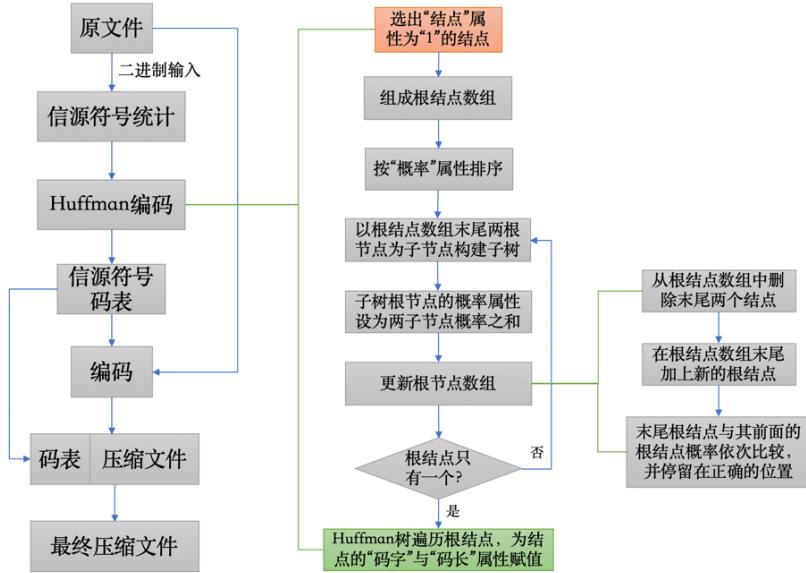


图 7: Huffman 编码具体流程

### 2.1.3 文件编码（压缩）

件编码（压缩）阶段的算法框图如图 8。

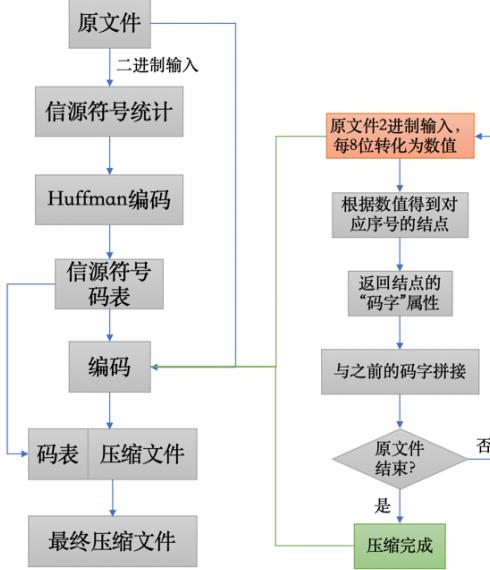


图 8: 文件编码具体流程

由于建立了 256 个结点的结点数组，因此文件编码也变得非常容易，只需要遍历原文件二进制序列，并将每个字节转化为数值，作为数组结点的下标进行寻址，返回结点的“码字”属性进行拼接即可。本阶段的时间复杂度为  $O(n)$ ， $n$  为原文件字节数。

#### 2.1.4 生成最终压缩文件

生成最终压缩文件阶段的算法框图如图 9。

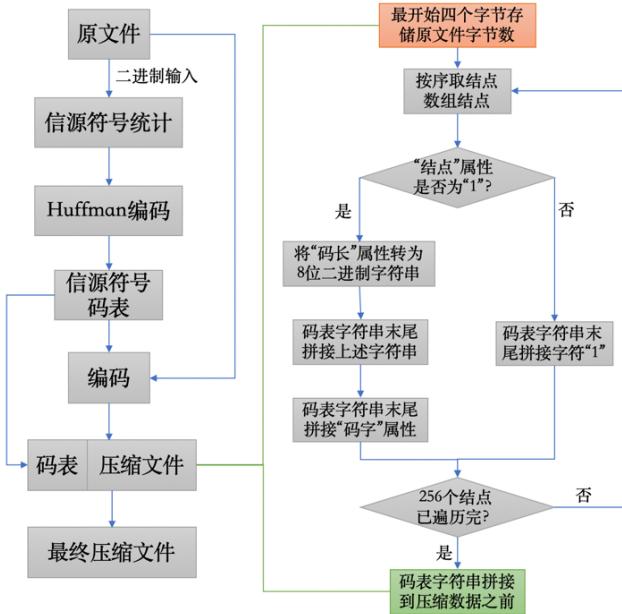


图 9: 生成最终压缩文件具体流程

由于计算机寻址的最小单位是字节，所以任何文件都必须是整字节大小，我们最后的压缩文件也不例外。若不是整字节，计算机会自动在文件的末尾补 0。为了之后能正确地译码，我们必须告知译码程序原文件的字节数以正确译码。因此在封装码表时，我们用最开头四个字节保存原文件的字节数，这样程序理论上可以对 4GB 的文件进行压缩和解压。在这之后按序遍历 256 个结点的结点数组，无效结点，即“结点”属性为“0”的结点只以字符“1”表示；而有效结点，即“结点”属性为“1”的结点以“0” + “码长” + “码字”这三段字符串的拼接来表示。在这里我们进行了一个合理的假设：“不存在码长为 256 的码字”，因为若出现码长为 256 的码字，这必将是一棵非常极端的 *Huffman* 树，其出现概率可忽略不计。这样设计，“码长”字符串就不可能以“1”开头或长度超过 8，译码时若遇到首字符为“1”也就能识别出这是个无效结点，并跳过。这方面的详细说明将在 2.2.1 重构码树时详细介绍。这样设计使封装码表时不需要传递信源符号，而是使用码字出现的顺序代替信源符号的传递，减小了传递码表对压缩效果的影响。本阶段时间复杂度为  $O(K)$ ，是常数时间复杂度。

## 2.2 译码实现思路

*Huffman* 解压缩（译码）实现流程如图 10 所示。

在 *Huffman* 编码时，我们把码表和压缩文件（有效数据）封装为最终压缩文件，解压过程也是对最终压缩文件进行解压。因此解压缩过程大体分为两个阶段：重构码树阶段和解码阶段。在重构码树阶段我们采用 2.1.4 生成最终压缩文件阶段的逆过程进行码表重构和码树重构；在译码阶段我们使用重构出的码树进行译码。解压缩过程的平均时间需求由译码过程决定，平均时间复杂度为  $O(n \log n)$ 。

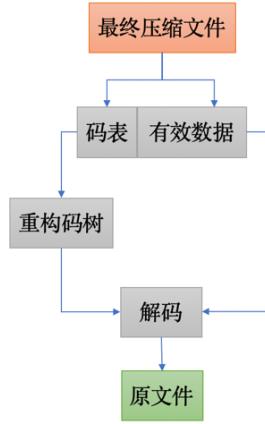


图 10: 译码流程

### 2.2.1 重构码树

重构码树阶段的算法框图如图 11。

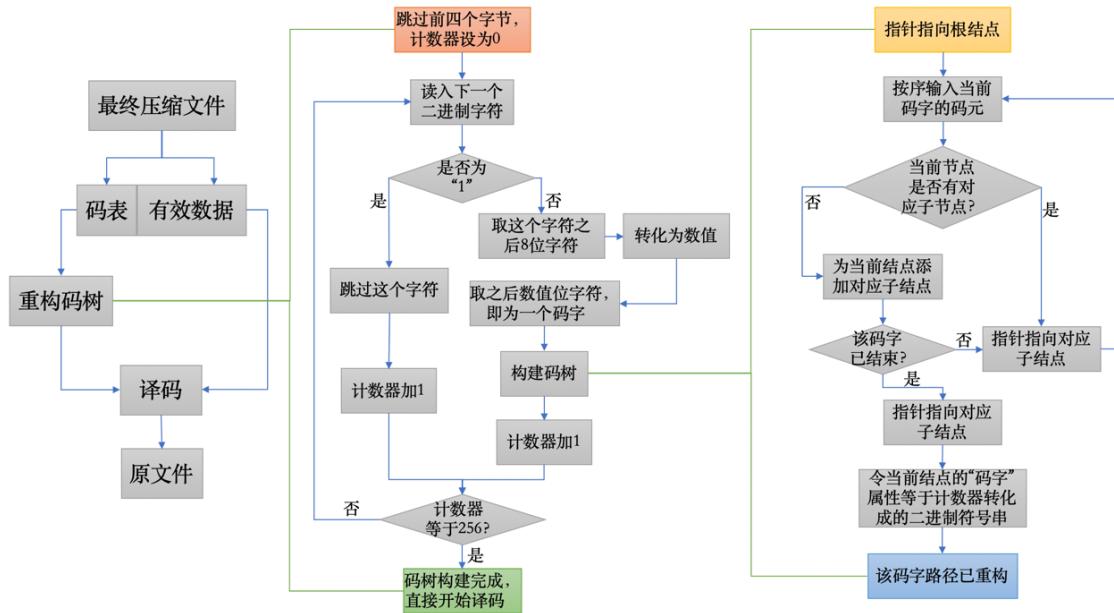


图 11: 重构码树具体流程

码表重构的过程是生成最终压缩文件的逆过程：最开始四个字节存储原文件字节数，处理好后跳过，然后开始重构码表。计数器的值即当前重构的信源符号（字节）的数值。依次读取符号，当遇到首字符为“1”则跳过，说明此处信源符号在本次编码中无效，不需重构。当遇到首字符为“0”则说明接下来是有效信源符号的码长和码字，于是重构该信源符号的码表。之后使用二叉树的生成算法，以码字为路径重构 *Huffman* 树，

框图中已详细描述，这里不加赘述。码树重构阶段的最大时间复杂度为  $O(K)$ ，是常数时间复杂度。

### 2.2.2 解码

解码阶段的算法框图如图 12。

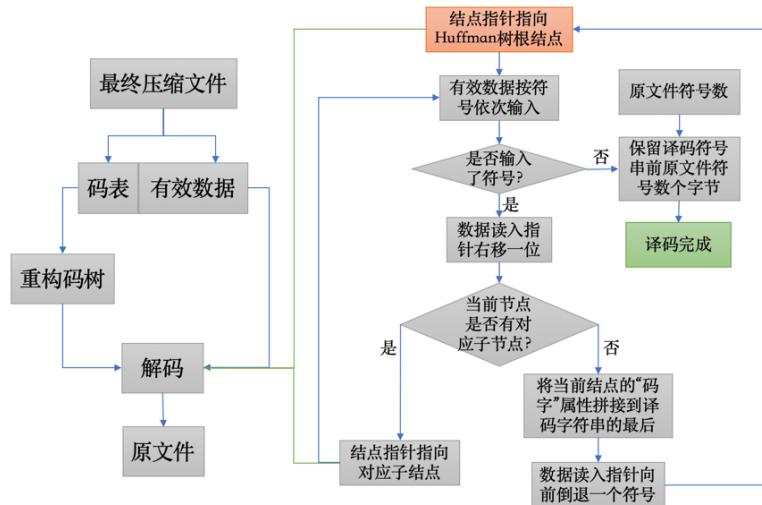


图 12: 解码具体流程

解码过程直接使用重构的 *Huffman* 树进行路径匹配，匹配到根结点则说明得到了一个信源符号（字节），框图中已详细说明，且原理也较简单，这里不加赘述。需要说明的是为了译码正确，最后需要选择译码后符号串的前原文件字节数个字节作为解压缩结果，来消除计算机自动补 0 的影响。本阶段最大时间复杂度为  $O(n)$ ， $n$  为原文件字节数。值得注意的是，若设某一特定文件的译码时间复杂度为  $t(n)$ ，则文件压缩率 = 压缩文件大小 / 原文件大小 =  $t(n)/Kn$ ，因为译码的时间复杂长度刚好正比于压缩文件的总码长。由于编码时直接采用数组下标寻址，而译码时使用二叉树寻址，所以译码的耗时是编码的常数倍。

## 3 编码结果和性能分析

我们使用了 *python* 语言来实现第 2 章的 *Huffman* 编解码算法。在本章及之后章节的所有实验数据都是在 *MacBook Pro (13 - inch, 2016, Four Thunderbolt 3 Ports)* 计算机上，在 *Spyder 3.3.6* 里的 *python 3.7.0* 环境下运行的结果。

### 3.1 编码结果

为了方便之后的说明，在本章及以后的章节中，我们使用以下描述：压缩率 = 原文件大小 / 压缩文件大小；压缩比 = 原文件大小 / 最终压缩文件（与码表一起封装后的压缩文件）大小。

#### 3.1.1 对 .txt 文件进行编解码

原文件打开后展示如图 13。

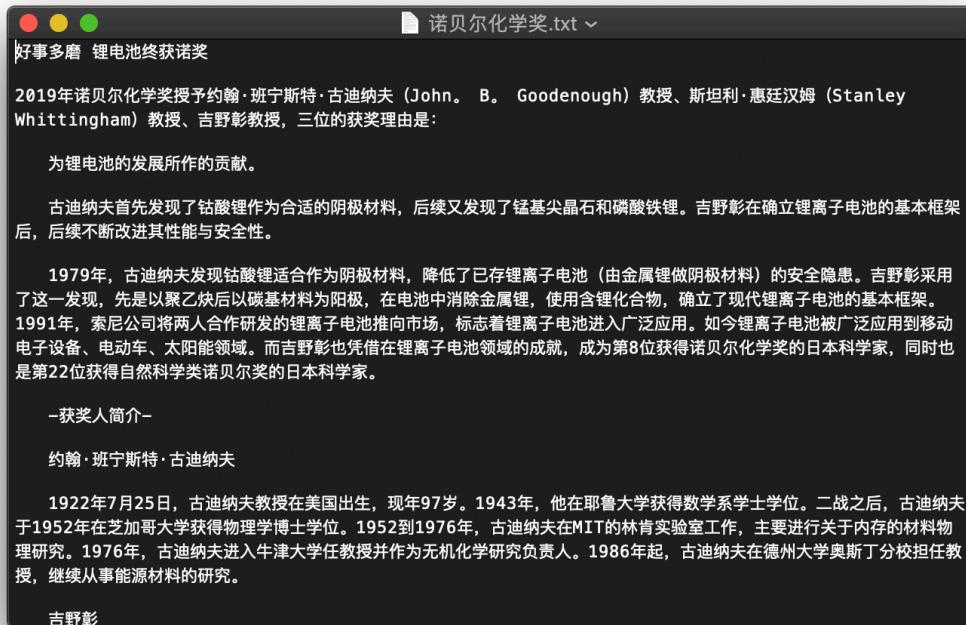


图 13: 原文件展示

对原文件《诺贝尔化学奖 .txt》进行编码的程序运行截图如图 14。

```
In [17]: runfile('/Users/hongxing/Desktop/大三上学期/信息论与编码/大作业/实验文件/info_huffman_lz.py', wdir='/Users/hongxing/Desktop/大三上学期/信息论与编码/大作业/实验文件')
please enter your operation
1:huffman_encode
2:huffman_decode
3:lz_encode
4:lz_decode

1
please input the encoding file path:
/Users/hongxing/Desktop/大三上学期/信息论与编码/大作业/实验文件/原文件/诺贝尔化学奖.txt
please input the target file path:

/Users/hongxing/Desktop/大三上学期/信息论与编码/大作业/实验文件/压缩文件/huf_txt_enc_file/
huf_txt1_Nobel.huf
source file bytes: 4039
data numbers: 411.625
code numbers: 3210.375
encoded file bytes: 3622
encode ratio: 0.794844020797227
compression ratio 0.896756622926467
finish encoding!
Running time: 0.03237600000011298 Seconds

In [18]: |
```

图 14: 编码的程序运行截图

可见《诺贝尔化学奖 .txt》原文件大小为 4039bytes，压缩后封装的码表大小为 411.625bytes，压缩文件

大小为 3210.375bytes，最终压缩文件大小为 3622bytes。压缩率为 1.2581，压缩比为 1.1151。耗时 0.03237s。  
对最终压缩文件《huf\_txt1\_Nobel.huf》进行解码的程序运行截图如图 15。

```
In [18]: runfile('/Users/hongxing/Desktop/大三上学期/信息论与编码/大作业/实验文件/info_huffman_lz.py', wdir='/Users/hongxing/Desktop/大三上学期/信息论与编码/大作业/实验文件')
please enter your operation
1:huffman_encode
2:huffman_decode
3:lz_encode
4:lz_decode

2
please input the decoding file path:
/Users/hongxing/Desktop/大三上学期/信息论与编码/大作业/实验文件/压缩文件/huf_txt_enc_file/
huf_txt1_Nobel.huf
please input the target file path:
/Users/hongxing/Desktop/大三上学期/信息论与编码/大作业/实验文件/解码文件/huf_txt_dec_file/
txt1_dec.txt
encoded file bytes: 3622
decoded file bytes: 4039
finish decoding!
Running time: 0.424999999999545 Seconds

In [19]:
```

图 15: 解码的程序运行截图

耗时 0.4250s。

打开译码后文件，与原文件对比展示如图 16。

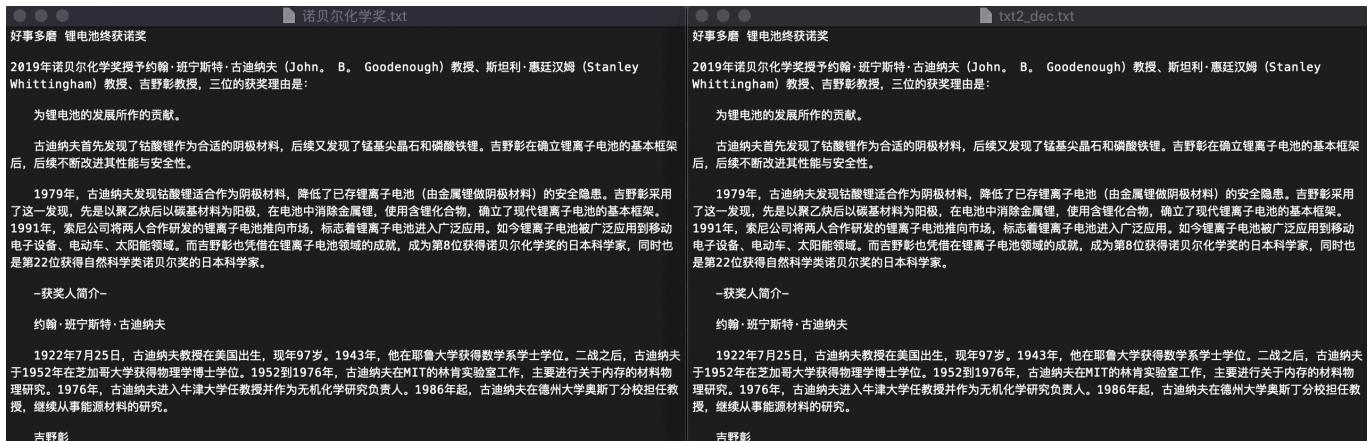


图 16: 译码后文件与原文件对比展示

### 3.1.2 .docx 文件进行编解码

原文件打开后展示如图 17、18。

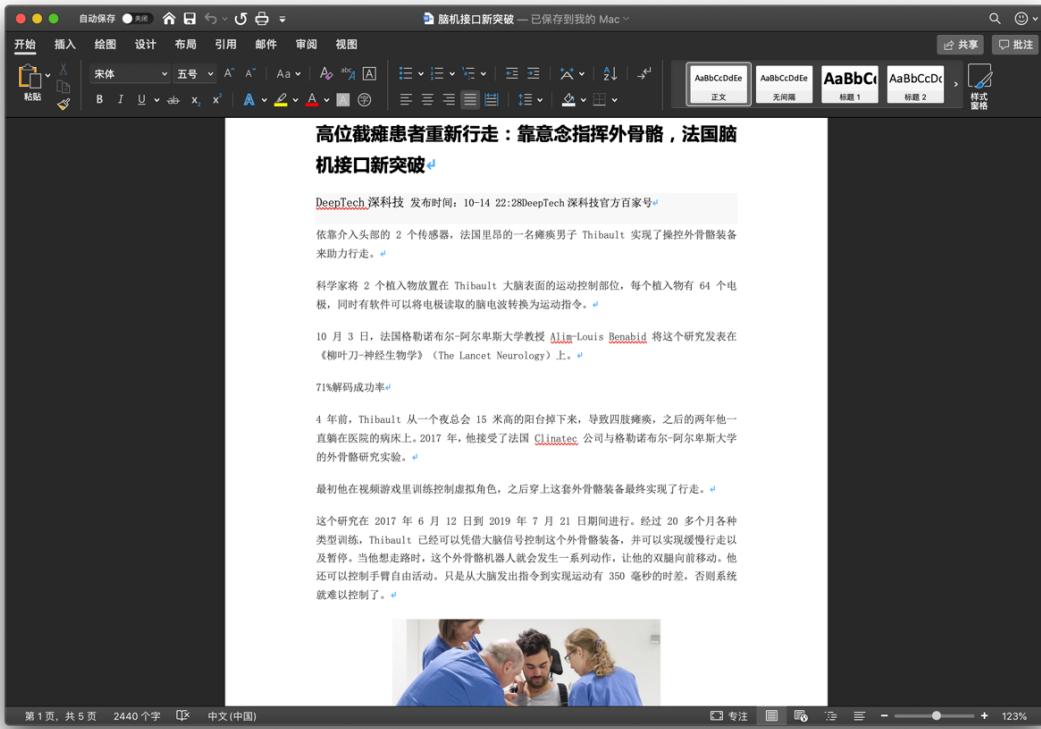


图 17: 原文件展示 1



图 18: 原文件展示 2

对原文件《脑机接口新突破.docx》进行编码的程序运行截图如图 19。

```
In [20]: runfile('/Users/hongxing/Desktop/大三上学期/信息论与编码/大作业/实验文件/info_huffman_lz.py', wdir='/Users/hongxing/Desktop/大三上学期/信息论与编码/大作业/实验文件')
please enter your operation
1:huffman_encode
2:huffman_decode
3:lz_encode
4:lz_decode

1
please input the encoding file path:
/Users/hongxing/Desktop/大三上学期/信息论与编码/大作业/实验文件/原文件/脑机接口新突破.docx
please input the target file path:
/Users/hongxing/Desktop/大三上学期/信息论与编码/大作业/实验文件/压缩文件/huf_docx_enc_file/
huf_docx1_brain.huf
source file bytes: 162678
data numbers: 775.5
code numbers: 161844.5
encoded file bytes: 162620
encode ratio: 0.9948763815635796
compression ratio 0.9996434674633324
finish encoding!
Running time: 0.6310960000000705 Seconds

In [21]: |
```

图 19: 编码的程序运行截图

可见《脑机接口新突破.docx》原文件大小为  $162678\text{bytes}$ , 压缩后封装的码表大小为  $775.5\text{bytes}$ , 压缩文件大小为  $161844.5\text{bytes}$ , 最终压缩文件大小为  $162620\text{bytes}$ 。压缩率为  $1.0052$ , 压缩比为  $1.0004$ 。耗时  $0.6310\text{s}$ 。  
对最终压缩文件《huf\_docx1\_brain.huf》进行解码的程序运行截图如图 20。

```
In [21]: runfile('/Users/hongxing/Desktop/大三上学期/信息论与编码/大作业/实验文件/info_huffman_lz.py', wdir='/Users/hongxing/Desktop/大三上学期/信息论与编码/大作业/实验文件')
please enter your operation
1:huffman_encode
2:huffman_decode
3:lz_encode
4:lz_decode

2
please input the decoding file path:
/Users/hongxing/Desktop/大三上学期/信息论与编码/大作业/实验文件/压缩文件/huf_docx_enc_file/
huf_docx1_brain.huf
please input the target file path:
/Users/hongxing/Desktop/大三上学期/信息论与编码/大作业/实验文件/解码文件/lz_docx_dec_file/
huf_docx_dec.docx
encoded file bytes: 162620
decoded file bytes: 162678
finish decoding!
Running time: 32.82988 Seconds

In [22]: |
```

图 20: 解码的程序运行截图

耗时  $32.8299\text{s}$ 。

打开译码后文件，与原文件对比展示如图 21。

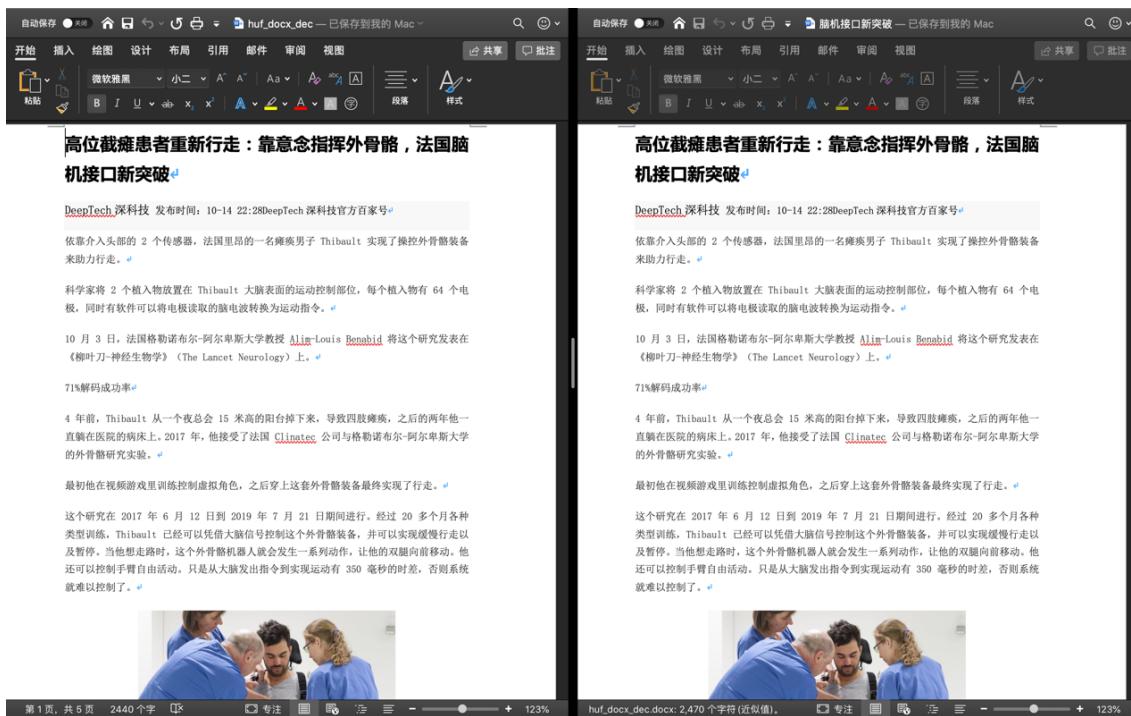


图 21: 译码后文件与原文件对比展示

### 3.1.3 编解码正确性

如图 16、21 所展示的，对 `.txt` 文件、`.docx` 文件进行编码后再进行解码，解码后文件均能正常打开，且内容、大小与原文件一样，所以本 *Huffman* 编码与解码的过程是正确有效的。

## 3.2 性能分析

为了更直观地分析，将实验结果制成表格如下：

	文件类型	原文件大小 /bytes	压缩文件大小 /bytes	码表大小 /bytes	最终压缩文件大小 /bytes	压缩率	压缩比	编码耗时 /s	译码耗时 /s
诺贝尔化学奖.txt	<code>.txt</code>	4039	3210.375	411.625	3622	11.2581	1.1151	0.03237	0.4250
脑机接口新突破.docx	<code>.docx</code>	162678	161844.5	775.5	162620	1.0052	1.0004	0.6310	32.8299
比值	<code>.docx/.txt</code>	40.28	50.41	1.88	44.89	0.8000	0.9009	19.49	77.25

### 3.2.1 压缩效率分析

从实验结果可以看出，我们的 *Huffman* 编码对 `.txt` 文件的压缩效果明显好于 `.docx` 文件。这是由于 `.txt` 和 `.docx` 两种文件的自身特性决定的。

查找资料可以知道，一般中文的 `.txt` 文件采用 *UTF-8* 编码，如图 22。而 *UTF-8* 编码是一种典型的整字节编码，或者说类似一种定长编码。它关注的是编码的通用性，目的是能表示尽可能多的字符，尽可能兼容其他定长编码集。所以对于一般的 `.txt` 文件来说，它有相当大的冗余度，因此我们的 *Huffman* 编码才能大展身手，压缩率达到 1.26。

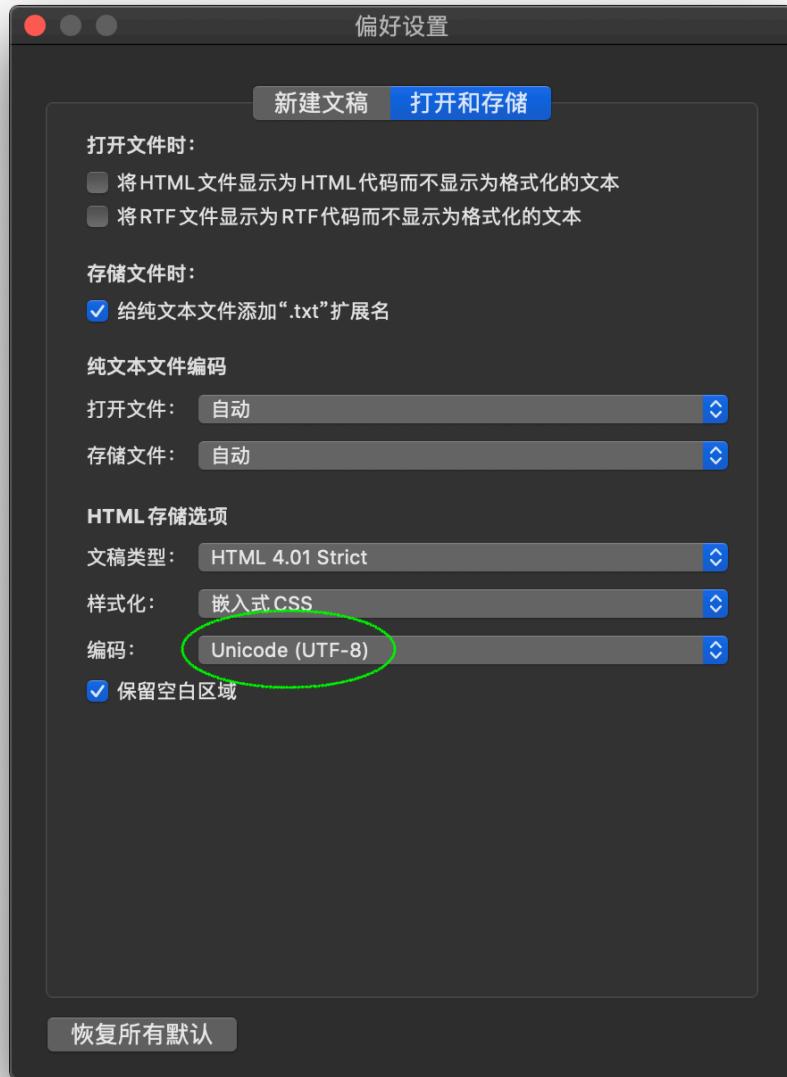


图 22: .txt 采取  $UTF_8$  编码

而 .docx 文件是基于 *Office Open XML* 标准的压缩文件格式，其本质上是一个 ZIP 压缩文件。3.1.2 节我们能够用 ZIP 解压软件打开 .docx 文件，并看到其内部文件结构就说明了这一点。如图 23。所以 .docx 文件原本就是一个用商业压缩软件压缩好了的文件，我们使用 *Huffman* 编码对其进行压缩只对其极少量的格式控制内容有效，而对占绝大部分的文本内容与图片内容是无效的，因为它们已经被压缩好了，其信源符号概率分布已经非常均匀，冗余度已经非常小。再加上我们的 *Huffman* 编码还会为压缩文件加上码表以帮助译码，所以 *Huffman* 编码对 .docx 文件的压缩效果非常差，压缩比只有 1.0004。从图 24可以看到，经过 ZIP 解压软件 Dr.Unarchiver 解压后的 .docx 文件形成一个文件夹，大小从 163KB 增大到了 258KB。



图 23: .docx 的内部文件结构

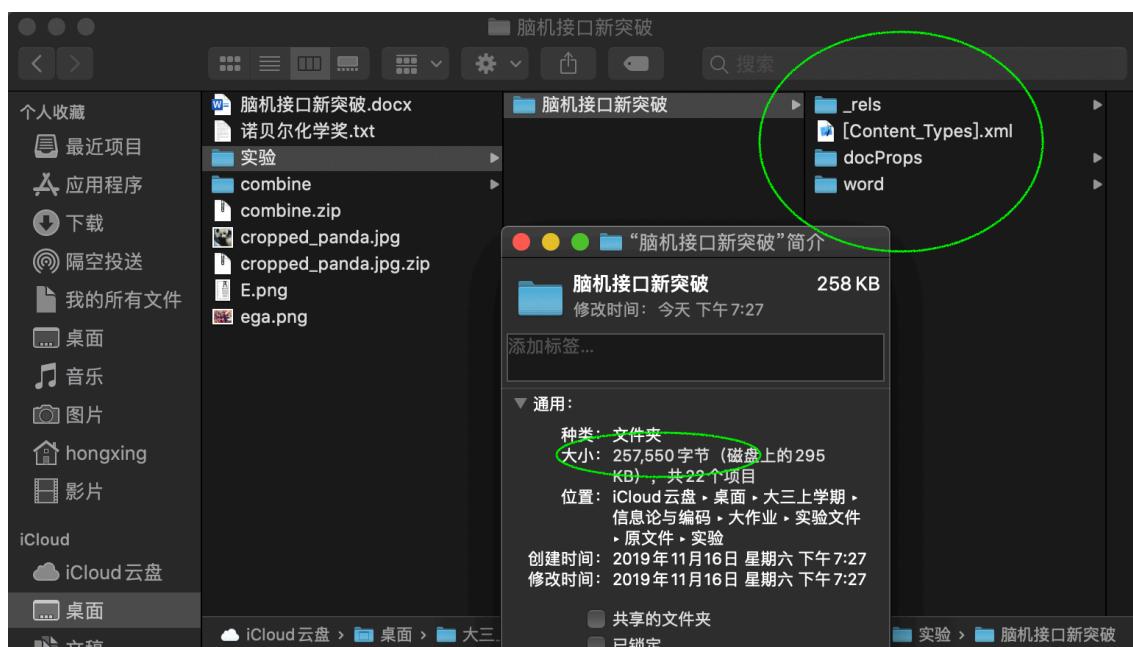


图 24: ZIP 解压软件解压后的.docx 文件

在第 4 章我们也会对 .docx 文件进行深入实验，对在此处提出的猜想进行验证。

### 3.2.2 运行时间分析

在第二章我们分析过编码和译码的时间复杂度都是  $O(n)$ ，不同的是常数因子，且一般而言译码的常数因子比编码的常数因子大一些，实验结果也说明了这一点。但是，从理论角度来说，编码的耗时与译码的耗时应该与原文件的大小成正比，但实验结果与理论似乎有些差距。进一步思考可以得出这个现象的解答：

第一，该算法的时间复杂度  $O(n)$  是在文件足够大时成立的，文件较小时，如实验中的 .txt 文件，其编解码时间与文件大小并不成线性关系。因为在文件较小时，全部 256 个信源符号可能并没有全部出现在文件中，甚至只出现一小部分，因此编解码相对较快。文件越大，包含越多种信源符号的概率越大，编解码的负担就越重，此时耗时与所含信源符号种类数与文件大小两种因素有关，因此是非线性的；当文件足够大时，有很大概率全部 256 种信源符号都出现在文件中，可以认为此时编解码的耗时只与文件大小有关了，这时耗时才会与文件大小成线性关系。

第二，编解码耗时也与文件性质有关。如.txt 文件，其由 UTF-8 编码，当文件较小时，其信源符号分布很不均匀，没有出现过的信源符号可能很多，因此小的 .txt 文件编解码很快。而 .docx 文件是经过压缩编码的，其信源符号概率分布已经很均匀，也就是说，全部 256 种信源符号全部都在文件中出现的概率很大，且个信源符号出现概率相等，因此即使是小的 .docx 文件其编解码速率也会比较慢。

从实验结果看出，该编码程序对小文件速度很快，但对较大的文件就显得力不从心了，与电脑上常见的商业压缩软件明显有很大差距。究其原因，我想是因为：第一，商业压缩软件综合考虑了压缩能力和运行效率，采用不断优化的压缩算法，因此时间复杂度与常数因子都比我们小一些；第二，商业压缩软件可能会调用硬件来为压缩算法加速，且压缩算法针对系统和硬件进行了优化，所以速度快很多。

## 4 比较实验

### 4.1 Huffman 编码压缩不同原文件

#### 4.1.1 深入 Huffman 编码压缩 .txt 原文件与 .docx 原文件实验

这一节继续第 3 章更深入地讨论 Huffman 编码压缩 .txt 与 .docx 原文件的实验，以为第三章中提出的解释提供更多的依据。

对图 25 所示文件进行 Huffman 编解码实验。

其中 empty.txt 是个空的 .txt 文件，可以看到它的大小为 0。其他文件均为随机从小说、代码、论文中摘抄来的混合文本组成的文件。以下列出各文件编解码实验结果：

文件名	原文件大小 /bytes	压缩文件大小 /bytes	码表大小 /bytes	最终压缩文件大小 /bytes	压缩率	压缩比	编码耗时 /s	译码耗时 /s
1.txt	416	299.375	242.625	542	1.3896	0.7675	0.00838	0.05290
2.txt	4039	3210.375	411.625	3622	1.2581	1.1151	0.02910	0.43223
3.txt	7883	6161.25	464.75	6626	1.2794	1.1897	0.03790	0.82718
4.txt	11976	9338.5	416.5	9755	1.2824	1.2277	0.05710	1.12518
5.txt	24191	19033.5	486.5	19520	1.2710	1.2393	0.11394	2.42758
6.txt	325320	232169.375	490.625	232660	1.4012	1.3983	1.10482	26.67480

从压缩比可以看出，随着原文件大小的增大，压缩比也在增大并趋向稳定，这主要是由于封装的码表在最终压缩文件中占比越来越小所致。我们也很容易可以看出随着原文件大小的增大，压缩比越来越靠近压缩率，说明码表对压缩效果的影响越来越小。另外，随着原文件增大，码表大小也在增大并趋向稳定，这主要是由于出现的信源符号种类增多并达到 256 所致。

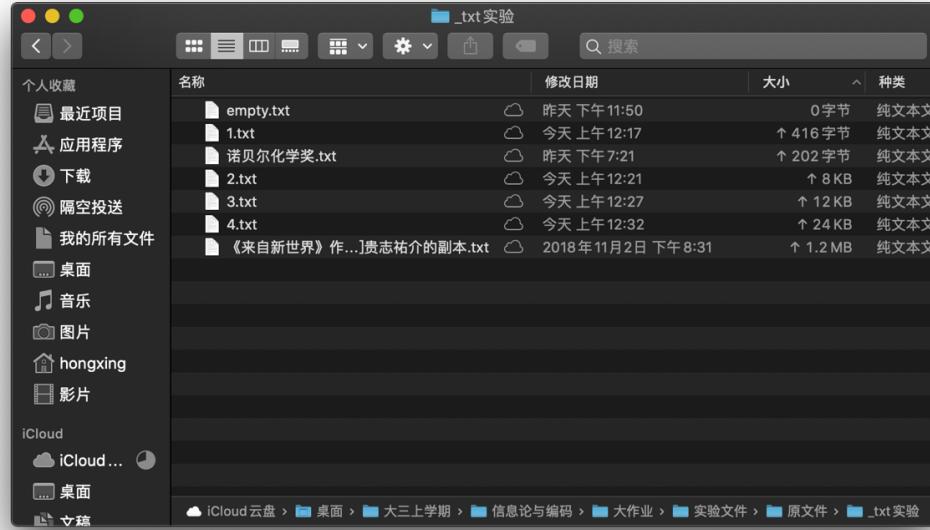


图 25: 实验文件

从编码耗时和译码耗时可以看出，当文件足够大时，耗时已经基本和原文件大小成线性关系，这也印证了时间复杂度为  $O(n)$  的结论。

而当对 `.docx` 进行实验时，情况稍有不同，这是因为正如我们第三章所说的 `.docx` 文件是一个含有已压缩文件的混合文件。下面我们将对图 26 所示的 `.docx` 文件进行实验。`.docx` 文件中的文本与对应的 `.txt` 文件的文本相同。

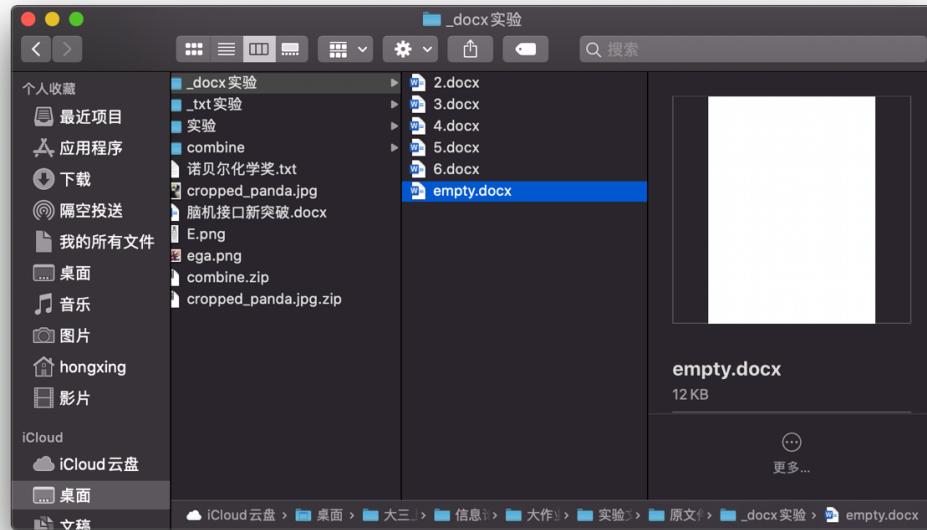


图 26: 实验文件

可以看出，即使是空的 *.docx* 文件仍然有 *12KB* 的大小，这就是 *.docx* 文件附带的格式控制数据，也是我们能够有效压缩的部分。以下列出各文件编解码实验结果：

文件名	原文件大小 /bytes	压缩文件大小 /bytes	码表大小 /bytes	最终压缩文件大小 /bytes	压缩率	压缩比	编码耗时 /s	译码耗时 /s
empty.docx	11906	10658.875	789.125	11448	1.1170	1.0400	0.08558	2.29320
2.docx	15119	13967.625	781.375	14749	1.0824	1.0251	0.09346	2.95471
3.docx	15938	14810.875	781.125	15592	1.0761	1.0222	0.08979	3.47227
4.docx	18490	17417.875	781.125	18199	1.0616	1.0160	0.11213	3.70192
5.docx	22234	21268.875	781.125	22050	1.0454	1.0083	0.11693	4.46147
6.docx	176324	176053.5	774.5	176828	1.0015	0.9971	0.65060	35.29224

从码表大小一列可以看出，在 *.docx* 文件很小时，封装码表就显然已经比同样大小的 *.txt* 文件的码表大得多，这刚好印证了我们之前的“在 *.docx* 文件中由于文本已被压缩，所以所有信源符号都会出现，且分布均匀”这一结论。且还可以看出文件越大，码表越小，这是因为文件中已包含所有信源符号，所以码表大小不再由信源符号种类数决定，而由信源符号的分布决定；而文件越大，*.docx* 文件已被压缩部分的信源符号分布就越均匀，*Huffman* 树的平均深度就越短。有理由相信在文件足够大时码表传递的 *Huffman* 树已是满树。

从压缩率与压缩比可以看出这两项都在随原文件的增大而减小，这恰恰印证了我们之前的另一个猜想：“由于 *.docx* 文件中文本部分已被压缩，所以 *Huffman* 编码对 *.docx* 文件的文本部分是起不到压缩作用的，只能对 *.docx* 的未经压缩的格式控制部分有效压缩。”压缩率和压缩比的降低是因为随着文件的增大，*.docx* 文件的格式控制部分（如 *empty.docx* 所示为 *12KB*）所占比例越来越小，即 *Huffman* 编码能有效压缩的部分占比越来越小，压缩能力当然会越来越差。在 *.docx* 文件足够大时，如 *6.docx*，能有效压缩的部分已经小到一定程度，可以看到压缩比已经小于 1，即 *Huffman* 编码无法再对 *.docx* 文件整体进行有效压缩。

而耗时部分也正如时间复杂度所表示的，当文件足够大时，编译码耗时与文件大小成线性关系。且经计算拟合可得，在笔者的 *Macbook* 上，译码时间最后趋近于编码时间的 50 倍。

#### 4.1.2 Huffman 编码压缩 *.BMP* 原文件与 *.png* 原文件实验

*.BMP* 文件是一种非压缩的图片格式，而 *.png* 是一种无损压缩的图片格式。我们将分别对由 *SketchBook* 中同一张图片导出的两种格式的图片进行 *Huffman* 编码，观察压缩效果。

两张图片展示如下。其中左为 *.bmp* 格式，大小为 *5MB*；右为 *.png* 格式，大小为 *1.8MB*。可见未压缩格式的图片比压缩格式的图片大得多。



图 27: tennkinoko.bmp



图 28: tennkinoko.png

对《tennkinoko.bmp》进行 *Huffman* 编码解码的效果如下图。左为原文件，右为编码后解码文件，大小均为 4,981,734 字节。



图 29: tennkinoko.bmp



图 30: huf\_dec\_bmp\_tennki.bmp

对《tennkinoko.png》进行 *Huffman* 编码解码的效果如下图。左为原文件，右为编码后解码文件，大小均为 1,785,621 字节。



图 31: tennkinoko.png



图 32: huf\_dec\_png\_tennki.png

得到压缩 *.bmp* 与 *.png* 文件的实验结果：

文件名	原文件大小 /bytes	压缩文件大小 /bytes	码表大小 /bytes	最终压缩文件大小 /bytes	压缩率	压缩比	编码耗时 /s	译码耗时 /s
tennkinoko.bmp	4981734	3941469.0	806.0	3942275	1.2639	1.2636	16.91097	787.37206
tennkinoko.png	1785621	1785621.0	774.0	1786395	1.0	0.9996	6.26295	369.82268

实验结果一目了然。对于未压缩格式的 *.bmp* 文件，*Huffman* 编码压缩效果较好，压缩率达到了 1.264；而对于压缩格式的 *.png* 文件，纯压缩文件和原文件一样大，压缩率为 1，且压缩比小于 1，这是因为最终压缩文件还要加上码表。这也印证了一个显而易见的结论：对于已压缩文件，由于其符号已经分布均匀，冗余度已经很小，很难进一步压缩；而分组码的压缩方法会附加上码表等冗余信息，反而会使文件增大。

## 4.2 LZ 编码与 *Huffman* 编码压缩性能对比

*LZ* 编码的原理由于不是重点，这里不再赘述。我们使用的算法和课本上描述的 *LZ78* 算法相同，而具体的技术细节如字符转换，文件输入输出与 *Huffman* 编码相同。下面我们直接使用 *LZ* 编码对 *Huffman* 编码已实验过的 *.txt* 文件和 *.docx* 文件进行实验，再在之后与 *Huffman* 编码进行性能对比。以下是实验结果表，为了方便观察，把之前进行 *Huffman* 编码的部分实验数据也放在其中。

编码算法	文件名	原文件大小 /bytes	最终压缩文件大小 /bytes	压缩比	编码耗时 /s	译码耗时 /s
LZ	诺贝尔化学奖.txt	4039	4546	0.8885	0.57398	0.10386
Huffman	诺贝尔化学奖.txt	4039	3622	1.1151	0.03237	0.42500
LZ	.docx	162678	178462	0.91156	370.54010	2.60782
Huffman	.docx	162678	162620	1.0004	0.6310	132.8299

分析编解码时间：由于 LZ 编码算法并未对建立字典的过程进行特别的优化，还是使用原始的字典搜索，且 LZ 编码的字典大小不像 Huffman 编码的编码表一样有上界，而是随文件的增大而增大。对于某些文件，LZ 编码的字典可能增大到一定程度就不再增大，时间复杂度最后趋近于  $O(n)$ ；但对于另一些特殊文件，字典可以一直增大，此时将达到最大时间复杂度  $O(n^2)$ 。所以本 LZ 编码的最大时间复杂度为  $O(n^2)$ 。在实验结果中也可看出，LZ 编码的耗时明显比 Huffman 编码的耗时高，且随文件大小加速增大。而 LZ 编码的译码仅仅是遍历序列，并用数组下标寻址码字，虽然其最大时间复杂度也为  $O(n)$ ，但常数因子比 Huffman 编码的译码小得多，所以在译码时间上 LZ 编码有明显优势。

分析压缩效果：从理论上来说，在信源序列足够长时，LZ 编码能使编码后序列冗余度无限小，即达到压缩极限。但显然我们实验的文件的符号序列是不够长的，即使没有像 Huffman 编码的最终文件那样携带码表，其压缩比也远小于 Huffman 编码，甚至使压缩后文件变大了。另外，压缩比也和文件的二进制流 01 序列的排布顺序有关，不同的文件会导致不同的二进制流排列顺序，进而会影响压缩的效果。对足够大的文件，使用 LZ 编码可能效果更好；对于较小的文件，可能使用 Huffman 编码效果会更为显著；或许也可以将几种编码算法结合在一起使用，如电脑上的一些商业压缩算法。从这里我们也可以得出这样一个结论：不同的编码算法对不同原文件的压缩效果不一样，在压缩文件时，应该选择对应的最有效的算法进行压缩以达到最高的压缩比。

## 5 总结与体会

### 5.1 总结

从本次大作业的实践中我们可以得到以下结论：

- 对不同文件使用同样的信源编码方法进行压缩，压缩的效果不同。使用朴素的 Huffman 编码等变长信源编码算法可以对冗余度较大的文件进行有效的压缩，但对于已压缩好、冗余度很小的文件不能进行有效压缩，甚至适得其反。
- 不同的信源编码算法对同样的文件的压缩效果不同。在压缩文件时，应该选择对应的最有效的算法进行压缩以达到最高的压缩比。
- 对文件进行编码存储和传输时，应综合考虑文件的用途、环境等自身特点来选择合适的编码方法。如果侧重于文件的通用性与易操作性，宜采用定长编码或整字节编码方法，如 .txt 文件；如果侧重于高效传输与存储，宜采用变长编码方法以压缩冗余度，如 .zip 文件；如果各方面都希望兼顾，则可以效仿 .png、.docx 等新型压缩格式。

### 5.2 体会

本次大作业是对信息论与编码理论的一次绝佳实践。通过这次大作业中的实验，我们对香农第一定理定理，即信源编码定理有了更深刻的理解。在对实验结果进行解释的过程中，我们查阅了大量资料，了解了 .txt、.docx、.png、.zip 等文件格式的具体内容与深刻思想，并且也进行了深入的思考。

同时，通过自己的实践，在看到自己的程序能实现对文件的有效压缩与解压时，不仅心中有满满的成就感，也对信源编码对文件高效存储与传输的巨大作用有了清晰的认识，对香农、维纳等信息论先贤们的伟大成就深感敬意，同时心中也暗下决心，要认真学习知识，积极思考，刻苦钻研，努力向先贤们看齐，希望自己有一天也能够取得这样的开创性成就，为人类的美好生活做出自己的贡献！

随着大作业的完成，《信息论与编码》这门课程也就告一段落了。但课程会结束，学习永远不会结束。信息论对身在信息安全专业的我们是非常重要的，可以说是信息安全的奠基学科之一。同时，《信息论与编码》这门课带给了我们全新的思想：用概率论的方法描述信息。这不仅是信息论的思想闪光点，更给我们在其他方向带来巨大启发。相信，信息论与编码将会使我们受益终身。