在第4章我们也会直接用ZIP文件进行压缩实验，我们将会看到经过Huffman编码后ZIP文件反而变大了，这很好地印证了我们此处的分析。

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。用数学语言描述就是:

begin{equation}

T(x) = \left\{

begin{align}

K\_1f(n)n\\

K\_2n

end{align}

end{equation}

其中K\_1、K\_2均为常数，f(n)正比于文件中信源符号种类，同时也与n有关，n是文件大小。

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。在第4章比较实验中我们会对这一点详细叙述。

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很多；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。

4.1.2 Huffman编码压缩.BMP原文件与.png原文件实验

4.1.3 Huffman编码压缩.zip原文件与.huf原文件实验

4.2 LZ编码与Huffman编码压缩性能对比

4.2.1 LZ编码压缩.txt原文件与.docx原文件实验

4.2.2 性能对比

4.3 商业压缩软件性能与Huffman编码压缩性能对比

5 总结与体会

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

import time

import math

import copy

from tqdm import tqdm

def file\_to\_bit(source\_file): #将二进制文件转换成二进制字符串

file1 = open(source\_file,'rb')

binary=file1.read()

file1.close()

lenth=binary.\_\_len\_\_() #统计二进制字节长度

#字节转换成bit字符串

number=[0 for i in range(0,lenth)] #字节流转换成数字

source=""

for i in range(0,lenth):

# print(i)

number[i]=int(binary[i])

l=bin(number[i])#辅助符号

l=l.lstrip('0b')

while l.\_\_len\_\_() < 8 : #对每一个字节补足左边省略的‘0’

l='0'+l

source=source+l

str\_len=bin(lenth)

str\_len=str\_len.lstrip('0b')

str\_len=str\_len.zfill(24) #记录字符串长度并用24位二进制表示，左边补‘0’

str\_len1=''

for i in range(0,24):

if (str\_len[i]=='1'):

str\_len1=str\_len1+'0'

elif (str\_len[i]=='0'):

str\_len1=str\_len1+'1'

return source,str\_len1 #返回二进制字符串和文本字节数

def inversion(binary): #对二进制字符串按位取反，避免对全0字节进行省略

len=binary.\_\_len\_\_()

len\_2=''

for i in range(0,len):

if (binary[i]=='1'):

len\_2=len\_2+'0'

elif (binary[i]=='0'):

len\_2=len\_2+'1'

return len\_2

def frequency(source):

num=[]

len=source.\_\_len\_\_()

temp=source

result=dict()

for i in range(0,int(len/8)):

j=8\*i

num.append(temp[j:j+8])

for i in num:

if i in result:

result[i]+=1

else:

result.update({i:1})

return result #对二进制字符串以8位为单位分析出现频率，并将结果用字典存放

class node: #建立节点类

def \_\_init\_\_(self):

self.frequency=0

self.name=None

self.l=None

self.r=None

self.code=None

def \_\_lt\_\_(self, other): #利用富比较排序各字节出现的频率

return self.frequency<other.frequency

def huffman(dict): #返回赫夫曼数的根节点

node\_ls=[]

for i in dict:

a = node()

a.frequency=dict[i]

a.name=i

node\_ls.append(a)

while len(node\_ls)>1:

node\_ls.sort(reverse=True)

temp\_1=node\_ls.pop()

temp\_2=node\_ls.pop()

new=node()

new.frequency=temp\_1.frequency+temp\_2.frequency

new.l=temp\_1

new.r=temp\_2

node\_ls.append(new)

return new

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。

速网和低速网 无线网和有线网 光纤网等几乎所有类型的计算机通信技术

网络互联层的主要任务是 为IP数据报分配一个全网唯一的传送地址 称为IP地址 实 现IP地址的识别与管理 IP数据报的路由机制 发送或接收时使IP数据报的长度与通信子网所 允许的数据报长度相匹配 例如 以太网所传输的帧长为1500字节 而ARPANET所传输的数据 包长度为1008字节 当以太网上的数据帧通过网络互联层IP协议转发给ARPANET时 就要进行 数据帧的分解处理

顺便指出 这里的 数据报 和以前提到的 数据包 packet 是不同的概念 数据包 是指分块的传输数据 它被用于早期的计算机通信网的文献中 而目前则普遍使用的 分组 一词 而数据报则是分组的一种传送方式或网络提供的一种无连接服务 在TCP/IP模型中 数 据报亦可指使用数据报服务来传送的具有一定格式的分组