蘭州大學
LANZHOU UNIVERSITY

Realize Huffman coding optimization based on TOP-DOWN mode

Homework summary report

题　　目：	基于top-down方式实现哈夫曼编码优化
授课教师：	刘忻
姓　　名：	周功海
学　　号：	320190903781
日　　期：	2021年11月30日

基于TOP-DOWN方式实现哈夫曼编码优化及与算术编码对比

周功海，320190903781

兰州大学信息科学与工程学院

摘　要： 首先以传统方式实现了哈夫曼编码，依据题目要求使用python3.8+pyqt5+graphviz实现哈夫曼树的编码解码以及UI界面展示。优化方面：哈夫曼编码作为一种无损数据压缩编码在计算机信息压缩中有广泛的应用。但传统的哈夫曼编码的实现方式是在构造哈夫曼树的基础上，从叶子节点向上到根节点逆向进行的。为了提高编码的效率，给出了一种新的哈夫曼编码实方式，该方式通过利用队列的数据结构，从哈夫曼树的根节点出发，向叶子节点进行编码，在编码过程中仅将哈夫曼树的每个节点进行一次扫描就可得到各叶子节点的哈夫曼编码。该方法不仅符合编码的思维方式，而且解决了原先编码过程中大量指针移动的问题，将哈夫曼编码的时间复杂度由原来的O(n^2)降为O(n)。除此以外，本文还集中介绍了当前最先进的压缩编码方式——算术编码，并将其与Huffman编码方式进行对比，使用python将其成功实现，并对学界当前最先进算术编码进行集中介绍。

关键词： python3.8，pyqt5，graphviz，哈夫曼编码，算术编码，

Realization of Huffman coding optimization based on TOP-DOWN mode and comparison with arithmetic coding

Zhou Gonghai

School of Information Science and Engineering, Lanzhou University

Abstract: First, the Huffman coding is implemented in a traditional way, and the coding and decoding of the Huffman tree and the UI interface display are realized using python3.8+pyqt5+graphviz according to the requirements of the subject. Optimization aspects:Huffman coding which is a lossless data compression coding bas a wide r 或e of using in the computer information compression. However, the traditional way to achieve Huffman coding is from leaf nodes to the root node on the-basis of the Huffman tree. In order to improve coding efficiency, presents a new implementation approach of the Huffman coding, which codes from the root node to leaf nodes of the Huffman tree by using the data structure of queue. In the coding process, every node is only scanned once before get the Huffman coding. This method not only coincide with the thinking of encoding mode, and solves the problem of a large number of pointers'movement, but also reduces the time complexity of the Huffman coding from O(n^2) to O(n).In addition, this article also focuses on the current most advanced compression coding method-arithmetic coding, and compares it with the Huffman coding method, using python to successfully implement it, and a concentrated introduction to the current most advanced arithmetic coding in academia.

Key Words: python3.8，pyqt5，graphviz，Huffman coding，Arithmetic coding

1　Experiment title

我选择的实验题目为：哈夫曼编码 (PS:以下实验过程均为本人单独完成)

1.1 哈夫曼编码问题

问题描述：打开一篇英文文章，统计该文章中每个字符出现的次数，然后以它们作为权值，对每一个字符进行编码，编码完成后再对其编码进行译码。

利用哈夫曼编码进行信息通信可以大大提高信道利用率，缩短信息传输时间，降低传输成本。但是，这要求在发送端通过一个编码系统对待传数据预先编码，在接收端将传来的数据进行译码（复原）。对于双工信道（即可以双向传输信息的信道），每端都需要一个完整的编译码系统。试为这样的信息收发站写一个哈夫曼编译码系统。
一个完整的系统应具有以下功能：
（1） I： 初始化 (Initialization)。从终端读入字符集大小 n ，以及 n 个字符和 n 个权值，建立哈夫曼树，并将它存于文件 hfmTree 中。
（2） E： 编码 (Encoding)。利用已建好的哈夫曼树（如不在内存，则从文件 htmTree 中读入），对文件 ToBeTran 中的正文进行编码，然后将结果存入文件 CodeFile 中。
（3）D： 译码(Decoding)。利用已建好的哈夫曼树将文件 CodeFile 中的代码进行译码，结果存入文件 TextFile 中。
（4）P： 印代码文件(Print)。将文件 CodeFile 以紧凑格式显示在终端上，每行 50 个代码。同时将此字符形式的编码写入文件 CodePrint 中。
（5） T： 印哈夫曼树 (TreePrinting)。将已在内存中的哈夫曼树以直观的方式（树或凹入表形式）显示在终端上，同时将此字符形式的哈夫曼树写入文件 TreePrint 中。
涉及知识点： 哈夫曼树
涉及论文：游洪跃 , 汪建武 , 陶郁 . 广义哈夫曼树及其在汉字编码中的应用 英文)[J].四川大学学报 自然科学版 , 2000(04):53 56.

1.2 优化哈夫曼编码

问题描述：传统的哈夫曼编码的实现方式是在构造哈夫曼的基础上，从叶子节点到根节点逆向进行的。为了提高编码的效率， 需设计 一种新的哈夫曼编码实现方式，该方式 的思路是：通过利用二叉排序树的数据结构，从哈夫曼的根节点出发，向叶子节点进行编码，在编码过程中仅将哈夫曼树的每个叶子节点进行一次扫描便可以得到各个叶子节点的哈夫曼编码。 请设计实例实现。
涉及知识点： 二叉排序树、哈夫曼编码

涉及论文：王防修 周康 基于二叉排序树的哈夫曼编码 [J]武汉工业学院学报 ,2011,30(04):45-48.

2 引言

在计算机科学和信息论中，哈夫曼编码是一种特殊类型的最优前缀码，通常用于无损数据压缩。查找或使用此类代码的过程通过Huffman 编码进行，这是一种由David A. Huffman在攻读博士学位期间开发的算法。麻省理工学院的学生，并发表在 1952 年的论文“A Method for the Construction of Minimum-Redundancy Codes”中。几十年来，这种编码方式已经渗透到我们日常学习工作的方方面面，例如数据压缩和数据传输等。

在数据结构课堂上，老师给我们详细介绍了哈夫曼编码，对我启发很大，本软件主要使用python3.8、PyQt5以及Graphviz实现输入文本的哈夫曼编码解码、二叉树的显示以及对服务器传输哈夫曼编码的功能。

2.1 哈夫曼算法简介

2.1.1 背景

随着现代社会的发展以及对于信息传输迫切需求，哈夫曼编码技术在近几十年中得到了飞速发展。哈夫曼编码技术是当前信息学科的基础，为整个工业信息压缩、传输等方面的相关功能发挥了巨大的作用。

哈夫曼编码算法是基于二叉树构建编码压缩结构的，它是数据压缩中经典的一种算法。算法根据文本字符出现的频率，重新对字符进行编码。因为为了缩短编码的长度，我们自然希望频率越高的词，编码越短，这样最终才能最大化压缩存储文本数据的空间。哈夫曼算法的输出可以看作是一个可变长度的代码表，用于对源符号（例如文件中的字符）进行编码。该算法根据源符号的每个可能值的估计概率或出现频率（权重）推导出该表。与在其他熵编码方法中一样，更常见的符号通常比不太常见的符号使用更少的位来表示。哈夫曼的方法可以有效地实现，如果这些权重被排序，则找到与输入权重的数量成线性时间的代码。

利用哈夫曼编码进行信息通信可以大大提高信道利用率，缩短信息传输时间，降低传输成本。但是，这要求在发送端通过一个编码系统对待传数据预先编码，在接收端将传来的数据进行译码（复原）。对于双工信道（即可以双向传输信息的信道），每端都需要一个完整的编/译码系统。

此次数据结构大作业所实现的基于哈夫曼编码的信息传输系统就着力于此方面，实现一个端对端的文本编码、解码、传输系统，以期望可以实现相关功能，提高信道利用效率，降低信息传输成本。

2.1.2 名词定义

下面列出本软件系统涉及到的名词定义，便于用户对本文档以及软件系统的理解。

(1)哈夫曼编码：（Huffman coding）哈夫曼编码使用特定的方法为每个符号选择表示，产生前缀码（有时称为“无前缀码”，即表示某个特定符号的位串永远不会是代表任何其他符号的位串的前缀）象征）。哈夫曼编码是一种广泛用于创建前缀代码的方法，以至于术语“哈夫曼代码”被广泛用作“前缀代码”的同义词，即使这种代码不是由哈夫曼算法产生的。

(2) 哈夫曼树：即HuffmanTree，表示用于使用哈夫曼编码对文件进行编码和解码的哈夫曼树的类。

(3) 初始化：即Initialization，即我们在文本框中输入或直接导入）。从终端读入字符集大小n，以及n个字符和n个权值，建立哈夫曼树，并将它存于文件hfmTree中。

(4)PyQt5: Qt 是一组跨平台的 C++ 库，它们实现了高级 API，用于访问现代桌面和移动系统的许多方面。其中包括定位和定位服务、多媒体、NFC 和蓝牙连接、基于 Chromium 的网络浏览器以及传统的 UI 开发。PyQt5 是 Qt v5 的一套全面的 Python 绑定。它以超过 35 个扩展模块的形式实现，使 Python 能够在所有支持的平台（包括 iOS 和 Android）上用作 C++ 的替代应用程序开发语言。

(5)Graphviz: Graphviz 是开源图形可视化软件。图可视化是一种将结构信息表示为抽象图和网络图的方法。它在网络、生物信息学、软件工程、数据库和网页设计、机器学习以及其他技术领域的可视化界面中具有重要应用。

2.2 哈夫曼编码算法数学表述

广义上的哈夫曼编码为：给定一组符号（Symbol）和其对应的权重值（weight），其权重通常表示成概率（%）。

而更为精确的数学表述为我们输入一组符号集合$A=\{a_1,a_2,a_3,\ldots,a_n\}$,此为表示大小的符号字母表$n$,元组$W=\{w_1,w_2,w_3\ldots，w_n\}$，这是（正）符号权重的元组（通常与概率成正比），即$W=weigth(a_i),i\in\{1,2,3\ldots,n\}$,输出方面，一组编码$C(S,W)=\{c_1,c_2,\ldots，c_n\}$,其$C$集合为一组二进制编码且$c_i$为$s_i$相应的编码，$1\leq i\leq n$。

目标为得到$L(C)=\sum_{i=1}^{n}wi\times length(ci)$为$C$的加权的路径长，对所有编码$T(S,W)$,则$L(c)\leq L(T)$。

实现哈夫曼编码的方式主要是创建一个二叉树和其节点。这些树的节点可以存储在数组里，数组的大小为符号（symbols）数的大小n，而节点分别是终端节点（叶节点）与非终端节点（内部节点）。

一开始，所有的节点都是终端节点，节点内有三个字段：

1.符号（Symbol）

2.权重（Weight、Probabilities、Frequency）

3.指向父节点的链接（Link to its parent node）

而非终端节点内有四个字段：

1.权重（Weight、Probabilities、Frequency）

2.指向两个子节点的链接（Links to two child node）

3.指向父节点的链接（Link to its parent node）

基本上，我们用'0'与'1'分别代表指向左子节点与右子节点，最后为完成的二叉树共有n个终端节点与$n-1$个非终端节点，去除了不必要的符号并产生最佳的编码长度。

过程中，每个终端节点都包含着一个权重（Weight、Probabilities、Frequency），两两终端节点结合会产生一个新节点，新节点的权重是由两个权重最小的终端节点权重之总和，并持续进行此过程直到只剩下一个节点为止。

实现哈夫曼树的方式有很多种，可以使用优先队列（Priority Queue）简单达成这个过程，给与权重较低的符号较高的优先级（Priority），算法如下：

1.把n个终端节点加入优先队列，则n个节点都有一个优先权Pi，$1 \leq i \leq n$

2.如果队列内的节点数>1，则：

⑴从队列中移除两个最小的Pi节点，即连续做两次remove（min（Pi）, Priority_Queue)

⑵产生一个新节点，此节点为（1）之移除节点之父节点，而此节点的权重值为（1）两节点之权重和

⑶把（2）产生之节点加入优先队列中Time Complexity

3.最后在优先队列里的点为树的根节点（root）

而此算法的时间复杂度(Time Complexity)为$nlogn$,因为有n个终端节点，所以树总共有$2n$个节点，使用优先队列每个循环须$O(logn)$。

此外，有一个更快的方式使时间复杂度降至线性时间（Linear Time)$O(logn)$，就是使用两个队列（Queue）创件哈夫曼树。第一个队列用来存储n个符号（即n个终端节点）的权重，第二个队列用来存储两两权重的合（即非终端节点）。此法可保证第二个队列的前端（Front）权重永远都是最小值，且方法如下：

1.把n个终端节点加入第一个队列（依照权重大小排列，最小在前端）

2.如果队列内的节点数＞1，则：

⑴从队列前端移除两个最低权重的节点

⑵将（1）中移除的两个节点权重相加合成一个新节点

⑶加入第二个队列

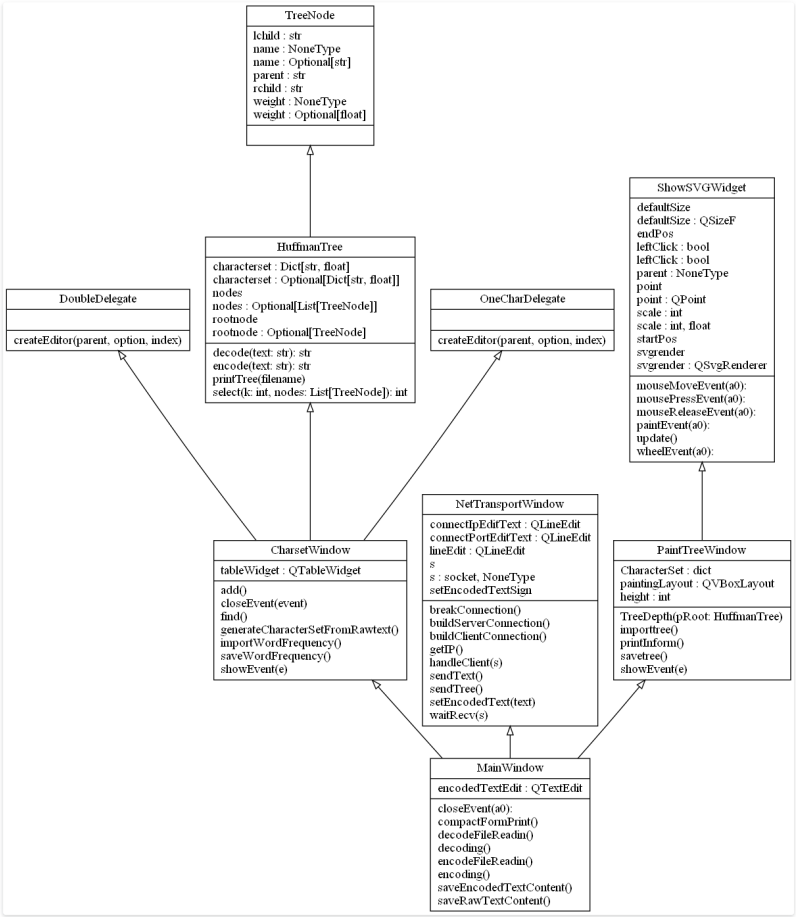
3.最后在第一个队列的节点为根节点

虽然使用此方法比使用优先队列的时间复杂度还低，但是注意此法的第1项，节点必须依照权重大小加入队列中，如果节点加入顺序不按大小，则需要经过排序，则至少花了$O(nlogn)$的时间复杂度计算。

但是在不同的状况考量下，时间复杂度并非是最重要的，如果我们今天考虑英文字母的出现频率，变量n就是英文字母的26个字母，则使用哪一种算法时间复杂度都不会影响很大，因为n不是一笔庞大的数字。

3 实验程序结构

如图所示：



图一.算法结构

4 算法设计

本题目大多数都是由C/C++实现的，而本人在自学python后发现用python处理本题目中的部分问题更为建议方便，因而本文多用python实现算法：

4.1 初始化（Initialization）

初始化有两种形式一种是直接根据原文生成，另一种是根据通过直接导入字符，而在这两种方法的基础上，还可对列表内的字符集进行增删改查，而对于现存的字符集还可进行保存文件这一操作，最终，在关闭该窗口时，程序会根据现有的字符集来进行建树。

4.1.1 根据原文生成

```
def add(self):
        # 加入一空行

 self.tableWidget.insertRow(self.tableWidget.rowCount
())
    def generateCharacterSetFromRawtext(self):
        # 根据原文生成字符集
        self.tableWidget.clearContents()
        self.tableWidget.setRowCount(0)
        def getFrequency(text: str) -> dict:
            # 字频(统计)
            cnt = {}
            for i in text:
                if i not in cnt:
                    cnt[i] = 1
                else:
                    cnt[i] += 1
            return cnt
        CharacterSet =
getFrequency(rawTextEdit.toPlainText())
        for i, j in CharacterSet.items():
            self.add()
            item1 = QTableWidgetItem(i)
            item2 = QTableWidgetItem(str(j))

 self.tableWidget.setItem(self.tableWidget.rowCount()
-1, 0, item1)

 self.tableWidget.setItem(self.tableWidget.rowCount()
-1, 1, item2)
```

4.1.2 直接导入字符

```
    def importWordFrequency(self):
        # 导入字频
        filePath, ok = QFileDialog.getOpenFileName(self, '选择文件')
        if ok:
            self.tableWidget.clearContents()
            self.tableWidget.setRowCount(0)
            with open(filePath, 'r', encoding='utf-8') as file:
                try:
                    frequency = file.read()
                except UnicodeDecodeError:
                    QMessageBox.critical(
                        self, "错误", "请确保打开的是UTF-8编码的文本文件", QMessageBox.OK)
                    return
            global CharacterSet
            CharacterSet = {}
            textlines = re.findall(r'([\s\S])\t(\S+)(\n|$)', frequency)
            if len(textlines) == 0:
                QMessageBox.critical(self, "错误", "字符集生成失败", QMessageBox.Ok)
                return
            for i, j, _ in textlines:
                try:
                    CharacterSet[i] = float(j)
                except ValueError:
                    QMessageBox.critical(
                        self, "错误", "字符集生成失败", QMessageBox.Ok)
                    self.tableWidget.clearContents()
                    self.tableWidget.setRowCount(0)
                    CharacterSet = {}
                    return
                self.add()
                item1 = QTableWidgetItem(i)
                item2 = QTableWidgetItem(j)
                self.tableWidget.setItem(
                    self.tableWidget.rowCount()-1, 0, item1)
                self.tableWidget.setItem(
                    self.tableWidget.rowCount()-1, 1, item2)
```

4.1.3 额外的增删改查

```
1
2       def add(self):
3           # 加入一空行
4
 self.tableWidget.insertRow(self.tableWidget.rowCount
())
5
6       def find(self):
7           # 对于字符或字频或字符与字频进行查找
8           a: str = self.wordFrequencyEdit.text()
9           b: str = self.frequencyEdit.text()
10          i: int = 0
11          if a and b:
12              while i < self.tableWidget.rowCount():
13                  if self.tableWidget.item(i, 0).text()
== a and self.tableWidget.item(i, 1).text() == b:
14
 self.resultLabel.setText(str(i+1))
15                      break
16                  i += 1
17          elif not a and b:
18              while i < self.tableWidget.rowCount():
19                  if self.tableWidget.item(i, 1).text()
== b:
20
 self.resultLabel.setText(str(i+1))
21                      break
22                  i += 1
23          elif a and not b:
24              while i < self.tableWidget.rowCount():
25                  if self.tableWidget.item(i, 0) and
self.tableWidget.item(i, 0).text() == a:
26
 self.resultLabel.setText(str(i+1))
27                      break
28                  i += 1
29          if i == self.tableWidget.rowCount():
30              self.resultLabel.setText("未找到")
31
32
```

4.1.4 保存字频

```
    def saveWordFrequency(self):
        # 保存文件
        filePath, ok =
QFileDialog.getSaveFileName(self, '选择文件')
        if ok:
            with open(filePath, 'w', encoding='utf-8')
as file:
                for i in
range(self.tableWidget.rowCount()):
                    m =
'\t'.join([self.tableWidget.item(
                        i, 0).text(),
self.tableWidget.item(i, 1).text()])
                    file.write(m+'\n')
```

4.1.5 构建二叉树

```
    def generateCharacterSetFromRawtext(self):
        # 根据原文生成字符集
        self.tableWidget.clearContents()
        self.tableWidget.setRowCount(0)
        def getFrequency(text: str) -> dict:
            # 字频(统计)
            cnt = {}
            for i in text:
                if i not in cnt:
                    cnt[i] = 1
                else:
                    cnt[i] += 1
            return cnt
        CharacterSet =
getFrequency(rawTextEdit.toPlainText())
        for i, j in CharacterSet.items():
            self.add()
            item1 = QTableWidgetItem(i)
            item2 = QTableWidgetItem(str(j))

 self.tableWidget.setItem(self.tableWidget.rowCount()
-1, 0, item1)

 self.tableWidget.setItem(self.tableWidget.rowCount()
-1, 1, item2)

    def closeEvent(self, event):
        # 关闭窗体
```

```
        if self.tableWidget.rowCount() == 0:
            return
        global CharacterSet
        CharacterSet = {}
        # 将表格中的字符集存入变量CharacterSet中
        for i in range(self.tableWidget.rowCount()):
            if self.tableWidget.item(i, 0) and self.tableWidget.item(i, 1):
                try:
                    CharacterSet[self.tableWidget.item(i, 0).text()] = float(
                        self.tableWidget.item(i, 1).text())
                except:
                    pass
        global HFTree
        # 将树依据现有的字符集进行更新
        if CharacterSet != {}:
            HFTree = HuffmanTree(CharacterSet)
            global showSVGWidget
            if showSVGWidget:
                HFTree.printTree('tmp')
                showSVGWidget.update()
                paintTreeWindow.printInform()
```

4.2 编码（Encoding）

编码则是根据内存中现有的树来对原文进行编码，而原文的读取方式有两种，一种是手动输入，一种是读取文件，而原文也可进行保存。

4.2.1 编码

```
    def encode(self, text: str) -> str:
        # 对text中的文本进行编码
        p, q = '', ''  # p是每个字符的编码，q是整篇文章的编码
        for i in text:
            for j in self.nodes:
                if i == j.name:
                    while j.parent:
                        if j.parent.lchild == j:
                            p += '0'
                        elif j.parent.rchild == j:
                            p += '1'
                        j = j.parent
                    q += p[::-1]
                    p = ''
```

```
                    break
            else:
                # 若当前字符并不在字符集中，则返回空的密文
                return None
        return q
    def encoding(self):
        if not HFTree:
            QMessageBox.critical(self, "错误", "当前无建好的树", QMessageBox.Ok)
        elif rawTextEdit.toPlainText() == '':
            QMessageBox.critical(self, "错误", "请输入原文", QMessageBox.Ok)
        else:
            t = HFTree.encode(rawTextEdit.toPlainText())
            if not t:
                QMessageBox.critical(self, "错误", "存在无效字符", QMessageBox.Ok)
                return
            self.encodedTextEdit.setText(t)
```

4.2.2 文件读入

```
    def encodeFileReadin(self):
        filePath, ok = QFileDialog.getOpenFileName(self, '选择文件')
        if ok:
            with open(filePath, 'r', encoding='utf-8') as file:
                try:
                    text = file.read()
                except UnicodeDecodeError:
                    QMessageBox.critical(
                        self, "错误", "请确保打开的是UTF-8编码的文本文件", QMessageBox.Ok)
                    return
            self.rawTextEdit.setText(text)
```

4.2.3 保存原文

```
def saveRawTextContent(self):
    filePath, ok = QFileDialog.getSaveFileName(self, '选择文件')
    if ok:
        with open(filePath, 'w', encoding='utf-8') as file:
            file.write(self.rawTextEdit.toPlainText())
```

4.3 译码（Decoding）

译码则是根据内存中现有的树来对密文进行译码，而密文的读取方式有两种，一种是手动输入，一种是读取文件，而密文也可进行保存

4.3.1 译码（Decoding）

```
def decode(self, text: str) -> str:
    # 在树中对text中的01串进行解码
    root: TreeNode = self.rootnode
    result = ""
    for i in text:
        if i == '0':
            root = root.lchild
        elif i == '1':
            root = root.rchild
        elif i == '\n':  # 紧凑格式中的'\n'需忽略
            continue
        else:
            return None
        if root.name:
            result += root.name
            root = self.rootnode
    if root != self.rootnode:
        return None
    else:
        return result
def decoding(self):
    if not HFTree:
        QMessageBox.critical(self, "错误", "当前无建好的树", QMessageBox.Ok)
    elif self.encodedTextEdit.toPlainText() == '':
        QMessageBox.critical(self, "错误", "请输入密文", QMessageBox.Ok)
    else:
        t = HFTree.decode(self.encodedTextEdit.toPlainText())
        if not t:
```

```
            QMessageBox.critical(self, "错误", "存在无效字符", QMessageBox.Ok)
            return
        self.rawTextEdit.setText(t)
```

4.3.2 文件读入

```
    def decodeFileReadin(self):
        filePath, ok = QFileDialog.getOpenFileName(self, '选择文件')
        if ok:
            with open(filePath, 'r', encoding='utf-8') as file:
                try:
                    encodedTextEdit = file.read()
                except UnicodeDecodeError:
                    QMessageBox.critical(
                        self, "错误", "请确保打开的是UTF-8编码的文本文件", QMessageBox.Ok)
                    return
            if not checkDecodedText(encodedTextEdit):
                QMessageBox.critical(self, "错误", "存在无效字符", QMessageBox.Ok)
                return

 self.encodedTextEdit.setText(encodedTextEdit)
```

③ 保存密文

```
    def saveEncodedTextContent(self):
        if not checkDecodedText(self.encodedTextEdit.toPlainText()):
            QMessageBox.critical(self, "错误", "存在无效字符", QMessageBox.Ok)
            return
        filePath, ok = QFileDialog.getSaveFileName(self, '选择文件')
        if ok:
            with open(filePath, 'w', encoding='utf-8') as file:

 file.write(self.encodedTextEdit.toPlainText())
```

4.4 打印代码文件（Print）

此处我们要实现以紧凑格式输出，且要存储文件

4.4.1 紧凑格式

```
def compactFormPrint(self):
    Text = self.encodedTextEdit.toPlainText()
    text = ''
    m = 50
    for i in Text.replace('\n', ''):
        text += i
        m -= 1
        if m == 0:
            text += '\n'
            m = 50
    self.encodedTextEdit.setPlainText(text)
```

4.4.2 存储

```
def saveEncodedTextContent(self):
    if not checkDecodedText(self.encodedTextEdit.toPlainText()):
        QMessageBox.critical(self, "错误", "存在无效字符", QMessageBox.Ok)
        return
    filePath, ok = QFileDialog.getSaveFileName(self, '选择文件')
    if ok:
        with open(filePath, 'w', encoding='utf-8') as file:

 file.write(self.encodedTextEdit.toPlainText())
```

4.5 打印哈夫曼树（Tree printing）

打印哈夫曼树中，包括生成树的信息、对控件中的图像进行操作、树信息的显示以及树的导入以及存储，还有查看字符集的相关操作

4.5.1 生成树的图片

```
def printTree(self, filename=None):
    # 生成树的图片
    dot = Digraph(comment="生成的树")
    dot.attr('node', fontname="STXinwei", shape='circle', fontsize="20")
    for i, j in enumerate(self.nodes):
        if j.name == '' or not j.name:
            dot.node(str(i), '')
        elif j.name == ' ':
            dot.node(str(i), '[ ]')  # 空格显示为'[ ]'
```

```
        elif j.name == '\n':
            dot.node(str(i), '\\\\n')  # 换行符显示为'\n' 转义 此处的还会被调用，因此需要四个斜杠
        elif j.name == '\t':
            dot.node(str(i), '\\\\t')  # 制表符显示为'\t'
        else:
            dot.node(str(i), j.name)
    dot.attr('graph', rankdir='LR')
    for i in self.nodes[::-1]:
        if not (i.rchild or i.lchild):
            break
        if i.lchild:
            dot.edge(str(self.nodes.index(i)), str(
                self.nodes.index(i.lchild)), '0', constraint='true')
        if i.rchild:
            dot.edge(str(self.nodes.index(i)), str(
                self.nodes.index(i.rchild)), '1', constraint='true')
    dot.render(filename, view=False, format='svg')
```

4.5.2 树图像的放大缩小

```
class ShowSVGWidget(QWidget):
    # 自定义控件，显示svg图片
    leftClick: bool
    svgrender: QSvgRenderer
    defaultSize: QSizeF
    point: QPoint
    scale = 1

    def __init__(self, parent=None):
        super().__init__(parent)
        self.parent = parent
        # 构造一张空白的svg图像
        self.svgrender = QSvgRenderer(
            b'<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 0 0"  width="512pt" height="512pt"></svg>')
        # 获取图片默认大小
        self.defaultSize = QSizeF(self.svgrender.defaultSize())
        self.point = QPoint(0, 0)
        self.scale = 1

    def update(self):
```

```
21          # 更新图片
22          self.svgrender = QSvgRenderer("tmp.svg")
23          self.defaultSize =
    QSizeF(self.svgrender.defaultSize())
24          self.point = QPoint(0, 0)
25          self.scale = 1
26          self.repaint()
27
28      def paintEvent(self, a0: QtGui.QPaintEvent) ->
    None:
29          # 绘画事件(回调函数)
30          painter = QPainter()  # 画笔
31          painter.begin(self)
32          self.svgrender.render(painter, QRectF(
33              self.point, self.defaultSize*self.scale))
     # svg渲染器来进行绘画，(画笔，QRectF(位置，大小))(F表示
    float)
34          painter.end()
35
36      def mouseMoveEvent(self, a0: QtGui.QMouseEvent) -
    > None:
37          # 鼠标移动事件(回调函数)
38          if self.leftClick:
39              self.endPos = a0.pos()-self.startPos
40              self.point += self.endPos
41              self.startPos = a0.pos()
42              self.repaint()
43
44      def mousePressEvent(self, a0: QtGui.QMouseEvent)
    -> None:
45          # 鼠标点击事件(回调函数)
46          if a0.button() == Qt.LeftButton:
47              self.leftClick = True
48              self.startPos = a0.pos()
49
50      def mouseReleaseEvent(self, a0:
    QtGui.QMouseEvent) -> None:
51          # 鼠标释放事件(回调函数)
52          if a0.button() == Qt.LeftButton:
53              self.leftClick = False
54
55      def wheelEvent(self, a0: QtGui.QWheelEvent) ->
    None:
56          # 根据光标所在位置进行图像缩放
57          oldScale = self.scale
58          if a0.angleDelta().y() > 0:
59              # 放大
60              if self.scale <= 5.0:
```

```
                self.scale *= 1.1
        elif a0.angleDelta().y() < 0:
            # 缩小
            if self.scale >= 0.2:
                self.scale *= 0.9
        self.point = a0.pos()-(self.scale/oldScale*
(a0.pos()-self.point))
        self.repaint()
```

4.5.3 树信息的显示

```
def printInform(self):
    # 更新树的信息

 self.treeHeightlabel.setText(str(self.TreeDepth(HFTre
e)))

 self.nodeCountlabel.setText(str(len(HFTree.characters
et)*2-1))

 self.leafCountlabel.setText(str(len(HFTree.characters
et)))
```

4.5.4 树文件的读取

```
def importtree(self):
    # 将树的信息导入到图片中
    filePath, ok = QFileDialog.getOpenFileName(self,
'选择文件')
    if ok:
        with open(filePath, 'r', encoding='utf-8') as
file:
            try:
                text = file.read()
            except UnicodeDecodeError:
                QMessageBox.critical(
                    self, "错误", "请确保打开的是UTF-8编
码的文本文件", QMessageBox.Ok)
                return
        global CharacterSet
        CharacterSet = self.CharacterSet
        textlines = re.findall(r'([\s\S])\t(\S+)\t\S+
(\n|$)', text)
        # 导入后重置字符集信息，并更新内存中的树
        for i, j, _ in textlines:
            CharacterSet[i] = float(j)
        global HFTree
```

```
        if CharacterSet != {}:
            HFTree = HuffmanTree(CharacterSet)
            global showSVGWidget
            HFTree.printTree('tmp')
            showSVGWidget.update()
            self.printInform()  # 将树的信息写在面板上
```

4.5.5 树的存储

```
def savetree(self):
    # 保存树的信息
    filePath, ok = QFileDialog.getSaveFileName(self, '选择文件')
    if ok:
        with open(filePath, 'w', encoding='utf-8') as file:
            for i, j in HFTree.characterset.items():
                m = '\t'.join([i, str(j), HFTree.encode(i)])
                file.write(m+'\n')
```

4.6 网络通信（Network）

网络通信包括服务端监听接口以及等待连接、客户端建立连接、树和密文的传输、等待接收以及断开连接

4.6.1 服务端监听端口

```
def buildServerConnection(self):
    # 服务端监听端口
    try:
        # 获取端口号
        port = int(self.lineEdit.text())
    except ValueError:
        QMessageBox.critical(self, "错误", "当前无已输入的端口号", QMessageBox.Ok)
        return
    # 建立一个套接字
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        # 设置监听端口并监听
        s.bind(("0.0.0.0", port))
        s.listen()
    except OSError:
        QMessageBox.critical(self, "错误", "端口已被占用", QMessageBox.Ok)
        return
```

```
18      # 等待客户端连接
19      self.stateLabel.setText("等待连接")
20      # 开启一个新的线程用于等待连接，防止程序阻塞，并利用daemon
    标记，以便于主线程结束时，自动结束带有此标记的所有线程
21      threading.Thread(target=self.handleClient,
22                       args=[s], daemon=True).start()
```

4.6.2 服务端等待连接

```
1  def handleClient(self, s: socket.socket):
2      # 服务器端等待连接
3      c = s.accept()[0]
4      self.stateLabel.setText("已连接")
5      self.s = c
6      # 启动等待接收的线程
7      threading.Thread(target=self.waitRecv, args=[c],
    daemon=True).start()
```

4.6.3 客户端建立连接

```
1  def buildClientConnection(self):
2      # 客户端建立连接
3      try:
4          # 获取IP地址
5          ip = self.connectIpEditText.text()
6          if ip == None:
7              QMessageBox.critical(
8                  self, "错误", "当前无已输入的IP地址",
    QMessageBox.Ok)
9              return
10         port = int(self.connectPortEditText.text())
11     except ValueError:
12         QMessageBox.critical(self, "错误", "当前无已输入
    的端口号", QMessageBox.Ok)
13         return
14     s = socket.socket(socket.AF_INET,
    socket.SOCK_STREAM)
15     try:
16         s.connect((ip, port))
17     except ConnectionRefusedError:
18         QMessageBox.critical(self, "错误", "连接失败",
    QMessageBox.Ok)
19         return
20     except OSError:
21         QMessageBox.critical(self, "错误", "IP或端口错
    误", QMessageBox.Ok)
22         return
```

```
23      self.stateLabel.setText("已连接")
24      self.s = s
25      # 连接成功，启动等待接收的线程
26      threading.Thread(target=self.waitRecv, args=[s],
daemon=True).start()
```

4.6.4 树和密文的传输

```
1  def sendTree(self):
2      # 发送树
3      if not self.s:
4          QMessageBox.critical(self, "错误", "请先建立连
接", QMessageBox.Ok)
5          return
6      global CharacterSet
7      if not CharacterSet or not HFTree:
8          QMessageBox.critical(self, "错误", "当前树为
空", QMessageBox.Ok)
9          return
10     content = 't'  # 发送树的标志
11     for i, j in CharacterSet.items():
12         content += i+"\t"+str(j)+'\n'
13     # 将其转化为Byte进行发送
14     self.s.sendall(content.encode())
15     QMessageBox.information(self, "提示", "发送成功",
QMessageBox.Ok)
16
17 def sendText(self):
18     # 发送密文
19     if not self.s:
20         QMessageBox.critical(self, "错误", "请先建立连
接", QMessageBox.Ok)
21         return
22     global encodedTextEdit
23     content = encodedTextEdit.toPlainText()
24     if not checkDecodedText(content):
25         QMessageBox.critical(self, "错误", "存在无效字
符", QMessageBox.Ok)
26         return
27     self.s.sendall(('c'+content).encode())
28     QMessageBox.information(self, "提示", "发送成功",
QMessageBox.Ok)
```

4.6.5 等待连接

```
def waitRecv(self, s: socket.socket):
    # 等待接受线程
    try:
        while True:
            data = s.recv(10000000)
            # 将内容转变为str类型
            data = data.decode()
            if data[0] == 't':
                data = data[1:]
                textlines =
re.findall(r'([\s\S])\t(\S+)(\n|$)', data)
                global CharacterSet
                CharacterSet = {}
                for i, j, _ in textlines:
                    try:
                        CharacterSet[i] = float(j)
                    except ValueError:
                        self.stateLabel.setText("接收
到无用数据")

 self.tableWidget.clearContents()

 self.tableWidget.setRowCount(0)
                        CharacterSet = {}
                        return
                global HFTree
                if CharacterSet != {}:
                    HFTree =
HuffmanTree(CharacterSet)
                    self.stateLabel.setText("已收到
树")
                else:
                    self.stateLabel.setText("收到空
树")
            elif data[0] == 'c':
                self.stateLabel.setText("已收到密文")
                data = data[1:]
                self.setEncodedTextSign.emit(data)
            else:
                self.stateLabel.setText("接收到无用数
据")
    except ConnectionResetError:  # 对方断开
        self.stateLabel.setText("连接断开")
        self.s = None
    except ConnectionAbortedError:  # 自己断开
        pass
```

4.6.6 断开连接

```
def breakConnection(self):
    # 断开连接按钮事件
    try:
        self.s.close()
        self.s = None
        self.stateLabel.setText("未连接")
    except:
        pass
```

5 项目成果功能展示：

本章主要对本软件的设计细节和实测性能进行详细的介绍，能够让读者更为深入地理解 每个步骤的意义以及实现原理。

5.1 初始化（Initialization）

初始化有两种形式一种是直接根据文本框输入生成，另一种是根据通过直接导入字符，而在这两种方法的基础上，还可对列表内的字符集进行增删改查，而对于现存的字符集还可进行保存文件这一操作，最终，在关闭该窗口时，程序会根据现有的字符集来进行建树。

5.1.1 文本输入

文本的输入主要有两种途径，一种是直接根据文本框输入生成，另一种是根据通过直接导入字符。这两种输入途径没有十分明显的区别，直接导入文本的时候最终文本还是会直接显示在文本框界面中。如图二、文本初始化所示：

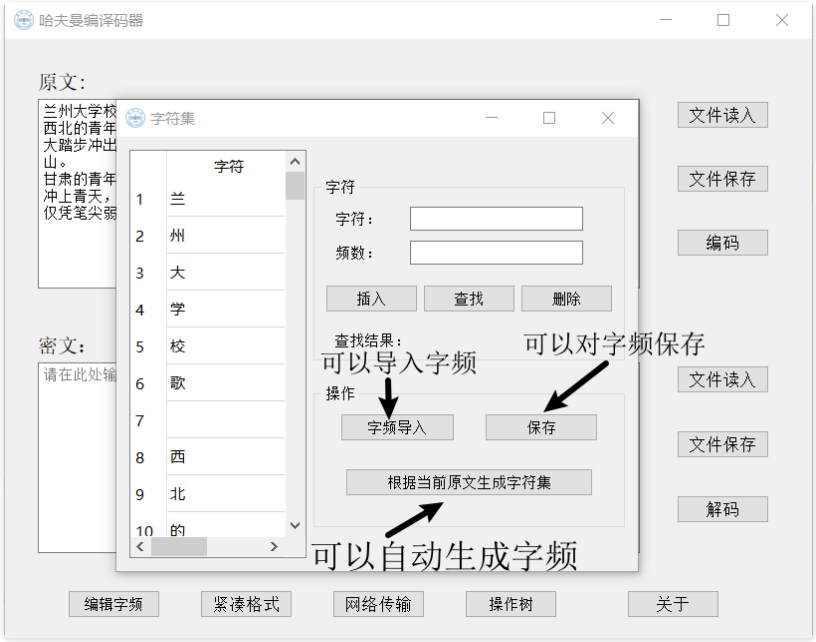
图二.文本初始化

5.1.2 额外的增删改查

在将文本输入或导入“原文：”的文本框后，还提供有随时修改文本功能。即可以在编码之前我们可以随时调整文本框内的内容，保证我们发送内容的准确性。

5.1.3 构建二叉树

在输入文本后点击编辑字频之后会弹出如图三的UI框，本程序提供自动生成字频功能、字频导入以及字频保存功能，依然提供两种方式实现字频的输入以方便用户的日常使用，使用户使用效率最大化。



图三、编辑字频

在字频统计好之后程序会根据每个字字频大小确定其权重大小，为之后的编码操作奠定基础。

5.1.4 保存字频

如上图三所示，如果文本内容较多，为了避免每次都需要自动生成字符集，这样可能会花费较多的时间，我们提供了字频保存的功能，方便提高效率。我们可以在不同环境、不同电脑上直接对保存字频生成的二叉树进行操作。

5.2 编码（Encoding）

5.2.1 编码

如图四、编码过程所示，点击编码后会将之前的字频所确定的各个字的节点生成一棵哈夫曼树（Huffmantree），之后将生成的哈夫曼树依次遍历编码即可完成对所输入内容的哈夫曼编码，生成密文。

图四、编码过程

5.2.2 保存原文

此处提供保存密文的功能，方便之后数据传输及记录。

5.3 译码（Decoding）

5.3.1 密文导入

如下图五、密文导入所示，我们在接收到别人通过网络传输发送的哈夫曼密文时，可以将密文复制进文本框中，或者选择直接导入密文

图五、密文导入

5.3.2 译码

点击译码时，循环读入一串哈夫曼序列，读到“0”从根结点的左孩子继续读，读到“1”从右孩子继续，如果读到一个结点的左孩子和右孩子是否都为0，如果是说明已经读到了一个叶子（字符），翻译一个字符成功，把该叶子结点代表的字符存在一个存储翻译字符的数组中，然后继续从根结点开始读，直到读完这串哈夫曼序列，遇到结束符便退出循环。

之后将生成的字符打印在“原文：”文本框里。实现将文本译码的功能。

图六、译码输出

5.3.3 文件读入和保存

可以将译码的文字内容通过“文件保存”存储到本地。

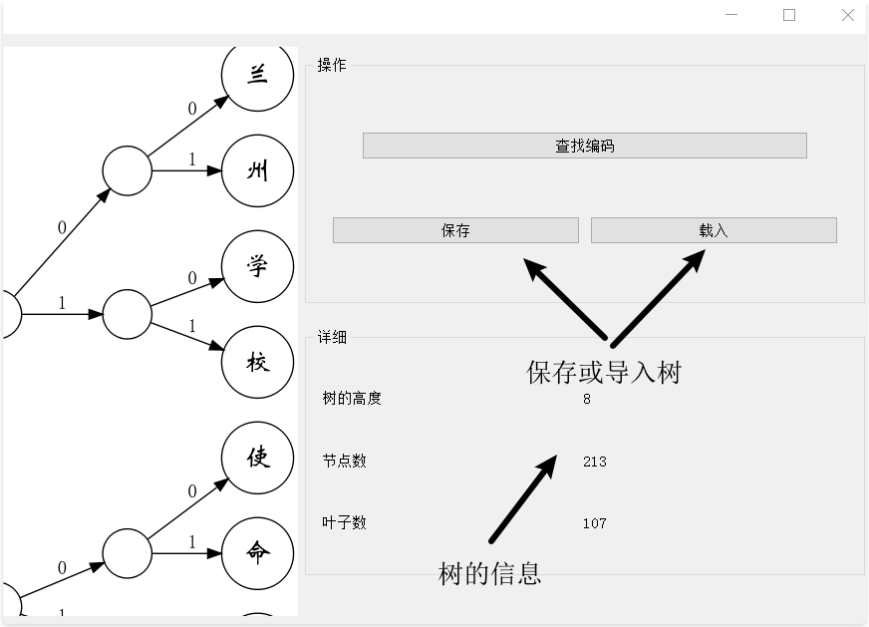
5.4 打印哈夫曼树

5.4.1 生成树的图片及存储

点击操作树，会生成如图所示的文本图片（其展示的树的图像不完全），可以“保存”和“载入”按钮进行树的存储或者导入功能。此树具备放大缩小展示的特性（缩小后不方便看清楚树的细节），在其界面角落还保留有对树的信息进行统计展示的功能。其内容主要包括树的高度，节点数以及叶子数等，本文案例中展示的“树的高度为8，树的节点数为213，树的叶子数为107”

除此以外，还提供有查找编码的功能，其主要是通过查询字符内容可以确定其字频数目，进而可以确定该字符的权重，可以反推出该字符在哈夫曼树中的位置。

如果对生成的哈夫曼树不满意，该程序还提供有对哈夫曼树中相关节点删除或者插入的功能，只要在“编辑字频”界面输入字符以及其字频（即权重），点击“插入”或“删除”按钮就可以完成对哈夫曼树的插入或删除操作。该功能对于我们需要紧急处理一些突发情况，例如在自动生成字频时发生错误，我们便可以利用此项功能对树，进而对编码实现修改。



图七、生成图的图片及存储

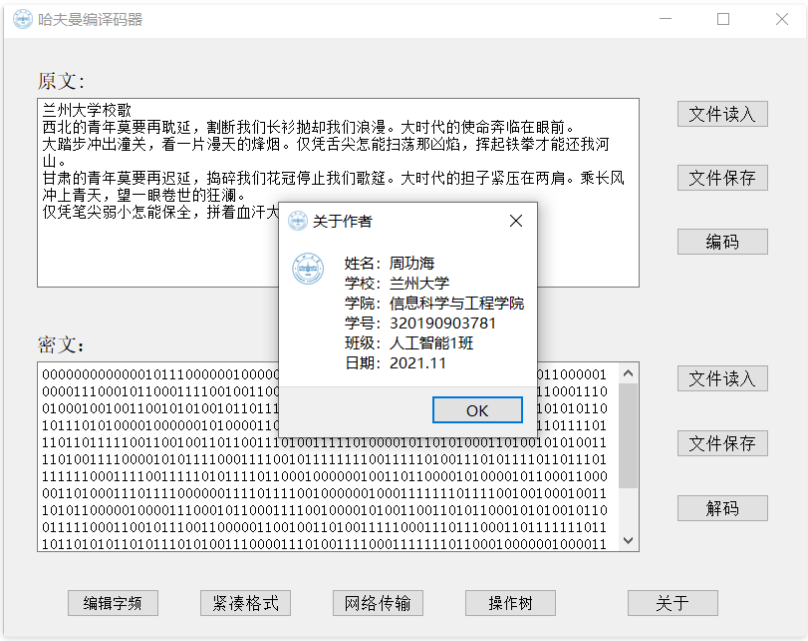
5.5 网络通信

图八、网络通信

如图八所示，本程序提供查询本机IP地址功能，可以开放本机端口作为哈夫曼编码的传输端口，连接好客户端的IP及接收端口后可以将本机的哈夫曼树和密文一起发送到客户端上，客户端可以导入哈夫曼树后运行解码操作就可以实现将文本转化的操作。

图九、网络传输哈夫曼树及密文

5.6 关于界面



图十、关于作者

由上图所示，在关于界面集中展示了我本人的相关信息，例如：姓名，学校，学院，学号，班级以及完成该作品的时间等。

5.7 项目总结

5.7.1 项目存在的优缺点：

项目缺点：

1. 无法限制密文的输入（由于textedit具有极为丰富的功能，例如带格式粘贴等，因此无法对密文的格式进行限制，只可在传输与读写时，进行内容判断）
2. 在服务端与客户端断开后，服务器端无法等待下一次重连
3. 每次接收的长度不可超过10000000
4. 由于浮点数的精度原因，可能无法将6.0与6判为相同

5. 保存图片的过程较为繁杂

项目优点：

1. 运用正则表达式对于字符、字频、端口号以及ip地址的输入进行了限制
2. 对多种非法操作进行特判
3. 根据光标所在位置进行图像缩放，并使用矢量图绘图，放大缩小时不失真

5.7.2 实验心得与体会（总结）

通过本次数据结构课程设计大作业的学习，对于数据结构中的算法有了更深的理解，尤其是关于哈夫曼树的构建以及哈夫曼编码器的编码译码，在写代码是出现了挺多的bug，但是进过不断调试之后，之前不怎么理解的代码也更加熟悉了。当自己的程序出现bug时应当先检查自己的程序是不是有一些小细节出了问题，同时通过编译器的报错来寻找错误的地方并且进行改进。如果实在寻找不出问题所在的话需要通过借阅网上资料，通过他们的经验来帮助自己完成程序的调试。

通过本次的课程设计，增强了自己单独设计程序的能力以及调试程序的能力，让自己受益匪浅。

6 优化哈夫曼编码

引言：哈夫曼编码作为一种无损数据压缩编码在计算机信息压缩中有广泛的应用。但传统的啥夫曼编码的实现方式是在构造哈夫曼树的基础上，从叶子节点向上到根节点逆向进行的。为了提高编码的效率，给出了一种新的哈夫曼编码实方式，该方式通过利用队列的数据结构，从哈夫曼树的根节点出发，向叶子节点进行编码，在编码过程中仅将哈夫曼树的每个节点进行一次扫描就可得到各叶子节点的哈夫曼编码。该方法不仅符合编码的思维方式，而且解决了原先编码过程中大量指针移动的问题，将哈夫曼编码的时间复杂度由原来的$O(n^2)$降为$O(n)$。

6.1 国内外研究哈夫曼编码的现状及方法

6.1.1 国内现状

国内已经出现利用哈夫曼编码进行图像压缩加密的密文可逆信息隐藏算法，这是一种基于自适应哈夫曼编码的密文域可逆信息隐藏算法，对不同的图像采用不同的哈夫曼码字编码腾出空间来嵌入秘密信息。首先利用自然图像相邻像素间的相关性对原始明文图像进行像素值预测，从最高有效位到最低有效位，对原始像素值和预测像素值的相同比特位进行自适应的哈夫曼编码标记。然后，利用流密码对原始明文图像进行加密。最后在腾出的空间，通过位替换来自适应的嵌入秘密信息，由于哈夫曼编码和解码的可逆性，合法接收者可以对原始明文图像和秘密信息实现分离的无损恢复和提取。

此外，Huffman code还被广泛应用于http/2 协议中，HPACK 是 HTTP/2 协议里用于头部压缩的一种技术，它又包括索引和编码两部分，其中索引是将 HTTP 头部替换为索引下标，编码则是将头部名、头部值进行哈夫曼编码，从而达到压缩头部大小，节省网络带宽和对端解析时的 CPU 消耗。

6.1.2 国外现状

国外出现了以Huffman编码为压缩方式进行像素加密的一种数据隐藏手段。在数据隐藏过程中将数据嵌入到数据媒体中。但在LSB的加密方法中，其对图像大小进行了文本限制。为了克服文本限制，他们采用Huffman编码的方式将图片进行压缩，为隐写图像提供了高嵌入能力和出色的不可察觉行，这使得文本信息更为安全。

6.2 自顶向下的哈夫曼编码算法

本算法采用对二叉树进行层次遍历的方式，利用队列对整个二叉树进行一次扫描，即可得到节点的哈夫曼编码

6.2.1 数据结构设计

6.2.1.1 哈夫曼树节点数据结构

在本结构体中，除了包含节点的被编码的信息域及其权重之外，还包含了存放节点编码的整型数组key[10]，指向其父节点的指针*par，指向其左右孩子节点的指针*Lchild和*Rchfld。具体如下：

```
typedef structNode
{
    char clata:
    int weight;
    char key[10];
    struct Node*Lehild,*RcNld,*par;
}BTNode, *BT
```

6.2.1.2 用于编码的队列的数据结构

本算法采用的是循环队列，front指向队头节点，rear指向队尾节点，count表示当前队列中节点的个数，data[]是模拟队列的数组。

```
typedef struct
{
    int front,rear;
    int count;
    BT data[queuesize]:
}drqueue;
```

6.2.2 算法描述

本算法从哈夫曼树的根节点开始，通过利用队列，按照层次遍历的方法依次对树中除根节点以外的每一个节点进行编码算法执行过程如下：

1)将哈夫曼树的根节点入队。

2)若当队列不为空，做以下操作：

(a)指针P指向当前队头节点；

(b)若当前队头节点无父节点，即根节点，则该根节点出队，并让其左孩子节点和右孩子节点先后入队；

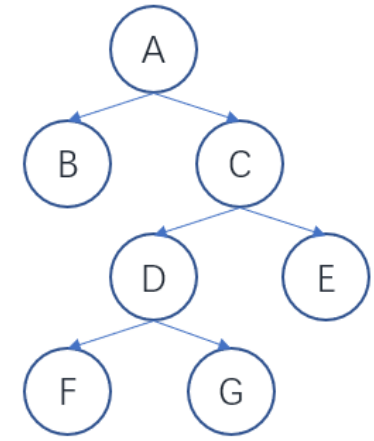
(c)若当前节点有父节点，则将父节点的哈夫曼编码赋给其左、右孩子节点，而后，若此节点为其父节点的左孩子，则在其父节点所赋给的编码后面加一个‘0’，若此节点为其父节点的右孩子，则在其父节点所赋给的编码后面加一个‘1’；由于根节点无编码，所以根节点的左右孩子节点不复制根节点的编码，直接分别得到‘O’，‘1’作为自己的编码；

(d)队头节点出队；若出队节点有左右孩子节点，则让其左右孩子分别人队，若出队节点没有左右孩子节点，转向e；

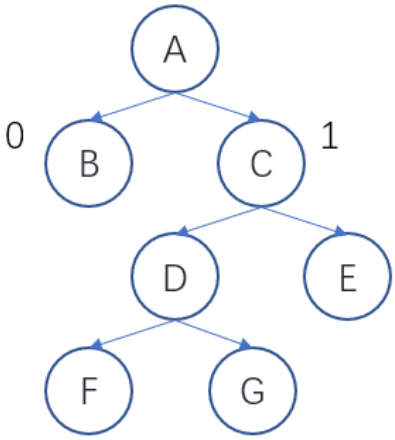
(e)判断队列是否为空。

3)若当前队列为空，则编码完成。

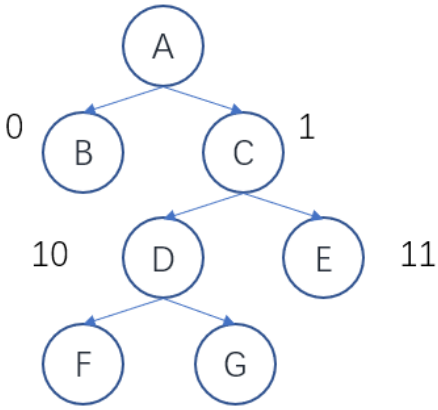
编码过程如图所示:



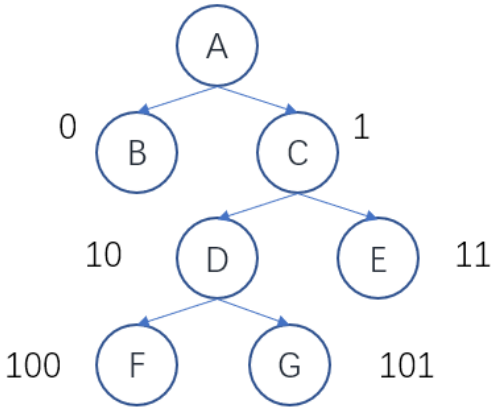
(a)编码前的哈夫曼树



(b)给第三层节点编码



(C)给第二层节点编码



(D)给第一层节点编码

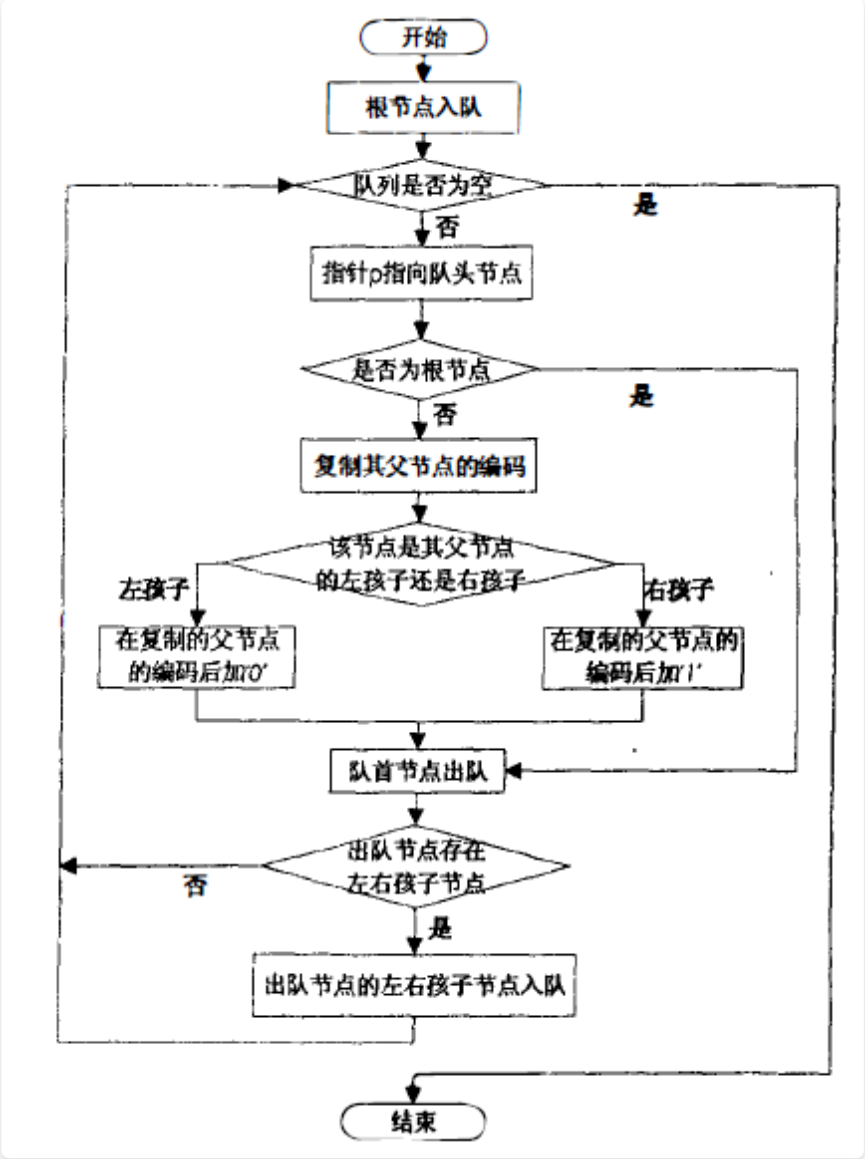


图1.算法程序框图

6.2.3 算法的具体实现(C语言)

```
BT lever(BT t)
{
    int i = 0;
    cirqueue* q;
    BT p;
    p = malloc(sizeof(p));
    p = t;
    q = (cirqueue* )malloc(sizeof(cirqueue));
    q -> rear = q -> front = q -> count = 0;
    q->data[q -> rear] = t; //根节点放入队列
    q -> count++;
    q->rear = (q->rear + 1) % queuesize; //重新设置队尾
    while (q -> count)//队列不空
    {
        if (q -> data[q->front])
        {
```

```
            p = q -> data[q->front]; //p指向当前队头节点
            if (p->par != NULL)
            {
                if (p->par->Lchild = p)
                    {
                    if (p -> par != NULL)//若存在父节点，将父节点的代码给其子节点
                        {
                    for (i = 0; p->par->key[i] = '0' || p->par -> key[i] = '1'; i++)
                        p -> key[i] = p->par -> key[i];
                        p->key[i] = '0';
                        else break;
                        }
                    else if (p->par->Rchild = p)
                        {
                        for (i = 0; p -> par->key[i] = '0'|| p->par->key[i] = '1'; i++)
                            p -> key[i] = p -> par->key[i];
                        p->key[i] = '1';
                        }
                    }
                        q->front = (q->front + 1) % queuesize; //重新设置队首
                        q->count--;//队首元素出队
                    if (q->count = queuesize)
                        printf("队列已满");
                    else
                    {
                        q->count++;
                        q->data[q -> rear] = p -> Lchild; //左孩子节点入队
                        q->rear = (q -> rear + 1) % queuesize;
                    }
                    if (q->count = queuesize)
                        printf("队列已满");
                    else
                    {
                        q->count++;
                        q->clata[q->rear] = p -> Rchild; //右孩子节点入队
                        q->rear = (q->rear + 1) % queuesize;
                    }
            }
```

```
55                          else // 若队头节点已经出队，重新设置队
    头节点
56                          {
57                          q->front = (q->front + 1) %
    queuesize;
58                          q -> count--;
59                          }
60                  }
61                  return t;
62          }
63  }
64
```

6.3 算法效率分析

若需参与编码的节点共有n个，则所建成的哈夫曼树共有$2n-1$个节点。由于本算法通过队列对节点进行编码，所以每个节点仅被扫描一次。故该算法在有$2n-1$个节点的哈夫曼树上的执行频度为$2n-1$，其时间复杂度为$O(n)$。

6.4 算法总结

哈夫曼编码是已经被证明的一种有效的墒编码方式。在诸如文本、图像、视频压缩及通信、密码等信息压缩编码标准中，哈夫曼编码被广泛使用。它的执行效率影响着其它各级编码系统的执行效率，尤其是在对数据处理能力的高要求与日俱增的情况下，更是如此。而文中介绍的这种新的哈夫曼编码方法和实现过程，不仅具有一般哈夫曼编码节省存储空间的特点，而且在算法的时间复杂度上有很大程度的改观，其应用优势更为显然。

6.5 未来展望

6.5.1 基于Huffman码的加密JPEG图像检索

学界已经有学者有提出了一种基于JPEG比特流中的Huffman码的加密JPEG图像检索方案。涉及三方：内容所有者、云服务器和授权用户。首先，内容所有者通过联合使用流密码、置换密码生成加密图像，然后将它们上传到云服务器。尤其是加密量化表之间的转换仍然有效。授权用户使用相同的密钥将加密的查询图像提交给服务器。其次，服务器从加密图像中提取Huffman码直方图作为特征。尽管在加密过程中哈夫曼码直方图发生了变化，但在特征比较后，将与查询图像内容相似的加密图像返回给用户。最后通过解密和哈希验证，用户可以获得经过认证的明文图像。实验结果表明，该方案在保证机密性、完整性和格式兼容性的同时，对不同质量因素的图像检索仍然有效。

6.5.2 生物数据压缩的 Huffman 代码

经典Huffman编码已被广泛用于压缩生物数据集。尽管通过经典的Huffman码可以获得数据大小的显着减少，但是通过考虑传输时间、能量消耗等的要求不同地处理二进制位，更有效的编码是可能的。许多技术已经修改了Huffman码算法，以获得不等字母成本的最佳前缀码，从而降低整体传输成本（时间）。在本文中，我们提出了一种新方法来提高此类扩展的压缩性能，即成本考虑方法 (CCA)，通过应用遗传算法将码字优化分配到符号。所提出的方法的想法是牺牲一些成本来最小化总位数，因此，遗传算法的工作原理是对成本进行惩罚。通过使用它来压缩一些标准生物数据集来评估该方法的性能。实验表明，所提出的方法在不显着增加成本的情况下显着提高了 CCA 的压缩性能。

6.5.3 GPU加速压缩的Huffman代码实现与优化

Huffman码是一种用于数据压缩的可变长度熵码。Huffman代码的压缩操作可以在 GPU 上非常有效地完成。然而，解压很难并行化，因为压缩后的数据没有分隔符来识别每个可变长度的码字。我们提出了一个带有间隙阵列 (GHCA) 的Huffman代码，它可以加速 GPU 解压缩。使用 Geforce RTX2080Ti GPU 的实验结果表明，GHCA 最多可能有 1.5% 的大小开销，但解压缩比之前发布的 Huffman 代码的 GPU 解码快 1.536-4713 倍。

6.5.4 使用Huffman编码的自适应放射图像压缩技术

首先，使用基于方差图像的出现直方图的自动阈值将 ROI 与图像背景分离，然后使用图像金字塔压缩和有损矢量量化 (VQ) 压缩技术以最大可能的压缩率压缩图像背景基于广义劳埃德算法（GLA）生成码本的方法。之后，使用低压缩比和最小细节损失的Huffman编码（HC）压缩ROI。最后，压缩图像是将压缩背景和压缩后的 ROI 结合起来得到的。通过计算不同压缩比下解压后原始图像和恢复图像之间的归一化互相关（NCC）和结构相似指数（SSIM）来评估结果。将获得的结果与使用有损 VQ、HC、离散余弦变换 (DCT) 和 JPEG2000 压缩方法在不进行分离的情况下压缩整个图像所获得的结果进行比较。结果表明，所提出的方法比使用VQ压缩或Huffman编码压缩完全压缩不分离的射线图像具有更可靠的性能。

6.6 哈夫曼编码与解码的缺陷

哈夫曼确实可以对数据进行压缩，但是无法逼近香农提出的熵极限。

信息熵：在信息论中，熵是接收的每条消息中包含的信息的平均量，又被称为信息熵、信源熵、平均自信息量。这里，“消息”代表来自分布或数据流中的事件、样本或特征。熵的单位通常为比特，但也用Sh、nat、Hart计量，取决于定义用到对数的底。依据Boltzmann's H-theorem，香农把随机变量X的熵值 H（希腊字母Eta）定义如下，其值域为$x_1, x_2 \ldots, x_n$：

$$H_r(x) = -\sum_{i=1}^{q} P(a_i) log_r P(a_i)$$

一般选用以2位底时，信息熵写成H(X)的形式，其中变量X是指某随机变量的整体。

r进制信息熵$H_r(x)$与二进制信息熵H(x)间的关系为

$$H_r(x) = \frac{H(x)}{logr}$$

这在计算一些字符的编码压缩极限时，很容易得出哈夫曼编码方式远高于其理论编码最短值，因此人们在此理论基础上发明了一种新的编码方式：即算术编码

7 算术编码

算术编码是 1980 年代发展起来的一种熵编码方法。

基本原理是将被编码的数据序列表示成 0 和 1 之间的一个间隔 (也就是一个小数范围)，该间隔的位置与输入数据的概率分布有关。信息越长，编码表示的间隔就越小，因而表示这一间隔所需的二进制位数就越多 (由于间隔是用小数表示的)。

算术编码是一种无损数据压缩方法，也是一种熵编码的方法。和其它熵编码方法不同的地方在于，其他的熵编码方法通常是把输入的消息分割为符号，然后对每个符号进行编码，而算术编码是直接把整个输入的消息编码为一个数，一个满足（0.0 ≤ n < 1.0）的小数n。

算术编码的压缩本质，就是在保留字符排列顺序的同时，对于更高频出现的字符，也就是概率更大的字符，赋予更大的小数区间。

简单来说，算术编码即为实现：

1. 假设有一段数据需要编码，统计里面所有的字符和出现的次数。
2. 将区间 [0,1) 连续划分成多个子区间，每个子区间代表一个上述字符， 区间的大小正比于这个字符在文中出现的概率 p。概率越大，则区间越大。所有的子区间加起来正好是 [0,1)。
3. 编码从一个初始区间 [0,1) 开始，设置:$low = 0$，$high = 1$,
4. 不断读入原始数据的字符，找到这个字符所在的区间，比如 [L, H)，更新:

$$low = low + (high - low) \times L$$
$$high = low + (high - low) \times H$$

最后将得到的区间 [low, high)中任意一个小数以二进制形式输出即得到编码的数据。

7.1 算术编码的分析及与哈夫曼编码差距

算术编码是一种从一整套符号开始并使用递归形式连续编码的方法。 算术编码不再有字母符号和码字的一一对应关系，您必须将算术码字分配给整个源符号序列（即一次不是一个数字）和码字。 它本身决定了 0 和 1 之间的实际间隔。

不论是否是二元信源，也不论数据的概率分布如何，其平均码长均能逼近信源的熵。

算术编码和Huffman的区别就在于：在算术编码中，输入序列（即被赋给单个码字的符号块）的长度，是可变的，可以说，算术编码是将可变长码字赋给可变长符号块

正是由于算术编码不需要为定长符号块分配整数长的码字，理论上能达到无损编码定理所规定的最低限。

在编码过程中，尽管在计算时有乘法运算，但可以通过移位实现，即通过加法和移位实现算术运算。在解码时，要除以符号区间概率，也可以通过移位实现，即通过减法和移位实现算术解码。这正是把这种编码方法称为算术码的原因。

算术编码从全序列出发，采用递推形式的连续编码，它不是将单个的信源符号映射成一个码字，而是将整个符号序列映射为实数轴上 [0,1）区间内的一个小区间，其长度等于该序列的概率。

随着输入符号越来越多，子区间分割越来越精细，因此表示其左端点的数值的有效位数也越来越多。

如果等整个符号序列输入完毕后再将最终得到的左端点输出，将遇到两个问题：第一，当符号序列很长时，将不能实时编解码；第二，有效位太长的数难以表示。

为了解决这个问题，通常采用两个有限精度的移位寄存器存放码字的最新部分，随着序列中符号的不断输入，不断地将其中的高位移出到信道上，以实现实时编解码。

算术编码和哈夫曼编码的相似程度很高——事实上，哈夫曼编码只是算术编码的一个特例。但是，算术编码将整个消息翻译成一个表示为指数b，而不是将消息中的每个符号翻译成一系列的以 b 为基数的数字，因此通常比哈夫曼编码更能达到最优熵编码。

因为算术编码不能一次压缩一个数据，所以在压缩iid字符串时它可以任意接近熵。相反，使用Huffman编码（到字符串）的扩展不会达到熵，除非字母符号的所有概率都是 2 的幂，在这种情况下，Huffman和算术编码都实现熵。

当Huffman编码二进制字符串时，即使熵低（例如（{0, 1}）具有概率{0.95, 0.05}），也不可能进行压缩。Huffman编码为每个值分配 1 比特，产生与输入长度相同的代码。相比之下，算术编码可以更加地压缩比特，接近最佳压缩比

$$1 - [-0.95log_2(0.95) + -0.05log_2(0.05)] \approx 71.4\%$$

解决Huffman编码次优性的一种简单方法是连接符号（阻塞）以形成新的字母表，其中每个新符号表示来自原始字母表的原始符号序列（在这种情况下是比特）。在上面的例子中，在编码之前对三个符号的序列进行分组将产生具有以下频率的新“超符号”：

- `000`: 85.7%
- `001`, `010`, `100`: 4.5% each
- `011`, `101`, `110`: 0.24% each
- `111`: 0.0125%

通过这种分组，Huffman编码每三个符号平均为 1.3 比特，或每符号 0.433 比特，而原始编码中每符号一比特，即 56.7%压缩。允许任意大的序列随意接近熵（就像算术编码一样），但需要大量代码才能这样做，因此不如算术编码那么实用。

另一种方法是通过基于Huffman的 Golomb-Rice 编码编码游程长度。这种方法允许比算术编码或甚至Huffman编码更简单和更快的编码/解码，因为后者需要表查找。在 {0.95, 0.05} 示例中，具有四位余数的 Golomb-Rice 代码实现了压缩率 71.1% ，远远低于使用三位块的压缩率。 Golomb-Rice 代码仅适用于伯努利输入，例如本例中的输入，因此它不能代替所有情况下的阻塞。

7.2 算术编码的效率

算术编码的最大优点之一在于它具有自适应性和高编码效率。

算术编码的模式选择直接影响编码效率。其模式有固定模式和自适应模式两种：

- 固定模式是基于概率分布模型进行编码。
- 自适应模式中，其各符号的初始概率都相同，但会随着符号顺序的出现而改变各符号的概率。

在无法进行信源概率模型统计的条件下，非常适于使用自适应模式的算术编码。

在信道符号概率比较均匀的情况下，算术编码的编码效率高于 Huffman 编码。但在实现上，比 Huffman 编码的计算过程复杂。

算术码也是变长码，编码过程中的移位和输出都不均匀，需要有缓冲存储器。

在误差扩散方面，也比分组码更严重。在分组码中，由于误码而破坏分组，过一段时间后常能自动恢复；但在算术码中，却往往会一直延续下去，因为它是从全序列出发来编码的。因而算术码流的传输也要求高质量的信道，或采用检错反馈重发的方式。

7.3 算术编码未来展望

7.3.1 算术编码与图像加密体制

算术编码的概念是在香农提出信息理论之后出现的。目前，算术编码应用于图像和图像数据压缩的例子很多。例如，最新的多媒体压缩标准jpeg2000、h.264等是算术编码应用的结果，借此实现了高效、安全的压缩和加密。

7.3.2 算术编码与量子通信

为了提高量子直接通信的光子利用率和通信效率，一些学者提出了一种基于算术编码的安全可行的确定性安全量子通信协议。首先使用喷泉码预先共享少量经典信息，完成测量基信息和解码信息的共享，然后对秘密信息进行简化算术编码，准备相应的单光子序列传输秘密信息信息。算术编码使得不必将窃听检测和机密信息传输分为两个步骤，提高了协议通信的效率。同时，算术编码特性使得用于窃听检测的光子无需丢弃，提高了光子利用率。该协议可以使用与 BB84 协议相同的物理设备来实现。安全性分析表明该协议具有较高的安全性。

7.3.3 使用同义词替换和算术编码的可逆自然语言水印

为了保护文本的版权并无害地恢复其原始内容，有学者提出了一种结合算术编码和同义词替换操作的新型可逆自然语言水印方法。通过分析同义词的相关频率，将承载载荷的同义词量化为不平衡的冗余二进制序列。量化后的二进制序列通过自适应二进制算术编码无损压缩，为容纳附加数据提供备用。然后，附加了水印的压缩数据通过同义词替换以可逆的方式嵌入到封面文本中。在接收端，通过对加水印文本中同义词的取值进行解码，提取水印和压缩数据，通过对提取的压缩数据进行解压，将替换后的同义词替换为原来的同义词，可以完美恢复原始上下文。同义词。实验结果表明，所提出的方法能够成功提取水印，并实现了对原始文本的无损恢复。此外，它还实现了高嵌入容量。

7.3.4 基于3D混沌映射和算术编码的改进图像选择性加密压缩技术

数字图像处理和通信的进步产生了对通过网络进行实时安全图像传输的巨大需求。然而，由于图像的海量数据容量和像素之间的高相关性等固有特征阻碍了传统联合加密压缩方法的使用，因此开发有效、快速和安全的依赖图像压缩加密系统仍然是一个研究问题。本文提出了一种用于部分图像加密压缩的新方法，它采用混沌 3D cat map 结合自适应阈值技术来去相关像素之间的关系，该技术用作有损压缩技术而不是使用复杂的量化技术，也作为一种提高密码图像安全性的替代技术。所提出的方案基于在轮廓波变换后对图像的最重要部分采用无损压缩和加密。然而，通过使用简单的阈值规则和算术编码来使图像完全无法识别，从而对最不重要的部分进行有损压缩。由于 3D 猫图对选择纯文本攻击的弱点，建议的方案结合了一种机制，根据图像的内容（上下文密钥）生成随机密钥。对基准图像进行了多次实验，以确保所提出技术的有效性。压缩分析和安全结果表明，所建议的技术对于实时图像的应用程序是有效且安全的。

7.4 实现代码

```python
# -*-coding:utf-8-*-
import time
import numpy as np
import pprint
import matplotlib.pyplot as plt
import random
alpha_dict = {
    'a': 0.0575,
    'b': 0.0128,
    'c': 0.0263,
    'd': 0.0285,
```

```
    'e': 0.0913,
    'f': 0.0173,
    'g': 0.0133,
    'h': 0.0313,
    'i': 0.0599,
    'j': 0.0006,
    'k': 0.0084,
    'l': 0.0335,
    'm': 0.0235,
    'n': 0.0596,
    'o': 0.0689,
    'p': 0.0192,
    'q': 0.0008,
    'r': 0.0508,
    's': 0.0567,
    't': 0.0706,
    'u': 0.0334,
    'v': 0.0069,
    'w': 0.0119,
    'x': 0.0073,
    'y': 0.0164,
    'z': 0.0007,
    ' ': 0.1928,
}
color = ['#dc2624', '#2b4750', '#45a0a2',
         '#e87a59', '#7dcaa9', '#649E7D',
         '#dc8018', '#C89F91', '#6c6d6c',
         '#4f6268', '#c7cccf'
         ]

# 计算信源熵
def calc_entropy(alpha_dict):
    entropy = 0
    gailv_jihe = list(alpha_dict.values())
    for i in range(len(alpha_dict)):
        entropy =
entropy+gailv_jihe[i]*np.log(gailv_jihe[i])
    return -entropy

# 计算累乘概率
def mul_pos(input_str=''):
    pre_possibility = 1
    input_list = list(input_str)
    for i in input_list:
        pre_possibility =
pre_possibility*alpha_dict.get(i)

    return pre_possibility
```

```
58
59  # 计算码长
60  def calc_machang(leic_possibility):
61      ma_length = np.ceil(-
    np.log(leic_possibility)/np.log(2))
62      return ma_length
63
64  # 十进制小数转换为二进制小数
65  def dec2bin(x):
66      x -= int(x)
67      bins = []
68      while x:
69          x *= 2
70          bins.append(1 if x >= 1. else 0)
71          x -= int(x)
72      return bins
73
74
75  # 二进制小数转换为十进制小数
76  def bin2dec(b):
77      d = 0
78      for i, x in enumerate(b):
79          d += 2**(-i-1)*x
80      return d
81
82  # 二进制小数进位
83  def bin_jinwei(input_bin=[]):
84      for i in range(len(input_bin)):
85          if input_bin[len(input_bin) - 1 - i] == 0:
86              input_bin[len(input_bin) - 1 - i] = 1
87              break
88          else:
89              input_bin[len(input_bin) - i - 1] = 0
90      return input_bin
91
92  def accu_pos(para_dict):
93      accu_pos_dict = {}
94      # print(len(alpha_dict))
95      for i in range(len(para_dict)):
96          pre_pos = 0
97          pre_alpha = list(para_dict.keys())[i]
98          if i == 0:
99              pre_pos = 0
100         else:
101             for j in range(i):
102                 pre_pos = list(para_dict.values())
    [j]+pre_pos
103         accu_pos_dict[pre_alpha] = pre_pos
```

```
104
105     return accu_pos_dict
106
107 # 计算待编码序列的累计概率
108 def calc_xulie_pos(xulie, accu_pos_dict={}):
109     xulie = list(xulie)
110     alpha_list = list(alpha_dict.keys())
111     pos_down = 0
112     pos_up = 0
113     accu = 1
114     plt.ion()
115     xianshi = ''
116     for i in range(len(xulie)):
117         xianshi = xianshi+xulie[i]
118         pos_down = pos_down +
    accu*accu_dict.get(xulie[i])
119         pos_up = pos_down +
    alpha_dict.get(xulie[i])*accu
120         accu = accu * alpha_dict.get(xulie[i])
121         # plt.xlim(pos_down,
    accu_dict[xulie[0]]+alpha_dict[xulie[0]])
122         # plt.xlim(pos_down, pos_up)
123     #     plt.ylim(0, 10)
124     #     plt.axvspan(pos_down, pos_up, 0, 0.2,
    alpha=(1-i*0.1), color=random.sample(color, 1)[0])
125     #     plt.axvline((pos_down+pos_up)/2, 0,
    0.3,color='black')
126     #     plt.text(pos_down, 5, 'P({})=
    {}'.format(xianshi, (pos_down+pos_up)/2))
127     #     plt.text(pos_down,2,'下界'+str(pos_down))
128     #     plt.text(pos_up, 3.5, '上界' +
    str(pos_down))
129     #     plt.title('概率区间')
130     #     plt.xlabel(xianshi)
131     #     plt.ylabel('概率')
132     #     plt.show()
133     #     time.sleep(1)
134     # plt.ioff()
135     # plt.close()
136     leiji_pos = (pos_down + pos_up) / 2
137     leiji_pos = pos_down
138
139     return leiji_pos
140
141 # # 解码过程
142 def decode_proceing(rec_pos, accu_pos_dict,
    machang=0):
143     pre_pos = rec_pos
```

```
    alpha_list = list(alpha_dict.keys())
    # print(alpha_list)
    jiema_result = ''
    for i in range(machang):
        for j in range(1, len(alpha_list)):
            leijigailv =
accu_pos_dict.get(alpha_list[j])
            if pre_pos > 0.8074:
                jiema_result = jiema_result+' '
                pre_pos = (pre_pos - accu_dict.get('
')) / alpha_dict.get(' ')
                break
            elif pre_pos < leijigailv:
                jiema_result =
jiema_result+alpha_list[j-1]
                pre_pos = (pre_pos-
accu_dict.get(alpha_list[j-1])) /
alpha_dict.get(alpha_list[j-1])
                break

    return jiema_result

def pos_change(leijigailv):
    bin_pos = dec2bin(leijigailv)
    return bin_pos

def bianma(bin_pos, pre_length=30):
    pre_result = []
    if len(bin_pos) == pre_length:
        pre_result = bin_pos
    elif len(bin_pos) >= pre_length:
        pre_result = bin_pos[:pre_length]
        pre_result = bin_jinwei(pre_result)

    else:

        for i in range(pre_length-len(bin_pos)):
            bin_pos.append(0)
            pre_result = bin_pos
    return pre_result

if __name__ == "__main__":
    resouse_entropy = calc_entropy(alpha_dict)
    print("信源熵为: {}".format(resouse_entropy))
    accu_dict = accu_pos(alpha_dict)
    pprint.pprint(accu_dict)
    f_test = open('test1.txt', 'r')
    text = f_test.read()
```

```
    a = ''
    total = len(text)*8
    total_ = 0
    count = 0
    count_wrong = 0
    all_unit = 0
    start_time = time.time()
    with open('result1.txt', 'a') as f:
        for i in range(len(text)):
            alp = text[i].lower()
            if alp in alpha_dict.keys():
                a += alp
                if alp == ' ':
                    all_unit += 1
                    print(a)
                    r = calc_xulie_pos(a,
accu_pos_dict=accu_dict)
                    lei_possibility = mul_pos(a)
                    N =
calc_machang(leic_possibility=lei_possibility)
                    total_ = total_+N
                    print(N)
                    print(r)
                    bin_pos = pos_change(r)
                    bin_pos =
bianma(bin_pos=bin_pos, pre_length=int(N))
                    print(bin_pos)
                    back_pos = bin2dec(bin_pos)
                    print(back_pos)
                    result =
decode_proceing(back_pos, accu_pos_dict=accu_dict,
machang=len(a))
                    print(result)
                    count += 1
                    f.write(result)
                    original_len = len(a)*8
                    print("压缩比为:
{:.2f}%".format(-N/original_len*100))
                    if a != result:
                        count_wrong += 1
                    if count == 12:
                        f.write('\n')
                        count = 0
                    a = ''
            else:
                a += ' '

    f_test.close()
```

```
    print("译码错误占比为:
{:.2f}%".format(count_wrong/all_unit*100))
    stop_time = time.time()
    run_time = stop_time - start_time
    print("运行时间为{}".format(run_time))
    print("总压缩比为:
{:.2f}%".format(total_/total*100))
```

测试内容为兰州大学校歌的英文版，如下所示：

Lanzhou University School Song
The young people in the northwest should not delay any longer, cutting off our gowns and leaving our romance behind. The mission of the great era is imminent.
Striding out of Tongguan, watching a cloud of beacon smoke. How can you sweep away the fierce flames only with the tip of your tongue?
The young people of Gansu must not delay any longer, smashing our corolla to stop our singing feast. The burden of the great era is squeezed on both shoulders. Riding the long wind to rush to the blue sky, looking at the raging waves of the world.
How can you keep the pen tip weak?

测试结果：

```
tip
16.0
0.6821662937041952
[1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1]
0.6821746826171875
tip
压缩比为:  -50.00%
译码错误占比为:  0.93%
运行时间为0.012696504592895508
总压缩比为: 52.37%
```

图1.运行结果

Reference

[1] Vitter J S. Algorithm 673: dynamic Huffman coding[J]. ACM Transactions on Mathematical Software (TOMS), 1989, 15(2): 158-167.
[2] Huffman D A. A method for the construction of minimum-redundancy codes[J]. Proceedings of the IRE, 1952, 40(9): 1098-1101.
[3] Larmore L L, Hirschberg D S. A fast algorithm for optimal length-limited Huffman codes[J]. Journal of the ACM

(JACM), 1990, 37(3): 464-473.

[4] Kanzariya N. Image Steganography for data hiding Using Huffman code, Zigzag and OPAP[J].

[5] MacKay D J C, Mac Kay D J C. Information theory, inference and learning algorithms[M]. Cambridge university press, 2003.

[6] Hao W, Xiang L, Li Y, et al. Reversible natural language watermarking using synonym substitution and arithmetic coding[J]. Comput. Mater. Contin, 2018, 55: 541-559.

[7] 朱怀宏, 吴楠, 夏黎春. 利用优化哈夫曼编码进行数据压缩的探索[J]. 微机发展, 2002, 12(5): 1-6.

[8] 李伟生, 李域, 王涛. 一种不甩建造 Huffman 树的高效 Huffman 编码算法[J]. 中国图象图形学报: A 辑, 2005, 10(3): 382-387.

[9] 王防修, 周康. 基于二叉排序树的哈夫曼编码[J]. 武汉工业学院学报, 2011, 30(4): 45-48.

[10] 孟欢, 包海燕, 潘飞. Xilinx 哈夫曼编码系统设计[J]. 电子产品世界, 2017, 24(11): 51-54.

[11] 王防修, 刘春红. 一种哈夫曼编码的改进算法[J]. 武汉轻工大学学报, 2016, 35(1): 88-91.

[12] 吕姣霖, 徐艳. 哈夫曼编码在图像压缩中的应用与分析[J]. 数字通信世界, 2021 (1): 189-190.

[13] 许子明. 哈夫曼编码译码功能的简单实现[J]. 科技风, 2018, 18.

[14] 石博文, 苑海朝, 路慧泽, 等. 基于二叉树和一维数组的哈夫曼编码[J]. 通信技术, 2017, 50(5): 867-872.

[15] 吴友情, 郭玉堂, 汤进, 等. 基于自适应哈夫曼编码的密文可逆信息隐藏算法[J]. 计算机学报, 2021, 44(4): 846-858.

[16] 王成山, 王继东. 基于能量阈值和自适应算术编码的数据压缩方法[J]. 电力系统自动化, 2004(24):56-60.

[17] 刘文松, 朱恩, 王健,等. JPEG2000算术编码器的算法优化和VLSI设计[J]. 电子学报, 2011, 39(11):2486-2491.

[18] Elia P. 無失真二值化影像壓縮使用可適性算數編碼[J]. 臺灣大學電信工程學研究所學位論文, 2019: 1-62.

[19] 孔奉波, 赖红, 熊海灵. 基于算术编码的确定性安全量子通信[J]. 光通信技术, 2019, 7.

[20] Lin C H. 可適性移動向量搜尋和編碼演算法[J]. 臺灣大學電信工程學研究所學位論文, 2020: 1-95