哈爾濱工業大學

数字媒体技术

实验报告

题　　目	编解码平台及应用
学　　院	计算机科学与技术
专　　业	软件工程
学　　号	1173710204
学　　生	陈东鑫
任 课 教 师	刘绍辉

哈尔滨工业大学计算机科学与技术学院

2020.3

实验三：编解码平台及应用

注意：请按照大家阅读文献的格式进行撰写，确保文档格式的规范性！

一、 实验内容或者文献情况介绍

- 尝试编写二进制算术编码、LZ77、LZ78、LZW 编码，并用文本数据或位图图像数 据进行基本测试
- 能对音频文件进行 mp3 编码和解码
  - 输入一段 PCM 的音频数据，然后采用 MP3 编码器对这段音频文件进行 mp3 压缩和解压缩，统计压缩前 后文件大小、压缩倍数、压缩时间
  - 理解 mp3 压缩的基本流程，查资料阅读文献画出 mp3 的处理流程图 - 能对 BMP 图像进行 JPEG 压缩，JBIG 压缩(专门针对二值图像)
  - 针对任意一幅 BMP 图像，采用 JPEG，JBIG 压缩和解压缩代码对其进行压缩和解压缩，然后阅读文献 ，写出 JPEG 压缩的基本流程
  - 了解 JPEG 系列标准的最新进展情况：https://jpeg.org
  - 尝试运行 JBIG 代码进行编解码：https://www.cl.cam.ac.uk/~mgk25/jbigkit/
- 能对视频进行 AVS、H.264 和 HEVC 压缩
  - 登录 http://www.avs.org.cn，了解 AVS 标准的最新进展；编译实验 AVS1 编码器和解码器，并用测试序列 进行实验，比较压缩前后的文件大小
  - 阅读 H.264 的基本编解码混合编码框架；编译 h.264 的参考编码器和解码器，并用测试序列进行实验， 比较压缩前后的文件大小
  - 阅读 HEVC 的基本编解码混合编码框架；编译 HEVC 的参考编码器和解码器，并用测试序列进行实验 ，比较压缩前后的文件大小
- 使用 FFMPEG 平台对视频进行编码、转码操作
  - 访问 ffmpeg 官网，学会使用 ffmpeg 进行视频操作

二、 算法简介及其实现细节

二进制算术编码：

首先将得到的数据二值化处理，这里用的是图像数据，故全部转为二进制，然后对图像进行分块，，每一块内的所有数据首尾相连视为一个数据。计算其中 0 所占的比例 ratio，作为[0,1)之间的分界点。

编码过程：

设置 low 为 0，high 为 1

若读入 1，则将 low 设置为 low+(high-low)*ratio

若读入 0，则将 high 设置为 low+(high-low)*ratio

重复执行上述过程，直到遍历所有数据

最终得到一个区间[low,high),在区间内取一个点 point，作为该块编码后的数据

编码过程代码：

```
        # 编码过程
        start = time.time()
        for i in range(block_y):
            for j in range(block_x):
                low = '0'
                high = '1'
```

```
                boundaryPoint = str(boundaryPoint_mat[i][j])

                for k in str_mat[i][j]:
                    if k == '0':
                        high = fo.add(low,
                                      fo.mul(fo.div(high, low), boundaryPoin
t))
                    if k == '1':
                        low = fo.add(low,
                                     fo.mul(fo.div(high, low), boundaryPoint
))

#                     while True:
#                         prefixLow = int(low * 10)
#                         prefixHigh = int(high * 10)
#                         if prefixLow == prefixHigh:
#                             low = low * 10 - prefixLow
#                             high = high * 10 - prefixHigh
#                             conPrefix += str(prefixLow)
#                         else:
#                             break

                while True:
                    rand = str(0.9 * random.random() + 0.1)
                    temp = fo.add(low, fo.mul(fo.div(high, low), rand))
                    if fo.compare(temp, low) > 0 and fo.compare(temp,
                                                                high) < 0:
                        coding_mat[i][j] = temp
                        break
        coding_mat = np.array(coding_mat, dtype=str)
        end = time.time()
        print('编码耗时:%fs' % (end - start))
```

解码过程：

计算分界点 boundry=low+(high-low)*ratio。观察 point 落在[low,boundry)和[boundry,high)的哪个区间内

若落在[low,boundry)内：

则 high = boundry, boundry= low+(high-low)*ratio

若落在[boundry,high)区间内：

则 low=boundry, boundry= low+(high-low)*ratio

重复执行上述过程，直到获得原始数据中的位数的数据。

解码过程代码：

```
# 解码过程
start = time.time()
encoding_mat = np.zeros((block_y, block_x)).tolist()
for i in range(block_y):
    for j in range(block_x):
        low = '0'
        high = '1'
        boundaryPoint = str(boundaryPoint_mat[i][j])
        code = coding_mat[i][j]
        length = shape_mat[i][j][0] * shape_mat[i][j][1] * cMax
        count = 0
        encoding = ''

        while count < length:
            point = fo.add(low, fo.mul(fo.div(high, low),
                                       boundaryPoint))
            if fo.compare(code, point) > 0:
                encoding += '1'
                count += 1
                low = point
            elif fo.compare(code, point) < 0:
                encoding += '0'
                count += 1
                high = point
        encoding_mat[i][j] = encoding

temp = np.zeros((self.height, self.width))
res_mat = np.zeros((self.height, self.width))
for i in range(self.height):
    for j in range(self.width):
        block_i = int(i / block)
        block_j = int(j / block)
        shape = shape_mat[block_i][block_j]
        startin = cMax * ((i % block) * shape[1] + j % block)
        endin = startin + cMax
        temp[i][j] = int(encoding_mat[block_i][block_j][startin:endin])
for i in range(8):
    res_mat[np.int0(temp / (10**i)) % 10 == 1] += 2**i

end = time.time()
print('解码耗时:%fs' % (end - start))
```

LZ77 编码：

编码过程：

为了编码待编码区， 编码器在滑动窗口的搜索缓冲区查找直到找到匹配的字符串。匹配字符串的开始字符串与待编码缓冲区的距离称为“偏移值”，匹配字符串的长度称为“匹配长度”。编码器在编码时，会一直在搜索区中搜索，直到找到最大匹配字符串，并输出(o, l)，其中 o 是偏移值， l 是匹配长度。然后窗口滑动 l，继续开始编码。如果没有找到匹配字符串，则输出(0, 0, c)，c 为待编码区下一个等待编码的字符，窗口滑动“1”。

主要步骤为：

1. 设置编码位置为输入流的开始
2. 在滑窗的待编码区查找搜索区中的最大匹配字符串
3. 如果找到字符串，输出(偏移值， 匹配长度)， 窗口向前滑动“匹配长度”
4. 如果没有找到，输出(0, 0, 待编码区的第一个字符)，窗口向前滑动一个单位
5. 如果待编码区不为空，回到步骤 2

编码过程代码：

```
1.  def coding(self):
2.      start = time.time()
3.      for i in range(self.block_y):
4.          for j in range(self.block_x):
5.              temp = self.str_mat[i][j]
6.              self.coding_mat[i][j] = self.match(temp, 5, 3)
7.      end = time.time()
8.      print('编码耗时:%fs' % (end - start))
9.
10. def match(self, string, lookahead, search):
11.     length = len(string)
12.     look_head = 0
13.     look_tail = 1
14.     search_head = 1
15.     search_tail = search_head + search
16.     res = [(0, 0, string[0])]
17.     flag = False
18.
19.     while True:
20.         for i in range(search_tail - search_head):
21.             search_this = search_tail - search_head
22.             match = string[look_head:look_tail].find(
23.                 string[search_head:search_tail - i])
24.             if match != -1:
25.                 flag = True
26.                 res.append((look_tail - look_head - match,
```

```
                            search_tail - search_head - i))
                search_tail = min(search_tail + (search_this - i), length)
                search_head = search_head + (search_this - i)
                look_tail = look_tail + (search_this - i)
                look_head = max(0, look_tail - lookahead)
                break
        if not flag:
            res.append((0, 0, string[search_head]))
            search_tail = min(search_tail + 1, length)
            search_head = search_head + 1
            look_tail = look_tail + 1
            look_head = max(0, look_tail - lookahead)
        flag = False
        if search_head >= search_tail:
            #                 print(search_head)
            #                 print(search_tail)
            break
    return res
```

解码过程：
解码过程较为简单，略
代码如下：

```
def encoding(self):
    start = time.time()
    for i in range(self.block_y):
        for j in range(self.block_x):
            temp = ''
            for k in self.coding_mat[i][j]:
                if k[0] == 0 and k[1] == 0:
                    temp += k[2]
                elif k[1] - k[0] == 0:
                    temp += temp[-k[0]:]
                else:
                    temp += temp[-k[0]:-k[0] + k[1]]
            self.encoding_mat[i][j] = temp

    temp = np.zeros((self.height, self.width))
    self.res_mat = np.zeros((self.height, self.width))
    for i in range(self.height):
        for j in range(self.width):
            block_i = int(i / self.block)
            block_j = int(j / self.block)
```

```
            shape = self.shape_mat[block_i][block_j]
            startin = self.cMax * ((i % self.block) * shape[1] + j % self.bl
ock)
            endin = startin + self.cMax
            temp[i][j] = int(self.encoding_mat[block_i][block_j][startin:end
in])
    for i in range(8):
        self.res_mat[np.int0(temp / (10**i)) % 10 == 1] += 2**i

    end = time.time()
    print('解码耗时:%f' % (end - start))
```

LZ78 编码：

在压缩时维护一个动态词典 Dictionary，其包括了历史字符串的 index 与内容；压缩情况分为三种：

若当前字符 c 未出现在词典中，则编码为(0, c)；

若当前字符 c 出现在词典中，则与词典做最长匹配，然后编码为(prefixIndex,lastChar)，其中，prefixIndex 为最长匹配的前缀字符串，lastChar 为最长匹配后的第一个字符；

为对最后一个字符的特殊处理，编码为(prefixIndex,)。

```
def coding(self):
    start = time.time()
    for i in range(self.block_y):
        for j in range(self.block_x):
            self.coding_mat[i][j] = self.match(self.str_mat[i][j])
    end = time.time()
    print('编码耗时:%fs'%(end-start))

def match(self,message):
    tree_dict = {}
    m_len = len(message)
    i = 0
    while i < m_len:
        # case I
        if message[i] not in tree_dict.keys():
            yield (0, message[i])
            tree_dict[message[i]] = len(tree_dict) + 1
            i += 1
        # case III
        elif i == m_len - 1:
            yield (tree_dict.get(message[i]), '')
            i += 1
```

```
        else:
            for j in range(i + 1, m_len):
                # case II
                if message[i:j + 1] not in tree_dict.keys():
                    yield (tree_dict.get(message[i:j]), message[j])
                    tree_dict[message[i:j + 1]] = len(tree_dict) + 1
                    i = j + 1
                    break
                # case III
                elif j == m_len - 1:
                    yield (tree_dict.get(message[i:j + 1]), '')
                    i = j + 1
    return tree_dict
```

解压缩能根据压缩编码恢复出（压缩时的）动态词典，然后根据 index 拼接成解码后的字符串。

解码过程代码如下：

```
def encoding(self):
    start = time.time()

    for i in range(self.block_y):
        for j in range(self.block_x):
            self.encoding_mat[i][j] = self.unmatch(self.coding_mat[i][j])

    temp = np.zeros((self.height, self.width))
    self.res_mat = np.zeros((self.height, self.width))
    for i in range(self.height):
        for j in range(self.width):
            block_i = int(i / self.block)
            block_j = int(j / self.block)
            shape = self.shape_mat[block_i][block_j]
            startin = self.cMax * ((i % self.block) * shape[1] + j % self.bl
ock)
            endin = startin + self.cMax
            temp[i][j] = int(self.encoding_mat[block_i][block_j][startin:end
in])
    for i in range(8):
        self.res_mat[np.int0(temp / (10**i)) % 10 == 1] += 2**i

    end = time.time()
    print('解码耗时:%f' % (end - start))

```

```
def unmatch(self,packed):
    unpacked = ''
    tree_dict = {}
    for index, ch in packed:
        if index == 0:
            unpacked += ch
            tree_dict[len(tree_dict) + 1] = ch
        else:
            term = tree_dict.get(index) + ch
            unpacked += term
            tree_dict[len(tree_dict) + 1] = term
    return unpacked
```

LZW 编码：

思想，把出现的字符串映射为记号，实现压缩。同时只保存这一记号序列，不保存映射，因为其自解释的特性，根据记号序列即可还原出原本的数据

编码过程：

编码器从原字符串不断地读入新的字符，并试图将单个字符或字符串编码为记号 (Symbol)。这里我们维护两个变量，一个是 P (Previous)，表示手头已有的，还没有被编码的字符串，一个是 C (current)，表示当前新读进来的字符。

1. 初始状态，字典里只有所有的默认项，例如 0->a，1->b，2->c。此时 P 和 C 都是空的。
2. 读入新的字符 C，与 P 合并形成字符串 P+C。
3. 在字典里查找 P+C，如果：
   - P+C 在字典里，P=P+C。
   - P+C 不在字典里，将 P 的记号输出；在字典中为 P+C 建立一个记号映射；更新 P=C。
4. 返回步骤 2 重复，直至读完原字符串中所有字符。

代码如下：

```
def coding(self):
    start = time.time()
    for i in range(self.block_y):
        for j in range(self.block_x):
            self.coding_mat[i][j] = self.match(self.str_mat[i][j])
    end = time.time()
    print('编码耗时:%fs' % (end - start))

def match(self, string):
    dictionary = {'0': 0, '1': 1}
    P = ''
    C = ''
    res = []
```

```
    count = 1
    for i in range(len(string)):
        C = string[i]
        if P + C in dictionary.keys():
            P = P + C
        else:
            res.append(dictionary[P])
            count = count + 1
            dictionary[P + C] = count
            P = C
        if i == len(string) - 1:
            res.append(dictionary[P])
    return res
```

解码过程：

解码器的输入是压缩后的数据，即记号流 (Symbol Stream)。类似于编码，我们仍然维护两个变量 pW (previous word) 和 cW (current word)，后缀 W 的含义是 word，实际上就是记号 (Symbol)，一个记号就代表一个 word，或者说子串。pW 表示之前刚刚解码的记号；cW 表示当前新读进来的记号。

1. 初始状态，字典里只有所有的默认项，例如 0->a，1->b，2->c。此时 pW 和 cW 都是空的。
2. 读入第一个的符号 cW，解码输出。注意第一个 cW 肯定是能直接解码的，而且一定是单个字符。
3. 赋值 pW=cW。
4. 读入下一个符号 cW。
5. 在字典里查找 cW，如果：
   a. cW 在字典里：
      (1) 解码 cW，即输出 Str(cW)。
      (2) 令 P=Str(pW)，C=Str(cW)的**第一个字符**。
      (3) 在字典中为 P+C 添加新的记号映射。
   b. cW 不在字典里：
      (1) 令 P=Str(pW)，C=Str(pW)的**第一个字符**。
      (2) 在字典中为 P+C 添加新的记号映射，这个新的记号一定就是 cW。
      (3) 输出 P+C。
6. 返回步骤 3 重复，直至读完所有记号。

代码：

```
def encoding(self):
    start = time.time()
    for i in range(self.block_y):
        for j in range(self.block_x):
            self.encoding_mat[i][j] = self.unmatch(self.coding_mat[i][j])

```

```
    temp = np.zeros((self.height, self.width))
    self.res_mat = np.zeros((self.height, self.width))
    for i in range(self.height):
        for j in range(self.width):
            block_i = int(i / self.block)
            block_j = int(j / self.block)
            shape = self.shape_mat[block_i][block_j]
            startin = self.cMax * (
                (i % self.block) * shape[1] + j % self.block)
            endin = startin + self.cMax
            temp[i][j] = int(
                self.encoding_mat[block_i][block_j][startin:endin])
    for i in range(8):
        self.res_mat[np.int0(temp / (10**i)) % 10 == 1] += 2**i

    end = time.time()
    print('解码耗时:%fs' % (end - start))
    return

def unmatch(self, sequence):
    string = ''
    dictionary = {'0': 0, '1': 1}
    pw = -1
    cw = sequence[0]
    P = ''
    C = ''
    count = 1
    index = 0
    string += self.get_key(dictionary, cw)
    for i in sequence[1:]:
        pw = cw
        cw = i
        key = self.get_key(dictionary, cw)
        if key != '':
            string += key
            P = self.get_key(dictionary, pw)
            C = self.get_key(dictionary, cw)[0]
            count=count+1
            dictionary[P+C] = count
        else:
            P = self.get_key(dictionary, pw)
            C = self.get_key(dictionary, pw)[0]
            count=count+1
            dictionary[P+C] = count
```

```
            string = string+P+C

    return string

def get_key(self, dictionary, value):
    for k, v in dictionary.items():
        if v == value:
            return k
    return ''
```

三、 实验设置及结果分析（包括实验数据集）

所用的图像是

结果输出到控制台，如下所示

二进制算术编码:
预处理耗时:0.289154s
编码耗时:32.863515s
解码耗时:51.240695s
检查是否出错
完全正确
LZ77 编码:
预处理耗时:0.307250s
编码耗时:0.935960s
解码耗时:0.391446
检查是否出错
完全正确
LZ78 编码:
预处理耗时:0.311201s
编码耗时:0.013949s
解码耗时:0.781455
检查是否出错

完全正确
LZW 编码:
预处理耗时:0.282032s
编码耗时:0.474167s
解码耗时:1.046911s
检查是否出错
完全正确

音频的 mp3 编码解码：
音频时长 382825ms
压缩前文件大小:67530614B
压缩后文件大小:6126280B
压缩耗时:12.844171s
解压耗时:0.142241s
压缩倍率:11.023103

图像的编码解码：
图像宽高:(1292,1929)
压缩前图像大小:7478150B
压缩后图像大小:437760B
压缩耗时:0.083579s
解压耗时:0.062831s
压缩倍率:17.082762