

《数字图像处理》实验报告

姓名：汪江豪 学号：22121630

实验 6

一. 任务 1

编程对图片 bridge.bmp 和 web.bmp 进行压缩.

1. 采用哈夫曼编码，实现压缩和解压缩
 2. 采用无损预测编码，并对误差进行哈夫曼编码，实现压缩和解压缩（该小题选做）
 3. 用平均均方误差的平方根（如下），对解压缩后的图像和原图进行比较，并计算压缩比
- a) 核心代码：

核心部分--哈夫曼编码部分：

```
class HuffmanNode:  
    def __init__(self, freq, symbol, left=None, right=None):  
        self.freq = int(freq)  
        self.symbol = int(symbol)  
        self.left = left  
        self.right = right  
        self.huff = None  
  
    # 重载小于运算符, 用于在优先队列中比较节点的频率  
    def __lt__(self, other):  
        return self.freq < other.freq  
  
# 构建哈夫曼树  
def build_huffman_tree(frequency):  
    heap = [HuffmanNode(freq, symbol) for symbol, freq in frequency.items()]  
    heapq.heapify(heap) # 将列表转换为优先队列  
    while len(heap) > 1:  
        left = heapq.heappop(heap)  
        right = heapq.heappop(heap)  
        left.huff = 0  
        right.huff = 1  
        newNode = HuffmanNode(  
            left.freq + right.freq, left.symbol + right.symbol, left, right  
        )  
        heapq.heappush(heap, newNode)  
    return heap[0]
```

```

# 递归构建哈夫曼编码,
def build_codes(node, val=None):
    if val is None:
        val = bitarray()
    # 构造哈夫曼键值对[symbol:编码]
    codes = {}
    if node:
        newVal = val.copy()
        if node.huff is not None:
            newVal.append(node.huff)
        if not node.left and not node.right:
            codes[node.symbol] = newVal
        codes.update(build_codes(node.left, newVal))
        codes.update(build_codes(node.right, newVal))
    return codes

# 哈夫曼编码
def huffman_encoding(data):
    # 从大到小给出[像素值: 频率]
    frequency = Counter(data)
    huffman_tree = build_huffman_tree(frequency)
    # 是一个字典[symbol:编码]
    huffman_codes = build_codes(huffman_tree)
    encoded_data = bitarray()

    cnt = 0
    for symbol in data:
        encoded_data += huffman_codes[symbol]
        cnt += 1
    print(f"编码完毕,{cnt}个像素值已编码")
    return encoded_data, huffman_codes

# 哈夫曼解码
def huffman_decoding(encoded_data, huffman_codes, length):
    # reverse_codes = [编码: symbol]
    reverse_codes = {code.to01(): symbol for symbol, code in huffman_codes.items()}
    decoded_data = []
    current_code = ""
    cnt = 0
    for bit in encoded_data:
        current_code += "1" if bit else "0"
        # 检索字典中的键
        if current_code in reverse_codes:
            decoded_data.append(reverse_codes[current_code])
            current_code = ""
    return decoded_data, length

```

```

        cnt += 1
        if cnt == length:
            break
        current_code = ""
print(f"解码完毕,{len(decoded_data)}个像素值已解码")
return decoded_data

```

使用哈夫曼压缩和解压图像部分：

```

# 压缩图片, 哈夫曼方式
def compress_image(img, output_path, huffman_codes_path):
    shape = img.shape
    # 将图片像素值转换为一维数组
    flat_img = img.flatten()
    print(f"源图像的像素个数: {flat_img.size}")
    encoded_data, huffman_codes = huffman_encoding(flat_img)
    # 写入 encoded_data
    with open(output_path, "wb") as f:
        encoded_data.tofile(f)
    # 写入 huffman_codes
    with open(huffman_codes_path, "w") as f:
        for symbol, code in huffman_codes.items():
            f.write(f"{symbol} : {code}\n")
    return huffman_codes, shape

# 解压图片, 哈夫曼方式
def decompress_image(encoded_path, huffman_codes, shape):
    encoded_data = bytearray()
    with open(encoded_path, "rb") as f:
        encoded_data.fromfile(f) # 从文件中读取 bit
    length = shape[0] * shape[1]
    decoded_data = huffman_decoding(encoded_data, huffman_codes, length)
    decoded_img = np.array(
        [int(pixel) for pixel in decoded_data], dtype=np.uint8
    ).reshape(shape)
    return decoded_img

```

使用无损预测编码对图像压缩的核心部分：

```

# 无损预测编码
def predictive_encoding(img):
    height, width = img.shape
    # 创建一个和原图像大小相同的数组
    img = img.astype(np.int16)
    pred_error = np.zeros((height, width), dtype=np.int16)
    for i in range(height):
        for j in range(width):

```

```

        if j == 0:
            pred_error[i][j] = img[i][j]
        else:
            pred_error[i][j] = img[i][j] - img[i][j - 1]
    return pred_error
# 无损预测解码
def predictive_decoding(pred_error):
    height, width = pred_error.shape
    img = np.zeros((height, width), dtype=np.int16)
    for i in range(height):
        for j in range(width):
            if j == 0:
                img[i, j] = pred_error[i, j]
            else:
                img[i, j] = pred_error[i, j] + img[i, j - 1]
    return img

```

无损预测—误差生成哈夫曼编码，对图像压缩和解压部分：

```

# 压缩图片, 无损预测方式
def compress_image_pre(img, output_path, huffman_codes_path):
    pred_error = predictive_encoding(img)
    flat_error = pred_error.flatten()
    encoded_data, huffman_codes = huffman_encoding(flat_error)
    with open(output_path, "wb") as f:
        encoded_data.tofile(f)
    with open(huffman_codes_path, "w") as f:
        for symbol, code in huffman_codes.items():
            f.write(f"{symbol} : {code.to01()}\n")
    return huffman_codes, pred_error.shape

# 解压图片, 无损预测方式
def decompress_image_pre(encoded_path, huffman_codes, shape):
    encoded_data = bitarray()
    with open(encoded_path, "rb") as f:
        encoded_data.fromfile(f)
    length = shape[0] * shape[1]
    decoded_data = huffman_decoding(encoded_data, huffman_codes, length)
    pred_error = np.array(
        [int(pixel) for pixel in decoded_data], dtype=np.int16
    ).reshape(shape)
    decoded_img = predictive_decoding(pred_error)
    return decoded_img

```

计算 RMSE（均方根误差）部分：

```
def calculate_rmse(original, decompressed):
```

```
mse = np.mean((original - decompressed) ** 2)

rmse = np.sqrt(mse)

return rmse
```

b) 实验结果截图

1. 哈夫曼编码压缩、解压图像:

(1) web.bmp

```
PS D:\Desktop\_DIP\dip_test6\code> & D:/Python/python.exe d:/Desktop/DIP/dip_test6/code/task1.py
开始压缩
源图像的像素个数: 12121088
编码完毕,12121088个像素值已编码
压缩时间: 2.3703秒
开始解压缩
解码完毕,12121088个像素值已解码
解压缩时间: 8.4349秒
RMSE: 0.0
```

	web_compressed.txt	2024/10/18 16:04	文本文档 11,243 KB
	web_decompressed.bmp	2024/10/18 16:04	BMP 文件 11,839 KB
	web_huffman_codes.txt	2024/10/18 16:04	文本文档 7 KB

web.bmp	2024/10/17 12:45	BMP 文件	11,839 KB
---------	------------------	--------	-----------

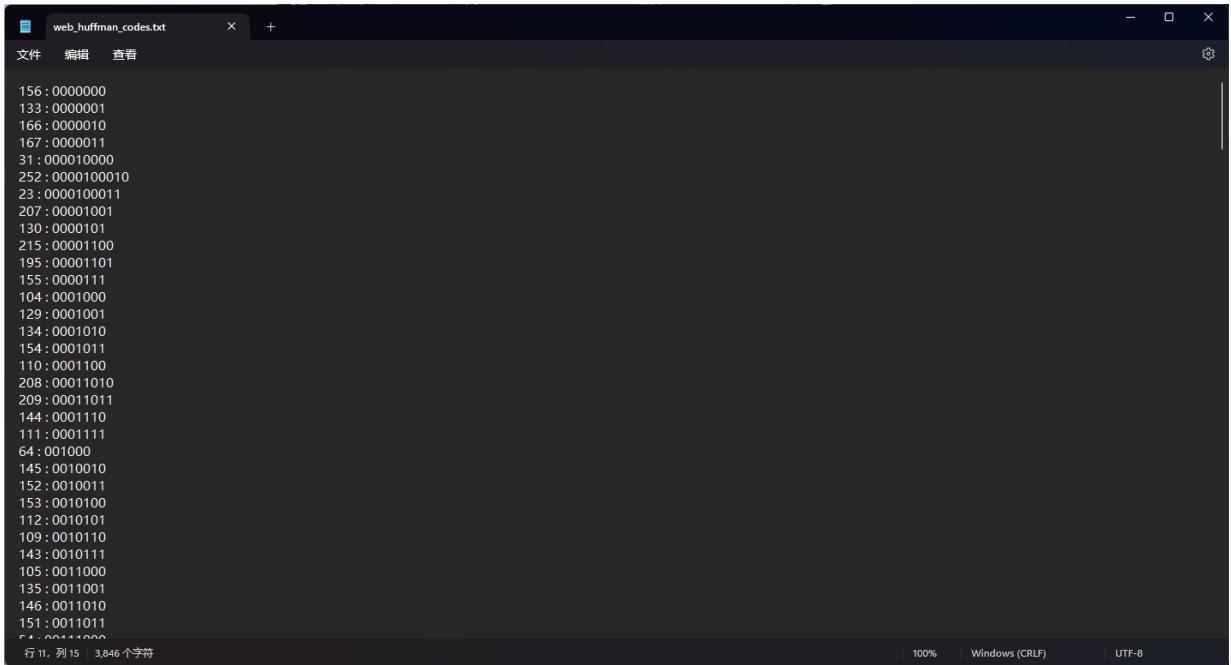
由于哈夫曼编码是无损压缩，所以 RMSE=0，压缩、解压缩后和源文件一样大。

生成的文件中：

`web_compressed.txt` 是将 `web` 图像像素值平铺展开后，经过哈夫曼编码的二进制 bit 流文件。

windows 无法直接查看，打开会显示乱码。其体积为 10.9MB，相比于源图像 11.5MB，压缩比为 94.8%

web_huffman_codes.txt 存储了 web 图像中所有像素值对应的哈夫曼编码。文件体积 3.99KB，共 249 行，也即源图像由 249 种不同灰度值组成。



```
156:0000000  
133:0000001  
166:0000010  
167:0000011  
31:000010000  
252:0000100010  
23:0000100011  
207:00001001  
130:0000101  
215:00001100  
195:00001101  
155:0000111  
104:0001000  
129:0001001  
134:0001010  
154:0001011  
110:0001100  
208:00011010  
209:00011011  
144:0001110  
111:0001111  
64:001000  
145:0010010  
152:0010011  
153:0010100  
112:0010101  
109:0010110  
143:0010111  
105:0011000  
135:0011001  
146:0011010  
151:0011011  
54:0011100  
行 11, 列 15 3,846 个字符
```

(2) bridge.bmp

```
PS D:\Desktop\DIPI\diptest6\code> & D:/Python/python.exe d:/Desktop/DIPI/dip_test6/code/task1.py  
开始压缩  
源图像的像素个数: 11130701  
编码完毕, 11130701个像素值已编码  
压缩时间: 2.2270秒  
开始解压缩  
解码完毕, 11130701个像素值已解码  
解压缩时间: 8.1163秒  
RMSE: 0.0
```

bridge_compressed.txt	2024/10/18 16:05	文本文档	10,133 KB
bridge_decompressed.bmp	2024/10/18 16:05	BMP 文件	10,883 KB
bridge_huffman_codes.txt	2024/10/18 16:05	文本文档	8 KB
bridge.bmp	2024/10/17 12:45	BMP 文件	10,883 KB

同理，哈夫曼编码为无损压缩，解压后图片大小不变，故 RMSE 为 0.

压缩后生成的 bridge_compressed.txt 为二进制 bit 文件，同样 windows 无法直接查看。后续不再展示该类文件。该文件大小为 9.89MB，相比于源图像 10.6MB，压缩比为 93.3%。

Bridge_huffman_codes.txt 中存储了各个像素值对应的哈夫曼编码。共 256 行，表示源图像由 256

个灰度级构成。

```
bridge_huffman_codes.txt +  
文件 编辑 查看  
175 : 000000000  
233 : 00000000100  
232 : 00000000101  
231 : 00000000110  
229 : 00000000111  
113 : 00000001  
112 : 00000010  
197 : 0000001100  
251 : 00000011010  
227 : 00000011011  
196 : 0000001110  
195 : 0000001111  
111 : 00000100  
110 : 00000101  
110 : 00000101  
90 : 0000011  
174 : 000010000  
228 : 00001000100  
226 : 00001000101  
194 : 0000100011  
109 : 00001001  
89 : 0000101  
4 : 000011  
173 : 000100000  
193 : 00010000010  
223 : 00010000110  
225 : 00010000111  
108 : 00010001  
107 : 000100010  
222 : 00010011000  
224 : 00010011001  
192 : 0001001101  
172 : 000100111  
.. .. ..  
行 10, 列 18 3,931 个字符 100% Windows (CRLF) UTF-8
```

2. 无损预测编码压缩、解压缩图像：

(1) web.bmp

```
PS D:\Desktop\_DIP\dip_test6\code> & D:/Python/python.exe d:/Desktop/DIP/dip_test6/code/task1.py  
开始压缩  
编码完毕,12121088个像素值已编码  
压缩时间: 8.4817秒  
开始解压缩  
解码完毕,12121088个像素值已解码  
解压缩时间: 7.1608秒  
RMSE: 0.0
```

web_compressed_pre.txt	2024/10/18 15:53	文本文档	3,580 KB
web_decompressed_pre.bmp	2024/10/18 15:53	BMP 文件	11,839 KB
web_huffman_codes_pre.txt	2024/10/18 15:53	文本文档	9 KB
web.bmp	2024/10/17 12:45	BMP 文件	11,839 KB

无损预测编码方式也为无损压缩，故解压后的图像 RMSE 计算为 0.

Web_compressed_pre.txt 为无损预测编码的误差进行哈夫曼编码后的结果，文件大小为 3.49MB，相比于源图像 11.5MB，压缩比为 30.3%，效果极好。

Web_huffman_codes.txt 为无损预测编码的各个误差值的哈夫曼编码。

web_huffman_codes_pre.txt			
文件	编辑	查看	
0:0			
1:10			
2:1100			
9:11010000000			
-16:1101000000100			
-41:110100000010100000			
190:110100000010100001			
40:110100000010100000			
44:1101000000101000110			
85:11010000001010001110			
-52:11010000001010001111			
28:1101000000101001			
56:110100000010101000			
132:11010000001010100100			
73:11010000001010100101			
-49:1101000000101010011			
34:11010000001010101			
254:11010000001010110			
98:110100000010101100			
54:1101000000101011010			
70:1101000000101011011			
39:110100000010101111			
-20:11010000001011			
19:11010000001100			
155:110100000011010000			
52:1101000000110100000			
115:110100000011010000110			
170:110100000011010000111			
-38:110100000011010001			
100:110100000011010010			
-44:1101000000110100110			
194:1101000000110100111			
74:11010000001101001111			
行 8, 列 24 8,740 个字符	100%	Windows (CRLF)	UTF-8

(2) bridge.bmp

```
PS D:\Desktop\DIPI\dip_test6\code> & D:/Python/python.exe d:/Desktop/DIP/dip_test6/code/task1.py
开始压缩
编码完毕,11130701个像素值已编码
压缩时间: 7.8136秒
开始解压缩
解码完毕,11130701个像素值已解码
解压缩时间: 9.6155秒
RMSE: 0.0
```

bridge_compressed_pre.txt	2024/10/18 15:58	文本文档	6,482 KB
bridge_decompressed_pre.bmp	2024/10/18 15:59	BMP 文件	10,883 KB
bridge_huffman_codes_pre.txt	2024/10/18 15:58	文本文档	9 KB
bridge.bmp	2024/10/17 12:45	BMP 文件	10,883 KB

同理，无损预测编码压缩为无损压缩，故解压后的图像和源图像等大，RMSE 为 0.

Bridge_compressed_pre.txt 为无损预测编码的误差的哈夫曼编码，同样为二进制 bit 流文件，windows

无法直接查看。文件大小为 6.32MB，相比于源图像 10.6MB，压缩比达到了 59.6%，效果较好。

Bridge_huffman_codes_pre.txt 文件存储了无损压缩中各个误差对应的哈夫曼编码。

```
0:000
10:001000
-16:00100100
16:00100101
-26:0010011000
26:0010011001
-20:001001101
20:001001110
-32:00100111100
32:00100111101
57:00100111100000
-73:00100111100001000
194:001001111000010010000
207:001001111000010010001
177:00100111100001001001
210:001001111000010010100
204:001001111000010010101
254:0010011110000100101100
247:0010011110000100101101
237:0010011110000100101110
199:0010011110000100101111
149:001001111000010011
-67:0010011110000101
64:001001111000011
-50:0010011110001
-44:0010011111001
-38:0010011111101
38:001001111110
-55:001001111111000
-71:0010011111110001000
-79:00100111111100100100
-82:00100111111100100101
111:001001111111100100110
```

c) 实验小结

由本次实验可知，哈夫曼编码压缩和无损预测编码压缩，两者均是无损压缩，不存在数据丢失。

如果直接对原图像的所有像素值进行哈夫曼编码，进行压缩，压缩比并不高。编码的平均码长接近图像的熵。

但如果首先对源图像的像素值进行无损预测编码，生成误差像素值，再对误差像素值进行哈夫曼编码，进行压缩，可以达到良好的压缩比。因为无损预测编码可以用邻近的像素来预测当前像素，所得误差像素值的熵比原图像小，码长更短。