Image Processing - Assignment 3

JPEG Codec

Beryl Li

1 Introduction

In assignment 3, I implement some components of JPEG codec including the three main blocks: Mapper, Quantizer, and Symbol Coder. The following methods appear in both encoder and decoder: Block based transform coding, Quantization of DCT coefficients, Huffman coding.

2 Methods

2.1 Block based transform coding

Block based transform coding will use discrete cosine transform to convert the values into frequency domain in encoder. In decoder, it will convert from frequency domain back to original values. Usually, we will distribute more bits to low frequency coefficients, such as DC term, because it is more important and fewer bits to high frequency coefficients.

2.2 Quantization of DCT coefficients

Quantization is applied after we convert the values into frequency domain. Quantization is a many-to-one mapping. Therefore, some different input will become the same after quantization. The process that makes output bins less than input bins is the part that makes JPEG codec lossy.

2.3 Huffman coding

Huffman coding converts each symbol into a bitstring according to their frequency. Every bitstring can have different length. A more frequent symbol presents fewer bits in Huffman coding, while a less frequent symbol requires more bits.

3 Experiments

When encoding, the input image will be converted into YCbCr color space from RGB color space first. Next, the image will be divided into several blocks and the size of each block is 8×8. For each block, its range will be converted into [-128, 127] from [0, 255] and then the DCT transform is applied. DCT coefficients are quantized according to the given quantization table. Lastly, DC term and AC term will be encoded by huffman

coding algorithm.
When decoding, not only the bitstring but also the huffman code dictionary are required because the dictionary is generated by the input image. The following procedure in decoding is the reversed procedure in encoding.

3.1 Experiment 1: sprint

In experiment 1, small artifacts can be detected after zooming-in (Fig 2), such as distortion of character in the left most image, small black dots appeared in the middle and right image.

(a) Original sprint image (b) Compressed sprint image

Figure 1: Sprint
Compression ratio: 6.77 : 1
RMSE: 4.08
SNR: 122.34

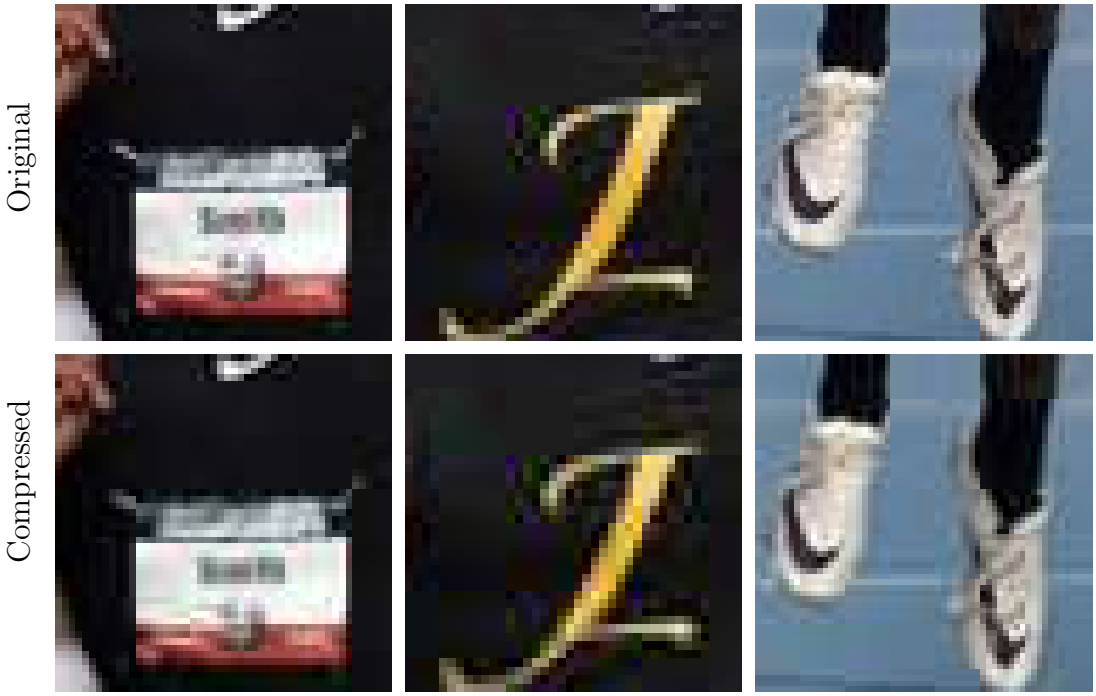

Figure 2: Sprint with zoomed-in details

3.2 Experiment 2: coins

In experiment 2, I compared three the Coins images, which are original image, compressed image with all coefficients, and compressed image with only 1/4 coefficients.

(a) Coins image (Original)

(b) Compressed coins image with all coefficients

(c) Compressed coins image with 1/4 coefficients

Figure 3: Coins

Table 1: Coins compression table

	Fig 3b	Fig 3c
Compression ratio	6.22 : 1	28.50 : 1
RMSE	6.45	7.75
SNR	416.60	79.23

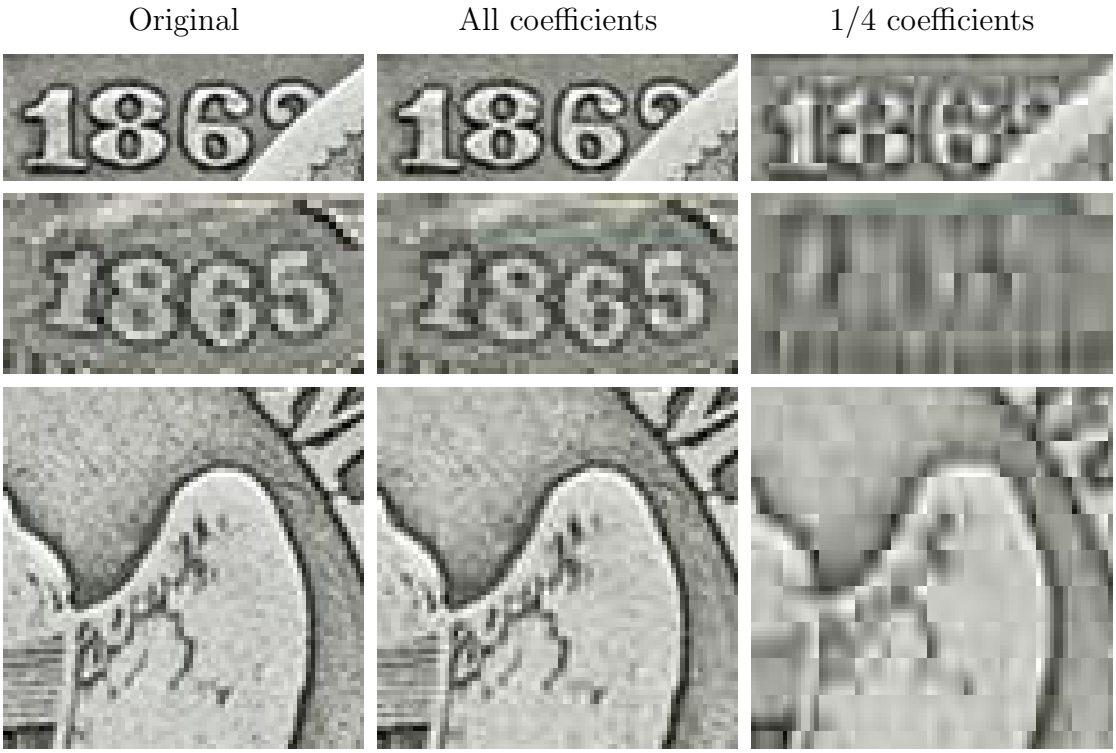

Figure 4: Coins with zoomed-in details

3.3 Experiment 3: leaves

In experiment 3, the block boundaries can be observed after zooming-in.

(a) Original leaves image (b) Compressed leaves image

Figure 5: Leaves
Compression ratio: 6.71 : 1
RMSE: 7.75
SNR: 79.23

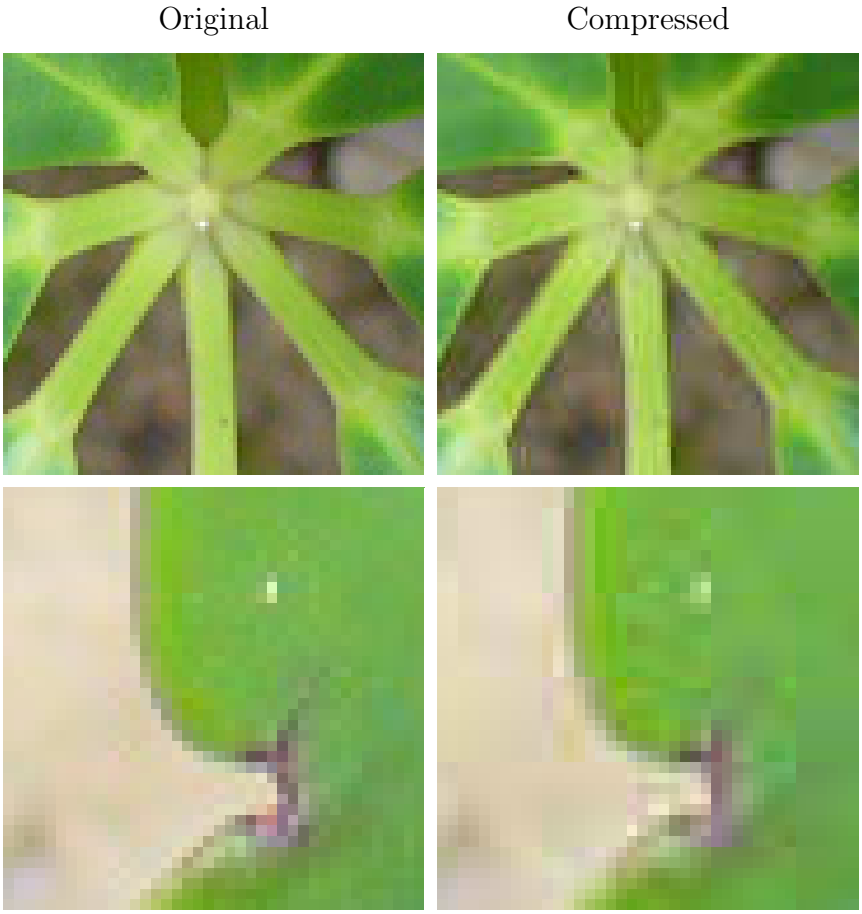

Figure 6: Leaves with zoomed-in details

3.4 Experiment 4: toy story

In experiment 4, the zoomed-in details (Fig 8) are some furry images. Although the compressed image is slightly different from the original one, it is difficult to distinguish the difference for furry images.

(a) Original toy story image (b) Compressed toy story image

Figure 7: Toy story
Compression ratio: 6.22 : 1
RMSE: 5.99
SNR: 198.76

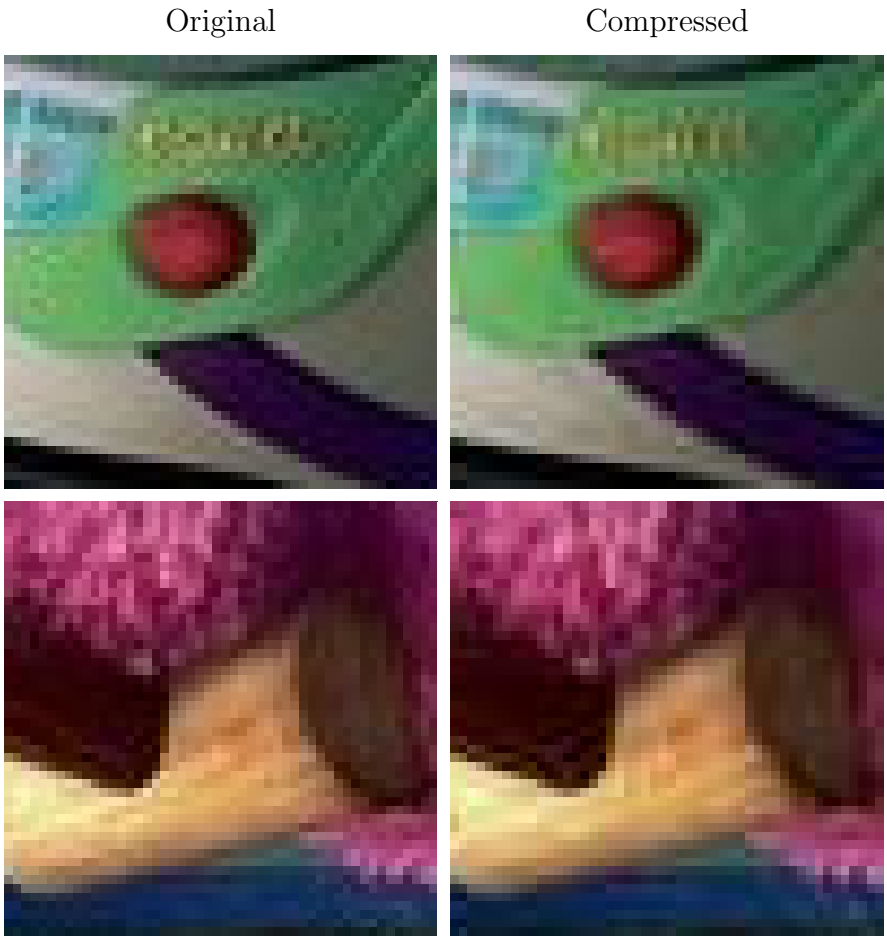

Figure 8: Toy story with zoomed-in details

3.5 Experiment 5: color wheel

In experiment 5, a lot of areas are smoothing in the original image. However, after compressed, a lot of noise can be observed in the right column of Fig 10.

(a) Original color wheel image (b) Compressed color wheel image

Figure 9: Toy story
Compression ratio: 7.62 : 1
RMSE: 2.18
SNR: 5607.85

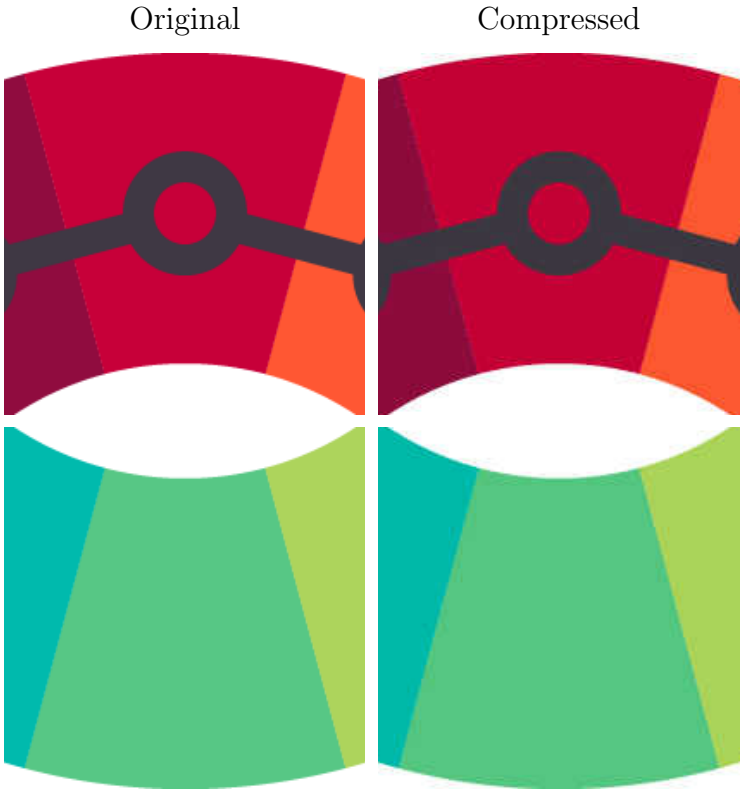

Figure 10: Color wheel with zoomed-in details

4 Discussions

For all the experiment, the original images and the compressed images look similar before zooming-in. However, after we zoom-in, we can observe several small artifacts that is hardly detected when we just look at the original image.

In my experiments, there are to main lossy parts. The first one is the quantization table used in all experiments, and the other one is the coefficients selection used in experiments 2 - Coins. If only the quantization table is adopted, only the last experiment - color wheel can clearly observe the distortion since it is a strong contrast between the original smooth image and the noised compressed image. If a lot of high frequency coefficients are discarded as experiment 2 did, the whole image still looks good, but if we zoom-in the image, we can find almost all the digits become unrecognizable.

In addition, the compression ratio increased significantly if only a few low frequency components are retained as experiment 2 showed. This made me realize the importance of transferring the images into frequency domain. Once the image is presented in frequency domain, it is easier for us to retain the important parts of the image and discard the less important parts.

In conclusion, I learned the importance of low frequency components and high frequency components in this assignment through how much bits we should allocate to these components.

5 Reference

1. https://github.com/ghallak/jpeg-python
2. https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

6 Codes

6.1 encoder.py

```
import pickle
import argparse
import numpy as np
from PIL import Image
from scipy import fftpack

from utils import load_quantization_table, zigzag_points
from huffman import HuffmanTree, getFreq


def dct_2d(image):
    return fftpack.dct(fftpack.dct(image.T, norm='ortho').T, norm='ortho')


def quantize(block, component):
    q = load_quantization_table(component)
    return (block / q).round().astype(np.int32)


def block_to_zigzag(block):
    return np.array([block[point] for point in zigzag_points(*block.shape)])


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument("--input", default='pic_png/color_wheel.png',
        help="path to the input image")
    parser.add_argument("--output", default='compressed_img/color_wheel.pkl',
        help="path to the output image")
    args = parser.parse_args(args=[])

    input_file = args.input
    output_file = args.output

    image = Image.open(input_file)
    ycbcr = image.convert('YCbCr')

    npmat = np.array(ycbcr, dtype=np.uint8)
    rows, cols = npmat.shape[0], npmat.shape[1]

    dc = np.empty((blocks_count, 3), dtype=np.int32)
    ac = np.empty((blocks_count, 63, 3), dtype=np.int32)
    for i in range(0, rows, 8):
        for j in range(0, cols, 8):
            try:
                block_index += 1
```

```
            except NameError:
                block_index = 0

            for c in range(3):
                block = npmat[i:i+8, j:j+8, c].astype(int) - 128
                dct_matrix = dct_2d(block)
                quant_matrix = quantize(dct_matrix, 'lum' if c == 0 else
                    'chrom')
                zz = block_to_zigzag(quant_matrix)
                dc[block_index, c] = zz[0]
                ac[block_index, :, c] = zz[1:]

    H_DC_Y = HuffmanTree(*getFreq(dc[:, 0]))
    H_DC_C = HuffmanTree(*getFreq(dc[:, 1:].flat))
    H_AC_Y = HuffmanTree(*getFreq(ac[:, :, 0].flat))
    H_AC_C = HuffmanTree(*getFreq(ac[:, :, 1:].flat))

    compressed_img = {'dc_y': H_DC_Y.encode(dc[:, 0]),
                      'dc_c': H_DC_C.encode(dc[:, 1:].flat),
                      'ac_y': H_AC_Y.encode(ac[:, :, 0].flat),
                      'ac_c': H_AC_C.encode(ac[:, :, 1:].flat),
                      'size': (rows, cols)}

    ratio = (H_DC_Y.bytes_original + H_DC_C.bytes_original +
        H_AC_Y.bytes_original + H_AC_C.bytes_original) \
        / (H_DC_Y.bytes_compressed + H_DC_C.bytes_compressed +
            H_AC_Y.bytes_compressed + H_AC_C.bytes_compressed)
    print(f'Compressio ratio: {ratio}')

    f = open(args.output,"wb")
    pickle.dump(compressed_img, f)
    f.close()
```

Code 1: Encoder

6.2 decoder.py

```
import pickle
import argparse
import numpy as np
from PIL import Image
from scipy import fftpack
import matplotlib.pyplot as plt

from utils import load_quantization_table, zigzag_points
from huffman import HuffmanTree


def read_compressed_img(img):
```

```
    pickle_file = open(img, 'rb')
    compressed_img = pickle.load(pickle_file)
    pickle_file.close()
    return compressed_img


def dequantize(block, component):
    q = load_quantization_table(component)
    return block * q


def idct_2d(image):
    return fftpack.idct(fftpack.idct(image.T, norm='ortho').T, norm='ortho')


def zigzag_to_block(zigzag):
    rows = cols = int(math.sqrt(len(zigzag)))

    if rows * cols != len(zigzag):
        raise ValueError("length of zigzag should be a perfect square")

    block = np.empty((rows, cols), np.int32)

    for i, point in enumerate(zigzag_points(rows, cols)):
        block[point] = zigzag[i]

    return block


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument("--input", default='compressed_img/color_wheel.pkl',
        help="path to the input image")
    parser.add_argument("--output",
        default='compressed_img/color_wheel_reconstruct.png')
    args = parser.parse_args(args=[])

    compressed_img = read_compressed_img(args.input)

    rows, cols = compressed_img['size']
    if rows % 8 == cols % 8 == 0:
        blocks_count = rows // 8 * cols // 8

    dict, bitsrting = compressed_img['dc_y']
    H_DC_Y = HuffmanTree(compressed_img['dc_y'][0])
    H_DC_C = HuffmanTree(compressed_img['dc_c'][0])
    H_AC_Y = HuffmanTree(compressed_img['ac_y'][0])
    H_AC_C = HuffmanTree(compressed_img['ac_c'][0])

    dc = np.empty((blocks_count, 3), dtype=np.int32)
    ac = np.empty((blocks_count, 63, 3), dtype=np.int32)
```

```
    dc[:, 0] = H_DC_Y.decode(compressed_img['dc_y'][1])
    temp = np.array(H_DC_C.decode(compressed_img['dc_c'][1]))
    dc[:, 1:] = np.reshape(temp, (-1, 2)).tolist()
    temp = np.array(H_AC_Y.decode(compressed_img['ac_y'][1]))
    ac[:, :, 0] = np.reshape(temp, (blocks_count, -1)).tolist()
    temp = np.array(H_AC_C.decode(compressed_img['ac_c'][1]))
    ac[:, :, 1:] = np.reshape(temp, (blocks_count, -1, 2)).tolist()


    npmat = np.empty((rows, cols, 3), dtype=np.uint8)
    for i in range(0, rows, 8):
        for j in range(0, cols, 8):
            try:
                block_index += 1
            except NameError:
                block_index = 0

            for c in range(3):
                zigzag = [dc[block_index, c]] + list(ac[block_index, :, c])
                quant_matrix = zigzag_to_block(zigzag)
                dct_matrix = dequantize(quant_matrix, 'lum' if c == 0 else
                    'chrom')
                block = idct_2d(dct_matrix) + 128
                block[block > 255] = 255
                block[block < 0] = 0
                npmat[i:i+8, j:j+8, c] = block

    npmat = np.array(npmat, dtype=np.uint8)
    image = Image.fromarray(npmat, 'YCbCr')
    image = image.convert('RGB')
    plt.imshow(image)
    plt.show()
    image.save(args.output)
```

Code 2: Decoder

6.3 huffman.py

```
from bitarray import bitarray


def getFreq(arr):
    dict = {}
    for value in arr:
        if value in dict.keys():
            dict[value] += 1
        else:
            dict[value] = 0

```

```
    value = list(dict.keys())
    freq = list(dict.values())
    return value, freq


class HuffmanTree:

    class __node:
        def __init__(self, value, freq, left=None, right=None):
            self.value = value
            self.freq = freq
            self.left = left
            self.right = right
            self.huff = '' # tree direction (0/1)


    def __init__(self, *args):
        if len(args) == 1:
            self.dict = args[0]
            self.__initNodes()
        if len(args) == 2:
            value = args[0]
            freq = args[1]
            self.__initDict(value, freq)


    def __initNodes(self):
        self.nodes = []
        self.nodes.append(self.__node(0, 0))

        for value, code in self.dict.items():
            current_node = self.nodes[0]
            for bit in code:
                if bit == '0':
                    if current_node.left == None:
                        current_node.left = self.__node(0, 0)
                    current_node = current_node.left
                if bit == '1':
                    if current_node.right == None:
                        current_node.right = self.__node(0, 0)
                    current_node = current_node.right
            current_node.value = value


    def __initDict(self, value, freq):
        self.dict = {}
        self.nodes = []

        for x in range(len(value)):
            self.nodes.append(self.__node(value[x], freq[x]))

```

```
63          while len(self.nodes) > 1:
64              self.nodes = sorted(self.nodes, key=lambda x: x.freq)
65
66              # pick 2 smallest nodes
67              left = self.nodes[0]
68              right = self.nodes[1]
69
70              # assign directional value to these nodes
71              left.huff = 0
72              right.huff = 1
73
74              # combine the 2 smallest nodes to create
75              # new node as their parent
76              newNode = self.__node(left.value+right.value,
                    left.freq+right.freq, left, right)
77
78              # remove the 2 nodes and add their
79              # parent as new node among others
80              self.nodes.remove(left)
81              self.nodes.remove(right)
82              self.nodes.append(newNode)
83
84          self.__buildDict(self.nodes[0])
85
86
87      def __buildDict(self, node, val=''):
88          newVal = val + str(node.huff) # huffman code for current node
89
90          if(node.left):
91              self.__buildDict(node.left, newVal)
92          if(node.right):
93              self.__buildDict(node.right, newVal)
94
95          if(not node.left and not node.right):
96              self.dict[node.value] = newVal
97
98
99      def encode(self, arr):
100         bitstring = ''
101
102         for value in arr:
103             bitstring += self.dict[value]
104
105         print(f'Number of Values: {len(arr)}')
106         print(f'Number of bytes required: {len(bitstring)/8:.3f}')
107
108         self.bytes_original = len(arr)
109         self.bytes_compressed = len(bitstring)/8
110
111         return (self.dict, bitstring)
112
```

```
113
114     def decode(self, bitstring):
115         arr = []
116
117         current_node = self.nodes[0]
118         for bit in bitstring:
119             if current_node.left == current_node.right == None:
120                 arr.append(current_node.value)
121                 current_node = self.nodes[0]
122             if bit == '0':
123                 if current_node.left != None:
124                     current_node = current_node.left
125             if bit == '1':
126                 if current_node.right != None:
127                     current_node = current_node.right
128         arr.append(current_node.value)
129
130         return arr
```

Code 3: Huffman Coding

6.4 utils.py

```
1  import numpy as np
2
3
4  def load_quantization_table(component):
5      if component == 'lum':
6          q = np.array([[16, 11, 10, 16, 24, 40, 51, 61],
7                        [12, 12, 14, 19, 26, 58, 60, 55],
8                        [14, 13, 16, 24, 40, 57, 69, 56],
9                        [14, 17, 22, 29, 51, 87, 80, 62],
10                       [18, 22, 37, 56, 68, 109, 103, 77],
11                       [24, 35, 55, 64, 81, 104, 113, 92],
12                       [49, 64, 78, 87, 103, 121, 120, 101],
13                       [72, 92, 95, 98, 112, 100, 103, 99]])
14     elif component == 'chrom':
15         q = np.array([[17, 18, 24, 47, 99, 99, 99, 99],
16                       [18, 21, 26, 66, 99, 99, 99, 99],
17                       [24, 26, 56, 99, 99, 99, 99, 99],
18                       [47, 66, 99, 99, 99, 99, 99, 99],
19                       [99, 99, 99, 99, 99, 99, 99, 99],
20                       [99, 99, 99, 99, 99, 99, 99, 99],
21                       [99, 99, 99, 99, 99, 99, 99, 99],
22                       [99, 99, 99, 99, 99, 99, 99, 99]])
23     else:
24         raise ValueError(("component should be either 'lum' or 'chrom', but
               '{comp}' was found").format(comp=component))
25     return q
```

```
26
27
28  def zigzag_points(rows, cols):
29      # constants for directions
30      UP, DOWN, RIGHT, LEFT, UP_RIGHT, DOWN_LEFT = range(6)
31
32      # move the point in different directions
33      def move(direction, point):
34          return {
35              UP: lambda point: (point[0] - 1, point[1]),
36              DOWN: lambda point: (point[0] + 1, point[1]),
37              LEFT: lambda point: (point[0], point[1] - 1),
38              RIGHT: lambda point: (point[0], point[1] + 1),
39              UP_RIGHT: lambda point: move(UP, move(RIGHT, point)),
40              DOWN_LEFT: lambda point: move(DOWN, move(LEFT, point))
41          }[direction](point)
42
43      # return true if point is inside the block bounds
44      def inbounds(point):
45          return 0 <= point[0] < rows and 0 <= point[1] < cols
46
47      # start in the top-left cell
48      point = (0, 0)
49
50      # True when moving up-right, False when moving down-left
51      move_up = True
52
53      for i in range(rows * cols):
54          yield point
55          if move_up:
56              if inbounds(move(UP_RIGHT, point)):
57                  point = move(UP_RIGHT, point)
58              else:
59                  move_up = False
60                  if inbounds(move(RIGHT, point)):
61                      point = move(RIGHT, point)
62                  else:
63                      point = move(DOWN, point)
64          else:
65              if inbounds(move(DOWN_LEFT, point)):
66                  point = move(DOWN_LEFT, point)
67              else:
68                  move_up = True
69                  if inbounds(move(DOWN, point)):
70                      point = move(DOWN, point)
71                  else:
72                      point = move(RIGHT, point)
```

Code 4: Utils

6.5 metric.py

```python
from pathlib import Path
import argparse
import numpy as np
from PIL import Image


def snr(img_1, img_2):
    img_1 = img_1.astype(int)
    img_2 = img_2.astype(int)
    error = np.mean(((img_1-img_2)**2))
    signal = np.mean(((img_1)**2))
    return signal / error


def rmse(img_1, img_2):
    return np.sqrt(np.mean((img_1-img_2)**2))


def crop(img_1, img_2, name_1, name_2):
    image = Image.fromarray(img_1[630:840, 375:585, :])
    image.save(name_1)

    image = Image.fromarray(img_2[630:840, 375:585, :])
    image.save(name_2)


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument("--input_1", default='pic_png/color_wheel.png')
    parser.add_argument("--input_2",
        default='compressed_img/color_wheel_reconstruct.png')
    parser.add_argument("--crop_name_1",
        default='compressed_img/color_wheel_crop_green.png')
    parser.add_argument("--crop_name_2",
        default='compressed_img/color_wheel_reconstruct_crop_green.png')
    args = parser.parse_args(args=[])

    img_1 = np.asarray(Image.open(args.input_1))
    img_2 = np.asarray(Image.open(args.input_2))

    print('RMSE', rmse(img_1, img_2))
    print('SNR', snr(img_1, img_2))

    crop(img_1, img_2, args.crop_name_1, args.crop_name_2)
```

Code 5: Metric