Optimized using
trial version
www.balesio.com

DAFTAR PUSTAKA

Ahmad, I. and Shin, S. (2021) "A novel hybrid image encryption–compression scheme by combining chaos theory and number theory," *Signal Processing: Image Communication*, 98. Available at: https://doi.org/10.1016/j.image.2021.116418.

Golts, A. and Schechner, Y.Y. (2021) "Image compression optimized for 3D reconstruction by utilizing deep neural networks," *Journal of Visual Communication and Image Representation*, 79. Available at: https://doi.org/10.1016/j.jvcir.2021.103208.

Gupta, A. *et al.* (2020) "Implementing lossless compression during image processing by integrated approach," *Materials Today: Proceedings* [Preprint], (xxxx). Available at: https://doi.org/10.1016/j.matpr.2020.10.052.

Hussain, A.J., Al-Fayadh, A. and Radi, N. (2018) *Image compression techniques: A survey in lossless and lossy algorithms*, *Neurocomputing*. Elsevier B.V. Available at: https://doi.org/10.1016/j.neucom.2018.02.094.

Li, Zhongqiang *et al.* (2021) "An optimized JPEG-XT-based algorithm for the lossy and lossless compression of 16-bit depth medical image," *Biomedical Signal Processing and Control*, 64(September 2019), p. 102306. Available at: https://doi.org/10.1016/j.bspc.2020.102306.

Mendivil, F. and Stenflo, Ö. (2021) "Extreme compression of grayscale images," *Communications in Nonlinear Science and Numerical Simulation*, 94. Available at: https://doi.org/10.1016/j.cnsns.2020.105546.

Perfilieva, I. and Hurtik, P. (2021) "The F-transform preprocessing for JPEG strong compression of high-resolution images," *Information Sciences*, 550, pp. 221–238. Available at: https://doi.org/10.1016/j.ins.2020.10.033.

Sayood, K. (2018a) *Huffman Coding*, *Introduction to Data Compression*. Available at: https://doi.org/10.1016/b978-0-12-809474-7.00003-3.

Sayood, K. (2018b) *Wavelet-Based Image Compression*, *Introduction to Data Compression*. Available at: https://doi.org/10.1016/b978-0-12-809474-7.00016-1.

Wang, Z. *et al.* (2021) "Deep image compression with multi-stage representation," *Journal of Visual Communication and Image Representation*, 79. Available at: https://doi.org/10.1016/j.jvcir.2021.103226.

Yuan, S. and Hu, J. (2019) "Research on image compression technology based on Huffman coding," *Journal of Visual Communication and Image*

Representation, 59, pp. 33–38. Available at: https://doi.org/10.1016/j.jvcir.2018.12.043.

Daubechies, I. (1988). Orthonormal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics*, 41(7), 909-996.

Huffman, D. A. (1952). A Method for the Construction of Minimum-Redundancy Codes. *Journal of the Franklin Institute*, 257(3), 163-177.

Optimized using
Trial Version
www.balesio.com

LAMPIRAN

Optimized using
Trial Version
www.pdfescape.com

de python kompresi gambar wavelet

```
 numpy as np
from PIL import Image, ImageOps
import os
from huffman_encoding import process_huffman
import pickle
import time

class wavelet:
    #Untuk Menegcek Ganjil atau Genapnya Gambar
    def check_odd_image(image):
        sizeh = len(image)
        sizew = len(image[0])
        newimagev = []
        result = []
        if(sizew % 2 != 0):
            for i in range(0, sizeh, 1):
                temp = image[i][-1]
                a = np.append(image[i], temp)
                newimagev.append(a)
        else:
            newimagev = image

        result = newimagev

        if(sizeh % 2 != 0):
            add = newimagev[sizeh-1]
            result.append(add)
        else:
            result = newimagev

        result = np.array(result)
        return result

    def frequency(image):
        result = []
        gambarLL = []
        gambarLH = []
        gambarHL = []
        gambarHH = []
        for j in range(0, len(image), 2):
            barisLL = []
            barisLH = []
            barisHL = []
            barisHH = []
            for i in range(0, len(image[j]), 2):
                a = int(image[j][i])
                b = int(image[j][i+1])
                c = int(image[j+1][i])
                d = int(image[j+1][i+1])
                ll = int(a + b + c + d) / 4
                lh = (int(a + b - c - d) / 4)
                hl = (int(a - b + c - d) / 4)
```

Optimized using
Trial Version
www.balesio.com

```
            hh = (int(a - b - c + d) / 4)
            barisLL.append(ll)
            barisLH.append(lh)
            barisHL.append(hl)
            barisHH.append(hh)

        gambarLL.append(barisLL)
        gambarLH.append(barisLH)
        gambarHL.append(barisHL)
        gambarHH.append(barisHH)

    result.append(gambarLL)
    result.append(gambarLH)
    result.append(gambarHL)
    result.append(gambarHH)

    return result

result = []
def compress(image):
    temp = []
    if (len(image)>1 or len(image[0])>1):
        image = wavelet.check_odd_image(image)
        frequency = wavelet.frequency(image)
        LL = frequency[0]
        LH = frequency[1]
        HL = frequency[2]
        HH = frequency[3]
        temp.append(LL)
        temp.append(LH)
        temp.append(HL)
        temp.append(HH)
        wavelet.result.append(temp)
    else:
        wavelet.result.append(image)
    return wavelet.result

def show_compressed(image):
    image = image[::-1]
    result = image[0]
    for i in range(1, len(image), 1):
        temp = []
        if(np.shape(result) != np.shape(image[i][0])):
            result = np.resize(result, new_shape=np.shape(image[i][0]))
        result = np.append(result, image[i][0], axis = 1)
        temp = np.append(image[i][1], image[i][2], axis = 1)
        result = np.append(result, temp, axis = 0)
    result = result[::-1]
    return result

def de_freq(image_array):
    result = []
    ll = image_array[0]
    lh = image_array[1]
    hl = image_array[2]
```

Optimized using
Trial Version
www.balesio.com

```
    h = image_array[3]

    or i in range(0, len(ll)):
      baris1 = []
      baris2 = []
      for a in range(0, len(ll[i])):
        baris1.append(ll[i][a] + lh[i][a] + hl[i][a] + hh[i][a])
        baris1.append(ll[i][a] + lh[i][a] - hl[i][a] - hh[i][a])
        baris2.append(ll[i][a] - lh[i][a] + hl[i][a] - hh[i][a])
        baris2.append(ll[i][a] - lh[i][a] - hl[i][a] + hh[i][a])
      result.append(baris1)
      result.append(baris2)
  result = np.array(result)
  return result

def reshaspe_image(image, shape):
  result = []
  wsize = len(image[0])
  hsize = len(image)
  h, w = np.shape(image)
  th, tw = shape

  if(w != tw):
    for baris in image:
      result.append(baris[:wsize-1])
  else:
    result = image
  if(h != th):
    result = result[:hsize-1]
  return result

def divide_compression(image):
  result =[]
  image = np.array(image)
  imwidth, imheight = np.shape(image)
  M = imwidth // 2
  N = imheight // 2
  for i in range(0, imwidth, M):
      for j in range(0, imheight, N):
        box = image[i:i+M, j:j+N]
        result.append(box)

  return result

def calculate_psnr(original_image, compressed_image):
  # Convert images to NumPy arrays
  original_array = np.array(original_image)
  compressed_array = np.array(compressed_image)

  # Calculate the mean squared error (MSE)
  mse = np.mean((original_array - compressed_array) ** 2)

  # Calculate the maximum pixel value
  max_pixel = np.max(original_array)
```

Optimized using
Trial Version
www.balesio.com

```
    alculate the PSNR using the formula: PSNR = 20 * log10(MAX) - 10 * log10(MSE)
    = 10 * np.log10(max_pixel ** 2 / mse)

    rn psnr

def calcualte_mse(original_image, compressed_image):
    # Convert images to NumPy arrays
    original_array = np.array(original_image)
    compressed_array = np.array(compressed_image)

    # Calculate the mean squared error (MSE)
    mse = np.mean((original_array - compressed_array) ** 2)

    return mse

start_time = time.time()
# ================ Define File Location ========================
image_path = './Data/Penelitian/'
filename = 'lineart1.jpg'
location = image_path+filename

# ================ Open Image File ================================
image_data = Image.open(location)
image_data = ImageOps.grayscale(image_data)
image_data.save(image_path + f'{filename}.jpg')
image_data = np.array(image_data)

# ================ Define The Shape of Original Image ============
shape = np.shape(image_data)

# ================ Compress Image Using DWT ======================
compress = wavelet.compress(image_data)

# ================ Processing the Compression Data ===============
compress = np.array(compress)
nshow = np.array(compress[0], dtype=np.uint8)
for i in range(len(nshow)):
    hifreq_image = Image.fromarray(nshow[i])
    hifreq_image.save(image_path + f'high_freq_{i}.jpg')
top = np.append(nshow[0], nshow[1], axis = 1)
bottom = np.append(nshow[2], nshow[3], axis = 1)
result = np.append(top, bottom, axis=0)
result = np.array(result, dtype=np.uint8)

#HUFFMAN ENCODE
compression_data, codewords, decompression_data = process_huffman(result)

#convert huffman data to pickle
compression_data = bytearray(compression_data)
compression_data_dumps = pickle.dumps(compression_data)
code_words_dumps = pickle.dumps(codewords)

# save as files
open(f'{image_path}huffman_wavelet.bit', 'wb').write(compression_data_dumps)
open(f'{image_path}codewords_huffman_wavelet.bit', 'wb').write(code_words_dumps)
```

Optimized using
Trial Version
www.balesio.com

pressed_image = Image.fromarray(decompression_data)
pressed_image.save(f'{image_path}wavelet_decompressed_{filename}')
# END OF HUFFMAN ENCODE

```
# ================ Save Compression result as Image =============
# result_asImage = Image.fromarray(result)
# result_asImage.save(image_path + 'hasil_kompresi.jpg')
# compression_path = image_path + 'hasil_kompresi.jpg'

# ================ Decompress Image Result ======================
result_defreq = wavelet.de_freq(nshow)
result_defreq = wavelet.reshaspe_image(result_defreq, shape)
result_defreq = np.array(result_defreq, dtype=np.uint8)

# ================ Save Decompression Result ====================
decompressed_image = Image.fromarray(result_defreq)
decompressed_image.save(image_path + f'wavelet_dekompresi_{filename}.bmp')
decompression_path = image_path + f'wavelet_dekompresi_{filename}.bmp'

# ================ Calculate Evaluation =========================
mse_value = calcualte_mse(image_data, result_defreq)
psnr_value = calculate_psnr(image_data, result_defreq)
original_size = os.path.getsize(location)
compression_size = os.path.getsize(f'{image_path}huffman_wavelet.bit')
codewords_size = os.path.getsize(f'{image_path}codewords_huffman_wavelet.bit')
decompresssion_size = os.path.getsize(decompression_path)
compression_ratio = 1 - (compression_size / original_size) * 100

# ================ Show Result ==================================
print('Original Size: ', original_size)
print('Compression Data Size: ', compression_size)
print('Codeworks Size: ', codewords_size)
print('Decompression Size: ', decompresssion_size)

# ================ Show Evaluation ==============================
print("==================== Evaluation Result ====================")
print("MSE: ", mse_value)
print("PSNR: ", psnr_value)
print("Ratio: ", compression_ratio)

end_time = time.time()
duration = end_time - start_time
print(f"Duration: ", duration, " s")
```

2. Kode python kompresi gambar huffman

```
import heapq
from collections import defaultdict
from PIL import Image, ImageOps
import numpy as np
import pickle
import os
```

Optimized using
Trial Version
www.balesio.com

time

HuffmanNode:
    def __init__(self, frequency, symbol=None, left=None, right=None):
        self.frequency = frequency
        self.symbol = symbol
        self.left = left
        self.right = right

    def __lt__(self, other):
        return self.frequency < other.frequency

def build_frequency_table(image_data):
    frequency_table = defaultdict(int)
    for pixel in image_data:
        frequency_table[pixel] += 1
    return frequency_table

def build_huffman_tree(frequency_table):
    heap = []
    for symbol, frequency in frequency_table.items():
        node = HuffmanNode(frequency, symbol)
        heapq.heappush(heap, node)

    while len(heap) > 1:
        node1 = heapq.heappop(heap)
        node2 = heapq.heappop(heap)
        merged_node = HuffmanNode(node1.frequency + node2.frequency, left=node1,
right=node2)
        heapq.heappush(heap, merged_node)

    return heap[0]

def build_codewords(node, current_code, codewords):
    if node.symbol is not None:
        codewords[node.symbol] = current_code
        return

    build_codewords(node.left, current_code + "0", codewords)
    build_codewords(node.right, current_code + "1", codewords)

def compress_image(image_data):
    frequency_table = build_frequency_table(image_data)
    huffman_tree = build_huffman_tree(frequency_table)
    codewords = {}
    build_codewords(huffman_tree, "", codewords)
    compressed_data = ""
    for pixel in image_data:
        compressed_data += codewords[pixel]
    return compressed_data, codewords

def decompress_image(compressed_data, codewords):
    reverse_codewords = {code: pixel for pixel, code in codewords.items()}
    current_code = ""
    decompressed_data = []

Optimized using
Trial Version
www.balesio.com

```
            bit in compressed_data:
            urrent_code += bit
            current_code in reverse_codewords:
                pixel = reverse_codewords[current_code]
                decompressed_data.append(pixel)
                current_code = ""
    return decompressed_data

def getbytes(bits):
    done = False
    while not done:
        byte = 0
        for _ in range(0, 8):
            try:
                bit = next(bits)
            except StopIteration:
                bit = 0
                done = True
            byte = (byte << 1) | bit
        yield byte

def calculate_psnr(original_image, compressed_image):
    # Convert images to NumPy arrays
    original_array = np.array(original_image)
    compressed_array = np.array(compressed_image)

    # Calculate the mean squared error (MSE)
    mse = np.mean((original_array - compressed_array) ** 2)

    # Calculate the maximum pixel value
    max_pixel = np.max(original_array)

    # Calculate the PSNR using the formula: PSNR = 20 * log10(MAX) - 10 * log10(MSE)
    psnr = 10 * np.log10(max_pixel ** 2 / mse)

    return psnr

def calcualte_mse(original_image, compressed_image):
    # Convert images to NumPy arrays
    original_array = np.array(original_image)
    compressed_array = np.array(compressed_image)

    # Calculate the mean squared error (MSE)
    mse = np.mean((original_array - compressed_array) ** 2)

    return mse

start_time = time.time()

# ============= File Location ====================
image_name = 'lineart1.jpg'
image_path = './Data/Penelitian/'
location = image_path+image_name
# ============= End of File Location ============
```

Optimized using
Trial Version
www.balesio.com

```
======= Using Image ====================
    _data = Image.open(location)
    _data = ImageOps.grayscale(image_data)
    _data =  np.array(image_data, dtype=np.uint8)
image_shape =  np.shape(image_data)
image_data = image_data.flatten()
# ============ End Of Using Image ============

# ============ Using File ====================
# image_data = open(location, 'rb').read()
# image_data = bytearray(image_data)
# image_data = list(image_data)
# ============ End Of Using File =============

# ================= Compress Data Using Huffman Encoding ==============
compressed_data, codewords = compress_image(image_data)
compressed_data_in_list = list(compressed_data)

# ============== Change Compressed data string to array =============
huffcode = []
for data in compressed_data_in_list:
   huffcode.append(int(data))

# ============== Change binary to INT ==================
intcode = []
for b in getbytes(iter(huffcode)):
   intcode.append(b)

# ============== Save in Data File Result ================================
intcode = np.array(intcode, dtype=np.uint8)
intcode = np.ndarray.tobytes(intcode)
print(intcode)
datapickle = pickle.dumps(intcode)
open("compressed_huffman_new.bit", "wb").write(datapickle)

codewords_data = pickle.dumps(codewords)
open("compressed_huffman_codeworks.bit", "wb").write(codewords_data)
# ================ End of Save in Data File ================================

# ============== Save in Image File
=====================================
# intcode = np.array(intcode, dtype=np.uint8)
# intcode = np.reshape(intcode, (-1, len(intcode)//2))
# intcode = Image.fromarray(intcode)
# intcode.save("compressed_huffman_new.jpg")
# ============== End of Save in Image File ================================

# ============== Decompressed Image Data ================================
decompressed_data = decompress_image(compressed_data, codewords)

# ============== Save Decompression Image Data ======================

# ============== Get File Size ======================================
original_size = os.path.getsize(location)
compressed_size = os.path.getsize('compressed_huffman_new.bit')
```

Optimized using
Trial Version
www.balesio.com

```
ession_ratio = (1 - (compressed_size / original_size)) * 100
alue = calcualte_mse(image_data, decompressed_data)
alue = calculate_psnr(image_data, decompressed_data)

# ================ Show Result Of Compressed Data in Binary and INT
====================
print("Compressed Data:", compressed_data)
print("Compressed in INT: ", len(intcode))
print("Original Data: ", len(image_data))
print("Codewords:", os.path.getsize('compressed_huffman_codeworks.bit'))
print("Decompressed data:", len(decompressed_data))

# ================ Show Evaluation Data
==================================
print("==================== Evaluation Data ============================")
print("MSE Evaluation: ", mse_value)
print("PSNR Evaluation: ", psnr_value)
print("Original Size: ", original_size, " byte")
print("Compression Size: ", compressed_size, " byte")
print("Compression Ration: ", compression_ratio, " %")

end_time = time.time()
calculation_time = end_time - start_time
print("Calculation Time: ", calculation_time)
```

3. Kode python kompresi gambar v-variable

```
import numpy as np
from sklearn.cluster import KMeans
from PIL import Image
import os
import pickle
from huffman_encoding import process_huffman
import time

def compress_image(image, num_clusters):
    value_sse = num_clusters
    # Convert the image to an RGB image
    image = image.convert('RGB')

    # Convert the image to a NumPy array
    image_array = np.array(image, dtype=np.uint8)

    # Reshape the image array to a 2D array
    flattened_image = image_array.reshape(-1, 2)

    # Perform clustering on the flattened image
    # while value_sse > 1:
    kmeans = KMeans(n_clusters=num_clusters)
    kmeans.fit(flattened_image)
    value_sse = kmeans.inertia_
    print(f"[+] Nilai SSE: {value_sse}, ukuran Cluster: {num_clusters}")
        # num_clusters += 10
```

Optimized using
Trial Version
www.balesio.com

pressed_image = kmeans.cluster_centers_[kmeans.predict(flattened_image)]

eshape the compressed image back to its original shape
pressed_image = compressed_image.reshape(image_array.shape)

    return compressed_image

def calculate_psnr(original_image, compressed_image):
    # Convert images to NumPy arrays
    original_array = np.array(original_image)
    compressed_array = np.array(compressed_image)

    # Calculate the mean squared error (MSE)
    mse = np.mean((original_array - compressed_array) ** 2)

    # Calculate the maximum pixel value
    max_pixel = np.max(original_array)

    # Calculate the PSNR
    psnr = 10 * np.log10(max_pixel ** 2 / mse)

    return psnr

def calculate_mse(original_image, compressed_image):
    # Convert images to NumPy arrays
    original_array = np.array(original_image)
    compressed_array = np.array(compressed_image)

    # Calculate the mean squared error (MSE)
    mse = np.mean((original_array - compressed_array) ** 2)

    return mse

def calculate_compression_ratio(original_size, compressed_size):
    # Calculate the compression ratio using the formula: Compression Ratio = Original Size / Compressed Size
    compression_ratio = original_size / compressed_size

    return compression_ratio

def calculate_sse(image, num_clusters):
    # Convert the image to an RGB image
    image = image.convert('RGB')

    # Convert the image to a NumPy array
    image_array = np.array(image, dtype=np.uint8)

    # Reshape the image array to a 2D array
    flattened_image = image_array.reshape(-1, 2)

    # Perform clustering on the flattened image
    kmeans = KMeans(n_clusters=num_clusters)
    kmeans.fit(flattened_image)
    sse_values = kmeans.inertia_

Optimized using
Trial Version
www.balesio.com

```
    rn sse_values

    ime = time.time()
    ify the image filename
image_path = './Data/Penelitian/'
image_name = 'lineart1.jpg'

# Load the original image using PIL
original_image = Image.open(image_path+image_name).convert('L')

# Display the original image
# original_image.show(title='Original Image')
# original_image = np.array(original_image, dtype=np.uint8)

# Compress the image using K-means clustering
num_clusters = 256
compressed_image = compress_image(original_image, num_clusters)

# Convert the compressed image array to PIL Image
compressed_image = Image.fromarray(np.uint8(compressed_image)).convert('L')

#HUFFMAN ENCODE
compression_data, codewords, decompression_data = process_huffman(compressed_image)

#convert huffman data to pickle
compression_data = bytearray(compression_data)
compression_data_dumps = pickle.dumps(compression_data)
code_words_dumps = pickle.dumps(codewords)

# save as files
open(f'{image_path}huffman_vvariable.bit', 'wb').write(compression_data_dumps)
open(f'{image_path}codewords_huffman_vvariable.bit', 'wb').write(code_words_dumps)

decompressed_image = Image.fromarray(decompression_data)
decompressed_image.save(f'{image_path}vvariable_decompressed_{image_name}')
# END OF HUFFMAN ENCODE

# Save the compressed image
# Calculate the PSNR
decompression_data = np.array(decompression_data, dtype=np.uint8)
psnr = calculate_psnr(original_image, decompression_data)
print('PSNR:', psnr)

# Calculate the MSE
mse = calculate_mse(original_image, decompression_data)
print('MSE:', mse)

sse = calculate_sse(original_image, num_clusters)
print('SSE: ', sse)

# Calculate the compression ratio
original_size = os.path.getsize(image_path + image_name)
compressed_size = os.path.getsize(image_path + 'huffman_vvariable.bit')
codewords_size = os.path.getsize(image_path + 'codewords_huffman_vvariable.bit')
print('Original Size:', original_size)
```

www.pdfelement.com
Optimized using
Trial Version

```
Compression Size:', compressed_size)
Codewords Size: ', codewords_size)
me = time.time()
on = end_time - strat_time
print('Time: ', duration, ' S')
```

4. Kode python kompresi gambar yang diusulkan

```
import numpy as np
from PIL import Image, ImageOps
from sklearn.cluster import KMeans
import os
import heapq
from collections import defaultdict
import pickle
import time
from huffman_encoding import process_huffman

class wavelet:
    #Untuk Menegcek Ganjil atau Genapnya Gambar
    def check_odd_image(image):
        sizeh = len(image)
        sizew = len(image[0])
        newimagev = []
        result = []
        if(sizew % 2 != 0):
            for i in range(0, sizeh, 1):
                temp = image[i][-1]
                a = np.append(image[i], temp)
                newimagev.append(a)
        else:
            newimagev = image

        result = newimagev

        if(sizeh % 2 != 0):
            add = newimagev[sizeh-1]
            result.append(add)
        else:
            result = newimagev

        result = np.array(result)
        return result

    def frequency(image):
        result = []
        gambarLL = []
        gambarLH = []
        gambarHL = []
        gambarHH = []
        for j in range(0, len(image), 2):
            barisLL = []
            barisLH = []
            barisHL = []
```

Optimized using
Trial Version
www.balesio.com

```
            barisHH = []
            for i in range(0, len(image[j]), 2):
                a = int(image[j][i])
                b = int(image[j][i+1])
                c = int(image[j+1][i])
                d = int(image[j+1][i+1])
                ll = int(a + b + c + d) / 4
                lh = (int(a + b - c - d) / 4)
                hl = (int(a - b + c - d) / 4)
                hh = (int(a - b - c + d) / 4)
                barisLL.append(ll)
                barisLH.append(lh)
                barisHL.append(hl)
                barisHH.append(hh)

            gambarLL.append(barisLL)
            gambarLH.append(barisLH)
            gambarHL.append(barisHL)
            gambarHH.append(barisHH)

        result.append(gambarLL)
        result.append(gambarLH)
        result.append(gambarHL)
        result.append(gambarHH)

        return result

    result = []
    def compress(image):
        temp = []
        if (len(image)>1 or len(image[0])>1):
            image = wavelet.check_odd_image(image)
            frequency = wavelet.frequency(image)
            LL = frequency[0]
            LH = frequency[1]
            HL = frequency[2]
            HH = frequency[3]
            temp.append(LL)
            temp.append(LH)
            temp.append(HL)
            temp.append(HH)
            wavelet.result.append(temp)
        else:
            wavelet.result.append(image)
        return wavelet.result

    def show_compressed(image):
        image = image[::-1]
        result = image[0]
        for i in range(1, len(image), 1):
            temp = []
            if(np.shape(result) != np.shape(image[i][0])):
                result = np.resize(result, new_shape=np.shape(image[i][0]))
            result = np.append(result, image[i][0], axis = 1)
            temp = np.append(image[i][1], image[i][2], axis = 1)
```

Optimized using
Trial Version
www.balesio.com

```
        result = np.append(result, temp, axis = 0)
    result = result[::-1]
    return result

def de_freq(image_array):
    result = []
    image_array = np.array(image_array, dtype=np.uint64)
    ll = image_array[0]
    lh = image_array[1]
    hl = image_array[2]
    hh = image_array[3]

    for i in range(0, len(ll)):
        baris1 = []
        baris2 = []
        for a in range(0, len(ll[i])):
            baris1.append(int(ll[i][a]) + int(lh[i][a]) + int(hl[i][a]) + int(hh[i][a]))
            baris1.append(int(ll[i][a]) + int(lh[i][a]) - int(hl[i][a]) - int(hh[i][a]))
            baris2.append(int(ll[i][a]) - int(lh[i][a]) + int(hl[i][a]) - int(hh[i][a]))
            baris2.append(int(ll[i][a]) - int(lh[i][a]) - int(hl[i][a]) + int(hh[i][a]))
        result.append(baris1)
        result.append(baris2)
    result = np.array(result)
    return result

def reshaspe_image(image, shape):
    result = []
    wsize = len(image[0])
    hsize = len(image)
    h, w = np.shape(image)
    th, tw = shape

    if(w != tw):
        for baris in image:
            result.append(baris[:wsize-1])
    else:
        result = image
    if(h != th):
        result = result[:hsize-1]
    return result

def divide_compression(image):
    result =[]
    image = np.array(image)
    imwidth, imheight = np.shape(image)
    M = imwidth // 2
    N = imheight // 2
    for i in range(0, imwidth, M):
            for j in range(0, imheight, N):
                box = image[i:i+M, j:j+N]
                result.append(box)

    return result

def compress_image_vvar(image, num_clusters):
```

Optimized using
Trial Version
www.balesio.com

```
    onvert the image to an RGB image
    iage = image.convert('RGB')

    # Convert the image to a NumPy array
    image_array = np.array(image, dtype=np.uint8)

    # Reshape the image array to a 2D array
    flattened_image = image_array.reshape(-1, 3)
    flattened_image = np.array(flattened_image, dtype=np.uint8)

    # Perform clustering on the flattened image
    kmeans = KMeans(n_clusters=num_clusters, n_init="auto", random_state=0)
    kmeans.fit(flattened_image)


    compressed_image = kmeans.cluster_centers_[kmeans.predict(flattened_image)]
    compressed_image = compressed_image.reshape(image_array.shape)
    return compressed_image

class HuffmanNode:
    def __init__(self, frequency, symbol=None, left=None, right=None):
        self.frequency = frequency
        self.symbol = symbol
        self.left = left
        self.right = right

    def __lt__(self, other):
        return self.frequency < other.frequency

def build_frequency_table(image_data):
    frequency_table = defaultdict(int)
    for pixel in image_data:
        frequency_table[pixel] += 1
    return frequency_table

def build_huffman_tree(frequency_table):
    heap = []
    for symbol, frequency in frequency_table.items():
        node = HuffmanNode(frequency, symbol)
        heapq.heappush(heap, node)

    while len(heap) > 1:
        node1 = heapq.heappop(heap)
        node2 = heapq.heappop(heap)
        merged_node = HuffmanNode(node1.frequency + node2.frequency, left=node1,
right=node2)
        heapq.heappush(heap, merged_node)

    return heap[0]

def build_codewords(node, current_code, codewords):
    if node.symbol is not None:
        codewords[node.symbol] = current_code
        return
```

www.palavio.com
Optimized using
trial version

```
        d_codewords(node.left, current_code + "0", codewords)
        d_codewords(node.right, current_code + "1", codewords)

    def compress_image_huffman(image_data):
        frequency_table = build_frequency_table(image_data)
        huffman_tree = build_huffman_tree(frequency_table)
        codewords = {}
        build_codewords(huffman_tree, "", codewords)
        compressed_data = ""
        for pixel in image_data:
            compressed_data += codewords[pixel]
        return compressed_data, codewords

    def decompress_image_huffman(compressed_data, codewords):
        reverse_codewords = {code: pixel for pixel, code in codewords.items()}
        current_code = ""
        decompressed_data = []
        for bit in compressed_data:
            current_code += bit
            if current_code in reverse_codewords:
                pixel = reverse_codewords[current_code]
                decompressed_data.append(pixel)
                current_code = ""
        return decompressed_data

    def getbytes(bits):
        done = False
        while not done:
            byte = 0
            for _ in range(0, 8):
                try:
                    bit = next(bits)
                except StopIteration:
                    bit = 0
                    done = True
                byte = (byte << 1) | bit
            yield byte

    def calculate_psnr(original_image, compressed_image):
        # Convert images to NumPy arrays
        original_array = np.array(original_image)
        compressed_array = np.array(compressed_image)

        # Calculate the mean squared error (MSE)
        mse = np.mean((original_array - compressed_array) ** 2)

        # Calculate the maximum pixel value
        max_pixel = np.max(original_array)

        # Calculate the PSNR using the formula: PSNR = 10 * log10(MAX) - 10 * log10(MSE)
        psnr = 10 * np.log10(max_pixel ** 2 / mse)

        return psnr

    def calcualte_mse(original_image, compressed_image):
```

Optimized using
Trial Version
www.balesio.com

    onvert images to NumPy arrays
    inal_array = np.array(original_image)
    pressed_array = np.array(compressed_image)

    # Calculate the mean squared error (MSE)
    mse = np.mean((original_array - compressed_array) ** 2)

    return mse

def calculate_sse(image, num_clusters):
    # Convert the image to an RGB image
    image = image.convert('RGB')

    # Convert the image to a NumPy array
    image_array = np.array(image, dtype=np.uint8)

    # Reshape the image array to a 2D array
    flattened_image = image_array.reshape(-1, 2)

    # Perform clustering on the flattened image
    kmeans = KMeans(n_clusters=num_clusters)
    kmeans.fit(flattened_image)
    sse_values = kmeans.inertia_

    return sse_values

start_time = time.time()
# ================ Define File Location ==========================
image_path = './Data/Penelitian/'
filename = 'AllType_3.bmp'
location = image_path+filename

# ================ Open Image File ================================
image_data = Image.open(location)
image_data = ImageOps.grayscale(image_data)
image_data = np.array(image_data)

# ================ Define The Shape of Original Image ============
shape = np.shape(image_data)

# ================ Compress Image Using DWT ======================
compress = wavelet.compress(image_data)

================ Divide Compressed Image ========================
LL, LH, HL, HH = compress[0]

HighFrequency = np.array([LH, HL, HH], dtype=np.uint8)

number_cluster = 3034
print("Cluster Size:", number_cluster)

compressed_HighFrequency = compress_image_vvar(HighFrequency, number_cluster)

lowFreq = np.array(LL, dtype=np.uint8)
hiFreq = np.array(compressed_HighFrequency)

www.pdfelement.com
Optimized using
Trial Version

RT HUFFMAN ENCODE
essed_data_huffman, codewords, decompressed_data_huffman =
s_huffman(hiFreq)
compressed_low_freq, codewords_lowfreq, decompressed_lowFreq =
process_huffman(lowFreq)

# High Frequency
compressed_data_dumps = bytearray(compressed_data_huffman)
compressed_data_dumps = pickle.dumps(compressed_data_dumps)
codewords_dumps = pickle.dumps(codewords)

open(f'{image_path}proposed_{filename}.bit', 'wb').write(compressed_data_dumps)
open(f'{image_path}proposed_codewords_{filename}.bit', 'wb').write(codewords_dumps)

lowFreq_image = Image.fromarray(lowFreq)
lowFreq_image.save(f'{image_path}proposed_lowfreq.jpg')

# Compressed VVAR
compressed_vvar = np.array([decompressed_lowFreq, decompressed_data_huffman[0],
decompressed_data_huffman[1], decompressed_data_huffman[2]])

decompressed_image = wavelet.de_freq(compressed_vvar)

decompressed_image = np.array(decompressed_image, dtype=np.uint8)

final_decompressed_image = Image.fromarray(decompressed_image)
final_decompressed_image.save(f"{image_path}proposed_decompressed_{filename}")

compressed_size = os.path.getsize(f'{image_path}proposed_{filename}.bit')
codewords_size = os.path.getsize(f'{image_path}proposed_codewords_{filename}.bit')
lowFreq_size = os.path.getsize(f'{image_path}proposed_lowfreq.jpg')
total_compress_size = compressed_size + codewords_size + lowFreq_size

print("================= Evaluation Result ============================")
print("Original Size: ", os.path.getsize(location))
print("Compression Size: ", compressed_size)
print("Codeword Size: ", codewords_size)
print("Low Freq Image: ", lowFreq_size)
print("Total Size: ", total_compress_size)
print("Decompression Size:
",os.path.getsize(f"{image_path}proposed_decompressed_{filename}"))

mse = calcualte_mse(image_data, decompressed_image)
psnr = calculate_psnr(image_data, decompressed_image)

print("MSE: ", mse)
print("PSNR: ", psnr)

end_time = time.time()
print("time: ", end_time - start_time, " Second")