Image Processing Assignment 8

Image Compression and Decompression using Huffman and Arithmetic Coding

Aditya Chavan

Roll No: 231080019

T.Y B.Tech IT

Date: October 18, 2025

1 Aim

To write a Python program for Image Compression and Decompression using:

- Huffman Coding and Decoding
- Arithmetic Coding and Decoding

2 Theory

Image compression reduces the amount of data required to represent an image while preserving visual quality. Lossless compression techniques such as Huffman and Arithmetic coding exploit statistical redundancy in the image data.

2.1 Huffman Coding

Huffman coding is a variable-length lossless compression technique that assigns shorter binary codes to frequent symbols and longer codes to rare ones.

Steps:

1. Calculate frequency of each pixel intensity.
2. Construct a binary Huffman tree.
3. Assign binary codes to each pixel value.
4. Replace pixel values with their corresponding codes.

Advantages:

- Simple and effective for data with skewed probability distributions.
- Produces exact decompression without any loss of information.

Disadvantages:

- Requires storing the Huffman tree along with the compressed data.
- Inefficient if symbol probabilities are nearly uniform.

2.2 Arithmetic Coding

Arithmetic coding represents the entire message as a fractional number in the range [0, 1). It achieves higher compression efficiency than Huffman coding by allowing fractional bits per symbol.

Steps:

1. Compute probabilities of all pixel values.
2. Define cumulative probability ranges for each symbol.
3. Narrow down the range for each successive symbol.
4. Output the final interval as the encoded value.

Advantages:

- Provides near-optimal compression close to entropy limit.
- Handles fractional probabilities more accurately.

Disadvantages:

- Computationally complex and slower than Huffman coding.
- Sensitive to rounding and floating-point precision.

3 Algorithm

3.1 Huffman Coding

1. Read image and flatten it into a 1D array.
2. Compute pixel frequency distribution.
3. Build a Huffman tree using frequencies.
4. Encode image pixels using generated codes.
5. Save encoded data and tree information to a file.

3.2 Arithmetic Coding

1. Compute frequency and cumulative tables.
2. Encode image data using scaled integer arithmetic.
3. Save encoded bitstream and header information.
4. Decode the bitstream using inverse arithmetic mapping.

4 Implementation

The implementation was done in Google Colab using Python and OpenCV.

4.1 Required Packages

```
!pip install opencv-python matplotlib numpy
```

4.2 Code Implementation

Google Colab Notebook: Click here to view the Colab Notebook

```python
import numpy as np
import cv2
import heapq
import pickle
from google.colab.patches import cv2_imshow
```

```python
img = cv2.imread("771476.jpg")
# Display
cv2_imshow(img)
```

```python
class HuffNode:
    def __init__(self, symbol, freq):
        self.symbol = symbol
        self.freq = freq
        self.left = None
        self.right = None
    def __lt__(self, other):
        return self.freq < other.freq

def build_huffman_tree(freqs):
    heap = [HuffNode(sym, f) for sym, f in freqs.items()]
    heapq.heapify(heap)
    while len(heap) > 1:
        a = heapq.heappop(heap)
        b = heapq.heappop(heap)
        parent = HuffNode(None, a.freq + b.freq)
        parent.left = a
        parent.right = b
        heapq.heappush(heap, parent)
    return heap[0]

def build_codes(node, code='', mapping=None):
    if mapping is None:
        mapping = {}
    if node.symbol is not None:
        mapping[node.symbol] = code or '0'
    else:
        build_codes(node.left, code + '0', mapping)
        build_codes(node.right, code + '1', mapping)
    return mapping
```

```python
def huffman_compress(img, out_file="compressed_image.huff"):
```

```python
    flat = img.flatten().tolist()
    freqs = {}
    for val in flat:
        freqs[val] = freqs.get(val, 0) + 1

    root = build_huffman_tree(freqs)
    codes = build_codes(root)

    bitstream = ''.join(codes[v] for v in flat)
    padding = 8 - len(bitstream) % 8
    bitstream += '0' * padding

    data = bytearray()
    for i in range(0, len(bitstream), 8):
        data.append(int(bitstream[i:i+8], 2))

    header = {'shape': img.shape, 'codes': codes, 'padding': padding}

    with open(out_file, 'wb') as f:
        pickle.dump((header, data), f)

    orig_size = img.size
    comp_size = len(data)
    ratio = orig_size / comp_size
    print(f"Huffman compressed: {orig_size} bytes     {comp_size} bytes (ratio: {ratio:.2f}:1)")
    return out_file
```

```python
def huffman_decompress(huff_file, out_image="reconstructed_from_huffman.png"):
    with open(huff_file, 'rb') as f:
        header, data = pickle.load(f)

    shape = header['shape']
    codes = header['codes']
    padding = header['padding']

    reverse_codes = {v: int(k) for k, v in codes.items()}
    bitstring = ''.join(format(byte, '08b') for byte in data)
    bitstring = bitstring[:-padding] if padding > 0 else bitstring

    current = ''
    decoded = []
    for bit in bitstring:
        current += bit
        if current in reverse_codes:
            decoded.append(reverse_codes[current])
            current = ''

```

```
    arr = np.array(decoded, dtype=np.uint8).reshape(shape)
    cv2.imwrite(out_image, arr)
    print(f"Huffman decompressed and saved as: {out_image}")
    return arr
```

```
def arithmetic_compress(img, out_file="compressed_image.arith"):
    flat = img.flatten().tolist()
    freqs = {}
    for val in flat:
        freqs[val] = freqs.get(val, 0) + 1
    total = len(flat)
    symbols = sorted(freqs.keys())

    cumul = {}
    cum = 0
    for s in symbols:
        cumul[s] = cum
        cum += freqs[s]

    SCALE = 1 << 16
    low = 0
    high = SCALE - 1
    output_bits = []
    pending_bits = 0

    for pixel in flat:
        range_size = high - low + 1
        high = low + (range_size * (cumul[pixel] + freqs[pixel]))
            // total - 1
        low = low + (range_size * cumul[pixel]) // total

        while True:
            if high < SCALE // 2:
                output_bits.append(0)
                output_bits.extend([1] * pending_bits)
                pending_bits = 0
                low = low * 2
                high = high * 2 + 1
            elif low >= SCALE // 2:
                output_bits.append(1)
                output_bits.extend([0] * pending_bits)
                pending_bits = 0
                low = (low - SCALE // 2) * 2
                high = (high - SCALE // 2) * 2 + 1
            elif low >= SCALE // 4 and high < 3 * SCALE // 4:
                pending_bits += 1
                low = (low - SCALE // 4) * 2
                high = (high - SCALE // 4) * 2 + 1
            else:
                break

```

```
    pending_bits += 1
    if low < SCALE // 4:
        output_bits.append(0)
        output_bits.extend([1] * pending_bits)
    else:
        output_bits.append(1)
        output_bits.extend([0] * pending_bits)

    bitstring = ''.join(str(b) for b in output_bits)
    padding = (8 - len(bitstring) % 8) % 8
    bitstring += '0' * padding

    encoded_bytes = bytearray()
    for i in range(0, len(bitstring), 8):
        encoded_bytes.append(int(bitstring[i:i+8], 2))

    compressed_data = {
        'shape': img.shape,
        'freqs': freqs,
        'cumul': cumul,
        'total': total,
        'symbols': symbols,
        'encoded': bytes(encoded_bytes),
        'padding': padding
    }

    with open(out_file, 'wb') as f:
        pickle.dump(compressed_data, f)

    orig_size = img.size
    comp_size = len(encoded_bytes)
    ratio = orig_size / comp_size
    print(f"Arithmetic compressed: {orig_size} bytes      {comp_size} bytes (ratio: {ratio:.2f}:1)")
    return out_file
```

```
def arithmetic_decompress(arith_file, out_image="reconstructed_from_arithmetic.png"):
    with open(arith_file, 'rb') as f:
        compressed_data = pickle.load(f)

    shape = compressed_data['shape']
    freqs = compressed_data['freqs']
    cumul = compressed_data['cumul']
    total = compressed_data['total']
    symbols = compressed_data['symbols']
    encoded = compressed_data['encoded']
    padding = compressed_data['padding']

    bitstring = ''.join(format(byte, '08b') for byte in encoded)
    if padding > 0:
```

```
        bitstring = bitstring[:-padding]

    SCALE = 1 << 16
    low = 0
    high = SCALE - 1
    code = 0

    for i in range(min(16, len(bitstring))):
        code = (code << 1) | int(bitstring[i])

    bit_pos = 16
    decoded = []
    pixels_needed = shape[0] * shape[1]

    for _ in range(pixels_needed):
        range_size = high - low + 1
        scaled_value = ((code - low + 1) * total - 1) // range_size
        symbol = None
        for s in symbols:
            if cumul[s] <= scaled_value < cumul[s] + freqs[s]:
                symbol = s
                break
        if symbol is None:
            symbol = symbols[-1]

        decoded.append(symbol)
        high = low + (range_size * (cumul[symbol] + freqs[symbol])) // total - 1
        low = low + (range_size * cumul[symbol]) // total

        while True:
            if high < SCALE // 2:
                low = low * 2
                high = high * 2 + 1
                code = (code * 2) % SCALE
                if bit_pos < len(bitstring):
                    code |= int(bitstring[bit_pos])
                    bit_pos += 1
            elif low >= SCALE // 2:
                low = (low - SCALE // 2) * 2
                high = (high - SCALE // 2) * 2 + 1
                code = ((code - SCALE // 2) * 2) % SCALE
                if bit_pos < len(bitstring):
                    code |= int(bitstring[bit_pos])
                    bit_pos += 1
            elif low >= SCALE // 4 and high < 3 * SCALE // 4:
                low = (low - SCALE // 4) * 2
                high = (high - SCALE // 4) * 2 + 1
                code = ((code - SCALE // 4) * 2) % SCALE
                if bit_pos < len(bitstring):
```

```python
                        code |= int(bitstring[bit_pos])
                        bit_pos += 1
                else:
                    break

    arr = np.array(decoded, dtype=np.uint8).reshape(shape)
    cv2.imwrite(out_image, arr)
    print(f"Arithmetic decompressed and saved as: {out_image}")
    return arr
```

```python
print("="*50)
print("HUFFMAN CODING")
print("="*50)
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
huff_file = huffman_compress(gray)
recon_huff = huffman_decompress(huff_file)

print("="*50)
print("ARITHMETIC CODING")
print("="*50)
arith_file = arithmetic_compress(gray)
recon_arith = arithmetic_decompress(arith_file)
```

5 Results and Analysis

5.1 Observations

- Huffman and Arithmetic Coding both achieved lossless image reconstruction.
- Arithmetic coding provided a slightly better compression ratio.
- Huffman coding was simpler and faster in execution.

6 Conclusion

This experiment successfully implemented Huffman and Arithmetic Coding algorithms for image compression and decompression. Both methods preserved image quality while reducing storage requirements. Huffman coding is efficient for simple probability distributions, whereas Arithmetic coding achieves higher compression for continuous-valued data.