Data Compression - Huffman Coding

Lakku Nalini Aishwarya - EE19BTECH11033
Alankritha C H - EE19BTECH11037

May 2, 2023

1 Introduction

Huffman coding is one of the fundamental compression algorithms used in image and video compression standards. Huffman coding is based on assigning variable-length codes to symbols in such a way that the most frequently occurring symbols have the shortest codes. The main idea behind Huffman coding is to calculate the frequency of input characters, sort them based on decreasing frequency, and use this information to construct a Huffman Tree that assigns binary codes to each character. This enables efficient compression of data, including image data, by exploiting the statistical regularities present in the input data.

2 Huffman Coding

A Huffman tree, also known as a Huffman coding tree, is a binary tree used in the Huffman coding algorithm for data compression. The construction of a Huffman tree typically involves the following steps:
1.Calculate the frequency of each symbol in the input data.
2.Sort the symbols based on their frequency in decreasing order.
3.Create a binary tree with each symbol as a leaf node.
4.Combine the two least frequent symbols and create a new internal node with a frequency equal to the sum of the two symbols' frequencies.
5.Repeat step 4 until all the symbols are combined into a single tree with a single root node.

Once the Huffman tree is constructed, each symbol is assigned a unique binary code based on the path from the root to its corresponding leaf node. The binary code for each symbol is generated by assigning 0 to left branches and 1 to right branches when traversing the tree.This results in a compressed representation of the input data that requires fewer bits to store than the original data, enabling faster transmission and more efficient storage.

3 Code

The code is uploaded in GitHub. Link

3.1 Taking input image

This code is designed to read in an image, with options for both color and grayscale input. Once the image is read in, it is stored in a convenient array format for further processing or analysis. Finally, the code outputs the array representation of the image to the terminal, providing a visual display of the image data.

```
print("Huffman Compression Program")
print("==================================================================")
def get_num_pixels(filepath):
    width, height = Image.open(filepath).size
```

```
    return width*height
h = int(input("Enter 1 if you want to input an colour image file, 2 for
                                                default gray scale case:"))
if h == 1:
    file = input("Enter the filename:")
    num_pixels=get_num_pixels(file)
    my_string = np.asarray(Image.open(file),np.uint8)
    shape = my_string.shape
    print(shape)
    a = my_string
    print ("Entered string is:",my_string)
    my_string = str(my_string.tolist())
elif h == 2:
    array = np.arange(0, 737280, 1, np.uint8)
    my_string = np.reshape(array, (1024, 720))
    num_pixels=737280
    print ("Entered string is:",my_string)
    a = my_string
    my_string = str(my_string.tolist())

else:
    print("You entered invalid input")                    # taking user input
```

3.2 Step 1: Frequency calculation

One of the first and most important steps is to calculate the frequency of each character or pixel in the input data. This frequency information is then typically stored in an array or list format for easy manipulation. The following code implements it.

```
letters = []
only_letters = []
for letter in my_string:
    if letter not in letters:
        frequency = my_string.count(letter)            #frequency of each
                                                        letter repetition
        letters.append(frequency)
        letters.append(letter)
        only_letters.append(letter)
```

3.3 Step 2: Sorting according to frequency

Once the frequency data is stored in the letter array, the next step is to sort the entries based on decreasing frequency.

The sorted frequency data can then be used to construct the Huffman Tree, which is a binary tree that assigns unique binary codes to each pixel value based on their frequency. For this, the letters are stored in nodes, which will be used to construct the tree.

```
nodes = []
while len(letters) > 0:
    nodes.append(letters[0:2])
    letters = letters[2:]                              # sorting according to frequency
nodes.sort()
huffman_tree = []
huffman_tree.append(nodes)                            #Make each unique character
                                                       as a leaf node
```

3.4 Step 3: Constructing Huffman Tree

The code presented below employs a recursive algorithm to construct a Huffman tree and generate binary codes for each node in the tree. It assigns 0 and 1 to nodes in the next level.

```
def combine_nodes(nodes):
    pos = 0
    newnode = []
    if len(nodes) > 1:
        nodes.sort()
        nodes[pos].append("1")                          # assigning values 1 and 0
        nodes[pos+1].append("0")
        combined_node1 = (nodes[pos] [0] + nodes[pos+1] [0])
        combined_node2 = (nodes[pos] [1] + nodes[pos+1] [1])
        # combining the nodes to generate pathways
        newnode.append(combined_node1)
        newnode.append(combined_node2)
        newnodes=[]
        newnodes.append(newnode)
        newnodes = newnodes + nodes[2:]
        nodes = newnodes
        huffman_tree.append(nodes)
        combine_nodes(nodes)
    return huffman_tree                                 # huffman tree generation
```

The following code prints the Huffman Tree, level wise.

```
newnodes = combine_nodes(nodes)

huffman_tree.sort(reverse = True)
print("Huffman tree with merged pathways:")

checklist = []
for level in huffman_tree:
    for node in level:
        if node not in checklist:
            checklist.append(node)
        else:
            level.remove(node)
count = 0
for level in huffman_tree:
    print("Level", count,":",level)              #print huffman tree
    count+=1
print()
```

The binary string for the input is generated using the following code. It uses the binary code from Huffman Tree for each letter and appends to the string.

```
letter_binary = []
if len(only_letters) == 1:
    lettercode = [only_letters[0], "0"]
    letter_binary.append(lettercode*len(my_string))
else:
    for letter in only_letters:
        code =""
        for node in checklist:
            if len (node)>2 and letter in node[1]:          #genrating binary code
                code = code + node[2]
        lettercode =[letter,code]
        letter_binary.append(lettercode)
```

```
print(letter_binary)
print("Binary code generated:")
for letter in letter_binary:
    print(letter[0], letter[1])

bitstring =""
for character in my_string:
    for item in letter_binary:
        if character in item:
            bitstring = bitstring + item[1]
binary ="0b"+bitstring
print("Your message as binary is:",binary)
                                        # binary code generated
```

3.5 Decoding the Huffman Tree

The decoding process involves traversing the Huffman tree using the binary codes as a guide, starting from the first level and moving down the tree based on the binary code. The following code decodes the binary string back to input array.

```
 output.write(bitstring)

bitstring = str(binary[2:])
uncompressed_string =""
code =""
for digit in bitstring:
    code = code+digit
    pos=0                                       #iterating and decoding
    for letter in letter_binary:
        if code ==letter[1]:
            uncompressed_string=uncompressed_string+letter_binary[pos] [0]
            code=""
        pos+=1
```

The code executes and publishes the uncompressed output based on whether the input file was a colour image or a grayscale image. By analyzing the uncompressed image and comparing it with the original image, we can assess the effectiveness of the Huffman coding algorithm in preserving the quality of the input data.

```
print("Your UNCOMPRESSED data is:")
if h == 1:
    temp = re.findall(r'\d+', uncompressed_string)
    res = list(map(int, temp))
    res = np.array(res)
    res = res.astype(np.uint8)
    res = np.reshape(res, shape)
    import numpy as np
    mse = (np.square(res - a)).mean(axis=None)
    print(mse)
    print(res)
    print("Observe the shapes and input and output arrays are matching or not")
    print("Input image dimensions:",shape)
    print("Output image dimensions:",res.shape)
    data = Image.fromarray(res)
    data.save('output.png')
    if a.all() == res.all():
        print("Success")
if h == 2:
```

```
temp = re.findall(r'\d+', uncompressed_string)
res = list(map(int, temp))
print(res)
res = np.array(res)
res = res.astype(np.uint8)
res = np.reshape(res, (1024, 720))
print(res)
data = Image.fromarray(res)
data.save('output.png')
print("Success")
```

3.6 Effectiveness Measure

By comparing the initial image and the uncompressed image, we can assess the loss in compression and evaluate the effectiveness of the Huffman coding algorithm.

```
uncompressed_file_size = len(my_string)*7
compressed_file_size = len(binary)-2
print("Your original file size was", uncompressed_file_size," bits. The compressed size is
print("This is a saving of ",uncompressed_file_size-compressed_file_size," bits")
print("Compression Ratio: ",uncompressed_file_size/compressed_file_size)
# Bits per pixel (BPP)
print("BPP: ",compressed_file_size/num_pixels)
output = open("compressed.txt","w+")
print("Compressed file generated as compressed.txt")
output = open("compressed.txt","w+")
```

4 Results

When the code is executed for different input photos, the associated Effectiveness measure and output uncompressed images are as follows:
Input file: 1

Figure 1: This is input image.

Effectiveness measure:

```
Your original file size was 4594254 bits. The compressed size is: 2177140
This is a saving of  2417114 bits
```

Figure 2: This is output image.

```
Compression Ratio:  2.110224422866697
BPP:  43.26159960258321
```

Input file: 2

Figure 3: This is input image.

Figure 4: This is output image.

Effectiveness Measure:

```
Your original file size was 2234764 bits. The compressed size is: 1124694
This is a saving of  1110070 bits
Compression Ratio:  1.9869973521686788
BPP:  54.530618181818184
```

Input file: 3
Effectiveness Measure

Figure 5: This is input image.

Figure 6: This is output image.

```
Your original file size was 224951258 bits. The compressed size is: 110132055
This is a saving of  114819203 bits
Compression Ratio:  2.0425593438713188
BPP:  47.80037109375
```

Input file: 4

Figure 7: This is input image.

Figure 8: This is output image.

Efectiveness Measure:

```
Your original file size was 31917571 bits. The compressed size is: 15988549
This is a saving of  15929022 bits
Compression Ratio:  1.9962768979223817
BPP:  54.83041495198903
```

5 Conclusion

By compressing image data using Huffman coding, we can achieve significant reductions in file size without sacrificing quality. This makes Huffman coding a valuable tool for applications where storage space or bandwidth is limited, such as in mobile devices, online image galleries, or video streaming

services.
However, it's important to note that the effectiveness of Huffman coding depends on the statistical regularities in the input data. If the input data has a uniform distribution or low redundancy, the compression gains may be limited. In such cases, other compression techniques such as wavelet compression or JPEG compression may be more effective.

6 Some Areas of Improvement

The codes obtained while doing the Huffman coding can further be coded using Canonical Huffman Coding which reduces the size of data required for decompression. Eg: Let the Huffman code of 4 bits be "0001" in terms of the canonical code its value is just 4 which is equal to its total number of bits present in its Huffman code.
Use fractal codes/ image generation along with the huffman codes. Though fractal algorithm is a lossy compression algorithm, image compression using this algorithm takes less execution time to compress the given images than Huffman Coding.

7 References

The following papers were referred to for the report:
1. Implementation of data compression using Huffman coding
2. Canonical Huffman Coding for Image Compression
3. Python Implementaion of Huffman Coding