



Dayananda Sagar College of Engineering
Department of Electronics and Communication Engineering
Shavige Malleshwara Hills, Kumaraswamy Layout, Bengaluru – 560 078.
(An Autonomous Institute affiliated to VTU, Approved by AICTE & ISO 9001:2008 Certified)
Accredited by National Assessment and Accreditation Council (NAAC) with 'A' grade

Assignment

Program: B.E.

Course: Digital Communication Systems

Course Code: 18EC5DCDCS

Branch: ECE

Semester : 5

Date:

A Report on

Huffman, Shannon Encoding and Decoding

Submitted by

USN: 1DS18EC091

NAME: Sudhamshu B N

USN: 1DS19EC426

NAME: Rajasudhan Gowda S A

Faculty In-charge

Prof. Navya Holla K

Signature of Faculty In-charge

Introduction

Huffman coding is a data compression algorithm used to reduce data size. It uses the Greedy algorithm as its technique for implementation. Huffman code assigns codewords depending on the frequency of the characters in some data. It assigns a shorter length codeword for a character with high frequency and a longer length codeword for a character with less frequency. In this way, it reduces the amount of space that the data takes.

Shannon Fano Algorithm is an entropy coding technique used for lossless data compression. It uses the probabilities of occurrence of a character and assigns a unique variable-length code to each of them.

In this mini project we are implementing both algorithms in Python.

Algorithms

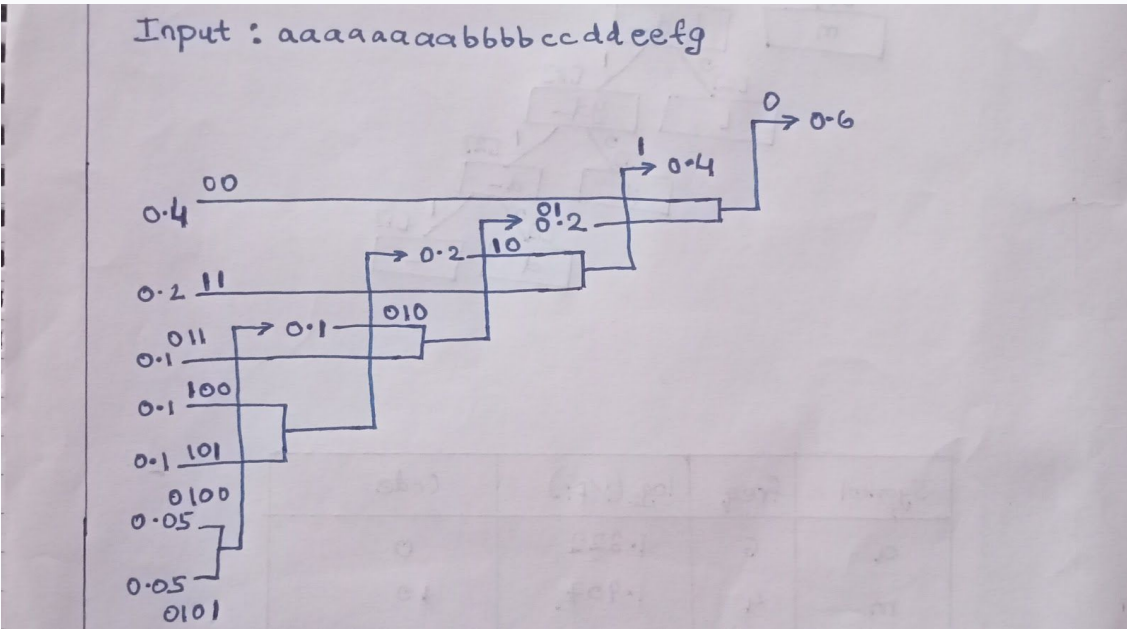
Huffman Encoding:

- Calculate the frequency of each character in the string.
- Sort the characters in increasing order of the frequency. These are stored in a priority queue
- Make each unique character as a leaf node
- Create an empty node. Assign the minimum frequency to the left child of the new node created and assign the second minimum frequency to the right child of the new node. Set the value of the new node as the sum of the above two minimum frequencies.
- Remove these two minimum frequencies from the queue and add the sum into the list of frequencies.
- Insert new node into the tree.
- Repeat steps 3 to 5 for all the characters
- For each non-leaf node, assign 1 to the left edge and 0 to the right edge.

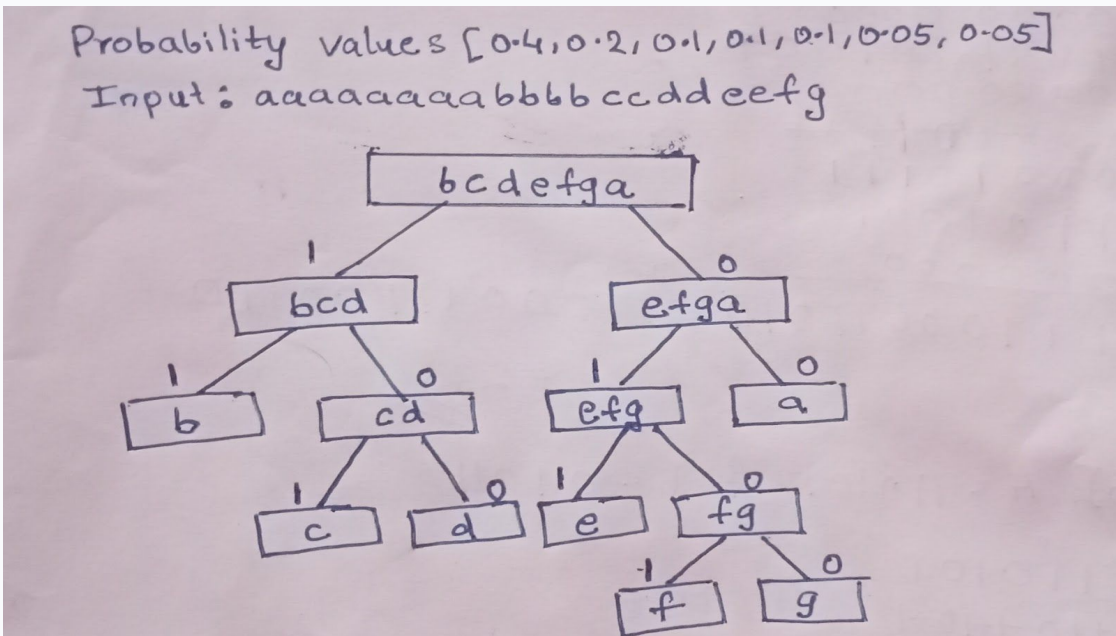
Huffman Decoding:

- To decode the encoded data we require Huffman tree and compressed code.
- Iterate through the binary encoded data, to find the corresponding encoded data.

Manual Calculation



Algorithm



Shannon Encoding:

- Calculate the frequency of each character in the string.
- Sort the characters in increasing order of the frequency. These are stored in a priority queue
- Recursively divide the symbols into two parts, each with approximately the same number of counts, until all parts contain only one symbol.
- Assign values to the nodes (at the left of branch 0 and at the right 1)

Shannon Decoding:

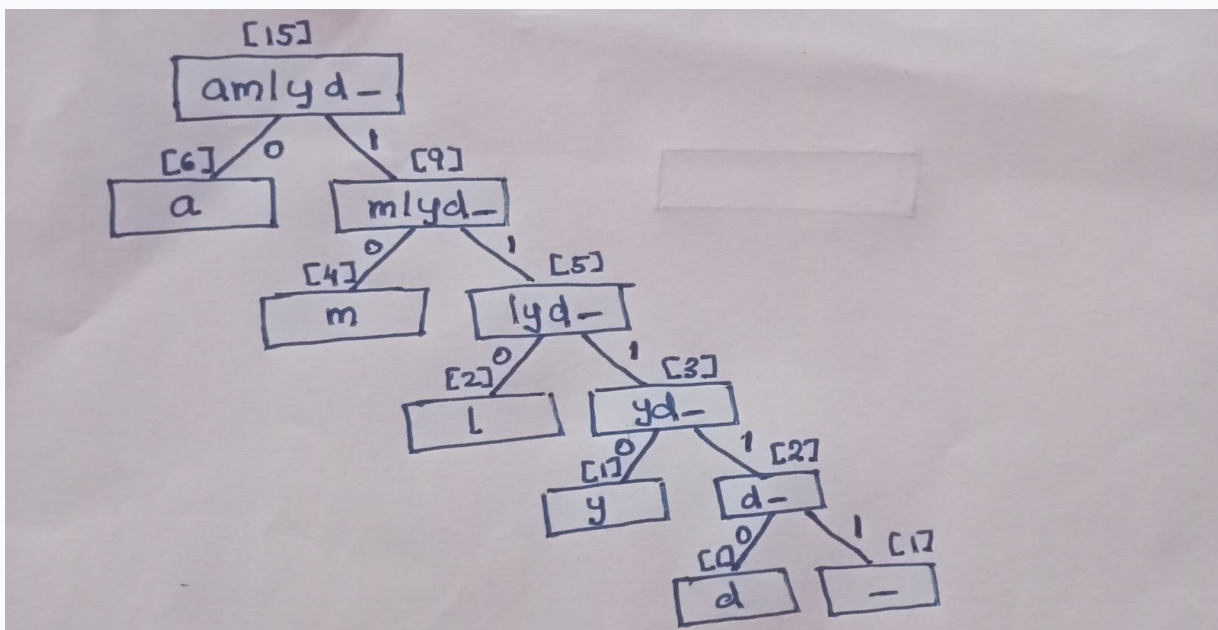
- To decode the data the same code as Huffman decoding is used.
- To decode the encoded data we require Shannon tree and compressed code.
- Iterate through the binary encoded data, to find the corresponding encoded data.

Manual Calculation

malayalam nadam

Symbol	freq	$\log_2(1/P_i)$	Code.
a	6	1.322	0
m	4	1.907	10
l	2	2.907	110
y	1	3.907	1110
d	1	3.907	11110
-	1	3.907	11111

Algorithm



Huffman Code

Click here:[sudhamshu091/Huffman-Encoding-and-Decoding \(github.com\)](https://github.com/sudhamshu091/Huffman-Encoding-and-Decoding)

```
print("Huffman Compression Program")
print("=====")
h = int(input("Enter 1 if you want to enter in command window, 2 if you are using some file:"))
if h == 1:
    my_string = input("Enter the string you want to compress:")
elif h == 2:
    file = input("Enter the filename:")
    with open(file, 'r') as f:
        my_string = f.read()
else:
    print("You entered invalid input")
len_my_string = len(my_string)          # taking user input
print("Entered string is:", my_string)
print("Your data is ", len_my_string * 7, "bits long")

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)

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

```

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])
        newnode.append(combined_node1)
        newnode.append(combined_node2)  # combining the nodes to generate pathways
        newnodes=[]
        newnodes.append(newnode)
        newnodes = newnodes + nodes[2:]
        nodes = newnodes
        huffman_tree.append(nodes)
        combine_nodes(nodes)
    return huffman_tree                # huffman tree generation

```

```

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()

```

```

letter_binary = []
if len(only_letters) == 1:
    lettercode = [only_letters[0], "0"]

```



```

letter_binary.append(letter_code*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("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:")
print(bitstring)                                # binary code generated

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:",compressed_file_size)
print("This is a saving of ",uncompressed_file_size-compressed_file_size,"bits")

print("Compressed file generated as compressed.txt")
output = open("compressed.txt","w+")
print("Decoding.....")
output.write(bitstring)

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

```

```
if code == letter[1]:
    uncompressed_string = uncompressed_string + letter_binary[pos][0]
    code = ""
pos += 1

print("Your UNCOMPRESSED data is:")    # uncompressed string
print(uncompressed_string)
```

Shannon Fano Code

Click here:[sudhamshu091/Shannon-Encoding-and-Decoding \(github.com\)](https://github.com/sudhamshu091/Shannon-Encoding-and-Decoding)

```
print("Shannon Compression Program")
print("=====")
import collections
h = int(input("Enter 1 if you want to enter in command window, 2 if you are using input as file :"))
if h == 1:
    message = input("Enter the string you want to compress:")
elif h == 2:
    file = input("Enter the filename:")
    with open(file, 'r') as f:
        message = f.read()
else:
    print("You entered invalid input")
print("Entered string is:",message)          #taking input from user

c = {}
def create_list(message):
    list = dict(collections.Counter(message))
    for key, value in list.items():
        print(key, ' : ', value)            #creating the sorted list according to the probability
    list_sorted = sorted(iter(list.items()), key = lambda k_v:(k_v[1],k_v[0]),reverse=True)
    final_list = []
    for key,value in list_sorted:
        final_list.append([key,value,""])
    return final_list

print("Shannon tree with merged pathways:")
def divide_list(list):
    if len(list) == 2:
        print([list[0]], [list[1]])          #printing merged pathways
```

```

    return [list[0]], [list[1]]
else:
    n = 0
    for i in list:
        n += i[1]
    x = 0
    distance = abs(2*x - n)
    j = 0
    for i in range(len(list)):          #shannon tree structure
        x += list[i][1]
        if distance < abs(2*x - n):
            j = i
    print(list[0:j+1], list[j+1:])      #printing merged pathways
    return list[0:j+1], list[j+1:]

```

```

def label_list(list):
    list1, list2 = divide_list(list)
    for i in list1:
        i[2] += '0'
        c[i[0]] = i[2]
    for i in list2:
        i[2] += '1'
        c[i[0]] = i[2]
    if len(list1) == 1 and len(list2) == 1:    #assigning values to the tree
        return
    label_list(list2)
    return c

```

```

code = label_list(create_list(message))
print("Shannon's Encoded Code:")
output = open("compressed.txt", "w+")        # generating output binary
letter_binary = []
for key, value in code.items():
    print(key, ' : ', value)
    letter_binary.append([key, value])
print("Compressed file generated as compressed.txt")

```

```

for a in message:
    for key, value in code.items():
        if key in a:

```

```

        print(key, ' : ', value)
        output.write(value)
output = open("compressed.txt","r")
intermediate = output.readlines()
bitstring = ""
for digit in intermediate:
    bitstring = bitstring + digit
uncompressed_string = ""
code = ""
for digit in bitstring:
    code = code+digit
    pos=0
    for letter in letter_binary:          # decoding the binary and genrating original data
        if code ==letter[1]:
            uncompressed_string=uncompressed_string+letter_binary[pos] [0]
            code=""
        pos+=1

print("Your UNCOMPRESSED data is:")
print(uncompressed_string)

```

Huffman Results

Executed Huffman Encoding and decoding program for probability values

[0.4, 0.2, 0.1, 0.1, 0.1, 0.05, 0.05] generated codes 00,11,101,100,011,0101,0100

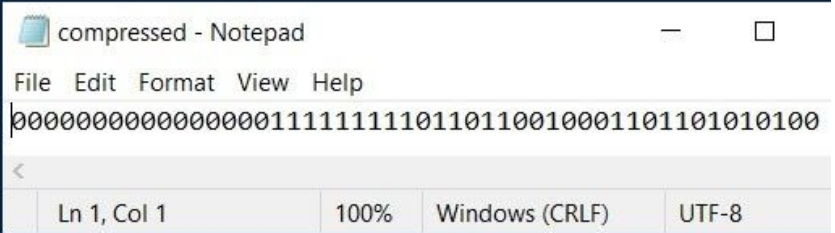
[8*a,4*b,2*c,2*d,2*e, 1*f, 1*g]

compressed file generated as 000000000000000001111111110110110010001101101010100

Decoded data:aaaaaaaabbbbccddeefg

```
Huffman Compression Program
=====
Enter 1 if you want to enter in command window, 2 if you are using some file:1
Enter the string you want to compress:aaaaaaaabbbbccddeefg
Entered string is: aaaaaaaabbbbccddeefg
Your data is 140 bits long
Huffman tree with merged pathways:
Level 0 : [[20, 'bcdefga']]
Level 1 : [[8, 'bcd', '1'], [12, 'efga', '0']]
Level 2 : [[4, 'efg', '1'], [8, 'a', '0']]
Level 3 : [[4, 'b', '1'], [4, 'cd', '0'], [8, 'a', '0']]
Level 4 : [[2, 'e', '1'], [2, 'fg', '0'], [4, 'cd', '0']]
Level 5 : [[2, 'c', '1'], [2, 'd', '0'], [2, 'fg', '0'], [8, 'a', '0']]
Level 6 : [[1, 'f', '1'], [1, 'g', '0'], [2, 'd', '0'], [4, 'b', '1']]

[['a', '00'], ['b', '11'], ['c', '101'], ['d', '100'], ['e', '011'], ['f', '0101'], ['g', '0100']]
Binary code generated:
a 00
b 11
c 101
d 100
e 011
f 0101
g 0100
Your message as binary is:
000000000000000001111111110110110010001101101010100
Your original file size was 140 bits. The compressed size is: 50
This is a saving of 90 bits
Compressed file generated as compressed.txt
Decoding.....
Your UNCOMPRESSED data is:
aaaaaaaabbbbccddeefg
```



Shannon Fano Results

Executed Shannon Encoding and decoding for input “malayalam madam”

for probability values [0.266, 0.4, 0.1333, 0.0666, 0.0666, 0.0666]

[a , m , l , y , d , “ ”]

Generated Codes: 0,10,110,1110,11110,11111

Binary code :1001100111001100101111110011110010

Decoded data: malayalam madam

```
Shannon Compression Program
=====
Enter 1 if you want to enter in command window, 2 if you are using input as file :1
Enter the string you want to compress:malayalam madam
Entered string is: malayalam madam
Shannon tree with merged pathways:
m : 4
a : 6
l : 2
y : 1
: 1
d : 1
[['a', 6, '']] [['m', 4, '']], [['l', 2, '']], [['y', 1, '']], [['d', 1, '']], [[' ', 1, '']]
[['m', 4, '1']] [['l', 2, '1']], [['y', 1, '1']], [['d', 1, '1']], [[' ', 1, '1']]
[['l', 2, '11']] [['y', 1, '11']], [['d', 1, '11']], [[' ', 1, '11']]
[['y', 1, '111']] [['d', 1, '111']], [[' ', 1, '111']]
[['d', 1, '1111']] [[' ', 1, '1111']]
Shannon's Encoded Code:
a : 0
m : 10
l : 110
y : 1110
d : 11110
: 11111
Compressed file generated as compressed.txt
m : 10
a : 0
l : 110
a : 0
y : 1110
a : 0
l : 110
a : 0
m : 10
: 11111
m : 10
a : 0
d : 11110
a : 0
m : 10
Your UNCOMPRESSED data is:
```

```
compressed - Notepad
File Edit Format View Help
1001100111001100101111110011110010
```

Discussions

Huffman encoding is more advantageous than Shannon encoding for the following reasons.

- Greedy algorithm
- Minimizes the length of code generated depending on frequency

Applications

- Huffman encoding is widely used in compression formats like GZIP, PKZIP (winzip) and BZIP2.
- Multimedia codecs like JPEG, PNG and MP3 use Huffman encoding.
- Huffman encoding still dominates the compression industry since newer arithmetic and range coding schemes are avoided due to their patent issues.
- Brotli compression algorithm by Google compresses data using a combination of a modern variant of the LZ77 algorithm, Huffman coding and 2nd order context modeling.

