Faculty of Engineering & Technology

Electrical & Computer Engineering Department

INFORMATION AND CODING THEORY

ENEE5304

Report For Course assignment

Prepared by:

Ahmed Zubaidia 1200105

Omar Masalmah 1200060

Instructor: Dr. Wael Hashlamoun

Section : 1

Date: 8/1/2024

Contents

Table of Figures :

Introduction

Purpose of the Report
This report investigates the implementation and efficiency of Huffman Coding as a method of data compression. With digital data growing exponentially, efficient storage and transmission have become imperative. Huffman Coding offers a solution by encoding data in variable-length codes, reducing the overall size of the data.

Understanding Huffman Coding

Huffman Coding is an optimal prefix code that is commonly used for lossless data compression.
The technique uses a binary tree to represent the most efficient binary code of different lengt hs for each symbol based on its frequency in the data set [1].

Application and Relevance

The relevance of the technique extends to text compression, telecommunications and multimedia. This report applies Huffman Coding to compress A single text - to Build a Fire by Jack London - to demonstrate the algorithm's effectiveness and practical utility [2].

By looking at the character frequency of the text and applying Huffman coding we show that compression efficiency is achievable relative to classical fixed-length Coding schemes such as ASCII. Results are expected to highlight the significance of Huffman Coding in practical applications where data compression is required.

Method (Theoretical Background)

Huffman Coding Algorithm Overview:

Huffman Coding is lossless data compression that uses a binary tree to assign variable-length codes to input characters with lengths based on their frequencies. The algorithm has several key steps:

1.Character Frequency Analysis: The first step would be analyzing the text to determine how frequently a character seems.

2. Tree Construction: Characters are queued in a binary tree with each leaf node representing a character and its frequency. The tree is constructed by removing the two nodes of least probability repeatedly and replacing them with a new node as the parent with a frequency the same as the sum of their frequencies.

3. Code Assignment: After the tree is built, each character gets a binary code showing the path to its corresponding leaf from the base of the tree. High frequency characters have shorter codes, and lower frequency characters have a longer code [1].

Huffman Coding vs. ASCII Encoding

In ASCII encoding, every character has a specific amount of bits (usually eight bits a character). In contrast, Huffman Coding assigns variable-length codes based on character frequency, with more frequent characters using fewer bits. This technique will decrease the total number of bits required to represent the text, in texts where certain characters show up more often compared to others [3].

Calculating Entropy and Efficiency

The effectiveness of Huffman Coding is often measured against entropy (minimum possible average code length per symbol) of data. Entropy is computed by the probabilities of every character in the text. Huffman Coding approximates this optimal length for maximum efficiency [4].

Implementing Huffman Coding

The underlying theory behind Huffman Coding is consistent but implementation varies. The choice of programming language and certain algorithmic optimizations may influence the actual compression achieved. Implementation using [chosen programming language] with its efficient data structures and libraries is used in this report.

Results and Analysis

Implementation and Character Frequency

Compressing the text to Build A Fire by Jack London using Huffman Coding algorithm. The text is 37,705 characters. Frequency analysis was performed to determine the frequency of each character which is important for building the Huffman tree. The most common characters were spaces,' e',' t',' a', and others, which indicated their commonality in English and the text.

Huffman Tree Construction and Code Assignment

A Huffman tree based on character frequencies was constructed. Within this tree, each leaf node represented a character from the text, and the path from the root to the leaf represented the binary code of that character. Characters with higher frequencies such as space (''),' e', and 't' received shorter codes while less frequent characters received longer codes.

Compression Results

Application of Huffman Coding to the text produced the following results:

- Total Characters: 37,705
- Entropy 4.25665 bits / character.
- Avg Bits per Character with Huffman Coding: 4.30245
- Total Bits for ASCII Encoding (NASCII): 301,64 0
- Total Bits for Huffman Encoding (Nhuffman): 162,224
- Compression Percentage: 53.78 %

These results demonstrate the efficacy of Huffman Coding. The compression ratio shows that the encoded data is significantly smaller than with ASCII encoding. The average bits per character with Huffman Coding is close to the theoretical bounds, indicating an efficient compression close to the theoretical bounds.

Analysis of Huffman Codes

The Huffman codes were generated for each differing length character. For example, common characters such as space ('') got a short code of' 111'and less common characters got longer codes. This variable code length is the heart of Huffman's algorithm efficiency adaptation to the actual text's character distribution.

Comparative Analysis with ASCII

Comparing the Huffman and ASCII encoding methods shows that Huffman Coding provides significant improvements in compression. ASCII, with its fixed-length encoding, doesn't adapt to character frequency, resulting in more extensive data representation. Huffman Coding is adaptable to compress data more effectively, especially for texts with skewed frequency distribution of characters.

Picture of the output

```
----------------------------------------------------------
Entropy:   4.25665 bits/character
Average:   4.30245 bits/character
Number Of Bits For ASCII:  301640
Number Of Bits For Huffman:  162224
Percentage Of Compression:   53.78%
--------------------------------------------------------------
```

Figure 1: Output of Entropy & AVG &number of bits

Symbol	Probability	Codewords	Length of codeword
!	7.956504442381647e-05	10101100111111	14
"	5.304336294921098e-05	00001001011110	14
'	0.0005304336294921098	0000100100	10
(space)	0.1869248110330195	111	3
,	0.011563453122927994	001001	6
-	0.0023604296512398887	00001000	8
.	0.010979976130486672	000011	6
:	5.304336294921098e-05	00001001011111	14
;	0.0006895637183397427	0000101011	10
?	2.652168147460549e-05	101011000001010	15
A	0.0013260840737302744	000010011	9
B	0.0006895637183397427	0010110000	10
C	0.0003182601776952659	00001010100	11
D	0.00013260840737302744	000010010110	12
E	0.00013260840737302744	1010110011110	13
F	0.0002386951332714494	101011001010	12
G	5.304336294921098e-05	10101100000100	14
H	0.0031030367325288423	00101101	8
I	0.0013791274366794855	000010100	9
J	2.652168147460549e-05	101011000001011	15
K	5.304336294921098e-05	10101100000111	14
L	7.956504442381647e-05	0000100101110	13
M	0.00013260840737302744	000010010100	12
N	0.0002652168147460549	101011001110	12
O	0.0002386951332714494	101011001011	12
P	0.00013260840737302744	000010010101	12
R	5.304336294921098e-05	10101100111110	14
S	0.0007691287627635592	0010110001	10
T	0.002864341599257393	00001011	8
U	5.304336294921098e-05	10101100000110	14

Figure 2: symbol, probability , codewords length part 1

U	5.304336294921098e-05	10101100000110	14
W	0.00045086858506829334	10101100100	11
Y	0.00018565177032223843	101011000000	12
a	0.058719002784776556	1001	4
b	0.012146930115369315	001010	6
c	0.02034212969102241	110001	6
d	0.04004773902665429	11010	5
e	0.10295716748441851	010	3
f	0.02081951995756531	110111	6
g	0.016390399151306193	101010	6
h	0.05731335366662246	1000	4
i	0.0512133669274632	0011	4
j	0.0005039119480175043	10101100110	11
k	0.008009547805330858	0010111	7
l	0.02981036997745657	10100	5
m	0.017849091632409494	110000	6
n	0.05482031560800955	0111	4
o	0.05203553905317597	0110	4
p	0.011033019493435884	001000	6
q	0.00045086858506829334	10101100001	11
r	0.03922556690094152	11001	5
s	0.0468372894841533	0001	4
t	0.07502983689165893	1011	4
u	0.02116430181673518	00000	5
v	0.004747380983954383	10101101	8
w	0.020448216416920833	110110	6
x	0.0009017371701365867	1010110001	10
y	0.009256066834637316	1010111	7
z	0.0016178225699509349	001011001	9
—	0.00037130354064447685	00001010101	11

Figure 3: symbol, probability, codewords length part 2

Conclusions

Efficacy of Huffman Coding:

Application of Huffman Coding to "To Build a Fire" by Jack London shows significantly reduced data size compared with traditional ASCII encoding. The algorithm used the frequency of characters to assign variable-length codes, thus producing a 53.78% compression ratio. This large efficiency gain demonstrates the benefit of Huffman Coding when data compression is required.

Near Optimal Efficiency

The average bits per character for Huffman Coding approached the calculated entropy of 4.25665 bits/character, suggesting near optimal compression for the data set. This efficiency demonstrates that Huffman Coding can adjust the code length for each character according to its frequency distribution within the text.

Practical Implications

The results validate Huffman Coding as practical in many domains where efficient data storage and transmission are required. Its flexibility lets it compress non-textual data in addition to image and audio files. Huffman Coding is fundamental to data compression algorithms and still relevant in modern data processing and communication systems.

References

[1] "1," [Online]. Available: https://brilliant.org/wiki/huffman-encoding/.

[2] "2," [Online]. Available: https://byjus.com/gate/huffman-coding-notes/#:~:text=for%20uncommon%20symbols.-,Applications%20of%20Huffman%20Coding,space%20required%20for%20digital%20images..

[3] "3," [Online]. Available: https://web.stonehill.edu/compsci/lc/textcompression.htm#:~:text=Unlike%20the%20ASCII%20code%20which,of%20bits%20for%20each%20character..

[4] "4," [Online]. Available: https://brilliant.org/wiki/entropy-information-theory/.

Appendix:

```
# Importing libraries
import heapq
from collections import Counter
import math
import docx2txt

# Read the text file
def readText():
    text = docx2txt.process("To+Build+A+Fire+by+Jack+London.docx")
    return text


def buildFrequencyDict(text):
    frequencyDict = {}
    for char in text:
        #change to lower
        #char = char.lower()
        if char == ' ':
                # rename it
                    char = '(space)'
        if char != '\n':  # Skip newline characters
            if char not in frequencyDict:
                frequencyDict[char] = 0
            frequencyDict[char] += 1
    return frequencyDict


# Calculate the total number of characters
def calculateTotalCharacters(frequencyDict):
    totalCharacters = 0
    for key, frequency in frequencyDict.items():
        totalCharacters += frequency
    return totalCharacters

# Calculate probabilitiies
def calculateProbabilities(frequencyDict, totalCharacters):
    probabilitiesOfEachChar = {}
    # sort
    frequencyDict = dict(sorted(frequencyDict.items(), key=lambda item:
item[1], reverse=True))
    print("----------------------------------------------------------")
    print("{:<10} {:<12} {:<20}".format("Symbol", "Frequency", "Probability
Of Char"))
    for key, frequency in frequencyDict.items():
        probabilitiesOfEachChar[key] = frequency / totalCharacters

        print(f"{key:<10} {frequencyDict[key]:<12}
{probabilitiesOfEachChar[key]}")
    print("----------------------------------------------------------")

    return probabilitiesOfEachChar

# Calculate Entropy
def calculateEntropy(probabilitiesOfEachChar):
```

```
    entropy = 0
    for key, probability in probabilitiesOfEachChar.items():
        entropy += -probability * math.log(probability, 2)

    return entropy


#****------------------***************-------------****************--------
----***************-----------***********
def buildHuffmanTree(frequency_dict):
    # Create a list of lists
    heap = []
    for char, frequency in frequency_dict.items():
        # Add the frequency and the character to the list
        heap.append([frequency, [char, ""]])


    # Convert the list into a min-heap
    heapq.heapify(heap)

    # Continue merging nodes until there is only one node (Huffman tree)
    while len(heap) > 1:
        # Pop the two nodes with the lowest weights from the heap
        low = heapq.heappop(heap)
        high = heapq.heappop(heap)

        # Add '0' as a prefix to the codewords of the low-weight node
        for pair in low[1:]:
            pair[1] = '0' + pair[1]

        # Add '1' as a prefix to the codewords of the high-weight node
        for pair in high[1:]:
            pair[1] = '1' + pair[1]

        # Merge the two nodes and push the result back into the heap
        heapq.heappush(heap, [low[0] + high[0]] + low[1:] + high[1:])

    codewordsForTheCharacters = sorted(heapq.heappop(heap)[1:], key=lambda p:
(len(p[-1]), p))
    return codewordsForTheCharacters



#****************************************************************************
********************************************************
# Build huffman tree in another way
class Node:
    def __init__(self, char=None, frequency=0, left=None, right=None):
        self.char = char
        self.frequency = frequency
        self.left = left
        self.right = right

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

def buildHuffmanTree2(frequency_dict):
```

```
    heap = []
    for c, frequency in frequency_dict.items():
        node = Node(char=c, frequency=frequency)
        heapq.heappush(heap, node)
    heapq.heapify(heap)

    while len(heap) > 1:
        low = heapq.heappop(heap)
        high = heapq.heappop(heap)
        merged_node = Node(frequency=low.frequency + high.frequency,
left=low, right=high)
        heapq.heappush(heap, merged_node)

    codewordsForTheCharacters = traverse_tree(heap[0])
    return codewordsForTheCharacters

def traverse_tree(node, code="", result=None):
    if result is None:
        result = []
    if node.char is not None:
        result.append((node.char, code))
    if node.left is not None:
        traverse_tree(node.left, code + "0", result)
    if node.right is not None:
        traverse_tree(node.right, code + "1", result)
    return sorted(result, key=lambda p: (len(p[1]), p))


#*****************************************************************************
*************************************************************************

# Find the average number of bits/character for the whole story using the
Huffman code.
def findAverageNumberOfBits(codewords, probabilitiesOfEachChar):
    averageNumber = 0
    for char, codeword in codewords:
        averageNumber += probabilitiesOfEachChar[char] * len(codeword)
#     print("Average Number Of Bits For The Whole Story: ",averageNumber)
    return averageNumber


# Find the  percentage of compression accomplished by using the Huffman
encoding as compared to ASCII code.
def findPercentageOfCompression(NASCII, Nhuffman):
    percentageOfCompression = (Nhuffman / NASCII) * 100
#     print("Percentage Of Compression: ",f"{percentageOfCompression: .2f}")
    return percentageOfCompression




# Main function
if __name__ == "__main__":
    text = readText()

    frequencyDict = buildFrequencyDict(text)
    totalCharacters = calculateTotalCharacters(frequencyDict)
```

```
    print("-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+-+-")
    # Step -a- & -b-
    print("Total Characters: ",totalCharacters)
    probabilitiesOfEachChar = calculateProbabilities(frequencyDict,
totalCharacters)

    # Step -c-
    entropy = calculateEntropy(probabilitiesOfEachChar)
    print("Entropy: ",f"{entropy: .5f} bits/character")

    # Step -1-
    codewords = buildHuffmanTree2(frequencyDict)

    # Step -3-
    # Average number of bits/character for the whole story using the Huffman
code
    averageNumberOfBits = findAverageNumberOfBits(codewords,
probabilitiesOfEachChar)
    print("Average: ",f"{averageNumberOfBits: .5f} bits/character")

    # Step -2-
    # Number of bits needed to encode the full story
    NASCII = totalCharacters * 8
    print("Number Of Bits For ASCII: ",NASCII)

    # Step -4-
    # Total number of bits needed to encode the entire story using Huffman
code.
    Nhuffman = 0
    for char, codeword in codewords:
        Nhuffman += frequencyDict[char] * len(codeword)
    print("Number Of Bits For Huffman: ",Nhuffman)


    # Step -5-
    # Percentage of compression accomplished by using the Huffman encoding as
compared to ASCII code.
    percentageOfCompression = findPercentageOfCompression(NASCII, Nhuffman)
    print("Percentage Of Compression: ",f"{percentageOfCompression: .2f}%")

    # Fill in the table to showcase some of your results, sorted as elphapet
    codewords.sort()  # Sort alphabetically based on the character

    print("-----------------------------------------------------------------
")
    print("{:<10} {:<27} {:<20} {:<15}".format("Symbol", "Probability",
"Codewords", "Length of codeword"))
    for char, codeword in codewords:
        print(f"{char:<10} {probabilitiesOfEachChar[char]:<27} {codeword:<20}
{len(codeword):<15}")
    print("-----------------------------------------------------------------
")
```