BIRZEIT UNIVERSITY

Faculty of Engineering & Technology
Electrical & Computer Engineering Department

ENEE5304, INFORMATION AND CODING THEORY

Course Project Report

“ Huffman Code”

Prepared by: Noor Hamayel 1202853
Dunia Al'amal Hamada 1201001

Instructor: Dr.Wael Hashlamoun

Section: 1

Jan-2024

Table of content

introduction

In this project, we use Jack London's story "To Build A Fire" as our text and explore the world of Huffman coding, an essential method in data compression. The core problem addressed here is the efficient encoding of text data. Huffman coding, so named after its creator David A. Huffman, is a popular lossless data compression method in computer engineering. By using their frequencies as a basis, this technique reduces the average length of codes assigned to input characters. Our goal is to comprehend and apply Huffman coding by examining "To Build A Fire" and assessing how well it preserves the original narrative while reducing text. This project closes the gap between theoretical ideas and real-world implementation while also improving our understanding of data compression techniques.

Theory

Introduction to Huffman Coding

Huffman Coding, developed by David Huffman, is a data compression technique renowned for reducing data size without compromising detail, especially effective in processing data with frequently recurring characters. This method employs a greedy algorithm that assigns code sizes based on character frequency: common characters receive shorter, variable-length codes, while rarer characters get longer ones. By utilizing variable-length encoding, Huffman Coding uniquely tailors code lengths for each character in the data stream, striking a balance between efficiency and integrity in data compression.[1]

Prefix Rule

Huffman coding prefers variable-length, prefix-free codes over fixed-length for better compression. In prefix-free codes, no codeword forms the beginning of another, ensuring clear, unique decoding of messages. This focus on efficient, prefix-free coding is key for optimal data compression.[2]

How does Huffman Coding work?

Huffman Coding works in two main steps: building a Huffman Tree and traversing this tree to assign and decode codes for each unique character. The process begins by calculating the frequency of each character in the input data. These characters, sorted by frequency, are then used to create a min-heap or priority queue. A leaf node is made for each character, and the two nodes with the lowest frequency are repeatedly extracted to form a new node, with their combined frequency as the root, until only one node remains. This final node is the root of the Huffman Tree. The path to each character in this tree defines its Huffman Code, used for efficient data compression.[3]

Results and Analysis

Table results

The table below shows a summary of the Huffman coding process for a text. It lists various characters found in the text, including letters, punctuation marks, and space. Each character is associated with several attributes: Frequency, Probability, Entropy, Huffman Code, and bits count.

Character	Frequency	Probability	Entropy	Huffman Code	Bits Count
space	7049	0.1869	0.4523	111	3
!	3	0.0001	0.0011	00011111011011	14
"	2	0.0001	0.0008	00011111011001	14
'	20	0.0005	0.0058	00011111001	11
,	436	0.0116	0.0744	000110	6
-	89	0.0024	0.0206	000111111	9
.	414	0.0110	0.0715	000100	6
:	2	0.0001	0.0008	00011111011010	14
;	26	0.0007	0.0072	0001111000	10
?	1	0.0000	0.0004	00011111011000	14
a	2264	0.0600	0.2436	1001	4
b	484	0.0128	0.0807	100000	6
c	779	0.0207	0.1156	110110	6
d	1515	0.0402	0.1863	11010	5
e	3887	0.1031	0.3379	010	3
f	794	0.0211	0.1173	00000	5
g	620	0.0164	0.0974	100001	6
h	2278	0.0604	0.2446	1010	4
i	1983	0.0526	0.2235	0110	4
j	20	0.0005	0.0058	00011111010	11
k	304	0.0081	0.0561	1011000	7
l	1127	0.0299	0.1514	10001	5
m	678	0.0180	0.1042	101101	6
n	2077	0.0551	0.2304	0111	4
o	1971	0.0523	0.2226	0011	4
p	421	0.0112	0.0724	000101	6
q	17	0.0005	0.0050	00011111000	11
r	1481	0.0393	0.1834	10111	5
s	1795	0.0476	0.2091	0010	4
t	2937	0.0779	0.2868	1100	4
u	800	0.0212	0.1179	00001	5
v	179	0.0047	0.0366	0001110	7
w	788	0.0209	0.1166	110111	6
x	34	0.0009	0.0091	0001111001	10
y	356	0.0094	0.0635	1011001	7
z	61	0.0016	0.0150	000111101	9
—	14	0.0004	0.0042	000111110111	12

Summary

The image below shows a summary of Huffman coding compression results, indicating a reduction from 301648 to 159063 bits, achieving an average of 4.22 bits per character and a 52.73% compression rate.

```
Summary:
Total Number Of Characters:   37706
N_ASCII:                      301648 bits
N_Huffman:                    159063 bits
Average Bits/Character (Huffman):4.22 bits/character
Entropy:                      4.17 bits/character
Compression Percentage:       52.73%
```

the probabilities, the lengths of the codewords, and the codewords for the following symbols : ['a', 'b', 'c', 'd', 'e', 'f', 'm', 'z', Space , '.']

Symbol	Probability	Huffman Code	Code Length
a	0.0600	1001	4
b	0.0128	100000	6
c	0.0207	110110	6
d	0.0402	11010	5
e	0.1031	010	3
f	0.0211	00000	5
m	0.0180	101101	6
z	0.0016	000111101	9
space	0.1869	111	3
.	0.0110	000100	6

Appendix

```
import heapq
from collections import Counter
from math import log2

# Calculates the frequency of each character in the text
def calculate_frequencies(text):
   frequencies = Counter(text)  # Uses Counter to count the frequency of each 
character
   return frequencies

# Calculates probabilities of each character based on their frequency
def calculate_probabilities(frequencies, total_characters):
   probabilities = {char: freq / total_characters for char, freq in 
frequencies.items()}
   return probabilities

# Implements the Huffman coding algorithm
def huffman_coding(probabilities):
   heap = [[weight, [char, ""]] for char, weight in probabilities.items()]
   heapq.heapify(heap)  # Creates a min-heap based on character frequencies
   while len(heap) > 1:
       lo = heapq.heappop(heap)  # Pop two smallest items
       hi = heapq.heappop(heap)
       for pair in lo[1:]:
           pair[1] = '0' + pair[1]  # Append '0' for lower frequency character
       for pair in hi[1:]:
           pair[1] = '1' + pair[1]  # Append '1' for higher frequency character
       heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
   huffman_codes = {char: code for char, code in heap[0][1:]}
   return huffman_codes

# Calculates total bits needed for encoding text using Huffman codes
def calculate_bits_needed(text, encoding):
   return sum(len(encoding[char]) for char in text)

def main():
   with open(r"/Users/noormacbook/Desktop/Coding/Assigment/story.txt", "r") as file:
       story = file.read().replace('\n', '').lower()  # Read and preprocess the story

   total_characters = len(story)
```

```
    frequencies = calculate_frequencies(story)  # Calculate character frequencies
    probabilities = calculate_probabilities(frequencies, total_characters)  # Calculate probabilities

    # Calculate the entropy of the alphabet
    entropy = sum(-p * (p and log2(p)) for p in probabilities.values())

    huffman_codes = huffman_coding(probabilities)  # Generate Huffman codes

    n_ascii = total_characters * 8  # Total number of bits using ASCII encoding
    n_huffman = calculate_bits_needed(story, huffman_codes)  # Total number of bits using Huffman encoding
    compression_percentage = ( n_huffman / n_ascii) * 100 # Calculate compression percentage

    # Print a table with character, frequency, probability, entropy, Huffman code, and bit count
    header =
f"{'Character':<12}{'Frequency':<12}{'Probability':<15}{'Entropy':<20}{'Huffman Code':<20}{'Bits Count':<12}"
    print(header)
    print('-' * len(header))
    for char in sorted(frequencies.keys()):
        display_char = "space" if char == ' ' else char  # Replace space ' ' with 'space'
        freq = frequencies[char]
        prob = probabilities[char]
        ent = -prob * log2(prob) if prob > 0 else 0
        code = huffman_codes[char]
        bits_count = len(code)

print(f"{display_char:<12}{freq:<12}{prob:<15.4f}{ent:<15.4f}{code:<20}{bits_count:<12}")

    # Print summary of results
    print("\nSummary:")
    print(f"{'Total Number Of Characters:':<30}{total_characters}")
    print(f"{'N_ASCII:':<30}{n_ascii} bits")
    print(f"{'N_Huffman:':<30}{n_huffman} bits")
    print(f"{'Average Bits/Character (Huffman):':<30}{n_huffman / total_characters:.2f} bits/character")
    print(f"{'Entropy:':<30}{entropy:.2f} bits/character")
```

```
    print(f"{'Compression Percentage:':<30}{compression_percentage:.2f}%")


# Specify the symbols
    symbols = ['a', 'b', 'c', 'd', 'e', 'f', 'm', 'z', ' ', '.']

    print()

    # Print a table for the specified symbols
    header = f"{'Symbol':<12}{'Probability':<15}{'Huffman Code':<20}{'Code
Length':<12}"
    print(header)
    print('-' * len(header))
    for symbol in symbols:
        display_char = "space" if symbol == ' ' else symbol  # Replace space with
'space'
        prob = probabilities.get(symbol, 0)
        code = huffman_codes.get(symbol, '')
        code_length = len(code)
        print(f"{display_char:<12}{prob:<15.4f}{code:<20}{code_length:<12}")


if __name__ == "__main__":
    main()
```

Conclusion

In summary, this project effectively used Huffman coding to compress the text "To Build A Fire," reducing the overall size of the data without compromising the integrity of the content. The project improved our knowledge of data compression by bridging the theoretical and practical applications of Huffman coding. With an average of 4.22 bits per character, we saw a 52.73% compression rate, highlighting the effectiveness of Huffman coding. This project improved our ability to implement algorithms and highlighted the significance of effective data encoding in computer engineering.

References

[1] https://www.scaler.com/topics/huffman-coding/

Access time: 9:15 AM, 7/1/2024

[2] https://home.cse.ust.hk/faculty/golin/COMP271Sp03/Notes/MyL17.pdf

Access time: 9:44 AM, 7/1/2024

[3]https://www.programiz.com/dsa/huffman-coding#:~:text=Huffman%20coding%20first%20creates%20a,concept%20of%20prefix%20code%20ie.

Access time: 11:24 AM, 7/1/2024