CIE 425 Project 1 Report

Name: Ahmed Muhammad Tarek
ID: 201500885
E-mail: s-ahmed.tarek@zewailcity.edu.eg

Table of Contents

Introduction

Huffman code is a lossless prefix-free data compaction scheme named after David A. Huffman who developed it during his study at MIT. Huffman submitted it as a term paper instead of taking a final examination in an information theory class he was taking that was taught by Robert Fano. Claude Shannon and Robert Fano independently developed a coding scheme closely related to Huffman code which is currently known as Shannon-Fano code. Both of the coding schemes require foreknowledge of the source alphabet probability distribution. They give the same results when the probabilities of the symbols are negative powers of two. However, Huffman code generally gives better results. It can be proven that Huffman code is the optimal symbol code. [1]

Theory

Huffman code is the optimal prefix-free symbol code which is proven to minimize the expectation of the codeword length $\bar{L} = \sum_{<i>} p_i l_i$ where p_i is the probability of the i^{th} symbol and l_i is the length of the i^{th} codeword. It can be proven that the maximum redundancy in the Huffman code is $p_i + 0.086$ where p_i is the highest probability in the source alphabet probability distribution (redundancy is the difference between the average codeword length and the entropy of the source). [1][2]

Huffman encoding algorithm [3]

1. Sort the source symbols descendingly in a table according to their probabilities
2. Merge the two source symbols with the smallest probabilities into an auxiliary symbol and add their probabilities
3. Assign 0 to one of the two merged symbols and 1 to the other
4. Sort the new entries descendingly
5. Mark the transitions of the entries from one position to another after each iteration
6. Repeat until the number of entries is equal to two
7. Assign 0 to one of the two entries and 1 to the other
8. To find the codeword of a certain symbol, work backwards in the formed schematic till you reach the desired symbol

Example (1)

Assume a discrete source whose alphabet is $\zeta = \{s_1, s_2, s_3, s_4\}$ and probability distribution $p(\zeta) = [0.4, 0.3, 0.2, 0.1]$

s_1	0.4		s_1	0.4		s_2, s_3, s_4	0.6	0
s_2	0.3		s_2	0.3	0	s_1	0.4	1
s_3	0.2	0	s_3, s_4	0.3	1			
s_4	0.1	1						

Working backwards we get:

Symbol	Codeword
s_1	1

s_2	00
s_3	010
s_4	011

In order to evaluate the efficiency of the code, we compute the entropy and the average length of codewords.

$$H(\zeta) = \sum_{i=0}^{3} -p_i \log p_i \approx 1.846$$

$$\bar{L} = \sum_{i=0}^{3} p_i l_i = 1.9$$

$$\eta = \frac{H(\zeta)}{\bar{L}} \approx 0.973\ (efficiency)$$

In the first iteration, an ambiguity is encountered. The new probability resulting from the addition is equal to one the probabilities in the table. On sorting, the new entry can be placed as high as possible or as low as possible provided that the choice made is maintained throughout the execution due to the deterministic nature of the implementation. The arbitrariness of the choice of the position of the new entry in case of equality does not affect the efficiency of the code, however; the variance of the code can be affected by this choice whose practical significance is going to be discussed later on.

It can be readily seen that the Huffman encoding process can be represented by a binary tree, each code word represents a path to a certain node which represents one of the source symbols.

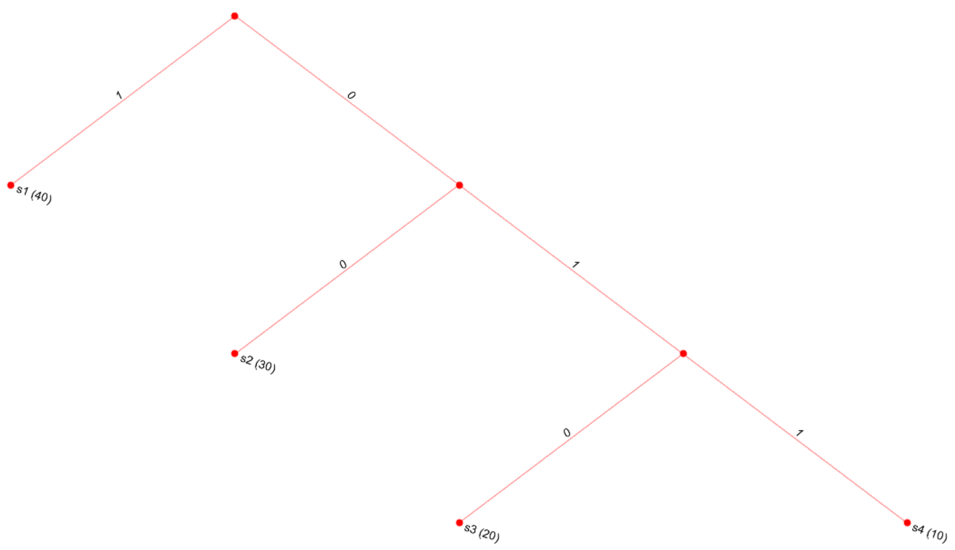


Fig (1) Huffman tree of example (1) using the implemented tree visualization function

Algorithms and Pseudocodes of Implemented functions

Huffman Dictionary Function (*HuffDict)*

Inputs:

1. *Symbols* which is a string vector of the source alphabet
2. *Prob* which is a vector of symbol probabilities
3. *Var* which is a Boolean input
   a. 0 represents minimum variance
   b. 1 represents maximum variance

Outputs:

1. *dict* is the Huffman dictionary that maps each symbol to its codeword
2. *averagelen* is the average length of the code
3. *variance* is the variance of the code
4. *fulldict* is equivalent to *dict* but it has an additional column for symbol frequencies
5. *tree* stores all the algorithm execution stages.

Setup:

1. Create a two-dimensional array of structures *tree* of size $(L)(L-1)$ where L is the number of symbols, each structure has three fields: Symbol, Frequency and Codeword
2. Multiply *prob* by 100 to convert the probabilities into frequencies
3. Sort *prob* and *symbols* descendingly according to *prob*
4. Store a symbol and its frequency at each entry of the first column of *tree* and leave all the other entries empty.

Pseudocode:

```
$k = L - 1$;
WHILE ($k > 1$)
    IF ($k \geq 1$)
        IF ($var = 0$)
            prob(k) = prob(k) + prob(k+1) + $1 \times 10^{-6}$ (in case of
            equality the new symbol is placed as high as possible)
            Delete prob(k+1);
            symbols(k) = concatenate(symbols(k), symbols(k+1));
            Delete symbols(k+1);
            Sort prob and symbols descendingly according to prob
        ELSE IF ($var = 1$)
            prob(k) = prob(k) + prob(k+1) (in case of equality the new
            symbol is placed as low as possible)
```

```
                Delete prob(k+1);
                symbols(k) = concatenate(symbols(k), symbols(k+1));
                Delete symbols(k+1);
                Sort prob and symbols descendingly according to prob;
                END IF;
            END IF;
            Assign the new symbols and frequencies to the $(L-k+1)th$ column of tree
            using a FOR loop;
            Set the $k^{th}$ codeword to 0 and the $k+1^{th}$codeword to 1 in the $(L-k)^{th}$
            column;
END WHILE;
FOR ($i = 1{:}L$)
    FOR ($j = 2{:}L-1$)
        FOR ($k = L-i+1{:}-1{:}1$)
            IF (symbols(i) is part of the symbol in tree(j,k))
                concatenate(symbol in tree(j,k),symbol in tree(i,1));
            ENDIF
        END FOR;
    END FOR;
END FOR;
fulldict= tree(:,1);
dict = remove(fulldict,Frequency);
averagelen = average length of the code;
variance = variance of the code;
Return dict, averagelen, variance, fulldict, tree;
```

Note: The function computes the average length and the variance of the code using the formulas:

$$\bar{L} = \sum_{<i>} p_i l_i$$

$$\sigma^2 = \sum_{<i>} p_i (l_i - \bar{L})^2$$

Huffman Encoder Function (*HuffEncode)*

Inputs:

1. *text* which is the text to be encoded
2. *dict* which is the Huffman dictionary that maps each symbol to its codeword

Outputs:

1. *encoded* which is the text after the encoding process

Setup:

1. Create *encoded* which is an array of strings of length L which is the length of *text*
2. m is the number of symbols which can be found from the dictionary

Pseudocode:

```
FOR ($i = 1 : L$)
    FOR ($j = 1 : m$)
        IF ($m^{th}$ symbol in dict equals $i^{th}$ character in text)
            encoded(i) = codeword of the $m^{th}$ symbol;
        END IF;
    END FOR;
END FOR;
```

Huffman Decoder Function (*HuffEncode)*

Inputs:

1. *encoded* which is the text to be decoded
2. *dict* which is the Huffman dictionary that maps each symbol to its codeword

Outputs:

1. *decoded* which is the text after the decoding process

Setup:

1. Create *decoded* which is an array of strings of length L which is the length of *encoded*
2. m is the number of symbols which can be found from the dictionary

Pseudocode:

```
FOR ($i = 1 : L$)
    FOR ($j = 1 : m$)
        IF ($m^{th}$ codeword in dict equals $i^{th}$ character in text)
```

```
                    decoded(i) = $m^{th}$symbol;
              END IF;
        END FOR;
END FOR;
```

The function which generates all Binary combinations of size n (*bincombinations)*

Inputs:

1. *n* is the size of the desired binary combinations

Outputs:

1. *allsequences* which is an array of strings of all the possible binary combinations

Pseudocode:

```
FOR ($i = 0: 2^n - 1$)
      allsequences(i+1) = string(dectobin($i$,$n$));
END FOR;
```

Note: dectobin takes the decimal number and the size of the desired binary number, then it converts the decimal number to its *n*-bit binary representation.

The function which calculates probabilities of source symbols (*CalculateCharProb)*

Inputs:

1. *text* which is the text we want to calculate the probability distribution of its characters
2. *chars* which is the source alphabet

Outputs:

1. *Prob* which is a vector of symbol probabilities

Pseudocode:

```
$L =$ length(chars);
FOR ($i = 1: L$)
      prob(i)= mean(text==chars(i)) (text==chars(i) is an array of boolean variables whose
      sum represents the number of occurrences of the $i^{th}$ character)
END FOR;
```

The function that maps each codeword to its actual node in the tree (*MapCodetoNode)*

Inputs:

1. *code* which is the codeword that is going to be mapped

Outputs:

1. *node* which is the index of the position of the node in the tree

Explanation:

Assume that the maximal binary tree has the following form:

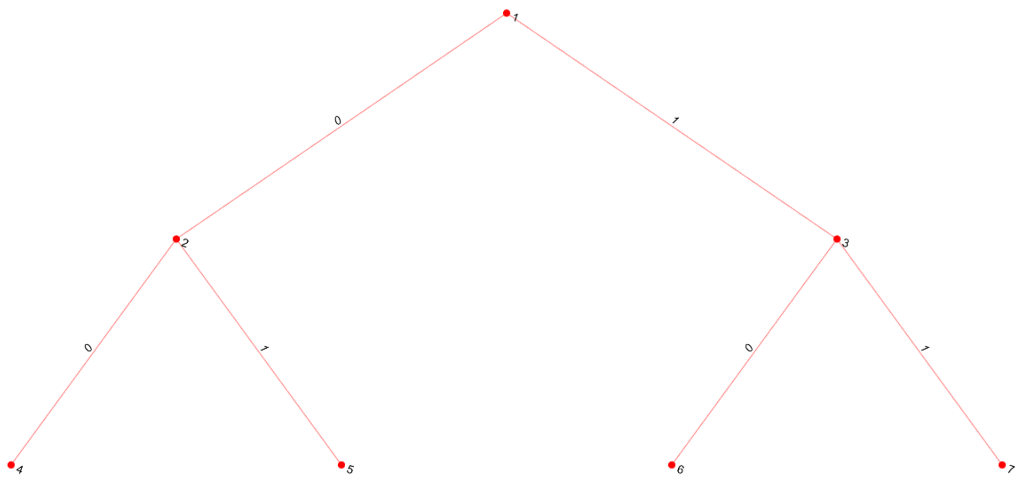


Fig (2) 2-level binary tree generated using the implemented tree visualization function (*HuffTree*)

Node index	Binary representation of node index	Path
2	10	0
3	11	1
4	100	00
5	101	01
6	110	10
7	111	11

It can be readily observed that we can map the path to the binary node index by padding 1 to the from the left side to the path itself.

Pseudocode:

```
node = concatenate(1,code);
```

Huffman Tree Visualization Function (*HuffTree)*

Inputs:

1. *fulldict* which is a Huffman dictionary with an additional field for symbol frequencies
2. *tree* which stores all the Huffman coding algorithm execution stages

Outputs:

1. *gtree* which is the binary Huffman tree (each codeword represents a path to its symbol on the tree)

Pseudocode:

```
Levels = length of the longest codeword+1;
Construct the maximal binary tree whose number of levels is equal to levels which
contains all possible paths.
FOR ($i = Levels - 1: -1: 1$)
    codes = all codewords of size $i$;
    allcodes = all possible paths (codewords) of size $i$; (generated using
    bincombinations function)
    indices  = NOT(allcodes==codes); (indices of nodes that are going to be
    removed)
    codes_to_remove = allcodes indexed by indices; (eliminate the actual
    codewords)
    nodes_to_remove = MapCodetoNode(codes_to_remove); (maps all codes to
    their actual nodes in the gtree)
    Remove all nodes in nodes_to_remove from gtree;
END FOR;
```

Note: The pseudocode gives a general idea of the function implementation, however; the implementation is more complicated.

Entropy Function (*Entropy)*

Inputs:

1. *prob* which is a symbol of vector probabilities

Outputs:

1. *entropy* which is the entropy of the source

Pseudocode:

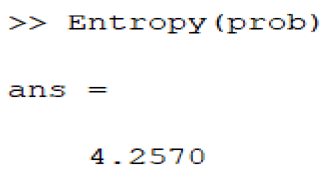
It directly applies the entropy formula to the *prob* vector.

```
$entropy = -prob \cdot \log prob$; ( · represents the vector dot product which is equivalent
to element by element multiplication then computing the sum of the resulting array)
```

Results and Discussion

Note: The results can be replicated by executing the main testing script sequentially

1.Entropy of symbol probabilities of the given text file



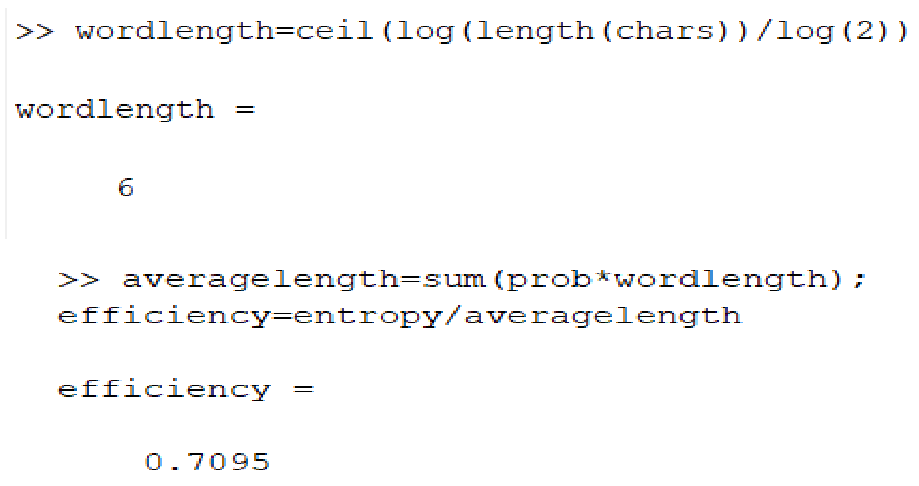
```
>> Entropy(prob)

ans =

    4.2570
```

Fig (3) Entropy of the source using the implemented *Entropy* Function

2. Number of bits per symbol required to construct a fixed length code and its efficiency



```
>> wordlength=ceil(log(length(chars))/log(2))

wordlength =

     6
```

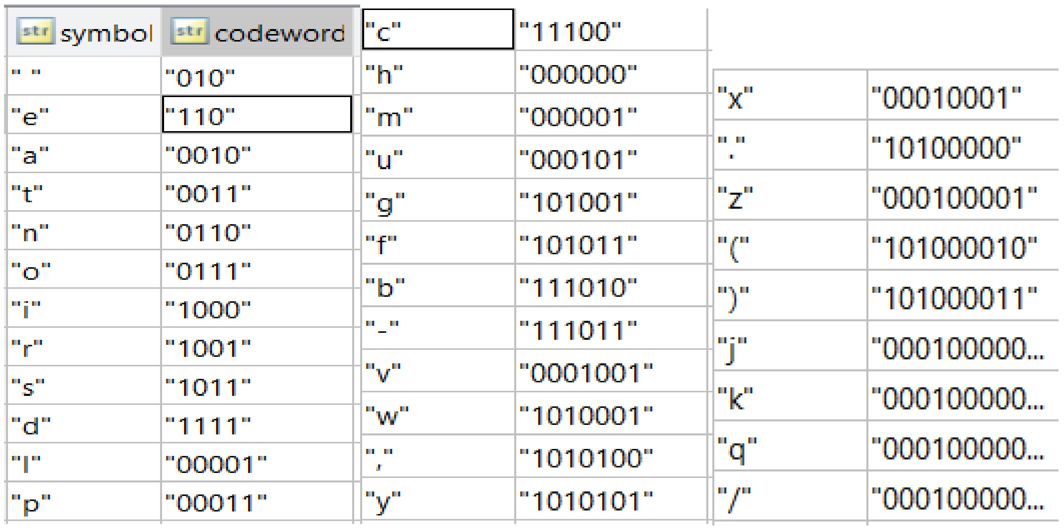
```
>> averagelength=sum(prob*wordlength);
efficiency=entropy/averagelength

efficiency =

    0.7095
```

Fig (4) Average length and efficiency of the fixed length code

3. Output of the Huffman Dictionary Function (*HuffDict)*



symbol	codeword
" "	"010"
"e"	"110"
"a"	"0010"
"t"	"0011"
"n"	"0110"
"o"	"0111"
"i"	"1000"
"r"	"1001"
"s"	"1011"
"d"	"1111"
"l"	"00001"
"p"	"00011"
"c"	"11100"
"h"	"000000"
"m"	"000001"
"u"	"000101"
"g"	"101001"
"f"	"101011"
"b"	"111010"
"-"	"111011"
"v"	"0001001"
"w"	"1010001"
","	"1010100"
"y"	"1010101"
"x"	"00010001"
"."	"10100000"
"z"	"000100001"
"("	"101000010"
")"	"101000011"
"j"	"000100000...
"k"	"000100000...
"q"	"000100000...
"/"	"000100000...

Fig (5) Huffman dictionary for the test text file

4. Tree Visualization using the implemented *HuffTree* function

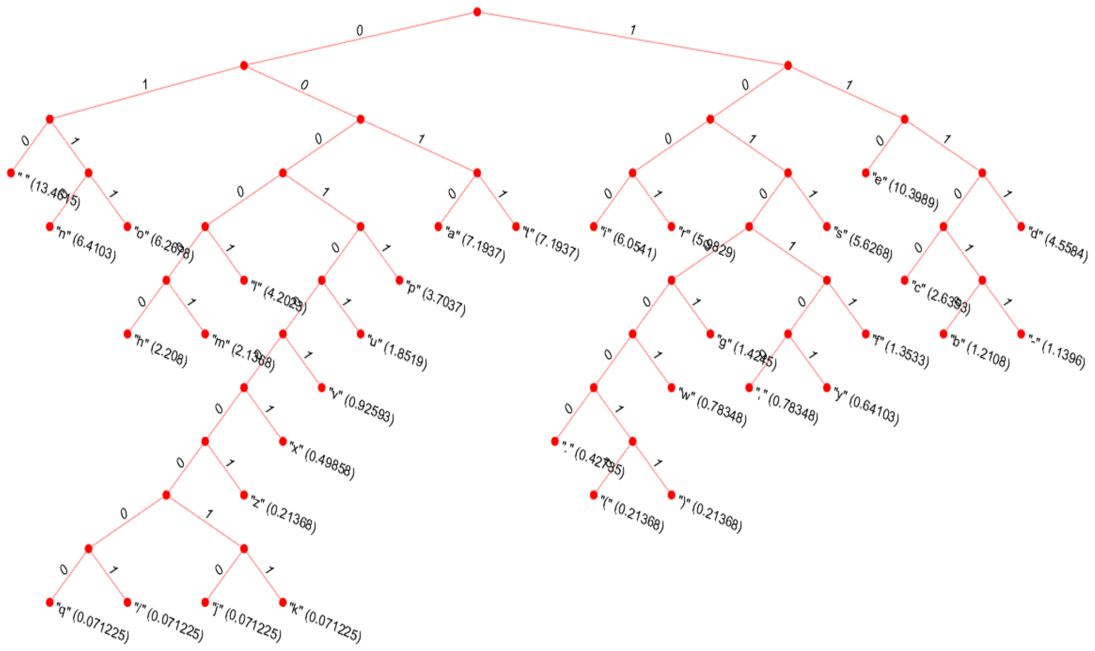
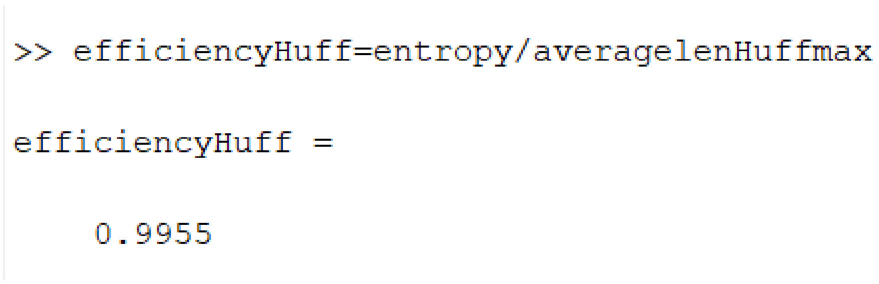


Fig (6) Huffman tree of the Huffman code in Fig (5)

5. You can check the output of the Huffman Decoder Function on passing to the encoded stream by checking the attached text file *decodedHuff.txt*

6.



```
>> efficiencyHuff=entropy/averagelenHuffmax

efficiencyHuff =

    0.9955
```

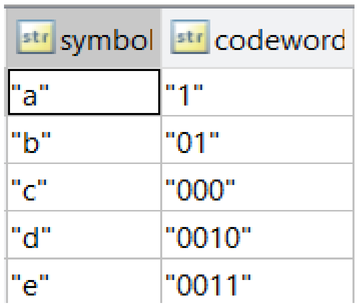
Fig (7) Efficiency of the constructed Huffman code

As you can see the efficiency of the Huffman code is much higher than the fixed length code.

7. Variance of the code [2]

In order to address the practical significance of the variance of the code, assume a discrete source whose alphabet is $\zeta = \{a, b, c, d, e\}$ and probability distribution $p(\zeta) = [0.4, 0.2, 0.2, 0.1, 0.1]$

Case 1: High Variance



symbol	codeword
"a"	"1"
"b"	"01"
"c"	"000"
"d"	"0010"
"e"	"0011"

Fig (8) Huffman dictionary of the source in high variance case

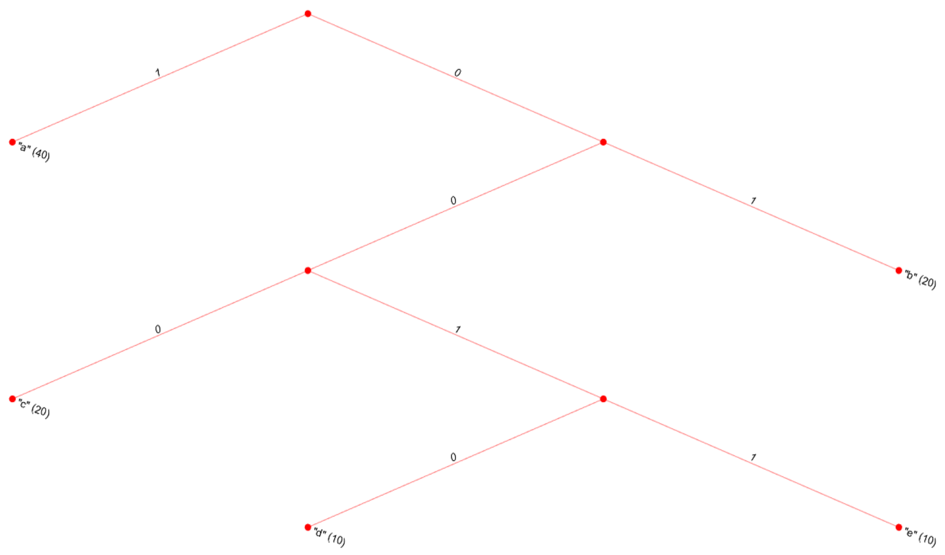
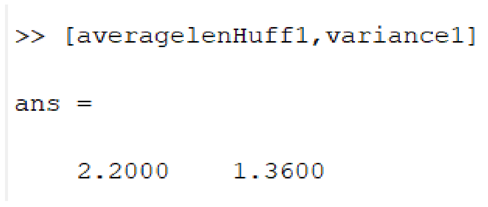


Fig (9) Huffman tree of the code in Fig (8)



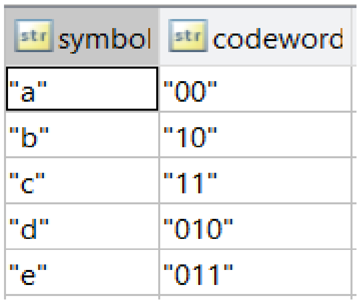
```
>> [averagelenHuff1,variance1]

ans =

    2.2000    1.3600
```

Fig (10) Average length and variance of the code in Fig (8)

Case 2: Low Variance



symbol	codeword
"a"	"00"
"b"	"10"
"c"	"11"
"d"	"010"
"e"	"011"

Fig (11) Huffman dictionary of the source in low variance case

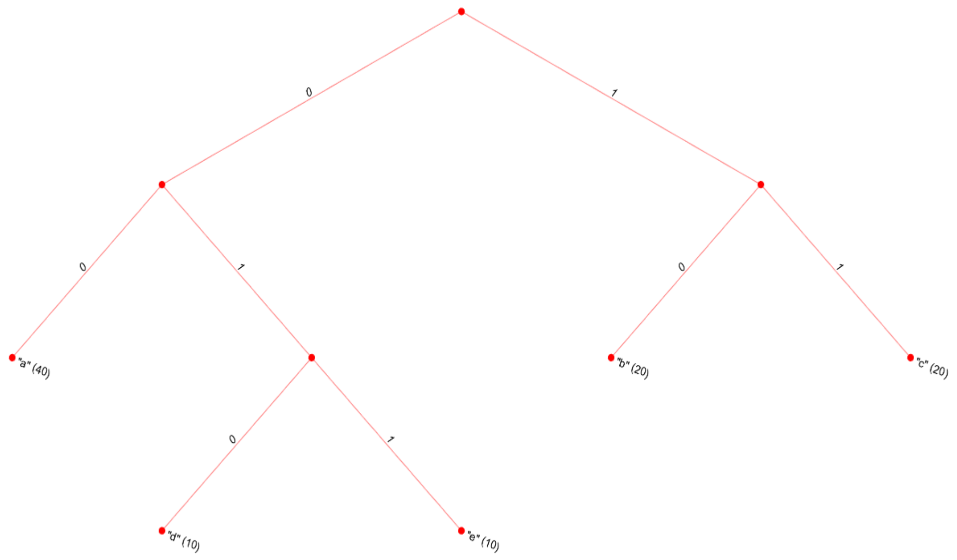
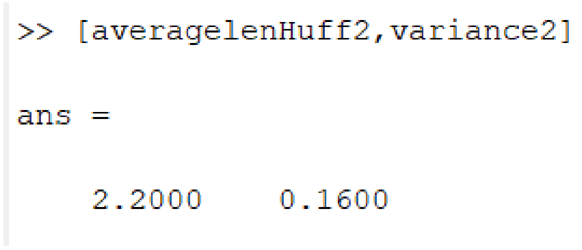


Fig (12) Huffman tree of the code in Fig (11)



```
>> [averagelenHuff2,variance2]

ans =

    2.2000    0.1600
```

Fig (13) Average length and variance of the code in Fig (11)

By comparing Fig (13) with Fig (10), we note that both codes have the same average length but the second code has lower variance. In our case of writing to a text file, the variance is irrelevant. However, if the data is going to be transmitted through a communication channel, a code with lower variance is better. As, the buffer that is used to read the data can be smaller in size which is cost-efficient.

Note: In some cases, the two implementations will have the same variance. However, using the low variance implementation will always have a variance lower than or equal to the variance of the high variance implementation.

Conclusion

Huffman Code efficiency is more than the efficiency of the fixed length code or equal to it at its worst case. The ambiguities that arise while performing the Huffman algorithm result in arbitrary choices which preserve the average codeword length, however; in some cases, they can change the variance of the code which can be practically significant

References

[1] D. Salomon, *A concise introduction to data compression*. New Delhi: Springer, 2011.

[2] R. Gallager, "Variations on a theme by Huffman," *IEEE Transactions on Information Theory*, vol. 24, no. 6, pp. 668–674, 1978.

[3] S. Haykin, *Communication systems*. London: John Wiley, 2001.