Lossless Coding

ELE 725 Lab 2

Dylan Pereira
500512067
Ryerson University
Toronto, Canada
dylan.pereira@ryerson.ca

Nishantan Jegarajah
500519391
Ryerson University
Toronto, Canada
king.rajah.18@gmail.com

Abstract— **In this lab, we present our result on lossless coding in the form of Entropy and Huffman Encoder. This process was done using C++ programming language where we utilized a vector arrays to generate values according to the given input message sequence "Huffman is the best compression algorithm." The entropy was then calculated by generating the frequency of symbols occurring with the message. The Entropy values generated are then used to create a Huffman binary tree that outlines the prefix codes for each symbol in the source. These prefix codes are then used to encode the source message into a bitstream.**

Keywords- Huffman Encoding, Entropy encoding, lossless encoding, binary tree, compression, statistical and structural redundancy.

I. Introduction/Theory

In this lab we studied the entropy-based methods and variable length coding (VLC) of lossless coding, specifically focusing on implementing a Huffman encoder. The purpose of this lab is to develop a greater understanding of the methods applied and how the Huffman process is utilized in lossless coding. We required some knowledge regarding media compression, encoding and decoding. The goal of compression is to reduce the amount of data used to convey the information. Media compression works by inputting an information source as data into an encoder for compression, the compressed data is then saved onto a storage media or network. When the encoded information is to be recovered, it is passed through a decoder where it will decompress the data and will output the media information.

There are two types of compression methods; 'lossless' and 'lossy.' In the lossless method, media is compressed and the recovered data has not lost any information in the process. The lossy method uses a technique where the recovered data is generated by approximating the input data, thus causing some original information to be lost in the process. Lossless compression is applied by looking for redundancies in information; there are many ways to exploit redundancies in information. Statistical redundancy relies on redundancy in terms of occurrences of symbols independently of how they are ordered, whereas structural redundancy looks for any redundancy in how the information is ordered and how it can be exploited.

II. Theory

A. Histogram

A sequence or message is a set of symbols of a given alphabet, audio samples, or images and video pixels. It is necessary to know how to calculate the probability/frequency of a particular symbol. The probability of a symbol is the number of occurrences of particular symbol divided by the total number of symbols in the sequence/message.

Generating a histogram using the data of probability of the symbols, allows us to know the probability distribution of the symbols present in the given signal. Using this histogram information, we can then apply variable length coding to allocate fewer bits to more frequently occurring symbols and more bits to less frequently occurring symbols, this will reduce the total size of the sequence or message. The number of bits needed to encode a media source is lower-bounded by its "Entropy".

B. Entropy

The entropy of a source H(S) is defined as the average number of bits per symbol required as a minimum for encoding it. Equation (1) is the formula for entropy, where pi represents the probability of a symbol occurring within the source, and the source has symbols {s1, s2… sn} that holds all symbols. The value of i, from 1 to the value n (the total number of symbols in the source), indicates the symbols of the source in order. The entropy is dependent on the values generated by the histogram, which includes the probability and self-information.

$$H(S) = \sum_{i}^{n} p_i \log_2\left(\frac{1}{p_i}\right) = -\sum_{i}^{n} p_i \log_2(p_i) \qquad (1)$$

C. Huffman Encoder

The Huffman encoding is a method of encoding a set of data so that every entry is assigned a unique binary code. A unique binary code is one where the beginning of each binary sequence does not appear in any other encoded binary sequence

Identify applicable sponsor/s here. *(sponsors)* in the data set. This is called a prefix code because the prefix of each code is unique; this means the correct symbol is identified every time you reading a sequence of bits.

Huffman encoder uses the bottom-up method for building the encoding tree, whereas the Shannon-Fano uses the top-down method. To build a Huffman tree you must first construct a tree of every unique data entry in the data set along with their frequency of occurrence. Symbols are sorted according to frequency/probability in ascending order. Then join the two smallest nodes under a new parent node. The new parent node should have frequency/probability equal to the sum of the two children nodes. The process is repeated to continue adding the two lowest frequency nodes until only one node is left which is called the root node. The branches are labeled 0 and 1 starting from the root to the intermediary nodes. Then traverse the binary tree from root to each leaf to build the associated code for each symbol.

III. Methodology

This experiment was conducted by building the encoding systems as a series of individual functional blocks in C++. There are four major blocks that are included in the program. The first stage takes the input message and captures the number of occurrences of each unique symbol. The second stage builds the Huffman tree using the information from the first stage. The third stage uses the prefix code dictionary built in the second stage to encode the input message into a bit stream. This experiment uses a character array as an input message. The type of message does not affect the construction of the algorithm, since integer or double messages can be casted into chars to be used in the algorithm. Since Huffman encoding is used in JPEG compression, image data is very frequently used as input. In order to accommodate image data, we would only need to separate the color channels and extract the integer pixel values (0 to 255) and convert them to chars. This process is not included in this experiment because it does not affect the algorithm structure.

Using C++ allows us to use vectors to represent data. Vectors are C++ built in structures that resemble dynamically growing arrays. Vectors can be treated like stacks for push and pop operations or can be accessed at a particular index using the '.at()' function. Our algorithm therefore works for arbitrary sized input strings. The first stage involves a nested for loop to generate the input vectors that represent the actual character and its recurrence value. The recurrence value vector is indexed at the same position as its corresponding character in its own vector for the entire algorithm. Recurrence values are then sorted from the smallest to the largest which also organizes the characters. At this point we have two sorted vectors representing the input data histogram.

In the second stage we build the Huffman tree. Since we are using vectors, we do not build the tree with arrays. Rather, we use 5 unique vectors to represent that array. Provided that symmetrically perform operations to each vector, we can treat the set as related by index. To build the tree we use the functions findLow() and addRow(). The former finds the lowest probability character from the input message. While the latter adds this character and its corresponding value to the binary tree vector set. New child and parent locations are updated at the appropriate indicies. We consistently call these two functions until parentTest() function returns true. The parentTest() is used to determine if there is only one term left with a parent. This term signifies the top of the Huffman tree which has no parents of its own; since we are building from bottom up, this is the terminal value.

We then loop through the binary tree vector set to generate two new vectors to represent the dictionary or encode table. In the third stage we use this table to encode the message by doing simple char matching and extracting the binary prefix in the same index as the matched char. For each letter of the input message, we push a prefix into a bits representation vector. Two functions are used to complete this last process: findBigParent() and leafTest(). The former is called recursively to traverse the binary tree from the letter up to the root node. Thus we reverse the order of the bits so that the bit stream is seen as read from top to bottom. The leafTest() function is what permits the call of findBigParent(), since its job is to test whether the start point is a leaf i.e. a comparable character.

IV. Results

Fig 1 shows the input message where all repetitions are removed in the second line: 'hufman istebcoprlg.' The occurrences of each of these letters is in line three. Line six shows us the repetition count of the characters in line seven as organized in increasing frequency. The line 'Val size is 18' tells us there are 18 unique characters in the input message. The binary tree is then shown underneath when it has been first initialized. All the values are set to -1 to represent that they are eligible for manipulation.

Fig 2 show the final binary tree vector set where each vector is seen column wise. This is set is larger than the initialized ones since we have added new parents to the set. The vectors in order from left to right are: index, (1) sum weighting, (2) parent, (3) left child and (4) right child. The fifth term is the character when applicable.

Fig 3 shows the result of the tree traversal process used to build the dictionary. The binary values on the left represent the prefix codes for each of the characters at the index given in the middle column for the corresponding letter given in the right column.

Fig 4 shows the that the bit stream generation can correctly select the bits representation for each character in the input message. The total space required to hold the message is a sum of all the bits: 4*30 + 5*6 + 3*5 = 165 bits. If the data was uncompressed, we would require 5 bits per symbol to contain 18 unique characters. This results in an uncompressed size of 41*5 = 205 bits. Using these values, we get the resulting compression ratio of 1.24.

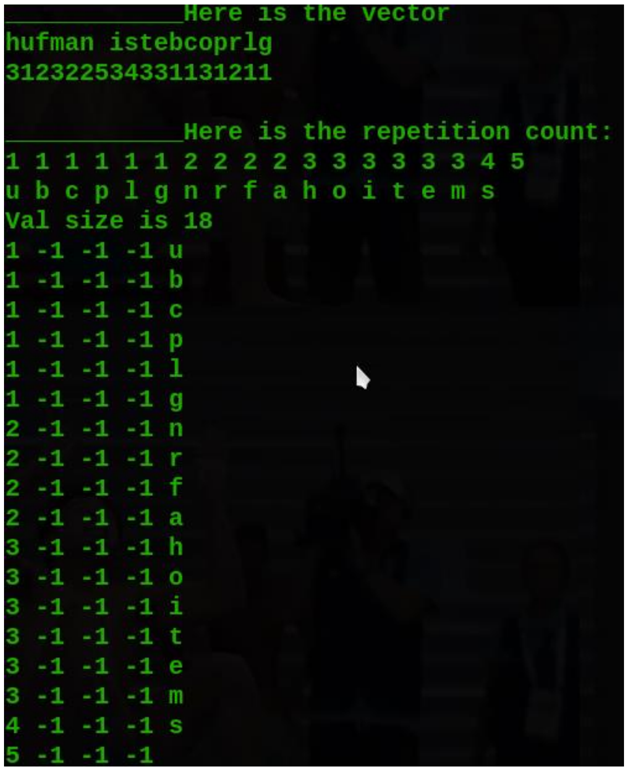
V. Discussion

Since we are only concerned with the functionality of this program we can confirm that the results of this process are able to encode the input sequence correctly. In the first part, we order unique characters by increasing frequency and we can verify the results from Fig 1 using manually by hand. In figure 2 we see the completed binary tree from the multi-function process. The new tree is 35 rows in size while the original initialized tree was 18 rows in size. Since the final tree must be 2N in size, where in is the number of unique characters, we can confirm the tree size to be correct. Furthermore, we have used online Huffman binary tree builders to verify the correctness of our tree based on the input string we have gotten. We also observe that the final tree has only one parent term that contains negative one. This condition is used as a trigger to end the tree building process and verifies that the tree is properly sequenced.

In figure 3 we see that the least frequency characters at the top are given the most number of bits. For example, ‘which occurs once is prefixed by ‘00010’ while the space character which occurs five times is prefixed by ‘001.’ As we increase in frequency we see the prefix codes are of size 5 bits, 4 bits and finally 3 bits.

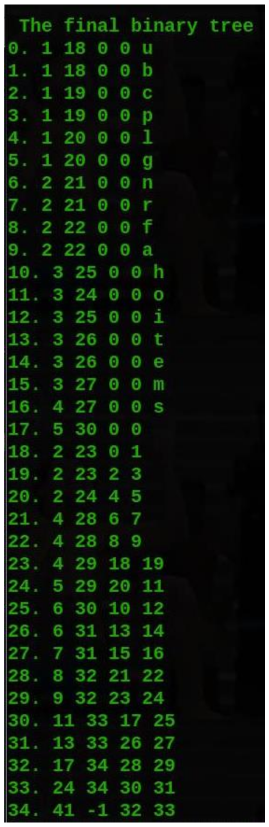
The compression ration that we determined in the results from fig 4 of 1.24 assume that only 18 symbols will ever be used. If the number of unique symbols increases, the compression ratio increases. If the data set size increases, the value of the compression does also. Using this small data set we see a substantial, near 25% reduction in message size without losing any input data.

A. Figures and Tables



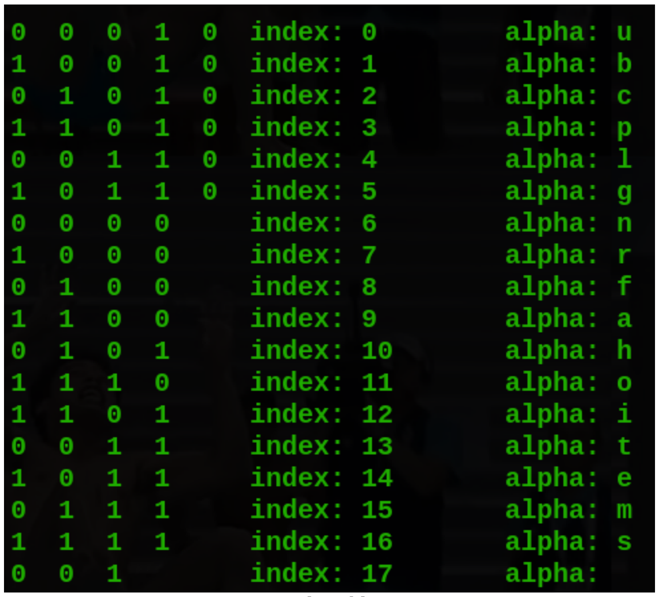
```
____________Here is the vector
hufman istebcoprlg
312322534331131211

____________Here is the repetition count:
1 1 1 1 1 1 2 2 2 2 3 3 3 3 3 3 4 5
u b c p l g n r f a h o i t e m s
Val size is 18
1 -1 -1 -1 u
1 -1 -1 -1 b
1 -1 -1 -1 c
1 -1 -1 -1 p
1 -1 -1 -1 l
1 -1 -1 -1 g
2 -1 -1 -1 n
2 -1 -1 -1 r
2 -1 -1 -1 f
2 -1 -1 -1 a
3 -1 -1 -1 h
3 -1 -1 -1 o
3 -1 -1 -1 i
3 -1 -1 -1 t
3 -1 -1 -1 e
3 -1 -1 -1 m
4 -1 -1 -1 s
5 -1 -1 -1
```

Figure 1. Histogram result and tree initialization



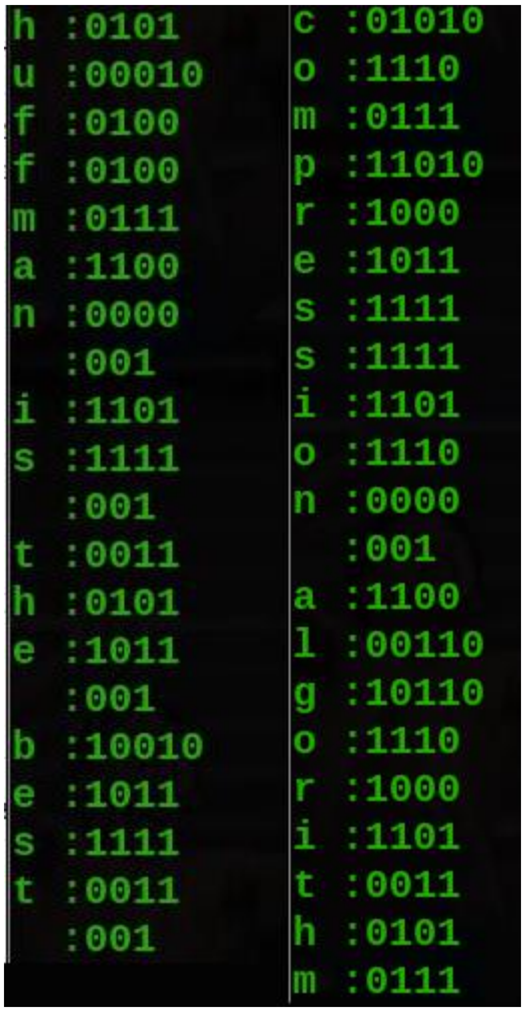
```
 The final binary tree
0. 1 18 0 0 u
1. 1 18 0 0 b
2. 1 19 0 0 c
3. 1 19 0 0 p
4. 1 20 0 0 l
5. 1 20 0 0 g
6. 2 21 0 0 n
7. 2 21 0 0 r
8. 2 22 0 0 f
9. 2 22 0 0 a
10. 3 25 0 0 h
11. 3 24 0 0 o
12. 3 25 0 0 i
13. 3 26 0 0 t
14. 3 26 0 0 e
15. 3 27 0 0 m
16. 4 27 0 0 s
17. 5 30 0 0
18. 2 23 0 1
19. 2 23 2 3
20. 2 24 4 5
21. 4 28 6 7
22. 4 28 8 9
23. 4 29 18 19
24. 5 29 20 11
25. 6 30 10 12
26. 6 31 13 14
27. 7 31 15 16
28. 8 32 21 22
29. 9 32 23 24
30. 11 33 17 25
31. 13 33 26 27
32. 17 34 28 29
33. 24 34 30 31
34. 41 -1 32 33
```

Figure 2. Binary Tree Vector Set



```
0  0  0  1  0  index: 0     alpha: u
1  0  0  1  0  index: 1     alpha: b
0  1  0  1  0  index: 2     alpha: c
1  1  0  1  0  index: 3     alpha: p
0  0  1  1  0  index: 4     alpha: l
1  0  1  1  0  index: 5     alpha: g
0  0  0  0     index: 6     alpha: n
1  0  0  0     index: 7     alpha: r
0  1  0  0     index: 8     alpha: f
1  1  0  0     index: 9     alpha: a
0  1  0  1     index: 10    alpha: h
1  1  1  0     index: 11    alpha: o
1  1  0  1     index: 12    alpha: i
0  0  1  1     index: 13    alpha: t
1  0  1  1     index: 14    alpha: e
0  1  1  1     index: 15    alpha: m
1  1  1  1     index: 16    alpha: s
0  0  1        index: 17    alpha:
```

Figure 3. Encode Table / Dictionary



```
h :0101      c :01010
u :00010     o :1110
f :0100      m :0111
f :0100      p :11010
m :0111      r :1000
a :1100      e :1011
n :0000      s :1111
  :001       s :1111
i :1101      i :1101
s :1111      o :1110
  :001       n :0000
t :0011        :001
h :0101      a :1100
e :1011      l :00110
  :001       g :10110
b :10010     o :1110
e :1011      r :1000
s :1111      i :1101
t :0011      t :0011
  :001       h :0101
             m :0111
```

Figure 4. Bit stream / Bits representation

VI. Conclusion

We have successfully been able to compress the input data losslessly using the Huffman binary tree approach. Each step was verified either manual by hand or by using online encoder tools. The most complex portions of this encoding system involve the traversal and construction of the binary tree, which requires multiple indices to be tracked and simultaneously updated with new inputs. Even with a small message size, we see a significant improvement in the amount of storage space saved.

References

[1] Khan, N. (n.d.). Basics of Multimedia Systems. Retrieved March 10, 2017. ELE725_L04.1_quantizationandaudio Lecture Slides

[2] Khan, N. (n.d.). Basics of Multimedia Systems. Retrieved March 10, 2017. ELE725_L04.2_quantizationandaudio Lecture Slides

[3] Khan, N. (n.d.). ELE725 Lab Manual. Retrieved March 10, 2017.