Data Compression Algorithms

To: MISS FARIBA GHASEMINAJM

Prateek Asthana Khanjan Kamlesh Dabhi

2020

Project Objective

In this project we will look at improvement of data compression ratios using LZW and Huffman encoding/decoding. We will study the different data compression techniques and the implementation of their encodings Measure the performance of Huffman and LZW by measuring various size of data. Then measure relative improved performance in the case of hybrid compression for both Huffman based LZW.

Lossless Data Compression

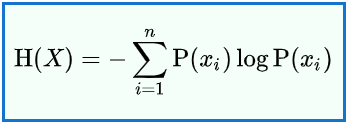
Huffman coding and LZW is used for lossless data compression. In lossless data compression the main idea is to compression and decompress the data without loosing any data. Comparing to lossy compression which only recovery a certain percentage of the original data. Lossy compression rates have high compression rates.

Lossless data compression is used in different applications than compared to lossy data compression. Zip files are a great example where lossless data compression is used. You could go and say we are creating something exactly like it. Though with different algorithms. Some other places where lossless compression is used is PNG and gifs, while other image formats use lossy compression data. Text documents or executable programs also use lossless compression data.

Lossless compression techniques follow a two-step program. First step they generate a statistical mode for the input data and than they map the data to the model. The models bit sequencing results in less data or compression.

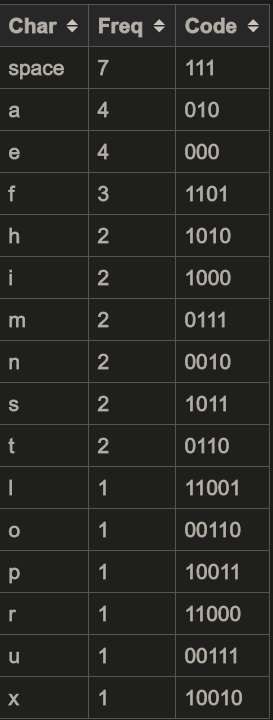
Huffman Coding

Huffman Coding is one of the algorithms that uses this two-step approach. This technique is also called arithmetic coding. In arithmetic coding you can achieve compression rates depending on your statistical model and that information is called information entropy. Entropy is dependent on the probability of the information. When the probability = 0 or 1 we can calculate entropy as zero bits. Entropy range is always between 0 and 1.

Given a discrete random variable X, with possible outcomes x1,…..,xn, which occur with probability P(x1),…..,P(xn), the entropy of x is formally defined as:

[1]

where Sigma denotes the sum over the variable's possible values log is the logarithm, the choice of base varying between different applications

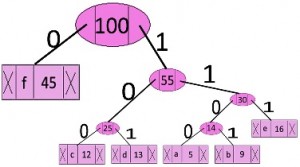
Moving back to the idea of arithmetic coding we can now understand that it is form of entropy encoding (independent of the specific characteristics of the medium). An example string “hello there” is 11 fixed bits, but if we convert the string using arithmetic encoding, we can store less bits which are not frequently occurring resulting in compression. This is different than Huffman encoding. Where you sperate the symbols and replace each with a code. This is called variable length code (maps source symbols to a variable number of bits). An example table for this is shown below:

This table is generated by calculating probability of frequency of occurrence for each value of the symbol. Common symbols are represented by fewer bits. Huffman coding is not optimal for all compression methods. There are other compression methods that can be used for better compression ratios such as asymmetric numerical systems. From the table we can see we are assigning char characters to integers. We coded the project in java. Because of that we can see the integer data types and their size.

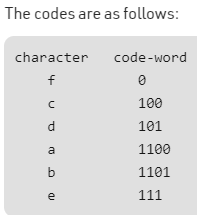
[5]

1 byte = 8 bits. We can see from the table above that char only takes 1 byte and int takes 4 bytes of space. Since we are discussing space, we can conclude that Huffman coding is less efficient when we are compressing text files. Thus, when compressing text files like we are in this project would yield lower efficiency than an algorithm where you can map to char data types.

Here is a visual example of the Huffman coding [2]



In the above picture each symbol is sorted from most frequent at the top to least frequent at the bottom shown by probability box beside the symbol. To achieve code for each symbol you must go from top to bottom.

[6]

Limpel Ziv Welch (LZW)

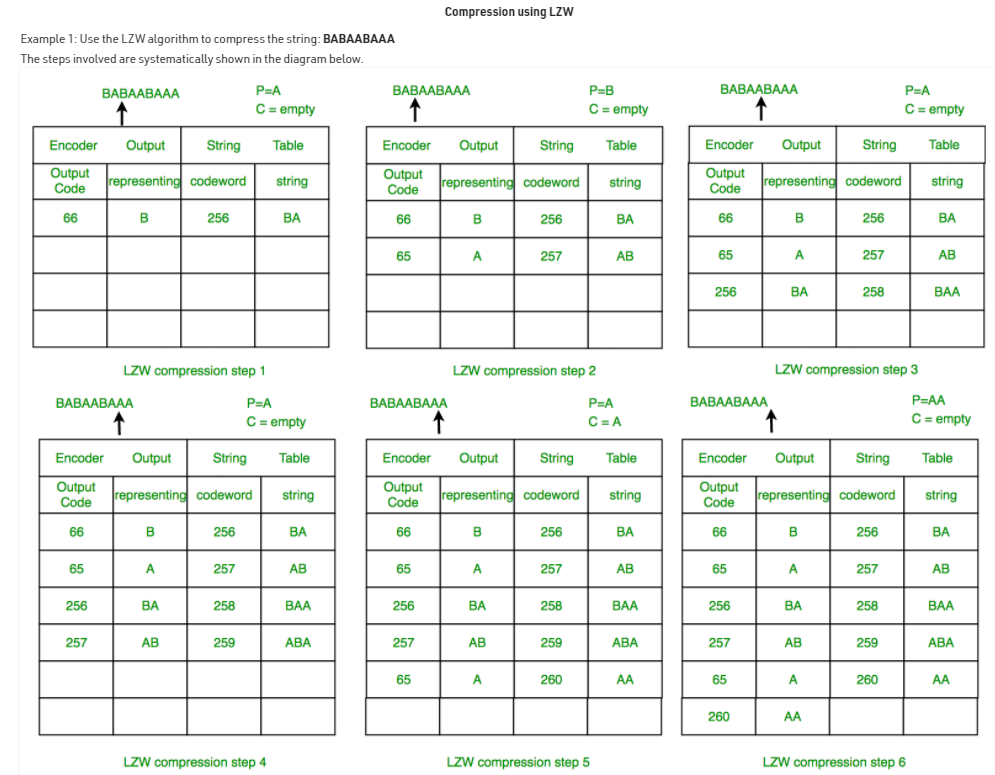
LZW is a simple algorithm with very high throughput in hardware implementations. It is a lossless compression algorithm. It is most used in unix file compression and is used in gif image format. The base idea of LZW is to rely on reoccurring patterns to save data space. This is also part of many PC utilities that claim to double your hard drive capacity.

LZW uses code tables and 4096 is a common choice for the table size. Codes are usually between 0-255 in the code table. This is mainly because they assume if you use 1 byte as a data memory so 2^8 would be 256.When the encoding begins it only holds values between 256-4095. The first 256 are already used.

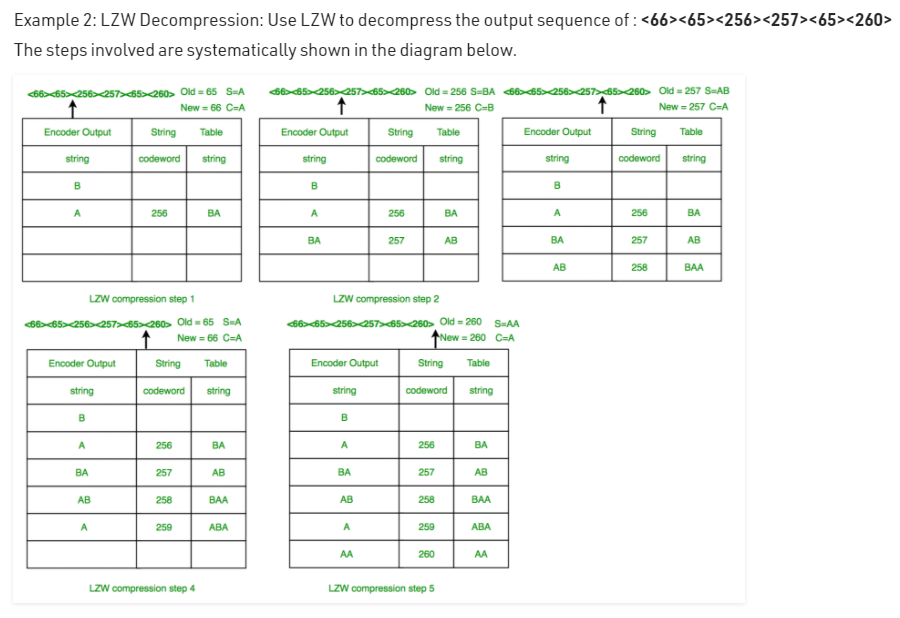
When the algorithm encounters first 8bit sequence it stores it in its dictionary. As the algorithm progresses through each unknown sequence it adds it to the dictionary, once it finds a sequence that already exists in the dictionary it will record the sequence until next unknown sequence appears. The known information is replaced with a code table pointer to the sequence in dictionary. The final code. This idea was adapted for other situations as well for example images where 12bit sequence was used as the starting point. [3] Some ideas such as variable-width for the starting sequence and reserving code to clear the idea to stop. But inputting this as part of basic would increase the encoding and decoding length of the binary code and increase the time required for the files to compress.

A visual example of limpel ziv welch:

Encoding [4]:

In step 6 we can see a full code table that is generated from encoding. Codeword is representing the position of the algorithm. String is representing the sequence that is identified. Output code is the code that calls the alphabet from the representing table.

Decoding [4]:



If you look at the labels on the second line and follow the table. Codeword is representing the position of the algorithm and string (far right) is representing the sequence identified, and string (far left) is representing the decoder output.

Methodology

We used agile methodology for this project. We followed four principals of agile methodology [8]. Individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, responding to change over following a plan.

It was crucial for us to have a working software otherwise, we would not be able to present our results and if we don’t have results, we cannot write comprehensive documentation. This is a class assignment and we do not the option of negotiation. Both of us have a lot of other courses and are in our final year. We believed it was best that we don’t hard define a plan as it creates unnecessary pressure for us. Because of the reasons listed above we felt agile methodology would be best approach for the software.

Design

We decided to code this in JAVA as it is the language, we are most familiar with. We used object-oriented approach towards coding this project. We coded both algorithms in separate files and used a main file to execute the hybrid Huffman-LZW sequence. We have attached the coded zip files. The pseudocode for following through each algorithms encoding and decoding part is shown below.

Huffman Encoding [7]

Procedure Huffman(C): // C is the set of n characters and related information

n = C.size

Q = priority\_queue()

for i = 1 to n

n = node(C[i])

Q.push(n)

end for

while Q.size() is not equal to 1

Z = new node()

Z.left = x = Q.pop

Z.right = y = Q.pop

Z.frequency = x.frequency + y.frequency

Q.push(Z)

end while

Return Q

Huffman Decoding [7]

n := S.length // S refers to bit-stream to be decompressed

for i := 1 to n

current = root

while current.left != NULL and current.right != NULL

if S[i] is equal to '0'

current := current.left

else

current := current.right

endif

i := i+1

endwhile

print current.symbol

endfor

LZW Encoding [4]

\* PSEUDOCODE

1 Initialize table with single character strings

2 P = first input character

3 WHILE not end of input stream

4 C = next input character

5 IF P + C is in the string table

6 P = P + C

7 ELSE

8   output the code for P

9 add P + C to the string table

10 P = C

11 END WHILE

12 output code for P

LZW Decoding [4]

\* PSEUDOCODE

1 Initialize table with single character strings

2 OLD = first input code

3 output translation of OLD

4 WHILE not end of input stream

5 NEW = next input code

6  IF NEW is not in the string table

7 S = translation of OLD

8   S = S + C

9 ELSE

10  S = translation of NEW

11 output S

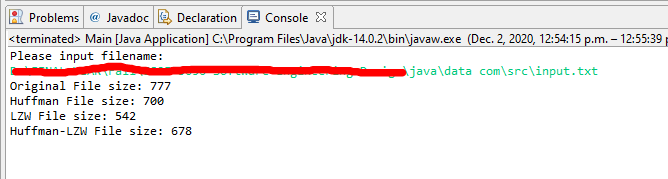
12   C = first character of S

13   OLD + C to the string table

14 OLD = NEW

15 END WHILE

Results:



The original file size is 777bytes (100%), after Huffman coding the file size decreased to 700 bytes (90%), performing only LZW compression the file size decreased to 542 bytes (69.75%). Upon performing Huffman coding than LZW in a sequence the file size decreased to 678 bytes (87.25%).

The results raised question regarding third iteration of as to why is performing Huffman-LZW resulted in larger file than simply just performing LZW. Upon investigating the binary instruction files, we could see that binary file size for Huffman-LZW is larger than LZW binary file. So, we believe when we perform double compression to a file it is less efficient than simply performing one highly efficient algorithm for compression. Simply because we are creating a lot of instructions to code an encoded file which is in binary.

References

1. <https://en.wikipedia.org/wiki/Huffman_coding>
2. <https://www.researchgate.net/profile/CHOUAKRI_Sid_Ahmed/publication/259007386/figure/fig4/AS:668221363453957@1536327796270/An-example-of-Huffman-coding-45-Linear-Prediction-The-applications-of-the-linear.png>
3. <https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch>
4. <https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/>
5. <https://docs.oracle.com/cd/E19253-01/817-6223/chp-typeopexpr-2/index.html>
6. <https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/>
7. <https://riptutorial.com/algorithm/example/23995/huffman-coding>
8. https://www.infoworld.com/article/3237508/what-is-agile-methodology-modern-software-development-explained.html