Lossless compression

Enis Mulić, University of Mostar

***Abstract* - Compression is a technique that reduces the number of bits needed to represent data. Compression helps in reducing the consumption of resources such as disk space and bandwidth. Compression is build into a broad range of technologies like storage systems, databases operating systems. However there are tradeoffs between the quantity of compression, the runtime, and the quality of the reconstruction.**

***Index terms* - data compression, lossless data compression, information entropy, Huffman coding, probability coding, arithmetic coding, Lempel-Ziv algorithms, LZ77, LZ78**

I. INTRODUCTION

Lossless compression refers to a type of compression where compressing and decompressing data will result in an exact replica of the original. The process of compression consists of two components, an encoding algorithm that generates a compressed representation of a given message, and a decoding algorithm that reconstructs the original message of an approximation of it from the compressed representation. [2]

Do to it being impossible to compress everything, all compression algorithms must assume that some input messages are more likely then others - i.e. an assumption that some characters have a higher chance of repeating than others.[2]

All compression algorithms have two components, the model, which captures the probability distribution of the input message, and the coder, which uses the probability distribution from the model to generate codes by lengthening messages with a low probability and shortening messages with a higher probability. [2]

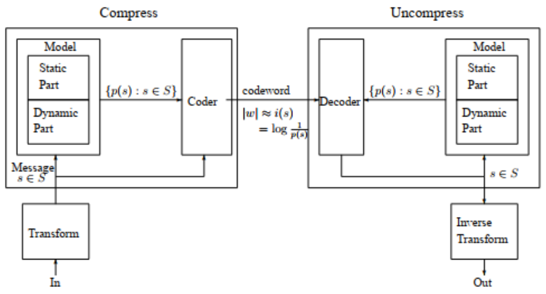


Figure 1 General framework of a model and coder

II. ENTROPY

Entropy represents the randomness or disorder of a system.

It is assumed that a system has a set of possible messages, and at a given time there is a probability distribution over those messages.[2]

Entropy is then defined as:

$$H(S) = \sum_{s \in S} p(s) \log_2 \frac{1}{p(s)}$$

where S is the set of possible messages, and p(s) is the probability of message $s \in S$. This indicates that the more even the probabilities the higher the entropy and the more biased the probabilities the lower the entropy. [2]

A concept closely related to entropy is the self information of a message, defined as

$$i(s) = \log_2 \frac{1}{p(s)}$$

Self information represents the number of bits of information contained in a message and the number of bits we should use to encode it. [2]

The definition of self information indicates that messages with higher probability contain less information. Which means that entropy is a probability weighted average of the self information of each message. [2]

III.Probability Coding

Typically probabilities are used for parts of messages rather then for the complete message, for example each character or word in a text. Each message component can be of a different type and come from its own probability distribution. [2]

There are two types of algorithms, those that assign a unique code for each message (Huffman codes) and those that "blend" the codes together from more than one message in a row (arithmetic codes). [2]

a. Prefix Codes

A code C for a message set S is a mapping from each message to a bit string. Each bit string is called a codeword. Codes are denoted using the syntax $C = \{(s_1, w_1), (s_2, w_2), \ldots, (s_n, w_n)\}$. As codewords have variable length this causes potential problems. If codewords are sent one after the other it could be hard or impossible to determine their beginning and ending. For example given the code $C = \{(a, 1), (b, 01), (c, 101), (d, 011)\}$, the sequence 1011 can be red as either *aba*, *ca*, or *ad*. In order to avoid this a special stop symbol can be added to the end of each codeword but this would result in more data needing to be sent. A more efficient solution is designing codewords which can always be uniquely deciphered. [2]

A prefix code is a special kind of uniquely decodable code in which no bit string is a prefix of another one, example $C = \{(a, 1), (b, 01), (c, 000), (d, 001)\}$. [2]

Prefix codes have an advantage over other uniquely decodable codes in that each message can be deciphered without having to see the start of the next message. This is important when sending messages from different probability distributions. [2]

A prefix code can be viewed as a binary tree:

- Each message is a leaf in the tree
- The code for each message is given by following a path from the root to the leaf and appending a 0 for each time a left branch is taken, and a 1 for each time a right branch is taken

This prefix-code tree can be also used to decode prefix codes. As the bits come in, the decoder can follow a path from the root unit it reaches a leaf and outputs the decoded message. The decoder then returns to the root for the next bit and repeats the process. [2]

IV. Huffman Codes

Huffman codes are optimal prefix codes generated from a probability distribution by the Huffman Coding Algorithm.[2]

The Huffman algorithm generates a prefix-code tree by starting with a forest of trees, one for each message. Each message contains a single vertex with the weight $w_i = p_i$ (Where p_i is the probability).[2]

Select two trees with the lowest weight roots (w_1 and w_2) and combine them into a single tree by adding a new root with weight $w_1 + w_2$, and make the trees the children. Repeat the aforementioned step until only a single tree remains. [2]

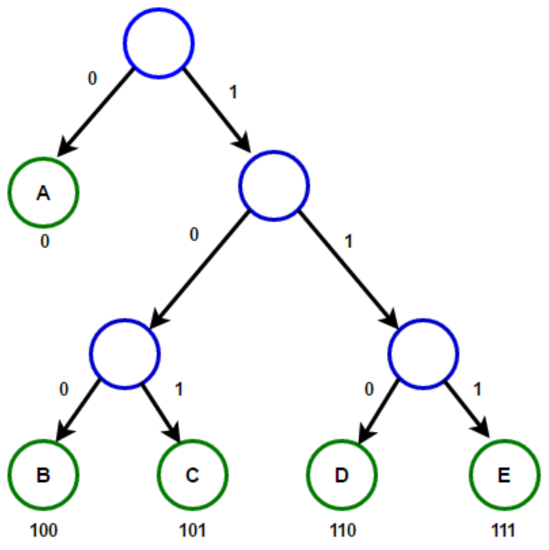


Figure 2 General framework of a model and coder [2]

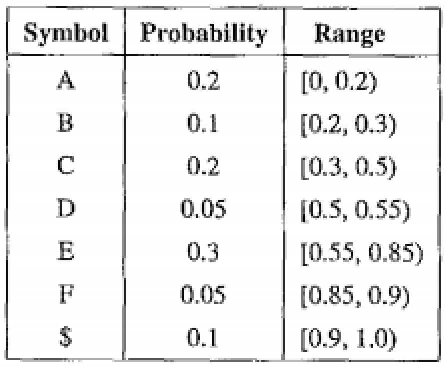
There are two families of Huffman encoding: Static Huffman algorithms and adaptive Huffman algorithms. Static Huffman algorithms calculate the frequency first and then generate the generate a common tree for both the compression and decompression process. Adaptive Huffman algorithms generate two trees, for both processes, while calculating the frequencies.[1]

V. Arithmetic Coding

Arithmetic coding is a technique for coding that allows the information of messages from a message sequence to be combined to share the same bits. Arithmetic codes can achieve better compression, but the transferring of the messages might get delayed because they need to be combined before they can be sent.[2]

In arithmetic coding each possible sequence of n messages is represented by a separate number in an interval between 0 and 1.[2]

The algorithm works recursively by encoding or decoding one symbol per recursion. Because the algorithm works well with smaller intervals, on each recursion, it partitions the interval, and retains on of the partitions as the new interval. In each one of the intervals lies a code string which is viewed as a magnitude. The data string is recovered by using magnitude comparisons on the code string in order to recreate how the encoder partitioned and retained each subinterval. [5]



Symbol	Probability	Range
A	0.2	[0, 0.2)
B	0.1	[0.2, 0.3)
C	0.2	[0.3, 0.5)
D	0.05	[0.5, 0.55)
E	0.3	[0.55, 0.85)
F	0.05	[0.85, 0.9)
$	0.1	[0.9, 1.0)

Figure 3

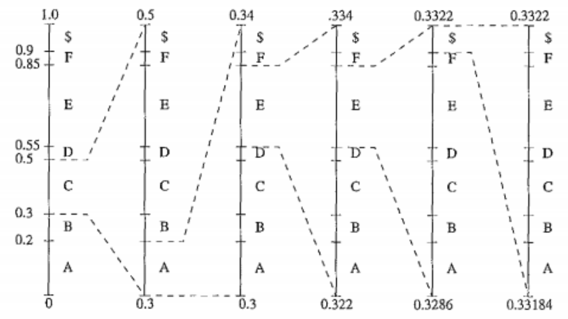
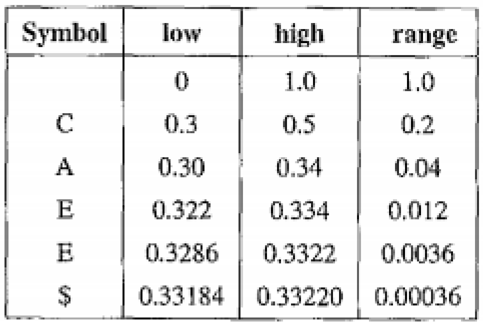


Figure 4



Symbol	low	high	range
	0	1.0	1.0
C	0.3	0.5	0.2
A	0.30	0.34	0.04
E	0.322	0.334	0.012
E	0.3286	0.3322	0.0036
$	0.33184	0.33220	0.00036

Figure 5

Arithmetic coding: encode symbols CAEE$; (Figure. 3) probability distribution of symbols; (Figure. 4) graphical display of shrinking ranges; (Figure. 5) new low, high, and range generated.[4]

VI. The Lempel-Ziv Algorithms

The Lempel-Ziv algorithms compress the message by creating a dictionary of previously seen strings. The Lempel-Ziv algorithms code groups of characters of varying lengths.

The algorithms work by looking through the part of a file before a given starting position and searching for the longest match to the string beginning at that starting position, and outputting some code that refers to the match.[2]

The two main variants of the algorithm were described by Ziv and Lempel in two separate papers in 1977 and 1978, and are referred to as LZ77 and LZ78. The two algorithms are distinct in how far back they search and how they find matches.[2]

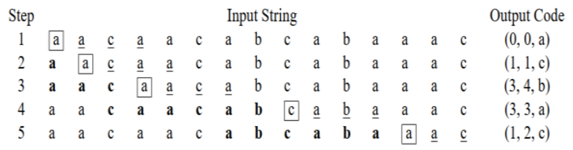
The LZ77 algorithm, based on the idea of a sliding window, looks for matches in a fixed distance back from the current position, while the LZ78 algorithm is based on a more conservative approach to adding strings to the dictionary.[2]

a. LZ77

The LZ77 algorithm uses a sliding window that moves along with the cursor (position). The window can be divided into two parts, the part before the cursor, called the dictionary, and the part starting at the cursor, called the lookahead buffer. The size of these two parts are parameters of the program and are fixed during execution of the algorithm. The basic algorithm loops executing the following steps: [2]

1) Find the longest match of a string starting at the cursor and completely contained in the lookahead buffer to a string starting in the dictionary. [2]

2) Output a triple (p, n, c) containing the position p of the occurrence in the window, the length n of the match and the next character c past the match [2]



Step	Input String															Output Code
1	$\boxed{a}$	a	c	a	a	c	a	b	c	a	b	a	a	a	c	(0, 0, a)
2	**a**	$\boxed{a}$	c	a	a	c	a	b	c	a	b	a	a	a	c	(1, 1, c)
3	**a**	**a**	**c**	$\boxed{a}$	a	c	a	b	c	a	b	a	a	a	c	(3, 4, b)
4	a	a	**c**	**a**	**a**	**c**	**a**	**b**	$\boxed{c}$	a	b	a	a	a	c	(3, 3, a)
5	a	a	c	a	a	c	**a**	**b**	**c**	**a**	**b**	**a**	$\boxed{a}$	a	c	(1, 2, c)

Figure 6 An example of LZ77 with a dictionary of size 6 and a look ahead buffer of size 4 [2]

3) Move the cursor n + 1 characters forward.[2]

The position p can be given relative to the cursor with 0 meaning no match, 1 meaning a match starting at the previous character, etc..[2]

To decode the message we consider a single step. We assume that the decoder has correctly constructed the string up to the current cursor, and we want to show that given the triple (p, n, c) it can reconstruct the string up to the next cursor position. To do this the decoder can look the string up by going back p positions and taking the next n characters, and then following this with the character c. A problem arises when n is greater then p, as in step 3 of the example in Figure 6.

The problem is that the string that needs to be copied overlaps the lookahead buffer, which the decoder hasn't filled yet. In this case the decoder can reconstruct the message by taking p characters before the cursor and repeat them after the cursor so that n positions would be filled.[2]

b. LZ78

Statistical models, such as the Huffman model, usually encode a single symbol at a time by generating a one-to-one symbol-to-code map. The basic idea behind a dictionary-based compressor is to replace an occurrence of a particular substring or group of bytes in a piece of data with a reference to a previous occurrence of that substring. [3]

The LZ78 algorithm, as previously mentioned, works by constructing a dictionary of previously appearing substrings, which we will call “phrases". The LZ78 algorithm constructs its dictionary on the fly, only going through the data once. This means that we don’t have to receive the entire document before starting to encode it. The algorithm parses the sequence into distinct phrases. [6]

VII. Limitations

Lossless data compression algorithms cannot guarantee compression for all input data sets. Meaning for any lossless data compression algorithm, there exists a data set that does not get smaller when the algorithm is applied, and for any lossless data compression algorithm that makes at least one data set smaller, there exists a data set which the algorithm makes bigger. [7]

This can be proven using the counting argument. Assuming that every file is represented as a string of bits of some arbitrary length N, means that there are 2^N possible bit string combinations. Knowing that the algorithm is supposed to decrease the length of the file this leaves us with the biggest value for length being N – 1, or 2^{N-1} possible combinations. This would mean that there would have to be an injection from 2^N to 2^{N-1}, which by the pigeonhole principle is impossible. [7]

References

[1] S.R. Kodituwakku; U.S. Amarasinghe. “Comparison of Lossless Data Compression Algorithms for Text Data”. In: Indian Journal of Computer Science and Engineering Vol 1. No 4 (Dec. 2010-2011), pp. 416–426.ISSN: 0976-5166.

[2] Guy E. Blelloch. Introduction to Data Compression. Carnegie Mellon University, 2013.

[3] Stanford University, Dictionary-based Compressors.https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossless/lz78/index.htm. Accessed: 2020-03-03.

[4] Ze-Nian Li; Mark S. Drew. Fundamentals of Multimedia. Pearson Education International.

[5] Jr. Glen G. Langod. “An Introduction to Arithmetic Coding”. In: IBM J. RES DEVELOPVol28.No 2 (Mar. 1984), pp. 135–149.

[6] Michel Goemans.Lempel-Ziv codes; lecture notes. https://math.mit.edu/%e2%88%bcgoemans/18310S15/lempel-ziv-notes.pdf Accessed: 2020-03-03. 2015.

[7]Stanford University lecture notes, The Pigeonhole Principle https://web.stanford.edu/class/archive/cs/cs103/cs103.1132/lectures/08/Small08.pdf Accessed: 2020-07-22