

Lossless Compression: A Review

Ilan Sutskov,
Signal and Image Processing Lab.
Dept. of EE, Technion - IIT.



1. [Introduction](#)
 - a. [About this chapter](#)
 - b. [Motivation](#)
 - c. [The basic theorems.](#)
2. [Common compression schemes for known probability distributions.](#)
 - a. [Introduction.](#)
 - b. [The Huffman codes.](#)
 - c. [The Arithmetic encoder/decoder.](#)
 - d. [Run-length encoding.](#)
3. [An introduction to unknown distribution.](#)
4. [Common compression schemes for unknown probability distributions.](#)
 - a. [Universal schemes versus Non-Universal schemes.](#)
 - b. [Double-pass methods.](#)
 - c. [Sequential algorithms.](#)
 - 1) [Compression using an arithmetic coder.](#)
 - 2) [Adaptive Huffman codes.](#)
 - 3) [The Lempel-Ziv algorithms.](#)
 - d. [Compression of non-stationary processes.](#)
5. [LOCO-I: ISO standard for lossless compression of images \(JPEG-LS\)](#)
6. [Bibliography.](#)

1. Introduction

1.1 About this chapter

This chapter is dealing with a topic called "Lossless Compression". The explanations are intended toward students familiar with probability theory (and familiarity with random processes is also an advantage). The basic definitions like probability, expectation, stationarity, etc. are not reviewed, and the reader is considered to be familiar with them.

We use special notation to discriminate among terms. Here is a list of notes.

X	A name of a random process/ a source.
X_1, \dots, X_n	A sequence of n random variables, which may be a part of the random process X .
x_1, \dots, x_n	A realization of the random process. Small letters always denote deterministic values.
A	The "alphabet". This is a set of all possible values that a random variable may obtain. E.g., if $A = \{0,1\}$ then we say that we have a binary alphabet, and our random variables may only take the values 0 or 1. We denote the size of the alphabet by $ A $ or simply A (not bold). For binary alphabet $A=2$. In our discussion the alphabet is always finite. We may consider k random variables as a single random vector (also called k -block) over the alphabet A^k .
E	The expectation operator.

1.2. Motivation

Lossless compression is a way of compactly representing data so that a perfect reconstruction is feasible. To understand the cause of success of such algorithms, we look at the following demonstration. Suppose we are given the sequence (read it from left to right)

1 1 1 0 1 1 0 0 1 0 1 1 1 1

emitted out of a binary i.i.d. (independent and identically distributed) source having the probabilities $P(1) = 0.9$ and $P(0) = 0.1$. We suggest the following mapping algorithm, which replaces blocks of fixed size (2 bits block) with strings of variable size:

Input block	Output block
0 0	0 0 0
0 1	0 0 1
1 0	0 1
1 1	1

Then, we get the sequence

1, 0 1, 1, 0 0 0, 0 1, 1, 1

which is 11 bits long instead of the original 14 bits long (we added the commas to ease the reading). One can explain the success of this code: since '1's are much more probable, the blocks '11' will appear many times, thus our algorithm represents them by short length codes, whereas blocks like '00', whose probability is low, are represented by long codes that appear rarely.

We should notice, however, that our codes must maintain the property of uniquely decipherability (UD) that means that we can interpret/decode a code string in only one way. A UD code is one that no commas are needed to separate between adjacent words within it. The code mentioned above is such a code, whereas a code like

Input block	Output block
0 0	1 0 0
0 1	0 0 1
1 0	0 1
1 1	1

is not UD, since we can't tell if the string '10011' is the code word of '00 11 11' or '11 01 11'.

Thus, the main goal of a good lossless compression scheme is to assign shorter codes to higher probability strings, while maintaining a UD code.

Notice that only stationary sources are considered - now and in the sequel.

1.3 . Basic theorems

One can immediately ask, is there a limit to the length of the code (that is - how short can we get)? The answer turns to be affirmative, but before we support it with a theorem, we define a term that plays a major role in the realm of information theory.

The *Entropy* of a single symbol (also called the 1st order entropy) is defined by

$$H_1(X) = \sum_{x \in \mathbf{A}} P(x) \log_2 \frac{1}{P(x)}$$

where \mathbf{A} is called the *alphabet*, and is the set of all possible symbols. We can expand the definitions to the k-th order entropy such that

$$H_k(X) = \frac{1}{k} \sum_{x_1, \dots, x_k \in \mathbf{A}^k} P(x_1, \dots, x_k) \log_2 \frac{1}{P(x_1, \dots, x_k)}$$

We'd like to state an important property of the entropy

$$\log_2 |\mathbf{A}| \geq H_k(X) \geq H_{k+1}(X) \geq 0$$

Let us denote by $L(X_1, \dots, X_n)$ the length function of a UD algorithm compressing the string X_1, \dots, X_n .

Theorem 1 [1]

For any UD code $\frac{EL(X_1, \dots, X_n)}{n} \geq H_n(X)$

That is, the expected per-letter rate is always greater than or equal to the entropy. The rate of a random variable is the number of bits that may be used to represent it. For example, if we are given a 256-gray-levels non-compressed image, then 8 bits are used to represent each pixel. In this case we say that the rate of the image is 8 bits per pixel (denoted by 8 bpp) or 8 bits per symbol.

Theorem 2 [1]

There exists an algorithm such that $\frac{EL(X_1, \dots, X_n)}{n} < H_n(X) + \frac{1}{n}$

Both theorems are due to C.E. Shannon, who started the area of information theory in 1948.

2. Common compression schemes for known probability distributions

2.1. Introduction.

First, let's consider the case of memoryless sources, that is source for which

$$P(X_k | X_{k-1}, X_{k-2}, \dots, X_1) = P(X_k)$$

Due to the independence among symbols, the entropy of any order is equal to the first order entropy $H_1(X)$. Does that mean that we can reach the entropy using only 1-symbol blocks? The answer in the general case is NO.

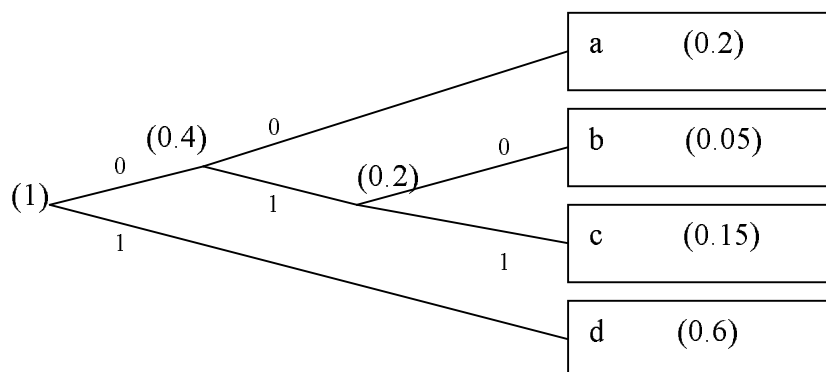
Though a rate of $H_1(X) + 1$ is reachable, it is totally unacceptable if, for example, our source was the binary source of the previous example, whose entropy is $H_1(X) \approx 0.46$. One can easily encode at a rate of 1 bit per symbol by not doing anything, so the upper bound of $0.46 + 1 = 1.46$ is meaningless. However, by using very long n-blocks, we can get (see [theorem 2](#)) a length that is less than $0.46 + \frac{1}{n} \dots$

We now introduce three very practical and common coding schemes: [the Huffman codes](#), the [arithmetic codes](#) and [run-length codes](#).

2.2. The Huffman codes [2]

The construction of Huffman codes is easy. We order at one side all the symbols, and from bottom to top we start connecting them in a tree structure. The rule of connection is that each time we connect the lowest two probabilities to form a new node in the tree. This node is assigned the probability of the sum of its two sons' probabilities. Here is an example: Consider a source with 4 letters alphabet {a,b,c,d} such that $P(a) = 0.2, P(b) = 0.05, P(c) = 0.15, P(d) = 0.6$

Then the proper tree is the following



Now our code is a walk in the tree from the root to the leaves, whereas each right turn is assigned the bit '1' and left turn is assigned the bit '0'. We get the code

a	00
b	010

c	011
d	1

Let's calculate the entropy of this source and the expected length that the Huffman code yields.

We assume that our source is i.i.d, so that all orders entropy are equal to the first order entropy, thus $H = 0.2 \log_2 \frac{1}{0.2} + 0.05 \log_2 \frac{1}{0.05} + 0.15 \log_2 \frac{1}{0.15} + 0.6 \log_2 \frac{1}{0.6} = 1.533$ bits per symbol.

The expected length of our Huffman code is

$$EL = 2 \cdot 0.2 + 3 \cdot 0.05 + 3 \cdot 0.15 + 1 \cdot 0.6 = 1.6 \text{ bits per symbol.}$$

Huffman code is proved to be the optimal code (in the means of expected length!) out of all UD codes using given fixed size blocks of data.

Note, that if the performance of 1-symbol Huffman code is not satisfying, then grouping symbols into 2-symbols block (or even k-symbol blocks) will improve our performance. In the case of a memoryless source will get $\frac{EL(X_1, \dots, X_n)}{n} \leq H_1(X) + \frac{1}{n}$

and in the case of a source with memory, using k-blocks we can get

$$\frac{EL(X_1, \dots, X_n)}{n} \leq H_k(X) + \frac{1}{n} \text{ which is even better !}$$

For example, using 2-symbols block (i.e. our symbols are 'aa', 'ab', 'ac', 'ad', 'ba', 'bb', ... and the probability of 'aa' is $0.2 \cdot 0.2 = 0.04$ because we assume that our source is i.i.d.) we get $\frac{EL(X_1, X_2)}{2} = 1.533$ (there is a difference between the entropy and this value, but it is only about 6 places after the decimal point.)

2.3. The arithmetic encoder/decoder [4]

If Huffman code is so good, what do we need other codes for ? The answer is in the structure of the Huffman code. First, using k-block exponentially enlarge the size of the table. For example, using a 256-symbol alphabet (as in the case of a gray-level image) and only 2-symbols block, we need a table with 65536 entries ! This is too big to handle most of the times. Second, we may wish to use blocks of variable length due to side information, the data structure or the source distribution structure.

The arithmetic coding scheme was originally introduced by Rissanen, and it is a U.S-patent (belongs to I.B.M). Its main advantage is the ability to work one symbol at a time.

Arithmetic coding is done by probability assignment. If every symbol x is assigned a probability $q(x)$, then the arithmetic encoder can generate a code whose length is

upper bounded by $E \log_2 \frac{1}{q(x)} + 2$ (we won't prove it of course). Observe that if $q(x)$

is the true probability function $P(x)$ then we get

$$\frac{EL(X_1, \dots, X_n)}{n} \leq H(X) + \frac{2}{n}$$

so in order to compress well, we use $q(x) = P(x)$, and we obtain a rate which is very close to the entropy again, while there is no need to handle large tables like Huffman code does.

Here is how it works. We take the $[0, 1]$ interval and slice it to pieces, so that the length of each piece equals the probability of the appropriate symbols (this is shown also by a diagram in the next page). We use the example given above to get the intervals: $[0, 0.2]$ $[0.2, 0.205]$ $[0.205, 0.4]$ $[0.4, 1]$.

Next, we find a real number that is in our interval (say our symbol was 'c', then we may choose the number 0.3, which is between 0.205 and 0.4). Upon receiving the next symbol, we slice again an interval according to the symbols' probabilities, but this time we slice our former appropriate interval (in our example we slice the interval $[0.205, 0.4]$ to $[0.205, 0.244]$ $[0.244, 0.244975]$ $[0.244975, 0.283]$ $[0.283, 0.4]$). We pick the proper interval according to our new symbol and pick a number within it (say this time the symbol is 'a', then we may choose the number 0.24, which is between 0.205 and 0.244). The chosen number forms our code.

The nesting property of the intervals causes this code to be [UD](#). In our example the number chosen 0.24 is between 0.205 and 0.4, so we know that the first symbol was 'c' and then it is also between 0.205 and 0.244 so we know that the second symbol was 'a'. One can immediately see that an arithmetic coder must also know the number of bits expected.

Here are some FAQ with their answers:

1) Q: Is there a way of choosing the number (that forms the code) progressively ? In our code we may first transmitted the number 0.51 and then 0.506. We would have liked to transmit a number like 0.50 (if it has the same meaning as 0.51) and then only add another digit to make it 0.506 (this example does not comply with the previous example of course).

A: Yes. There is a way to choose the number such that it is surely the prefix of the code number of a longer sequence.

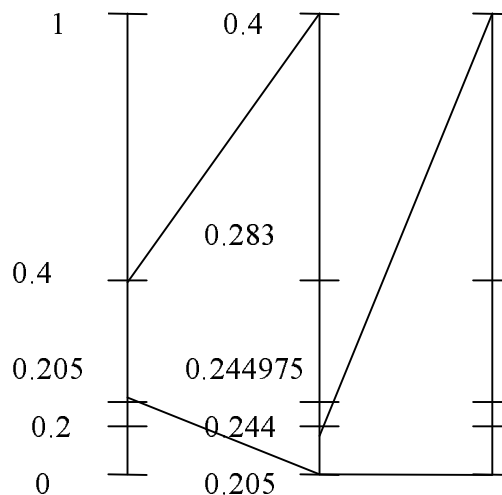
This method makes the arithmetic code a progressive code, so that indeed we can transmit the code symbol by symbol.

2) Q: What makes this algorithm work well ?

A: At each time instant, we choose a number out of an interval whose length is proportional to the probability. Thus, for a symbol with low(high) probability we need longer(shorter) code to specify our number. Observe, for example, that our first symbol 'c' was transmitted by the number 0.3, which is one digit after the zero, whereas if we have got the symbol 'b' instead (which is of lower probability) we must have send at least two digits number such as 0.202 .

3) Q: Is it true that we need infinite precision to encode/decode a very long string ?

A: No. There is a way of solving this precision issue. We do it by expanding the interval to $[0,1]$ at each time instant, as can be seen in the following diagram.



4) Q: What can we do if the probability distribution changes during time ?

A: Using an arithmetic coder, this is not a problem. Just adjust the interval slicing every time such a change occurs. Note that the change must be known to the decoder also, so that a change will be made by it or else a mistake will occur.

2.4. Run-Length encoding [5]

The run-length codes are not as generic as its predecessors, therefore they are optimal only for a small class of sources. Run-length codes are good for sources which emits long sequences of constant values (for those who are familiar with Markov processes, run-length codes are usually good for Markov sources with low probability of transients).

Run-length encoding is simple. For each constant value sequence two values are transmitted: the constant value and the number of times it repeats.

An example for an application, for which run-length coding works well, is the compression of a text page image (like in a facsimile). Since many rows of the page are "white", it may suitably be compressed by run-length encoding. For each such long row of "whites" its value is transmitted and then its length.

Note that the length of the "run" is basically an unbound integer number. Though there are ways for constructing [UD](#) codes for the integers, they usually cost too much if the length is quite short. Therefore, for practical means we may wish to bound the length of the "run" to be represented by a constant amounts of bits. Adjusting this maximal length is an important parameter in the construction of a run-length code.

3. An introduction to unknown distribution

Though theoretically we can compress at a rate that is equal to the entropy, and no other algorithm can compress better (in the expected length sense), one usually doesn't know the probability distribution of the source that emitted the given sequence. So what can we say now about the minimum expected length ?

We follow again the same guidelines of the [previous section](#). First we lower bound the best achievable expected performance, and then we try to suggest algorithms that can be shown to have a rate close to the lower bound.

Theorem 3

For many classes of sources, the expected code length is lower bounded by

$$\frac{EL(X_1, \dots, X_n)}{n} \geq H_n(X) + O\left(\frac{\log_2 n}{n}\right)$$

Especially, if the source belongs to a parametric class having K freedom degrees, then

$$\frac{EL(X_1, \dots, X_n)}{n} \geq H_n(X) + \frac{K}{2} \frac{\log_2 n}{n}$$

For example: an i.i.d. source with alphabet $A=\{a,b,c\}$ is having $K=2$ parameters which are the probability of 'a' and the probability 'b'. The probability of 'c' equals 1 minus these probabilities.

This theorem is due to Rissanen [6].

The new term added to the lower bound (compare with theorem 1), $O\left(\frac{\log_2 n}{n}\right)$, is usually known by the name “The Model Cost”, that is – it is the extra cost needed to specify the model itself (the cost of “telling” the decoder what the probabilities are).

In the next section, some lossless compression algorithms will be described, and their performance will be compared to the lower bound of theorem 3. We will see that this bound is tight and we can approach it closely.

4. Common compression schemes for unknown probability distributions.

4.1. Universal schemes versus Non-Universal schemes.

At the [previous section](#) we saw that the [model cost](#) is $O\left(\frac{\log_2 n}{n}\right)$. This is true when no a-priori information is involved. However this cost can be reduced if some a-priori information is given. Consider the example of image compression. It is well **known** that natural images are usually smooth, and due to the physical properties of cameras, the gray scale values of pixels are usually close one to another in a small neighborhood. Hence, we can encode only the difference between neighbor-pixels. This helps us reducing the size of the effective alphabet and having a better compression. This is a form of a-priori information. If the image is not “natural”, then this assumption is not valid, and we may even achieve worse performance. The alternative is to transmit the statistics first to the decoder, and then use them to compress the image. This is a **Universal encoding**, for which [theorem 3](#) holds. The example using the knowledge about natural images is using **Non-Universal encoding** since it is good only for some images, though we know that only those interest us.

The following subsections consider universal encoding (while the last section is the only example of non-universal encoding). Understanding universal encoding is important for the understanding of non-universal encoding too, because then we know where our weak points are.

4.2. Double-pass algorithms

Double-pass algorithms are algorithms that scan the data twice. At the first scan, statistics are gathered and useful empirical probabilities are evaluated. These statistics are transmitted to the decoder (this is the model cost). At the second scan, the sequence itself is transmitted using the statistics that were previously transmitted.

For example, histogram of symbols' appearance is calculated and transmitted to the decoder, and after that the estimated probabilities are the empirical probabilities given by the normalized histogram. Finally, the sequence is encoded using the empirical probabilities (by Huffman codes or arithmetic coder), and the expected length of this part can be proved to be at least as short as the entropy. So we stay with the question: how much does the transmission of the histogram costs?

We know that our source may emit symbols out of A options, thus we have A bins at the histogram. Our sequence is n symbols long, so the biggest value of every column is n , and it takes about $\log_2(n)$ bits to transmit it. Overall, we can transmit the histogram using $(A-1)\log_2(n)$ (the value of the last bin can be deterministically calculated and does not have to be transmitted), and the cost per symbol is

$(A-1)\frac{\log_2(n)}{n}$. This cost is the same as the cost of the model in [theorem 3](#), except for a factor of $\frac{1}{2}$.

We remember that the statistics of 1-block symbols can get us close only to the entropy of a single symbol. We can get better performance when using larger blocks. Let's say that we want to use k-blocks, then we need the statistics of k-blocks. The histogram now has A^k bins, and the cost of each bin is again at most $\log_2(n)$, so the cost of the histogram is $(A^k - 1) \frac{\log_2 n}{n}$. The cost of the model increases while we expect a decrease in the code length of the sequence compression. More about this subject may be found in [7].

Hence, using two-pass methods, one can approach the lower bound easily. The problems of two-pass algorithms are two: It has delay that is equal to the length of the entire sequence, and creation of k-histograms may be of high storage complexity. Therefore, it may be also useful to inspect the [sequential methods](#) (also called "on-line" methods or sometimes "adaptive algorithms").

4.3. Sequential algorithms

There are many universal sequential algorithms, most of which represent different trade-offs among computational complexity, storage complexity and compression ratio. We will review now several of the most famous algorithms.

4.3.1. compression using an arithmetic coder

[As previously seen](#), an arithmetic coder can provide us with performance upper bounded by

$$\frac{EL[X_1, \dots, X_n]}{n} \leq \frac{1}{n} E \log \left(\frac{1}{q(X_1, \dots, X_n)} \right) + \frac{2}{n}$$

where $q(\cdot)$ is the probability measure assigned **by us**. Were $q(\cdot)$ equal to the true source probability, then the entropy was reached. Our target using arithmetic coder, therefore, is to find probability assignments that are close to the true source probabilities.

First, for our algorithm to be sequential, we note the following equality

$$p(X_1, \dots, X_n) = p(X_1) \prod_{i=2}^n p(X_i | X_1, \dots, X_{i-1})$$

which leads to

$$\log_2 p(X_1, \dots, X_n) = \log_2 p(X_1) + \sum_{i=2}^n \log_2 p(X_i | X_1, \dots, X_{i-1})$$

and it follows that instead of trying to estimate the true $p(X_1, \dots, X_n)$ as one term, we may estimate **sequentially** the conditional probabilities $p(X_i | X_1, \dots, X_{i-1})$ by a learning algorithm resulting in $q_i(X_i | X_1, \dots, X_{i-1})$. In general, the update procedure of $q_i(X_i | X_1, \dots, X_{i-1})$ depends on the class to which our source belongs. A well-known probability assignment, known as the **Krichevsky and Trofimov** estimator [8,9] is given by the formula

$$q_i(x | X_1, \dots, X_{i-1}) = \frac{n(x) + \frac{1}{2}}{i - 1 + \frac{A}{2}}$$

where $n(x)$ counts the appearances of the symbol x within the sequence X_1, \dots, X_{i-1} (that is our past), and A is the size of the alphabet as always.

This estimator was proved to reach the single-symbol entropy plus a term equals to $\frac{A-1}{2} \frac{\log_2 n}{n}$, thus it is optimal for the compression of i.i.d. sources. Using alphabet extensions (like treating every k symbols as one extended symbol), the k -th order entropy may be reached, plus a higher cost of model.

Though not verified, we believe that almost all so-called “adaptive arithmetic coders” are using the Krichevski-Trofimov estimators.

4.3.2. Adaptive Huffman codes [3]

Again, [as with the adaptive arithmetic coder](#), the question reduces to proper probability assignments (this time needed for the construction of the Huffman tree). We remark that the optimal probability assignment, relating [to the estimator of Krichevski and Trofimov](#), is reducing to ordering by frequency of appearance. That is, the K&T probability estimated for a symbol is bigger than the probability estimated for another symbol if and only if the former occurred more times than the later. This makes the problem of constructing the Huffman tree depend only on the number of occurrences for each letter.

The adaptive Huffman coding was deeply investigated, and some very fast VLSI implementations are known for it.

UNIX’s “compact” is applying adaptive Huffman to its files.

4.3.3. The Lempel-Ziv algorithms


The LZ algorithms were invented in 1977-8 by A.Lempel and J.Ziv. These algorithms are of very low computational complexity, and not that high storage complexity either. They are asymptotically optimal, which means that they converge to the source entropy (the lowest bound) when the sequence is long enough (infinitely long, in fact). The LZ algorithms were adopted by Microsoft for the compression of DOS files, thus becoming one of the most famous algorithms ever. However, it should be noticed that in a non-asymptotic manner, i.e. when the sequence is finite and not so long – which is usually the case with some applications like image compression etc., the LZ algorithms do not promise to be close to the best performance possible. Hence, for some applications LZ may not suit.

The LZ family has two main versions called LZ77 and LZ78 (after the year of invention). We will now briefly describe both.

The LZ77 algorithm [10]

LZ77, as also LZ78, is based on the concept of distinctive parsing. The algorithm is coding string by string, of variable lengths. The string encoded at each instant is the longest string that exists in the past (the past is known to the decoder too) including a continuation to the future (a short demo will soon explain). It is encoded by transmitting two numbers: the length of the phrase and its starting point. For example, consider the compression of the sequence 'a b c d e d e d f', when the substring 'a b c d e' was already encoded. Then the longest string following it is 'd e d', which is of length 3 and starts at position 4, as depicted below

Position	1	2	3	4	5	6	7	8	9
Sequence	a	b	c	d	e	d	e	d	f



and the code word for the string 'ded' is : (3,4).

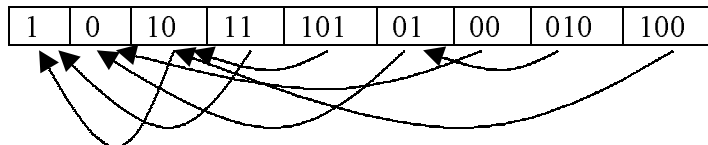
Though not proved here, this simple algorithm is as powerful as stated before.

UNIX's "gzip" and DOS's "pkzip" are applying LZ77 to their files.

The LZ78 algorithm [1,11]

The main drawback of the LZ77 is that though it is conceptually simple, the search for the longest phrase is quite complex. The LZ78 is much less complex though its performance is usually worse than LZ77.

In LZ78 the sequence is first parsed into distinct phrases this way: from our starting point we look for the longest string forward that exists in the past. Then, our phrase is this string plus the symbol that comes next. For example, we parse the binary sequence '1 0 1 0 1 1 1 0 1 0 1 0 0 0 1 0 1 0 0' to



(the arrows are showing for each phrase the longest string that constructed it). Then, for each phrase we transmit a pointer to its longest prefix (string) and we transmit the last symbol of it (in this example, for the phrase '010' we transmit a pointer to '01' which is the number 6 and then we transmit the bit '0').

As with the LZ77, this algorithm too was proved to achieve the source entropy asymptotically. However, as said before, it is known that the performance of the LZ78 for "short" sequences is worse than that of LZ77.

Due to the specific parsing process, a tree model, enabling low storage complexity and computational complexity may compactly represent the phrases.

UNIX's "compress" program is using a variation of LZ78.

4.4. Compression of non stationary processes

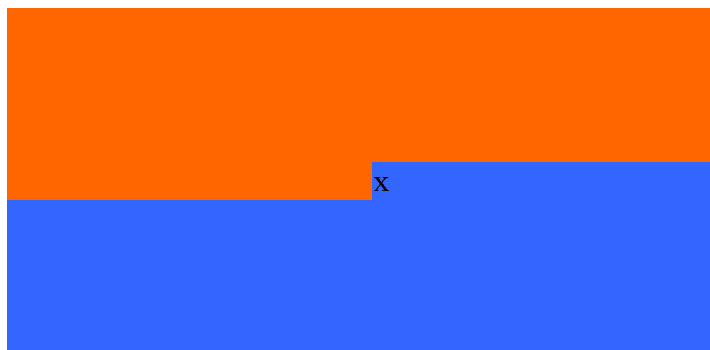
In some applications, it is common to consider the data as having only "local stationarity", i.e. its characteristics/statistical behavior are constant only in a somewhat small neighborhood. Hence, learning probabilities from the far past may be unwise, and a common solution is the use of a sliding window. Here, the probabilities are learned only from the most recent symbols of the past, and the algorithm adapts itself to the local neighborhood.

5. LOCO-I: ISO standard for lossless compression of images (JPEG-LS). [12,13,14]

This section is dedicated to a single compression algorithm initially called LOCO-I. On 1997 this algorithm was accepted as the new standard of the JPEG group in ISO for lossless compression, and it is called JPEG-LS [12]. The LOCO algorithm has two modes of operation: lossless and near lossless. In the later, the reconstructed sequence is allowed to be different from the original by a predetermined distance for each symbol. This section is reviewing only the lossless mode of LOCO.

LOCO is an algorithm designated specifically for the compression of images. This means that some a-priori information related to natural images is embedded inside. We will emphasize the use of this information for two purposes: to show the practical way of designing an algorithm to use this information and to show the difference between universal encoding and non-universal encoding.

Here is how LOCO works. The image is raster-scanned row by row. Each pixel is independently encoded as follows. Assume that the red (upper) area was already encoded and the blue (lower) area was not yet encoded.



The next symbol to be encoded (which is currently “blue”) will be denoted by x , and its neighbors by a, b and c , as follows

c	b
a	x

First, the pixel x is predicted using the neighbors. This prediction is performed by a constant predictor, which essentially assumes one of the two options: x is either on the plane defined by a, b and c , or x is in one side of an edge. In the former case, x is predicted by the equation $\hat{x} = a + b - c$, and in the later \hat{x} is replicated from a or b according to some rule. Then, the prediction error $e = x - \hat{x}$ is calculated. Since a, b and c are given to the decoder too, the given the error e , the decoder can perfectly reconstruct the value x . This technique is called DPCM (differential pulse code modulation), and it is a form of using an a-priori information. Our information is the knowledge that natural images are “smooth” the four pixels are almost linearly set. Note that for images which are not “smooth”, the performance may consequentially be very bad.

The error is then encoded using an entropy coder. Here another a-priori assumption is used: assuming that our predictor is good, then it will have many events zero error, some less amount of events error of value 1 (or -1), less 2 (or -2) etc. This behavior is nicely modeled by a two-sided geometric distribution (TSGD). Therefore, [a Huffman code](#) designed for TSGD is used, resulting in a code called Golomb-Rice code.

Since the image is usually only “locally stationary” and not stationary (another a-priori assumption), the error is not encoded using only one Huffman table, but rather several tables, each designed for the “context” of the encoded pixel.

A context of a pixel is defined by the following neighborhood of the pixel x

c	b	d
a	x	

The gradients {d-b, b-c, c-a} are vector quantized to yield a number between 0 and 354. Each such number is a measure of activity of the pixel x’s neighbors. For each context, the two parameters of the TSGD model are estimated from its previous occurrences (these parameters are the mean value and the variance), and then the Huffman table for the pixel’s context is built. Within this context, the pixel’s value is encoded.

Note that the quantization of the contexts is an a-priori information too. In light of our [previous discussions](#), the reasoning behind the quantization can be explained. It is a trade-off between model cost and success of compression assuming a model. If, on one hand, no quantization was done, then the model cost was reduced to nothing, but almost no statistics were gathered for each context (too many contexts compare to the short length of the image). In that case, the TSGD model estimation was poor, and no compression is likely to be achieved. On the other hand, if fewer contexts were defined, then we lose the “local stationarity” separation, and the Huffman tables are mismatch the information, and again poor results are expected. The quantization used is a trade-off between using correct Huffman tables (good compression under the model) and the model cost (amount of statistics in a context).

Finally, [run-length encoding](#) is utilized too by LOCO. Since Huffman codes are at least 1 bit long for each pixel, long sequences of constant value are encoded using run-length coding, resulting in a per-bit rate that is less than 1 bit per pixel.

Let's understand these two theorems. The first one says that we cannot find any compression algorithm whose *expected* length is shorter than the entropy. The second theorem justifies the definition of the entropy - not only it is a lower bound on our expected length, also it is a reachable bound. So one can get close to the entropy, but cannot get below it.

We would like to conclude with the following important observation: Using an algorithm that utilizes blocks of length k , the following holds

$$\frac{EL(X_1, \dots, X_n)}{n} \geq H_k(X)$$

whereas we already remarked that the k-th order entropy is bigger (or equal) to the n-th entropy (when $k < n$).

Bibliography

- [1] T. Cover and J. Thomas, *Elements of Information Theory*, New York: Wiley, 1991.
- [2] D.A. Huffman, "A method for the construction of minimum redundancy codes", Proc. of the IRE, Vol. 40. pp. 1098-1101, 1952.
- [3] R.G. Gallager, "Variations on a Theme by Huffman", IEEE Trans. On Information Theory, Vol. 24, No. 6, pp. 668-674, Nov. 1978.
- [4] P.G. Howard and J.S. Vitter, "Analysis of Arithmetic Coding for Data Compression", Information Processing and Management, Vol. 28, No. 6, pp. 749-763, Nov. 1992.
- [5] J.D. Murray and W. VanRyper, *Encyclopedia of Graphics File Formats*, 2nd edition, O'Reilly & Assoc. Inc., 1996.
- [6] J.J. Rissanen, "Universal Coding, Information, Prediction and Estimation", IEEE Trans. On Information Theory, Vol. 30, No. 4, pp. 629-636, July 1984.
- [7] L.D. Davisson, "Universal Noiseless Coding", IEEE Trans. On Information Theory, Vol. 19, No. 6, pp. 783-795, Nov. 1973.
- [8] R.E. Krichevsky and V.K. Trofimov, "The performance of Universal Encoding", IEEE Trans. On Information Theory, Vol. 27, No. 2, pp. 199-207, March 1981.
- [9] M.J. Weinberger, J.J. Rissanen and R.B. Arps, "Applications of Universal Context Modeling to Lossless Compression of Gray-Scale Images", IEEE Trans. On Image Processing, Vol. 5, No. 4, pp. 575-586, April 1996.
- [10] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression", IEEE Trans. On Information Theory, Vol. 23, No. 3, pp. 337- 343, May 1977.
- [11] A.D. Wyner and J. Ziv, "The Sliding-Window Lempel-Ziv Algorithm is Asymptotically Optimal", Proc. on the IEEE, Vol. 82, No. 6, pp. 872-877, June 1994.
- [12] ISO/IEC JTC1/SC29 WG1 ITU-T SG8 (JPEG/JBIG) "CD 14495, Lossless and near-lossless coding of continuous tone still images (JPEG-LS)"
- [13] M.J. Weinberger, G. Seroussi and G. Sapiro, "LOCO-I: A low complexity, context-based, lossless image compression algorithm", Proc. 1996 Data Compression Conference (Snowbird, Utah, USA), pp. 140-149, March 1996.
- [14] M.J. Weinberger, G. Seroussi and G. Sapiro, "The LOCO-I Lossless Image Compression Algorithm: Principles and Standardization into JPEG-LS", Hewlett-Packard Internal Report HPL-98-193, Nov. 1998.