CDI15

5. *Other lossless procedures*

303 SXD

4.1. Run-length encoding

4.2. Differencing

4.3. Move-to-front coding

4.4. Residual coding

4.1. Run-length encoding (RLE)

It is used in text and image compression procedures.

Idea. If x appears n consecutive times in the input stream (*run length of* n), encode those occurrences as $\{n, x\}$, or just nx .

To distinguish, in the coded sequence, a character n from the counter n in nx, instead of nx we will write $´nx$, where ´ is an 'escape' character.

Example. RLE(bbaaabaabb)→´2b´3ab´2a´2b.

We note that a run length of 2 needs 3 characters. So it is better to apply the $´nx$ rule only when $n > 2$. In this way we have

RLE(aaabbbbaabbbbaa)→´3a´4baa´4baa.

Remark. The compression ratio is not that impressive. For a message of N characters, with r runs of (average) length $n \geq 3$, it is

$$(N - r \cdot n + 3 \cdot n\,)/N = 1 - n \cdot (r - 3)/N.$$

For example, if $r = 20$ and $n = 13$ in a message of length $N = 400$, the compression would be 0.5.

We see that the compression improves when n or r increase. For text this seldom occurs (except maybe for some particular characters, like _), but it may (and will) be of use in other contexts (monochrome images, for example, and in particular in fax encodings).

Digram encoding. Consists in replacing commonly occurring pairs of characters by a single character not used in the message.

the_fat_cat_sat_on_the_mat

th→x, e_→y, at→z

xyfz_cz_sz_on_xymz

This a compression of 18/26≃0.69

This can easily be modified in order to take into account substitutions of longer substrings.

the_fat_cat_sat_on_the_mat

th→x, e_→y, at_→z

xyfzczszon_xymat

which amounts to a compression 16/26≃0.62

In computer programs, for example, we could substitute reserved words by a single character c, or by ´c if there are more such reserved words than characters available. This is usually referred to as p*attern substitution.*

4.2. Differencing

This method is also called *relative encoding*.

Suppose we have the sequence of integer values

23,21,23,25,23,25,23,21,23,23,25,23.

We can compress it starting with the first item followed by the differences of the remaining items with the first:

23,-2,0,2,0,2,0,-2,0,0,2,0

If there were a large jump, as in

23,21,23,25,23,25,23,51,53,53,55,54

then we could encode as

23,-2,0,2,0,2,0,51,2,2,4,3

For this to work, the decoder will need a flag (1 bit) to know whether the value it is decoding is an actual value or a difference. So the encoder will sent a {0,j} if the current encoded item j is a value and {1,j} if it is a differ-

ence. The decoding will return j if the flag is 0 and j+v if it is a difference, where v is the last decoded value.

Remark. The flag values can be sent in blocks. Suppose that the differences are bytes. Then the compressor can send, for example, 8 flag values (as a byte, say) just after sending 8 characters.

4.3. Move-to-front coding

A	x	k
abcde	a	0
	a	0
	b	1
bacde	e	4
ebacd	e	0
	c	3
cebad	e	1
ecbad	a	3
aecbd	d	4
daecb	e	2
edacb	a	3
cedab	c	0
cedab		

Suppose we have the message

M={a,a,b,e,e,c,e,a,d,e,a,c}.

The alphabet here is A={a,b,c,d,e}. In this example, the move-to-front scheme operates as is explained presently (see the table).

Conventions

x current character; initially the first character of M.

k coded value of x;

A current state of the *dictionary*.

At the start, A is the alphabet ordered in some way (lexicographically, say). A step of the encoding works as follows:

k is set to the *number of characters in* A *preceding* x;

x is moved to the front of A;

x is set to next character .

So the coded message, according to the table, is

{0,0,1,4,0,3,1,3,4,2,3,0}.

x	k
a	0
a	0
b	1
e	4
e	4
c	2
e	4
a	0
d	3
e	4
a	0
c	2

Remark. The original M has 12·8=96 bits. How many bits do we need to represent the encoding?

According to the frequencies of the characters in the message, we could use the source

S={{4,a},{1,b},{2,c},{1,d},{4,e}}

to assign an entropy of 2.085 to the alphabet. Thus the expected number of bits would be 25.2. For comparison: if we do a Huffman encoding of A, then it turns out that we can represent the encoding with 30 bits.

4.4. Residual coding

This is a sort of generalization of Differencing (Section 4.2). The model there was that the data would present small variations around a constant value. In the same way, we could give a rule that generates values that are small variations of the real data. Then the rule is called the *model* and the differences between the data and the model values are called *residuals.*

***Example*.** The list of integers

{41,43,46,52,62,71,82,98,115,131}

be encoded as by providing the model $f: n \to n^2 - n + 41$ and the residuals

{0,0,-1,-1,1,0,-1,1,2,0},

as the first 10 values of f are

{41,43,47,53,61,71,83,97,113,131}.

Remark. The values supplied by f are prime for n in 1..40.

JPEG-LS

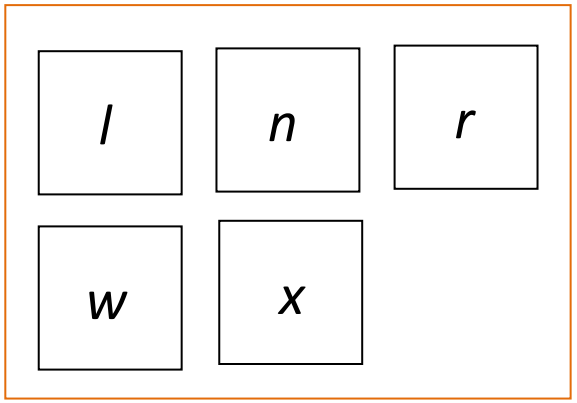
We will discuss this algorithm (JPEG lossless) for gray scale images. For color images it works by compressing in a similar way each of the three color planes.

A gray scale uncompressed digital image is a matrix whose entries correspond to the grey level of the pixels. If we use 256 levels of gray, then an uncompressed image amounts to $m \times n$ bytes.

The algorithm successively looks at the rows from top to bottom, and in in each row it scans the pixels from left to right (this order is called *raster order*). At each pixel, it tries to predict its value on the basis of (some) previously seen pixels (pixels to the left of the current row or pixels in previous rows) and this prediction leads to a *residual* with respect to the actual value. These residuals is what is used for the compression.

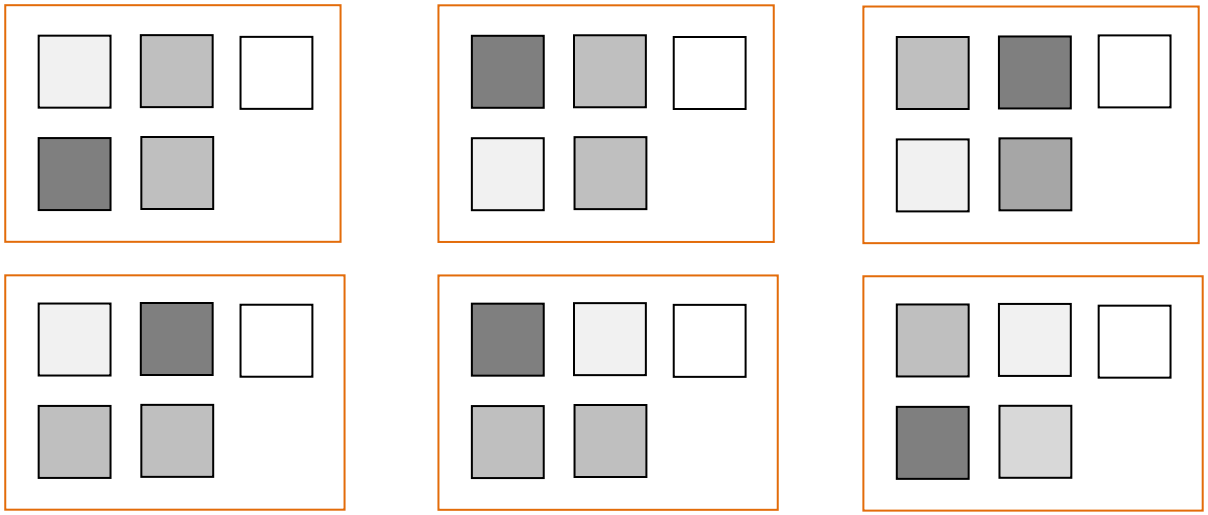
The whole idea lies in the fact that in most images each pixel is highly correlated with the nearby pixels.

In the case of JPEG LS, only 4 other pixels are used for the prediction of a pixel x: those denoted *w*, *l*, *n* and *r* on the figure .

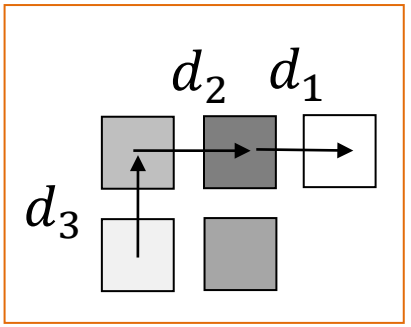


The first step in the prediction is to compute a guess $\hat{x}$ for the value of x:

$$\hat{x} = \begin{cases} \max(w,n) \text{ if } \max(n,w) \le l \\ \min(w,n) \text{ if } \min(n,w) \ge l \\ n + w - l \quad \text{otherwise} \end{cases}$$

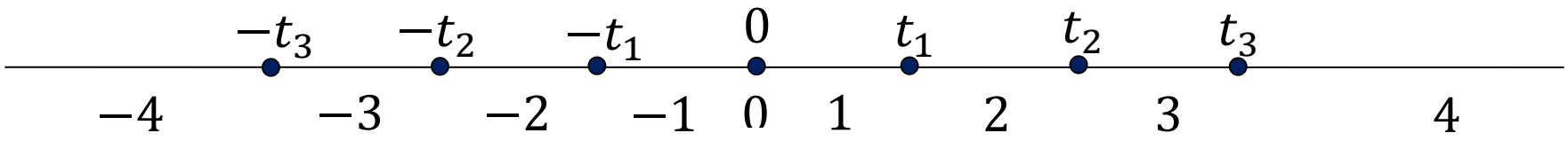


In a second step the above prediction is refined. Here is the outline of how it goes. The following differences (gradients) are computed



$$d_1 = r - n,\ d_2 = n - l,\ d_3 = l - w.$$

With these, a *context vector* $q = [q_1, q_2, q_3]$, $d_i \mapsto q_i$, is computed by assigning to each d_i a value in$\{-4, -3, -2, -1,0,1,2,3,4\}$ according to the interval in which it lies in the following scheme (t_1, t_2, t_3 are positive numbers, which in principle can be chosen by the user):



More formally, the intervals are

$(-\infty, -t_3), [-t_3, t_2), [-t_2, t_1), [-t_1, t_0), [0], (-0, t_1], (t_1, t_2], (t_2, t_3], (t_3, \infty)$

and the corresponding values are $-4, -3, -2, -1,0,1,2,3,4$, respectively.

Thus the number of possible contexts is $9^3 = 729$. These are reduced to 365 with the following rule:

> If q is non-zero and its first non-zero component is negative, then do $q \mapsto -q$ and retain this change in a variable SIGN (-1 if a change has been effected, $+1$ otherwise).

The 365 contexts are further enumerated in the range 0..364 (the standard does not specify in which order) and the value is multiplied by SIGN. The result, added to the first step prediction, is the refined prediction.

In a final step the residual d (the difference between the refined prediction and the actual value) is normalized so that it lies in the range

$$-\frac{M}{2}..\frac{M}{2}$$

where M is the number of grey levels ($M = 256$, for example):

$$\begin{cases} 0 & \text{if } d = 0 \\ d + M & \text{if } d < M/2 \\ d - M & \text{if } d > M/2 \end{cases}$$

Remark. The residuals, which we expect to be small (this is the whole business about any prediction), are further encoded by means of (adaptively selected codes based on) Golomb codes. This adds to the compression capacity of the algorithm.

The following table and images have been taken from [Sayood-2006]. Images are 256x256, and grey levels are in the range 0..255. So each uncompressed image is $16^2 \times 16^2 = 2^{16} = 65536$ bytes.

Image	Old JPEG*	JPEG-LS	CALIC*
Sena	31,055	27,339	26,433
Sensin	32,429	30,344	29,213
Earth	32,137	26,088	25,280
Omaha	48,818	50,765	48,249

* See below.

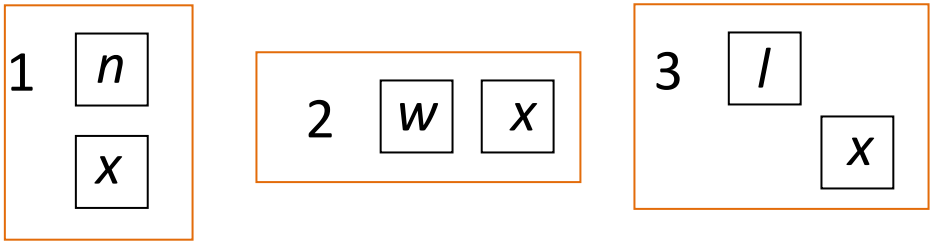
Old JPEG [Wallace-1973]. For lossless still compression, it provides eight different predictive schemes from which the user can select. The first scheme makes no prediction, so there is no more to say. The other 7 are as follows:

1 $\hat{x} = n;$ 2. $\hat{x} = w;$

3. $\hat{x} = l;$ 4. $\hat{x} = w + n - l;$

5. $\hat{x} = w + (n - l)/2 = w + d_2/2;$

6 $\hat{x} = n + (w - l)/2 = n - d_3/2;$ 7. $\hat{x} = (w + n)/2.$



CALIC: Context Adaptive Lossless Image Compression [Memon-Wu-1994]

Uses contexts and prediction. Works in two modes: grey-scale and bi-level. It is much more complex than JPEG-LS, but still compresses (as seen in the table above) a bit more than JPEG-LS does.

[Wu-Memon-1996] X. Wu and N.D. Memon. *CALIC—A context based adaptive lossless image coding scheme*. IEEE Transactions on Communications, May 1996.

[Memon-Wu-1997] N.D. Memon and X. Wu. *Recent developments in context-based predictive techniques for lossless image compression*. The Computer Journal, Vol. 40:127-136, 1997. [LOCO-I, from Low Complexity].

[Wallace-1991] G.K. Wallace. *The JPEG still picture compression standard*. Communications of the ACM, 34:31-44, April 1991.

Note. JPEG-LS is based on the LOCO-I algorithm (LOw COmplexity LOssless COmpression for Images). Its official standard number is ISO-14495-1/ITU-T.87.

Good writing is the art of lossy text compression

(Strung and White)