Glicbawls — Grey Level Image Compression By Adaptive Weighted Least Squares

Bernd Meyer, Peter Tischer
School of Computer Science and Software Engineering,
Monash University
Clayton, Victoria, Australia, 3800

Email: [bmeyer,pet]@csse.monash.edu.au

1 Introduction

In recent years, most research into lossless and near-lossless compression of greyscale images could be characterized as belonging to either of two distinct groups.

The first group, which is concerned with so-called "practical" algorithms, encompasses research into methods that allow compression and decompression with low to moderate computational complexity, while still obtaining impressive compression ratios. Some well-known algorithms coming from this group are LOCO [4], CALIC [1] and P2AR[2].

The other group is mainly concerned with determining what is theoretically possible. Algorithms coming from this group are usually characterized by extreme computational complexity and/or huge memory requirements. While their practical applicability is low, they generally achieve better compression than the best practical algorithm of the same time, thus proving beyond a doubt that the practical algorithms fail to exploit some redundancy inherent in the images. Well-known examples are UCM[3] and TMW [5]

What has been largely missing so far is an algorithm that combines the compression rates of the impractical algorithms with the moderate computational requirements of the practical ones. In this paper, we present Glicbawls, an algorithm that achieves that goal for natural images.

The current implementation can compress and decompress greyscale images with 1 to 30 bits per pixel, using raw or ASCII .PGM files, in both lossless and near-lossless mode. Colour images (raw and ASCII .PPM files) are also supported, and while compression rates for them are not world class, they are usually better than PNG's.

Due to the simplicity of the Glicbawls algorithm, a full featured encoder/decoder can be implemented in 1839 bytes of C code.[1] Even when including the decoder code with each compressed image the compression rates achieved are still extremely competitive

[1] This is also due to some rather atrocious abuses of the C language — Glicbawls was originally developed as an entry for the International Obfuscated C Contest.

and allow placing an absolute upper bound on the amount of information contained in an image.

2 Overview

Wu's P2AR[2] algorithm uses a single linear predictor whose weights are recalculated for each pixel. This is done by applying the least squares algorithm to the pixels in a rectangular window around the current pixel. By choosing a rectangular window, the computational complexity of calculating the least squares predictor can be markedly reduced, thus making the method practical.

However, giving equal weight to the contribution of each pixel within the rectangular window, and no weight to pixels outside the window, does not accurately reflect the correlations that exist within typical images. Pixels close to the current pixel should have more influence on the choice of weights than pixels farther away.

Glicbawls uses a predictor similar to Wu's, except that the predictor weights are recalculated by taking *all* previous pixels into account. Each pixel's contribution to the least squares algorithm is *weighted* by a factor of 0.8^{d_i} where d_i is the Manhattan-distance between the pixel i and the current pixel. Given this choice of weights, least squares predictors can be calculated using an algorithm similar in efficiency to that used in [2]

The resulting prediction errors are modeled by using the modified t-distribution introduced in [5]. The spread parameter σ of the distribution is calculated from the weighted average of the squared prediction errors for all previous pixels, with each pixel's contribution being given the weight 0.7^{d_i}. This average can also be calculated efficiently.

The actual entropy coding of the prediction errors according to the t-distribution is handled by a straightforward arithmetic coder.

3 Least Squares Predictor

The calculation of weighted least squares predictors is at the very heart of the Glicbawls algorithm. Only through an efficient implementation of this calculation could Glicbawls achieve its goal of practicality.

3.1 Definition of Predictor

At any point during the encoding, there are a number N of previous pixels, with pixel values $p_1, p_2, \cdots, p_N$ at coordinates $(x_1, y_1), \cdots, (x_N, y_N)$, as well as a current pixel at coordinates (x_C, y_C) whose pixel value p_C is to be predicted. For each of those pixels p_i, the 12 causal neighbours with a Manhattan-distance of three or less[2] are referred to as $\mathbf{n}_i = \begin{pmatrix} n_{i,1} \\ \vdots \\ n_{i,12} \end{pmatrix}$.

[2]Which neighbouring pixel ends up at what position in n does not matter, as long as it is consistent at each step

For each pixel p_i, a matrix $\mathbf{A}_i$ and a vector $\mathbf{b}_i$ can be calculated as

$$\mathbf{A}_i = \mathbf{n}_i \mathbf{n}_i{}^T \tag{1}$$
$$\mathbf{b}_i = p_i \mathbf{n}_i \tag{2}$$

From those, $\mathbf{A}_C$ and $\mathbf{b}_C$ are calculated as

$$\mathbf{A}_C = \sum_{i=1}^{N} 0.8^{|x_C - x_i| + |y_C - y_i|} \mathbf{A}_i \tag{3}$$

and

$$\mathbf{b}_C = \sum_{i=1}^{N} 0.8^{|x_C - x_i| + |y_C - y_i|} \mathbf{b}_i \tag{4}$$

Then solving the linear equation system[3]

$$\mathbf{A}_C \mathbf{w} = \mathbf{b}_C \tag{5}$$

will give the weights $\mathbf{w} = \begin{pmatrix} w_1 \\ \vdots \\ w_{12} \end{pmatrix}$ of the linear predictor that can be used to predict p_C from $\mathbf{n}_C$.

3.2 Efficient Calculation

Glicbawls encodes and decodes images in scanline order. This implies that y_C can never be smaller than the y_i of any previously seen pixel, and thus $|y_C - y_i| = y_C - y_i$.

Assuming the indices of the previously seen pixels p_i to be ordered according to their x_i coordinates in such a way that $x_i \leq x_{i+1}$, we can define N_q as the index of the first pixel with $x_i \geq q$ and M_q as the index of the last pixel with $x_i \leq q$. In effect, $p_{N_1} \cdots p_{M_1}$ will be the set of all pixels in column one, and $p_{N_q} \cdots p_{M_q}$ the set of all pixels in column q. Obviously, $M_q + 1 = N_{q+1}$.

We can then rewrite equation (3), using X for the number of columns in the image, as

$$\mathbf{A}_C = \sum_{q=1}^{X} (0.8^{|x_C - q|} \sum_{i=N_q}^{M_q} 0.8^{(y_C - y_i)} \mathbf{A}_i) \tag{6}$$

Defining $\mathbf{B}_q$ as

$$\mathbf{B}_q = \sum_{i=N_q}^{M_q} 0.8^{(y_C - y_i)} \mathbf{A}_i \tag{7}$$

[3]Actually, this is a simplification. See section 3.3 for one additional step that has been omitted from this explanation.

equation (6) can be written as

$$\mathbf{A}_C = \sum_{q=1}^{x_C-1} (0.8^{(x_C-q)}\mathbf{B}_q) + \sum_{q=x_C}^{X} (0.8^{(q-x_C)}\mathbf{B}_q) \tag{8}$$

We can refer to the two parts of that sum as $\mathbf{E}$ and $\mathbf{F}$, i.e.

$$\mathbf{E} = \sum_{q=1}^{x_C-1} (0.8^{(x_C-q)}\mathbf{B}_q) \tag{9}$$

$$\mathbf{F} = \sum_{q=x_C}^{X} (0.8^{(q-x_C)}\mathbf{B}_q) \tag{10}$$

Then $\mathbf{E}$ contains the contributions of all previous pixels *left* of the current pixel, while $\mathbf{F}$ contains the contributions of all pixels *above* the current pixel that are not in $\mathbf{E}$ (see figure 3.2). As can easily be seen, in the case where the current pixel is the top left corner pixel (i.e. the first one in scanline ordering), both are matrices containing only zeros. The same is true for all $\mathbf{B}_q$.

Whenever a pixel has been fully encoded or decoded, it is added to the set of previously seen pixels, and the algorithm then proceeds with the next pixel in scanline order. Let l be the index of the pixel just added to the set of previously seen pixels.

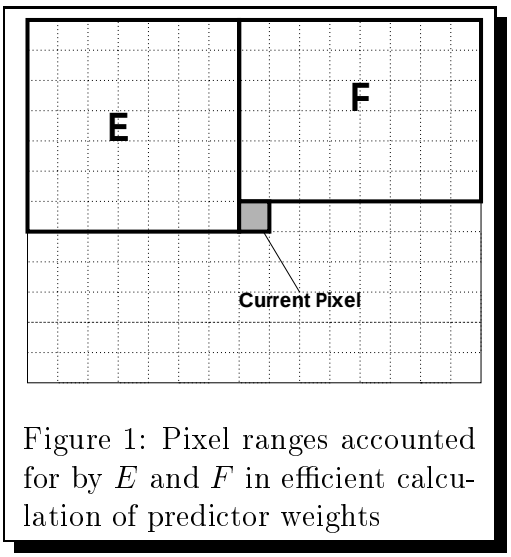

Figure 1: Pixel ranges accounted for by E and F in efficient calculation of predictor weights

3.2.1 Non-border Case

Usually, the next pixel is at coordinates $(x_C + 1, y_C)$, i.e. just to the right of the current pixel. Proceeding to it does not change y_C at all. As the $\mathbf{B}_q$ do not depend on x_C at all, the only $\mathbf{B}_q$ that changes is the $\mathbf{B}$ the $\mathbf{A}_i$ from the just finished pixel gets added to, which is $\mathbf{B}_{x_C}$. Thus

$$\mathbf{B}_q \leftarrow \begin{cases} \mathbf{B}_q & \text{if } q \neq x_C, \\ \mathbf{B}_q + \mathbf{A}_l & \text{if } q = x_C. \end{cases} \tag{11}$$

When proceeding to the right, a whole new column of pixels is added to the range of pixels covered by $\mathbf{E}$. The contributions of all the pixels in that column are contained in $\mathbf{B}_{x_C}$. On the other hand, for all the pixels that were already covered by $\mathbf{E}$ before, the Manhattan-distance from the current pixel is going to become one larger than it was before, and thus their contribution gets weighted down by a uniform factor of 0.8. Thus

$$\mathbf{E} \leftarrow 0.8\mathbf{E} + \mathbf{B}_{x_C} \tag{12}$$

using the already updated $\mathbf{B}_{x_C}$.

Similarly, proceeding to the right means that a whole column of pixels is removed from the range covered by $\mathbf{F}$. The contributions of those pixels are contained in the *pre-update* version of $\mathbf{B}_{x_C}$. For all pixels that remain in the range covered by $\mathbf{E}$, the Manhattan-distance from the current pixel is going to become one smaller, and thus their contributions get weighted up by a uniform factor of $\frac{1}{0.8}$. Thus

$$\mathbf{F} \leftarrow \frac{\mathbf{F} - \mathbf{B}_{x_C}}{0.8} \tag{13}$$

using the *pre-update* $\mathbf{B}_{x_C}$.

3.2.2 Border Case

Whenever the pixel just encoded or decoded was the rightmost pixel of a scanline, the next pixel is at coordinates $(0, y_C + 1)$, i.e. the leftmost pixel of the next scanline. Proceeding to it increases y_C.

The $\mathbf{B}_q$ do not depend on x_C at all, but they *do* depend on y_C. Increasing y_C by one will increase the exponent in their definition by one, and thus their values need to be scaled down by a factor of 0.8.

Of course, the $\mathbf{A}_i$ from the pixel just encoded or decoded needs to be added to the appropriate $\mathbf{B}_q$ *before* that scale-down:

$$\mathbf{B}_q \leftarrow \begin{cases} 0.8\mathbf{B}_q & \text{if } q \neq x_C, \\ 0.8(\mathbf{B}_q + \mathbf{A}_l) & \text{if } q = x_C. \end{cases} \tag{14}$$

When moving to the leftmost pixel of a new scanline, there will be no pixels to the left of the current pixel, and thus

$$\mathbf{E} \leftarrow 0 \tag{15}$$

$\mathbf{F}$, however, needs to be recalculated completely from the updated $\mathbf{B}_q$, to contain the influence of all the pixels seen so far:

$$\mathbf{F} \leftarrow \sum_{q=1}^{X} (0.8^{(x_i - 1)} \mathbf{B}_q) \tag{16}$$

3.2.3 Implementation Issues

While the above algorithm is mathematically correct, repeated application of equation (13) will lead to numerical instability. Also, calculating $\mathbf{F}$ directly according to equation (16) is inefficient.

The solution is to pre-calculate all values for $\mathbf{F}$ each time work on a new scanline is started. This can be done by defining

$$\mathbf{F}_q = \begin{cases} \mathbf{B}_q + 0.8\mathbf{F}_{q+1} & \text{if } q \leq X, \\ 0 & \text{if } q = X + 1. \end{cases} \tag{17}$$

At the start of each scanline, all $\mathbf{F}_1, \cdots, \mathbf{F}_X$ are calculated (starting at $\mathbf{F}_X$ and working downward, i.e. working from right to the left) and stored. These precalculated values are then used instead of the single, constantly maintained $\mathbf{F}$ described in the algorithm above. Except for differences introduced by numerical inaccuracies, they are identical. However, the storage required for the precalculated values roughly doubles the memory requirements of the efficient calculation.

All of the above has dealt only with $\mathbf{A}_i$ and $\mathbf{A}_C$ and has completely ignored the $\mathbf{b}_i$ and $\mathbf{b}_C$. The exact same methods are used for them. Simply replacing every $\mathbf{A}$ with $\mathbf{b}$ in the above will suffice to arrive at an efficient method for calculating $\mathbf{b}_C$.

3.3 Bias

One problem commonly encountered when using least squares predictors, especially ones calculated from relatively few observations[4], is that they tend to overfit data, sometimes producing horrendously large predictor weights. However, large weights are usually not desirable for predicting in the presence of noise, as they tend to amplify the noise in the predicted value.

To avoid this problem as much as possible, Glicbawls adds a bias u towards an averaging predictor to the equation system before calculating the weights. A matrix $\widehat{\mathbf{A}_C}$ and a vector $\widehat{\mathbf{b}_C}$ are defined as

$$\widehat{\mathbf{A}_C} = \mathbf{A}_C + u\mathbf{I} \tag{18}$$

$$\widehat{\mathbf{b}_C} = \mathbf{b}_C + \frac{1}{12}\begin{pmatrix} u \\ \vdots \\ u \end{pmatrix} \tag{19}$$

Instead of actually using equation (5) to calculate the predictor weights, we use

$$\widehat{\mathbf{A}_C}\mathbf{w} = \widehat{\mathbf{b}_C} \tag{20}$$

The strength of this bias, u, is initially given a value of 80, and then dynamically adapted during the encoding or decoding of an image. For each pixel, *two* predicted values are calculated, P_1 (which is used for coding) using a bias of u, and P_2 using a smaller bias of $0.9u$. For both, the prediction errors are calculated ($e_1 = P_1 - p_C$ and $e_2 = P_2 - p_C$). Then u is adjusted as follows:

$$u \leftarrow \begin{cases} u + e1 - e2 & \text{if } e_1 > 0 \\ u + e2 - e1 & \text{if } e_1 \leq 0 \end{cases} \tag{21}$$

Experiments show that this way of changing u allows for rapid adjustment in case the characteristics of the image change strongly, while being fairly robust in the presence of small, random, noise induced differences between P_1 and P_2

[4] Or, in the case of Glicbawls, calculated from many observations, relatively few of which are given enough weight to dominate the calculation.

4 Prediction Error Modeling

4.1 Modified t-distribution

Once the predictor has predicted a pixel value P for the current pixel, a modified t-distribution is centered around that prediction. The probability of the current pixel value being less than X is given by the formula

$$p(x < X) = K \int_{-\infty}^{X-P} (\frac{1}{1 + \frac{v^2}{13\sigma^2}})^{\frac{13}{2}} dv \tag{22}$$

and the probability of the current pixel value being between X_1 and X_2 by

$$p(X_1 \leq x < X_2) \quad = p(x < X_2) - p(x < X_1) \tag{23}$$

with K chosen so that $p(x < \infty) = 1$. Because in Glicbawls the probabilities passed to the arithmetic coder are always *ratios* of values given by these formulae, the constant factor K can be ignored for the calculation.

This distribution is similar to a Normal Distribution, but has the useful property of having more weight in the tails, and thus being more forgiving about large mispredictions by the least squares predictor. Its other useful property is that the integral can be solved analytically.[5] Ignoring the term K, that probability can be calculated as

$$p(x < X) = \quad \frac{(X-P)}{\sigma} r_2(\frac{(X-P)^2}{13\sigma^2} + 1) + c \tag{24}$$

$$r_n(y) = \begin{cases} \frac{(n-1)}{n} \frac{r_{n+2}(y)}{y} + \frac{1}{\sqrt{y}} & \text{if } n < 14, \\ \frac{1}{\sqrt{y}} & \text{if } n = 14. \end{cases} \tag{25}$$

where c is an integration constant that cancels out in equation 23.

4.2 Calculation of σ

To determine the distribution parameter σ and thus the width of the distribution, the weighted average S of the squared prediction errors for all previously seen pixels is calculated. Each pixel's contribution e_i^2 is being weighed proportionally to 0.7^{d_i}. This average can be calculated efficiently using the methods described in section 3.2. As the result of the calculation, however, is supposed to a scalar value (rather than an equation system), scaling has to be applied to ensure that the weights sum up to one. σ is then calculated as

$$\sigma = 0.964\sqrt{S} \tag{26}$$

The factor 0.964 was arrived at empirically and provides a slight compensation for the shape of the used distribution, which is slightly wider than a Normal Distribution with

[5] By looking it up in [6], the book the German engineering term "Bronstein-integrierbar" is derived from

the same σ.[6]

4.3 Avoiding Numerical Instability

Using the definitions given so far on a machine with limited numeric precision can in rare cases lead to a probability being calculated as zero. If that happens, the program would either fail or enter an infinite loop.

For this reason, $\widehat{p}$ is defined as

$$\widehat{p}(X_1 \leq x < X_2) = p(X_1 \leq x < X_2) + 10^{-6}(X_2 - X_1) \tag{27}$$

and $\widehat{p}$ is used instead of p for for coding. Because in Glicbawls, all probabilities passed to the arithmetic coder are *ratios* of two $\widehat{p}$ thus calculated, it is not necessary to explicitly renormalize the $\widehat{p}$ to ensure they add up to 1.

For most natural images, the difference of this modification on the compressed image size is negligible.

4.4 Coding

Pixel values are encoded by repeatedly subdividing the possible value range into two (roughly) equal sized parts, and encoding which part contained the actual value. Encoding stops when the possible value range has been reduced to a size small enough to ensure reconstruction with the required accuracy. For lossless compression, this is the case when the size of the value range reaches one, but for near-lossless compression with a maximum allowed error of e, reaching a size of $2e + 1$ is sufficient.

When encoding a greyscale image with a maximum pixel value of M, the initial range (i.e. the range of possible non-quantized pixel values) is $R_0 = [-0.5, M + 0.5)$

Whenever a value is known to be in the range $R_i = [X_1, X_2)$ and a further reduction in the size of the range is necessary, a value $\overline{X}$ is calculated that splits R_i into $R_{i,1} = [X_1, \overline{X})$ and $R_{i,2} = [\overline{X}, X_2)$.[7]

Given that the pixel value is known to be in R_i, the probability r_1 of it being in $R_{i,1}$ is

$$r_1 = \frac{\widehat{p}(X_1 \leq x < \overline{X})}{\widehat{p}(X_1 \leq x < X_2)} \tag{28}$$

A simple binary arithmetic coder is used to encode which part of R_i the actual pixel value was in (or, in the case of decoding, to provide that information). The range bounds are then adjusted, and if necessary, another iteration is taken.

[6] The actual implementation contains some extra detail which cannot be discussed in the limited space available. In particular, measures to ensure that σ does not become unreasonably small, even in the face of large areas of perfect prediction. For natural and thus noise-containing images, the influence of these measures is negligible.

[7] In the case of near-lossless compression, the total possible range is first divided into "bins" of size $2e + 1$, and $\overline{X}$ is chosen to be on a boundary between bins. The bins are aligned in such a way that the predicted pixel value is centered in the middle of a bin, thus minimizing the expected number of bits needed to encode the pixel.

5 Weight Adjustment

There is one more twist to the Glicbawls algorithm. As described so far, the unavoidably large prediction errors near edges in the image will dominate the least squares algorithm. The resulting predictors are well suited for predicting pixels near edges, but are generally suboptimal for non-edge regions.

Due to the way the σ parameter is calculated, however, the Glicbawls algorithm usually *expects* predictions for pixels near edges (i.e. in the vicinity of previous larger prediction errors) to be less accurate — σ will be larger for those pixels. Figuratively speaking, "getting it wrong" is not as much of a problem for those pixels as for the others.

For this reason, the influence of pixels for which σ was large should be reduced. In Glicbawls, this is done not by using the definitions in equations (1) and (2), but rather the following scaled versions:

$$\mathbf{A}_i = \frac{\mathbf{n}_i \mathbf{n}_i{}^T}{\sigma} \tag{29}$$

$$\mathbf{b}_i = \frac{p_i \mathbf{n}_i}{\sigma} \tag{30}$$

Different powers of σ were tried for the weight adjustment. There is no single "best" choice, different images compress best with different exponents, but simply reducing weights by $\frac{1}{\sigma}$ gives the best overall performance over a large suite of test images.

6 Self Extraction

Due to the small size of the C source that implements the Glicbawls algorithm, including it with the compressed data is feasible. The total overhead for including the complete (gzip-compressed) source code as well as a small shell script that extracts, uncompresses and compiles it and then uses the resulting executable to decompress an image, is no more than 1370 bytes. The resulting file is a shell script that will output the image to its standard output.

As that file contains *everything* needed to recreate the image file on any UNIX machine which has gunzip and a C compiler installed, its total size can serve as an absolute upper bound for the amount of information contained in the image.

7 Colour Images

In order to keep the Glicbawls code size small, colour images are essentially treated as greyscale images in which the colour components are interleaved on a per-pixel basis. Each row of the greyscale image is three times as wide as those of the colour image it was derived from. The values in columns $3n + 0$ of the greyscale image are the R components of the colour pixels in the matching columns n, the values in columns $3n+1$ are the G components, and the values in columns $3n + 2$ the B components.

When dealing with colour images, the local neighbourhood used to predict pixel values is modified by multiplying all horizontal offsets by 3. Effectively, this means that

	balloon	barb	barb2	board	boats	girl	gold	hotel	zelda	lenna	Avg
	lossless compression										
LOCO	2.90	4.65	4.66	3.64	3.92	3.90	4.47	4.35	3.87	4.24	4.06
CALIC	2.83	4.41	4.53	3.56	3.83	3.77	4.39	4.25	3.75	4.10	3.94
P2AR		3.98			3.64		4.30			3.96	3.78
Glicbawls	**2.64**	**3.92**	**4.31**	**3.39**	**3.63**	**3.56**	**4.28**	**4.18**	**3.54**	**3.90**	**3.74**
	near-lossless compression, e=1										
LOCO	1.64	3.15	3.17	2.20	2.48	2.45	3.00	2.87	2.37	2.71	2.60
Glicbawls	**1.32**	**2.43**	**2.81**	**1.94**	**2.17**	**2.11**	**2.75**	**2.66**	**2.05**	**2.39**	**2.26**
	near-lossless compression, e=5										
LOCO	.73	1.70	1.65	.86	1.13	1.26	1.51	1.38	1.19	1.37	1.28
Glicbawls	**.33**	**1.04**	**1.28**	**.60**	**.84**	**.81**	**1.23**	**1.14**	**.65**	**.89**	**.88**
	lossless compression, self extracting file										
Glicbawls	**2.67**	**3.94**	**4.34**	**3.42**	**3.65**	**3.59**	**4.30**	**4.20**	**3.56**	**3.94**	**3.76**

Table 1: Compression results, in bits per pixel, for Glicbawls compared to CALIC, LOCO and P2AR in lossless and near lossless modes

pixel values from one colour band are predicted based only on values from the same band. However, as the *weights* used for predicting pixel values are calculated as before, data in one colour band will still effect the predictions the others.

Also, no adjustment is made in the calculation of σ parameter. This means that the magnitude of prediction errors in one colour band will influence the expected magnitude of prediction errors in other bands.

8 Results

Table 1 lists file sizes (in bits per pixel) obtained by running Glicbawls[8], CALIC using arithmetic coding[9], LOCO[10] and P2AR[2][11] on a variety of test images. In all cases, Glicbawls provides the best compression rates of all programs compared.

Table 2 lists file sizes (in bits per pixel) obtained by running Glicbawls as well as several well-established colour compression programs on a number of photographic test images.[12]. While certainly not being competitive with state-of-the-art methods, Glicbawls consistently outperforms both Locoe and Pngcrush, two programs in common use today.

Table 3 lists file sizes for the so-called "artistic" (i.e. non-photographic) CLEGG, FRYMIRE and SERRANO images. As can clearly be seen, Glicbawls is not suitable for such images.

Quite surprisingly, given that it was designed as a compression program for continuous tone greyscale images, Glicbawls performs reasonably well on the eight CCITT sample fax pages as well.[13] The total set is compressed to 242.2kB, for a compression ratio of 16.56:1. This is ahead of the Group4 fax standard (15.5:1), but loses to IBM's Q-coder (19.0:1) and JBIG (19.7:1).

[8] As available from `http://www.csse.monash.edu.au/~bmeyer/glicbawls`

[9] As available from `ftp://ftp.csd.uwo.ca/pub/from_wu/`

[10] As available from `http://www.hpl.hp.com/loco/`

[11] The average for P2AR was estimated based on the available results

[12] Comparison values: `http://www.geocities.com/SiliconValley/Bay/1995/artest14.html`

[13] The binary images are treated as two-level greyscale images.

	lena	monarch	peppers	sail	tulips	Avg
	lossless colour compression					
BMF	12.28	8.21	9.19	10.02	9.31	9.80
Rkim	12.56	8.37	9.15	10.26	9.39	9.95
Locoe	13.60	11.29	11.75	15.61	12.54	12.96
Pngcrush	14.51	12.52	12.99	16.17	13.85	14.01
Glicbawls	**12.74**	**10.14**	**10.56**	**13.60**	**10.71**	**11.55**

Table 2: Colour compression results, in bits per pixel, for Glicbawls compared to a range of existing methods

	clegg	frymire	serrano	Avg
	lossless colour compression			
BMF	4.28	1.26	1.27	2.27
Rkim	10.43	3.95	3.26	5.88
Locoe	7.30	6.06	4.70	6.02
Pngcrush	5.41	1.63	1.71	2.92
Glicbawls	**15.08**	**12.98**	**11.65**	**13.24**

Table 3: Colour compression results for "artistic" images, in bits per pixel

9 Conclusion

We have presented an algorithm for lossless and near-lossless compression of greyscale images which consistently achieves higher compression ratios than CALIC while having a computational complexity low enough to be practical.

Including the source code of the decompressor with the compressed image allows the creation of self-extracting files, thus giving absolute upper bounds for the information contained in images.

While originally developed for greyscale images, the algorithm can also handle colour as well as bi-level images, achieving respectable compression on them.

References

[1] X. Wu and N. Memon, "Context-based, Adaptive, Lossless Image Codec", *IEEE Trans. on Communications*, vol. 45, no. 4, April 1997.

[2] X. Wu, K.U. Barthel and W. Zhang, "Piecewise 2D Autoregression for Predictive Image Coding", *International Conference on Image Processing conference proceedings*, Vol 3, 1998

[3] M. J. Weinberger, J. J. Rissanen and R. B. Arps, "Applications of universal context modelling to lossless compression of gray-scale images," *IEEE Trans. Image Processing*, 5, 1996, 575-586.

[4] M. Weinberger, G. Seroussi and G. Sapiro, "LOCO-I: A Low Complexity, Context-Based, Lossless Image Compression Algorithm", *Proceedings IEEE Data Compression Conference, Snowbird, Utah*, March-April 1996

[5] B. Meyer and P.E. Tischer, "TMW — a New Method for Lossless Image Compression", *International Picture Coding Symposium PCS97 conference proceedings*, September 1997

[6] I.N. Bronstein, K.A. Semendjajew, G. Musiol and H. Mühlig, "Taschenbuch der Mathematik, 2. Auflage", *Verlag Harri Deutsch*, Frankfurt am Main, 1995