Aritmetic Compression of Weighted Finite Automata

Jarkko Kari [*]

Academy of Finland

Pasi Fränti

Computer Science Department

University of Turku

20500 Turku, Finland

Abstract

Karel Culik and the first author have demonstrated how Weighted Finite Automata (WFA) provide a strong tool for image compression [1, 2, 3, 4]. In the present article we introduce an improved method for the last step of the compression algorithm: for compressing the WFA that approximates the given image. Our method is based on arithmetic compression of sparse matrices.

1 Introduction

The image compression based on Weighted Finite Automata (WFA) is a relatively efficient fractal compression method that gives good compression results. The principal idea is to infer a WFA A that represent a good approximation of the image to be compressed, and to remember A instead of the image. Inference algorithms for finding a suitable WFA have been discussed extensively in [1, 2, 3, 4], but the the important last step of expressing the WFA as a bitstring has been mostly ignored.

In the present article we introduce a way of writing a WFA in a compressed form that improves the compression results reported in [1, 2, 3, 4]. The method is based on arithmetic coding with three adaptive models for different parts of the WFA. First we give a short description of WFA and the recursive inference algorithm. Then we describe the arithmetic encoding of the WFA. Finally, examples and compression results are presented.

2 Preliminaries

Weighted Finite Automata (WFA) are finite automata with each edge labeled besides an input symbol also by a real number. Initial and final states are replaced by initial and final distributions that give for each state a real number. A WFA A over alphabet Σ with state set $Q = \{1, 2, \ldots, n\}$ is represented compactly by $|\Sigma|$ transition matrices W_a, $a \in \Sigma$, of size $n \times n$, a row vector I of size $1 \times n$ and a column vector F of size $n \times 1$. For all $i, j \in Q$, $a \in \Sigma$, the element (i, j) of W_a is the weight of the transition from state i to state j with input symbol a. The initial and final distribution values of state i are the i'th elements of I and F, respectively.

[*]The address of the first author is: Mathematics Department, University of Turku, 20500 Turku, Finland

The WFA A defines a function $f_A : \Sigma^* \to I\!\!R$ by

$$f_A(a_1a_2\dots a_k) = IW_{a_1}W_{a_2}\dots W_{a_k}F,$$

where ordinary matrix products are used. Equivalently, f_A can be determined as follows: Define for each state $i \in Q$ a function $\psi_i : \Sigma^* \to I\!\!R$ recusively by

$$\begin{aligned} \psi_i(\varepsilon) &= F_i, \text{ and} && (1)\\ \psi_i(aw) &= (W_a)_{i,1}\psi_1(w) + (W_a)_{i,2}\psi_1(w) + \dots + (W_a)_{i,n}\psi_n(w), && (2) \end{aligned}$$

for all $w \in \Sigma^*, a \in \Sigma$. Then

$$f_A(w) = I_1\psi_1(w) + I_1\psi_2(w) + \dots + I_1\psi_n(w). \qquad (3)$$

If the set $\{f : \Sigma^* \to I\!\!R\}$ of functions is taken as a linear space where addition and multiplication by a constant are defined in the natural way pointwise, then (2) can be written as

$$(\psi_i)_a = (W_a)_{i,1}\psi_1 + (W_a)_{i,2}\psi_1 + \dots + (W_a)_{i,n}\psi_n, \qquad (2')$$

where $(\psi_i)_a$ is the function $(\psi_i)_a(w) = \psi_i(aw), \forall w \in \Sigma^*$. Similarly (3) becomes

$$f_A = I_1\psi_1 + I_1\psi_2 + \dots + I_1\psi_n. \qquad (3')$$

In other words, the initial distribution gives the coefficients in the linear expression of f_A using functions ψ_i, and the i'th row of transition matrix W_a provides the coefficients in a similar linear expression for $(\psi_i)_a$. Clearly (1), (2') and (3') uniquely define f_A.

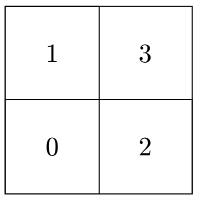

Figure 1: The addresses of quadrants

Let $\Sigma = \{0, 1, 2, 3\}$. A correspondence between functions $\Sigma^* \to I\!\!R$ and grey-tone images is obtained as follows. Words over Σ are understood as addresses of subsquares of the unit square $[0, 1] \times [0, 1]$: Each letter of Σ refers to one quadrant of a square as shown in Fig. 1. We assign ε as the address of the root of the quadtree representing an image, i.e. it refers to the whole unit square. Each letter of Σ is the address of a child of the root, i.e. a quadrant of the unit square. Every word in $\Sigma^\star$ of length k, say w, is then an address of a unique node of the quadtree at depth k, i.e. an address of a subsquare of size $2^{-k} \times 2^{-k}$. The children of this node have addresses $w0, w1, w2$ and $w3$. For example, the squares addressed by words of length three are shown in Figure 2.

A function $f : \Sigma^* \to I\!\!R$ defines *a multiresolution image*, that is, an image in every resolution $2^k \times 2^k$: The grey-tone intensity of the pixel with address $w \in \Sigma^k$ is $f(w)$. Typically the values of f are supposed to be within the interval $[0, 1]$, in which case 0 is interpreted as 'white', 1 as 'black' and intermediate values as intermediate intensities, but other interpretations are also possible. The different resolutions are compatible if the multiresolution image f is *average preserving*:

$$f(w) = \frac{1}{4}[f(w0) + f(w1) + f(w2) + f(w3)]$$

111	113	131	133	311	313	331	333
110	112	130	132	310	312	330	332
101	103	121	123	301	303	321	323
100	102	120	122	300	302	320	322
011	013	031	033	211	213	231	233
010	012	030	032	210	212	230	232
001	003	021	023	201	203	221	223
000	002	020	022	200	202	220	222

Figure 2: The addresses of subsquares in resolution 8×8.

for each $w \in \Sigma^\star$. The function f_A computed by WFA A is average preserving if

$$(W_0 + W_1 + W_2 + W_3)F = 4F,$$

that is, if the final distribution F is an eigenvector of $W_0 + W_1 + W_2 + W_3$ corresponding to eigenvalue 4.

With this interpretation of alphabet $\Sigma = \{0, 1, 2, 3\}$ the meaning of (2') becomes obvious: It states that the quadrant a of the image ψ_i is the linear combination of images $\psi_1, \psi_2, \ldots, \psi_n$ with coefficients given by the i'th row of matrix W_a.

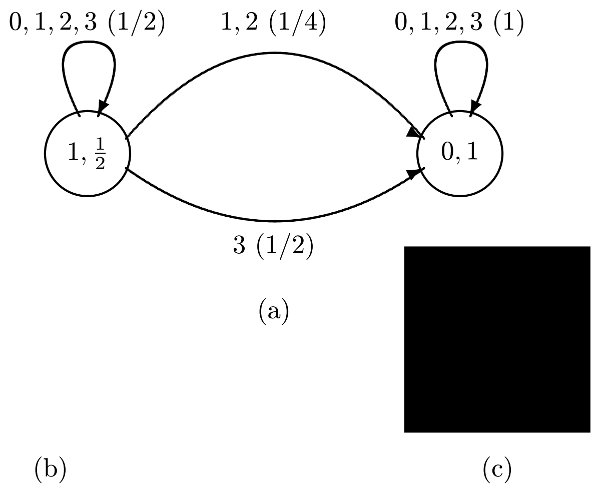

Figure 3: (a) WFA A defining the linear grayness function f_A, (b) $f_A = \psi_1$, (c) ψ_2.

Example 1: Consider the WFA A shown in Fig. 3(a). The transitions are labeled with symbols of Σ and weights, which are inside parentheses. The initial and final distribution values of the states are shown inside the nodes. The images ψ_1 and ψ_2 are shown in Fig. 3(b) and (c), respectively. Since the initial distribution is $(1, 0)$, $f_A = \psi_1$. Fig. 4 shows the linear expressions indicated by the outgoing transitions from the first state. The outgoing transitions from the second state simply indicate that all four quadrants of ψ_2 are the same image ψ_2.

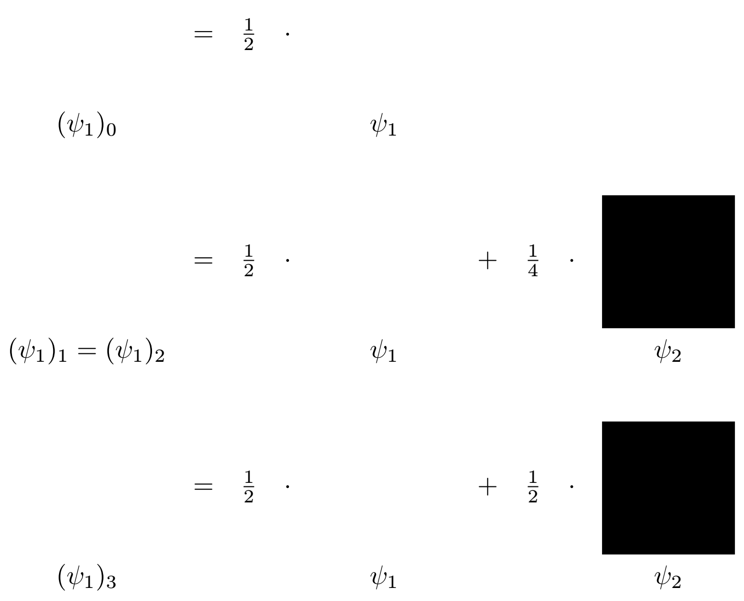

Figure 4: Linear expressions defined by the outgoing transitions from state 1 in Fig. 3.

3 A scetch of the recursive inference algorithm

The inference problem for WFA means finding a WFA that approximates well a given grey-tone image. Based on the discussion of the previous section this can be rephrased as follows: Try to find multiresolution images $\psi_1, \psi_2, \ldots, \psi_n$ such that

(i) all four quadrants of all ψ_i can be expressed as linear combinations of $\psi_1, \psi_2, \ldots, \psi_n$,

(ii) a good approximation of image f can be expressed as a linear combination of $\psi_1, \psi_2, \ldots, \psi_n$.

The coefficients of (i) and (ii) define the transition matrices and the initial distribution of the WFA, respectively, while the final distribution is given by the average intensities of images ψ_i, $1 \leq i \leq n$.

Algorithm 1 is a scetch of our implementation of the recursive inference algorithm discussed in [2, 3, 4]. See those references for more details. Basically the algorithm processes all four quadrants of an image

(a) by trying to express the quadrant as a linear combination of existing images ψ_i and

(b) by choosing the quadrant as a new image that is recursively processed.

The alternative that yields better result is chosen. The alternative (a) corresponds to adding new edges that express the coefficients of the linear combination, while (b) means adding a new state for the quadrant and just one transition with weight 1. The alternative is chosen that gives smaller value of

$$\textit{cost} = \textit{error} + G \cdot \textit{size},$$

where G is a real number parameter given to the algorithm, *error* is the square difference between the quadrant and its approximation (by square difference we mean the sum of the squares of the differencies in the pixel values in the two images), and *size* is the number of bits required to store the the new edges and states.

Global variables:

n	:	number of states in the automaton,
ψ_i	:	image of state i, $1 \le i \le n$,
weight$[i][a][j]$	:	the weight of the transition from state i to state j with label a,
child$[i][a]$	:	number j such that $\psi_j = (\psi_i)_a$, if such j exists, 0 otherwise. (Indicates what choice was done for quadrant a of state i in function *build*.)

Functions $d_k(f,g) = \sum_{w\in\Sigma^k} (f(w) - g(w))^2$ compute the square distance between images f and g in resolution $2^k \times 2^k$.

Initially, $n =$ the numer of elements in the initial basis, $\psi_1, \psi_2, \ldots, \psi_n$ are the images in the basis, and values of *weight* and *child* are initiated to 0.

The WFA for image $f : \Sigma^k \to I\!R$ of resolution $2^k \times 2^k$ is constructed by calling *build*(f, k, ∞).

```
float build($\psi, k, min$)

/* Approximates image $\psi : \Sigma^k \to I\!R$ of resolution $2^k \times 2^k$ by a WFA. In the end of the routine the
image $\psi_n$ of the last state is an approximation of $\psi$ such that the value of $cost = error + G\cdot size$ is
minimized, provided $cost < min$. In this case the routine returns $cost$. Otherwise (if $min$ could not
be improved) value $\infty$ is returned. The function uses local variables $s[a]$ and $t[i][a]$ to remember the
chosen values of child and weight until the end of the routine, that is, until the number of the new
state approximating $\psi$ is known. */

If $min \le 0$ or $k = 0$ then return($\infty$);
$cost \leftarrow 0$;
do steps 1–5 with $\varphi = \psi_a$ for all $a \in \Sigma$:

    1. Find $r_1, r_2, \ldots r_n$ such that the value of

           $cost1 \leftarrow d_{k-1}(\varphi, r_1\psi_1 + \ldots + r_n\psi_n) + G \cdot size1$

       is small, where $size1$ denotes the increase (in bits) in the size of the automaton caused by
       adding edges to states $1, 2, \ldots, n$ with label $a$ and weights $r_1, r_2, \ldots, r_n$;

    2. $n_0 \leftarrow n$;

    3. $cost2 \leftarrow G \cdot size2 + build(\varphi, k-1, min\{min - cost, cost1\} - G\cdot size2)$, where $size2$ is the increase
       in the size of the WFA caused by an edge with weight 1 to a new state;

    4. If $cost2 \le cost1$ then $cost \leftarrow cost + cost2$, $s[a] \leftarrow n$, $t[n][a] \leftarrow 1$;

    5. If $cost1 < cost2$ then $cost \leftarrow cost + cost1$, $n \leftarrow n_0$, $s[a] \leftarrow 0$, $t[i][a] \leftarrow r_i, \forall i, 1 \le i \le n_0$;

If $cost > min$ return($\infty$);
$n \leftarrow n + 1$;
For all $a \in \Sigma$ set $child[n][a] \leftarrow s[a]$;
For all $a \in \Sigma$ and $i = 1, 2, \ldots, n-1$ set $weight[n][a][i] \leftarrow t[i][a]$;
$\psi_n \leftarrow$ the multiresolution image defined to state $n$;
return($cost$);
```

Algorithm 1: Outline of the recursive inference algorithm for WFA.

Parameter G controlls the quality of the approximation and the compression rate. One bit increase in the size of the compressed file is allowed if the improvement it provides in the image quality is at least G. The algorithm thus chooses from the square error vs. file size -graph the point where the derivative is $-G$.

The Algorithm 1 is slightly different from the algorithm described in [2, 3, 4]. The main difference is that the new state is added only *after* its quadrants are processed. Therefore its image is not available for the linear combinations in its subquadrants. This prevents loops in the automaton. This also means that before first call to function *build*, the WFA has to be initiated to a fixed initial basis. See [4] for more details on this point.

After Algorithm 1 has been executed, the transition matrices of the WFA can be read from the variable *weight*. Element (i, j) of W_a is *weight*$[i][a][j]$. Images ψ_i of the states (except the states of the initial basis) are approximations of some subsquares of the original image. Variable *child* describes the relation between the states and the quadtree representation of the image: If state i approximates the subsquare with address w, then state *child*$[i][a]$ approximates the subsquare with address wa, provided *child*$[i][a] \neq 0$. If *child*$[i][a] = 0$, the quadrant a was expressed as a linear combination of existing states, so there is no state in the WFA for subsquare wa.

4 Compressing the WFA

Algorithm 1 produces a WFA A that still has to be written as a bitstring. The initial distribution of the WFA A will always be of the simple form $(0, 0, \ldots, 0, 1)$ since the image ψ_n of the state added in the end of the outermost recursive call of *build*, i.e. the last state of A, is the approximation of the original image. The final distribution, on the other hand, is uniquely determined by the transitions and the initial basis: Since there are no loops, the transitions tell how each ψ_i is eventually build up from the images of the initial basis, and the average intensity of each state (= the final distribution) can be computed. Therefore, initial and final distributions do not need to be stored — it is sufficient to remember the four transition matrices.

The transition matrices can be stored in a variety of ways. The only requirement is that during the execution of Algorithm 1 we have to be able to compute (or at least estimate well) the increase in the size of the compressed WFA caused by new edges and states on steps 1 and 3. The algorithm is build in such a way that it takes into account how the WFA is stored in the end, and it attempts to produce WFA that are well suited for that particular method.

Our choice is to use arithmetic coding with adaptive models. A *model* is a method for calculating the probability distribution for the next symbol to be encoded. The model gets as input the *context* of the symbol, i.e. some information computable without knowing the symbol. Arithmetic encoder get as input the probability distribution and the actual next symbol. The decoder uses the same model to compute the same probability distribution for the same context. Based on this information it can decode from the compressed string the actual encoded symbol. The model is called *adaptive* if the probability distribution it provides depends on the history of symbols already encoded. Consult reference [6] for more details.

In our application the weights will be encoded separately from their positions in the matrices. In otherwords we store bitmatrices that indicate the non-zero elements of the transition matrices. The edges added on step 4 of Algorithm 1 are special: their weight is always one. They are stored by encoding the variable *child*. So we use three models: one for variable *child*, one for the bitmatrix of other edges, and one for the weights. Since adaptive models are used, in order to allow the costs be estimated correctly on steps 1 and 3, the encoding has to be done in the same order as the edges are added during the execution

External routines *encode_child*, *encode_bitmatrix* and *encode_weight* that take care of the arithmetic encoding and updating the models are used. Routine *initiate_column* initiates the probabilities of 0 and 1 in an unused context of the bitmatrix model.

Before calling *encode* initiate $n \leftarrow$ the numer of elements in the initial basis.

The encoding of the WFA produced by Algorithm 1 is done by calling *encode*(m, k), where m is the number of states in the WFA, and the size of the original image is $2^k \times 2^k$.

```
encode(i, k)

/* Encodes the subtree of the quadtree rooted at state i. State i represents a subimage in resolution
2^k × 2^k. */

For all $a \in \Sigma$ do
    If $child[i][a] \neq 0$
    {
      $encode\_child(1, k)$ ;
      $encode(child[i][a], k-1)$;
    }
    Else
    {
      $encode\_child(0, k)$ ;
      For all $j = 1, 2, \ldots, n$ do
        If $weight[i][a][j] = 0$ $encode\_bitmatrix(0, j)$ ;
        Else
        {
          $encode\_bitmatrix(1, j)$ ;
          $encode\_weight(weight[i][a][j], k)$ ;
        }
    }

$n \leftarrow n+1$;   /* = i */
$initiate\_column(n)$.
```

Algorithm 2: Outline of a recursive algorithm for encoding a WFA produced by Algorithm 1.

of Algorithm 1.

Algorithm 2 is a scetch of the encoding algorithm. In the following we exlplain in more details the algorithm and the models that are used. Algorithm 3 is a scetch of the corresponding decoding algorithm.

4.1 The model for variable *child*

Four bits are encoded for each state i: For every $a \in \Sigma$ we encode 1 if $child[i][a] \neq 0$ and 0 otherwise. This uniquely determines the underlying quadtree structure. Note that the actual value of $child[i][a]$ can be deduced by numbering the nodes in the same depth first order as Algorithm 1 traverses the tree.

The depth of the state in the tree is used as context. The idea is that the probability of bit 1 is greater

External routines *decode_child*, *decode_bitmatrix* and *decode_weight* take care of the arithmetic decoding and updating the models. Routine *initiate_column* is used to initiate the probabilities of 0 and 1 in a new context of the bitmatrix model.

Before calling *decode* initiate $n \leftarrow$ the numer of elements in the initial basis.

A WFA produced by by Algorithm 1 is decoded by calling $decode(k)$, if the size of the original image is $2^k \times 2^k$.

```
decode(k)

/* Decodes a subtree whose root represents an image at resolution $2^k \times 2^k$. Local variables $s[a]$ and $t[j][a]$ are used to remember the values of child and weight until the end of the routine. */

For all $a \in \Sigma$ do
    If $decode\_child(k) = 1$  $decode(k-1)$; $s[a] = n$;
    Else
    {
      $s[a] = 0$;
      For all $j = 1, 2, \ldots, n$ do
        If $decode\_bitmatrix(j) = 1$ then $t[j][a] \leftarrow decode\_weight(k)$;
        Else $t[j][a] \leftarrow 0$;
    }

$n \leftarrow n + 1$;
For all $a \in \Sigma$ set $child[n][a] \leftarrow s[a]$;
For all $a \in \Sigma$
    If $s[a] = 0$ then, for all $j = 1, 2, \ldots, n-1$ set $weight[n][a][j] \leftarrow t[j][a]$;
    Else $weight[n][a][s[a]] \leftarrow 1$;
initiate_column(n).
```

Algorithm 3: Outline of a recursive algorithm for decoding a WFA encoded by Algorithm 2.

for the nodes close to the root than for those closer to the leaves. In particular, a node is a leaf iff all four bits are 0's.

Initially, in every context the probabilities of both bits are set to 1/2. After encoding x 0's and y 1's at the particular depth, the probabilities are updated to $(x+1)/(x+y+2)$ and $(y+1)/(x+y+2)$ for 0 and 1, respectively. The same probabilities should be used in Algorithm 1 for calculating *size1* and *size2* on steps 1 and 3:

$$\begin{aligned} \textit{size1} &= -\log_2\left(\frac{x+1}{x+y+2}\right) + \textit{size1'}, \text{ and} \\ \textit{size2} &= -\log_2\left(\frac{y+1}{x+y+2}\right), \end{aligned}$$

where *size1'* is the increase in the size of the automaton due to the new edges with weights $r_1, r_2, \ldots, r_n$.

Bit b at depth k is encoded by routine *encode_child*(b,k) that (i) encodes the bit using arithmetic coding and (ii) updates the model for context k. The corresponding routine for decoding is *decode_child*(k) that returns the encoded bit and updates the model.

If *child*$[i][a] \neq 0$, nothing else than bit 1 needs to be stored for quadrant a. If *child*$[i][a] = 0$, the weights *weight*$[i][a][j]$, $1 \leq j \leq n$ found on step 1 have to be encoded in addition to the bit 0. For each j a bit indicating whether *weight*$[i][a][j] = 0$ is encoded using the model of subsection 4.2. If *weight*$[i][a][j] \neq 0$ the weight is stored using the method of subsection **??**.

4.2 The model for the bitmatrices

A bit indicating whether *weight*$[i][a][j] = 0$ is encoded using the column j as the context. This is based on the observation that some images are more frequently used in linear combinations than others. It is sufficient to encode a bit for the columns that correspond to states that existed when step 1 of Algorithm 1 was executed. Others correspond to states that were created later and were therefore not yet available.

A new context is initiated when a new state becomes available. The probabilities of bits 0 and 1 are then initiated to p_0 and p_1 according to their occurences in the bit matrices (in all contexts) so far. Later, after encoding x 0's and y 1's in the context, the probabilities are updated to $(x+p_0)/(x+y+1)$ and $(y+p_1)/(x+y+1)$ for 0 and 1, respectively.

Bits are encoded by routine *encode_bitmatrix*(b,j) that also updates the model for context j. The corresponding routine for decoding is *decode_bitmatrix*(j).

4.3 Encoding the weights

Finally non-zero weights $r_j =$*weight*$[i][a][j]$ have to be encoded. Consider their precision: Assume p bits after the binary point are stored. The precision should be increased by one bit if the image quality gets improved by at least G. The improvement is at most $4^{p+1} \cdot d_{k-1}(\psi_j, 0)$, and on the average

$$\ldots$$

where

$$d_{k-1}(\psi_j, 0) = \sum_{w \in \Sigma^{k-1}} \psi_j(w)^2.$$

Therefore the precision should be increased if

$$\frac{d_{k-1}(\psi_j, 0)}{3 \cdot 4^{p+1}} - \frac{d_{k-1}(\psi_j, 0)}{3 \cdot 4^{p+2}} < G,$$

that is, if

$$p < \log_4 d_{k-1}(\psi_j, 0) + \log_4 \frac{1}{G} - 2.$$

Instead of weight r_j, normalized weight

$$r'_j = nor_{k-1}[j] \cdot r_j$$

is stored, where

$$nor_{k-1}[j] = \sqrt{\frac{d_{k-1}(\psi_j, 0)}{4^{k-1}}}$$

is the square root of the mean square distance of ψ_j from zero in resolution $4^{k-1} \times 4^{k-1}$. For r'_j precision

$$p = k - 2 + \left\lfloor \log_4 \frac{1}{G} \right\rfloor$$

is used. Advantages of using the normalized weight are the uniformity of the precisions (the same number of bits at the same resolution) and better distribution of the weights.

For encoding the weights, an interval $I = [x, x + m \cdot 2^{-s}]$ is divided into m subintervals of length 2^{-s}. First, information concerning which subinterval contains r'_j is encoded using arithmetic encoding (here $(-\infty, x)$ and (y, ∞) are possible intervals). If $r'_j \in I$, the last $p - s$ bits are written as such. If $r'_j < x$ (or $r'_j > y$), the positive real number $r = r'_j - x$ (or $r = y - r'_j$, respectively) is stored as follows: $m = \lfloor \log_2(r + 1) \rfloor$ is written in unary (this requires $m + 1$ bits), and $r + 1 - 2^m$ is written as such ($m + p$ bits).

Weights r'_j are encoded in precision p by routine *encode_weights*(r'_j, p) that also updates the model. The corresponding routine for decoding is *decode_weights*(p).

5 Results

Our set of test images consists of four well-known 8-bit gray scale images of resolution 512 × 512 (lena, airplane), or 256 × 256 (bridge, camera). Lena is the green component of the original rgb-image lena.

Recommendations for the compression algorithm was set up by the JPEG (*Joint Photographic Experts Group*) working group. The bit rate requirement for "useful" image quality was set to 0.25 bits per pixel, and for 'recognizable" image quality to 0.083 bits per pixel [5]. We try to meet the first requirement.

		WFA		**W+WFA**		**JPEG**	
G	**Image**	**bpp**	**mse**	**bpp**	**mse**	**bpp**	**mse**
0.010	Lena	0.20	70.90	0.19	70.83	0.20	113.91
	Airplane	0.23	71.10	0.20	64.67	0.21	117.06
	Bridge	0.33	84.15	0.30	76.40	0.30	94.42
	Camera						
0.005	Lena	0.29	51.87	0.28	46.54	0.28	66.57
	Airplane	0.35	45.12	0.29	40.87	0.29	62.68
	Bridge	0.52	40.55	0.46	37.45	0.45	55.95
	Camera						

Table 1: Test results for several well-known images.

Table 1 contains the test results for these images when compressed by the recursive inference algorithm (WFA), by the recursive inference algorithm applied to the Mallat form of the W6 wavelet transform (W+WFA, see [2]), and by JPEG. The inference algorithm was applied with parameter values $G = 0.01$ and $G = 0.005$. For each case the bit rate of JPEG was set to match as closely as possible with the ones given by the W+WFA method. The results of both WFA and W+WFA clearly outperforms the results of JPEG. The compression results for Lena and $G = 0.005$ are illustrated in Fig. 5. The degderations of the methods are however better visualized in Fig. 6, where magnifications from Lena are shown.

Figure 5: Test image Lena a) original, b) compressed by JPEG, c) by WFA, d) by W+WFA.

The drawback of the WFA inference algorithm is its high compression time. The higher the automaton size (and thus the image quality) the higher the running time. However, the decoding remains relatively fast. This is an important property e.g. for an image archieving system where the images are stored only once, but retrieved often. The compression and decompression times with *Sun SparcServer 690M* are given in Table 2.

Finally, Table 3 contains more details about the encoding results of the WFA. For all four WFA for Lena, we have listed the numbers of bits required to encode the three different parts of the automaton: the undelying quadtree structure (i.e. variable *child*), the bitmatrices and the weights. The numbers reported are the base 2 logarithms of the probabilities provided by the models to the arithmetic encoder. Also the numbers of states and edges in the WFA are listed. The numbers do not include the initial basis which does not need to be stored. The number of edges contains only the edges whose weights are stored — the edges with weight 1 forming the quadtree structure are excluded.

Figure 6: Magnifications of Lena.

		WFA		**W+WFA**	
G	**Image**	**comp.**	**decomp.**	**comp.**	**decomp.**
0.010	Lena	412.9	10.5	518.1	22.0
	Airplane	598.9	15.2	547.2	25.7
	Bridge	1480.5	64.5	1078.7	44.4
	Camera				
0.005	Lena	614.6	13.9	981.0	36.0
	Airplane	1066.9	23.9	964.6	33.2
	Bridge	3101.5	149.0	2341.0	103.2
	Camera				

Table 2: Compression/decompression times.

On the average, a little more than two bits per state are needed for the quadtree — a clear improvement to the trivial four bits per state. The numbers of bits per encoded weight varies between 4.0 and 5.9. For parameter value $G = 0.005$ one bit higher precision is used for the weights than in case $G = 0.01$. Typically the combination with wavelets produces more states but fewer edges, therefore the bitmatrices occupy a greater share of the compressed file.

G	type	bits in quadtree	bits in bitmatrices	bits in weights	states	edges
0.010	WFA	1 088	25 850	25 072	477	4 843
	W+WFA	1 874	32 624	14 431	831	3 534
0.005	WFA	1 368	40 090	34 962	601	5 948
	W+WFA	2 326	48 144	23 087	1 145	4 972

Table 3: Entropies for the different parts of the WFA for Lena, and numbers of states and edges.

References

[1] K. Culik II and J. Kari, Image Compression Using Weighted Finite Automata, *Computer and Graphics* vol.17 (3), 305-313 (1993).

[2] K. Culik II and J. Kari, Image-data Compression Using Edge-Optimizing Algorithm for WFA Inference, *Journal of Information Processing and Management*, to appear.

[3] K. Culik II and J. Kari, Inference Algorithm for WFA and Image Compression, in: *Fractal Image, Encoding and Compression*, ed. Y.Fisher, Springer-Verlag, to appear.

[4] K. Culik II and J. Kari, A Recursive Algorithm for Image Compression with Finite Automata, *DCC'94*, submitted.

[5] W.B. Pennebaker, J.L. Mitchell, JPEG Still Image Data Compression Standard, Van Nostrand Reinhold, 1993.

[6] I. Witten, R. Neal, J. Clearly, Arithmetic Coding for Data Compression, *Comm. ACM* vol. 30 (6), 520–539 (1987).