Image Compression with Function Trees

S. Battiato, G. Gallo, S. Nicotra
{battiato, gallo, snicotra}@dmi.unict.it
Department of Mathematics and Computer Science
Viale Andrea Doria, 6 – 95125 Catania
Italy

ABSTRACT

A new compact scheme, which outperforms in many cases quad-tree decomposition, for coding of digital images is proposed and analyzed. It is based on a suitable generalization of *function-trees*, introduced in [1]. In particular a new top-down approach to build function-trees both for binary pictures and gray level images is presented. The proposed coding scheme can be adapted to compress digital images both for lossless and lossy compression and it has useful application into fractal image compression.

KEYWORDS: Quad-tree, Image Compression, Fractal Compression.

1. INTRODUCTION

The quad-trees are powerful and simple data structures for representing and/or compressing digital images [2][3][4]. It is possible to find applications of quad-tree decomposition in many different contexts, such as the compression of sub-band coefficients in wavelet decomposition and coding of the sub-blocks data in EBCOT algorithm [5][6]. The quad-tree decomposition is an important tool for fractal image compression where many suitable smart variants of quad-trees are used and applied (see [7] for a detailed survey). *Function-tree* representation, introduced in [1], is another possible improvement of the original scheme proposed by Samet [3]. The main innovation of *function-trees* concerns the introduction into the node decomposition of classical quad-trees of suitable new nodes representing sub-squares whose black pixels form *trapezoidal regions*, i.e. regions that may be described as the area below the graph of a discrete function. Using such information it is possible to obtain some improvement in terms of compression size due to the fact that *trapezoidal regions* can be efficiently encoded with boundary representation. A typical *trapezoidal region* is shown in Fig. 1 together with its boundary representation. In this paper the original proposal of *function-trees* is enriched and used to derive a compression scheme able to outperform in many cases the results obtained with classical quad-tree decomposition, both for binary and gray level images.

The paper is structured as follows. In Section 2, after a brief review of the *function-tree* decomposition, a top down version of the algorithm is presented and compared to the bottom-up technique originally proposed by [1]. The next Section is entirely devoted to the description of an original compression scheme, applied first to black/white images and then generalized to gray level images. In Section 4 a comparative analysis of the method proposed with the classical quad-tree decomposition is presented together with some experimental results. A final Section closes the paper tracking directions for future related works and applications [8][9][10][11][12][13].

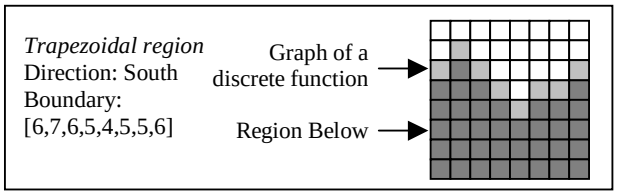

Figure 1. *An example of Trapezoidal Region and its boundary representation.*

2. BUILDING FUNCTION TREES

Function-trees are connected with the key concept of quad-tree decomposition. In particular a *function-tree* can be defined as a quad-tree with four possible kinds of sub-nodes: each node is used to code different areas of the input image depending on their content.

More precisely let us suppose to have as input a binary image *I*, of *n x n* pixels, and suppose for simplicity sake that $n=2^k$, for some integer *k*. The nodes in a *function-tree* of the image *I*, at level *t*, correspond to a $2^{k-t} \times 2^{k-t}$ sub-image of the related sub-square decomposition. Nodes may be leaves or internal nodes. More precisely we have:

- a **leaf** if the relative sub-square is a:
  - a *black* region;
  - a *white* region;
  - a *trapezoidal* region*;*
- an **internal node** (*gray*) otherwise.

Nodes have to be represented by a suitable data structure, i.e. a record storing all the relevant information. The structure of a typical node, together with a rough estimate of its content in bit is schematically reported in Figure 2.

Color (*white, black, graph, gray*)				2 bit
Info (size, boundary, ...)				array of bits
Direction (N,S,W,E)				2 bit
First	Second	Third	Four	Pointers

Figure 2. *Scheme of the record relative to a node in a function-tree.*

Function-trees may be computed in the very same fashion than quad-trees. In short a typical algorithm recursively reads the input, creates a new node in the tree and checks if the corresponding sub-region is entirely *black*, *white* or a *trapezoidal region* (see below for more details). In these cases, the node is a *leaf* in the tree containing all the information useful in the sub-square decoding phase. For example if the sub-square relative to the node is entirely black, the relative node will contain the color of the region.

If a non-leaf node is encountered, the original input is split in four *n/2 x n/2* sub-images and the algorithm is recursively applied to each one of such regions. This produces in the tree, a *gray* node that is the parent of 4 ordered children built by the relative recursive calls. If the input size is a power of two, after at most $log_2 n$ recursive calls, the algorithm terminates.

The pseudo-code of the algorithm is described in Box 1.

```
procedure function_tree(binary_image)
1.  new_node(tree);
2.  if is_black(binary_image)
3.  then
        tree.color="black"
        return(tree);
4.  if is_white(binary_image)
5.  then
        tree.color="white"
        return(tree);
6.  if is_a_function(binary_image,boundary,dir)
7.  then
        tree.color='graph';
        tree.info=boundary;
        tree.direction=dir;
        return(tree);
8.  // otherwise divide binary_image in four sub
9.  // images and go recursively
10. split_in_four(binary_image,nw,ne,sw,se);
11. tree.first=function_tree(nw);
12. tree.second=function_tree(ne);
13. tree.third=function_tree(sw);
14. tree.fourth=function_tree(se);
15. end function_tree;
```

Box 1. *Pseudo-code of the function-tree algorithm.*

The overall complexity of the algorithm is related with the tests performed respectively on lines 2, 4, 6 and on the number of recursive calls. More precisely:

- function **is_black** on line 2, takes $O(n^2)$, when it is called on a *n*n* sub-square.
- function **is_white** on line 4, takes $O(n^2)$, when it is called on a n*n sub-square.
- function **is_a_function** on line 6 checks (for all direction) if each column of sub-square is suitable as a subset of a trapezoidal region; it tries to find sequences as 0*01*1 or 1*10*0 depending on the direction. Here we are adopting regular expression language.

It is possible to perform the test of "is_a_function" using different strategies and heuristics that are illustrated below. For reference sake suppose that we are trying to test if a sequence 000…111 has been encountered.

The most naive method is: scan the sequence from left to right until finding boundary flag "1", then check if the remaining sub-sequence is entirely made up of "1"'s.

There is also another interesting strategy based on properties of binary numbers. As a matter of fact sequences which represent "good columns", when regarded as binary representations of number, come from integers that may be obtained as $\Sigma^k_{i=0} 2^i$. It's well known that such quantities have the form of $2^{k+1} - 1$.

Let $y = (a_1,...,a_k)$ be a bit sequence , $x = \Sigma^k_{i=0} a_i * 2^i$, let $x' = x + 1$. If $log_2 (x')$ is an integer than y is a "good sequence".

For example: $y = (00111111)_2$, $x = 31 \rightarrow x' = 32$ and $log_2 (32) = 5$ is an integer.

Both techniques are synthetically described in Box 2.

```
procedure simple(col c)
n = length(c);
for i = 1 to n
    if c(i)==0
    i = i + 1;
    else
        while i < n
        if c(i) == 1
            return('no');
        end if
        end while
        return('yes')
    end if
end for
```

```
procedure binary(col c)
x = binary_to_integer(c);
y = log2( x+1 );
if is_an_integer( y )
    return( "yes" );
else return( "no" );
```

Box 2. *Alternative pseudo-codes to test if a bit sequence represents a "good column" (see text).*

Both techniques need $\theta(n)$ time in case of positive answer (all bits in sequence must be analyzed). Moreover an efficient implementation (software or hardware) of binary digits operations could effectively reduce complexity time.

Finally, time complexity of *function_tree* procedure can be described by the following recurrence:

$T(n^2) = n^2 + 4\ T(n^2/4) \rightarrow T(n^2) = O(n^2 \log n)$ by Master Theorem [14]. A suitable estimate of the complexity in space is obtained observing that the space used by *function-trees* is always less or equal to quad-tree. It is important to note that our approach builds a *function-tree* in a top-down way that is different than the one described in the original paper [1]. The new approach allows more proper data processing.

A typical example of *function-tree* relative to a digital image (i.e. "circle" image) is reported in Fig. 3. The figure contains also, for sake of comparison, the analogous quad-tree decomposition.

Some numerical results obtained applying the *function-trees* decomposition, over three different b/w images (shown in Fig. 3,4) are reported in Table 1.

Images	*Circle*	*Lena*	*Comb*
Size	16x16	64x64	128x128
Quad - tree			
Black	56	1329	942
White	20	985	865
Internal	25	771	602
Total	**101**	**3085**	**2409**
Function - tree			
Black	0	528	0
White	0	444	0
Graph	4	445	1
Internal	1	413	0
Total	**5**	**1830**	**1**

Table 1. *Comparisons of number of nodes for images in figure 3,4 for functions trees and quad-trees.*

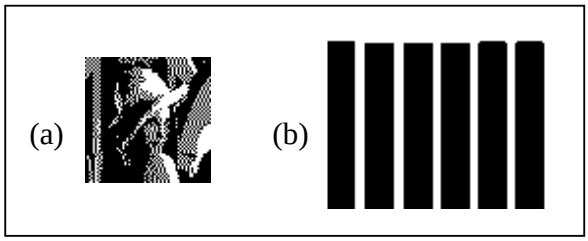

Figure 4. *(a) Lena and (b) Comb images*.

The saving obtained in terms of numbers of nodes is evident and allows to obtain significative compression ratio improvement.

3. A SIMPLE COMPRESSION SCHEME

Function-tree decomposition of digital images provides a tool for several efficient techniques of data processing. Several strategies and heuristics can be applied in order to obtain from *function-trees* an effective compression schemes. In this paper we present a simple compression scheme and compare it with an analogous scheme based on the classical quad-tree data structure. Although our results are somehow interesting we do not claim that the proposed scheme should be adopted exclusively for lossless compression: better performances may be obtained allowing lossy data management.

In their basic structure *function-trees* are very suitable to efficiently store binary images. In order to obtain a compression scheme for gray level pictures we suggest different solutions.

A first simple, but effective, method is based on the

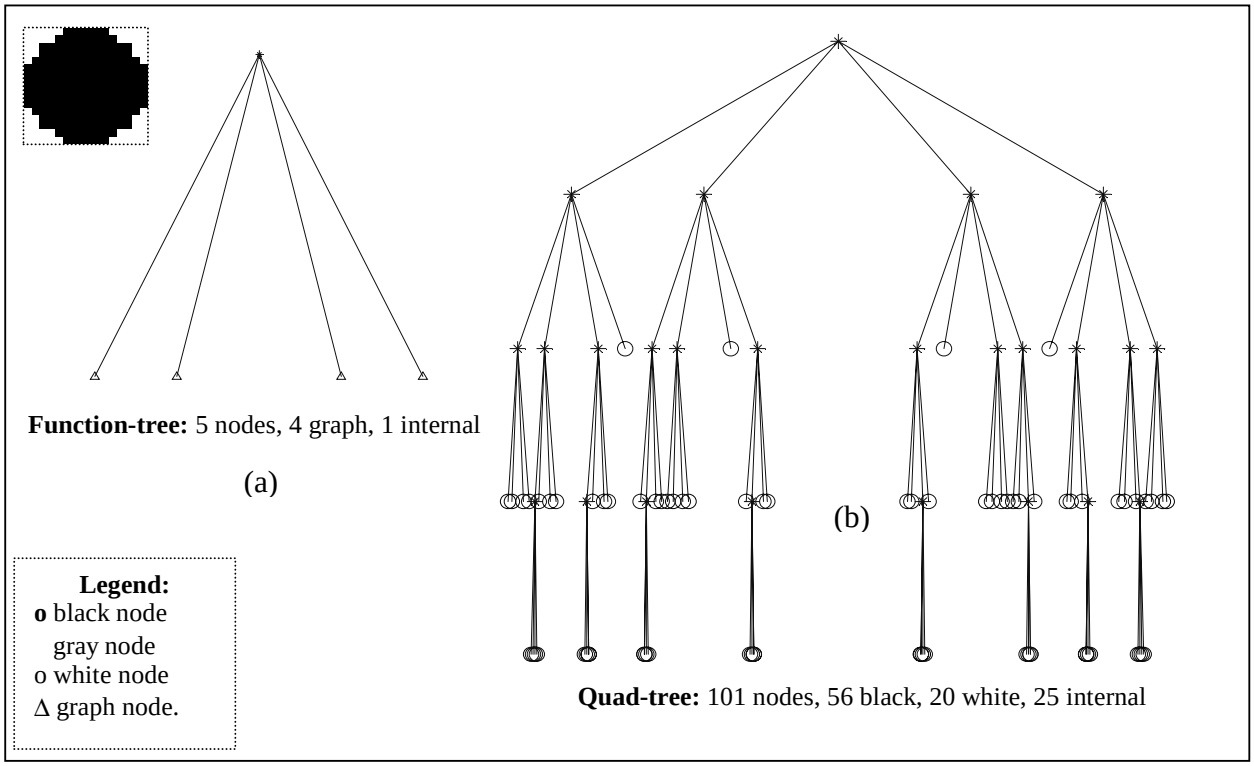

Figure 3. *(a) Function-tree and (b) quad-tree relative to a little black filled circle (16x16 pixels).*

well-known concept of bit planes. A typical gray levels (e.g. 8 bit, 256 levels) image may be split into 8 binary images (i.e. 8 bit planes). It becomes hence obvious to apply the previous algorithm to such planes. In this way an image is transformed into a forest of 8 *function-trees*. Encoding and decoding algorithms based on this idea have been implemented and tested on a large database of images to verify the real power of the proposed method. See Figure 5 for an illustrative typical example of the technique.

Although simple and effective, the algorithm presented above has some weaknesses, in term of compression performance when applied to real scenes.

Real scenes representation involves a large amount of middle to high frequencies contributions. Such contributions appear mainly in the least significative bit planes. In particular, the last four bit planes contain "pulviscular" b/w sets and require *function-trees* with an excessive number of nodes.

Better results have been obtained adopting gray code [2] instead of classical binary code. Gray codes are easily obtained by binary code and have the property that two consecutive decimal numbers have gray code representations that differ in only, and only one, bit. A typical application of gray-codes can be found in [2], where it is applied to reduce the number of gray levels in an image while minimizing the visual artifacts.

Adopting gray codes before bit planes decomposition we have obtained better results in terms of number of nodes. Even in this case, the least significative bit planes are too fragmented and although *function-trees* produce a space reduction with respect to binary version, results are not still satisfactory.

For these reasons in our experiments we adopted a suitable combination of bit-planes with classical entropy coding such (Run Length Encoding and Huffman Compression) [2].

The idea is to restrict *function-tree* encoding to the first four more significative bit planes and to process the other four bit planes separately and precisely with RLE followed by Huffman. In this case too gray code decomposition of bit planes performs better than binary. This is in agreement with other findings that show gray code decomposition improves RLE [2].

It is possible to generalize the method to colors images simply applying the method to each one of the three corresponding color planes. Alternatively it is possible to preliminary convert, the input image in a YUV-like color space, applying the technique to the full luminance plane (e.g. the Y-plane) but subsampling the chrominance planes in the classical way (e.g. 4:2:2 or 4:2:0 format [15]).

The next section shows in detail the results obtained with the proposed method compared with the results obtained with classical loss-less quad-tree decomposition.

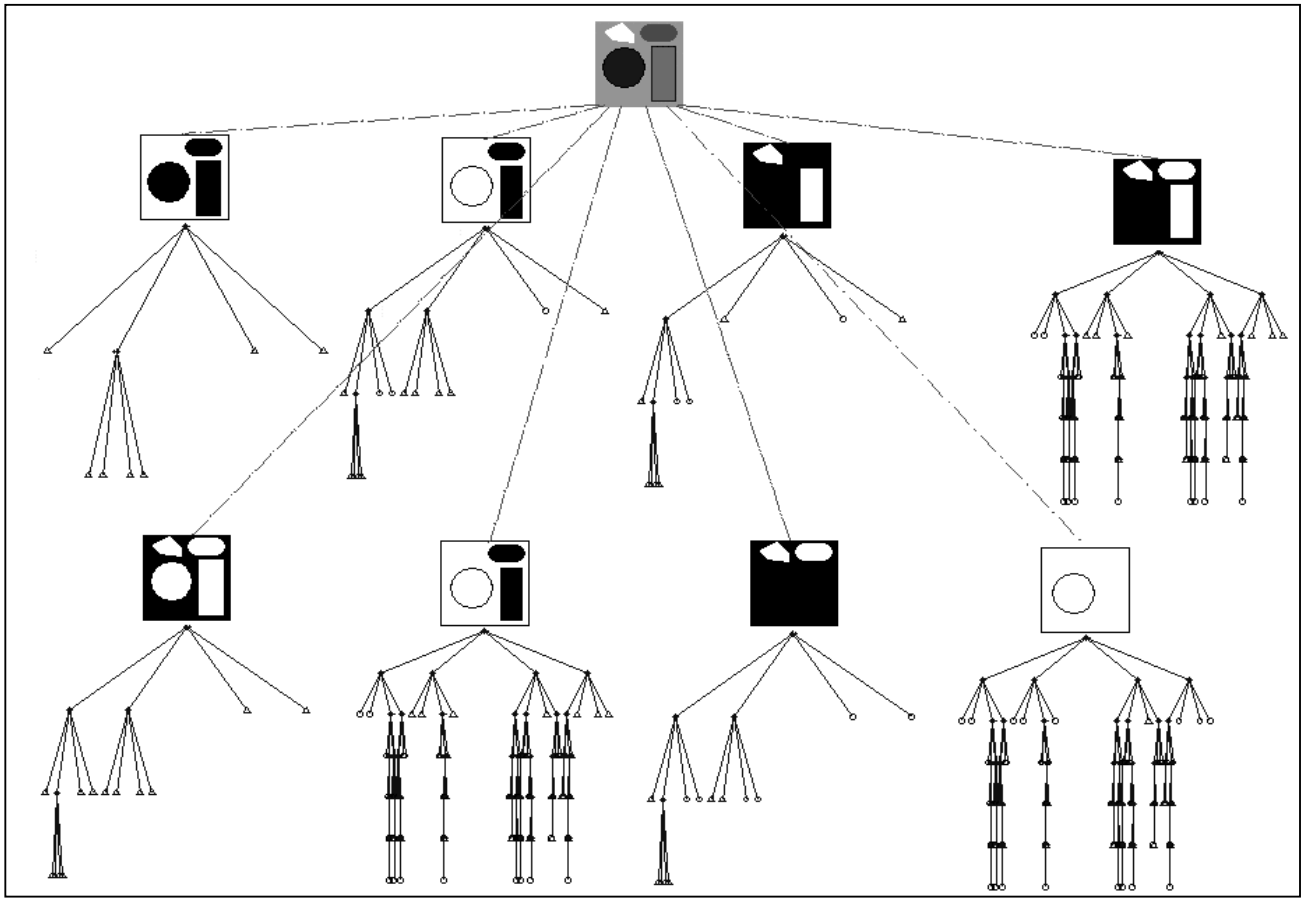

Figure 5. *Function-trees forest relative to eight bit-planes of a synthetic gray-level image.*

4. EXPERIMENTAL RESULTS

All the techniques described in the previous Section have been implemented and tested over different kinds of images (real, synthetic, textures).

As a benchmark measure we choose the number of nodes in the structure and their space requirements both for *function-trees* and quad-trees. In this way we are able to provide comparisons that are not dependent over the specific implementation platform. We adopted a standard code to build as quad-trees [2] and *function-trees* according to the details reported below.

To encode a *function-tree* a depth-first visit is used. Each node of tree is a record with the following fields:

- [Color]: Is a two-bit code to indicate four possible colors (e.g. 00 Black, 01 White, 10 Graph, 11 Gray)
- [Direction]: *(Only for graph node):* two bit code (N,S,W,E)
- [Info] *(Only for graph node):* stores the boundary representation of a trapezoidal region.
  If the node is at level t of the tree, than it represents a $[\ m/2^t\ ,\ m/2^t\ ]$ sub-square, where m is the size of processed image. The boundary representation is hence a $m/2^t$*-vector*, whose elements are less than $m/2^t$. A binary code for each element hence takes $log_2(m/2^t)$ bit, so size of field [Info] is: $m/2^t * log_2(m/2^t)$.

Similarly, a node in a quad-tree is a record with the following fields:

- [Color]: Is a one-bit code indicating if node is Homogeneous or Gray.
- [Info]: Binary code of a gray level *[0-255]* (8 bit are needed for this field)

Tables 2,3 report some results obtained with photograph of real scenes and synthetic images.

Similar experiments have been performed on textures. These kinds of images typically show a strong high frequency component over all bit-planes and so no effective compression can be obtained using *function-tree* decomposition.

N	Byte	Quadtree	4+4 binary code	4+4 gray code
1	65536	68268	1128	1081
2	65536	26318	14901	9992
3	65536	34631	18073	12200
4	65536	65256	21715	17610
5	65536	29481	20632	17617
6	65536	34731	16933	18864
7	65536	61674	36798	27903
8	65536	63034	49377	43772

Table 3. *Results on synthetic images (256x256) depicted in Figure 8. Image n. 1 is a generalisation of "Comb", therefore allows an excellent compression ratio.*

N	Image	Byte	Quadtree	4+4 binary code	4+4 gray code
1	*Bird*	65536	64593	46871	38956
2	*Bridge*	65536	67940	75271	65257
3	*Camera*	65536	65812	56597	46986
4	*Kodak-19*	65536	66856	53653	43472
5	*Kodak-02*	65536	67371	55104	48261
6	*Kodak-07*	65536	66552	61432	52143
7	*Barbara*	262144	270587	260640	223097
8	*Boat*	262144	271793	250732	212446
9	*Gold Hill*	262144	271528	253491	206838
10	*Kodak-04*	262144	266015	225496	183910
11	*Kodak-17*	262144	265306	219161	184596
12	*Kodak-05*	262144	269890	274673	247462
13	*Lena*	262144	266878	230320	190545
14	*Mandrill*	262144	272793	324294	275670
15	*Peppers*	262144	271346	236256	194904
16	*Whatshat*	262144	229015	236875	175837

Table 2. *Results on both classical test images* [16] *and some images from Kodak collection*[17].

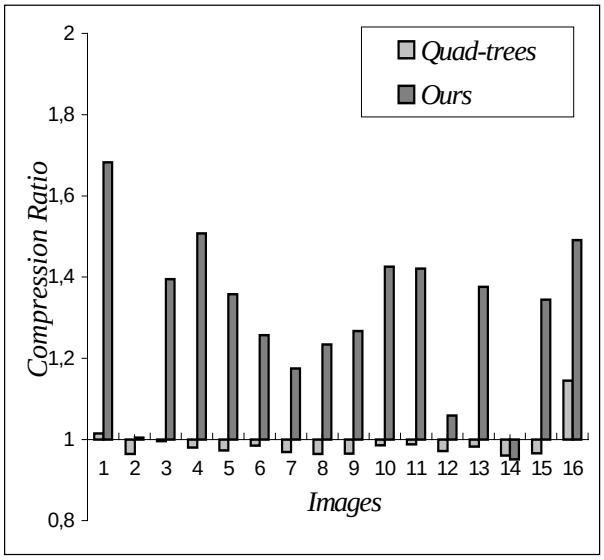

Figure 6. *Plot of compression ratios achieved by quad-trees and function-trees over the images in table 2.*

5. CONCLUSIONS AND FUTURE WORKS

This paper presents a compression scheme based on *function-trees*. Early experiments confirm that this method can be used to code gray level image. A comparison with the classical loss-less quad-tree decomposition is also presented and discussed.

Future development of the technique will include heuristics to generalize the method to color images.

Finally this paper inscribes itself into a line of research whose goal is to find a possible application of *function-trees* to fractal compression theory where quad-tree decomposition has been already used. In particular in a fractal compression framework the propose technique seems to be able to greatly speed up the range/domain

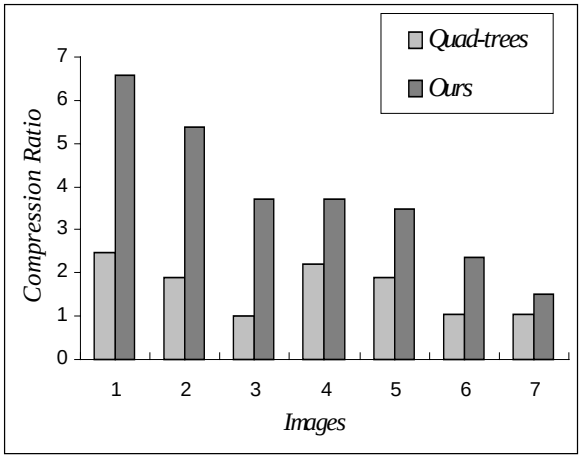

Figure 7. *Plot of compression ratios achieved by quad-trees and function-trees over the images in table 3. Observe that for sake of readability the values obtained for image n. 1 have not been plotted because its special structure gives unusually good compression ratio when processed with function-trees.*

coupling phase. Experiments and evaluation of this tool in this specific application are currently done [1] [7][12][13].

REFERENCES

[1] G. Gallo, D. Spampinato,"Function Trees: An Efficient Data Structure for Compact Binary Pictures", in *Proceedings WSCG95*, Pilsen, Czeck Republic, 1995;

[2] A.K. Jain, "*Fundamentals of Digital Image Processing*", Prentice Hall, Englewood Cliffs, 1989;

[3] H. Samet, "The Quadtree and Related Hierarchical Data Structures", *ACM Comput. Surv.* 16,2 – 1984;

[4] H. Samet, "*The Design and Analysis of Spatial Data Structures"*. Addison-Wesley Reading, MA – 1990;

[5] D. Taubman and A. Zakhor, "Orientation Adaptive Subband Coding of Images", *IEEE Trans. on Image Processing.* Vol. 3, No. 4, pp. 421-437, July 1994;

[6] D. Taubman, "High Performance Scalable Image Compression with EBCOT", *IEEE Trans. on Image Processing*, Vol. 9, No. 7, pp. 1158-1170, 2000;

[7] B. Wohlberg, G. de Jager, "A Review of Fractal Image Coding Literature", *IEEE Trans. on Image Processing*, vol. 8, n. 12, December 1999.

[8] M. Polvere, M. Nappi, "Speed-Up In Fractal Image Coding: Comparison of Methods", *IEEE Trans. On Image Processing* ,vol. 9, n. 6, June 2000;

[9] J. Cardinal, "Fast Fractal Compression of Grayscale Images", *IEEE Transactions on Image Processing*, vol. 10, n. 1, January 2001;

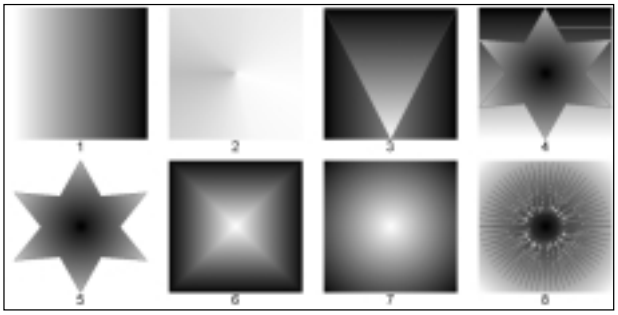

Figure 8. *Thumbnails of synthetic images.*

[10] D. Saupe, R. Hamzaoui, "Complexity Reduction Methods for Fractal Image Compression" in *Proc. IMA Conf. Image Processing: Mathematical Methods and Applications*, J.M. Blackledge, Ed., Oxford, England, pp. 211-229, Sept. 1994;

[11] D. Saupe, "Accelerating Fractal Image Compression by Multidimensional Nearest Neighbor Search", in *Proc. IEEE Data Compression Conf.*, J.A. Storer and M. Cohn, Eds. Snowbird, UT, Mar. 1995, pp. 222-231;

[12] Y. Fisher, "*Fractal Image Compression"*, Springer-Verlag, 1995;

[13] M. Barnsley, "*Fractals Everywhere"*. New York: Academic, 1988;

[14] T.H. Cormen, C.E. Leiserson, R.L. Rivest, "*Introduction to Algorithms"* – Mc Graw Hill – 1990;

[15] W. Pennebaker, J. Mitchell, *JPEG: Still Image Compression Standard"*, Norstrand Reinhold, New York, 1992;

[16] Waterloo BragZone site: *http://links.uwaterloo.ca/bragzone.base.html*

[17] Kodak's PhotoCD system: *ftp://www.cipr.rpi.edu/pub/image/still/KodakImages/*