

Image compression for fast wavelet-based subregion retrieval [☆]

A.S. Poulakidas, A. Srinivasan, Ö. Eğecioğlu, O. Ibarra ^{*}, T. Yang

Department of Computer Science, University of California, Santa Barbara, CA 93106, USA

Abstract

In an image browsing environment there is need for progressively viewing image subregions at various resolutions. We describe a storage scheme that accomplishes good image compression, while supporting fast image subregion retrieval. We evaluate analytically and experimentally the compression performance of our algorithm. We also provide results on the speed of the algorithm to demonstrate its effectiveness, and present an extension to a client/server environment. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Image compression; Wavelet; Subregion retrieval; Digital library; Quadrees; Huffman code

1. Introduction

Wavelet-based algorithms have become prominent in many areas of applied sciences, such as signal and image processing. A two-dimensional wavelet transform maps a given image to another that resembles the original, with half the resolution. Three additional coefficient matrices are also created. These matrices combined with the resembling image, can reproduce the original image by a two-dimensional inverse wavelet transform. (This inverse transform is also referred to as *reconstruction*, while the first transform is referred to as *decomposition*. See Fig. 1.) The decomposition can be applied to the resembling image recursively to produce smaller and smaller images, to a prescribed maximum depth. The smallest resembling image created is referred to as a *thumbnail*. Instead of the original image, this thumbnail and a set of coefficient matrices are stored.

In the Alexandria Digital Library (ADL) project [3] at UC Santa Barbara, wavelets are used to support hierarchical image storage and browsing. In this paper we describe

[☆] Supported by NSF/NASA/ARPA Grant No. IRI94–11330.

^{*} Corresponding author.

E-mail addresses: omer@cs.ucsb.edu (Ö. Eğecioğlu), ibarra@cs.ucsb.edu (O. Ibarra).

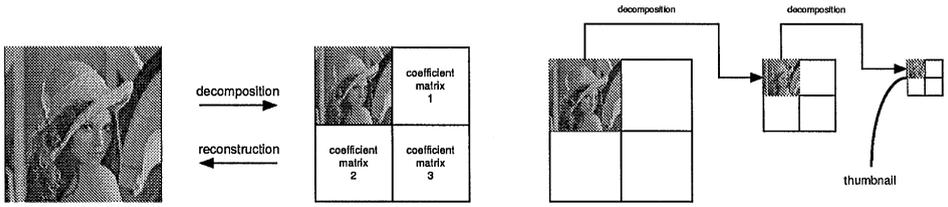


Fig. 1. (Left) Image decomposition and reconstruction, (Right) Multi-level decomposition of an image.

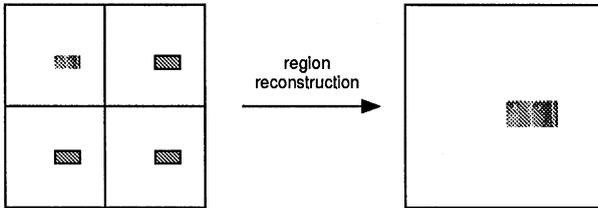


Fig. 2. Subregion reconstruction.

and analyze a compact scheme to store quantized coefficient matrices produced by wavelet decomposition. We briefly discuss the necessity of this scheme for the ADL environment:

1. There is a large number of satellite images and air-photos of average size 30 MBytes. Therefore the storage scheme must be compact to achieve a good *compression* ratio. Typically, real-valued wavelet coefficient matrices produced by the decomposition algorithm are quantized before images are ready for storage and retrieval. The challenge is to find the right coding for quantized coefficient matrices.
2. At high resolutions the size of an image in ADL is usually too large to browse. Common user requests are to access subregions of images. Therefore, the storage scheme should provide a means for *fast subregion retrieval*.

A request for an image subregion at some resolution translates to requests for subregions from stored coefficient matrices plus a reconstruction computation (see Fig. 2). Typical compression algorithms do not support subregion retrieval: to retrieve a subregion all the values may need to be uncompressed. To solve this problem, indexing on the compressed data has been proposed (e.g. [5]). The disadvantage is that indexing structures need extra space to be stored, thus deteriorating the compression performance.

In this paper we describe a quadtree-based indexing that is not external to the compression algorithm but contributes to the compression itself, thus eliminating the need for extra storage space for indexing.

There have been several applications of quadtrees to image compression (e.g. [9, 12]). These typically involve dividing an image into blocks such that some type of averaging can be performed in the block without introducing too much error. Our approach in the use of quadtrees is different: We construct a quadtree for the quantized coeffi-

cient matrices obtained from a wavelet transform. There is no error introduced since no averaging is performed. Errors are only introduced by the application of wavelet transforms and quantization.

There is extensive research on selecting an appropriate wavelet transform (e.g. [1, 6]), and on the quantization that will provide compression by minimizing the bit allocation, while preserving high image quality (e.g. [4, 10]). To further reduce storage requirements, often an extra phase of compression is applied to quantized values, typically using entropy coding, hybrid schemes using runlength/Huffman coding [1], or other methods such as EZW [8] and SPIHT [7]. Our scheme provides an alternative technique for this phase, which results in similar compression performance but with superior retrieval speeds.

In Section 2 we give a description of the storage scheme and the algorithm used to retrieve subregions from the compressed data. In Section 3 we analytically evaluate the compression performance of the scheme and experimentally compare it with other popular compressions like JPEG. In Section 4 we experimentally show that the indexing incorporated in our scheme allows faster image subregion retrieval. In Sections 5 we describe how the algorithm adapts to a client/server environment.

2. The storage scheme

The effective compression ratio of our scheme depends on the histograms of quantized coefficient matrices. In practical situations, the histogram distribution is typically symmetrical with peak centered at zero [6], similar to those shown in Fig. 3. The peaks of the histograms become shorter and wider as the level of the decomposition increases. However, this is of not much importance, since the size of coefficient matrices becomes smaller as the level of decomposition increases. Just the coefficient matrices at the first level of decomposition add up to more than 75% of the total space size of coefficient matrices. Thus, it can be assumed that the quantized coefficient matrices contain many zeros.

Furthermore, because of the locality observed in most images, and since coefficient matrices consist of difference values among weighted neighboring pixels, it is expected that quantized matrices will contain many two-dimensional blocks consisting of zeros only. *Quadrees* are used to eliminate unnecessary storage of two-dimensional blocks of zeros. Furthermore, they provide a natural indexing mechanism to easily access subregions.

A quadtree is used to store a coefficient matrix as follows: the root corresponds to the entire matrix. If this matrix contains some non-zero values, the root will have four children, which correspond to the top-left, top-right, bottom-left and bottom-right equally sized submatrices/blocks in the matrix. If any of these blocks is found to be composed of zeros the corresponding tree node becomes a *0-leaf* in the quadtree, and we say that this block is “reduced” to a 0-leaf. The remaining blocks are partitioned into four blocks and four children are added to their corresponding nodes in the quadtree.

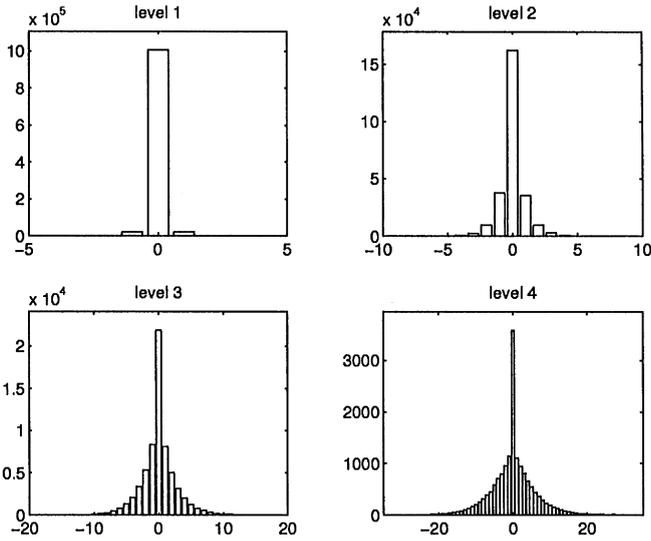


Fig. 3. Histograms of quantized coefficient matrices produced at various levels of wavelet decomposition.

Each new block is examined to see if it can be reduced. This partitioning continues until the algorithm determines that a block is small enough and should not be further divided. This block will be represented in the quadtree as a *stored-leaf*, and another compression method will be used to store its contents.

There are three kinds of leaves in our quadtrees: 0-leaves that represent blocks of zeros, stored-leaves that correspond to non-zero blocks that are stored, and *null-leaves* that correspond to empty blocks (these appear when the matrix is not square or its dimensions are not a power of 2).

Note that the quadtree scheme used by us is quite different from the zerotree of EZW [8]. In the latter, zero values occurring at different levels of resolution from the same region in space can be compressed significantly. Our scheme compresses blocks of zeros occurring at the same level of resolution. Unlike EZW, it does not require descendants of a region at finer scales to be zero in order to get significant compression. On the other hand, neither does it take advantage of such “self-similarity” which EZW can exploit. A similar statement can be made in a comparison with SPIHT [7]. Our quadtree scheme also includes indexing information that enables fast subregion retrieval.

Although quadtrees efficiently compress zero values, they do not contribute to the compression of the non-zero data that are represented by the stored-leaves. We therefore used *Huffman coding* to compress them. We had also tried Arithmetic Coding [11], but the compression was not as good as with Huffman coding.

We thus used a hybrid method that takes advantage of both quadtree and Huffman coding as follows: (1) Use the quadtree method to partition coefficient matrices and (2) compress all the stored-leaves of the quadtree using Huffman coding (see Fig. 4).

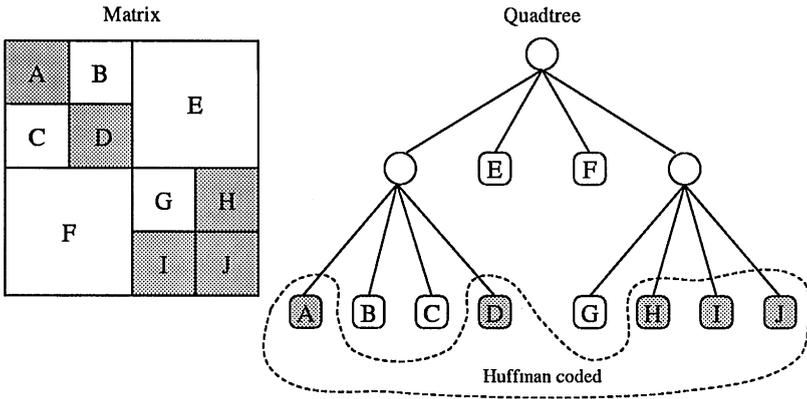


Fig. 4. The combination of quadtrees and Huffman coding in our storage scheme. The shaded blocks (leaves) correspond to non-zero blocks (stored-leaves). Non-shaded blocks (leaves) correspond to zero blocks (0-leaves).

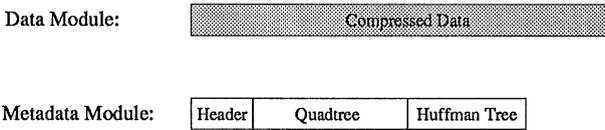


Fig. 5. The data components stored by the quadtree-based scheme.

A single Huffman tree is built for the matrix and the data are stored in two modules (see Fig. 5).

2.1. The compression algorithm

Storing the quadtree. Using a compact representation to store the quadtree is important, since in many cases the quadtree takes more than 50% of the total storage required for a coefficient matrix.

We use preorder coding to represent a quadtree. We store a 00 2-bit string when we encounter an internal node in the preorder traversal of the quadtree, a 01 2-bit string when we encounter a 0-leaf or a null-leaf, and when we encounter a stored-leaf we store a 1 bit followed by the *bit-offset*¹ in the file where the compressed block is stored.

To store these bit-offsets the naive approach is to allocate a constant number of bits for each bit-offsets, as in Fig. 6(a), but this results in high storage requirements. Moreover, bit-offsets can take any value in the range [0, file bit size), and they tend to be evenly distributed in that range, thus making encoding-based compression inadequate for their storage.

¹ To eliminate the internal fragmentation at the byte level that appears when a block is compressed to a number of bits which is not a multiple of 8, bit-offsets instead of byte-offsets need to be used.

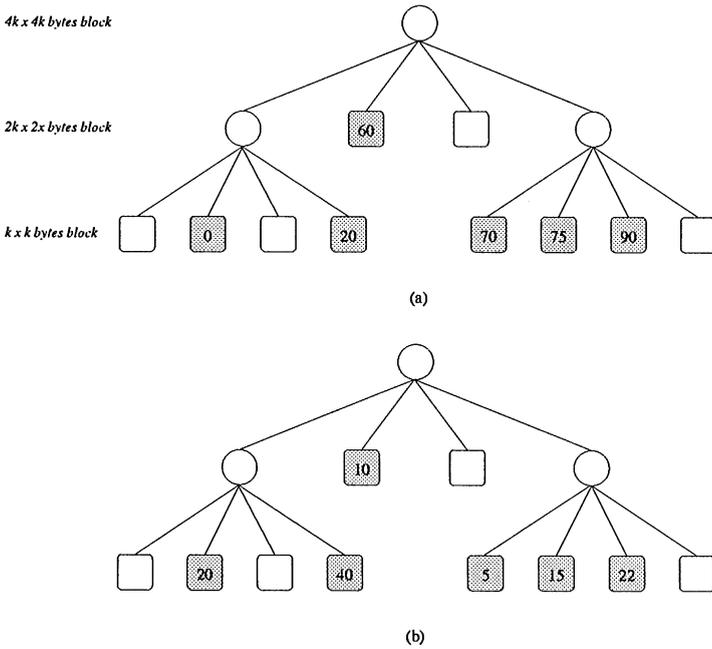


Fig. 6. The nodes at each level of the tree correspond to blocks whose size is given on the left of the tree. (a) Typical quadtree representation. The bit representation of this quadtree is 00 00 01 1 (0) 01 1 (20) 1 (60) 01 00 1 (70) 1 (75) 1 (90) 01, where the numbers in parentheses represent the offset in decimal and require c bits each, for some constant c . (b) Improved quadtree representation. The bit representation of this quadtree is 00 00 01 1 (20) 01 1 (40) 1 (10) 01 00 1 (5) 1 (15) 1 (22) 01, where the numbers in parentheses represent the increment in decimal. The increment (10) is allocated $2 \log k + 5$ bits since its leaf points to a $2k \times 2k$ byte block, while all the other increments are allocated only $2 \log k + 3$ bits since they point to $k \times k$ blocks.

But there is a bound for the *difference* between two successive bit-offsets: If a bit-offsets points to a compressed $k \times l$ block that allocates a byte per value, the next bit-offset will have a value at most $8kl$ larger. This implies that the *increment* between these two successive bit-offsets only requires $\log kl + 3$ bits. See Fig. 6(b). Furthermore, by storing only the increments between successive bit-offsets, we can easily reconstruct the quadtree with the actual bit-offsets when traversing it in preorder. This strategy leads to a big space saving over the typical approach: the storage requirements are experimentally found to be about 1/3 of the requirements if 34 bits were allocated per pointer.

Pruning the quadtree. To further reduce the storage requirements, an adaptive control mechanism is provided that automatically *prunes* a quadtree to improve the compression, while making sure that the desired trade-off with the region retrieval speed is not violated. There exists a trade-off because there are parts of some coefficient matrices that do not compress well under quadtrees. For these parts the corresponding subtree of the quadtree should be minimal in order to achieve the best compression. On the

other hand, if the quadtree is too small insufficient indexing may lead to a slow sub-region retrieval. The desired trade-off is determined by a parameter that specifies the maximum non-zero area that is allowed to be a stored-leaf in the quadtree.

Pruning a node means that the subtree rooted with this node becomes a stored-leaf. The pruning is done by performing a postorder traversal of the quadtree. For each node visited, the storage requirements when the node is pruned and not pruned are compared. If it is better not to prune the node, it and all its ancestors will not be pruned.

More precisely, for each Node of the quadtree the storage requirements are computed using the following relations:

$$\begin{aligned} \text{Huffman_code(Node)} &= \sum_{C \text{ is child of Node}} \text{Huffman_code}(C) \\ \text{pruned_size(Node)} &= \text{Huffman_code(Node)} \\ &\quad + 1 + \text{increment_bits(Node)} \\ \text{not_pruned_size(Node)} &= 2 + \sum_{C \text{ is child of Node}} \min \begin{cases} \text{not_pruned_size}(C) \\ \text{pruned_size}(C) \end{cases} \end{aligned}$$

where $\text{Huffman_code}(A)$ and $\text{increment_bits}(A)$ are the bits needed to store the Huffman code and the bit-offset increment for the block corresponding to A . If $\text{pruned_size}(\text{Node}) < \text{not_pruned_size}(\text{Node})$, it is better not to prune Node.

We take advantage of the fact that, given a common Huffman tree and two blocks A and B , $\text{Huffman_code}(A) + \text{Huffman_code}(B) = \text{Huffman_code}(AB)$, where AB is the concatenation of A and B . This leads to the following Theorem on the quality of our pruning strategy.

Theorem 1. *Given a quadtree Q and a Huffman tree H constructed from a matrix M , the above algorithm produces an optimal (in terms of space requirements) quadtree Q' . Furthermore, the computation can be done in linear time with respect to the size of M . (Note that Q and H are linear in the size of M .)*

Summarizing, compressing a quantized coefficient matrix is a three-pass process: In the first pass the quadtree is built in a preorder fashion and a Huffman tree is constructed. In the second pass the quadtree is pruned. In the final pass the quadtree is stored in preorder, and the stored-leaves are Huffman encoded and stored. The total running time and memory requirements are linear in the size of the matrix.

2.2. Subregion retrieval

When a subregion of a coefficient matrix needs to be retrieved, information has to be extracted from its quadtree. Note that there is no need to reconstruct the quadtree. By traversing the preorder coded quadtree we can get all the information that is needed about zero-blocks and the location of non-zero data that compose the subregion. This saves memory and requires, in the worst case, linear time with respect to the size of the stored quadtree.

Besides the reconstruction of the (normally small) Huffman tree, the only other computations needed are the filling of zero blocks in the subregion and loading from the disk and decoding of non-zero blocks. These take time proportional to the subregion size.

Most disk accesses involve reading small chunks of data that reside on neighboring disk locations. On top of the Unix disk I/O system, an extra buffering mechanism is implemented, thus reducing the number of disk reads and the total retrieval time.

3. The compression performance

We would like to estimate the effectiveness of the quadtree compression on a matrix whose values have a known distribution. We will analytically predict compression performance by assuming that the zeros are randomly scattered in the matrix. The compression should be a function of the distribution of values in the matrix, especially of the number of zeros (non-zeros are compressed using other techniques, e.g. Huffman coding). When the zeros are not uniformly distributed, as is the case with realistic images, the compression using our scheme will only be better. Thus the analysis below can be considered somewhat like a lower bound on the compression obtainable.

To better measure our compression performance we will also provide an experimental comparison of our storage scheme with other widely used compression schemes, like JPEG.

3.1. Analysis

Assume that the matrix has dimensions $2^n \times 2^n$ for some integer $n > 2$, and that initially there are z zeros scattered in the matrix randomly and independent of each other. Then there are at most n partitioning phases (*levels*) in the quadtree scheme where blocks of different size are examined to see if they are composed of zeros. Level l is the phase in which blocks of dimension $2^l \times 2^l$ are examined. Obviously $0 \leq l \leq n$, but for performance reasons level $l = 0$ is not considered.

In order to compute the compression performance of this quadtree, we need to know, at each level l , the number of blocks A_l that are found to be composed of zeros. The larger the number of reduced blocks is and the higher level they are found, the better the contribution of the quadtree to the compression is.

Let R_l be the combined size of the areas which are not reduced in levels higher than l . Obviously $R_n = 2^{2n}$, and

$$R_l = R_{l+1} - A_{l+1}2^{2(l+1)}. \quad (1)$$

Let Z_l be the number of zeros not reduced till level l . It is useful to use $S_l = 2^{2l}$ to denote the size of the areas that are examined at level l . Finally, let C_l be the number of the candidate for reduction areas at level l . Then $Z_l = z - (R_n - R_l)$ and $C_l = R_l/S_l$.

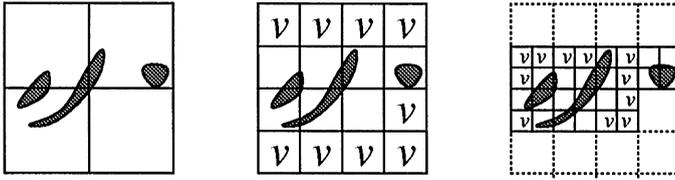


Fig. 7. Quadtree-based partition and reduction on a $2^n \times 2^n$ matrix. From left to right are levels $n - 1$, $n - 2$ and $n - 3$. Darker areas consist of non-zeros.

Example. Consider the matrix in Fig. 7. At level $n - 1$ 4 blocks, each of size $2^{2(n-1)}$, are examined, but none is reduced. Therefore, $S_{n-1} = 2^{2(n-1)}$, $C_{n-1} = 4$, $A_{n-1} = 0$. At level $n - 2$ 9 blocks are found to consist of zeros and are reduced. Therefore, $S_{n-2} = 2^{2(n-2)}$, $C_{n-2} = 16$, $A_{n-2} = 9$ and $R_{n-3} = 2^{2n} - 9 \cdot 2^{2(n-2)}$. At level $n - 3$ $S_{n-3} = 2^{2(n-3)}$, $C_{n-3} = 28$, $A_{n-3} = 11$ and $R_{n-4} = 2^{2n} - 9 \cdot 2^{2(n-2)} - 11 \cdot 2^{2(n-3)}$.

3.1.1. Probability distribution of A_l

A_l is a random variable that takes values from the set $\{0, 1, \dots, C_l\}$. Our aim is to express its distribution in terms of Z_l , S_l and C_l . All of them can be expressed in terms of l and R_l . Given (1), we hope then to be able to compute characteristics of all the A_l 's given just n and z .

The total number of ways to distribute Z_l zeros among C_l blocks of size S_l each is $\binom{C_l S_l}{Z_l}$. Let $B_{c,s}^m$ be the number of ways to distribute m zeros in c blocks of size s each, such that no block is full of zeros. The number of ways to distribute Z_l zeros among C_l blocks of size S_l each such that exactly i of them are full of zeros is $\binom{C_l}{i} B_{C_l-i, S_l}^{Z_l-i S_l}$. Then

$$\Pr(A_l = i) = \frac{\binom{C_l}{i} B_{C_l-i, S_l}^{Z_l-i S_l}}{\binom{C_l S_l}{Z_l}}. \tag{2}$$

We need to find $B_{c,s}^m$. Let $\mathbf{J}^k = \{\mathbf{j} \in \mathcal{N}^k \mid 1 \leq j_1 < j_2 < \dots < j_k \leq c\}$. For $j \in \mathbf{J}^1$, let Ω_j be the set of configurations where the j -th area is full of zeros. For $1 \leq k \leq c$ and $\mathbf{j} \in \mathbf{J}^k$, let $\Omega_{\mathbf{j}} = \Omega_{j_1} \cap \Omega_{j_2} \cap \dots \cap \Omega_{j_k}$.

Note that given k , $|\Omega_{\mathbf{j}}| = \binom{cs - ks}{m - ks}$ for any $\mathbf{j} \in \mathbf{J}^k$. Furthermore, $|\mathbf{J}^k| = \binom{c}{k}$. Then

$$\begin{aligned} |\Omega_1 \cup \Omega_2 \cup \dots \cup \Omega_c| &= \sum_{\mathbf{j} \in \mathbf{J}^1} |\Omega_{\mathbf{j}}| - \sum_{\mathbf{j} \in \mathbf{J}^2} |\Omega_{\mathbf{j}}| + \sum_{\mathbf{j} \in \mathbf{J}^3} |\Omega_{\mathbf{j}}| - \dots + (-1)^{c+1} \sum_{\mathbf{j} \in \mathbf{J}^c} |\Omega_{\mathbf{j}}| \\ &= \sum_{k=1}^c (-1)^{k+1} \binom{c}{k} \binom{cs - ks}{m - ks}. \end{aligned}$$

Since $B_{c,s}^m = \binom{cs}{m} - |\Omega_1 \cup \Omega_2 \cup \dots \cup \Omega_c|$, it follows that

$$B_{c,s}^m = \sum_{k=0}^c (-1)^k \binom{c}{k} \binom{cs - ks}{m - ks}. \tag{3}$$

Eq. (3) does not seem promising for simplification or appropriate for computation, and nor does (2). Passing to expected values,

$$\begin{aligned}
 E(A_l) &= \sum_{i=0}^{C_l} i \Pr(A_l = i) \\
 &= \sum_{i=0}^{C_l} i \frac{\binom{C_l}{i} B_{C_l-i, S_l}^{Z_l-iS_l}}{\binom{C_l S_l}{Z_l}}.
 \end{aligned}
 \tag{4}$$

The numerator is

$$\begin{aligned}
 &\sum_{i=0}^{C_l} i \binom{C_l}{i} B_{C_l-i, S_l}^{Z_l-iS_l} \\
 &= \sum_{i=0}^{C_l} i \binom{C_l}{i} \sum_{k=0}^{C_l-i} (-1)^k \binom{C_l-i}{k} \binom{(C_l-i)S_l - kS_l}{Z_l - iS_l - kS_l} \\
 &= \sum_{i=0}^{C_l} \sum_{k=i}^{C_l} (-1)^{k-i} i \binom{C_l}{i} \binom{C_l-i}{k-i} \binom{C_l S_l - k S_l}{Z_l - k S_l} \\
 &= \sum_{k=0}^{C_l} \sum_{i=0}^k (-1)^{k-i} i \binom{C_l}{i} \binom{C_l-i}{k-i} \binom{C_l S_l - k S_l}{Z_l - k S_l} \\
 &= \sum_{k=0}^{C_l} \binom{C_l S_l - k S_l}{Z_l - k S_l} \sum_{i=0}^k (-1)^{k-i} i \binom{C_l}{i} \binom{C_l-i}{k-i}.
 \end{aligned}
 \tag{5}$$

The inner sum is

$$\begin{aligned}
 &\sum_{i=0}^k (-1)^{k-i} i \binom{C_l}{i} \binom{C_l-i}{k-i} \\
 &= (-1)^k \sum_{i=0}^k (-1)^i \binom{i}{1} \binom{C_l}{i} \binom{C_l-i}{k-i} \\
 &= (-1)^k \sum_{i=0}^k (-1)^i \binom{C_l}{1} \binom{C_l-1}{i-1} \binom{C_l-i}{k-i} \\
 &= (-1)^k C_l \sum_{i=0}^k (-1)^i \binom{C_l-1}{i-1} (-1)^{k-i} \binom{k-C_l-1}{k-i} \\
 &= C_l \sum_{i=-1}^{k-1} \binom{C_l-1}{i} \binom{k-C_l-1}{k-1-i} \\
 &= C_l \sum_i \binom{C_l-1}{i} \binom{k-C_l-1}{k-1-i}.
 \end{aligned}$$

The Vandermonde’s convolution states that

$$\sum_i \binom{r}{i} \binom{s}{n-i} = \binom{r+s}{n},$$

for integer n . If we apply it to the above expression it becomes $C_l \binom{k-2}{k-1}$. This is 0 for all k , except for $k=1$ when it becomes C_l . Substituting in (5) gives

$$\begin{aligned} \sum_{i=0}^{C_l} i \binom{C_l}{i} B_{C_l-i, S_l}^{Z_l-iS_l} &= C_l \binom{C_l S_l - S_l}{Z_l - S_l} \\ &= C_l \frac{\binom{C_l S_l}{Z_l} \binom{Z_l}{S_l}}{\binom{C_l S_l}{S_l}}. \end{aligned}$$

Eq. (4) now becomes

$$E(A_l) = C_l \frac{\binom{Z_l}{S_l}}{\binom{C_l S_l}{S_l}}. \tag{6}$$

Fig. 8 illustrates how good an estimation $E(A_l)$ is for the actual value of A_l . The peaks in the figure appear narrow enough to justify the use of $E(A_l)$ instead of A_l . The quality of this estimation can actually be measured: in a similar way $E(A_l)$ was computed, we can compute the variance, $\sigma_{A_l^2}$, of A_l .

$$\begin{aligned} \sigma_{A_l^2} &= E(A_l^2) - E(A_l)^2 \\ &= \sum_{i=0}^{C_l} i^2 \Pr(A_l = i) - E(A_l)^2 \\ &= \sum_{i=0}^{C_l} (i-1)i \Pr(A_l = i) + \sum_{i=0}^{C_l} i \Pr(A_l = i) - E(A_l)^2 \\ &= \sum_{i=0}^{C_l} (i-1)i \Pr(A_l = i) + E(A_l) - E(A_l)^2 \\ &= \sum_{i=0}^{C_l} (i-1)i \frac{\binom{C_l}{i} B_{C_l-i, S_l}^{Z_l-iS_l}}{\binom{C_l S_l}{Z_l}} + E(A_l) - E(A_l)^2. \end{aligned} \tag{7}$$

The numerator of the first term is

$$\begin{aligned} \sum_{i=0}^{C_l} (i-1)i \binom{C_l}{i} B_{C_l-i, S_l}^{Z_l-iS_l} \\ = \sum_{k=0}^{C_l} \binom{C_l S_l - k S_l}{Z_l - k S_l} \sum_{i=0}^k (-1)^{k-i} (i-1)i \binom{C_l}{i} \binom{C_l - i}{k - i}. \end{aligned} \tag{8}$$

The inner sum is

$$\begin{aligned} \sum_{i=0}^k (-1)^{k-i} (i-1)i \binom{C_l}{i} \binom{C_l - i}{k - i} \\ = (-1)^k C_l \sum_{i=0}^k (-1)^i (i-1) \binom{C_l - 1}{i - 1} \binom{C_l - i}{k - i} \end{aligned}$$

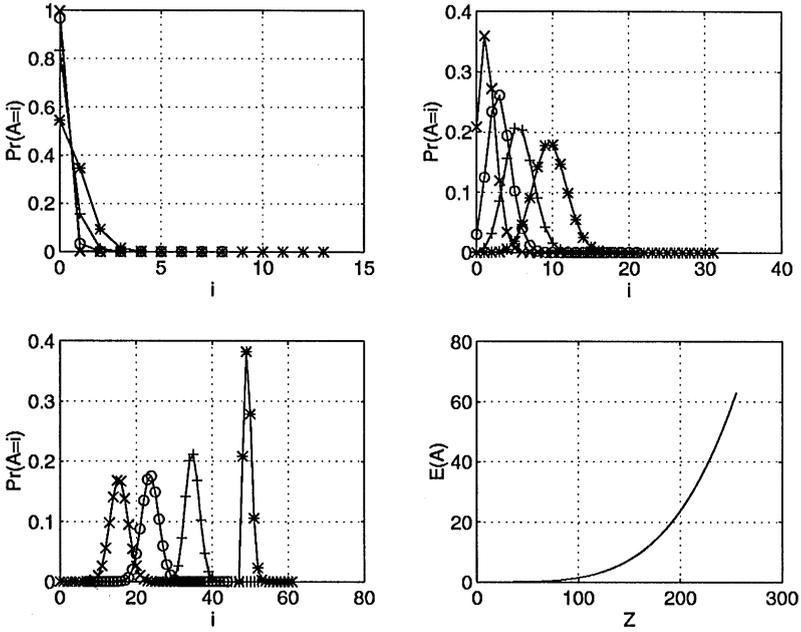


Fig. 8. The plots involve the probabilities of $A_2 = i$ when $C_2 = 64$. Z_2 is varying and each of the first three plots show the values of $A_2 = i$, as calculated by (2), for four values of Z_2 , using x, o, +, and * for the smallest to largest value of Z_2 . Top left: $Z_2 = 20, 40, 60, 80$; top right: $Z_2 = 100, 120, 140, 160$; bottom left: $Z_2 = 180, 200, 220$. Bottom right: plot of $E(A_2)$ as calculated by (6).

$$\begin{aligned}
 &= (-1)^k C_l (C_l - 1) \sum_{i=0}^k (-1)^i \binom{C_l - 2}{i - 2} \binom{C_l - i}{k - i} \\
 &= (-1)^k C_l (C_l - 1) \sum_{i=0}^k (-1)^i \binom{C_l - 2}{i - 2} (-1)^{k-i} \binom{k - C_l - 1}{k - i} \\
 &= C_l (C_l - 1) \sum_{i=-2}^{k-2} \binom{C_l - 2}{i} \binom{k - C_l - 1}{k - 2 - i} \\
 &= C_l (C_l - 1) \sum_i \binom{C_l - 2}{i} \binom{k - C_l - 1}{k - 2 - i} \\
 &= C_l (C_l - 1) \binom{k - 3}{k - 2},
 \end{aligned}$$

which is 0 for all k , except for $k = 2$ when it becomes $C_l (C_l - 1)$. Substituting in (8) gives

$$\begin{aligned}
 \sum_{i=0}^{C_l} (i - 1) i \binom{C_l}{i} B_{C_l - i, S_l}^{Z_l - i S_l} &= C_l (C_l - 1) \binom{C_l S_l - 2 S_l}{Z_l - 2 S_l} \\
 &= C_l (C_l - 1) \frac{\binom{C_l S_l}{Z_l} \binom{Z_l}{2 S_l}}{\binom{C_l S_l}{2 S_l}}
 \end{aligned}$$

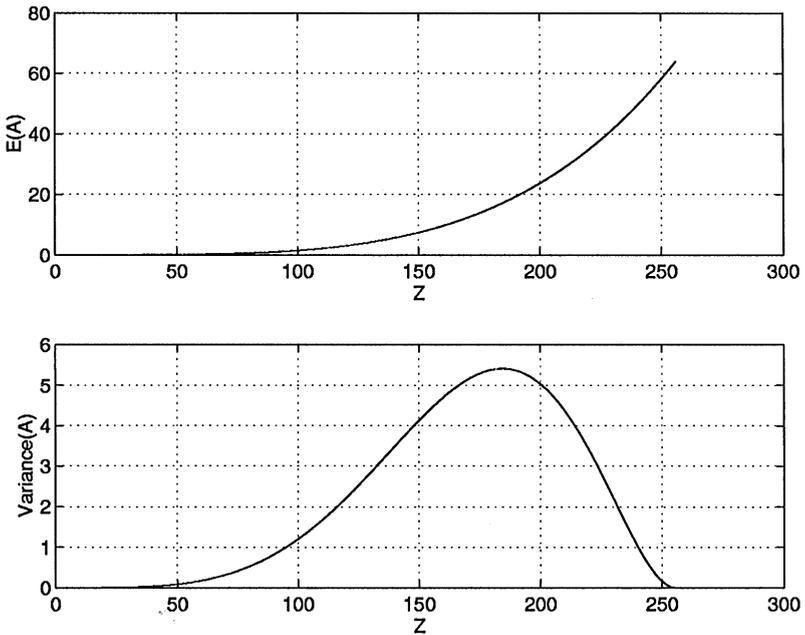


Fig. 9. Plots of $E(A_1)$ and $\sigma_{A_1}^2$ where $C_1 = 64$ and Z_1 is varying.

and then (7) becomes

$$\sigma_{A_1}^2 = C_1(C_1 - 1) \frac{\binom{Z_1}{2S_1}}{\binom{C_1 S_1}{2S_1}} + C_1 \frac{\binom{Z_1}{S_1}}{\binom{C_1 S_1}{S_1}} - C_1^2 \frac{\binom{Z_1}{S_1}^2}{\binom{C_1 S_1}{S_1}^2}. \tag{9}$$

As Fig. 9 illustrates, when the number of zeros becomes large the expected number of blocks that are reduced grows rapidly and the actual number is very close to the expected (the variance becomes smaller rapidly). As discussed in the previous chapter, after the wavelet decomposition and the quantization of the coefficients, the number of zeros in the coefficient matrices is indeed large. This justifies the combination of quadrees and wavelet transforms as a compression scheme.

3.1.2. Estimating the compression performance

Assume no pruning is used. At each level $l > 1$ in the quadtree there will be $C_l - A_l$ internal nodes and A_l 0-leaves. In either case 2 bits are been stored, therefore $2C_l$ bits are needed to store level l . For $l = 1$, 2 bits will be stored for the A_1 0-leaves and $1 + 2l + 3$ bits are needed to store the bit-offset increments for each of the $C_1 - A_1$ nonzero leaves. The total number of bits stored in the quadtree is

$$Qbits = 6C_1 - 4A_1 + \sum_{l=2}^n 2C_l.$$

For $l < n$, $C_l = 4(C_{l+1} - A_{l+1}) = 4^{n-l} - \sum_{i=l+1}^n 4^{i-l} A_i$. Substituting above

$$\begin{aligned}
 Qbits &= 6 \cdot 4^{n-1} - 6 \sum_{i=2}^n 4^{i-1} A_i - 4A_1 + 2 \sum_{l=2}^n 4^{n-l} - 2 \sum_{l=2}^n \sum_{i=l+1}^n 4^{i-l} A_i \\
 &= 6 \cdot 4^{n-1} - 6 \sum_{i=2}^n 4^{i-1} A_i - 4A_1 + 2 \frac{4^{n-1} - 1}{3} - 2 \sum_{i=3}^n 4^i A_i \sum_{l=2}^{i-1} 4^{-l} \\
 &= \frac{20 \cdot 4^{n-1} - 2}{3} - 4A_1 - 6 \sum_{i=2}^n 4^{i-1} A_i - 2 \sum_{i=3}^n 4^i A_i \frac{1 - 4^{2-i}}{12} \\
 &= \frac{20 \cdot 4^{n-1} - 2}{3} - 4A_1 - 24A_2 - \sum_{i=3}^n \frac{10 \cdot 4^i - 16}{6} A_i \\
 &= \frac{20 \cdot 4^{n-1} - 2}{3} - \sum_{l=1}^n \frac{5 \cdot 4^l - 8}{3} A_l.
 \end{aligned} \tag{10}$$

At level l there are $A_l 2^{2l}$ zeros reduced. The total number of zeros reduced is

$$Qreduced = \sum_{l=0}^n 2^{2l} A_l. \tag{11}$$

The quadtree scheme stores $Qbits$ extra bits and saves the storage of $Qreduced$ zeros. We measure the compression performance of the quadtree by the ratio

$$Qratio = \frac{Qbits}{8Qreduced}.$$

To evaluate its expected compression performance, $E(Qratio)$, the expected values of $Qbits$ and $Qreduced$ can be derived from (10) and (11) by substituting each A_k with $E(A_k)$:

$$E(Qbits) = \frac{20 \times 4^{n-1} - 2}{3} - \sum_{l=1}^n \frac{5 \times 4^l - 8}{3} E(A_l), \tag{12}$$

$$E(Qreduced) = \sum_{l=0}^n 2^{2l} E(A_l). \tag{13}$$

3.1.3. Comparison with experimental results

We can approximately compute $E(A_n), E(A_{n-1}), \dots, E(A_1)$ by using (1) and (6). To evaluate this approximation, the expected $Qratio$ is compared with that observed when applying the compression scheme on matrices where z zeros are randomly placed. The size of the matrices and z were given various combinations of realistic values. In the experiments the observed values of $Qratio$ were consistently within 5% of the expected value of $Qratio$. This suggests that we can analytically estimate the compression performance of the quadtree scheme within a 5% error range.

If we compare the estimated $Qratio$ or the one observed when compressing random matrices, with the $Qratio$ on coefficient matrices of actual images with the same dimensions and number of zeros, we find that the coefficient matrices compress significantly better. The reason is that, as mentioned earlier, in our analysis we assume that zeros

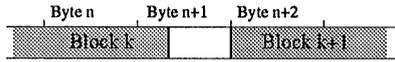


Fig. 10. Fragmentation at the compressed non-zero blocks.

are randomly scattered, while actual coefficient matrices normally have groups of zeros, due to localities in the images. The difference of the analysis and the experiments can be used to quantify the image localities' effect. The analytical results can also be used as pessimistic estimates of expected compressions.

3.2. The effect of the optimizations

The quadtree-based storage scheme uses three basic optimizations to improve its compression performance. This section serves to justify them and show their effect.

The first optimization is the elimination of the internal fragmentation. Internal fragmentation occurs because the compressed size of a non-zero data block is not necessarily a multiple of 8 bits. If a compressed data block was required to be stored at byte offsets, half a byte in average would be wasted (Fig. 10). On the other hand, to eliminate internal fragmentation bit-offsets have to be used in the quadtree instead of byte-offsets, which require 3 extra bits each. Therefore it appears that in average we save half a byte and add 3 bits for each non-zero data block. This trade is to our advantage and is also justified experimentally. Furthermore, the program code that is used to eliminate internal fragmentation in the non-zero data was used to eliminate it in the quadtree data as well, where offsets do not need to start at the beginning of a byte, thus reducing the quadtree size.

Storing bit-offset increments in place of bit-offsets results in substantial reduction in the size of the quadtree. Finally, the quadtree pruning further reduces its size while increasing the size of the compressed non-zero data. By Theorem 1 this increase is less than the reduction. In Fig. 11 the successive effect of these optimizations is illustrated in a typical example.

3.3. Experimental comparison with other compression methods

To experimentally give a perspective of the compression performance of the quadtree scheme, we compare it with the popular lossy JPEG method and with the lossless GIF, TIFF LWZ, and TIFF PackBits.

While making these comparisons, we wish to note that our scheme has been developed keeping in view speed of subregion retrieval, including the inverse wavelet transform, too. Thus we chose a wavelet [1] that could be inverted fast, even though that it was not the best one for the purposes of compression. Also, the quadtree includes indexing information to enable subregion retrieval, a constraint not face by the other schemes with which we compare. However, our scheme still compresses well.

The comparisons for eight representative greyscale images are shown in Figs. 12 and 13, where we use the peak-signal-to-noise ratio (PSNR) as the criterion for the

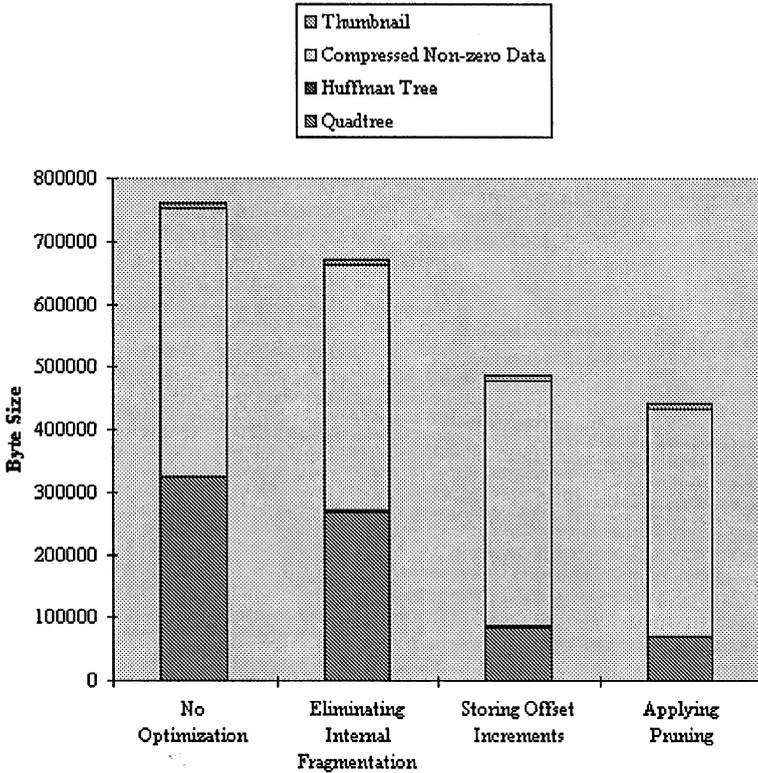


Fig. 11. Typical effect on compression of the optimizations. The Huffman tree portions are non-visible because of their small size.

reconstruction quality.² The solid lines correspond to the quadtree scheme, while the dashed correspond to the JPEG. The vertical dotted lines indicate the performance of the best of the lossless methods.

We observe that the quadtree scheme tends to perform better when the PSNR is high or very low. For most images there is a range of moderate values of the PSNR where JPEG is slightly better. For most applications, for the reconstruction quality to be acceptable the PSNR has to be higher than 35 dB. Whenever this is true, for all the images we considered the quadtree scheme achieved better compression than JPEG.

Therefore, if compression better than that achieved by lossless methods is needed (area in the plots to the right of the vertical dotted lines) but good reconstruction quality

² The PSNR for an $N \times M$ 8-bit image I and its reconstructed equivalent \hat{I} is given in dB by

$$PSNR = 20 \log_{10} \left(\frac{255}{\sqrt{(1/NM) \sum_{i=1}^N \sum_{j=1}^M [I(i,j) - \hat{I}(i,j)]^2}} \right).$$

The higher the PSNR the higher is the reconstruction quality.

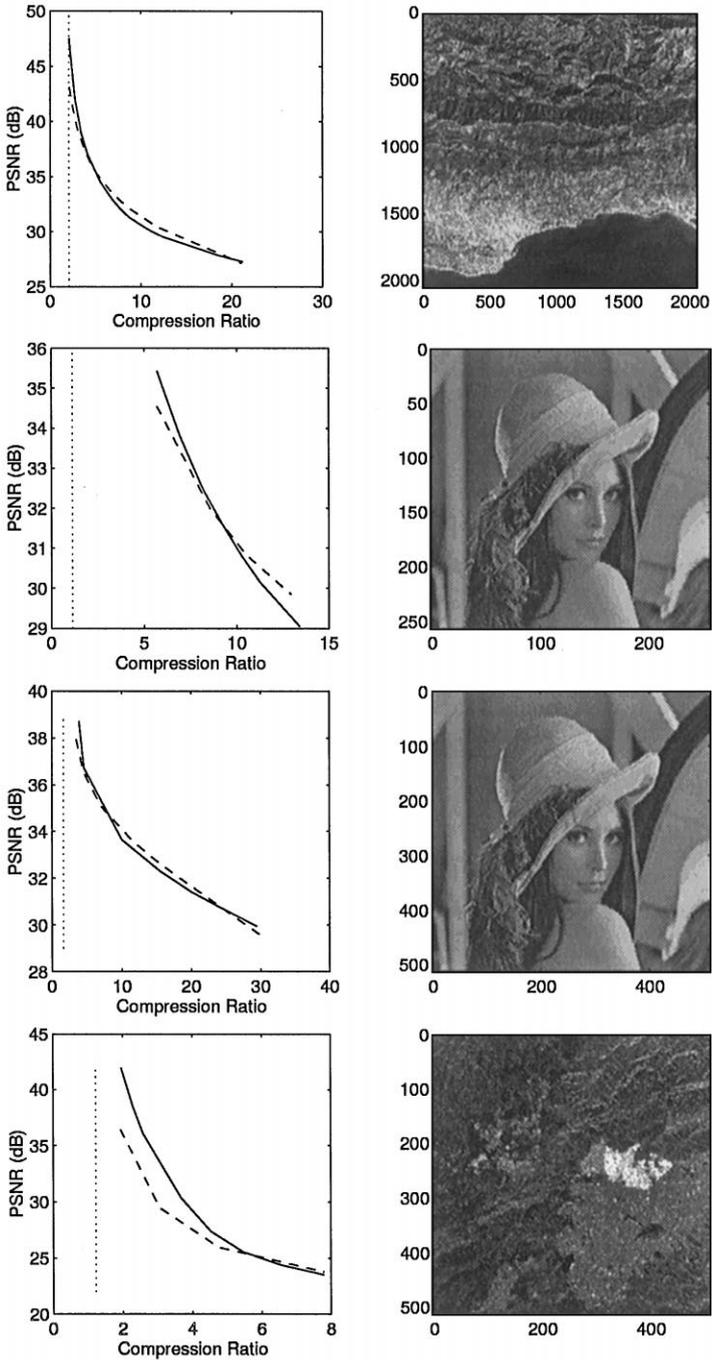


Fig. 12. Compression performance on the first four representative images. Note: There are other versions of the “lena” image (second and third from the top) available, and they differ in ease of compression. Thus some caution is called for in comparing with other works.

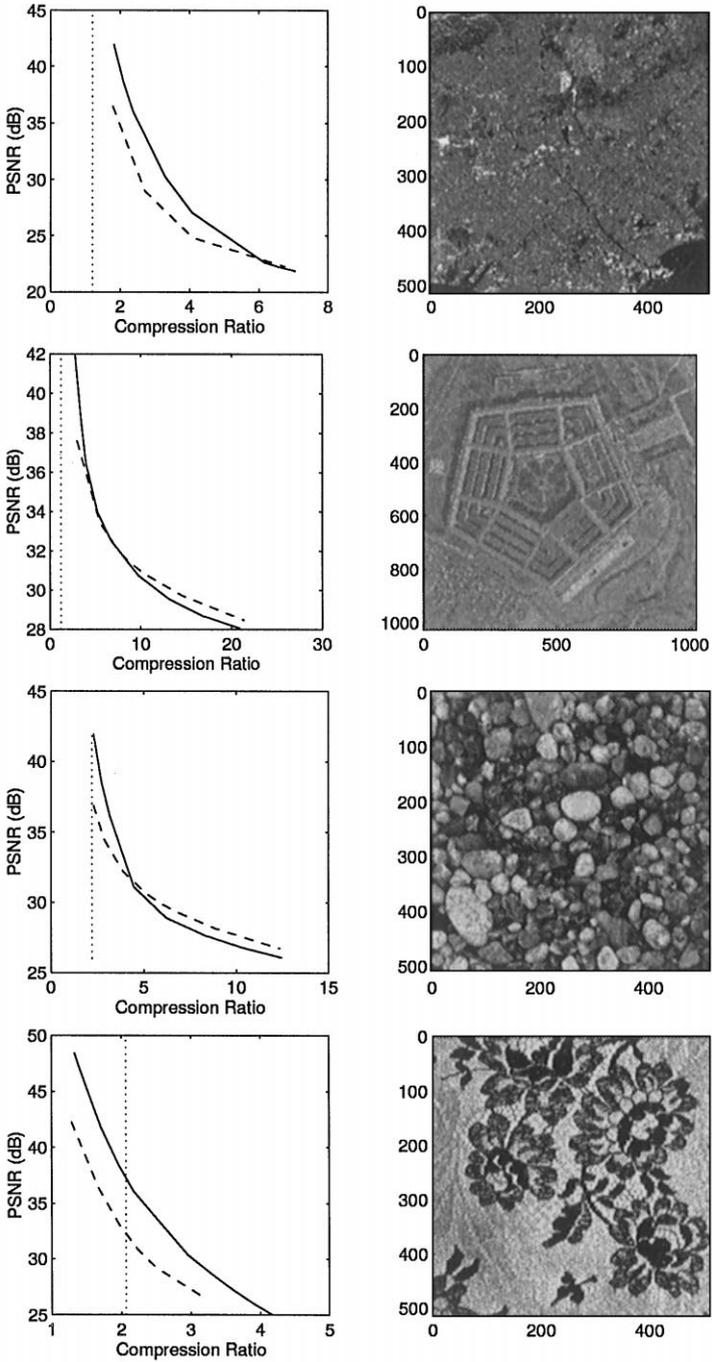


Fig. 13. Compression performance on the last four representative images.

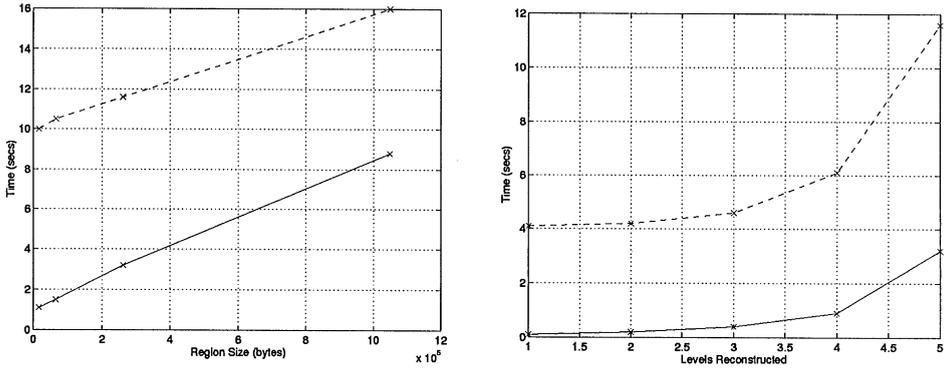


Fig. 14. Comparison of Retrieval+Reconstruction times on a 2048×2048 image. On the left reconstruction is performed for 5 levels for subregions of size 128×128 , 256×256 , 512×512 and 1024×1024 . The subregions are reconstructed for 5 levels. On the right a typical-sized 512×512 subregion is reconstructed at various levels.

is required (area where PSNR > 35 dB), the quadtree scheme seems more appropriate than the other compression methods considered in this section.

4. Experiments on subregion retrieval

We ran our experiments on a SUN SPARC 5 (75 MHz) workstation with its own local SCSI-2 disk. To test our subregion retrieval time we compared with another storage scheme that compresses quantized matrices using a combination of runlength and Huffman coding, resulting in a comparable space reduction to ours. Both schemes use the same wavelet transforms and Huffman coding. However for each subregion retrieval from a matrix, the adversary scheme requires the *whole* matrix to be loaded and uncompressed. Once data are uncompressed, only those portions needed to reconstruct the desired subregion are actually used by both schemes. In Fig. 14 the two schemes are compared. The solid lines refer to the times achieved by our method and the dashed lines refer to the adversary scheme. There is a significant benefit to our method due to its indexing capability.

5. Adaptation to client/server environment

Section 2 describes the storage scheme running on a single computer. The case where the user is logged on a client computer and requests image subregions from a server, where the images are stored, is of significant importance nowadays. In this environment there is a choice of executing the computational tasks on either the server or the client.

The image subregion retrieval can be decomposed into the following three tasks: loading of compressed coefficient data, uncompressing these data, and executing the

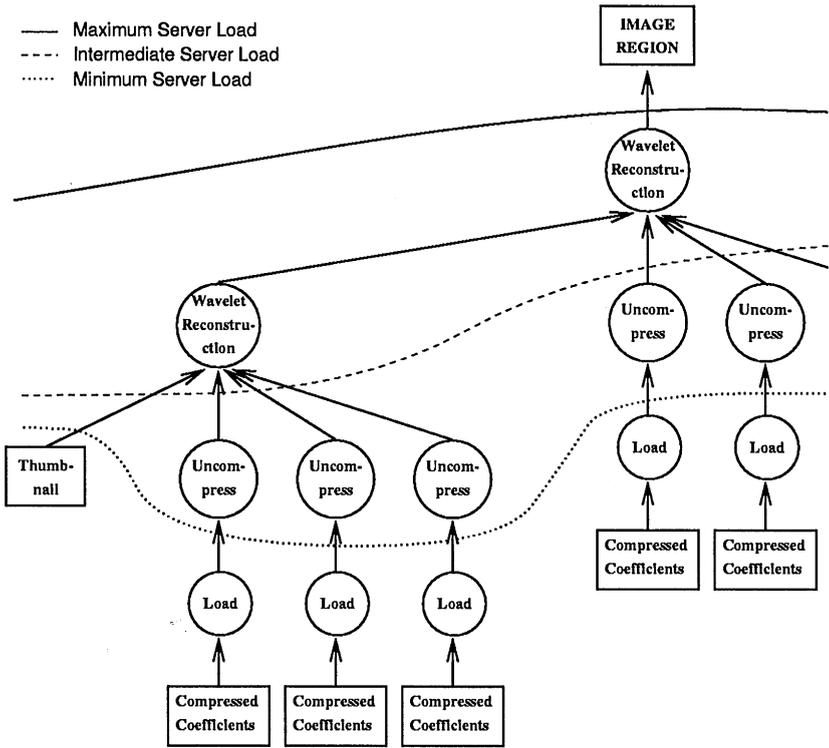


Fig. 15. The tasks involved in retrieving an image subregion.

wavelet reconstruction to produce the required image subregion. They are shown in Fig. 15, where the three curved lines specify three possible partitions of tasks between a server and a client:

- P1 all tasks executed at the server (solid line) and the requested image subregion is sent to the client,
- P2 the wavelet reconstruction is performed at the user's computer (dashed line), and
- P3 the server only loads the compressed data and the thumbnail (dotted line) and sends them to the client for processing.

Partition P1 is appropriate when the load on the server is low or the client is much slower. It has the disadvantage that the image subregion is sent over the network uncompressed.

Partition P2 results in a lighter load to the server: it only loads and uncompresses the coefficient data required to reconstruct the image subregion. The client needs to apply wavelet reconstruction in order to produce the subregion. The advantages of P2 over P1 are that the load on the server is lighter and if the client needs to view part of the reconstructed image subregion in an even higher resolution, the server only needs to send the extra coefficients to get to this higher resolution (*progressive delivery*).

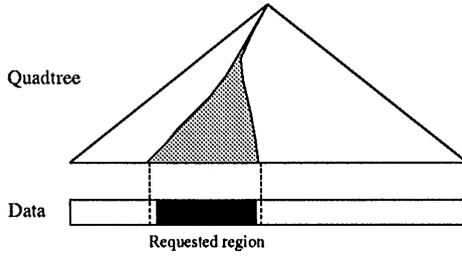


Fig. 16. Only a minimal subtree of the quadtree that “covers” the requested subregion is needed in order to recover the subregion. This is demonstrated here in 1-D. The 2-D case is a direct generalization.

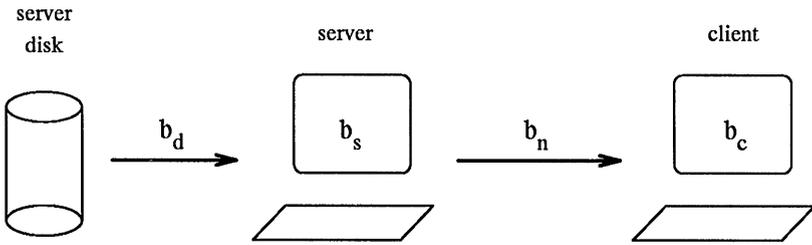


Fig. 17. Model of computational environment.

If partition P3 is used, the server only needs to load the compressed coefficients needed to reconstruct the image subregion and send them to the client. This greatly reduces the need for network bandwidth since the data that are sent are compressed. Furthermore, as part of these data, only a subtree of the quadtree that “covers” the subregion needs to be included (see Fig. 16).

There exists a fourth partition where the whole image is shipped to the client. The client then does all the computation. This partition, named P4, is practical only if the server load is very high.

5.1. Comparing the partitions

According to the available computation power on a server and a client, the network bandwidth between them and the size and resolution of the requested image subregion, each of P1, P2, P3 or P4 may be more appropriate.

We assume the computational environment of Fig. 17. The disk to server bandwidth is denoted by b_d , while the bandwidth of the network connection between the server and the client is denoted by b_n . We use b_s to measure the speed of the server. It indicates how much faster the server is than a SUN SPARC 5 (75 MHz). The value refers to the available processing power. If for instance the server hardware is 8 times faster than a SPARC 5, but only 20% of the server’s processing power is available (the rest is consumed by other processes), b_s would be 1.6. The respective measure for the client is b_c .

Each one of the four partitions has been implemented and experimented with. The connections between the client and the server were implemented using Unix sockets.

Assume that the image compression ratio is r and that the compressed data required to reconstruct the subregion of interest have size rs , where s is the size of the uncompressed subregion.³ Let S be the size of the whole uncompressed image.

Let T_i be the response time when partition P_i is used. Our experiments show that the following approximations are reasonable:

$$T_1 = \frac{rs}{b_d} + \frac{6rs + 23.7s}{10^6 b_s} + \frac{s}{b_n}, \quad (14)$$

$$T_2 = \frac{rs}{b_d} + \frac{6rs + 2s}{10^6 b_s} + \frac{s}{b_n} + \frac{21.7s}{10^6 b_c}, \quad (15)$$

$$T_3 = \frac{rs}{b_d} + \frac{7.3rs}{10^6 b_s} + \frac{rs}{b_n} + \frac{2rs + 23.7s}{10^6 b_c}, \quad (16)$$

$$T_4 = \frac{rS}{b_d} + \frac{0.44rS}{10^6 b_s} + \frac{rS}{b_n} + \frac{6rs + 23.7s}{10^6 b_c}. \quad (17)$$

A scheduler can predict the response time for each partition using the above formulas, and then choose the partition corresponding to the minimum response time. To increase the throughput of the server, the scheduler can dynamically adjust b_s to compensate for concurrent requests.

Example.. Assume the following parameters for the computing environment: $b_d = 3$ MBytes/s, $b_s = 2$, $b_n = 0.1$ MBytes/s, and $b_c = 1.5$. Let $r = 0.1$, which is reasonable. Then $T_1 = 22.18 \times 10^{-6}s$. P_2 transfers some of the computation to the client, which is slower. Thus it is no surprise that $T_2 = 25.8 \times 10^{-6}s$ is larger than T_1 . P_3 provides the best performance: $T_3 = 17.33 \times 10^{-6}s$. P_4 is definitely worse in this environment than P_3 , even if s is as large as S : $T_4 = 1.05 \times 10^{-6} \cdot S + 16.2 \times 10^{-6}s$.

Example.. By modifying the previous example such that the connection between the server and the client has a typical Ethernet bandwidth, $b_n = 1$ MByte/s, and assuming that both computers have the same processing power, $b_s = b_c = 2.5$, the predictions become

$$T_1 = 10.75 \times 10^{-6}s$$

$$T_2 = 10.75 \times 10^{-6}s$$

$$T_3 = 9.98 \times 10^{-6}s$$

$$T_4 = 0.15 \times 10^{-6}S + 9.72 \times 10^{-6}s$$

³ An approximation will be made: it is assumed that these rs data can be read from the disk with no overhead. This is not the case, since to read the quadtree data related to the subregion, in the worst case the whole quadtree may need to be read. The approximation is reasonable because

- the quadtree data are not a dominant proportion of the whole data, and
- reading the quadtree from the server disk and processing it is a small fraction of the total service time.

If $s < 0.57S$, P_3 is the most appropriate partition, otherwise P_4 is better. If the client was a little slower, $b_c = 2$, P_1 would be the most efficient partition.

6. Conclusions

We have implemented an encoding scheme that provides both good compression and fast retrieval of subregions of an image. We observed from the experiments that the compression ratio is better than popular compression schemes like JPEG and similar to that obtained using other hybrid compression schemes that do not provide indexing. We also provided analytical evaluation of the compression behavior. Our use of the quadrees produces an indexing mechanism in a natural way while it performs a compression to eliminate blocks of zeros. The retrieval times for the subimages shows that our scheme produces a significant benefit due to its indexing capability. We have extended the scheme to a distributed environment, where the various computational tasks can be performed either on the client or the server side.

Acknowledgements

We wish to acknowledge the advice given by anonymous referees, which has enabled us to clarify certain points.

References

- [1] E.H. Adelson, E. Simoncelli, Subband image coding with three-tap pyramids, Picture Coding Symposium 1990, Cambridge, MA.
- [2] D. Andresen, T. Yang, D. Watson, A. Poulakidas, Dynamic processor scheduling with client resources for fast multi-resolution WWW image browsing, Proc. 11th Int. Parallel Processing Symp. April 1997, to appear.
- [3] Alexandria Digital Library, <http://alexandria.sdc.ucsb.edu/>.
- [4] M. Barlaud, P. Sole, T. Gaidon, M. Antonini, P. Mathieu, Pyramidal lattice vector quantization for multiscale coding, IEEE Trans. Image Process. 3(4) (1994) 367–381.
- [5] Kodak FlashPix Format, <http://www.kodak.com/daiHome/flashPix/flashPixHome.shtml>.
- [6] S.G. Mallat, A theory for multiresolution signal decomposition: the wavelet representation, IEEE Trans. Pattern Anal. Mach. Intell. 11(7) (1989) 674–693.
- [7] A. Said, W.A. Pearlman, A new, fast, and efficient image codec based on set partitioning in hierarchical trees, IEEE Trans. Circuits Systems Video Techno. 6(3) (1996) 243–249.
- [8] J.M. Shapiro, Embedded image coding using zerotrees of wavelet coefficients, IEEE Trans. Signal Process. 41(12) (1993) 3445–3462.
- [9] E. Shusterman, M. Feder, Image compression via improved quadtree decomposition algorithms, IEEE Trans. Image Process. 3(2) (1994) 207–215.
- [10] N. Strobel, S.K. Mitra, B.S. Manjunath, An approach to efficient storage, retrieval, and browsing of large scale image databases, Proceedings of SPIE Conference on Image Storage and Archiving System, Vol. 2606, pp. 324–335, Philadelphia, Pennsylvania, October 1995.
- [11] I.H. Witten, R.M. Neal, J.G. Cleary, Arithmetic coding for data compression, Commun. ACM 30(6) (1987) 520–540.
- [12] H.-S. Wu, R.A. King, R.I. Kitney, Improving the performance of the quadtree-based image approximation via the generalized DCT, Electron. Lett. 29(10) (1993) 887–888.