JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 26, 1917-1930 (2010)

Short Paper

Linear Quadtree Construction in Real Time*

CHI-YEN HUANG AND YU-WEI CHEN+
Department of Information Management
National Taipei College of Business
Taipei, 100 Taiwan
+*Graduate Institute of Information and Logistics Management*
National Taipei University of Technology
Taipei, 106 Taiwan

The paper presents a novel method for encoding the linear quadtree of a given image. In this method, the pixels of the image are scanned in row major order. In each encountered pixel, the result codes in the linear quadtree are updated simultaneously in real time. The linear quadtree is thus obtained after all pixels are processed. This method is quite different from those in previous studies, which need huge memory space to store the input pixels for further processing, or need post processing to rearrange the sequence of the final codes to put them back in increasing order. Moreover, since this method adopts simpler and more efficient operations than previous methods, it is found to be faster in experiments.

Keywords: linear quadtree, linear bintree, spatial data structure, image encoding, locational code

1. INTRODUCTION

Image representing is an important issue in the field of image processing. Many efficient methods for compressing images have previously been proposed. Although these methods reduce the storage space required by the compressed data is less than that of the original, they increase the complexity of data manipulation. Hence, spatial data structures are now widely discussed as a means to save the storage space while enabling fast image data computation. Based on the structure of the encoding scheme, the spatial data structures are classified into five types [1]. The linear tree is an important approach with many useful properties. It has been shown to be storage-saving [2] and good for direct manipulation, *e.g.* geometry properties [3, 4], set operations [1], neighbor finding [1, 5-7], connected component labeling [8], mirroring [9, 10], dilation [11], rotation [12], transformation [13, 14], conversion [15, 16] and compression [17]. The linear tree has thus been studied intensively [18-20].

The method of building the linear quadtree is important in the field of spatial data structures. Clearly, the linear quadtree of an image can be calculated from the correspond-

Received May 1, 2008; revised March 26 & June 2 & July 23 & August 3, 2009; accepted August 5, 2009.
Communicated by Suh-Yin Lee.
* The authors would like to thank the National Science Council of R.O.C., Taiwan for financially supporting this research under Contract No. NSC 97-2221-E-027-118.

ing quadtree by traversing the quadtree in preorder and collecting the leaf nodes. Based on the concept, some studies were proposed in early years [4]. However, an algorithm to build the linear quadtree directly is useful. Shaffer and Samet [21] presented an efficient method for this aim. The input pixels of the image are in row-major order in their algorithm. The method maintains some adjunctive data structures to guarantee that each insertion is a maximal image block. Holroyd *et al*. presented another such method [22], in which the input pixels are assumed to be in Morton number order. Holroyd *et al*.'s method is efficient but needs additional effort to transform the input pixel sequence to the row-major order in practical usage.

This study presents a novel method for constructing linear quadtrees. The proposed method also assumes that the pixels of the image are in row major order, which is a normal sequence when an image is scanned. The linear quadtree is updated synchronously in each non-white encountered pixel. The result linear quadtree can thus be obtained without post processing after all pixels of the image have been processed.

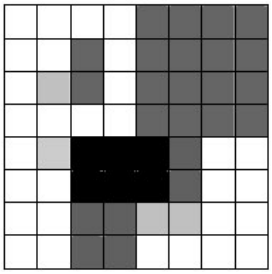
The method proposed in this study has the following unique properties. (1) The image can be encoded in real time. This property is useful in the case of the scanning process is interrupted; (2) The resulting linear quadtree does not need a post-process to preserve the increasing property [1]; (3) The disc space requirement adopted in the method is relatively low; (4) The proposed algorithm is faster than that of [21], since it adopts simple and more efficient operations.

The rest of this paper is organized as follows. Section 2 presents the tree and linear quadtree structures. Additionally, some previous studies for the construction of the linear quadtree are introduced. Section 3 presents some important properties and theorems for the proposed method. Section 4 presents the method itself. Experiments are presented in section 5. Finally, some conclusions are given in the last section.

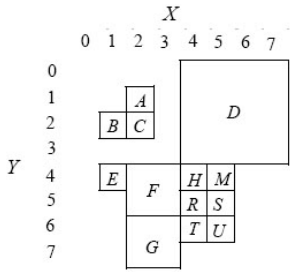
2. PRELIMINARIES

The quadtree of a $2^N \times 2^N$ image is built as follows. The root node of the tree denotes the whole image. If the image is in one color, then it is represented by the root node. Otherwise, four sons are added. Each son represents one-quarter of the image, as northwest, northeast, southwest and southeast. If the subdivided image is the same color, then the process is ended; otherwise, the process is repeated recursively for the son. In summary, the quadtree representation is obtained by recursively subdividing the image into quadrents, subquadrants, and so on, until the maximal block with uniform color is obtained.

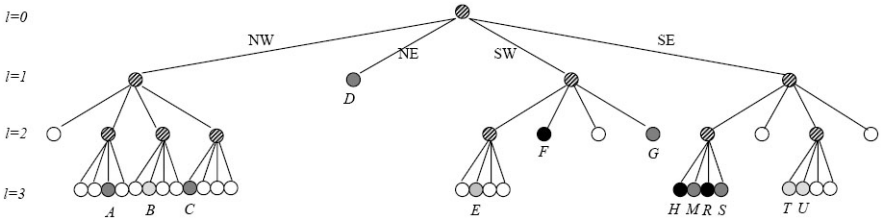
Given a $2^3 \times 2^3$ image with four gray levels as shown in Fig. 1 (a), the corresponding image blocks and quadtree are also shown in Figs. 1 (b) and (c), respectively. The linear quadtree of the image can thus be obtained by traversing the quadtree in preorder, collecting the leaf nodes, and denoting each non-white leaf node by a unique bit string. The organization of the bit string is different in the proposed linear quadtrees [4], including FD, FL, and VL locational codes. Although the encoding schemes are different, all of the bit strings are formed by the *location* of the image block residue, the *size* of the image block and its *color*. The *location* and the *size* of the image can be replaced by the *path* from the root to the node and the *level* of the node in the corresponding quadtree, respectively. They are equivalent [4].



(a) A $2^3 \times 2^3$ example image.



(b) The corresponding image blocks of (a).



(c) The corresponding quadtree of (a).

Fig. 1. A $2^3 \times 2^3$ example image and its corresponding image blocks and quadtree.

A general format is adopted throughout this paper to describe the construction method. However, the method can be adapted and applied for the proposed linear quadtrees (bintrees). Each image block is represented by a record of type, <path, level, color>. This record type is called a linear code in the rest of this paper. In a linear code, the path, level, and color denote the path from the root to the node, the level of the node and the color of the node, respectively. In the path code, 0, 1, 2 and 3 are adopted to denote the northwest, northeast, southwest and southeast quadrants, respectively. For instance, the path code of block *A* in Fig. 1 (b) is $(012)_4$ (= $(000110)_2$). The resulting number is the same as that obtained by interleaving the bits that comprise the values of the pixel's *x* and *y* coordinates in its upper left corner, which is called the image block's location. Given a $2^N \times 2^N$ image, if an image block at location (*x*, *y*), its location code is given by $(b_{N-1}a_{N-1}\ b_{N-2}a_{N-2}\ldots b_1a_1b_0a_0)_2$ where $x = (a_{N-1}a_{N-2}\ldots a_1a_0)_2$ and $y = (b_{N-1}b_{N-2}\ldots b_1b_0)_2$. The location of block *A* in Fig. 1 (b) is (2, 1). Thus $2 = (010)_2$ and $1 = (001)_2$, and the path code of block *A* is $(000110)_2$.

The first method for constructing quadtree is the naive algorithm, in which the pixels of the image are scanned in row-major order. The quadtree is initially comprised of a single "white" leaf node covering the whole image. If the encountered pixel is part of an existing leaf of the same color, then no change is made for the quadtree; otherwise, the corresponding quadtree is updated by a insert routine. Meanwhile, four sons with the same color may then be merged to produce a father leaf may then occur. The linear quadtree can thus be obtained by traversing and accumulating the non-white leaf nodes of the quadtree in preorder after the quadtree was built [23].

Based on the above description, the image shown in Fig. 1 (a) is denoted by <($(012)_4$, 3, 2); ($(021)_4$, 3, 1); ($(030)_4$, 3, 2); ($(100)_4$, 1, 2); ($(201)_4$, 3, 1); ($(210)_4$, 2, 3); ($(230)_4$, 2, 2); ($(300)_4$, 3, 3); ($(301)_4$, 3, 2); ($(302)_4$, 3, 3); ($(303)_4$, 3, 2); ($(320)_4$, 3, 1); ($(321)_4$, 3, 1)>.

Shaffer *et al.* later presented an improved method [21] to build the linear quadtree

directly. It also scans the image in row-major order. In the method, an array called the active node array is maintained to keep the nodes currently being processed, and insert the maximal nodes without merging routines. The method is a significant improvement. However, the insertion process in the active nodes array is still as complicated as before, and disc space requirement is relatively high [22].

Further, an improved algorithm has been presented [22]. The method eliminates many merging routines and complicated leaf insertion activities, since it assumes that the input pixels are in Morton scan order. It does not need a post-process to rearrange the sequence of the final codes to preserve their order. However, the method needs to be stored on disc [22] before it is encoded, meaning the image cannot be processed in real time. Morton scan order is not a natural image input format.

3. PROPERTIES AND THEOREMS

Some important properties for calculating the linear codes and preserving their order are presented as follows. These will be used in the proposed algorithm.

Theorem 1 Given a pixel of path code $(b_{N-1}a_{N-1}b_{N-2}a_{N-2}\ldots b_1a_1b_0a_0)_2$ at location (x, y), where $x = (a_{N-1}a_{N-2}\ldots a_1a_0)_2$ and $y = (b_{N-1}b_{N-2}\ldots b_1b_0)_2$, the path code of the pixel at location $(x \pm \Delta x, y \pm \Delta y)$ is denoted as $((((b_{N-1}a_{N-1}b_{N-2}a_{N-2}\ldots b_1a_1b_0a_0)_2 \vee (\underbrace{0101....0101}_{2N})_2) \pm (d_{N-1}fd_{N-2}f\ldots d_1fd_0f)_2) \wedge (\underbrace{1010....1010}_{2N})_2) \vee ((((b_{N-1}a_{N-1}b_{N-2}a_{N-2}\ldots b_1a_1b_0a_0)_2 \vee (\underbrace{1010....1010}_{2N})_2) \pm (fc_{N-1}fc_{N-2}\ldots fc_1fc_0)_2) \wedge (\underbrace{0101....0101}_{2N})_2)$, where $\Delta x = (c_{N-1}c_{N-2}\ldots c_1c_0)_2$, $\Delta y = (d_{N-1}d_{N-2}\ldots d_1 d_0)_2$, and $f = 0$ and 1 if addition and subtraction operations, respectively, applied in the aforementioned equation.

Proof: $(b_{N-1}a_{N-1}b_{N-2}a_{N-2}\ldots b_1a_1b_0a_0)_2 \vee (\underbrace{0101....0101}_{2N})_2$ preserves y and sets the other bits as 1. Then, $y + \Delta y$ is calculated as $(((b_{N-1}a_{N-1}b_{N-2}a_{N-2}\ldots b_1a_1b_0a_0)_2 \vee (\underbrace{0101....0101}_{2N})_2) + (d_{N-1}0d_{N-2}0\ldots d_10d_00)_2)$ while $y - \Delta y$ is calculated as $(((b_{N-1}a_{N-1}b_{N-2}a_{N-2}\ldots b_1a_1b_0a_0)_2 \vee (\underbrace{0101....0101}_{2N})_2) - (d_{N-1}1d_{N-2}1\ldots d_11d_01)_2)$. Thus, $((((b_{N-1}a_{N-1}b_{N-2}a_{N-2}\ldots b_1a_1b_0a_0)_2 \vee (\underbrace{0101....0101}_{2N})_2) \pm (d_{N-1}fd_{N-2}f\ldots d_1fd_0f)_2) \wedge (\underbrace{1010....1010}_{2N})_2)$ preserves $y \pm \Delta y$, and sets the other bits to be 0. Likewise, $((((b_{N-1}a_{N-1}b_{N-2}a_{N-2}\ldots b_1a_1b_0a_0)_2 \vee (\underbrace{1010....1010}_{2N})_2) \pm (fc_{N-1}fc_{N-2}\ldots fc_1fc_0)_2) \wedge (\underbrace{0101....0101}_{2N})_2)$ calculates $x \pm \Delta x$ and sets the other bits to be 0. Finally, the path code of the pixel at location $(x \pm \Delta x, y \pm \Delta y)$ thus can be obtained by *OR*ing these two results. ❑

The calculation shown in Theorem 1 is clearly of a constant time. Theorem 1 indicates that given two pixels, denoted A_1 and A_2, the path code of A_2 can be obtained in constant time if the path code of A_1 and the distance between A_1 and A_2 are known.

For instance, in Fig. 1 (b), the path code of pixel in the left upper corner of the im-

age is $(000000)_2$ (= $(000)_4$) since $x = (000)_2$ and $y = (000)_2$. Considering the pixel in the left upper corner of D, it follows that $\Delta x = (100)_2$ and $\Delta y = (000)_2$. By Theorem 1, $((((000000)_2 \vee (010101)_2) + (000000)_2) \wedge (101010)_2) \vee (((((000000)_2 \vee (101010)_2) + (010000)_2) \wedge (010101)_2) = ((010101)_2 \wedge (101010)_2) \vee ((111010)_2 \wedge (010101)_2) = (010000)_2$ (= $(100)_4$).

The path code of the next non-white pixel, which is the pixel at (5, 0), can be obtained by $((((01000)_2 \vee (010101)_2) + (000000)_2) \wedge (101010)_2) \vee (((((010000)_2 \vee (101010)_2) + (00001)_2) \wedge (010101)_2) = ((010101)_2 \wedge (101010)_2) \vee ((111011)_2 \wedge (010101)_2) = (010001)_2$ (= $(101)_4$) since $\Delta x = (001)_2$ and $\Delta y = (000)_2$.

Property 1 Merging property.

Given four path codes, P_0, P_1, P_2, and P_3 at level l, $1 \le l \le N$ where

$P_0 = (b_{N-1}a_{N-1}b_{N-2}a_{N-2}\ldots b_{N-l-1}a_{N-l-1}0000\ldots00)_2$,
$P_1 = (b_{N-1}a_{N-1}b_{N-2}a_{N-2}\ldots b_{N-l-1}a_{N-l-1}0100\ldots00)_2$,
$P_2 = (b_{N-1}a_{N-1}b_{N-2}a_{N-2}\ldots b_{N-l-1}a_{N-l-1}1000\ldots00)_2$, and
$P_3 = (b_{N-1}a_{N-1}b_{N-2}a_{N-2}\ldots b_{N-l-1}a_{N-l-1}1100\ldots00)_2$,

if P_0, P_1, P_2, and P_3 have the same color, then they can be merged and represented by $(b_{N-1}a_{N-1}b_{N-2}a_{N-2}\ldots b_{N-l-1}a_{N-l-1}0000\ldots00)_2$ with level $l - 1$.

Blocks H, M, R, and S in Fig. 1 (b) are assumed to have uniform color, and thus can be merged. The path codes of H, M, R, and S are (110000), (110001), (110010), and (110011), respectively, and are at level 3. By Property 1, They can be merged and represented by (110000) at level 2, which is the linear code corresponding to the father of nodes H, M, R, and S in Fig. 1 (c).

A $2^3 \times 2^3$ image with Morton number is shown in Fig. 2. Based on the increasing property [1], the pixels must be in the sequence of <0, 1, 2, 3, …, 61, 62, 63>. Thus, an insertion approach is needed when the image is scanned in row major order. In the first row, the encountered pixel can be appended to the next of the previous encountered pixel. The sequence is <0, 1, 4, 5, 16, 17, 20, 21>. $P_{(x,y)}$ denotes the next pixel of $P_{(x-1,y)}$. Next, the first pixel of the second row, Pixel 2, is appended to the next of Pixel 1 and Pixel 3 is the next of Pixel 2. Then, Pixel 6 is appended to the next of Pixel 5, successively. After

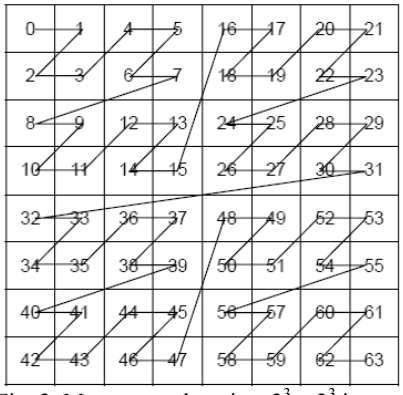


Fig. 2. Morton numbers in a $2^3 \times 2^3$ image.

the second row was processed, the sequence is <0, 1, 2, 3, 4, 5, 6, 7, 16, 17, 18, 19, 20, 21, 22, 23>. The following property is applied.

Property 2 Preceding pixel property.

If the pixels of the image are scanned and stored in row major order, then given a pixel $P_{(x,y)}$ in a $2^N \times 2^N$ image, where $P_{(x,y)}$ denotes the pixel at location (x, y) and $0 \leqq x, y \leqq 2^N - 1$, it follows that

if $y = 0$, then $P_{(x,y)}$ denotes the next pixel of $P_{(x-1,y)}$;

if $y \bmod 2^k = 2^{k-1}$ ($k = 1, 2, 3, 4, \ldots$), and if $x = (2^k \times z)$, where $z = 0, 1, 2, 3, \ldots$, then $P_{(x,y)}$ denotes the next pixel of $P_{(x+(2^k-1),y-1)}$; otherwise, $P_{(x,y)}$ denotes the next pixel of $P_{(x-1,y)}$.

In Fig. 2, the linear code of pixel $P_{(4,2)}$ is the next linear code of $P_{(7,1)}$ if the image is processed in row major order. It can be adapted in Property 2 by $k = 2$. Now, for maintaining the positions which the pixels of the next row are inserted, a pointer list is needed. Property 2 can be rewritten as follows.

Property 3 Position property.

Consider a $2^N \times 2^N$ image with Pixel $P_{(x,y)}$, where $0 \leqq x, y \leqq 2^N - 1$. Let Q_x denotes the xth memory space in Linking list Q, and I_x denote the xth pointer in Pointer list I. Let $I_x = x$, and save the linear code of $P_{(x,0)}$ into Q_x if $P_{(x,0)}$ is not initially white. To preserve the increasing property, the linear code of $P_{(x,y)}$, where $0 \leqq x \leqq 2^N - 1$ and $1 \leqq y \leqq 2^N - 1$, must be inserted into Q_p if Q_p is empty, where

$$p = \begin{cases} I_{x+2^k-1} & \text{if } y \bmod 2^k = 2^{k-1} (k = 1, 2, 3, 4, \ldots) \text{ and } x = (2^k \times z), \text{ where } z = 0, 1, 2, 3, \ldots, \\ I_{x-1} & \text{otherwise;} \end{cases}$$

the value of I_x is adjusted as p. If Q_p already contains a linear code, then the linear code of $P_{(x,y)}$ is inserted into the next value of Q_p. The address of the next of Q_p is saved in I_x. Notably, if $P_{(x,y)}$ is white, then the value of I_x is adjusted as p defined above.

To reduce the calculation effort, Position property can be implemented as a lookup table to obtain the p value. The size is $2^N \times N$, since there are N ($= \log_2 2^N$) different row types. The lookup table for a $2^3 \times 2^3$ image is shown in Table 1. Given a $2^3 \times 2^3$ image, there are three different types beside Row 0 ($y = 0$). Based on Position property, Row 1, 3, 5, and 7 are the first type, which is Type 1 in Table 1, where "←" represents that the value is obtained from the left column. In this type, the linear codes of $P_{(x,y)}$ are inserted into the next of I_{x+1} if $x = 0$, 2, 4, and 6. The linear codes of the other pixel are inserted into the next of the previous pixel residue. The second type, which is Type 2 in Table 1, is the

Table 1. The lookup table for a $2^3 \times 2^3$ image.

		I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7
Type 1	$P =$	I_1	←	I_3	←	I_5	←	I_7	←
Type 2	$P =$	I_3	←	←	←	I_7	←	←	←
Type 3	$P =$	I_7	←	←	←	←	←	←	←

pixels at Row 2 and 6. In this type, the linear codes of $P_{(x,y)}$ are inserted into the next of I_{x+3} if $x = 0$ and 4 and the others are inserted into the next of the previous pixel residue. Finally, the third type, which is Type 3 in Table 1, comprises the pixels at Row 4. The linear code of $P_{(x,y)}$ is inserted into the next of I_{x+7} if $x = 0$. Moreover, the bit strings denoting Δx and Δy, $(fc_{N-1}fc_{N-2}\ldots fc_1fc_0)_2$ and $(d_{N-1}fd_{N-2}f\ldots d_1fd_0f)_2$, described in Theorem 1 can also be applied by the lookup table.

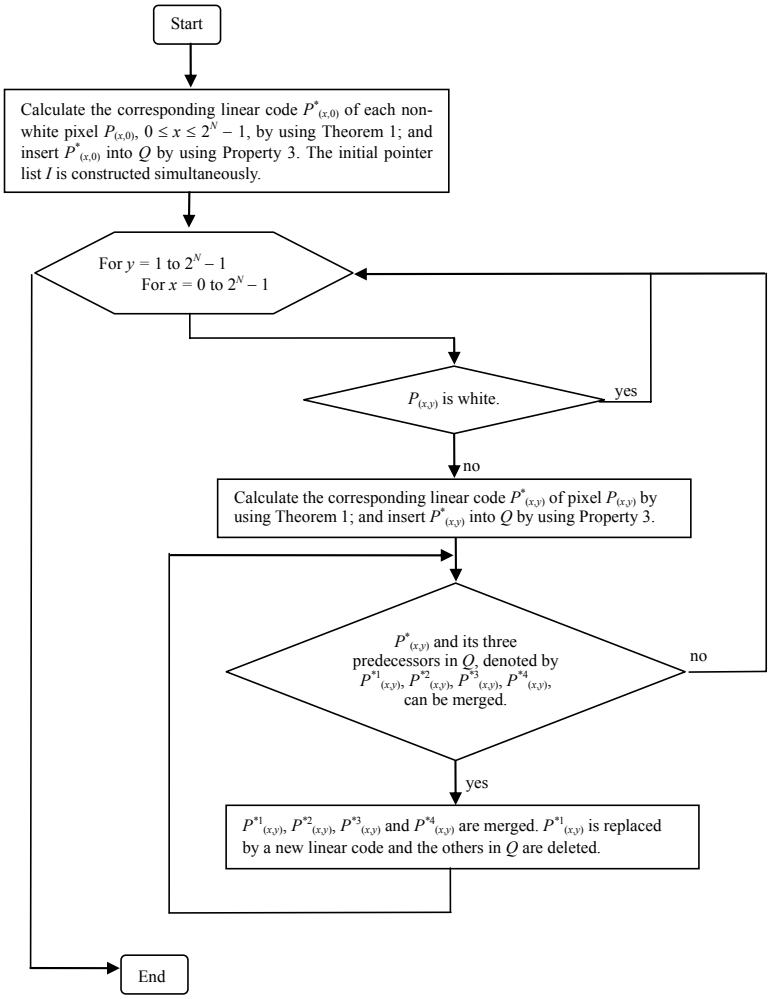


Fig. 3. The block diagram for the proposed method.

4. PROPOSED METHOD

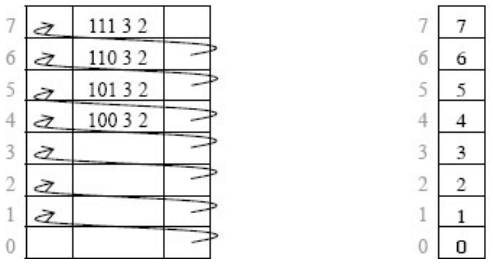
The proposed method calculates the linear code of each encountered non-white pixel according to Theorem 1. The linear code is then inserted into the linear quadtree by the method described in Position property. A merge routine based on Merging property is then

performed if necessary. The resulting linear quadtree of the image can thus be obtained after all pixels are processed. Notably, I_x should be updated simultaneously for each encountered pixel to conform to Position property. The method can be summarized as a block diagram shown in Fig. 3.

The construction method is then simulated by the image shown in Fig. 1. An empty result linear quadtree (Q) and a pointer list (I) are first created. Both of them are initially of size 2^N. The size of the result linear quadtree is variable during the encoding process, and that of the pointer list is fixed. Notably, the image is of size $2^N \times 2^N$.

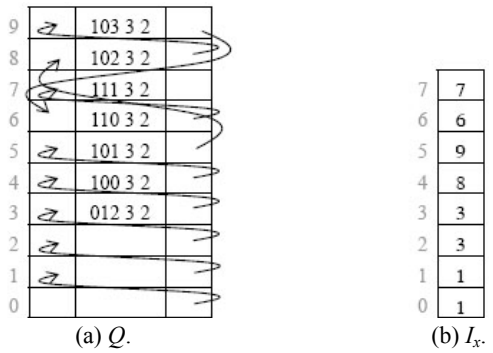
In the first row, Row 0, the linear code of $P_{(x,0)}$ is first put into position x in the resulting linear quadtree if it is not white. Pointer I_x is set to x. Fig. 4 shows the result. In Fig. 4 (a), each row has three columns. The middle column is the linear code represented by the path, level and color. The right column and the left column are the physical addresses of its next and previous linear code. For simplicity, they are represented by arrows in the remainder of this study. Fig. 4 (b) shows the pointer list.

The second row is then processed successively. $P_{(0,1)}$, which is the pixel at (0, 1), is encountered. It is white, and thus does not have to be encoded. Then, Pointer I_0 is set to be 1 by Property 3. Similarly, the next pixel $P_{(1,1)}$ is white. Pointer I_1 is set to be 1 by Property 3. Next, $P_{(2,1)}$ is encountered. It is not white. By Theorem 1, its linear code is (012 3 2). By Property 3, (012 3 2) should be put into the position of Pointer I_3. Hence, (012 3 2) is put into position 3 in Fig. 5 (a). Simultaneously, by Property 3, Pointer I_2 is set to be 3. See Fig. 5 (b).



(a) Q. (b) I_x.

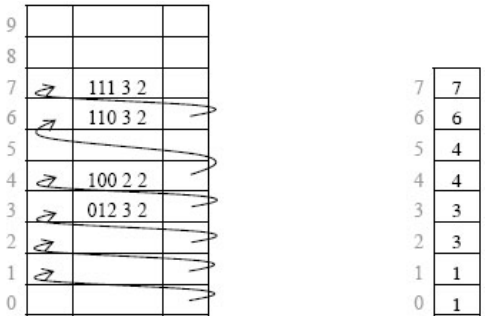
Fig. 4. The resulting linear quadtree and the pointer list after $P_{(7,0)}$ is processed.



(a) Q. (b) I_x.

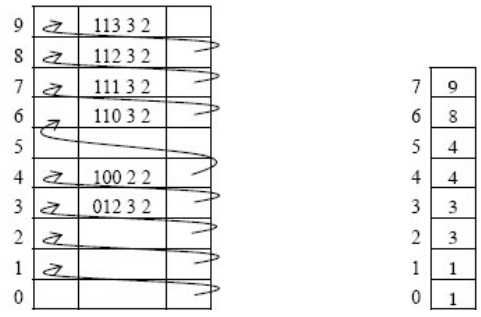
Fig. 5. The resulting linear quadtree and the pointer list after $P_{(5,1)}$ is inserted.

$P_{(3,1)}$ is processed successively, and is white. Likewise, I_3 in Fig. 5 (b) is set to be 3 (= I_2). $P_{(4,1)}$ is not white. By Theorem 1, its linear code is (102 3 2). By Property 3, (102 3 2) should be put into the position after the occupied position Q_{I_5} (= Q_5). Thus, (102 3 2) is put into position 8; the related links is modified properly, and I_4 is set to be 8. See Fig. 5 (b). Likewise, the next linear code, (103 3 2), is put into position 9, and I_5 is set to be 9. Fig. 5 shows the result up to this point.



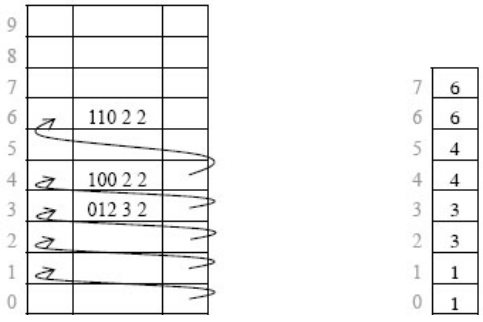
(a) Q. (b) I_x.

Fig. 6. The result of Fig. 5 after merging.



(a) Q. (b) I_x.

Fig. 7. The resulting linear quadtree and the pointer list after $P_{(7,1)}$ is inserted.



(a) Q. (b) I_x.

Fig. 8. The result of Fig. 7 after merging.

Based on Property 1, (100 3 2), (101 3 2), (102 3 2), and (103 3 2) can be merged and represented by (100 2 2). Fig. 6 shows the result after merging. The associated I_x are updated concurrently. I_4 and I_5 are set to be at the physical address of (100 2 2) residue.

Likewise, Fig. 7 shows the linear quadtree and associated physical addresses after $P_{(7,1)}$ is inserted, and Fig. 8 shows the result after merging.

The final linear quadtree thus can be obtained directly by retrieving the linear codes in the result linear quadtree list after all pixels of the image were processed. The obtained result for the image shown in Fig. 1 (a) is <(012 3 2) (021 3 1) (030 3 2) (100 1 2) (201 3 1) (210 2 3) (230 2 2) (300 3 3) (301 3 2) (302 3 3) (303 3 2) (320 3 1) (321 3 1)>.

The algorithm is summarized below.

Algorithm Building the linear quadtree
Input: $N \leftarrow$ the resolution of the image.
$P^*_{(0,0)} \leftarrow (\underbrace{0000 \ldots 0000}_{2N})_2$ /* $P^*_{(x,y)}$ denotes the liner code of $P_{(x,y)}$ in the algorithm. */
$Q \leftarrow \varnothing$
1: $y \leftarrow 0$
2: $\Delta y \leftarrow 0$; $\Delta x \leftarrow -1$
3: For $x = 0$ to $2^N - 1$
4: $\Delta x \leftarrow \Delta x + 1$
5: If $P_{(x,0)}$ is not white,
6: calculate $P^*_{(x,0)}$ by using Theorem 1 and insert $P^*_{(x,0)}$ into Q by using Property 3;
7: $\Delta x \leftarrow 0$;
8: endif
9: $I_x \leftarrow x$.
10: endfor
11: $\Delta y \leftarrow -1$
12: For $y = 1$ to $2^N - 1$
13: $\Delta y \leftarrow \Delta y + 1$
14: $\Delta x \leftarrow -1$
15: For $x = 0$ to $2^N - 1$
16: $\Delta x \leftarrow \Delta x + 1$
17: If $P_{(x,y)}$ is not white,
18: calculate $P^*_{(x,y)}$ by using Theorem 1 and insert $P^*_{(x,y)}$ into Q by using Property 3;
19: $\Delta x \leftarrow 0$; $\Delta y \leftarrow 0$
20: $I_x \leftarrow$ the address where $P^*_{(x,y)}$ is saved in Q
21: If $P^*_{(x,y)}$ and its three predecessors in Q, denoted by $P^{*1}_{(x,y)}$, $P^{*2}_{(x,y)}$, $P^{*3}_{(x,y)}$ and $P^{*4}_{(x,y)}$, can be merged by applying Merging property,
22: $x_e \leftarrow$ the x value of I_x which corresponding to $P^{*2}_{(x,y)}$
23: While $P^{*1}_{(x,y)}$, $P^{*2}_{(x,y)}$, $P^{*3}_{(x,y)}$ and $P^{*4}_{(x,y)}$ can be merged,
24: $P^{*1}_{(x,y)}$ is replaced by a new linear code based on Property 1 and the others in Q are set to be empty.
25: $x_s \leftarrow$ the x value of I_x which corresponding to $P^{*1}_{(x,y)}$
26: endwhile
27: $<I_{x_s}, I_{x_e}> \leftarrow x_s$
28: endif

29: Otherwise,
30: $I_x \leftarrow p$ shown in Eq. (1).
31: endif
32: endfor
33: endfor

The *For-loop* in Statement 3 of the proposed algorithm calculates the linear codes of $P_{(x,0)}$, which are the pixels in the first row. I_x is saved simultaneously in Statement 9. The nested *For-loops* in Statements 12 and 15 calculate the linear codes of the pixels from $P_{(0,1)}$ to $P_{((2^N-1)\times(2^N-1))}$. Once a linear code is calculated, the liner code and its three predecessors in Q are checked whether they can be merged. If so, these four liner codes are merged in Statement 23 and the corresponding I_x is updated in Statement 27.

The steps for inserting and merging the linear codes shown in the algorithm can be easily applied [24] since the data structure of the result linear quadtree is a list. The step for calculating I_x can be performed based on Position property by the lookup table described above.

5. EXPERIMENTATIONS

Some experiments were performed to measure the performance of the algorithm. Three images, with respect to binary, gray level and color, were scanned to resolutions $2^7 \times 2^7$ and $2^8 \times 2^8$. Figs. 9, 10 and 11 show the images with resolution $2^8 \times 2^8$. The images shown in Figs. 10 and 11 are of 16 gray levels and 64 colors, respectively. The programs were coded in the *C* programming language, and executed on a personal computer with a 3.0 MHz processor.

Fig. 9. City map.

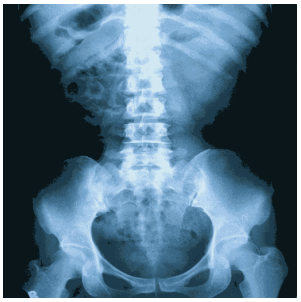


Fig. 10. An X-ray picture.

Fig. 11. A fish.

Two methods namely the algorithm proposed in [21], called Algorithm_1, and the proposed algorithm were measured in the experiment. Table 2 shows the experimental results. The method presented in [22] is not included because it is of a different input format.

In summary, the proposed method in the paper can be divided into two parts, namely maintaining the resulting linear quadtree and maintaining the pointer list. Experimental results indicate that the lookup table is useful.

Table 2. Experimental results.

Image	Resolution ($2^N \times 2^N$)	Size of linear codes	CPU time in 10^{-2} second(s)		
			Algorithm_1	The proposed method	
				Without lookup table	With lookup table
Fig. 8 (City)	$N = 7$	3706	6.0	5.9	4.6
	$N = 8$	15149	23.3	19.8	15.3
Fig. 9 (X-ray)	$N = 7$	33002	39.6	9.1	7.1
	$N = 8$	132271	154.5	35.9	28.5
Fig. 10 (Fish)	$N = 7$	11809	18.3	5.2	4.1
	$N = 8$	48039	67.7	15.4	12.2

6. CONCLUSION

This study presents a novel method for building the linear quadtree from a given image. From the theoretical point of view, the time complexity for encoding an image is at least of $O(2^N \times 2^N)$ if the size of the image is $2^N \times 2^N$, since each pixel in the image should be checked regardless of its color. Both the proposed method and that of [21] belong to this type. Moreover, an algorithm with good empirical performance is required. The proposed method has been indicated to be simple, easy and efficient. Moreover, the image can be encoded in real time. The proposed method does not require a large disk space either to save the input pixels or to maintain a complex data structure.

Results of this study demonstrate that the usage of the lookup table is useful. However, additional space is required for storing these tables before the image is encoded. This is a trade-off in implementation.

REFERENCES

1. C. Y. Huang and K. L. Chung, "Fast operations on images using interpolation-based bintrees," *Pattern Recognition*, Vol. 28, 1995, pp. 409-420.
2. C. A. Shaffer, R. Juvvadi, and L. S. Health, "Generalized comparison of quadtree and bintree storage requirements," *Image and Vision Computing*, Vol. 11, 1993, pp. 402-412.
3. H. Samet and M. Tamminen, "Computing geometric properties of images represented by linear quadtrees," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 7, 1985, pp. 229-240.
4. H. Samet, *Applications of Spatial Data Structures*, Addison Wesley, Reading, Massachusetts, 1990.
5. G. Schrack, "Finding neighbors of equal size in linear quadtrees and octrees in constant time," *CVGIP: Image Understanding*, Vol. 55, 1992, pp. 221-230.
6. C. Y. Huang and K. L. Chung, "Faster neighbor finding on image represented by bincodes," *Pattern Recognition*, Vol. 29, 1996, pp. 1507-1518.
7. K. L. Chung and C. Y. Huang, "Finding neighbors on bincode-based images in $O(n \log \log n)$ time," *Pattern Recognition Letters*, Vol. 17, 1996, pp. 1117-1124.
8. H. Samet and M. Tamminen, "Efficient component labeling of images of arbitrary

dimension represented by linear bintrees," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 10, 1988, pp. 578-586.
9. G. Schrack and I. Gargantini, "Mirroring and rotating images in linear quadtree from with few machine instructions," *Image and Vision Computing*, Vol. 11, 1993, pp. 112-118.
10. L. S. Wilke and G. F. Schrack, "Improved mirroring and rotation functions for linear quadtree leaves," *Image and Vision Computing*, Vol. 13, 1995, pp. 491-495.
11. D. C. Mason, "Dilation algorithm for a linear quadtree," *Image and Vision Computing*, Vol. 5, 1987, pp. 11-20.
12. K. L. Chung and J. G. Wu, "Fast implementations for mirroring and rotating bincode-based images," *Pattern Recognition*, Vol. 30, 1998, pp. 1961-1967
13. C. Y. Huang and K. L. Chung, "Transformations between bincodes and the DF-expression," *Computers and Graphics*, Vol. 19, 1995, pp. 601-610.
14. C. Y. Huang and K. L. Chung, "Manipulating images by using run-length Morton codes," *International Journal of Pattern Recognition and Artificial Intelligence*, Vol. 11, 1997, pp. 889-907.
15. Z. Chen and I. P. Chen, "A simple recursive method for converting a chain code into a quadtree with a lookup table," *Image and Vision Computing*, Vol. 19, 2001, pp. 413-426.
16. F. Y. Shih and W. T. Wong, "An adaptive algorithm for conversion from quadtree to chain codes," *Pattern Recognition*, Vol. 34, 2001, pp. 631-639.
17. Y. K. Chan and C. C. Chang, "Block image retrieval based on a compressed linear quadtree," *Image and Vision Computing*, Vol. 22, 2004, pp. 391-397.
18. P. M. Chen, "Variant code transformations for linear quadtrees," *Pattern Recognition Letters*, Vol. 23, 2002, pp. 1253-1262.
19. Y. C. Hu and J. H. Jiang, "Low-complexity progressive image transmission scheme based on quadtree segmentation," *Real-Time Imaging*, Vol. 11, 2005, pp. 59-70.
20. K. L. Chung, Y. W. Liu, and W. M. Yan, "A hybrid gray image representation using spatial-and DCT-based approach with application to moment computation," *Journal of Visual Communication and Image Representation*, Vol. 17, 2006, pp. 1209-1226.
21. C. A. Shaffer and H. Samet, "Optimal quadtree construction algorithms," *Computer Vision*, *Graphics*, *and Processing*, Vol. 37, 1987, pp. 402-419.
22. F. C. Holroyd and D. C. Mason, "Efficient linear quadtree construction algorithm," *Image and Vision Computing*, Vol. 8, 1990, pp. 218-224.
23. H. Samet, "An algorithm for converting rasters to quadtrees," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 3, 1981, pp. 93-95.
24. Y. Langsam, M. J. Augenstein, and A. M. Tenbaum, *Data Structures using C and C++*, Prentice Hall, Upper Saddle River, New Jersey, 1996.

Chi-Yen Huang (黃其彥) received the M.S. and Ph.D. degrees in the Department of Industrial Management and Information Management, respectively, from National Taiwan University of Science and Technology. He is currently an Associate Professor in Department of Information Management, National Taipei College of Business, Taipei, Taiwan, R.O.C. His current research interests include spatial data structures, database, and software engineering.

Yu-Wei Chen (陳育威) received the B.S. and Ph.D. in the Department of Information Management from National Taiwan University of Science and Technology, Taipei, Taiwan, in 1993 and 1999, respectively. He is currently an Assistant Professor in Graduate Institute of Information and Logistics Management, National Taipei University of Technology, Taipei, Taiwan. His current research interests include video-on-demand, peer to peer, multimedia information network, fault tolerance, and parallel processing.