Efficient Rotating and Mirroring Operations on the Image Data Based on CBLQ *

Ye-In Chang, Hue-Ling Chen, Yuan-Shu Hsiao, and Ta-Wei Liu
Dept. of Computer Science and Engineering
National Sun Yat-Sen University
Kaohsiung, Taiwan, R.O.C
E-mail: changyi@cse.nsysu.edu.tw

Abstract

The *Constant Bit-Length Linear Quadtree (CBLQ)* has been proposed as one of well-known encoding schemes for representing binary images. It not only saves the storage efficiently, but also keeps the level of detail property for images. Based on CBLQ, set operations on images can be easily derived. In this paper, we propose efficient strategies for rotating and mirroring images based on CBLQ. Our strategies can obtain the code of the rotated or mirrored image directly from the code of the original image, instead of from the rotated/ mirrored image. From our simulation, we show that the CPU-time for all rotating or mirroring cases are the same by using our strategy based on CBLQ.

Keywords: CBLQ, mirroring, quadtree, rotation, spatial data

1. Introduction

Representation and manipulation of digital binary images are two important tasks in image processing, pattern recognition, pictorial database, computer graphics, geographic information systems, and the other related applications [3, 5, 6, 7, 8, 9, 11, 12]. A *Quadtree* [8, 10] is one well-known representation of hierarchical data structures for representing the binary image. Take Figure 1 for example. A binary image in Figure 1-(a) is successively sub-divided into four equal-size sub-images in the NW (northwest), NE (northeast), SW (southwest), and SE (southeast) quadrants. A homogeneously colored quadrant is represented by a leaf node in the quadtree. Otherwise, the quadrant is represented by an internal node and further divided into four subquadrants until each subquadrant has the same color. The corresponding quadtree for this binary image is shown in Figure 1-(b).

Since the quadtree uses a lot of pointer data, it consumes a considerable amount of storage and possibly result in increase of processing time due to high rate of page faults. A linear quadtree has been proposed as a linear array of certain type of structure elements such that the tree structure is implicitly preserved [4]. It is one kind of the set-of-codes representation which represents the colored quadrants as a set of numbers. The *Constant Bit-length Linear Quadtrees (CBLQ)* [4] is proposed to keep the level of detail property for the linear quadtree. In the CBLQ representation, the code length of each interesting structure element is constant, instead of variant in length, according to the number of elements. Based on CBLQ representation, the operations can be performed on images efficiently because the information of images can be easily obtained [4].

Rotating and mirroring operations are two important operations on image manipulation [2, 6, 7, 11, 12]. Basically, there are three rotating cases: **Rotate_90**, **Rotate_180**, **Rotate_270**. These cases are equivalent to rotating of 270^o, 180^o, and 90^o about the center in the anticlockwise direction, respectively. On the other hand, there are four mirroring cases [2] which are described as follows. The **Mirror_X** case and the **Mirror_Y** case mean mirroring about X-axis and Y-axis, respectively. The **Mirror_MD** case means mirroring about the main diagonal from the SW to NE direction, where **MD** means the main diagonal. The **Mirror_CD** case means mirroring about the cross diagonal from the SE to NW direction, where **CD** means the cross diagonal.

If we want to retrieve the image which is ro-

*This research was supported in part by the National Science Council of Republic of China under Grant No. NSC-93-2213-E-110-027.

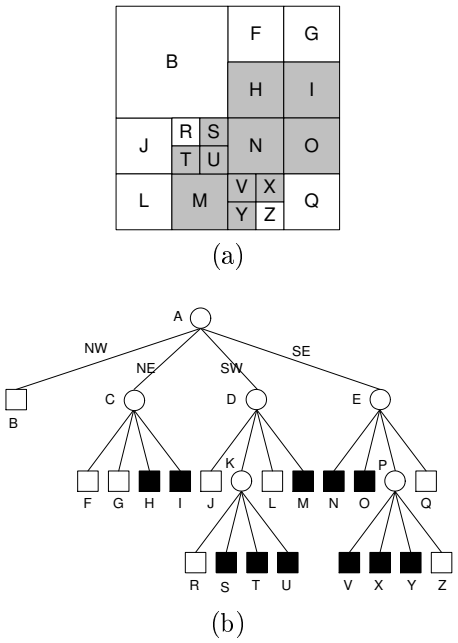

Figure 1: An example of the quadtree: (a) the sample image; (b) its quadtree.

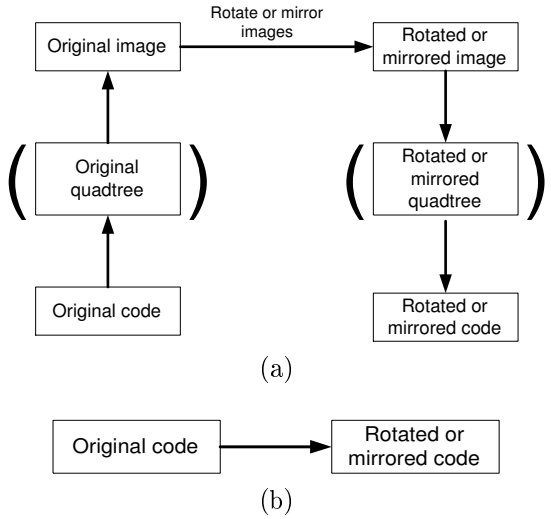

Figure 2: The process for obtaining the rotated or mirrored code: (a) the original three steps ; (b) our proposed strategy.

tated or mirrored in the database, we must derive the code of the new image which has been rotated or mirrored from the original image. That is, to achieve this goal, originally, we must perform three steps as shown in Figure 2-(a). First, we must derive the original image A from the original code. Second, we rotate or mirror the original image A to the new one B. Third, we derive the related code from the new image B. The process for obtaining the rotated or mirrored code is time-consuming. Therefore, it is more efficient to obtain the code of the rotated or mirrored image from the code of the original one directly. Therefore, we propose the strategies to obtain the rotated or mirrored code directly from the original one, which is the efficient process shown in Figure 2-(b). Since the CBLQ has good compression improvement to represent and manipulate the images efficiently, we propose the strategies for rotating and mirroring images based on the CBLQ representation.

The rest of this paper is organized as follows. In Section 2, we briefly describe the Constant Bit-length Linear Quadtrees (CBLQ) [4]. In Section 3, we present one strategy to obtain the codes of the rotated images represented by CBLQ. In Section 4, we present another one strategy to obtain the codes of the mirrored images represented by CBLQ. In Section 5, we show the performance study. Finally, we give the conclusion.

2. Constant Bit-Length Linear Quadtree (CBLQ)

It is assumed that the image is a square binary $2^n \times 2^n$ array composed of unit square pixels that can be black or white, where n is referred as the *resolution* parameter in the horizontal or vertical direction. A CBLQ representation of a picture is obtained by traversing its corresponding quadtree in the breadth-first order [4]. In the traversal, a special digit is obtained from each node according to its attribute. If the node is a leaf node, then digit 1(0) is obtained if it is a black (white) node. If the node is an internal node, then digit 2 is taken unless all its children are leaf nodes. Digit 3 is gained if the node is an internal node and its children are leaf nodes. For the convenience, digits 0 and 1 (2 and 3) are called leaf (internal) labels. Take Figure 3 as an example, the corresponding CBLQ representation of Figure 3-(a) is 1222 0303 3311 0301 1010 1010 1100 1100 0011, which is obtained by traversing the quadtree in Figure 3-(b) in the breadth-first order.

3. The Strategy to Rotate Binary Images Represented by CBLQ

In this section, we present the strategy which maps the original image to the rotated one directly from the codes of the original image based on the CBLQ representation [4]. Generally, we assume that the input image is a $2^n \times 2^n$ binary image, where n is the resolution parameter in the horizontal and vertical directions. Based on CBLQ,

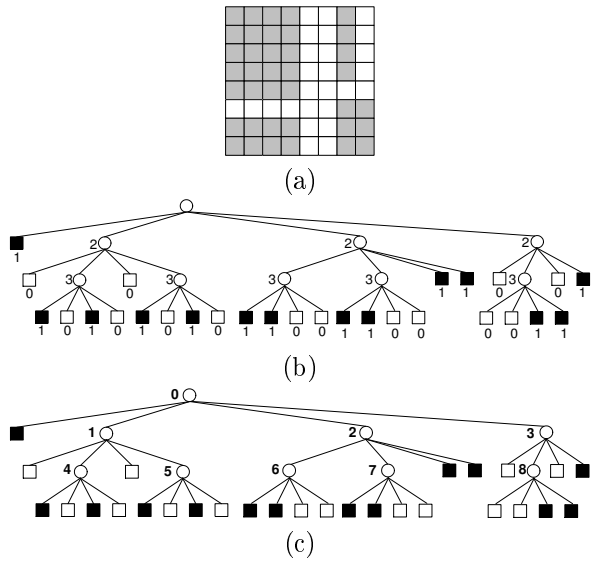

Figure 3: An example of the CBLQ representation: (a) the example image; (b) the corresponding quadtree with the labels; (c) the corresponding quadtree with the ID numbers of the internal nodes.

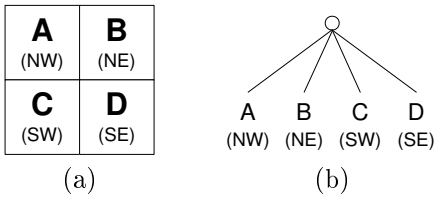

Figure 4: Definition of four blocks: (a) blocks A, B, C, and D; (b) the corresponding quadtree.

we represent four quadrants in the quadtree as four blocks. It means that the NW, NE, SW, and SE quadrants are represented as blocks A, B, C, and D, respectively, as shown in Figure 4-(a). The order for blocks A, B, C, and D is called a *Morton order* [4]. Moreover, blocks A, B, C, and D are the first, second, third, and fourth children of the root node (or the internal node) from the left to right in the corresponding quadtree, respectively. Figure 4-(b) shows the location of blocks A, B, C, and D in the corresponding quadtree, which are positions A, B, C, and D.

In our strategy, we define one data structure, *L_Group*, which will be used for code conversion. From the view point of the quadtree and the CBLQ representation, every four labels from the same parent node can be grouped together in one *L_Group*. We assume that the ID of the root node is 0, where ID means the identification for the internal node. The ID of *L_Group* is equal to the ID of the parent node. In Figure 3-(c), each ID of the internal node shown is assigned in the breath-first order by increasing the previous ID by 1. Take Figure 3-(b) as an example. The root node is assumed at the zeroth level. The four labels 0, 3, 0, and 3 at the second level can be grouped into a *L_Group*. They are derived from the same parent node which is labeled with 2 at the first level. Moveover, since the ID of this parent node is 1 after the $ID = 0$ for the root node, the ID of this *L_Group* is 1 shown in Figure 3-(c).

There is a property for the rotating cases: the image and the corresponding quadtree will be changed at the same time during the rotating process. When the image is rotated, its corresponding quadtree will be traversed in the breadth-first order. At the same time, the nodes in this quadtree will be moved to the new position. Take Figure 5-(a) as an example. The four children of N_1, are labelled as 1, 2, 3, and N_2 at the first level in Figure 5-(b). According to Figure 4-(b), they locate at positions A, B, C, and D, respectively, where N_i denotes an internal node. After being rotated by 90 degrees clockwise, their location have been changed from blocks A, B, C, and D in Figure 5-(a) to blocks B, D, A, and C in Figure 5-(c). It means that the four children at the first level in Figure 5-(b) are changed their positions A, B, C, and D to positions B, D, A, and C, respectively, in Figure 5-(d). The four children at the first level in Figure 5-(d) are labels with 3, 1, N_2, and 2 on positions A, B, C, and D, respectively. The process of changing the positions of four children is executed at each level until the end of the quadtree. It is similar to the other rotating cases.

From the examples shown as above, we conclude the mapping rules which are used for changing the positions in three rotating cases.

- Rule $R1$. (**Rotate_90**):
  $C \rightarrow A$; $A \rightarrow B$; $D \rightarrow C$; $B \rightarrow D$.
- Rule $R2$. (**Rotate_180**):
  $D \rightarrow A$; $C \rightarrow B$; $B \rightarrow C$; $A \rightarrow D$.
- Rule $R3$. (**Rotate_270**):
  $B \rightarrow A$; $D \rightarrow B$; $A \rightarrow C$; $C \rightarrow D$.

Note that in the mapping rules, we always maintain the order A, B, C, and D in the result (the right part of the mapping rule, *i.e.*, the right part of "$\rightarrow$"). Therefore, the order of the process in each mapping rule is important.

We take Figures 6-(a) and (b) as examples to illustrate the image of Rotate_90 case which uses

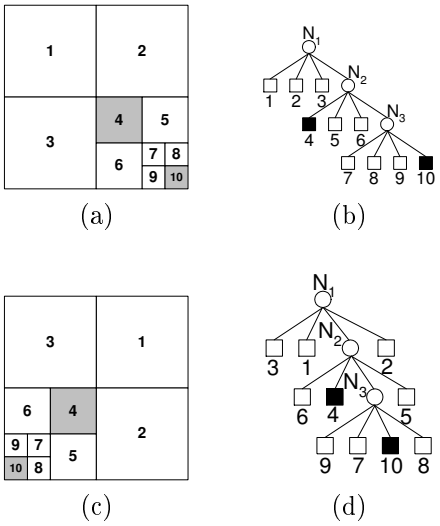

Figure 5: An example for the rotating case: (a) the original image; (b) the quadtree for the original image; (c) the image after being rotated by 90 degrees clockwise; (d) the quadtree for the image after being rotated.

the mapping rule $R1$. According to rule $R1$, The sixteen data blocks in block A in Figure 6-(a) are moved, as the result of the sixteen data blocks shown in block B in Figure 6-(b). It is similar to the sixteen data blocks in each block B, C, and D. They are moved to blocks D, A, and C, respectively. The other mapping rules which are used for the other cases can be easily derived in the similar way. The images of other rotating cases which use different mapping rules are shown in Figures 6-(c) and(d).

Although we use the image to illustrate how the mapping rule used in one case, actually, we change the CBLQ representation of the origin image by using the mapping rule. After processing by the mapping rule, the CBLQ representation for a certain case of the rotated image can be obtained directly. Figure 7 shows the algorithm to obtain the CBLQ representation of the rotated image directly from the one of the original image by using the mapping rule. Table 1 shows the meaning of the parameters used in Figure 7. First, in the process of procedure *RM_CBLQ*, we call Procedure *CreateInternalID*(I_Codes, ID_Array) as shown in line 3. We scan I_Codes of the original image to compute the ID numbers of internal nodes. The ID number of the first internal node is 1. If $I\_Codes[i] = 2$ or 3 which is a label of an internal node, the ID number of this internal node is stored in $ID\_Array[i]$ by adding 1 to the

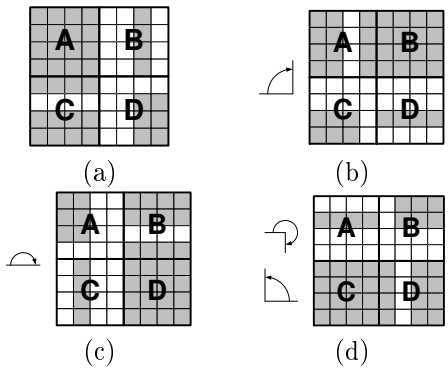

Figure 6: Rotating cases about the center in the clockwise direction: (a) the original image; (b) the Rotate_90 case; (c) the Rotate_180 case; (d) the Rotate_270 case.

```
1:procedure RM_CBLQ(I_Codes, I_Case)
2:begin
3:    Call CreateInternalID(I_Codes, ID_Array);
4:    repeat
5:        t := Dequeue(I_Queue);
6:        Call IQueueAdd(t, I_Case, ID_Array, I_Queue);
7:        Call OCodesOut(t, I_Case, I_Codes, O_Codes);
8:    until I_Queue is empty;
9:end
```

Figure 7: Procedure RM_CBLQ

previous one, where $0 \leqslant i \leqslant |I\_Codes|$. Otherwise the value -1 is stored in ID_Array which indicates $I\_Codes[i]$ is not an internal node.

Second, we process I_Queue by the loop of statements from line 5 to 7 in Figure 7 until I_Queue is empty. Initially, we add 0 to I_Queue, where 0 is the ID number of the root node. We will add the ID number of the internal node to I_Queue in the following statements. As shown in line 5, we delete an element t from I_Queue by calling function *Dequeue*. The variable t is an

Table 1: Parameters used for Procedure RM_CBLQ

parameter	content
I_Codes	an array with an index starting from 0 which stores the labels of the original image;
I_Case	the type of the rotating or mirroring case;
ID_Array	an array with an index starting from 0, which stores the ID numbers of internal nodes and $\|ID\_Array\| = \|I\_Codes\|$;
I_Queue	a queue which stores the ID numbers of L_Groups and will be read in order;
O_Codes	an array with an index starting from 0, which stores the labels after being rotated or mirrored and $\|O\_Codes\| = \|I\_Codes\|$;

ID number of L_Group which represents one parent node that has four child nodes in the quadtree. Since these four child nodes are represented as four labels based on CBLQ, one L_Group has four labels. The variable t helps compute the start and en position of the processing range in the ID_Array and I_Codes. Since each L_Group has four labels, the start position is obtained as $t \times 4$ and the end position is obtained as $(t+1) \times 4 - 1 = t \times 4 + 3$.

Procedure $IQueueAdd$ as shown in line 6 is used to find the ID number of the internal node in the ID_Array and put it into I_Queue. In this procedure, we regard $ID\_Array[t \times 4]$, $ID\_Array[t \times 4+1]$, $ID\_Array[t \times 4+2]$, and $ID\_Array[t \times 4+3]$ as positions A, B, C, and D in the quadtree, respectively. We also can regard them as blocks A, B, C, and D in the image, respectively. Then, we check the number in $ID\_Array[t \times 4 \,..\, t \times 4+3]$ to see whether it indicates the internal node or not. If the number is larger than 0, which indicates the internal node, we add it into I_Queue; otherwise, we ignore it. The ID number in I_Queue indicates the internal node that will be processed later, which implies their descendent nodes will be moved later at the same time.

In line 7, procedure $OCodesOut$ is used to map the labels in I_Codes to O_Codes according to the mapping rule for I_Case. In Procedure $IQueueAdd$, similarly, we regard $I\_Codes[t \times 4]$, $I\_Codes[t \times 4 + 1]$, $I\_Codes[t \times 4 + 2]$, and $I\_Codes[t \times 4 + 3]$ as positions A, B, C, and D in the quadtree, respectively. We store $I\_Codes[t \times 4 \,..\, t \times 4 + 3]$ in O_Codes according to the mapping rule for I_Case. The labels in $I\_Codes[t \times 4 \,..\, t \times 4 + 3]$ in their order will be affected by the mapping rule for I_Case. Therefore, according to the mapping rule for I_Case, we can obtain the new order of the labels and store them in O_Codes in the new order. Then, we will repeat the above three steps until I_Queue is empty, and we can obtain the rotated codes.

We take Figure 6-(a) as an example to illustrate the algorithm shown in Figure 7. There are two input variables for Procedure RM_CBLQ. One is I_Codes which is represented as 1222 0303 3311 0301 1010 1010 1100 1100 0011 based on the CBLQ representation, and the other one is I_Case which is the type of the Rotate_90 case. The initialization of Procedure RM_CBLQ is listed as follows.

• Initialization:
I_Codes:
1222 0303 3311 0301 1010 1010 1100 1100 0011
ID_Array:
I_Queue: 0
O_codes:

First, we store the ID numbers of L_Group in *ID_Array*:
-1 1 2 3 -1 4 -1 5 6 7 -1 -1 -1 8 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1.
Then, we repeat those steps from line 5 to 7 in Figure 7 until I_Queue is empty. We describe the process of the first loop as follows.

• The first loop:
Dequeue → 0 ; $0 \times 4 = 0$;
Read $I\_Codes[0..3]$ and $ID\_Array[0..3]$
I_Codes:
[1222] 0303 3311 0301 1010 1010 1100 1100 0011

ID_Array:
[-1 1 2 3] -1 4 -1 5 6 7 -1 -1 -1 8 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1

I_Queue: 2, 3, 1
O_Codes: **[2122]**

Note that the number with underline in I_Queue denotes the new-added ID number of the internal node in this loop.

Here we explain our strategy for the rotating case from view point of the image. In the first loop is shown in Figure 8, the blocks marked by the bold frame are visited in a *Morton order*, *i.e.*, blocks A, B, C, and D in order. The left part in each of Figures 8-(a), (b), (c), or (d) is the original image. The right part in each of Figures 8-(a), (b), (c), or (d) is the result after the block movement. Figure 8-(a) shows that the data item in block C is moved to block A. Figure 8-(b) shows that the data item in block A is moved to block B. Figure 8-(c) shows that the data item in block D is moved to block C. Figure 8-(d) shows that the data item in block B is moved to block D. Therefore, the four blocks are moved to the related positions according to the mapping rule $R1$ for the Rotate_90 case.

On the other hand, we explain our strategy from the view point of the code. In the process of the first loop, in the second step, we delete the ID number 0 from I_Queue. It means that the first loop is processing in L_Group with ID number 0. It starts at the number "0", which is obtained by the number "0" being multiplied by 4, and ends at the number "3", which is obtained by the starting number "0" being added by

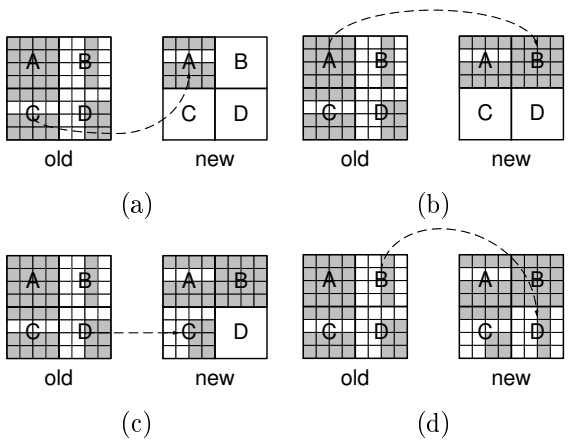

Figure 8: The movement of the data item in the first loop of the Rotate_90 case from the view point of the image: (a) from block C to block A; (b) from block A to block B; (c) from block D to block C; (d) from block B to block D.

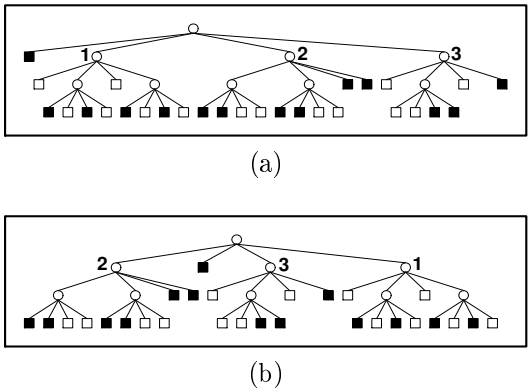

Figure 9: The movement of the ID numbers of each internal node: (a) the original position of the ID numbers; (b) the positions of the ID numbers after the first loop.

3. In other words, the region ranging from 0 to 3 in *I_Codes* and *ID_Array* will be processed to output the corresponding *O_Codes* and *I_Queue*, respectively. Therefore, we have *I_Codes*[0..3] = [1, 2, 2, 2], and *ID_Array*[0..3] = [-1, 1, 2, 3]. In the third step, since the second to fourth elements in *ID_Array*[0..3] indicate the ID numbers of the internal nodes, they are added in *I_Queue* that will be processed after. But the first element in *ID_Array* is not added in *I_Queue* because it indicates a leaf node. Therefore, we add 2, 3, 1 to *I_Queue* according to the mapping rule for the Rotate_90 case in order. Figure 9 shows the movement of the ID numbers of the internal node from the view point of the quadtree in the first loop.

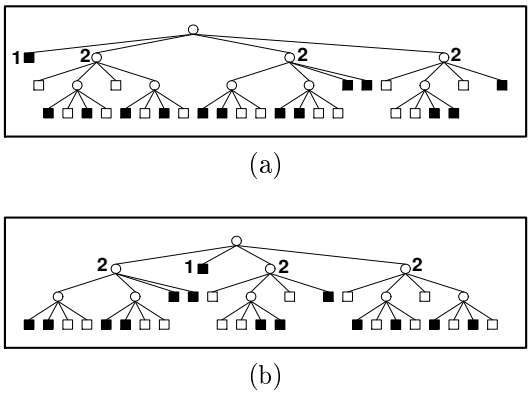

Figure 10: The movement of the CBLQ labels: (a) the original position of each CBLQ label; (b) the positions of the CBLQ labels after the first loop.

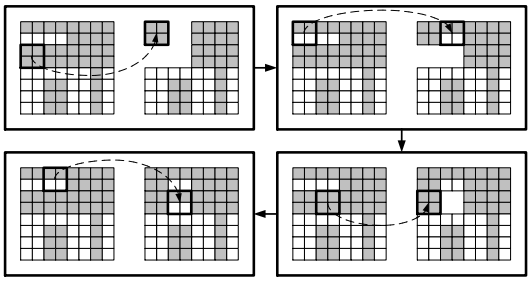

Figure 11: The second loop of the Rotate_90 case from the view point of the image

In the fourth step, according to the mapping rule for *I_Case*, we rearrange the order for the labels in *I_Codes*[0..3], *i.e.*, [1, 2, 2, 2] in order, to obtain the labels in *O_Codes*[0..3], *i.e.*, [2, 1, 2, 2] in order. Figure 10 shows the movement of the labels from the view point of the quadtree in the first loop. At this point, the first loop is finished.

The process of other loops are presented in Table 2 and Table 3. The output is *O_Codes* " 2122 1313 0013 0033 0101 0101 1010 1100 1100" in the Rotate_90 case. Note that from the view point of the image, the size of the blocks that are moved in the second loop shown in Figure 11 becomes smaller than that in the first loop shown in Figure 8. It indicates the movement and traverse of the nodes from the first level to the last level of the quadtree. Similarly, the size of the blocks that are moved in the 5th loop in Figure 12 becomes smaller than that in the second loop shown in Figure 11.

Table 2: The process of the $2nd$ loop to the $5th$ loop in the Rotate_90 case

Loop 2
Dequeue → 2 ; $2 \times 4 = 8$; Read *I_Codes*[8..11] and *ID_Array*[8..11] *I_Codes*: 1222 0303 **[3311]** 0301 1010 1010 1100 1100 0011 *ID_Array*: -1 1 2 3 -1 4 -1 5 **[6 7 -1 -1]** -1 8 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 *I_Queue*: 3, 1, 6, 7 *O_Codes*: 2122 **[1313]**
Loop 3
Dequeue → 3 ; $3 \times 4 = 12$; Read *I_Codes*[12..15] and *ID_Array*[12..15] *I_Codes*: 1222 0303 3311 **[0301]** 1010 1010 1100 1100 0011 *ID_Array*: -1 1 2 3 -1 4 -1 5 6 7 -1 -1 **[-1 8 -1 -1]** -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 *I_Queue*: 1, 6, 7, 8 *O_Codes*: 2122 1313 **[0013]**
Loop 4
Dequeue → 1 ; $1 \times 4 = 4$; Read *I_Codes*[4..7] and *ID_Array*[4..7] *I_Codes*: 1222 **[0303]** 3311 0301 1010 1010 1100 1100 0011 *ID_Array*: -1 1 2 3 **[-1 4 -1 5]** 6 7 -1 -1 -1 8 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 *I_Queue*: 6, 7, 8, 5, 4 *O_Codes*: 2122 1313 0013 **[0033]**
Loop 5
Dequeue → 6 ; $6 \times 4 = 24$; Read *I_Codes*[24..27] and *ID_Array*[24..27] *I_Codes*: 1222 0303 3311 0301 1010 1010 **[1100]** 1100 0011 *ID_Array*: -1 1 2 3 -1 4 -1 5 6 7 -1 -1 -1 8 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 **[-1 -1 -1 -1]** -1 -1 -1 -1 -1 -1 -1 -1 *I_Queue*: 7, 8, 5, 4 *O_Codes*: 2122 1313 0013 0033 **[0101]**

Table 3: The process of the $6th$ loop to the last loop in the Rotate_90 case

Loop 6
Dequeue → 7 ; $7 \times 4 = 28$; Read *I_Codes*[28..31] and *ID_Array*[28..31] *I_Codes*: 1222 0303 3311 0301 1010 1010 1100 **[1100]** 0011 *ID_Array*: -1 1 2 3 -1 4 -1 5 6 7 -1 -1 -1 8 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 **[-1 -1 -1 -1]** -1 -1 -1 -1 *I_Queue*: 8, 5, 4 *O_Codes*: 2122 1313 0013 0033 0101 **[0101]**
Loop 7
Dequeue → 8 ; $8 \times 4 = 32$; Read *I_Codes*[32..35] and *ID_Array*[32..35] *I_Codes*: 1222 0303 3311 0301 1010 1010 1100 1100 **(0011)** *ID_Array*: -1 1 2 3 -1 4 -1 5 6 7 -1 -1 -1 8 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 **[-1 -1 -1 -1]** *I_Queue*: 5, 4 *O_Codes*: 2122 1313 0013 0033 0101 0101 **[1010]**
Loop 8
Dequeue → 5 ; $5 \times 4 = 20$; Read *I_Cdoes*[20..23] and *ID_Array*[20..23] *I_Codes*: 1222 0303 3311 0301 1010 **[1010]** 1100 1100 0011 *ID_Array*: -1 1 2 3 -1 4 -1 5 6 7 -1 -1 -1 8 -1 -1 -1 -1 -1 -1 **[-1 -1 -1 -1]** -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 *I_Queue*: 4 *O_Codes*: 2122 1313 0013 0033 0101 0101 1010 **[1100]**
Loop 9
Dequeue → 4 ; $4 \times 4 = 16$; Read *I_Codes*[16..19] and *ID_Array*[16..19] *I_Codes*: 1222 0303 3311 0301 **[1010]** 1010 1100 1100 0011 *ID_Array*: -1 1 2 3 -1 4 -1 5 6 7 -1 -1 -1 8 -1 -1 **[-1 -1 -1 -1]** -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 *I_Queue*: *O_Codes*: 2122 1313 0013 0033 0101 0101 1010 1100 **[1100]**

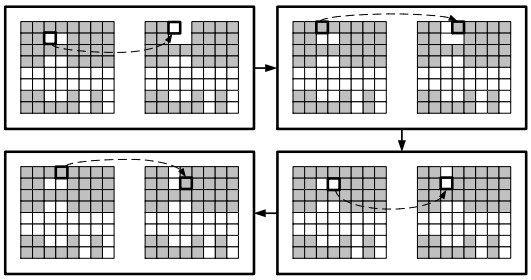

Figure 12: The 5th loop of the Rotate_90 case from the view point of the image

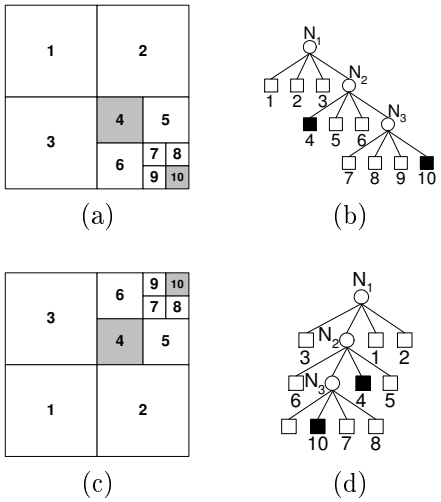

Figure 13: An Example for the mirroring case: (a) the original image; (b) the quadtree for the original image; (c) the image after being mirrored about X-axis; (d) the quadtree for the image after being mirrored.

4. The Strategy to Mirror Binary Images Represented by CBLQ

In this section, we present the strategy which maps the original image to the mirrored one directly from the codes based on the CBLQ representation [4]. The property for the rotating cases is the same for the mirroring case, *i.e.*, the image and the corresponding quadtree change at the same time during the mirroring process. Take Figure 13-(a) as an example. Figure 13-(c) shows the case of mirroring Figure 13-(a) about X-axis to obtain the result. In other words, after mirroring of Figure 13-(a) about X-axis, the four children at the first level in Figure 13-(b) have been changed their positions from A, B, C, and D to C, D, A, and B, respectively, as shown in Figure 13-(d). The four children at the first level in Figure 13-(d) are labeled with labels 3, N_2, 1, and 2 on positions A, B, C, and D, respectively. The process of changing the positions of four children is also executed at each level until the end of the quadtree. It is similar to the other mirroring cases.

Our strategy uses the same process as the rotating strategy as shown in Figure 7 to obtain the code for the mirrored image. However, different from the rotating strategy, the mirroring strategy uses different rules for the mirroring cases as shown In the process of the mirroring strategy as shown in Figure 7, the mapping rules for the mirroring cases are used in *I_Case* to changed the positions of labels in the corresponding quadtree. Then, we can obtain the mirrored code of the image directly from the code of the original image. The mapping rules used for four mirroring cases are listed as follows.

- Rule $M1$. (**Mirror_X**):
  $C \rightarrow A$; $D \rightarrow B$; $A \rightarrow C$; $B \rightarrow D$.
- Rule $M2$. (**Mirror_Y**):
  $B \rightarrow A$; $A \rightarrow B$; $D \rightarrow C$; $C \rightarrow D$.
- Rule $M3$. (**Mirror_MD**):
  $D \rightarrow A$; $B \rightarrow B$; $C \rightarrow C$; $A \rightarrow D$.
- Rule $M4$. (**Mirror_CD**):
  $A \rightarrow A$; $C \rightarrow B$; $B \rightarrow C$; $D \rightarrow D$.

We take Figures 6-(a) as examples to illustrate the image of Mirror_X case which uses the mapping rule $M1$. According to rule $M1$, The sixteen data blocks in block A in Figure 14-(a) are moved, as the result of the sixteen data blocks shown in block C in Figure 14-(b). It is similar to the sixteen data blocks in each block B, C, and D. They are moved to blocks D, A, and B, respectively. The other mapping rules which are used for the other cases can be easily derived in the similar way. The images of other rotating cases which use different mapping rules are shown in Figures 14-(c), (d), and (e).

5. Performance

In this section, we show the results of the performance study of our strategies to rotate and mirror images based on CBLQ. We use the CPU-time as the performance measure for our strategy based on CBLQ. In our simulation, we randomly generate pixel arrays of size 32×32 and 64×64 pixels with controlled amount of black density. We generate images with densities of 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, and 90% black pixels. That is, we generate 18 combinations (from

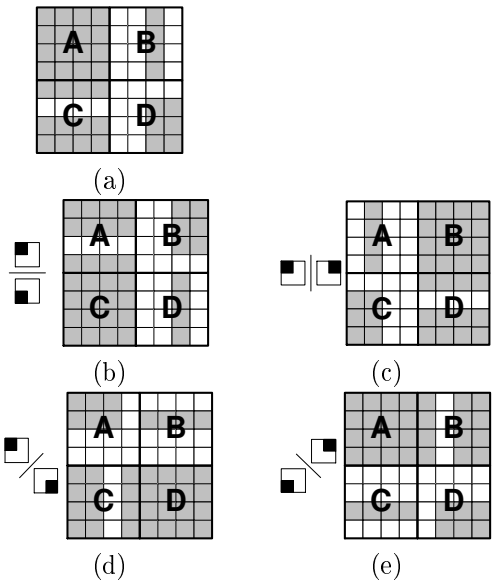

Figure 14: Mirroring cases: (a) the original image; (b) the Mirror_X case; (c) the Mirror_Y case; (d) the Mirror _MD case; (e) the Mirror _CD case.

Table 4: Comparison of the CPU-time (ms) on different rotating cases based on the CBLQ with size 32×32

Case \ Black denstiy	10%	20%	30%	40%	50%	60%	70%	80%	90%
Rotate_90	8	14	17	19	20	19	17	13	8
Rotate_180	8	14	17	19	20	19	17	13	8
Rotate_270	8	14	17	19	20	19	17	13	8

32×32 with 10% black pixels to 64×64 with 90% black pixels). 500 binary images which are randomly placed black pixels with the uniform distribution are generated for each combination. For each combination, we compare the average CPU-time of the seven cases (the Rotate_90 case, the Rotate_180 case, the Rotate_270 case, the Mirror_X case, the Mirror_Y case, the Mirror_MD case, and the Mirror_CD case) based on the CBLQ.

The results of the CPU-time for those cases in rotating and mirroring strategies based on the CBLQ are shown in Tables 4, 5, 6, and 7. From these tables, they both show that the CPU-time for those cases is the same, whether the rotating cases or mirroring cases, for each of the 18 combinations of size and black density. Since those seven cases use different mapping rules to help the code conversion, they use the same process to obtain the result.

Moreover, we can see that the larger the size of

Table 5: Comparison of the CPU-time (ms) on different mirroring cases based on the CBLQ with size 32×32

Case \ Black denstiy	10%	20%	30%	40%	50%	60%	70%	80%	90%
Mirror_X	8	14	17	19	20	19	17	13	8
Mirror_Y	8	14	17	19	20	19	17	13	8
Mirror_MD	8	14	17	19	20	19	17	13	8
Mirror_CD	8	14	17	19	20	19	17	13	8

Table 6: Comparison of the CPU-time (ms) on different rotating cases based on the CBLQ with size 64×64

Case \ Black denstiy	10%	20%	30%	40%	50%	60%	70%	80%	90%
Rotate_90	63	121	157	177	185	178	157	121	63
Rotate_180	63	121	157	177	185	178	157	121	63
Rotate_270	63	121	157	177	185	178	157	121	63

Table 7: Comparison of the CPU-time (ms) on different mirroring cases based on the CBLQ with size 64×64

Case \ Black denstiy	10%	20%	30%	40%	50%	60%	70%	80%	90%
Mirror_X	63	121	157	177	185	178	157	121	63
Mirror_Y	63	121	157	177	185	178	157	121	63
Mirror_MD	63	121	157	177	185	178	157	121	63
Mirror_CD	63	121	157	177	185	178	157	121	63

the image is, the longer the needed CPU-time is. This is because the length of the code representing the image becomes long, as the size of the image becomes large. We also can see that the CPU-time is increased when the black density is no larger than 50%, and the CPU-time is decreased when the black density is larger than 50%. It means that when the black density of combination A is equal to the white density of combination B, *i.e.*, the black density is 20% and 80% (the white density is 20%), the CPU-time of combination A and that of combination B are similar. This is because the CBLQ representation is used to code all nodes in the quadtree, and when the black density of combination A is equal to the white density of combination B, the number of nodes of combination A in the quadtree and that of combination B in the quadtree are similar.

6. Conclusion

In this paper, we have proposed the strategies to obtain the code of the rotated or mirrored image based on CBLQ representation. Our strategy can obtain the code of the rotated or mirrored image directly from the code of the original image, instead of from the rotated/ mirrored image. From our simulation, we have shown that the CPU-time for all cases , whether the rotating cases or the mirroring cases, are the same by using our strategy on the images which are represented by CBLQ. When the black density is equal to the white density (50%) base on CBLQ, it costs the most CPU time among all combination of the black or white densities.

Acknowledgement

The authors also like to thank "Aim for Top University Plan" project of NSYSU and Ministry of Education, Taiwan, for partially supporting the research.

References

[1] P. M. Chen, "Variant Code Transformations for Linear Quadtrees," *Pattern Recognition Letters*, Vol. 23, No. 11, pp. 1253–1262, Sept. 2002.

[2] K. L. Chung and J. G. Wu, "Fast Implementations for Mirroring and Rotating Bincode-Based Images," *Pattern Recognition*, Vol. 31, No. 12, pp. 1961–1967, Dec. 1998.

[3] M. Hasegawa, N. Suzuki, T. Takahashi and S. Kato, "A Lossless Coding Method Using Shape Information and Rotation Compensation for Digital Museum Images," *Proc. of Int. Conf. on Image Processing*, pp. 193–196, 2003.

[4] T. W. Lin, "Set Operations on Constant Bit-Length Linear Quadtrees," *Pattern Recognition*, Vol. 30, No. 7, pp. 1239–1249, July 1997.

[5] S. K. Lodha, N. M. Faaland, J. C. Renteria, "Topology Preserving Top-Down Compression of 2D Vector Fields Using Bintree and Triangular Quadtrees," *IEEE Trans. on Visualization and Computer Graphics*, Vol. 9, No. 4, pp. 433–442, Oct. 2003.

[6] A. Makadia and K. Daniilidis, "Rotation Recovery from Spherical Images without Correspondences ," *IEEE Trans. On Pattern Analysis and Machine Intelligence*, Vol. 28, No. 7, pp. 1170–1175, July 2006.

[7] B. Obara, "An image processing algorithm for the reversed transformation of rotated microscope images," *Computers and Geosciences,* Vol. 33, No. 7, pp. 853–859, July 2007.

[8] Frank. Y. Shih and Wai-Tak Wong, "An Adaptive Algorithm for Conversion from Quadtree to Chain Codes," *Pattern Recognition*, Vol. 34, No. 3, pp. 631–639, 2001.

[9] H. H. Wang and N. Kulathuramaiyer, "Rotation Invariance in Color-spatial Based Image Retrieval," *Proc. of Int. Conf. on Information Technology in Asia*, pp. 156–161, 2003.

[10] Y. H. Yang, K. L. Chung and Y. H. Tsai, "A Compact Improved Quadtree Representation with Image Manipulations," *Image and Vision Computing,* Vol. 18, No. 3, pp. 223–231, Feb. 2000.

[11] X. Yang, J. Pei and W. Xie, "Rotation Registration of Medical Images Based on Image Symmetry," *Proc. of Int. Conf. on Intelligent Computing*, pp. 68–76, 2005.

[12] H. Zhong, W. F. Sze, Y. S. Hung, "Reconstruction from Plane Mirror Reflection," *Proc. of the 18th Int. Conf. on Pattern Recognition,* pp. 715–718, 2006.