Variant code transformations for linear quadtrees

Pei-Min Chen

Department of Computer and Information Science, Soochow University, 56, Kuer-Yang St., Section 1, Taipei, Taiwan, ROC

Received 25 April 2001; received in revised form 17 July 2001

Abstract

In this paper, general guidelines and specific algorithms for code transformations between breadth-first (BF) and depth-first (DF) linear quadtrees are proposed. Each algorithm has time and space complexities in $O(l)$, where $l$ is the length of the code of a BF or DF linear quadtree. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Linear quadtree; Code transformation; Depth-first traversal; Breath-first traversal

1. Introduction

A quadtree is a well-known hierarchical data structure used to represent an image or a picture in various applications (Samet, 1990a,b). In general, for an image, its linear quadtree representation requires less storage space than its bit pattern representation does. Thus, variant linear quadtree representations were proposed. A linear quadtree can be regarded as a list of location and/or color codes such that the hierarchical structure of a quadtree is preserved by arranging the sequence of these codes according to depth-first (DF) or breadth-first (BF) traversing of the quadtree (Samet, 1990a,b). In general, for the same picture, the amount of storage used to represent a BF linear quadtree is less than that used to represent a DF one. However, since a BF linear quadtree is a non-block oriented structure and is more position-sensitive than a DF one, it seems that image operations (Samet, 1990a; Hunter and Steiglitz, 1979) performed on the former are more complex and more time-consuming than those performed on the latter (Gargantini, 1982; Chang and Chang, 1994; Lin, 1997; Wang, 1991). Hence, we can manipulate an image more effectively and efficiently if this image is stored or transmitted by its BF linear quadtree and operated on by its DF one.

In this paper, general guidelines for code transformations between BF and DF linear quadtrees are proposed. The algorithms of code transformations for specific BF and DF linear quadtrees are also derived. Here, one typical DF linear quadtree scheme, Gargantini (1982), and two BF linear quadtree schemes with different styles, FBLQ (Chang and Chang, 1994) with variant length and CBLQ (Lin, 1997) with constant length, are selected to depict the guidelines and algorithms. Each algorithm has time and space complexities in $O(l)$, where $l$ is the length of the code of a BF or DF linear quadtree. These complexities are generally much reduced compared to...
those of the BF or DF encoding processes for a corresponding image, which are at least of the same order as the number of pixels in this image.

The remainder of this paper is organized as follows. In Section 2, three BF and DF linear quadtree representations are described. In Section 3, the general guidelines and formal algorithms for code transformations are first proposed, and then two examples are given to demonstrate the concept behind the code transformations. Section 4 gives analysis of these transformation algorithms. Concluding remarks are finally made in Section 5.

2. Linear quadtrees

A quadtree is generally used to describe a square binary image. The corresponding quadtree of an image is constructed by successively subdividing this image into four equal-size subimages: NW, NE, SW, and SE quadrants. A homogeneously colored quadrant of the image is represented by a leaf node in the tree. Otherwise, the quadrant is represented by an internal node and further divided into subquadrants until each subquadrant has the same color pixels. Thus, the quadtree provides a compact structure for an image. A sample image and its corresponding quadtree are shown in Fig. 1. In this section, three different linear quadtree representations are introduced. Each linear quadtree is represented by a list of numbers, of which the sequence and value-setting conserve the structure of the quadtree. Basically, the sequence and value-setting of these numbers are mutual relative. In general, the sequence is generated according to the manner of tree traversal, the DF or BF approach, and the value-setting of these numbers is dependent on the approach of tree traversal. If DF traversal is used to represent a linear quadtree, the values, called location codes, of each number often represent the four directions, NW, NE, SW, and SE, of the quadrants. However, during scanning of a quadtree in the BF style, the values of the numbers often depict the types of nodes (internal or leaf) and the colors of the leaf nodes (black or white). The former values are called position codes, and the latter are called color codes.

One of the typical DF linear quadtree representations was proposed by Gargantini (1982). In an image, only black quadrants are converted into Gargantini codes, and each quadrant is represented by a sequence of location codes (0, 1, 2, and 3 for NW, NE, SW, and SE, respectively) and don’t-care symbols, ‘X’. This sequence of a quadrant is constructed by listing the location codes of the nodes in the path from the root to this quadrant. For example, if a black quadrant is coded as ‘00X’, this quadrant is the leftmost leaf node at the third level of the quadtree with depth 4. The purpose of appended symbol ‘X’ is to make Gargantini code with a fixed length. The Gargantini codes for the image in Fig. 1 are: ‘00X, 01X, 112, 113, 12X, 201, 230, 3XX’, which are shown in Fig. 2.

Fig. 1. A sample image and its corresponding quadtree.
Usually there are two kinds of codes for BF linear quadtree representations: position codes for internal/leaf nodes and color codes for the colors of nodes. Level-wise and left-to-right orderings are used to construct a list of BF linear quadtree codes. The position codes of nodes are listed level by level from the second to the bottom level and the codes within each level is left-to-right order. However, the color codes of leaf nodes follow after their corresponding position codes. Due to the different ways used to represent position codes and color codes, there are two kinds of BF linear quadtree representations: variant length and constant length. FBLQ is one of the BF representations, in which the position codes are ‘1’ for an internal node and ‘0’ for a leaf node, respectively, and the color codes are ‘1’ for black and ‘0’ for white. In the FBLQ representation, each internal or leaf node has a position code. However, only a leaf node is associated with a color code. As a result, the code length of the FBLQ representation is variant. The FBLQ linear quadtree for the image in Fig. 1 is shown in Fig. 3. These codes are ‘1110 1 0000 1100 0100 010 1001 00 0011 0100 1000’, where color codes are underlined. CBLQ is another BF representation, in which the code length is constant. CBLQ uses four different values to describe position codes and color codes together: 0 for white, 1 for black, 2 for an internal node, and 3 for an internal node with all its child nodes being external. If we use two bits to represent a CBLQ code, then the byte length of CBLQ codes is equal to the number of internal nodes in the quadtree. The CBLQ codes for the image in Fig. 1 are also depicted in Fig. 3. They are ‘3221 1100 0310 3003 0011 0100 1000’.

Next, the size of the storage requirement and the performance in terms of image operations on different linear quadtrees will be briefly discussed. For convenience, let \( I \), \( B \), and \( W \) stand for the total number of internal, black, and white nodes in a quadtree, respectively. Also, let \( S_G \), \( S_F \), and \( S_C \) stand for the size of the storage requirement (in bit length) for Gargantini, FBLQ, and CBLQ codes,
respectively. Then, for a \(2^n \times 2^n\) image, the size of the storage requirement of different linear quad-trees, respectively, is

\[
S_G = B \times 3n, \quad (1)
\]

where \(n\) is the largest level in the quadtree,

\[
S_F = I + W + B + W_I B_I, \quad (2)
\]

where \(W_I\) and \(B_I\) are the numbers of white and black nodes not in the bottom level, the \(n\)th level, of the quadtree, respectively,

\[
S_C = 8 \times I, \quad (3)
\]

where the number 8 is the bit length of a byte.

As described above, Gargantini codes represent black nodes only, and each black node requires \(n\) location codes or don’t-care symbols. Each black node also needs three bits to represent four location codes and one don’t-care symbol. Thus, the total number of bits required by a quadtree represented by Gargantini codes is \(B \times 3n\). Considering the FBLQ representation, it is obvious that each internal node is described by a position code, that each leaf node is described by a color code, and, in addition, that each leaf node not in the bottom level of the quadtree is also described by a position code. Therefore, the total bit length for a FBLQ linear quadtree is \(8 \times I\), where \(8\) is the bit length of a byte. To compare these storage requirements, the following equation (Dyer, 1982) is applied:

\[
I = \frac{1}{3} (B + W - 1).
\]

On average, \(B\) is likely to equal to \(W\), i.e.,

\[
I \approx \frac{2}{3} B. \quad (4)
\]

Thus,

\[
\frac{S_F}{S_G} = \frac{I + B + W + B_I + W_I}{B \times 3n} \approx \frac{(2/3)B + 3B}{B \times 3n} = \frac{11}{9n}, \text{ where we let } \frac{1}{2}B = \frac{1}{2}W = B_I = W_I \quad (5)
\]

and

\[
\frac{S_C}{S_G} \approx \frac{8 \times (2/3)B}{B \times 3n} = \frac{16}{9n}. \quad (6)
\]

From Eqs. (5) and (6), we know that, in general, the size of the storage requirement for Gargantini codes is greater than that for FBLQ or CBLQ codes, respectively. In addition, these ratios will increase if the size of the transformed image becomes larger. If the size of an image is \(256 \times 256\), the ratios for Eqs. (5) and (6) will reach 15.3% and 22.2%.

Considering the performance of image operations on these linear quadtrees, additional data structures, e.g., arrays or tables, and more complex algorithms will be needed to perform set operations on FBLQ and CBLQ linear quadtrees. However, these set operations can be directly manipulated using Gargantini codes. It also seems that Gargantini representation is superior to the other two linear quadtree representations in terms of the performance of other kinds of operations, e.g., neighbor finding, windowing, region expansion, etc.

3. Code transformations

In this section, guidelines for transforming a DF linear quadtree into its corresponding BF quadtree and vice versa are first introduced. Then, related algorithms and examples are given. The guidelines for transforming a DF linear quadtree into its corresponding BF quadtree are as follows:

1. According to the hierarchy of the quadtree, scan the DF codes level by level and record the number of occurrences of each quaternary code in every level.
2. Produce a color code if the bottom-of-the-sub-quadtree is reached. Otherwise, a position code for an internal node is generated.

The ways to transform a Gargantini linear quadtree into its corresponding FBLQ and CBLQ quadtrees are shown in Algorithms 1 and 2, respectively, and these two transformations are demonstrated by Fig. 4. Algorithm 1 and 2 first scan Gargantini codes level by level and produce...
the intermediate data by counting the number of occurrences of each quaternary code in every level. Since Gargantini codes start from the second level, the above scanning begin from the same level. Then use the values in the intermediate data and Gargantini code at the next level to generate FBLQ/CBLQ position and color codes. Repeat the above processes until the bottom level is reached. Finally, utilize the Gargantini codes at the bottom level and the intermediate data of the previous level to produce the final color codes. ‘NOP’ in these two algorithms denotes no operation.

**Algorithm 1.** Gargantini to FBLQ code transformation.

**Input.** A sorted Gargantini linear quadtree and the number, \( n \), of levels of this quadtree.

**Output.** The corresponding FBLQ linear quadtree.

**Process.**

**Step 1.** Scan all the Gargantini codes corresponding to the quadrants at the current level, \( v \), (starting from the second level). If the bottom level of this quadtree is reached, then jump to Step 6.

Otherwise, record the number of occurrences \( t_{vw} \) for the \( w \)th subquadtree in the \( v \)th level as intermediate data, where \( 1 \leq w \leq 4^{v-1} \). The value of \( t_{vw} \) is

- 0 if there is no Gargantini code for the \( w \)th subquadtree,
- \( i \) if the number of Gargantini codes for the \( w \)th subquadtree is \( i \), or
- NOP if the Gargantini code is ‘X’.

**Step 2.** Produce FBLQ position codes for the \( w \)th subquadtree in the \( v \)th level based on the value of the intermediate data \( t_{vw} \):

1. If \( t_{vw} > 1 \), or \( t_{vw} = 1 \) and the Gargantini code of the \( w \)th subquadtree in the \( v + 1 \)st level is not ‘X’, or
2. 0 if \( t_{vw} = 0 \), or \( t_{vw} = 1 \) and the Gargantini code of the \( w \)th subquadtree in the \( v + 1 \)st level is ‘X’.

**Step 3.** Produce FBLQ color codes sequentially for the \( w \)th subquadtree in the \( v \)th level based on the value of the intermediate data \( t_{vw} \):

1. If \( t_{vw} = 1 \) and the Gargantini code of the \( w \)th subquadtree in the \( v + 1 \)st level is ‘X’, or
2. 0 if \( t_{vw} = 0 \).

**Step 4.** Eliminate those \( t_{vw} \) values which are already used to produce color codes.
Step 5. Repeat Steps 1–4 for the next level until the bottom level of this quadtree is reached.

Step 6. Produce FBLQ color codes based on the remained \( t_{(v-1)w} \) values and the corresponding scanned Gargantini codes at the bottom level:

- 1 if \( t_{(v-1)w} \geq 1 \) and the value of the scanned Gargantini code in the corresponding \( w \)th subquadtree is the same as its quaternary position,
- 0 if \( t_{(v-1)w} \geq 1 \) and the value of the scanned Gargantini code in the corresponding \( w \)th subquadtree is not equal to its quaternary position, or
- NOP if the scanned Gargantini code is ‘X’.

Algorithm 2. Gargantini to CBLQ transformation.

Input. A sorted Gargantini linear quadtree and the number, \( n \), of levels of this quadtree.

Output. The corresponding CBLQ linear quadtree.

Process.

Step 1. Scan all the Gargantini codes corresponding to the quadrants at the current level, \( v \), (starting from the second level). If the bottom level of this quadtree is reached, then jump to Step 6. Otherwise, record the number of occurrences \( t_{vw} \) for the \( w \)th subquadtree in the \( v \)th level as intermediate data, where \( 1 \leq w \leq 4^{v-1} \). The value of \( t_{vw} \) is

- 0 if there is no Gargantini code for the \( w \)th subquadtree,
- \( i \) if the number of Gargantini codes for the \( w \)th subquadtree is \( i \), (this process is called descendant locating) or
- NOP if the Gargantini code is ‘X’.

Step 2. For the \( w \)th subquadtree in the \( v \)th level, scan its Gargantini codes in the \( v + 2 \)nd level and calculate the total number \( d \) of don’t-care symbols, ‘X’.

Step 3. Produce CBLQ codes for the \( w \)th subquadtree in the \( v \)th level based on the value of the intermediate data \( t_{vw} \):

- 3 if \( t_{vw} \neq 0 \) and one of the following conditions holds: \( d = t_{vw} \) or \( v + 1 \)st level is the bottom level,
- 2 if \( t_{vw} \neq 0 \) and \( d \neq t_{vw} \),
- 1 if \( t_{vw} = 1 \) and the Gargantini code of the \( w \)th subquadtree in the \( v + 1 \)st level is ‘X’, or
- 0 if \( t_{vw} = 0 \).

Step 4. Eliminate those \( t_{vw} \) values which are already used to produce color codes.

Step 5. Repeat Steps 1–4 for the next level until the bottom level of this quadtree is reached.

Step 6. Produce CBLQ codes based on the the remained \( t_{(v-1)w} \) values and the corresponding scanned Gargantini codes at the bottom level:

- 1 if \( t_{(v-1)w} \geq 1 \) and the value of the scanned Gargantini code in the corresponding \( w \)th subquadtree is the same as its quaternary position,
- 0 if \( t_{(v-1)w} \geq 1 \) and the value of the scanned Gargantini code in the corresponding \( w \)th subquadtree is not equal to its quaternary position, or
- NOP if the scanned Gargantini code is ‘X’.

The guidelines for transforming a BF linear quadtree into its corresponding DF quadtree are as follows:

1. Scan the BF codes sequentially, and store the position of a BF code if the corresponding node of this BF code is an external node.
2. Produce a complete DF code if a color code in the BF linear quadtree is reached.

In generating Gargantini codes from other BF codes, only the black leaf nodes are transformed. Thus, a complete Gargantini code is generated only for a black leaf node in the quadtree. Algorithms 3 and 4 transform FBLQ and CBLQ linear quadtrees into the corresponding Gargantini codes, respectively, and these transformations are demonstrated by Fig. 5. These two algorithms first scan position codes sequentially and transform the position codes into the corresponding location codes. If the scanned position code stands for a leaf node and its color is black, then the corresponding Gargantini codes are produced. Otherwise, if the color is white, then no Gargantini code is produced. However, if the scanned position code stands for an internal node, then the transformed location code is appended to the location code list which represents the ancestor of this internal node. This list is used to generate Gargantini codes if the descendants of the internal node are black leaf nodes. At the end of these two algorithms, the color codes at the bottom level are scanned.
black color (node) is reached, the corresponding Gargantini codes are generated by appending the location code of this node to its ancestor’s location code list.

**Algorithm 3.** FBLQ to Gargantini transformation.

*Input.* An FBLQ linear quadtree and the number, \( n \) of levels of this quadtree.

*Output.* The corresponding Gargantini codes.

*Process.*

1. **Step 1.** Scan the position codes, \( p \), at the current level, \( v \), of an FBLQ linear quadtree sequentially.

2. **Step 2.** Generate the location code, \( j \), for each position code, \( p \), as intermediate data if \( p = 1 \) for the corresponding \( j \)th quadrant, and append this new location code to the previously generated location code list, which specifies the ancestor of this current quadrant, such that this new list of appended location codes represents a specific subquadrant at the \( v \)th level.

Then, store this appended code list as a new intermediate data.

3. **Step 3.** Append the \( n - v \) number of don’t-care symbols ‘X’ to the previously stored location code list in the intermediate data as the corresponding Gargantini codes if \( p = 0 \) and its associated color code is 1.

4. **Step 4.** Repeat Steps 1, 2, and 3 until all FBLQ position codes are processed.

5. **Step 5.** Scan each color code, \( c \), of the bottom level and generate the location code, \( j \), if \( c = 1 \) for the corresponding \( j \)th quadrant. Then, append this new location code to the previously generated location code list, which specifies the ancestor of the current quadrant, as the corresponding Gargantini codes.

**Algorithm 4.** CBLQ to Gargantini transformation.

*Input.* A CBLQ linear quadtree and the number of levels of this quadtree.

*Output.* The corresponding Gargantini codes.

*Process.*

1. **Step 1.** Scan CBLQ codes, \( c \), at the current level, \( v \), sequentially.

2. **Step 2.** Generate the location code, \( j \), for each scanned code, \( c \), if \( c = 2 \) or \( c = 3 \), for the corresponding \( j \)th quadrant, and append this new location code to the previously generated location code list, which specifies the ancestor of the current quadrant, such that this new list of appended location codes represents a specific subquadrant at the \( v \)th level.

Then, store this appended code list as a new intermediate data.

Append a specific number of don’t-care symbols ‘X’ to the previously stored location codes in
the intermediate data as the corresponding Gargantini code if \( c_i = 1 \).

Step 3. Append the \( n - v \) number of don’t-care symbols ‘X’ to the previously stored location codes in the intermediate data as the corresponding Gargantini codes if \( c = 1 \).

Step 4. Repeat Steps 1, 2, and 3 until the bottom level of this quadtree is reached.

Step 5. Scan the rest of the codes, \( c \), in the bottom level and generate the location code, \( j \), if \( c = 1 \) for the corresponding \( j \)th quadrant. Then, append this new location code to the previously generated location code list, which specifies the ancestor of the current quadrant, as the corresponding Gargantini codes.

For Step 3 in both Algorithms 3 and 4, the number of don’t-care symbols ‘X’ to be appended is determined by the difference between the level number of the specified black leaf node and that of the nodes in the bottom level. The analyses of these algorithms will be presented in Section 4. The complexity of each algorithm in time and space is probably much less than that of the BF or DF encoding processes, which are at least of the same order as the number of pixels in the image.

4. Analyses

Since all four algorithms use intermediate data to record the occurrences or the positions of subquadtrees in order to produce target codes, the intermediate data play an important role in the design and analysis of these algorithms. Without loss of generality, analyzing the complexity of these four algorithms requires that we focus on the number of comparisons as well as the size of space required for both the transformed codes and the intermediate data. We will first scrutinize the space requirement for the intermediate data. Then, we will use this result to analyze these algorithms.

Lemma 1. The space complexity for the intermediate data in both Algorithms 1 and 2 is \( O(I + W_t + B_t) \), where \( I \) is the number of internal nodes and \( W_t \) and \( B_t \) are the number of white leaf nodes and black leaf nodes above the bottom level, respectively.

Proof. Algorithms 1 and 2 use intermediate data to record the number of occurrences of each subquadtree whenever a non-‘X’ symbol is scanned. Thus, each piece of intermediate data corresponds to a subquadtree, and this data will expand by another four intermediate data if there are descendants of this subquadtree except for the subquadtrees containing only leaf nodes in the bottom level. Hence, the order of space required for intermediate data in Algorithm 1 or 2 is the number of nodes not in the bottom level, i.e., \( O(I + W_t + B_t) \).

Lemma 2. The space complexity for the intermediate data in both Algorithms 3 and 4 is \( O((n - 1)I) \), where \( I \) is the number of internal nodes and \( n \) is the number of levels in the quadtree.

Proof. Whenever an FBLQ or CBLQ code is scanned, if its corresponding quadtree node is an internal node, then its location value for the subquadtree in the current level is appended to its ancestor’s location code list to generate a new intermediate data. Thus, the total space required for intermediate data in Algorithm 3 or 4 can be calculated as \( I_I + 2 \times I_{I_2} + 3 \times I_{I_3} + \cdots + (n - 1) \times I_{I_{n-1}} \) where \( I_I \) is the number of internal nodes in the \( I_{I_1} \)th level and \( I_I + I_{I_2} + I_{I_3} + \cdots + I_{I_{n-1}} = I \), where \( I \) is the total number of internal nodes in this quadtree.

However, \( I_I + 2 \times I_{I_2} + 3 \times I_{I_3} + \cdots + (n - 1) \times I_{I_{n-1}} \) \( < (n - 1)I \).

Thus, the space complexity is \( O((n - 1)I) \).

Theorem 1. The space complexity of Algorithm 1 or 2 is \( O(l) \), where \( l \) is the length of the Gargantini codes.

Proof. The space in both Algorithms 1 and 2 is mainly used to store Gargantini codes and intermediate data. From (1), (4), (5), and Lemma 1, the storage requirements (in bytes) for Gargantini codes and the intermediate data are \( B \times n \) and
The space complexity of each algorithm is $O(n \times B + (3/5)B)$, which can be simplified to $O(nB) = O(l)$, where $l$ is the length of the Gargantini codes.

Theorem 2. The space complexity of Algorithm 3 is $O(l)$, where $l$ is the length of the generated Gargantini codes.

Proof. The space in Algorithm 3 is mainly used to store FBLQ codes and intermediate data. Since one quaternary digit is enough to represent a location code in the intermediate data, from (2) and Lemma 2, the storage requirements (in bits) for FBLQ codes and the intermediate data are $l + W + B + W_i + B_i$ and $2(n - 1) \times I$, respectively, where $n$ is the number of levels in the quadtree. Also, from (4) and (5), these storage requirements are approximately $(11/3)B$ and $(4(n - 1)/3)B$, respectively. Thus, the total space complexity of this algorithm is $O((4n + 7/3)B)$, which can be simplified to $O((4n/3)B) = O((4/9)B \times 3n) = O((4/9)l) = O(l)$, where $l$ is the length of the generated Gargantini codes.

Theorem 3. The space complexity of Algorithm 4 is $O(l)$, where $l$ is the length of the generated Gargantini codes.

Proof. The space in Algorithm 4 is mainly used to store CBLQ codes and intermediate data. Since one quaternary digit is enough to represent a location code in the intermediate data, from (3) and Lemma 2, the storage requirements (in bits) for CBLQ codes and the intermediate data are $8l$ and $2(n - 1) \times I$, respectively, where $n$ is the number of levels in the quadtree. Also, from (4), these storage requirements are approximately $(16/3)B$ and $(4(n - 1)/3)B$, respectively. Thus, the total space complexity of this algorithm is $O((4n + 12/3)B)$, which can be simplified to $O((4n/3)B) = O((4/9)B \times 3n) = O((4/9)l) = O(l)$, where $l$ is the length of the generated Gargantini codes.

Theorem 4. The time complexity of Algorithm 1 or 2 is $O(l)$, where $l$ is the code length of a Gargantini linear quadtree.

Proof. Basically, in Algorithms 1 and 2, the Gargantini codes are first transformed into intermediate data, which are then used to produce FBLQ or CBLQ codes. Hence, the time complexity of Algorithm 1 or 2 consists of the time used to perform the above two operations. That is, the time complexity is determined by the length of the Gargantini codes, $l$, and the length of the intermediate data, $m$. Thus, the time complexity can be calculated as $O(l + m)$. However, from Theorem 1, $O(l + m) = O(l)$.

Theorem 5. The time complexity of Algorithm 3 or 4 is $O(l)$, where $l$ is the code length of an FBLQ or CBLQ linear quadtree.

Proof. In Algorithms 3 and 4, the intermediate data are used to store the list of location codes (from the root node to the node being processed) for the internal nodes. They are referred to whenever FBLQ or CBLQ black nodes are reached. Hence, the time complexity is mainly determined by the input data, and the FBLQ or CBLQ codes. That is, the time complexity is on the order of the length of the FBLQ or CBLQ codes, $O(l)$.

5. Conclusion

In general, BF linear quadtree representation requires less storage space than DF representation does. For a 256x256 image, the ratio of the storage space for both kinds of representations may reach 15.3%. However, in general, DF linear quadtree representation can make performance of image operations easier and faster compared to BF representation. The related code transformation guidelines and algorithms have been proposed. In this case, we can use BF and DF linear quadtree representation interchangeably to exploit their respective advantages. The complexity of each proposed code transformation algorithm in time and space measurement is in $O(l)$, where $l$ is the length of the codes of a BF or DF linear quadtree. This complexity is much less than that of either the BF or DF encoding processes, which are at least on the same order as the number of pixels in the
image. Thus, an image can be stored or transmitted in its corresponding BF linear quadtree representation and can be manipulated in its corresponding DF linear quadtree representation so that the advantages of both BF and DF representation can be fully exploited with negligible additional cost in terms of time and space.

References


