Image and Vision Computing 19 (2001) 413–426

www.elsevier.com/locate/imavis

A simple recursive method for converting a chain code into a quadtree with a lookup table

Z. Chen[*], I.-P. Chen

Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan, ROC

Received 7 December 1999; revised 17 July 2000; accepted 4 September 2000

Abstract

We present a simple recursive method for converting a chain code into a quadtree. We generate the quadtree black nodes level by level, starting from the finest resolution (i.e. the bottom) level toward the coarsest resolution (i.e. the top) level. Meanwhile, at each resolution level a new object border is revealed after the removal of the black nodes. The chain code elements for this new object border can be easily generated. Thus, the quadtree black nodes and the chain code elements of the new object border are generated side by side and their generations constitute a basic cycle of the conversion process. We show the generations can be done with the aid of a lookup table. Finally, we compare our method with the existing two-phase methods exemplified by Samet's method in terms of the total numbers of major operations including node allocations, node color filling, and node pointer linking. It indicates that our method is conceptually simpler and runs faster. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Chain code; Quadtree; Node allocation; Node linking; Color filling; Table lookup

1. Introduction

Chain code and quadtree are two widely used data structures useful for the hierarchical or multi-resolution object shape representation and image compression [1–16]. The former is a boundary (or edge-based) representation, while the latter is a region-based representation. They can be derived from each other. Here, we address the conversion from the chain code to the quadtree one.

Consider an object in Fig. 1(a) whose chain code consists of a sequence of grid segments (or code elements) of equal length along the object boundary. The code elements have four possible pointing directions: 0(right), 1(up), 2(left) and 3(down). The quadtree representation is given in Fig. 1(e) in which four child nodes, indicated by 0, 1, 2, and 3, are linked to a parent node. The quadtree has different levels of resolutions, ranging from the finest to the coarsest. By convention, the self-intersection of the chain code is assumed not allowed unless the intersection occurs between the starting point and the ending point of the chain code, and also assume that the grid cell located on the right-hand side of a code element lies inside the object. The grid cells of size $1 * 1$ along the object border, which are encompassed by the chain code, are not necessarily the legal black nodes in the quadtree representation. Any quadtree black node must be a maximal square block inside the object derived from a recursive partition of the square image. A quadtree node can only have an area of 4^0, 4^1, 4^2,..., or 4^N, where $4^N = 2^N * 2^N$ is the size of the square image. The conversion of the quadtree nodes from the chain code has drawn the attention of many researchers [17–20]. There are four popular methods: Samet [17], Mark and Abel [18], Webber [19], and Lattanzi and Shaffer [20]. The inputs of these methods are in the form of a chain code or polygon. Their outputs are pointer-based quadtree or linear quadtree. These methods are essentially a two-phase method, which was originally proposed by Samet. Samet's method is first briefly described below.

Phase One (initiating color filling and generating the tentative quadtree): in this phase, the chain code is traced to get the border cells, which are colored in black (the shaded cells 1, 2, 3, etc., in Fig. 2(a)). A tentative or intermediate quadtree corresponding to border cells are constructed, as shown in Fig. 2(b). In the tentative quadtree, the border pixel nodes are colored in black. The parents or ancestors of any black node or gray node are colored in gray. All other nodes without having any black or gray descendants are called "indeterminate" nodes whose color is to be determined in the next phase. To construct the quadtree,

[*] Corresponding author. Tel.: +886-3-5731875; fax: +886-3-5723148.
E-mail address: zchen@csie.nctu.edu.tw (Z. Chen).

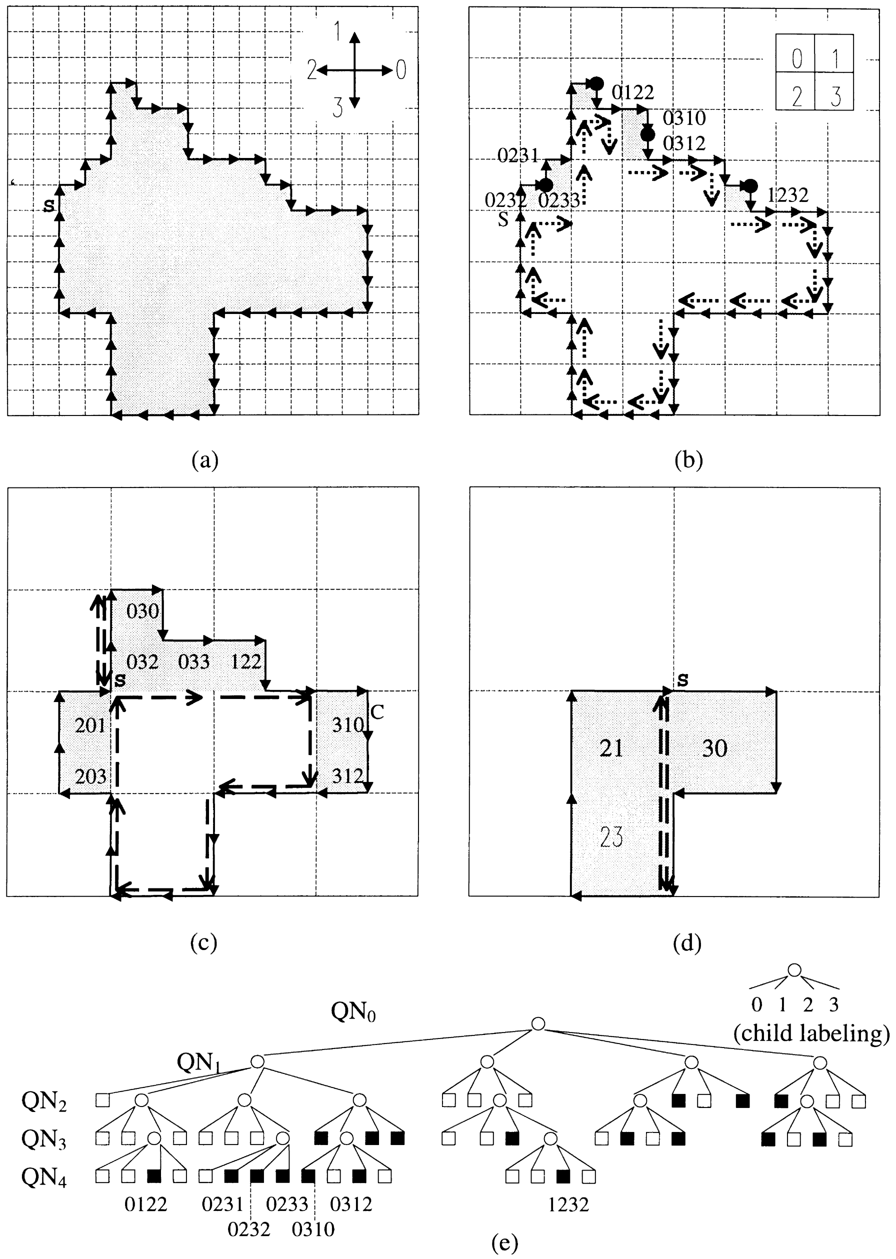


Fig. 1. (a) An object with a chain code, $CC_4 = 1010111030033000303000333322222223333222211111221111$. The code element marked with S is the starting element. (b) The shaded area is the pixel-level black node of QN_4. The center points in the parent nodes are marked as solid dots. The dash-lined arrows indicate the new code elements generated. (c) The chain code CC_3 and the quadtree nodes QN_3 all at level 3. (d) The chain code CC_2 and the quadtree nodes QN_2 all at level 2. (e) The final quadtree representation.

memory spaces have to be allocated to accommodate the above black, gray and indeterminate nodes and pointer links between parent and child nodes must be established.

Phase Two (completing the node coloring and the final quadtree): those “indeterminate” nodes right inside the object border are colored in black and those right outside the object border are in white. Similarly, through the new object borders the colors of all other “indeterminate” nodes are gradually determined as black or white. In addition to the color filling of “indeterminate” nodes, a node that was initially in a gray color is re-colored as black if its four son nodes are expanded and found all black; namely, a node merge takes place. After merge, the four son nodes are deleted and their memory allocations are freed.

The other three methods are basically similar to Samet’s method. In Mark and Abel’s method, the chain code representation is derived from a vector representation of the polygonal shape of the object. After the method traces the chain code to get the border grid cells, it uses the location code of each border grid cell to infer the colors of the

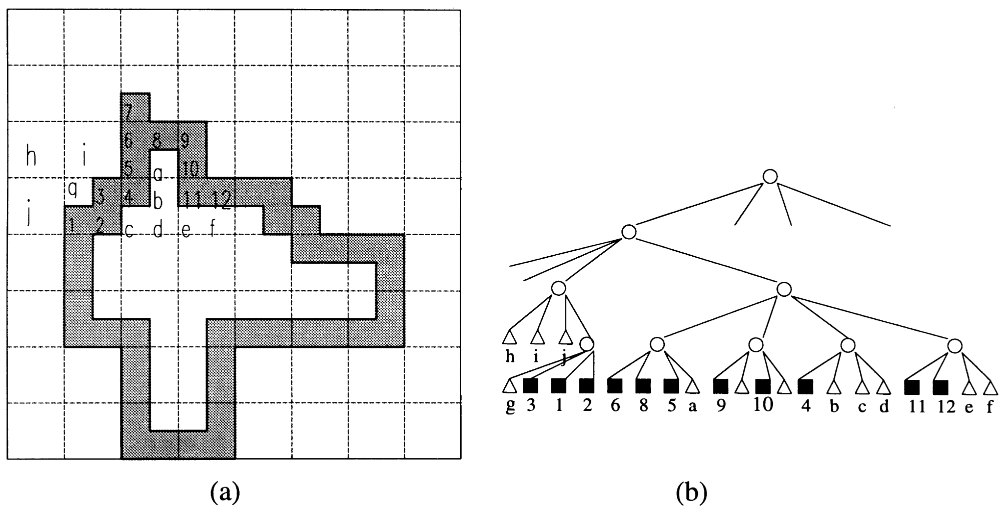


Fig. 2. (a) The border grid cells used in Samet's method. (b) Part of the tentative quadtree generated in Samet's method, the nodes a to j are labeled "indeterminate", i.e. their colors are to be decided later.

"indeterminate" nodes. Webber's method also first constructs the initial tentative quadtree and, then infers the colors of "indeterminate" nodes which is similar to Samet's method. But Webber's method shifts the chain code representation to a certain optimal position to help the color filling operation. Lattanzi and Shaffer's method gets the locational code of border grid cells which is similar to Mark and Abel's method. Then the border grid cells are sorted according to Morton order. The colors of "indeterminate" nodes are determined in a way similar to Samet's method.

The major drawback in the above two-phase methods is the need of merging four child nodes into a parent node. This is due to the lack of some mechanisms for judging whether the border grid cells are legal black nodes in the final quadtree? In Fig. 2 the border grid cells 1, 2, 3 are the final legal quadtree black nodes, but the border grid cells 4 and 5 are not, because they are contained in some other larger quadtree black nodes. It is not difficult to find out, during the chain code tracing, that the parent node containing the border grid cells 1, 2, 3 is pierced through by the chain code, so the three grid cells are the maximal black nodes of the quadtree. However, it is not so trivial to see immediately that the border grid cell 4 or 5 is contained in some larger quadtree black node, so neither of them is qualified as a quadtree black node.

We shall present a simple method to determine which border grid cells should be outputted as the legal quadtree black nodes during tracing the chain code. We then remove these black nodes to reveal the new object border. The new object border is associated with a new chain code whose element length is doubled. Quadtree black nodes are generated recursively and at the same time the chain code elements of the new object border are also generated to be used in the next pass or run. The process is repeated until the quadtree black nodes at all levels are generated. Fig. 1(b)–(d) illustrates the result of this recursive generation process, where the solid lines are the chain code elements at the current resolution level, the dashed lines are the chain code elements at the next coarser level. Furthermore, we point out it is possible to design a lookup table to speed up the conversion process. Comparisons with the existing methods indicate a better performance of the proposed method.

The rest of the paper is organized as follows. Section 2 describes the lemmas for validating a border grid cell as a legal quadtree black node at each resolution level. And the generation of the chain code elements for the new object border is given. Section 3 describes the table lookup method for generating the quadtree black nodes and the new chain code elements. Section 4 provides a performance comparison between the proposed method and the existing two-phase methods. Section 5 is the concluding remarks.

2. Generation of quadtree black nodes and new chain code elements

Let the notations of quadtree nodes and chain codes at various resolution levels be defined first. Let QN_i, $i = N, N-1, \ldots, 0$, be the set of quadtree black nodes at the ith level. (Here the Nth level is the finest resolution level.) Let CC_i denote the chain code of the object at the ith resolution level and CC_N is the given object chain code. We shall generate all sets of black nodes of QN_i, $i = N, N-1, \ldots, 0$, from the finest resolution level ($i = N$) to the coarsest resolution level ($i = 0$). First of all, refer to the grid points of the square image at the various resolution level in Fig. 1. We classify the grid points at each resolution level into three types, depending on: (1) if it is a grid point at the next

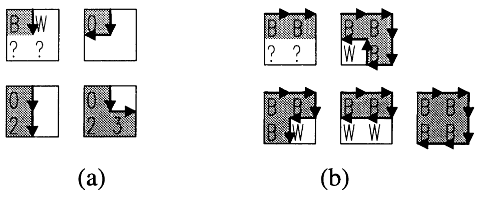


Fig. 3. Chain code sequences considered in Lemma 1 (a) and Lemma 2 (b).

coarser level (i.e. a grid point of a parent block); or (2) if it is the center point of a parent block; or (3) if it is a midpoint of the chain code element of the parent block (or simply the edge midpoint of a parent block).

We can choose to generate all quadtree black nodes at a given level first and then obtain the new chain code at a coarser level after the removal of all black nodes at the current level. Here the new chain code is obtained by tracing the new object border. However, the sequential generation of the quadtree black nodes and the new chain code is a waste of time. Instead, we will generate the quadtree black nodes and the new chain code elements side by side. We observe that if a parent node is pierced through by the chain code, then the black son nodes are the legal quadtree black nodes. And the new code elements associated with the new object border after removal of the generated black nodes can be immediately determined. In addition, it must be pointed out that only the four son nodes belonging to a common parent node are involved in the decision about whether they are the legal quadtree black nodes or not? Therefore, four consecutive code elements at a time are scanned in a basic cycle of the generation process. On the other hand, the starting point of the chain code is chosen to be a grid point at the parent level because the chain code at the next coarser level is defined over such grid points.

The following are two important lemmas for determining the legal quadtree black nodes during the chain code scanning.

Lemma 1. *If and only if any code element of CC_i enters the center point of a parent node, then there will be one to three black nodes of QN_i at the current level. The exact number of black nodes depends on the actual pattern of the succeeding code elements.*

Proof. (1) The 'if ' part. In Fig. 3(a) assume a code element of CC_i at the current level enters the center point of a parent node from the north direction (for the other directions the proof proceeds similarly). Since the right-hand-side grid cell of the code element is inside the object, so the color of child node 0 is black and the color of child node 1 is white. The colors of child nodes 2 and 3 depend on the types of the succeeding code elements, as shown in Fig. 3(a). Therefore, there is one or up to three current-level black nodes.

(2) The 'only if ' part. There are four grid cells or son nodes around the center point of a parent node. If these four nodes are all white (black), then the center point of the parent node is outside (inside) the object. So the center point of the parent node is not passed through by the chain code. On the other hand, if one to three black grid cells are around the center point of the parent node, then the center point of the parent node is located on the boundary of object and it will be reached by a code element. □

Lemma 2. *If two consecutive code elements of CC_i, $i = N, N-1, \ldots, 1$, at the current level are in the same direction and lie on the border of a common parent node, then there will be two to four black grid cells of QN_i at the current level. The exact number of quadtree black nodes depends on the actual pattern of the succeeding code elements.*

Proof. Consider the two code elements lying on the north border of a parent node (the proof works similarly for other borders), as shown in Fig. 3(b). Since the right-hand side grid cell of the code element is inside the object, so the colors of grid cell 0 and grid cell 1 are black.

The colors of grid cell 2 and grid cell 3 are determined by whether there is any succeeding code element cutting through the center point of the parent node:

1. If there is a code element entering the center point of the parent node, then at least one of grid cell 2 or grid cell 3 will be white (Fig. 3(b)). So there are two to three black grid cells at the current level.
2. If not, then these four grid cells are all black, and these four cells should not be outputted as the current level black nodes, since they are included in the black parent node at the next coarser level. The output of the black parent node will be handled at the next coarser level. □

As shall be seen, the output of the black quadtree nodes will be taken care of by the following-up chain code tracing. So the two code elements are replaced with a code element of a double length in the same direction to be used at the next coarser level.

We now describe how to scan the chain code CC_i at the current level by starting from the grid point at the next coarser level, and fetch at most 4 code elements at one time. We then generate the quadtree black nodes of QN_i at the current level and the next coarser level chain code CC_{i-1} in the following way.

When starting from a grid point at the next coarser level, the first code element must move to a midpoint of a parent node. Fig. 4 illustrates the 19 possible code element patterns when the initial code element goes in the upward direction; for the other cases of the initial code direction the analysis works similar. With respect to the first code direction, the second code element has two possible directions: (1) it will go to a second grid point at the next coarser level; or (2) it will go to the center point of the parent node.

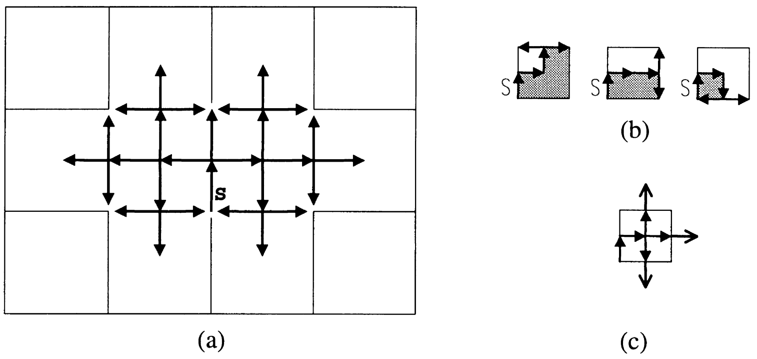


Fig. 4. (a) The 19 possible patterns of a code sequence consisting of four consecutive code elements, assuming the starting code element (labeled by s) is in the upward direction. (b) The cases in which the fourth code element stops at a grid point at the next coarser level. (c) The cases in which the fourth code element stops at a center point of a neighboring coarser node.

For the first case, these two code elements are in the same direction and, thus, satisfying Lemma 2. So we simply replace these two code elements by a code element of a double length at the next coarser level. And the next conversion cycle is to start from the second grid point.

For the second case the third code element must go to a midpoint of the parent node. Then the second and third code elements satisfy Lemma 1, so one to three black nodes of QN_i at the current level will be produced, depending on the direction of the fourth code element. Meanwhile, which grid point to start within the next conversion cycle depends on the fourth code element type. There are two possibilities: First, if the fourth code element moves to a grid point at the next coarser level (refer to Fig. 4(b)), then this grid point is chosen to be the new starting point. Second, if the fourth code element ends at a center point of a neighboring parent node (refer to Fig. 4(c)), then the grid point at the next coarser level located on the right-hand-side of the third and fourth code elements is the new starting point. In this case, we insert, after the third code element, a code element from the end point of the third code element to the new starting grid point. In this way, this pattern of four code elements falls in the same category of the preceding one.

After adding an extra code element after the third code element, we need to balance off by adding another code element in the reverse direction. This new code element is inserted before the fourth code element, so the inserted one and the original fourth one will become the first two code elements in the next conversion cycle.

When the above black nodes of QN_i are outputted, then a new object border is exposed which can be represented by some (or none when finished) new code element(s) at the next coarser level, each with a double length. It should be noticed that the new chain code may contain pairs of code elements in opposite directions and these pairs of elements cancel out by themselves (see examples in Section 3). So we need a post-processing stage to remove such code element to make sure that the processed chain code will satisfy our assumption of no self-intersection.

Next, consider the selection of an appropriate starting point. If the starting point of the chain code of CC_i at the current level is not a grid point of its parent node, then there are two ways to adjust the code element sequence. One way is to shift along the chain code until we come across a grid point at the next coarser level, and this grid point is chosen to be the new starting point. But even if such a grid point exists, it is sometimes a waste of time to find it, so we will not do so. Instead, we choose the second way, which works as follows:

1. If the starting code element goes from a midpoint to a grid point (Fig. 5(a)), then the grid point is chosen as the new starting point and the chain code is shifted by one element.
2. If the starting code element goes from a midpoint to a center point (Fig. 5(b)), then a pair of code elements with the opposite directions is inserted between the starting middle point and a grid point on the right-hand-side of the starting code element. The grid point chosen is the new starting point and the chain code is modified accordingly.
3. If the starting code element goes from a center point to a midpoint (Fig. 5(c)), then by shifting to the second code element, the chain code now starts from a midpoint. We can apply one of the above two ways to choose a new starting point.

Once we choose a grid point as the starting point for scanning the chain code, we can fetch four succeeding code elements to analyze. Notice that there are only a finite

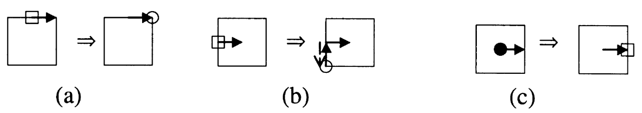


Fig. 5. The three cases for the adjustment of the starting point. The square indicates the midpoint; the circle indicates the grid point at the next coarser level.

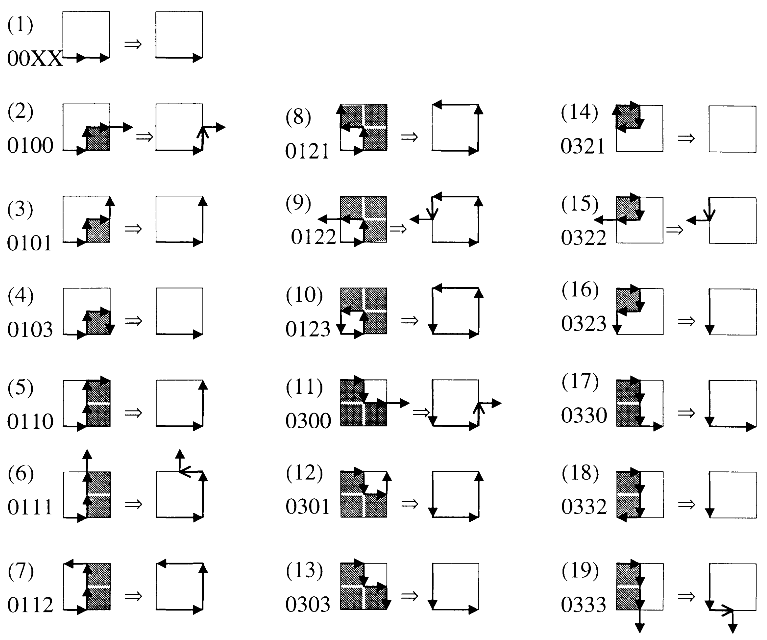


Fig. 6. The typical 19 patterns in the table lookup operation. The input is the four code elements. The shaded blocks are the output black quadtree nodes. The long arrowheads are the output new chain code elements of CC_{i-1} and the short arrowheads are the output adjusted code elements of CC_i.

number of patterns that these code elements can form, we shall propose a table lookup method for the generation tasks we have described above.

3. The table lookup method

In this section a table lookup method for generating black quadtree nodes and the new chain code elements is presented. We know that the total number of possible combinations of four code elements is $256 = 4^4$. Among these 256 combinations, some are illegal under the assumption that the self-intersection of the chain code is prohibited. There are totally 108 legal patterns, as described below. First, when the first two code elements are in the same direction, there are four categories denoted as 00XX, 11XX, 22XX, and 33XX, where XX may be from {00, 01,...,33}. Each of these four categories has only $3 * 3$ legal patterns. Secondly, when the first two code elements are not in the same direction, the number of legal patterns of four code elements is $72 = 4 * 2 * 3 * 3$. Totally there are $108 = 4 * 9 + 72$ legal patterns.

Now consider a typical code element pattern. Fig. 4(a) shows all the possible combinations of at most four code elements, assuming a starting code element (marked by s) is in direction 1. Fig. 6 shows the generated black nodes (shown by shaded areas), the generated code elements at the next coarser level (shown by long arrows), and the possible adjusted code elements at the current level (shown by short arrows). They are collected in a lookup table given in Table 1(a). Using input code pattern 0100 as an example (refer to Fig. 6). There is an output code element of CC_{i-1}, which is direction 0 (the long arrow pointing rightward in the right side of Fig. 6). There is a black quadtree node of QN_i, which is quadrant 3 (the lower right shaded region). Also there are two adjusted code elements of CC_i, which are directions 1 and 0 (the short arrows in the right side). Similarly, we can rotate all the patterns in Fig. 6 counterclockwise by 90, 180, and 270° to derive the results shown in Table 1(b)–(d).

The chain code in Fig. 1(a) is used as an illustrative example to show how to use the lookup table for creating the possible black quadtree nodes and the new chain code elements. The initial chain code is given by

$$CC_4 = 10101110300330003030003333 \\ 2222223333222221111221111.$$

The letter "s" indicates the starting point. The scanning of the chain code is performed as follows:

1. The starting point "s" of CC_4 is a grid point at the next coarser level. Fetch four code elements CC_4 (1) to CC_4 (4) whose code pattern is 1010. From the lookup table, we output three black nodes with Morton orders of 0231, 0232, and 0233 and two new code elements CC_3 (1) and CC_3 (2) at the next coarser level, whose pattern is 01. These results are shown in Fig. 1(b).
2. Since there are no output of the adjusted code elements at the current level, fetch the next four code elements from the remaining chain code whose code pattern is 1110.
3. From the lookup table corresponding to the input pattern

Table 1
The lookup table for generating the new chain code element(s) of CC_{i-1} at the next coarser level, the quadtree black node(s) of QN_i, and the adjusted code elements of CC_i for all 4^4 possible input code patterns of CC_i. The input code element marked as X is an arbitrary element

(a)				(b)				(c)				(d)			
Input code pattern of CC_i	New code element(s) of CC_{i-1}	Adjusted code elements of CC_i	Black quadtree nodes of QN_i	Input code pattern of CC_i	New code element(s) of CC_{i-1}	Adjusted code elements of CC_i	Black quadtree nodes of QN_i	Input code pattern of CC_i	New code element(s) of CC_{i-1}	Adjusted code elements of CC_i	Black quadtree nodes of QN_i	Input code pattern of CC_i	New code element(s) of CC_{i-1}	Adjusted code elements of CC_i	Black quadtree nodes of QN_i
00XX	0	XX	–	1000	0	10	2,3	20XX				3000	30	10	32
0100	0	10	3	1001	01	–	2,3	2100	2	10	3	3001	301	–	32
0101	01	–	3	1002				2101	1	–	3	3002			
0102				1003	0	–	2,3	2102				3003	30	–	32
0103	0	–	3	1010	01	–	1,2,3	2103	–	–	3	3010	301	–	1,2,3
0110	01	–	1,3	1011	01	21	1,2,3	2110	1	–	1,3	3011	301	21	1,2,3
0111	01	21	1,3	1012	012	–	1,2,3	2111	1	21	1,3	3012	3012	–	1,2,3
0112	012	–	1,3	1013				2112	12	–	1,3	3013			
0113				102X				2113				302X			
0120				1030	0	–	2	2120				3030	30	–	2
0121	012	–	0,1,3	1031				2121	12	–	0,1,3	3031			
0122	012	32	0,1,3	1032	–	–	2	2122	12	32	0,1,3	3032	3	–	2
0123	0123	–	0,1,3	1033	1	03	2	2123	123	–	0,1,3	3033	3	03	2
013X				11XX	1	XX	–	213X				31XX			
02XX				120X				22XX	2	XX	–	320X			
0300	30	10	0,2,3	1210	1	–	1	2300	230	10	0,2,3	3210	–	–	1
0301	301	–	0,2,3	1211	1	21	1	2301	2301	–	0,2,3	3211	3	21	1
0302				1212	12	–	1	2302				3212	2	–	1
0303	30	–	0,2,3	1213				2303	230	–	0,2,3	3213			
031X				1220				231X				3220			
0320				1221	12	–	0,1	2320				3221	2	–	0,1
0321	–	–	0	1222	12	32	0,1	2321	2	–	0	3222	2	32	0,1
0322	0	32	0	1223	123	–	0,1	2322	2	32	0	3223	23	–	0,1
0323	3	–	0	1230	1230	–	0,1,2	2323	23	–	0	3230	230	–	0,1,2
0330	30	–	0,2	1231				2330	230	–	0,2	3231			
0331				1232	123	–	0,1,2	2331				3232	23	–	0,1,2
0332	3	–	0,2	1233	123	03	0,1,2	2332	23	–	0,2	3233	23	03	0,1,2
0333	3	03	0,2	13XX				2333	23	03	0,2	33XX	3	XX	–

1110, we obtain the output of a new code element of CC_{i-1} (i.e. “1” in this case). There are no outputs of black quadtree nodes, but two adjusted code elements: 10. We fetch two more code elements from the remaining chain code CC_i.
4. In this way, the remaining code elements are scanned four at a time. The outputted black nodes are 0122, 0310, 0312, and 1232. At the end of the first pass of the chain code scanning, we obtain a new chain code at the next coarser level which is $CC_3 =$ 01103003003322233221121. Since the initial starting point of CC_3 is not a grid point at the next coarser level so it is adjusted to start at the second position, resulting in the modified chain code being 11030030033222332211210 (see Fig. 1(c)).
5. For the second pass of scanning, the chain code CC_3 is scanned to construct QN_3, resulting the black nodes 030, 032, 033, 122, 310, 312, 201, and 203 in order (refer to Fig. 1(c)). The new chain code at the next coarser level CC_2 is 1300323211. Since there is a self-intersection in the first two code elements 13, they are cancelled out. The new starting point of CC_2 is not a grid point at the next coarser level, so the chain code is adjusted to 03232110.
6. For the third pass, the new chain code CC_2 is scanned to construct QN_2, producing the nodes 21, 23, and 30 (refer to Fig. 1(d)). The resultant chain code at the next coarser level is $CC_1 = 31$. After the cancellation of pairs of opposite code elements, CC_1 becomes empty, and so is CC_0. No black nodes of QN_1 and QN_0 are generated.
7. The final quadtree is shown in Fig. 1(e).

An overall description of the above table lookup algorithm for the conversion process is given below.

Algorithm:

Input: The chain code of the object, CC_N, placed in NextCC[ ].
Output:
(1) The output of black quadtree nodes QN_i placed in OutputNode[ ] and the output of new chain code CC_{i-1} at the next coarser level placed in OutputCode[ ] for $i = N, N-1, \ldots, 1$.
(2) The final quadtree representation of the object.
The data structures used in the algorithm are described below.

```
(a) short Morton[N];//$2^N * 2^N$ is the size of square image.
(b) short CurCC [M], NextCC[M];//M is a sufficient large number.
(c) short InputPattern[4], OutputNode[3], OutputCode[4], AdjustedCode[2];
(d) int indexCurCC, indexNextCC;//index of CurCC [ ], NextCC[ ].
(e) struct node {
      short color;
      struct node * parent, * son[4]
    }
```

The array Morton [ ] stores the Morton order of the black node(s) listed in the data OutputNode [ ]. The array InputPattern [ ] stores the input code pattern used for table lookup. The arrays OutputNode [ ], OutputCode [ ], and AdjustedCode [ ] store the outputs of the table lookup operation.

Next, the functions called are described below.

```
(a) PointerWrite (P,j,S) {P.son [j] = S; S.parent = P;}
(b) AdjustInputPattern ( ) {
      InputPattern[0] = AdjustedCode[0];InputPattern[1] = AdjustedCode[1];
      InputPattern[2] = CurCC[indexCurCC + 1];
      InputPattern[3] = CurCC[indexCurCC + 2];
      IndexCurCC + = 2;}
(c) AppendNextCCWithOutputCode ( ) {
      for ($i = 0$; $i <$ Length(OutputCode); $i$++)
      NextCC[indexNextCC + $i$] = OutputCode[$i$];
      indexNextCC + = Length(OutputCode);}
(d) GenerateQTNode ( ) {
      int $i$;
      struct node * CurNode, * NextNode;
      CurNode = ROOT;//Start from ROOT;
      for ($i = 0$; $i <$ Length(Morton); $i$++) {
        If (CurNode.son[Morton[$i$]] = nil) {
          NextNode = Allocate ( );
          PointerWrite(CurNode, Morton[$i$], NextNode);
          Coloring(NextNode, ‘Gray’);
        }
        CurNode = CurNode.son[Morton[$i$]];
      }
```

Table 2
The comparison of major operations between Samet’s method and our method

Operation	Method	Samet’s method		Our method[a]
		Phase 1	Phase 2	
Node allocation	Black node	Yes	No	Yes
	Gray node	Yes	No	Yes
	Indeterminate node	Yes	No	No
Node coloring	Black	Yes	Yes	Yes
	Gray	Yes	No	Yes
	White	No	Yes	No
Pointer writing		Yes	No	Yes
Node merge	Node deallocation	No	Yes	No
	Node recoloring	No	Yes	No
	Pointer rewriting	No	Yes	No
Chain code	Reading	Yes	No	Yes
	Updating	No	No	Yes
Depth-first tree traversal		No	Yes	No

[a] Only the black nodes and gray nodes are constructed in our method. To produce the complete quadtree, the child nodes, which are not colored black, are taken as white by default.

```
    for ($i = 0$; $i <$ Length(OutputNode); $i++$) {
        NextNode = Allocate( );
        PointerWrite(CurNode, OutputNode[$i$], NextNode);
        Coloring(NextNode, 'Black');
    }
}
```

There are other minor functions. The function FetchFourInputPattern( ) fetches four code elements from CurCC[ ].The function NotEndOfCurCC( ) checks whether the end of CurCC is reached. The function Cancellation(-NextCC) performs any possible cancellations between pairs of opposite code elements to eliminate the self-intersection of code elements in CC_{i-1}.

Next, the main procedure of the algorithm is given:

```
Main( ) {
    While (Length(NextCC)! = 0) {//Start a new pass of the algorithm.//
        CurCC = NextCC;                indexCurCC = 0;
        indexNextCC = 0;
        SelectStartingPoint(CurCC);//Select a grid point at the next coarser level.//
        while (NotEndOfCurCC( )) {
            if (Length(AdjustedCode) = = 0) {
                FetchFourInputPattern( );//
                //Fetch 4 code elements from CurCC[ ].
            }
            else {
                AdjustInputPattern( );//Fetch 2 code elements from
                //AdjustedCode[ ] and 2 code elements from CurCC[ ].
            }
            TableLookup(InputPattern, OutputNode, OutputCode, AdjustedCode);
            GenerateQTNode( );//Generate $QN_i$ at the current level//
            AppendNextCCWithOutputCode( );
            //Generate $CC_{i-1}$ at the next coarser level.
        }
        Cancellation(NextCC);//Eliminate any possible self-intersection of chain code in $CC_{i-1}$.//
        }
    }
```

4. Performance comparison

It is difficult to give an exact comparison between our method and the two-phase methods, because they contain some different operations. But we shall give a gross estimate of their computation times. For comparison purpose, major operations of our method and the two-phase methods represented by Samet's method are listed in Table 2. To gain some insight into the comparison, we show the more detailed results of the quadtree generation by the two methods in Figs. 8 and 9 for two representative objects shown in Fig. 7(a) and (b). We summarized the execution numbers of major operations for all objects in Fig. 7 in Table 3. From these examples, we conclude:

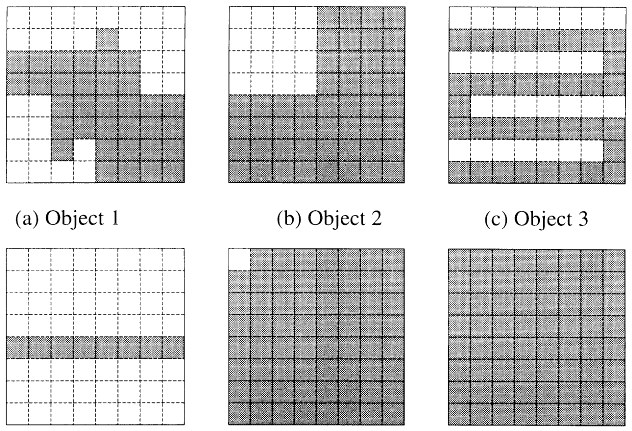

(a) Object 1 (b) Object 2 (c) Object 3
(d) Object 4 (e) Object 5 (f) Object 6

Fig. 7. Six object images used in the performance evaluation.

1. The two-phase method allocates more nodes than required in the final quadtree representation, while our method only generates the black nodes in the final quadtree representation.
2. The two-phase method invokes a coloring process utilizing a local, single code element to decide the color of its corresponding quadtree node, while our method uses a look-ahead or global process to determine any possible black nodes in the finalized quadtree. As a consequence, the two-phase method may produce a temporary black node, resulting in a later node merge; our method does not produce any temporary black nodes.
3. The two-phase method has to check the color of nodes which are white in phase 2; our method only concerns with the black nodes. The white nodes in the final quadtree are by default.
4. The color determination mechanism in the two-phase method is by checking the spatial relation between the particular node and the object boundary location; while a predetermined table is lookedup in our method.
5. The order of node expansion (i.e. its construction) in phase 2 of the two-phase method is decided by the depth-first traversal of the tentative quadtree, while in our method, it is always by the chain code tracing at the various resolution levels.

Based on the above observation, we can compare the computation time required by these two methods as

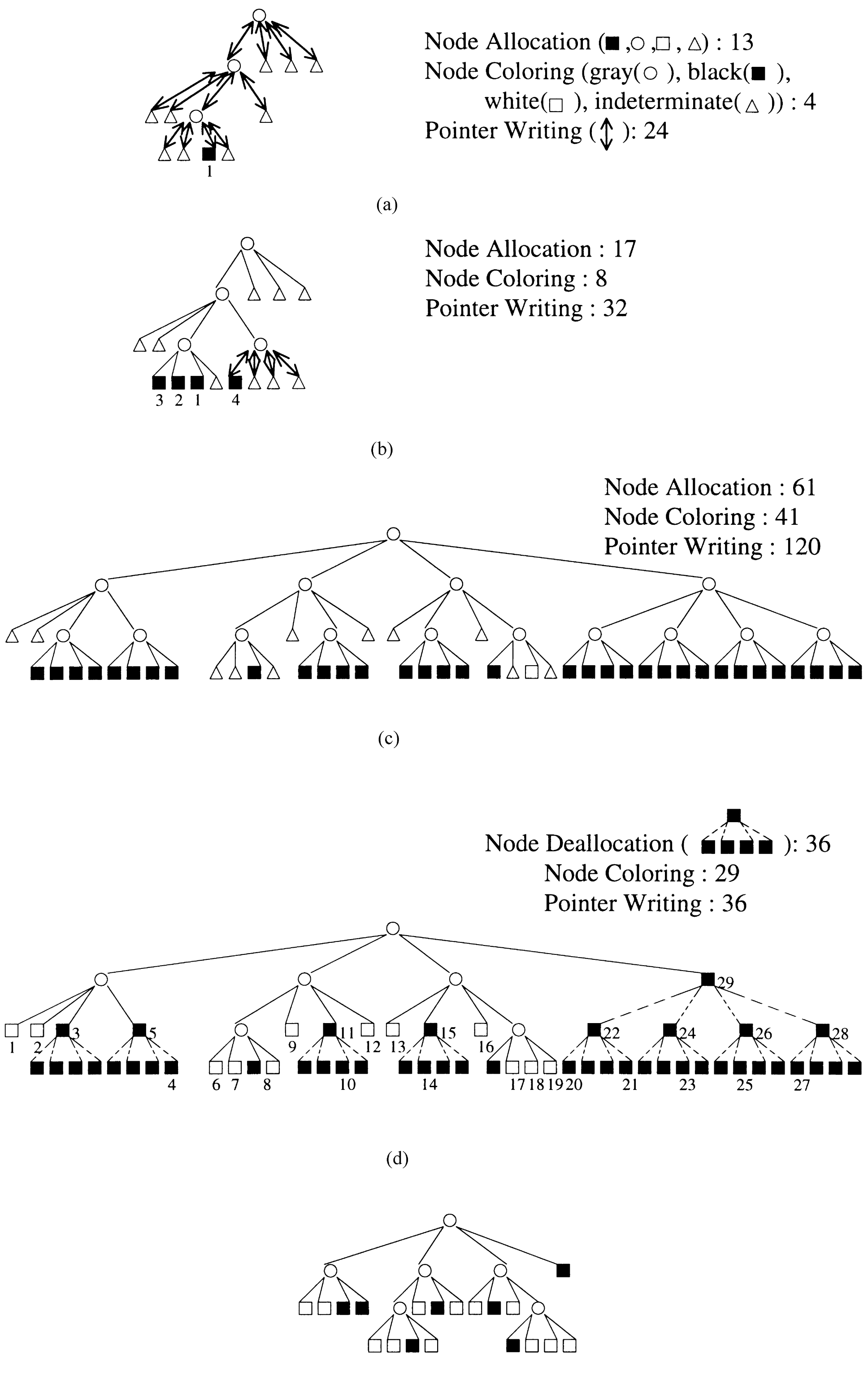
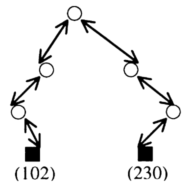
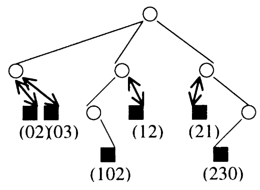


Fig. 8. The quadtree generation for Object 1 by the two methods. (a)–(e) The intermediate results of Samet's method. The numbers indicate the sequence of the operations taking place. (f)–(i) The intermediate results of our method. The numbers in brackets indicate the Morton code of the generated black nodes.



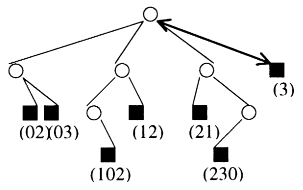
Node Allocation : 7
Node Coloring : 7
Pointer Writing : 12

(f)



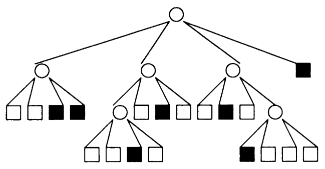
Node Allocation : 12
Node Coloring : 12
Pointer Writing : 22

(g)



Node Allocation : 13
Node Coloring : 13
Pointer Writing : 24

(h)

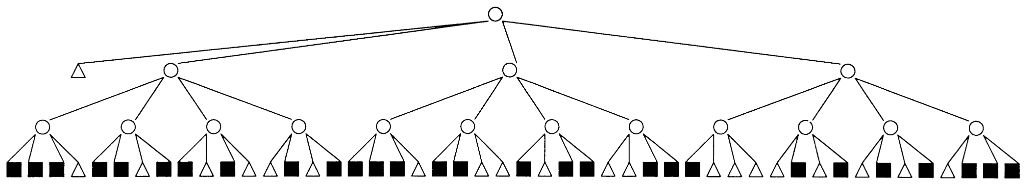


Node Allocation : 25
Node Coloring : 25
Pointer Writing : 48

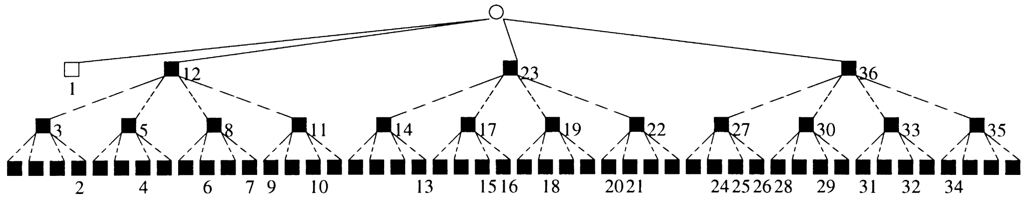
(i)

Fig. 8. (*continued*)

follows. The main common operations executed by both the methods are: (1) memory allocation for node constructions and memory deallocation after node merging; (2) write/rewrite of the color field in the node data structure; (3) write/rewrite of the five pointers in the node data structure; and (4) read of the chain code element(s). We shall examine the execution of the first three operations. The reading time of the chain code element(s) in the last operation is negligible. Overall speaking, the dominant time is the memory allocation time for node constructions. The main differences between these two methods are: (1) the decision on the next node(s) to be expanded; and (2) the color determination process used by two methods, as already described above. In regard to the node expansion order phase 2 of the two-phase method does the depth-first traversal of the tentative quadtree. Our method traces the chain code elements to get the black node(s) for expansion. Furthermore, the table lookup operation also produces the new chain code elements and/or the adjustment in the current chain code. Generally speaking, it takes more time to implement the depth-first tree traversal than the table lookup operation. For simplicity, we ignore these different operations required by the individual methods. We shall concentrate on the



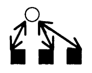
(a)



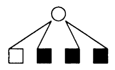
(b)

(Empty tree)

(c)



(d)



(e)

Fig. 9. The quadtree generation for Object 2 by the two methods. (a) and (b) The generation results obtained by Phase 1 and Phase 2 of Samet's method. (c)–(e) The generation results obtained by our method.

execution time of the common operations taking place in both methods. Lemmas 3–5 summarize the differences in the total operation numbers of: (i) node allocation/deallocation; (ii) node coloring; and (iii) node pointer linking.

Lemma 3. *The total number of nodes expanded (or generated) by Samet's method is larger than that of the regular quadtree by a number that is equal to the total number of nodes that are deleted due to any node merges taking place. The total number of nodes expanded by our method is equal to that of the regular quadtree.*

Lemma 4. *The total number of color filling operations in Samet's method is larger than that of our method by a number that is equal to the total number of merged nodes times 1.25 (i.e. the total number of black nodes deleted plus their re-colored parent nodes).*

Lemma 5. *The total number of pointer links constructed in Samet's method is larger than that of our method by a number that is equal to the total number of merged nodes times 3 (i.e. the total number of pointer links constructed*

Table 3
The comparison of the numbers of operations used by Samet's method and our method

Method	Operation	Objects					
		Object 1	Object 2	Object 3	Object 4	Object 5	Object 6
Samet's method	Node allocation	61	65	85	29	69	69
	Node coloring	70	80	85	29	83	86
	Pointer writing	156	188	168	56	192	204
	Node deallocation	36	60	0	0	56	68
Our method[a]	Node allocation	25	5	85	29	13	1
	Node coloring	25	5	85	29	13	1
	Pointer writing	48	8	168	56	24	0
	Node deallocation	–	–	–	–	–	–

[a] Refer to Lemmas 3–5.

between the parent nodes and their child nodes before and after node merges).

The computational time required by each of the two methods is roughly equal to the summation of products of the average execution time per operation and the total operation number of each operation over the three main operations. From Lemmas 3–5, the computational time difference between the two methods is approximately equal to

$$\text{Computational Time Difference} = \text{Ave\_Time\_of\_Node\_Allocation} * N_{\text{DEL}} + \text{Ave\_Time\_of\_Node\_Coloring} * N_{\text{DEL}} * 1.25 + \text{Ave\_Time\_of\_Pointer\_Writing} * N_{\text{DEL}} * 3.$$

where N_{DEL} is the total number of nodes deleted in Phase 2 of Samet's method.The overhead of our method is the need of generating the chain code for each level. The new chain code is generated by using the lookup table, so the generation is fast. The total number of table lookup operations is to be calculated below. Recall that four code elements are fetched at a time. The length of the new chain code at the next coarser level is roughly one half of that of the current level, that is, $\mathrm{B}_{i-1} \approx 0.5 * \mathrm{B}_i$, where B_i is the length of chain code at the ith level. Therefore, the total number of table lookup operations is approximately equal to

$$(\mathrm{B}_N + \mathrm{B}_{N-1} + \mathrm{B}_{N-2} + \cdots + \mathrm{B}_0)/4 \approx (\mathrm{B}_N + \mathrm{B}_N/2 + \mathrm{B}_N/4 + \cdots + 1)/4 \approx (1/2) * \mathrm{B}_N,$$

where B_N is the length of the original chain code, that is, the boundary length of the object. So this number is bounded and its operation time is minor, compared to other operation times.

5. Conclusion

In this paper we have presented a simple recursive method for converting a chain code representation into a quadtree one. Lemmas for the determination of the quadtree black nodes are derived. We generate the quadtree black nodes level by level from the finest one to the coarsest one. Meanwhile, at each resolution level a new object border is revealed after the removal of the black nodes at that level. The chain code for this new object border can be easily generated. Thus, the generation of the quadtree black nodes and the chain code of the new object border constitutes a basic cycle of the conversion process. We show the generation can be done with a table lookup. Our conversion method runs faster than Samet's method in terms of the total execution time of major operations, as indicated by the lemmas. Some representative examples are given to illustrate the better performance of our method.

References

[1] H. Freeman, Computer processing of line-drawing images, ACM Computing Survey 6 (1) (1974) 57–97.

[2] P. Zingaretti, M. Gasparroni, L. Vecci, Fast chain coding of region boundaries, IEEE Transactions on Pattern Analysis and Machine Intelligence 20 (1998) 407–416.

[3] H. Samet, Applications of Spatial Data Structures, Addison-Wesley, New York, 1989.

[4] H. Samet, The Design and Analysis of Spatial Data Structures, Addison-Wesley, New York, 1989.

[5] H. Samet, The quadtrees and related hierarchical data structures, ACM Computing Survey 16 (1984) 187–260.

[6] J. Koplowitz, J. DeLeone, Multilevel resolution of digital binary images, Proceedings of the IEEE International Symposium on Information Theory, 1995, p. 361.

[7] P. Nunes, F. Pereira, F. Marques, Multi-grid chain coding of binary shapes, Proceedings of the IEEE International Conference on Image Processing, 1997, pp. 114–117.

[8] G.R. Martin, R.A. Packwood, M.K. Steliaros, Scalable description of shape and motion for object-based coding, Proceedings of the IEEE International Conference on Image Processing and its Applications, 1999, pp. 157–161.

[9] W. Li, W. Li, A fast fractal image coding technique, Proceedings of the International Conference on Signal Processing, 1998, pp. 775–778.

[10] R.A. Packwood, M.K. Steliaros, G.R. Martin, Variable size block matching motion compensation for object-based video coding, Proceedings of the International Conference on Image Processing and Its Applications, 1997, pp. 56–60.

[11] Y.-C. Chang, B.-K. Shyu, J.-S. Wang, Region-based fractal image compression with quadtree segmentation, Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal, 1997, pp. 3125–3128.

[12] I. Gargantini, H.H. Atkinson, Linear quadtrees; a blocking technique for contour filling, Pattern Recognition 17 (1984) 285–293.

[13] H.H. Atkinson, I. Gargantini, T.R.S. Walsh, Filling by quadrants or octrants, Computer Vision, Graphics, and Image Processing 33 (1986) 138–155.

[14] G.M. Hunter, K. Steiglitz, Operations on images using quad trees, IEEE Transactions on Pattern Analysis and Machine Intelligence 1 (1979) 145–153.

[15] G.M. Hunter, K. Steiglitz, Linear transformation of pictures represented by quad trees, Computer Graphics and Image Processing 10 (1979) 289–296.

[16] H. Samet, R.E. Webber, On encoding boundaries with quadtrees, IEEE Transactions on Pattern Analysis and Machine Intelligence 6 (1984) 365–369.

[17] H. Samet, Region representation: quadtree from chain codes, Communications of ACM 23 (1980) 163–170.

[18] D.M. Mark, D.J. Abel, Linear quadtrees from vector representations of polygons, IEEE Transactions on Pattern Analysis and Machine Intelligence 7 (1985) 344–349.

[19] R.E. Webber, Analysis of Quadtree Algorithms, PhD dissertation, Computer Science Department, University of Maryland, College Park, MD, 1984.

[20] M.R. Lattanzi, C.A. Shaffer, An optimal boundary to quadtree conversion algorithm, CVGIP: Image Understanding 53 (1991) 303–312.