ABSTRACT

New Algorithms for Mid-Crack Codes in Image Processing

by
Wai-Tak Wong

The chain code is a widely-used description for a contour image. Recently, a mid-crack code algorithm has been proposed as another more precise method for image representation. New algorithms using this new mid-crack code for image representation, restoration, and skeletonization are developed. The efficiency and accuracy can be increased obviously.

Firstly, the conversion of a binary image with multiple regions into the mid-crack codes is presented. A fast on-line implementation can be achieved using tables look-up. The input binary image may contain several object regions and their mid-crack codes can be extracted at the same time in a single-pass row-by-row scan. The perimeter and area of each region can be obtained during the execution of the algorithm. The inclusion relationship among region boundaries also can be easily determined.

Secondly, a simple and fast algorithm for the restoration of binary images based on mid-crack codes description is proposed. The algorithm developed has the advantages of speed, simplicity, and less storage. The algorithm also can be applied to gray-scale images with multiple regions efficiently.

Thirdly, it was observed that there exist four problems when running on some images with an in-contour in the restoration algorithm by Chang and Leu. We present the problems by a counterexample and propose simple improvements to modify the results so that the modified algorithm will allow the robustness, flexibility and correctness of the region filling and the complete reconstruction of an image. The idea of the improvement is similar to that of the restoration from mid-crack code description.

Finally, a new thinning algorithm for binary images based on the safe-point testing and mid-crack code tracing is established. Thinning is treated as the deletion of nonsafe border pixels from contour to the center layer-by-layer. The deletion is determined by masking a 3×3 weighted template and table look-up. The resulting skeleton does not require cleaning or pruning. The skeleton obtained possesses single-pixel thickness and preserves connectivity. The algorithm is very simple and efficient since only boundary pixels in each iteration are processed and look-up tables are used.

New Algorithms for Mid-Crack Codes in Image Processing

by
Wai-Tak Wong

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science
Department of Computer and Information Science
May, 1992

APPROVAL PAGE

New Algorithms for Mid-Crack Codes
in Image Processing

by
Wai-Tak Wong

[illegible] 5/7/92

Dr. Frank Y. Shih, Thesis Advisor
Assistant Professor of Computer and
Information Science Department, NJIT

BIOGRAPHICAL SKETCH

Author: Wai-Tak Wong

Degree: Master of Science in Computer Science

Date: May, 1992

Date of Birth:

Place of Birth:

Undergraduate and Graduate Education:

- Master of Science in Computer Science, New Jersey Institute of Technology, Newark, NJ, 1992
- Bachelor of Science in Chemical Engineering, Nation Taiwan University, Taipei, Taiwan, Republic of China, 1986

Major: Computer and Information Science

**This thesis is dedicated to
our almighty GOD and Jesus Christ**

ACKNOWLEDGEMENT

First, I would like to thank my GOD and Jesus Christ.

At this time, I would also like to thank my advisor Dr. Frank Y. Shih. Without his help in finanical support and education, I think I cannot grow up as well as my present status. Next, thanks the technical support from Yui-Liang Chen. At last, I am glad to thank my wife and my parents for their endless love.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The contour representation of a binary image is determined by specifying a starting point and a sequence of moves around the borders of each region. Current methods of contour tracing are based on the chain code or crack code concept [1,2,3]. The chain code moves along a sequence of the center of border points, while the crack code moves along a sequence of ''cracks'' between two adjacent border points. Typically, they are based on the 4- or 8-connectivity of the segments, where the direction of each segment is encoded by using a numbering scheme, such as 3-bit numbers $\{i \mid i = 0, 1, \cdots, 7\}$ denoting an angle of $45i^\circ$ counter-clockwise from the positive x-axis for a chain code, or 2-bit numbers $\{i \mid i = 0, 1, 2, 3\}$ denoting an angle of $90i^\circ$ for a crack code. The elementary idea of the chain or crack coding algorithm is to trace the border-pixels or cracks and sequentially generate codes by considering the neighborhood adjacency relationship.

The chain and the crack codes can be viewed as a connected sequence of straight line segments with specified lengths and directions. An obvious disadvantage of the chain code is observed when we use it to compute the area and perimeter of an object. Referring to Fig. 1, the inside chain code appears to underestimate the area and perimeter while the outside chain code overestimates them. The disadvantages in the crack code are that much more codes are generated and the perimeter is much overestimated. The mid-crack code [4], located in between, should make a more accurate computation of the geometric features.

When the contours of multiple objects in an image are extracted, one of the most common problems is to fill the region inside each contour. A region consists of a group

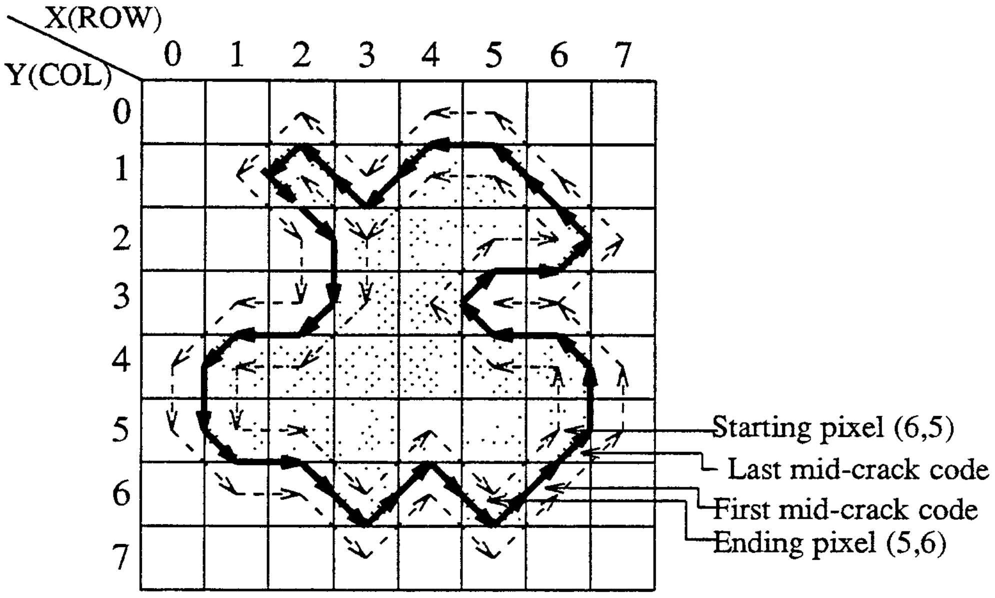


Fig. 1. Silhouette with the inside and outside chain coded contours (dashed lines) and the mid-crack coded contour (solid line).

of adjacent, connected pixels. The task of filling primitives can be divided into two parts: the decision of which pixels to fill and the easier decision of with what value to fill them. If the image is binary, we assign the same value to each pixel lying on a scan line running from the left edge to the right edge; i.e., fill each span from $x_{\min}$ to $x_{\max}$. Spans exploit a primitive's *spatial coherence*: the fact that primitives often do not change from pixel to pixel within a span or from scan-line to scan-line. We exploit coherence in general by looking for only those pixels at which changes occur.

Skeletonization or thinning is a very important preprocessing step in pattern analysis such as industrial parts inspection [5], fingerprint recognition [6], optical character recognition [7], and biomedical diagnosis [8]. One advantage of skeletonization is the reduction of memory space required for storing the essential structural information presented in a pattern. Moreover, it simplifies the data structure required in pattern analysis. Most of the skeletonization algorithms require iterative passes through the whole image, or at least through each pixel of the object considered. At each pass, a relatively complicated analysis over each pixel's neighborhood must be performed, that makes the algorithms time-consuming.

In Chapter 2, a few relative literatures are reviewed. In Chapter 3, a new single-pass algorithm for extracting the mid-crack codes of multiple regions is presented. In Chapter 4, a counterexample of a fast algorithm for restoration of images based on chain codes description is described. In Chapter 5, a new algorithm for the restoration of binary and gray-scale images by using contour mid-crack codes description is introduced. In Chapter 6, a new safe-point thinning algorithm based on the mid-crack code tracing is illustrated. In Chapter 7, future approach is proposed and we make a conclusion.

CHAPTER 2

LITERATURE REVIEW

2.1 Contour Representation

A method of using the run-length coding to generate the chain code was presented in [9]. But, the COD-S and ABS-S tables must be produced in the first run-length step before the chain code generation and linking phases start. They also suggested a useful concept to deal with the inclusion relationship between boundaries. Besides, a RC-code (raster-scan chain code) introducing the max-point and min-point concept, especially, a linking concept of relation links was proposed in [10]. Another single-pass algorithm [11] for generating the chain code adopts the use of a step-by-step concept, such as chain link generation, a link segment data structure, and a junction of links.

The mid-crack code [4] is a variation and an improvement of the traditional tracing methods between the chain code and the crack code. In contrast to Freeman chain code, which moves along the center of pixels, the mid-crack code moves along the edge mid-point of a pixel producing codes of links. For the horizontal and vertical moves, the length of a move is 1, and for diagonal moves, it is $\sqrt{2}/2$. If the crack is located in between two adjacent object pixels in the vertical direction, it is said to be on a vertical crack. Similarly, if the crack is located in between two adjacent object pixels in the horizontal direction, it is said to be on a horizontal crack.

Fig. 2 shows the Freeman chain code and the mid-crack code on the vertical and the horizontal cracks. There are two restrictions on the moves in the mid-crack code. If the move is from the vertical crack, the codes 0 and 4 are not allowed. Similarly, the codes 2 and 6 are not allowed in the moves from the horizontal crack. The experimental verification of the mid-crack code in the area and perimeter computations is shown in [4,12,13] where the mean perimeter error value is -0.074% and the mean area error is -0.006%. Therefore the mid-crack code is a desirable alternative method in contour tracing with its benefit in accuracy.

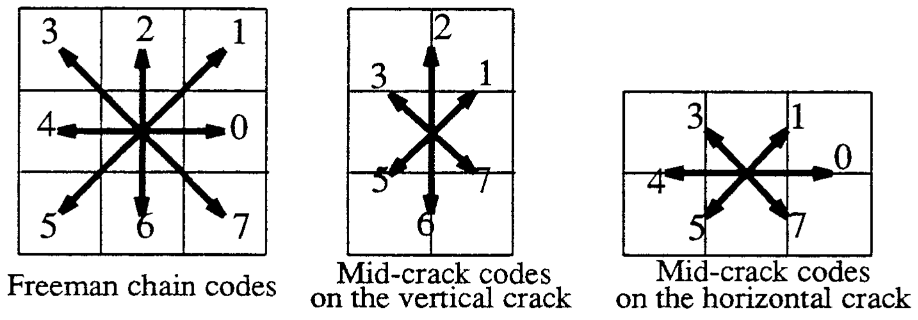


Fig. 2. The Freeman chain codes and the mid-crack codes on the vertical or horizontal cracks.

It is possible to invert a closed mid-crack code sequence into its corresponding silhouette. An internal boundary of the 8-connectedness object (or foreground) being traced counter-clockwise can be reconstructed by treating it as the 4-connectedness background being traced clockwise. A disadvantage of the mid-crack code is that it is always longer than the Freeman chain code. An algorithm of conversion between a mid-crack code sequence and a chain code sequence is described in [4] to complement the defects as a compression process.

2.2 Region Filling

There are several algorithms performing the region-filling based on chain codes [14] by the technique of parity checking [15,16] or a method of seed growing [15,17]. The technique of parity checking is often-used because only a single scan is required instead of the iterations required in the seed-growing method. However, it is possible to produce an incorrect count of the number of intersections if points from two or more sides are mapped on the identical pixel. Besides, a problem will arise if the test line is tangent to the contour. A fast algorithm for the restoration of an image was presented by Chang and Leu [18]. The algorithm uses the technique of parity checking based on contour direction chain codes description [15,16,19]. The goal of the algorithm is to convert the chain codes description (boundary representation) [1] into the y-axis partition (region representation) [20] because it is much easier to derive geometric properties for a shape from the y-axis representation. Merrill changes the original boundary into an augmented boundary, such that the extrema and inflections are repeatedly listed and require for conditions checking.

2.3 Thinning Algorithm

A practical problem with the definition of skeleton is that circular neighborhoods cannot be represented exactly on a discrete grid. The reasonable compromise is reached that the generated skeleton must be essential to preserve the object's topology and to represent informatively the pattern's shape. Many skeletonization algorithms are available in the literature. Different algorithms produce slightly different skeletons. Rosenfeld *et. al.* [21,22] classified skeletonization algorithms as being parallel or sequential. The parallel algorithms, suited for parallel computers are usually simpler than the sequential ones. However, a sequential algorithm will be faster than a parallel algorithm if they are implemented on a serial computer.

A parallel skeletonization algorithm called ''*safe-point thinning algorithm*'' (SPTA) [23] where the safe-point testing is conducted by examining a set of Boolean expressions on eight neighbors of each edge-point to determine whether it is a safe or nonsafe point. The nonsafe points are then removed by a two-scan algorithm. A decision tree is constructed to minimize the number of neighbors to be examined. Since the algorithm is performed by scanning all object pixels at each iteration until no more nonsafe point exists, it is inefficient compared with our algorithm.

Some thinning algorithms are based on the contour generation method [24,25,26]. The essential idea is first to convert the input image into chain codes for each closed contour, and then to trace around the contour. If a boundary point is removed, the algorithm will generate new chain codes to replace the old one. The new chain codes are generated from two directional vectors, inward and outward, of each boundary point. All cases of two directional vectors are tabulated as a look-up table. The contour tracing will process each contour layer-by-layer iteratively until no further deletion is occurred. The algorithm performs better since it can reduce the running time to only the processing of boundary points.

CHAPTER 3

MID-CRACK CODES EXTRACTION

3.1 Introduction

In this chapter, a fast algorithm for extracting the mid-crack code in a single-pass raster-scan fashion and an extension to parallel implementation are presented. The developed algorithm needs only one phase to generate all the code sequences though a binary image is composed of several objects. Two types of boundaries, external and internal, are also considered, such that the inclusion relationship among region boundaries can be easily determined using the same algorithm.

The method in [4] is based on the mid-crack following scheme, which has the disadvantage that it requires access to the points of the image in an arbitrary order, since a border may be of any shape and size. Our method of constructing mid-crack codes of all the borders in a single row-by-row scan of the image, allows completely parallel implementation. Besides, the move-length in [4] is only one mid-crack at a time. Our algorithm, which can extract more than one mid-crack code determined from the codes linking step, is more efficient. In addition, our algorithm can deal with an image containing multiple objects with holes in still a single raster scan.

In Section 3.2.1, the look-up table and the move table for efficient accessibility are described. In Section 3.2.2, the mid-crack codes linking procedure is presented. In Section 3.2.3, the boundary type and inclusion relationship are discussed. In Section 3.2.4, the algorithm is given. In Section 3.2.5, the perimeter computation and area adjustment are included. In Section 3.3, experimental results are provided. In Section 3.4, a conclusion is made.

3.2 Methodology

3.2.1 Look-up Table and Move Table

A set of 3×3 window masks containing every variety of mid-crack codelinks initialized from the mid-cracks around a central pixel, is illustrated in Fig. 3. We could summarize this into five following types of encoding based on the number of codelinks: no-code, one-code, two-code, three-code and four-code links. Each codelink is associated with a headcode which is illustrated in Fig. 4. Then, we set up a look-up table shown in Table 1. The mid-crack codes are based on 8-connectedness and counter-clockwise tracing for the external boundary and clockwise tracing for the internal boundary. The index value is discussed next. The index value 0 reflecting a single pixel which is treated as a noise. The related information such as the total number of codelinks, all headcodes, and all codelinks can be obtained from drawing the mid-crack codes surrounding the central pixel, as illustrated in Fig. 3.

A 3×3 window, which is incorporated with different weights at each element exploring the presence of eight neighboring locations, is shown in Fig. 5a. If an object pixel occurs, the weighted window is convolved with the 3×3 neighborhood centered at that pixel. Assume that the binary image has the object pixel ''1'' and the background pixel ''0.'' This convolution is performed to calculate the index value of the look-up table. An example of the window operation is illustrated in Fig. 5b. The related information with respect to the mid-crack codes surrounding a pixel in Table 1 can be retrieved by the use of the index value. Then, a series of operations are applied to concatenate these individual codelinks to their suitable boundary links.

From a codelink, we can determine the relative move in column and row with respect to the current location. A move table listing the relative coordinates for all the moves, is shown in Table 2. For example, the code 0 indicates one-pixel move in x-axis

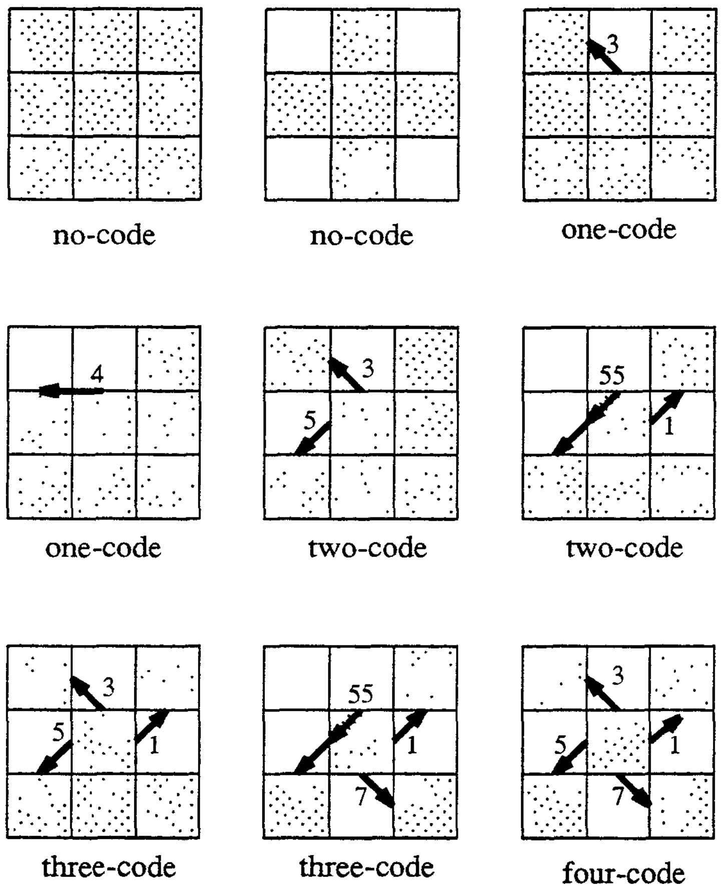
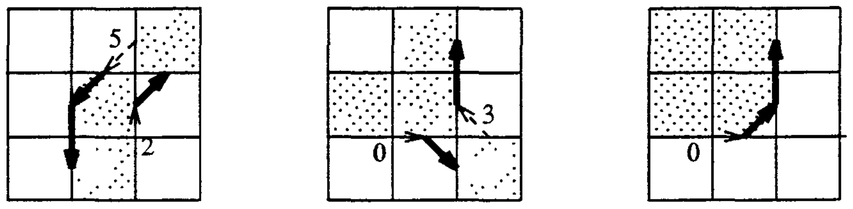


Fig. 3. Examples of five types of code-links initialized from the mid-cracks around the central pixel.

(or column), and no move in y-axis (or row). The destination of moves acts as an important role as we search for a right link in the linking head or tail step which will be discussed next.

Table 1. The Look-Up Table

Index	Number of Codelinks	Headcode	Codelink
0	0	Nil	Nil
1	1	7	3317
2	1	6	217
3	1	7	217
4	1	5	1175
5	2	5,0	117, 3
...	...	...	...
130	2	6.3	77, 2
131	2	7,3	77, 2
132	2	5,3	775, 1
133	3	7,5,3	77, 3, 1
134	2	6,3	77, 1
...	...	...	...
251	0	Nil	Nil
252	1	5	4
253	1	5	3
254	0	Nil	Nil
255	0	Nil	Nil



two-code : headcode = 5, 2

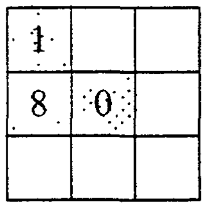
two-code : headcode = 0, 3

one-code : headcode = 0

Fig. 4. Headcode (dashed line) determination.

1	2	4
8	0	16
32	64	128

Weights in the Window
(a)



index = 1+8 = 9
(b)

Fig. 5. Index value calculation in the window operation.

3.2.2 Mid-Crack Codes Linking in the Raster Scan Algorithm

When the first object pixel is fetched, we quickly obtain its codelinks simply by a look-up table. After that, we create a new linked list in the connectedness structure array shown in Fig. 6. A list in the connectedness structure includes: head and tail coordinates, codelinks, thead, perimeter computation, and area adjustment. A list-typed data structure to store the temporary boundary links is employed in the representation. If the boundary link is connected with the codelink, we will not only increase the code sequence, but also change the coordinates of head and tail positions. Then in the process of scanning the next pixel, its codelinks are joined into the existing links by checking the conditions in the link-tail and link-head steps of the algorithm.

Table 2. The Move Table

Move	Relative Coordinates	
	Column(dx)	Row(dy)
0	+1	0
1	+1	-1
2	0	-1
3	-1	-1
4	-1	0
5	-1	+1
6	0	+1
7	+1	+1

If the current codes have connectivity with the neighboring codes, there are two kinds of concatenation ways: head concatenation and tail concatenation. In the head concatenation case, we connect the codelink with the specific boundary link in Step 6 of the algorithm presented in Section 5. In the tail concatenation case, we connect the specific boundary link in Step 7 of the algorithm with the codelink. In the link-head step, we check the coordinates of head and the first code of the searched boundary link in the connectedness structure array. In the link-tail step, we check the coordinates of tail and ''thead'' of the link. It is a condition that the thead can determine with which link to join suitably. If none of the links satisfies the condition, we create a new link in the connectedness structure array to store this code information. After the object pixels are scanned

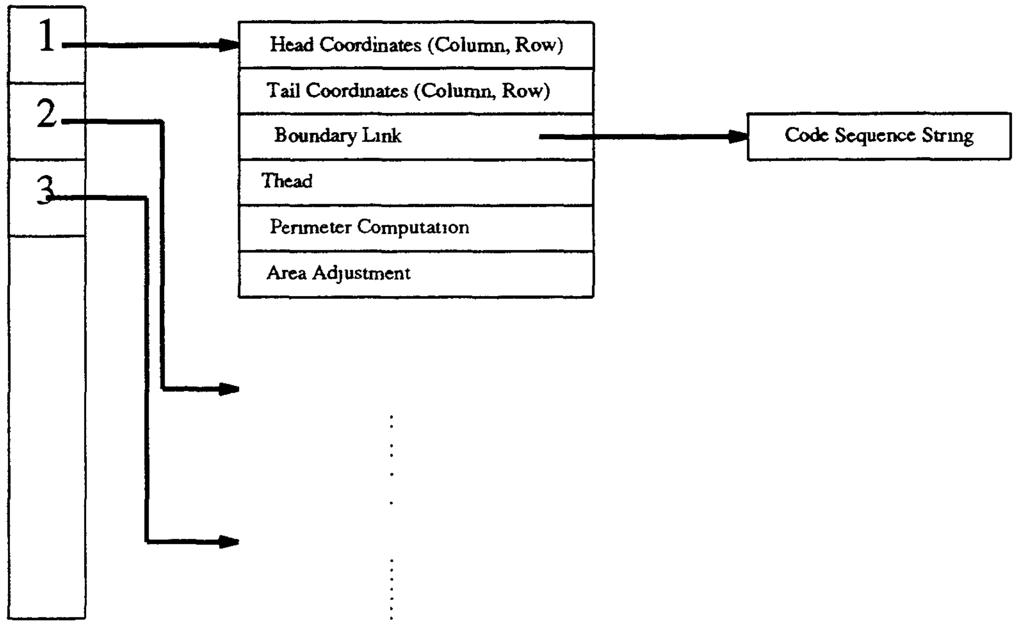


Fig. 6. The linked list in a connectedness structure array.

completely, each linked-list sequence in a block of the array represents the codes of an internal or external contour of an object. After the whole image is scanned, the boundary links of different objects exist in different blocks of the array. The external and internal boundary links can be differentiated by an additional step discussed in the next section.

3.2.3 Boundary Type and Inclusion Relationship

After all the mid-crack codelinks are extracted, we can easily determine the tracking direction. If a link is traced counter-clockwise, it is an external boundary; otherwise an internal boundary. A quick method to determine the tracking direction is proposed. A link tracking must be stopped at a pixel which is located in the lower-right corner. We can observe that only two cases with the first two codes "45" and "35," shown in Fig. 7, could happen in an internal boundary when the tracing is completed. The other cases belong to the external boundary.

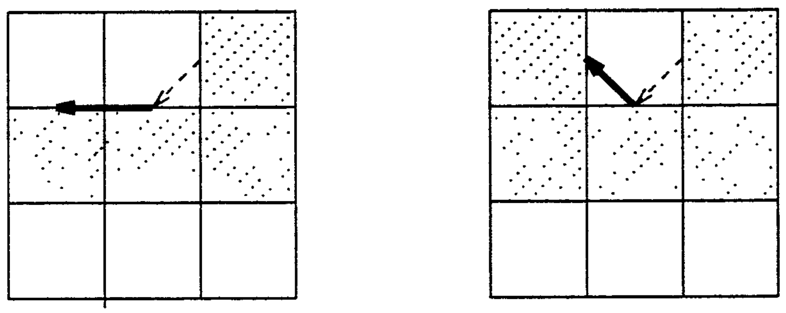
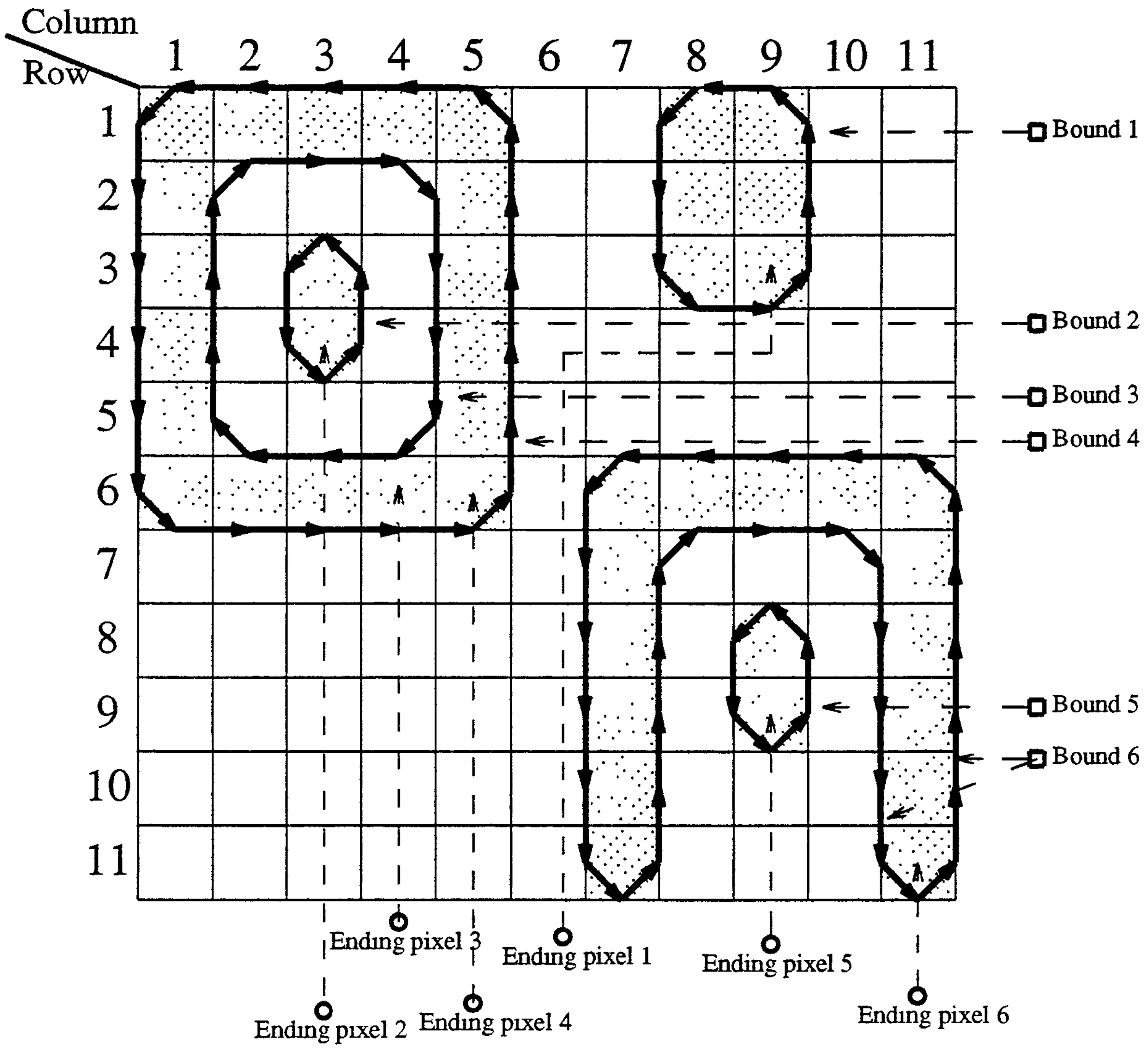


Fig. 7. The cases in the internal boundary.

A further analysis is needed if we want to know the inclusion relationship between boundaries. A five-object image with external and internal boundaries is shown in Fig. 8. We can find the ‘‘nearest link’’ in the connectedness structure array by comparing the difference between the (x,y)-coordinates of the ending pixel of the closed boundary link and the head and tail coordinates of the tested boundary link. The tested link with the least difference is the nearest link. For example, the link labeled Bound 2 is closed, its nearest link Bound 3 is found. Hence, the link number, Bound 2, is recorded into the field of Bound 3 in the connectedness structure array. Each time when the Bound 3 is joined into other links, the marked Bound 2 is transferred accordingly. After knowing the relationship of the different links, we can determine which object owns an internal link or even an external link. In Fig. 8, the ending pixel 2 is contained in an external boundary, Bound 2, and is bounded by an internal boundary, Bound 3. If Bound 3 is not an internal boundary, Bound 2 will not be bounded by it. For example, the ending pixel 5 is a component of Bound 5, but it is not bounded by Bound 6 since the boundary types of Bound 5 and 6 are the same as the external boundaries.



Ending Pixel	Cord(Col,Row)	Bounded by	Head(Col,Row)	Tail(Col,Row)
1	(9,3)	None	...	...
2	(3,4)	Bound 3	(5,4)	(1,4)
3	(4,6)	Bound 4	(4,6)	(5,5)
4	(5,6)	None	...	...
5	(9,9)	Bound 6	(11,9)	(7,9)
6	(11,11)	None	...	...

Fig. 8. A five-object image with external and internal boundaries.

3.2.4 The Algorithm

A system flow-chart is described in Fig. 9. Given a binary image of n columns and m rows, three two-dimensional arrays, named A[0][i], A[1][i] and A[2][i], are created to represent three rows, where $0 \le i \le n+1$. Since the 3×3 window operation is applied everywhere in an image including the image boundary, each array is extended to have two more elements at 0 and n+1. Let three array-pointers, named P_0, P_1 and P_2, point to the corresponding array: $P_0 \rightarrow$ A[0][0], $P_1 \rightarrow$ A[1][0] and $P_2 \rightarrow$ A[2][0]. Initially, let A[0][i] be all zeros and A[1][0] = A[2][0] = A[1][n+1] = A[2][n+1] = 0. The first two rows of the image are stored into A[1][k] and A[2][k], where $1 \le k \le n$. The following steps are performed:

1. Search for the object pixels in the middle row pointed by P_1. If the last pixel in this row is reached, go to the pointer adjustment in Step 10.

2. Accumulate the total number of object pixels. Get the pixel's coordinates, such that Col = i and Row = the current row in processing.

3. A 3×3 window shown in Fig. 5a is convolved with a 3×3 neighborhood centered at this pixel to compute the index value used in Table 1.

4. From Table 1, we extract the number of codelinks, denoted by ''c_num,'' and respond according to the following cases. In the zero-code case, go back to Step 1. In the one-code case, go to Step 5 once, and then back to Step 1. In the two-code case, go to Step 5 twice; similarly, three times in the three-code case and four times in the four-code case. After all codelinks are done, go back to Step 1.

5. In the linking step, first calculate the destination where the codelink goes to. Calculate HCol and HRow using eqs. (1) and (2), where dx and dy are retrieved in the move table shown in Table 2 corresponding to the first code of the codelink from Step 4.

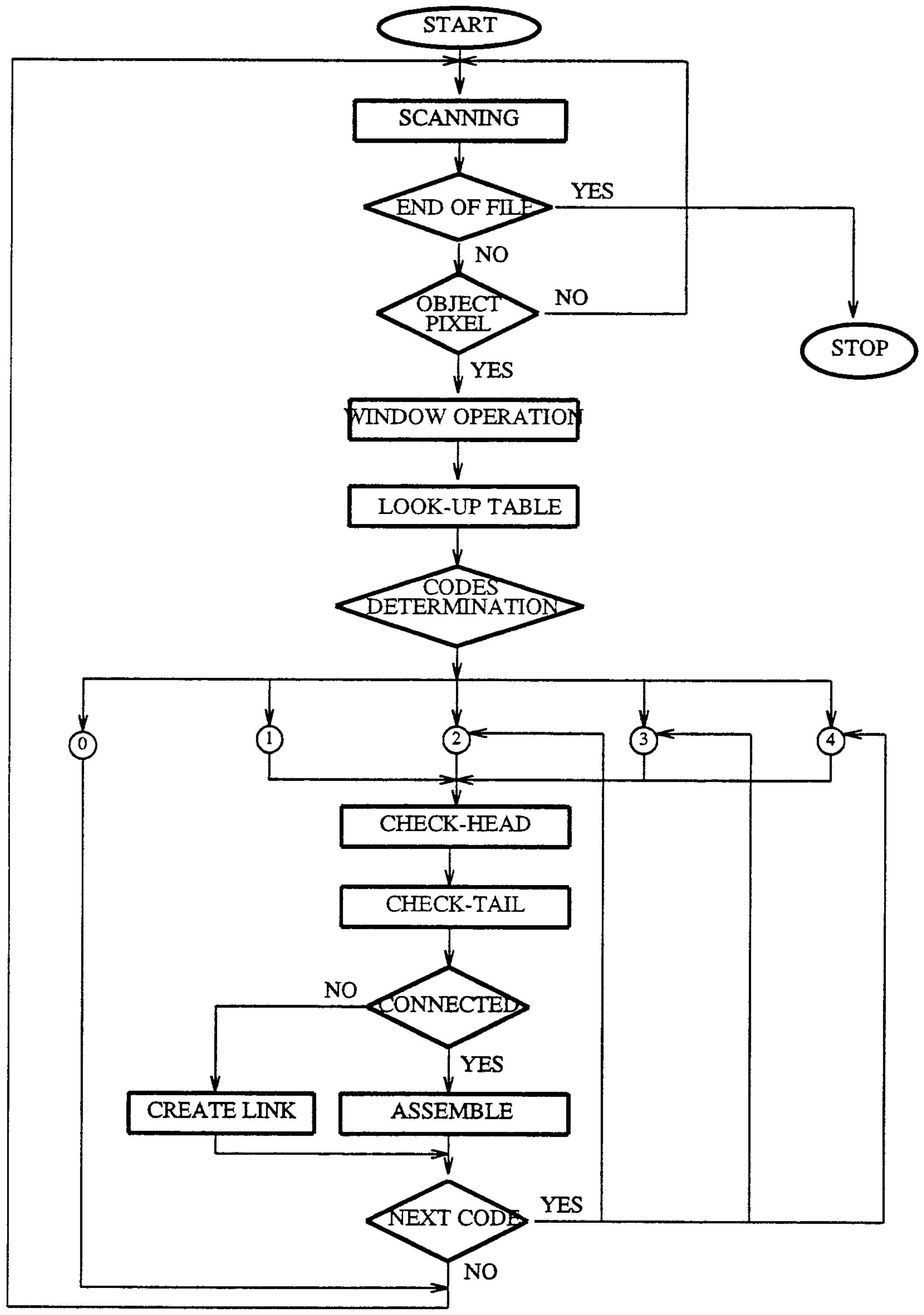


Fig. 9. The system flow-chart.

$$\text{HCol} = \text{Col} + dx, \tag{1}$$

$$\text{HRow} = \text{Row} + dy, \tag{2}$$

where

$$dx = \text{move}\,[\text{table}\,[\text{index}\,].\text{codelink}\,[q\,][0]].\text{column}, \tag{3}$$

$$dy = \text{move}\,[\text{table}\,[\text{index}\,].\text{codelink}\,[q\,][0]].\text{row}, \tag{4}$$

$$\text{headcode} = \text{table}\,[\text{index}\,].\text{headcode}\,[q\,], \tag{5}$$

$$\text{tailcode} = \text{table}\,[\text{index}\,].\text{codelink}\,[q\,][0], \tag{6}$$

where q is the order number of codelinks and $0 \le q \le \text{c\_num} - 1$. The codelink[$q$][0] is the first code of the qth codelink. In Fig. 8, the pixel (3,4) can be described as: index = 2, c_num = 1, and q = 0. From Table 1 when q = 0, table[2].codelink[0][0] = 2. Therefore, dx = move[2].column = 0 and dy = move[2].row = -1.

Go to the check-head step 6 and the check-tail step 7, and get the return values of the connected link number for the link-head and link-tail, respectively. The link-head and link-tail are the link number in the connectedness structure array. If the head and tail both are not connected, go to the create-link step 8. If only one of the head and tail is connected, go to the concatenation step 9 accompanied with an indicating symbol ''HEAD'' or ''TAIL'' correspondingly. If both head and tail are connected, there are two cases. 1) If link-head is not equal to link-tail, go to the concatenation step 9 twice with the symbol ''TAIL.'' 2) If link-head is equal to link-tail, go to the concatenation step once with the symbol ''HEAD.'' After Steps 8 and 9 are done, go back to Step 4.

6. In the check-head step, we check two conditions. First condition: Col and Row are equal to the head coordinates of the searched boundary link in the connectedness structure array. Second condition: the headcode is equal to the first code of the boundary link. If both conditions are satisfied, we call the head of this codelink connected; otherwise disconnected. Return the connected link number to Step 5.
7. In the check-tail step, we also check two conditions. First condition: HCol and HRow are equal to the tail coordinates of the searched boundary link in the connectedness structure array. Second condition: the tailcode is equal to the thead of the boundary link. If both conditions are satisfied, we called the tail of this codelink connected; otherwise disconnected. Return the connected link number to Step 5.
8. In the create-link step, we search for the empty link, which is joined into another link and will not be used any more in the connectedness structure array. If there is no empty link available, a new link is created. Then we store the information including head and tail coordinates, codelink, and thead into such a link. Return to Step 5.
9. In the concatenation step, we have an indicating symbol, HEAD or TAIL, to determine the control flow. If the symbol is HEAD, we replace x- and y-coordinates of the headcode by HCol and HRow computed by eqs. (1) and (2). Then, we concatenate the codelink with the code sequence of the specified link in Step 6. If the symbol is TAIL, we replace x- and y-coordinates of the tailcode and thead by the Col, Row and headcode computed in Step 2 and eq. (5), respectively. Then we concatenate the code sequence of the specified link in Step 7 to the codelink. A little difference is seen in the case of both the head and tail being connected. If the link-head is equal to the link-tail, then it means that the boundary link is closed by this codelink. In this case, no matter how to concatenate them using the case ''HEAD'' or ''TAIL,'' the resulting codes are the same. In our experiment, we concatenate them using the

case of HEAD. Otherwise, we perform the following two steps. First step: we concatenate the codelink to the code sequence of the link-tail. Second step: we concatenate the code sequence of link-head to the code sequence of link-tail, replace tail coordinates and thead in the link-tail by tail coordinates and thead in the link-head, and set the head coordinates to a negative value indicating an empty link which can be reused in Step 8. Return to Step 5.

10. Up to now, We have completed one row of the input image. The array pointers are then adjusted as: $P_0 \rightarrow$ A[j mod 3][0], $P_1 \rightarrow$ A[(j+1) mod 3][0], and $P_2 \rightarrow$ A[(j+2) mod 3][0], where j is initialized as 0 and increased by 1 in each iteration. For example in the second iteration, $P_0 \rightarrow$ A[1][0], $P_1 \rightarrow$ A[2][0], and $P_2 \rightarrow$A[0][0]. The next row of the input image is then stored into the array A[(j+2) mod 3][k]. If $j < m$, the image is not yet scanned completely and must go back to Step 1; otherwise, the scanning is over.

3.2.5 Perimeter Computation and Area Adjustment

The look-up table can be added two more items to deal with the perimeter computation and area adjustment which are obtained simultaneously during the execution of mid-crack codes. We know the moving length of the codes 1, 3, 5 and 7 is $\sqrt{2}/2$ and of 0, 2, 4 and 6 is 1. For the simplicity of computation, the perimeter of an object region is estimated by accumulating all the moving lengths of the codes. For example for an object pixel with the index value 5 in the look-up table, it has two codelinks "117" and "3." The perimeter of the first codelink is $3 \times \sqrt{2}/2$ and of the second codelink is $\sqrt{2}/2$.

Therefore, we can add this information to the look-up table and extend the connectedness structure array with a perimeter item. If the connectedness of the codelink occurs, we can accumulate the perimeter value in the concatenation step.

The area of an object region is simply the number of points in the region. In Step 2 of the algorithm, the total number of object pixels is computed. However, as we mentioned previously, the enclosed area of mid-crack codes possesses more accurate value. Therefore the area based on counting the number of pixels needs some adjustment. A few examples of the area adjustment are illustrated in Fig. 10. The vertical moves, i.e., the codes 0, 2, 4 and 6, do not need the adjustment since they move along the cracks in between two pixels. For diagonal moves, i.e., the codes 1, 3, 5 and 7, the area enclosed is different from the digitized image area by $\pm 1/8$ of a unit-pixel square. All the cases enumerated with their adjustments are given in Table 3. If an image contains only external boundaries, the area adjustment can be achieved in parallel at the same time of the algorithm execution. If any internal boundary is involved, the area adjustment requires the knowledge of boundary types which can be determined by using the procedure presented in Section 4, after the codes of a boundary are entirely obtained.

3.3 Experimental Results

Two results are shown. They are considered using 8-connectedness for the object regions. The program was implemented in C on a Sun Sparc workstation. The CPU execution times for both results are less than 100 ms. In Fig. 11, we are concerned with an image containing an object with a hole. The result is shown as follows:

The total number of pixels = 29.

(1) Starting pixel = (4,6).

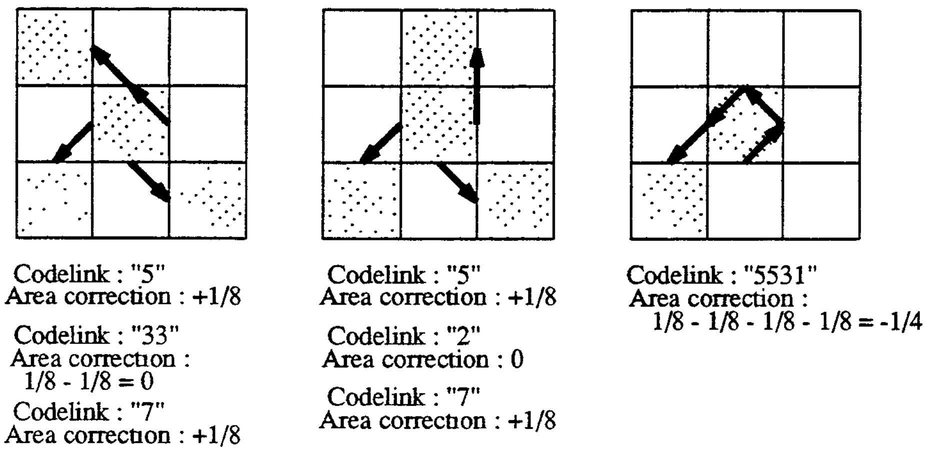


Fig. 10. A few examples of the area adjustment.

Table 3. All the cases enumerated with their area adjustments.

Area Correction of Moves in Counter-clockwise Tracing								
Move	0	1	2	3	4	5	6	7
Cross Object		-1/8		-1/8		-1/8		-1/8
Cross Background		+1/8		+1/8		+1/8		+1/8
Along Cracks	0		0		0		0	

Codes: 3571

Perimeter = 5.66, Area = 0.5, Boundary type: Internal

(2) Starting pixel = (8,8).

Codes:

33177117667121767111210765455555777111175555777753333553113335557753355553111133333177771113333333

Perimeter = 70.26, Area = 29, Boundary type: External

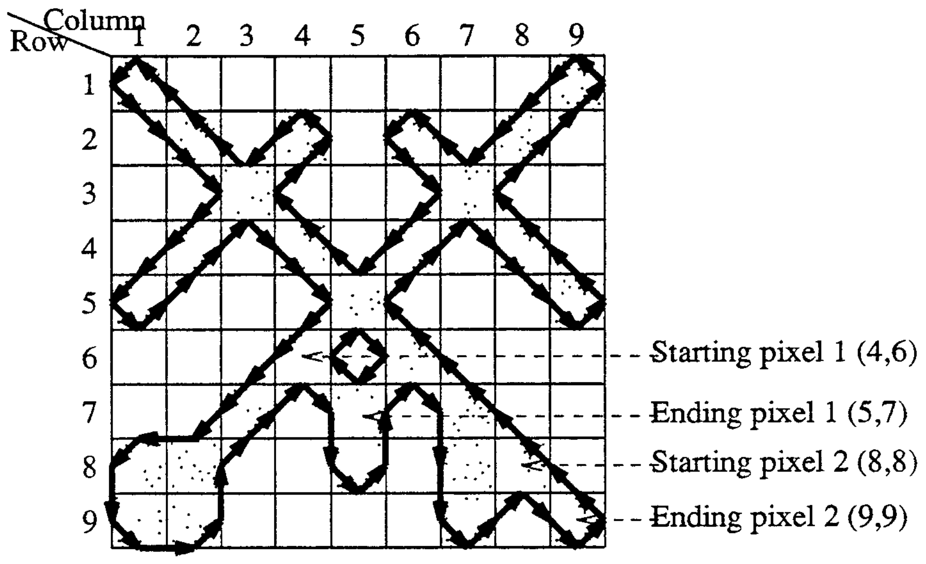


Fig. 11. An image contains an object with a hole.

In Fig. 8, we are concerned with an image with multiple objects. The result is shown as follows:

The total number of pixels = 43.

(1) Starting pixel = (9,2).

Codes: 2107665432

Perimeter = 8.83, Area = 5.5, Boundary type: External

(2) Starting pixel = (3,3).

Codes: 217653

Perimeter = 4.83, Area = 1.5, Boundary type: External

(3) Starting pixel = (3,6).

Codes: 45666700122234

Perimeter = 12.83, Area = 11.5, Boundary type: Internal

(4) Starting pixel = (5,5).

Codes: 2100007666665444432222

Perimeter = 20.83, Area = 18, Boundary type: External

(5) Starting pixel = (9,8).

Codes: 217653

Perimeter = 4.83, Area = 1.5, Boundary type: External

(6) Starting pixel = (11,10).

Codes: 21766667001222217666665444432222

Perimeter = 25.66, Area = 14.5, Boundary type: External

3.4 Conclusions

The paper describes an efficient algorithm for encoding a complex image into mid-crack codes, which is a more precise method based on the subpixel measurement capabilities. The algorithm only requires a single row-by-row scan of the image and uses a 3×3 window for the codelinks generation. Simultaneously, all the connected links of line segments are encoded and joined together into a data structure. Also, the algorithm has an advantage in detecting the inclusion relationship between boundaries. Besides, the perimeter computation and area adjustment can be performed in parallel.

The algorithm is nearly memoryless, since it scans only three rows at a time and does not store the entire image. If there are multiple objects in an image, such links exist in the array after the encoding and linking processes. Therefore, the algorithm can extract all mid-crack codes in a single scan. Our method is based on tables which are suited for all various cases in an image. If the 4-connectedness for the foreground is dealt with instead of 8-connectedness, the only change is to slightly adjust the look-up table. The applications of the mid-crack code description are still an ongoing research.

CHAPTER 4

RESTORATION USING MID-CRACK CODE DESCRIPTION

4.1 Introduction

In computer vision, image processing, and object recognition and inspection, the precision requirement of geometric properties is very important. By applying our region-filling algorithm based on the mid-crack code representation, we convert the codes into y-partition data structure from which the run-length codes and the formulas of geometric features can be easily derived [3,16]. Our algorithm is significantly different from the method proposed by Merrill [18] which exists three restrictions, such as a) the boundary must be closed, b) it cannot handle the cases where the test line intersects the boundary tangentially, and c) the closed boundary cannot loop back on itself. However, our midcrack-code based algorithm does not have these restrictions. This chapter is organized as follows. In Section 4.2.1, the mid-crack code description rules are introduced. In Section 4.2.2, the region-filling algorithm developed is described. In Section 4.2.3, the algorithm applied to gray-scale images with multiple regions is presented. In Section 4.3, experimental results are illustrated. In Section 4.4, some conclusions are made.

4.2 Methodology

4.2.1 Mid-Crack Code Description Rules

In the contour description, the border points are represented in the form of a string of eight moving-directional codes. The mid-crack code description can be seen as movement along the mid-points of border cracks which produce codes of links. The codes from 0 to 7 are assigned to indicate the eight moving directions of 0° to 315° in intervals of 45°, as shown in Fig. 2. For the horizontal or vertical move, the length of a move is 1, and for the diagonal move, it is $\sqrt{2}/2$. If the crack is located in between two adjacent pixels in the vertical direction, it is said to be on a *vertical crack*. Similarly, if the crack is located in between two adjacent pixels in the horizontal direction, it is said to be on a *horizontal crack*. There are two restrictions on the moves in the mid-crack code as illustrated in Fig. 2. If the move is from a vertical crack, the codes 0 and 4 are not allowed. Similarly, the codes 2 and 6 are not allowed in the moves from a horizontal crack.

We use the tracking rule of 8-connectedness counterclockwise. Let us assume that the description form of all m contours in a binary image is expressed as follows:

$$\begin{array}{c}
s x_{00} y_{00} d_{00} d_{01} \cdots\cdots\cdots\cdots d_{0n_0} e \\
s x_{10} y_{10} d_{10} d_{11} \cdots\cdots\cdots\cdots d_{1n_1} e \\
\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
\cdots\cdots\cdots\cdots d_{i(j-1)} d_{ij} \cdots\cdots \\
\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
s x_{m0} y_{m0} d_{m0} d_{m1} \cdots\cdots\cdots\cdots d_{mn_m} e\#
\end{array}$$

where "#" is the end symbol of the description; "s" and "e" are the starting and ending symbols for each contour, respectively; "x" and "y" are the coordinates of the starting point; "d" is the directional code. The subscript of "d_{ij}" indicates that d is the code in the ith contour and the jth move. Each contour could be either an in-contour or an out-contour. In mid-crack code description, since the starting and ending pixels in each contour must be adjacent to each other, the opened curve occurred in the chain-code description should not exist here. Besides, there are four cracks surrounding a pixel. Therefore the problems in the chain code description of no-code for an isolated point and redundantly tracing a same pixel twice for a multi-way junction pixel will not again happen here.

A simple example of the mid-crack code description for a single-object image is illustrated in Fig. 1. The resulting mid-crack codes are:

s (6,5)11771177076545677533554333101343 21*e#*

Referring to Fig. 1, the starting pixel (e.g. (6,5)) is defined as the pixel which is pointed by the link of the first mid-crack code located at the ending pixel (e.g. (5,6)) in a raster scan. As mentioned above, the counterclockwise tracing indicated by the arrow directions is adopted. Since the mid-crack codes are generated based on pixel-by-pixel according to a look-up table and then are concatenated using the code-linking from the head to the tail of arrows, it is in nature that the generated sequences of the mid-crack code description are in a clockwise direction.

3.2.1 Region-filling algorithm of binary image

In the mid-crack code extraction algorithm [15], an object pixel may be visited at its four cracks for none, once, twice, three times, or four times, as illustrated in Fig. 12. If an object pixel is not visited on its vertical crack (e.g. the case in Fig. 12a), it must be completely surrounded in the horizontal direction by other object pixels. Hence, in our region-filling algorithm the location of this pixel will not be stored. If an object pixel is visited once on its vertical crack (e.g. the case in Fig. 12b), it will be either a starting border or an ending border in the horizontal direction. Hence, the location of this pixel will be stored once. If an object pixel is visited twice on its vertical crack (e.g., the cases in Figs. 12c, 12d, and 12e), it must be one of the cases, such as a singular point, an isolated point, or a multi-way junction point. No matter what case, the object pixel is always a starting border as well as an ending border in the horizontal direction. Hence, the

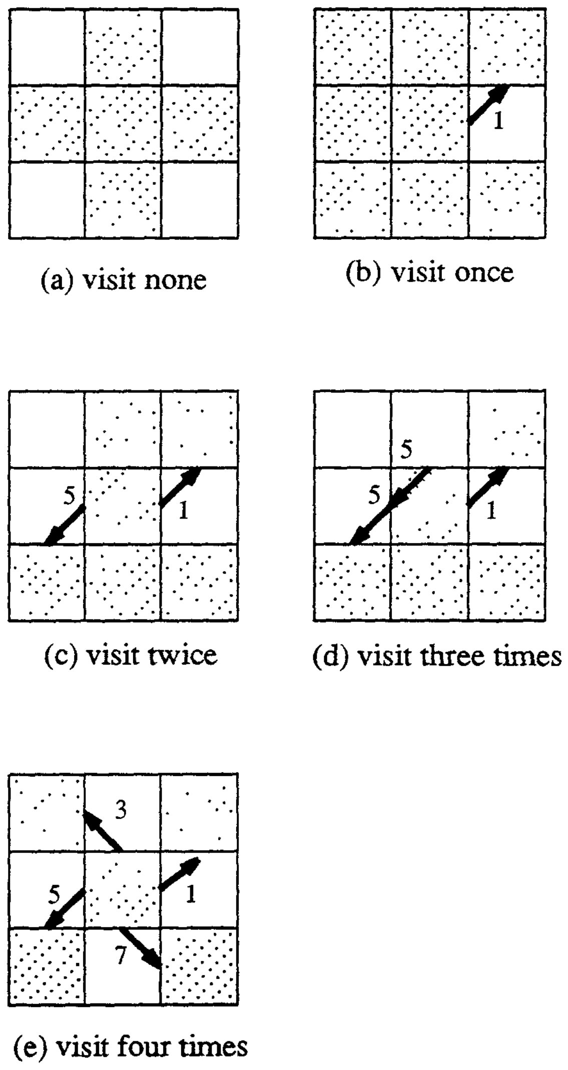


Fig. 12.The cases of an object pixel being visited none, once, twice, three times, or four times.

location of this pixel will be stored twice. By using this method, it is easy to extract the starting border and the ending border in each horizontal direction in order to reconstruct all the object pixels in an external boundary or an internal boundary.

There are two kinds of tracking directions: forward and backward, in the mid-crack code description. Referring to Fig. 1, since the counterclockwise rule is used, the forward and backward tracking is defined as the same direction or the reverse direction of the contour tracking, respectively. In other words, the forward tracking approach visits the mid-crack codes beginning with the last code (tail of link) of the code sequence, and the backward tracking approach starts with the first code (head of link) of the code sequence. In our region-filling algorithm, the forward tracking is chosen in order to maintain consistency with the contour tracing direction.

In the mid-crack code description rules, the starting pixel is the pixel which is pointed by the link of the first mid-crack code located at the ending pixel in the row-by-row scan. We can easily observe that for an ending pixel in an external boundary must be the form of

$$\begin{matrix} \mathbf{x} & \mathbf{x} & \mathbf{x} \\ \mathbf{x} & 1 & 0 \\ 0 & 0 & 0 \end{matrix}$$

where **x** denotes "*don't care*". Hence there are totally $2^4 = 16$ cases. If all **x**'s are zeroes, the pixel is an isolated point; its mid-crack codes will be obtained immediately as "1357." For an ending pixel in an internal boundary, its 3×3 neighborhood must be the form of

$$\begin{array}{ccc} 1 & 0 & 1 \\ \mathbf{x} & 1 & \mathbf{x} \\ \mathbf{x} & \mathbf{x} & \mathbf{x} \end{array}$$

or

$$\begin{array}{ccc} 0 & 0 & 1 \\ 1 & 1 & \mathbf{x} \\ \mathbf{x} & \mathbf{x} & \mathbf{x} \end{array}$$

Hence, there are totally $2^5 + 2^4 = 48$ cases. However, in Fig. 13 it is shown that the first code in the mid-crack code description must be (a) 1, 2, 3, or 4 in an external boundary, or (b) 3 or 4 in an internal boundary. The adjustment for a starting pixel from the mid-crack coordinate system to the chain-code coordinate system will depend on the first code, as described next.

Because the mid-crack code visits the mid-point of a crack, a unit-length in the mid-crack code coordinate system is designed as equating a half-pixel in the normally-used chain code coordinates, as shown in Fig. 14. The coordinate system used in extracting the mid-crack codes is the chain code coordinate [15]. Hence, the (x,y) values of the input mid-crack codes must be converted into new values in the mid-crack code coordinate system by an adjustment table shown in Table 4. From a move sequence of the mid-crack code, we can establish a move table to evaluate the location of the next code. Accordingly, the move table in [15] also needs to be modified to fit in the mid-crack code coordinate system, as shown in Table 5. When the location of a move is on the vertical mid-crack (i.e., y is not an integer), it must be the starting or ending mid-crack surrounding a horizontal region. Hence, its x- and y-coordinates are recorded by converting back to the chain code coordinate system for the purpose of region filling. The reversed conversion is performed by truncating y into an integer and adjusting x according to Table 6. The procedure is repeated until the last code is reached. The algorithm is

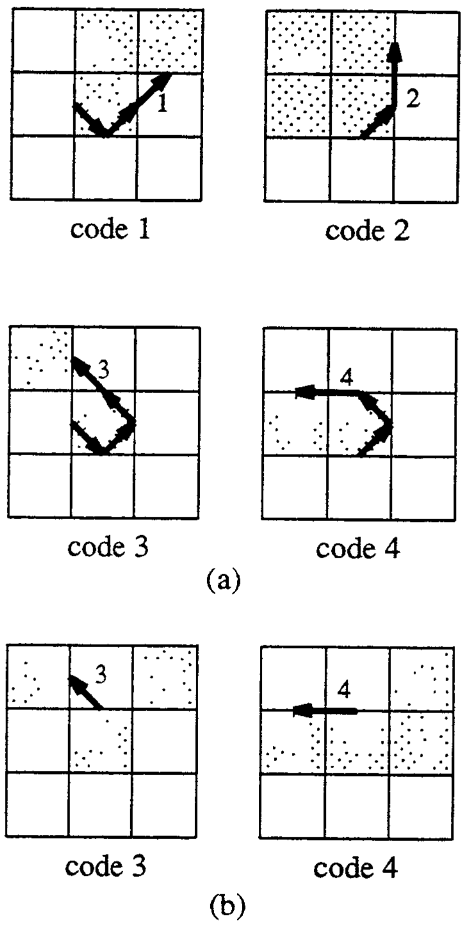


Fig. 13.The cases illustrate that the first number of the mid-crack code description must be (a) 1, 2, 3 or 4 in an external boundary, or (b) 3 or 4 in an internal boundary.

quite efficient such that only three simple look-up tables are used. After all the codes of a contour are processed, we can check the coordinates of the last point location. If it is equal to the starting method, our algorithm is described as follows:

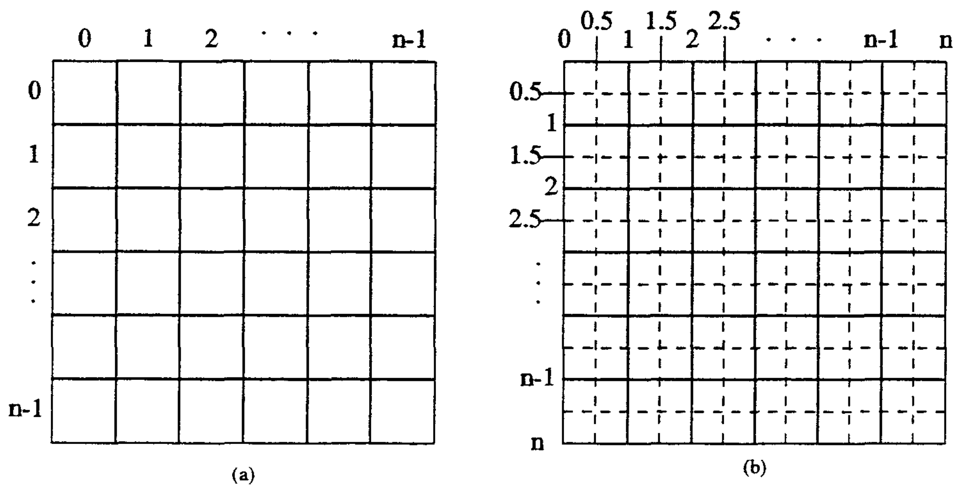


Fig. 14. The coordinate systems of (a) the chain code. (b) the mid-crack code.

Table 4. The Starting-Point Adjustment Table

First code	X Adjustment	Y Adjustment
1	+0.5	+1
2	+1	+0.5
3	+1	+0.5
4	+0.5	0

1. Read the starting pixel (x_{i0}, y_{i0}).

2. Read the first direction code d_{i0}. Adjust the starting point (x_{i0}, y_{i0}) in the chain code coordinates to be (X_0, Y_0) in the mid-crack code coordinates according to d_{i0} and Table 4.

Table 5. The Move Table

Code	X Adjustment	Y Adjustment
0	+1	0
1	+0.5	-0.5
2	0	-1
3	-0.5	-0.5
4	-1	0
5	-0.5	+0.5
6	0	1
7	+0.5	+0.5

3. Read the next direction code D_i (beginning from the last code of the code sequence). Evaluate the (X, Y) coordinates of the next location according to X_0, Y_0, D_i, and Table 5.
4. If the value of Y is not an integer, record (X, Y) by truncating Y into an integer and adjusting X according to Table 6.
5. Let $X=X_0$, $Y=Y_0$.
6. Do Step 3 and obtain the next location until the first code is reached.
7. Check whether the location of the last visited point is the same as the starting point in Step 2. If it is the same, the contour is correct; otherwise, it is incorrect.

Table 6. The *X*-Coordinate Conversion Table

Code	*X* Adjustment
1	-1
2	-1
3	-1
5	0
6	0
7	0

8. Repeat Steps 1 through 7 with another contour until the ending symbol ''#'' is reached.

4.3 Experimental Results

Fig. 15 illustrates an image containing eight contours. Their starting and ending pixel coordinates are shown in Table 7. Their mid-crack codes description is obtained as follows:

Contour 1: *s*(9,3)21767755533311*e*

Contour 2: *s*(4,6)4567001234*e*

Contour 3: *s*(12,5)356712*e*

Contour 4: *s*(6,8)457013*e*

Contour 5: *s*(2,10)4566701223*e*

Contour 6: *s*(5,9)110077766655677557754443333323333222107711*e*

Contour 7: *s*(10,9)3565570122*e*

Contour 8: *s*(12,9)2107776665456555432221121l*e#*

As an example of the contour 6, behind the starting symbol ''s'' the ''(5,9)'' is the coordinate of the ending pixel, and that ''1'', ''1'', ''0'', etc., are the directional codes. We present the result of applying our algorithm to the above contour 6 in Table 8. We can convert all the results into the following data structure of list-record:

Contour 1:

(1;9,9)

(2;8,10)

(3;9,9)

(4;9,9)

Contour 2:

(4;2,6)

(5;2,6)

Contour 3:

(4;12,14)

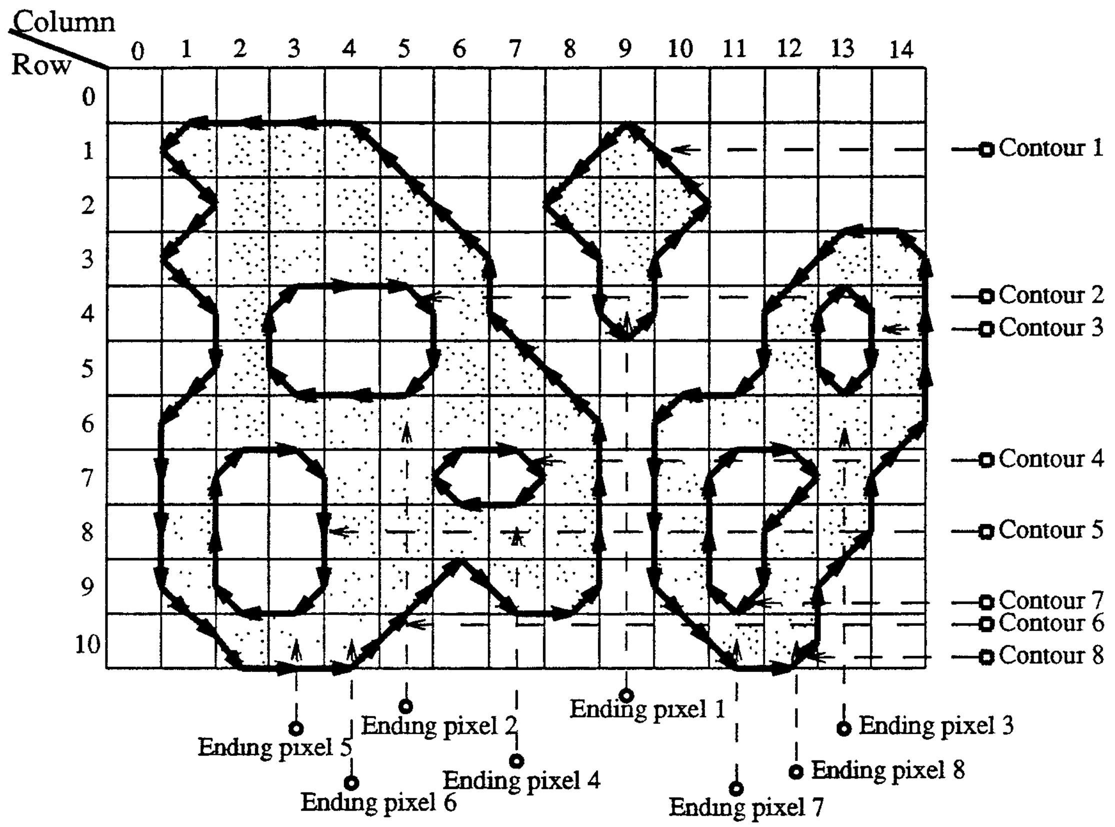


Fig. 15. An image of eight contours with their mid-crack code description.

(5;12,14)

Contour 4:

(7;5,8)

Contour 5:

(7;1,4)

(8;1,4)

(9;1,4)

Table 7. The Starting and Ending Pixels in Fig. 15

Contour	Starting Pixel(Col,Row)	Bounded by	Ending Pixel(Col,Row)
1	(9,3)	None	(9,4)
2	(4,6)	Bound 6	(5,6)
3	(12,5)	Bound 8	(13,6)
4	(6,8)	Bound 6	(7,8)
5	(2,10)	Bound 6	(3,10)
6	(5,9)	None	(4,10)
7	(10,9)	Bound 8	(11,10)
8	(12,9)	None	(12,10)

Contour 6:

(1;1,4)

(2;2,5)

(3;1,6)

(4;2,6)

(5;2,7)

(6;1,8)

(7;1,8)

(8;1,8)

(9;1,5,7,8)

(10;2,4)

Contour 7:

(7;10,13)

(8;10,12)

(9;10,12)

Contour 8:

(3;13,14)

(4;12,14)

(5;12,14)

(6;10,14)

(7;10,13)

(8;10,13)

(9;10,12)

(10;11,12)

where the first number is the *y*-coordinate, and then the *x*-coordinate of the starting and ending pixels, and so forth. The combined list-record of all eight contours is described in Table 9. According to this table, we can fill the pixels into the region. The precise number for the area of three objects bounded by a mid-crack code description in Fig. 15 is 46, 5½, and 20½. The area of reconstruction using our algorithm is just to accumulate the number of object pixels which is 45, 6, and 20, respectively.

Table 8. The Result of Applying Algorithm on Contour 6 of Fig. 15

Code in seq.	Coordinate	Recorded Coord.	Code in seq.	Coordinate	Recorded Coord.
Starting location	(5.5, 10)	...	4	(1.5, 1)	...
1	(6, 9.5)	(5, 9)	5	(1, 1.5)	(1, 1)
1	(6, 9)	...	7	(1.5, 2)	...
7	(7, 9.5)	(7, 9)	7	(2, 2.5)	(1, 2)
7	(7.5, 10)	...	5	(1.5, 3)	...
0	(8.5, 10)	...	5	(1, 3.5)	(1, 3)
1	(9, 9.5)	(8, 9)	7	(1.5, 4)	...
2	(9, 8.5)	(8, 8)	7	(2, 4.5)	(2, 4)
2	(9, 7.5)	(8, 7)	6	(2, 5.5)	(2, 5)
2	(9, 6.5)	(8, 6)	5	(1.5, 6)	...
3	(8.5, 6)	...	5	(1, 6.5)	(1. 6)
3	(8, 5.5)	(7, 5)	6	(1, 7.5)	(1, 7)
3	(7.5, 5)	...	6	(1, 8.5)	(1, 8)
3	(7, 4.5)	(6, 4)	6	(1, 9.5)	(1, 9)
2	(7, 3.5)	(6, 3)	7	(1.5, 10)	...
3	(6.5, 3)	...	7	(2, 10.5)	(2, 10)
3	(6, 2.5)	(5, 2)	7	(2.5, 11)	...
3	(5.5, 2)	...	0	(3.5, 11)	...
3	(5, 1.5)	(4, 1)	0	(4.5, 11)	...
3	(4.5, 1)	...	1	(5, 10.5)	(4, 10)
4	(3.5, 1)	...	1	(5.5, 10)	...
4	(2.5, 1)	...	End symbol		

Table 9. The Combined List-Record of All Contours in Fig. 15

Y(Row)	X(Col)
1	1, 4, 9, 9
2	2, 5, 8, 10
3	1, 6, 9, 9, 13, 14
4	2, 2, 6, 6, 9, 9, 12, 12, 14, 14
5	2, 2, 6, 7, 12, 12, 14, 14
6	1, 8, 10, 14
7	1, 1, 4, 5, 8, 8, 10, 10, 13, 13
8	1, 1, 4 ,8, 10, 10, 12, 13
9	1, 1, 4, 5, 7, 8, 10, 10, 12, 12
10	2, 4, 11, 12

4.4 Multi-region Filling Approach

In the previous sections, we only discuss the restoration of a binary image with multiple objects. Now, the same algorithm can be applied to a gray-scale (or color) image with multiple regions. If a picture contains more than two types of regions, it may still be possible to segment it by applying several thresholds. For example, in pictures of white blood cells the nucleus is generally darker than the cytoplasm, which is in turn darker than the background. A gray-level term is added to a contour mid-crack code description to represent the gray value of the region bounded by its contour. For example:

s (50)(6,5)117711770765456775335543331013432 1*e*

The number "50" following the "*s*" denotes the gray level of the region.

The restoration of the whole image by filling each region with its corresponding gray-value (or color) from its contour description is straightforward by applying the same proposed algorithm. A critical problem is how to extract those contours in a very efficient way. In an ordinary method multiple regions with different gray-levels are extracted from dealing with one region at a time after thresholding. Therefore, the extraction routine is repeated again and again. Referring to the algorithm for mid-crack codes extraction [15], if the boundary between two regions shown in Fig. 16, is considered, we can assume that each region has its own boundary by viewing each region as an independent object as shown in Fig. 17. Based on the assumption, we observe that two contours may coexist in one boundary if two regions are neighboring to each other. The mid-crack codes of Fig. 17 are shown in Table 10.

In Fig. 17 if a region contains no interior subregion, it produces an out-contour in the mid-crack code description. If a region contains k regions inside, then it will need $k + 1$ out-contours and k in-contours to represent the complicated region. Referring to the masking step in the mid-crack code extraction algorithm [15], we process the 3×3 neighborhood using the following criterion: the neighboring pixels having the same gray value as the central pixel are assigned to be 1, otherwise 0, and then an index value is generated to be used in the look-up table. In the create-link step, we assign a new gray value to differentiate the created link from the links of other regions having the different gray level in the linking step.

The mid-crack coding algorithm [15] can be used to obtain all the contours in an

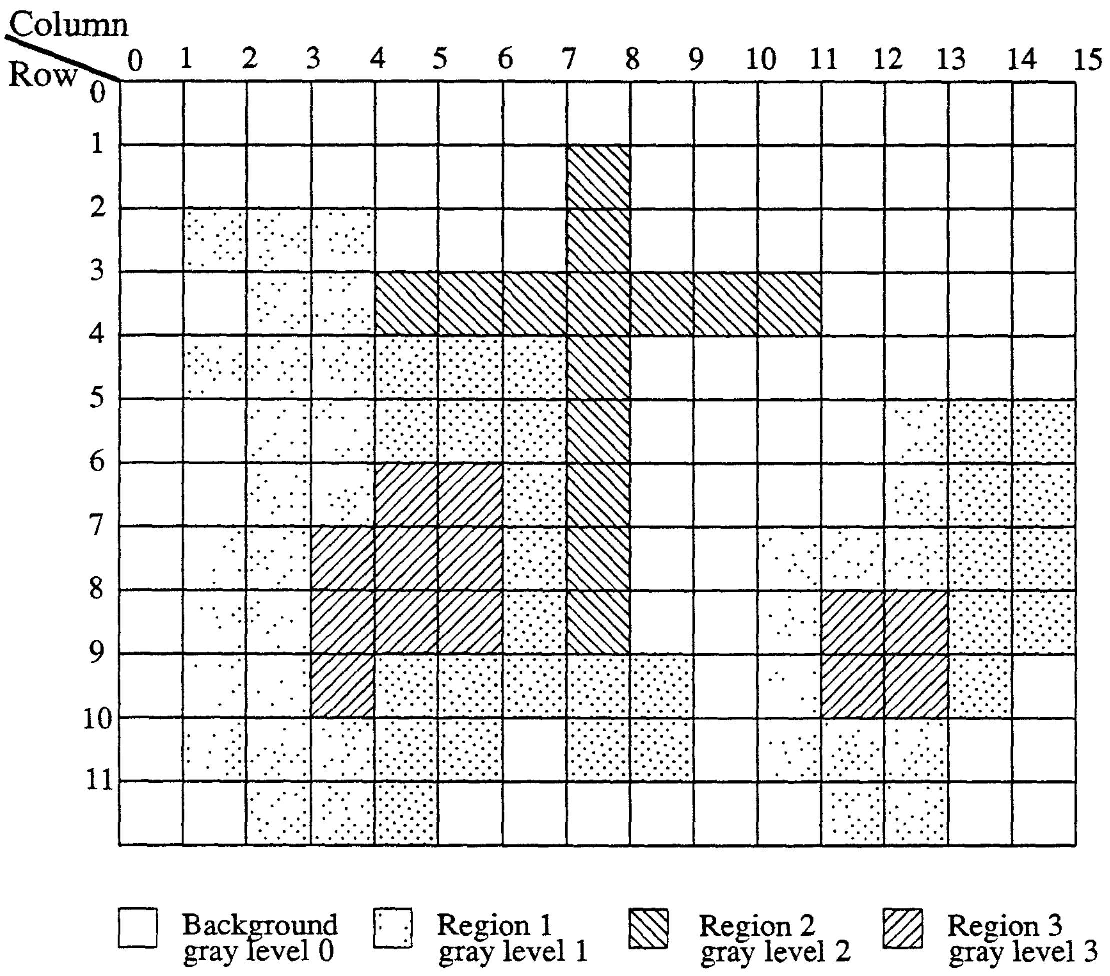


Fig. 16. A gray-scale image with multiple regions.

image. Each contour is encoded with its gray level. According to our region-filling algorithm, the result is shown in Table 11. Our region-filling algorithm is efficient in speed since it applies the mid-crack coding which only requires a single-pass raster scan and utilizes look-up tables.

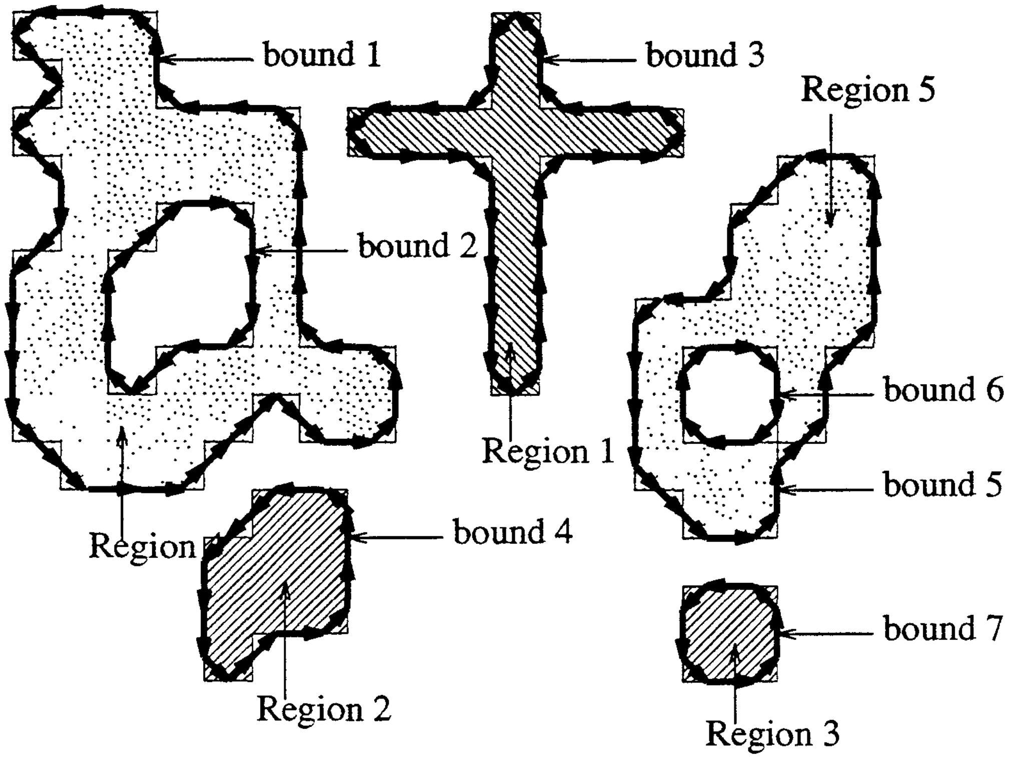


Fig. 17. The mid-crack contours of Fig. 16.

4.5 Conclusion

In the mid-crack code contour description, we do not need to take care of any particular pixel, such as a singular point, an isolated point, a multi-way junction point, etc. Instead of considering the characteristics of the input and output codes of a point, we are concerned with the coordinate of the cracks in the mid-crack code description. Since multi-way junction points are easy to process in the mid-crack code, no additional operations are used to solve this kind of problem, i.e., the chaining process passes through such a point more than once. Our method also can convert the mid-crack code description into the y-axis partition which is a region representation. Therefore, it is much easier to derive the geometric properties for an object. The developed algorithm can be also applied to gray-scale images with multiple regions very efficiently.

Table 10. The Mid-Crack Codes of Fig. 16

Bound	Code Sequence
1	s(g_1)(5,10)1100777666556775577544323443222234321077711e
2	s(g_1)(2,9)35545667011122e
3	s(g_2)(7,7)21766667007544565323443100122e
4	s(g_3)(4,8)11766555432210e
5	s(g_1)(12,10)2107776665456555432222111 1e
6	s(g_1)(11,10)45670123e
7	s(g_3)(12,8)21076543e

(g_1 : gray level 1 g_2 : gray level 2 g_3 : gray level 3)

Table 11. The Final Result of Fig. 16 by Applying Our Algorithm

Y(Row)	X(Col)
1	$(g_2;\ 7,7)$
2	$(g_1;\ 1,3)$ $(g_2;\ 7,7)$
3	$(g_1;\ 2,3)$ $(g_2;\ 4,10)$
4	$(g_1;\ 1,6)$ $(g_2;\ 7,7)$ $(g_1;\ 13,14)$
5	$(g_1;\ 2,6)$ $(g_2;\ 7,7)$ $(g_1;\ 12,14)$
6	$(g_1;\ 2,3)$ $(g_3;\ 4,5)$ $(g_1;\ 6,6)$ $(g_2;\ 7,7)$ $(g_1;\ 12,14)$
7	$(g_1;\ 1,2)$ $(g_3;\ 3,5)$ $(g_1;\ 6,6)$ $(g_2;\ 7,7)$ $(g_1;\ 10,14)$
8	$(g_1;\ 1,2)$ $(g_3;\ 3,5)$ $(g_1;\ 6,6)$ $(g_2;\ 7,7)$ $(g_1;\ 10,10)$ $(g_3;\ 11,12)$ $(g_1;\ 13,14)$
9	$(g_1;\ 1,2)$ $(g_3;\ 3,3)$ $(g_1;\ 4,8,10,10)$ $(g_3;\ 11,12)$ $(g_1;\ 13,13)$
10	$(g_1;\ 1,5,7,8,10,12)$
11	$(g_1;\ 2,4,11,12)$

(g_1 : gray level 1 g_2 : gray level 2 g_3: gray level 3)

CHAPTER 5
RESTORATION USING CHAIN CODES DESCRIPTION

5.1 Introduction

In [1], the object's connectivity of an image adopted is eight-connectedness, and the contour tracing rule is counterclockwise for an out-contour (external boundary) and clockwise for an in-contour (internal boundary). We implemented the Chang's algorithm and tested on numerous variant types of images. It was observed that there exist four problems when running on some images with an in-contour. A counterexample of using the algorithm is given in Fig. 18. Its chain codes description is obtained as follows [7]:

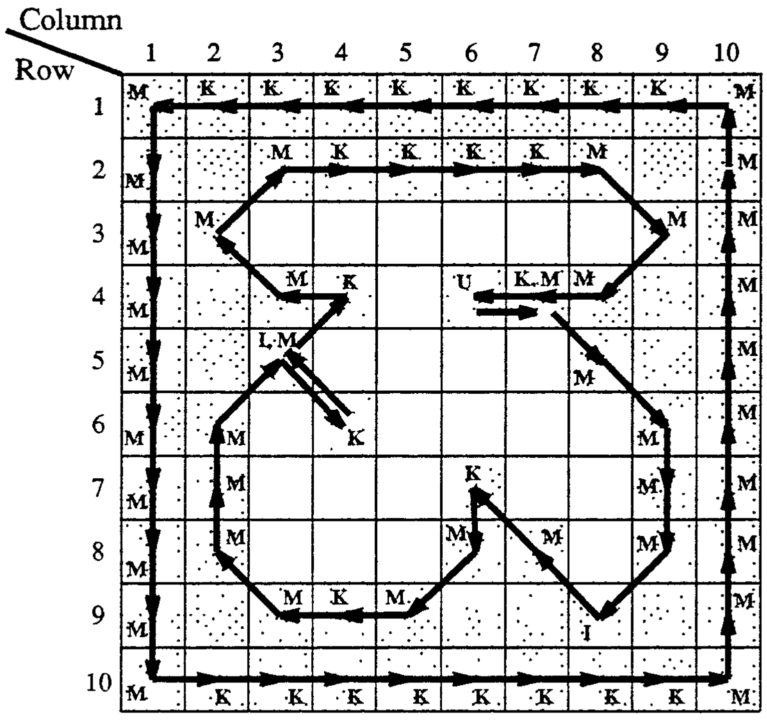
s(1,1)666666666000000000222222222444444444**e**

s(3,2)0000075440776653365443221731431**e$**,

where ''**s**'' is the starting symbol and ''**e**'' is the ending symbol for each contour; two numbers enclosed by a pair of parentheses are the *x*- and *y*-coordinates of the starting point; ''**$**'' is the ending symbol of the description of an image.

The step-by-step results of applying the Chang's algorithm [1] to the in-contour of Fig. 18 is tabulated in Table 12. For clear visualization, the resulting classification is labeled along with the corresponding point in Fig. 18, where **I, M, K,** and **U** denote a singular point, a marking point, a skipping point, and an unsuitable point, respectively. By analyzing the results, the following four problems will arise:

(1) The listed chain codes description of the aforementioned out-contour and in-contour of Fig. 18 is correct according to [7]. However, by Chang's algorithm there exists an unsuitable point at (6,4) and by the quoted rule in [1] ''the description is considered as wrong if an unsuitable point is visited,'' the in-contour chain codes description is mistakenly determined as incorrect.



M : Marking Point K : Skipping Point
I : Singular Point U : Unsuitable Point

Figure 18. A counterexample of using Chang's Algorithm.

(2) The points at (4,4), (4,6) and (6,7) of Fig. 18 that are classified as skipping points will disappear in the reconstructed image according to Chang's algorithm. Hence, the algorithm does not ensure complete reconstruction.

(3) The fourth row of Fig. 18 cannot be reconstructed because there exist three marking points at (3,4), (7,4) and (8,4) and an unsuitable point at (6,4), that violates the parity checking rule.

(4) Disregarding the previous problems, the area of the reconstructed image is always less than that of the original image because the in-contour's pixels are filled in by the background value.

Table 12. Result of Applying Chang's Algorithm to the In-Contour in Figure 18.

Pixel Coordinates	$d_{i(j-1)}\ d_{ij}$	Point	Pixel Coordinates	$d_{i(j-1)}\ d_{ij}$	Point
		Chang's Algorithm			Chang's Algorithm
(4, 2)	0 0	Skipping	(6, 7)	3 6	Skipping
(5, 2)	0 0	Skipping	(6, 8)	6 5	Marking
(6, 2)	0 0	Skipping	(5, 9)	5 4	Marking
(7, 2)	0 0	Skipping	(4, 9)	4 4	Skipping
(8, 2)	0 7	Marking	(3, 9)	4 3	Marking
(9, 3)	7 5	Marking	(2, 8)	3 2	Marking
(8, 4)	5 4	Marking	(2, 7)	2 2	Marking
(7, 4)	4 4	Skipping	(2, 6)	2 1	Marking
(6, 4)	4 0	Unsuitable	(3, 5)	1 7	Singular
(7, 4)	0 7	Marking	(4, 6)	7 3	Skipping
(8, 5)	7 7	Marking	(3, 5)	3 1	Marking
(9, 6)	7 6	Marking	(4. 4)	1 4	Skipping
(9, 7)	6 6	Marking	(3, 4)	4 3	Marking
(9, 8)	6 5	Marking	(2, 3)	3 1	Marking
(8, 9)	5 3	Singular	(3, 2)	1 0	Marking
(7, 8)	3 3	Marking			

The usefulness and accuracy of the algorithm in [1] are dubious due to the above defects. An improved algorithm is proposed and discussed next.

5.2 Improvements and Results

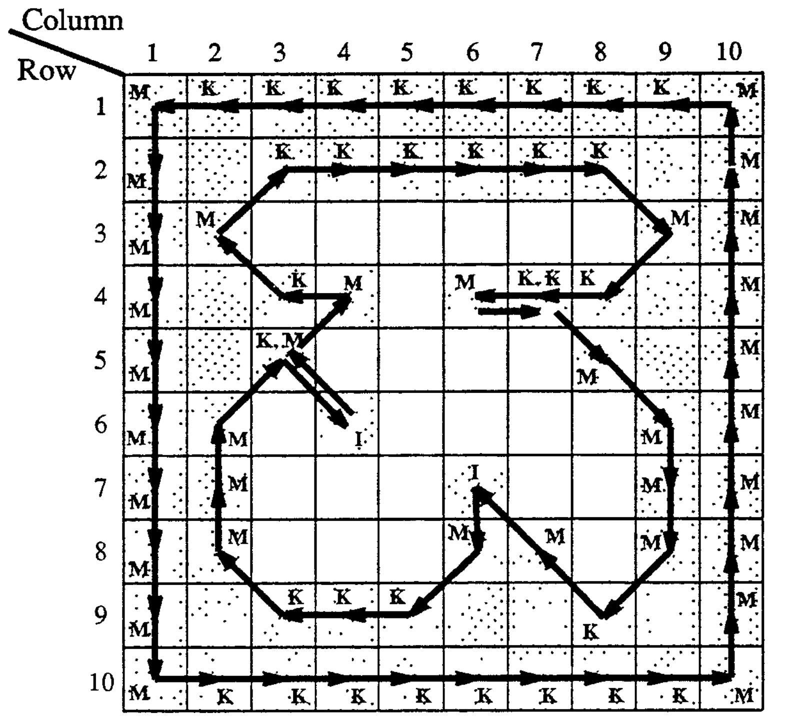
Referring to the y-axis partition algorithm [5], the coordinates of all border pixels of an image are sorted and partitioned into sets so that each set contains only points which have the same y-coordinate. If an object is divided into row by row (y-axis partition), the object pixels are bounded by the starting and ending border pixels in the horizontal direction. Once the coordinates of the two pixels are known, the precise object pixels of this region can be filled in. If the object has internal boundaries (hole borders), the problem can be solved by processing the internal and external boundaries together in the y-axis partition step after the starting and ending border pixels in horizontal direction are extracted. In other words, a couple of starting and ending border pixels will bound a region without regard to what kind of border (i.e. external or internal) it is. Hence, the originally separated internal and external y-axis partitions can be improved by merging them together.

By carrying out the aforementioned concept, the two look-up tables in [1], one for an out-contour and the other for an in-contour, are not necessarily needed. The procedure in extracting the starting and ending border pixels for an in-contour is applied to be identical to that for an out-contour. An example of applying Chang's algorithm only with his Table 12 to Fig. 18 is shown in Fig. 19, where **I, M, K,** represent the same meanings as previously. The step-by-step results of the in-contour are tabulated in Table 13. It has to be noted that the labels of an out-contour in Fig. 18 remains unchanged in Fig. 19, but those of an in-contour are significantly modified.

By comparing the in-contour labels in Fig. 19 with those in Fig. 19, the unsuitable point (labeled as **U**) in Fig. 18 disappears. Besides, the points at (4,4), (4,6) and (6,7) that are no longer skipping points will be preserved in the reconstructed image. The fourth row of Fig.19 that has even number of marking points will now comply with parity-checking rule.

After all the starting and ending border pixels are extracted, they are sorted into another data structure as the Merrill's y-axis partition [5] that is shown in Table 14. As expected, the reconstructed image using Table 14 is exactly the same as the original image.

One of the common problems in region filling is to allow complete reconstruction



M : Marking Point K : Skipping Point I : Singular Point

Figure 19. An example of using the improved algorithm.

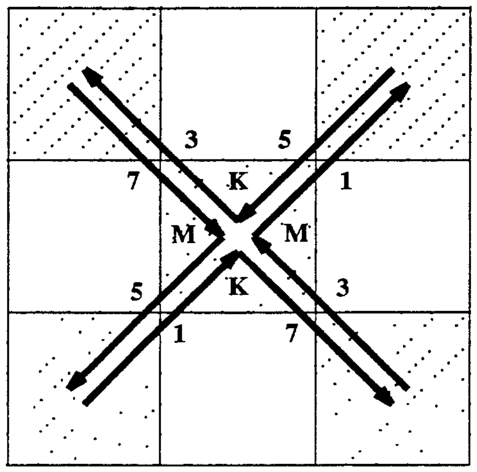
while dealing with multi-junction points by using the chain codes description [3]. An example of a four-way junction point is given in Fig. 20. There are four pairs of links jumping into and out of the central pixel. A problem of unconstructivity will arise by applying Chang's algorithm if any pair of links is not in the same contour as the remaining three pairs. By applying our improvements, the result shows that this four-way junction point is classified and counted twice as both a marking point and a skipping point, so that the complete reconstruction of this type can be easily achieved. The other types such as a three-way junction point and a two-way junction point can be similarly derived.

Table 13. Result of Applying Our Improvement to the In-Contour in Figure 19.

Pixel Coordinates	$d_{i(j-1)}$ d_{ij}	Point	Pixel Coordinates	$d_{i(j-1)}$ d_{ij}	Point
		Improved Algorithm			Improved Algorithm
(4, 2)	0 0	Skipping	(6, 7)	3 6	Singular
(5, 2)	0 0	Skipping	(6, 8)	6 5	Marking
(6, 2)	0 0	Skipping	(5, 9)	5 4	Skipping
(7, 2)	0 0	Skipping	(4, 9)	4 4	Skipping
(8, 2)	0 7	Skipping	(3, 9)	4 3	Skipping
(9, 3)	7 5	Marking	(2, 8)	3 2	Marking
(8, 4)	5 4	Skipping	(2, 7)	2 2	Marking
(7, 4)	4 4	Skipping	(2, 6)	2 1	Marking
(6, 4)	4 0	Marking	(3, 5)	1 7	Skipping
(7, 4)	0 7	Skipping	(4, 6)	7 3	Singular
(8, 5)	7 7	Marking	(3, 5)	3 1	Marking
(9, 6)	7 6	Marking	(4, 4)	1 4	Marking
(9, 7)	6 6	Marking	(3, 4)	4 3	Skipping
(9, 8)	6 5	Marking	(2, 3)	3 1	Marking
(8, 9)	5 3	Skipping	(3, 2)	1 0	Skipping
(7, 8)	3 3	Marking			

5.3 Conclusion

The Chang's algorithm is very efficient and precise to process the out-contour. Nevertheless, there exist four problems when it is applied to complicated image with an in-contour. We have presented the problems by a counterexample and have proposed simple improvements to modify the classification result. Hence, it is feasible to accomplish the robustness, flexibility, efficiency and correctness of the restoration of an image by using the modified Chang's algorithm.



M : Marking Point **K** : Skipping Point

Fig. 20 An example illustrating how the four-way junction point is processed in the improved algorithm.

Table 14. The Result of Figure 19 in Y-axis Partition.

Y(Row)	X(Col)
1	1, 10
2	1, 10
3	1, 2, 9, 10
4	1, 4, 6, 10
5	1, 3, 8, 10
6	1, 2, 4, 4, 9, 10
7	1, 2, 6, 6, 9, 10
8	1, 2, 6, 7, 9, 10
9	1, 10
10	1, 10

CHAPTER 6

THINNING USING MID_CRACK CODES TRACING

6.1 Introduction

This paper is intended to take the advantages of the safe-point thinning algorithm and the contour tracing method. It will be improve the thinning speed and the skeletal shape. The ''safe-point test'' is to determine the deletion of the boundary pixels and the contour tracing is achieved by the mid-crack code. The contour tracing is first in a single scan and the locations of all the contour pixels of the object are stored in buffers. Then in the following step, the pixels in the buffers are processed for the next iteration is generated by look-up tables and set back to the buffers. This technique, compared with the conventional implementation methods for which the repeated scans of the whole image must be used to find the contour pixels, requires much less computation.

The proposed algorithm can perform skeletonization efficiently when processing multiple objects in parallel. The solution we propose is first to retrieve the necessary information during codes extraction, such as boundary-link, coordinates of starting pixel and the first code of the codelink connected with the tail of a boundary-link, into a corresponding object oriented data structure. Then, all the contours of multiple objects are applied simultaneously by the proposed skeletonization algorithm as well as tables look-up.

In Section 6.2.1, the definition of the safe-points and nonsafe points are described clearly. In Section 6.2.2, we will present how to set up the ''safe-point table'' and the ''mid-crack code generation table''. In Section 6.2.3, the ''move table'' and ''coordinates conversion table'' are set up since the algorithm involves contour tracing and coordinates conversion between chain code and mid-crack code. In Section 6.2.4, the proposed algorithm is presented in detail. In Section 6.3, experimental results with the performance are given. Lastly in Section 6.4, we make a conclusion of our algorithm.

6.2 Methodology

6.2.1 Definition of Safe-Point

A pixel in a binary image can be either a black point with the value ''1'' or a white point with the value ''0.'' Mostly the black point indicates foreground and the white point indicates background. Let p_i $(i = 0, 1, \cdots, m-1)$ denote the object's pixels, where m is the total number of pixels in the object. The eight neighbors of p_i shown in Fig. 21 are denoted as n_j $(j = 0, 1, \cdots, 7)$. The points n_0 , n_2, n_4, and n_6 are considered as four neighbors of p_i. If all eight (or four) neighbors are considered while processing p_i , then the object is said to be eight-connected (or four-connected). In this paper, eight-connectedness for the object is used.

The definitions of the border point, end point, break point, and excessive erosion point are given as follows :

- A *border point* is an object point which has at least one of four neighbors exists in the background.
- An *end point* is a border point which has only one eight-neighboring border point.
- A *break point* is a border point and its deletion will cause the loss of connectivity. Eight examples of the break point p_i are illustrated in Fig. 22.
- An *excessive erosion* point is a border point and its deletion will cause the loss of the original objects shape. A few examples of the excessive erosion point p_i are

illustrated in Fig. 23.

n_3	n_2	n_1
n_4	p_i	n_0
n_5	n_6	n_7

Fig. 21. The eight neighbors of the point p_i.

n_4	p_i	n_0

(a)

		n_1
	p_i	
n_5		

(b)

	n_2	
	p_i	
	n_6	

(c)

n_3		
	p_i	
		n_7

(d)

n_3		n_1
	p_i	n_0
n_5		

(e)

n_3		n_1
	p_i	n_0
n_5	n_6	n_7

(f)

n_3	n_2	n_1
	p_i	
n_5		n_7

(g)

n_3		n_1
	p_i	
n_5		n_7

(h)

Fig. 22. A few examples showing that the p_i is a break point.

The thinned, single-pixel wide skeleton must preserve the connectedness and the shape of the original object. In order to ensure the connectedness and shape preservation, the safe-point must be kept. The *safe-point* is defined as one of the end point, the break

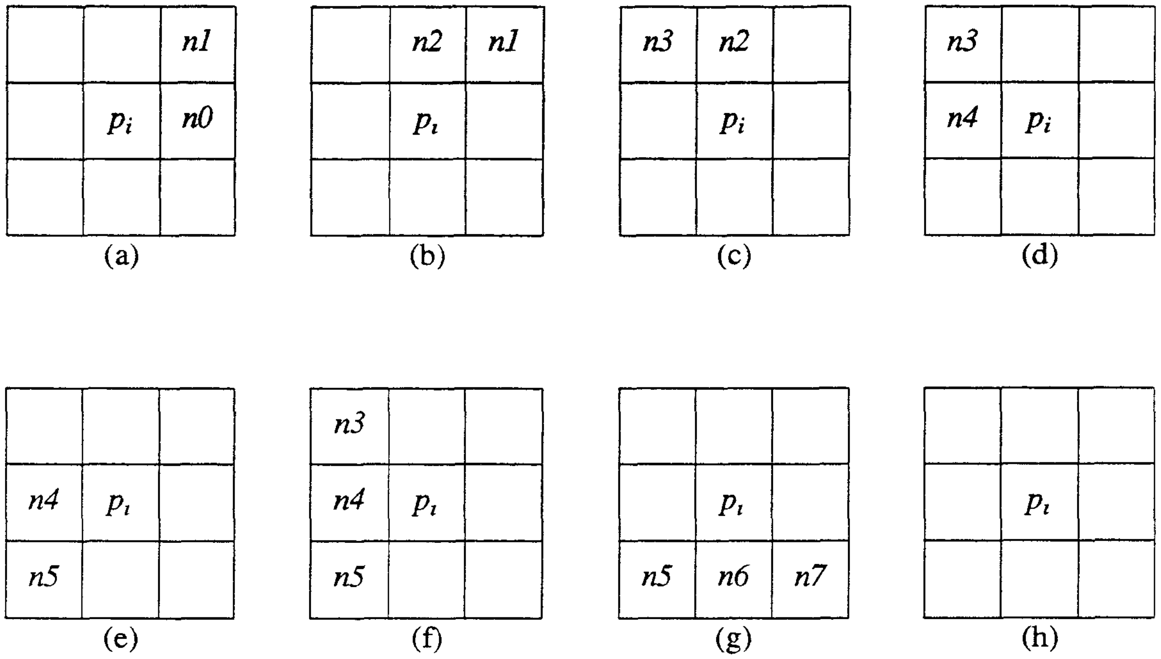


Fig. 23. A few examples showing that the p_i is an excessive erosion point.

point, or the excessive erosion point. All possible permutations of safe and nonsafe points in a local 3×3 window can be tabulated into a look-up table which will be presented in Section 4.

An observation is made that if the excessive erosion pixels remain in the skeleton, the unnecessary short skeletal branches would arise. Two types of resulting skeletons will be illustrated in our experiments. With a simple deletion of excessive erosion pixel, the generated skeleton will reflect the rough object's shape structure but the detail information is disregarded. The skeleton will not require further cleaning or pruning and is useful for generic shape recognition.

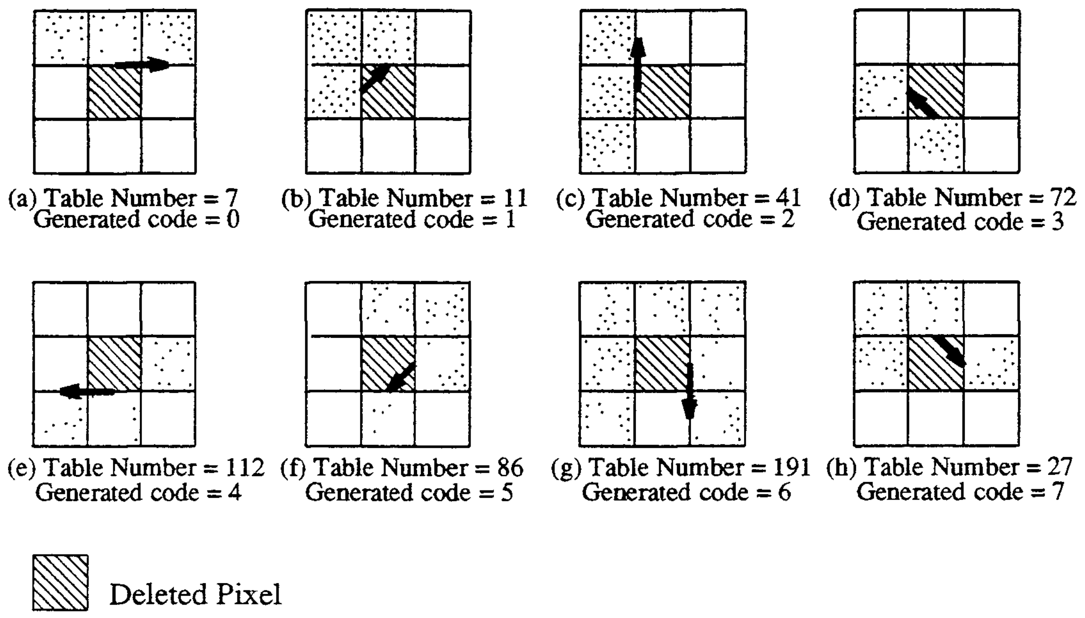


Fig. 24. A few examples illustrating how a new mid-crack code is generated when a border pixel is deleted

6.2.2 Safe-Point Table and Mid-Crack Code Generation Table

All the variant moves of mid-crack codes can be sufficiently described in a local 3×3 window. A 3×3 template incorporated with different weights at different neighbors to indicate the presence of eight neighboring pixels is shown in Fig. 5a. An object pixel and its eight neighbors are then correlated with the 3×3 template. Note that the object pixel is represented as "1" and the background pixel as "0." This correlation is performed to obtain the index value for table look-up which will be described next. An example of the index calculation is shown in Fig. 5b.

Let a codelink be a link of connected codes at the central pixel of a 3×3 window. The information such as total number of codelinks, all the headcodes, all the theads, and all the codelinks can be easily obtained by drawing the mid-crack codes surrounding a

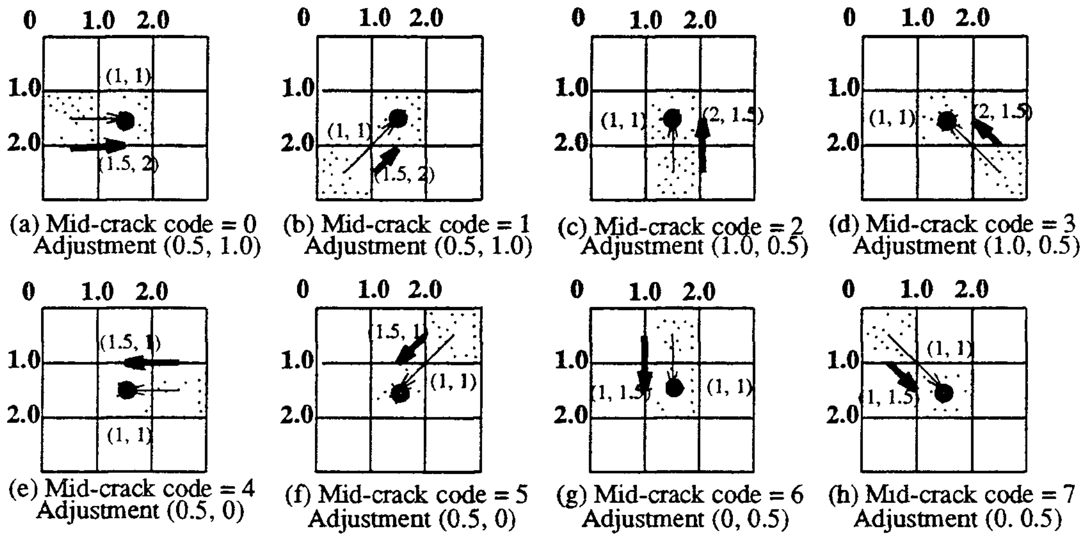

0 1.0 2.0
1.0
2.0
(1, 1)
(1.5, 2)
(a) Mid-crack code = 0 Adjustment (0.5, 1.0)
(b) Mid-crack code = 1 Adjustment (0.5, 1.0)
(2, 1.5)
(c) Mid-crack code = 2 Adjustment (1.0, 0.5)
(d) Mid-crack code = 3 Adjustment (1.0, 0.5)
(1.5, 1)
(e) Mid-crack code = 4 Adjustment (0.5, 0)
(f) Mid-crack code = 5 Adjustment (0.5, 0)
(1, 1.5)
(g) Mid-crack code = 6 Adjustment (0, 0.5)
(h) Mid-crack code = 7 Adjustment (0. 0.5)

Fig. 25. An illustrative example of the conversion table.

central pixel [11]. In this paper, counterclockwise direction is adopted in contour tracing. Therefore, a look-up table is accordingly established in Table 1 and is used to retrieve above information based on the computed index value.

Let *"Algorithm* α *"* denote the one preserves the excessive erosion points and *"Algorithm* β *"* denote the one deletes them. According to the safe-point definition, a safe-point table using the same computed index value can be established as shown in Table 15 to determine whether a border point is deleted or not. If a border point is deleted, a new mid-crack code will be generated to keep track of the next border point. A few examples illustrating how a new mid-crack code is produced when a border pixel is deleted are shown in Fig. 24. Note that in Figs. 24a and 24c, the central pixel being an excessive erosion point is preserved in algorithm α to produce "Nil" and is deleted in algorithm β to produce new codes. All the permutations can be tabulated in Table 16. With Tables 15 and 16, It is feasible and efficient to delete nonsafe points and to continuously trace the boundary at the same time.

Table 15. The Safe-Point Table

Index	Safe Point		Index	Safe Point	
	Algorithm α	Algorithm β		Algorithm α	Algorithm β
0	1	1	43	0	0
1	1	1	...	...	...
2	1	1	95	Nil	Nil
3	1	0	96	1	0
4	1	1	97	1	1
5	1	1	...	...	...
6	1	0	142	1	1
7	1	0	143	1	1
8	1	1	144	1	0
9	1	0	...	...	...
...	...	...	191	0	0
20	1	0	192	1	0
21	1	1	193	1	1
22	0	0	...	...	...
...	...	...	252	0	0
40	1	0	253	0	0
41	1	0	254	Nil	Nil
42	0	0	255	Nil	Nil

Notations : 1 : Safe-point 0 : Nonsafe point Nil : Not exist

Table 16. The Generated Code Table

Index	Code Generated		Index	Code Generated	
	Algorithm α	Algorithm β		Algorithm α	Algorithm β
0	Nil	Nil	43	1	1
1	Nil	Nil	...	...	...
2	Nil	Nil	95	Nil	Nil
3	Nil	0	96	Nil	4
4	Nil	Nil	97	Nil	Nil
5	Nil	Nil	...	...	...
6	Nil	0	142	Nil	Nil
7	Nil	0	143	Nil	Nil
8	Nil	Nil	144	Nil	6
9	Nil	2	...	...	...
...	...	...	191	6	6
20	Nil	6	192	Nil	4
21	Nil	Nil	193	Nil	Nil
22	7	7	...	...	...
...	...	...	252	3	3
40	Nil	2	253	2	2
41	Nil	2	254	Nil	Nil
42	1	1	255	Nil	Nil

Nil : No code being generated

6.2.3 Move Table and Coordinates Conversion Table

The chain codes move along the center of border pixels, while the mid-crack codes move along cracks between two adjacent border pixels. Hence, the coordinate system of chain codes is counted at the middle location between discrete grids and that of mid-crack codes is counted at the location of discrete grids, as shown in Fig. 8. A move table listing the relative coordinates in columns and rows for all the moves from pixel to pixel in the chain code coordinate system is shown in Table 17. For instance, the code 0 indicates one pixel move in x-axis (or in column) and no move in y-axis (or in row). Instead of using mid-crack coded contour for thinning, the boundary pixels are traced whose coordinates are the same as using the chain code coordinate system adjusted by the chain/mid-crack codes coordinate conversion. Since the currently tracing location for thinning in mid-crack code coordinate system is needed for next tracing selection, the chain code coordinates must be converted into new values in the mid-crack code coordinate system. All the conversions from chain code to mid-crack code coordinate systems varied with respect to mid-crack codes from 0 to 7 are illustrated in Fig. 25. In the figure, all the chain code coordinates (1,1) of the central pixel are added with the x- and y-adjustments to be the mid-crack code coordinates of current tracing locations. Once the mid-crack code coordinates are obtained, the current location is compared to the destination location to determine whether the boundary tracing in this iteration is completed. These conversions can be tabulated in a look-up table as shown in Table 18.

Table 17. The Move Table in the Chain-Code Coordinate System

Move	Relative Coordinates	
	Coloumn	Row
0	+1	0
1	+1	-1
2	0	-1
3	-1	-1
4	-1	0
5	-1	+1
6	0	+1
7	+1	+1

6.2.4 The Algorithm

A binary image is first applied by the mid-crack code extraction algorithm [11] to obtain boundary links, starting pixel's coordinates and "headcodes" into an object-oriented data structure, described in the Appendix. A boundary link is a link of connected codelinks (defined in Section 4) for all the border pixels. The "headcode" is the first code of a boundary link connected with the tail of a codelink. All boundary pixels of an object including exterior and interior are processed by thinning algorithm iteratively layer-by-layer until no more nonsafe point in a boundary is found. The algorithm will be ended when all the objects in the image are completed.

As aforementioned, two algorithms α and β can be used. Algorithm α preserves the excessive erosion points, however, algorithm β explores a cleaner skeleton by deleting them. The difference of the two algorithms is that algorithm α treats the ending pixel from the mid-crack code extraction as the last processed pixel in a layer (in the other

Table 18. The Conversion Table from Chain Code to Mid-Crack Code Coordinate Systems

Code	*X* Adjustment	*Y* Adjustment
0	+0.5	+1.0
1	+0.5	+1.0
2	+1.0	+0.5
3	+1.0	+0.5
4	+0.5	0
5	+0.5	0
6	0	+0.5
7	0	+0.5

word, the starting pixel is the first processed pixel), but algorithm β treats it as the first processed pixel. Therefore, the ending pixel can be preserved in algorithm α no matter it is safe or not in the beginning. On the other hand, the ending pixel can be removed in algorithm β if it is a nonsafe point in the beginning. The safe-point table given in Table 15 containing the situation of the two algorithms is used, The pseudo codes of the thinning algorithm α is provided in the Appendix. The essential thinning algorithm α when applying an image with a single boundary is described as follows:

1. The value of the current border pixel, destination pixel and headcode are initialized.
2. The current border pixel is correlated with a weighted 3×3 template to calculate the index value for table look-up.
3. The first code of the codelink is extracted from Table 1 using the index and the corresponding headcode.

4. The current pixel is examined to determine whether it is a safe or nonsafe point from Table 15 using the index.
5. If it is nonsafe, replace its headcode by the generated mid-crack codes from Table 16 using the index from Step 1. The finish flag (Finish) is assigned to ''False'' and the current pixel value is assigned to zero (a background pixel). Otherwise, replace the headcode by the first code from Step 3.
6. Adjust the chain code and mid-crack code coordinates of the current pixel by the first code from Step 3 and Tables 17 and 18.
7. If the mid-crack code coordinates of the current pixel are different from those of the destination pixel, go to Step 2. Otherwise go to Step 8.
8. The current pixel is correlated with a weighted 3×3 template to obtain the index value.
9. The first code of the codelink is extracted from Table 1 using the index and the corresponding headcode.
10. The current pixel is examined to determine whether it is a safe or nonsafe point from Table 15 using the index. If the current pixel is nonsafe, go to Step 11. Otherwise go to Step 12.
11. The current pixel value is assigned to zero and set the finish flag to be ''False''.
12. The chain code coordinates of the current pixel are updated using Table17 and the first code from Step 9. They are then adjusted by Table 18 to be the mid-crack code coordinates of the destination point in the next layer. The code number of Table 18 is obtained from the generated mid-crack codes in Step 10.
13. The headcode is replaced by the generated mid-crack code from Table 16 using the same index in Step 8. The updated current pixel is correlated with the 3× 3 template, and the first code of the codelink is extracted from Table 1 using the

computed index value in Step 8.

14. The headcode in next layer is replaced by the first code, and the chain code coordinates of the starting pixel in the next layer are updated by the current pixel coordinates, Table 17 and the headcode. Go to Step 16.

15. The chain code coordinates of the current pixel are adjusted by Table 18 and the headcode to be the mid-crack code coordinates of the destination point in the next layer. The chain code coordinates of the starting pixel in the next layer are adjusted by the current pixel coordinates, Table 17 and the headcode. The headcode in the next layer is replaced by the first code. Go to Step 16.

16. If the finish flag is "False," go to Step 2 to process the next layer boundary; otherwise, the algorithm is ended.

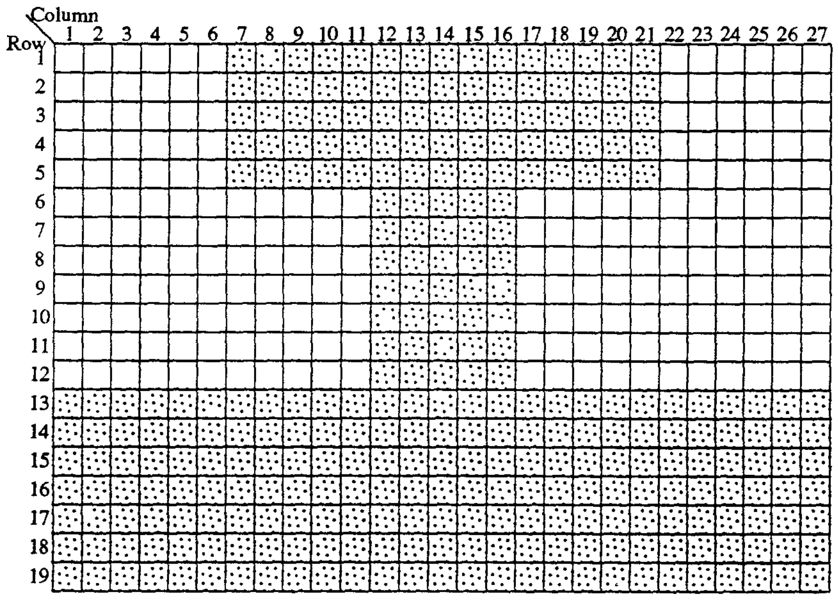


Fig. 26(a). An input image with the symmetric shape.

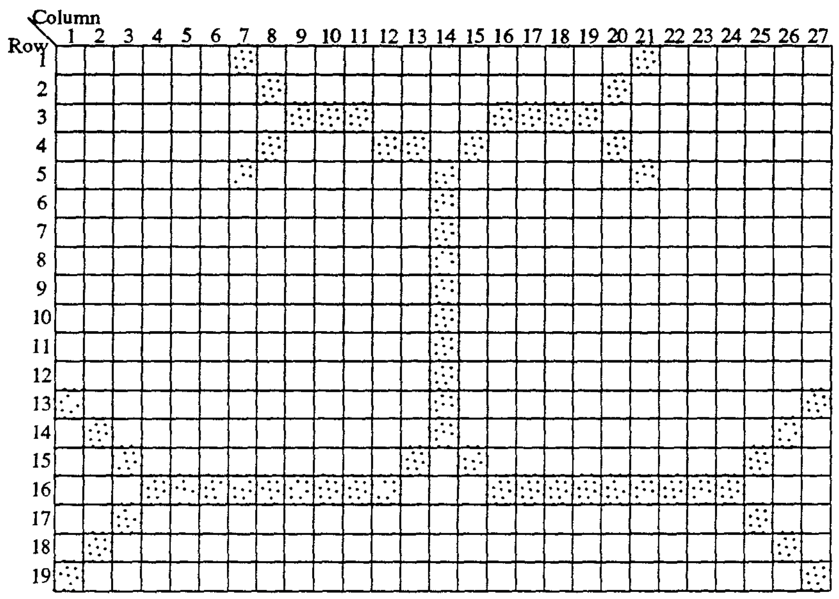


Fig. 26(b). The result of applying Algorithm α to Fig. 26(a).

6.3 Experimental Results

Two results are shown in Figs. 26 and 27. The eight-connectedness for foreground and four-connectedness for background are used. In Fig. 26(a), an image with the symmetric shape is shown. The results of applying algorithms α and β are shown in Figs. 26(b) and 26(c), respectively. In Fig. 27(a), an image with a hole and notches is shown. The results of applying algorithms α and β are shown in Figs. 27(b) and 27(c), respectively. The algorithms have been experimented with successfully on various images with several holes in an object or arbitrary shaped objects. The algorithms are written in C language and implemented on a SUN SPARC workstation under Unix environment. It takes less than 100 ms of cpu time to produce the above resulting figures. A 64 $\times$ 64 image also is tested and takes less than 700 ms of cpu time.

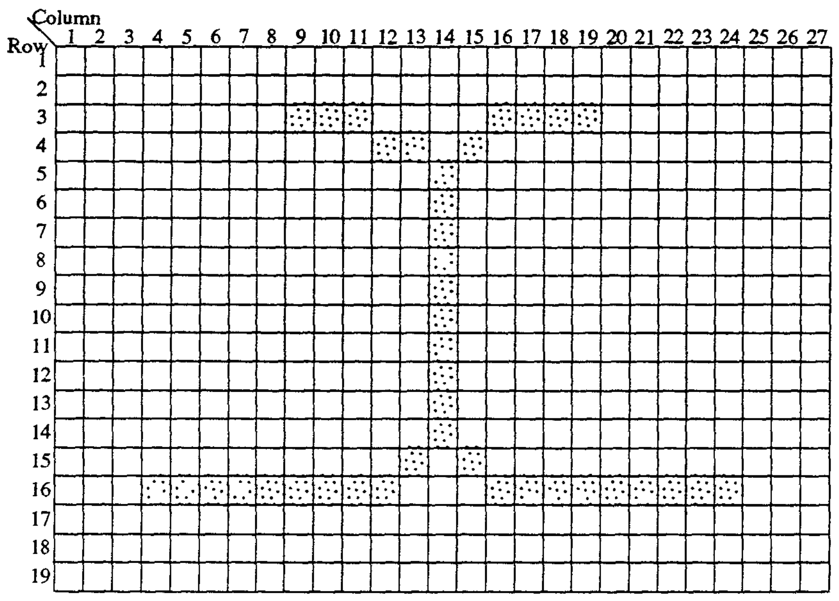


Fig. 26(c). The result of applying Algorithm β to Fig. 26(a).

6.4 Conclusion

The paper presents an efficient thinning algorithm for binary images based on the safe-point testing and mid-crack code tracing. The developed algorithm only needs to process the border pixels at each iteration. The established look-up tables are used to speed up the processing. The algorithm can produce two slightly different results simply by different look-up tables. One of the results which removes the excessive erosion points is a shrink version of the original shape. The other is similar to the medial axis transformation[16].

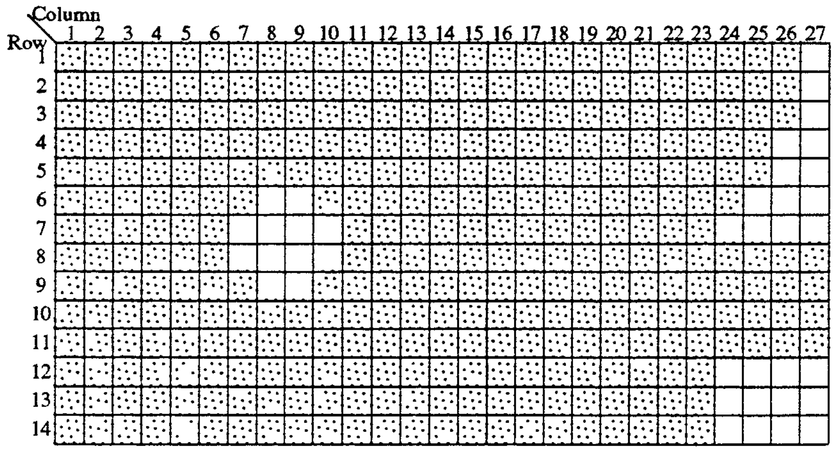


Fig. 27(a). An input image with a hole and notches.

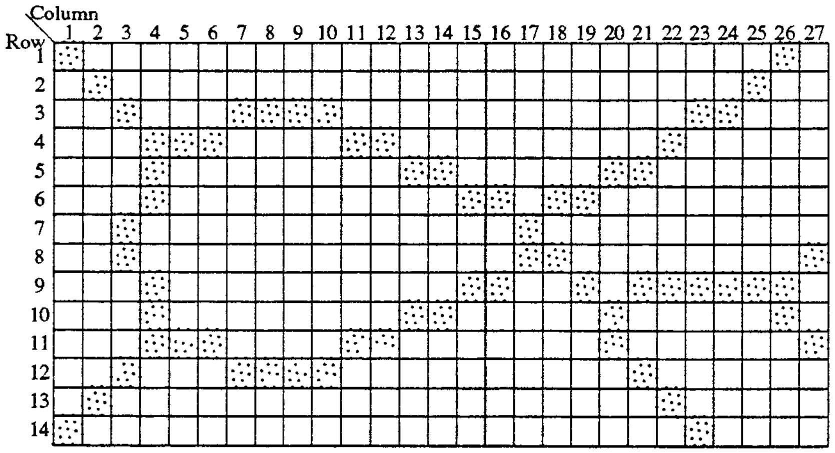


Fig. 27(b). The result of applying Algorithm α to Fig. 27(a).

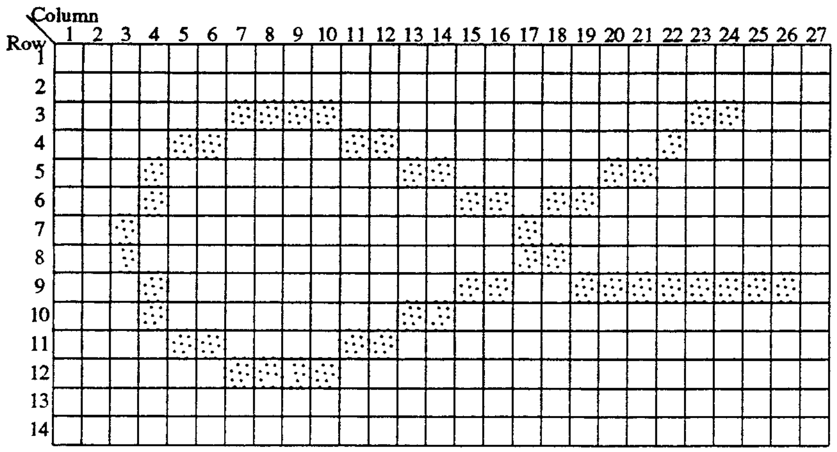


Fig. 27(c). The result of applying Algorithm β to Fig. 27(a).

CHAPTER 7
FUTURE WORK AND CONCLUSION

7.1 Future Work

Until now, our developed algorithms are based on sequential methods. Our next stage will turn to develop parallel algorithms to speed up the processing time. The other approach is to implement the shape recognition by string matching based on mid-crack code contour. However, there exist many applications which using chain codes description that can be replaced by this new mid-crack code approach to take the advantages as we discussed above.

7.2 Conclusion

In this article, we have accomplished a few experiments from which, the results shown that the mid-crack code can be extracted by a fast on-line algorithm accompanied with look-up tables. Besides, a set of algorithms for restoration of binary and grayscale images can be processed efficiently. Other approach in thinning based on mid-crack codes tracing also has a good result. It is expected that the research result will lead to fundamental advances in the precision measurement and recognition by use of the new code. In future works, we will concentrate on algorithms used in parallel computers. So, the research findings produced will also have substantial utility for advancing computer vision technologies.

APPENDIX

Pseudo codes of the thinning algorithm α :

```
typedef struct coordinate_type{
    int x;
    int y;
    }coordinate;        /* Chain code coordinates    */
typedef struct fcoordinate_type{
    float fx;
    float fy;
    }fcoordinate;       /* Mid-crack code coordinates */
typedef struct Boundary_type{
    coordinate  Start;
    fcoordinate Fend;
    char      Headcode;
    int       finish_flag = False;
    }Boundary_record;      /* Object boundary record     */
typedef struct Object_type{
    Boundary_record Bound[MaxNumber1];
    int          finish_flag = False;
    int          Boundary_number;
    }Object_record;      /* Object record          */
Object_record Object[MaxNumber2];
```

```
/* function ObjectThinning */
void ObjectThinning(Dataptr, ROW, COL)
int *Dataptr;              /* Point to the input binary image   */
int ROW, COL;                 /* Total row and column of the image */
{
 int i, j;
 for(i=0 ; i < Object_Number ; i++){                /* For each objects */
   for(j=0 ; j < Object[i].Boundary_number ; j++){  /* For each boundaries */
      Initialize(&Object[i].Bound[j].Headcode, &Object[i].Bound[j].Start,
               &Object[i].Bound[j].Fend);
   }
 }
 for(i=0 ; i < Object_Number ; i++){
   while(Object[i]_finish_flag == False){
    Object[i]_finish_flag = True;
    for(j=0 ; j < Object[i].Boundary_number ; j++){
      while(Object[i].Bound[j].Boundary_finish_flag == False){
        BoundaryThinning(Dataptr, ROW, COL, &Object[i].Bound[j].Headcode,
        &Object[i].Bound[j].Start, &Object[i].Bound[j].Fend
        &Object[i].Bound[j].finish_flag);
      }
      if (Object[i].Bound[j].finish_flag == False)
        Object[i]_finish_flag = False;
      }
    }
  }
}


/* Algorithm α */
int BoundaryThinning(dataptr, row, col, headcode, start, fend, Finish);
int  *dataptr, row, col;
char *headcode;
coordinate *start;
fcoordinate *fend;
int  *Finish;
{
 int end_point = False, table_number;
 coordinate current;
 fcoordinate fcurrent;
 char firstcode;
   current = *start;
   *Finish = True;
   while(end_point == False){
        table_number = MASKING(dataptr, current, col);
        firstcode = FIND_FIRST_CODE(headcode, table_number);
        if(SAFE_POINT_TABLE[table_number] != SAFE){
```

```
        *headcode = ADJUST_HEAD
                (CODE_GENERATED_TABLE[table_number]);
        SET_ZERO(dataptr, current);
        *Finish = False;
      }
      else
        *headcode = ADJUST_HEAD(firstcode);
     current += MOVE(firstcode);
     fcurrent = current + CONVERSION(firstcode);
     if (fcurrent == *fend)
        end_point = True;
  }
  table_number = MASKING(dataptr, current, col);
  firstcode = FIND_FIRST_CODE(headcode, table_number);
  if(SAFE_POINT_TABLE[table_number] != SAFE){
    SET_ZERO(dataptr, current);
    *Finish = False;
    current += MOVE(firstcode);
    *fend = current +
          CONVERSION(CODE_GENERATED_TABLE[table_number]);
    *headcode = ADJUST_HEAD(CODE_GENERATED_TABLE[table_number]);
    table_number = MASKING(dataptr, current, col);
    firstcode = FIND_FIRST_CODE(headcode, table_number);
    *headcode = ADJUST_HEAD(firstcode);
    *start = current + MOVE(headcode);
    }
  else{
    *fend = current + CONVERSION(headcode);
    *start = current + MOVE(firstcode);
    *headcode = ADJUST_HEAD(firstcode);
    }
}
```

BIBLIOGRAPHY

1. Freeman, H. ''Computer processing of line drawing images.'' *Comput. Surveys*, **6,** (1974): **1,** 57-97.

2. Freeman, H., and J. M. Glass. ''On the quantization of line-drawing data.'' *IEEE Trans. Syst. Man Cybern.*, **5,** (1969): 70-79.

3. Rosenfeld, A., and A. C. Kak. *Digital Picture Processing*, Vol. 2, Academic Press, 1982.

4. Dunkelberger, K. A., and O. R. Mitchell. ''Contour tracing for precision measurement.'' *Proc. IEEE Inter. Conf. Robotics and Automation*, St. Louis, (1985): 22-27.

5. Shih, F. Y., and O. R. Mitchell. ''Industrial parts recognition and inspection by image morphology.'' *Proc. IEEE Inter. Conf. Robotics and Automation*, Philadelphia, PA., **3,** (1988): 1764-1766.

6. Moayer, B. and K. S. Fu. ''A tree system approach for fingerprint pattern recognition.'' *IEEE Trans. Comput.*, **25,** (1976): **3,** 262-275.

7. Ogawa, H. and K. Taniguchi. ''Thinning and stroke segmentation for handwritten Chinese character recognition.'' *Pattern Recognition*, **15,** (1982): 299-308.

8. Dill, A. R., Levine, M. D., and P. B. Noble. ''Multiple resolution skeletons.'' *IEEE Trans. Pattern Anal. Mach. Intell.*, **9,** (1987): **4,** 495-504.

9. Kim, S.-D., Lee, J.-H., and J.-K. Kim ''A new chain-coding algorithm for binary images using run-length codes." *Comput. Vision, Graphics, and Image Processing*, **41,** (1988): 114-128.

10. Cederberg, R. L. T. ''Chain-link coding and segmentation for raster scan devices.'' *Comput. Graphics and Image Process.*, **10, (1979)**: 224-234.

11. Chakravarty, I. ''A single-pass, chain generating algorithm for region boundaries.'' *Comput. Graphics and Image Process.*, **15,** (1981): 182-193.

12. Koplowitz, J. ''On the performance of chain codes for quantization of line drawings.'' *IEEE Trans. Pattern Anal. Mach. Intell.*, **3,** (1981): 180-185.

13. Saghri, J. A., and H. Freeman. ''Analysis of the precision of generalized chain

codes for the representation of planar curves.'' *IEEE Trans. Pattern Anal. Mach. Intell.*, **3,** 1981, **5,** 533-539.

14. Cai, Z. ''Restoration of binary images using contour direction chain codes description.'' *Computer Vision, Graphics, Image Processing*, **41,** (1988): 101-106.

15. Pavlidis, T. ''Filling algorithms for raster graphics.'' *Comput. Graphics Image Process.*, **10,** (1979): 126-141.

16. Ackland, B. D., and N. Weste. ''The edge flag algorithm – a fill method for raster scan display.'' *IEEE Trans. Comput.*, **30,** (1981): 41-47.

17. Shani, U. ''Filling regions in binary raster images.'' *SIGGRAPH*, (1980): 321-327.

18. Chang, L.-W., and K.-L. Leu. ''A fast algorithm for the restoration of images based on chain codes description and its applications.'' *Comput. Vision, Graphics, and Image Processing*, **50,** (1990): 296-307.

19. Ali, S. M., and R. E. Burge. ''A new algorithm for extracting the interior of bounded regions based on chain coding.' *Computer Vision, Graphics, Image Processing*, **43,** (1988): 256-264.

20. Merrill, R. D. ''Representation of Contours and Regions for efficient Computer Search.'' *Commun. of ACM* **16,** (1973): **2,** 69-82.

21. Rosenfeld, A. and J. L. Pfaltz. ''Sequential operations in digital picture processing.'' *J. ACM*, **13,** (1966): **4,** 471-494.

22. Stefanelli, R. and A. Rosenfeld. ''Some parallel thinning algorithms for digital pictures.'' *J. ACM*, **18,** (1971): **2,** 255-264.

23. Naccache, N. J., and R. Shinghal. ''SPTA : A proposed algorithm for thinning binary patterns.'' *IEEE Trans. Syst. Man Cybern.*, **14,** (1984): **3,** 409-418.

24. Kwok, P. C. K. ''A thinning algorithm by contour generation.'' *Comm. ACM,* **31,** (1988): **11,** 1314-1324.

25. Vossepoel, A. M., Buys, J. P., and G. Koelewijn. ''Skeletons from chain-coded contours'' *Proc. IEEE Inter. Conf. Pattern Recognition,* Altantic City, NJ., (1980): 70-73.

26. Xu, W., and C. Wang. "CGT : A fast thinning algorithm implemented on a sequential computer." *IEEE Trans. Syst. Man Cybern.*, **17**, (1987): **5**, 847-851.

27. Shih, F. Y., and W.-T. Wong. "A new single-pass algorithm for extracting the mid-crack codes of multiple regions." *Journal of Visual Communication and Image Representation*, **3**, (1992): **1**.

28. Shih, F. Y., and W.-T. Wong. "Restoration of binary and gray-scale images using contour mid-crack codes description." *Journal of Visual Communication and Image Representation*, to be published(1992).

29. Shih, F. Y., and W.-T. Wong. "Restoration of binary and gray-scale images using contour mid-crack codes description." *Proc. IEEE Inter. Conf. Pattern Recognition*, Netherlands, to be published(1992).