Fast Chain Coding of Region Boundaries

Primo Zingaretti, *Member, IEEE*,
Massimiliano Gasparroni, *Member, IEEE*, and
Lorenzo Vecci, *Member, IEEE*

Abstract—A fast single-pass algorithm to convert a multivalued image from a raster-based representation into chain codes is presented. All chain codes are obtained in linear time with respect to the number of chain segments that are generated at each raster according to a set of templates. A formal statement and the complexity and performance analysis of the algorithm are given.

Index Terms—Boundary representations, chain coding, run-length coding, multivalued images, image coding, image processing, shape analysis.

1 Introduction

REGIONS resulting from segmentation must be represented and described in a form suitable for further computer processing. The main methods are based either on the internal characteristics of regions, such as run-length or quadtree representations [14], or on region boundaries, such as chain codes [5].

Chain coding is a very popular error-free coding technique, and is also used extensively in the fields of second-generation image coding [7], image processing [13], and shape analysis [4], [8].

Solutions to chain coding of binary images have been widely investigated in literature. Rosenfeld [12] described a number of algorithms for pixel-to-vector conversion. Sobel [15] used the concept of eight-neighborhood code to develop an efficient method for tracking region boundaries in a binary image. Morrin [9] presented a class of black and white image compression algorithms based on chain-link encoding and decoding of object boundaries. His encoding algorithm searches for objects sequentially and processes each object by a contour follower. Rosenfeld and Kak [13] described a method to convert a row-wise representation of regions (e.g., runs) directly into a boundary representation. More recently, Kim et al. [6] proposed a two-step algorithm to compute chain codes from a particular run-length coded representation that preserves connectivity information between runs. The exacting memory requirements of boundary tracking methods, which need to access the entire image data, can be overcome by the simultaneous chain coding of all region boundaries, but requiring a higher level of algorithmic complexity. If the coding algorithm is able to work in raster scan mode, then only few lines need to be stored. For example, by introducing the RC-code and the concepts of max-point and min-point, Cederberg's algorithm [1] requires two lines at each step to code binary images. Pavlidis [11] developed a two-pass algorithm that encodes the connectivity relationship between adjacent lines into a structure called the Line Adjacency Graph and needs only one raster line stored at a time. Chakravarty [2] proposed an algorithm that converts region boundaries into chain-encoded line structures in a single pass by convolving a 3×3 window with the image. The heart of Chakravarty's chain coding algorithm is the search process that determines whether the link currently decoded from a 2×2 subwindow is connected to some existing chain segment. So, to increase efficiency, a binary search algorithm is used to keep the chain segments sorted.

• *The authors are with Istituto di Informatica, Università di Ancona, 60131, Ancona, Italy. E-mail: zinga@inform.unian.it.*

Manuscript received 9 Sept. 1996. Recommended for acceptance by P. Flynn. For information on obtaining reprints of this article, please send e-mail to: tpami@computer.org, and reference IEEECS Log Number 106438.

A common characteristic of all these algorithms is that they work with binary images and extensions to multivalued images have been scarcely discussed. Using a sequential processing, the input image could be transformed in as many binary images as the number of gray levels or labels. On the contrary, this paper presents a fast chain coding algorithm that processes concurrently all region boundaries in a single scan of the image, using a unified framework for binary, labeled and gray-level images. The generation of chain segments at each raster adopts a set of templates that describe all possible adjacencies between pairs of region parts on two consecutive rows. The algorithm does not require sorting techniques to store the concurrently growing chain segments into ordered lists, so all boundaries can be obtained in linear time with respect to the number of chain segment generations, which are often much less than boundary pixels.

In the following section, we first give some definitions, then we introduce our boundary description scheme and describe the set of templates used in chain segment generation. The chain coding algorithm is presented in Section 3. An example is reported in Section 4, while complexity and performance of the algorithm are analyzed in Section 5. Concluding remarks are given in Section 6.

2 Definitions

We assume the *boundary* of a region constituted by the set of pixels that belong to the region and are also adjacent to another region. The *chain code* of a region is determined by specifying a starting pixel and the sequence of unit vectors obtained from going either left, right, up, or down in moving from pixel to pixel along the boundary. In contrast to the Freeman chain codes [5], which move along the centers, our chain codes, sometimes called *crack codes*, move along the edge points of the boundary pixels. That is, vectors all have length one and lie on a four-connected fixed grid, allowing each direction to be represented by two bits.

2.1 A Boundary Description Scheme Supporting a Set Theory

Our scheme for the description of region boundaries is based on the following assumptions.

ASSUMPTION 1. *An image is partitioned into a collection of eight-connected sets without any distinction between regions and holes.*

A hole of a set S is defined as a connected set H entirely surrounded by S. Unfortunately, the surroundness property has many possible definitions according to what type of connectedness is used (usually eight-connectedness for regions and boundaries and four-connectedness for holes, in binary images). Moreover, in multivalued images, the distinction between regions and holes causes problems to the interpretation of the inclusion and adjacency relationships between two connected sets, so we adopt the following assumption.

ASSUMPTION 2. *For any eight-connected set S, we consider two types of boundaries: the boundary surrounding the set, called external boundary, and the boundaries, if any exist, that surround any subset included in S, called internal boundaries.*

Our algorithm represents external and internal boundaries by chain codes in clockwise and counterclockwise order, respectively.

The definition of new subsets or partitions of an image in terms of given ones is often necessary, so we adopt the following assumption to make these set-theoretic operations possible on chain codes.

ASSUMPTION 3. *Eight-connectedness is used for external boundaries and four-connectedness for internal boundaries.*

In addition, the following constraint holds.

CONSTRAINT. *Except for the outer image boundary, any boundary edge point belongs either to two external boundaries or to one external and one internal boundary.*

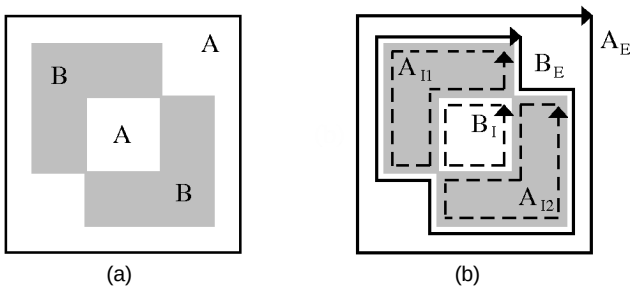


Fig. 1. (a) Multiple connected regions. (b) Their four-direction chain coding in external (solid lines) and internal (dashed lines) boundaries.

This Constraint and Assumption 1 do not permit a connectedness type for external and internal boundaries different from that in Assumption 3. In fact, the adoption of four-connectedness for both boundaries would violate either the set-theory or the above Constraint, while the adoption of eight-connectedness for internal boundaries would require that some regions are described by more than one external boundary. On the contrary, all our regions are described by exactly one external boundary minus the existing internal boundaries. Fig. 1 shows the four-direction chain coding of the eight-connected regions A and B in external (A_E and B_E) and internal (A_{I1}, A_{I2}, and B_I) boundaries. By performing set theoretic operations the two regions are described by:

$$A = A_E - A_{I1} - A_{I2} \quad \text{and} \qquad B = B_E - B_I$$

2.2 Generation of Chain Segments by Adjacency Look-Up-Tables

Our algorithm operates in raster scan mode and is mainly based on the analysis of consecutive runs. On the contrary of the algorithm by Kim et al. [6], we use neither connectivity information between runs nor their coordinates, during the phase of run-length coding. Really, we only need of simultaneously analyzing two rows to detect any variation of a pixel value either on the current row i or on the previous row $i-1$. Thus, our runs, which we name "minimal runs," are different from usual ones, here named "maximal runs." Any maximal run, in any of the two rows, is split in as many minimal runs as maximal runs exist in the other row that are four-adjacent to it, see Fig. 2. On the contrary of maximal runs, each minimal run in row $i-1$ is four-adjacent to exactly one minimal run in row i. This property is at the basis of our method for the generation of chain segments.

DEFINITION 1. *The pair of adjacent minimal runs for which chain segments have to be generated is named CURR_P, while the pair of adjacent minimal runs that precedes (follows)* CURR_P *is named* PREV_P (NEXT_P).

COROLLARY. *At least one of the minimal runs of* PREV_P *and* NEXT_P *belongs to a maximal run with a label different from those of* CURR_P.

DEFINITION 2. *The abscissa of the left (right) edge point of a* CURR_P *is named the starting (ending) position of the* CURR_P.

The generation of chain segments according to a set of pixel patterns (templates) is one of the basic characteristics of our algorithm. We have analyzed all possible combinations of adjacency between each CURR_P and its PREV_P or NEXT_P to discriminate all necessary templates. A cross-tabulation of all resulting different templates, with separation of the cases with equal (first nine templates) or different (48 templates below) labels for the minimal runs of CURR_P, is shown in Fig. 3. The proposed algorithm selects one of these 57 templates and generates chain segments according to it. For this reason, we name this table *Adjacency-Look-Up-Table*, A_LUT. In particular, according to the image type (binary, gray, or labeled images, four- or eight-connected regions), different A_LUTs (B4, G4, B8, G8) can be defined using only a subset of the 57 templates, as reported in Fig. 4.

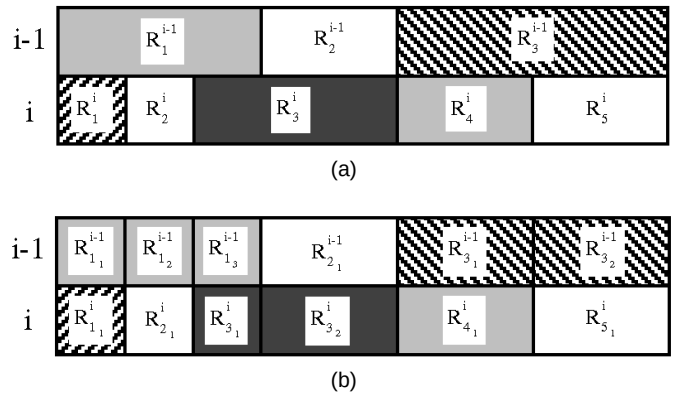


Fig. 2. (a) Usual "maximal run" coding. (b) Our "minimal run" definition.

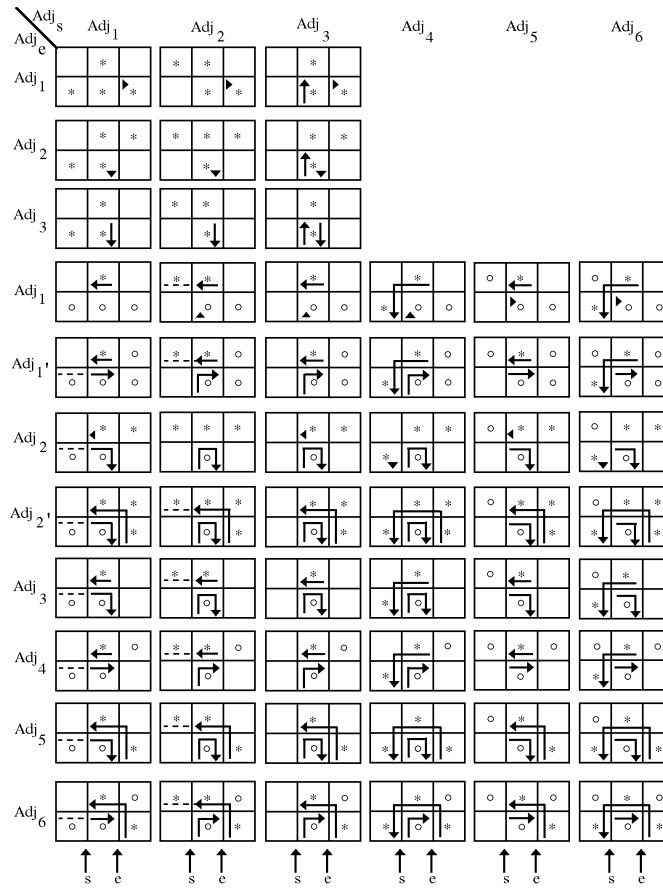
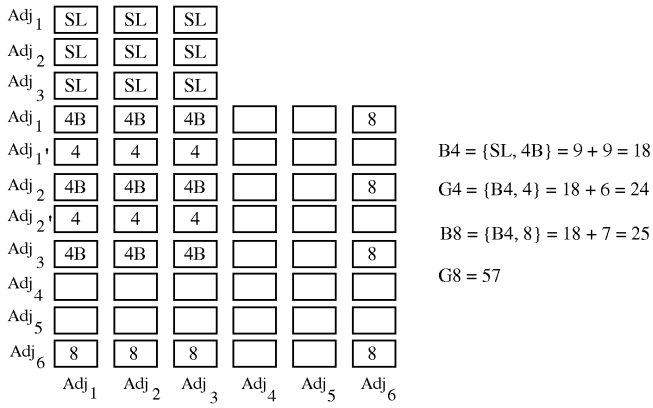


Fig. 3. The set of templates used for the generation of chain segments. Column and row indexes indicate the type of adjacency at the starting (left arrows at the bottom of each column) and ending (right arrows) position of CURR_P, respectively. Blank cells correspond to pixels with labels different from those in CURR_P.

Two general considerations with regard to these templates follow. First, any number of pixels may be between the starting and ending position of the CURR_P, but only one chain segment will be generated for each row. Second, all the chains on the lower (upper) row have a clockwise (counterclockwise) direction, even if they can become either internal or external boundaries.



SL = equals labels, four-connected regions;
4B = different labels, four-connected regions;
4 = different labels, four-connected regions;
8 = templates with different labels used to process eight-connected regions;
B4 = binary images, eight-connected regions;
B8 = templates for processing binary images with eight-connected regions;
G4 = gray/labeled images, four-connected regions;
G8 = gray/labeled images, eight-connected regions.

Fig. 4. A_LUTs and image types. Boxes are in positional correspondence with templates in Fig. 3.

3 The Chain Coding Algorithm

A set of chain lists is the output of the algorithm. Each list stores all the chain codes relative to regions with the same label (in the input image, two or more regions may have the same label if they are not connected). During the processing, the algorithm needs the two auxiliary sets of *pending_chain* and *temp_chain* lists, whose record structure is reported in the following C pseudocode. The symbol PC (TC), when not ambiguous, will be used indifferently to denote a pending_chain (temp_chain) record or only its *code_string* field.

The algorithm scans the image in raster fashion and processes each intermediate row, represented by a list of minimal runs, in two phases. Firstly, using the A_LUT templates, function *create_temp_chain()* analyzes the sequence of CURR_Ps, from left to right, and appends new TCs to the proper temp_chain list. Then, function *link_up()* connects the previously generated TCs with existing PCs, which will represent the searched internal or external chains, once closed. In addition, there is a separated processing for the runs in the first and last raster of the image: function *process_first_row()* generates PCs with strings of the type "10 … 03," while function *process_last_row()* generates TCs with strings of the type "2 … 2."

3.1 Create_temp_chain

The types of adjacency at the starting, Adj_S, and ending, Adj_e, position of CURR_P are defined by:

$$Adj_S = Adj_1 \vee Adj_2 \vee Adj_3 \vee Adj_4 \vee Adj_5 \vee Adj_6$$

$$Adj_e = Adj_1 \vee Adj_{1'} \vee Adj_2 \vee Adj_{2'} \vee Adj_3 \vee Adj_4 \vee Adj_5 \vee Adj_6 = Adj_S \vee Adj_{1'} \vee Adj_{2'}$$

If we regard $Adj_{1'}$ and $Adj_{2'}$ respectively as Adj_1 and Adj_2, we obtain that the adjacency type in the starting position of CURR_P is always equal to the adjacency type in the ending position of PREV_P. Then, for all the CURR_Ps of any intermediate row, function *create_temp_chain()* selects one of the templates of Fig. 3 by analyzing only the type of adjacency in the ending position of each CURR_P. In fact, the type of adjacency at the starting position of NEXT_P, the CURR_P of next processing, is the same as the one at the ending position of the current CURR_P. Once one of the templates has been selected, function *chain_from_A_LUT*() operates according to the following cases:

- *solid segment with an ending arrow*: generation of the chain segment and allocation of a TC;
- *arrow without any attached segment*: a new TC is allocated, but no chain segment is generated; in addition, an ending or a starting position is, respectively, memorized according to whether the arrow is directed outwards (e.g., $A\_LUT_{Adj_2,Adj_4}$) or inwards (e.g., $A\_LUT_{Adj_1,Adj_4}$) the cell in which it is located;
- *dashed segment*: the associated TC was allocated at the beginning of the maximal run, together with the writing of one extreme point, while the chain segment is generated now at once (e.g., $A\_LUT_{Adj_4,Adj_1}$);
- *no graphic symbol*: only the length of the sequence "0 … 0" or "2 … 2" is updated for the chain segment associated with the run (e.g., $A\_LUT_{Adj_2,Adj_2}$).

Algorithm

```
// Data types of main global variables
typedef struct chain * chain_ptr;              // pointer to a record in a chain list
typedef struct chain {
    int  on_x, on_y, off_x, off_y;             // coordinates of starting and ending positions of the chain segment
    int  dir;                                  // direction of coding (C = clockwise, CC = counterclockwise)
    int  flag;                                 // used to signal if a connection with a TC has already occurred
    char *code_string;                         // string of unit directions describing the chain segment
    chain_ptr  next;                           // pointer to the next record in the list
    chain_ptr  prev;                           // pointer to a "suspended" record in the list
};
chain_ptr  * temp_chain, * pending_chain;      // pointers to the auxiliary chain lists
chain_ptr  * ext_chain, * int_chain;           // pointers to the output chain lists
typedef struct minimal_run * run_ptr;          // pointer to a record in a minimal run list
typedef struct minimal_run { int label; int length; run_ptr next};


main()
{   process_first_row();                       // generation of PCs for the first row
    for (i=2; i<=r; i++) {                     // r = rows in the image
            create_temp_chain(i);              // phase 1: selection of templates from the A_LUT and generation of chain segments (TCs)
            link_up(i);                        // phase 2: connection of TCs to PCs
    }
    process_last_row();                        // generation of TCs for the last row
    link_up(r+1);                              // connection of TCs of the last row to PCs
}
```

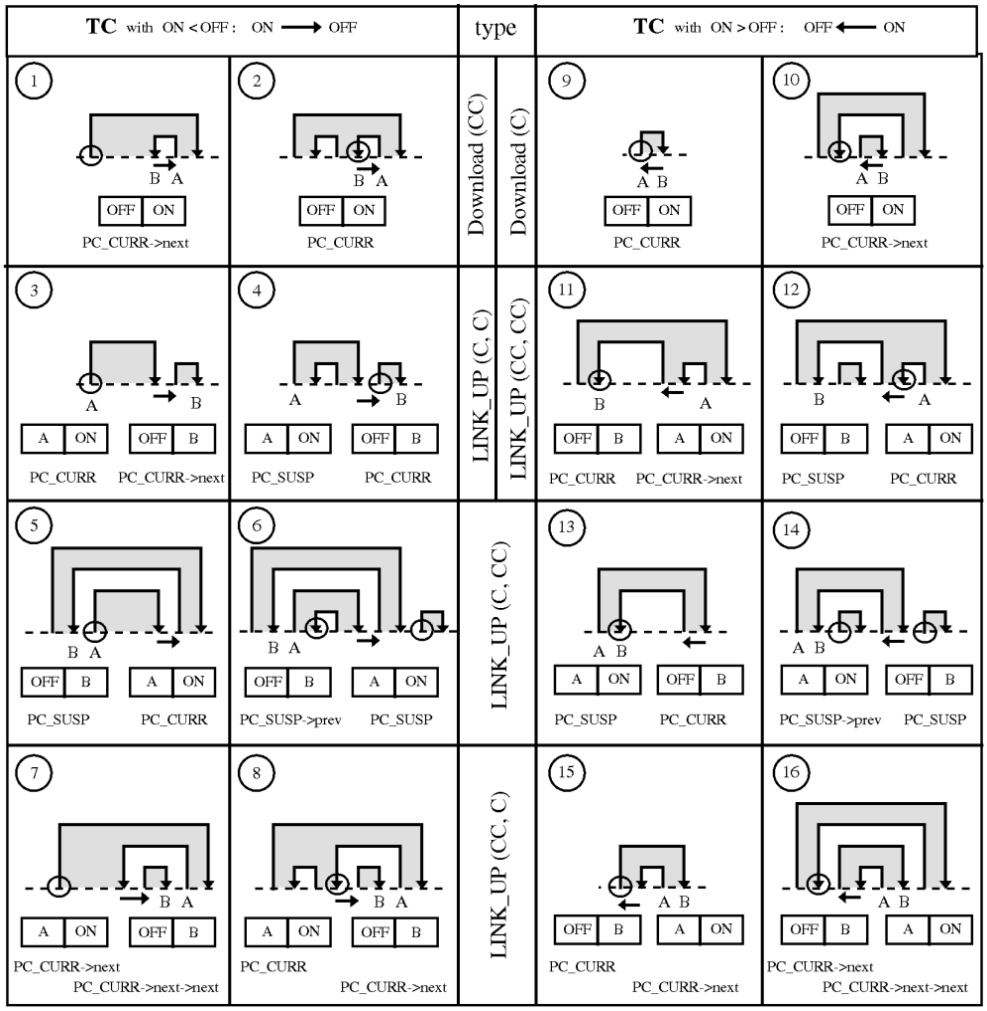


Fig. 5. Merges for different types of connections between Type-2 TCs and PCs. The starting (ending) position of each Type-2 TC is denoted by ON (OFF), while A (B) denotes the starting (ending) position of the PC that must be linked up with ON (OFF). In all cases the PC {A - B} will result from the merge. The box number, circled, is referenced in the formal statement of function *link_up*(). The circles over the chains denote the *min*(*on_x*, *off_x*) of CURR_P (in the cases 6 and 14 both positions are possible), and the two PCs that must be merged are reported on the bottom of each box.

Algorithm (continued)

```
void  create_temp_chain(int  i)
{   run_ptr   p = pointer to the first minimal run in row i-1;
    run_ptr   q = pointer to the first minimal run in row i;
    int Adj_s, Adj_e;                  // types of adjacency at the starting and ending position of CURR_P: Adj_1, ..., Adj_6
    Adj_e= Adj_3;                      // adjacency type of the image left boundary
    for (j=0; j<nb_adj_run_i; j++) {   // nb_adj_run_i = number of pairs of adjacent minimal runs between row i-1 and row i
        Adj_s = Adj_e;                 // the adjacency in the starting position does not need to be analysed
        // phase 1A: selection of the template A_LUT_{Adj_e, Adj_s}
            if (p->label == q->label)  Adj_e = get_end_adj( Adj_1, Adj_2, Adj_3 );
            else {                     // p->label != q->label
                if (p->length == q->length) Adj_e = get_end_adj( Adj_3, Adj_4, Adj_5, Adj_6 );
                else if (p->length > q->length) Adj_e = get_end_adj( Adj_2, Adj_2' );
                else Adj_e = get_end_adj( Adj_1, Adj_1' );          // p->length < q->length
             }
        // phase 1B: N^j_{t_i} chain segment generations;  sum_{j=1}^{nb_adj_run_i} N^j_{t_i} = N_{t_i}
            chain_from_A_LUT( Adj_s, Adj_e);
        p = p->next; q = q->next;      // next pair of minimal runs in row i-1 and in row i  (that is, next CURR_P)
    }
}
```

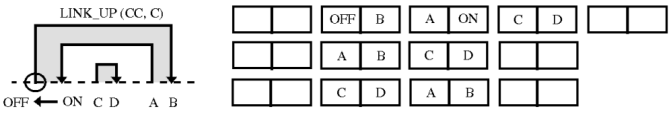


Fig. 6. Alteration of PC order, due to the PC {C - D}, in a LINK_UP(CC, C) merge.

3.2 Link_up

The task of function *link_up()* is to transfer the content of temp_chain lists into the right PCs. Two chain segments are attached when the head of one is connected to the tail of the other. A key aspect of our algorithm is that connections between TC and PC extremes do not need sorting procedures, even if several chains are built concurrently. For this purpose, TCs are classified into three types according to whether only one (Type-1), both (Type-2), or none (Type-3) of their two extremes connect with a PC; in the last case a new PC must be generated.

Function *create_temp_chain()* automatically generates ordered lists of TCs. That is, the starting position of a TC is greater than both the starting and the ending position of the previous TC. Unfortunately, PCs can not be ordered in the same way. However, if we keep the PCs ordered on the minimum between their starting and ending position, *min(on_x, off_x)*, this minimum extreme connects with a TC before any of the two extremes of the next record. Function *link_up()*, by using the two auxiliary pointers PC_CURR and PC_SUSP, can keep this order without needing sorting techniques. PC_CURR always points to the PC that will be firstly used in the search for the concatenation with the TC pointed by TC_CURR. Whenever PC_CURR is advanced before the PC which it points to has taken part in two connections, this PC becomes a "suspended" record and is pointed by PC_SUSP. The chain of suspended records is updated by PC_SUSP->prev. When PC_CURR can not be advanced because the last PC of the list has been processed, the variable *back* is set to TRUE to signal that all remaining TCs must be connected in sequence, that is, without any search, with PC_SUSPs. Finally, PCs involved in a merge preserve the old running direction, which always starts correctly.

Owing to this control strategy, connections with Type-1 TCs must be searched at maximum at the three records pointed by PC_CURR, PC_CURR->next, and PC_SUSP, while Type-3 TCs, inserted just before or after the record pointed by PC_CURR, automatically maintain in order the pending_chain lists.

The situation for Type-2 TCs, as shown in Fig. 5, is a little more complex and must comply with both 6 types of connections between Type-2 TCs and PCs (Download(CC), Download(C), LINK_UP(C, C), LINK_UP(CC, CC), LINK_UP(C, CC), LINK_UP(CC, C) and four different pairs of PCs involved in subsequent merges (PC_CURR ↔ PC_CURR->next, PC_SUSP ↔ PC_CURR, PC_SUSP->prev ↔ PC_SUSP, PC_CURR->next ↔ PC_CURR->next->next).

In practice, one extreme of the Type-2 TC is linked up with either the starting or the ending position of a PC; then, if the other extreme of the TC links up with the other extreme of the same PC a closed boundary is obtained and the chain is downloaded from the pending_chain list, cases 1, 2, 9, and 10 of Fig. 5; otherwise, the PC must be merged with another PC according to one of the other cases of Fig. 5.

Infrequent situations that alter the record order occur whenever PCs exist between *max* (ON, OFF) and *min* (A, B) positions, like in the example shown in Fig. 6. Then, in the cases 7, 8, 15, and 16 of Fig. 5, if both the extremes of the PC {A - B} resulting from the merge still have to be linked up (flag = 0) and another record follows in the list (next NULL), the algorithm executes function *check_on_off*() to control, and eventually fix up, the order of the records in the pending_chain list.

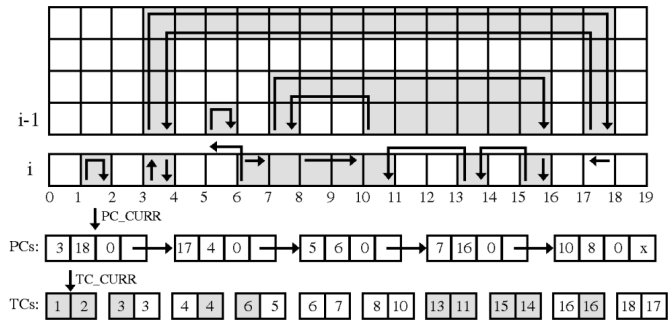


Fig. 7. An example. Processing at row i. From left to right, the starting and ending positions, the *flag* field and the pointer to the next record (x = NULL pointer) are visualized for PCs. TCs are represented in the order they are generated and for each record the starting and ending positions are reported. White and gray backgrounds indicate, respectively, the extreme points that can take part or not in a connection with a PC.

4 An Example

The chain coding of the gray regions shown in the upper part of Fig. 7 is a very significant example of how our algorithm works. The PCs that are active before starting the processing of row *i* and the TCs generated by function *create_temp_chain()* are shown in the lower part of Fig. 7. The first operation of function *link_up()* involves the Type-3 TC {1 - 2} which is inserted before PC_CURR in the pending_chain list. Then, the *off_x* of the Type-1 TC {3 - 3} is compared with the *on_x* of PC_CURR and connected to it. PC_CURR must be advanced to the actual third PC in order to connect the third TC {4 - 4}. This makes the actual second PC a suspended record and, consequently, PC_SUSP is initialized (Fig. 8a). For the same reason, PC_SUSP and the list of suspended records must be updated during the processing of the Type-1 TC {6 – 5}. An interesting case involves the first Type-2 TC {6 - 7}. First, the extreme 6 is attached to the corresponding extreme in the actual fourth PC; then, the fourth PC is merged with the fifth PC, resulting in the situation of Fig. 8b. The processing of the next Type-2 TC {8 - 10} results in the downloading of the internal boundary {8 - 8}. Fig. 8c reports the situation after the processing of the two Type-3 TCs {13 - 11} and {15 - 14}. As PC_CURR now points to the last record in the list and PC_CURR->flag = 2, the processing of the next Type-1 TC {16 - 16} sets the variable *back* to TRUE, connects TC to PC_SUSP, and updates PC_SUSP (Fig. 8d). Finally, without any search because *back* is TRUE, PC_SUSP is connected with the last TC and merged with PC_SUSP->prev.

5 Algorithm Complexity

First, the algorithm complexity is analyzed in terms of routine calls, searches for concatenations and memory allocations. Then, results of tests performed on sample and real images are reported.

5.1 Number of Routine Calls and Searches for Concatenations

The two major functions of the proposed chain coding algorithm, *create_temp_chain()* and *link_up()*, are called once for each row *i* of the image, except for the first and the last row where they are substituted by functions of similar complexity.

The speed of function *create_temp_chain()* is mainly determined by $nb\_adj\_run_i$, the number of pairs of adjacent runs between two consecutive rows. In fact, for $nb\_adj\_run_i$ times function *create_temp_chain()* accesses a template from the A_LUT and performs the automatic generation of the corresponding TCs. The time required by each execution depends on the number of TC generations. Anyway, the maximum number of TCs that function

Algorithm (continued)

```
void link_up(int i)          // N_{s_i} = N_{s_{i-1}};  x_i = 0; y_i = 0
{  for (j=1; j<= nb_active_label ; j++) {    // 1 ≤ nb_active_label ≤ nb_lab
     chain_ptr PC_CURR = pointer to the first record of pending_chain list with label j;
     chain_ptr TC_CURR = pointer to the first record of temp_chain list with label j;
     chain_ptr PC_SUSP = NULL;          // pointer to the last "suspended" (that is, incompletely processed) record
     int back = FALSE;
     for (k=1; k<= L^j_{t_i} ; k++) {          // L^j_{t_i} = number of TCs with label j; Σ_{j=1}^{nb_active_label_i} L^j_{t_i} = N_{t_i}
        switch (high_low (k)) {          // high_low (k) = 1 or 2 or 3
           case 1: // Type-1 TC: only one connection.
              forward_test();            // if both extremes of PC_CURR already linked up (PC_CURR->flag = 2)
                                         // and last record not yet reached (back = FALSE) then advance PC_CURR
              if (back) { connect(TC_CURR, PC_SUSP); update(PC_SUSP); }
              else if (TC_CURR->on_x == PC_CURR->off_x || TC_CURR->off_x == PC_CURR->on_x)
                 connect(TC_CURR, PC_CURR);
              else if (PC_CURR->next != NULL && (TC_CURR->on_x == PC_CURR->next->off_x ||
                                                 TC_CURR->off_x == PC_CURR->next->on_x )) {
                 connect(TC_CURR, PC_CURR->next); create_new(PC_SUSP); advance(PC_CURR); }
                 else { connect(TC_CURR, PC_SUSP); update(PC_SUSP); }
              break;

           case 2: // Type-2 TC: two connections. See Fig. 5.
              forward_test();
              if (back) { connect(TC_CURR, PC_SUSP); merge(PC_SUSP, PC_SUSP->prev); }          // 6, 14
              else if (TC_CURR->on_x == PC_CURR->off_x || TC_CURR->off_x == PC_CURR->on_x) {
                 connect( TC_CURR, PC_CURR);                                  // 2, 3, 4, 5, 8, 9, 11, 12, 13, 15
                 if (PC_CURR->flag == 1 && PC_CURR->on_x == PC_CURR->off_x)
                    download(PC_CURR);    // ext_chain or int_chain according to PC_CURR->dir;    // 2, 9
                 else if (PC_CURR->next != NULL && (TC_CURR->on_x == PC_CURR->next->off_x ||
                                                    TC_CURR->off_x == PC_CURR->next->on_x )) {
                    merge(PC_CURR, PC_CURR->next);                                  // 3, 8, 11, 15
                    if (PC_CURR->flag == 0) check_on_off(PC_CURR); }                // only for 8, 15
                 else merge(PC_CURR, PC_SUSP);                                      // 4, 5, 12, 13
              }
              else if (PC_CURR->next != NULL && (TC_CURR->on_x == PC_CURR->next->off_x ||
                                                 TC_CURR->off_x == PC_CURR->next->on_x )) {
                 connect(TC_CURR, PC_CURR->next);                                   // 1, 7, 10, 16
                 if (PC_CURR->next->flag == 1 && PC_CURR->next->on_x == PC_CURR->next->off_x)
                    download(PC_CURR->next);                                        // 1, 10
                 else if (PC_CURR->next->next != NULL && (TC_CURR->on_x ==
                    PC_CURR->next->next->off_x || TC_CURR->off_x == PC_CURR->next->next->on_x )) {
                    merge(PC_CURR->next, PC_CURR->next->next);                      // 7, 16
                    if (PC_CURR->next->flag == 0) check_on_off(PC_CURR->next); }
              }
              else {     // TC_CURR->off_x = PC_SUSP->on_x (6) or TC_CURR->on_x = PC_SUSP->off_x (14)
                 connect(TC_CURR, PC_SUSP); merge(PC_SUSP, PC_SUSP->prev); }        // 6, 14
              break;
           case 3: // Type-3 TC: no connection with PCs.
              create new_PC from TC_CURR;      // better, transfer TC_CURR to the pending_chain list
              if (PC_CURR->flag == 0) insert new_PC before PC_CURR;
              else {
                 if (PC_CURR->flag == 1)  create_new(PC_SUSP);
                 append new_PC after PC_CURR and set PC_CURR to new_PC; }
              break;
        }
        advance(TC_CURR);
}  }  }

void check_on_off (p) {                  // see Figure 6
   while ((p->next != NULL) && (min(p->next->on_x, p->next->off_x) < min(p->on_x, p->off_x))) {
      exchange_data (p, p->next);
      p = p->next; }
}
```

chain_from_A_LUT() can generate is two and, by assuming each template is equally likely to occur, its average number is 1.33. Then, the number of N_{t_i} TCs generated at row i by function *create_temp_chain()* satisfies the following relations:

$$1 \le N_{t_i} \le 2 * nb\_adj\_run_i$$

and

$$N_{t_i} \cong nb\_adj\_run_i * 1.33$$

The process aiming at finding concatenations between last generated and existing chain segments may be a critical situation. A key feature of our algorithm is that it does not require sorting techniques to connect the stored chain segments; so, for example, it can run faster than Chakravarty's chain coding algorithm [2], which uses a binary search to reduce the number of comparisons necessary to find a concatenation.

Now we will show that the total number of comparisons, NOC, to find concatenations is linear with respect to the total number of TCs,

$$N_t = \sum_{i=1}^{r+1} N_{t_i}.$$

Owing to the above described control strategy, Type-1 TCs can be linked up only with one of the three PCs: PC_CURR, PC_CURR->next, and PC_SUSP, requiring at maximum two comparisons (the connection with the third PC becomes mandatory when these two comparisons fail). Then, by assuming that con-

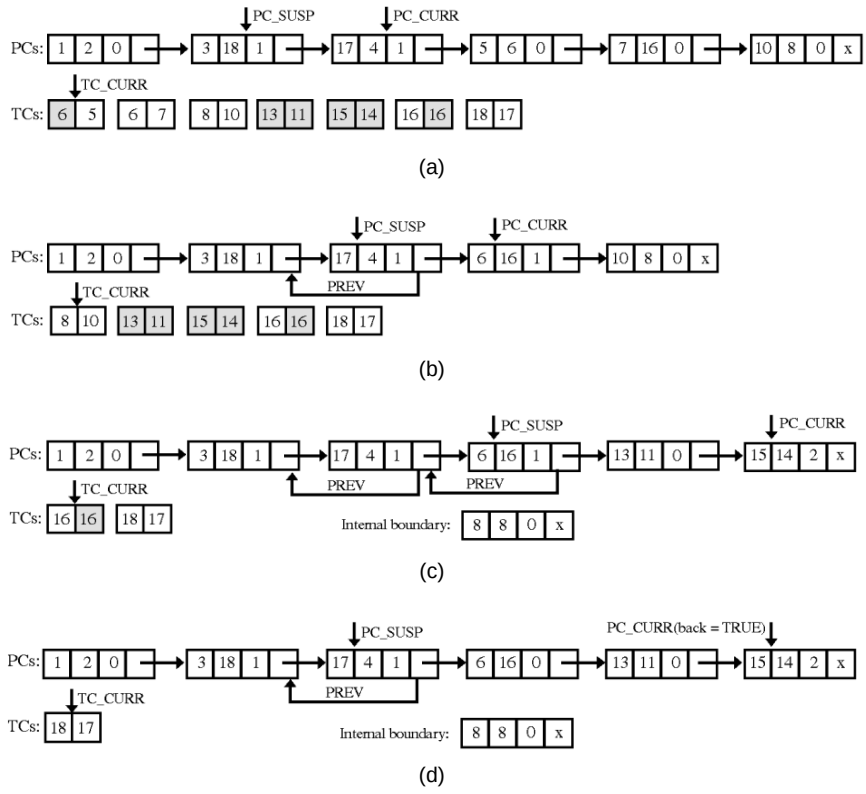


Fig. 8. Situation after the processing of: (a) Type-1 TC {4 - 4}, (b) Type-2 TC {6 - 7}, (c) Type-3 TC {15 - 14}, (d) Type-1 TC {16 - 16}.

nections of a TC with each one of the above PCs are equally likely to occur, Type-1 TCs, N_t^1, require 1.66 * N_t^1 comparisons on the average. On the contrary of Type-1 TCs, for which the PC extreme to be searched is known, the search for a concatenation must be performed both on the starting and the ending position of a PC in the case of Type-2 TCs. In addition, after a *type-y* concatenation between a TC and a PC, Type-2 TCs involve also a *type-x* concatenation between two PCs. Then, following the order adopted in our implementation for the search of *type-y* concatenations, two comparisons at least and three at maximum are necessary in the 10 cases when the TC must be connected with the PC_CURR, and four comparisons at least and five at maximum are necessary in the other six cases. By assuming equal probability for all 16 cases, Type-2 TCs, N_t^2, require 3.37 * N_t^2 comparisons on the average (two comparisons for cases 2 and 9; three comparisons for cases 3, 4, 5, 8, 11, 12, 13, and 15; four comparisons for cases 1, 7, 10, and 16; five comparisons for cases 6 and 14). Type-3 TCs, N_t^3, do not require any search for concatenation, but just a record insertion in the pending_chain list.

Therefore, all chain codes can be obtained in time proportional to the number of chain segment generations, which are much less than the number of boundary pixels. In fact, the greatest number of comparisons to find concatenations is five, and in the hypothesis of equal probability for all events and assuming:

1) $N_t^1 = N_t^2 = N_t^3 = N_t/3$,
2) $N_t^1 = N_t^2 = N_t/2$,

function *link_up*() requires:

1) $NOC = 0.56 * N_t$,
2) $NOC = 1.26 * N_t$

comparisons on the average. Values between these two extremes have been verified in the analyzed real images. Moreover, this very interesting result of $NOC \cong N_t$ is further reduced by the adoption of the variable *back* to signal that the last PC in the list has been already processed: when *back* = TRUE, TC_CURR is always connected with the current PC_SUSP, sewing up all suspended records without making any test.

5.2 Memory Allocations

In terms of execution time, an expensive function is the dynamic-memory allocation of new records, which is mainly determined by the number of TC generations and concatenations. In particular, each concatenation implies four dynamic memory operations: allocation of a new code_string, freeing of the two code_strings to be concatenated and freeing of one of the two involved records.

After the processing of row i, the number of suspended PCs that will be processed on the next row, $N_{s_{i+1}}$, is given by:

$$N_{s_{i+1}} = \left(N_{s_i} - x_i\right) + \left(N_{t_i} - y_i\right) - N_{c_i} \tag{1}$$

where N_{c_i} is the number of chains closed by function *link_up*(), x_i and y_i are, respectively, the number of *type-x* and *type-y* concatenations. Moreover, the following constraints hold:

$$0 \le x_i < y_i;\ 1 \le y_i \le \min\left(N_{t_i}, 2N_{s_i}\right);$$

$$y_i = N_t^1 + N_t^2;\ N_{s_1} = N_{c_1} = x_1 = y_1 = 0.$$

After the execution of *process_last_row*() there will be no suspended PC. So, from (1):

$$\sum_{i=1}^{r+1}\left(N_{t_i} - x_i - y_i - N_{c_i}\right) = 0$$

or

$$\sum_{i=1}^{r+1}\left(x_i + y_i\right) = \sum_{i=1}^{r+1}\left(N_{t_i} - N_{c_i}\right)$$

TABLE 1
VALUES ON SAMPLE AND REAL IMAGES

Image	nb_lab	N_C	Length	N_t^1	N_t^2	N_t^3	NOC^1	NOC_y^2	NOC_x^2
Chess	2	~N^2	$4N^2$	~4N	~N^2	~N^2	~4N	~N^2	~N^2
H_lines	2	N	~$2N^2$	0	N	N	0	N	N
V_lines	2	N	~$2N^2$	~$2N^2$	N	N	~$2N^2$	N	N
Scene1	18	494	21,568	8,294	946	945	9,549	1,612	1,094
Scene2	40	179	32,579	13,622	568	560	14,610	947	702
Scene3	58	5,189	96,105	25,130	7,199	7,191	28,429	9,967	7,865
Scene4	109	157	27,042	9,880	253	251	10,889	354	292

by which the total number of dynamic-memory allocations, *tot_alloc*, and releases, *tot_free*, can be expressed independently from the number of concatenations:

$$tot\_alloc = record\_alloc + code\_string\_alloc = \sum_{i=1}^{r+1} N_{t_i} + \sum_{i=1}^{r+1}\left(N_{t_i} + x_i + y_i\right) = \sum_{i=1}^{r+1}\left(3N_{t_i} - N_{c_i}\right)$$

$$tot\_free = record\_free + code\_string\_free = \sum_{i=1}^{r+1}\left(x_i + y_i\right) + 2\sum_{i=1}^{r+1}\left(x_i + y_i\right) = 3\sum_{i=1}^{r+1}\left(N_{t_i} - N_{c_i}\right)$$

The difference between *tot_alloc* and *tot_free*, $2N_c = 2\sum_{i=1}^{r+1} N_{c_i}$, represents the number of records plus the relative code_strings in the final chain lists.

5.3 Tests on Sample and Real Images

The proposed algorithm has been implemented in C language on a Sun SPARCstation and tested on binary, labeled, and gray-level images. In the last case, the algorithm performs an on-line segmentation of the image by thresholding the input grayscale into preset intervals.

Table 1 reports the values of major variables used in the algorithm analysis for three sample binary images of N × N pixels (Chess: a chessboard; H_lines and V_lines: alternate black and white rows and columns, respectively) and four 256 gray-levels, 512 × 512 pixels, real images of different complexity in terms of number of labels and boundaries (Scene 1-4). The number of records in the final chain lists, N_C, is usually greater than the number of labels, *nb_lab*, because more connected sets may have the same label and the algorithm provides both the external and, if any exists, the internal boundaries of a region. In addition to the number of TCs, Table 1 also reports the number of comparisons necessary to find concatenations involving Type-1 TCs, NOC^1, and Type-2 TCs, computed separately for *type-y* and *type-x* concatenations, NOC_y^2 and NOC_x^2.

Table 2 reports the major results of our algorithm. The total number of comparisons, $NOC = NOC^1 + NOC_y^2 + NOC_x^2$, has a linear behavior with respect to N_t; in particular, the ratio between NOC and N_t is near one. As previously pointed out, in general, N_t is less than the number of boundary pixels, *Length*, and often half this value for the analyzed real images, as it can be seen from the third column of Table 2. The last two columns of Table 2 show how the model assumed for computing the average number of search operations gives values higher than those obtained in real images: In Section 5.1, the ratios $\frac{NOC^1}{N_t^1}$ and $\frac{NOC_y^2 + NOC_x^2}{N_t^2}$ were estimated to be 1.66 and 3.37, without taking into consideration the use of the variable *back*.

6 SUMMARY AND DISCUSSION

Chain coding is a useful technique for many applications. This paper has presented a fast single-pass algorithm to convert a multivalued image from a raster based representation (e.g., a run-length coding) into a four-direction chain code. All possible adjacencies between minimal runs have been described and the data structures to process the information efficiently have been identified.

All regions are processed concurrently, i.e., at the same time, in a single scan of the image. A very important characteristic of our algorithm is that the searches for concatenations between chain segments, *NOC*, are about equal to the number N_t of generated chain segments, which, furthermore, are often much less than boundary pixels, *Length*. This result, which is at the basis of the superior performance against existing techniques, is obtained by using a set of templates that permit to detect region boundaries from a comparison in run-by-run mode instead of pixel-by-pixel mode. For example, our technique has a computational time saving with respect to the algorithms by Chakravarty [2] and Cederberg [1], which both work on a pixel basis, due to the minor number of accesses to templates ($N_t/Length$, third column of Table 2). As already mentioned, further saving may be obtained because the sorting of growing chain segments is not required, as in Chakravarty's algorithm. A fair comparison with our algorithm is possible by adapting Chakravarty's algorithm to compute only line segments, not necessarily closed, of the foreground in binary images. This test yields an equal ratio NOC/N_t for the H_lines image, while for the V_lines image the presence of many pending segments at each line imposes a heavy computational burden of $\log_2(N/2)$ on Chakravarty's algorithm. Finally, regarding Kim et al.'s algorithm [6], a comparative analysis of complexity is not directly possible, even if both algorithms are based on runs. Anyway, as well as Morrin's algorithm [9], Kim et al.'s algorithm is, in

TABLE 2
RESULTS ON SAMPLE AND REAL IMAGES

Image	$\frac{NOC}{N_t}$	$\frac{N_t}{Length}$	$\frac{NOC^1}{N_t^1}$	$\frac{NOC_y^2 + NOC_x^2}{N_t^2}$
Chess	~1	~0.5	~1	~2
H_lines	1.00	~1/N	0.00	2.00
V_lines	1.00	~1	~1	2.00
Scene1	1.20	0.47	1.15	2.86
Scene2	1.10	0.45	1.07	2.90
Scene3	1.17	0.41	1.13	2.48
Scene4	1.11	0.38	1.10	2.55

essence, a sequential contour follower; on the contrary, we are able to encode several boundaries in parallel. It comes out that our algorithm may result much better once implemented on hardware able to exploit its capabilities, and, in any case, our run-length coding module is simpler than Kim et al's module, which needs to extract the ABS and COD tables for successive chain coding.

In conclusion, while we are now working on determining both inclusion and adjacency relationships between regions and junction points at the same time as chain code generation, future research will be addressed to exploit the ability of the algorithm to encode several boundaries in parallel by implementing this algorithm on specialized hardware [3], [10].

REFERENCES

[1] L.T. Cederberg, "Chain-Link Coding and Segmentation for Raster Scan Devices," *Computer Graphics and Image Processing*, no. 10, pp. 224-234, 1979.

[2] I. Chakravarty, "A Single-Pass, Chain Generating Algorithm for Region Boundaries," *Computer Graphics and Image Processing*, no. 15, pp. 182-193, 1981.

[3] L.T. Chen, L.S. Davis, and C.P. Kruskal, "Efficient Parallel Processing of Image Contours," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 15, no. 1, pp. 113-129, Jan. 1994.

[4] M.A. Fischler and H.C. Wolf, "Locating Perceptually Salient Points on Planar Curves", *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 16, no. 2, pp. 69-81, Feb. 1993.

[5] H. Freeman, "Computer Processing of Line-Drawing Image," *Computer Surveys*, no. 6, pp. 57-97, 1974.

[6] S.D. Kim, J.H. Lee, and J.K. Kim, "A New Chain-Coding Algorithm for Binary Images Using Run-Length Codes," *Computer Vision, Graphics, and Image Processing*, no. 41, pp. 114-128, 1988.

[7] M. Kunt, A. Ikonomopoulos, and M. Kocher, "Second-Generation Image Coding Techniques," *Proc. IEEE*, vol. 73, no. 4, pp. 549-574, 1985.

[8] F. Mokhtarian, "Silhouette-Based Isolated Object Recognition Through Curvature Scale Space," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 17, no. 5, pp. 539-544, May 1995.

[9] T.H. Morrin, "Chain-Link Compression of Arbitrary Black-White Images," *Computer Graphics and Image Processing*, no. 5, pp. 172-189, 1976.

[10] C.J. Nicol, "A Systolic Approach for Real-Time Connected Component Labelling," *Computer Vision and Image Understanding*, vol. 61, no. 1, pp. 17-31, 1995.

[11] T. Pavlidis, "A Minimum Storage Boundary Tracing Algorithm and Its Application to Automatic Inspection," *IEEE Trans. Systems, Man, and Cybernetics*, vol. 8, no. 1, pp. 66-69, 1978.

[12] A. Rosenfeld, "Algorithms for Image/Vector Conversion," *Computer Graphics*, no. 12, pp. 135-139, 1978.

[13] A. Rosenfeld and A.C. Kak, *Digital Image Processing*. New York: Academic Press, cap 11, 1982.

[14] H. Samet, *The Design and Analysis of Spatial Data Structures*, Reading Mass.: Addison-Wesley, 1990.

[15] I. Sobel, "Neighbourhood Coding of Binary Images for Fast Contour Following and General Binary Array Processing," *Computer Graphics and Image Processing*, no. 8, pp. 127-135, 1978.