This is the peer reviewd version of the followng article:

A Warp Speed Chain-Code Algorithm Based on Binary Decision Trees / Allegretti, Stefano; Bolelli, Federico; Grana, Costantino. - (2020), pp. 1-8. (Intervento presentato al convegno Joint 9th International Conference on Informatics, Electronics and Vision and 4th International Conference on Imaging, Vision and Pattern Recognition, ICIEV and icIVPR 2020 tenutosi a Kitakyushu, Fukuoka, Japan nel Aug 26-29) [10.1109/ICIEVicIVPR48672.2020.9306532].

Institute of Electrical and Electronics Engineers Inc.
Terms of use:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

19/04/2024 15:50

(Article begins on next page)

A Warp Speed Chain-Code Algorithm Based on Binary Decision Trees

Stefano Allegretti, Federico Bolelli, and Costantino Grana

Dipartimento di Ingegneria "Enzo Ferrari"
Università degli Studi di Modena e Reggio Emilia, Modena, Italy
Email: {*name.surname*}@unimore.it

***Abstract*—Contours extraction, also known as chain-code extraction, is one of the most common algorithms of binary image processing. Despite being the raster way the most cache friendly and, consequently, fast way to scan an image, most commonly used chain-code algorithms perform contours tracing, and therefore tend to be fairly inefficient. In this paper, we took a rarely used algorithm that extracts contours in raster scan, and optimized its execution time through template functions, look-up tables and decision trees, in order to reduce code branches and the average number of load/store operations required. The result is a very fast solution that outspeeds the state-of-the-art contours extraction algorithm implemented in OpenCV, on a collection of real case datasets.**

***Contribution*—This paper significantly improves the performance of existing chain-code algorithms, by smartly introducing decision trees to reduce code branches and the average number of load/store operations required.**

***Index Terms*—Chain-Code, Contours Extraction, Decision Trees, Algorithms Optimization**

I. Introduction

Contours extraction is a common algorithm in binary image processing, usually exploited to represent objects shapes. It has a key role in image storage and transmission, but it is also important for what concerns shape recognition and shape analysis in pattern recognition [1], [2], [3]. In a binary image, a contour is a sequence of foreground pixels that separates an object (connected component) from the background. The output of a contour extraction algorithms is a set of contours, that can be represented in several ways. The simplest is a list of coordinates, but more compact representations also exist: as an example, in Freeman chain-code [4] —the first approach for representing digital curves presented in literature— only one coordinate is stored for a contour, and then all other pixels are identified by a number (from 0 to 7) that encodes its relative position w.r.t. the previous one.

In particular, the chain-code scheme defines eight codes in the directions shown in Fig. 1b. In this case, movements from the center of one pixel to the center of an adjacent one are described. Chain-codes having even value have unit length while those having odd value have length $\sqrt{2}$.

978-1-7281-9331-1/20/$31.00 ©2020 IEEE

Alternative representation of object outlines have been used in the literature, namely crack-codes and midcrack-codes. The crack code scheme defines four codes in the directions shown in Fig. 1a, where each line has length one pixel. The contour is formed by moving on the outer edge of every pixel of the object contour, which means moving along the cracks between adjacent pixels having complementary values. Usually it is assumed that the direction of travel is such that the object pixels are on the right-hand side of the crack. The midcrack-code [5], instead, defines the eight codes shown in Fig. 1c. This encoding scheme defines movement from the midpoint of one crack to the midpoint of an adjacent one. Even valued codes have unit length while odd-valued codes have length $1/\sqrt{2}$. In [6] it is shown that both crack and midcrack coded strings may be obtained from the chain-code string of the contour, and the other way around. For this reason and because of its popularity, in the following we will focus on chain-code only.

Since chain-code techniques allows to preserve information while providing a considerable data reduction, many pattern recognition and topology algorithms based on this representation have been proposed in the last decades.

In [1] the author proposes a new algorithm to describe and generate spirals —a recurring pattern in nature (fingerprints, teeth, galaxies and so on) and thus an important topic for computer vision and pattern recognition— by means of the *slope chain-code*, a variation of the Freeman chain-code originally presented in [7]. In [8] a face identification methodology based on chain-codes encoded face contours is presented, allowing higher efficiency with respect to normal template matching algorithms for face detection.

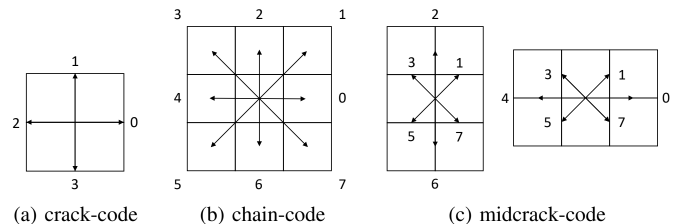


Fig. 1. Different contour coding schemes: (a) crack-code, (b) chain-code, and (c) midcrack-code.

Chain-code has also been used for many object recognition tasks [9]: various shape features extraction, contour smoothing and correlation for shape comparison may be obtained directly from this representation [10]. In [11], a feature extraction method using the chain-code representation of fingerprint ridge contours is presented. The representation allows efficient image quality enhancement and detection of fine minutiae feature points. The authors estimate the direction field from a set of selected chain-codes. Thus, being able to enhance the original fingerprint image using a dynamic filtering scheme that takes advantage of the estimated direction flow of the contours.

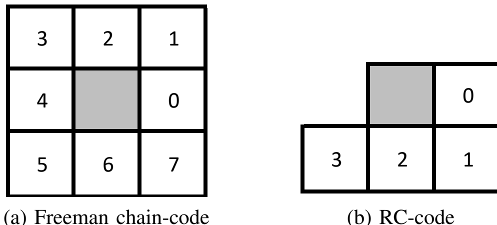
In general, contours extraction is applied as one of the first steps of an image processing pipeline, therefore, it should be as fast as possible. Most contours extraction algorithms follow a boundary tracing approach, that hinders the performance causing several cache misses. Anyway, some works for raster scan chain-code extraction algorithms have been proposed in literature [12], [13], [14].

With this paper, we introduce different optimizations of the algorithm proposed by Cederberg [13]. The use of decision trees, already proven to be a winning strategy for improving the performance of image processing algorithms [15], [16], [17], [18], [19], [20], has shown the best performance on this task: our proposals are able to significantly outperform current software implementations of chain-code algorithms. In Section IV, a thorough evaluation of different variations of our algorithm (presented in Section III) is performed also in comparison with state-of-the-art implementations. The source-code of the proposed algorithm as well as the benchmarking code used to measure performance is publicly available in [21].

II. Contours Extraction

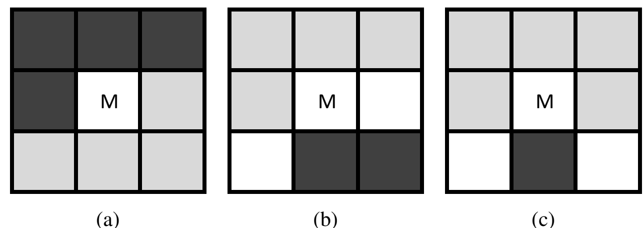
The chain-code variation proposed by Freeman [4], which is the most popular, encodes the coordinates of one pixel belonging to the contour, and then follows the boundary, encoding the direction in which the next pixel shall be found. Since each pixel has only eight neighbors, it is sufficient to use a number from 0 to 7 to identify the next contour point (Fig. 2a). Cederberg [13] proposed another variation, called Raster-scan Chain-code (RC-code), that could ease the retrieval when examining the image in a raster scan fashion. In the RC-code, several coordinates are listed for each contour: they represent the first pixels that are hit in some border during the raster scan process. Each of these pixels is called *MaxPoint*, and is linked to two chains (R-chains), a left and a right one. Every contour pixel met during the scan can either be a MaxPoint (if it is not connected to any already known R-chain) or the next link of some existing R-chain. A MaxPoint can either be an *outer* one, when it is a transition from background to object, or an *inner* one, when at object-background transition. An important difference between outer and inner MaxPoints is that, for inner MaxPoints, the R-chains are swapped, so that the right R-chain actually appears to the left, and viceversa. MaxPoints can be identified by templates in Fig. 3. When proceeding in raster scan, only four directions are possible, so a link is represented by a number from 0 to 3 (Fig. 2b). Templates corresponding to chain links are depicted in Fig. 4. The same pixel can be a link for multiple R-chains. Specifically, a border point that is a link for both a left and a right R-chain is called *MinPoint*. MinPoints have the same templates of MaxPoints, but rotated by 180 degrees. A MinPoint determines the end of two R-chains, which can be merged together. The merging of two R-chains consists in recognizing that the left R-chain continues the same contour traced by the right one, and therefore an ordering between MaxPoints of the same contour can be established.

An example of scan is depicted in Fig. 5. It considers the C++ implementation of the MaxPoint structure described in Listing 1. Members *row* and *col* are the MaxPoint coordinates, *left* and *right* are the chains and *next* is the array index of the following MaxPoint. In Fig. 5a, a contour pixel is met for the first time, and it is recognized as a MaxPoint (M_0). The first foreground pixel of the next row (Fig. 5b) is part of the same contour, but when it is met during the raster scan it does not appear linked to any previously known chain, and it is therefore labeled as MaxPoint (M_1). Then, in Fig. 5c, the scan reaches a MinPoint that links together M_0 left chain and M_1 right chain. After that, the scan can proceed until M_2 is met (Fig. 5d). It configures as an inner MaxPoint, so its R-chains appear swapped. The left R-chain will eventually meet M_0 right one, and the right R-chain will meet M_1 left one. After the end of the scan, all contours are coded in the list of MaxPoint structures, which is the final representation of the RC-code.



(a) Freeman chain-code (b) RC-code

Fig. 2. (a) in Freeman chain-code, each contour pixel is coded with a number for 0 to 7, representing its relative position to the previous one. (b) in RC-code, that is built with a raster scan of the image, only four directions are possible from a link to the next one.



(a) (b) (c)

Fig. 3. MaxPoint templates. Dark squares are background, white squares are foreground and gray squares represent the concept of indifference. All outer MaxPoints correspond to template (a), while inner MaxPoints can either match (b) or (c).

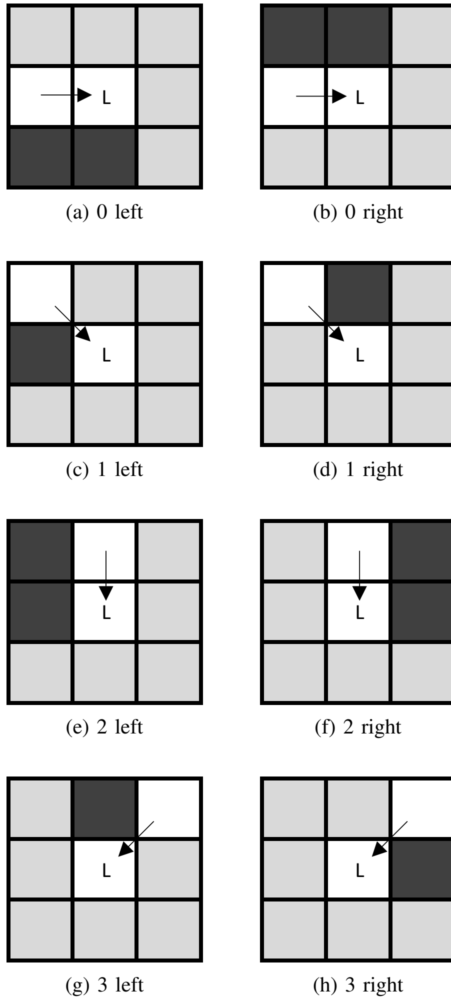
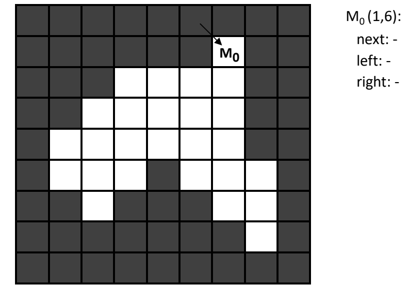


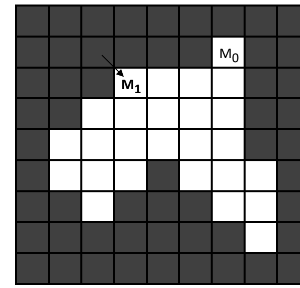
Fig. 4. Chain link templates. Dark squares are background, white squares are foreground and gray squares represent the concept of indifference.

```
struct MaxPoint {
  unsigned row, col;
  Chain left, right;
  unsigned next;
}
```

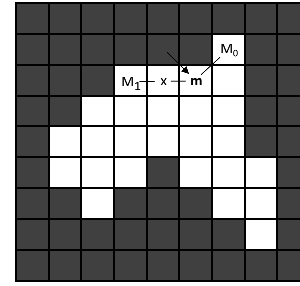
Listing 1. Definition of the MaxPoint data structure in C++.



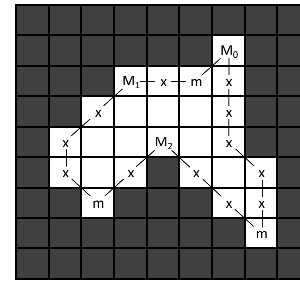
(a)



(b)



(c)



(d)

Fig. 5. Example of RC-code extraction. The algorithm proceeds in raster scan, and in (a), (b) and (c) the arrow points to the pixel currently analyzed. Symbols M, m and x respectively represent MaxPoints, MinPoints and chain links. To the right, MaxPoints are listed, along with their description.

The reconstruction of a contour starts from a MaxPoint and follows its right R-chain until the end; then it follows, in reverse order, the connected left R-chain, and the process goes on until the starting MaxPoint is met again. The RC-code can be converted to Freeman chain-code following the same procedure.

When computing the RC-code, is it sufficient to look at the 3×3 neighborhood of a pixel to recognize its nature, *i.e.*, whether it is a MaxPoint, a MinPoint or a chain link.

Multiple possibilities are viable for the implementation of the *Chain* class, the simplest of which is just the `vector` from the standard library. Anyway, since only 4 kinds of link exist, our *Chain* class has a more efficient implementation that represents a link with only 2 bits.

The RC-code retrieval algorithm uses two data structures: a vector of MaxPoint structures Mv, and a vector of active R-chains Cv, sorted in the order that the raster scan is supposed to meet them along the row. In Cv, each R-chain is simply represented by the index of its MaxPoint in Mv. This implies that both left and right R-chains of a MaxPoint share the same value in Cv, but there is no risk of confusion because right and left chains always alternate in an image row, starting with a left one. During a row scan, pos holds the position in Cv

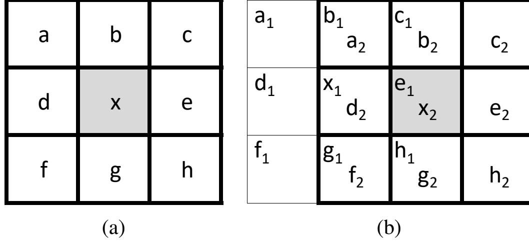


Fig. 6. (a) is the neighborhood mask, containing pixels whose value affects the status of x. (b) illustrates the concept of pixel prediction: 6 pixels of the mask are still inside the mask after the horizontal shift, though with different names.

of the next R-chain that is expected to be met, and to which the next link will be attached.

To sum up, a pixel can be none, one or more of the following: outer/inner MaxPoint, outer/inner MinPoint, left/right link of type 0, 1, 2 or 3.

Thus, from the RC-code point of view, 12 boolean predicates (2 MaxPoints, 2 MinPoints, 8 Links) totally describe a pixel. We say that they constitute the pixel *status*. For each pixel, the procedure is:

- Retrieve the pixel status from its neighborhood.
- Perform an action depending on the status.

The action to be performed on the current pixel can be schematized in the following sequence of steps:

- Move pos backward for every link of type 0. Those are horizontal links, and must be attached to the previously met chain on the same row.
- Iterate through links, from 0 to 3, left before right. For each one, attach it to the chain at position pos, and increment pos.
- If inner MinPoint, merge the last left R-chain met and the chain preceding it, then remove both from Cv.
- If outer MinPoint, merge the last right R-chain met and the chain preceding it, then remove both from Cv.
- If outer MaxPoint, add a new element to Mv, and add its R-chains to Cv before pos. Note that adding the two R-chains to Cv means adding twice the last index of Mv.
- If inner MaxPoint, add a new element to Mv, and add its R-chains to Cv. Same as the previous case, this translates to adding the last index of Mv twice. The two R-chains must be inserted before pos, or before $pos - 1$ if the last R-chain met is a right one.

In our implementation, the action is executed by a C++ function composed of a sequence of *if* statements. This function requires the pixel status as a parameter and its pseudocode is reported in Algorithm 1.

III. Optimization

Our execution time optimization of the RC-code retrieval algorithm concerns both the status retrieval and the action execution.

As regards the former, first of all we computed the status corresponding to every possible 3×3 neighborhood, and we

Algorithm 1: This algorithm provides the Cederberg RC-code action implementation, *i.e.* the instructions required to perform an action on the current pixel, knowing its $status$.

Input: Mv, Cv, pos, r, c, $status$
Procedure `PerformAction()`: **1**
  $last_found_right \longleftarrow false$ **2**
  $pos \longleftarrow pos -$ `CountHorizLinks`$(status)$ **3**
  for $type = 0$ **to** 3 **do** **4**
    if `IsLeftLinkType`$(status, type)$ **then** **5**
      $Mv[Cv[pos]]$.left.push_back($link$) **6**
      $pos \longleftarrow pos + 1$ **7**
      $last_found_right \longleftarrow false$ **8**
    if `IsRightLinkType`$(status, type)$ **then** **9**
      $Mv[Cv[pos]]$.right.push_back($link$) **10**
      $pos \longleftarrow pos + 1$ **11**
      $last_found_right \longleftarrow true$ **12**
  if `IsInnerMinPoint`$(status)$ **then** **13**
    $Mv[Cv[pos-2]]$.next $\longleftarrow Cv[pos-1]$ **14**
    Cv.erase(from=$pos - 2$, to=pos) **15**
  if `IsOuterMinPoint`$(status)$ **then** **16**
    $Mv[Cv[pos-1]]$.next $\longleftarrow Cv[pos-2]$ **17**
    Cv.erase(from=$pos - 2$, to=pos) **18**
  if `IsOuterMaxPoint`$(status)$ **then** **19**
    Mv.emplace_back(r, c) **20**
    Cv.insert(at=pos, cnt=2, val=Mv.size) **21**
    $last_found_right \longleftarrow true$ **22**
  if `IsInnerMaxPoint`$(status)$ **then** **23**
    Mv.emplace_back(r, c) **24**
    if $last_found_right$ **then** **25**
      Cv.insert(at=$pos - 1$, cnt=2, val=Mv.size) **26**
    else **27**
      Cv.insert(at=pos, cnt=2, val=Mv.size) **28**

recorded it in a decision table, to be used as a Look-Up Table (LUT) at runtime, in order to avoid the expensive template matching operation. A reduced version of the LUT is depicted in Fig. 8. When using the LUT to identify a pixel status, 9 pixels must be tested (Fig.6a). Anyway, except for borders, 6 out of those 9 pixels were already tested in the previous scan step, and therefore there is no need to check them again (Fig.6b). This approach is known as pixel prediction [22], and we adopted it to speed up the LUT variation of the RC-code extraction algorithm (LUT_PRED). Then, we observed that, in a number of cases, not all neighbor pixels need to be tested in order to perform the corresponding action. For example, if the central pixel is background, then the status is already established: the pixel is neither a MaxPoint nor a MinPoint nor a link of any kind. This simple observation suggests that the order with which the conditions are verified impacts on the number of load/store operations required. A specific ordering

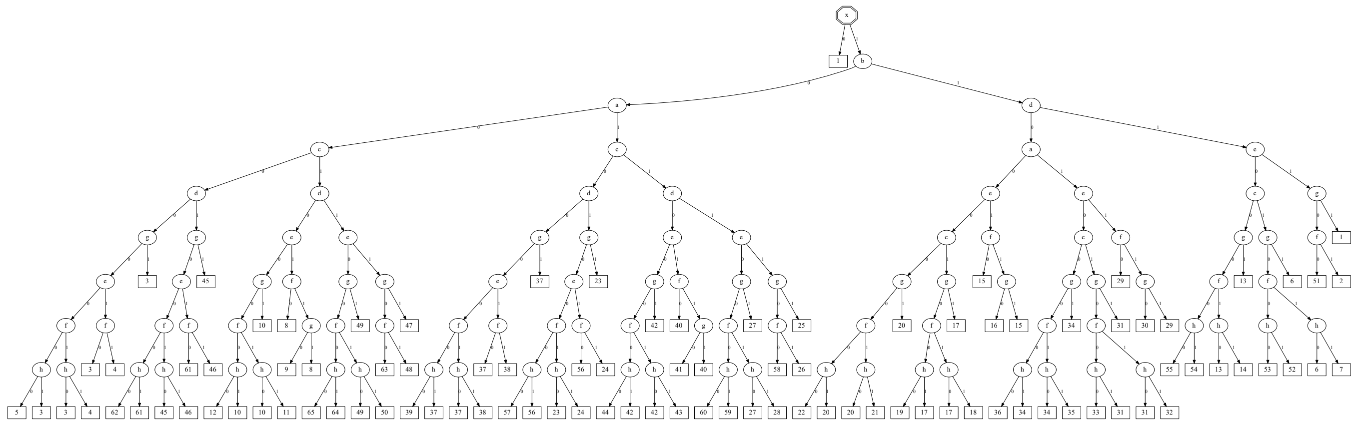


Fig. 7. Binary decision tree which provides the chain-code pixel status with minimal load/store operations. Ellipsis represent conditions to check and leaves represent the status. A status correspond to the group of actions to be performed for the current pixel. Best viewed online. An enlarged detail of the root rightmost branch is reported in Fig. 9.

of condition tests can be represented by a decision tree. A decision tree is a full binary tree, meaning that every non leaf node has exactly two children. Every non leaf node n has a condition, $\mathcal{C}(n)$, corresponding to a neighbor pixel that must be checked when the execution flow reaches that node. Its left and right children l and r contain the pixel to be read if the value of $\mathcal{C}(n)$ is 0 or 1, respectively. Finally, each leaf contains a status.

What we are now looking for is an optimal decision tree, *i.e.*, a tree that allows to minimize the average amount of condition tests needed to reach a leaf. The transformation of a decision table into an optimal decision tree has been deeply studied in the past and we use the dynamic programming technique proposed by Schumacher [23], which guarantees to obtain an optimal solution. One of the basic concepts involved in the creation of a simplified tree from a decision table is that, if two branches lead to the same status, the condition from which they originate may be removed. With a binary notation, if both the condition outcomes 10110 and 11110 lead to status s, we can write that 1-110 leads to status s, thus removing the need of testing condition 2. The dash implies that both 0 or 1 may be substituted in that condition, and represents the concept of indifference. The saving given by the removal of a test condition is called gain in the algorithm, and we conventionally set it to 1.

The conversion of a decision table (with n conditions) to a decision tree can be interpreted as the partitioning of an n-dimensional hypercube (n-cube in short) where the vertexes correspond to the 2^n possible rules. Including the concept of indifference, a t-cube corresponds to a set of rules and can be specified as an n-vector of t dashes and $n-t$ 0's and 1's. For example, 01-0- is the 2-cube consisting of the four rules $\{01000{,}01001{,}01100{,}01101\}$. In summary, Schumacher algorithm proceeds in steps as follows:

- Step 0: all 0-cubes, that is all rules, are associated to a single corresponding status and a starting gain of 0; this means that if we need to evaluate the complete set of

x	a	b	c	d	e	f	g	h	link 0 left	link 0 right	link 1 left	link 1 right	link 2 left	link 2 right	link 3 left	link 3 right	min outer	min inner	max outer	max inner	status
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	3
1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	4
1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	3
1	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	0	3
1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	3
1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	3
1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	3
1	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	3
1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	4
1	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	4
1	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	3
1	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	0	3
1	0	0	0	1	0	0	0	0	1	1	0	0	0	0	0	0	1	0	0	0	62
1	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	61
1	0	0	1	1	1	0	0	0	1	1	0	0	0	0	1	0	0	1	0	0	63
1	0	0	1	1	1	0	0	1	1	1	0	0	0	0	1	0	0	1	0	0	63
1	0	0	1	1	1	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	47
1	0	0	1	1	1	0	1	1	0	1	0	0	0	0	1	0	0	1	0	0	47
1	0	0	1	1	1	1	0	0	0	1	0	0	0	0	1	0	0	1	0	1	48
1	0	0	1	1	1	1	0	1	0	1	0	0	0	0	1	0	0	1	0	1	48
1	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	0	1	0	0	47
1	0	0	1	1	1	1	1	1	0	1	0	0	0	0	1	0	0	1	0	0	47
1	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	51
1	1	1	1	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	51
1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	1	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1
1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	2
1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	2
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1

Fig. 8. Reduced version of the table representing the correspondence between pixel neighborhood and status. The left side contains every possible value of the 9 pixel neighbors, whose names are those of Fig. 6. The complete table has, therefore, $2^9 = 512$ rows. The right side contains the status, expressed with 12 boolean predicates, one for each possible feature of a pixel. Finally, the rightmost column contains the status expressed in a more compact way, as a number from 1 to 65. In fact, only 65 different status are actually possible.

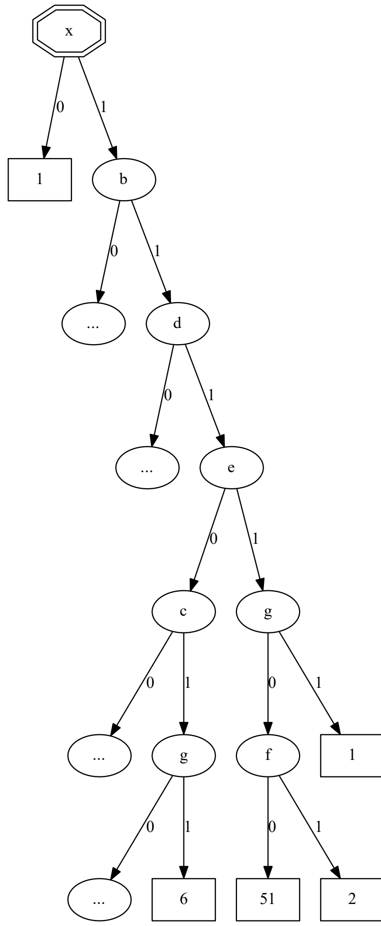


Fig. 9. Detail of the decision tree used to extract the chain-code status of pixel x by the proposed algorithm. Status are expressed in the compact representation of the rightmost column of the table in Fig. 8. If x is background (0) the corresponding status is 1, meaning that no operation is required. A similar situation occurs when x and all the surrounding pixels in the cardinal directions are foreground (rightmost leaf). On the other hand when x, b, c, d, and f are foreground and g is background the decision tree leads to status 2, which means that the current pixel x is a inner MaxPoint: an important thing to be noticed is that in this case the status of x is detected without the need to check pixels a, e and h

Algorithm 2: Decision tree implementation. Border pixels check are omitted for readability purposes.

```
Input: I binary image; r, c row and column indexes.
Output: status of pixel I[r, c].
Function DecisionTree():                                1
    if I[r, c] then                          // x       2
        if I[r − 1, c] then                  // b       3
            if I[r, c − 1] then              // d       4
                if I[r, c + 1] then          // e       5
                    if I[r + 1, c] then      // g       6
                        return 1                        7
                    else                                8
                        if I[r + 1, c − 1] then  // f   9
                            return 2                    10
                        else                            11
                            return 51                   12
                else                                    13
                    // ... the rest of the tree         14
```

Algorithm 3: The complete Cederberg RC-code extraction algorithm.

```
Input: I binary image.
Output: Mv vector of MaxPoints.
Function ExtractChainCode():                            1
    Mv ⟵ vector < MaxPoint >                            2
    Cv ⟵ vector < int >                                 3
    // Cv contains, for each chain, the index           4
       of its MaxPoint in Mv
    for r = 0 to I.rows - 1 do                          5
        pos ⟵ 0                                         6
        for c = 0 to I.cols - 1 do                      7
            status ⟵ DecisionTree(I, r, c)              8
            PerformAction(Mv, Cv, pos, status, r, c)    9
    return Mv                                           10
```

conditions, we do not get any computational saving.

- Step t: all t-cubes are enumerated. Every t-cube may be produced by the merge of two $(t-1)$-cubes in t different ways (for example 01-0- may be produced by the merge of {01-00,01-01} or of {0100-,0110-}). For each of these ways of producing the t-cube (denoted as r in the following formulas) we compute the corresponding gain G_r as

$$G_r = G_r^0 + G_r^1 + \delta[S_r^0 - S_r^1]$$

where ${G^0}_r$ and G_r^1 are the gains of the two $(t-1)$-cubes, and S_r^0 and S_r^1 are the corresponding status. δ is the Kronecker function that provides a unitary gain if the two status are the same or no gain otherwise, modeling the fact that if the status are the same we "gain" the opportunity to save a test. The gain assigned to the t-cubes is the maximum of all G_r, which means that we choose to test the condition allowing the maximum saving. Analogously we have to assign a status to the t-cube. This may be a real status if all rules of the t-cube are associated to the same status, or 0 otherwise: a conventional way of expressing the fact that we need to branch to choose which action to perform. In formulas:

$$S = S_r^0 \cdot \delta[S_r^0 - S_r^1]$$

where r may be chosen arbitrarily, since the result is always the same.

The algorithm continues to execute Step t until $t = n$, which effectively produces a single vector of dashes. The tree may be constructed by recursively tracing back through the merges at each t-cube. A leaf is reached if a t-cube has a status $S \neq 0$. The result is represented in Fig. 7, and an enlarged detail of the rightmost branch is reported in Fig. 9. An excerpt of the

TABLE I

AVERAGE RUN-TIME EXPERIMENTAL RESULTS ON CHAIN-CODE ALGORITHMS IN MILLISECONDS. ASU IS THE AVERAGE SPEED-UP OVER OPENCV. THE STAR IDENTIFIES NOVEL VARIATIONS ON PREVIOUS ALGORITHMS. LOWER IS BETTER.

	3DPeS	*Fingerprints*	*Hamlet*	*Medical*	*MIRflickr*	*Tobacco800*	*XDOCS*	ASU
Suzuki85 (OpenCV)	0.814	1.332	9.252	3.436	1.291	10.089	50.578	1.000
Cederberg_LUT*	2.392	1.733	18.378	7.980	1.960	27.262	118.311	0.499
Cederberg_LUT_PRED*	1.524	1.376	12.652	5.371	1.458	17.682	82.825	0.705
Cederberg_Tree*	**0.613**	**1.092**	**6.749**	**2.950**	**1.136**	**7.534**	**47.545**	**1.231**

code implementing the tree is reported in Algorithm 2.

We also managed to save some computational time by making the action execution function template on the status, instead of passing the status as a parameter. We noticed that only 65 different status can actually occur for a pixel. The general action is composed of a sequence of *if* statements: using C++ templates, 65 different functions are compiled, each only containing the part of the code that must be executed for that status. This allows to avoid a large number of checks and jumps at runtime. Most of those functions result in very small pieces of machine code, and we experimentally verified that no performance issues related to code size have been introduced. Finally, the complete Cederberg RC-code extraction algorithm is reported in Algorithm 3.

IV. EXPERIMENTAL RESULT

In this section, the quality of the optimization introduced is evaluated, and the resulting algorithm is compared to a state-of-the-art alternative. The results discussed in the following have been obtained on a desktop computer running Windows 10 Pro (x64, build 10.0.18362) with an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz, using MSVC 19.15.26730 with optimizations enabled. All discussed algorithms have been proved to be correct, *i.e.*, the output result is exactly the one required by the chain-code extraction task. The experiments are performed on the publicly available YACCLAB dataset [24], [25], which covers most of the applications for binary image processing. The dataset contains six real-world 2D datasets: *3DPeS* is a surveillance dataset for which basic motion segmentation was used to generate the foreground masks [26]. *Fingerprints* is a dataset containing artifical and real-world captured fingerprints, binarized with adaptive thresholding [27]. *Medical* is a dataset containing 343 histological images [28]. *MIRflickr* is an Otsu-binarized version of the MIRflickr dataset [29]. *Tobacco800* is a dataset containing typewritten documents, digitalized using different techniques, therefore resulting in very diverse images [30]. *XDOCS* is a dataset containing high-resolution, handwritten historical documents from the Italian civil registries [31], [32], [33]. The datasets have a highly variable resolution, density, amount of components and originate from highly different sources, captured through various means (scans, photos, microscopy). Full description in [34].

In order to easily compare multiple algorithms, we modified an open source benchmarking framework for thinning algorithms, THeBE [35], and we adapted it to chain-code algorithms. The reference algorithm is the contours extraction algorithm implemented in OpenCV 3.4.7 (Suzuki85) [36], which uses an extremely optimized contour following approach, while the algorithm proposed by Cederberg [13] has been implemented in multiple variants, with increasing optimization of the status retrieval:

- Cederberg_LUT - This variation of the algorithm employs a Look-Up Table that links every neighborhood configuration to a certain status.
- Cederberg_LUT_PRED - This optimization of Cederberg_LUT employs prediction to only check new pixels at every step.
- Cederberg_Tree - Is the optimization based on the optimal binary decision tree.

The Average Speed Up (ASU) is used as measure of the improvement w.r.t. the reference algorithm, Suzuki85.

As it can be observed by comparing LUT and LUT_PRED implementations (Table I), the use of prediction significantly improve performance, simply reducing the number of load operations required at each step of the process. As explained in Section III, six pixels remain inside the mask after its horizontal movement and the PRED optimization removes the need to read them again from the input image. The sped up of LUT_PRED w.r.t LUT implementation ranges from $\times 1.26$ to $\times 1.57$. Anyway, both of the optimizations fails to compete with the carefully designed algorithm in OpenCV (ASU < 1).

Even though the OpenCV contour following approach access the input image in a non cache friendly manner, it requires to update less data structures and thus provides better performances. If we consider experimental results on the *Fingerprints* dataset, we can see that the total execution time of OpenCV and LUT_PRED algorithms is almost the same: *Fingerprints* dataset contains many long vertical connected components, leading to multiple cache misses when using Suzuki85 algorithm. In this specific scenario, the additional data structures required by LUT_PRED compensate for cache misses.

Finally, the algorithm based on the optimal decision tree reduces even more the number of load/store operations, while preserving cache friendly accesses to the input image. This algorithmic solution significantly improves state-of-the-art performance (ASU=1.23).

V. CONCLUSION

We considerably improved the time performance of a raster scan contours extraction algorithm, introducing several optimization such as template functions, look-up tables and, most of all, optimal decision trees.

The result is a very fast algorithm that outspeeds the state-of-the-art solution implemented in the most widespread computer vision library —OpenCV—, over a collection of real datasets representing common use cases for contours extraction.

REFERENCES

[1] E. Bribiesca, “The Spirals of the Slope Chain Code,” *Pattern Recognition*, 2019.

[2] F. Pollastri, F. Bolelli, R. Paredes, and C. Grana, “Improving Skin Lesion Segmentation with Generative Adversarial Networks,” in *2018 IEEE 31st International Symposium on Computer-Based Medical Systems (CBMS)*. IEEE, 2018, pp. 442–443.

[3] ——, “Augmenting data with GANs to segment melanomaskin lesions,” in *Multimedia Tools and Applications*. Springer, 2019, pp. 1–18.

[4] H. Freeman, “On the Encoding of Arbitrary Geometric Configurations,” *IRE Transactions on Electronic Computers*, vol. EC-10, no. 2, pp. 260–268, June 1961.

[5] K. Dunkelberger and O. Mitchell, “Contour tracing for precision measurement,” in *Proceedings. 1985 IEEE International Conference on Robotics and Automation*, vol. 2. IEEE, 1985, pp. 22–27.

[6] G. Wilson, “Properties of contour codes,” *IEE Proceedings-Vision, Image and Signal Processing*, vol. 144, no. 3, pp. 145–149, 1997.

[7] E. Bribiesca, “A Geometric Structure for Two-Dimensional Shapes and Three-Dimensional Surfaces,” *Pattern Recognition*, vol. 25, no. 5, pp. 483–496, 1992.

[8] S. Kundu and B. Ray, “An Efficient Chain Code Based Face IdentificationSystem for Biometrics,” in *2015 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*. IEEE, 2015, pp. 559–564.

[9] J. W. McKee and J. Aggarwal, “Computer Recognition of Partial Views of Curved Objects,” *IEEE Transactions on Computers*, no. 8, pp. 790–800, 1977.

[10] M. D. Levine, *Vision in man and machine*. McGraw-Hill College, 1985.

[11] Z. Shi and V. Govindaraju, “A chaincode based scheme for fingerprint feature extraction,” *Pattern Recognition Letters*, vol. 27, no. 5, pp. 462–468, 2006.

[12] B. Batchelor and B. Marlow, “Fast generation of chain code,” *IEE Proceedings E (Computers and Digital Techniques)*, vol. 127, no. 4, pp. 143–147, 1980.

[13] R. L. Cederberg, “Chain-Link Coding and Segmentation for Raster Scan Devices,” *Computer Graphics and Image Processing*, vol. 10, no. 3, pp. 224 – 234, 1979. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0146664X79900029

[14] P. Zingaretti, M. Gasparroni, and L. Vecci, “Fast Chain Coding of Region Boundaries,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 4, pp. 407–415, 1998.

[15] S. Allegretti, F. Bolelli, M. Cancilla, F. Pollastri, L. Canalini, and C. Grana, “How does Connected Components Labeling with Decision Trees perform on GPUs?” in *Computer Analysis of Images and Patterns*. Springer, September 2019, pp. 39–51.

[16] F. Bolelli, S. Allegretti, L. Baraldi, and C. Grana, “Spaghetti Labeling: Directed Acyclic Graphs for Block-Based Connected Components Labeling,” *IEEE Transactions on Image Processing*, pp. 1999–2012, October 2019.

[17] F. Bolelli, L. Baraldi, M. Cancilla, and C. Grana, “Connected Components Labeling on DRAGs,” in *International Conference on Pattern Recognition (ICPR)*. IEEE, 2018, pp. 121–126.

[21] S. Allegretti, F. Bolelli, and C. Grana, “BACCA: Benchmark Another Chain Code Algorithm.” GitHub Repository, 2020, https://github.com/prittt/BACCA.

[18] F. Bolelli, M. Cancilla, and C. Grana, “Two More Strategies to Speed Up Connected Components Labeling Algorithms,” in *International Conference on Image Analysis and Processing*. Springer, 2017, pp. 48–58.

[19] L. He, Y. Chao, and K. Suzuki, “Two Efficient Label-Equivalence-Based Connected-Component Labeling Algorithms for 3-D Binary Images,” *IEEE Transactions on Image Processing*, vol. 20, no. 8, pp. 2122–2134, 2011.

[20] K. Wu, E. Otoo, and K. Suzuki, “Optimizing two-pass connected-component labeling algorithms,” *Pattern Analysis and Applications*, vol. 12, no. 2, pp. 117–135, 2009.

[22] C. Grana, L. Baraldi, and F. Bolelli, “Optimized Connected Components Labeling with Pixel Prediction,” in *Advanced Concepts for Intelligent Vision Systems (ACIVS)*. Springer, 2016, pp. 431–440.

[23] H. Schumacher and K. C. Sevcik, “The Synthetic Approach to Decision Table Conversion,” *Commun ACM*, vol. 19, no. 6, pp. 343–351, 1976.

[24] F. Bolelli, M. Cancilla, L. Baraldi, and C. Grana, “Toward reliable experiments on the performance of Connected Components Labeling algorithms,” *Journal of Real-Time Image Processing*, pp. 1–16, 2018.

[25] C. Grana, F. Bolelli, L. Baraldi, and R. Vezzani, “YACCLAB - Yet Another Connected Components Labeling Benchmark,” in *2016 23rd International Conference on Pattern Recognition (ICPR)*. Springer, December 2016, pp. 3109–3114.

[26] D. Baltieri, R. Vezzani, and R. Cucchiara, “3DPeS: 3D People Dataset for Surveillance and Forensics,” in *Proceedings of the 2011 joint ACM workshop on Human gesture and behavior understanding*. ACM, 2011, pp. 59–64.

[27] D. Maltoni, D. Maio, A. Jain, and S. Prabhakar, *Handbook of Fingerprint Recognition*. Springer Science & Business Media, 2009.

[28] F. Dong, H. Irshad, E.-Y. Oh *et al.*, “Computational Pathology to Discriminate Benign from Malignant Intraductal Proliferations of the Breast,” *PloS one*, vol. 9, no. 12, p. e114885, 2014.

[29] M. J. Huiskes and M. S. Lew, “The MIR Flickr Retrieval Evaluation,” in *International Conference on Multimedia Information Retrieval*. New York, NY, USA: ACM, 2008, pp. 39–43.

[30] D. Lewis, G. Agam, S. Argamon, O. Frieder, D. Grossman, and J.Heard, “Building a test collection for complex document information processing,” in *Proc. 29th Annual Int. ACM SIGIR Conference*, 2006, pp. 665–666.

[31] F. Bolelli, “Indexing of Historical Document Images: Ad Hoc Dewarping Technique for Handwritten Text,” in *Italian Research Conference on Digital Libraries (IRCDL)*. Springer, 2017, pp. 45–55.

[32] F. Bolelli, G. Borghi, and C. Grana, “Historical Handwritten Text Images Word Spotting Through Sliding Window Hog Features,” in *19th International Conference on Image Analysis and Processing (ICIAP)*. Springer, 2017, pp. 729–738.

[33] ——, “XDOCS: An Application to Index Historical Documents,” in *Italian Research Conference on Digital Libraries*. Springer, 2018, pp. 151–162.

[34] S. Allegretti, F. Bolelli, and C. Grana, “Optimized Block-Based Algorithms to Label Connected Components on GPUs,” *IEEE Transactions on Parallel and Distributed Systems*, pp. 423–438, August 2019.

[35] F. Bolelli and C. Grana, “Improving the Performance of Thinning Algorithms with Directed Rooted Acyclic Graphs,” in *Image Analysis and Processing - ICIAP 2019*. Springer, September 2019, pp. 148–158.

[36] S. Suzuki and K. Abe, “Topological Structural Analysis of Digitized Binary Images by Border Following,” *Computer Vision, Graphics, and Image Processing*, vol. 30, no. 1, pp. 32 – 46, 1985.