Journal of Pattern Recognition Research 1 (2010) 10-22

Received January 18, 2010. Accepted July 1, 2010.

In-place Algorithm for Connected Components Labeling

Tetsuo Asano *t-asano@jaist.ac.jp*
School of Information Science, JAIST,
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan

Hiroshi Tanaka *gja164@jaist.ac.jp*
School of Information Science, JAIST,
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan

Abstract

Connected components labeling of a binary image is one of the most fundamental operations in image processing. This paper presents two linear-time algorithms on this topics, one for just reporting the number of connected components and the other for labeling every element of value 1 (or white pixel) by its component index. An important feature here is that they can be implemented without using any extra array other than the one for an input image and they are suitable for embedded software.

Keywords: In-place algorithm Components labeling Constant work space Image processing.

WWW.JPRR.ORG

1. Introduction

Demand for embedded software is growing toward highly functional hardware. Embedded software has more severe constraint in its size of local memory available. For example, to design an intelligent scanner a number of algorithms should be embedded in the scanners. In most of those cases, the amount of work space available for such software is severely limited. In the sense algorithms which require a limited amount of work space and run reasonably fast are desired.

In this paper we consider the connected components labeling, one of the most fundamental and important problems in image processing. We first consider a simpler problem of finding the number of connected components in a given digital image or binary matrix. We prove that the problem can be solved in linear time, that is, in time proportional to the number of pixels, without using any extra array other than the one for an input image. Then, we extend the algorithm for the second problem of connected components labeling in which every pixel value should be replaced with the index of a component to which the pixel belongs. Note that each pixel of value 0 must remain 0. Our algorithms runs in linear time in the same framework as above.

This kind of topics may belong to Image Processing or Computer Vision. Unfortunately, as far as the authors know, there are few studies in this direction. As images are growing in size, space-efficient algorithms become more important. The algorithms in this paper may be good sources to other space-efficient algorithms.

This paper is organized as follows. In Section 2 we give mathematical statements of our problems after preparing some notations and definitions. Then, in Section 3 we present an algorithm for computing the number of connected components without using any extra array. We extend the algorithm in Section 4 to the problem of connected components labeling again without using any extra array. In Section 5 we conclude the paper with some open problems.

© 2010 JPRR. All rights reserved. Permissions to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or to republish, requires a fee and/or special permission from JPRR.

2. Problem Statement

Let G_n be an $\sqrt{n} \times \sqrt{n}$ binary image (or matrix) such that each pixel value $G(x, y)$ is either 0 or 1. [1] A pixel of value 1 (resp., 0) is called a **white pixel** (resp., **black pixel**). For a pixel $p(x, y)$, we define the neighborhoods

$$N_4(p) = N_4(x, y) = \{(x, y), (x+1, y), (x-1, y), (x, y+1), (x, y-1)\}, \text{and}$$
$$N_8(p) = N_8(x, y) = \{(x', y') \mid x' = x-1, x, x+1, y' = y-1, y, y+1\}.$$

Two white pixels p and q are called **4-connected** (resp., **8-connected**) if there exists a sequence of white pixels $(p = p_0, p_1, \ldots, p_k = q)$ in G_n such that $p_{i+1} \in N_4(p_i)$ (resp., $p_{i+1} \in N_8(p_i)$) for every $i = 0, 1, \ldots, k-1$.

A **4-connected component** (resp., **8-connected component**) of a binary image G_n is a maximal set of white pixels in G_n such that any two of them are 4-connected (resp., 8-connected).

Let $C_1, C_2, \ldots, C_\kappa$ be (4- or 8-)connected components in a binary image G_n. Assume the raster order over pixels of G_n, which is equivalent with the lexicographical order of those pixels in (y, x), that is,

$$(x, y) < (x', y') \text{ if and only if } \ y < y' \ \text{ or } \ y = y' \ \text{ and } \ x < x'.$$

Assuming this ordering, a pixel in a component C_i that is smallest in the lexicographical order is referred to as the **canonical pixel** of the component. The same lexicographical order defines a natural order among components. A component C_i precedes another component C_j if C_i's canonical pixel precedes C_j's canonical pixel in the order. When $(C_1, C_2, \ldots, C_\kappa)$ is the lexicographical order of all components in a binary image G_n, the **index** of the i-th component C_i is defined by $i + 1$.

We can imagine a unit square for each pixel. Then, each side of the unit square is called an edge. A **boundary edge** is an edge between two consecutive pixels of different colors. Throughout the paper we orient each boundary edge so that white pixel always lies to its left. Then, by the definition of the canonical pixel of a component, the vertical edge to the left of the canonical pixel is downward.

Given any boundary edge e, it is easy to follow the whole boundary containing e. Figure 1 illustrates local rules to follow a boundary of a 4-connected component. Rules for other directions are just symmetric. Of course, we need different rules for 8-connected components, which will be described in a later section. So, for the time being, we implicitly assume the 4-connectivity.

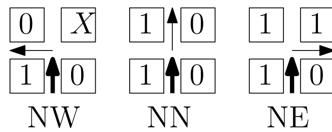

Fig. 1: Rules for following a boundary of a 4-connected component. Bold arrows are current boundary edges and thin ones are the succeeding edges. The pixel denoted by X is either 0 or 1.

We have defined a 4-connected component as a set of white pixels, but it is also possible to characterize it by its boundary. In general, a connected component has a single external boundary and any number of internal boundaries delimiting internal holes. Following our

[1] We assume a square image for simplicity, but extension to rectangular images is straightforward.

convention each external boundary is oriented in a counter-clockwise manner while each internal one is clockwisely oriented.

We can apply the same lexicographic order to vertical boundary edges. Two vertical edges are ordered using their bottom endpoints. Now, we can define a uniquely defined edge for each boundary. It is the downward edge on the boundary that is smallest in the lexicographical order. The edge is called the **canonical edge** of the boundary.

A binary image shown in Figure 2 contains two connected components C_1, and C_2. The component C_1 has two holes defined by two internal boundaries while the component C_2 has no hole (recall that black pixels are connected under the 8-connectivity). There are four boundaries in total, and two among them are external. The canonical edges are depicted by bold arrows in the figure. The lexicographical order of the four canonical edges is $e_1 \leq e_2 \leq e_3 \leq e_4$.

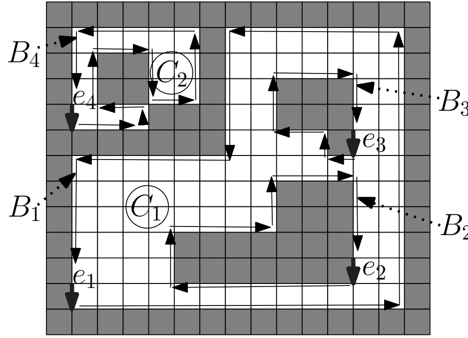

Fig. 2: Example of a binary matrix containing two connected components C_1 and C_2. The component C_1 contains two holes while C_2 has none. Bold arrows are canonical edges of the four boundaries.

Lemma 1 (Boundary Ordering Lemma) *Let $(B_1, B_2, \ldots, B_m)$ be a sequence of all boundaries (internal and external) in a binary image, which are sorted in the lexicographical order of their canonical edges. Then, the row major raster scan hits those canonical edges in the same order. In particular, for component C_i the canonical edge of the external boundary for C_i precedes that of any internal boundary of C_i.*

Proof See Figure 3. The canonical edge of a boundary is defined as the lowest (and leftmost) downward edge on the boundary. We move to the left from the canonical edge e of an internal boundary B_i until we encounter a downward boundary edge e'. If e' is an edge of another internal boundary B_j then we have $e_c(B_j) < e_c(B_i)$ since $e_c(B_j) \leq e' < e = e_c(B_i)$. Thus, the edge $e_c(B_j)$ must have been found before we find the canonical edge $e_c(B_i)$. This proves the first half of the lemma. The latter half of the lemma is immediate from the fact that any internal boundary of a component must be enclosed by an external boundary since any internal boundary cannot intersect any external boundary. □

3. Problem Statement

With these definitions, we consider the following two problems related to connected components.

Problem 1: Given a binary image G_n with n pixels, find the number of connected components in G_n.

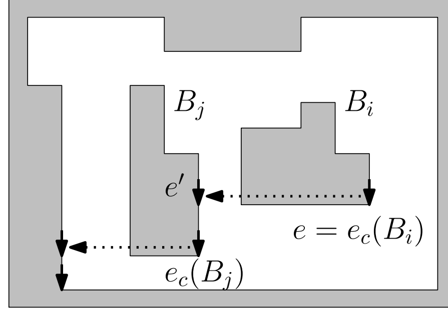

Fig. 3: The canonical edge of an external boundary precedes any canonical edge of an internal boundary of the same connected component.

Problem 2: Given a binary image G_n with n pixels, label each white pixel by the index of a connected component to which the pixel belongs.

Our interest here is to design algorithms without using any extra array, or in-place algorithms for the problems.

[**Basic assumption on work space**]

Input is a 0-1 array with n elements in total. Output of Problem 2 is given on the array. Since the maximum index can be $O(n)$, each array element must have $O(\log n)$ bits. We also assume the same amount of work space for Problem 1.

4. Known Algorithms

The connected components labeling is one of the most fundamental problems in digital image processing and there are a number of results on this topics. A recent book by Klette and Rosenfeld [2] nicely surveys geometric methods for digital picture analysis. The book includes two different algorithms for connected components labeling. One of them is a simple method in which a label is propagated from an initial pixel using a stack or queue. The algorithm in the book is described using a function to put a label L_k to white pixels starting from a pixel p as follows (just as it is in the book):

FILL Algorithm for Connected Component Labeling

1. Label p with L_k
2. Put p into a stack and put a temporary label to p.
3. If the stack is empty, stop.
4. Pop r out of the stack.
5. Label with L_k all pixels $q \in A(r)$ that have value 1 and have not yet been labeled, and put these qs into the stack.
6. Go to Step 3.

In the algorithm, $A(r)$ denotes a set of white pixels in the neighborhood of a pixel r, $N_4(r)$ or $N_8(r)$.

The time complexity of the algorithm is linear in the number of pixels since any pixel is never pushed into the stack more than once. What about the largest possible size of the stack? Suppose we have a binary image shown to the left in Figure 4, which repeats the same pattern every sixth row, that is, row i is repeated in row $i + 6$. If we start from the lower left corner of the white component and we check the four neighbors in the order of East, North, West, and South, the algorithm works as follows:

(1) We first move to East until it reaches the rightmost column while putting Northern neighbors into a stack, and
(2) go upward by four rows (since Eastern neighbors are black) and then we are forced to move to West until the left end while putting Southern neighbors into the stack, and
(3) go upward again by two rows.
Then, we have the same situation as the beginning. In this example, almost half of the white elements are stored in the stack when we reach the upper left corner. This means that we need a work space of size $\Omega(n)$ for a binary image with n pixels. Of course, if we use a different ordering of the four directions, then we can reduce the maximum size of the stack for this example, but it is also possible to design a worst case example for the new order.

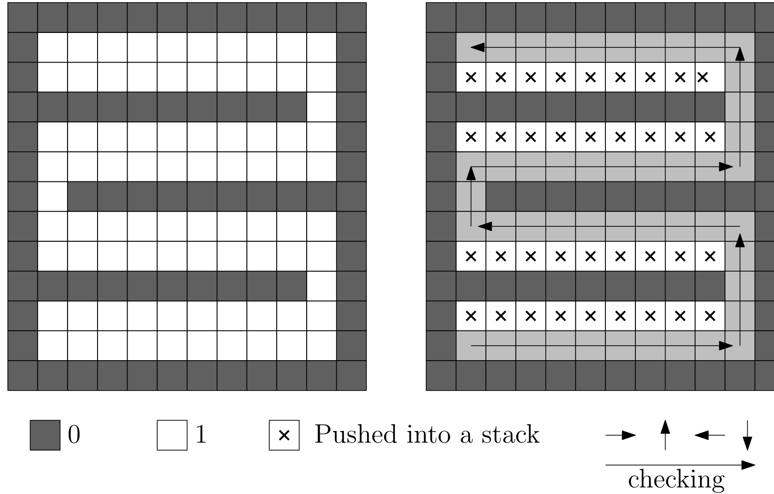

Fig. 4: A worst case example of a binary image with n pixels for which the stack size is $\Omega(n)$. Neighbors are checked in the order of North, East, West, and South. Starting from the lower left corner labels are propagated to the right and then upward, and so on. Pixels with checks are stored in the stack for later processing.

How about using a queue? In practice a queue of size $O(\sqrt{n})$ may be enough. Unfortunately, we can design bad examples for which the queue size is $\Theta(n)$. Initially we have a path of white pixels connecting the top middle pixel to the center pixel of the image which is one pixel wide. Then, we extend paths of white pixels in many different directions. All of those paths are one pixel wide.

Starting from the center pixel, we create five paths, three of them extend in the directions of South, East and West, and two of them extend to the North after leaving the center horizontally by two pixels. All of those paths have length $\sqrt{n}/4$. Figure 5 illustrates those paths. The endpoints for the first three paths are called **F-endpoints** (depicted by circles in the figure) while the other two are **P-endpoints** (depicted by squares). At each F-endpoint we create five paths of white pixels with three F-endpoints and two P-endpoints in the same manner at the half distance $\sqrt{n}/8$. At each P-endpoint we create three paths with one F-endpoint in the orthogonal direction and two P-endpoints in the opposite directions along the previous path again in the same distance $\sqrt{n}/8$. The results after the second expansion are shown in Figure 6.

In this way we repeat expansions until the length of the paths becomes some small constant. Since every time we halve the length of those paths, the number of iterations is $O(\log n)$.

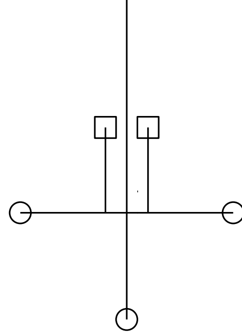

Fig. 5: First expansion from the center. Three F-endpoints (circles) and two P-endpoints (squares) are generated in the same distance from the center.

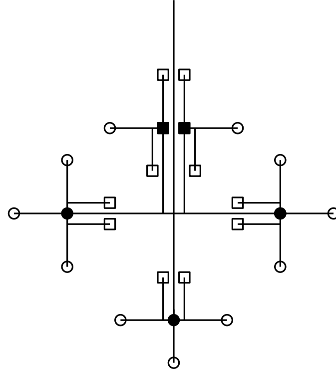

Fig. 6: Second expansion from the endpoints generated in the first iteration, which are indicated by black circles and black squares.

Figure 7 shows an example of the worst cases for queue.

Suppose in the k-th iteration we have F_k F-endpoints and P_k P-endpoints. Then, one F-endpoint is extended to three F-endpoints and two P-endpoints in the same distance from center pixel. One F-endpoint is extended to one F-endpoint and two P-endpoints. Thus, we have

$$F_{k+1} = 3F_k + P_k, \text{ and } P_{k+1} = 2F_k + 2P_k. \tag{1}$$

We first observe that $F_k \geq P_k$ for any k since

$$F_k - P_k = 3F_{k-1} + P_{k-1} - (2F_{k-1} + 2P_{k-1}) = F_{k-1} - P_{k-1} = \cdots = F_1 - P_1 = 1.$$

Therefore, we have

$$P_k = 2F_{k-1} + 2P_{k-1} \geq 4P_{k-1} \geq \cdots \geq 4^{k-1}P_1.$$

By the construction the number of iterations is $\Theta(\log n)$. The total number of F-endpoints is obviously at most n, and thus we have

$$n \geq F_k \geq P_k = \Omega(n).$$

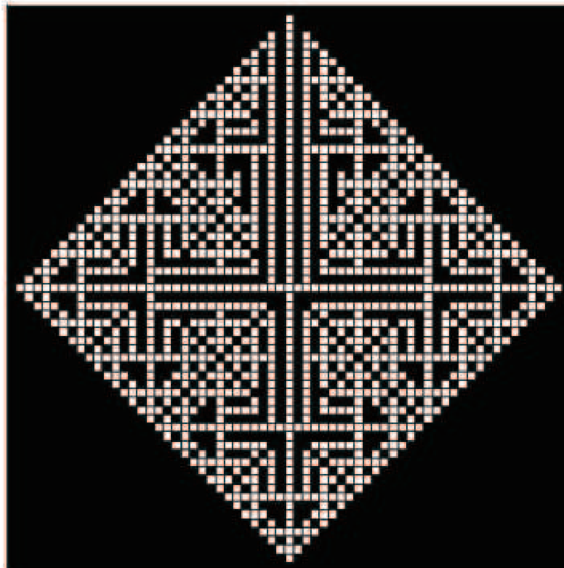

Fig. 7: A worst example of a binary image with n pixels for an algorithm using a queue for which the queue size is $\Theta(n)$.

4.1 Algorithm Using Union-Find Tree

Another famous algorithm is the one known as the Rosenfeld-Pfaltz labeling algorithm [3]. The book [2] describes the algorithm as follows (just as it is in the book).

Rosenfeld-Pfaltz Algorithm for Connected Components Labeling

1. In the first scan, propagate the labels until the end of the picture is reached.
   1.1 If the current pixel p is adjacent to one or more previously visited pixels that all have the same label, assign that label to p, and continue the scan.
   1.2 If the current pixel p is adjacent to two or more previously visited pixels that have different labels, assign the smallest of those labels(e.g., L) to p, enter the other labels into the table as being equivalent to L, and continue the scan.
   1.3 Otherwise, assign to p a label that has not yet been used and continue the scan.
2. Determine the equivalence classes of the labels by computing the transitive closure of the equivalent pairs of labels detected in Step 1. Choose one label from each equivalence class as its pivot.
3. Scan the picture a second time, and replace every label with the pivot of its equivalence class.

The algorithm is more sophisticated than the previous one in that it uses a data structure for union and find operations. In our case the algorithm executes at most n *union* operations to unite two equivalence classes and at most n *find* operations to find the smallest label. A sequence of $O(n)$ *union* and *find* operations can be done in $O(n\alpha(n))$ time using a data structure called *union-find tree* [6]. Here $\alpha(n)$ is a very slowly growing function, known as a functional inverse of Ackermann's function. So, the running time is almost linear in the image size, and indeed it runs in linear time in practice. The size of the data structure is, unfortunately, $O(n)$. Thus, theoretically speaking, this is not an improvement of the FILL algorithm.

Theorem 1 *Given a binary image with n pixels, the FILL algorithm and the Rosenfeld Pfaltz algorithm run in $O(n)$ and $O(n\alpha(n,n))$ time, respectively, using $O(n)$ work space in the worst case.*

Proof The proof of the theorem depends on the following famous theorem[6].

Theorem 2 (Tarjan and van Leeuwen 1984) *The algorithms with either union rule combined with either compaction rule run in $O(n + m\alpha(m + n, n))$ time on a sequence of at most $m - 1$ unions and m finds.*

In our case, we have at most n *union*s and $2n$ *find*s, and thus it takes $O(n + n\alpha(n,n)) = O(n\alpha(n,n))$ time. □

Our goal here is to present an algorithm which runs in $O(n)$ time without using any extra array.

5. Algorithm 1 for Counting Connected Components

To ease our argument we implicitly assume that a given binary image G_n is surrounded by black pixels, that is, $G(x,0) = G(x,\sqrt{n}+1) = 0$ for each $x = 0, 1, \ldots, \sqrt{n}+1$ and $G(0,y) = G(\sqrt{n}+1, y) = 0$ for each $y = 0, 1, \ldots, \sqrt{n}+1$.

A key operation of our algorithm for counting connected components is a *boundary labeling* on a boundary. Given an edge e on a boundary and a new label $\lambda \geq 2$, we follow the boundary from e. For each downward edge e' on the boundary, we put λ at the pixel adjacent to its left (to the East) of e' (which used to be white). Once we apply the operation to a boundary, each downward edge on the boundary corresponds to the transition from 0 to $\lambda \geq 2$. Recall that the downward boundary edge used to be a $0 \to 1$ edge.

Boundary-Labeling(e, λ)

```
e_s = e.
do{
    if e is downward then{
        Let p the pixel lying to the left (East) of e.
        Replace the pixel value of p by λ.
    }
    e = the next edge on the boundary.
} while(e! = e_s)
```

We scan the given image G_n in the row-major raster order from bottom to top. Whenever we find a transition from 0 to 1 in the scan, which induces a downward boundary edge, we traverse the boundary from the edge. Note that the first downward edge we find in the scan must be the canonical edge of an external boundary, say B_1, by Lemma 1. Then we perform the Boundary-Labeling($e_c(B_1), 2$) using a parameter 2. Then, we continue the raster scan to find the next downward boundary edge between 0 and 1. Note that all the downward edges on B_1 are neglected in the scan since their left pixels have been changed to 2 by the boundary labeling operation on B_1.

Now there are two cases on the next downward boundary edge e. If it is on an external boundary, then we perform Boundary-Labeling($e, 2$) again for the boundary. On the other

hand, if it belongs to an internal boundary, we perform Boundary-Labeling($e, 3$) using a different parameter 3. Thanks to the boundary labeling operation, no boundary is followed more than once, and boundaries are followed in the order of their canonical edges.

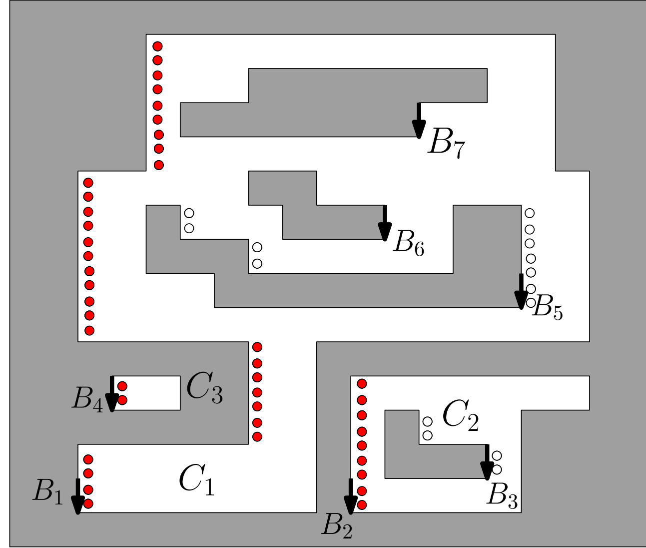

Fig. 8: Boundary labels after traversing the boundary B_5. Each downward edge is labeled by 2 (filled circle) for an external boundary and 3 (empty circle) for internal one.

Figure 8 illustrates how pixel values are changed by boundary labeling. It shows the image after finishing the boundary labeling for the boundaries $B_1, \ldots, B_5$ and we are now going to execute the boundary labeling on B_6. Filled circles indicate pixel values 2 for external boundaries and empty circles indicate pixel values 3 for internal boundaries.

In the argument above we have implicitly assumed that it is possible to determine whether a boundary followed by a downward edge found in the raster scan is external or internal. How can we determine it without following the boundary? Surprisingly it is quite easy!!

Let e be a downward boundary edge found in the raster scan. By the definition, e's left is black (value 0) and e's right is white (value 1). What about pixels below them? There are only two cases depending on the direction of the next edge e' of e. Figure 9 illustrates two different situations around the edge e. Let b_0 and b_1 be values of pixels below the black pixel and white pixel, respectively. Since e is the edge found by the raster scan, the next edge e' cannot be downward. The reason is as follows. Suppose the next edge e' is the downward for contradiction. The right pixel b_1 of e' cannot have label 2 or 3 because it belongs to a boundary which has not been followed before e. Thus, the right pixel must be 1, i.e., $b_1 = 1$. But, then the next edge e' must have been found in the raster scan before e, which is a contradiction. In the same reason, b_1 (x in the figure) cannot be 0.

In the first case (a) in the figure the next edge e' is to the left. Since e is the canonical edge, there is no downward edge below e on the boundary containing e. Therefore, the boundary must be clockwisely oriented, which means that the boundary is internal.

In the second case (b) in the figure, the next edge e' is to the right. By the same reason the boundary must be counter-clockwisely oriented, that is, it must be external.

In this way, we can determine the orientation of a boundary based on local information around the canonical edge of the boundary. Thus, it is easy to count the number of connected components. The algorithm is described as follows using a pseudocode.

Algorithm 1 for Counting Connected Components

$C = 0$.
for $y = 1$ to $\sqrt{n}$
for $x = 1$ to $\sqrt{n}$ {
  if $G(x, y) = 0$ and $G(x + 1, y) = 1$ then // external boundary
    Let e be the edge between the two pixels (x, y) and $(x + 1, y)$
    if $G(x + 1, y - 1) = 0$ then {
      $C = C + 1$.
      Perform Boundary-Labeling$(e, 2)$ by following the boundary starting from e.
    } else { // internal boundary
      Perform Boundary-Labeling$(e, 3)$ by following the boundary starting from e.
}
Output the value of C as the number of connected components.

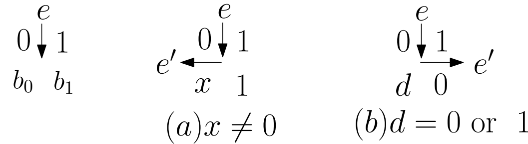

Fig. 9: Two situations around the canonical edge (downward boundary edge).

Theorem 3 *Given a binary image with n pixels, Algorithm 1 counts the number of connected components in* $\Theta(n)$ *time without using any extra array (using exactly two bits for each pixel).*

Proof We have already proved the correctness of the algorithm. Algorithm 1 examines each pixel at most twice, once in a raster scan, and at most once to follow boundaries. The labels we put are 2-bits integers $0, 1, 2, 3$. □

6. Algorithm 2 for Connected Components Labeling

Next we consider Problem 2 of labeling pixels by component indices. We show how to adapt Algorithm 1 to solve Problem 2. We have put 2 or 3 to the right of each downward edge depending on whether it is on an external or internal boundary, respectively. Our goal here is to put the index of a component at each pixel in the component. It is rather easy to adapt Algorithm 1 to put the index of a component to the right of each downward edge on the external boundary of the component since we can easily detect the canonical edge of each external boundary.

How should we handle internal boundaries? A key observation is again Boundary Ordering Lemma.

The algorithm is described as follows: Following the row major raster order we scan pixels from left to right. Whenever we find a transition from 0 to 1, which corresponds to a downward edge, we follow the boundary while putting a label to the right of each downward edge. The vertical edge must be on a canonical edge of some boundary. If its lower right pixel is black, then it must be an external boundary. Otherwise, it is internal. Figure 10 contains two external boundaries and three internal boundaries. Their canonical edges

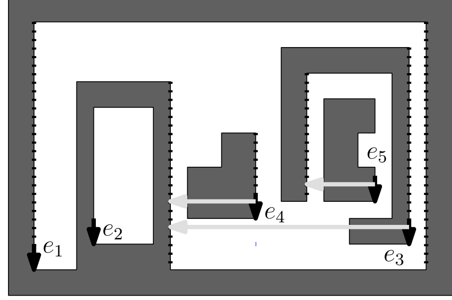

Fig. 10: An example of a binary image having two components and three holes. Their canonical edges are ordered as $(e_1, e_2, e_3, e_4, e_5)$. When e_3 is found, we can find the index of the component containing it by walking to the left until it hits a downward edge. Since each downward edge on the external boundary starting from e_1 is labeled by 2, we get the value at the downward edge. Then, this label is put along the boundary from e_3, which is later used when we find e_5. In the figure the downward edges are shown by bold dotted lines.

$e_1, \ldots, e_5$ are indicated by bold arrows. Their lexicographical order is $(e_1, e_2, e_3, e_4, e_5)$. So, if we perform a raster scan, we find those edges in this order.

When we find an external boundary, we create a new label $\lambda \geq 2$ and put the label to the Eastern pixel of every downward edge on the boundary by following it. On the other hand, when the canonical edge we find is internal one, then we need a label assigned to the component. For the purpose we walk to the West within the component (while pixel values are 1) until we encounter a black pixel p. The pixel q just before the black pixel p has the label we want. The reason is as follows. The transition from the pixel q to the black pixel p corresponds to a downward edge. Due to the raster order, the edge must have been scanned before. Thus, by induction we can assume that it must have been labeled by the index of the connected component. Therefore, if we keep the most recent index (label) and use it for every canonical edge of an internal boundary, the correct index can be propagated. Once every downward edge is properly handled and correct indices are put to their Eastern pixels, it suffices to propagate them to the East as far as pixel values are 1s.

Now, we have Algorithm 2 as follows.

Algorithm 2 for Connected Components Labeling

$C = 0$.
for $y = 1$ to $\sqrt{n}$
for $x = 1$ to $\sqrt{n}$ {
  if $G(x, y) > 1$ then $D = G(x, y)$.
  if $G(x, y) = 0$ and $G(x + 1, y) = 1$ then // external boundary
    Let e be the edge between the two pixels (x, y) and $(x + 1, y)$
    if $G(x + 1, y - 1) = 0$ then {
      $C = C + 1$.
      Perform Boundary-Labeling$(e, C + 1)$ by following the boundary starting from e.
    } else { // internal boundary
      Perform Boundary-Labeling(e, D) by following the boundary starting from e.
}
for $y = 1$ to $\sqrt{n}$
for $x = 1$ to $\sqrt{n}$ {
  if $G(x - 1, y) > 1$ and $G(x, y) = 1$ then $G(x, y) = G(x - 1, y)$.

// Propagate component index to the right.

The algorithm above scans an image twice, once for boundary labeling and once for propagating component indices. We can combine the two scans into one as follows.

Algorithm 2' for Connected Components Labeling

$C = 0$.
for $y = 1$ to $\sqrt{n}$
for $x = 1$ to $\sqrt{n}$ {
  if $G(x-1, y) > 1$ and $G(x, y) = 1$ then $D = G(x, y) = G(x-1, y)$.
    // Propagate component index to the right.
  if $G(x, y) = 0$ and $G(x+1, y) = 1$ then // external boundary
    Let e be the edge between the two pixels (x, y) and $(x+1, y)$
    if $G(x+1, y-1) = 0$ then {
      $C = C + 1$.
      Perform Boundary-Labeling$(e, C+1)$ by following the boundary starting from e.
    } else { // internal boundary
      Perform Boundary-Labeling(e, D) by following the boundary starting from e.
}

Theorem 4 *Given a binary image with n pixels, Algorithm 2' labels each white pixel by the index of a component to which the pixel belongs in $\Theta(n)$ time without using any extra array other than an array for an input image.*

7. Extension to 8-Connectivity

So far we have assumed 4-connectivity. In this section we adapt our algorithms for 8-connectivity. The rules for following boundaries must be modified as shown in Figure 11. We also need to modify the rule for determining the orientation of a boundary. A new rule is illustrated in Figure 12.

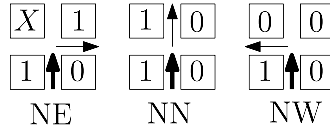

Fig. 11: Rules for following a boundary of a 8-connected component. The pixel indicated by the symbol X is either 0 or 1.

Fig. 12: Two situations around the canonical edge (downward boundary edge) for 8-connectivity.

Theorem 5 *Algorithms 1 and 2' can be adapted for 8-connected components following new rules depicted in Figure 11 and Figure 12.*

We omit a formal proof although it is not trivial to show that this change suffices to label pixels in 8-connected components.

8. Conclusions

In this paper we have presented two in-place algorithms, one for counting the number of connected components and the other for connected components labeling. Both of them run in linear time and need no extra array other than an input array. An interesting open question is how fast we can count the number of connected components when only one bit is available for each pixel and no other extra array is allowed in addition to the input array.

References

[1] Z. Galil and G.F. Italiano: "Data structures and algorithms for disjoint set union problems," ACM Comput.Surv., 23(3), pp.319-344, 1991.

[2] R. Klette and A. Rosenfeld: "Digital Geometry: Geometric Methods for Digital Picture Analysis," Elsevier, 2004.

[3] A. Rosenfeld and J.L. Pfaltz: "Sequential Operations in Digital Picture Processing," J. ACM, 13, pp. 471-494, 1966.

[4] C. Ronse and P.A. Devijver: "Connected Components in Binary Images: The Detection Problem," Wiley, New York, 1984.

[5] "LEDA: A library for Efficient Data Types and Algorithms," Algorithmic Solutions Software GmbH, Germany.

[6] R.E. Tarjan and J. van Leeuwen: "Worst-case analysis of set union algorithms," J. ACM, 31(2), pp.245-281, 1984.