Pattern Anal Applic (2009) 12:117–135
DOI 10.1007/s10044-008-0109-y

THEORETICAL ADVANCES

Optimizing two-pass connected-component labeling algorithms

Kesheng Wu · Ekow Otoo · Kenji Suzuki

Received: 5 January 2007 / Accepted: 23 December 2007 / Published online: 4 March 2008

Abstract We present two optimization strategies to improve connected-component labeling algorithms. Taking together, they form an efficient two-pass labeling algorithm that is fast and theoretically optimal. The first optimization strategy reduces the number of neighboring pixels accessed through the use of a decision tree, and the second one streamlines the union-find algorithms used to track equivalent labels. We show that the first strategy reduces the average number of neighbors accessed by a factor of about 2. We prove our streamlined union-find algorithms have the same theoretical optimality as the more sophisticated ones in literature. This result generalizes an earlier one on using union-find in labeling algorithms by Fiorio and Gustedt (Theor Comput Sci 154(2):165–181, 1996). In tests, the new union-find algorithms improve a labeling algorithm by a factor of 4 or more. Through analyses and experiments, we demonstrate that our new two-pass labeling algorithm scales linearly with the number of pixels in the image, which is optimal in computational complexity theory. Furthermore, the new labeling algorithm outperforms the published labeling algorithms irrespective of test platforms. In comparing with the fastest known labeling algorithm for two-dimensional (2D) binary images called contour tracing algorithm, our new labeling algorithm is up to ten times faster than the contour tracing program distributed by the original authors.

Keywords Connected-component labeling · Optimization · Union-find algorithm · Decision tree · Equivalence relation

K. Wu (✉) · E. Otoo
Lawrence Berkeley National Laboratory,
University of California, Berkeley, CA, USA
e-mail: KWu@lbl.gov

E. Otoo
e-mail: EJOtoo@lbl.gov

K. Suzuki
Department of Radiology, The University of Chicago,
Chicago, IL, USA
e-mail: suzuki@uchicago.edu

1 Introduction

Connected-component labeling is a procedure for assigning a unique label to each object (or a connected component) in an image [7, 17, 34, 36]. Because these labels are key for other analytical procedures, connected-component labeling is an indispensable part of most applications in pattern recognition and computer vision, such as character recognition [6, 9, 23, 37]. In many cases, it is also one of the most time-consuming tasks among other pattern-recognition algorithms [4]. Therefore, connected-component labeling continues to be an active area of research [1, 9, 12, 21, 22, 24, 29, 32, 38, 43, 47]. In this paper, we present two optimization strategies to improve labeling algorithms. Through extensive testing, we demonstrate these optimization strategies greatly enhance the labeling algorithms on all machines tested.

To illustrate the new optimization strategies, we consider the problem of labeling binary images stored in two-dimensional (2D) arrays. These images are typically the output from another image-processing step, such as segmentation [20, 35, 44]. A binary image contains two types of pixels: *object pixel* and *background pixel*. The connected-component labeling problem is to assign a label to each object pixel so that connected (or neighboring) object pixels have the same label. There are two common ways of defining connectedness for a 2D image: 4-connectedness

and 8-connectedness [33]. In this paper, we use the 8-connectedness as illustrated in Fig. 1a. Our optimization strategies can be applied to higher dimensional images, but the new labeling algorithm combining the two optimization strategies even outperforms the fastest labeling algorithms designed specifically for 2D images.

1.1 Background

There are a number of different approaches to labeling-connected components. The simplest approach repeatedly scans the image to determine appropriate labels until no further changes can be made to the assigned labels [34]. A label assigned to an object pixel before the final assignment is called a *provisional label*. For a 2D image, a *forward scan* assigns labels to pixels from left to right and top to bottom. A *backward scan* assigns labels to pixels from right to left and bottom to top. Each time a pixel is scanned, its neighbors in the scan mask are examined to determine an appropriate label for the current pixel. In the illustration shown in Fig. 1, the current pixel being examined is marked as **e** and the four neighbors in the scan masks are designated as **a**, **b**, **c** and **d**. If there is no object pixel in the scan mask, the current pixel receives a new provisional label. On the other hand, if there are object pixels in the scan mask, the provisional labels of the neighbors are considered equivalent, a representative label is selected to represent all equivalent labels, and the current object pixel is assigned this representative label. A common strategy for selecting a representative is to use the smallest label. A more sophisticated labeling approach may have a separate data structure for storing the equivalence information or a different strategy to select a representative of the equivalent labels. Without considering the issues such as image formats or parallelization, we divide the labeling algorithms into three broad categories: multi-pass algorithms, two-pass algorithms and one-pass algorithms.

1. **Multi-pass algorithms** ([7, 19, 33, 36, 38]): The basic labeling algorithm described in the preceding paragraph is the best known example of this group. They may require a large number of passes before reaching the final labels. Given an image with p pixels, a labeling algorithm is said to be optimal if it uses $O(p)$ time. Because the number of passes over the image depends on the content of the image, multi-pass algorithms are not considered to be optimal.
   To control the number of passes, one may alternate the direction of scans or directly manipulate the equivalence information. The most efficient multi-pass algorithm we know of is that of Suzuki et al. [38]. It uses a label connection table to reduce the number of scans. In tests, this algorithm uses not more than four scans and was observed to be much faster than many well-known algorithms [38]. In later discussions, we refer to this algorithm as scan plus connection table, or *SCT*.
2. **Two-pass algorithms** ([18, 27, 28, 30]): Many algorithms in this group operate in three distinct phases.
   (a) *Scanning phase:* In this phase, the image is scanned once to assign provisional labels to object pixels, and to record the equivalence information among provisional labels.
   (b) *Analysis phase:* This phase analyzes the label equivalence information to determine the final labels.
   (c) *Labeling phase:* This third phase assigns final labels to object pixels using a second pass through the image.

   Depending on the data structure used for representing the equivalence information, the analysis phase may be integrated into the scanning phase or the labeling phase. One of the most efficient data structures for representing the equivalence information is the *union-find* data structure [10, 12]. Because the operations on the union-find data structure are very simple, one expects the analysis phase and the labeling phase to take less time than the scanning phase. Indeed, many two-pass algorithms have the theoretically optimal time complexity of $O(p)$, where p is the number of pixels in an image. In this paper, we use an algorithm by Fiorio and Gustedt [12] as the representative of this group. Because the equivalence information is stored in a union-find data structure, we refer to this algorithm as scan plus union-find, or *SUF*.
3. **One-pass algorithms** ([7, 9, 22, 33, 42]): An algorithm in this group scans the image to find an unlabeled object pixel and then assigns the same label to all connected object pixels. By definition, one-pass algorithms go through the image only once, but typically with an irregular access pattern. For example, an approach proposed by Udupa and Ajjanagadde [42] avoids the second pass by tracing boundary faces with containment trees. Similar to other one-pass algorithms, this approach is recursive in nature. In practice, the irregular access of pixels leads to slow performance.

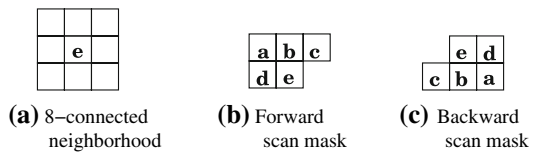


Fig. 1 The masks and the neighborhood of pixel **e**. Notice that all the pixels in the forward and backward scan masks are in the neighborhood of pixel **b**

To overcome this problem, one may limit the scope of this irregular accesses. Recently, Hu et al. [22] demonstrated that it was possible to outperform SCT with such a method. For 2D images that we plan to use for illustrations, the most efficient one-pass algorithm is the contour tracing (CT) algorithm by Chang et al. [9]. Because of this, we choose to use CT as the representative of one-pass algorithms. The implementation of CT used in later tests is distributed by the original authors of the algorithm.[1] Note that CT is also the most efficient sequential (i.e., not parallel) labeling algorithm in literature.

1.2 Overview of key points

Generally, one expects a one-pass algorithm to be faster than a two-pass algorithm and a two-pass algorithm in turn to be faster than a multi-pass algorithm. However, this is not always the case, as demonstrated in [38]. One reason that a multi-pass algorithm like SCT could be faster than a two-pass algorithm is that SCT performs only sequential and local memory accesses, whereas a two-pass algorithm needs random memory accesses to maintain and update the union-find data structure. The sequential memory accesses are much better supported on modern computers than are random memory accesses. on the basis of this observation, our optimization strategies seek to minimize the number of random memory accesses. We show the usefulness of each of the optimization strategies with both analyses and timing measurements. By combining the optimization strategies, we aim at producing a two-pass algorithm that is more efficient than the fastest known algorithm, namely, the contour tracing algorithm.

Our first optimization strategy minimizes the number of neighbors visited during a scan and therefore reduces the number of memory accesses. Assuming that the current pixel is designated **e** (see Fig. 1), existing scanning procedures examine all four neighbors **a**, **b**, **c**, and **d**. With our optimization, if **b** is an object pixel, the other three pixels are not accessed. This is possible because all other three pixels are neighbors of **b**. When the label equivalence information is recorded, it is possible to derive the correct label of **b** (and therefore that of **e**) later. If **b** is a background pixel, the order to examine the other pixels is given as a decision tree. We are not aware of any existing labeling algorithms that reduce memory accesses in this manner. The SCT approach proposed by Suzuki et al. [38] has some resemblance to ours; however, theirs reduces the accesses to the label array and the label connection table, but not the image pixels. Later, we show that using a decision tree can significantly speed up SCT.

Our second optimization strategy simplifies the data structure and the algorithms used to solve the union-find problem [5, 8, 10, 11, 13, 14]. Because union-find involves relatively simple operations, the time spent on union-find was expected to be a small fraction of a two-pass algorithm. However, this is not the case (see [13]). This has motivated a number of research efforts to find more efficient data structures to implement union-find [13, 15, 39, 40]. Our union-find data structure is implemented with a single array. Even though this particular implementation strategy has been suggested before [11], using it effectively in a connected component labeling algorithm is new.

Although the basic versions of union-find algorithms are simple, to achieve the best performance, they need to incorporate a number of well-known optimization strategies, which can significantly complicate the implementation. In their work, on the scan plus union-find algorithm, Fiorio and Gustedt [12] found that by using a relative simple optimization strategy called path-compression in union-find algorithms along with some extra flattening of the active union-find trees (defined in Sect. 4.2), they can achieve the optimal $O(p)$ performance. We find it possible to achieve the same optimal performance without the extra flattening operations. This simplifies the use of union-find in labeling algorithms and also improves their overall performance.

Another key contribution of this paper is the development of a two-pass labeling algorithm that combines the above two optimization strategies. This algorithm generates consecutive final labels, which are preferred over nonconsecutive ones in most applications. We analyze this new labeling algorithm for its correctness, its worst-case time complexity, and its average time complexity. In addition, we conduct extensive timing measurements on three different platforms to ensure that the performance advantages we report are not due to any particularity of a specific hardware.

Previously, we have published a limited performance study on the optimization strategies [45]. Since then, an independent study has confirmed their effectiveness [47], which makes it more interesting to carefully study them and fully understand the reasons for their efficiency.

1.3 Organization

The remainder of this paper is divided into six sections. The next section describes the decision tree used for minimizing the number of neighbors visited during a scan. Section 3 contains the description of the new union-find solution. In Sect. 4, we analyze the correctness of the optimization strategies, the worst-case time complexity of the new labeling algorithm, and it expected average

[1] An implementation of the Contour Tracing algorithm distributed by the original authors of the algorithm is available from http://www.iis.sinica.edu.tw/~fchang/03src.html.

execution time on random images. In Sect. 5, we present timing results that confirm the expected performance advantages of the two optimization strategies, and compare the perform of the new labeling algorithm against the fastest known labeling algorithm. A summary and discussion on future work are given in Sect. 6. We conclude this paper with a section discussing the originality and contribution.

2 Minimizing scan cost

In this section, we briefly describe the generic scanning procedure used by most connected-component labeling algorithms, and then a decision tree to minimize the cost of such a scanning procedure. To make the description concrete, we apply this optimization strategy to the scan plus connection table (SCT) algorithm of Suzuki et al. [38]. To use the decision tree in a scanning procedure, one needs to maintain enough label equivalence information. The connection table used in SCT is a minimalistic data structure that satisfies this requirement and produces the correct final labels.

2.1 The basic scanning procedure

Let I denote the 2D array representing an image. A pixel is a *background pixel* if $I[i, j] = 0$, and an *object pixel* if $I[i, j] = 1$.[2] We use an array L of the same size and shape as I for storing the labels. In our implementation of the labeling algorithms, we use one array to hold both I and L. However, for clarity, we will continue to describe them as two separate arrays. The problem of connected-component labeling is to fill the array L with (integer) labels so that the neighboring object pixels have the same label. We name the pixel in the scan mask (illustrated in Fig. 1) as **a**, **b**, **c**, **d** and **e**, and also use the same letters in place of their (i, j) coordinates in the following discussion. With this notation, $L[\mathbf{e}]$ denotes the label of the current pixel, $I[\mathbf{b}]$ denotes the pixel value of the neighbor directly above **e** in the forward scan mask, and so on. Let l be an integer variable initialized to 1. The assignment of a provisional label for **e** during the first scan can be expressed as follows ($i \in (\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$):

$$L[\mathbf{e}] \leftarrow \begin{cases} 0, & I[\mathbf{e}] = 0, \\ l, (l \leftarrow l + 1), & I[i] = 0, \forall i, \\ \min\limits_{i \mid I(i)=1} (L[i]), & \text{otherwise}. \end{cases} \tag{1}$$

[2] Note that we have made an arbitrary choice of denoting a background pixel by 0 and an object pixel by 1; however, there are other equally valid choices [38]. It is also possible to use other types of labels than the integers used in this paper.

The above expression states that $L[\mathbf{e}]$ is assigned to 0 if $I[\mathbf{e}] = 0$. It is assigned a new label l, and l is increased by 1, if its neighbors in the scan mask are all background pixels. Otherwise, it is assigned the minimum of the provisional labels already assigned to a neighbor in the scan mask.

In later scans, labels for object pixels are modified to be the minimum labels of their neighbors, as described by the following expression (which is the last case of Eq. 1):

$$L[\mathbf{e}] \leftarrow \min_{i \in (\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d}) \mid I[i]=1} (L[i]), \quad \text{if } I[\mathbf{e}] = 1, \text{and } I[i] = 1, \exists i \in (\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}). \tag{2}$$

The above formulas can be used for both forward scan and backward scan. In principle, we can apply them to any type of scan on any image format.

The simplest multi-pass algorithm repeats the above scanning procedure until the label array L no longer changes. Initially, pixels in a connected component may receive different provisional labels. We say that these labels are equivalent, and we have chosen to use the smallest label as their representative. As labels are discovered to be equivalent, the pixels not yet scanned will take on the smallest label of its neighbors in the scan mask. Eventually, each pixel will receive the smallest provisional label assigned to any pixel in the connected component, but it may take many scans. One successful technique to reduce the number of scans is using the label connection table [38], which we describe next.

2.2 Scan plus connection table

The connection table proposed by Suzuki et al. [38] is a one-dimensional (1D) array that has as many elements as the number of provisional labels. Let T denote this connection table. In the first scan, the arrays L and T are updated as follows ($i \in (\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$):

$$L[\mathbf{e}] \leftarrow \begin{cases} 0, & I[\mathbf{e}] = 0, \\ l, (T[l] \leftarrow l, l \leftarrow l + 1), & I[i] = 0, \forall i, \\ \min\limits_{i \mid I(i)=1} (L[i]), (T[L[i]] \leftarrow & \\ L[\mathbf{e}], \forall i \mid I[i] = 1), & \text{otherwise}. \end{cases} \tag{3}$$

In the subsequent scans, we only update the labels of object pixels that have other object pixels in their scan masks. The formula for updating L and T follows from the last case in Eq. 3 and is given as

$$\begin{aligned} L[\mathbf{e}] &\leftarrow \min_{i \in (\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d}) \mid I[i]=1} (L[i]), \\ T[L[i]] &\leftarrow L[\mathbf{e}], \forall i \in (\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}) \mid I[i] = 1, \\ &\text{if } I[\mathbf{e}] = 1, \text{and } I[i] = 1, \exists i \in (\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}). \end{aligned} \tag{4}$$

Because the connection table passes the label equivalence information to all the pixels with the same provisional

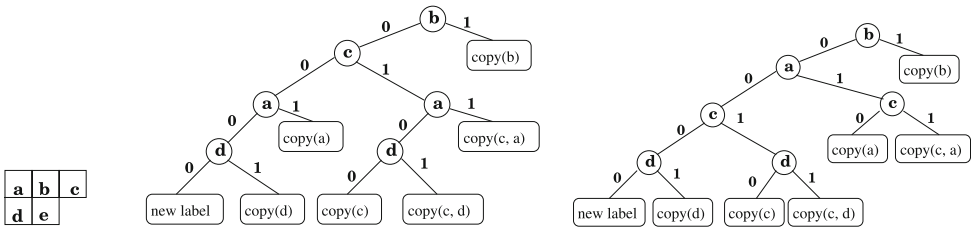


Fig. 2 The decision trees used in scanning for 8-connected neighbors. The two decision trees are equivalent. We use the first in this paper. **a** Forward scan mask, **b** decision tree 1, **c** decision tree

labels, the labels can propagate much faster than in other multi-pass algorithms. The above formulas indicate that all four neighbors in the scan masks need to be visited. We refer to this basic version of SCT as SCT-4.

2.3 Decision tree

In Fig. 2a, it is clear that all the neighbors in the scan masks are neighbors of **b**. If there is enough equivalence information for accessing the up-to-date label of **b**, then there is no need to examine the rest of the neighbors. On the basis of this observation, we present a set of decision trees that organize the scan operation in a specific order as illustrated in Fig. 2. Two equivalent trees are shown. We can produce two more equivalent trees by swapping the labels **a** and **d**. Because they are equivalent, one may use any one of them.

A decision tree is invoked to handle the case when the current pixel is an object pixel. In the first scan pass, if all neighbors in the scan mask are background pixels, a new label is generated. In subsequent scans, this branch of the decision tree performs no operation. All other branches of the decision tree deal with the case where some neighbors in the scan mask are object pixels. Using this decision process, we minimize the accesses to array I.

The decision trees presented in Fig. 2 need three functions in their leaf nodes. They are defined as follows (using the same arrays L and T defined previously):

1. The one-argument copy function, such as copy(**a**), contains one statement:
$$L[\mathbf{e}] \leftarrow T[L[\mathbf{a}]]. \tag{5}$$
2. The two-argument copy function, such as copy(**c**, **a**), contains three statements:
$$\begin{aligned} L[\mathbf{e}] &\leftarrow \min(T[L[\mathbf{c}]], T[L[\mathbf{a}]]), \\ T[L[\mathbf{c}]] &\leftarrow L[\mathbf{e}], \\ \text{and, } T[L[\mathbf{a}]] &\leftarrow L[\mathbf{e}]. \end{aligned} \tag{6}$$
3. The new label function performs the three statements below, which replicate the second case in Eq. 3.
$$L[\mathbf{e}] \leftarrow l, \quad T[l] \leftarrow l \text{ and } l \leftarrow l + 1. \tag{7}$$

The use of a decision tree minimizes the number of neighbors visited in determining a label for pixel **e**. We formalize this observation later after we have explained the concept of union find. In the following discussions, we denote the SCT algorithm that employs a decision tree as SCT-1.

3 Array-based union-find

The connection table helps SCT to propagate the label equivalence information quickly. The ultimate version of this would be to bypass all the repeated scans by directly working with the equivalence information, which leads to theoretically optimal two-pass labeling algorithms [10, 12]. Our challenge is to make the union-find algorithms simple enough so that these optimal algorithms are also fast. Our approach is to implement the union-find data structure with a single array. To provide a context for the union-find algorithms, we briefly review a two-pass labeling algorithm that uses them.

A two-pass labeling algorithm employing a union-find data structure generally starts with a scanning phase by using one of the scanning procedures described in the previous section. During the scanning phase, it also builds up the union-find data structure to record the equivalence information among the provisional labels. After the scanning phase, it analyzes the union-find data structure to determine the final label for each provisional label. This is the analysis phase, which does not access the image array. Finally, it passes through the image a second time to convert all the provisional labels into their final values. This is the labeling phase. Next, we proceed to describe the union-find problem in general and then given the details of the proposed array-based union-find data structure and algorithms.

3.1 General union-find data structure

A union-find data structure can be viewed conceptually as rooted trees, where each node of a tree is a provisional label and each edge represents the equivalence between two labels [16]. By definition, all labels in a tree are equivalent. The label associated with the root of a tree is usually chosen as the final label for all provisional labels in

the tree. We will refer to the union-find data structure and the associated algorithms simply as the union-find in the future.

There are only three operations on a union-find data structure: (1) unite two trees, (2) find the root of a given node, and (3) make a new tree with a single node. The first two operations are commonly referred to as *union* and *find*, respectively, hence the name union-find. The find operation starts from a node and follows the edges until it reaches the root of the tree. This operation returns the root label. The union operation adds an edge from the root of one tree to the root of another. The input arguments to a union operation can be two arbitrary nodes, and two find operations are needed for finding the root nodes of their respective trees. In general, the cost of a union operation is dominated by the two find operations. Therefore, an efficient find algorithm is critical to the overall efficiency of union-find operations.

A natural to represent the edges in trees is to use software pointers. In most cases, nodes of a pointer-based rooted tree are scattered randomly by the memory management system. A find operation follows the pointers to the root and traverses the memory in an unpredictable manner. This is typically slow.

A number of authors have suggested storing these rooted trees in arrays because an array resides in consecutive memory locations [3, 11, 36]. Figure 3 shows an example of such an array. Usually, the complexity of a union-find problem is defined as the cost of an arbitrary combination of m union and find operations on a union-find data structure with n nodes. Because each operation touches at least one node, the time complexity of m operations cannot be less than $O(m)$. We say that a union-find is *linear* if it has $O(m)$ time complexity. Such an approach is also said to be *optimal*. A naive approach may require $O(mn)$ time. Common optimization techniques to speed up these operations are path compression [2] and weighted union [16, 40]. Using both these techniques, union-find is nearly linear in general [39, 41]. Under some restricted settings [14, 26] or some special classes of inputs [11, 25, 46], m union and find operations can be proven to take $O(m)$ time. However, these approaches are too cumbersome to implement with arrays or not applicable in a real application.

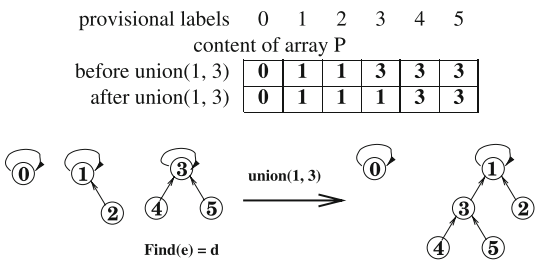


Fig. 3 An array representation of the rooted trees

Our array-based union-find approach only uses path compression. This allows us to implement all necessary algorithms as simple iterative procedures on a single array. In the next section, we prove that these algorithms lead to an optimal union-find for connected component labeling. We also use a special union rule that allows us to access elements of the array in a regular pattern, which makes the algorithms efficient.

3.2 Proposed union-find

Following examples in the literature [11, 36], we call the array that contains the equivalence information array P (short for the parent array). Array P can be filled in a way similar to that of the connection table T introduced in Sect. 2.2. In particular, every time a new provisional label is generated, array P is extended by one element denoted by the assignment $P[l] \leftarrow l$. This operation adds a new single-node tree to the union-find trees. In other cases, a reference to T[i] needs to be replaced by either a find or a union operation. Next, we describe these two operations using pseudo-code. Our implementations are in C++.[3] The two basic operations for finding the root of a tree and changing all nodes on a path to point to a new root are defined as `findRoot` and `setRoot`.

Function `findRoot(P, i)`

```
findRoot(P, i)
Input: An array P and a node i.
Output: The root node of tree of node i.
//Find the root of the tree of node i.
begin
    root ⇐ i ;
    while P[root] < root do root ⇐ P[root] ;
    return root ;
end
```

Procedure `setRoot(P, i, root)`

```
setRoot(P, i, root)
InOut: An array P.
Input: A node i of the tree.
Input: The root node of the tree of node i.
//Make all nodes in the path of node i point to root.
begin
    while P[i] < i do
        j ⇐ P[i] ; P[i] ⇐ root ; i ⇐ j ;
    end
    P[i] ⇐ root ;
end
```

[3] In C++ convention, all indices to arrays start from 0. The word *array* or *vector* in all pseudo-code segments is a short-hand of C++ STL type `std::vector<unsigned>`.

With the function findRoot and procedure setRoot, we can easily define the functions for union and find operations. We note that these two functions are iterative rather than recursive as in a pointer-based union-find implementation. In function findRoot, the variable root takes on a sequence of values. This sequence forms a path from the starting node i to the root of the tree. This path is known as a *find path*. The procedure setRoot changes all nodes on the find path to point directly to the specified new root. This operation is the path compression.

Function find(*P*, *i*)

```
find(P, i)
InOut: An array P.
Input: A node i of tree of node i.
Output: The root node of tree of node i.
//Find the root of the tree of node i
//and compress the path in the process.
begin
    root ⇐ findRoot(P, i) ;
    setRoot(P, i, root) ;
    return root ;
end
```

Function union(*P*, *i*, *j*)

```
union(P, i, j)
InOut: An array P.
Input: Two nodes i and j.
Output: The root of the united tree.
//Unite the two trees containing nodes
//i and j, and return the new root.
begin
    root ⇐ findRoot(P, i) ;
    if i ≠ j then
        rootj ⇐ findRoot(P, j) ;
        if root > rootj then root ⇐ rootj; ;
        setRoot(P, j, root); ;
    end
    setRoot(P, i, root) ;
    return root ;
end
```

3.3 New labeling algorithm

With functions find and union, we now describe how to implement the three different phases of a two-pass labeling algorithm, the scanning phase, the analysis phase, and the labeling phase. We start by describing how to modify the scanning phase to update the union-find data structure. To use the basic scanning procedure defined by Eq. 3, we perform the operations as follows ($i \in$(**a**, **b**, **c**, **d**)):

$$L[\mathbf{e}] \leftarrow \begin{cases} 0, & I[\mathbf{e}] = 0, \\ l, (P[l] \leftarrow l, l \leftarrow l+1), & I[i] = 0, \forall i, \\ \min\limits_{i|I[i]=1} (\texttt{findRoot}(P, L[i])), & \\ (\texttt{setRoot}(P, L[i], L[\mathbf{e}]), & \\ \forall i \mid I[i] = 1), & \text{otherwise.} \end{cases} \tag{8}$$

To use a decision tree in the scanning phase, we need to redefine the three functions used at the leaf nodes of the decision tree (shown in Fig. 2): the new label function, one-argument copy function and the two-argument copy function. Note that the new label function is the second case in the above equation. The one-argument copy function, copy(**a**), previously defined by Eq. 5, is simplified to be

$$L[\mathbf{e}] \leftarrow L[\mathbf{a}]. \tag{9}$$

The third function, the two-argument copy function, copy (**c**, **a**) previously defined by Eq. 6, is now simply

$$L[\mathbf{e}] \leftarrow \texttt{union}(P, L[\mathbf{c}], L[\mathbf{a}]). \tag{10}$$

The above union function always selects the root with the smaller label as the root of the combined tree, which means that the parent of a node always has a smaller label than its own label (i.e., $P[i] \leq i$), and furthermore, the root of a tree always has the smallest label in the tree. This has two important consequences: the memory access pattern in findRoot and setRoot is more predictable than using other union strategies, and we can produce consecutive final labels efficiently by using the procedure flattenL.

The procedure flattenL carries out the analysis phase of the two-pass labeling algorithm. After which, the third phase of assigning the final labels can be expressed as the following equation:

$$L[i,j] \leftarrow P[L[i,j]], \quad \forall i,j. \tag{11}$$

If there is no need for consecutive labels, one may use the procedure flatten instead of flattenL because flatten is less time consuming than flattenL.

One important characteristics of two above algorithms is that their computational complexities are not affected by the actual content of array P. No matter how the union-find trees are shaped, the costs of both flatten and flattenL are the same. Therefore, there is no need to keep the height of the union-find tree as short as possible to reduce the cost of the analysis phase. Later, we show that this is true even if other union-find data structures are used.

```
Procedure flattenL(P, size)
flattenL(P, size)
InOut: An array P.
Input: The size of the array P.
//Flatten the Union-Find tree and
//relabel the components.
begin
    k ⇐ 1 ;
    for i ⇐ 1 to size-1 do
        if P[i] < i then
            P[i] ⇐ P[P[i]] ;
        else
            P[i] ⇐ k ;  k ⇐ k + 1 ;
        end
    end
end
```

```
Procedure flatten(P, size)
flatten(P, size)
InOut: A parent array P
Input: The size of the array P
//Flatten the Union-Find tree
begin
    for i ⇐ 1 to size-1 do P[i] ⇐ P[P[i]] ;
end
```

4 Analyses of expected performance

We now show the correctness of our proposed algorithms and their expected time requirement. One of the main results of our analyses is that any two-pass algorithm using the path compression in union-find has the worst-case time complexity of $O(p)$. There is no need to flatten the union-find trees immediately after scanning each row as recommended in [12]. We use SUF (scan plus union-find) as a short-hand for any two-pass labeling algorithm and use SAUF (scan plus array-based union-find) to denote the version that uses our array-based union-find describe in the previous section.

4.1 Correctness of algorithms

The main results of this section are stated in the form of lemmas and theorems. The first two lemmas concern the union-find algorithms. Their proofs do not require explicit details of the scanning procedure. For completeness, one can assume that Eq. 8 is used for defining the scanning procedure. We then show that the use of a decision tree achieves the same result as checking all four neighbors. We conclude this subsection by showing that the use of a decision tree minimizes the number of neighbors visited during a scan.

Lemma 1 *The array P produced by the array-based union and find algorithms satisfies $P[i] \leq i$, $\forall i$.*

Proof Each element of the array $P[i]$ is initialized to i. During both union and find procedures, the value of $P[i]$ never increases. Therefore, the lemma is true. □

The procedure `flatten` can be used to produce final labels for the connected components. However, the labels may be discontinuous. For example, the array P may contain 0, 1, 2, and 4, but not 3. In many applications, consecutive labels are preferred. In these cases, one may use the procedure `flattenL` to generate consecutive labels. The following lemma formalizes this property.

Lemma 2 *Given that there are k connected components, the procedure* `flattenL` *changes array P to contain all integers between 0 and k.*

Proof Label 0 is reserved for the background pixels. If there is one connected component, we must have $P[0] = 0$ and $P[1] = 1$. Clearly, the lemma is true for $k = 1$. To prove the lemma by induction, we assume that it is true for the first i elements of array P and prove that, after executing the procedure `flattenL` for one more iteration, the lemma is true for P with $(i+1)$ elements. We observe that `flattenL` only changes one value of P in any iteration and does not go back to change any values already examined. If there are $(k-1)$ connected components represented by the first i elements of P, then $P[0{:}i\text{-}1]$ must contain the final label already, i.e., $P[0]\ldots P[i\text{-}1]$ must contain all integers between 0 and $(k-1)$. At the ith iteration, depending on the value of $P[i]$, the procedure `flattenL` may perform one of two possible actions. If $P[i] = i$, then $P[i]$ is assigned the value of variable k. In this case, there are k components and the content of $P[0]\ldots P[i]$ is between 0 and k. The correctness of the lemma is maintained. On the other hand, if $P[i] < i$, then the content of $P[P[i]]$ must be an integer less than k and a correct final label for the tree that contains node $P[i]$ and i. In this case, there are $(k-1)$ components, and the lemma is also correct. By induction, the lemma is true for any i. □

Lemma 3 *Let S_0 denote the scanning phase without a decision tree, and let S_1 denote the scanning phase with a decision tree. The connected-component labeling algorithm SUF using either S_0 or S_1 produces the same final labels.*

Proof To produce the same final labels, the scanning phase needs to ensure that each union-find tree contains all provisional labels assigned to the pixels that are connected. Because the final labels are always produced with a flattening of union-find trees, different scanning procedures

must perform the same union operations but may perform different find operations. We say that two union-find trees are equivalent if they contain the same set of provisional labels. We say that two sets of union-find trees are equivalent if each tree from one set is equivalent to exactly one tree from the other set.

With the above definitions, to prove this lemma, we need to show that S_0 and S_1 produce equivalent sets of union-find tree. To do this, we observe that they produce exactly the same trees after scanning the first row of an image and the first pixel of the second row because each union-find tree contains only a single node. To generalize this, we assume that S_0 and S_1 have produced equivalent sets of trees up to pixel **d** in the scan mask. We need to show that, after a label is assigned to **e**, the two sets of trees remain equivalent. To prove this, we show that there are only two union operations that may possibly involve two distinct trees; all remaining apparent union operations performed by S_0 are operating on a single tree and therefore are actually find operations.

If pixel **b** is an object pixel, the provisional labels assigned to all neighbors of **e** in the scan mask must be in one tree. If pixel **b** is a background pixel, pixel **c** may belong to one union-find tree, and **a** and **d** may belong to another tree. The two union operations that may involve two distinct trees must involve **c** and one of **a** or **d**. These two cases are captured by the decision trees as two invocations of the two-argument copy function. Therefore, the decision trees correctly capture the equivalence information. The union-find trees produced by S_0 and S_1 are equivalent. □

To prove that using the decision tree actually minimize the work performed in the scanning phase, we need to quantify the costs of operations. For this purpose, we count the number of accesses to pixel values $I[i]$, $i \in$(**a**, **b**, **c**, **d**). The actual number of operations is bounded by a small constant times the number of pixels accessed, where the constant depends on the details of the scanning procedure such as the union-find algorithms.

Theorem 1 *The use of a decision tree minimizes the number of pixels accessed during the scanning phase of the connected-component labeling algorithm.*

Proof Lemma 3 implies that using the decision tree in a scanning phase performs all necessary work. To prove that it actually performs the minimal amount of work, we show that any modification to the order of accessing the neighbors leads to one of the three outcomes, an equivalent decision tree, a more expensive decision tree, or an incomplete scanning phase.

In Sect. 2.3, we mentioned four equivalent decision trees. They all access pixel **b** first, and the four variations represent all possible arrangements of the other three neighbors. Without any specific information about which neighbor is more likely to be an object pixel, the four variations are equivalent in the sense that they have the same average cost overall all possible combinations of pixel values.

If we rearrange the decision tree so that we access a pixel other than **b**, it is easy to see that it leads to accessing at least two pixels before reaching a leaf node and therefore increase the cost of the scanning phase.

Inspecting a decision tree, such as the one in Fig. 2b, makes it clear that removing any node or leaf or subtree leads to incomplete scanning operation. Furthermore, we cannot remove any operations from a leaf node. More specifically, if we neglect any of the new label operation or the one-argument copy functions, we will not assign provisional labels to some pixels. The two-argument copy function is simply a union operation as shown in Eq. 10. If we remove any union operations or replace then with find operations, we neglect to record the equivalence information between provisional labels. Note that the actual cost of union-find operations is counted in the constant mentioned above, and the fact that it is actually a constant follows from Theorem 2. In terms of the number of accesses to neighboring pixels, using the decision tree indeed minimizes this cost measure. □

4.2 Worst-case complexity

Fiorio and Gustedt [12] proved that the worst-case time complexity of a two-pass algorithm with the path compression in its union-find is $O(p)$, where p is the number of pixels in the image. A key step in their approach is that they flatten the union-find trees after scanning each line of the image. Our thesis is that these flattening operations are not necessary. We just showed that the find operations can be skipped without affecting the final labels and without adding any extra work to the last two phases of the Scan plus array-based union-find (SAUF) algorithm. We next show that this is true for any union-find with path compression.

To be precise, we define the *cost of a find operation* to be the number of nodes on the find path. This definition ensures that the cost of a find operation is at least one. We define the *cost of a union operation* to be the cost of the two find operations it invokes.

Theorem 2 *Given an arbitrary union-find tree with t nodes, the total cost of executing a find operation with path compression on each node is no more than* $3t$.

Proof For convenience, let us number the nodes of the tree from 0 to $t - 1$ and assign the root of the tree to be node 0. We define the degree d_i of node i to be the number

of children of the node. In each find path, there is a starting node and the root. In all t find operations, there are t distinct starting nodes. The root node appears t times as well. In one case, the root appears also as the starting node. Altogether, the t find paths include $2t - 1$ nodes at the beginning and the end of the paths. To compute the total cost, we need to account for the nodes that appear in the middle of the find paths.

With path compression, node i can appear in the middle of a find path at most d_i times. Because the path compression ensures that all nodes on a find path point to the root directly, after appearing in the middle of find path d_i times, all children of node i must directly point to the root of the tree. The total number of nodes that appear in the middle of t find paths is $\sum d_i$. In any tree with t nodes, $\sum d_i = t - 1$. Because the root is never in the middle of any find path, the total number of nodes that appear in the middle of the find paths is actually less than $t - 1$. The total cost of t find operations is no more than $3t - 2 < 3t$. □

After the scanning phase, the union-find data structure may contain an arbitrary number of trees. However, the total number of provisional labels (i.e., the number of nodes in all trees) is not more than the number of object pixels, which, in turn, is not more than the total number of pixels p. In the most general case, the analysis phase (the second phase) of a two-pass algorithm performs a find operation (with path compression) on each provisional label. The total cost of the analysis phase then is not more than $3p$. This proves the following lemma regarding the computational complexity of the analysis phase.

Lemma 4 *The worst-case time of the analysis phase of a two-pass connected-component labeling algorithm using any union-find with pass compression is $O(p)$, where p is the number of pixels in the image being labeled.*

After flattening of a union-find tree, the steps to assign the final labels clearly costs $O(p)$ (see Eq. 11).

So far, we have analyzed the last two phases of a two-pass labeling algorithm. Now we turn our attention to the first phase, the scanning phase. We have shown (see Lemma 3) that using a decision tree produces the same final labels as using the straightforward scanning strategy, and using either scanning procedure does not change the cost of the analysis phase or the labeling phase of a two-pass algorithm. Therefore, we choose the simple scanning strategy for the analysis of the worst-case complexity of the scanning phase. This scanning procedure is defined by Eq. 8, but may use different union rules or different union-find data structures.

During a forward scan, only the labels of the pixel in the preceding line may directly affect the labels to be used. We call these labels the *active labels*. These active labels form their own union-find trees we call the *active union-find trees* or the *active trees* for short.

After scanning of the first line, each union-find tree contains a single node. The above statement is clearly true. We next examine what happens while scanning an arbitrary line i. By construction, the scanning procedure always assigns the label of a root to a pixel. What we need to show then is that, as new pixels are assigned labels, the labels used earlier either remain as roots or are connected to roots through labels used more recently. A root of a tree may become non-root only through union operations such as the following. The label assigned to pixel **d** was a root when the assignment was made. While determining a label for pixel **e**, a union involving **d** and **c** is performed and the root of the tree containing the label of **c** becomes the parent of the label of **d**. In this case, pixel **e** is assigned the label of the root of the newly united tree and the label of **d** is a child of the new root. This is the only mechanism by which a root becomes a non-root. In this process, the old label becomes a child of the new label. This process may be repeated many times, but the earlier labels always connect to the roots through other labels that have been used more recently.

Lemma 5 *The total cost of a scanning phase of a two-pass labeling algorithm (using any union-find with path compression) with flattening of active trees after scanning each line is $O(p)$.*

Proof Following from the argument above, the active labels form their own union-find trees. Therefore, the cost of flattening the active tree is proportional to the number of pixels on the line. The process to assign labels to the pixels on the next line of the image will only involve these active labels and new labels that are in single-node union-find trees. The cost of each find and union operation is bounded by a small constant. The total cost of assigning labels to the next line is again proportional to the number of pixels on the line. Overall, the total time is $O(p)$. □

If we do not flatten the active trees, we cannot account for the costs in the same way. However, we expect that the total cost of a scanning phase without the flattening of active trees to be not more than the total cost of a scanning phase with the flattening. This is because the process of flattening the active trees are simply a series of find operations on the active labels. If we do not perform these find operations explicitly, the procedure of assigning a new label may invoke them anyway.

Lemma 6 *The total cost of a scanning phase of a two-pass labeling algorithm (using any union-find with path compression) without flattening of active trees is $O(p)$.*

Proof In the process of assigning a provisional label to pixel **e**, it may perform find operations on the labels of **a**, **b**,

c, and **d**. Instead of associating the cost of these find operations with **e**, we associate the cost of each find operation with its starting pixels **a**, **b**, **c**, or **d**. This leaves a small constant cost of assigning the provisional label to be associated with pixel **e**. While labeling a 2D images, each pixel may be the starting point of up to 4 find operations. Because these find operations involve only the active trees or newly generated single-node trees, the total cost of all find operations after scanning each row is at worst proportional to the number of object pixels in the row. Therefore, the total cost of all find operations is at worst proportional to the number of object pixels. Accounting for other constant costs per pixel, the total cost of the scanning phase is $O(p)$. □

Theorem 3 *The total time required by a two-pass labeling algorithm using any union-find with path compression is $O(p)$, where p is the number of pixels in the 2D image.*

Proof A two-pass labeling algorithm can be divided into three phases: scanning phase, analysis phase, and labeling phase. Lemmas 2 and 2 show that the scanning phase takes at most $O(p)$ time with or without flattening of active trees. Lemma 2 shows that the analysis phase takes $O(p)$ time by the use of a series of find operations with path compression, or one of the simplified algorithms, `flatten` and `flattenL`. The labeling phase, as defined by Eq. 11, obviously takes $O(p)$ time. Overall, the total time is at worst $O(p)$. □

4.3 Average performance on random images

For the expected performance of the scan plus array-based union-find (SAUF) algorithm, consider a random image with n rows and m columns where each pixel has a probability q of being an object pixel. We also refer to q as the density of object pixels ($0 \leq q \leq 1$). The total number of pixels $p = mn$, of which $n_o = qmn$ are expected to be object pixels.

To illustrate the probability model, we first consider the number of provisional labels produced by a forward scan. A new provisional label is generated if all neighboring pixels in the scan mask are background pixels. Each pixel has the probability $(1 - q)$ of being a background pixel. Assuming that each pixel is generated independently, the probability of all four pixels being background pixels is $(1 - q)^4$.

In a 2D image, pixels normally have four neighbors in the forward scan mask. There are also four special cases that contain fewer pixels in their scan masks.

1. The top-left pixel that has no neighbors in the scan mask.
2. The pixels on the top-most row (except the left-most pixel), each of which has one neighbor to the left.
3. The pixels on the left-most column (except the top-most pixel), each of which has two neighbors.
4. The pixels on the right-most column (except the top-most pixel) each of which has three neighbors.

Including the normal case, there are five different scan masks used during a forward scan. An illustration of these five scan masks is shown in Table 1. The same table also lists the number of instances (in column 2 under the heading of **instances**) for each case and the probabilities of an object pixel receiving a new label (in column 6 under the heading of **labels**). Multiplying the density q and the values in columns 2 and 6 of Table 1, we get an estimate of the number of provisional labels produced for each case. The following equation shows the total number of provisional labels expected:

$$n_p = q\Big(1 + (m-1)(1-q) + (n-1)(1-q)^2 + (n-1)(1-q)^3 + (m-2)(n-1)(1-q)^4\Big). \quad (12)$$

Using the same probability model, we can estimate the time required by SAUF to label a random 2D binary image. To do this, we divide the operations performed by SAUF into six independent categories.

Table 1 The expected numbers of operations per object pixel used by the SAUF algorithm

Mask	Instances	Expected values			
		(3) Neighbors	(4) Copy	(5) Union	(6) Labels
e	1	0	1	0	1
d e	$m-1$	1	q	0	$1-q$
b c e	$n-1$	$2-q$	$q(2-q)$	0	$(1-q)^2$
a b d e	$n-1$	$3-3q+q^2$	$1-(1-q)^3$	0	$(1-q)^3$
a b c d e	$(m-2)(n-1)$	$(2-q)^2$	$4q-8q^2+7q^3-2q^4$	$q^2(1-q)(2-q)$	$(1-q)^4$

The dominant case is shown in the last row

1. *Work done per pixel*: Work performed on every pixel, such as reading a pixel value from main memory to a register, testing whether a pixel is a background pixel or an object pixel, and assigning the final label to each pixel (the last phase of any two-pass algorithm).
2. *Unaccounted work done per object pixel*: Work performed on an object pixel that is not already counted in the next four categories.
3. *Time for visiting the neighbors*: This is the major part of the scanning procedure. The process of traversing a decision tree requires multiple if-tests. Because each if-test is for a different neighbor, the amount of time required in this category should be proportional to the number of neighbors visited. The expected number of neighbors to be visited for each object pixel is shown in column 3 under the heading of **neighbors** in Table 1.
4. *Copying a provisional label or assigning a new label*: This includes two types of terminal nodes on a decision tree shown in Fig. 2, the new label operation and the one-argument copy function. The amount of work performed for each copy or assignment is a small constant. The expected number of copy (or new label) operations to be performed for each object pixel is shown in column 4 under the heading of **copy** in Table 1.
5. *Union operations*: This is a case where the two-argument copy function is invoked by a decision tree. Each union operation has the same cost as two find operations. Based on Theorem 2, we can say that the average cost of a find operation is a constant, and therefore the average cost of a union operation is a constant. The probability of performing a union operation for each object pixel is shown in column 5 under the heading of **union** in Table 1.
6. *Flattening operation*: This is the second phase of the SAUF algorithm. The total cost of this operation is proportional to the number of provisional labels. The probability of assigning a new label to an object pixel is shown in column 6 under the heading of **labels** in Table 1.

To illustrate how we obtain values in Table 1, we briefly describe how we compute the values in columns 3, 4, and 5 in the last row (the normal case). Column 3 contains the average number of neighbors accessed. In the normal case, the computation of this quantity is based on the decision tree shown in Fig. 2b. We associate each edge labeled "1" with the probability q and each edge labeled "0" with the probability $(1-q)$. There is one path from the root to a leaf that is of length 1 (i.e., when $I[\mathbf{b}] = 1$). Note that the path length is the number of neighboring pixels accessed. The probability of taking this path is q. There are two paths of length 3. The probabilities of taking these paths are $(1-q)q^2$ and $(1-q)^2q$. The total probability of accessing 3 neighbors is $(1-q)q$. There are four paths of length 4. The probabilities of taking these four paths are $(1-q)^4$, $(1-q)^3q$, $(1-q)^3q$, and $(1-q)^2q^2$. The total probability of accessing 4 neighbors is $(1-q)^2$. The average number of accesses to neighbors is $q + 3q(1-q) + 4(1-q)^2 = (2-q)^2$. This value is entered in the row for the normal case (case 5) under the column heading **neighbors** in Table 1.

Among the seven paths in a decision tree, there are two leading to a two-argument copy function. These two paths have probabilities of $(1-q)q^2$ and $(1-q)^2q^2$. The two-argument copy function invokes the union operation, and therefore the total probability of invoking a union operation is $q^2(1-q)(2-q)$. This value is entered in the row for the normal case under the heading of **union**. The remaining 5 paths lead to either a one-argument copy function or a new label function. We enter their total probability under the heading of **copy**. The other four rows of Table 1 are computed similarly.

For a typical image, where m and n are sufficiently large, the normal case should dominate the four special cases. Only considering the normal case, we can make a few observations. Our first observation is that the probability of performing a union approaches 0 for both small q ($q \to 0$) and large q ($q \to 1$). This agrees with our expectation.

Theorem 4 *In the scanning phase of two-pass labeling algorithm on a random 2D binary image, the average number of neighboring pixels visited following a decision tree is* 7/3, *and consequently, using a decision tree speeds up the scanning procedure by a factor of* 12/7.

Proof In the normal case, the number of neighbors visited is given by a quadratic formula, $(2-q)^2$. As the density q increases from 0 to 1, the quadratic formula quickly drops from 4 to 1. Using this formula, we can compute an average number of neighbors visited. If the density q is uniformly sampled between 0 and 1, we can compute the average number of neighbors visited by simply integrating the function $f(q) = (2-q)^2$ over q from 0 to 1, which yields 7/3.

The naive scanning procedure always accesses all 4 neighbors. On average, the speedup of using a decision tree is 12/7. □

Based on the probabilities shown in Table 1, we define the number of instances of six categories as

$$
\begin{aligned}
p &= mn,\\
n_o &= qmn,\\
n_n &= q(m-1+(n-1)(5-4q+q^2)+\\
&\quad (m-2)(n-1)(2-q)^2),\\
n_c &= q+q^2(m-1+(n-1)(5-4q+q^2)\\
&\quad +(m-2)(n-1)(4-8q+7q^2-2q^3)),\\
n_u &= q^3(1-q)(2-q)(m-2)(n-1),
\end{aligned}
$$

Let constants $C_1,\ldots,C_6$ represent the average cost per operation of the six categories identified, then the total execution time of SAUF is (note that n_p is defined in Eq. 12):

$$
\begin{aligned}
t_S &= C_1p + C_2n_o + C_3n_n + C_4n_c\\
&\quad + C_5n_u + C_6n_p.
\end{aligned}
\tag{13}
$$

5 Performance measurements

In this section, we report the timing measurements of various connected-component labeling algorithms. We use these measurements to verify the expected performance advantages of the two optimization strategies and the resulting labeling algorithm SAUF. The decision tree shown in Fig. 2b was implemented in all test programs that required a decision tree.

5.1 Test setup

To measure the performance of labeling algorithms, we used four different sets of binary images. We previously conducted a limited performance study in which we used random binary images only [45]. For this study, we used three additional sets of images from various applications. Some sample images are shown in Fig. 4, and summary descriptions of these images are given in Table 2. We applied Otsu thresholding [31] on the intensity to turn the application images into binary images. The random binary images used in this study were smaller than in our previous study, so that they were closer to the application images in size. Users who apply our algorithms on large images may see larger performance improvements as demonstrated in [45].

To ensure that our measurements are not biased by a particular hardware environment, we elected to run the same test cases on three different machines listed in Table 3. With each machine, we also chose to use a different compiler.

5.2 Effectiveness of the decision tree

We implemented two variants of the scan plus connection table algorithm, namely, SCT-4 and SCT-1. A summary of

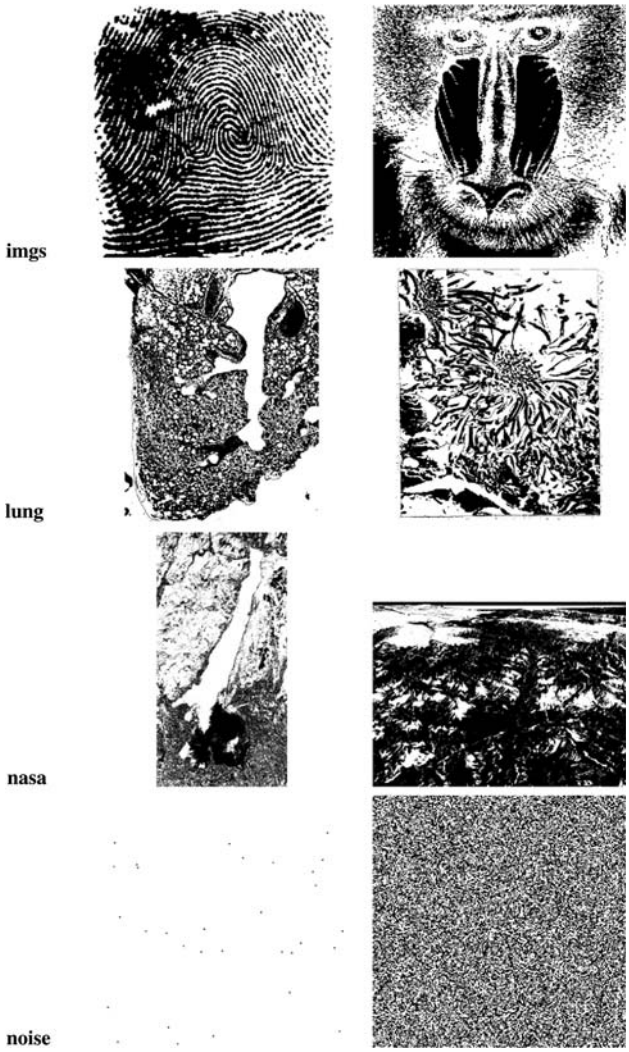


Fig. 4 A sample of the binary images used in tests. Object pixels are shown as *black*

timing results is given in Table 4. Because the four sets of test images have significantly different sizes, we show the average time for each set separately. The timing measurements were made for each test image. The test on each image was repeated enough times so that at least one second is used altogether. A minimum of five iterations was always used. The time values reported are wall clock time. The speedup of SCT-1 is measured against SCT-4. Each speedup value is computed for one test image and the speedup values reported in Table 4 are averages.

On each test platform, the two algorithms, SCT-4 and SCT-1, show consistent relative performances on the three sets of application images. The performance characteristics are slightly different for random binary images (marked **noise**). This is partly because the application images typically contain well-shaped connected components, whereas the random images contain irregular components. This

Table 2 Summary information about test images, where **N** is the number of images in the test set, **P** is the average number of pixels in an image, **O** is the average number of object pixels, **C** is the average number of connected components, and **Q** is the average number of pixels per component

Name	*N*	*P*	*O*	*C*	*Q*	Description
imgs	54	254,558	94,256	1,088	3,633	Images used in [38]
lung	64	468,220	315,898	3	198,211	Mouse lung structure images from lbl.gov
nasa	63	8,294,591	5,041,424	17,289	638	Satellite images from nasa.gov
noise	78	1,750,000	875,000	35,434	309,246	Random binary images (500 × 500, 1,000 × 1,000, 2,000 × 2000)

Table 3 Information about the three test machines

CPU type	Clock (MHz)	Cache (KB)	Memory (MB)	OS	Compiler
UltraSPARC	450	4,096	4,096	Solaris 8	Forte workshop 7
Pentium 4	2,200	512	512	Linux 2.4	gcc 3.3.3
Athlon 64	2,000	1,024	512	Windows XP	Visual Studio.NET

irregularity slightly reduces the effectiveness of SCT-1. On the application images, SCT-1 is about twice as fast as SCT-4. Based on the number of neighbors accessed, Theorem 3 predicts a speed up of 12/7, which is about 1.7. On random images, the actual observed speedup values shown in Table 4 are close to 1.7, which confirms our analyses.

In all test cases, SCT-1 is never slower than SCT-4. Because of this, we use a decision tree in all subsequent tests.

Table 4 Summary of timing measurements on the multi-pass algorithms: the time values are in milliseconds and the values reported for speedup are the averages of speedups computed for each individual image

	Time (ms)		speedup
	SCT-4	SCT-1	$\frac{\text{SCT}-1}{\text{SCT}-4}$
UltraSPARC			
imgs	96	44	2.1
lung	215	95	2.3
nasa	5,776	2,880	2.1
noise	940	501	1.9
Pentium 4			
imgs	15	8	2.0
lung	29	14	2.2
nasa	782	383	2.1
noise	173	97	1.8
Athlon 64			
imgs	14	8	1.7
lung	26	14	1.8
nasa	687	394	1.8
noise	141	87	1.7

5.3 Effectiveness of array-based union-find

To assess the effective of the new union-find algorithms, we compare the new scan plus array-based union-find (SAUF) algorithm with the scan plus union-find (SUF) algorithm [12]. We implemented two versions of SUF, SUF1 that flattens the active union-find trees after scanning each line of the image as suggested by Fiorio and Gustedt [12], and SUF0 that does not perform the extra flattening operation. In Lemmas 5 and 6, we prove that both should use $O(p)$ time, and suggested that SUF0 could be faster than SUF1. This expectation is confirmed by the timing results shown in Table 5. SAUF is considerably faster than both SUF1 and SUF0. Since all of them use the same scanning procedure, the performance differences are due to different union-find algorithms. The simpler union-find algorithms in SAUF are clearly more efficient.

As in the previous table, Table 5 reports the elapsed time. In this table, the speedup was measured against SUF1. In our tests, SUF0 was at least 30% faster than SUF1 on relatively small test images. On larger images, the performance differences were much larger. For example, on the lung structure images, SUF0 was five times as fast as SUF1 on two of the three test machines. From our analyses, we expected SUF0 to be faster than SUF1; however, the observed performance difference was much larger than anticipated. Our new labeling algorithm SAUF was usually four times or more as fast as SUF1, and about twice as fast as SUF0. The performance differences were even larger when many provisional labels were combined into a small number of final labels, as in the test image set **lung**. In these cases, the union-find algorithms need to unite more provisional labels and a set of more efficient union-find algorithms makes a bigger difference. On **lung** images, the

Table 5 Summary of timing measurements on the three two-pass algorithms; the time values are in milliseconds and the speedup values are relative to SUF1

	SUF1	SUF0		SAUF	
	Time	Time	Speedup	Time	Speedup
UltraSPARC					
imgs	83	62	1.3	22	3.7
lung	366	134	2.7	53	6.8
nasa	5,279	3,231	1.6	1,164	4.6
noise	1,056	742	1.4	243	4.4
Pentium 4					
imgs	25	16	1.7	5	5.5
lung	131	25	5.2	10	13.3
nasa	1,506	576	2.5	182	7.9
noise	332	186	1.9	47	6.6
Athlon 64					
imgs	17	11	1.7	4	4.7
lung	86	17	5.1	7	11.7
nasa	1,073	429	2.4	134	7.5
noise	237	140	1.9	34	6.5

SAUF is more than 10 times faster than SAUF1 on two of the three test machines.

5.4 Verifying performance model for SAUF

As shown in Table 2, we used 78 random binary images of various sizes for this set of tests. For each image, we computed the average time used by SAUF on each of the test machines. We used these 78 average time values to compute the six constants $C_1,\ldots,C_6$ for each machine. The computation used a linear least-square formulation to minimize the fitting error with a non-negative constraint.[4] The results of $C_1,\ldots,C_6$ for all three test machines are shown in Table 6. Because the three computers used different types of CPUs, caches and operating systems, these constants are different. we next study their similarity and differences to understand the our performance model further.

Overall, we see that our performance model captures the actual work quite well because C_2 is 0 on all three machines. Category 2 was introduced as a catch-all category. The value of C_2 being 0 indicates there is no need for such term.

On all three machines, both C_1 and C_3 were computed as positive values. The value of C_1 is the average time spent on per pixel operations such as reading a pixel from memory to register and assigning the final labels. This value is positive because at a density of 0, SAUF uses some time to label the image. The value C_3 is the average time used to access a neighboring pixel during the scanning phase, which involves accessing the pixel value of the neighbor and performing an if-test to see if it an object pixel. Both of which consume a number of clock cycles.

[4] The computation uses the function `lsqlin` from the optimization toolbox of MATLAB.

Table 6 The constant values (10^{-8} s) of Eq. 13 produced with a non-negative least-square fitting of measured time values

	UltraSPARC	Pentium 4	Athlon 64
C_1	9.5	1.1	1.2
C_2	0	0	0
C_3	5.3	0.8	3.7
C_4	0.2	0	0
C_5	0	6.7	4.1
C_6	0	11.5	5.4

The constant C_4 represents the average cost of a copy operation and the operation to assign a new label. We expected it to be small. This was indeed the case as shown in Table 6. The values C_5 and C_6 are zero on **UltraSPARC**, but are nonzero on the others. This is due to the different sizes of CPU caches as shown in Table 3. These two constants measure the average cost of a union operation and the operation to compute a final label in the analysis phase. They involve operations on array P, which is usually less than 1 MB because the array has less than 1/4 million (4-byte) elements in most cases as shown in Fig. 6. An array of this size can fit in cache on **UltraSPARC**, but not on the other systems.

With the six constants shown in Table 6, we can use Eq. 13 to compute the expected time. In Fig. 5, we show the measured time along with the expected time. We see that the expected time agrees with the measured time to within $\pm 10\%$ in most cases. The only case where the expected values never intersected any observed values is when random images of size $1{,}000 \times 1{,}000$ were labeled on **UltraSPARC**. In this particular case, the estimated time is about 1/4 larger than the actual measured values. This discrepancy is largely because we used the same parameters for the smaller images and the larger ones. On this particular machine, the smaller images fit into the cache, which makes the labeling algorithm more efficient on smaller images than on larger ones.

The estimated number of provisional labels for random images is given in Eq. 12. As a sanity check for the performance model, we compared this estimated number of provisional labels with the actually observed numbers. We plotted the estimated and the observed numbers of provisional labels in Fig. 6. The estimated values are close to the observed values for $q < 0.2$. For higher densities, the differences between estimated and observed values become

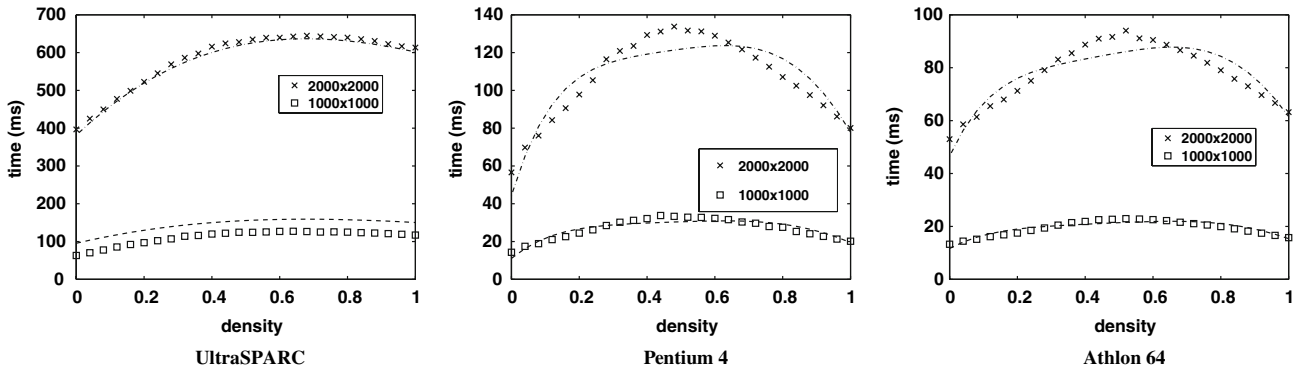


Fig. 5 The measured time (in ms) used by SAUF agrees with the performance model (shown as *broken lines*) described by Eq. 13

more pronounced because the independence assumption used for our estimation becomes more unreliable as q increases.

In Theorem 3, we prove that SAUF has time complexity of $O(p)$, which is theoretically optimal. In Fig. 7, we plot the maximum observed time and the average observed time versus the image size p. It is clear that both the maximum time and the average time scales linearly with p.

5.5 Comparison with contour tracing

According to the current literature, the Contour Tracing algorithm is the fastest algorithm for connected-component labeling on 2D binary images [9]. To demonstrate the effectiveness of our optimization strategies, we compared SAUF versus the contour tracing algorithm. We used the implementation of the contour tracing algorithm distributed by the original authors. In the following discussions, we denote it as CTo to emphasis that the program is from the original authors.

Table 7 shows the average time used by SAUF and the contour tracing algorithm. In 8 out of the 12 cases shown, SAUF is noticeably faster than CTo, particularly on larger images. Of the three sets of large images, the images in the set **nasa** are scenery photos which have more well-shaped components than others. The contour tracing algorithm was more efficient in identifying these well-shaped components because there are fewer pixels on their boundaries. On the smaller images, SAUF and CTo perform about the same overall. The average speedup of SAUF over CTo, across the four sets of test images and on three machines, is about 1.5.

Figure 8 shows the relative performance of SAUF over CTo on a set of large random images. In this case, the images still fit in memory as in earlier tests. On these images with 100 million pixels (requiring about 400 MB), SAUF is between 6 and 10 times faster than CTo (except the special case of the empty image with no object pixels). The relative performance differences between SAUF and CTo are much larger in this test for two reasons. Firstly, the components in random images are not well-shaped, and a majority of the pixels in a components are on the boundary. CTo performs more work on boundary

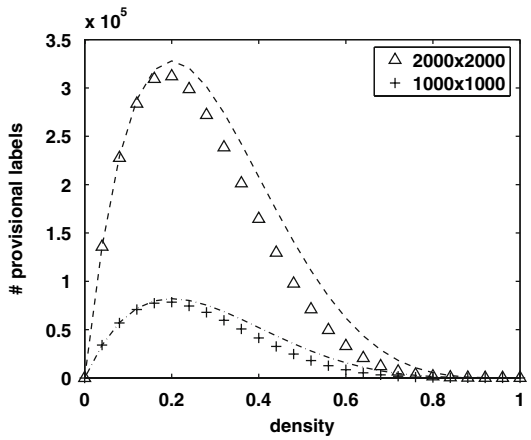


Fig. 6 The actual number of provisional labels observed plotted against the estimated labels (shown as *broken lines*)

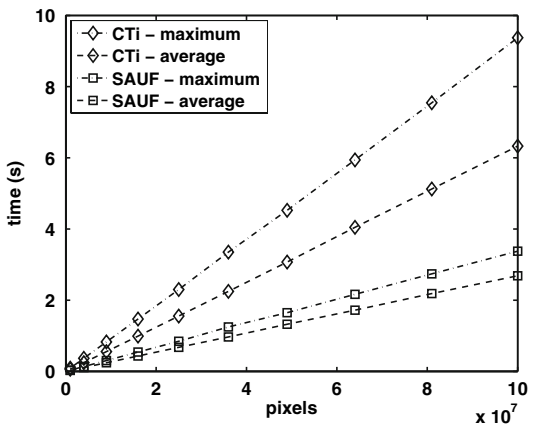


Fig. 7 Time (s) used by SAUF scales linearly with the number of pixels in the images to be labeled

Table 7 Average time (in milliseconds) used by CTo and SAUF to label the test images; the overall average speedup is 1.5

	Time (ms)		Speedup
	CTo	SAUF	$\frac{\text{SAUF}}{\text{CTo}}$
UltraSPARC			
imgs	21	22	1.0
lung	127	53	2.4
nasa	793	1,164	0.8
noise	327	243	1.5
Pentium 4			
imgs	4	5	0.8
lung	21	10	2.1
nasa	358	182	1.3
noise	67	47	1.3
Athlon 64			
imgs	4	4	1.0
lung	20	7	2.7
nasa	191	134	1.4
noise	59	34	1.7

pixels than on interior ones. Therefore CTo performs more work per pixel on random images than on scenery images. Secondly, as the image sizes increases, the random memory accesses used by CTo becomes relatively more expensive because of the increased likelihood of cache line address collision which leads to the same words to be loaded from main memory to cache more times. CTo also uses more memory than SAUF, which increases the likelihood that some inactive program or data may need to be swapped out of memory to make space for the active program and data. This also increases the observed elapse time.

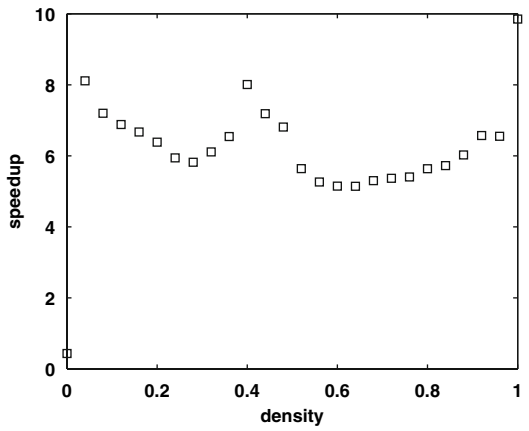


Fig. 8 Speedup of SAUF over CTo on a set of large random images (10,000 × 10,000)

6 Summary and future work

We have presented two strategies for optimizing the connected-component labeling algorithms. The first strategy minimizes the work in the scanning phase of a labeling algorithm; and the second reduces the time needed for manipulating the equivalence information among the provisional labels. Our analyses show that a two-pass algorithm using these strategies has the optimal worst-case time complexity $O(p)$, where p is the number of pixels in an image. We also showed with extensive tests that the new algorithm SAUF significantly outperforms well-known algorithms, such as the scan plus connection table [38] and scan plus union-find [12]. On the set of 2D images used for our timing measurements, the contour tracing (CT) algorithm is the fastest known method [9]. On relatively small images, SAUF outperform CT by 50%. However, on larger images, we observed a factor of 10 improvement for SAUF, because the memory access pattern of SAUF is more regular than CT and the relative advantage of SAUF increases as the image size increases.

The optimization strategies are straightforward to implement and can be extended to higher dimensional images. It also produces consecutive labels, which are convenient for applications.

More work remains to be done for a better understanding of the performance features and trade-offs of these strategies. For example, it may be useful to mix the contour tracing algorithm and SAUF. A derivation of a bound on the maximum number of scans needed by the SCT algorithm would help us to understand SCT better. It should also be interesting to apply the two optimization strategies to parallel algorithms for connected-component labeling and for different image formats.

7 Originality and contribution

This paper presents a new two-pass connected-component labeling algorithm based on two optimization strategies, the first one uses a decision tree to minimize the number of neighbors examined during the scanning phase, and the second one streamlines the union-find algorithms to minimize the work needed to manage label equivalence information. These optimization can be used in other labeling algorithms separately and are novel in their own right. We have not seen any other published labeling algorithm that uses a decision tree to minimize work. The second strategy combines an effective way of using union-find algorithms for labeling [12] with an array-based implementation for union-find [2, 11]. The novelty of this approach is that we are able to remove a significant amount

of unnecessary work while keeping the algorithms simple enough for an array-based implementation.

Combined together, the two optimization strategies form a powerful two-pass labeling algorithm that are faster than known labeling algorithms for 2D images. The new two-pass labeling algorithm is efficient because it performs the minimal amount of work necessary to find the connected components and it does so with a relatively small amount of random memory accesses. These are confirmed with both theoretical analyses and extensive timing comparisons.

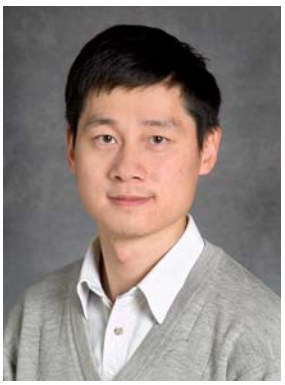
Acknowledgment This work was supported in part by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

References

1. Agarwal PK, Arge L, Ke Yi (2006) I/O-efficient batched union-find and its applications to terrain analysis. In: SCG '06: Proceedings of the 22nd annual symposium on computational geometry. ACM Press, New York, pp 167–176
2. Aho AV, Hopcroft JE, Ullman JD (1974) The design and analysis of computer algorithms. Addison–Wesley, Reading
3. Aho AV, Ullman JD, Hopcroft JE (1983) Data structures and algorithms. Addison–Wesley, Reading
4. Alnuweiri HM, Prasanna VK (1992) Parallel architectures and algorithms for image component labeling. IEEE Trans Pattern Anal Mach Intell 14(10):1014–1034
5. Alstrup S, Ben-Amram AM, Rauhe T (1999) Worst-case and amortised optimality in union-find. In: Proceedings of 31th Annual ACM symposium on theory of computing (STOC'99). ACM Press, New York, pp 499–506
6. Amin A, Fisher S (2000) A document skew detection method using the Hough transform. Pattern Anal Appl 3(3):243–253
7. Ballard DH (1982) Computer vision. Prentice-Hall, Englewood
8. Bollobás B, Simon I (1985) On the expected behavior of disjoint set union algorithms. In: STOC '85: Proceedings of the 17th annual ACM symposium on theory of computing. ACM Press, New York, pp 224–231
9. Chang F, Chen C-J, Lu C-J (2004) A linear-time component-labeling algorithm using contour tracing technique. Comput Vis Image Underst 93(2):206–220
10. Dillencourt MB, Samet H, Tamminen M (1992) A general approach to connected-component labeling for arbitrary image representations. J ACM 39(2):253–280
11. Doyle J, Rivest RL (1976) Linear expected time of a simple union-find algorithm. Inf Process Lett 5(5):146–148
12. Fiorio C, Gustedt J (1996) Two linear time union-find strategies for image processing. Theor Comput Sci 154(2):165–181
13. Fiorio C, Gustedt J (1997) Memory management for union-find algorithms. In: Proceedings of 14th symposium on theoretical aspects of computer Science. Springer, New York, pp 67–79
14. Gabow HN, Tarjan RE (1983) A linear-time algorithm for a special case of disjoint set union. In: STOC '83: Proceedings of the 15th annual ACM symposium on theory of computing. ACM Press, New York, pp 246–251
15. Galil Z, Italiano GF (1991) Data structures and algorithms for disjoint set union problems. ACM Comput Surv 23(3): 319–344
16. Galler BA, Fisher MJ (1964) An improved equivalence algorithm. Commun ACM 7(5):301–303
17. Gonzalez RC, Woods RE (2002) Digital Image Processing, 2nd edn. Prentice-Hall, New Jersy
18. Gotoh T, Ohta Y, Yoshida M, Shirai Y (1987) Component labeling algorithm for video rate processing. In: Proceedings of SPIE 1987. Advances in image processing, vol 804, pp 217–224
19. Haralick RM (1981) Some neighborhood operations. Plenum Press, New York, pp 11–35
20. Haralick RM, Shapiro LG (1985) Image segmentation techniques. Comput Vis Graph Image Process 29(1):100–132
21. Hayashi H, Kudo M, Toyama J, Shimbo M (2001) Fast labelling of natural scenes using enhanced knowledge. Pattern Anal Appl 4(1):20–27
22. Qingmao H, Guoyu Q, Nowinski WL (2005) Fast connected-component labelling in three-dimensional binary images based on iterative recursion. Comput Vis Image Underst 99:414–434
23. Kim J-H, Kim KK, Suen CY (2000) An HMM-MLP hybrid model for cursive script recognition. Pattern Anal Appl 3(4):314–324
24. Knop F, Rego V (1998) Parallel labeling of three-dimensional clusters on networks of workstations. J Parallel Distrib Comput 49(2):182–203
25. Knuth DE, Schönhage A (1978) The expected linearity of a simple equivalence algorithm. Theor Comput Sci 6:281–315
26. Lucas JM (1990) Postorder disjoint set union is linear. SIAM J Comput 19(5):868–882
27. Lumia R (1983) A new three-dimensional connected components algorithm. Comput Vis Graph Image Process 23(2):207–217
28. Lumia R, Shapiro L, Zungia O (1983) A new connected components algorithm for virtual memory computers. Comput Vis Graph Image Process 22(2):287–300
29. Moga AN, Gabbouj M (1997) Parallel image component labeling with watershed transformations. IEEE Trans Pattern Anal Mach Intell 19(5):441–450
30. Naoi S (1995) High-speed labeling method using adaptive variable window size for character shape feature. In: IEEE Asian Conference on computer vision, vol 1, pp 408–411
31. Otsu N (1979) A threshold selection method from gray level histograms. IEEE Trans Syst Man Cybern 9:62–66
32. Regentova E, Latifi S, Deng S, Yao D (2002) An algorithm with reduced operations for connected components detection in itu-t group 3/4 coded images. IEEE Trans Pattern Anal Mach Intell 24(8):1039–1047
33. Rosenfeld A (1970) Connectivity in digital pictures. J ACM 17(1):146–160
34. Rosenfeld A, Kak AC (1982) Digital picture processing, 2nd edn. Academic Press, San Diego
35. Shi J, Malik J (2000) Normalized cuts and image segmentation. IEEE Trans Pattern Anal Mach Intell 22(8):888–905
36. Stockman GC, Shapiro LG (2001) Computer Vision. Prentice-Hall, Englewood
37. Suri JS, Singh B, Reden L (2002) Computer vision and pattern recognition techniques for 2-d and 3-d MR cerebral cortical segmentation: a state-of-the-art review. Pattern Analy Appl 5(1):46–76
38. Suzuki K, Horiba I, Sugie N (2003) Linear-time connected-component labeling based on sequential local operations. Comput Vis Image Underst 89(1):1–23
39. Tarjan RE, van Leeuwen J (1984) Worst-case analysis of set union algorithms. J ACM 31(2):245–281
40. Tarjan RE (1975) Efficiency of a good but not linear set union algorithm. J ACM 22(2):215–225
41. Tarjan RE (1977) Reference machines require non-linear time to maintain disjoint sets. In: STOC '77: Proceedings of the 9th

annual ACM symposium on theory of computing. ACM Press, pp 18–29
42. Udupa JK, Ajjanagadde VG (1990) Boundary and object labelling in three-dimensional images. Comput Vis Graph Image Process 51(3):355–369
43. Wang K-B, Chia T-L, Chen Z, Lou D-C (2003) Parallel execution of a connected component labeling operation on a linear array architecture. J Inf Sci Eng 19:353–370
44. Wang Y, Bhattacharya P (2003) Using connected components to guide image understanding and segmentation. Mach Graph Vis 12(2):163–186
45. Wu K, Otoo E, Shoshani A (2005) Optimizing connected component labeling algorithms. In: Proceedings of SPIE medical imaging conference 2005, San Diego, CA, 2005. LBNL report LBNL-56864
46. Yao AC (1985) On the expected performance of path compression algorithms. SIAM J Comput 14(1):129–133
47. Yapa RD, Koichi H (2007) A connected component labeling algorithm for grayscale images and application of the algorithm on mammograms. In: SAC'07: Proceedings of the 2007 ACM symposium on applied computing. ACM Press, New York, pp 146–152

Author Biographies



Kesheng Wu is a staff computer scientist at Lawrence Berkeley National Laboratory. His work primarily involves data management, data analyses and scientific computing. He is the lead developer of FastBit bitmap indexing software for searching over large datasets. He also led the development of a software package call TRLan, which computes eigenvalues of large symmetric matrices on parallel machines. He received a Ph.D. in computer science from the University of Minnesota, an M.S. in physics from the University of Wisconsin-Milwaukee, and a B.S. in physics from Nanjing University, China. His homepage on the web is http://lbl.gov/kwu.

Ekow Otoo holds a B.Sc. degree in Electrical Engineering from the University of Science and Technology, Kumasi, Ghana, and a Ph.D. degree in Computer Science from McGill University, Montreal, Canada. From 1987 to 1999, he was a tenured faculty at Carleton University, Ottawa, Canada. He has served as a consultant to Bell Northern Research, and the GIS Division, Geomatics Canada. He is presently a consultant with Mathematical Sciences Research Institute, Ghana, and a staff scientist/engineer, LBNL, Berkeley. He is a member of the ACM and IEEE. His research interests include database management, data structures, algorithms, parallel and distributed computing.

Kenji Suzuki received his Ph.D. degree from Nagoya University in 2001. In 2001, he joined Department of Radiology at University of Chicago. Since 2006, he has been Assistant Professor of Radiology, Medical Physics, and Cancer Research Center. His research interests include computer-aided diagnosis, machine learning, and pattern recognition. He published 110 papers including 45 journal papers. He has served as an associate editor for three journals and a referee for 17 journals. He received Paul Hodges Award, RSNA Certificate of Merit Awards, Cancer Research Foundation Young Investigator Award, and SPIE Honorable Mention Award. He is a Senior Member of IEEE.