LBNL-59102

Two Strategies to Speed up Connected Component Labeling Algorithms

Kesheng Wu, Ekow Otoo, Kenji Suzuki,

Abstract— **This paper presents two new strategies to speed up connected component labeling algorithms. The first strategy employs a decision tree to minimize the work performed in the scanning phase of connected component labeling algorithms. The second strategy uses a simplified union-find data structure to represent the equivalence information among the labels. For 8-connected components in a two-dimensional (2D) image, the first strategy reduces the number of neighboring pixels visited from 4 to 7/3 on average. In various tests, using a decision tree decreases the scanning time by a factor of about 2. The second strategy uses a compact representation of the union-find data structure. This strategy significantly speeds up the labeling algorithms. We prove analytically that a labeling algorithm with our simplified union-find structure has the same optimal theoretical time complexity as do the best labeling algorithms. By extensive experimental measurements, we confirm the expected performance characteristics of the new labeling algorithms and demonstrate that they are faster than other optimal labeling algorithms.**

Index Terms— **Connected component labeling, optimization, union-find algorithm, decision tree, equivalence relation.**

I. Introduction

Connected component labeling is a procedure for assigning a unique label to each object (a group of connected components) in an image [1], [2], [3], [4]. These labels are the keys for any subsequent analysis procedure and are used for distinguishing and referencing the objects. This makes connected component labeling an indispensable part of nearly all applications in pattern recognition and computer vision. For example, before a computer can detect or classify any object in an image, be it a car, a person, or a lesion, groups of similar pixels are identified and labeled. Each group is generally referred to as an object. Identifying all pixels in a group enables one to compute the information required for subsequent processing, such as area size, height, width, and perimeter. Clearly, connected component labeling is one of the most fundamental algorithms of image analysis. In many cases, it is also one of the most time-consuming tasks among other pattern-recognition algorithms [5]. For these reasons, connected component labeling continues to remain an active area of research. Some recent work is included in references [6], [7], [8], [9], [10], and [11]. This paper presents two strategies that significantly speed up the commonly used algorithms for connected component labeling.

To illustrate the new optimization strategies, we consider the problem of labeling binary images stored in 2-dimensional (2D) arrays. These images are typically the output from another image-processing step, such as segmentation [12], [13], [14]. Each pixel in a binary image is called either an object pixel or a 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 in a 2D image, i.e., 4-connectedness and 8-connectedness [15]. In this paper, we use the 8-connectedness as illustrated in Fig. 1(a).

There are a number of different approaches to labeling the connected components. The simplest approach repeatedly scans the image to determine appropriate labeling until no further changes can be made to the assigned labels [3]. A label assigned to an object pixel is called a *provisional label* before the final assignment. 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, as illustrated in Figs. 1(b) and (c), are examined for determining an appropriate

K. Wu and E. Otoo are with Lawrence Berkeley National Laboratory. Their e-mail addresses are {KWu, EJOtoo}@lbl.gov.

K. Suzuki is with the University of Chicago. His e-mail address is suzuki@uchicago.edu.

label to be assigned to the current pixel. 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 any 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. One simple 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. Based on these and other features, labeling algorithms can be grouped into five different categories.

1) **Multi-pass algorithms** ([1], [4], [11], [16], [15]): The basic labeling algorithm described above is the best known example of this group. An obvious short-coming of this algorithm is that the number of scans can be large. To control the number of iterations, one may alternate the direction of scans or directly manipulate the equivalence information. A recent example of such an algorithm was given by Suzuki et al. [11]. It performs sequential scans, and uses a label connection table to reduce the number of scans. In most tests, this algorithm uses no more than four scans, which is less than used by others in this group. In later discussion, we refer to this algorithm as Scan plus Connection Table, or SCT.
2) **Two-pass algorithms** ([8], [17], [18], [19], [20], [21]): 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 all object pixels, and to record the equivalence information about the provisional labels.
    b) *Analysis phase:* This phase analyzes the label equivalence information to determine the final label of each provisional label.
    c) *Labeling phase:* This third phase assigns the final labels to the object pixels by doing 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 other two. One of the most efficient data structures for representing the equivalence information is the *union-find* data structure [17], [8]. Because the operations on the union-find data structure are very simple, one would expect the analysis phase and the labeling phase to take less time than the scanning phase. Because a multi-pass algorithm typically repeats the scanning phase multiple times, one would expect a two-pass algorithm to be faster than a multi-pass algorithm. Indeed, there are a number of two-pass algorithms that not only perform well in practice, but also have a theoretical worst-case time complexity $\mathcal{O}(p)$, where p is the number of pixels in the image. Given an image in a 2D array, any labeling algorithm must visit every pixel at the minimum. Thus, $\mathcal{O}(p)$ complexity is theoretically optimal. In this paper, we use one such optimal algorithm by Fiorio and Gustedt [8] as the representative of this group. Because the equivalence label 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** ([1], [7], [15]): 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. The most efficient algorithm in this group is the Contour Tracing (CT) algorithm by Chang et al. [7]. Algorithms in this group need to go through the image only once, typically with an irregular access pattern. For example, each time an unlabeled object pixel is found, the CT algorithm follows the boundary of the connected component until it returns to the starting position. It then fills in the labels for the object pixels in the interior of the component.
4) **Algorithms for hierarchical image formats** ([17], [22], [23], [24], [25]): There are many labeling algorithms that are designed for more complex image formats than the simple 2D array. Most of these algorithms are built on the algorithms from groups 1 and 2. This paper aims to improve the basic building blocks of algorithms from groups 1 and 2, which indirectly benefits algorithms in this group.

5) **Parallel algorithms** ([5], [26], [27], [16], [9], [10]): Because connected component labeling is considered a bottleneck in many image analysis applications, a large number of parallel algorithms have been developed. Most of them utilize the basic steps used in the first two groups. Our optimization strategies should benefit these algorithms as well.

In general, 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 has been reported [11]. One reason that, in practice, 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 most modern computers than are random memory accesses. In fact, in the past few years, the CPU clock rate has been increasing significantly faster than that of the speed of memory accesses. This makes random memory accesses relatively more expensive than before. Based on this observation, our optimization strategies seek to minimize the number of random memory accesses. By combining the optimization strategies, we aim at producing a two-pass algorithm that is more efficient than the fastest known algorithm from the first three groups [7].

Our first optimization strategy minimizes the number of neighbors visited during a scan and therefore reduces the number of memory accesses. A straightforward scanning procedure examines the four neighbors **a**, **b**, **c**, and **d** in turn [2], [3]. With our optimization, if **b** is an object pixel, the other three pixels are not examined. A decision tree is used for deciding which neighbors to examine if **b** is a background pixel. This optimization strategy has the potential of reducing the amount of work during a scan by up to a factor of four. In Suzuki et al. [11], the authors pointed out a way to reduce the amount of work that is more suitable for a hardware implementation. Our strategy is intended for a software implementation.

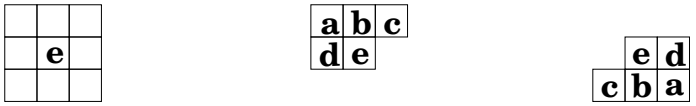
Our second optimization strategy simplifies the data structure and the algorithms used to solve the union-find problem. A considerable amount of work has been done on union-find [28], [29], [30], [17], [31], [32]. Because union-find involves relatively simple operations, the time spent on union-find is expected to be a small fraction of a two-pass algorithm. However, this is not the case (see, for example, ref. [31]), which has motivated a number of research efforts [31], [33], [34], [35]. Our new algorithms for union-find are based on an array data structure [30]. Because this data structure requires less computer memory than do commonly used ones and because the new algorithms access the memory in a more regular manner, using our union-find data structure and algorithms significantly reduces the overall time for connected component labeling. More specifically, with a common union-find solution, the Contour Tracing algorithm was shown to outperform the SUF algorithm [7]. However, our new Scan plus Array-based Union-Find (SAUF) algorithm is usually faster than the same Contour Tracing algorithm[1].

The remainder of this paper is divided into five sections. The next section describes the decision tree used for minimizing the number of neighbors visited during a scan. Sections III contains the description of the new union-find solution. In Section IV, we analyze the correctness of the two optimization strategies, and prove the worst-case time complexity of the new labeling algorithm that employs the two strategies. In addition, we derive a performance model for the expected time needed by the new algorithm to label random binary images. In Section V, we present timing results that confirm the improvement in performance of the two optimization strategies through extensive set of experiments. The timing results also verify the performance model for the new labeling algorithm on random images. A summary and discussion on future work are given in Section VI.

II. Minimizing the cost of scanning operations

In this section, we describe a decision tree used for minimizing the work required in the scanning phase used by most connected component labeling algorithms. As an example, we apply this optimization strategy to the Scan plus Connection Table (SCT) algorithm of Suzuki et al. [11]. We prove that a decision tree indeed minimizes the number

[1]In our tests, we compare against an implementation of the Contour Tracing algorithm distributed by the original authors of the algorithm. It is available from <http://www.iis.sinica.edu.tw/∼fchang/03src.html>.



(a) 8−connected neighborhood (b) Forward scan mask (c) Backward scan mask

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**.

of neighbors visited during scans later in Section IV.

A. The basic scanning procedure

Given a 2D image stored in a 2D array, the simplest scanning procedure for performing connected component labeling is to visit each pixel in turn, and assign a label to each object pixel that is either a label of its neighbors' or a new distinct label if its neighbors are all background pixels. Let I denote the 2D array for an image. Let $I[i,j] = 0$ denote a background pixel, and let $I[i,j] = 1$ denote an object pixel. We use an array L with 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. 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 [11]. For simplicity, we have also chosen to use integer labels, but it is possible to use different types of labels as well. We name the pixel in the scan mask as 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 being scanned, and $I[\mathbf{b}]$ denotes the pixel value of the neighbor directly above **e** in the vertical direction. 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:

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

The above expression means that $L[\mathbf{e}]$ is assigned 0 if $I[\mathbf{e}] = 0$. It is assigned a new label l, and l is increased by 1, if $\mathbf{a}, \mathbf{b}, \mathbf{c}$, and $\mathbf{d}$ in the scan mask are all background pixels. Otherwise, it is assigned the minimum of the provisional labels already assigned to the scan mask. In later scans, the labels for the object pixels are modified to be the minimum labels of their neighboring object pixels, as described by the following expression (which is the last case of Equation (1)):

$$\begin{aligned} L[\mathbf{e}] \leftarrow & \min_{i \in (\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d}) | I[i]=1} (L[i]), \\ & \text{if } I[\mathbf{e}] = 1, \text{and } \ \exists i \in (\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d}) \\ & \text{such that } I[i] = 1. \end{aligned} \tag{2}$$

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. In a multi-pass algorithm, this basic scanning procedure is repeated until the label array L no longer changes. After the first scan, the labels may change because pixels in one connected component could have been assigned multiple labels. We say that these labels are equivalent, and we have chosen arbitrarily to use the smallest label as the representative of the equivalent labels. As labels are discovered to be equivalent during a scan, the pixels not yet scanned will take on the smaller label. However, pixels already scanned during this pass will not change. In general, many scans are required for converting all equivalent labels to the smallest one. One successful strategy used for reducing the number of scans is the use a label connection table [11], which we briefly describe next.

B. Scan plus connection table

The connection table proposed by Suzuki et al. [11] 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:

$$L[\mathbf{e}] \leftarrow \begin{cases} \text{If } I[\mathbf{e}] = 0, \\ 0, \\ \text{if } \forall i \in (\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}), I[i] = 0, \\ l, T[l] \leftarrow l, l \leftarrow l + 1, \\ \text{otherwise}, \\ \min\limits_{i \in (\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d}) | I[i]=1} \left(T[L[i]]\right), \\ \quad \forall i \in (\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}) \mid I[i] = 1, \\ \quad T[L[i]] \leftarrow L[\mathbf{e}] \end{cases} \tag{3}$$

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 is as follows (which are equivalent to the last case in Equation (3)):

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

Because the connection table passes the label equivalence information to all the pixels with the same provisional labels, the labels can propagate much faster than in similar algorithms. In many tests, SCT usually required no more than 4 scans [11].

The above formulae indicate that all four neighbors in the scan masks need to be visited. In later discussion, we refer to this basic version of SCT as SCT-4 because it always visits the four neighbors. In the paper that proposed SCT [11], the authors also suggested an optimization in the appendix. Their optimization reduced the number of neighbors visited from 4 to 2 in many cases. In later discussion, we refer to this improved version as SCT-2. The decision trees to be described next are another step in this optimization process. They can further reduce the number of neighbors visited.

C. Decision tree

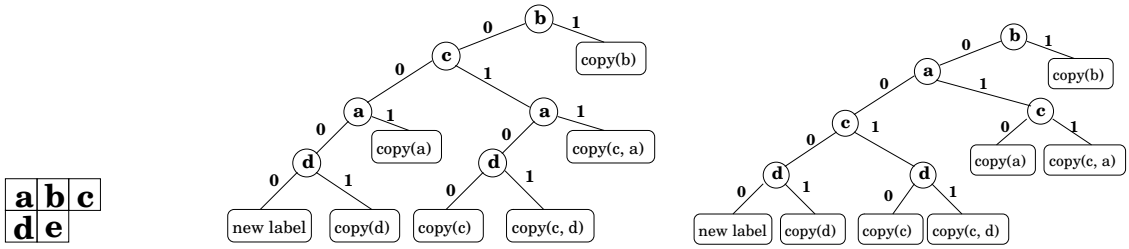
In Fig. 2(a), it is clear that all the neighbors in the scan masks are neighbors of **b**. If there is enough equivalence information for accessing the up-to-date label of **b**, then there is no need to examine the rest of the neighbors. Based on this observation, we present a set of decision trees that each organizes 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 the four. Throughout this paper, the decision tree of Fig. 2(b) is assumed.

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. By using this detailed decision process, we minimize the number of neighbors visited during the scans.

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

1) The one-argument copy function, copy(**a**), contains one statement:
$$L[\mathbf{e}] \leftarrow T[L[\mathbf{a}]]. \tag{5}$$
2) The two-argument copy function, 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.
$$L[\mathbf{e}] \leftarrow l, \quad T[l] \leftarrow l, \text{ and } l \leftarrow l + 1. \tag{7}$$
The statements of Equation (7) exactly replicate the second case in Equation (3).

It is clear that a scanning procedure following a decision tree will do less work than using either of the straightforward scans suggested by Equations (3) and (4). The use of a decision tree minimizes the number of neighbors visited for determining a label for pixel **e**. It is much easier to formalize this observation once we have explained the concept of union-find. For this reason, we will not give a formal analysis of the decision trees until Section IV, after the union-find data structure is explained. In the following discussion, we call the SCT algorithm that employs a decision tree SCT-1.



(a) Forward scan mask (b) Decision tree 1 (c) Decision tree 2

Fig. 2. The decision trees used in scanning for 8-connected neighbors.

III. Simplifying union-find

The connection table used in the previous section is one way of controlling the number of scans required for labeling the connected components. Another way of reducing the number of scans is to use the union-find data structure, which results in two-pass algorithms for connected component labeling [17], [8], [33], [36]. These two-pass algorithms are typically executed in three phases. A *scanning* phase for examining the image and assigning provisional labels to object pixels. This phase also fills the union-find data structure and records the label equivalence information. The second phase, called the *analysis* phase, flattens the union-find data structure so that the final labels are easily accessible. The last phase, called the *labeling* phase, assigns the final label to every object pixel.

Using union-find in connected component labeling has been well studied in [17], [8], [33], [34]. Because the algorithms used for maintaining and manipulating the union-find data structure are very simple, one may expect that these operations take a negligible amount of time compared with the scanning phase. However, this is not the case in practice. For example, two recent publications by two separate groups have shown that the two-pass algorithms are not as efficient as expected [7], [11]. One of the main reasons for this performance problem is that most union-find algorithms perform a large number of random memory accesses [31]. To minimize random memory accesses, we present a simple variant of the union-find data structure in this section. In later sections, we analyze our union-find approach and measure its impact on two-pass connected component labeling.

A. Union-find

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 an equivalence between two labels [36]. It is easy to see that 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:

- make a new tree of a single node,
- find the root of a given node, and
- unite two trees.

The second and third operations are commonly referred to as *find* and *union*, 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. Two find operations are needed for finding the root nodes of their respective trees before the roots can be connected. In general, the main cost of a union operation is for the two find operations. Therefore, an efficient find algorithm is critical to the overall efficiency of union-find operations.

A natural way of representing rooted trees in software is to use pointers. In most cases, nodes of a pointer-based rooted tree are scattered randomly by the memory management system of a computer. A find operation would have to follow the pointers to the root and thus would traverse the memory in an unpredictable manner. This operation is typically slow.

A number of authors have suggested storing these rooted trees in arrays, because an array resides in consecutive memory locations [37], [30], [4]. Fig. 3 shows an example of such an array. Usually, the complexity of a union-find problem is defined as

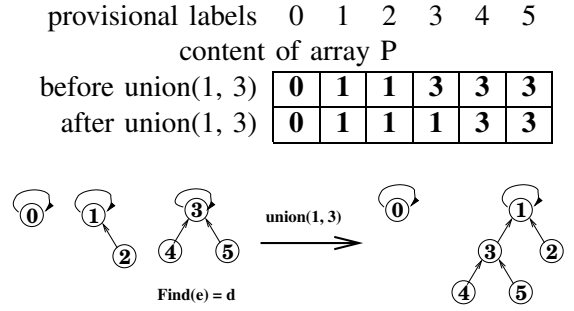


Fig. 3. An array representation of the rooted trees.

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 $\mathcal{O}(m)$. We say that a union-find is *linear* if it has $\mathcal{O}(m)$ time complexity. We also consider such an approach to be *optimal*. A naive approach may require $\mathcal{O}(mn)$ time. Two types of optimization techniques are commonly used to speed up the naive approach for union and find operations: path compression [38] and weighted union [36], [34]. In the worst case, all known union-find approaches require a superlinear time to perform m union and find operations [39], [35]. However, under some restricted settings [32], [40] or some inputs [30], [41], [42], m union and find operations may take a linear time. These approaches are generally cumbersome to implement with arrays.

Our proposed approach is based on two observations. First, using a single array, we can efficiently implement the union-find algorithms with path compression. Second, the union-find algorithms with only path compression (i.e., without weighted union) can also achieve the linear time complexity on average under certain conditions [30], [41]. Based on these observations, we expect our simple union-find approach to perform quite well. In fact, we can prove that our union-find approach takes a linear time (even in the worst case) to perform any m union and find operations in the connected component labeling algorithms. This enables the overall time complexity of a labeling algorithm to be linear in the number of pixels p, which is optimal for any labeling algorithm that takes a 2D array as input [7], [8], [11], [17].

B. Simplified algorithms

Following an example in the literature [30], [4], we call the array that contains the equivalence information the P array (short for parent links). Array P can be filled in a way similar to that of the connection table T introduced in Section II-B. In particular, every time a new provisional label is generated, array P is extended by one element by use of assignment statement $P[l] \leftarrow l$. 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 by using pseudo-codes. Our implementations, however, are in C++ with extensive use of C++'s Standard Template Library (STL)[2]. 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
```

[2]In C++ convention, all indices to arrays start from 0. The word *array* or *vector* in all pseudo-code segments is a shorthand 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 is typical in a pointer-based union-find definitions On most computer systems, the iterative functions can execute more efficiently than the equivalent recursive functions. In the 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
```

With functions `find` and `union`, Equation (3) can be redefined as follows:

$$L[\mathbf{e}] \leftarrow \begin{cases} \text{If } I[\mathbf{e}] = 0, \\ 0, \\ \text{if } \forall i \in (\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d}), I[i] = 0, \\ l, P[l] \leftarrow l, l \leftarrow l+1; \\ \text{otherwise}, \\ \min\limits_{i \in (\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d}) \mid I[i]=1} \left(\texttt{findRoot}(P, L[i])\right), \\ \quad \forall i \in (\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d}) \mid I[i] = 1, \\ \quad \texttt{setRoot}(P, L[i], L[\mathbf{e}]). \end{cases} \tag{8}$$

To use a decision tree as shown in Fig. 2, we need to define three functions used at the leaf nodes. We note that the new label function is the second case in the above equation and the one-argument copy function, copy(**a**), previously defined by Equation (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**) defined by Equation (6), can be redefined as

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

The above union function always selects the root with the smaller label as the root of the combined tree. This leads to the fact that the parent of a node always has a smaller label than its own label (i.e., P[i] $\leq$ i), and 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`.

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

Note that after the procedure `flattenL` is invoked, the array P no longer describes union-find trees. It can be used only to assign the final labels using Equation (11). It is easy to see that the final labels are consecutive, as shown in the next section.

If there is no need for consecutive labels, one may use the procedure `flatten` instead of the procedure `flattenL`. It is easy to see that procedure `flatten` is cheaper than `flattenL`.

After invoking the `flatten` procedure, one can use Equation (11) to assign the final labels as well. The result of calling the `flatten` procedure is that every node of union-find trees points to its root. One

Procedure `flattenL(P, size)`

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

important characteristics of both `flatten` and `flattenL` is that their computation complexities are not affected by the actual content of the array P. No matter how the union-find trees are shaped, the costs of both `flatten` and `flattenL` are the same. This may not be the case if the flattening procedure were implemented as a series of calls to the function `find`. We can employ these simpler flattening procedures because we have used a special union rule.

IV. Analyses

We now present some analyses to show the correctness of our proposed algorithms and their worst-case time complexities. 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 $\mathcal{O}(p)$. There is no need to flatten the union-find trees immediately after scanning each row as recommended in [8].

A. Correctness of algorithms

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

Lemma 1: The array P produced by the simplified 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. □

Lemma 2: The procedure `flatten` changes each node to point directly to the root of the tree containing it.

Proof: Following the previous lemma, it is straightforward to prove this by induction. □

The procedure `flatten` can be used to produce final labels for the 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 of `flattenL`.

Lemma 3: 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. Further, 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 the procedure `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-1] must contain the final label already, i.e., P[0] ... P[i-1] must contain all integers between 0 and $(k-1)$. At the i^{th} 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 $P[0] \dots P[i]$ 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. □

If the equivalence information is correctly captured, the above `union` and `find` functions can be used for producing the final labels by the use of Equation (11). The most straightforward way of capturing the equivalence information in the scanning phase is to visit all four neighbors in the scan mask, as described in Equation (8). Next, we prove that the use of a decision tree achieves the same goal.

Lemma 4: Let $S0$ denote the scanning phase without a decision tree, and let $S1$ denote the scanning phase with a decision tree. A connected component labeling algorithm using either S0 or S1 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 all union operations but could perform different find operations. We say that two union-find trees are equivalent if they contain the same 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.

To prove that S0 and S1 produce two equivalent sets of union-find trees, 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. To generalize this, we assume that S0 and S1 have produced equivalent sets of trees up to pixel **d** in the scan mask and 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 S0 are operating only 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 S0 and S1 are equivalent. □

Another way to interpret the above lemma is that a scanning procedure using a decision tree does all the necessary work. Our intuition is that it actually does a minimal amount of work. The following theorem formalizes this intuition.

Theorem 1: The use of a decision tree minimizes the number of neighbors visited during the scanning phase of a connected component labeling algorithm.

Proof: From the proof of Lemma 4, we know that with the use of a decision tree, the potential union operations are performed. All operations that are clearly find operations are avoided. To prove this theorem, we need to show two more facts: (1) the decision tree visits the minimal number of neighbors before deciding that a union operation is required, and (2) the union operations invoked cannot be replaced with less expensive operations even if they do not actually unite two trees.

To show the first fact, we observe that the provisional label of **c** may belong to a different union-find tree than those of **a** and **d**, only if **b** is a background pixel. Therefore, we cannot avoid visiting **b**. The decision tree invokes the two-argument copy function (i.e., the union operation) only if **c** and at least one of **a** and **d** are object pixels. To decide whether a union operations is necessary, one always needs to visit **c**. The choice of visiting **a** first in Fig. 2 is an arbitrary choice; we could easily find reasons for visiting **d** first. The important point is that the decision tree invokes the union operation as soon as it detects the first object pixel between **a** and **d**. This minimizes the number of neighbors visited before deciding that a union operation is required.

A union operations may actually degenerate into two find operations when the two input nodes belong to the same union-find tree. We could avoid these two find operations if we can detect the two input node belong to the same union-find tree. However, because detecting this requires exactly the same two find operations, the work performed by the `union` function can not be avoided even if

the two input nodes belong to the same union-find tree. Overall, the use of a decision tree minimizes the amount of work performed during the scanning phase. □

In the previous section, we applied a decision tree on the Scan with Connection Table (SCT) algorithm. To see that SCT with a decision tree (SCT-1) would eventually produce the same labels as the straightforward SCT-4, we observe that the connection table T essentially captures the same information as array P used in our simplified union-find. The main difference is that SCT relies on repeated scans to achieve the effect of the path compression used in the union-find.

This similarity between the connection table and the union-find trees also suggests that the number of iterations required by SCT is related to the height of the union-find trees. However, a rigorous analysis is complicated by the change of scanning directions in SCT. We leave that analysis for future work.

B. Worst-case complexity

Fiorio and Gustedt [8] have proved that the worst-case time complexity of a two-pass algorithm with the path compression in its union-find is $\mathcal{O}(p)$, where p is an number of pixels in the image. A key in their approach is that they flatten the union-find trees after scanning each row of the image. After each object pixel in a row receives a provisional label, their algorithm revisits each active label and applies the find operation with path compression on the provisional labels. This may add another pass through the image. Our thesis is that this extra pass through the image is not necessary. Earlier in this section, we 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 algorithm. Next, we show that a two-pass algorithm using *any* union-find with path compression has the same $\mathcal{O}(p)$ time complexity with or without flattening the active trees after scanning each row.

Our analysis of the worst-case time complexity proceeds from the third phase of the two-pass algorithm to the first phase. It is obvious that the labeling phase as shown in Equation (11) requires $\mathcal{O}(p)$ time. We next show that the analysis phase requires $\mathcal{O}(p)$ time at most and then prove that the scanning phase requires $\mathcal{O}(p)$ time also.

To be able to give precise expressions for the cost of some operations, 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 requires. We use the path lengths of the united tree to measure the cost of the find operations. However, it is possible to use find paths before the union operations as well without affecting the final cost analysis.

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. Let the degree (i.e., the number of children) of node i be d_i. 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 scheme 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 no more than the number of object pixels, which, in turn, is no 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 at most $3p$. This proves the following lemma regarding the computational complexity of the analysis phase.

Lemma 5: The worst-case time of the analysis phase of a two-pass connected component labeling algorithm using a union-find with pass compression is $\mathcal{O}(p)$, where p is the number of pixels in the image being labeled.

After flattening of a union-find tree, any subsequent find operation costs at most 2. If there is ever a need to perform multiple iterations of find operations on each node, the total cost of each iteration is proportional to the number of nodes in the trees. The proportionality constant for the first iteration is about 50% larger than that of subsequent iterations.

Next, we examine the cost of the scanning phase. We have shown (see Lemma 4) that using a decision tree produces the same final labels as using the straightforward scanning strategy, and the above lemma indicates that the two strategies do not change the cost of the analysis phase and the labeling phase. Therefore, we choose the simple scanning strategy for the next analysis. This simple scanning strategy would invoke find operations on the provisional labels of all object pixels in the scan mask, perform union operations if necessary, and assign the label of the root node (of a possibly new united tree) to pixel **e**. This procedure is similar to that defined by Equation (8), but may use different union rules and different union-find data structures.

Lemma 6: In using the simple forward scan procedure to assign provisional labels in a two-pass algorithm, the provisional labels assigned to object pixels, after scanning a row, are either at the roots of union-find trees or connected to roots through provisional labels used in the row.

Proof: After scanning of the first row, each union-find tree contains a single node. The above lemma is clearly true. We next examine what happens while scanning an arbitrary row i. By construction of the scanning procedure, when a particular pixel is assigned a provisional label, the label must be the root of a union-find tree. 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. Using the names given in a scan mask, we can describe such a union operation as follows. The label assigned to pixel **d** was a root of a tree when the assignment was performed. While a label is determined 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 of newly formed trees through other labels that have been used more recently. □

Let the term *active labels* refer to the provisional labels assigned to the pixels of a row just scanned. The above lemma indicates that the active labels occupy the top of the union-find trees and form a set of valid union-find trees of their own, which we call *active trees*. If we flatten the active trees as suggested by Fiorio and Gustedt [8], the total cost, in the worst case, is proportional to the number of object pixels in the row according to Theorem 2. We can amortize the cost of this flattening operation to the object pixels in the last row scanned. This adds a constant cost of the operation of each object pixel. Because future scan operations involve only the active labels, after flattening of the active trees, each future find operation costs at most 2. Therefore, the cost of assigning a provisional label and the cost of a union operation are constants. This proves the following lemma, which generalizes an earlier result in [8] by allowing any union-find with path compression.

Lemma 7: The total cost of a scanning phase *with flattening* of active trees after scanning each row is $\mathcal{O}(p)$.

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 is no more than the total cost of a scanning phase with the flattening of active trees. 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 separately, the procedure of assigning a new label may invoke them anyway. With the flattening of active trees, we perform two sets of find operations. Without the flattening of active trees, we perform only one set of find operations.

Lemma 8: The total cost of a scanning phase *without flattening* of active trees is $\mathcal{O}(p)$.

Proof: In the process of assigning a provisional label to the pixel **e**, it is necessary to

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 to its starting pixels **a**, **b**, **c**, or **d**. This leaves a small constant cost of assigning the provisional label (and possibly linking two trees) 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, the total cost of all find operations is at worst proportional to the number of active labels, which is no more than the number of object pixels in the row of the image. 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 $\mathcal{O}(p)$. □

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

Proof: A two-path algorithm can be divided into three phases: scanning, analysis, and labeling. Lemmas 7 and 8 show that the scanning phase takes at most $\mathcal{O}(p)$ time with or without flattening of active trees. Lemma 5 shows that the analysis phase takes $\mathcal{O}(p)$ 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 in Equation (11), obviously takes $\mathcal{O}(p)$ time. Overall, the total time is at worst $\mathcal{O}(p)$. □

C. Expected performance on random images

Next, we study the expected performance of the Scan plus Array-based Union-Find (SAUF) algorithm on random binary images. The random images considered here contain n rows and m columns, and each pixel has a probability q of being an object pixel. We also refer to q as the density of object pixels. The total number of pixels is $p = mn$, of which qmn are expected to be object pixels.

As a way of illustrating the probability model used, 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 $(1-q)$ for being a background pixel. Assuming that each pixel is generated independently, the probability for all four pixels to be background pixels is $(1-q)^4$. In a 2D image, pixels normally have four neighbors in the 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 actually five different scan masks used during a forward scan. An illustration of these five scan masks is shown in Table I. The same table also lists the number of instances (in column 3 under the heading of **instances**) for each case and the probabilities of an object pixel receiving a new label (in column 7 under the heading of **labels**). Multiplying the density q and the values in columns 3 and 7 of Table I, 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:

$$n_p = q\left(1 + (m-1)(1-q) + (n-1)(1-q)^2 + (n-1)(1-q)^3 + (m-2)(n-1)(1-q)^4\right). \tag{12}$$

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

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 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, which is visiting the neighbors according to a decision tree. The process of traversing a decision tree requires multiple if-tests and a non-trivial amount of time. Because each if-test is for

TABLE I

THE EXPECTED NUMBERS OF OPERATIONS PER OBJECT PIXEL USED BY THE SAUF ALGORITHM.

			expected values			
	mask	**instances**	3) **neighbors**	4) **copy**	5) **union**	6) **labels**
1	**e**	1	0	1	0	1
2	**d e**	$m-1$	1	q	0	$1-q$
3	**b c** **e**	$n-1$	$2-q$	$q(2-q)$	0	$(1-q)^2$
4	**a b** **d e**	$n-1$	$3-3q+q^2$	$1-(1-q)^3$	0	$(1-q)^3$
5	**a b c** **d e**	$(m-2)(n-1)$	$(2-q)^2$	$4q-8q^2+7q^3-2q^4$	$q^2(1-q)(2-q)$	$(1-q)^4$

a different neighbor, the amount of time required in this category should be proportional to the number of neighbors visited during the scanning phase. The expected number of neighbors to be visited for each object pixel is shown in column 4 under the heading of **neighbors** in Table I.

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. To account for time spent in this category of work, we need to estimate the number of times a copy or an assignment is performed. The expected number of copy (or new label) operations to be performed for each object pixel is shown in column 5 under the heading of **copy** in Table I.
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. To account for the total cost of all union operations, we need to estimate the number of union operations performed. The probability of performing a union operation for each object pixel is shown in column 6 under the heading of **union** in Table I.
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 7 under the heading of **labels** in Table I.

Next, we use the same probability model used for estimating the number of provisional labels to estimate the work of categories 3, 4, and 5. In category 3, the number of if-tests performed is the number of neighbors visited. For the normal case, the computation of these quantities are based on the decision tree shown in Fig. 2(b). We associate each edge labeled "1" with the probability q and each edge labeled "0" with the probability $(1-q)$. Take the example of computing the number of if-tests required to reach a leaf of the decision tree. There is one path from the root to a leaf that is of length 1 (i.e., if **b** is an object pixel). 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 taking a path of length 3 is $(1-q)q$. There are four paths of length 4. The probabilities of taking each of these four paths are $(1-q)^4$, $(1-q)^3q$, $(1-q)^3q$, and $(1-q)^2q^2$. The total probability of taking a path of length 4 is $(1-q)^2$. The average path length (or the average number of neighbors visited) 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 I. Among the seven paths, there are two leads to a two-argument copy function, which are better known as the union operation. The two paths have probabilities of $(1-q)q^2$ and $(1-q)^2q^2$. The total probability of invoking a union function 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 leading to either a simply copy function or a new label function, we enter their total probability under the heading of **copy**. We have repeated the same evaluation for all four

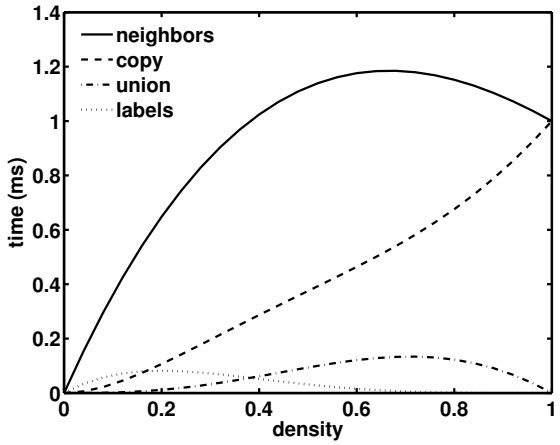


Fig. 4. Relative contributions from different categories of work performed by SAUF.

TABLE II
THE DOMINANT TERMS FOR DIFFERENT CATEGORIES OF WORK PERFORMED BY SAUF AND THEIR AVERAGES OVER ALL POSSIBLE q.

	formula	average
3) **neighbors**	$q(2-q)^2$	11/12
4) **copy**	$q^2(4-8q+7q^2-2q^3)$	2/5
5) **union**	$q^3(1-q)(2-q)$	1/15
6) **labels**	$q(1-q)^4$	1/30

other cases and entered the probabilities in Table I.

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 ($q \to 0$) and large ($q \to 1$) densities. This agrees with our expectation.

Theorem 4: Following a decision tree to determine a provisional label for an object pixel of a typical random 2D image, $7/3$ neighboring pixels are visited on average.

Proof: In the normal case, the number of neighbors visited is a simple 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 $7/3$. □

To show the relative importance of categories 3 – 6 defined on page 13, we display their probabilities in the normal case multiplied by the density q in Fig. 4. If the average cost per operation are about the same, we see that the total cost of category 6 is the lowest and that of category 3 is highest. Table II gives the average values of the functions shown in Fig. 4.

Based on the probabilities shown in Table I, we have the following formula for the expected execution time of SAUF, where the constants C_1, …, C_6 represent the average cost per operation of the six categories identified (n_p was defined in Equation (12)).

$$\begin{aligned} p &= mn, \\ n_o &= qmn, \\ n_n &= q\left(m-1+(n-1)(5-4q+q^2)+\right. \\ &\quad \left.(m-2)(n-1)(2-q)^2\right), \\ n_c &= q+q^2\left(m-1+(n-1)(5-4q+q^2)\right. \\ &\quad \left.+(m-2)(n-1)(4-8q+7q^2-2q^3)\right), \\ n_u &= q^3(1-q)(2-q)(m-2)(n-1), \\ t_S &= C_1p+C_2n_o+C_3n_n+C_4n_c+ \\ &\quad C_5n_u+C_6n_p. \end{aligned} \tag{13}$$

In the next section, we will use timing results to estimate the constants C_1, …, C_6 on different test machines.

V. EXPERIMENTAL RESULTS

In this section, we report the timing measurements of our software implementation of various connected component labeling algorithms. We also verify the performance model developed for SAUF on random images. The decision tree shown in Fig. 2(b) was implemented in all test programs that requires a decision tree.

A. Test setup

To measure the performance of various 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 [43]. For this study, we used three additional sets of images from various applications. Some sample images are shown in Fig. 5, and summary descriptions of these images are given in Table III. We applied Otsu thresholding [44] on the intensity to turn the application images into binary images. The random binary images used in

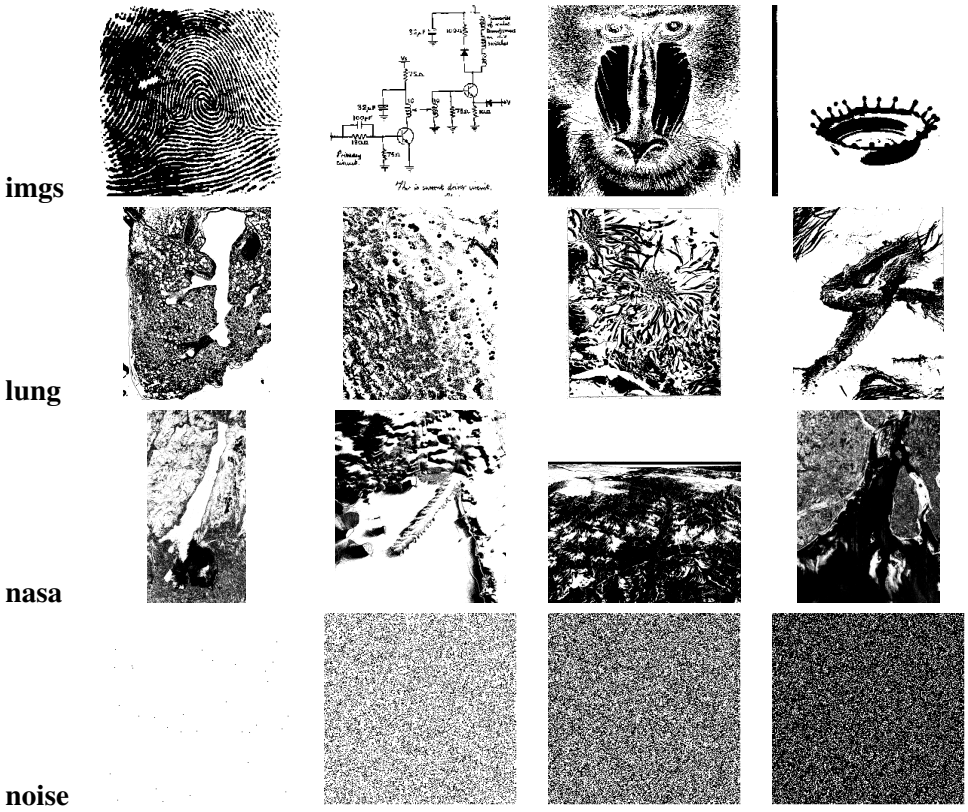


Fig. 5. A sample of the binary images used in tests. Object pixels are shown as black.

TABLE III

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 [11]
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 x 500, 1000 x 1000, 2000 x 2000)

this study were smaller than in our previous study, so that they were closer to the application images in size. Testing on these images may better reflect what can be expected in a real application. The test image set **imgs** also included some pathologic cases (illustrated in Fig. 6) used in the analysis by Suzuki et al. [11]. These images were used in part for verifying the correctness of the programs.

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 as listed in Table IV. With each machine, we also chose to use a different compiler. This should make it easier to identify the differences in performance due to the algorithmic differences.

B. Timing the multi-pass algorithms

We implemented three variants of the Scan plus Connection Table algorithm, namely, SCT-4, SCT-2 and SCT-1, in software using C++ programming language, and timed them on the three machines listed in Table IV. A summary of the timing results is given in Table V. Because the four sets of test images have significantly different sizes, we showed the average time for each set separately. The timing measurements were made for each test image. The test on each image was repeated enough

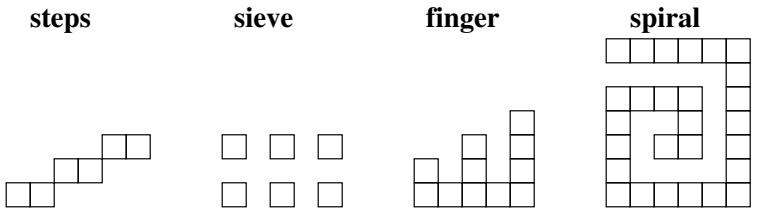


Fig. 6. Some pathologic test (pixel) patterns included in image set **imgs**.

TABLE IV
INFORMATION ABOUT THE TEST MACHINES.

CPU type	Clock (MHz)	Cache (KB)	Memory (MB)	OS	Compiler
UltraSPARC	450	4096	4096	Solaris 8	Forte workshop 7
Pentium 4	2200	512	512	Linux 2.4	gcc 3.3.3
Athlon 64	2000	1024	512	Windows XP	Visual Studio .NET

times so that at least one second is used. A minimum of five iterations was always used. The time values reported are wall clock time. The speedup of SCT-2 and SCT-1 were measured against SCT-4. Each speedup value is computed for one test image and the speedup values reported in Table V are averages.

On each test platform, the three algorithms, SCT-4, SCT-2, 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 connected components. This irregularity slightly reduces the effectiveness of both SCT-1 and SCT-2.

On the application images, SCT-1 is about twice as fast as SCT-4 and about 20% faster than SCT-2. Theorem 4 states that the average number of neighbors visited with the use of a decision tree is $7/3$. Since SCT-4 always visits 4 neighbors, we expect a speed up of $12/7$, i.e., about 1.7. The actual observed speedup value shown in Table V is close to 1.7 for random images. The actual speedups are larger (around 2) for application images. In all test cases, SCT-1 is never slower than either SCT-2 or SCT-4. For this reason, we used a decision tree in all of the subsequent tests.

TABLE V
SUMMARY OF TIMING MEASUREMENTS ON THE THREE MULTI-PASS ALGORITHMS. THE TIME VALUES ARE IN MILLISECONDS AND THE SPEEDUP VALUES ARE RELATIVE TO SCT-4.

	SCT-4	SCT-2		SCT-1	
	Time	Time	Speedup	Time	Speedup
UltraSPARC					
imgs	96	52	1.8	44	2.1
lung	215	118	1.8	95	2.3
nasa	5776	2946	1.9	2880	2.1
noise	940	571	1.6	501	1.9
Pentium 4					
imgs	15	9	1.6	8	2.0
lung	29	19	1.5	14	2.2
nasa	782	433	1.7	383	2.1
noise	173	117	1.5	97	1.8
Athlon 64					
imgs	14	10	1.4	8	1.7
lung	26	19	1.3	14	1.8
nasa	687	454	1.5	394	1.8
noise	141	110	1.3	87	1.7

C. Timing the two-pass algorithms

In this subsection, we compare the new Scan plus Array-based Union-Find (SAUF) algorithm with other two-pass algorithms that uses a pointer-based union-find with both path compression and weighted union. One of the algorithms flattens the active union-find trees after scanning each row of the image as suggested by Fiorio and Gustedt [8]. We refer to this algorithm as SUF1. The other, which does not perform the extra flattening operation, is referred to as SUF0. In their analysis, Fiorio and Gustedt concluded that flattening the active trees after scanning each row is important to reduce the worst-case time complexity [8]. Earlier in this paper, we presented a refined analysis and

TABLE VI
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	5279	3231	1.6	1164	4.6
noise	1056	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	1506	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	1073	429	2.4	134	7.5
noise	237	140	1.9	34	6.5

showed that this extra flattening is unnecessary. The timing measurements shown in Table VI confirm our analysis.

As in the previous table, Table VI reports the elapsed time used by various algorithms. 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 expected. Our new labeling algorithm SAUF was usually four times or more as fast as SUF1, and about twice as fast as SUF0. The performance difference was even larger when many provisional labels were combined into a small number of final labels, as in the test image set **lung**.

D. Comparison with contour tracing algorithms

The Contour Tracing algorithm is one of the most efficient algorithms for connected component labeling [7]. In this subsection, we present some timing results of the Contour Tracing algorithm and justify the performance characteristics of the SAUF algorithm and the Contour Tracing algorithm.

In our tests, we used two versions of the Contour Tracing algorithm. The first is from the original authors of the algorithm, and this is referred to as CTo (for CT original)[1]. The second is our implementation of an in-place version of the algorithm referred to as CTi (for CT in-place). The original version places the input image in a larger array so as to avoid the need to check whether a pixel is on the boundary of the image. It is expected to take less time than the in-place version, but it requires more memory. The in-place version avoids the need for copying the image into a large array, but uses more if-tests to check for pixels on the image boundary. Because the input array to our labeling algorithms contains only 0 and 1, CTi starts to label the object pixels with the number 2 rather than 1. This allows us to distinguish easily the pixels that have been labeled from those that have not. Because the Contour Tracing algorithm also marks some background pixels as -1 to indicate that they have been visited, to produce the same output as other algorithms, our in-place version needs to change the value -1 back to 0. We also take the opportunity to reduce all positive labels by 1. This change makes CTi a two-pass algorithm rather than a one-pass algorithm.

TABLE VII
THE AVERAGE SPEEDUP OF SAUF OVER CTO. THE OVERALL AVERAGE SPEEDUP IS 1.5.

	UltraSPARC	**Pentium 4**	**Athlon 64**
imgs	1.0	0.8	1.0
lung	2.4	2.1	2.7
nasa	0.8	1.3	1.4
noise	1.5	1.3	1.7

Table VIII shows the average time used by SAUF and the two version of Contour Tracing algorithms. On the three larger sets of test images, SAUF usually uses less time than do CTo and CTi. Table VII shows the performance of SAUF relative to CTo. In 8 out of the 12 cases shown in Table VII, SAUF is noticeably faster than CTo. Of the three sets of large images, the images in the set named **nasa** are scenery photos which have more well-defined connected components than do the connected components in images from **lung** and **noise** sets. The Contour Tracing algorithm was relatively more efficient in identifying these well-defined components because there are fewer pixels on the boundaries of the components. For the smaller images, SAUF and CTo performs about the same overall.

TABLE VIII
AVERAGE TIME (IN MILLISECONDS) USED BY CTO, CTI, AND SAUF TO LABEL THE TEST IMAGES.

	UltraSPARC			Pentium 4			Athlon 64		
	CTo	CTi	SAUF	CTo	CTi	SAUF	CTo	CTi	SAUF
imgs	21	28	22	4	7	5	4	5	4
lung	127	158	53	21	32	10	20	24	7
nasa	793	1294	1164	358	258	182	191	170	134
noise	327	468	243	67	97	47	59	74	34

TABLE IX
AVERAGE TIME (IN MILLISECONDS) USED BY CTO, CTI, AND SAUF TO LABEL THE FOUR PATHOLOGIC TEST IMAGES.

	UltraSPARC			Pentium 4			Athlon 64		
	CTo	CTi	SAUF	CTo	CTi	SAUF	CTo	CTi	SAUF
steps	14	15	8	2	2	1	2	2	1
sieve	17	16	8	2	3	1	2	2	1
finger	78	71	36	11	15	6	11	11	5
spiral	98	88	26	15	31	5	14	18	4

As shown in Table IX, on the four pathological test images illustrated in Fig. 6, SAUF has an average speedup of 2.4 over CTo.

The algorithm SAUF contains both optimization strategies, a decision tree to minimize work during scanning phase and the simplified algorithms to reduce the time spent in union-find. From Table VII, we see that it is often twice as fast as the original version of the Contour Tracing algorithm. The average speedup of SAUF over CTo, across the four sets of test images and on three machines, is about 1.5. In a previous test [7], the approach of Scan plus Union-Find was shown to take an average of 60% more time than the Contour Tracing (CTo) algorithm. Because CTo is exactly the same program that was used in [7], clearly the optimization strategies did pay off.

Even though the input data to all test cases fit in the memory of all three machines, we observed many cases where the in-place version CTi was faster than the original version CTo. This counters the expectation that CTo is faster than CTi. This unexpected observation can be explained as follows. In the recent years, the increase in the CPU speed has significantly outpaced the increase in memory speed. This makes operations in CPU relatively cheaper than memory accesses, which make the in-place version more attractive today than in the past.

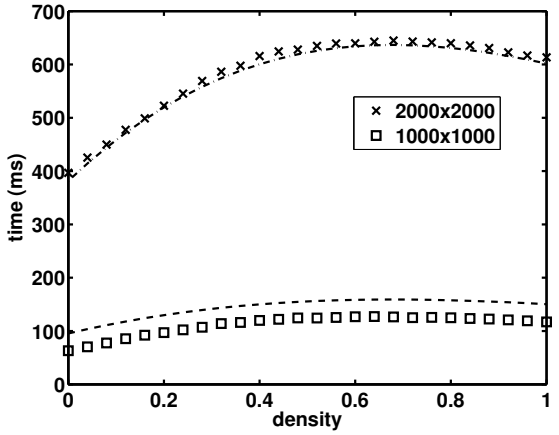
There are also other considerations in favor of using CTi as well. In our tests, the images are stored in files and are read into memory for each test. The time used for reading the images is not reported because we intended to compare the labeling algorithms and not the I/O speed of the systems. In the implementation of CTo, an image is directly read into a larger array. If the input image is already in memory, one has to copy the image into a larger array. The in-place version can avoid this copying and therefore could be more competitive. In addition, if the down-stream analysis function can cope with the negative number (-1) used to mark some background pixels, it would not be necessary to have the second pass through the image array in CTi. In this case, the in-place version would be even more competitive against the original version.

TABLE X
THE CONSTANT VALUES (10^{-8} SECONDS) OF EQUATION (13) PRODUCED WITH A CONSTRAINED LEAST-SQUARE FITTING OF MEASURED TIME VALUES.

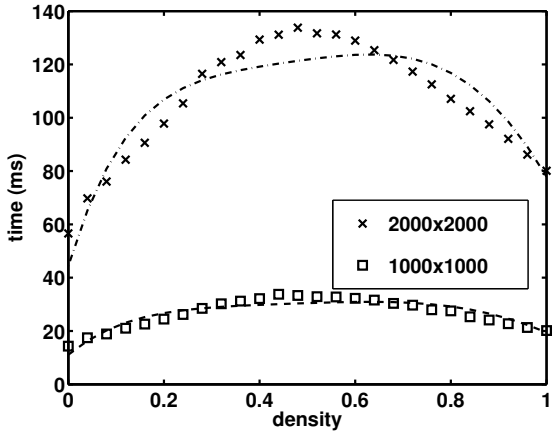
	UltraSPARC	Pentium 4	Athlon 64
C_1	9.5	1.1	1.2
C_2	0	0	0
C_3	5.3	0.8	3.7
C_4	0.2	0	0
C_5	0	6.7	4.1
C_6	0	11.5	5.4

E. Performance on random images

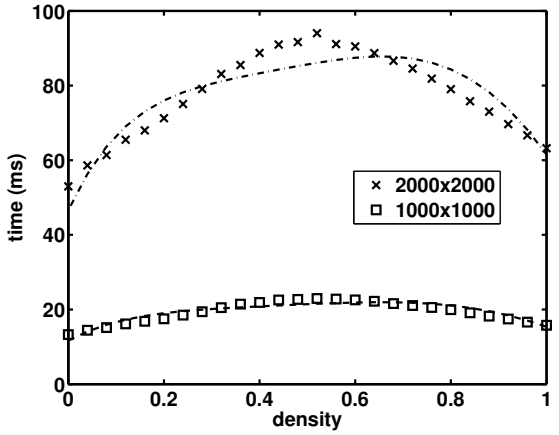
We developed a performance model for the time needed by SAUF to label random binary images. Next, we show some timing measurements that support the performance model.



UltraSPARC



Pentium 4



Athlon 64

Fig. 7. The measured time (in milliseconds) used by SAUF agrees with the performance model (shown as broken lines) described by Equation (13).

As shown in Table III, we used 78 random binary images of various sizes in our 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, ..., C_6 for each machine. The computation used a linear least-square formulation to minimize the fitting error with a non-negative constraint[3]. The results of C_1, ..., C_6 for all three test machines are shown in Table X. Because the three computers used different types of CPUs and different operating systems, and because our performance model does not capture some important factors like cache sizes, memory bandwidth, memory access latency, and so on, we expect these constants to be different for different machines. Category 2 was introduced as a catch-all category. The value of C_2 is computed to be 0 on all three machines, which indicates that the other five categories model the performance quite well.

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 for visiting a neighboring pixel during the scanning phase. The process of visiting a neighbor involves accessing the pixel value of the neighbor and performing an if-test on it. Both of these operations consume a number of clock cycles. The cost of visiting neighbors dominates the overall shape of the timing curves shown in Fig. 7.

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 in as shown in Table X. The values C_5 and C_6 are zero for the **UltraSPARC**, but are nonzero for the two others. This is likely due to the different sizes of CPU caches on these machines as shown in Table IV. Because the terms involving these two constants are relatively small as shown in Fig. 4 and Table II, it is also likely that the curve-fitting errors have a stronger influence on C_5 and C_6 than on C_1 and C_3.

With the six constants shown in Table X, we can use Equation (13) to compute the expected time. In Fig. 7, we show the measured time along with the

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

expected time. We see that the expected time agrees with the measured time to within 10% in most cases. A case with noticeable discrepancy occurred when random images of size 1000 x 1000 were labeled on the **UltraSPARC**. In this particular case, the estimated time is about 1/4 larger than the actual measured time. Considering that there are many important factors not captured by the performance model and that we used the same constants for images of different sizes, this discrepancy is not unexpected.

In Fig. 8, we use the random images to illustrate the relative strengths of SAUF, CTi and CTo. This figure can be considered a more detailed view of the last row of Table VIII. In this figure, the horizontal axes are densities, which are fractions of pixels that are object pixels. The worst-case time complexity of all three of the algorithms are linear in the number of pixels in the image. However, the worst-case linear relations were hardly ever achieved in the tests conducted. The Contour Tracing algorithms perform more work on boundary pixels, and as a result they should take longer on images with more pixels on the boundaries of the connected components. For random images, when the object pixel density is near a half, we find more pixels in the boundary. Therefore, the two versions of the Contour Tracing algorithm took the longest time when the density was nearly one half. SAUF shows less dependency on the density q. Overall, we see that CTo took less time than SAUF when q is either very small or nearly 1. For a large range of densities from 0.1 to 0.9, SAUF is significantly faster than both CTi and CTo. When the density is near 0.5, SAUF can be 3 to 4 times faster than CTo.

The estimated number of provisional labels for random images is given in Equation (12). As a sanity check for the performance model, we compare this estimated number of provisional labels against the actually observed number. We plotted the estimated and the observed number of provisional labels in Fig. 9(a). The estimated values are close to the observed values for $q < 0.2$. For higher densities, the differences between estimated and observed values become more pronounced. These differences are due to the fact that the independence assumption becomes more unreliable as q increases.

Because each union operation is likely to reduce the number of final labels by one, we can subtract the estimated number of union operations from the number of provisional labels to produce an

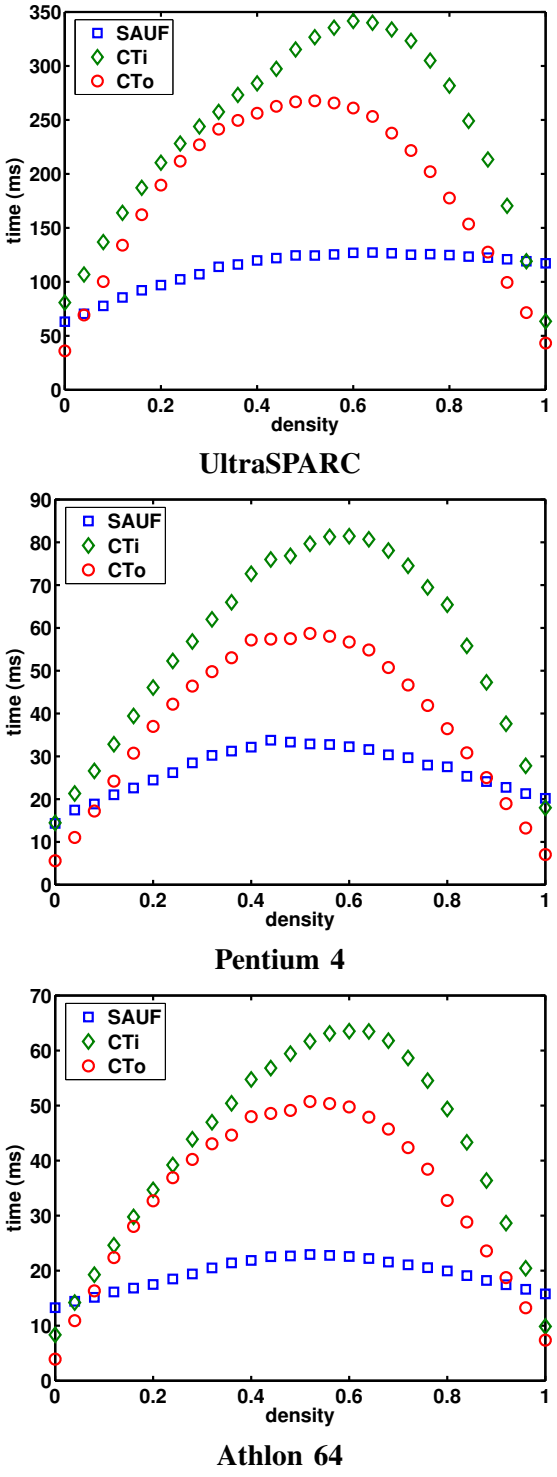
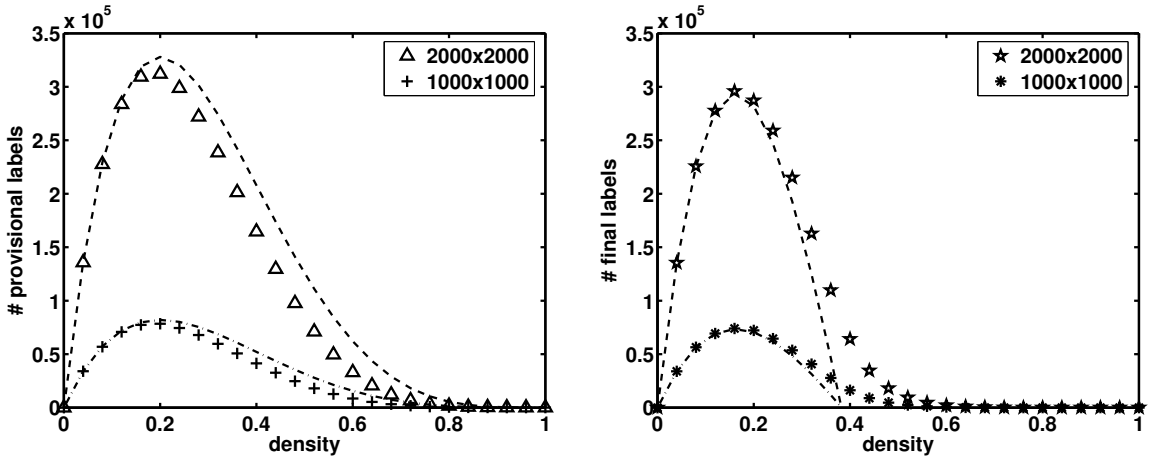


Fig. 8. Time (in milliseconds) used for labeling random images with different densities of object pixels (image size 1000 x 1000).



(a) Provisional labels **(b)** Final labels

Fig. 9. The actual number of provisional labels and final labels observed plotted with estimated labels shown as broken lines.

estimate of the number of final labels (or number of components). The estimate is shown in Fig. 9(b) as broken lines. Because many union operations actually involve two provisional labels that already belong to the same union-find tree, our estimation of the number of final labels is an underestimation, as illustrated in Fig. 9(b).

VI. 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; whereas 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 same worst-case time complexity as do the best-known labeling algorithms. We also showed with extensive tests that the new two-pass algorithm named SAUF significantly outperforms other well-known two-pass algorithms and multi-pass algorithms. It even outperforms the Contour Tracing algorithm by 50% on average. The new algorithm is relatively straightforward to implement. 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 would be helpful to formalize the arguments given in the previous section to decide when to use the Contour Tracing algorithm and when to use SAUF. A derivation of a bound on the maximum number of scans needed by the SCT algorithm, as mentioned in Section IV, 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.

Acknowledgments

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] D. H. Ballard, *Computer Vision*. Englewood, New Jesey: Prentice-Hall, 1982.

[2] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 2nd ed. New Jerse: Prentice Hall, 2002.

[3] A. Rosenfeld and A. C. Kak, *Digital Picture Processing*, 2nd ed. San Diego, CA: Academic Press, 1982.

[4] G. C. Stockman and L. G. Shapiro, *Computer Vision*. Englewood, New Jesey: Prentice Hall, 2001.

[5] H. M. Alnuweiri and V. K. Prasanna, "Parallel architectures and algorithms for image component labeling," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, no. 10, pp. 1014–1034, 1992.

[6] H. M. Alnuweiri and V. K. P. Kumar, "Fast image labeling using local operators on mesh-connectedcomputers," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 13, no. 2, pp. 202–207, 1991.

[7] F. Chang, C.-J. Chen, and C.-J. Lu, "A linear-time component-labeling algorithm using contour tracing technique," *Comput. Vis. Image Underst.*, vol. 93, no. 2, pp. 206–220, 2004.

[8] C. Fiorio and J. Gustedt, "Two linear time union-find strategies for image processing," *Theor. Comput. Sci.*, vol. 154, no. 2, pp. 165–181, 1996.

[9] F. Knop and V. Rego, "Parallel labeling of three-dimensional clusters on networks of workstations," *Journal of Parallel and Distributed Computing*, vol. 49, no. 2, pp. 182–203, March 1998.

[10] A. N. Moga and M. Gabbouj, "Parallel image component labeling with watershed transformations," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, no. 5, pp. 441–450, 1997.

[11] K. Suzuki, I. Horiba, and N. Sugie, "Linear-time connected-component labeling based on sequential local operations," *Comput. Vis. Image Underst.*, vol. 89, no. 1, pp. 1–23, 2003.

[12] R. M. Haralick and L. G. Shapiro, "Image segmentation techniques," *Computer Vision, Graphics, and Image Processing*, vol. 29, no. 1, pp. 100–132, Jan. 1985.

[13] J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 8, pp. 888–905, August 2000.

[14] Y. Wang and P. Bhattacharya, "Using connected components to guide image understanding and segmentation," *Machine Graphics & Vision*, vol. 12, no. 2, pp. 163–186, 2003.

[15] A. Rosenfeld, "Connectivity in digital pictures," *J. ACM*, vol. 17, no. 1, pp. 146–160, 1970.

[16] R. M. Haralick, *Some Neighborhood Operations*. New York: Plenum Press, 1981, pp. 11–35.

[17] M. B. Dillencourt, H. Samet, and M. Tamminen, "A general approach to connected-component labeling for arbitrary image representations," *J. ACM*, vol. 39, no. 2, pp. 253–280, 1992.

[18] T. Gotoh, Y. Ohta, M. Yoshida, and Y. Shirai, "Component labeling algorithm for video rate processing," in *Proc. SPIE 1987*, ser. Advances in Image Processing, vol. 804, 1987, pp. 217–224.

[19] R. Lumia, "A new three-dimensional connected components algorithm," *Comput. Vision, Graphics, and Image Process*, vol. 23, no. 2, pp. 207–217, 1983.

[20] R. Lumia, L. Shapiro, and O. Zungia, "A new connected components algorithm for virtual memory computers," *Comput. Vision, Graphics, and Image Process*, vol. 22, no. 2, pp. 287–300, 1983.

[21] S. Naoi, "High-speed labeling method using adaptive variable window size for character shape feature," in *IEEE Asian Conference on Computer Vision, 1995, vol. 1*, 1995, pp. 408–411.

[22] J. Hecquard and R. Acharya, "Connected component labeling with linear octree," *Pattern Recogn.*, vol. 24, no. 6, pp. 515–531, 1991.

[23] H. Samet, "Connected component labeling using quadtrees," *J. ACM*, vol. 28, no. 3, pp. 487–501, 1981.

[24] H. Samet and M. Tamminen, "An improved approach to connected component labeling of images," in *Proceedings, CVPR '86 (IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Miami Beach, FL, June 22–26, 1986)*. IEEE, 1986, pp. 312–318.

[25] ——, "Efficient component labeling of images of arbitrary dimension represented by linear bintrees," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 10, no. 4, pp. 579–586, 1988.

[26] P. Bhattacharya, "Connected component labeling for binary images on a reconfigurable mesh architecture," *J. Syst. Archit.*, vol. 42, no. 4, pp. 309–313, 1996.

[27] R. Cypher, J. Sanz, and L. Snyder, "An erew pram algorithm for image component labeling," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, no. 3, pp. 258–262, March 1989.

[28] S. Alstrup, A. M. Ben-Amram, and T. Rauhe, "Worst-case and amortised optimality in union-find," in *Proc. 31th Annual ACM Symposium on Theory of Computing (STOC'99)*. ACM Press, 1999, pp. 499–506.

[29] B. Bollobás and I. Simon, "On the expected behavior of disjoint set union algorithms," in *STOC '85: Proceedings of the seventeenth annual ACM symposium on Theory of computing*. ACM Press, 1985, pp. 224–231.

[30] J. Doyle and R. L. Rivest, "Linear expected time of a simple union-find algorithm." *Inf. Process. Lett.*, vol. 5, no. 5, pp. 146–148, 1976.

[31] C. Fiorio and J. Gustedt, "Memory management for union-find algorithms," in *Proceedings of 14th Symposium on Theoretical Aspects of Computer Science*. Springer-Verlag, 1997, pp. 67–79.

[32] H. N. Gabow and R. E. Tarjan, "A linear-time algorithm for a special case of disjoint set union," in *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*. ACM Press, 1983, pp. 246–251.

[33] Z. Galil and G. F. Italiano, "Data structures and algorithms for disjoint set union problems," *ACM Comput. Surv.*, vol. 23, no. 3, pp. 319–344, 1991.

[34] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *J. ACM*, vol. 22, no. 2, pp. 215–225, 1975.

[35] R. E. Tarjan and J. van Leeuwen, "Worst-case analysis of set union algorithms," *J. ACM*, vol. 31, no. 2, pp. 245–281, 1984.

[36] B. A. Galler and M. J. Fisher, "An improved equivalence algorithm," *Commun. ACM*, vol. 7, no. 5, pp. 301–303, 1964.

[37] A. V. Aho, J. D. Ullman, and J. E. Hopcroft, *Data Structures and Algorithms*. Addison Wesley, 1983.

[38] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, Mass.: Addison - Wesley, 1974.

[39] R. E. Tarjan, "Reference machines require non-linear time to maintain disjoint sets," in *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*. ACM Press, 1977, pp. 18–29.

[40] J. M. Lucas, "Postorder disjoint set union is linear," *SIAM J. Comput.*, vol. 19, no. 5, pp. 868–882, 1990.

[41] A. C. Yao, "On the expected performance of path compression algorithms," *SIAM J. Comput.*, vol. 14, no. 1, pp. 129–133, 1985.

[42] D. E. Knuth and A. Schönhage, "The expected linearity of a simple equivalence algorithm." *Theor. Comput. Sci.*, vol. 6, pp. 281–315, 1978.

[43] K. Wu, E. Otoo, and A. Shoshani, "Optimizing connected component labeling algorithms," in *Proceedings of SPIE Medical Imaging Conference 2005, San Diego, CA*, 2005, a draft appeared as LBNL report LBNL-56864.

[44] N. Otsu, "A threshold selection method from gray level histograms," *IEEE Trans. Systems, Man and Cybernetics*, vol. 9, pp. 62–66, Mar. 1979.