A General Approach to Connected-Component Labeling for Arbitrary Image Representations

MICHAEL B. DILLENCOURT

University of California, Irvine California

HANNAN SAMET

University of Maryland, College Park, Maryland

AND

MARKKU TAMMINEN

Helsinki University of Technology, Espoo, Finland

Abstract. An improved and general approach to connected-component labeling of images is presented. The algorithm presented in this paper processes images in *predetermined order*, which means that the processing order depends only on the image representation scheme and not on specific properties of the image. The algorithm handles a wide variety of image representation schemes (rasters, run lengths, quadtrees, bintrees, etc.). How to adapt the standard UNION–FIND algorithm to permit reuse of temporary labels is shown. This is done using a technique called *age balancing*, in which, when two labels are merged, the older label becomes the father of the younger label. This technique can be made to coexist with the more conventional rule of *weight balancing*, in which the label with more descendants becomes the father of the label with fewer descendants. Various image scanning orders are examined and classified. It is also shown that when the algorithm is specialized to a pixel array scanned in raster order, the total processing time is linear in the number of pixels. The linear-time processing time follows from a special property of the UNION–FIND algorithm, which may be of independent interest. This property states that under certain restrictions on the input, UNION–FIND runs in time linear in the number of FIND and UNION operations. Under these restrictions, linear-time performance can be achieved without resorting to the more complicated Gabow–Tarjan algorithm for disjoint set union.

1. *Introduction*

Connected-component labeling [10] is a fundamental task common to virtually all image processing applications in two and three dimensions. For a binary image, represented as an array of d-dimensional pixels or *image elements*, connected component labeling is the process of assigning labels to the BLACK

Prior to publication of this paper, Dr. Markku Tamminen passed away. At the time this research was conducted, Dr. Tamminen was associated with the Laboratory for Information Processing Science, Helsinki University of Technology, Espoo, Finland.

Authors' addresses: M. B. Dillencourt, Department of Information and Computer Science, University of California, Irvine, CA 92717; H. Samet, Center for Automation Research, University of Maryland, College Park, MD 20742-3411.

image elements in such a way that adjacent BLACK image elements are assigned the same label [8, 10]. Here, "adjacent" may mean 4-adjacent or 8-adjacent [9]. Connected-component labeling can be characterized [7] as a transformation of a binary input image, B, into a symbolic image, S, such that

(1) All image elements that have value WHITE will remain so in S; and,
(2) Every maximal connected subset of BLACK image elements in B is labeled by a distinct positive integer in S.

This definition can be extended to other representations of images (e.g., quadtrees, octrees, and bintrees) [11, 12] in an obvious way. In these representations, the *image elements* are the portions of the image corresponding to leaf nodes. Throughout this paper we assume that in all representations considered, image elements correspond to rectangular areas of the image, and the length and width of each image element is an integral multiple of the length of a pixel.

A binary image defines a graph, in which the nodes are the BLACK image elements and the edges correspond to pairs of adjacent BLACK image elements. If the image fits in memory, and if the representation of the image does not constrain the order in which edges may be visited, then the components of the image may be efficiently labeled using a *depth-first* component-labeling strategy [5]. However, in some image representation schemes,this strategy may not be appropriate. For example, in large pixel arrays stored in raster order, or in pointerless quadtree representations [18], random access into the image can produce large numbers of page faults, so it is preferable to process the image in sequential order. In this paper, we address the problem of labeling the components of an image that is to be processed in a *predetermined* order—that is, in an order that is determined by the image representation scheme rather than by the specific characteristics of the image.

A typical implementation of predetermined-order component labeling consists of two passes. In the first pass, each pair of adjacent BLACK image elements is examined in succession, and a set of equivalence classes is maintained. Each BLACK image element is initially assigned a temporary label, and the temporary label is placed in its own equivalence class. For each pair of adjacent BLACK image elements, the equivalence classes containing the temporary labels assigned to the two image elements are merged. When the first pass is complete, the equivalence classes correspond to components (i.e., two image elements belong to the same component if and only if their temporary labels are in the same equivalence class). In the second pass, each equivalence class is assigned a unique permanent label, and each image element is assigned the label of the equivalence class to which its temporary label belongs. In both passes, the process of keeping track of the equivalence classes is facilitated by the use of a disjoint set-union algorithm. Although several such algorithms are known, the UNION–FIND algorithm [1] is the simplest and the most commonly used. This algorithm maintains each set as a tree, and uses path compression and weight balancing to yield almost linear behavior.

The contributions of this paper are fourfold. First, we present a unified single algorithm for predetermined-order connected-component labeling. The algorithm handles a wide variety of image representation schemes, including arrays, quadtrees, bintrees, and their multidimensional generalizations [13]. Second, we show how to adapt the UNION–FIND algorithm to permit the reuse of temporary labels, by introducing the notion of age-balancing, and we show

how to simultaneously implement age-balancing and weight-balancing. Third, we give several criteria for "good" scanning orders, and we introduce the notion of *admissible* and *weakly admissible* scanning orders. Finally, we show that when our algorithm is specialized to 2-dimensional pixel arrays using a raster scanning order,[1] we obtain an algorithm that runs in time linear in the image size. The linear-time bound is based on the observation that geometric constraints that apply in the case of 2-dimensional pixel arrays lead to an order of operations for which the UNION–FIND algorithm, using path compression but not necessarily using weight balancing, performs in linear time. Our linear-time bound does not require the more complicated Gabow–Tarjan algorithm [3]. The basic assumption of the Gabow–Tarjan algorithm, namely that we know the set into which an element will ultimately be merged when we create the element, does not appear to apply in the application discussed here.

Section 2 contains basic facts about the UNION–FIND algorithm. Section 3 discusses image scanning orders and the fundamental concepts underlying our algorithm. Section 4 is a general formulation of an algorithm for predetermined-order connected-component labeling that satisfies the twin goals of running efficiently and reducing storage requirements. Section 5 presents a correctness proof, and Section 6 contains an analysis of the algorithm's running time. In Section 7, we show how to adapt the general algorithm to the case of a two-dimensional array representation of an image, and we analyze the storage and execution-time requirements of the specialized algorithm. Section 8 contains some final remarks.

Connected-component labeling is a problem that has received much attention in the literature [4, 6, 7, 10, 11, 15]. Recently, Schwartz, et al. [17] have independently reported a linear-time algorithm for labeling raster-scanned 2-dimensional binary arrays. Their algorithm makes use of deeper properties of raster-scanned images, and it is not clear how to generalize their approach to other image representations. For a comparison with our approach, see [14].

2. *The* UNION–FIND *Algorithm*

The UNION–FIND algorithm is a general algorithm for keeping track of disjoint sets of elements. This algorithm makes use of a tree to represent each set. The trees are represented using only father links. Typically, the root of the tree representing a set contains a pointer to application data relevant to all elements of the set. The UNION–FIND algorithm supports three basic operations:

(1) MAKESET(A) creates a new set containing the single element A.
(2) FIND(A) finds the root of the tree that contains the element A.
(3) UNION(A, B) combines the two sets whose roots are at A and B by making one of the root elements the father of the other. The root of the combined tree is sometimes called the *survivor*.

The UNION–FIND algorithm can be made to run quite fast provided two optimizations are performed:

Path-compression. On each FIND(A) operation, all nodes encountered along the path from A to the root of the tree containing A (including A, but not

[1] Throughout this paper, *raster scanning order* means that the rows are processed from top to bottom and, within each row, pixels are processed from left to right.

including the root) have their father pointers reset to point to the root of the tree.

Weight-Balancing. When a UNION operation is performed, the smaller tree is made a subtree of the larger tree (i.e., the root with more nodes is the survivor).

With these two optimizations, any set of m FIND and UNION operations can be performed in time $O(m\alpha(m))$, where α is the inverse of Ackerman's function and grows extremely slowly. See [19] for more details on the exact formulation, and see [20] for some related results. In most practical cases $\alpha(m) \leq 5$. It is worth noting that both optimizations must be used to obtain the $O(m\alpha(m))$ time bound. If only one of the optimizations is used (i.e., either path-compression or weight-balancing but not both), then the worst-case bound is $\Omega(m \log m)$. If neither of the optimizations is used, then the worst-case bound is $\Omega(m^2)$.

In this paper, we use a modified version of the UNION–FIND algorithm. We add a fourth operation, RECYCLE(A), which removes an element A from the set to which it belongs and makes it eligible for reuse, *provided* A is not the father of another entry. It is the responsibility of the surrounding application to ensure that this constraint on A is satisfied. Clearly, the RECYCLE operation can be executed in constant time. We refer to the interval of time between the creation of an element (via MAKESET) and its return (via RECYCLE) as an *incarnation* of the element.

Our algorithm for component labeling uses a concept called *age-balancing*. In essence, age-balancing says that when we merge two components, the younger tree (the tree whose root began its current incarnation more recently) is made a subtree of the older one. We show that age-balancing permits us to use the RECYCLE operation, thereby lowering our space requirement significantly. We also show how to make age-balancing and weight-balancing coexist, so that the $O(m\alpha(m))$ bound on the UNION–FIND algorithm is maintained.

It is shown in [19] that the UNION–FIND algorithm exhibits superlinear worst-case behavior. Lemma 2.1, below, shows that under certain restrictions, the UNION–FIND algorithm is linear. Define a *chain from the root* to be a sequence of elements $A_0, A_1, \ldots, A_k$ such that A_0 is the root of the tree containing A_k and such that A_{i-1} is the father of A_i for $i = 1, \ldots, k$. We say that a sequence of UNION and FIND operations obeys the *Stable Tree Property* if, whenever $A_0, A_1, \ldots, A_k$ is a chain from the root with $k \geq 2$, then once the command FIND(A_k) is issued, A_0 remains the root of the tree until all the elements $A_1, \ldots, A_k$ have been recycled (see Figure 1). Define the *depth* of the root node in a tree to be 0, and the depth of a nonroot node to be one more than the depth of its father.

LEMMA 2.1. *Suppose that in the execution of the* UNION–FIND *algorithm, the Stable Tree Property holds. Then, the* UNION–FIND *algorithm, using path compression, runs in time linear in the number of operations, even without weight-balancing. More precisely, for any sequence of* UNION *and* FIND *operations, the total number of links traversed is at most* $2F + 3U$, *where* F *and* U *are the total number of* FIND *and* UNION *operations, respectively.*

PROOF. We first show that the Stable Tree Property implies that the following two properties hold: (1) For each incarnation of an element A, A

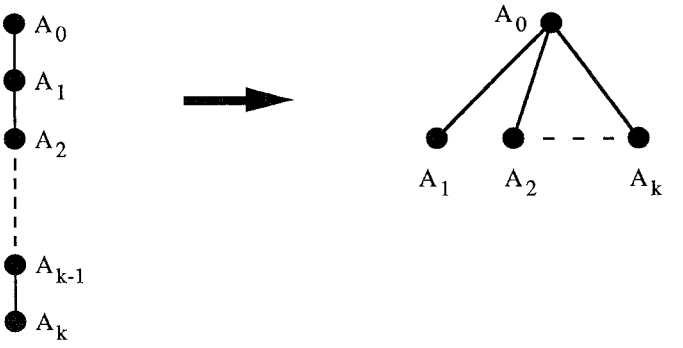

FIG. 1. The Stable Tree Property: If $A_0, A_1, \ldots, A_k$ is a chain and $k \geq 2$, then once the command FIND(A_k) is issued, A_0 remains the root of the tree until all the elements $A_1, \ldots, A_k$ have been recycled.

participates in at most one FIND operation in which A is at a depth exceeding 1. (2) In a chain from the root $A_0, A_1, \ldots, A_k$ with $k \geq 2$, for each $i > 1$ (but not necessarily for $i = 1$), there must have been a previous UNION(A_{i-1}, A_i) operation.

Let B be A's father after the first FIND operation in which A participates and in which A has a depth exceeding 1. After this operation, A's depth is 1. Because of path compression, B is the root of the tree to which A belongs. By the Stable Tree Property, B cannot acquire a father for the remainder of A's incarnation, so A's depth remains 1. This proves (1). To prove (2), observe that A_{i-1} can become the father of A_i in one of only two ways. One way is through a UNION(A_{i-1}, A_i) operation. The second possibility is through a FIND operation in which A_{i-1} is the root and the depth of A_i is at least 2. But if A_{i-1} had become the father of A_i through a FIND operation in which the depth of A_i exceeded 1, the Stable Tree Property would imply that A_{i-2} could not then have become the father of A_{i-1} before A_i was recycled. Thus, the second possibility cannot have occurred, which proves (2).

To prove the time bound asserted in the lemma, we "charge" each link traversed in each UNION and FIND operation as follows. Each UNION operation requires setting one father link, which we charge to the UNION operation. If a FIND operation is performed on an entry at depth 0 or 1, at most two links must be examined to verify that fact (in this case, no links must be modified), and we charge those 2 links to the FIND operation. If a FIND operation is performed at depth $k \geq 2$, then $k + 1$ links must be examined (to determine the root), and $k - 1$ links must be modified (to implement path compression), for a total cost of $2k$. Assume the command is FIND(A_k), and let $A_k, A_{k-1}, \ldots, A_0$ be the path from A_k to the root. Charge to links to each of the $k - 1$ commands UNION(A_i, A_{i-1}) for $i = 2, \ldots, k$. (By (2), we know each of these UNION commands occurred previously.) By (1), each element A_i participates in at most one FIND command that results in charging some UNION command. Hence, each UNION command gets charged by at most one FIND command. It follows that the total number of links traversed is at most $2F + 3U$, where F is the number of FIND commands and U is the number of UNION commands. □

It is instructive to briefly compare Lemma 2.1 with the Gabow–Tarjan linear-time algorithm for a special case of set union [3]. Essentially, the

Gabow–Tarjan algorithm requires the structure of the unions to be known in advance. Although this is a valid assumption in many applications, it is not true in the case of the component-labeling problem discussed in this paper. Moreover, the Gabow–Tarjan algorithm is more complicated to implement than UNION–FIND, although its performance is apparently roughly comparable in practice. Lemma 2.1 says that the standard UNION–FIND algorithm, using path-compression (but not necessarily using weight-balancing), runs in linear time under certain restrictions on the sequence of input commands.

3. *Active Elements, Live Labels, and Scanning Orders*

A *scanning order* defines the order in which image elements are processed, or *scanned*. Given a d-dimensional image, each image element has neighbors in at most $2 \cdot d$ *directions*. In two dimensions, this property is known as 4-*adjacency* [9].[2] An image element and each of its neighbors are adjacent to each other along a *border* of the image element. These directions are grouped into d *pairs* of opposite directions.

A preprocessing phase initializes the boundaries of the image to WHITE and hence the neighbors in these directions are considered to have been scanned initially. At any instant during the scan, the image is partitioned into three subsets:

(1) *Inactive image elements.* Scanned image elements whose $2 \cdot d$ borders are all shared with image elements that have already been scanned (or with the image boundary). These image elements do not have unscanned neighbors.
(2) *Active image elements.* Scanned image elements that have no more than $2 \cdot d - 1$ of their borders shared with image elements that have already been scanned (or with the image boundary). These image elements have at least one unscanned neighbor. Borders between scanned image elements and unscanned image elements are called *active borders*. Note that any scanned image element that is adjacent to an active border element is necessarily active.
(3) *Unscanned image elements.* Image elements that have not yet been scanned.

These three subsets are illustrated in Figure 2 for a raster scanning order.

Each image element has associate with it a *label* (this is a temporary label in the first pass and a permanent label in the second pass). Two labels are *equivalent* if two image elements associated with those labels are known to be in the same component because of adjacency information that has already been processed. We refer to labels that are associated with at least one active image element (or equivalent to such a label) as *alive* (or *live*), and we say that a label associated only with inactive image elements is *dead*. Only active image elements can cause distinct components to subsequent merge. Thus, a dead label will never be referenced again (on the current pass over the image), and storage used to represent a dead label can be recycled and subsequently reused. The algorithm that we present in the next section is based on exploiting this observation.

[2] The methods of this paper can also be adapted to 8-adjacency, in which each image element has neighbors in $3^d - 1$ directions.

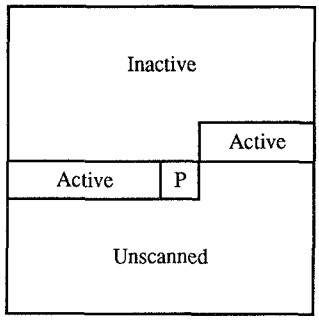

FIG. 2. Image partition after scanning pixel P in raster scanning order.

The definitions of this section, and the algorithm presented in the next section, are applicable to any scanning order. Nevertheless, as a practical matter, some scanning orders are better than others. For the purposes of the algorithm discussed in this paper, a good scanning order is one that limits the maximum number of labels that can be simultaneously alive.

A scanning order is said to be *weakly admissible* if each image element is processed once, and when processing any image element P, all of the P's neighbors in at least one direction of every direction pair either do not exist or have been scanned already. If there is a set of distinguished directions, one from each direction pair, such that whenever an image element is encountered its neighbor in each of the distinguished directions has already been processed (provided it exists), the scanning order is *admissible*. (The difference between being admissible and weakly admissible is whether the choice of directions depends on the particular image element.) Any admissible order is weakly admissible. A scanning order that is not weakly admissible is called *inadmissible*. There are many scanning orders that are admissible (e.g., left to right and top to bottom for a $2 \cdot d$ array; NW, NE, SW, SE for a quadtree [12]; left, right for a bintree [16]; etc.). Three $2 \cdot d$ examples of scanning orders that are weakly admissible but not admissible are shown in Figure 3(a)–(c). The "zigzag" order of Figure 3(a) fails to be admissible because when Pixel 1 is visited its EAST neighbor has not been scanned while when Pixel 9 is visited its WEST neighbor has not been scanned. Similar reasoning shows that the "spiral" order of Figure 3(b) and the scanning order of Figure 3(c), which grows inward from the corners of the image, are not admissible, although these three examples are all weakly admissible. An inadmissible scanning order for a quadtree is NW, NE, SE, SW. This is illustrated in Figure 3(d). In this example, when Node 3 is scanned, neither its west neighbor nor its east neighbor has been scanned. Other examples of inadmissible scanning orders are the "alternating-row" order of Figure 3(e) and the "longest-diagonal-first" order of Figure 3(f). When the alternating-row scanning order is used for an $N \times N$ image, the number of simultaneously active image elements can be as high as $N^2/4$.

Admissible scanning orders are of interest for three reasons. First, the fact that a scanning order is admissible imposes an upper bound on the maximum number of image elements that may be simultaneously active (Proposition 3.1(a), below). This limits the number of labels that may be alive simultaneously, and hence limits the storage required by the algorithm of Section 4. Second, admissibility implies that the region formed by those image elements that have been scanned is *orthogonally convex*, which means that the intersec-

1	2	3	4	5	6	7	8
16	15	14	13	12	11	10	9
17	18	19	20	21	22	23	24
32	31	30	29	28	27	26	25
33	34	35	36	37	38	39	40
48	47	46	45	44	43	42	41
49	50	51	52	53	54	55	56
64	63	62	61	60	59	58	57

(a)

1	2	3	4	5	6	7	8
28	29	30	31	32	33	34	9
27	48	49	50	51	52	35	10
26	47	60	61	62	53	36	11
25	46	59	64	63	54	37	12
24	45	58	57	56	55	38	13
23	44	43	42	41	40	39	14
22	21	20	19	18	17	16	15

(b)

1	5	17	21	42	34	10	2
9	13	25	29	46	38	14	6
33	37	49	57	54	50	26	18
41	45	53	61	62	58	30	22
24	32	60	64	63	55	47	43
20	28	52	56	59	51	39	35
8	16	40	48	31	27	15	11
4	12	36	44	23	19	7	3

(c)

1	2	5	6	17	18	21	22
4	3	8	7	20	19	24	23
13	14	9	10	29	30	25	26
16	15	12	11	32	31	28	27
49	50	53	54	33	34	37	38
52	51	56	55	36	35	40	39
61	62	57	58	45	46	41	42
64	63	60	59	48	47	44	43

(d)

1	2	3	4	5	6	7	8
33	34	35	36	37	38	39	40
9	10	11	12	13	14	15	16
41	42	43	44	45	46	47	48
17	18	19	20	21	22	23	24
49	50	51	52	53	54	55	56
25	26	27	28	29	30	31	32
57	58	59	60	61	62	63	64

(e)

1	9	23	35	45	53	59	63
16	2	10	24	36	46	54	60
29	17	3	11	25	37	47	55
40	30	18	4	12	26	38	48
49	41	31	19	5	13	27	39
56	50	42	32	20	6	14	28
61	57	51	43	33	21	7	15
64	62	58	52	44	34	22	8

(f)

FIG. 3. Examples of weak admissibility and inadmissibility. (a)–(c) are weakly admissible but not admissible, while (d)–(f) are inadmissible. (a) A pixel array, in which rows are alternately scanned left-to-right and right-to-left ("zigzag order"). (b) A pixel array scanned in "spiral order." (c) A pixel array which is scanned inward in four waves emanating from the corners of the image. (d) A quadtree scanned in NW, NE, SE, SW order. (e) A pixel array scanned in a "alternating row" order. (f) A pixel array scanned in "longest diagonal first" order.

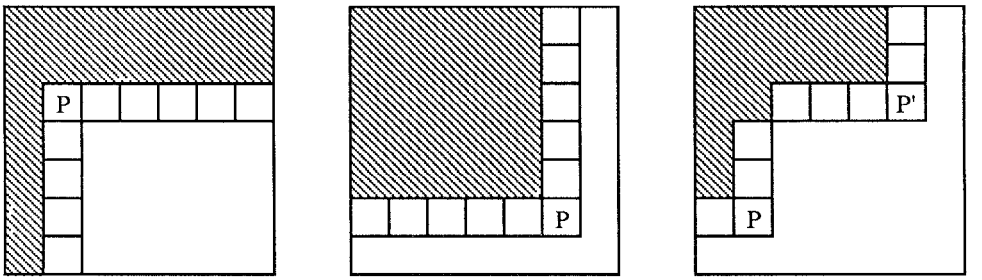

FIG. 4. Illustration of the proof of Proposition 3.1. The inactive image elements are shaded. (a) A staircase with no convex bends. (b) A staircase with one convex bend. (c) A staircase with two convex bends.

tion of any line parallel to one of the axes and the region either is empty or consists of a single interval. Orthogonal convexity in turn implies that the region formed by the scanned image elements is *simply connected*, which (in two dimensions) means that it is connected and has no holes. Although simple connectedness is not of great importance for connected-component labeling, it is crucial for other operations, such as boundary extraction (see [2]). Third, if a scanning order is known to be admissible, then it is only necessary to look at each adjacency once. For example, if a quadtree is scanned using a NW, NE, SW, SE scanning order, then a component labeling algorithm need only examine the NORTH and WEST neighbors of the image element being scanned. These properties are summarized in the following proposition:

PROPOSITION 3.1. *If a scanning order is admissible, then*

(*a*) *For an* $N \times N$ *2-d image, the number of image elements that may be active simultaneously is at most* $2N - 2$.

(*b*) *For an image of any dimension, at any time during the scan, the region formed by those image elements that have already been scanned is an orthogonally convex set. In particular, it is simply connected.*

PROOF. Assume without loss of generality, that the north and west neighbors of each image element have already been scanned when the image element is scanned. The active image elements, as well as the active borders, form a "staircase" from southwest to northeast. A convex bend is an image element that has active borders both to the south and east. If there are no convex bends, then either the active image elements form a straight horizontal or vertical run, or the staircase runs north from the south border of the image, takes one right turn, and then continues to the east border of the image as illustrated in Figure 4(a). In the former case, there are at most N active image elements. In the latter case, there are at most $2N - 2$ active image elements, with equality holding if and only if the image element labeled P is the pixel at the northwest corner of the image. Each convex bend reduces the upper bound on the number of active image elements by one. For example, if there is just one convex bend, then the maximum number of active image elements is

1	2	3		38	37	36	35
4	5	6		42	41	40	39
7	8	9		46	45	44	43
			51	50	49	48	47
22	23	24	25				
18	19	20	21		34	33	32
14	15	16	17		31	30	29
10	11	12	13		28	27	26

FIG. 5. In a weakly admissible scanning order, the number of active image elements may be as high as $4N - 7$.

$2N - 3$ as shown in Figure 4(b), while two convex bends reduce the maximum number of active image elements to $2N - 4$ as shown in Figure 4(c).

The proof of (b) is a simple inductive argument. □

Proposition 3.1(b) becomes false if "admissible" is replaced by "weakly admissible," as illustrated by Figure 3(b). In fact, Figure 3(c) shows that the scanned region need not be connected when a weakly admissible scanning order is applied. Figure 3(f) shows that the converse of Proposition 3.1(b) fails, even when "admissible" is replaced by "weakly admissible." Figure 5 shows that if "admissible" is replaced by "weakly admissible" in Proposition 3.1(a), the bound can be as high as $4N - 7$ (the case $N = 8$ is illustrated). We conjecture that this is indeed the upper bound.

4. *A General Algorithm for Connected-Component Labeling*

It should be clear from the discussion in Section 3 that by reusing labels appropriately, we can devise an algorithm for connected-component labeling that reduces the amount of storage needed for labels. The algorithm, which appears as an appendix of this paper, is executed in two passes. The first pass writes out an intermediate file consisting of image elements and temporary labels. The second pass processes this file in reverse order (the file can be conceptualized as a stack) and assigns final labels to each image element as the end result is output. Thus, the requirement that the whole image not be kept in internal memory is satisfied. The first pass processes each image element I in turn, according to some predetermined scanning order. If I is BLACK, then the temporary labels of all scanned BLACK image elements that are 4-adjacent to I are collected and an appropriate temporary label is associated with I. Regardless of the color of I, the set of active image elements is then updated to reflect the fact that some image elements may have become inactive when I was processed. This may cause temporary labels to become dead, in which case they may be eligible for reuse.

In this section, the algorithm is presented in the form of a skeleton which is applicable to images of arbitrary dimensionalities, as well as to array, run-length, quadtree, bintree, etc., representations of these images. For this reason, we leave the precise implementation details of the data structures (e.g., the set of active image elements) unspecified. The choice of data structure would depend on factors such as the image representation and the scanning order.

These factors affect the final formulation of the algorithm as some of the steps may become trivial or even unnecessary. In Section 7, we describe a very efficient implementation for a two-dimensional array representation of an image. A discussion of how the algorithm can be used for a bintree representation of an image of arbitrary dimensionality appears in [16].

Before presenting the algorithm, we first discuss how to effectively keep track of the active image elements and the equivalence relations among labels in Pass 1. An active image element ceases to be active (and is removed from the set of active image elements) when all $2 \cdot d$ of its borders are adjacent in their entirety to image elements that have already been scanned or to the image boundary. This situation is detected by keeping track of how many borders of each active image element are active. We use the term, *active border element*, to refer to elements of these borders.

A slight complication arises when using hierarchical image representations such as quadtrees, as some borders of active image elements may be only partially active. For example, consider the quadtree in Figure 6 whose BLACK boxes are labeled A–G and whose WHITE blocks are labeled 1–6. Assume that it is scanned in the order NW, NE, SW, SE. After processing the block labeled 5, blocks A, 3, B, 4, and 5 are active. Block A is active because part of its southern border is adjacent to the unscanned block C. A situation of this type is detected by use of procedure PARTIALLY—ACTIVE.

As we stated in the introduction, the basic strategy behind our algorithm is to maintain sets of equivalent temporary labels, using UNION–FIND, and to merge sets whenever an equivalence between two temporary labels is noted. In order to achieve effective reuse of space, we use the concept of *age balancing*, introduced in Section 2, which says that when two trees are merged, the oldest label becomes the root of the combined tree. This conflicts with the rule of *weight balancing* (see Section 2), which says that when two trees are merged, the root of the tree with the most descendants becomes the root of the combined tree. The conflict is resolved by representing a temporary label using two tightly coupled data structures: a *surrogate record* and a *temporary label record*. The correspondence between coupled surrogate and temporary label records is explicitly represented by pointers. Each active image element contains a pointer to the appropriate temporary label record. Each equivalence class is maintained as a tree of surrogate records. When two equivalence classes are merged, weight balancing is achieved by making the surrogate record with more descendants the root of the combined tree. Age balancing is then achieved by altering the association between the surrogate records and temporary label records, if necessary, to preserve the invariant that the temporary label record associated with the root of the combined tree is the oldest temporary label record associated with a surrogate record in the tree. This is done by exchanging the pointers to a temporary label records in the two surrogate records, and making the corresponding exchange between the pointers to surrogate records stored in the temporary label records.

We can now describe the data structures for our algorithm. Each active image element, say I, is represented as a 4-tuple consisting of the fields COLOR, DSCR, NBORDERS, and TLABEL. COLOR(I) is the color of I. DSCR(I) contains information about I—for example, the length of its side for a quadtree. NBORDERS(I) indicates how many of the borders of I are active. It is initialized when I is added to the set of active image elements. In general the initial value

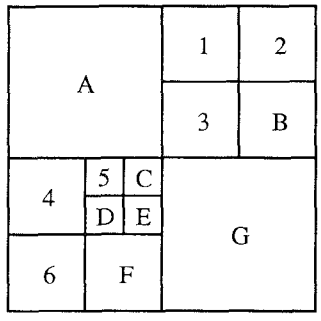

FIG. 6. Example quadtree illustrating the concept of partially active border elements. BLACK blocks are labeled A–G and WHITE blocks are labeled 1–6.

must be computed by examining I's neighbors to determine which ones have been scanned. For an admissible scanning order, the initial value is easier to compute: it is d for a d-dimensional image except when some of the borders are adjacent to the image boundary in which case it is less than d. The value is given by the function NUM_ACTIVE. TLABLE(I) points to the temporary label record of the temporary label associated with I when I became active. Once TLABEL(I) is set, it never changes until I becomes inactive.

Each surrogate record S is represented by a 3-tuple consisting of the fields TLABEL, FATHER, and COUNT. TLABEL(S) points to the corresponding temporary label record. FATHER(S) is used to implement the sets of equivalence classes: it points to another surrogate corresponding to a temporary label with which the temporary label corresponding to S has been merged. COUNT(S) contains the number of surrogate records that are descendants of S in the tree of surrogate records used to represent equivalence classes of temporary labels. COUNT(S) is used to implement weight balancing, and also to determine when it is safe to recycle S.

Each temporary label record T is represented by a 3-tuple consisting of the fields SURG, STAMP, and NACTIVE. SURG(T) points to the corresponding surrogate record. STAMP(T) is a time-stamp, used for the time-comparison between temporary label records necessary to implement age-balancing. NACTIVE(T) indicates the number of active image elements whose TLABEL field is T.

The key to effective reuse of temporary labels is the observation that if S and T are a coupled surrogate record/temporary label record pair representing a label, the temporary label may be recycled when the following two conditions are satisfied: (1) the label is no longer associated with any active image elements (NACTIVE(T) = 0), and (2) the surrogate record is not referenced by the surrogate record of a younger temporary label (COUNT(S) = 0). For example, consider the quadtree given in Figure 7. All WHITE areas are marked with numbers with the order reflecting the time at which they were visited using a NW, NE, SW, SE scanning order, which is admissible. BLACK areas are marked with letters, some of which correspond to the temporary labels that they are assigned as the first pass is executed. Four temporary labels are needed—A, B, C, and D with A the oldest and D the youngest. The cell marked with W causes B and C to be merged with B being retained. The cell marked with X causes A and B to be merged with A being retained. At the time that the cell marked with Y is processed, four temporary labels are in use. Processing Y causes path compression so that FATHER(SURG(C)) is set to point to SURG(A). However, at this time there are no active image elements whose

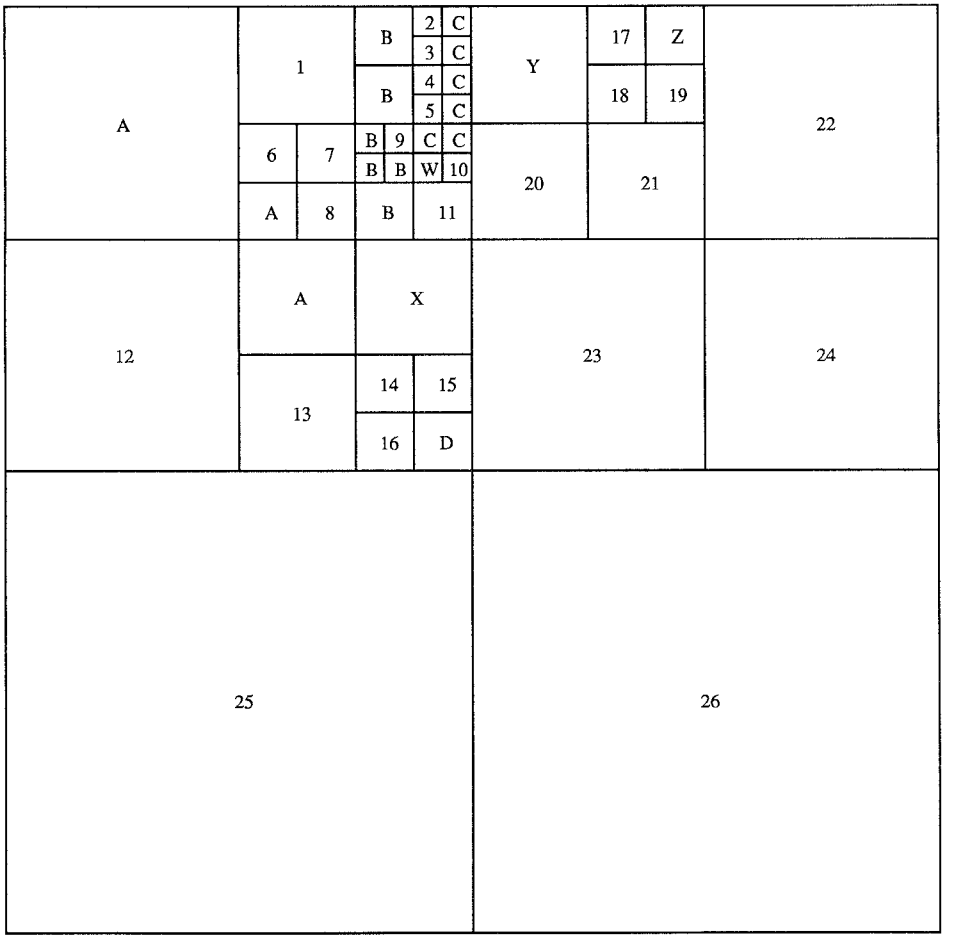

FIG. 7. Example quadtree illustrating a temporary label that becomes dead as a result of path compression.

TLABEL field is B and since FATHER(SURG(C)) no longer points at B, we find that both COUNT(SURG(B)) and NACTIVE(B) are zero. Thus, temporary label B can be recycled, and reused when block Z is subsequently processed, even though a younger temporary label, (C in this case) is still active. Notice that if we did not recycle B at this point, then we would have no easy way to detect when it could be reused again, since the link from C to B has been removed by the path compression and there are no active image elements whose TLABEL fields point to B. Notice also that it is important to perform path compression on the first FIND(C) operation when node Y is processed. Otherwise, each FIND operation performed (one for each of the four scanned neighbors) would require walking the complete path C–B–A instead of the compressed path C–A.

The first pass of the algorithm traverses the image elements and applies procedure PROCESS_ELEMENT_PASS_1 to each image element, say *I*. Initially, there are no active image elements. During this pass an intermediate output file is constructed. The output file consists of three types of records:

WHITE, BLACK, and EQUIVALENCE. Each record contains the field TYPE that indicates its type. A record of type WHITE corresponds to a WHITE image element and has no additional fields. A record of type BLACK corresponds to a BLACK image element and contains a second field, called TLABEL, which contains the temporary label associated with the image element at the time of output. The temporary label is specified as an index between 1 and the maximum number of temporary labels that have been used so far. A record of type EQUIVALENCE corresponds to a temporary label that becomes dead. A record of this type has two additional fields: the tempoary label itself, called TLABEL, and the temporary label that became its father through the surrogate structure, called FATHER, which may have a value of NULL.

Procedure PROCESS—ELEMENT—PASS—1 makes use of procedures COLLECT—ADJACENT, ASSIGN—TEMP—LABEL, REMOVE—ACTIVE—ELEMENTS, and REMOVE—ACTIVE—TEMP—LABELS. As each image element is processed, it is added to ACTIVE, the set of active image elements. If the image element is WHITE, then a record of type WHITE is output.

For each BLACK image element I, procedure COLLECT—ADJACENT performs a FIND operation, with path compression, on (the surrogate record of) the temporary label of each BLACK image element A that is 4-adjacent to I. During the path compression, temporary labels that are no longer associated with any active image elements are made available for reuse (recall the example given in Figure 7). The set of oldest representatives of equivalence classes containing temporary labels associated with elements of A is accumulated in TLABELSET.

Procedure ASSIGN—TEMP—LABEL is used to associate a temporary label with I. If TLABELSET is empty, then a new tempoary label is allocated. Otherwise, the temporary labels in TLABELSET are merged. Pointers to the surrogate record with the most descendants and the oldest temporary label record are accumulated in S—MAXCOUNT and L—MINSTAMP, respectively. The label L—MINSTAMP is retained, pointers are switched if necessary to ensure that S—MAXCOUNT = SURG(L—MINSTAMP), and weight-balancing is applied to the surrogate structure. The COUNT field of the surviving surrogate record is updated to reflect the number of temporary labels that have been merged. After the call to ASSIGN—TEMP—LABEL, PROCESS—ELEMENT—PASS—1 increments the NACTIVE field of the temporary label record corresponding to the surviving surrogate record and writes a record of type BLACK to the output file.

Procedure REMOVE—ACTIVE—ELEMENTS updates ACTIVE, the set of active image elements, by removing those image elements that have become inactive. If a removed image element is BLACK, the NACTIVE fields of the associated temporary label is decremented. If this causes the NACTIVE field of the associated temporary label to become 0, then the label is added to the set INACTIVE, which is an initially empty list of candidates for recycling. Procedure REMOVE—ACTIVE—TEMP—LABELS is called for each surrogate record corresponding to an element of the set INACTIVE. It checks whether the surrogate record can be recycled and, if so, recycles it. If the surrogate record is recycled, that may make its father eligible for recycling, and so on. It is possible that this recycling up a chain could cause a temporary label in the INACTIVE set to be recycled before its turn comes up in the loop at the end of PROCESS—ELEMENT—PASS—1. An example of this phenomenon is shown in Figure 8, where the quadtree is being processed in NW, NE, SW, SE order. When the block marked Y is processed by procedure REMOVE—ACTIVE—TEMP—LABELS, INACTIVE con-

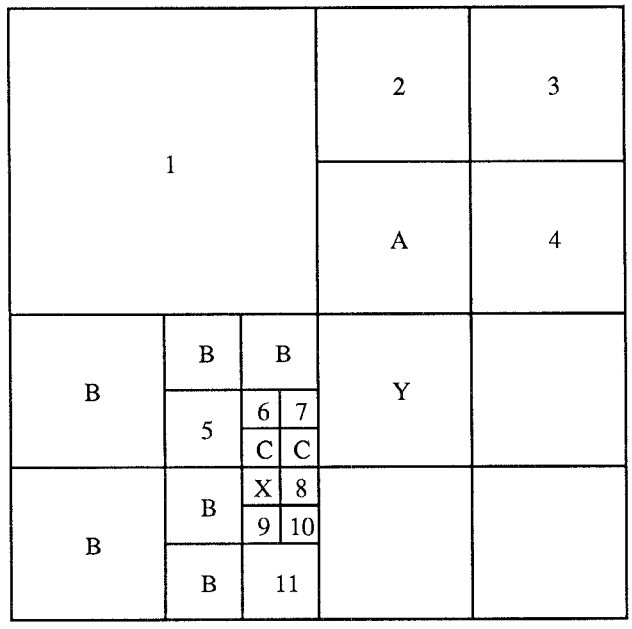

FIG. 8. An example illustrating why the function INUSE is required in the loop at the end of PROCESS_ELEMENT_PASS_1.

tains the labels A, B, and C. If C is processed before B, then B will be recycled before its turn comes up in the loop. For this reason, the primitive INUSE checks whether the label is still in use (i.e., has not been recycled). This primitive can be implemented by, for instance, having each temporary label that has been recycled store a negative value in its NACTIVE field.

The second pass processes the intermediate file of records output in the first pass in reverse order by applying procedure PROCESS_ELEMENT_PASS_2 to each record. The data structures for the second pass are much simpler than those for the first pass. There is no need to support weight-balancing in the implementation of UNION_FIND (this statement is justified in Section 6), so there is no need for surrogate records. The temporary label records require two fields: FATHER to support UNION_FIND, and LABEL to hold the permanent label. These fields may share storage with the fields in the temporary label records used in Pass 1, so no new storage is necessary for Pass 2.

Recall that there are up to three fields in each record of the intermediate file, called TYPE, TLABEL, and FATHER. Whenever a record corresponding to an equivalence relation ⟨'EQUIVALENCE',L,NULL⟩ is encountered in the intermediate file, a unique (permanent) label is generated and associated with L. If a record ⟨'EQUIVALENCE',L,F⟩ is encountered, L is linked to the temporary label F (i.e., FATHER(L) is set to F). This link is used by a FIND operation (which includes path compression) to obtain the correct label when a record corresponding to a BLACK node with temporary label L (or its equivalent sons) is subsequently encountered. WHITE nodes, (and GRAY nodes for certain hierarchical representations) do not require any special handling on the second pass, although they are written to the final output file as "place holders."

As an example, consider the 7×5 image in Figure 9(a), scanned according to a raster scanning order, where B and W correspond to BLACK and WHITE

B	B	B	B	W
W	W	W	B	W
W	B	W	B	W
W	W	W	B	W
B	B	B	B	B
W	W	W	W	W
W	B	W	W	W

(a)

(B,1)	(B,1)	(B,1)	(B,1)	(W)
(W)	(W)	(W)	(B,1)	(W)
(W)	(B,2)	(W)	(B,1)	(W)
(W)	(W) (E,2,Ω)	(W)	(B,1)	(W)
(B,2)	(B,2)	(B,2)	(B,1)	(B,1)
(W)	(W)	(W) (E,2,1)	(W)	(W) (E,1,Ω)
(W)	(B,1)	(W) (E,1,Ω)	(W)	(W)

(b)

C2	C2	C2	C2	W
W	W	W	C2	W
W	C3	W	C2	W
W	W	W	C2	W
C2	C2	C2	C2	C2
W	W	W	W	W
W	C1	W	W	W

(c)

FIG. 9. Illustration of the application of the connected-component labeling algorithm to a two-dimensional raster-scanned image representing as an array. (a) A 7×5 two-dimensional image. (b) The corresponding contents of the intermediate file. (c) The final labeled image.

pixels, respectively. The output of Pass 1 is shown in Figure 9(b), where the records of type EQUIVALENCE have been placed in the cell associated with the pixel which triggered its output. The final output is shown in Figure 9(c), where the final component labels are generated in the order C1, C2, C3 (recall that the second pass scans the intermediate file in reverse order).

5. *Proof of Correctness*

The algorithm described in Section 4 specifies a method for connected-component labeling using UNION–FIND, with reuse of temporary labels. The key

to the correctness of the algorithm is the correctness of the reuse of space in both passes. Propositions 5.2 and 5.3, below, establish that space is reused correctly. To give a precise formulation of Proposition 5.2, we need the following lemma.

LEMMA 5.1. *The mapping between temporary label records and surrogate records given by the* TLABEL *field is well-defined. More precisely, during the execution of the first pass, for each temporary label record* L, *at any given time there is a unique surrogate record* S *associated with* L. *For each such pair* L *and* S, SURG(L) = S, TLABEL(S) = L, *and* L *is in use* (*i.e., has been allocated but not recycled*) *if and only if* S *is in use.*

PROOF. When a temporary label is created (by the procedure NEW—TEMP—LABEL), a surrogate record is also created and linked to it. These links survive deallocation (recycling) of temporary labels (by RETURN—TO—AVAIL) and reallocation (by NEW—TEMP—LABEL). The links are only altered at one place in the code, namely in ASSIGN—TEMP—LABEL, and it is easily verified that if there was a well-defined association before the links are swapped then there is also a well-defined association afterwards. Since the conclusion of the lemma is vacuously true at the start of Pass 1, it follows by induction that it can never become false. □

Lemma 5.1 provides the justification for certain shorthand terminology used in the previous section and the current section. For example, if L1 and L2 are temporary labels, the statement that "L2 is the father of L1" is equivalent to the more formal statement "SURG(L2) = FATHER(SURG(L1))." Of course, it must be understood that relationships of this kind can change in two ways: When FATHER links in the forest of surrogate records are changed, and when the associa-
tion between temporary label records and surrogate records changes in ASSIGN—TEMP—LABEL. Similarly, the statement that "label 2 is recycled" is a shorthand way of saying "the temporary label record used to represent label 2 (and the associated surrogate record) are recycled."

We can now state precisely what it means for temporary labels to be reused correctly in Pass 1.

PROPOSITION 5.2. *The following properties hold throughout Pass* 1. (*a*) *No temporary label is recycled as long as it is the father of a temporary label.* (*b*) *No temporary label is recycled as long as it is associated with an active image element.*

PROOF. The key observations are that the COUNT field of a surrogate record represents its total number of descendants, and that the NACTIVE field of a temporary label record always represents the number of active image elements to which the corresponding temporary label has been assigned. Parts (a) and (b) then follow from observing that a temporary label is recycled only when it has the value 0 in its NACTIVE field and the COUNT field of its associated surrogate record also has the value 0. It remains to show that these two fields are correctly maintained.

The COUNT field of a surrogate record is initialized to zero when the associated label record is allocated in ASSIGN—TEMP—LABEL. It is altered in three places.

(1) In COLLECT—ADJACENT, when path compression occurs along a chain from the root $L_0, L_1, \ldots, L_k$ because of a FIND operation on L_k, the COUNT fields are updated according to the following two rules: (i) for $1 \leq i < k$, COUNT(L_k) is decreased by the old value of COUNT(L_{k+1}) + 1, and (ii) COUNT(L_0) is decreased by the number of labels that are recycled. These rules reflect the fact that if $1 \leq i < k$, L_{k+1} and its descendants quit being descendants of L_k irrespective of whether they are recycled, but the only labels that quit being descendants of L_0 are the labels that are recycled.
(2) In ASSIGN—TEMP—LABEL, when trees are merged, the COUNT field of the new root is increased appropriately.
(3) In REMOVE—ACTIVE—TEMP—LABELS, the number of nodes that have been deleted along the current path are accumulated in the variable DELETED—COUNT and this value is subtracted from each node.

Place (1) ensures that the COUNT field is handled correctly as path compression and node recycling occur during a FIND operation. Places (2) and (3) guarantee that the COUNT field is updated properly during UNION operations and during RECYCLE operations that stem from image elements becoming inactive, respectively. Thus, the COUNT field is correctly maintained.

The NACTIVE field is only updated in two places–in PROCESS—ELEMENT—PASS—1 when an image element is assigned a temporary label, and in REMOVE—ACTIVE—ELEMENTS when the image element becomes inactive. By inspecting the code, we see that it is maintained correctly. □

Recall that in Pass 1, when several temporary labels are merged, the oldest of the labels is kept. This strategy, which is called *age balancing*, is supported by the use of the STAMP field in the temporary label. In Pass 2, a permanent label is assigned to each BLACK image element as it is encountered. This is done by first performing a FIND on the image element's temporary label, say L_1, to find the temporary label L_2 that is the root of the tree to which L_1 belongs. The image element is then assigned the permanent label LABEL(L_2). To establish the correctness of Pass 2, we have to show that when an intermediate record corresponding to a BLACK image element with associated temporary label L_1 is encountered, L_2 has already been assigned the appropriate LABEL field and has not been subsequently reused. This follows from the following proposition.

PROPOSITION 5.3. *The following properties hold throughout Pass* 2. (*a*) *When a temporary label is encountered as the* TLABEL *field of a* BLACK *record in the* INTERMEDIATE *file then it is in a tree whose root has the correct permanent label in its* LABEL *field.* (*b*) *No temporary label is reused while it is still the father of an nonreused temporary label.*

PROOF. Let L_1 and L_2 be two temporary labels encountered during Pass 2. Suppose that L_2 is the temporary label returned by a FIND on L_1 during the processing of an image element belonging to component C. It is easy to show, by induction on the total number of UNION operations performed while processing C, that L_2 was the first (and hence, by age-balancing, the last) temporary label to be associated with C during Pass 1. This means that the last record pertaining to C that was written during Pass 1 was $\langle$'EQUIVALENCE',L_2,$\Omega\rangle$. Moreover, a record of this type is written for each component.

Hence, the first record pertaining to component C to be processed in Pass 2 was the record $\langle$'EQUIVALENCE',L_2,$\Omega\rangle$, so L_2 is the temporary label with the correct LABEL field. This proves (a).

Assertion (b) follows from the age-balancing performed during Pass 1 and the fact that the intermediate file is read in reverse order in Pass 2. In more detail, the argument is as follows. For the purposes of this proof, let image element k be the kth image element processed during Pass 1. Suppose that L_1 and L_2 are two temporary labels in Pass 2, with L_2 the father of L_1, and let C be the common component with which they are associated. This particular use of L_1 corresponds to an incarnation of the temporary label L_1 in Pass 1; suppose the incarnation began when image element k_1 was being processed. Similarly, suppose the particular use of L_2 corresponds to a (Pass 1) incarnation that began when image element K_2 was being processed. Since L_2 is the father of L_1 in Pass 2, either (1) there was a record $\langle$'EQUIVALENCE',L_1,$L_2\rangle$ written during Pass 1, or (2) L_2 is the root of the tree containing L_1. (Notice that either or both of these conditions may hold.) In either case, it follows from the age-balancing rule that $k_2 < k_1$. In Pass 2, L_2 will not be reused until after image element k_2 is processed. This occurs *after* image element k_1 is processed, at which time L_1 becomes eligible for reuse. □

We summarize the results of this section in the following theorem, which follows from Proposition 5.2, Proposition 5.3, and the remarks at the start of the section.

THEOREM 5.4. *The algorithm of Section* 4 *correctly labels the connected components of an image.*

6. *Analysis of the Algorithm*

The time-complexity of the algorithm is determined by the following quantities: the cost of processing the image elements, the cost of examining all the active neighbors of all image elements, and the cost of the UNION–FIND operations. Let I be the total number of image elements, and let E be the number of all adjacent pairs of image elements.

The main procedures in both Pass 1 and Pass 2 (PROCESS_ELEMENT_PASS_1 and PROCESS_ELEMENT_PASS_2, respectively) are each executed I times. If the scanning order is admissible, each pair of adjacent nodes is examined at most twice in Pass 1–once to find the neighbors of an image element and once to consider them for possible removal from the ACTIVE set. In the absence of admissibility, each adjacency pair may be examined four times. In Pass 1, each image element induces at most one UNION operation, and each adjacency pair induces at most one FIND operation. The total time required by all UNION and FIND operations in Pass 1 is thus $O(E\alpha(E))$, since the UNION–FIND implementation in Pass 1 combines weight-balancing and path-compression. It is then straightforward to verify that the total time required by Pass 1 is $O(E\alpha(E))$. The only possible point of difficulty is REMOVE_ACTIVE_TEMP_LABELS, and it is not hard to show that the total time required by all calls to REMOVE_ACTIVE_TEMP_LABELS is $O(E)$. This last statement follows from two easily verified facts: (1) the total number of calls to REMOVE_ACTIVE_TEMP_LABELS in all of Pass 1 is at most $2E$ (at most E if the scanning order is admissible),

and (2) each temporary label in INACTIVE is at depth at most 2, so each call to REMOVE—ACTIVE—TEMP—LABELS requires at most three iterations of the **while** loop.

In Pass 2, each image element induces at most one UNION (corresponding to an EQUIVALENCE record) and at most one FIND operation. It follows from Proposition 5.3(a) that the Stable Tree Property holds in Pass 2, so Lemma 2.1 can be applied, and we get a bound of $O(I)$ for the total cost of the UNION–FIND operations in Pass 2. This last observation justifies the statement, made in Section 4, that weight-balancing is not necessary in Pass 2.

It follows from the preceding paragraphs that the worst-case time-complexity is $O(I + E\alpha(E))$. For almost any representation of an image, and certainly for all the ones considered here (bintrees, quadtrees, arrays, etc.), $E = O(I)$, so the worst-case time-complexity reduces to $O(I\alpha(I))$ in these important cases. This statement remains true for d-dimensional extensions of these representations if d is treated as a constant. The complexity is actually $O(d \cdot I\alpha(I))$ for 4-adjacent labeling and $O(3^d \cdot I\alpha(I))$ for 8-adjacent labeling.

7. *Component Labeling in a Pixel Array Using Raster-Scanning Order*

The algorithm described in Section 4 is formulated in a general manner that is independent of the representation of the image and of the scanning order. In this section, we show how to adapt it for the array representation of a two-dimensional image, processed in raster-scanning order. The resulting algorithm runs in time linear in the number of pixels, which is optimal. Rather than give a detailed description of the raster-scan algorithm, we show how the general algorithm can be simplified to obtain it. A detailed description of the algorithm, and some empirical results, can be found in [15].

Assume an $N \times N$ image such that the origin is at the upper left corner of the image so that rows and columns are numbered from 1 to N. A pixel at position (i, j) is in row i and column j. When we process a pixel at position (i, j), the set of active image elements consists of the pixels at positions (i, p) such that $0 < p < j$ and $(i - 1, p)$ such that $j \leq p \leq N$. The active border elements are the southern sides of the active image elements and the eastern side of the active image element at position $(i, j - 1)$. For simplicity we assume that all pixels of (the fictitious) row 0 and column 0 are WHITE.

It is easy to see that the set of active elements can be represented as a one-dimensional array. The array representation, coupled with the raster scanning order, is useful for two important reasons. First, when collecting the temporary labels of the 4-adjacent neighbors of an active image element in PROCESS—ELEMENT—PASS—1 and COLLECT—ADJACENT we need not perform a search–a single array access suffices. For a pixel at position (i, j), the 4-adjacent active image elements are found at positions $(i, j - 1)$ and $(i - 1, j)$, which correspond to positions $j - 1$ and j in the array. Determining the temporary labels associated with these pixels only requires one FIND operation for the temporary label associated with of $(i - 1, j)$. The temporary label associated with $(i, j - 1)$ is current since this was the most recent pixel processed. Second, there is no need to have an NBORDERS counter associated with an active pixel. This is because, once an image element's eastern neighbor has been processed, only one active border element remains (the southern side), and when this border element ceases to be active, so does the pixel.

The raster-scanning algorithm and the representation of the active elements can be further simplified by making use of the observation that active image elements become inactive in the same relative order that they became active. This observation facilitates keeping track of temporary labels that have been merged. It also simplifies the process of removing elements that are no longer active from the set of active elements. In particular, there is no need for the loop in REMOVE_ACTIVE_TEMP_LABELS (see [15]).

We now show that the specialized algorithm runs in linear time. More precisely, we show that the total number of links that must be traversed is bounded by $9B$, where B is the number of black pixels. Our proof is based on showing that Lemma 2.1 applies. This means that weight balancing is not needed, so the surrogate and temporary label records described in Section 4 may be combined into a single record structure and the COUNT field is unnecessary. These observations simplify considerably the data structures required. The following lemma captures the intuitive notion that an older component cannot be "surrounded" by a younger component.

LEMMA 7.1. *Let L and M be two active temporary labels that are the roots of* UNION-FIND *trees, with M younger than L. Let P be an active pixel that is labeled with a label L' that is a descendant of L. Then all active pixels that are labeled with M or its descendants are on the same side (left/right) of P.*

PROOF. Let Q be the oldest pixel labeled with L (i.e., the first pixel to receive this label in the current incarnation of the label L). Suppose that X and Y are two active pixels labeled with descendants of M, one on either side of P, as illustrated in Figure 10. There are disjoint 4-connected paths Π_{PQ} and Π_{XY}, each consisting of black pixels that have already been processed, connecting X to Y and P to Q, respectively. It follows that Π_{XY} must contain some pixel above Q. But then the temporary label M is older than L, which is a contradiction. □

PROPOSITION 7.2. *Pass* 1 *requires at most $5B$ links to be accessed, where B is the number of black pixels processed.*

PROOF. Each BLACK pixel induces one call to UNION and one call to FIND. The key to the proof is establishing that Lemma 2.1 applies. We first establish the following claim.

Claim 1. Suppose that when a pixel P is processed, with pixel V immediately above P and TLABEL(V) = A_k, the call FIND(TLABEL(V)) causes the chain $A_k, \ldots, A_0$ to be collapsed due to path compression. Then, all active pixels labeled with A_j, $1 \le j \le k$, are to the right of V.

Suppose the claim is false, and let m be the smallest value of j for which it is false. Then there is some pixel Q to the left of P such that Q is active and labeled with A_m when P is processed. Let R be the pixel that caused the temporary label A_m to be merged with A_{m-1}. R must be to the right of Q (since otherwise Q would not be labeled with A_m) and to the left of P (since by the time P is encountered, A_{m-1} is a descendant of A_0). Let L and T be, respectively, the pixels to the left of and above R. Then R, L, and T must be BLACK, and R must be labeled with A_{m-1} (see Figure 11). If $m > 1$, this contradicts the definition of m, so we may assume that $m = 1$.

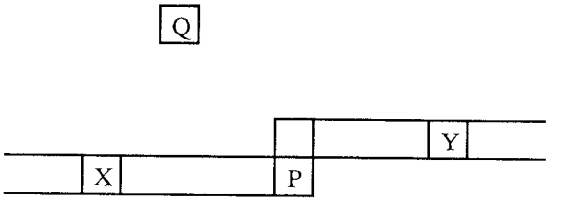

FIG. 10. Illustration of the proof of Lemma 7.1.

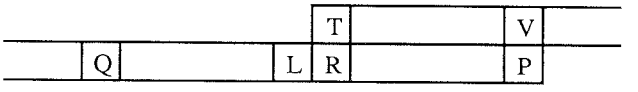

FIG. 11. Illustration of the proof of Claim 1 of Proposition 7.2.

Now consider the situation immediately before R is processed. V is labeled with A_k, Q is labeled with A_1, and either L or T is labeled with A_0 (if $L = Q$, T is labeled with A_0). A_k is in the tree rooted at A_1, which is disjoint from (and younger than) the tree rooted at A_0. Lemma 7.1 then implies that V and Q must be on the same side of R, which is a contradiction of our initial assumption. This contradiction establishes Claim 1.

Claim 2. Suppose that when a pixel P is processed, with V the pixel immediately above P and TLABEL(V) = A_k, the call FIND(TLABEL(V)) causes the chain $A_k, \ldots, A_0$ to be collapsed. Then, the labels $A_k, \ldots, A_1$ will all become dead before a pixel labeled with a temporary label whose root is older than A_0 is encountered.

Suppose Claim 2 is false. Then some pixel Q, encountered after pixel P, is associated with a temporary label (say A^*) that is older than A_0, and when Q is encountered, some pixel R labeled with A_j is still active. By Claim 1, R is to the right of V. Since R is still active when Q is encountered, Q is to the right of P, and R must be to the right of the pixel above Q (see Figure 12). Since A_0 is younger than A^*, and pixels P and R are on opposite sides of Q, this is a violation of Lemma 7.1. Claim 2 follows by contradiction.

Claim 2 implies that the Stable Tree Property holds, so we can apply Lemma 2.1. Hence, the total number of links traversed in Pass 1 is no more than $2F + 3U = 2B + 3B = 5B$. □

Pass 2 requires at most $4 \cdot B$ links to be accessed. This is because we do a FIND on a new temporary label immediately after encountering it, so age-balancing implies that no temporary label is ever at a depth greater than 2 in the UNION–FIND structure. Thus no FIND operation requires accessing more than three links or resetting more than one, and there is at most one FIND per black pixel.

The storage requirement is the storage for N the active image elements plus the storage for the live temporary labels. At any one time, there can be no

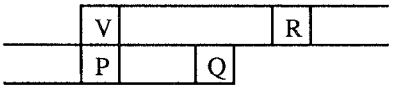

FIG. 12. Illustration of the proof of Claim 2 of Proposition 7.2.

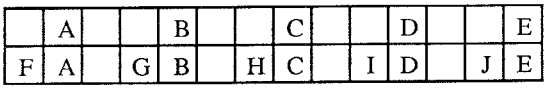

FIG. 13. The number of simultaneously live labels in a raster-scanned $N \times N$ pixel array can be as high as $\lfloor 2N/3 \rfloor$.

more than $\lceil 2N/3 \rceil$ live nodes, where $\lceil \cdot \rceil$ denotes the ceiling function. This follows from the (easily verified) fact that at any moment during Pass 1, of any three consecutive active pixels, either one is white or at least two have the same temporary label. Figure 13 shows that the bound of $\lceil 2N/3 \rceil$ is tight. In the figure, $N = 14$, blank pixels are white, and the letters indicate labels.

The results of this section are summarized in the following theorem.

THEOREM 7.3. *The total number of links processed in connected-component labeling of an $N \times N$ image represented as an array and processed in raster-scan order, using the methods of this section, is bounded by $9B$, where B is the number of black pixels. Hence, the total time requirement is $O(N)$. The storage requirement is also $O(N)$.*

8. *Concluding Remarks*

We have presented an efficient algorithm for connected-component labeling of images for arbitrarily specified scanning orders. We have proposed a criterion for good scanning orders, namely admissibility. We have shown that our algorithm, when specialized to a 2-dimensional pixel array processed in raster-scan order, runs in time linear in the number of pixels. This analysis is based on the observation that the UNION–FIND algorithm runs in linear time provided the sequence of input operations satisfies the Stable Tree Property, a fact that may be of interest in other situations as well.

We leave as an open problem a detailed worst-case analysis of the storage and time requirement of the approach of Section 4 for other representations such as quadtrees and 3D-pixel arrays. We conclude with an example showing that the Stable Tree Property does not necessarily hold for the UNION–FIND structures that arise in arbitrary admissible scanning orders. The quadtree of Figure 14 is scanned in NW, NE, SW, SE order. Cells marked with letters are BLACK, and cells marked with numbers are WHITE. The letters in some of the cells are the temporary labels associated with them. The nodes marked X, Y, and Z respectively cause D to become the father of E, C the father of D, and B the father of C. The node marked U causes path compression to occur along the path E–D–C–B. When node V is subsequently encountered, it causes label A to become the father of label B, even though labels C, D, and E are still alive. This shows that the Stable Tree Property does not necessarily hold for quadtrees.

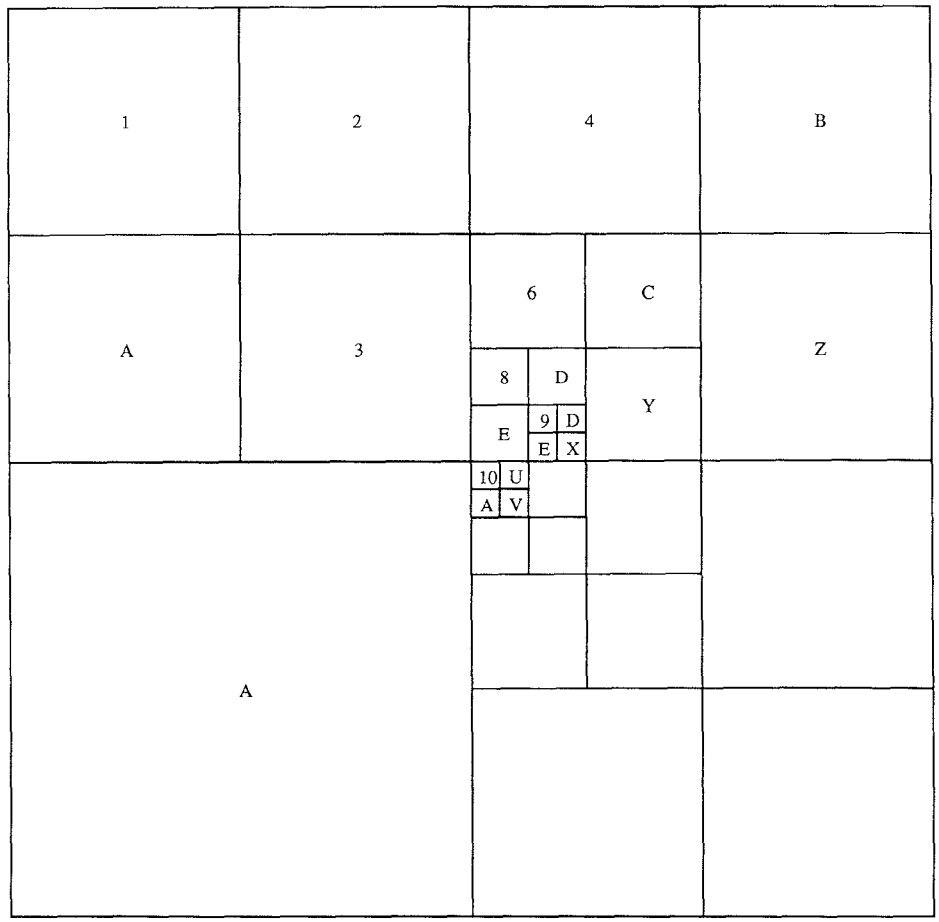

FIG. 14. The Stable Tree Property does not necessarily hold for quadtrees scanned in an admissible order.

Appendix. A General Algorithm for Connected-Component Labeling

```
procedure PROCESS_ELEMENT_PASS1(I, ACTIVE);
   /* Add image element I to the set of active image elements during the first pass of
      connected-component labeling of an image and output appropriate records to the
      intermediate file pointed at by INTERMEDIATE for the second pass. The contents of
      these records are specified within angle brackets */
begin
  value pointer image_ELEMENT I;
  reference pointer image_ELEMENT SET ACTIVE;
  global integer MAXSTAMP, MAXLABEL; /* initially 0 at start of PASS1 */
  global pointer file INTERMEDIATE;
  pointer temp_LABEL SET TLABELSET,INACTIVE;
  pointer image_ELEMENT A;
  if COLOR(I) = GRAY then
    begin /* this case handles quadtrees, octrees, bintrees, etc. */
      output(INTERMEDIATE,⟨'GRAY'⟩);
      DECOMPOSE_AND_RECUR(I, PROCESS_ELEMENT_PASS1);
    end
  else
    begin
```

```
        addtoset(I,ACTIVE);
        NBORDERS(I) ← NUM__ACTIVE(I);
        /* NUM__ACTIVE indicates how many of I's borders are ACTIVE */
        if COLOR(I) = WHITE then output(INTERMEDIATE,⟨'WHITE'⟩)
        else /* I is BLACK */
          begin
            TLABELSET ← empty;
            foreach A in ACTIVE suchthat FOUR__ADJACENT(A,I) do
              COLLECT__ADJACENT(A,TLABELSET);
          ASSIGN__TEMP__LABEL(I,TLABELSET);
          NACTIVE(TLABEL(I)) ← NACTIVE(TLABEL(I)) + 1;
          output(INTERMEDIATE,⟨'BLACK', TLABEL(I)⟩);
        end;
      INACTIVE ← empty;
      foreach A in ACTIVE suchthat
        FOUR__ADJACENT(A,I) and not PARTIALLY__ACTIVE(DSCR(A), DSCR(I)) do
          REMOVE__ACTIVE__ELEMENTS(A,ACTIVE,INACTIVE);
      foreach L in INACTIVE do
        if INUSE(L) then REMOVE__ACTIVE__TEMP__LABELS(SURG(L));
    end;
  end;
  procedure COLLECT__ADJACENT(A,TLABELSET);
    /* Collect the temporary labels of BLACK active image elements that are 4-adjacent to I.
       */
  begin
    value pointer image__ELEMENT A;
    reference pointer temp__LABEL SET TLABELSET;
    pointer surrogate S, S1, S2;
    integer PATHCOUNT ← 0;
    if COLOR(A) = BLACK then
      begin
        S1 ← S ← SURG(TLABEL((A));
        while not null(FATHER(S)) do S ← FATHER(S); /* FIND */
        while S1 ≠ S do /* path compression */
          begin
            S2 ← FATHER(S1);
            /* PATHCOUNT contains value of COUNT(S1) from before start of path
               compression */
            if S2 ≠ S then
              begin
                COUNT(S2) ← COUNT(S2) − PATHCOUNT − 1;
                PATHCOUNT ← PATHCOUNT + COUNT(S2) + 1; /* old COUNT(S2) */
              end;
            if COUNT(S1) = 0 and NACTIVE(TLABEL(S1)) = 0 then
              begin
                RETURN__TO__AVAIL(TLABEL(S1));
                COUNT(S) ← COUNT(S) − 1;
              end
            else
              FATHER(S1) ← S;
            S1 ← S2;
          end;
        addtoset(S,TLABELSET);
      end;
  end;
  procedure ASSIGN__TEMP__LABEL(I, TLABELSET);
    /*Assign a temporary label to image element I. TLABELSET contains the temporary labels
      of all BLACK image elements that are 4-adjacent to I. If TLABELSET is empty, then
      allocate a temporary label and assign it to I. Otherwise, determine L__MINSTAMP, the
      oldest temporary label, and S__MAXCOUNT, the surrogate with the most descendants.
      In this case, first achieve age-balancing and weight-balancing by ensuring that
```

```
      S__MAXCOUNT is the surrogate for L__MINSTAMP. Next merge the labels in TLABELSET
      */
begin
  value pointer image__ELEMENT I;
  reference pointer temp__LABEL SET TLABELSET;
  pointer temp__LABEL L__MINSTAMP,L;
  pointer surrogate S__MAXCOUNT;
  pointer global integer MAXLABEL, MAXSTAMP;
  if empty(TLABELSET) then
    begin /* no BLACK active image elements are 4-adjacent to I. */
      L__MINSTAMP ← NEW__TEMP__LABEL( );
        /* returns pointer to temp__LABEL record properly coupled with a
          surrogate record */
      NACTIVE(L__MINSTAMP) ← COUNT(SURG(L__MINSTAMP)) ← 0;
      FATHER(SURG(L__MINSTAMP)) ← NIL;
      STAMP(L__MINSTAMP) ← MAXSTAMP ← MAXSTAMP + 1;
    end
  else
    begin
      L__MINSTAMP ← ARBITRARY(TLABELSET); /* pick some arbitrary element of
        TLABELSET */
      S__MAXCOUNT ← SURG(L__MINSTAMP);
      foreach L in TLABELSET do
        begin
          if STAMP(L) < STAMP(L__MINSTAMP) then
            L__MINSTAMP ← L; /* determine oldest temporary label */
          if COUNT(SURG(L)) > COUNT(S__MAXCOUNT) then
            S__MAXCOUNT ← SURG(L); /* determine surrogate with largest
              subtree */
        end;
      /* ensure S__MAXCOUNT is surrogate for L__MINSTAMP */
      if S__MAXCOUNT ≠ SURG(L__MINSTAMP) then
        begin
          L ← TLABEL(S__MAXCOUNT);
          TLABEL(S__MAXCOUNT) ↔ TLABEL(SURG(L__MINSTAMP));
          SURG(L) ↔ SURG(L__MINSTAMP);
        end;
      foreach L in TLABELSET do
        begin /* UNION */
          S ← SURG(L);
          if S ≠ S__MAXCOUNT then
            begin
              FATHER(S) ← S__MAXCOUNT;
              COUNT((S__MAXCOUNT) ← COUNT(S__MAXCOUNT) + COUNT(S) + 1;
            end;
        end;
    end;
  TLABEL(I) ← L__MINSTAMP;
end;
procedure REMOVE__ACTIVE__ELEMENTS(A,ACTIVE,INACTIVE);
  /* Remove image element A from the set ACTIVE if it is no longer active. If the removed
      image element is BLACK, decrement the NACTIVE field of the associate temporary
      label. If this field becomes 0, add the temporary label to INACTIVE */
begin
  value pointer image__ELEMENT A;
  reference pointer image__ELEMENT SET ACTIVE;
  reference pointer temp__LABEL SET INACTIVE;
  NBORDERS(A) ← NBORDERS(A) − 1; /* I can be adjacent to A along only one border */
  if NBORDERS(A) = 0 then
    begin /* image element A is no longer ACTIVE */
      REMOVE__FROM__SET(A,ACTIVE);
```

```
      if COLOR(A) = BLACK then
        begin
          NACTIVE(TLABEL(A)) ← NACTIVE(TLABEL(A)) − 1;
          if NACTIVE(TLABEL(A)) = 0 then addtoset(TLABEL(A),INACTIVE);
        end;
    end;
end;
procedure REMOVE_ACTIVE_TEMP_LABELS(S);
  /* Recycle the temporary label associated with surrogate record S if it is legal to do so. If
     S can be reused, then check if its father can also be reused, and so on. Update COUNT
     fields all the way up to the root */
begin
  value pointer surrogate S;
  pointer surrogate S1;
  integer DELETED_COUNT ← 0;
  while not null(S) do
    begin
      COUNT(S) ← COUNT(S) − DELETED_COUNT;
      S1 ← S;
      S ← FATHER(S);
      if NACTIVE(TLABEL(S1)) = 0 and COUNT(S1) = 0 then
        begin /* temporary label with surrogate S can be reused */
          DELETED_COUNT ← DELETED_COUNT + 1;
          output(INTERMEDIATE,⟨'EQUIVALENCE',TLABEL(S1),TLABEL(S)⟩); /*
            TLABEL(NIL) = NIL! */
          RETURN_TO_AVAIL(TLABEL(S1));
        end;
    end;
end;
procedure PROCESS_ELEMENT_PASS2(R);
  /* Assign the final component label to the objectcorresponding to R during the second
     pass of connected = component labeling of an image. */
begin
  value INTERMEDIATE_RECORD R;
  global integer MAXLABEL; /* initially 0 at start of pass2 */
  if TYPE(R) = 'BLACK' then /* format is ⟨'BLACK',TLABEL⟩ */
    output(LABEL(FIND(TLABEL(R))))
  else if TYPE(R) = 'EQUIVALENCE' then /* format is ⟨'EQUIVALENCE',
    TLABEL,FATHER⟩ */
    begin
      FATHER(TLABEL(R)) ← FATHER(R); /* UNION */
      if null(FATHER(R)) then LABEL(TLABEL(R)) ← MAXLABEL ← MAXLABEL + 1;
    end
  else output(TYPE(R)); /* WHITE or GRAY node */
end;
pointer temp_LABEL PROCEDURE FIND(L);
  /* Find the root of the tree to which L belongs, using path compression */
begin
  value pointer temp_LABEL L;
  pointer temp_LABEL L1, L2;
  if null(FATHER(L)) then return(L);
  L1 ← L;
  while not null(FATHER(L)) do L ← FATHER(L); /* find root */
  while FATHER(L1) ≠ L do /* path compression */
    begin
      L2 ← FATHER(L1);
      FATHER(L1) ← L;
      L1 ← L2;
    end;
  return(L);
end;
```

ACKNOWLEDGMENTS. We thank John Canning and Azriel Rosenfeld for helpful discussions and comments.

REFERENCES

1. AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.
2. DILLENCOURT, M. B., AND SAMET, H. Extracting region boundaries from maps stored as linear quadtrees. In *Proceedings of the 3rd International Conference on Spatial Data Handling* (Sydney, Australia, Aug.) 1988, pages 65–77.
3. GABOW, H. N. AND TARJAN, R. E. A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci. 30*, 2 (Apr. 1985), 209–221.
4. HARALICK, R. M. Some neighborhood operations. In M. Onoe, K. Preston, and A. Rosenfeld, eds. *Real Time/Parallel Computing Image Analysis*. Plenum Press, New York, 1981, pp. 11–35.
5. HOPCROFT, J., AND TARJAN, R. Efficient algorithms for graph manipulation. *Commun. ACM 16*, 6 (June 1973), 372–378.
6. LUMIA, R. A new three-dimensional connected components algorithm. *Comput. Graph., Vision, Image Proc. 23*, 2 (Aug. 1983), 207–217.
7. LUMIA, R., SHAPIRO, L., AND ZUNIGA, O. A new connected components algorithm for virtual memory computers. *Comput. Graph., Vision, and Image Proc. 22*, 2 (May 1983), 287–300.
8. PARK, C. M., AND ROSENFELD, A. Connectivity and genus in three dimensions. Computer Science Technical Report TR-156. Univ. Maryland, College Park, Md., May 1971.
9. ROSENFELD, A., AND KAK, A. C. *Digital Picture Processing*. Academic Press, Orlando, Fla., second edition, 1982.
10. ROSENFELD, A., AND PFALTZ, J. L. Sequential operations in digital picture processing. *J. ACM 13*, 4 (Oct. 1966), 471–494.
11. SAMET, H. Connected component labeling using quadtrees. *J. ACM 28*, 3 (July 1981) 487–501.
12. SAMET, H. The quadtree and related hierarchical data structures. *ACM Comput. Surv. 16*, 2 (June 1984), 187–260.
13. SAMET, H. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, Mass., 1990.
14. SAMET, H., AND TAMMINEN, M. A general approach to connected component labeling of images. Computer Science Technical Report TR-1649. Univ. Maryland, College Park, Md., Aug. 1986.
15. SAMET, H., AND TAMMINEN, M. An improved approach to connected component labeling of images. In *Proceedings of Computer Vision and Pattern Recognition 86* (Miami Beach, Fla., June) 1986, pp. 312–318.
16. SAMET, H., AND TAMMINEN, M. Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *IEEE Trans. Pattern Analysis and Machine Int. 10*, 4 (July 1988), 579–586.
17. SCHWARTZ, J. T., SHARIR, M., AND SIEGEL, A. An efficient algorithm for finding connected components in a binary image. Robotics Research Technical Report 38. New York Univ. New York, Feb. 1985 (Revised July, 1985).
18. SHAFFER, C. A., SAMET, H., AND NELSON, R. C. QUILT: A geographical information system based on quadtrees. *Int. J. Geograph. Inf. Syst. 4*, 2 (April–June 1990), 103–131.
19. TARJAN, R. E. Efficiency of a good but not linear set union algorithm. *J. ACM 22*, 2 (Apr. 1975), 215–225.
20. TARJAN, R. E., AND VAN LEEUWEN, J. Worst-case analysis of set union algorithms. *J. ACM, 31*, 2 (Apr. 1984), 245–281.

RECEIVED SEPTEMBER 1986; REVISED AUGUST 1989 AND SEPTEMBER 1990, ACCEPTED FEBRUARY 1990