Connected component labeling for arbitrary binary image representations

Michael B. Dillencourt
Department of Information and Computer Science
University of California
Irvine, CA 92717

Hanan Samet
Computer Science Department
University of Maryland
College Park, MD 20742

Markku Tamminen
Laboratory for Information Processing Science
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.). We show how to adapt the standard UNION-FIND algorithm to permit reuse of temporary labels. 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.

1 Introduction

Connected component labeling [7] 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 the same label to all adjacent BLACK image elements [5, 7]. The elements may be 4-adjacent or 8-adjacent [6]. Connected component labeling can be characterized [4] as a transformation of a binary input image, B, into a symbolic image, S, such that (1) all image elements which have value WHITE will remain so in S; and, (2) every maximally 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) [8, 9] in an obvious way. In these representations, the *image elements* are the portions of the image corresponding to leaf nodes. Throughout this paper we will 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 may be thought of as 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 [3]. 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 [14], 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. While 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.

In this paper, 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 [10]. 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.

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 approach to predetermined-order connected component labeling that satisfies the twin goals of running efficiently and reducing storage requirements.

Section 5 addresses the correctness of the algorithm and an analysis of its running time. Section 6 contains some final remarks.

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 one of the root elements a 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 the following 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 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 [15] for more details on the exact formulation. 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 will 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*. 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 will show that age-balancing permits us to use the RECYCLE operation, thereby lowering our space requirement significantly. We will 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.

3 Active Elements and Live Labels

A *scanning order* defines the order in which image elements are processed, or *scanned.* The algorithm of this paper is formulated for *4-adjacency* [6], in which each image element of a d-dimensional image has neighbors in $2 \cdot d$ *directions*, although the methods of this paper can also be adapted to 8-adjacency, in which each image element has neighbors in 3^d directions. An image element and 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. *Inactive image elements* are scanned image elements whose $2 \cdot d$ borders are adjacent in their entirety to image elements that have already been scanned (or to the image boundary). These image elements do not have unscanned neighbors. *Active image elements* are scanned image elements that have no more than $2 \cdot d - 1$ of their borders adjacent in their entirety to image elements that have already been scanned (or to the image boundary). These image elements have at least one unscanned neighbor. Borders between scanned and unscanned image elements are called *active borders.* Finally, *unscanned image elements* are image elements that have not yet been scanned. These three subsets are illustrated in Figure 1 for a raster scanning order.

Each image element has associated 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 subsequently 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 presented below is based on exploiting this observation.

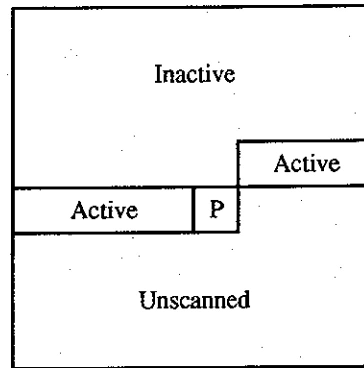

Figure 1: Image partition after scanning pixel P in raster scanning order.

4 A General Algorithm for Connected-Component Labeling

The general 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.

The algorithm is presented in the form of a skeleton which is applicable to images of arbitrary dimensionalities, as well as to array, runlength, quadtree, bintree, etc., representations of these images. We leave the precise implementation details of the data structures (e.g., the set of active image elements) unspecified. The choice of data structures 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. A very efficient implementation for a two-dimensional array representation of an image is described in [11]. A discussion of how the algorithm can be used for a bintree representation of an image of arbitrary dimensionality appears in [12].

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. 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 age balancing. This conflicts with weight balancing, which is necessary to obtain $O(m\alpha(m))$ performance. 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 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 - e.g., 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 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. TLABEL(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) $= 0$).

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 temporary 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. 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 temporary 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. 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 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 2(a), scanned according to a raster scanning order, where B and W correspond to BLACK and WHITE pixels respectively. The output of Pass 1 is shown in Figure 2(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 2(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 Correctness and Analysis

In this section we briefly sketch an argument for the correctness of the algorithm described in Section 4 and provide an upper bound on irs running time. More detail can be found in [2].

In the first pass, no temporary label is recycled as long as it is the father of a temporary label, and no temporary label is recycled as long as it is associated with an active image element. These two observations imply that no temporary label is recycled prematurely, so space is reused correctly in the first pass.. In the second pass, no temporary label is reused while it is still the father of a non-reused temporary label. Moreover, 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. It follows that space is reused correctly in the second pass, and each image element is assigned the correct permanent label.

The time-complexity of the algorithm is determined by the following quantities: the

B	B	B	B	W
W	W	W	B	W
W	W	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)

Figure 2: Illustration of the connected-component labeling algorithm on a 2-dimensional raster-scanned image. (a) A 7×5 two-dimensional image. (b) The output of the first pass. (c) The final output.

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. Each adjacency pair is examined at most 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. Hence the total time required by Pass 1 is $O(E\alpha(E))$. In the full paper, it is shown that Pass 2 runs in $O(I)$ time. Thus the worst-case time-complexity of the general algorithm 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 ($O(d \cdot I\alpha(I))$ in d dimensions).

6 Concluding Remarks

We have presented an efficient algorithm for connected-component labeling of images for arbitrarily specified scanning orders. For raster-scanned arrays, the algorithm runs in time linear in the number of pixels [11]. Thus in this case, our general algorithm problem is as fast as the algorithm of [13], which is specifically formulated for raster-scanned arrays and does not appear to be easily generalized to other image representations. We leave as an open problem a detailed worst-case analysis of the storage and time requirement of the general algorithm for other representations such as quadtrees and 3D-pixel arrays.

Acknowledgments

We thank John Canning and Azriel Rosenfeld for helpful discussions and comments. This research was supported in part by the National Science Foundation under grant IRI-8802457.

References

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

[2] M. B. Dillencourt, H. Samet, and M. Tamminen. Connected-component labeling of binary images. Computer Science CS-TR-2303, University of Maryland, College Park, MD, August 1989.

[3] J. Hopcroft and R. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, June 1973.

[4] R. Lumia, L. Shapiro, and O. Zuniga. A new connected components algorithm for virtual memory computers. *Computer Vision, Graphics, and Image Processing*, 22(2):287–300, May 1983.

[5] C.M. Park and A. Rosenfeld. Connectivity and genus in three dimensions. Computer Science Technical Report TR-156, University of Maryland, College Park, MD, May 1971.

[6] A. Rosenfeld and A.C. Kak. *Digital Picture Processing.* Academic Press, New York, NY, second edition, 1982.

[7] A. Rosenfeld and J.L. Pfaltz. Sequential operations in digital image processing. *Journal of the ACM*, 13(4):471–494, October 1966.

[8] H. Samet. Connected component labeling using quadtrees. *Journal of the ACM*, 28(3):487–501, July 1981.

[9] H. Samet. The quadtree and related hierarchical structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

[10] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1989.

[11] H. Samet and M. Tamminen. A general approach to connected component labeling of images. In *Proceedings of Computer Vision and Pattern Recognition 86*, pages 312–318, Miami Beach, June 1986.

[12] H. Samet and M. Tamminen. Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(4):579–586, July 1988.

[13] J. T. Schwartz, M. Sharir, and A. Siegel. An efficient algorithm for finding connected components in a binary image. Robotics Research 38, New York University, New York, NY, February 1985. Revised July, 1985.

[14] C. A. Shaffer, H. Samet, and R. C. Nelson. QUILT: a geographic information system based on quadtrees. Computer Science Technical Report TR-1885.1, University of Maryland, College Park, MD, July 1987.

[15] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.

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 object corresponding 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;
```