AN IMPROVED APPROACH TO CONNECTED COMPONENT LABELING OF IMAGES*

Hanan Samet

Computer Science Department
University of Maryland
College Park, Maryland 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 that is particularly useful for processing large images which do not fit in core. Its virtue is a very small equivalence table. As the image is processed, equivalence classes are reused when they cease to be associated with any "active" component. This technique is equally applicable to images represented by hierarchical data structures such as quadtrees, octrees, and the general d-dimensional representation known as the bintree. Equivalences are processed by using UNION-FIND. An implementation of the algorithm is given for a two-dimensional array. For an $N \times N$ array, the number of equivalence classes is bounded by $N/2$. The amount of internal storage is $O(N)$. Its total processing cost is linear in the number of pixels.

1. INTRODUCTION

Connected component labeling [3] is a fundamental task common to virtually all image processing applications in two as well as three dimensions. For a binary image, represented as an array of d-dimensional pixels termed *image elements*, it is the process of assigning the same label to all adjacent BLACK image elements [3]. The elements may be 4-adjacent or 8-adjacent [2].

Connected component labeling can be performed by depth-first and breadth-first techniques. They differ in the time at which equivalences between labels of adjacent BLACK image elements are propagated. The depth-first approach labels each component in its entirety one-by-one. It requires the whole image to be readily accessible. If the cost of determining if two BLACK image elements are adjacent is constant, then an algorithm employing this approach can be devised that runs in time proportional to the product of the dimensionality of the image and the number of BLACK image elements.

In most applications we must process images which are much bigger than the capacity of internal memory. Thus the depth-first approach is inappropriate and we focus on the breadth-first approach. This approach examines each pair of adjacent BLACK image elements in succession and constructs an equivalence table where initially each BLACK image element is in a separate equivalence class. For each such pair, a two stage process (also known as *UNION-FIND* [8]) is applied. It makes use of a tree to represent each equivalence class. First, determine the equivalence classes associated with both BLACK image elements that comprise the pair by using FIND. FIND traverses father links in the tree to locate the root. If the classes differ, then they are combined using UNION. UNION merges two trees by making the father of the root of one tree point at the root of the other tree. Path compression [7] is applied as part of FIND to make sure that after UNION no equivalence class on the FIND path is more than one link away from the root of its tree. Stage two assigns a final label to each BLACK image element (corresponding to the component of which it is a member). When path compression is used, the results of Tarjan and van Leeuwen [9] let us deduce that the worst-case execution time of the total (i.e., both stages) task is almost linear.

*The support of the National Science Foundation under Grant DCR–83–02118 is gratefully acknowledged.

Several algorithms for two-dimensional arrays based on the breadth-first approach have been reported [1,3]. They require two passes if the images contain any adjacent BLACK image elements. In this paper we present an improved breadth-first approach based on reducing the number of equivalence classes that can be active at any instant. We assume 4-adjacency although our methods could also be adapted to 8-adjacency. The technique is general in the sense and is equally applicable to images represented by hierarchical data structures such as quadtrees, octrees, and the general d-dimensional representation known as the bintree [4]. We conclude with an adaptation of the algorithm to a two-dimensional array representation of an image whose total cost is linear in the number of pixels.

2. NEW ALGORITHM

A *scanning order* defines the order in which image elements are processed. Given a d-dimensional image, each image element has neighbors in $2 \cdot d$ *directions*. An image element and its neighbors are adjacent in a given direction along a *border* of the image element. These directions are grouped into d pairs each element of which is opposite to the other. A scanning order is said to be *admissible* if, when processing any image element P, all of P's neighbors in at least one direction of every direction pair have already been processed (i.e., scanned). A preprocessing phase initializes the boundaries of the image to WHITE and hence the neighbors in their direction are said to have been processed (i.e., scanned).

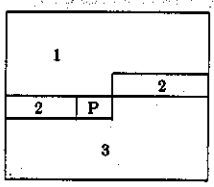
There are many scanning orders that are admissible (e.g., left to right and top to bottom for a 2-d array; NW, NE, SW, SE for a quadtree; left, right for a bintree; etc.). Admissible scanning orders are of interest because it is easy to see that a connected component labeling algorithm that visits each image element using such a scanning order insures that a transition was made through all adjacencies between image elements. Of equal importance, each image element will only be visited once by this process. Of course, the second stage of the breadth-first approach requires that the BLACK image elements be visited one more time so that their final label can be assigned.

At any instant of time, say t, after processing image element P, the scan partitions the image into three subsets:

(1) 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.
(2) Scanned image elements having at least d but no more than $2 \cdot d - 1$ borders adjacent in their entirety to image elements that have already been scanned or to the image boundary.
(3) Unscanned image elements.

These three subsets are illustrated in the figure below for a raster scanning order (i.e., process each row from left to right starting at the top row) and labeled appropriately. At time t, only image elements that are members of the second subset can cause distinct components consisting, in part, of members of the first subset to merge. We use the term *active* to describe the set of image elements comprising the second subset, their unscanned borders, and the equivalence classes in which these image elements are members. At this time, if two distinct

components having image elements that are members of the first subset do not contain any image elements that are active, then they can never become equivalent because all chains of adjacencies are propagated via the active borders. Alternatively, this property means that an equivalence class without at least one active member will never be referenced again. Thus the storage used to represent it can be used to represent another component. To see that this is true, suppose that after processing image element P in the figure below, there are no more active image elements that participate in a given equivalence class, say E associated with component C. Therefore, no subsequent image elements can belong to component C and thus we can reuse equivalence class E to keep track of another component. The components are assigned unique labels by pass two of the algorithm. Thus the number of different equivalence classes (and hence the size of the equivalence table) is bounded by the number of classes that can be active simultaneously. This number is, in turn, bounded by the number of image elements that can be active simultaneously. The algorithm presented below exploits these properties.



For example, for an $N \times N$ binary array that is raster-scanned (see the above image), the active elements consist of one element with two active borders (i.e., P) and $N-1$ elements that have just one active border. Moreover, since all of the equivalence classes must be distinct, in the case of 4-adjacency, we find that only $N/2$ equivalence classes can be active simultaneously and hence the size of the equivalence table is bounded by $N/2$. Generalizing these properties to a d-dimensional binary array, with each side having width N, we find that the maximum number of active elements is N^{d-1} while the upper bound on the number of distinct equivalence classes, and consequently on the equivalence table, is $N^{d-1}/2$.

Our algorithm is executed in two passes. The first pass creates an intermediate file consisting of image elements and equivalence classes while 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 a final image file is output. Thus the requirement that the whole image not be kept in internal memory is satisfied. The algorithm is applicable to both arrays and hierarchical representations such as quadtrees and bintrees, as well as to images of arbitrary dimension. Here we leave the issue of the exact representation (i.e., data structure) of the set of active image elements unspecified. The choice would depend on factors such as the image representation (e.g., array, quadtree, etc.) and on the scanning order. These factors have important ramifications on the final formulation of the algorithm as some of the steps become trivial or even unnecessary.

Before presenting the algorithm, we comment on how to effectively keep track of the active image elements, and the number of active image elements that are members of each equivalence class as well as what other information must be recorded for each equivalence class. 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 these borders. In addition, for hierarchical image representations (e.g., quadtrees) some borders of active image elements are only partially active. For example, when an active image element corresponding to a quadtree block, say I, is 4-adjacent to another block, say A, such that A is larger. In such a case A would only cease to be active if none of its sides is partially adjacent to an unscanned element. This situation is detected by use of procedure PARTIALLY_ACTIVE.

Each equivalence class, say E, is represented by a 3-tuple consisting of the fields NACTIVE, FATHER, and LABEL. NACTIVE(E) indicates the number of active image elements in class E, plus how many other equivalence classes have been merged into E. FATHER(E) points to an equivalence class with which E has been merged. LABEL(E) serves two purposes. On the first pass it is used as a time stamp to indicate when class E was first assigned to an image element in the component that it is currently representing. On the second pass it is used to store the label that has been assigned to the component that is currently associated with class E.

Each active image element, say I, is represented as a 4-tuple consisting of the fields COL, DSCR, NBORDERS, and EQC. COL(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. The initial value is usually d for a d-dimensional image except when some of the borders are adjacent to boundary elements in which case it is less than d. EQC(I) points to the equivalence class that was initially associated with I. Equivalence classes are only accessed via the active image elements. Thus there is no explicit data structure corresponding to the set of equivalence classes - they are just records. Once an equivalence class, say E, is initially associated with active image element I by being assigned to EQC(I), EQC(I) is not updated. Therefore we need not know how to locate the elements that should be updated when two or more (e.g., when $d \geq 2$) equivalence classes are merged. Instead, in order to access the most current value of the equivalence class containing I, FIND is applied to EQC(I) which means that the chain starting at FATHER(EQC(I)) is traversed until encountering an equivalence class F such that FATHER(F) is NULL.

If image element I causes several different equivalence classes to be merged, say O and Y_i ($1 \leq i \leq m$) such that O was assigned to an image element at an earlier time than any of Y_i, then the older of the classes (i.e., O) is the one that is retained. By *retained* we mean that O is the class that is now associated with I and that all subsequently scanned image elements that are members of the merged classes will be referenced by O. Such a merge is called *age balancing*. The EQC field of active image elements whose value was Y_i will retain this value after the merge. The merge is recorded by setting the FATHER field of the younger classes to the older class (i.e., FATHER(Y_i) is set to point to O for each of the Y_i). The age comparison between the classes is determined by the values of their LABEL fields. In addition, the number of classes that have been merged into O is incremented by m (i.e., NACTIVE(O)←NACTIVE(O)+m). Note that our concept of "merging" differs from the traditional one used with UNION-FIND algorithms as we may merge more than two classes (e.g., when $d \geq 2$ and also when more than one active image element is 4-adjacent to I). Once a chain of classes has been traversed by a FIND and after a UNION, path compression is applied to update all FATHER pointers on the path to point to the most current class (i.e., F).

The first pass traverses the image elements in an admissible scanning order applying PROCESS_ELEMENT_PASS1, given below, to each image element, say I. Initially, there are no active image elements. During this process an intermediate output file is constructed. It contains all of the image elements as well as pointers to (indices of) equivalence classes. This file consists of three types of records. Type WHITE corresponds to a WHITE image element. Type BLACK corresponds to a BLACK image element in which case it also includes the class in which the image element is contained at the time of output. The class can be specified as an index between 1 and the maximum number of classes that have been used so far. Type EQCLASS corresponds to a class which is no longer active in which case it also includes the contents of its FATHER field (which may be NULL).

```
procedure PROCESS_ELEMENT_PASS1(I,ACTIVE);
/* Add image element I to the ACTIVE set of image elements during
   the first pass and output appropriate records to the intermediate
   file pointed at by INTERMEDIATE for the second pass. The con-
   tents of these records are specified within < >. */
begin
 value image_element I;
```

```
reference image_element set ACTIVE;
global integer MAXLABEL;
global pointer stack INTERMEDIATE;
eq_class set ESET;
eq_class E,J,Q,T;
image_element A;
integer MINLABEL;
if GRAY(COL(I)) then
 begin  /* This case handles quadtrees, octrees, bintrees, etc. */
  push(INTERMEDIATE,<'GRAY'>);
  DECOMPOSE_AND_RECUR(I,PROCESS_ELEMENT_PASS1);
 end
else
 begin
  addtoset(I,ACTIVE);
  NBORDERS(I)←NBORDERS(I)+NUM_ACTIVE(I);
  if WHITE(COL(I)) then push(INTERMEDIATE,<'WHITE'>)
  else /* I is BLACK */
   begin
    ESET←empty;
    MINLABEL←MAXLABEL+1;
    foreach A in ACTIVE such that FOUR_ADJACENT(A,I) do
     /*  Collect the equivalence classes of BLACK active  */
     begin  /* image elements that are 4-adjacent to I. */
      if BLACK(COL(A)) then
       begin
        addtoset(EQC(A),ESET);
        T←EQC(A);
        while not null(FATHER(T)) do
         T←FATHER(T); /* FIND */
        if LABEL(T)<MINLABEL then /* Determine oldest class */
         begin /* Use LABEL field */
          MINLABEL←LABEL(T);
          E←T;
         end;
       end;
     end;
    if empty(ESET) then  /* No BLACK active image elements */
     begin  /* are 4-adjacent to I. */
      E←create(eq_class);
      EQC(I)←E;
      FATHER(E)←NIL;
      MAXLABEL←MAXLABEL+1;
      LABEL(E)←MAXLABEL;
     end
    else
     begin
      EQC(I)←E; /* E is the oldest class */
      foreach J in ESET do
       begin /* UNION operation with path compression */
        while not null(J) and J neq E and FATHER(J) neq E do
         begin
          T←FATHER(J);
          FATHER(J)←E;
          J←T;
          NACTIVE(E)←NACTIVE(E)+1;
          if not null(J) then NACTIVE(J)←NACTIVE(J)-1;
          while not null(J) and NACTIVE(J)=0 do
           begin
            T←FATHER(J);
            returntoavail(J);
            J←T;
            if not null(J) then NACTIVE(J)←NACTIVE(J)-1;
           end;
         end;
       end;
      NACTIVE(E)←NACTIVE(E)+1;
      push(INTERMEDIATE,<'BLACK',EQC(I)>);
     end;
    foreach A in ACTIVE such that FOUR_ADJACENT(A,I) do
     /*  Remove image elements that are no longer active  */
     begin  /* by virtue of their 4-adjacency with I. */
      if not PARTIALLY_ACTIVE(DSCR(A),DSCR(I)) then
       begin
        NBORDERS(A)←NBORDERS(A)-1;
        if NBORDERS(A)=0 then
         begin
          removefromset(A,ACTIVE);
          if BLACK(COL(A)) then
           begin
            Q←EQC(A);
            NACTIVE(Q)←NACTIVE(Q)-1;
            while NACTIVE(Q)=0 do
             begin /* Equivalence class Q can be reused */
              push(INTERMEDIATE,<'EQCLASS',Q,FATHER(Q)>);
              T←Q;
              Q←FATHER(Q);
              returntoavail(T);
              if null(Q) then return
              else NACTIVE(Q)←NACTIVE(Q)-1;
             end;
           end;
         end;
       end;
     end;
   end;
end;
```

The second pass processes the file of records output in the first pass in reverse order applying PROCESS_ELEMENT_PASS2 to each record. Whenever a record, say R, corresponding to an equivalence class (i.e., of type EQCLASS), say E, is encountered, a unique label is generated and associated with E if the father field specified in the record (i.e., FATHER(R)) is NULL. Otherwise, E is linked to the class specified in the father field, say F (i.e., FATHER(E) is set to F). This link is used by the FIND operation to obtain the correct label when a record corresponding to a BLACK node in class E (or its equivalent sons) is subsequently encountered. It can be shown (see Theorem 1) that when the link from E to F is made, E is guaranteed not to be associated with any other component that is still active (in the backward pass meaning of "active"). WHITE (and GRAY for certain hierarchical representations) nodes do not require any special handling on the second pass and are output into the final result as "place holders".

```
procedure PROCESS_ELEMENT_PASS2(R);
/*  Assign the final component label to the image element correspond-
    ing to R during the second pass.  */
begin
 value stack_entry R;
 global integer MAXLABEL;  /* Initially 0 */
 if TYPE(R)='BLACK' then output(LABEL(FIND(EQC(R))))
 else if TYPE(R)='EQCLASS' then
  begin
   FATHER(EQC(R))←FATHER(R);
   if null(FATHER(R)) then
    LABEL(EQC(R))←MAXLABEL←MAXLABEL+1;
  end
 else output(TYPE(R)); /* WHITE or GRAY node */
end;
```

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

As an example, consider the 7×5 image given above where B and W correspond to BLACK and WHITE pixels respectively. Assume that the image has been raster-scanned. Let the origin be at the upper left corner of the image with rows and columns numbered from 1 to N. A pixel at (i,j) is in row i and column j. The image is redrawn

below where each cell contains a 1-tuple, 2-tuple, or 3-tuple consisting of the information output for it on the first pass. The first entry in the tuple is the type (W for WHITE, B for BLACK, and E for EQCLASS). The second entry indicates the equivalence class associated with BLACK or EQCLASS tuples. The third entry is the father of the class for an EQCLASS tuple. A NULL pointer is specified by Ω. Note that records of type EQCLASS have been placed in the cell associated with the pixel which triggered its output.

(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)

Once class 1 is assigned to pixel (1,1) this class is propagated to the neighbors of this pixel as the image is scanned. When pixel (3,2) is encountered, we assign class 2 to it. After processing pixel (4,2) class 2 is no longer active and is reused upon encountering pixel (5,1). When processing pixel (5,4) we merge classes 1 and 2 and retain class 1 as it is older than 2. Pixel (5,5) is assigned to class 1. In addition, the FATHER field of class 2 is set to point to 1. Once pixel (6,3) has been processed, class 2 is no longer active and an EQCLASS record is output containing 2 and a pointer to 1 (its father). Once pixel (6,5) has been processed, class 1 is no longer active, and again an EQCLASS record is output containing 1 and a pointer to NULL as 1 has no father. Class 1 is reused upon encountering pixel (7,2) but only briefly as it ceases to be active after processing pixel (7,3).

The second pass processes the output of the first pass in reverse order. The first two items are (W). Upon encountering (E,1,Ω) a label is generated for the first component, say C_1. The next occurrence of (E,1,Ω) causes a label to be generated for the second component, C_2. (E,2,1) causes the FATHER field of class 2 to be set to point at class 1. Thereafter, all occurrences of class 2 will be associated with C_2. (E,2,Ω) at (4,2) results in reusing class 2 and thus generates a label for the third component, say C_3. From now on, all occurrences of class 2 will be associated with C_3.

The key to the correctness of our algorithm is being able to prove that it does not reuse equivalence classes while they still contain active image elements. Such a proof must demonstrate that both the forward and backward (i.e., the first and second) passes operate in a manner much akin to a postorder tree traversal. In other words, we must show that the following property holds:

Tree Property: When equivalence class O is encountered as the first representative of component C_1, O will not be reused to represent another component C_2 until each equivalence class Y that is subsequently encountered, which is merged with O so that O is retained, no longer is used to represent elements of C_1. Moreover, an equivalence class, say E, that represents component C_1, is not reused to represent another component C_2 as long as there exists an equivalence class whose FATHER field is E.

Theorem 1: The tree property is satisfied by both the forward and backward passes of the algorithm .
Proof: See [6]. •

3. LABELING TWO-DIMENSIONAL ARRAYS

Adapting the algorithm of Section 2 to the array representation of a binary image is quite straightforward. We use a raster scanning order which is clearly admissible. Assume an $N \times N$ image with the origin at the upper left corner of the image. Rows and columns are numbered from 1 to N. A pixel at position (i,j) is in row i and column j and when it is processed, the set of active image elements consists of 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 $(i,j-1)$. For simplicity we assume that each pixel of the leftmost column (i.e., $(p,0)$ such that $0<p \leq N$) is WHITE as are all pixels in row 0 (i.e., $(0,p)$ such that $0<p \leq N$).

From this discussion it is easy to see that the set of active elements can be represented as an array. The array representation, coupled with the raster scanning order, facilitates many operations. First, when collecting the equivalence classes of the 4-adjacent neighbors of an active image element in PROCESS_ELEMENT_PASS1 we need not do a search - just do an array access. Using a raster scanning order, for a pixel at (i,j), the 4-adjacent active image elements are found at $(i,j-1)$ and $(i-1,j)$. Determining the equivalence classes associated with these pixels only requires one FIND operation for the class of $(i-1,j)$. The class of $(i,j-1)$ is current since this was the most recent pixel processed. Second, there is no need to have an NBORDERS counter with each active image element, say I, since there is only one active border element (i.e., the southern side) and thus when this border ceases to be active, then so does I.

Our algorithm and the representation of the active elements is simplified considerably by observing that active image elements become inactive in the same relative order that they became active. To see this, note that the set of active elements is just the last N image elements that were scanned. In other words, when a pixel, say P, becomes active we know that the equivalence class, say E, in which P is contained at the time it became active cannot outlive (i.e., its NACTIVE field will have a nonzero value) any other equivalence class to which E subsequently becomes equivalent. This is useful for several reasons. First, it facilitates keeping track of equivalence classes that have been merged. It enables us to avoid having to increment the NACTIVE field of the surviving equivalence class whenever a UNION operation or path compression occurs. 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 at the end of PROCESS_ELEMENT_PASS1.

Second, it means that the array for the active elements can be implemented as an image buffer whose size is the width of a row plus one. Third, the "push" operations in PROCESS_ELEMENT_PASS1 can be modified to output the pixel at $(i-1,j)$ after processing the pixel at (i,j). Moreover, when outputting a pixel at $(i-1,j)$ we also check if it is the last active element of its equivalence class, and if yes, then output the class of its father. Thus there is no need to output a special record of type EQCLASS for the equivalence pair. The result is that all of the "push" operations have been combined and occur once the pixel is removed from the set of active elements.

These modifications mean that the output of the first pass can simply be a list of pointers to equivalence classes where we allocate two classes, pointed at by BLACK_PIXEL and WHITE_PIXEL, to correspond to flags to distinguish between BLACK and WHITE intermediate output entries. In an actual implementation, the pointers are replaced by indices. Below we give the result of applying the first pass to the 7×5 image discussed earlier. Each cell contains a 1-tuple, 2-tuple, or 3-tuple defined as follows. For a WHITE pixel, the output is a 1-tuple containing a pointer to WHITE_PIXEL (e.g., W). For a BLACK pixel, the output is a 2-tuple containing a pointer to its equivalence class and a pointer to BLACK_PIXEL. If the BLACK pixel was one that caused its equivalence class, say E, to cease being active, then the output is a 3-tuple with the pointer to FATHER(E) as the third entry. Note that the order of the entries in the 2-tuple and the 3-tuple is slightly different from that used in Section 2.

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

The second pass of the algorithm needs the following minor modification to cope with the above implementation. As the pointers to the equivalence classes are removed from the intermediate file,

there are three possibilities. If the pointer is equal to WHITE_PIXEL, then nothing is done. If the pointer is equal to BLACK_PIXEL, then the next item is a pointer to its class and the appropriate label is obtained by FIND. If the pointer is neither BLACK_PIXEL or WHITE_PIXEL, then it is a pointer to a father, say F. The next two items are BLACK_PIXEL and a pointer to the class associated with the pixel, say E. This results in the appropriate pointer being set (i.e., FATHER(E) is set to F). If the father was NULL, then a new label is generated (e.g., entry (1,B,Ω) for pixel (7,2)). Otherwise, the label is obtained by determining the equivalence class presently associated with the pixel by use of FIND starting with F and FATHER(E) is reset if necessary (e.g., entry (2,B,1) for pixel (5,3)).

The code for the algorithm is given in the Appendix. The entire process is controlled by procedure LABEL_ARRAY. It scans the image pixel by pixel and row by row. The first pass is implemented by procedures PROCESS_PIXEL and PUT_PIXEL. The second pass is implemented by procedure PHASEII.

Analyzing the storage requirements of the above algorithm is not difficult. Assume an $N \times N$ image. In order to place our bounds in a proper perspective, we compare them to the algorithm of Lumia *et al.* [1]. We qualify references to their algorithm by *old* and to our algorithm by *new*. The only internal storage that is necessary to run the new algorithm is that required for the active elements and the equivalence table. The buffer for the active elements requires $N+1$ entries, each represented by a record of two fields. The equivalence table requires at most $N/2$ entries, each represented by a record of three fields. The old algorithm also makes two passes over the data. It has similar internal storage requirements in the sense that it can be implemented using a buffer of the size of two rows and an equivalence table whose width is at most N. Each time a row has been processed, the equivalences are determined, and each element in the row is updated to contain its most current equivalence class value.

The external storage requirements of the two algorithms depend on the size of the intermediate file. The new algorithm outputs type information about each pixel (we ignore the end of file marker here) as well as an equivalence class pointer for each BLACK pixel. Moreover, an additional pointer is output for some BLACK pixels which signals that a class can be reused. This situation arises whenever two classes have been merged (i.e., a UNION), or the last pixel of the component has been output. Since the number of classes is bounded by $N/2$, only $\log_2(N/2)$ bits are required to specify each class. Assuming that the image contains B BLACK pixels, C components, that U UNION operations are performed (recall that a UNION is only done for vertically adjacent pixels), one end of file marker, and that each field is encoded by $\log_2(N/2)$ bits, then the intermediate file requires $(N^2+B+C+U)\cdot\log_2(N/2)$ bits. However, $U+C \leq B$ and thus $(N^2+2\cdot B)\cdot\log_2(N/2)$ is an upper bound on this quantity. Similarly, the old algorithm must also use an intermediate file to store the labels assigned to the various pixels. In this case, since the maximum number of classes is $N^2/2$, each class can be encoded using $2\cdot\log_2(N/2)$ bits. Thus the total external storage requirements are $2\cdot N^2\cdot\log_2(N/2)$ bits. If the probability of a pixel being BLACK is greater than 0.5, this is lower than the new algorithm's bound. However, unlike the new algorithm's bound, this is not an upper bound, but instead is the actual external storage required.

Although the specification of the old algorithm does not indicate how equivalences are handled, for the sake of comparison of execution times assume that UNION-FIND is used as it is the optimal method for such tasks. For each pixel the new algorithm does a maximum of one UNION and one FIND on the first pass and one FIND on the second pass for a total of one UNION and two FINDs. In contrast, the old algorithm requires one UNION and two FINDs for each pixel on each of its two passes for a total of two UNIONs four FINDs.

We can tighten our analysis of the new algorithm even further by making use of the following results about the cost of FIND and UNION. Assume path compression on both passes. From the following theorem we have that the sum of the links traversed by these operations on the two passes is not greater than eight times the number of pixels in the image (five for the first pass and three for the second pass). The proof assumes that on the first pass, a FIND and a UNION (actually UNION_FIND) are done for every pixel in the image although they are only done once for each pair of vertically adjacent BLACK pixels. No UNION is done on the second pass.

Theorem 2: The average time necessary to process equivalences for each pixel in the new algorithm is bounded by a constant and thus the new algorithm is linear in the number of pixels in the image.
Proof: See [6].

In order to verify the practical efficiency of the algorithm we ran a number of experiments with both an optimized and unoptimized version of our program using randomly generated images varying the probabilities of a pixel being BLACK. Optimization consisted mainly of replacing functions by macros. The experiments were performed on a VAX 11/750 running UNIX BSD 4.2. The times are "user" CPU-times as reported by UNIX taken as averages of three independent runs. The programs were written in C. The results are tabulated in Tables 1 and 2 for a number of different probabilities and row sizes. It is clear that for a given probability of a pixel being BLACK, the time per pixel is constant as indicated by Theorem 2. Note that since the algorithm is sequential, and makes use of little internal memory, the execution times per pixel are independent of the image size and thus can be extrapolated even for larger images.

Table 1. Execution times (secs.) for the unoptimized program.

Row Size	Probability of BLACK Pixel				
	0.1	0.25	0.5	0.75	0.9
128	4.92	5.64	10.4	7.52	8.0
256	19.7	22.7	42.3	30.3	32.8

Table 2. Execution times (secs.) for the optimized program.

Row Size	Probability of BLACK Pixel	
	0.1	0.5
128	3.37	4.33
256	13.4	16.9
512	54.3	68.8

It would be interesting to compare our results with those published for the old algorithm (see [1]). Unfortunately, this is not easy. First, we do not know what types of images they used. Second, it is difficult to factor out programming skill. Third, different computers and operating systems can also bias the comparison. Nevertheless, using a metric of amount of time per pixel we find that for the images that they reported, their execution times ranged between 681 to 1462 microseconds per pixel on a VAX11/780 for images ranging in size between 6K and 10.24M pixels. In contrast, our results depended on the probability of the pixel being BLACK but were between 204 (300) and 264 (645) microseconds per pixel for the optimized (unoptimized) program on a VAX11/750 for images ranging in size between 16K and 262K. The speed of the VAX 11/750 is about 0.6 of the VAX11/780 and thus for a more realistic comparison our execution times ranged between 122 (180) and 158 (387) microseconds of VAX 11/780 CPU time per pixel for the optimized (unoptimized) program.

4. CONCLUDING REMARKS

An improved approach to breadth-first connected component labeling has been presented that is appropriate for images that must be kept in external storage and when the size of the equivalence table must be kept to a minimum. It is general in the sense that it is applicable to array representations as well as hierarchical data structures such as quadtrees and bintrees and in fact has been used for the bintree [5]. Applying this approach to an array representation of a two-dimensional yields an algorithm that has linear worst-case time complexity. For data of higher dimensions, the analysis is complicated by the ability of the chains of 4-adjacent elements that form a connected component to "wiggle" their way around each other. For hierarchical representations such as the quadtree and bintree, the analysis is complicated by the fact that image elements do not become inactive in the same relative order that they became active. This means that the

cost of path compression will not necessarily be recovered by subsequent FIND operations. Although our technique involved use of the UNION-FIND algorithm, its complexity was reduced due to the use of age balancing and the limitation on the number of different equivalence classes that can be active at any instant of time. In practice, the number of links that are manipulated by the combination of UNION and FIND operations is small and empirical tests show that our algorithm compares favorably with existing methods.

ACKNOWLEDGMENTS

We have benefitted greatly from discussions with John Canning and Mike Dillencourt.

REFERENCES

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

2. A. Rosenfeld and A.C. Kak, *Digital Picture Processing*, Second Edition, Academic Press, New York, 1982.

3. A. Rosenfeld and J.L. Pfaltz, Sequential operations in digital image processing, *Journal of the ACM 13*, 4(October 1966), 471-494.

4. H. Samet, The quadtree and related hierarchical data structures, *ACM Computing Surveys 16*, 2(June 1984), 187-260.

5. H. Samet and M. Tamminen, Efficient component labeling of images of arbitrary dimension, Computer Science TR-1480, University of Maryland, College Park, MD, February 1985.

6. H. Samet and M. Tamminen, A general approach to connected component labeling of images, Computer Science TR-1649 University of Maryland, College Park, MD, June 1986.

7. R. Sedgewick, *Algorithms*, Addison-Wesley, Reading, MA, 1983.

8. R.E. Tarjan, Efficiency of a good but not linear set union algorithm, *Journal of the ACM 22*, 2(April 1975), 215-225.

9. R.E. Tarjan and J. van Leeuwen, Worst-case analysis of set union algorithms, *Journal of the ACM 31*, 2(April 1984), 245-281.

APPENDIX - TWO-DIMENSIONAL ARRAY ALGORITHM

```
procedure LABEL_ARRAY(N);
/* Perform connected component labeling for an N×N array. The
   active elements are stored in an array of records of type pixel
   pointed at by P. A record of type pixel has two fields, COL and
   EQC, corresponding to its color and a pointer to the equivalence
   class currently associated with it. */
begin
 value integer N;
 global integer MAXLABEL;
 global pointer eq_class stack EQ_CLASS_STACK,
  INTERMEDIATE;
 integer I,J;
 pointer pixel array P[0:N];
 pointer pixel PIXEL_ABOVE;
 MAXLABEL←0; /* Initialize time stamp */
 EQ_CLASS_STACK←empty;
 /* Initialize stack for intermediate output for the second pass: */
 INTERMEDIATE←empty;
 /* Initialize position to the left of the first pixel: */
 COL(P[0])←'WHITE';
 PIXEL_ABOVE←create(pixel);
 /* The image boundary is WHITE: */
 COL(PIXEL_ABOVE)←'WHITE';
 for J←1 step 1 until N do
  begin /* Initialize first row */
   P[J]←readpixel();
   if COL(P[J]) = BLACK then
    PROCESS_PIXEL(P[J],P[J-1],PIXEL_ABOVE);
  end;
 for I←1 step 1 until N-1 do
  begin /* Process rows 2 to N and output rows 1 to N-1 */
   for J←1 step 1 until N do
    begin
     COL(PIXEL_ABOVE)←COL(P[J]);
     EQC(PIXEL_ABOVE)←EQC(P[J]);
     P[J]←readpixel();
     if COL(P[J]) = BLACK then
      PROCESS_PIXEL(P[J],P[J-1],PIXEL_ABOVE);
     PUT_PIXEL(PIXEL_ABOVE);
    end;
  end;
 for J←1 step 1 until N do PUT_PIXEL(P[J]); /* Last row */
 PHASEII();
end;

procedure PROCESS_PIXEL(C,L,A);
/* Process pixel C. Pixel L is to the left of C and pixel A is above C.
   If they are all BLACK, then merge their equivalence classes. */
begin
 value pointer pixel C,L,A;
 if COL(L) = 'BLACK' then
  begin
   if COL(A) = BLACK then UNION_FIND(C,EQC(L),EQC(A))
   else ADD_PIXEL(C,L);
  end
 else if COL(A) = BLACK then ADD_PIXEL(C,FIND(EQC(A)))
 else EQC(C)←GET_EQ_CLASS();
end;

procedure ADD_PIXEL(C,E);
/* Add pixel C to equivalence class E. */
begin
 value pointer pixel C;
 value pointer eq_class E;
 EQC(C)←E;
 NACTIVE(E)←NACTIVE(E)+1;
end;

procedure UNION_FIND(C,L,A);
/* Set the equivalence class of pixel C to the result of the merge of
   the classes pointed at by L and the root of the FIND chain starting at A if they are not already the same. Retain the oldest of
   the two classes. Set the FATHER field of the younger class to
   point at the older one. Apply path compression to elements of the
   chain starting at A once the oldest class has been determined. */
begin
 value pointer pixel C;
 value pointer eq_class L,A;
 pointer eq_class E,O,T;
 E←A; /* Perform FIND without path compression: */
 while not null(FATHER(A)) do A←FATHER(A);
 if L = A then
  begin
   EQC(C)←L;
   NACTIVE(L)←NACTIVE(L)+1;
   O←L;
  end
 else if LABEL(L) > LABEL(A) then
  begin
   EQC(C)←A;
   FATHER(L)←A;
   NACTIVE(A)←NACTIVE(A)+1;
   O←A;
  end
 else
  begin
   EQC(C)←L;
   FATHER(A)←L;
```

```
    NACTIVE(L)←NACTIVE(L)+1;
    O←L;
   end;
   while FATHER(E) neq O and not null(FATHER(E)) do
   begin /* Path compression */
    T←FATHER(E);
    FATHER(E)←O;
    E←T;
   end;
  end;
  pointer eq_class procedure FIND(E);
  /* Determine the equivalence class containing the class pointed at by
     E. Perform path compression at the same time - i.e., when a class
     requires more than one link to reach the head of the class. In this
     case the appropriate link is set. */
  begin
   value pointer eq_class E;
   pointer eq_class R,T;
   if null(FATHER(E)) then return(E)
   else
    begin
    R←E;
    do R←FATHER(R) until null(FATHER(R));
    do
     begin /* Short circuit the path by linking E to R */
      T←FATHER(E);
      FATHER(E)←R;
      E←T;
     end
     until null(FATHER(E));
    return(R);
   end;
  end;

  pointer eq_class procedure GET_EQ_CLASS();
  /*    Generate an equivalence class record. Check if
     EQ_CLASS_STACK contains any that can be reused. If none,
     then allocate a new one. Each class record is of type eq_class
     with three fields, LABEL, NACTIVE, and FATHER, correspond-
     ing to a unique identifier that is associated with it, the number of
     active image elements that are members of it, and a pointer to the
     class into which it has been UNIONed respectively. When NAC-
     TIVE goes to zero the class is returned to EQ_CLASS_STACK by
     DELETE_ACTIVE and is available for reuse. */
  begin
   global pointer eq_class stack EQ_CLASS_STACK;
   global integer MAXLABEL;
   pointer eq_class E;
   E←if empty(EQ_CLASS_STACK) then create(eq_class)
     else pop(EQ_CLASS_STACK);
   MAXLABEL←MAXLABEL+1;
   LABEL(E)←MAXLABEL;
   NACTIVE(E)←1;
   FATHER(E)←NIL;
   return(E);
  end;

  procedure PUT_PIXEL(C);
  /* Output information about pixel C in the intermediate file (a stack
     pointed at by INTERMEDIATE). BLACK_PIXEL and
     WHITE_PIXEL are pointers to records of type eq_class which are
     used to denote a BLACK and WHITE pixel in the output respec-
     tively. Note that for a BLACK pixel the equivalence class is out-
     put first so that on the second pass the pointer to the
     BLACK_PIXEL class will be encountered first. */
  begin
   value pointer pixel C;
   global pointer eq_class BLACK_PIXEL,WHITE_PIXEL;
   global pointer eq_class stack INTERMEDIATE;
   if COL(C)='WHITE' then push(INTERMEDIATE,WHITE_PIXEL)
   else
    begin
     push(INTERMEDIATE,EQC(C));
     push(INTERMEDIATE,BLACK_PIXEL);
     DELETE_ACTIVE(EQC(C));
    end;
  end;

  procedure DELETE_ACTIVE(E);
  /* Decrement the counter of active image elements that are in the
     equivalence class pointed at by E. If this counter is 0, then no
     more image elements can be equivalent to this class and it is
     returned to EQ_CLASS_STACK for reuse. In addition, output a
     pointer to the class, if any, to which this class is equivalent. */
  begin
   value pointer eq_class E;
   global pointer eq_class stack EQ_CLASS_STACK,
    INTERMEDIATE;
   NACTIVE(E)←NACTIVE(E)-1;
   if NACTIVE(E) = 0 then
    begin
     push(EQ_CLASS_STACK,E); /* Recycle equivalence class */
     push(INTERMEDIATE,FATHER(E));
    end;
  end;
  procedure PHASEII();
  /* Process the output of the first pass in reverse order. This output is
     found in the stack of pointers to equivalence class records pointed
     at by INTERMEDIATE. As each stack item is popped, if it is a
     pointer to a WHITE pixel (i.e., WHITE_PIXEL), then no label is
     assigned. If it is a pointer to a BLACK pixel (BLACK_PIXEL),
     then pop the stack to get the next stack item and look up the
     label associated with the most current equivalence class binding
     by using FIND. Otherwise, the stack item popped originally
     points to a class which is the father of the class associated with
     the next two items on the stack (i.e., a BLACK pixel and its
     class). In this case, set the appropriate FATHER link, and if the
     father was NULL, then create a new label. */
  begin
   global pointer eq_class stack BLACK_PIXEL,WHITE_PIXEL,
    INTERMEDIATE;
   global integer MAXLABEL;
   pointer eq_class E,S;
   MAXLABEL←0; /* Initialize component counter */
   while not empty(INTERMEDIATE) do
    begin
     E←pop(INTERMEDIATE);
     if E = WHITE_PIXEL then output('WHITE')
     else if E = BLACK_PIXEL then
      output(LABEL(FIND(pop(INTERMEDIATE))));
     else /* A father pointer */
      begin /* Pop pointer to BLACK_PIXEL: */
       S←pop(INTERMEDIATE);
       S←pop(INTERMEDIATE); /* Pop equivalence class pointer */
       FATHER(S)←E;
       if null(E) then
        begin /* A new equivalence class */
         MAXLABEL←MAXLABEL+1;
         LABEL(S)←MAXLABEL;
         output(LABEL(S));
        end
       else
        begin
         FATHER(S)←FIND(E); /* Path compression */
         output(LABEL(FATHER(S)));
        end;
      end;
    end;
  end;
```