Connected Component Labeling Using Quadtrees

HANAN SAMET

University of Maryland, College Park, Maryland

ABSTRACT. An algorithm is presented for labeling the connected components of an image represented by a quadtree. The algorithm proceeds by exploring all possible adjacencies for each node once and only once. As soon as this is done, any equivalences generated by the adjacency labeling phase are propagated Analysis of the algorithm reveals that its average execution time is of the order ($W + B \cdot \log B$) where B and W correspond to the number of blocks comprising the foreground and background, respectively, of the image

KEY WORDS AND PHRASES: quadtrees, image processing, pattern recognition, connectivity

CR CATEGORIES: 3.63, 8.2

1. *Introduction*

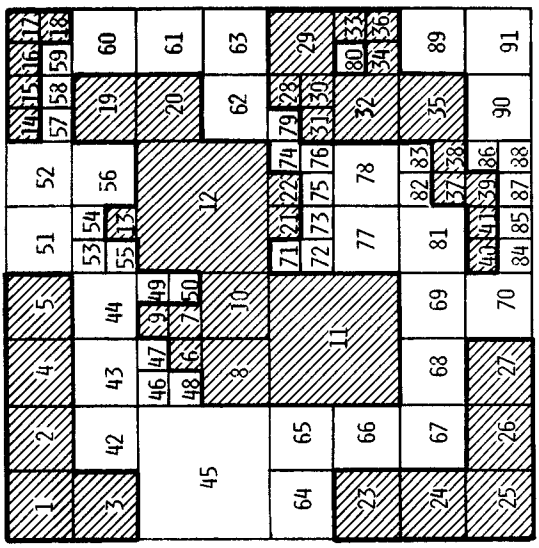
Connected component labeling [13, 14] is the process of identifying the disjoint elements of an image (e.g., the regions labeled 1–5 are the connected components of the image in Figure 1a). As such, it is a basic operation in image processing and is applied once the image has been segmented (e.g., the separation of object points from the background on the basis of a threshold value). Segmentation is the first step in processing an image and customarily results in the labeling of object points with a 1 and background points with a 0. Algorithms for object counting generally involve a connected component labeling process, even if this process is not explicit.

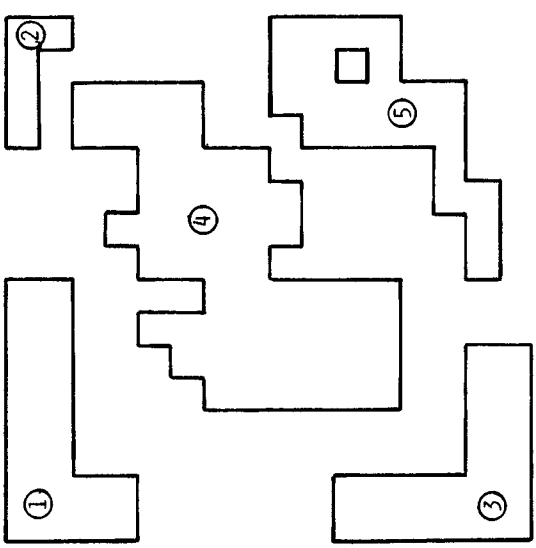
The most common representations used in image processing are the binary array and the run-length representation [14]. Using these representations, connected component labeling involves scanning the image by rows and forming equivalence classes. These equivalence classes are subsequently merged and the image's components labeled accordingly. This procedure has execution time on the order of the area (i.e., number of picture elements) plus the time required to process equivalences.

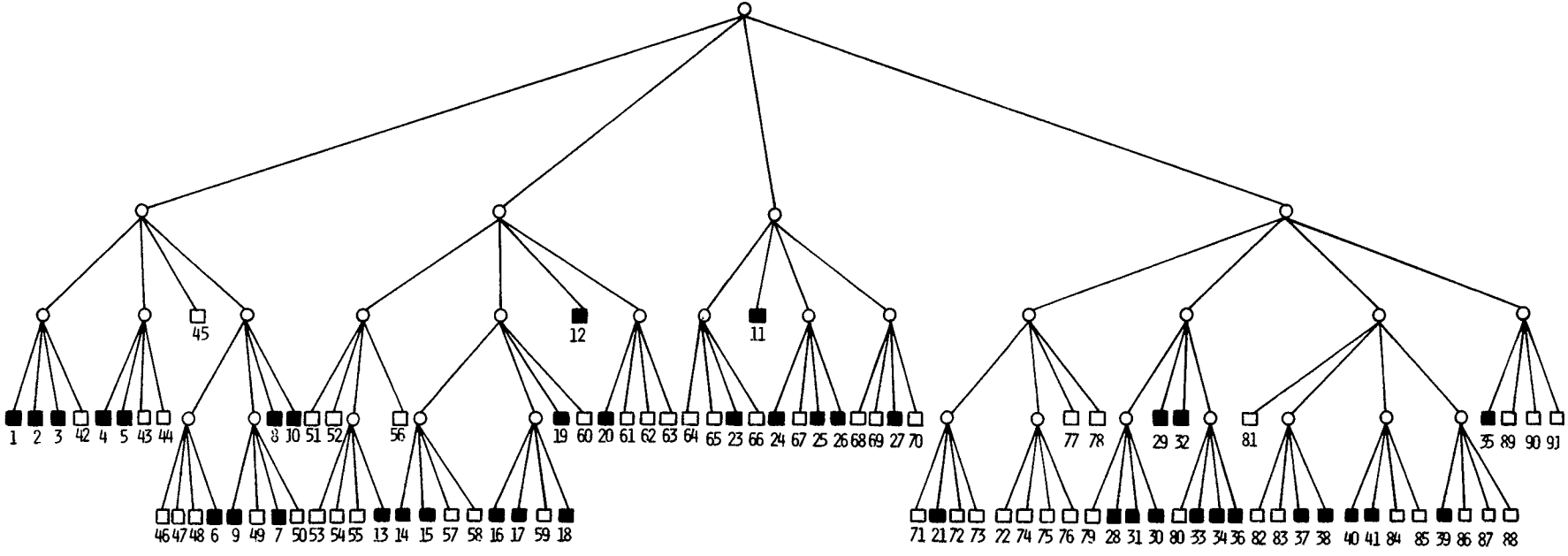
The quadtree is an approach to region representation which is based on the successive subdivision of an image array into quadrants. It results in the representation of an image as a collection of maximal blocks of standard sizes and positions (powers of 2). It can be quite compact and lends itself to set operations such as union and intersection [5, 6], as well as the computation of various region properties [5–8] (see below for a more formal definition of a quadtree). It was first proposed by Klinger [1, 8]; compare also [4, 12, 16, 17]. Recently it has been used by Hunter and Steiglitz [5–7] in the domain of computer graphics, and also in [3, 15] for cartographic applications.

This work was sponsored by the Defense Advanced Research Projects Agency and the U.S. Army Night Vision Laboratory under Contract DAAG-53-76C-0138 (DARPA Order 3206).

Author's address Computer Science Department, University of Maryland, College Park, MD 20742


1
2
3
4
5
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
(b)


1
2
3
4
5
(a)



(c)

FIG 1 An image, its maximal blocks, and the corresponding quadtree Blocks in the image are shaded. (a) Sample image. (b) Block decomposition of the image in (a) (c) Quadtree representation of the blocks in (b)

More formally, the quadtree is defined as follows. Without loss of generality, assume that the given binary image is a 2^n by 2^n array of unit square "pixels." If the image does not cover the entire array, then we subdivide the array into quadrants, subquadrants, ..., until we obtain blocks (possibly single pixels) that are entirely contained in the region or entirely disjoint from it (i.e., all 1's or 0's, respectively). This process is represented by a tree of outdegree 4 in which the root node represents the entire array. The four sons of the root node represent the quadrants, and the leaf nodes correspond to those blocks of the array for which no further subdivision is necessary. Since the array was assumed to be 2^n by 2^n, the tree height is at most n. It should be clear that the representation does not depend on the image containing only one region. As an example, Figure 1b is a block decomposition of the region in Figure 1a, while Figure 1c is the corresponding quadtree. In general, BLACK and WHITE square nodes are leaf nodes corresponding to blocks consisting entirely of 1's and 0's, respectively. Circular nodes, also termed GRAY nodes, denote nonterminal nodes.

In the following sections we present and analyze an algorithm for labeling the connected components of an image represented by a quadtree. It is based on an analogy with the method used in conjunction with the binary array representation. Included is a formal description of the algorithm along with motivating considerations. The actual algorithm is given using a variant of ALGOL 60 [11].

2. *Definitions and Notation*

Let each node in a quadtree be stored as a record containing seven fields. The first five fields contain pointers to the node's father and its four sons, labeled NW, NE, SE, and SW. Give a node P and a son I, these fields are referenced as FATHER(P) and SON(P, I), respectively. At times it is useful to use the function SONTYPE(P), where SONTYPE(P) = Q iff SON(FATHER(P), Q) = P. The sixth field, named NODETYPE, describes the contents of the block of the image which the node represents—that is, WHITE, if the block contains no 1's; BLACK, if the block contains only 1's; and GRAY, if it contains pixels of both types. Alternatively, BLACK and WHITE nodes are terminal nodes, while GRAY nodes are nonterminal nodes. The seventh field, named REGION, identifies the connected component containing the block represented by the node. This field is only meaningful for BLACK nodes. It is set as a result of the connected component labeling algorithm. LABELED(P) indicates whether node P has already been labeled.

Let the four sides of a node's block be called its N, E, S, and W sides. They are also termed its boundaries, and at times we speak of them as if they are directions (e.g., in Figure 1, node 3 is node 1's neighbor in the southern direction). The interrelationship between a block's four quadrants and its sides is facilitated by use of the predicate ADJ and the function REFLECT. ADJ(B, I) is true if and only if quadrant I is adjacent to side B of the node's block; for example, ADJ(N, NE) is true. REFLECT(B, I) yields the SONTYPE value of the block of equal size that is adjacent to side B of a block having SONTYPE value I; for example, REFLECT(W, NW) = NE, REFLECT(E, NW) = NE, REFLECT(N, NW) = SW, and REFLECT(S, NW) = SW. Figure 2 shows the relationship between the quadrants of a node and its boundaries.

Given a quadtree corresponding to a 2^n by 2^n array, we say that the root node is at level n, and that a node at level i is at a distance of $n - i$ from the root of the tree. In other words, for a node at level i we must ascend $n - i$ FATHER links to reach the root of the tree. Note that the farthest node from the root of the tree is at level ≥ 0.

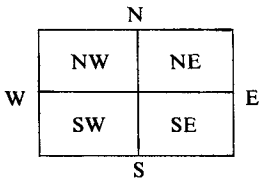


FIG 2 Relationship between a block's four quadrants and its boundaries

3. *Informal Description of the Algorithm*

The connected component labeling algorithm has three phases. The first phase traverses the tree and explores all possible adjacencies between pairs of BLACK nodes. During this process all BLACK nodes are labeled. Should any equivalences be discovered between regions already labeled, then their component identifiers are added to a list of pairs of equivalences. Once the entire tree is traversed in this manner, the second phase processes pairs of equivalences to yield equivalence classes (e.g., [9, 18]). Finally, the third phase traverses the tree one more time, assigning the same component identifier (i.e., label) to all members of an equivalence class.

Phase one traverses the tree in postorder (i.e., the sons of a node are visited first). In particular, the sons are visited in the order NW, NE, SW, and SE. For each BLACK terminal node, say P, we explore the eastern and southern adjacencies. This means that all of the node's BLACK adjacent southern and eastern neighbors are visited. If they have not been previously visited, then they are labeled with the label of P. If P does not already have a label, then it is assigned the label of one of its adjacent neighbors if it has a label. If adjacent BLACK nodes have already been assigned labels that are different, then the labels are added to the list of equivalences that will be merged in the second phase.

The key to the algorithm is that phase one ensures that every adjacency of two BLACK nodes will be explored once and only once. To see this, note that the traversal starts at the NW-most son, if possible, and the brothers are traversed in the order NW, NE, SW, and SE. Clearly, by the time any BLACK node is visited, its northern and western adjacencies have already been explored. Thus the northern and western adjacencies need not be reexplored. This is because each node labels all of its adjacent eastern and southern neighbors.

As an example of the application of the algorithm, consider the image given in Figure 1a. Figure 1b is the corresponding block decomposition, and Figure 1c is its quadtree representation. All of the BLACK nodes have numbers ranging between 1 and 41, while the WHITE nodes have numbers ranging between 42 and 91. The BLACK nodes have been numbered in the order in which they were labeled by phase one. The WHITE nodes have been numbered in the order in which they were visited (i.e., the argument to procedure LABEL). Thus node 1 has been labeled before nodes 2, 3, etc. Figure 3 shows the labels assigned to the five components. Phase two of the algorithm will merge the equivalence pair $B \equiv D$ to form component 4 and the equivalence pairs $F \equiv G$ and $G \equiv H$ to form component 5.

4. *Formal Statement of the Algorithm*

The following ALGOL-like procedures specify the connected component labeling algorithm. Actually, we only present the procedures corresponding to the first and third phases of the algorithm. Phase two can be achieved by using a variant of Algorithm E in [9, p. 354].

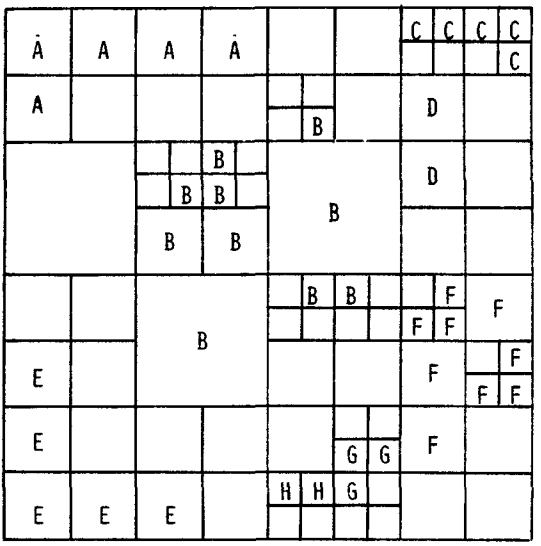


FIG. 3. Result of the application of phase one to Figure 1b.

The main procedure is termed COMPONENT and is invoked with a pointer to the root of the quadtree representing the image. The global variable MERGES is used to accumulate all the equivalence relations formed by adjacent BLACK nodes. MERGES is subsequently processed by phase two to yield a set of equivalence classes—that is, one class per component. LABEL implements phase one by traversing the tree and controlling the exploration of adjacent BLACK nodes. GTEQUAL_ADJ_NEIGHBOR locates a neighboring node of greater or equal size along a specified side of a given node's corresponding block. If no such neighboring BLACK or WHITE node exists, then GTEQUAL_ADJ_NEIGHBOR returns a pointer to a GRAY node of equal size. If this is also impossible, then the node is adjacent to the border of the image, and NULL is returned. In the case of a GRAY node, procedure LABEL_ADJACENT continues the search recursively by examining all BLACK and WHITE adjacent neighbors of smaller size. Otherwise, LABEL_ADJACENT assigns a label to the adjacent neighbor if it is BLACK. Unique labels are generated by procedure GENREGION (not given here) and assigned by procedure ASSIGN_LABEL. Procedure UPDATE corresponds to phase three and results in the postorder traversal of the tree in order to propagate the equivalences, thereby uniquely labeling each component.

```
procedure COMPONENT(QUADTREE),
/*Label all of the connected components of the tree rooted at QUADTREE*/
begin
    value node QUADTREE;
    pairlist MERGES;
    MERGES ← empty;
    LABEL(QUADTREE),
    Process equivalences specified by MERGES,
    UPDATE(QUADTREE);
end;


procedure LABEL(P),
/*Assign labels to node P and its sons*/
```

```
begin
    value node P;
    node Q,
    quadrant I,
    if GRAY(P) then
        begin
            for I in {'NW', 'NE', 'SW', 'SE'} do LABEL(SON(P, I));
        end
    else if BLACK(P) then
        begin
            Q ← GTEQUAL_ADJ_NEIGHBOR(P, 'E');
            if not NULL(Q) then LABEL_ADJACENT(Q, 'NW', 'SW', P),
            Q ← GTEQUAL_ADJ_NEIGHBOR(P, 'S');
            if not NULL(Q) then LABEL_ADJACENT(Q, 'NW', 'NE', P);
            if not LABELED(P) then REGION(P) ← GENREGION ( ),
        end
    else return, /*A WHITE node*/
end;

node procedure GTEQUAL_ADJ_NEIGHBOR(P, D);
/*Return the neighbor of node P in horizontal or vertical direction D which is greater than or equal in size
to P. If such a node does not exist, then a GRAY node of equal size is returned If this is also impossible,
then the node is adjacent to the border of the image and NULL is returned*/
begin
    value node P;
    node Q,
    value direction D,
    if (not NULL(FATHER(P))) and ADJ(D, SONTYPE(P)) then
        /*Find a common ancestor*/
        Q ← GTEQUAL_ADJ_NEIGHBOR(FATHER(P), D)
    else Q ← FATHER(P);
    /*Follow the reflected path back to locate the neighbor*/
    return (if not NULL(Q) and GRAY(Q) then SON(Q, REFLECT(D, SONTYPE(P)))
            else Q),
end,

procedure LABEL_ADJACENT(R, Q1, Q2, P),
/*Find all descendants of node R adjacent to node P—i e , in quadrants Q1 and Q2*/
begin
    value node P, R,
    value quadrant Q1, Q2,
    if GRAY(R) then
        begin
            LABEL_ADJACENT(SON(R, Q1), Q1, Q2, P),
            LABEL_ADJACENT(SON(R, Q2), Q1, Q2, P),
        end
    else if BLACK(R) then ASSIGN_LABEL(P, R)
    else return, /*A WHITE node*/
end,

procedure ASSIGN_LABEL(P, Q),
/*Assign a label to nodes P and Q if they do not already have one If both have different labels, then enter
them in MERGES*/
begin
    value node P, Q,
    if LABELED(P) and LABELED(Q) then
        begin
            if REGION(P) ≠ REGION(Q) then add ⟨REGION(P), REGION(Q)⟩ to MERGES,
        end
    else if LABELED(P) then REGION(Q) ← REGION(P)
```

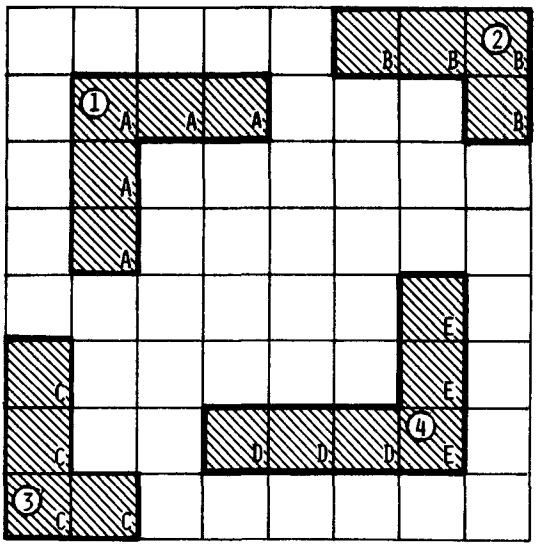


FIG 4. Sample component shapes and the labels that they generate.

```
    else if LABELED(Q) then REGION(P) ← REGION(Q)
    else REGION(P) ← REGION(Q) ← GENREGION( );
end;

procedure UPDATE(P);
/*Propagate the equivalences represented by MERGES in the quadtree rooted at P*/
begin
    value node P;
    quadrant I;
    if GRAY(P) then
        begin
            for I in {'NW', 'NE', 'SW', 'SE'} do UPDATE(SON(P, I));
        end
    else if BLACK(P) then REGION(P) ← LOOKUP (REGION(P), MERGES)
    else return; /*A WHITE node*/
end;
```

5. *Analysis*

The running time of the connected component labeling algorithm is determined by the time necessary to execute its three phases. Prior to analyzing this value we first examine the spatial configurations confronted by the algorithm and how they affect its execution time. It should be clear that the greater the number of BLACK nodes, the more time is spent exploring adjacencies in phase one. Phase two is more dependent on the shapes of the various components. The execution time of phase two is dominated by the number of equivalence pairs that are generated in phase one. An equivalence pair is generated whenever an adjacency of a previously labeled node is explored and it is found that the adjacent neighbor has already been assigned a different label.

The situation giving rise to the generation of an equivalence pair can be best seen by examining Figure 4. Components 1–3 do not result in the generation of equivalence pairs because of the manner in which phase one explores adjacencies—that is, in the eastern and southern directions, thereby processing the quadrants in the order NW, NE, SW, SE. Thus for components 1–3 we see that the nodes or blocks

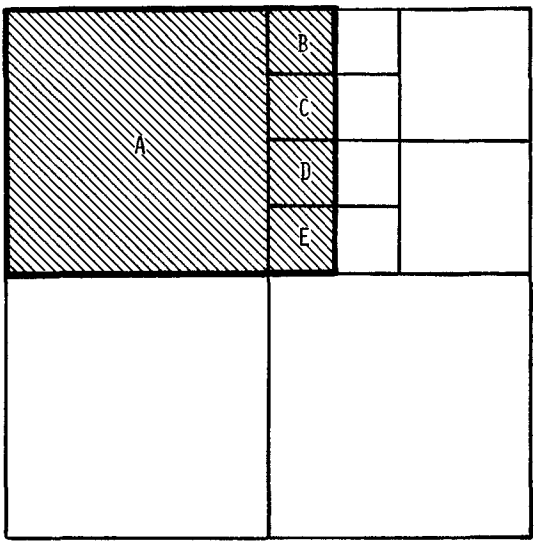


FIG. 5. Sample image demonstrating the maximum number of nodes that must be visited for an adjacency.

comprising the quadtree are processed in the order in which they are adjacent. However, this is not always the case for a component having the form of component 4 in Figure 4 (in this case we have the equivalence of D and E). This is especially true when the vertical and horizontal segments are not comprised of single blocks. For example, in the image represented by Figure 1a we find that no equivalence pairs were generated for the components 1–3, whereas this was not true for the components 4 and 5. In particular, as demonstrated in Figure 3, we have the equivalences $B \equiv D$ for component 4 and $F \equiv G$ and $G \equiv H$ for component 5. Note that if the block labeled 40 had been WHITE rather than BLACK, then block 41 would have been labeled with G and no equivalence pair would have been generated.

Phase one depends on the speed of the combination of procedures LABEL_ADJACENT and GTEQUAL_ADJ_NEIGHBOR. These procedures are invoked in phase one (i.e., in procedure LABEL) twice as many times as one has BLACK nodes. The actual amount of work performed by these procedures is more accurately represented by considering the number of nodes that are visited when an adjacency is being explored. Recall that we must find the neighbor, and, if it is GRAY, then visit all adjacent neighbors of a smaller size. In the worst case we are at level $n - 1$, with a GRAY neighbor and all adjacent neighbors at level 0. In such a case we must visit 2^n nodes. For example, consider Figure 5 where $n = 3$ and we wish to visit the blocks adjacent to the block labeled A (i.e., blocks B, C, D, and E). We must visit the root of the quadtree as well as A's neighboring GRAY node and all of its NW and SW sons—that is, a complete binary tree of height 2 (termed A's adjacency tree in [5]). In total, $2^3 = 8$ nodes are visited. In general, let the space be partitioned into a 2^n by 2^n array. Assume a random image in the sense that a BLACK node is equally likely to appear in any position and level in a quadtree. This means that we assume that all neighbor pairs (i.e., configurations of adjacent nodes of varying sizes) have equal probability. This is different from the more conventional notion of a random image which implies that every block at level 0 (i.e., pixel) has an equal probability of being BLACK or WHITE. Such an assumption would lead to a very low probability of any nodes corresponding to blocks of size larger than 1.

Clearly, for such an image the quadtree is the wrong representation. We have the following result.

THEOREM 1. *The average of the maximum number of nodes visited by LABEL_ADJACENT is five.*

PROOF. Given a node P at level i and a direction D, there is a maximum of $2^{n-i} \cdot (2^{n-i} - 1)$ neighbor pairs. $2^{n-i} \cdot 2^0$ have their nearest common ancestor at level n, $2^{n-i} \cdot 2^1$ at level $n-1, \ldots,$ and $2^{n-i} \cdot 2^{n-i-1}$ at level $i+1$. For each node at level i having a common ancestor at level j, the maximum number of nodes that will be visited by GTEQUAL_ADJ_NEIGHBOR and SUM_ADJACENT is

$$(j-i) + (j-i-1) + \sum_{k=0}^{i} 2^k = 2 \cdot (j-i-1) + 2^{i+1}.$$

This is obtained by observing that the common ancestor is at a distance of $j-i$ and that a node at level i has a maximum of 2^i adjacent neighbors (all appearing at level 0). Assuming that node P is equally likely to occur at any level i and at any of the $2^{n-i} \cdot (2^{n-i} - 1)$ positions at level i, then the average of the maximum number of nodes visited by GTEQUAL_ADJ_NEIGHBOR and SUM_ADJACENT is

$$\frac{\sum_{i=0}^{n-1} \sum_{j=i+1}^{n} 2^{n-i} \cdot 2^{n-j} \cdot (2 \cdot (j-i-1) + 2^{i+1})}{\sum_{i=0}^{n-1} 2^{n-i} \cdot (2^{n-i} - 1)}. \tag{1}$$

Expression (1) can be rewritten to yield

$$\frac{\sum_{i=0}^{n-1} \sum_{j=0}^{n-1-i} 2^{2n-2i-1-j} \cdot (2j + 2^{i+1})}{\sum_{i=0}^{n} 2^i \cdot (2^i - 1)}. \tag{2}$$

The numerator of (2) can be rewritten as follows:

$$\sum_{i=0}^{n-1} 2^{2n-2i} \cdot \sum_{j=0}^{n-1-i} \frac{j}{2^j} + \sum_{i=0}^{n-1} 2^{2n-i} \cdot \sum_{j=0}^{n-1-i} \frac{1}{2^j}. \tag{3}$$

But

$$\sum_{j=0}^{n-1-i} \frac{j}{2^j} = 2 - \frac{n+1-i}{2^{n-1-i}}. \tag{4}$$

Also,

$$\sum_{j=0}^{n-1-i} \frac{1}{2^j} = 2 \cdot \left(1 - \frac{1}{2^{n-i}}\right). \tag{5}$$

Substituting (4) and (5) into (3) yields

$$\sum_{i=0}^{n-1} \left(2^{2n-2i} \left(2 - \frac{n+1-i}{2^{n-1-i}}\right) + 2^{2n-i} \cdot 2 \left(1 - \frac{1}{2^{n-i}}\right)\right)$$

$$= 2^{2n+1} \cdot \sum_{i=0}^{n-1} \frac{1}{2^{2i}} - 2^{n+1} \cdot \sum_{i=0}^{n-1} \frac{n+1}{2^i} + 2^{n+1} \cdot \sum_{i=0}^{n-1} \frac{i}{2^i}$$

$$+ 2^{2n+1} \cdot \sum_{i=0}^{n-1} \frac{1}{2^i} - n \cdot 2^{n+1}. \tag{6}$$

But

$$\sum_{i=0}^{n} \frac{1}{2^{2i}} = \frac{1}{3} \left(4 - \frac{1}{2^{2n}}\right). \tag{7}$$

Making use of (7) in (6) leads to

$$\sum_{i=0}^{n-1}\left(2^{2n-2i}\left(2-\frac{n+1-i}{2^{n-1-i}}\right)+2^{2n-i}\cdot 2\left(1-\frac{1}{2^{n-i}}\right)\right)$$

$$=2^{2n+1}\cdot\frac{4}{3}\cdot\left(1-\frac{1}{2^{2n}}\right)-2^{n+2}\cdot(n+1)\cdot\left(1-\frac{1}{2^n}\right)$$

$$+2^{n+1}\cdot\left(2-\frac{n+1}{2^{n-1}}\right)+2^{2n+2}\cdot\left(1-\frac{1}{2^n}\right)-n\cdot 2^{n+1}$$

$$=\frac{20}{3}\cdot 2^{2n}-(3n+2)\cdot 2^{n+1}-\frac{8}{3}. \tag{8}$$

The denominator of (2) can be simplified as follows:

$$\sum_{i=0}^{n}2^i\cdot(2^i-1)=\sum_{i=0}^{n}4^i-\sum_{i=0}^{n}2^i$$

$$=\frac{4^{n+1}-1}{3}-(2^{n+1}-1)$$

$$=\frac{1}{3}\cdot(2^{2n+2}-3\cdot 2^{n+1}+2). \tag{9}$$

Substituting (8) and (9) into (2) yields

$$\frac{\frac{20}{3}\cdot 2^{2n}-(3n+2)\cdot 2^{n+1}-\frac{8}{3}}{\frac{1}{3}\cdot(2^{2n+2}-3\cdot 2^{n+1}+2)}=5-\frac{3\cdot(3n+7)\cdot 2^{n+1}+18}{2^{2n+2}-3\cdot 2^{n+1}+2}$$

$$\approx 5 \quad \text{as} \quad n \quad \text{gets large}$$

$$\leq 5. \qquad \text{Q.E.D.}$$

The speed of phase two of the algorithm depends on the method used for processing equivalence relations and on the number of pairs of equivalences and different objects of the set on which the equivalences are defined. We use a variant of Algorithm E presented in [9, p. 354]. The basis of this algorithm is the construction of a set of trees such that each tree corresponds to an equivalence class. In particular, the root of the tree denotes the head of the equivalence class, while its subtrees correspond to members of the equivalence class. Algorithm E has a maximum execution time that is proportional to the square of the number of equivalence pairs. It can be speeded up by use of the following rules [9, p. 572]. The weighting rule ensures that whenever two equivalence classes are merged, the class containing the fewest elements is made a subtree of the tree corresponding to the other class. The collapsing rule stipulates that whenever the head of an equivalence class is sought (i.e., the appropriate tree is searched), all nodes accessed during the process are made direct sons of the nodes corresponding to the head of the equivalence class. Using such methods, equivalences can be processed in time proportional to the product of the number of equivalence pairs and the log of the size of the set on which the equivalences are defined. For an even better time estimate, although still not a linear one, see [18].

Recall that equivalence pairs are generated during phase one only when we are exploring the adjacencies of a node that is already labeled and whose neighbor has also been labeled before, albeit with a different label. We now prove the following lemma.

LEMMA 1. *Phase one generates a maximum of one equivalence pair for each adjacency that is explored (i.e., each call to procedure LABEL explores two adjacencies).*

PROOF. There are two cases depending on the direction of the adjacency.

Case a. An adjacency in the eastern direction can yield at most one equivalence pair regardless of the size of the neighbor. This is clearly true if the neighbor is larger (e.g., blocks 38 and 35 in Figure 1b). Similarly, if the neighbor is smaller, then only the northernmost such neighbor could have been previously labeled (e.g., blocks 12 and 20 in Figure 1b) because only it could have been the southern neighbor of a previously labeled node.

Case b. An adjacency in the southern direction can only yield an equivalence pair if the neighbor is larger. No equivalence pair may result if the neighbor is smaller. This should be clear, since southern neighbors could only be labeled already if they are adjacent to a western neighbor which has been visited previously. Q.E.D.

Letting B denote the number of BLACK nodes, we have the following theorem.

THEOREM 2. *$2 \cdot B \cdot \log B$ is an upper bound on the execution time of phase two.*

PROOF. By the above lemma, phase one generates a maximum of one equivalence pair for each adjacency that is explored. Recall that phase one explores two adjacencies for each BLACK node. Also, the set on which the equivalence pairs are defined has a maximum number of objects equal to the number of BLACK nodes, that is, B. Thus when the speeded up equivalence merging algorithm of [9] is used, we have $2 \cdot B \cdot \log B$ as the upper bound for the execution time. Q.E.D.

The speed of phase three of the algorithm can be obtained in a straightforward way. The quadtree must be traversed, and for each BLACK node P, REGION(P) must be set to the head of the equivalence class obtained as a result of phase two. Since collapsing is assumed to have taken place during phase two, the lookup operation takes a constant amount of time (i.e., traverse one link in the tree of equivalence classes). Letting B and W correspond to the number of BLACK and WHITE leaf nodes in the quadtree, respectively, we obtain the total number of nodes in the quadtree as follows.

LEMMA 2. *The number of nodes in a quadtree having B and W leaf nodes is bounded by $\frac{4}{3} \cdot (B + W)$.*

PROOF. Let G denote the number of nonterminal nodes. Given G nonterminal nodes and $B + W$ terminal nodes, we have $G + B + W - 1$ edges (since the tree is an acyclic graph). Counting another way, by the number of sons, we have that there are $4 \cdot G$ edges. Thus $4G = G + B + W - 1$, or $G + B + W = (4 \cdot (B + W) - 1)/3$. But $G + B + W$ corresponds to the number of nodes in the quadtree, and our result follows. Q.E.D.

THEOREM 3. *The upper bound of the execution time of phase three is proportional to $\frac{4}{3} \cdot (B + W)$.*

PROOF. A direct result of Lemma 2, since each node is visited once. Q.E.D.

Using Lemma 2, we obtain an upper bound on the average execution time of phase one.

THEOREM 4. *The upper bound on the average execution time of phase one is $\frac{1}{3}(34 \cdot B + 4 \cdot W)$.*

PROOF. From Theorem 1 we have that for each adjacency involving a BLACK node, phase one results in a bounded average of five nodes being visited. Recall that two adjacencies are visited for each node. Also, from Lemma 2 we have that the tree traversal component of phase one visits $\frac{4}{3} \cdot (B + W)$ nodes. Therefore, we have $2 \cdot B \cdot 5 + \frac{4}{3} \cdot (B + W) = \frac{1}{3} \cdot (34 \cdot B + 4 \cdot W)$ nodes being visited. Q.E.D.

At this point we come to the main result.

THEOREM 5. *The average execution time of the connected component labeling algorithm is of order* $(W + B \cdot \log B)$.

PROOF. A direct result of the contributions of phases one, two, and three, as indicated by Theorems 4, 2, and 3, respectively. Q.E.D.

Connected component labeling can also be achieved by adapting the initial stage of the quadtree transformation algorithm of Hunter and Steiglitz [5, 7]. This involves labeling the BLACK blocks on the boundary of each component and then propagating the labels to the interior nodes. Using such a technique, connected component labeling could be done in time proportional to the number of leaf nodes. This is due, in part, to their use of additional links in their quadtree representation which are called ropes and nets. In essence, these links enable them to avoid the GTEQUAL_ADJ_NEIGHBOR and LABEL_ADJACENT procedures that we execute. They do so by explicitly storing such information with each leaf node through the use of links. This requires considerably more space than our representation, which only uses SON and FATHER links.

6. *Concluding Remarks*

An algorithm has been presented for labeling the connected components of a binary image that is represented by a quadtree. The algorithm's average execution time was shown to be of order $(W + B \cdot \log B)$, where B and W correspond to the number of blocks comprising the objects and the background of the image, respectively. It was also shown that the number of BLACK blocks (i.e., the image complexity) dominates the execution time of the algorithm. The $B \cdot \log B$ term arises from the need to process equivalences. Lower bounds, although not linear, can be obtained by use of an algorithm such as that presented in [18]. In general, processing equivalences is not really a problem. This is because in actuality very few equivalence pairs are generated. An example of the worst case, in terms of the number of equivalence pairs that are generated, for a 2^3 by 2^3 image is given in Figure 6. Note that some variant of the worst case in terms of configurations leading to the generation of an equivalence pair will arise no matter which order of traversing the adjacencies is adopted (i.e., NW, SW, NE, and SE sons in order instead of NW, NE, SW, and SE sons in order as done in our algorithm).

It should be clear that phase two of our algorithm can be combined with phase one by performing the merge dictated by the equivalence immediately in procedure ASSIGN_LABEL instead of using the list MERGES and executing phase two. We chose the present approach in order to simplify the presentation of the analysis. Also note that Lemma 1 ensures that the upper bound of the execution time of phase two is not affected by generating the same equivalence pair more than once (e.g., in Figure 7 the equivalence $A \equiv B$ is generated once by blocks 6 and 2 and once by blocks 9 and 2).

The importance of our connected component labeling algorithm is that it lends support to the usefulness of the quadtree representation, aside from the obvious

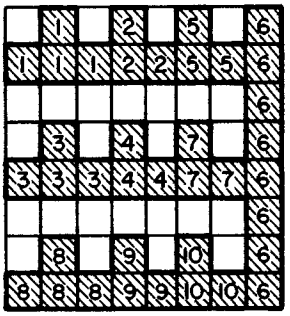


FIG. 6 Sample image for $n = 3$ which results in the generation of a maximum number of equivalence pairs.

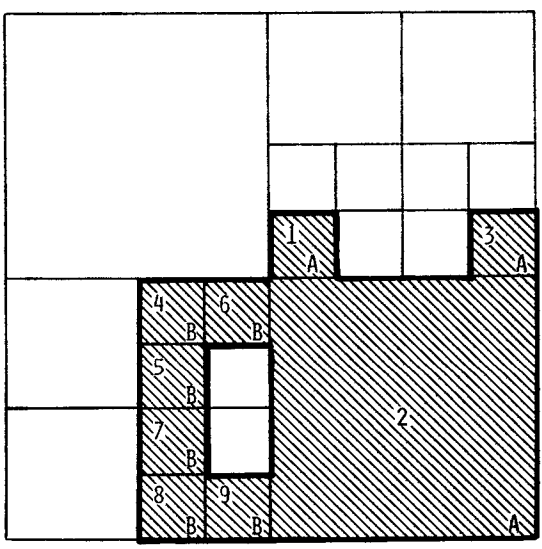


FIG. 7 Sample image demonstrating the generation of the same equivalence pair more than once.

benefits gained from its compactness, by showing that methods used in binary array image representations can be adapted to the quadtree. In particular, we presented a method which scans the image in a sequential manner (i.e., a postorder traversal of the quadtree) and generates a number of equivalences that are bounded by twice the number of blocks. The analogous binary array result is obtained by scanning the image a row at a time, starting at the top from left to right, and labeling adjacencies to the right and down. The number of equivalences is bounded by twice the number of pixels (i.e., blocks). Thus we see that by aggregating pixels into larger size blocks we obtain an algorithm whose execution time is proportional to the number of blocks comprising the image and not their size. Dyer has obtained an analogous result in [2] where he observes that one may apply the connected component labeling algorithm presented here to both the BLACK and WHITE nodes, thereby also yielding the number of holes in the image, which enables the computation of the genus of the image. In fact, Dyer also shows that the genus of an image represented by a quadtree can be computed in the same manner as presented by Minsky and

Papert [10] for the pixel representation—that is, by counting the number of occurrences of various local patterns in the image. The importance of Dyer's result is that it serves to reinforce our observation that it is the number of blocks that is critical and not the area which they comprise.

ACKNOWLEDGMENTS. I would like to thank Kathryn Riley for typing the manuscript and Pat Young for drawing the figures. I have benefitted greatly from discussions with Charles R. Dyer, Jack Minker, Paul McMullin, and Azriel Rosenfeld.

REFERENCES

1. ALEXANDRIDIS, N., AND KLINGER, A. Picture decomposition, tree data structures, and identifying directional symmetries as node combinations. *Comput. Graph. Image Proc. 8* (1976), 43–77.
2. DYER, C. R. Computing the Euler number of an image from its quadtree. *Comput. Graph. Image Proc. 13* (1980), 270–276.
3. DYER, C. R., ROSENFELD, A., AND SAMET, H. Region representation: Boundary codes from quadtrees. *Commun. ACM 23*, 3 (March 1980), 171–179.
4. HOROWITZ, S. L., AND PAVLIDIS, T. Picture segmentation by a tree traversal algorithm. *J. ACM 23*, 2 (April 1976), 368–388.
5. HUNTER, G. M. Efficient computation and data structures for graphics. Ph.D. Diss., Dep. of Electrical Engineering and Computer Science, Princeton Univ., Princeton, N.J., 1978.
6. HUNTER, G. M., AND STEIGLITZ, K. Operations on images using quadtrees. *IEEE Trans. Pattern Anal. Machine Intell. PAMI-1*, 2 (April 1979), 145–153.
7. HUNTER, G. M., AND STEIGLITZ, K. Linear transformation of pictures represented by quadtrees. *Comput. Graph. Image Proc. 10* (1979), 289–296.
8. KLINGER, A., AND DYER, C. R. Experiments in picture representation using regular decomposition. *Comput. Graph. Image Proc. 5* (1976), 68–105.
9. KNUTH, D. E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 2nd ed. Addison-Wesley, Reading, Mass., 1973, pp. 353–355, 360, 572.
10. MINSKY, M., AND PAPERT, S. *Perceptrons—An Introduction to Computational Geometry*. M.I.T. Press, Cambridge, Mass. 1969.
11. NAUR, P., ED. Revised report on the algorithm language ALGOL 60. *Commun. ACM 3*, 5 (May 1960), 299–314.
12. RISEMAN, E. M., AND ARBIB, M. A. Computational techniques in the visual segmentation of static scenes. *Comput. Graph. Image Proc. 6* (1977), 221–276.
13. ROSENFELD, A. Connectivity in digital pictures. *J. ACM 17*, 1 (Jan. 1970), 146–160.
14. ROSENFELD, A., AND KAK, A. C. *Digital Picture Processing*. Academic Press, New York, 1976, Sec. 8.1.
15. SAMET, H. Region representation: Quadtrees from boundary codes. *Commun. ACM 23*, 3 (March 1980), 163–170.
16. TANIMOTO, S. L. Pictorial feature distortion in a pyramid. *Comput. Graph. Image Proc. 5* (1976), 333–352.
17. TANIMOTO, S. L., AND PAVLIDIS, T. A hierarchical data structure for picture processing. *Comput. Graph. Image Proc. 4* (1975), 104–119.
18. TARJAN, R. E. On the efficiency of a good but not linear set union algorithm. Tech. Rep. 72-148, Computer Science Dep., Cornell Univ., Ithaca, New York, November 1972.

RECEIVED MAY 1979, REVISED MARCH 1980, ACCEPTED APRIL 1980