Correspondence

Efficient Component Labeling of Images of Arbitrary Dimension Represented by Linear Bintrees

HANAN SAMET AND MARKKU TAMMINEN

Abstract—**An algorithm is presented to perform connected component labeling of images of arbitrary dimension that are represented by a linear bintree. The bintree is a generalization of the quadtree data structure that enables dealing with images of arbitrary dimension. The linear bintree is a pointerless representation. The algorithm uses an active border which is represented by linked lists instead of arrays. This results in a significant reduction in the space requirements, thereby making it feasible to process three- and higher dimensional images. Analysis of the execution time of the algorithm shows almost linear behavior with respect to the number of leaf nodes in the image, and empirical tests are in agreement. The algorithm can be modified easily to compute a $(d-1)$-dimensional boundary measure (e.g., perimeter in two dimensions and surface area in three dimensions) with linear performance.**

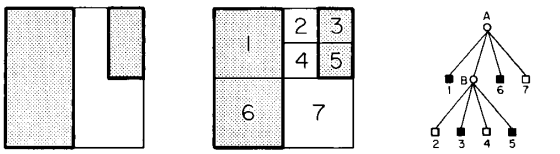
Index Terms—**Computer-aided design, computer graphics, connected component labeling. DF-expressions, hierarchical data structures, image processing, linear quadtress, octrees, quadtrees.**

I. Introduction

Hierarchical data structures such as the region quadtree [7] and the octree [3], [4], [9] have been the subject of much research in recent years (for a survey, see [13]). Their variants have found use in a number of domains including image processing, computer graphics, cartography, and computer-aided design.

Quadtrees can be implemented in a number of different ways. The most common implementation is in the form of an explicit tree (Fig. 1) which requires the use of pointers. In the interest of saving space, pointerless quadtree representations in the form of lists are often used. They can be classified into two categories. The first treats the image as a collection of leaf nodes (termed a *linear quadtree* in [2]) where each leaf is encoded by a base 2^d number (4 for $d = 2$), termed a *location code*, corresponding to a sequence of directional codes that locate the leaf along a path from the root of the tree. The second represents the image in the form of a preorder traversal of the nodes of its quadtree (termed a *DF-expression* [6]). We shall use the term linear quadtree for the entire class of pointerless quadtree representations.

In our discussion, we are primarily concerned with binary images. Two pixels of a two-dimensional image are said to be *4-adjacent* if they are adjacent to each other in the horizontal or vertical directions. A BLACK region is a maximal *four-connected* set of BLACK pixels—that is, a set S such that for any pixels, p, q, in S, there exists a sequence of pixels $p = p_0, p_1, \cdots, p_n = q$ in S such that p_{i+1} is 4-adjacent to p_i, $0 \leq i < n$. BLACK regions are termed *components*. A pixel is said to have four edges, each of which is of unit length. Similar definitions can be formulated in terms of blocks for images represented by quadtrees. For example, two disjoint blocks P and Q are said to be *4-adjacent* if there exists a pixel p in P and a pixel q in Q such that p and q are 4-adjacent. 8-adjacency for blocks is defined analogously.



(a) (b) (c)

Fig. 1. An image, its maximal blocks, and the corresponding quadtree. Blocks in the image are shaded; background blocks are blank. (a) Image. (b) Block decomposition of the image in (a). (c) Quadtree representation of the blocks in (a).

Now, consider a d-dimensional image represented as an array of d-dimensional pixels termed *image elements*. Each image element has $2 \cdot d$ *borders* (e.g., an edge in two dimensions and a face in three dimensions), each of which has unit size. Two image elements are said to be *4-adjacent* if they are adjacent in the sense that they share a border in its entirety (i.e., it has a nonzero $(d-1)$-dimensional measure). BLACK regions are defined analogously as in two dimensions and likewise for blocks.

Connected component labeling [11] is a fundamental task common to virtually all image processing applications in two as well as in three dimensions. For a binary image, represented as an array of d-dimensional pixels (or a collection of d-dimensional blocks in the case of quadtrees, octrees, etc.), it is the process of assigning the same label to all 4-adjacent BLACK image elements.

Connected component labeling is also a problem in graph theory. A connection between the image and graph problems is accomplished by conceptualizing the image as an undirected graph and searching the graph for connected image elements. Formally, we say that a *vertex* corresponds to a BLACK image element and an *edge* corresponds to a $(d-1)$-dimensional adjacency between two BLACK image elements (i.e., they are connected).[1] Finding the connected components of the undirected graph of the image yields its *connection graph*.[2] The two principal approaches to performing connected component labeling correspond loosely to the two ways in which a graph can be searched—i.e., depth-first and breadth-first 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 that 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* [19]) 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

Manuscript received July 22, 1986; revised October 12, 1987. This work was supported in part by the National Science Foundation under Grant DCR-8605557 and in part by the Finnish Academy.

H. Samet is with the Department of Computer Science, University of Maryland, College Park, MD 20742.

M. Tamminen is with the Laboratory for Information Processing Science, Helsinki University of Technology, Espoo, Finland.

IEEE Log Number 8718594.

[1] In the case of a nonbinary image, it is necessary to redefine the concept of a vertex and an edge to include color information.

[2] Note that in [15], the concepts of an edge and a vertex are different.

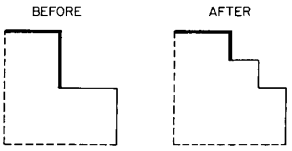


Fig. 2. The active border before and after processing block 4 in Fig. 1.

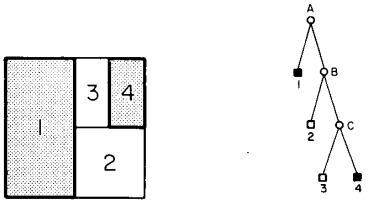
and the right subtree to positive values. The linear bintree for the image of Fig. 7 has (*B*(((BWWB as its DF expression.
lence class on the FIND path is more than one link away from the root of its tree. Stage two assigns a final label to each BLACK image element (corresponding to the component of which it is a member). When path compression is used, the results of Tarjan and van Leeuwen [20] let us deduce that the worst case execution time of the total (i.e., both stages) task is almost linear.

Use of a quadtree representation for two-dimensional images results in connected component labeling algorithms whose computational complexity is on the order of the number of leaf nodes in the tree rather than the number of pixels [12]. Similar techniques are used in octree systems. One such algorithm using a pointer-based quadtree is reported in [12]. In [15], a general framework for the computation of geometric properties (including connected component labeling) of two-dimensional images represented by linear quadtrees is discussed. The basic idea is that at any instant (i.e., after processing m leaf nodes), the state of the traversal can be visualized as a staircase (termed an *active border*). In particular, given a traversal that visits sons of nonterminal nodes in the order SW, SE, NW, NE, the part of the image described by the m leaf nodes is to the left and below the staircase. For example, for Fig. 1, a traversal in the above order (without GRAY nodes) is 6, 7, 1, 4, 5, 2, 3. Fig. 2 shows the active border before and after processing node 4. A heavy line indicates that the adjacent block is BLACK, while a light line corresponds to an adjacent WHITE block. Assuming a 2^n by 2^n image, such a framework requires 2 arrays of 2^n records with three fields apiece to represent the active border.

The techniques described above are good for two-dimensional data, but for data of three and higher dimensions (e.g., octree), direct extensions are not very practical due to large storage requirements. For example, for a 2^n by 2^n by 2^n object, they require three arrays of 4^n records with three fields apiece for the active border. In order to obtain efficient multidimensional algorithms, in this paper we develop a new representation of the active border. It takes advantage of the fact that much of the storage is unnecessary and represents the active border with linked lists instead of arrays. As can be seen from the algorithm, this is a nontrivial modification. The result is a connected component labeling algorithm for data of arbitrary dimensions that is very efficient from both the standpoints of space and time. A three-dimensional octree implementation of the algorithm was used to facilitate the conversion from a boundary representation to an octree [18]. In that application, the objects were not allowed to have holes. However, the connected component labeling algorithm works in the presence of holes as well.

II. Connected Component Labeling Using Bintrees

The region quadtree is based on the successive subdivision of a d-dimensional image into 2^d equal-size quadrants until homogeneous blocks (i.e., *BLACK or WHITE*) are encountered. A binary image tree (termed *bintree*) [17] is defined analogously, except that at each stage, we subdivide the image into two parts. Again, as in the case of the linear quadtree, when $d = 2$, at odd stages we partition along the x coordinate and at even stages on the y coordinate. In d dimensions, the coordinate axes are similarly cyclically chosen for partitioning. Fig. 3 is the bintree corresponding to the image of Fig. 1(a). We assume that for the $x(y)$ partition, the left subtree corresponds to the west (south) half of the image and the right subtree corresponds to the east (north) half.



(a) (b)

Fig. 3. The bintree corresponding to Fig. 1. (a) Block decomposition. (b) Bintree representation of the blocks in (a).

In our algorithms, we use a linear image representation in the form of a preorder traversal of its bintree (i.e., a DF-expression [6]). This representation is also applicable to the bintree. The traversal yields a string, termed a *linear bintree*, over the alphabet "(," "*B*," and "*W*" corresponding to the GRAY (i.e., nonleaf), BLACK, and WHITE nodes, respectively. For example, the DF-expression for the bintree corresponding to the two-dimensional image of Fig. 1(a) is (*B*(*W*(*WB*. Modifying our algorithms to deal with other linear bintree representations is simple.

We label the connected components of an image represented by a bintree by using the breadth-first approach. It requires us to inspect all BLACK border elements. This is done by traversing the linear bintree. Again, as in the case of the linear quadtree, when $d = 2$, at any instant (i.e., after processing m leaf nodes), the state of the traversal can be visualized as a staircase (termed an *active border*) consisting of *active border elements*. This set can be further decomposed into d sets of active border elements. For example, when $d = 2$, a traversal of the bintree such that the western and southern halves are traversed before the eastern and northern halves, respectively, the m processed leaf nodes describe a portion of the image that is to the left and below the staircase. A traversal in this order (without GRAY nodes) for Fig. 3 is 1, 2, 3, 4. Each active border element must be given the color of the adjacent d-dimensional cube that has already been processed. As each leaf in the list of nodes in the linear bintree is processed, its d border elements adjacent to the active border (the S and W border elements in the two-dimensional case) are examined and UNION-FIND is applied to any resulting adjacencies between BLACK elements. In addition, the active border is updated to reflect the new active border elements (i.e., the other d unprocessed border elements of the node—the N and E border elements in the two-dimensional case). Fig. 4 shows the active border before and after processing block 3 of Fig. 3. A heavy line indicates that the adjacent block is BLACK, while a light line corresponds to an adjacent WHITE block.

We represent the active border as a singly linked list of records of type *borderlist*, with fields DATA and NEXT, which contains pointers to records corresponding to the active border elements comprising it. For example, in Fig. 4, the active-x border is the list (X_1, X_2, X_3) and the active-y border is the list (Y_1, Y_2). Each active border element is represented as a record of type *borderelement* having three fields, SIZ, COL, and LAB, corresponding, respectively, to the size (length in two dimensions, area in three dimensions, etc.), color, and label (i.e., equivalence class) of the side of the block adjacent to the already processed border element. Initially, there are d active border elements of all size $2^{n\cdot(d-1)}$ and color WHITE.

Connected component labeling is performed by procedures COMPONENTS, TRAVERSE, and INCREMENT. They are given in the Appendix using a variant of Algol. Procedure COMPONENTS is invoked with the DF expression encoding of the bintree. First, it initializes the active border. Next, it invokes procedure TRAVERSE which controls the traversal of the bintree nodes. During this process, each BLACK leaf node is assigned a label (i.e., equivalence class) and a copy is made of the DF-expression (in

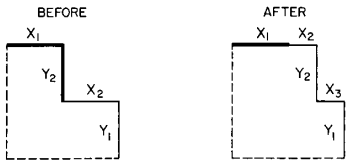


Fig. 4. The active border before and after processing block 3 of Fig. 3.

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 is applied as part of FIND to make sure that after FIND, no equivareverse order). Finally, it applies procedure PHASEII to adjust the component labels to their final values. PHASEII just reverses the DF-expression while applying FIND once for each BLACK image element. For each leaf node, TRAVERSE calls procedure INCREMENT d times—i.e., once for each of the d borders to perform the actual updating of the active borders and to propagate equivalences among labels. If a BLACK leaf node is not identified with any existing image component, then a new label (i.e., equivalence class) is generated.

Procedure TRAVERSE is the key to the algorithm. Using list pointers ACTIVE_BORDER[0], ACTIVE_BORDER[1], $\cdots$, ACTIVE_BORDER[$d-1$], it keeps track of the heads of the lists of the elements of the d active borders. For example, assume that $d = 2$ and let YL and XL correspond to ACTIVE_BORDER[0] and ACTIVE_BORDER[1], the active-y and active-x borders, respectively, starting at the node currently being processed. We use Fig. 5, the state of the active borders (XL and YL) before and after each call to TRAVERSE, to illustrate our discussion.

First, in the case of a nonleaf node, its block is split in two and the procedure is applied recursively to the two halves. Parameter CURRENT_COORD indicates the direction along which the block should be partitioned. CURRENT-COORD cycles through all the directions. VOLUME is the DIMENSION-dimensional area of the block and WIDTH is the width of the block along directions CURRENT_COORD through DIMENSION-1; its width along the remaining directions is WIDTH/2. After finishing one half, say the first half of a partition on the x(y) coordinate, the pointer to the active-y (active-x) (note the change in the order of x and y) border is reset to point at the element of the active-y (active-x) border it pointed at just prior to the partition (although its actual value may have changed during processing). It must be reset because we have swept through its entire range, while this is not true for the remaining active borders. For example, after processing node 1 and before processing node B, the value of YL is reset, but the active-y border entry is now different from what it was when starting with node 1. The pointer to the active-x border retains the value it had at the end of the first half (e.g., XL after processing node 1 and before processing node B).

Second, in the case of a leaf node, procedure INCREMENT is invoked DIMENSION times to process all active border elements bordering on the edge of the new leaf. There are three possible cases, depending on whether the entering edge is smaller than its corresponding active border entry, equal in size, or larger. Fig. 6 illustrates the effect of each of these cases on the active border when $d = 2$. Note that the list comprising the active border will grow in size, stay the same, or shrink, respectively, for the three cases. If the new leaf is BLACK, then the connected component information is also updated if the corresponding active border entry or part thereof is BLACK. Once procedure INCREMENT has been applied to all the active borders, TRAVERSE determines if the leaf that has just been processed was BLACK and was only adjacent to WHITE nodes. If this was the case, then a new label (i.e., equivalence class) is generated. Just prior to termination, TRAVERSE advances the list pointers to elements of the active borders so that

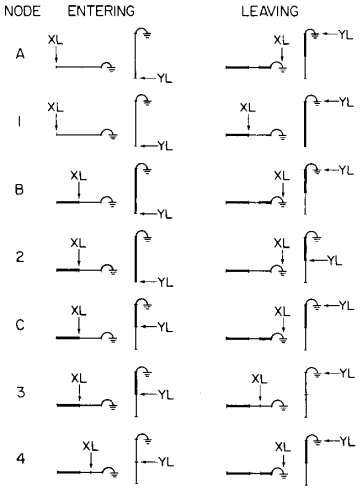


Fig. 5. State of the active borders (XL) and (YL) before and after each call to TRAVERSE for $d = 2$.

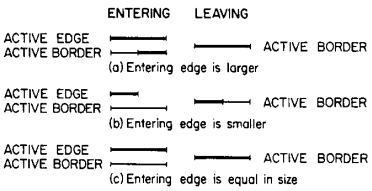


Fig. 6. The state of the active border in procedure INCREMENT. The comparisons are between the entering edge and its corresponding active border entry.

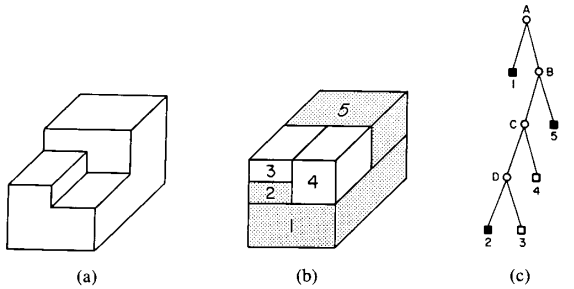


Fig. 7. An object, its maximal blocks, and the corresponding bintree. Blocks in the object are shaded. (a) Object. (b) Block decomposition of the object in (a). (c) Bintree representation of the blocks in (b).

they point to the edges corresponding to the block associated with the node to be processed next.

The equivalence classes are represented by trees. The root of each tree is the representative element of the class. The links between the remaining nodes in the tree reflect equivalences. Each node in the tree corresponds to a record of type *eq_class* with one field called FATHER. The nodes in the tree are pointed at by the LAB field of the active border elements and the leaf nodes.

As another example of the use of our algorithm, consider a three-dimensional image (i.e., $d = 3$) in the form of a bintree (e.g., Fig. 7). Assume that the first partition is in the z direction, followed by y and z, alternating among the three directions thereafter. Using the coordinate system of Fig. 8, we say that the left subtree corresponds to the negative x, y, and z directions relative to the origin

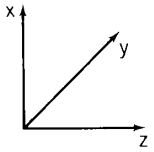


Fig. 8. Bintree coordinate system.

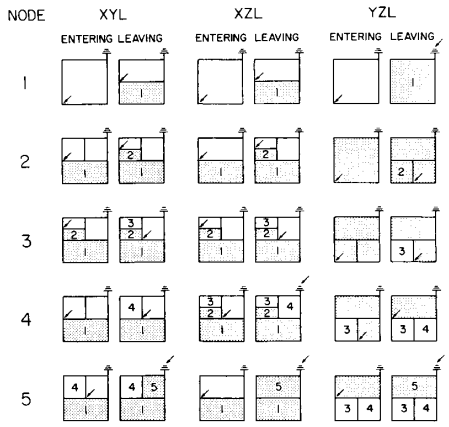


Fig. 9. The state of the active borders (XYL, XZL, and YZL) at selected calls to TRAVERSE for $d = 3$. The actual arrows designate the value of XYL, XZL, and YZL as is appropriate. BLACK blocks in the image are shaded.

and the right subtree to positive values. The linear bintree for the image of Fig. 7 has (B(((BWWB as its DF-expression.

The algorithm for labeling connected components of a three-dimensional bintree is almost identical to that presented for the two-dimensional case. The main difference is that we now have an active-*xy*, active-*xz*, and active-*yz* border consisting of active border elements which correspond to faces instead of edges. As an example of the application of the algorithm, let YZL, XZL, and XYL correspond to ACTIVE_BORDER[0], ACTIVE_BORDER[1], and ACTIVE_BORDER[2], the active-*yz*, active-*xz*, and active-*xy* borders, respectively. Fig. 9 shows the state of the active-*xy*, active-*xz*, and active-*yz* borders before and after each call to TRAVERSE corresponding to the leaf nodes in Fig. 7.

III. Analysis

A. Theoretical

A theoretical analysis of the worst case running time of the algorithm finds that it is $\Omega(B + (2E + B) \cdot \alpha(B, 2B + 2E))$ where B is the number of BLACK image elements and E is the number of adjacencies between them. α is the inverse of the Ackermann function and grows very slowly. These bounds are a direct result of using the UNION-FIND method to process equivalences. See [20] for more details on the exact formulation. In most practical cases, $\alpha(V, 2V + 2E) < 3$, and thus the algorithm is "almost" linear. Its worst-case storage requirements are $O(B)$.

B. Empirical

Ignoring the contribution of UNION-FIND (given in Section III-A), the time to traverse a d-dimensional linear bintree is directly proportional to the number of leaf nodes. This is better than results obtained when using neighbor finding methods [12] and is comparable to techniques which transmit neighbors as parameters to the traversal algorithm [5], [14]. The space requirements of the algorithms reported here are generally better than of those used in [15] by virtue of using linked lists instead of arrays. However, the worst cases are still the same (i.e., a checkerboard image).

In order to gain some more insight into the performance of our algorithms we conducted experiments. The first series of tests helps to understand the behavior of the algorithm for "simple" (i.e., highly aggregated) images. The test images consisted of two-dimensional and three-dimensional digital balls of varying diameters (i.e., M). Each ball was generated so that each pixel (voxel) intersecting the boundary is BLACK, as are the elements totally contained in the ball. Programs were written in the C programming language and were executed on a VAX 11/750. Three experiments were performed with balls having a diameter, say M, which was a power of two and the ball was embedded in a d-dimensional cube of side length M.

1) The connected components of a two-dimensional ball (i.e., disk) with $M = 512$, 1024, 2048, and 4096.

2) The perimeter of a two-dimensional ball with diameters identical to those used in 1). This helps determine the effect of the UNION-FIND algorithm because perimeter computation is closely related to connected component labeling when the step of merging equivalence classes is omitted.

3) The connected components of a three-dimensional ball with $M = 32$, 64, and 128.

Table I shows the results of the first two experiments. "Leaf nodes" denotes the number of BLACK and WHITE leaf nodes in the bintree. "Labels" corresponds to the number of different equivalence classes that were generated (i.e., the number of nodes that could not be directly labeled from their neighbors that had already been processed). "Max edges" indicates the maximum number of records of type *borderelement* that were required in the active border lists. It is useful for determining the space requirements and comparing to the case when the active border is represented as an array [15]. It is interesting to note that for $M = 4096$, the method of [15] required 8192 edge records, whereas the method described here only required a maximum of 382 edge records. Execution times are given in CPU seconds. The programs implemented were not optimized; subroutine calls alone accounted for about 50 percent of the processing time. We see that for this class of images, the space requirements are small. Table II shows the results of the third experiment. Column headings are analogous to those used in Table I, except that in three dimensions, perimeter means surface area.

The data of Tables I and II are revealing from several standpoints. First, they illustrate Hunter's theorem [3] for two-dimensional images that as the resolution is doubled, the quadtree grows linearly in the number of nodes. Furthermore, for three-dimensional images, the number of nodes in the octree grows with the second power (i.e., proportionally to the boundary of the image [9]). In our experiments, these results also hold for the number of labels (i.e., equivalence classes) that are generated. However, no such easy conclusions can be drawn about the maximum number of edges (faces) whose number seems to grow more slowly. Nevertheless, it is interesting to note that in Table II, the maximum number of faces for $M = 128$ was 2514, whereas in actuality, were we to apply an array method similar to [15], then 49,152 array entries (border elements) would be required to represent the active border.

Second, our data show that the execution time of our connected component labeling algorithm is approximately linearly related to the number of leaf nodes in the tree. By measuring separately the amount of time necessary for computing the perimeter, we saw that the contribution of the equivalence class merging task (i.e., UNION-FIND) to connected component labeling is relatively small. We also measured the contribution of PHASEII and found it to be approximately 10 percent of the time. Of course, program optimizations will increase the relative contribution of the UNION-FIND step.

Third, from the limited data that we have, it seems that for a given image size (i.e., in terms of leaf nodes), the three-dimensional case requires approximately 30 percent more time than the two-dimensional case. This is not surprising since the active border now has three lists instead of two and INCREMENT must process each of these lists.

TABLE I
TWO-DIMENSIONAL CONNECTED COMPONENT LABELING (CCL) AND PERIMETER

M	Leaf Nodes	BLACK Nodes	Labels	Max Edges	CCL (seconds)	Perimeter (seconds)
512	2840	1524	86	157	1.8	1.6
1024	5840	3028	178	211	3.4	3.1
2048	11 740	5916	343	287	6.8	5.9
4096	23 680	11 928	687	382	13.7	12.2

TABLE II
PERFORMANCE OF THREE-DIMENSIONAL CONNECTED COMPONENT LABELING (CCL)

M	Leaf nodes	BLACK Nodes	Labels	Max Faces	CCL (seconds)	Perimeter (seconds)
32	3296	1656	50	571	2.9	2.7
64	13 960	6896	189	1187	11.3	10.6
128	58 712	29 676	755	2514	46.8	43.1

TABLE III
COMPARISON OF THREE-DIMENSIONAL CONNECTED COMPONENT LABELING METHODS

NBANDS	Voxels	Image Size Leaf Nodes	BLACK Nodes	Components	CPU Time (seconds) Method of [8] (VAX 11/780)	CPU Time (seconds) New (VAX 11/750)	Storage Requirements (New Method) Labels	Storage Requirements (New Method) Maximum Faces
2	20 000	10 806	4837	214	72.8	7.5	653	325
4	40 000	20 114	9903	312	145.0	14.5	912	553
8	80 000	39 329	19 577	410	290.7	28.5	1321	1065
16	160 000	78 988	39 383	704	582.3	56.1	2248	2331
32	320 000	157 479	78 537	1194	1147.5	111.7	3993	4873

In order to get a more complicated benchmark, we used images similar to those used in a study performed by Lumia [8] of a different connected component labeling algorithm for three-dimensional images that are represented by arrays. Using these tests to compare the two algorithms is not fair as they are different. In particular, Lumia's method does not use the UNION-FIND algorithm. Also, Lumia's method makes use of a matrix-like representation, whereas we use a three-dimensional bintree. Nevertheless, the results are worth noting from a qualitative standpoint. The test images consist of a number, NBANDS, of two-dimensional 100×100 checkerboard sections as described in more detail in [8]. For the purpose of building a bintree, we embedded each image into a 128×128 space with the checkerboard sections as yz planes. In contrast to the previous series, the test images exhibited almost no aggregation—an average leaf node of the bintree contained only two voxels. Thus, these images are a "bad case" for our algorithm. Table III summarizes our results and also contains a column describing the performance of Lumia's algorithms (on a VAX 11/780). Table III verifies the conclusions of Table II in that processing time is approximately proportional to the number of leaf nodes and storage requirements are small. Processing times of the new algorithm compare very favorably to those reported by Lumia, even when the new method is applied without the two voxel/leaf compression that results from use of the bintree (in this case, processing times would be approximately doubled).

To help interpret Table III further, we note some more details. 1) PHASEII requires approximately 10 percent of the processing time as does image input. 2) We generated the bintree procedurally from Lumia's description. This required more than twice the CPU time necessary for connected component labeling.

IV. CONCLUDING REMARKS

We have presented a general algorithm for labeling connected components of d-dimensional images represented by linear bintrees. Our implementation was for a linear quadtree in the form of a DF-expression. However, the same algorithm can be used for a linear quadtree in the form of a set of location codes. We used a generalized data structure, termed a bintree, which can be easily adapted to any d-dimensional image. The algorithm is an improvement over a previous method described in [15] in that it has been formulated to treat images of arbitrary dimensionality. Also, it requires considerably less space, and hence is feasible for such images. Nevertheless, the approach described in [15] still has merit. It is a general method for computing geometric properties of two-dimensional images represented by linear quadtrees. These properties included perimeter and genus (i.e., Euler number) as well as connected component labeling. The perimeter could be easily computed within the framework presented here. However, it is not immediately clear how the d-dimensional genus could be computed using the methods presented here. The problem is that to compute the genus in two dimensions for quadtrees [1], we can make use of the method of Minsky and Papert [10] which, unfortunately, is not easily generalizable to images of higher dimensions. On the other hand, genus computation is not a very critical operation in image processing systems. For other work involving the application of hierarchical methods to images of arbitrary dimension see [21], [5].

It is difficult to compare our techniques to existing methods since the efficiency of our algorithms is directly proportional to the amount of aggregation that exists in the image. Our largest three-

dimensional image contained over 2 million voxels, yet its labeling required less than 47 s of CPU time (Table II), while an image containing 320 000 voxels required about 112 s of CPU time (Table III). The strength of our techniques lies in the ability to take advantage of homogeneity. The results reinforce our earlier observation that quadtree methods derive their significance from the saving in execution time that they yield, which is also directly proportional to the space saving.

Our method requires an equivalence table which may have as many elements as there are BLACK blocks in the entire image. However, this problem can be easily overcome by making use of the concept of active equivalence classes. This concept has been analyzed in great detail in [16]. In particular, we say that an equivalence class E is *active* as long as there is at least one image element, which refers to it, in the active border. Whenever no element of the active border refers to E, then we reuse E by associating its storage space with another component. Using such a method means that the number of different labels (i.e., equivalence classes) is bounded by the maximum number of active border elements. A general algorithm, applicable to array as well as hierarchical image representations and employing this concept, is given in [16], and its incorporation in the procedures of this paper is straightforward. In particular, the only required change is that with each equivalence class, a count is maintained of the number of active border elements that refer to it. The control structure of the modified algorithm remains the same.

Appendix.
Code for the Algorithm

```
procedure COMPONENTS(DIMENSION, WIDTH_OF_UNIVERSE, DF);
/* Label the connected components of a WIDTH_OF_UNIVERSE
   by WIDTH_OF_UNIVERSE by ··· by WIDTH_OF_UNIVERSE (WIDTH_OF_UNIVERSE = 2^n) DIMENSION-dimensional image represented by DF, a preorder traversal of its bintree. PHASEII_NODES points to the start of the list of nodes used in the second phase of the algorithm. Each node is represented by a record of type node having two fields, COL and LAB, corresponding to its color and the equivalence class which is assigned to it. */
begin
   global value integer DIMENSION;
   value integer WIDTH_OF_UNIVERSE;
   global value pointer dfnodelist DF;
   global pointer nodelist PHASEII_NODES;
   pointer borderlist array ACTIVE_BORDER
     [0:DIMENSION-1];
   integer J;
   /* Initialize each element of ACTIVE_BORDER to represent one active borderelement of size WIDTH_OF_UNIVERSE^DIMENSION-1 and adjacent to WHITE blocks in each of the DIMENSION directions: */
   for J ← 0 step 1 until DIMENSION-1 do
      begin
         ACTIVE_BORDER[J] ← create(borderlist):
         DATA(ACTIVE_BORDER[J]) ← create(border element);
         SIZ(DATA(ACTIVE_BORDER[J])) ← WIDTH_OF_UNIVERSE ↑ (DIMENSION-1);
         COL(DATA(ACTIVE_BORDER[J])) ← 'WHITE';
         LAB(DATA(ACTIVE_BORDER[J])) ← NIL;
      end;
   if not empty (DF) then
      begin
         PHASEII_NODES ← NIL;
         TRAVERSE(WIDTH_OF UNIVERSE ↑ DIMENSION,
                  ACTIVE_BORDER, 0,
                  WIDTH_OF_UNIVERSE);
         PHASEII(PHASEII_NODES); /* Set the final label of each leaf using FIND */
      end;
end;
procedure TRAVERSE(VOLUME, ACTIVE_BORDER, CURRENT_COORD, WIDTH);
/* Compute the contribution of a node whose corresponding DIMENSION-dimensional rectangular parallelepiped has volume VOLUME. CURRENT_COORD of its sides have width WIDTH/2 and the remaining DIMENSION-CURRENT_COORD sides are of width WIDTH. For each nonterminal node, CURRENT_COORD indicates the direction along which the corresponding block should be partitioned and TRAVERSE is recursively applied to the two halves. ACTIVE_BORDER contains pointers to the active borders in the DIMENSION directions. ACTIVE_BORDER[0], ACTIVE_BORDER[1], ··· , and ACTIVE_BORDER[DIMENSION-1] point to the part of the active border that is adjacent to the parallelepiped currently being processed. */
begin
   value real VOLUME, WIDTH;
   reference pointer borderlist array ACTIVE_BORDER
   [0:DIMENSION-1];
   value integer CURRENT_COORD;
   global integer DIMENSION;
   global pointer dfnodelist DF;
   global pointer nodelist PHASEII_NODES;
   pointer borderlist TEMP;
   pointer node CURRENT_NODE;
   integer J;
   CURRENT_NODE ← create (node);
   COL(CURRENT_NODE) ← NEXT(DF);
      /* Get the next element in the preorder traversal */
   LAB(CURRENT_NODE) ← NIL;
   if COL(CURRENT_NODE) = 'GRAY' then
      begin /* Nonleaf node */
         /* Add CURRENT_NODE to the front of PHASEII_NODES so that the second phase can update the labels to their final equivalence classes. */
         addtolist (PHASEII_NODES, CURRENT_NODE);
         TEMP ← ACTIVE_BORDER[CURRENT_COORD];
          /* Save pointer to start of ACTIVE_BORDER [CURRENT_COORD] */
         if (CURRENT_COORD + 1) mod DIMENSION = 0
         then WIDTH ← WIDTH/2;
         VOLUME ← VOLUME/2;
         TRAVERSE(VOLUME, ACTIVE_BORDER,
                  (CURRENT_COORD + 1) mod DIMENSION, WIDTH);
                    /* Partition on CURRENT_COORD */
         ACTIVE_BORDER[CURRENT_COORD] ← TEMP;
         TRAVERSE(VOLUME, ACTIVE_BORDER,
                  (CURRENT_COORD + 1) mod DIMENSION, WIDTH);

      end
   else
      begin /* Leaf node */
         /* Compute each border element's contribution to the active border. In computing the "size" parameter we must distinguish between active borders 0 ··· CURRENT_COORD_1 and CURRENT_COORD ··· DIMENSION-1. */
         for J ← 0 step 1 until DIMENSION-1 do
            INCREMENT(CURRENT_NODE, ACTIVE_BORDER[J],
            if J ≥ CURRENT_COORD then VOLUME/WIDTH
            else 2*VOLUME/WIDTH;
         if COL(CURRENT_NODE) = 'BLACK' then
```

```
            begin
              /* Assign CURRENT_NODE's equivalence class
              to its corresponding active border elements */
              if null (LAB(CURRENT_NODE)) then /* New
              equivalence class */
                begin
                  LAB(CURRENT_NODE) ← create
                  (eq_class);
                  FATHER(LAB(CURRENT_NODE)) ← NIL;
                end;
              for J ← 0 step 1 until DIMENSION-1 do
                LAB(DATA(ACTIVE_BORDER[J])) ← LAB
                (CURRENT_NODE);
            end;
          /* Advance the pointer to the start of the appropriate ac-
          tive border: */
          for J ← 0 step 1 until DIMENSION-1 do
            ACTIVE_BORDER[J] ← NEXT
            (ACTIVE_BORDER[J]);
          /* Add CURRENT_NODE to the front of PHASE-
            II_NODES so that the second phase can update the
            labels to their final equivalence classes. */
          addtolist(PHASEII_NODES, CURRENT_NODE);
        end;
end;
procedure INCREMENT(LEAF, ACTIVE_BORDER, SIZE);
/* Update the active border for a side of leaf node LEAF having
   size SIZE. SIZE corresponds to width in the two-dimensional
   case and to area in the three-dimensional case. ACTIVE_BOR-
   DER is a pointer to a list of border elements constituting the
   active border in the present direction. LEAF is adjacent to the
   first border element in ACTIVE_BORDER. */
begin
  value pointer node LEAF;
  value pointer borderlist ACTIVE_BORDER;
  value integer SIZE;
  pointer borderlist NEIGHBOR, Q; /* Auxiliary variables */
  integer I;
  if SIZE > SIZ(DATA(ACTIVE_BORDER)) then
    begin /* Neighbor is a nonleaf node—case (a) of Fig. 6 */
      I ← 0;
      NEIGHBOR ← ACTIVE_BORDER;
      while I NEQ SIZE do
        begin
          /* Update the active border for all border elements
            that are adjacent to the side of LEAF that is being
            processed. */
          if COL(LEAF) = 'BLACK' and COL
          (DATA(NEIGHBOR)) = 'BLACK' then
            LAB(LEAF) ← UNION(LAB(LEAF), FIND
            (LAB(DATA(NEIGHBOR))));
          I ← I + SIZ(DATA(NEIGHBOR));
          NEIGHBOR ← NEXT(NEIGHBOR);
        end;
      Q ← NEXT(ACTIVE_BORDER);
      NEXT(ACTIVE_BORDER) ← NEIGHBOR;
      borderlist_dispose (Q, NEIGHBOR);
      /* Reclaim storage for active border elements starting
        at Q up to but not including NEIGHBOR */
  end
    else /* Neighbor is a leaf—cases (b) and (c) of Fig. 6
    */
      begin
        if COL(LEAF) = 'BLACK' and COL
        (DATA(ACTIVE_BORDER)) = 'BLACK' then
          LAB(LEAF) ← UNION(LAB(LEAF), FIND
          (LAB(DATA(ACTIVE_BORDER))));
        if SIZE < SIZ(DATA(ACTIVE_BORDER)) then
          /* Neighbor is larger—case (b) of Fig. 6 */
          begin /* Update the active border */
            NEIGHBOR ← create(borderlist; /* Add a new
            border element */
            DATA(NEIGHBOR) ← create(borderelement);
            /* Compute unprocessed portion of the active bor-
              der: */
            SIZ(DATA(NEIGHBOR)) ← SIZ
            (DATA(ACTIVE_BORDER))-SIZE;
            COL(DATA(NEIGHBOR)) ← COL
            (DATA(ACTIVE_BORDER));
            LAB(DATA(NEIGHBOR)) ← LAB
            (DATA(ACTIVE_BORDER));
            NEXT(NEIGHBOR) ← NEXT (ACTIVE_BOR-
            DER);
            /* Update head of active border list: */
            NEXT(ACTIVE_BORDER) ← NEIGHBOR;
          end;
          end;
    /* Update the active border to reflect the new leaf */
    SIZ(DATA(ACTIVE_BORDER)) ← SIZE;
    COL(DATA(ACTIVE_BORDER)) ← COL(LEAF);
end;

pointer nodelist procedure PHASEII(OLD_NODE_LIST);
/* Update the equivalence classes of all elements of list NODES
   that are leaf nodes (i.e., their LAB field) to the correct equiv-
   alence class by using FIND. During this process, list NODES
   will be reversed and the original order of the DF expression will
   be restored. nodelist has two fields, DATA and NEXT. */
begin
  value pointer nodelist OLD_NODE_LIST;
  pointer nodelist NEW_NODE_LIST
  NEW_NODE_LIST ← NIL;
  while not null(OLD_NODE_LIST) do
    begin
      if COL(DATA(OLD_NODE_LIST)) NEQ 'GRAY' then
        LAB(DATA(OLD_NODE_LIST)) ← FIND
        (LAB(DATA(OLD_NODE_LIST)));
      addtolist(NEW_NODE_LIST, DATA (OLD_NODE_
      LIST));
      OLD_NODE_LIST ← NEXT(OLD_NODE_LIST);
    end;
  return(NEW_NODE_LIST);
end;

pointer eq_class procedure UNION(LABEL1, LABEL2);
/* Merge equivalence class LABEL1 with equivalence class
   LABEL2. If LABEL1 is not NIL and if LABEL1 is not equal
   to LABEL2, then set the FATHER field of LABEL1 to
   LABEL2. */
begin
  value pointer eq_class LABEL1, LABEL2;
  if not null(LABEL1) and LABEL1 NEQ LABEL2 then
    FATHER(LABEL1) ← LABEL2;
  return(LABEL2);
end;

pointer eq_class procedure FIND(LABEL1);
/* Determine the equivalence class containing equivalence class
   LABEL1. Perform path compression at the same time—i.e.,
   when an equivalence 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 LABEL1;
  pointer eq_class R, TEMP;
  if null(FATHER(LABEL1)) then return(LABEL1)
  else
    begin
      R ← LABEL1;
      do R ← FATHER (R) until null (FATHER (R));
```

```
        do
          begin /* Short circuit the path by linking LABEL1 to
          R */
            TEMP ← FATHER(LABEL1);
            FATHER(LABEL1) ← R;
            LABEL1 ← TEMP;
          end
          until null(FATHER(LABEL1));
        return (R);
      end;
end;
```

ACKNOWLEDGMENT

We thank M. Mantyla, A. Rosenfeld, and R. E. Webber for their comments.

REFERENCES

[1] C. R. Dyer, "Computing the Euler number of an image from its quadtree," *Comput. Graphics Image Processing*, vol. 13, pp. 270–276, July 1980.
[2] I. Gargantini, "An effective way to represent quadtrees," *Commun. ACM*, vol. 25, pp. 905–910, Dec. 1982.
[3] G. M. Hunter, "Efficient computation and data structures for graphics," Ph.D. dissertation, Dep. Elec. Eng. Comput. Sci., Princeton Univ., Princeton, NJ, 1978.
[4] C. L. Jackins, and S. L. Tanimoto, "Oct-trees and their use in representing three-dimensional objects," *Comput. Graphics Image Processing*, vol. 14, pp. 249–270, Nov. 1980.
[5] ——, "Quad-trees, oct-trees, and *k*-trees—A generalized approach to recursive decomposition of Euclidean space," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-5, pp. 533–539, Sept. 1983.
[6] E. Kawaguchi and T. Endo, "On the method of binary picture representation and its application to data compression," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-2, pp. 27–35, Jan. 1980.
[7] A. Klinger, "Patterns and search statistics," in *Optimizing Methods in Statistics*, J. S. Rustagi, Ed. New York: Academic, 1971, pp. 303–337.
[8] R. Lumina, "A new three-dimensional connected components algorithm," *Comput. Vision, Graphics, Image Processing*, vol. 23, pp. 207–217, Aug. 1983.
[9] D. Meagher, "Geometric modeling using octree encoding," *Comput. Graphics Image Processing*, vol. 19, pp. 129–147, June 1982.
[10] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA: M.I.T. Press, 1969.
[11] A. Rosenfeld and J. L. Pfaltz, "Sequential operations in digital image processing," *J. ACM*, vol. 13, pp. 471–494, Oct. 1966.
[12] H. Samet, "Connected component labeling using quadtrees," *J. ACM*, vol. 28, pp. 487–501, July 1981.
[13] ——, "The quadtree and related hierarchical data structures," *ACM Comput. Surveys*, vol. 16, pp. 187–260, June 1984.
[14] H. Samet, "A top-down quadtree traversal algorithm," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-7, Jan. 1985; also, Univ. Maryland, College Park, Comput. Sci. TR-1237.
[15] H. Samet and M. Tamminen, "Computing geometric properties of images represented by linear quadtrees," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-7, Mar. 1985.
[16] H. Samet and M. Tamminen, "An improved approach to connected component labeling of images," in *Proc. CVPR86 Conf.*, Miami Beach, FL, June 1986, pp. 312–318.
[17] M. Tamminen, "Component on quad- and octtrees," *Commun. ACM*, vol. 27, pp. 248–249, Mar. 1984.
[18] M. Tamminen and H. Samet, "Effective octree conversion by connectivity labeling," in *Proc. SIGGRAPH '84 Conf.*, Minneapolis, MN, July 1984, pp. 43–51.
[19] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *J. ACM*, vol. 22, pp. 215–225, Apr. 1975.
[20] R. E. Tarjan and J. van Leeuwen, "Worst-case analysis of set union algorithms," *J. ACM*, vol. 31, pp. 245–281, Apr. 1984.
[21] M. Yau and S. N. Srihari, "A hierarchical data structure for multidimensional digital images," *Commun. ACM*, vol. 26, pp. 504–515, July 1983.

A Multilevel Parallel Processing Approach to Scene Labeling Problems

HSI-HO LIU, TZAY Y. YOUNG, AND AMITAVA DAS

***Abstract*—A parallel tree search procedure and multilevel array architectures are presented for scene labeling problems. In the scene labeling problem, when the number of variables is not large, e.g., less than 100, its solutions and search space are not expected to increase very fast with problem size. The multilevel arrays only use a polynomial number of processors at each level and a bus or mesh connection for interlevel communication. This approach is very efficient compared to the binary-tree machine if the number of nodes in the search tree of labeling problem increases polynomially with the number of variables. Because of its regular and simple structure, the multilevel array is particularly suitable for VLSI implementation.**

***Index Terms*—Multilevel array, parallel processing, scene labeling, tree search, VLSI.**

I. INTRODUCTION

The labeling problem plays an important role in scene labeling [2], [12] and matching [1] of the computer vision, and many other problems such as graph coloring and graph homomorphisms [2]. In this correspondence, we present a parallel processing approach for the scene labeling problem. The number of applicable labels in the scene labeling problem is usually a small constant, and each variable is also constrained by a small number of variables. From our experience in several medium-size problems (e.g., problems with 100 variables), the number of solutions and search effort did not increase very fast.

The labeling algorithms can generally be divided into three groups [10]: basic tree search such as backtracking, filtering [14], or discrete relaxation [12], and hybrid approach [3]. Backtracking algorithms are guaranteed to find any solutions, but are extremely inefficient. Filtering can reduce the number of applicable labelings for each variable, yet a tree search is still needed to find out which labeling is compatible with which, and that can be very time consuming. Some hybrid algorithms such as forward checking have been shown to be very efficient [3]. However, it needs to pass a lot of information from node to node along every tree path, which makes it not a good candidate for parallel, particularly VLSI, implementation. The approach we adopt here applys filtering first and then uses the multilevel parallel tree search to find the solutions.

II. THE CONSISTENT LABELING PROBLEM AND PARALLEL PROCESSING

A. Problem Definition

Let $A = \{a_1, \cdots, a_n\}$ be a set of units or variables to be labeled, $L = \{l_1, \cdots, l_m\}$ a set of labels, T a set of variable constraint relations, and R a set of label compatibility relations. The labeling problem is to find all consistent labelings where a labeling of variables $(a_1, \cdots, a_n)$ is an n tuple $(l_{j_1}, \cdots, l_{j_n})$, $l_{j_i} \in L_i$, $i = 1, \cdots, n$, and a consistent labeling is a labeling which satisfies all compatibility relations. Let $L_i \subseteq L$ be the set of labels applicable to variable a_i, and let d_i be the number of labels in L_i, $i = 1, \cdots, n$. The variable constraint relation T can be represented

Manuscript received December 23, 1986; revised June 23, 1987. This work was supported in part by the National Science Foundation under Grant DCR 85-09737.

H.-H. Liu was with the Department of Electrical and Computer Engineering, University of Miami, Coral Gables, FL 33124. He is now with the Department of Electrical Engineering, Vanderbilt University, Nashville, TN 37235.

T. Y. Young and A. Das are with the Department of Electrical and Computer Engineering, University of Miami, Coral Gables, FL 33124.

IEEE Log Number 8718597.