CVGIP: IMAGE UNDERSTANDING
Vol. 54, No. 2, September, pp. 215–223, 1991

Linear Time Distance Transforms for Quadtrees

CLIFFORD A. SHAFFER

Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061-0106

AND

QUENTIN F. STOUT

Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Michigan 48109-2122

Received August 28, 1989; accepted May 16, 1990

Linear time algorithms are given for computing the chessboard distance transform for both pointer-based and linear quadtree representations. Comparisons between algorithmic styles for the two representations are made. Both versions of the algorithm consist of a pair of tree traversals. © 1991 Academic Press, Inc.

1. INTRODUCTION

The (region) quadtree, shown in Fig. 1, is a widely studied data structure for representing digitized images. An extensive survey of quadtrees and their use in image processing and graphics appears in Samet [1, 2]. Unfortunately, the two-dimensional nature of the information they store makes quadtree usage more subtle than, say, binary search tree usage, and efficient algorithms often demand special quadtree techniques. This problem is intensified by the fact that there are at least three distinct representations for the quadtree structure reported in the literature. Each has unique advantages and disadvantages, with the result that each representation has applications for which it is most suited. The *pointer-based* quadtree representation maintains the explicit tree structure as illustrated by the tree of Fig. 1b. The *linear* quadtree [3] replaces the tree structure with a sorted linear list containing only the leaf nodes from the original tree.[1] The sort key is obtained by assigning to each leaf node an address derived by interleaving the bits of the x and y coordinates of the upper left pixel for the corresponding block in the image. The resulting records appear in the list in the same order as they would be visited by a depth-first traversal of the pointer-based quadtree. The *DF-expression* [4] is a third quadtree representation obtained by listing only the values of the nodes (both internal and leaf) in order as they occur when performing a preorder traversal of the tree structure.

The pointer-based and linear quadtree representations are the two most often appearing in the literature as base representations for describing algorithms. The flavor of the resulting algorithms are often different, depending on which of the two representations is used. In the past, the pointer-based quadtree has typically been used for applications where the entire image can be maintained in main memory. The linear quadtree is most appropriate for applications where large images are maintained in disk files with portions brought into memory as needed. This is primarily due to the fact that good paging algorithms are now known for the linear representation (specifically, index the list with a B-tree), but no effective paging algorithms have been presented for the pointer-based representation.

Pointer-based algorithms are concerned with tree-oriented operations such as finding the father or son of a node, or *neighbor-finding* operations [5, 6]. In comparison, linear quadtree algorithms access the node list by means of list search and insertion operations. While algorithms for the two representations may appear quite different, algorithms yielding a particular run time complexity for one representation can usually be converted to an equally efficient algorithm in the other representation (one example of such a conversion appears in Shaffer and Samet [7]).

The purpose of this paper is twofold. First, linear time algorithms are provided to solve the problem of generating the quadtree chessboard distance transform of Samet [8]. By providing such an algorithm, a linear time solution to the computation of the Quadtree Medial Axis Transform (QMAT) of Samet [9] is also implied since Samet's

[1] The method of [3] explicitly stores only the black nodes of a binary image. For simplicity, this paper assumes that all nodes are stored in the linear list; however, being a traversal, our linear quadtree algorithm could reconstruct the white nodes during processing if they were not stored explicitly.

1049-9660/91 $3.00

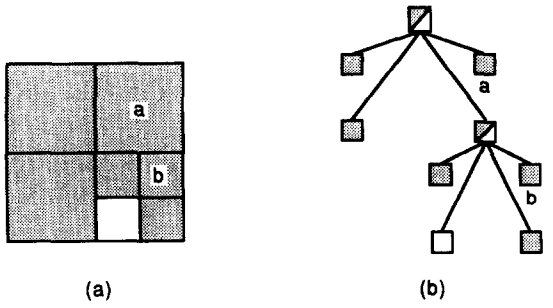

FIG. 1. A 4 × 4 image and its quadtree.

algorithm generates the QMAT from the distance transform in linear time. While our algorithms are similar to previous ones, they are the first with linear worst-case times. Second, by presenting algorithms both in terms of pointer-based and linear quadtree representations, the similarities and differences of the two approaches can be appreciated.

Section 2 provides further definitions for the data structures and terminology. Section 3 presents the distance transform algorithm in terms of the pointer-based quadtree representation. Section 4 presents the distance transform algorithm in terms of the linear quadtree representation. Section 5 presents our conclusions. Finally, PASCAL-like pseudo-code for both distance transform algorithms is provided in the Appendix.

2. DEFINITIONS AND FUNCTIONS

Given a $2^n \times 2^n$ image array of black or white pixels, its pointer-based quadtree representation is recursively constructed as follows: the root represents the entire image, and if the entire image is white or black then the root is a *white* or *black leaf*, respectively.[2] Otherwise, the root is a *gray (interior) node* with pointers to its four children (denoted NW, NE, SW, and SE) representing the four $2^{n-1} \times 2^{n-1}$ subimages in the quadrants. Two nodes are *adjacent* if and only if neither is a descendant of the other and if the image squares they represent share an edge or corner. In Fig. 1, b is adjacent to a and to the white pixel. The *depth* of the root is 0, and, recursively, the depth of a child of a node p is 1 more than the depth of p.

Given nodes p and q, neither of which is a descendant of the other, we say that p *is N* (equivalently *E*, *S*, *W*) of q if some pixel in p's image square is due north (south, east, west) of some pixel in q's image square, and that p *is NW (NE, SW, SE) of* q if p is not N nor W (not N nor E, not S nor W, not S nor E) of q and some pixel in p's image square is in the northwest (northwest, southwest, southeast) quadrant of some pixel in q's square. *Direction* will mean one of N, E, S, W, NW, NE, SW, or SE. Notice that if neither p nor q is a descendant of the other, then there is a direction C such that p is C of q and q is $-C$ of p, where $-C$ is the direction opposite of C. In Fig. 1, a is N of b, b is S of a, and the white pixel is also S of a.

Throughout we use the phrase *the C neighbor of p*, where p is a node and C is a direction. If p's C edge or corner is on the border of the image then there is no such neighbor. Otherwise, the neighbor is the node (possibly gray) of greatest depth less than or equal to p's which is adjacent to p in the indicated direction. In Fig. 1, a is the N neighbor of b, and b's parent is the S neighbor of a. *Neighbor* means a C neighbor for some direction C. Note that if q is a neighbor of p, then depth(q) ≤ depth(p) and q represents a square at least as large as the square p represents. It is important to keep in mind that neighbor is not a symmetric relation, in that p can be a neighbor of q while q is not a neighbor of p.

Most linear quadtree representations assume some sorted list implementation (such as a B-tree) is used that allows for efficient key search and dynamic record insertion and deletion. Our algorithm accesses the node list only by means of a visit to each node of the tree in order, combined with reconstruction of the identical node list in reverse order. Thus, the only list operations that need to be supported are some form of "next node" operation, and an operation that inserts a new node at the head of a list of output nodes.[3]

3. POINTER-BASED DISTANCE TRANSFORM ALGORITHM

The *Chessboard* or l_∞ distance between two points a and b can be defined as $\max(|a_x - b_x|, |a_y - b_y|)$. Samet [8] studied the problem of assigning to each black node the smallest distance to a white node, where the *distance from a black node b to a white node w*, denoted $d(b, w)$, is the l_∞ distance from the center point of the image square b represents to the nearest edge or corner of the image square w represents. The l_∞ *distance transform problem* (also called the chessboard distance transform problem) is to determine $dt(b) = \min\{d(b, w)$: w a white node$\}$ for each black node b. (If b is the root node, then $dt(b)$ is defined to be infinite.) Figure 2 shows the distances for the image in Fig. 1.

Our pointer-based quadtree algorithm is based on "top-down quadtree traversals" as described in Samet [10]. Top-down quadtree algorithms also appear in Jack-

[2] While we describe all concepts and algorithms in terms of binary images, they are easily extended to multi-color images as well.

[3] An alternative, though unusual, implementation for the linear quadtree suitable for use in this algorithm could be created with simple stack operations. Nodes would be POPed off an input list, processed, and PUSHed onto an output list, thus reversing the node order.

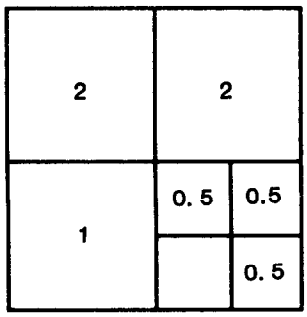

FIG. 2. Distance to nearest white node for the black nodes of Fig. 1.

ins and Tanimoto [11], Rosenfeld *et al.* [12] and Samet and Webber [13]. There are many such traversals, depending on the relative order in which nodes are to be visited. For example, one can visit all children after their parent (a postorder traversal), or before (a preorder traversal). For our purposes the parent/children ordering is immaterial since no work is ever performed at interior nodes, but the relative order in which different children are visited is important. In general, a stack is used to store a path to the root (either explicitly, or implicitly through recursion), where if an interior node is being visited then it causes its children to be visited, while if a leaf is being visited then some action is performed and the leaf is removed from the stack. "Top-down quadtree traversals" are distinguished from top-down tree traversals such as preorder traversal by the property that a call to a node p also passes pointers to p's eight or fewer neighbors. Note that if q is a child of p, then each of q's neighbors is either (1) one of p's neighbors, (2) one of p's children, (3) or a child of one of p's neighbors. Using this fact, q's neighbors can be determined in constant time if p's neighbors are known, and hence the overhead for an entire traversal can be performed in linear time. Since the distinguishing feature of "top-down quadtree traversals," compared to standard top-down tree traversals applied to quadtrees, is the fact that each node is accompanied by its neighborhood, in the rest of the paper they will be called *neighbored traversals*.

Throughout, N, B, and W will denote the total number of nodes, the number of black nodes, and the number of white nodes, respectively, in the quadtree. Time is always the worst-case time measured as a function of N. Since $B + W \leq N < (4/3) * (B + W)$, $N = \Theta(B + W)$.

The average time for the algorithm presented by Samet [8] is $\Theta(N)$. Samet did not analyze the worst-case time of his algorithm, but it is easy to show that it is $\Theta(N + B * H)$, where H is the height of the quadtree. Samet [10] states that the time of his algorithm "cannot be lowered by use of the top-down method since its computation time is not dominated by the cost of neighbor finding." While this is true for average time under his assumption concerning the expected distribution of black nodes, the top-down method can be used to reduce the worst case time complexity. The algorithm described in this section is similar to Samet's, but requires only linear time in the worst case.

For a black node b, radius(b) denotes the l_∞ radius of b's square, i.e., radius(b) is half of the length of a side of the square. It should be clear that $dt(b) \geq$ radius(b). Two properties about chessboard distances will be used later to prove the linear upper bound for our algorithm.

Property 1. If p and q are adjacent black nodes then $dt(p) \leq dt(q) +$ radius(p) + radius(q). To see why this is so, let q be a closest white node to q. The triangle inequality shows that $d(p, w) \leq d(q, w) +$ radius(p) + radius(q) (see Fig. 3), and since $dt(p) \leq d(p, w)$ and $dt(q) = d(q, w)$, this gives the result.

Property 2. If p is a black node other than the root, then $dt(p) < 3 *$ radius(p). This is because there must be a white node beneath p's parent, and any such white node is within $3 *$ radius(p) of p.

Both distance transform algorithms store with each black node p a field D (indicated as $p.D$ in PASCAL notation) which is initially ∞, and which equals $dt(p)$ at the end of the algorithm. The value of $p.D$ never increases, and at any time during the algorithm, if $p.D < \infty$ then there is a white node w such that $p.D = d(p, w)$.

The pointer-based algorithm consists of two neighbored traversals, named NW_to_SE and SE_to_NW, which can be performed in either order. These are just mirror images of each other, interchanging the roles of north and south, and of east and west, so only the NW_to_SE traversal will be explained. In this traversal, when a node p is visited, all of the nodes in the NW, N, and W directions have already been visited. This is insured by visiting the children of each node in the order NW, NE, SW, and SE. If p is white, then for each black neighbor q in the E, SW, S, and SE directions, $q.D$ is set equal to radius(q). If p is gray then its children are visited. Finally, if p is black, first $p.D$ is set equal to radius(p) if any of its neighbors in the NW, N, NE, or W

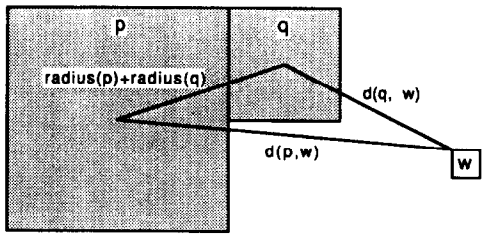

FIG. 3. The triangle inequality applied to chessboard distances.

directions are white. Then for each black neighbor q in the E, SW, S, and SE, directions, $q.D$ is set equal to the *minimum* of $q.D$ and $p.D$ + radius(p) + radius(q).

Algorithm 1, named POINTER_TRANSFORM, encodes the procedure described above.

THEOREM. *The l_∞ distance transform algorithm described above is correct and always finishes in time linear in the number of nodes of the quadtree.*

Proof. The time is linear because each neighbored traversal uses linear time. For any black node p, either p is the root and initialization sets $p.D = \infty$, or else there is some nearest white node w. This white node is in some direction C of p, and by Lemmas 1 and 2 below, at the end of the appropriate traversal $p.D = dt(p)$. Since $p.D$ is never less than $dt(p)$, the distances are all correctly determined. Q.E.D.

LEMMA 1. *For each black node p, if a nearest white node is N or W of p then $p.D = dt(p)$ at the end of the NW_to_SE traversal, and if it is S or E of p then $p.D = dt(p)$ at the end of the SE_to_NW traversal.*

Proof. Let w be a nearest white node to p, where w is C of p for $C \in \{N, E, S, W\}$. There is a pixel on the C border of p's image square such that some pixel of w's image square is C of the border pixel. Between w's image square and this border pixel there are only black pixels, since w is a nearest white. Therefore there is a sequence of black nodes $q_1, \ldots, q_k$ ($q_k = p$), where the C border of q_1's image square touches w's image square, q_{i+1} is a $-C$ neighbor of q_i for $1 \le i \le k - 1$, and

$$\begin{aligned} dt(q_{i+1}) &= d(q_{i+1}, w) \\ &= d(q_i, w) + \text{radius}(q_i) + \text{radius}(q_{i+1}) \\ &\quad \text{for } 1 \le i \le k - 1. \end{aligned}$$

(See Fig. 4a). Property 1 ensures that q_i is larger than q_{i-1} for $i > 1$. All of these claims are straightforward, except perhaps the fact that q_{i+1} is a $-C$ neighbor of q_i, since this requires that depth(q_{i+1}) $\le$ depth(q_i). To see that this is true, note that if it is false then $d(q_{i+1}, w) \ge$ radius(q_{i+1}) + 2*radius(q_i) $\ge$ 3*radius(q_{i+1}). Since $d(q_{i+1}) <$ 3*radius(q_{i+1}) is always true, this would imply that w is not the closest white to q_{i+1}, and hence not to p.

During the visit to q_1 in the appropriate traversal, $q_1.D$ is set equal to radius(q_1). This is because either w is a C neighbor of q_1 and the visit to q_1 sets $q_1.D$, or q_1 is a $-C$ neighbor of w and the visit to w (which preceded the visit to q_1) set $q_1.D$. During the remainder of the traversal, q_i is visited before q_{i+1}, and the visit to q_i results in $q_{i+1}.D$ being set equal to $d(q_{i+1}, w)$. When $i = k - 1$, this sets $p.D = d(p, w) = dt(p)$. Q.E.D.

LEMMA 2. *For each black node p, if a nearest white node is NW or NE of p, then $p.D = dt(p)$ at the end of the NW_to_SE traversal, while if it is SE or SW of p, then $p.D = dt(p)$ at the end of the SE_to_NW traversal.*

Proof. The proof is quite similar to that of the previous lemma. If a nearest white node w is, say, NE of p, then there are a sequence of black squares $q_1, \ldots, q_k = p$, and an integer j, $1 \le j \le k - 1$, such that q_1 is adjacent to w; each q_{i+1} is a SW neighbor of q_i for $j \le i \le k - 1$; either each q_{i+1} is a S neighbor of q_i for $1 \le i \le j - 1$, or each q_{i+1} is a W neighbor of q_i for $1 \le i \le j - 1$; and

$$\begin{aligned} dt(q_{i+1}) &= d(q_{i+1}, w) \\ &= d(q_i, w) + \text{radius}(q_i) + \text{radius}(q_{i+1}) \\ &\quad \text{for } 1 \le i \le k - 1. \end{aligned}$$

(See Fig. 4b.) Once again, during the visit to q_1 in the NW_to_SE traversal, $q_1.D$ = radius(q_1), and during the visit to q_i the value of $q_{i+1}.D$ is set to the correct value. Q.E.D.

It might appear that there is a problem in that when a leaf node that is a SE son of its parent is visited during the NW_to_SE traversal, its NE neighbor has not yet been visited and thus does not contain the correct distance transform value. However, once again, Property 1 ensures that q_i is larger than q_{i-1}. The result is that no node $q_i, j < i < k$ is a SE son. Likewise for NW sons during the SE_to_NW traversal.

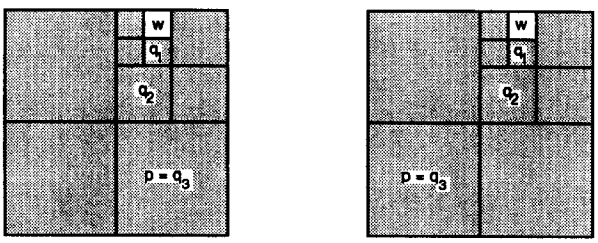

FIG. 4. Paths from node p to the nearest white node w.

4. LINEAR QUADTREE DISTANCE TRANSFORM ALGORITHM

This section presents a linear time two-pass chessboard distance transform algorithm for linear quadtrees. The two passes of this algorithm serve the same functions as the two traversals of the pointer implementation, visiting the leaf nodes in the same order, but accomplish their goals in a slightly different fashion. The first pass calculates the distance transform for each node M with respect to those nodes that precede M in the node list (i.e., M's

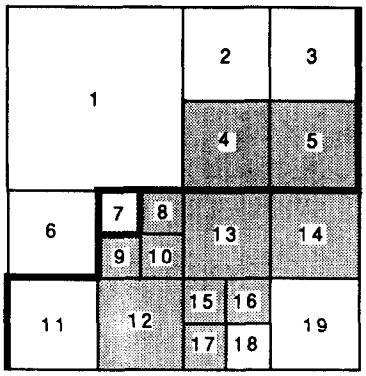

FIG. 5. The active border of a quadtree block decomposition after node 6 is processed. Dashed lines show the active border after node 7 is processed.

value after the first pass will be the distance to the nearest WHITE node preceding M). This is accomplished by examining the distance transform values of those blocks adjacent to M that have been processed already. M is then output to a temporary node list with its value set to the (partially calculated) distance transform. The nodes are output so that the result of the first pass will be a node list in reverse order from the input node list. The second pass calculates the distance transform for each node M with respect to those nodes that now precede M in the node list (i.e., those nodes that, in the original input tree, came after M). This ensures that each node examines all of its neighbors to deduce the correct distance transform value. The algorithm requires exactly two list searches and two list insertions for each node, with all searches performed in order and all insertions performed in the reverse of the input order (i.e., insertions to the head of the node list).[4]

The primary difference between this algorithm and the one presented in Section 3 is that in the linear quadtree, neighbor information is not passed down to leaf nodes from the internal nodes. Instead, information about each node's previously visited neighbors is stored in an *active border table* [14]. Since the node list is processed in order, the border of the region of the corresponding image consisting of the blocks that have already been processed has the shape of a staircase. For example, consider Fig. 5 where the blocks have been assigned labels matching their order in the input list. The heavy line represents the state of the *active border* after block 6 is processed. The broken line along the southern and eastern border of block 7 shows the change in the active border after block 7 is processed. The active border of a $2^n \times 2^n$ image consists of sets of horizontal and vertical segments such that the total length of each of these sets is 2^n pixel widths. Thus, a complete description of the neighboring nodes along the active border can be maintained using two tables each containing 2^n records.

The distance transform algorithm must visit corner-adjacent neighbors as well as side-adjacent neighbors. It is therefore necessary to maintain, in addition to two edge tables, a table containing the value at each potential node corner (referred to as a *vertex* in Samet and Tamminen [14].) This allows the algorithm to retrieve efficiently the distance transform for a node's NW neighbor in cases where neither edge table retains a record corresponding to the NW neighbor's value (e.g., in Fig. 5 node 1 is the NW neighbor of node 13). Since a vertex may fall anywhere within a range from 0 to 2^n along the length of each axis (with the vertices at 0 being identical), the vertex table must be of size $2 \cdot 2^n + 1$. A line segment with equation $X = Y + c$ will intersect the active border only once; therefore, the vertex table can be organized to store the record for the vertex at (X, Y) in location $c = 2^n + X - Y$ (2^n is added to yield a range from 0 to $2 \cdot 2^n$).

Algorithm 2, named LINEAR_TRANSFORM, encodes the procedure described in this section. As with Algorithm 1, a tree consisting of a single black leaf would have a distance transform value of ∞. Arguments very similar to those used in Theorem 1 of the previous section can be used to demonstrate the correctness of the algorithm.

Since procedure DOPASS is executed twice for each node of the tree, the algorithm is $O(N)$ if the sum of the calls to DOPASS are also $O(N)$. Assuming that REVERSE_ORDER_INSERT (which inserts a node at the head of the node list) operates in constant time, the only point that must be considered is the cost of the two **while** loops. Each iteration of these loops represents a comparison between two neighboring blocks of the quadtree, and each such neighbor pair is investigated exactly once on each traversal. For every pair of side adjacent leaf nodes M and M', the smaller node (say M') has no other neighbor along their common side other than M. Thus, the total number of side-adjacent pairs that must be examined is $O(N)$, yielding a total cost for each traversal of $O(N)$.

As an example of how DOPASS processes a node, consider node 13 from Fig. 5 during pass 1. Since 13 is not white, *trval*'s value is initially set to ∞. W neighbors are checked first, beginning with node 8 (the W neighbor of 13's upper left corner). Since 8's D value + radius (and consequently the value stored in the edge table) is 1, *trval* is set to be 2. Since 8 is not 13's only W neighbor, the edge table at 10's upper right corner is also checked. This does not affect *trval*, nor does the check of 13's SW neighbor (node 12). Visiting the N edge table provides the D value based on node 14, followed by a check on the NE

[4] Alternatively, both passes could be performed working forward from the end of the node list, with all insertions being to the end of the newly created list. This may be preferable for disk-based processing.

neighbor (node 5). Neither reduces 13's D value. Finally, the NW neighbor is checked and found to be white. Thus, 13's D value is set to 13's radius. *x_edge*, *y_edge*, and *vert* are updated to reflect 13's final D value.

5. CONCLUSIONS

Two linear time algorithms have been presented to calculate the chessboard distance transform for quadtrees. Algorithms have been presented for both the pointer-based and linear quadtree representations, where both algorithms make two passes through the quadtree, with the second pass in the reverse order of the first pass. While the algorithms are quite similar, some differences may be noted. The primary difference is in the method whereby neighbor information is provided to the current node. The pointer-based quadtree algorithm makes use of neighbored traversals, with each traversal supplying necessary neighbor information to each black node through parameters passed to the traversal function. The linear quadtree algorithm makes use of a set of active border tables to store all necessary information about neighbors of the current node. Accesses to the quadtree are in terms of list operations.

A further difference between the two algorithms is that the linear quadtree algorithm is purely iterative, while the pointer-based quadtree algorithm is recursive (or stack-driven). The linear quadtree algorithm requires a "next node" operation to support a visit to each leaf node in order, and construction of an output tree where the value for each leaf is provided in reverse order. The pointer quadree algorithm requires a stack and an operation to access children.

As mentioned previously, the choice between a pointer-based or linear quadtree representation is usually determined by whether the application is disk or RAM based. Thus it does not make sense to declare that one or the other of these representations is "better" since the application often determines which is appropriate. Instead, these algorithms are complementary in that they provide a means whereby the distance transform may be computed in linear time in whichever representation is selected.

APPENDIX

The algorithms presented in this appendix are written in PASCAL with the extension of the **for** ⟨variable⟩ **in** ⟨set⟩ construct. This construct iterates ⟨variable⟩ over each item in ⟨set⟩. In addition, the following operations are assumed to be predefined.

GRAY, WHITE, and BLACK are boolean operations which are true iff the node value is gray (i.e., an internal node), white, or black, respectively.

SON(*node*, *quad*) returns the son of node *node* in quadrant *quad*. SONI is identical to SON except that if *node* is a leaf node, then *node* is returned instead of its son.

QUAD(*side*, *side*) returns the quadrant bounded by the two sides; e.g., QUAD(N, W) = NW.

OPQUAD(*quad*) and OPSIDE(*side*) return the opposite quadrant and side, respectively; e.g., OPQUAD(NW) = SE; OPSIDE(N) = S.

CSIDE(*side*) and CCSIDE(*side*) return the side clockwise and counter-clockwise to *side*, respectively; e.g., CSIDE(N) = E.

SIDE1(*quad*) and SIDE2(*quad*) return the first and second sides adjacent to *quad*, respectively; e.g., SIDE1(NW) = N, SIDE2(NW) = W.

WIDTH, RADIUS, X_OF, Y_OF, and VALUE return a node's width, a node's radius, x and y coordinates of a node's upper left corner, and a node's value, respectively.

REVERSE_ORDER_INSERT(*out*, *x*, *y*, *width*, *color*, *D*) appends to the head of list *out* a quadtree node with upper-left corner at (x, y), width *width*, color value *color*, and distance transform value D.

ALGORITHM 1. Pointer-based quadtree chessboard distance transform algorithm.

```
type
  direction = (N, E, S, W, NW, NE, SW, SE);
  neighbors = array [direction] of ↑NODE;
  corder = array [1..4] of direction;
end;

{ Compute and return the distance transform for pointer-based quadtree intree. It is assumed that all .D components
  of black nodes are initially infinite. }
procedure POINTER_TRANSFORM(intree : QUADTREE);
var
  i : direction;
  narray : neighbors;
  childorder : corder;
```

```
begin
  for i in (N, S, E, W, NW, NE, SW, SE) do narray[i] := nil;
  childorder[1]:=NW; childorder[2]:=NE;
  childorder[3]:=SW; childorder[4]:=SE;
  TRAVERSE(root, narray, childorder); { NW_to_SE traversal }
  childorder[1]:=SE; childorder[2]:=SW;
  childorder[3]:=NE; childorder[4]:=NW;
  TRAVERSE(root, narray, childorder) { SE_to_NW traversal }
end;

{ Perform a neighbored traversal, visiting the children in the order specified by childorder. }
procedure TRAVERSE(node : ↑ NODE; narray : neighbors; childorder : corder);
var
  d,child : direction;
  cnarray : neighbors;
begin
  if WHITE(node) then begin
    for d in (SIDE1(childorder[4]), childorder[3], SIDE2(childorder[4]), childorder[4]) do
      if BLACK(narray[d]) then
        narray[d] ↑ .D := RADIUS(narray[d])
  end
  else if BLACK(node) then begin
    for d in (OPQUAD(childorder[4]), SIDE1(childorder[2]), childorder[3],
          SIDE2(childorder[3])) do
      if WHITE(narray[d]) then
        node ↑ .D := RADIUS(node)
    for d in (SIDE1(childorder[4]), childorder[3], SIDE2(childorder[4]), childorder[4]) do
      if BLACK(narray[d]) then
        narray[d] ↑ .D := min(narray[d] ↑ .D, node ↑ .D+RADIUS(node)+RADIUS(narray[d]))
  end
  else begin { GRAY node code derived from Samet [10] }
    for i:=1 to 4 do begin
      child:= childorder [i];
      cnarray[child] := SONI(narray[child], QUAD(OPSIDE(child), CSIDE(child)));
      cnarray[QUAD(child, CSIDE(child))] := SONI(narray[QUAD(child, CSIDE(child))],
              QUAD(OPSIDE(child), CCSIDE(child)));
      cnarray[CSIDE(child)] := SONI(narray[CSIDE(child)], QUAD(child, CCSIDE(child)));
      cnarray[QUAD(OPSIDE(child), CSIDE(child))] := SONI(narray[CSIDE(child)],
              QUAD(OPSIDE(child), CCSIDE(child)));
      cnarray[OPSIDE(child)] := SON(node, QUAD(OPSIDE(child), CSIDE(child)));
      cnarray[QUAD(OPSIDE(child), CCSIDE(child))] :=
          SON(node, QUAD(OPSIDE(child), CCSIDE(child)));
      cnarray[CCSIDE(child)] := SON(node, QUAD(child, CCSIDE(child)));
      cnarray[QUAD(child, CCSIDE(child))] :=
          SONI(narray[child], QUAD(OPSIDE(child), CCSIDE(child)));
      TRAVERSE(SON(node, child), cnarray, childorder)
  end
end
```

ALGORITHM 2. Linear quadtree chessboard distance transform algorithm.

```
type
  EDGE = record
    length : integer; { length of edge segment described }
    posit : integer; { coordinate of edge segment in other dimension }
    D : real { value of edge segment, i.e., the distance transform value along that segment}
  end;
```

```
var
  n : integer; { assume initialized as depth of the quadtree; i.e., width of quadtree = 2^n}
  x_edge, y_edge : array [0..2^n] of EDGE;
  vert : array [0..2 * 2^n] of integer; { Vertex array }

{ Compute the chessboard distance transform for each node of a quadtree. The value stored in the edge tables is the
  sum of the distance transform of the node along that edge plus its radius. The reason for this is that dt(node) =
  radius(node) + radius(neighbor) + dt(neighbor) }
function LINEAR_TRANSFORM(intree : QUADTREE) : QUADTREE;
var
  temp, outtree : QUADTREE;
  nd : ↑ NODE:
begin
  y_edge[0].length := x_edge[0].length := 2^n; { initialize }
  y_edge[0].D := x_edge[0].D := vert[2^n] := ∞;
  y_edge[0].posit := x_edge[0].posit := 0;
  foreach nd in intree do { first pass }
    DOPASS(temp, nd, X_OF(nd), Y_OF(nd), WIDTH(nd), true);
  y_edge[0].length := x_edge[0].length := 2^n; { re-initialize }
  y_edge[0].D := x_edge[0].D := vert[2^n] := ∞;
  y_edge[0].posit := x_edge[0].posit := 0;
  foreach nd in temp do { second pass }
    DOPASS(outtree, nd, X_OF(nd), Y_OF(nd), WIDTH(nd), false);
  LINEAR_TRANSFORM := outtree
end;

{ Calculate the distance transform value with respect to the preceding nodes of a node with value trval, upper left
  corner (fx, fy), and width width. The node with its resulting distance transform value will be inserted into linear
  quadtree out with its position modified so that the resulting file is in reverse order from the input file. firstp is 'true'
  iff this is the first pass. }
procedure DOPASS(out : QUADTREE: nd: ↑ NODE; fx, fy, width : integer; firstp : boolean);
var
  oldval, cur,, outval, newx, newy, oldcurr, trval : integer;
begin
  if WHITE(nd) then trval := 0 else if firstp then trval := ∞ else trval := nd ↑ .D;
  if trval <> 0 then begin { non-white node–process distance transform }
    curr := fy; { first do west (east) edge }
    while curr < fy + width do begin
      trval := min(trval, width/2 + y_edge[curr].D); { check each neighbor }
      oldcurr := curr; curr := curr + y_edge[curr].length
    end; { now oldcurr points to last segment of edge }
    vert[2^n + fx − (fy + width)] := y_edge[oldcurr].D; { NW of curr's S neighbor }
    { Check SW (NE) corner. If haven't seen SW (NE) neighbor, don't update. }
    if y_edge[fy].length > width then curr := fy; else curr := fy + width;
    if y_edge[curr].posit = fx then { We have visited the corner neighbor }
      trval := min(trval, width/2 + y_edge[curr].D);
    curr := fx; { now do north (south) edge }
    while curr < fx + width do begin
      trval := min(trval, width/2 + x_edge[curr].D); { check each neighbor }
      oldcurr := curr; curr := curr + x_edge[curr].length
    end; { now oldcurr points to last segment of edge }
    vert[2^n + fx + width − fy] := x_edge[oldcurr].D; { NW of curr's E neighbor }
    { now check NE (SW) corner }
    if x_edge[fx].length > width then curr := fx; else curr := fx + width;
    if x_edge[curr].posit = fy then trval := min(trval, width/2 + x_edge[curr].D);
```

```
    { Finally, do NW (SE) corner }
    trval := min(trval, width/2 + vert[2^n + fx - fy])
  end; { now, trval is set to be the distance transform value for the node }
  { insert node into output tree (in reverse) }
  newx := 2^n - (fx + width); newy := 2^n - (fy + width);
  REVERSE_ORDER_INSERT(out, newx, newy, width, VALUE(nd), trval);
  { update tables }
  if trval <> 0 then { if black node, then add radius }
    trval := trval + width/2; { store transform + width in table }
  if x_edge[fx].length > width then begin { just updating part of segment }
    x_edge[fx + width].length := x_edge[fx].length - width;
    x_edge[fx + width].D := x_edge[fx].D;
    x_edge[fx + width].posit := x_edge[fx].posit
  end;
  x_edge[fx].length := width; x_edge[fx].D := trval;
  x_edge[fx].posit := x_edge[fx].posit + width;
  if y_edge[fy].length > width then begin
    y_edge[fy + width].length := y_edge[fy].length - width;
    y_edge[fy + width].D := y_edge[fy].D;
    y_edge[fy + width].posit := y_edge[fy].posit
  end;
  y_edge[fy].length := width; y_edge[fy].D := trval;
  y_edge[fy].posit := y_edge[fy].posit + width;
  vert[2^n + fx - fy] := trval
end;
```

REFERENCES

1. H. Samet, "Design and Analysis of Spatial Data Structures: Quadtrees, Octrees, and Other Hierarchical Methods," Addison-Wesley, Reading, MA, 1989.
2. H. Samet, "Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS," Addison-Wesley, Reading, MA, 1990.
3. I. Gargantini, An effective way to represent quadtrees, *Commun. ACM* **25,** 1982, 905–910.
4. E. Kawaguchi and T. Endo, On a method of binary picture representation and its application to data compression, *IEEE Trans. Pattern Anal. Mach. Intelligence* **2,** 1980, 27–35.
5. H. Samet, Neighbor finding techniques for images represented by quadtrees, *Comput. Graphics Image Process.* **18,** 1982, 37–57.
6. H. Samet and C. A. Shaffer, A model for the analysis of neighbor finding in pointer-based quadtrees, *IEEE Trans. Pattern Anal. Mach. Intelligence* **7,** 1985, 717–720.
7. C. A. Shaffer and H. Samet, Optimal quadtree construction algorithms, *Comput. Vision Graphics Image Process.* **37,** 1987, 402–419.
8. H. Samet, Distance transform for images represented by quadtrees, *IEEE Trans. Pattern Anal. Mach. Intelligence* **4,** 1982, 298–303.
9. H. Samet, A quadtree medial axis transform, *Commun. ACM* **26,** 1983, 680–693.
10. H. Samet, A top-down quadtree traversal algorithm, *IEEE Trans. Pattern Anal. Mach. Intelligence* **7,** 1985, 94–98.
11. C. L. Jackins and S. L. Tanimoto, Quad-trees, oct-trees, and k-trees—A generalized approach to recursive decomposition of Euclidean space, *IEEE Trans. Pattern Anal. Mach. Intelligence* **5,** 1983, 533–539.
12. A. Rosenfeld, H. Samet, C. Shaffer, and R. E. Webber, "Application of Hierarchical Data Structures to Geographic Information Systems, University of Maryland, Computer Science TR-1197, June, 1982.
13. H. Samet and R. E. Webber, On encoding boundaries with quadtrees, *IEEE Trans. Pattern Anal. Mach. Intelligence* **6,** 1984, 365–369.
14. H. Samet and M. Tamminen, Computing geometric properties of images represented by linear quadtrees, *IEEE Trans. Pattern Anal. Mach. Intelligence* **7,** 1985, 229–240.