Algorithm to expand regions represented by linear quadtrees

Clifford A Shaffer and Hanan Samet*

An algorithm is presented that changes to 'black' those 'white' pixels within a specified distance of any 'black' pixel in an image represented by a linear quadtree. This function is useful for answering queries in a geographic information system such as 'Find all wheat fields within five miles of a flood plain.' The algorithm works by computing the chessboard distance to nearby black pixels for large white nodes, and either leaves them white, changes them to black or repeats the process on each subquadrant, as required. Small white nodes are a priori within the given radius and require no further calculation. Thus only a small percentage of the nodes of the quadtree need extensive processing. The algorithm is easily applied to multicoloured images by treating all nonwhite colours as 'black'.

Keywords: linear quadtrees, hierarchical data structures, cartography, geographic information systems, region expansion

A useful feature for a geographic information system is the ability to generate a map which is 'black' at all pixels within a specified distance of the black regions of an input map. For the sake of discussion, the 'black' pixels of the input map are defined as those pixels initially within the regions of interest, while those pixels outside such regions are defined as 'white'. In the case of a multicoloured image, consider that all nonwhite pixels are in the regions of interest and that those white pixels within a specified distance of a nonwhite pixel will be set to black. The process which performs this task is sometimes referred to as region dilation or expansion. In this paper, the process will be referred to as the Within function. This function is important for answering queries such as 'Find all wheat fields within five miles of a flood plain.' Such a query would be processed by applying the Within function to a map whose black regions represent flood plains and then intersecting the result with a map representing the wheat fields. To simplify the presentation, it is assumed in the remainder of this paper that the function is performed on a binary image.

The quadtree data structure has proved useful for representing cartographic data. Today, the term 'quadtree' is used in a general sense to describe a class of data structures whose common property is that they are based on the principle of recursive decomposition of space. This paper is concerned with the 'region quadtree' as defined by Klinger[1] and will use the term quadtree to refer to it. Figure 1 is an example of a region and its corresponding quadtree. For a comprehensive survey of quadtrees and related hierarchical data structures, see Samet[2].

Quadtrees are of interest, in part, because they enable the solution of problems in a manner that focusses the work on the areas where the information is of the greatest density. The Within algorithm presented in this paper takes advantage of the quadtree structure to decrease the number of nodes for which expensive processing must be performed. This is accomplished by recognizing that only white nodes need processing (black nodes will not be modified) and that sufficiently small white nodes will always be within a given distance of a black pixel.

The linear quadtree technique[3,4] has become quite popular since it allows the storage of quadtrees in such a way that they may be manipulated efficiently as disc files. The technique is currently being used to store maps in an experimental geographic information system at the University of Maryland, USA[5]. In the linear quadtree, leaf nodes are represented by use of a locational code corresponding to a sequence of directional codes that locate the leaf along a path from the root of the tree. This collection is usually stored as a list sorted in increasing order of locational codes. Such an ordering is useful because it is the order in which the leaf nodes of the quadtree would be visited by a depth-first traversal of the quadtree. The Within algorithm described here,

Computer Science Department, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, USA
*Center for Automation Research and Computer Science Department, University of Maryland, College Park, MD 20742, USA

0262–8856/88/03162–07 $03.00

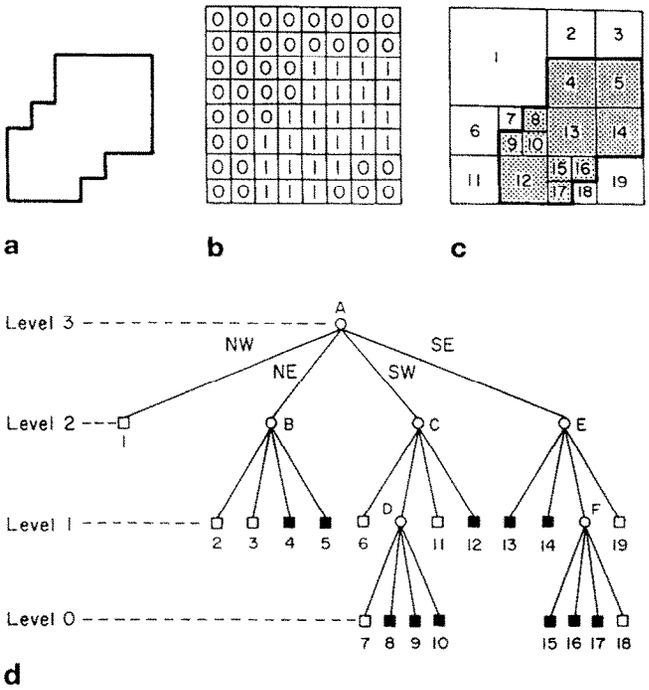

Figure 1. **a**, *A region;* **b**, *its binary array;* **c**, *block decomposition of the region in* **a** *(blocks in the region are shaded);* **d**, *quadtree representation of the blocks in* **c**

while easily modifiable for use with pointer-based quadtrees, is designed for use with the linear quadtree. Therefore it is important for the reader to realize that in the algorithm presented here there is a distinction made between functions that manipulate node addresses and functions that manipulate the node list that represents an actual quadtree.

The next section presents necessary definitions and notation. Following this, an algorithm for computing the Within function is presented.

DEFINITIONS AND NOTATION

The linear quadtree is implemented by first labelling each pixel of the quadtree with an address. The addressing schemes most commonly used are variations on one suggested by Morton[6] for use in indexing maps in the Canada Geographic Information System[7]. Such schemes will be referred to as 'Morton sequencing'. Their application to quadtrees was realized independently by Gargantini[3] and Abel and Smith[4]. Morton sequencing makes use of an addressing scheme which is equivalent to interleaving the bits of the binary representations of the x and y coordinates (each represented by a fixed number of digits) of a representative pixel in the node's block. In Figure 2, for example, 3-bit binary representations of the row and column coordinates are indicated along the bottom and right sides of an 8 × 8 array. The locational code of each pixel is formed by interleaving the bits so that the y bit precedes the x bit at each position. In Figure 2, these pixel addresses are represented with base-4 digits (i.e. each pair of x and y bits corresponds to a single base-4 digit). When the addresses of the pixels are sorted in increasing order, the result is equivalent to a depth-first traversal such

	000	001	010	011	100	101	110	111
000	000	001	010	011	100	101	110	111
001	002	003	012	013	102	103	112	113
010	020	021	030	031	120	121	130	131
011	022	023	032	033	122	123	132	133
100	200	201	210	211	300	301	310	311
101	202	203	212	213	302	303	312	313
110	220	221	230	231	320	321	330	331
111	222	223	232	233	322	323	332	333

Figure 2. Morton code address scheme for labelling pixels

that quadrants are visited in the order NW, NE, SW and SE.

Several linear quadtree variants have been proposed that differ in terms of what specific leaf nodes will be stored. For example, the original formulation[3] observes that the white nodes of the quadtree may be regenerated from the positions of the black nodes. There are various space/time tradeoffs involved[8]. In this paper, the linear quadtree is defined to explicitly store all leaf nodes.

Given the above method for addressing pixels, node addresses can in turn be generated by stipulating that each node will be given the address of the least valued pixel contained within the block that it represents. Figure 3 shows the block decomposition for the image in Figure 1 with each block given the address (in base 4) of the least valued pixel contained within that block. Note that the node in the NW quadrant of the image in Figure 3 has a 0 value in the most significant position (indicating a NW branch), all nodes in the NE quadrant have a 1 value in the most significant position, etc. This method of addressing blocks is incomplete as there is no indication of the block size. However, there are a number of ways to remedy this, one of which is to append to the address the level at which the block is found. Regardless of the method used to address quadtree blocks, the list of quadtree blocks is kept sorted in increasing order of locational code.

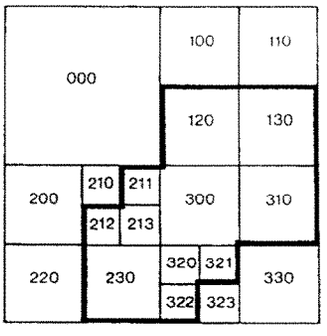

Figure 3. Morton code addresses for the blocks of Figure 1

Given this sorted list of quadtree blocks, some means must be found to organize it so that insertions, deletions and node searches may be performed efficiently. In addition, it is important that the organization method lends itself to offline storage of large images. The B-tree[9] is a data structure that meets these requirements. B-trees are very efficient in that the number of accesses necessary to retrieve a given key from secondary storage is kept low. This is partly because the tree is always balanced, and partly because the branching factor is very high. Both Abel and Smith[4] and Samet *et al.*[5] use a linear quadtree encoding in conjunction with B^+-trees to store images.

Throughout, this paper will use the 'chessboard distance metric'[10], which is defined as $\max(d_x, d_y)$ where d_x and d_y are the horizontal and vertical distances, respectively, between two points. Thus, the locus of points within chessboard distance R of a point will form a square of side length $2R$. In the authors' algorithm, the 'distance between two nodes' refers to the minimum chessboard distance between their borders. Although the Euclidean distance might be considered to be more accurate, the chessboard distance metric is more suited to the quadtree representation. Therefore only the latter will be treated in this paper.

'WITHIN' ALGORITHM

The Within function changes to black those white pixels of an image which are within distance R of a black pixel. The authors have previously reported[11] an algorithm for computing the Within function which works by expanding each black block of the input image by R units (where R is the radius) and inserting all the nodes making up this expanded square into the output tree. This leads to many redundant node insertions. In addition, many of the nodes inserted are small, and are eventually merged to form larger nodes. A polygon dilation algorithm has recently been presented[12] which traverses the image in two passes, modifying the values of white nodes based on the values of the nodes seen previously during the current pass through the node list.

The new algorithm presented here is based on the chessboard distance transform algorithm of Samet[10]. An outline of the algorithm is given in Figure 4, and a more detailed encoding is given in Appendix 1. The difference between the two algorithms is that the detailed version shows how many of the calculations can be performed in an efficient manner. This is crucial in obtaining satisfactory performance.

The algorithm does the following for each node N of the input quadtree. If N is black, then it is inserted into the output tree. If N is white, and its width is less than or equal to $(R + 1)/2$, then it must lie entirely within R pixels of some black node. This is true because one of the siblings of N must contain a black pixel whose distance to the border of N is at most $(R - 1)/2$. Thus N is made black and inserted into the tree. If N is white and has a width greater than $(R + 1)/2$, then the distance from the border of N to the borders of nearby nodes (i.e. a subset of those nodes within radius R) is computed. If this distance is such that N is completely within radius R of a black pixel, then N is inserted as a black node

```
procedure WITHIN(INTREE, OUTTREE, RADIUS);
/* Create a map OUTTREE which is BLACK for all WHITE pixels of INTREE within
   RADIUS units of a non-WHITE pixel. */
begin
 global node list INTREE, OUTTREE; /* input and output quadtrees */
 global integer RADIUS; /* radius value */
 node pointer ND; /* pointer to current node */

 for ND in INTREE do
  if TYPE(ND) = 'BLACK' then INSERT(OUTTREE, ND);
  else /* WHITE node */
   if width(ND) < (RADIUS+1)/2 then INSERT(OUTTREE, ND);
   else /* A large WHITE node */
    DOLARGE(ND);
end;

procedure DOLARGE(ND);
/* Process a large WHITE node */
begin
 node pointer ND;
 global node list INMAP, OUTMAP;
 global integer RADIUS;
 integer D; /* chessboard distance */
 quadrant I;

 D ← Chessboard distance from the center of N to the border of
      the nearest BLACK node in the direction of N's neighbors
      in the 8 principal directions;
 if D+WIDTH(ND)/2 ≤ RADIUS then INSERT(OUTTREE, SET(ND, 'BLACK'));
 else if D-WIDTH(ND)/2 > RADIUS then INSERT(OUTTREE, ND);
 else
  for I in {'NW','NE','SW','SE'} do
   WITHIN(SON(ND,I));
end;
```

Figure 4. Outline of an algorithm to compute the Within function.

into the output tree. If N is completely outside that radius, then it is inserted as white. Otherwise, portions of N will be inserted as black and portions as white.

When N is a white node of width greater than $(R + 1)/2$, the distance computation is performed as follows. First, consider the horizontal and vertical neighbours of N. The diagonal neighbours of N will be considered later. For each direction d in {N,E,S,W}, the address of the equal sized neighbouring block D in that direction is computed, and the actual block in the linear quadtree node list (say D') is then located.

If the d-direction equal sized neighbour of N does in fact exist (i.e. N is not on the d edge of the image), then there are three cases to consider: D is white, grey or black. If D is white, then normally there will be no black pixels in direction d that are within distance R of N. A special case occurs when the width of N is less than R (but greater than $(R + 1)/2$). In this case, if the actual neighbouring node D' also has width less than R, it is possible that the neighbour of D' in direction d will contain black pixels which are closer to some parts of N than any other black pixels (e.g. the upper rightmost pixels of node N in Figure 5). Thus the neighbour of D' in direction d must also be visited if the widths of N and D' are equal and are less than R and if d' is white.

If D is grey, then its subquadrants are visited to determine whether they contain black blocks within distance R of N. 'Grey' means that the equal sized neighbour of N is actually made up of more than one leaf block. For each black subquadrant of D (say D') within distance R of N, N will be decomposed as necessary, with those portions within distance R of D' inserted as black.

If D is black, then the distance from N to D is 0. In such a case, if the width of N is less than R, then N is inserted as black into the output tree. Otherwise, each quadrant of N is compared in turn against D. If

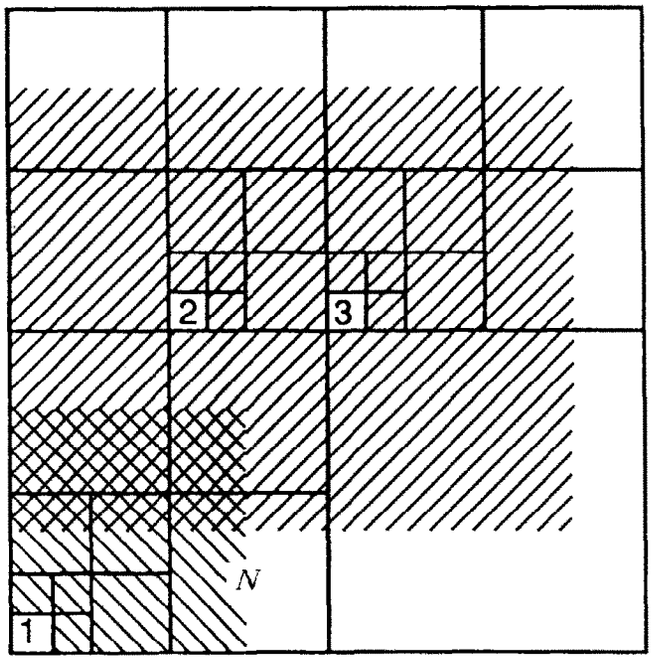

Figure 5. Quadtree block decomposition for an image whose pixels 1, 2 and 3 are black, with all other pixels being white. The shading shows all pixels within a distance of five pixels of the black nodes. Note that the rightmost pixels at the top of node N are within five pixels distance of nodes 2 and 3, but not within five pixels distance of node 1

the quadrant is entirely within distance R of D, then it is inserted as black. If the quadrant is both partially within and partially outside of distance R from D, then N is subdivided further.

In a similar manner, the neighbours of N in each direction d in {NW, NE, SW, SE} must also be visited. As with the adjacent neighbour case, if the equal sized neighbour D is white, then normally D will not affect N. However, if N and D have widths less than R, and D is white, then neighbours of D may contain black pixels within distance R of certain pixels of N. These neighbours must also be visited. As an example, if D is the NW neighbour of N, then the N, W and NW neighbours of D may need to be visited. Grey corner neighbours of N must have their subquadrants examined to locate any black pixels within distance R of N, again possibly causing a decomposition of N. When D is black, it will be at distance 0 from the border of N, thereby possibly causing a decomposition of N.

Table 1 compares execution times for new and old algorithms on the images shown in Figures 6 and 7. Both images are represented by 512 × 512 pixel quadtrees. The quadtree of Figure 6 contains 4693 nodes while that of Figure 7 contains 3253 nodes. The times in Table 1 represent the number of CPU seconds required to execute the algorithms on a Vax 11/785 running BSD 4.3 Unix. Each algorithm is applied to the two images for radius values ranging between 1 and 16.

The new algorithm is an improvement over the old one for two reasons. First, only large white nodes need excessive computation; since most nodes in a quadtree are small, very few nodes generate much work. Secondly, although input nodes may be visited several times when neighbouring nodes compute their chessboard distances, the number of duplicate insertions for a newly created black node will be greatly reduced.

In certain extreme cases, it is possible for the old algorithm to perform more efficiently than the new algorithm. This occurs when the image contains relatively few black nodes, and these nodes are spaced well apart. For such images, the numbers of merges and duplicated insertions required by the old algorithm are minimized, while the number of large white nodes requiring additional computation by the new algorithm is maximized. This can be seen in Table 1 for radius 2 expansion on the 'ACC' land use image. In this case, the old algorithm is slightly faster than the new algorithm. However, as illustrated by the remainder of the test data, such situations are rare and are significant only for small radius values.

Table 1. Execution times for the Within function

Distance	Figure 6 time (seconds)		Figure 7 time (seconds)	
	New algorithm	Old algorithm	New algorithm	Old algorithm
1	14.5	32.9	10.9	15.4
2	19.4	23.7	14.7	13.7
3	18.5	53.0	14.7	28.5
4	22.7	30.3	17.6	20.0
5	34.0	68.5	26.9	39.3
6	35.1	49.6	28.9	30.2
7	29.1	90.2	27.0	53.4
8	30.3	52.6	26.4	36.3
9	43.3	103.1	39.3	63.9
10	41.7	75.2	37.0	46.8
11	53.4	126.1	49.7	77.4
12	49.0	76.3	44.4	53.1
13	67.2	138.8	63.1	87.4
14	60.2	98.3	54.7	65.7
15	51.3	161.7	52.1	101.9
16	45.2	94.9	44.9	68.9

Figure 6. Floodplain map

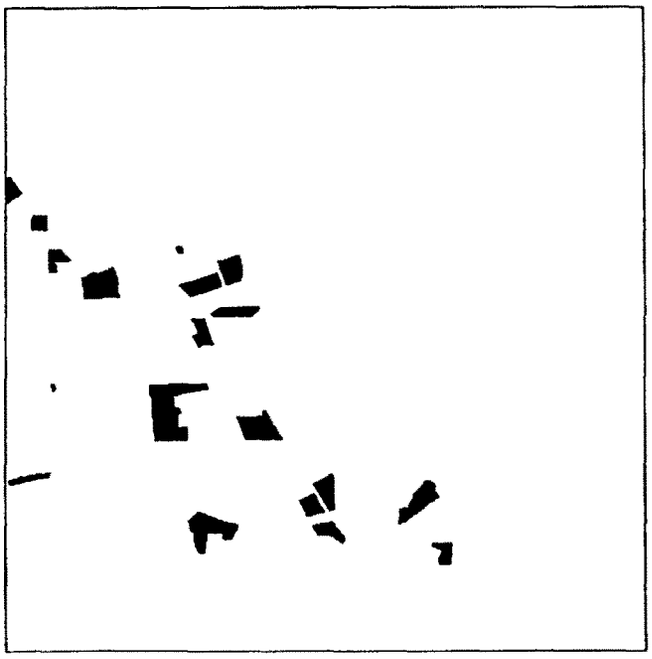

Figure 7. 'ACC' land use class map

CONCLUSIONS

A new algorithm has been presented which computes the Within function for a linear quadtree. This algorithm is a considerable improvement over an earlier naïve algorithm which simply expanded each black node and inserted the result as a series of quadtree nodes. Given a distance R for which the Within function is being computed, the new algorithm computes the chessboard distances to nearby black nodes only for white nodes larger than $(R + 1)/2$, changing subquadrants to black as required.

ACKNOWLEDGEMENTS

The authors are grateful to D C Mason for pointing out a long standing error in an earlier version of this algorithm, and to Chuang Heng Ang for suggesting the necessary modifications to fix this error. The support of the US National Science Foundation under Grant DCR–86–05557 is gratefully acknowledged.

REFERENCES

1 **Klinger, A** 'Patterns and search statistics' in **Rustagi, J S (ed.)** *Optimizing methods in statistics* Academic Press, New York, NY, USA (1971) pp 303–337

2 **Samet, H** 'The quadtree and related hierarchical data structures' *ACM Comput. Surv.* Vol 16 No 2 (June 1984) pp 187–260

3 **Gargantini, I** 'An effective way to represent quadtrees' *Commun. ACM* Vol 25 No 12 (December 1982) pp 905–910

4 **Abel, D J and Smith, J L** 'A data structure and algorithm based on a linear key for a rectangle retrieval problem' *Comput. Vision, Graphics, Image Process.* Vol 24 No 1 (October 1983) pp 1–13

5 **Samet, H, Rosenfeld, A, Shaffer, C A and Webber, R E** 'A geographic information system using quadtrees' *Pattern Recogn.* Vol 17 No 6 (1984) pp 647–656

6 **Morton, G M** *A computer oriented geodetic data base and a new technique in file sequencing* IBM, Ottawa, Canada (1966)

7 **Comeau, M A** 'A coordinate reference system for spatial data processing' *CLDS Tech. Bull.* No 3 (November 1981)

8 **Shaffer, C A, Samet, H, and Nelson, R C** 'QUILT: a geographic information system based on quadtrees' *Computer Science TR 1885* University of Maryland, College Park, MD, USA (July 1987)

9 **Comer, D** 'The ubiquitous B-tree' *ACM Comput. Surv.* Vol 11 No 2 (June 1979) pp 121–137

10 **Samet, H** 'A distance transform for images represented by quadtrees' *IEEE Trans. Pattern Anal. Mach. Intell.* Vol 4 No 3 (May 1982) pp 298–303

11 **Samet, H, Rosenfeld, A, Shaffer, C A, Nelson, R C and Huang, Y–G** 'Application of hierarchical data structures to geographical information systems, phase III' *Computer Science TR 1457* University of Maryland, College Park, MD, USA (November 1984)

APPENDIX 1: FULL VERSION OF THE 'WITHIN' ALGORITHM

The Within algorithm presented here assumes the existence of a number of functions for accessing the linear quadtree and manipulating quadtree nodes. The edges of a block in the quadtree are labelled N, S, E and W for north, south, east and west respectively. The quadrants are named NW, NE, SW and SE. NIL is the null pointer (in this algorithm, a value of NIL is returned by neighbour-finding functions if the desired node does not exist). INSERT(TREE, NODE) inserts

NODE into node list TREE at the correct position, splitting or merging nodes as necessary to maintain a region quadtree. FIND__NEIGHBOR(TREE, NODE, D, ABUT) returns the actual existing neighbour of NODE in direction D (which may be of any size), specifically at the corner where side D meets side ABUT. DIAGONAL(NODE, QUAD) manipulates the address field of NODE to return the address of the equal sized neighbour in the diagonal direction QUAD. FIND(TREE, NODE) returns the actual node in TREE which contains the address of NODE. FIND__DIAGONAL(TREE, NODE, QUAD) is equivalent to performing a FIND operation on the node created by DIAGONAL. NODETYPE(NODE) returns the type of the node (e.g. the colour). FATHER(NODE) modifies the address field of NODE so as to return the address for the father of NODE. SON(NODE, QUAD) modifies the address field of NODE so as to return the address for the son of NODE in quadrant QUAD. WIDTHOF (NODE) returns the width of NODE (always a power of two). XOF(NODE) and YOF(NODE) return the x and y coordinates for the upper left corner of NODE, respectively. COPY__FIELDS(NODE1, NODE2) copies all fields of NODE1 to NODE2, and returns a pointer to NODE2. SETNODE(ND, X, Y, DEPTH, VALUE) sets the descriptor for ND to have upper left corner (X, Y), depth DEPTH and value VALUE. QUAD(D1, D2) returns the quadrant bounded by sides D1 and D2; CSIDE(D) returns the side in the clockwise direction with respect to side D; and CCSIDE(D) returns the side in the anticlockwise direction with respect to side D. **log**(N) returns the base 2 logarithm of N.

When reading the following algorithm, it is important to keep in mind that the algorithm operates on a linear quadtree. Thus most node operations manipulate the address or value fields of a node template without actually querying or modifying the node list. Only INSERT modifies the node list; only FIND, FIND__NEIGHBOR and FIND__DIAGONAL query the node list.

The full listing of the Within algorithm is as follows.

```
procedure WITHIN(INTREE, OUTTREE, RADIUS);
/* Create a map OUTTREE which is BLACK for all WHITE pixels of INTREE within
   RADIUS units of a non-WHITE pixel. */
begin
  global node list INTREE, OUTTREE; /* input and output quadtrees */
  global integer RADIUS; /* radius value */
  node pointer ND; /* pointer to current node */

  for ND in INTREE do
   begin
    if NODETYPE(ND) neq WHITE then
      INSERT(OUTTREE, ND); /* Don't modify non-WHITE node */
    else if WIDTHOF(ND) ≤ (RADIUS+1)/2 then
     begin /* small node must be within RADIUS */
      NODETYPE(ND) ← BLACK; INSERT(OUTTREE, ND);
     end;
    else DOLARGE(ND, XOF(ND), YOF(ND), WIDTHOF(ND));
   end;
end;


procedure DOLARGE(ND, X, Y, WID);
/* Compute the distance from node ND (which has upper left corner (X, Y) and width
   WID) to the non-WHITE pixels in each neighboring node. */
begin
  global node list INTREE, OUTTREE; /* input and output quadtrees */
  global integer RADIUS; /* radius value */
  value node pointer ND; /* pointer to large WHITE node being processed */
  value integer X, Y, WID; /* position and width of current node */
  node pointer Q, DNEIGH, DREAL, DPTR; /* local node pointers */
  direction D; /* current neighbor direction */

  Q ← create(node); DREAL ← create(node);
  for D in {'N', 'E', 'S', 'W'} do
   begin /* for each cardinal direction */
    COPY_FIELDS(ND, Q); /* make a copy for temporary usage */
    DNEIGH ← FIND_NEIGHBOR(INTREE, Q, D, CCSIDE(D));
    if (DNEIGH neq NIL) and (NODETYPE(DNEIGH) = WHITE) and
      (RADIUS>WIDTHOF(DNEIGH)) and (WID=WIDTHOF(DNEIGH))) then
       /* must visit DNEIGH's D direction neighbor */
       DNEIGH ← FIND_NEIGHBOR(INTREE, DNEIGH, D, CCSIDE(D));
    CHECK_NODE(ND, DNEIGH, X, Y, WID);

    /* visit diagonal neighbor in direction QUAD(D, CSIDE(D)) */
    COPY_FIELDS(ND,Q); /* make a copy for temporary usage */
    DNEIGH ← DIAGONAL(Q, QUAD(D, CSIDE(D)));
    if DNEIGH neq NIL then
     begin /* DNEIGH is in the tree */
      COPY_FIELDS(DNEIGH, DREAL); /* we want to preserve DNEIGH */
      DPTR ← FIND(INTREE, DREAL); /* find the real neighbor block in tree */
      CHECK_NODE(ND, DPTR, X, Y, WID);
      if (RADIUS > WIDTHOF(ND)) and (NODETYPE(DPTR) = WHITE) then
       begin /* DPTR is small - must visit DPTR's neighbors */
        COPY_FIELDS(DNEIGH, DREAL);
        DPTR ← FIND_NEIGHBOR(INTREE, DREAL, D, CCSIDE(D));
        CHECK_NODE(ND, DPTR, X, Y, WID);
        COPY_FIELDS(DNEIGH, DREAL);
        DPTR ← FIND_DIAGONAL(INTREE, DREAL, QUAD(D, CSIDE(D)));
        CHECK_NODE(ND, DPTR, X, Y, WID);
        COPY_FIELDS(DNEIGH, DREAL);
        DPTR ← FIND_NEIGHBOR(INTREE,DREAL,CSIDE(D),CCSIDE(CSIDE(D)));
        CHECK_NODE(ND, DPTR, X, Y, WID);
       end;
     end;
   end;
end;


procedure CHECK_NODE(ND, Q, X, Y, WID);
/* Examine Q, a node within RADIUS pixels of ND. ND has upper left corner (X, Y)
   and width WID. If Q is GRAY, then apply DIST_NODE to the ancestor of Q which
   is an equal-sized neighbor of ND. If Q is non-WHITE, change the appropriate por-
   tions of ND's block to BLACK. */
begin
  value node pointer ND, Q;
  value integer X, Y, WID;

  if(Q neq NIL) then
   begin
    if WIDTHOF(ND) > WIDTHOF(Q) then
     begin /* Q is GRAY - i.e., composed of subblocks smaller than ND */
      while WIDTHOF(ND) > WIDTHOF(Q) do Q ← FATHER(Q);
      DIST_NODE(Q, WIDTHOF(Q), X, Y, WID);
     end;
    else if NODETYPE(Q) neq WHITE then /* Q is non-WHITE leaf node */
     begin /* compute distance, and either insert or split ND */
      T ← SCOMPARE(XOF(Q), YOF(Q), WIDTHOF(Q), X, Y, WID);
      if T + WIDTHOF(ND) ≤ R then /* node within radius */
       begin
        NODETYPE(ND) ← BLACK; INSERT(OUTTREE,ND);
       end;
      else if T < R then /* node beyond radius */
        SPLITDIST(X, Y, WID, XOF(Q), YOF(Q), WIDTHOF(Q));
     end;
    /* else Q is a WHITE node - do nothing */
   end;
end;


procedure DIST_NODE(Q, WID, X, Y, W);
/* Find the distance from node Q with width WID to the block with upper left corner
   (X, Y) and width W. */
begin
  global node list OUTTREE; /* output quadtree */
  global integer RADIUS; /* radius value */
  value node pointer Q; /* current node */
  value integer WID, X, Y, W; /* width of Q and block descriptor */
  node pointer QSON; /* local node pointer */
  integer SDIST; /* distance to QSON */

  QSON ← create(node);
  COPY(ND, QSON);
  if WIDTHOF(QSON) neq 1 then /* not a pixel-sized node */
    FIND(INMAP, QSON ← SON(QSON, SW));
  else FIND(INMAP, QSON);
  if WIDTHOF(QSON) = WID then /* found the leaf node */
    if NODETYPE(QSON) neq WHITE then
     begin
      SDIST ← SCOMPARE(XOF(QSON),YOF(QSON),WIDTHOF(QSON),X,Y,W);
      if SDIST + W ≤ RADIUS then
        INSERT(OUTTREE, CREATE_NODE(X, Y, log(WID), BLACK));
      else if SDIST < R then
        SPLITDIST(X, Y, W, XOF(QSON), YOF(QSON), WIDTHOF(QSON));
      return;
     end;
    else
      return;
  WID ← WID/2;
  if SCOMPARE(XOF(ND), YOF(ND), WID, X, Y, W) < R then
    DIST_NODE(SON(COPY_FIELDS(ND, QSON), SW), WID, X, Y, W);
  if SCOMPARE(X_OF(ND)+WID,Y_OF(ND),WID,X,Y,W) < R then
    DIST_NODE(SON(COPY_FIELDS(ND, QSON), SE), WID, X, Y, W);
  if SCOMPARE(X_OF(ND),Y_OF(ND)+WID,WID,X,Y,W) < R then
    DIST_NODE(SON(COPY_FIELDS(ND, QSON), NW), WID, X, Y, W);
  if SCOMPARE(X_OF(ND)+WID,Y_OF(ND+WID,WID,X,Y,W) < R then
    DIST_NODE(SON(COPY_FIELDS(ND, QSON), NE), WID, X, Y, W);
end;
```

```
procedure SPLITDIST(GX, GY, GW, FX, FY, FW);
/* Change to BLACK that part of the block described by GX, GY, and GW that is
   within RADIUS of the block represented by FX, FY, and FW. */
begin
  value integer GX, GY, GW, FX, FY, FW;
  integer WID, T;
  node pointer ND;

  ND ← create(node);
  WID ← GW/2;
  T ← SCOMPARE(GX, GY, WID, FX, FY, FW);
  if T + WID ≤ R then
    INSERT(OUTTREE, SETNODE(ND, GX, GY, log(WID), BLACK));
  else if T < R then SPLITDIST(GX, GY, WID, FX, FY, FW);
  T ← SCOMPARE(GX+WID, GY, WID, FX, FY, FW);
  if T + WID ≤ R then
    INSERT(OUTTREE, SETNODE(ND,GX+WID,GY,LOG(WID),BLACK));
  else if T < R then SPLITDIST(GX+WID, GY, WID, FX, FY, FW);
  T ← SCOMPARE(GX, GY+WID, WID, FX, FY, FW);
  if T + WID ≤ R then
    INSERT(OUTTREE,SETNODE(ND,GX,GY+WID,LOG(WID),BLACK));
  else if T < R then SPLITDIST(GX, GY+WID, WID, FX, FY, FW);
  T ← SCOMPARE(GX+WID, GY+WID, WID, FX, FY, FW);
  if T + WID ≤ R then
    INSERT(OUTTREE, SETNODE(ND,GX+WID,GY+WID,LOG(WID),BLACK));
  else if T < R then SPLITDIST(GX+WID, GY+WID, WID, FX, FY, FW);
end;


integer procedure SCOMPARE(X1, Y1, W1, X2, Y2, W2);
/* Find the chessboard distance between two squares (closest approach) */
begin
  value integer X1, Y1, W1; /* description for square 1 */
  value integer X2, Y2, W2; /* description for square 2 */
  integer XDIST, YDIST;

  if X1 < X2 then XDIST ← X2 - (X1 + W1);
  else XDIST ← X1 - (X2 + W2);
  if Y1 < Y2 then YDIST ← Y2 - (Y1 + W1);
  else YDIST ← Y1 - (Y2 + W2);
  return(max(XDIST, YDIST));
end;
```