References
1. Berztiss, A.T. *Data Structures: Theory and Practice.* 2nd Ed. Academic Press, New York, 1975.
2. Cavouras, J.C. On the conversion of programs to decision tables: Methods and objectives. *Comm, ACM 17,* 8 (Aug. 1974), 456–462.
3. Cooper, C.B., Böhm and Jacopini's reduction of flowcharts, *Comm. ACM 10,* 8 (Aug. 1967), 463, 473
4. Imbrasha, M. and Rajaraman, V. Detection of logical errors in decision table programs. *Comm. ACM 21,* 12 (Dec. 1978), 1016–1025.
5. Knuth, D.E., Structured programming with GOTO statements. *Computing Surveys* 6, 4 (Dec. 1974), 261–301.
6. Ledgard, H.F. and Marcotty, M. A genealogy of control structures. *Comm. ACM 18,* 11 (Nov. 1975), 629–639.
7. Lew, A. Optimal conversion of extended-entry decision tables with general cost criteria. *Comm. ACM 21,* 4 (Apr. 1978), 269–279.
8. Lew, A., and Tamanaha, D. Decision table programming and reliability. *Proc. 2nd Intl. Conf. on Software Engr.*, 1976, 345–349.
9. Maes, R. On the representation of program structures by decision tables: A critical assessment. *Comptr. J. 21,* 4 (Nov. 1978), 290–295.
10. McCabe, T.J. A complexity measure. *IEEE Trans. on Software Engrg., SE-2,* 4 (Dec. 1976), 308-320.
11. Metzner, J.R., and Barnes, B.H. *Decision Table Languages and Systems.* Academic Press, New York, 1977.
12. Montalbano, M. *Decision Tables.* Science Research Associates, Chicago, 1974.
13. Peterson, W.W., Kasami, T., and Tokura, N. On the capability of while, repeat, and exit statements. *Comm. ACM 16,* 8 (Aug. 1973), 503-512.
14. Pollack, S.L., Hicks, H., and Harrison, W.J. *Decision Tables: Theory and Practice,* Wiley, New York, 1971.
15. Pooch, U.W. Translation of decision tables. *Computing Surveys 6,* 2 (June 1974), 125-151.
16. Trakhtenbrot, B.A. *Algorithms and Automatic Computing Machines.* D.C. Heath and Co., Lexington, Mass., 1963.

Graphics and Image Processing

James Foley*
Editor

An Effective Way to Represent Quadtrees

Irene Gargantini
The University of Western Ontario

A quadtree may be represented without pointers by encoding each black node with a quaternary integer whose digits reflect successive quadrant subdivisions. We refer to the sorted array of black nodes as the "linear quadtree" and show that it introduces a saving of at least 66 percent of the computer storage required by regular quadtrees. Some algorithms using linear quadtrees are presented, namely, (*i*) encoding a pixel from a $2^n \times 2^n$ array (or screen) into its quaternary code; (*ii*) finding adjacent nodes; (*iii*) determining the color of a node; (*iv*) superposing two images. It is shown that algorithms (*i*)-(*iii*) can be executed in logarithmic time, while superposition can be carried out in linear time with respect to the total number of black nodes. The paper also shows that the dynamic capability of a quadtree can be effectively simulated.

CR Categories and Subject Descriptors: I.4.2 [**Image Processing**]: Compression—*quadtrees*
General Term: Algorithms
Additional Key Words and Phrases: digital images, image encoding

1. Introduction

The quadtree [2, 3, 7, 8] is a hierarchical representation of a $2^n \times 2^n$ binary array that is made of unit square pixels that can be black or white. We refer to the subset

* Former editor of Graphics and Image Processing. Robert Haralick is the current editor of the department, which has recently been renamed Image Processing and Computer Vision. (See April '82 *Communications*, pp. 311-312.)

This work was sponsored by the Canadian Government under the NSERC grant #A7136.

Author's Present Address: I. Gargantini, The University of Western Ontario, Department of Computer Science, London, Canada N6A 5B9.

of black pixels as the "region" and to the subset of white pixels as the "region's background." As extensively discussed in the literature [1–3, 9–11], a quadtree can be effectively used to describe the successive partitions of a $2^n \times 2^n$ array ($n \geqq 1$) into quadrants, to separate a region from its background, and to represent a set of pixels belonging to the same quadrant (at any level of subdivision) as a single node. The quadtree can be defined as a tree whose nodes are either leaves or have four sons. A node can be grey, white, or black and is, in general, stored as a record with six fields, five of which are pointers (to the NW, NE, SW, and SE quadrants) and the sixth is a color's identifier.

The present paper introduces a new structure, called a "linear quadtree," with the following characteristics:

(*i*) Only black nodes are stored.

(*ii*) The encoding used for each node incorporates adjacency properties in the four principal directions.

(*iii*) The node representation implicitly encodes the path from the root to the node.

The main advantages of linear quadtrees, with respect to quadtrees, are:

(*i*) Space and time complexity depend only on the number of black nodes.

(*ii*) Pointers are eliminated.

2. Encoding Black Pixels

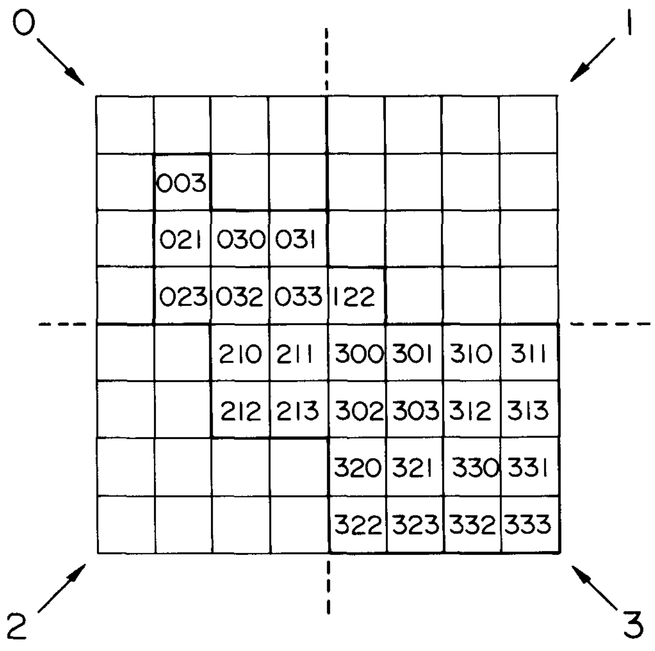
The first step is to devise an algorithm to encode black pixels. There are two distinct cases of this problem, one of which is associated with image processing [2, 8, 9] where the input is randomly generated. In the other case, found mainly in numerical analysis, all the black pixels in the $2^n \times 2^n$ array may be randomly accessed and the order of input may thus be predetermined.

Since the first case is more general, we concentrate on it. We adopt the following conventions:

(*i*) The NW quadrant is encoded with 0, the NE with 1, the SW with 2, and the SE with 3. Each black pixel is then encoded in a weighted quaternary code, i.e., with digits 0, 1, 2, 3 in base 4, where each successive digit represents the quadrant subdivision from which it originates. Thus, the digit of weight 4^{n-h}, $1 \leqq h \leqq n$, identifies the quadrant to which the pixel belongs at the hth subdivision. (See Fig. 1.)

(*ii*) The pair of integers (I, J), with $I, J = 0, 1, \ldots, 2^n - 1$ identifies the position of a pixel in the $2^n \times 2^n$ array. The encoding procedure consists of mapping the pair (I, J) into an integer K in base 4, which expresses the successive quadrants to which the (I, J) pixel belongs. For example, if $n = 3$ and $(I, J) = (6, 5)$, K will be 321. This means that pixel (6, 5) belongs to the SE quadrant in the first subdivision, to the SW quadrant in the second, and to the NE in the third (final) subdivision. (See Fig. 1.)

Fig. 1. Quadrants Labeling and Generation of Quaternary Codes.



To find K, we first write I and J as

$$I = c_{n-1}2^{n-1} + c_{n-2}2^{n-2} + \cdots + c_0$$
$$J = d_{n-1}2^{n-1} + d_{n-2}2^{n-2} + \cdots + d_0 \qquad (1)$$

where c_i, d_i is either 0 or 1. (Since I, J lie between 0 and $2^n - 1$, 2^{n-1} is the highest power in the binary representation.) Thus, c_i and d_i indicate which quadrant (I, J) belongs to at each level of subdivision. In fact, if (I, J) is in the NW quadrant, both highest powers of 2 are missing, i.e., $c_{n-1} = d_{n-1} = 0$. If pixel (I, J) is in the NE quadrant, the highest power of 2 in the I representation is missing, while the one relative to J is not, i.e., $c_{n-1} = 0$, $d_{n-1} = 1$. Similarly, if (I, J) is in the SW quadrant, $c_{n-1} = 1$, $d_{n-1} = 0$, and if (I, J) is in the SE quadrant, then $c_{n-1} = d_{n-1} = 1$. Once the first (largest) quadrant is identified, we partition it into four quadrants and repeat the procedure for $(n - 2)$ and so on, until the last quadrant consists of only four elements. Therefore, we have:

```
 procedure ENCODING (n, c, d, K); value n;
 reference c, d, K; integer n; integer array c, d, K;
(* given n, c and d, find quaternary code K *)
 begin integer l;
   for l ← n − 1 step − 1 until 0 do
         begin
         if c[l] = 0 and d[l] = 0 then K[l] = 0;
         if c[l] = 0 and d[l] = 1 then K[l] = 1;
         if c[l] = 1 and d[l] = 0 then K[l] = 2;
         if c[l] = 1 and d[l] = 1 then K[l] = 3
         end
 end;
```

In our previous example, $I = 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$, $J = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$, and thus, $K = 321$.

After we have encoded all the black pixels into their corresponding quaternary codes, we sort them and store them in an array or list. (If we can access the input randomly, we can choose which pixel to encode first and produce all the codes already sorted.) Condensation [9] can then be applied: if four pixels have the same representation except for the last digit, we eliminate them from the list and replace them with a code of $(n - 1)$ quaternary digits followed by some kind of marker, here denoted by X.

For instance, if pixels 310, 311, 312, and 313 are all in the array, they are replaced by $31X$. Similarly, if $30X$, $31X$, $32X$, $33X$ are present, we replace them by $3XX$ and so forth. After condensation is complete, we obtain an array (containing only black pixels or a covering thereof) which is still sorted if we suitably encode marker X with an integer > 3. We refer to this sorted array as the linear quadtree. The region shown in Fig. 1, for instance, is encoded by the sequence, 003 021 023 $03X$ 122 $21X$ $3XX$. Note that the linear quadtree corresponds to the postorder traversal of black nodes of a quadtree, e.g., Fig. 2.

The process of decoding a quaternary code is left to the reader. We note that the concept of labeling square pixels with quaternary codes and cubic pixels (voxels) with octal codes is also discussed in [4–6, 12].

3. Adjacency

Most frequently, storing pixels is motivated by some kind of application such as determination of the boundary of a region, determination of connectivity, and the like. Directly or indirectly, these problems require finding the pixel adjacent to a given one in a specified direction. With the newly proposed structure, the determination of adjacency is quite simple.

Let $K = (K_{n-1}K_{n-2} \ldots K_0)_4$ be the given pixel and $S = (S_{n-1}S_{n-2} \ldots S_0)_4$ its adjacent node in the southern direction. As an example, we show how to determine digits $S_{n-1}, S_{n-2}, \ldots, S_0$.

We distinguish two cases: in the first one, K and S belong to the same quadrant relative to the nth subdivision; in the other, K and S do not. If the last digit K_0 is either 0 or 1, then the node adjacent to K in the southern direction lies in the same quadrant (relative to the nth subdivision) as K. (See Fig. 3.) Therefore, $S_0 = 2$ if $K_0 = 0$, and $S_0 = 3$ if $K_0 = 1$. All other digits are the same. For instance, for $K = 2101$ (see Fig. 3), the adjacent node in the southern direction is $S = 2103$.

In the second case, i.e., if K_0 is either 2 or 3, a transition from two 'different' quadrants occurs. We note that $S_0 = 0$ if $K_0 = 2$, and $S_0 = 1$ if $K_0 = 3$. $S_1 = (K_1 + 2)$ modulo 4 in both cases. (See Fig. 3.) For all other digits, we remark that S_i, $2 \leqq i \leqq n - 1$ depends only on K_i and K_{i-1}. Basically, we increment K_i by 2 if K_{i-1} was either 2 or 3. For example, if $K = 0033$, $S = 0211$. The algorithm for determining the node adjacent to K in the southern direction is given below.

```
procedure S_ADJ_TO(n, K, S); value n;
reference K, S; integer n; integer array K, S;
(* given n and node K, find adjacent node S *)
  begin integer i, j;
    S[0] ← (K[0] + 2) mod 4;
    j ← i ← 1;
    while i ≠ n and j ≠ n − 1 do
      begin
        if K[i − 1] = 0 or K[i − 1] = 1 then
          for j ← i step 1 until n − 1 do S[j] ← K[j]
        else begin
          S[i] ← (K[i] + 2) mod 4; i ← i + 1
        end
  end;
```

Algorithms for finding adjacent nodes in other directions have structures similar to those shown above.

Fig. 2. Quadtree for Region of Fig. 1.

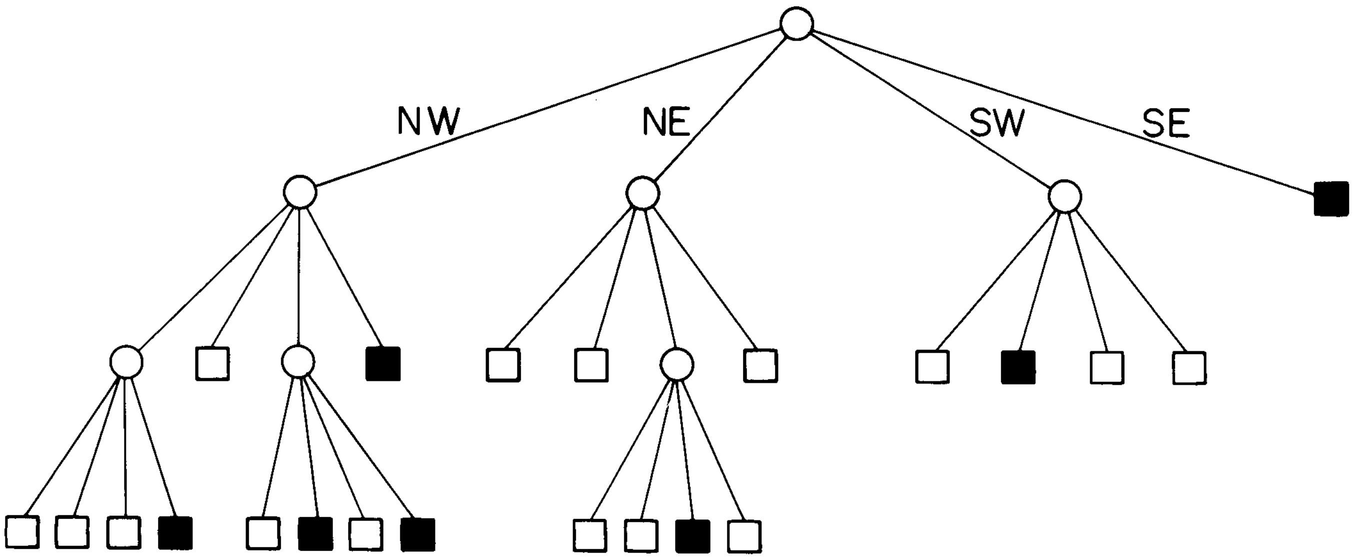
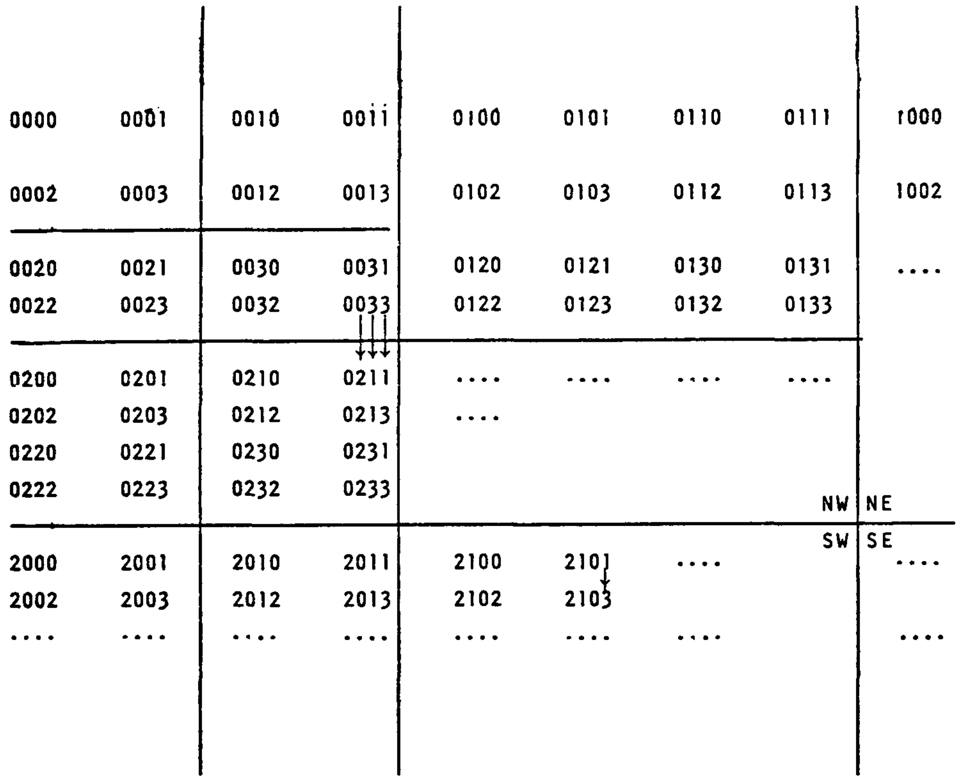


Fig. 3. Quaternary Codes for $n = 4$.



4. Search

Let K' denote the pixel adjacent to K in the wanted direction. Since we store only black nodes, we have to determine whether K' is black or white by searching the linear quadtree. Let Y denote the node used as comparand. Its representation consists of $1 \leqq k \leqq n$ quaternary digits (0, 1, 2, 3) and $(n - k)$ markers that we denote, as before, by X's. The criterion used to decide if K' is black or white is contained in the following informal algorithmic description.

Apply a binary search and distinguish the following cases:

(*i*) A node Y is found with n digits identical to the digits of K', i.e., $K' = Y$. It follows that K' is black.

(*ii*) A node Y is found with $k < n$ digits identical to those of K' and with $(n - k)$ markers (X's) in the remaining (weighted) positions. In this case, $K' \subset Y$, and it follows that K' is black.

(*iii*) If neither of the previous cases hold, then it follows that K' is white.

5. Dynamic Structure

One of the most useful features of the quadtrees is that an increase of the resolution parameter, for instance, from n to $n + 1$, requires further subdivision of the terminal nodes *only*, leaving the remaining part of the tree unchanged. Roughly speaking, a 'refinement' of the image merely requires a 'refinement' of the leaves. Quadtrees, therefore, are excellent structures for the storage of (slowly) changing pictures. We refer to this property as the 'dynamic refinement' of a quadtree.

At this point the reader may wonder how much we have lost, in terms of dynamic refinement, by replacing a quadtree with a list or array. We distinguish two cases. The first corresponds to refining only black nodes; in this case, the creation of a new array to store the updated blocks at level $(n + 1)$ requires no more nodes than in a traditional quadtree. The array at level n can be released to the free space.

The second case occurs when we want to refine all nodes, black and white. In a quadtree we refine all terminal nodes. Refining a white node is treated in exactly the same way as refining a black one. In our linear quadtree we have to supply a fast algorithm capable of identifying the white nodes not explicitly stored. We can compete with quadtrees, in terms of structure flexibility, only if we can generate the *largest* possible coverings of white pixels in the correct position as we traverse the list of black nodes. We can do this by analyzing the encoding of successive pairs of black nodes.

Let us start with an example. Take $n = 4$ and two consecutive black blocks, $Q = 010X$ and $Q1 = 0331$. (See Fig. 3.) We generate two sequences of white blocks: one sorted in increasing order; the other sorted in decreasing order. To create the first one, we consider the encoding of Q, and increase by one, one digit at a time, and starting from the rightmost, all *quaternary* digits, while inserting as many X's as possible at the right of the increased digit. We keep going until we produce all white blocks with encoding less than $Q1$. In the example we produce the sequence

$$011X, 012X, 013X, 02XX$$

To generate the descending sequence, we basically follow the same scheme: we decrease (by one) the rightmost *quaternary* $Q1$-digits, one at a time, and produce all white blocks with encoding greater than the last element of the previous sequence. In the example, we obtain

$$0330, 032X, 031X, 030X$$

We now formalize the above discussion. Let the mixed quaternary representation of Q be

$$Q = q_{n-1}4^{n-1} + \cdots + q_i 4^i + \cdots + q_0 \qquad (2)$$

where either $i = 0$ or i is the subscript for which $q_i \neq X$ and $q_{i-1} = \cdots = q_0 = X$. Let $i1$ of

$$Q1 = jq'_{n-1}4^{n-1} + \cdots + q'_{i1}4^{i1} + \cdots + q'_0 \qquad (3)$$

be similarly defined. i and $i1$ indicate which quaternary digits should be increased and decreased, respectively, in order to start the generation of the two sequences. (As before, Q and $Q1$ are two successive elements of the linear quadtree with $Q < Q1$.)

The procedure used in the example is basically correct, but still requires some improvement. First, we note that if Q and $Q1$ have the first j digits in common, these very same digits must belong to all the white blocks between Q and $Q1$. (In the example, there is only one digit, 0.) Second, we note that the last block produced in the ascending sequence ($2XX$ in the example) has the digit in the $(n - j - 1)$ position equal to $(q'_{n-j-1} - 1)$ and all other digits represented by X's. The last block produced is, therefore, the *largest block closest* to $Q1$ 'from below.'

Let U be formally given by

$$U = u_{n-1}4^{n-1} + \cdots + u_{n-j}4^{n-j} + u_{n-j-1}4^{n-j-1} + \cdots + u_0$$

where

$$u_{n-1} = q_{n-1} = q'_{n-1}, \cdots, u_{n-j} = q_{n-j} = q'_{n-j};$$

$$u_{n-j-1} = q'_{n-j-1} - 1;$$

$$u_{n-j-2} = \cdots = u_0 = X \qquad 0 \leqq j \leqq n-1$$

For $j = 0$, Q and $Q1$ do not have any common digits, and for $j = n - 1$, U has no X's. By construction, $Q \leq U < Q1$.

The introduction of U would not be necessary if we could always produce its value as the last code of the ascending sequence, as in the example above. Unfortunately, this is not always the case. Consider, for instance, $Q = 0100$ and $Q1 = 0113$. The two sequences are 0101 0102 0103 and 0112 0111 0110. U, equal to $010X$, is not a white block (it actually covers Q) and, therefore, should not be generated by either sequence. The decreasing sequence, on the other hand, should produce all quaternary codes $> U$. Our algorithm, therefore, should produce the ascending sequence in the way described before, but should generate the descending sequence between $Q1$ and U.

The algorithm is formally given below as 'procedure WHITEBLOCKS.' It accepts, as input, n, i, $i1$, U and two arrays Q and $Q1$ containing the coefficients of expansions (2) and (3), respectively. It outputs all the white blocks (if any) with encoding between Q and $Q1$. The function SUM_OF_POWERS is designed to evaluate forms of type (2).

```
procedure WHITEBLOCKS (n, i, i1, Q, Q1, U);
value n, i, i1, U; reference Q, Q1;
integer n, i, i1, U; integer array Q, Q1;

(* WHITEBLOCKS generates all white blocks between Q and
Q1;
Q[0], Q[1], .. are the Q-digits; Q1[0], Q1[1], ... are the
Q1-digits;
Q1SUM is the Q1-mixed quaternary code *)

begin integer m, k, S, X, Q1SUM;
integer array LAST[0: n − 1], LAST1[0: n − 1];
Q1SUM ← SUM_OF_POWERS(Q1);

(* to generate the ascending sequence call WHITE with formal
   parameter CHOICE = 1; to generate the largest white blocks
   set formal parameter LIMIT = Q1SUM; initially LAST =
   Q; after each invocation of WHITE, LAST contains a white
   block *)

for m ← 0 step 1 until n − 1 do LAST [m] ← Q[m];
k ← i;
while SUM_OF_POWERS(LAST) < Q1SUM and k ≦ n −
1 do

(* SUM_OF_POWERS(LAST) must be evaluated here, since
   WHITE may not be called *)

  begin
  if k ≠ 0 then LAST [k − 1] ← X;
  S ← LAST [k];
  if S ≠ 3 then WHITE (k, 3 − S, 1, Q1SUM, LAST);
  k ← k + 1
  end;

(* to generate the descending sequence call WHITE with for-
   mal parameter CHOICE = −1; to generate all white blocks
   > U and < Q1SUM set formal parameter LIMIT = U;
   initially LAST1 = Q1; after each invocation of WHITE,
   LAST contains a white block *)

for m ← 0 step 1 until n − 1 do LAST1[m] ← Q1[m];
k = i1;
 while SUM_OF_POWERS(LAST1) > U and k ≦ n − 1 do
  begin
  if k ≠ 0 then LAST1 [k − 1] ← X;
  S ← LAST1[k];
  if S ≠ 0 then WHITE (k, S, −1, U, LAST1);
  k ← k + 1
  end
 end  (* of WHITEBLOCKS *);

procedure WHITE (k, SS, CHOICE, LIMIT, WHITE-
NODE);
value k, SS, CHOICE, LIMIT; reference WHITENODE;
```

```
integer k, SS, CHOICE, LIMIT; integer array WHITENODE;

begin integer s, m, SUM; Boolean FLAG;
FLAG ← true;
for s ← 1 step 1 until SS do
    if FLAG = true do
        begin
        WHITENODE [k] ← WHITENODE [k] + CHOICE;
        SUM ← SUM_OF_POWERS (WHITENODE);
(* CHOICE = 1 corresponds to the production of a white block
   with SUM < Q1SUM;
   CHOICE = −1 corresponds to the production of a white
   block with SUM > U *)
          if (CHOICE = 1 and SUM < LIMIT) or
          (CHOICE = −1 and SUM > LIMIT) then
                   for m ← n − 1 step − 1 until 0 do
                            output WHITENODE [m]
          else FLAG ← false
          end
end (* of WHITE *);

integer procedure SUM_OF_POWERS (n, Y);
value n; reference Y; integer array Y;
(* evaluation of Y[0] + Y[1]*4 +···+ Y[n − 1]*4**(n − 1)
   *)
begin integer m;
SUM_OF_POWERS ← Y[n − 1];
for m ← n − 2 step − 1 until 0 do
          SUM_OF_POWERS ← SUM_OF_POWERS* 4 + Y[m]
end (* of SUM_OF_POWERS *);
```

6. Union and Intersection of Linear Quadtrees

The most extensive discussion on union, intersection, and superposition (the latter being defined as the occlusion of white nodes by black ones) can be found in [2]. Our data structure suggests a simple procedure, based on merging the two linear quadtrees into a new sorted array. If two nodes are the same, we store only one. If one block covers the other, we store only the larger. Execution time is, of course, proportional to the total number of black nodes of the two regions. Superposition of $k > 2$ regions can be performed in parallel by applying a k-way merge. Condensation should be incorporated into the merging phase, so that superposition of k regions can be carried out in linear time. Intersection can be treated similarly.

7. Complexity Considerations

Let NP be the total number of black pixels, W the number of white nodes, and M the total number of nodes of a quadtree. N and n are the number of black nodes and the resolution parameter, respectively. Space complexity in terms of number of bits is easily evaluated. Each quaternary digit requires two bits, while the marker X requires three bits. Since no node can start with an X, the number of bits for each quaternary code is $3(n-1)+2$. A linear quadtree, therefore, can be stored in $(3(n-1)+2)N$ bits of memory. Space complexity, measured in terms of numbers of nodes, is N for linear quadtrees and M for regular quadtrees, where M can be as high as $4nN + 1$ [11]. If we assume that the six fields of a quadtree's node can be stored in three memory locations, linear quadtrees reduce the storage requirement by a factor of $N/(3M)$, with $N < M$. This means a savings of more than 66 percent of the storage required by quadtrees in all cases and a savings higher than 90 percent when M is close to the above-mentioned bound.

Worst-case time complexity of the given algorithms is also easily evaluated. Encoding a black pixel requires time proportional to n; encoding a region including condensation, requires $O(n * NP)$ time; finding an adjacent pixel and determining its color can be carried out in $O(n)$ time; refining black nodes requires $O(n * N)$ time and refining all terminal nodes can be executed in $O(n * (N + W))$ time. Finally, superposition of two regions formed of N_1 and N_2 black nodes can be carried out in time proportional to $O(N_1 + N_2)$. These time estimates are established in the presence of a sequential model of computation. However, some of the presented procedures, for example, encoding a region, could be executed much faster if parallel processing were available.

Acknowledgments. The author wishes to thank the referees for their criticism and Mr. Zale Tabakman for his excellent job of programming and testing the algorithms.

Received 8/81; revised 10/81; accepted 8/82

References
1. Dyer, C.R., Rosenfeld, A., and Samet, H. Region representation: Boundary codes from quadtrees. *Comm. ACM 23*, 3 (March 1980), 171–179.
2. Hunter, G.M. and Steiglitz, K. Operations on images using quadtrees. *IEEE Trans. on Pattern Analysis and Machine Intell. 1*, 2 (April 1979), 145–153.
3. Hunter, G.M. and Steiglitz, K. Linear transformations of pictures represented by quadtrees. *Comptr. Graphics and Image Processing 10*, 3 (July 1979), 289–296.
4. Jackins, C.L. and Tanimoto, S.L. Oct-trees and their use in representing three-dimensional objects. *Comptr. Graphics and Image Processing 14*, 3 (Nov. 1980), 249–270.
5. Jones, L. and Iyengar, S.S. Representation of a region as a forest of quad-trees. *Proc. IEEE-PRIP 81 Conference* Dallas, TX, IEEE Publ. 81 CH1595-8, (1981), 57–59.
6. Kawaguchi, E. and Endo, T. On a method of binary-picture representation and its application to data compression. *IEEE Trans. on Pattern Analysis and Machine Intell. 2*, 1 (Jan. 1980), 27–35.
7. Klinger, A. *Patterns and Search Statistics, Optimizing Methods in Statistics.* Rustagi, J.D. (Ed.) Academic Press, New York, 1971.
8. Klinger, A. and Rhodes, M.L. Organization and access of image data by areas. *IEEE Trans. on Pattern Analysis and Machine Intell. 1*, 1 (Jan. 1979), 50–60.
9. Samet, H. An algorithm for converting rasters to quadtrees. *IEEE Trans. on Pattern Analysis and Machine Intell. 3*, 1 (Jan. 1981) 93–95.
10. Samet, H. Connected component labeling using quadtrees. *JACM 28*, 3 (July 1981), 487–501.
11. Samet, H. Region representation: quadtrees from boundary codes. *Comm. ACM 23*, 3 (March 1980), 163–170.
12. Srihari, S.N. Representation of three-dimensional digital images. TR-162, Comptr. Sci. Dept. State Univ. of New York at Buffalo, July 1980.