Neighbor Finding in Quadtrees

Hanan Samet

Computer Science Department, University of Maryland
College Park, MD 20742

ABSTRACT

Image representation plays an important role in image processing applications. Recently there has been a considerable interest in the use of quadtrees. This has led to the development of algorithms for performing image processing tasks as well as converting between the quadtree and other representations. Common to these algorithms is a traversal of the tree and the performance of a given computation at each node. These computations typically require the ability to examine adacencies between neighboring nodes. Algorithms are given for determining such adjacencies in the horizontal, vertical, and diagonal directions.

Introduction

Region representation is an important aspect of image processing with numerous representations finding use. Recently, there has emerged a considerable amount of interest in the quadtree [3-8,11]. This stems primarily from its hierarchical nature which lends itself to a compact representation. It is also quite efficient for a number of traditional image processing operations such as computing perimeters [14], labeling connected components [13], finding the genus of an image [1], and computing centroids and set properties [19]. Development of algorithms to convert between the quadtree representation and other representations such as chain codes [2,10], rasters [12,18], binary arrays [11], and medial axis transforms [15,16,20] lend further support to this importance.

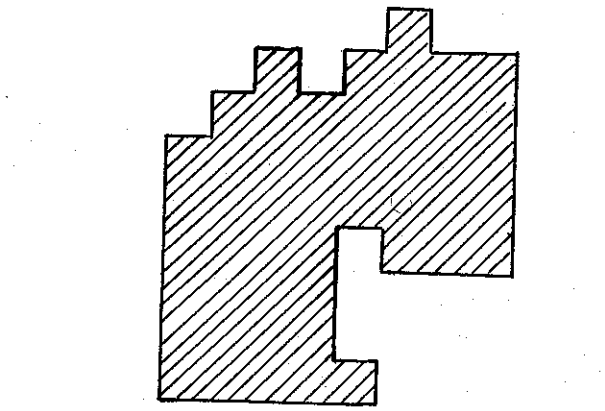
In this paper we discuss methods for moving between adjacent blocks in the quadtree. We first show how transitions are made between blocks of equal size and then generalize our results to blocks of different size where the destination block is either of larger or smaller size than the source block. Such blocks are termed neighbors. Note that the transitions that we discuss also include those along diagonal, as well as horizontal and vertical, directions. The importance of these methods lies in their being the cornerstone of many of the quadtree algorithms (e.g., [1,2,10,12-16, 18-20]), since they are basically tree traversals with a "visit" at each node. More often than not these visits involve probing a node's neighbors. The significance of our methods lies in the fact that they do not use coordinate information, knowledge of the size of the images, or storage in excess of that imposed by the nature of the quadtree data structure.

Definitions and notation

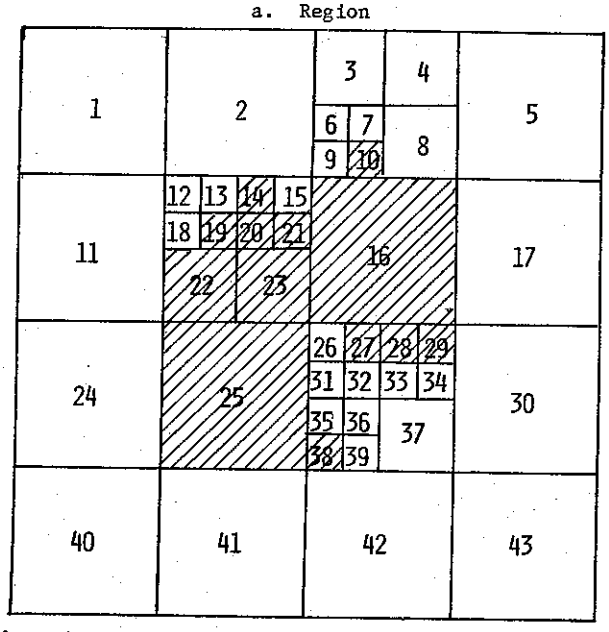
The quadtree is an approach to image representation based on the successive subdivision of the image into quadrants. It is represented by a tree of outdegree 4 in which the root represents a block and the four sons represent in order the NW, NE, SW, and SE quadrants. We assume that each node is stored as a record containing six fields. The first five fields contain pointers to the node's father and its four sons which correspond to the four quadrants. If P is a node and I is a quadrant, then these fields are referenced as FATHER(P) and SON(P,I) respectively. We can determine the specific quadrant in which a node, say P, lies relative to its father by use of the function SONTYPE(P) which has a value of I if SON(FATHER(P),I) = P. The sixth field, NODETYPE, describes the contents of the block of the image which the node represents-i.e., WHITE if the block contains no 1's, BLACK if the block contains only 1's, and GRAY if it contains 0's and 1's. Alternatively, BLACK and WHITE are terminal nodes, while GRAY nodes are non-terminal nodes. For example, Figure 1b is a block decomposition of the region in Figure 1a while Figure 1c is the corresponding quadtree.

Let the four sides of a node's block be called its N, E, S, and W sides. They are also termed its boundaries and at times we speak of them as if they are directions. We define the following predicates and functions to aid in the expression of operations involving a block's quadrants and its boundaries. ADJ(B,I) is true if and only if quadrant I is adjacent to boundary B of the node's block, e.g., ADJ(W, SW) is true. REFLECT(B,I) yields the SONTYPE value of the block of equal size that is adjacent to side B of a block having SONTYPE value I, e.g., REFLECT(N, SW) = NW. COMMONSIDE(Q1,Q2) indicates the boundary of the block containing quadrants Q1 and Q2 that is common to them; e.g., COMMONSIDE(SW,NW) = W. If Q1 and Q2 are not adjacent brother quadrants (e.g., NE and SW) or if Q1 and Q2 are the same, then the value of COMMONSIDE is undefined. OPQUAD(Q) is the quadrant which does not share a block boundary

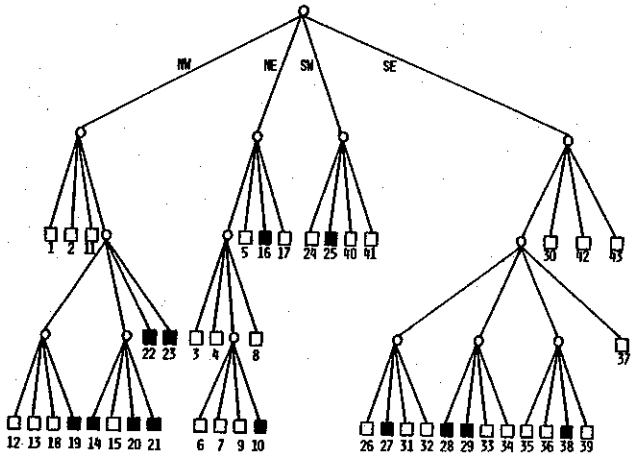
CH1595-8/81/0000/0068$00.75 © 1981 IEEE



a. Region



b. Block decomposition of the region in (a).



c. Quadtree representation of the blocks in (b).

Figure 1. A region, its maximal blocks, and the corresponding quadtree. Blocks in the region are shaded, background blocks are blank.

ADJ(S,Q)	NW	NE	SW	SE
N	T	T	F	F
E	F	T	F	T
S	F	F	T	T
W	T	F	T	F

Table 1. ADJ(S,Q)

REFLECT(S,Q)	NW	NE	SW	SE
N	SW	SE	NW	NE
E	NE	NW	SE	SW
S	SW	SE	NW	NE
W	NE	NW	SE	SW

Table 2. REFLECT(S,Q)

Q	OPQUAD(Q)
NW	SE
NE	SW
SW	NE
SE	NW

Table 3. OPQUAD(Q)

COMMONSIDE(Q1,Q2)	NW	NE	SW	SE
NW	Ω	N	W	Ω
NE	N	Ω	Ω	E
SW	W	Ω	Ω	S
SE	Ω	E	S	Ω

Table 4. COMMONSIDE(Q1,Q2)

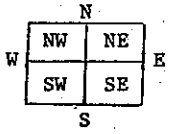


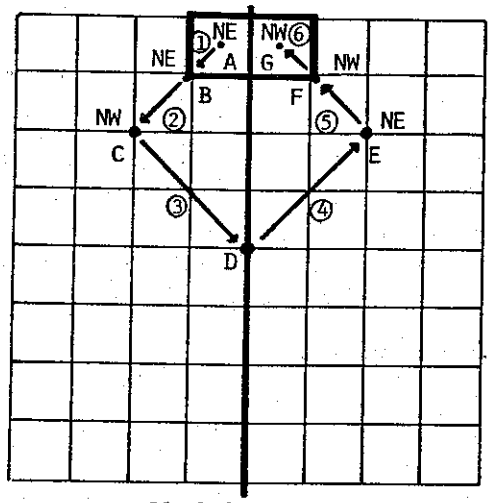
Figure 2. Relationship between a block's four quadrants and its boundaries.

with quadrant Q; e.g., OPQUAD(SW) = NE. Figure 2 shows the relationship between the quadrants of a node and its boundaries while Tables 1-4 contain the definitions of the ADJ, REFLECT, OPQUAD, and COMMONSIDE relationships respectively. Ω corresponds to an undefined value.

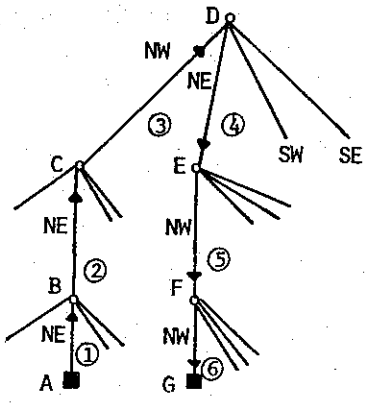
For a quadtree corresponding to a 2^n by 2^n array we say that the root is at level n, and that a node at level i is at a distance of n-i from the root of the tree. In other words, for a node at level i, we must ascend n-i FATHER links to reach the root of the tree. Note that the farthest node from the root of the tree is at a level $\geq \emptyset$. A node at level $\emptyset$ corresponds to a single pixel in the image. Also, we say that a node is of size 2^s if it is found at level s in the tree.

Neighbor finding algorithms

Given a node corresponding to a specific block in the image, its neighbor of equal size in the horizontal or vertical direction is determined by locating a common ancestor. Next, we retrace the path while making mirror image moves about an axis formed by the common boundary between the blocks associated with the two nodes. The common ancestor is simple to determine--e.g., to find an eastern neighbor, the common ancestor is the first ancestor node which is reached via its NW or SW son. For example, the eastern neighbor of node A in Figure 3a is G. It is located by ascending the tree until the common ancestor, D, is found. This requires going through a NE link to reach B, a NE link to reach C, and a NW link to reach D. Node

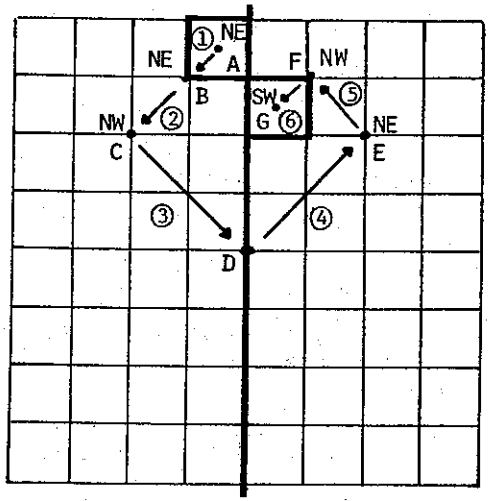


a. Block decomposition

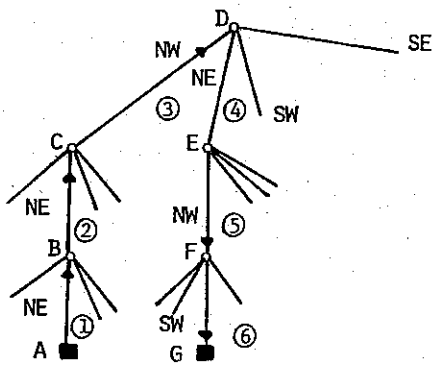


b. Tree representation

Figure 3. Process of locating the eastern neighbor of Node A (i.e., G).



a. Block decomposition



b. Tree representation

Figure 4. Process of locating the SE neighbor of Node A (i.e., G).

G is now reached by backtracking along the previous path with the appropriate mirror image moves. This requires descending a NE link to reach E, a NW link to reach F, and a NW link to reach G. Figures 3a and 3b show how the eastern neighbor of node A is located. The algorithm for locating an equal sized neighbor in a given horizontal or vertical direction is given below using a variant of ALGOL 60 [9]. Note that we assume that the neighbor in the specified direction does indeed exist (i.e., we are not on the border of the image).

```
node procedure EQUAL_ADJ_NEIGHBOR(P,D);
/* Locate an equal-sized neighbor of node P in
   horizontal or vertical direction D */
begin
  value node P;
  value direction D;
  return (SON(if ADJ(D,SONTYPE(P)) then
                EQUAL_ADJ_NEIGHBOR(FATHER(P),D)
              else FATHER(P),
              REFLECT(D,SONTYPE(P))));
end;
```

Finding a node's neighbor in the diagonal direction (i.e., its corresponding block touches the given node's block at a corner) is more complex. Given a node corresponding to a specific block in the image, its neighbor of equal size in a diagonal direction is determined by a three step process. First, we locate the given node's nearest ancestor who is also adjacent (horizontally or vertically) to an ancestor of the sought neighbor. Next, we make use of EQUAL_ADJ_NEIGHBOR to access the ancestor of the sought neighbor in the direction of the adjacency. Finally, we retrace the remainder of the path while making directly opposite moves (i.e., 180° opposite so that a NW move becomes a SE move). The nearest ancestor of the first step is the first ancestor which is not reached by a link equal to the direction of the desired neighbor--e.g., to find a SE neighbor, the nearest such ancestor is the first ancestor node which is not reached via its SE son. For example, the SE neighbor of node A in Figure 4a is G. It

is located by ascending the tree until the nearest neighbor, B, which is also adjacent horizontally (in this case) to an ancestor of G, i.e., F, is found. This requires going through a NE link to reach B. Node F is now reached by applying EQUAL_ADJ_NEIGHBOR in the direction of the adjacency (i.e., east). This forces us to go through a NE link to reach C and a NW link to reach D. Backtracking results in descending a NW link to reach E and a NW link to reach F. Finally, we backtrack along the remainder of the path making 180° moves--i.e., we descend a SW link to reach G. Figures 4a and 4b show how the SE neighbor of node A is located. Note that, at times, EQUAL_ADJ_NEIGHBOR may not need to be applied. This is the case when the nearest ancestor of the first step is reached by a link equal to the direction opposite that of the desired neighbor (e.g., the SW neighbor of node 16 is 25 with the nearest ancestor of step 1 in Figure 1, being node A). The algorithm for locating an equal size neighbor in a given diagonal direction is given below. Once again, we assume that the neighbor in the specified direction does indeed exist (i.e., we are not on the border of the image).

```
node procedure EQUAL_CORNER_NEIGHBOR(P,C);
/ *Locate an equal-sized neighbor of node P in
   the direction of quadrant C */
begin
  value node P
  value quadrant C;
  return(SON(if SONTYPE(P)=OPQUAD(C) then FATHER(P)
             else if SONTYPE(P)=C then
                EQUAL_CORNER_NEIGHBOR(FATHER(P),C)
             else EQUAL_ADJ_NEIGHBOR(
                   FATHER(P),
                   COMMONSIDE(SONTYPE(P),C)),
             OPQUAD(SONTYPE(P))));
end;
```

It is often the case that neighbors are of different sizes. In such a case, we say that we want the neighboring terminal nodes having equal or greater size (e.g., the eastern neighbor of node 23 in Figure 1, is 16). If such a node does not exist, then we return a GRAY node of equal size if possible (e.g., the northern neighbor of node 23 in Figure 1 is J). Otherwise the node is adjacent to the border of the image (not the region) and NULL is returned since there is no neighbor in the specified direction (e.g., the northern neighbor of node 2 in Figure 1 is NULL). When a node does not have a neighboring terminal node of equal or greater size, returning a GRAY node of equal size is reasonable because the given node whose neighbor is being sought has more than one neighboring terminal node in the given direction. The algorithms for locating neighbors of equal or greater size in horizontal and vertical directions as well as diagonal directions are given below using procedures GTEQUAL_ADJ_NEIGHBOR and GTEQUAL_CORNER_NEIGHBOR respectively. Note that a neighbor in a diagonal direction, say C, will not always abut against corner C of the node whose neighbor is sought (e.g., node 16 is a non-abutting NE neighbor of node 23 in Figure 1).

```
node procedure  GTEQUAL_ADJ_NEIGHBOR(P,D):
/* Locate a neighbor of node P in horizontal or
   vertical direction D.  If such a node does not
   exist, then return NULL */
begin
  value node P;
  value direction D;
  node Q;
  if not NULL(FATHER(P)) and ADJ(D,SONTYPE(P)) then
    /* Find a common ancestor */
    Q←GTEQUAL_ADJ_NEIGHBOR(FATHER(P),D)
  else Q←FATHER(P);
  /* Follow the reflected path to locate the
     neighbor */
  return (if not NULL(Q) and GRAY(Q) then
             SON(Q, REFLECT(D,SONTYPE(P)))
          else Q);
end;

node procedure GTEQUAL_CORNER_NEIGHBOR(P,C);
/* Locate a neighbor of node P in the direction
   of quadrant C.  If such a node does not exist,
   then return NULL */
begin
  value node P;
  value quadrant C;
  node Q;
  if not NULL(FATHER(P)) and
     SONTYPE(P)≠OPQUAD(C) then
    /* Find a common  ancestor */
    if SONTYPE(P)=C then
       Q←GTEQUAL_CORNER_NEIGHBOR(FATHER(P),C)
    else Q←GTEQUAL_ADJ_NEIGHBOR(
                FATHER(P),
                COMMONSIDE(SONTYPE(P),C))
  else Q←FATHER(P);
  /* Follow opposite path to locate the neighbor */
  return (if not NULL(Q) and GRAY(Q) then
             SON(Q,OPQUAD(SONTYPE(P)))
          else Q);
end;
```

If neighbors are of different sizes, we may wish to know the size of the adjacent or abutting neighbor. In such a case, we want our neighbor finding algorithms to return both a pointer to the neighboring node and a value from which the node's size can be easily computed. This is relatively straightforward when we know the level in the tree at which is found the node whose neighbor is being sought. In fact, such an algorithm need only increment the level counter by 1 for each link that is ascended while locating the common ancestor, and then decrement the level counter by 1 for each link that is descended while locating the appropriate neighbor. The algorithms for locating neighbors of equal or greater size, with their corresponding level positions, in horizontal and vertical directions as well as diagonal directions, are given below using procedures GTEQUAL_ADJ_NEIGHBOR2 and GTEQUAL_CORNER_NEIGHBOR2 respectively. Note the use of reference parameters to transmit and return results. An alternative is to define a record of type

'block' having two fields of type 'node' and 'integer' whose values are a pointer to the neighboring node and its level respectively.

```
procedure GTEQUAL_ADJ_NEIGHBOR2(P,D,Q,L);
/* Return in Q the neighbor of node P in horizontal or vertical direction D. L denotes
   the level of the tree at which node P is
   initially found and the level of the tree
   at which node Q is ultimately found. If such
   a node does not exist, then return NULL */
begin
  value node P;
  value direction D;
  reference node Q;
  reference integer L;
  L←L+1;
  if not NULL(FATHER(P)) and
     ADJ(D,SONTYPE(P)) then
   /* Find a common ancestor */
   GTEQUAL_ADJ_NEIGHBOR2(FATHER(P),D,Q,L)
  else Q←FATHER(P);
  /* Follow the reflected path to locate the
     neighbor */
  if not NULL(Q) and GRAY(Q) then
    begin
      Q←SON(Q,REFLECT(D,SONTYPE(P)));
      L←L-1;
    end;
end;
```

```
procedure GTEQUAL_CORNER_NEIGHBOR2(P,C,Q,L);
/* Return in Q the neighbor of node P in the direction of quadrant C. L denotes the level of
   the tree at which node P is initially found
   and the level of the tree at which node Q is
   ultimately found. If such a node does not
   exist, then return NULL */
begin
  value node P;
  value quadrant C;
  reference node Q;
  reference integer L;
  L←L+1;
  if not NULL(FATHER(P)) and
     SONTYPE(P)≠OPQUAD(C) then
   /* Find a common ancestor */
   if SONTYPE(P)=C then
      GTEQUAL_CORNER_NEIGHBOR2(FATHER(P),C,Q,L)
   else GTEQUAL_ADJ_NEIGHBOR2(
              FATHER(P),
              COMMONSIDE(SONTYPE(P),C),Q,L)
  else Q←FATHER(P);
  /* Follow the opposite path to locate the neighbor */
  if not NULL(Q) and GRAY(Q) then
    begin
      Q←SON(Q,OPQUAD(SONTYPE(P)));
      L←L-1;
    end;
end;
```

At times we may wish to locate an adjacent horizontal or vertical neighbor regardless of its size. In such a case, we also specify a corner of the block corresponding to the node whose neighbor is being sought. The neighboring node must be adjacent to this corner (e.g., node 21 is the northern neighbor of node 23 which is adjacent to the NE corner of node 23). The algorithm for computing such a neighbor is given below by procedure CORNER_ADJ_NEIGHBOR which makes use of GTEQUAL_ADJ_NEIGHBOR.

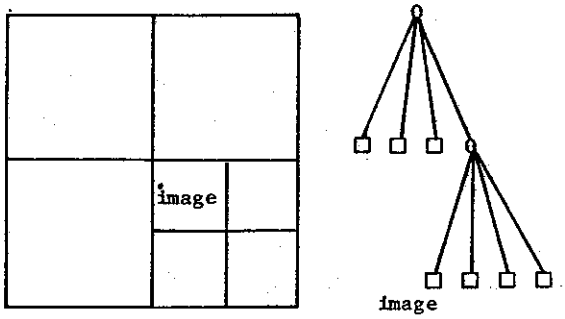
```
node procedure CORNER_ADJ_NEIGHBOR(P,D,C);
/* Locate a neighbor of node P in horizontal or
   vertical direction D which is adjacent to
   corner C of node P. If such a node does not
   exist, then return NULL */
begin
  value node P;
  value direction D;
  value quadrant C;
  P←GTEQUAL_ADJ_NEIGHBOR(P,D);
  while GRAY(P) do P←SON(P,REFLECT(D,C));
       /* Descend to the desired corner */
  return (P);
end;
```

Similarily, in the case of a diagonal neighbor, we may also wish to locate the neighbor in the given direction regardless of its size (e.g., node 20 is a NE neighbor of node 22 in Figure 1 which is smaller in size). The algorithm for locating an arbitrary-sized diagonal neighbor is given below by procedure CORNER_CORNER_NEIGHBOR which makes use of GTEQUAL_CORNER_NEIGHBOR.

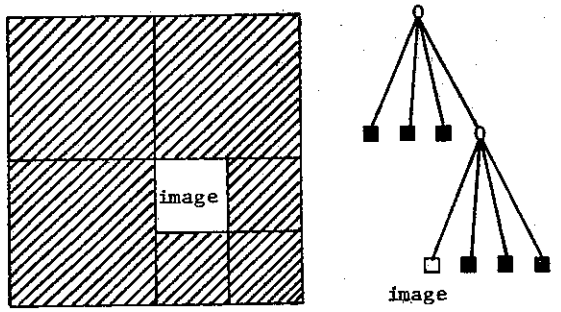
```
node procedure CORNER_CORNER_NEIGHBOR(P,C);
/* Locate a neighbor of node P in the direction of
   quadrant C which abuts against corner C of node
   P. If such a node does not exist, then return
   NULL */
begin
  value node P;
  value quadrant C;
  node Q;
  Q←GTEQUAL_CORNER_NEIGHBOR(P,C);
  while GRAY(Q) do Q←SON(Q,OPQUAD(C));
       /* Descend to the desired corner */
  return (Q);
end;
```

It should be clear that procedures similar to CORNER_ADJ_NEIGHBOR and CORNER_CORNER_NEIGHBOR can be constructed that also return the level at which the desired neighboring node is found. This will not be done here.

The procedures outlined about always return NULL when a neighbor in a specified direction does not exist. This situation arises whenever the node whose neighbor is sought is adjacent to the border of the image along the specified direction. At times the NULL pointer is not convenient. Instead, we could assume that the image is surrounded by WHITE blocks as in Figure 5a or by BLACK blocks as in Figure 5b. The choice of WHITE or BLACK for the surrounding blocks depends on the particular application. For example, we use WHITE in the case of the quadtree to boundary code conversion algorithm [2] while BLACK is more useful in the case of the computation of distance [15] and the construction of a Quadtree Medial Axis Transform [16].



a. Image surrounded by WHITE blocks.



b. Image surrounded by BLACK blocks.

Figure 5. Technique to avoid lacking a neighbor in a given direction.

Concluding remarks

The above techniques should be contrasted with other methods of locating neighbors [3-5,8]. In [8], a method is described for moving between adjacent blocks of equal size that are brothers (i.e., have the same father node). This method does not make use of the tree structure; instead coordinate information and knowledge of the size of the image are used to locate a neighboring brother in a given horizontal or vertical direction. This is accomplished by a number of primitives termed MOVE UP, MOVE DOWN, MOVE RIGHT, and MOVE LEFT. Transitions to non-brother neighboring blocks require the use of approximations through the use of primitives named MORE, LESS, and GAMMA. The disadvantages of these methods is that they require computation (rather than chasing links) and are clumsy when adjacent blocks are not brothers as well as when they are of different sizes than the block whose neighbor is sought.

In [3-5] a number of algorithms are described for operating on images using quadtrees. Transitions between neighboring blocks are made by use of explicit links from a node to its adjacent neighbors in the horizontal and vertical directions. This is achieved through the use of adjacency trees, "ropes", and "nets". An adjacency tree exists whenever a leaf node, say X, has a GRAY neighbor, say Y, of equal size. In such a

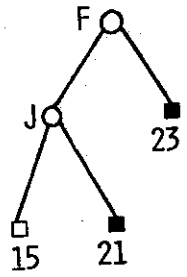


Figure 6. Adjacency tree for the western neighbor of node 16 in Figure 1.

case, the adjacency tree of X is a binary tree rooted at Y whose nodes consist of all sons of Y (BLACK, WHITE, and GRAY) that are adjacent to X. For example, for node 16 in Figure 1, the western neighbor is GRAY node F with an adjacency tree as shown in Figure 6. A rope is a link between adjacent nodes of equal size at least one of which is a leaf node. For example, in Figure 1, there exists a rope between node 16 and nodes G, 17, H, and F. Similarly, there exists a rope between node 37 and nodes M and N; however, there does not exist a rope between node L and nodes M and N.

The algorithm for finding a neighbor using a roped quadtree is quite simple. We want a neighbor, say Y, on a given side, say D, of a block, say X. If there is a rope from X on side D, then it leads to the desired neighbor. If no such rope exists, then the desired neighbor must be larger. In such a case, we ascend the tree until encountering a node having a rope on side D which leads to the desired neighbor. In effect, we have ascended the adjacency tree of Y. For example, to find the eastern neighbor of node 21 in Figure 1, we ascend through node J to node F which has a rope along its eastern side leading to node 16.

At times it is not convenient to ascend nodes searching for ropes. A data structure named a net is used in [3-5] to obviate this step by linking all leaf nodes to their neighbors regardless of their relative size. Thus in the previous example there would be a direct link between nodes 21 and 16 along the eastern side of node 21. The advantage of ropes and nets is that the number of links that must be traversed is reduced (for a detailed comparative analysis of execution times see the expanded version of this paper in [17]. However, the disadvantage is that the storage requirements are considerably increased since many additional links are necessary. In contrast, our methods are implemented by algorithms that make use of the existing structure of the tree--i.e., four links from a non-leaf node to its sons, and a link from a non-root node to its father.

References

1. C.R. Dyer, Computing and the Euler number of an image from its quadtree, Computer Graphics and Image Processing 13, 1980, 270-276.

2. C.R. Dyer, A. Rosenfeld, and H. Samet, Region representation: boundary codes from quadtrees, Communications of the ACM, March 1980, 171-179.

3. G.M. Hunter, Efficient computation and data structures for graphics, Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, New Jersey, 1978.

4. G.M. Hunter and K. Steiglitz, Operations on images using quadtrees, IEEE Transactions on Pattern Analysis and Machine Intelligence 1, 1979, 145-153.

5. G.M. Hunter and K. Steiglitz, Linear transformation of pictures represented by quadtrees, Computer Graphics and Image Processing 10, 1979, 289-296.

6. A. Klinger, Patterns and Search Statistics in Optimizing Methods in Statistics, J.S. Rustagi, Ed., Academic Press, New York, 1971.

7. A. Klinger and C.R. Dyer, Experiments in picture representation using regular decomposition, Computer Graphics and Image Processing 5, 1976, 68-105.

8. A. Klinger and M.L. Rhodes, Organization and access of image data by areas, IEEE Transactions on Pattern Analysis and Machine Intelligence 1, 1979, 50-60.

9. P. Naur (Ed.), Revised report on the algorithmic language ALGOL 60, Communications of the ACM 3, 1960, 299-314.

10. H. Samet, Region representation: quadtrees from boundary codes, Communications of the ACM, March 1980, 163-170.

11. H. Samet, Region representation: quadtrees from binary arrays, Computer Graphics and Image Processing, May 1980, 88-93.

12. H. Samet, An algorithm for converting from rasters to quadtrees, IEEE Transactions on Pattern Analysis and Machine Intelligence, January 1981, 93-95.

13. H. Samet, Connected component labeling using quadtrees, Journal of the ACM, July 1981.

14. H. Samet, Computing perimeters of images represented by quadtrees, Computer Science TR-755, University of Maryland, College Park, Maryland, April 1979, to appear in IEEE Transactions on Pattern Analysis and Machine Intelligence.

15. H. Samet, A distance transform for images represented by quadtrees, Computer Science TR-780, University of Maryland, College Park, Maryland, July 1979, to appear in IEEE Transactions on Pattern Analysis and Machine Intelligence.

16. H. Samet, A quadtree medial axis transform, Computer Science TR-803, University of Maryland, College Park, Maryland, August 1979.

17. H. Samet, Neighbor finding techniques for images represented by quadtrees, Computer Science TR-857, University of Maryland, College Park, Maryland, January 1980, to appear in Computer Grahpics and Image Processing.

18. H. Samet, Algorithms for the conversion of quadtrees to rasters, Computer Science TR-979, University of Maryland, College Park, Maryland, November 1980.

19. M. Shneier, Linear-time calculations of geometric properties using quadtrees, Computer Science TR-770, University of Maryland, College Park, Maryland, May 1979, to appear in Computer Graphics and Image Processing.

20. M. Shneier, Path-length distances for quadtrees, Information Sciences 23, February 1981, 45-67.