FAST REGION EXPANSION FOR QUADTREES*

Chuan-Heng Ang
Hanan Samet

Computer Science Department and
Institute of Advanced Computer Studies and
Center for Automation Research
University of Maryland
College Park, MD 20742

Clifford A. Shaffer

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

Abstract

A one-pass algorithm is presented to perform region expansion in images that are represented by quadtrees. The algorithm changes to BLACK those WHITE pixels within a specified distance of any BLACK node in the image. This process is sometimes referred to as *polygon expansion* or *image dilation*. The algorithm is especially well-suited to images that are represented using a pointer-less representation such as the linear quadtree. The algorithm yields a significant improvement over previous approaches by (1) reducing the number of BLACK nodes that must be considered for expansion, and (2) reducing the number of nodes that must be inserted as a result of the expansion. This is achieved by the introduction of the concepts of a merging cluster and a vertex set. Empirical tests show that the execution time of this algorithm generally decreases as the radius of expansion increases, whereas for previous approaches the execution time generally increases with the radius of expansion. The algorithm is an important component of a graphics editor for applications in cartography, computer-aided design, and robotics.

*The support of the National Science Foundation under Grant DCR-86-05557 is gratefully acknowledged.

1. Introduction

This paper addresses the efficient computation of the region expansion operation. This operation finds use in a number of applications related to graphics editing. It is useful in computer-aided design when we want to find all objects near a cursor or near a particular set of objects. It can also be used to simplify the process of planning a collision-free path for a robot in a two-dimensional environment consisting of obstacles. In this case, a finite-width robot is treated as a point and the obstacles are expanded by an amount equal to the size of the robot. This paper primarily aims at computer cartography applications where it is desirable to provide graphic answers to queries such as "Find all wheatfields within five miles of the floodplain". Such an answer is computed by expanding the floodplain region in the image and intersecting the result with another image that represents the wheatfields.

The region expansion task is also known as *polygon expansion* and *image dilation*. In this paper we refer to it as the *WITHIN* function. From an implementation standpoint, given a binary image M represented as an array, WITHIN generates a new image which is BLACK at all pixels within a specified distance of the BLACK regions of M. The BLACK pixels of M correspond to those pixels which are initially within the regions of interest, while those pixels outside of such regions are defined as WHITE. In the case of a multi-colored image, we say that the regions of interest are those that contain the non-WHITE pixels. In this case, the result of the WITHIN function is that all WHITE pixels within a specified distance of a non-WHITE pixel are set to BLACK.

Region expansion is often expensive since a large image requires a great deal of computation as many pixels must be examined. Of course, the cost of this examination process can be greatly reduced when the objects in the image are very well-defined - e.g., consisting of primitive instances of known shapes. However, we are interested in images such as maps where the shapes of the regions are poorly defined. The algorithm presented here executes this function on an image represented by a region quadtree (Klinger, 1971, Samet, 1984a). The motivation behind the use of the region quadtree as an image representation is a desire to take advantage of the homogeneity of the image. If such homogeneity exists, then the space requirements can be reduced substantially by aggregating similarly colored pixels into blocks. In particular, the quadtree has the property that it acts as a dimension reducing device. For example, for a simple polygon the storage requirements of its region quadtree are proportional to its perimeter (Hunter, 1979), whereas the storage requirements of its array representation are proportional to its area. Even more important, the aggregation leads to a reduction in the execution time of many primitive functions (e.g., set operations (Hunter, 1979)). In particular, algorithms that use the quadtree representation have an execution time that is proportional to the number of blocks in the image rather than to their individual sizes (e.g., connected component labeling (Samet, 1981)).

Performing region expansion with a quadtree representation is not so simple. We can always simulate the algorithm for an array representation of the image by processing each pixel in order and then rebuilding the quadtree. However, this is expensive as it requires that every pixel be visited twice. Instead, our goal is to reduce the number of node operations so that the algorithm takes advantage of the aggregation in the image.

Nevertheless, the amount of time necessary to build a quadtree from the array representation of the expanded image is a good yardstick for measuring the efficiency of any algorithm that is proposed since at worst the quadtree could be converted to an array, the array WITHIN operation performed, and the quadtree rebuilt all in time proportional to the array-to-quadtree conversion step. Thus one of our goals is that as the radius of expansion gets sufficiently large, the execution time of any reasonable algorithm should start decreasing.

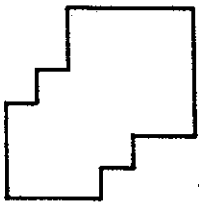
Section 2 briefly reviews the definitions of a quadtree and also the implementation that we used to test our algorithms. Section 3 briefly describes three alternative prior implementations of the WITHIN function, and outlines their strengths and weaknesses. Section 4 presents a new algorithm that overcomes the shortcomings of the methods described in Section 3. Section 5 analyzes the execution time of the new algorithm and discusses how it is confirmed by empirical tests. Section 6 reviews the differences between the algorithms.

2. A Linear Quadtree Implementation

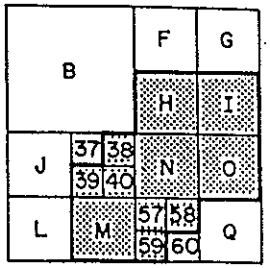
The region quadtree decomposes an image into homogeneous blocks. If the image is all one color, it is represented by a single block. If not, the image is decomposed into quadrants, subquadrants, ..., until each block is homogeneous. In order to simplify the presentation, unless noted otherwise, we assume that the image is binary. Figure 1 illustrates the region quadtree. The region of Figure 1a is represented by a binary array in Figure 1b. The resulting quadtree block decomposition is shown in Figure 1c, with the tree structure represented in Figure 1d. When a quadtree is represented by means of such a tree structure, it is referred to as a *pointer-based quadtree*. Although we have defined a quadtree representation only for region data, it can be adapted easily to deal with other types of data such as points and lines (e.g., (Nelson, 1986, Samet, 1985b)).

Often the volume of data is so high that it is preferable to store the quadtree in disk files. In such a case, a pointer-based representation may require many disk pages to be accessed. Thus alternative representations such as the linear quadtree (Gargantini, 1982, Abel, 1983) are used. The linear quadtree represents an image as a collection of the leaf nodes that comprise it. Each leaf node is represented by its locational code which corresponds to a sequence of directional codes that locate the leaf along a path from the root of the tree. Assuming that the origin of the coordinate system of the image is located at its upper left corner, then the locational code of a node is the same as the result of interleaving the bits that comprise the x, y coordinates of the upper left corner of the node. In addition, the depth of the node relative to the root must also be recorded. The collection of nodes making up the linear quadtree is usually stored as a list sorted in increasing order of the locational codes. Such an ordering is useful because it is the order in which the leaf nodes of the quadtree are visited by a depth-first traversal of the quadtree. This representation is employed in the QUILT system (Shaffer, 1987c) which was used to test the algorithms described in this paper.

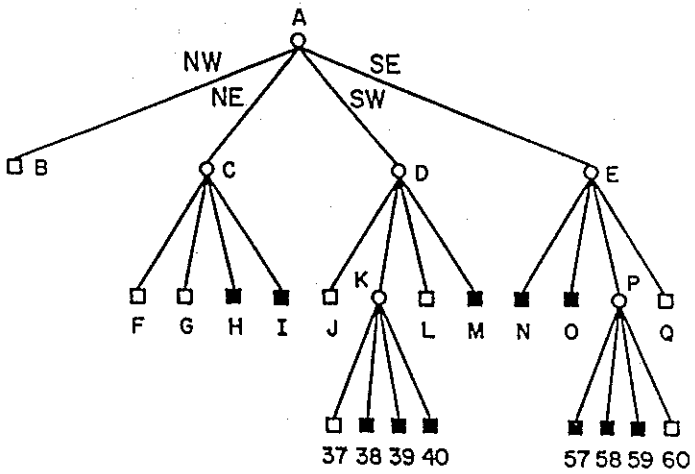
The algorithms that we describe can be used (with appropriate modifications) in both pointer-based and linear quadtrees. However, they are primarily designed for use



(a) Region



(b) Block decomposition of the region in (a). Blocks in the region are shaded.



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

Figure 1. A region, its maximal blocks, and the corresponding quadtree.

with the linear quadtree. This will not make a big difference in the algorithms; however, it does mean that we can discuss the algorithms in terms of node searches and insertions into a sorted node list, rather than lower level tree manipulation functions.

All versions of the WITHIN algorithm use the *Chessboard distance metric* (Rosenfeld, 1982). It defines the distance between points (x_1,y_1) and (x_2,y_2) as $\mathrm{MAX}(|x_1-x_2|,|y_1-y_2|)$.

All empirical tests were performed by implementing the different algorithms in the QUILT system, a quadtree-based cartographic information system and cartographic editor. Three 512×512 images, named Center, ACC, and Pebble, were used to compare the execution time of the four algorithms. These images are shown in Figures 2a, 2b, and 2c. Center is a map of the 100 year flood plain in the Russian River Valley. ACC is a single landuse class from a landuse class map of the same area. Pebble is an image of pebbles. They contain 4693, 3253, and 44950 leaf nodes respectively. For Center and ACC, expansions of 8 pixels are also shown. The algorithms are denoted as WITHINi ($1 \le i \le 4$). Figures 3a, 3b, and 3c show the execution times for these algorithms on the three images using radius values that range from 1 to 32. They were executed on a VAX11/785 running the 4.3BSD version of UNIX. In order to illustrate the relative variation in the performance of the algorithms, we plot the natural logarithms of the execution times.

3. Three Polygon Expansion Algorithms

The simplest region expansion algorithm, termed *WITHIN1* (Samet, 1984c), visits each node of the quadtree. Each BLACK node, say B, is expanded by R units and the new square (of width $\mathrm{WIDTH}(B)+2\cdot R$) is decomposed into quadtree blocks and inserted into the output quadtree. The execution time of WITHIN1 increases directly with R since the number of blocks in a quadtree is proportional to the total perimeter of the regions that comprise it (Hunter, 1978). Although the algorithm that we have implemented aggregates the resulting blocks before inserting them into the output quadtree, WITHIN1 still requires many duplicate insertions and subsequent mergings of BLACK nodes. Note that the execution times for even values of R are generally smaller than those for odd values due to the effects of node aggregation which reduces the number of blocks inserted into the output quadtree. In other words, a quadtree node expanded by an even number of pixels R can be represented with fewer quadtree blocks than one expanded by $R+1$ pixels.

A second algorithm, termed *WITHIN2* (Shaffer, 1987b), tries to avoid the excessive insertion and merging required by WITHIN1 by focusing the work on the WHITE nodes of the quadtree instead of the BLACK nodes. Again, assume that R is the radius of expansion. WHITE nodes of width less than or equal to $(R+1)/2$, as well as their brothers, are inserted into the output quadtree as BLACK nodes. WHITE nodes of width greater than $(R+1)/2$ have their distance from the closest BLACK node determined by use of a modified Chessboard distance transform (Samet, 1982). Those portions of these large WHITE nodes that lie within radius R of a nearby BLACK node are output as BLACK nodes. The problem with this approach is that many nodes of the

(a)

(b)

(c)

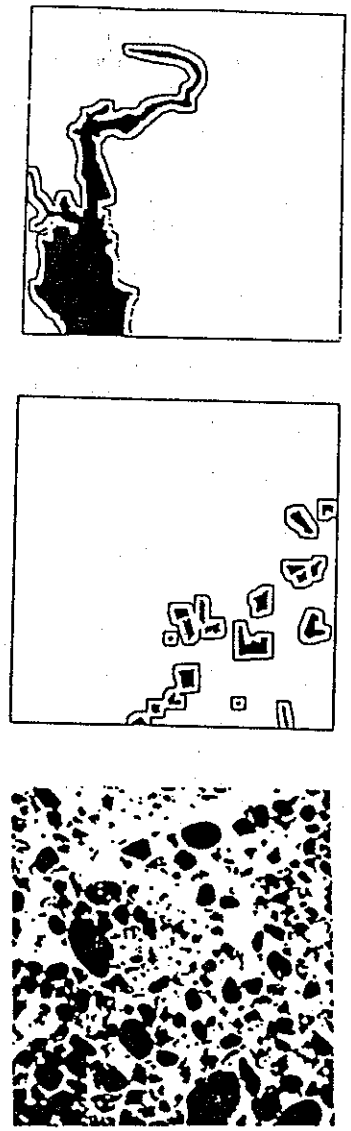


Figure 2. Three images: (a) Center, (b) ACC, (c) Pebble.

(a)

(b)

(c)

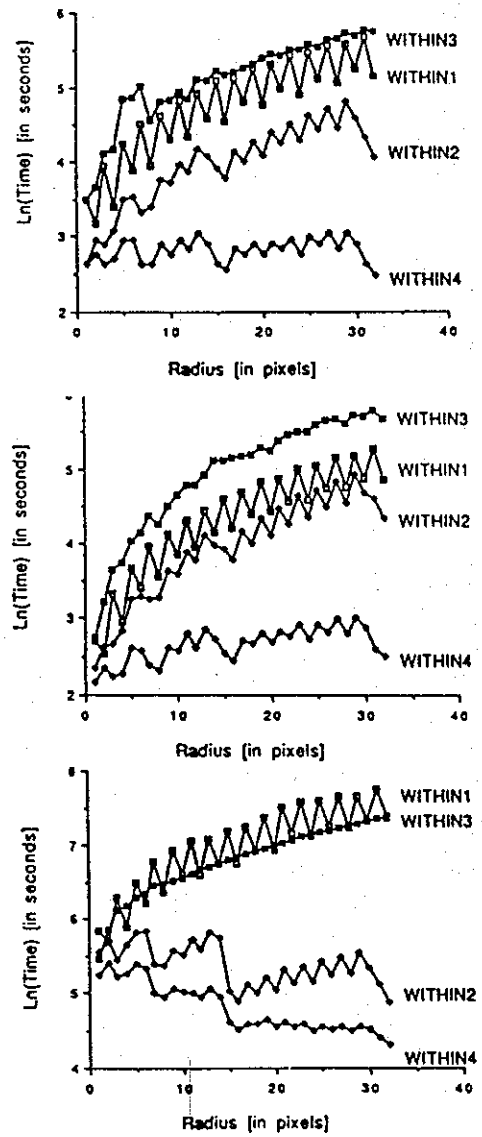


Figure 3. Execution times: (a) Center, (b) ACC, (c) Pebble.

input quadtree will be visited more than once while finding the distance from large WHITE nodes to the nearby BLACK nodes. In addition, there are many redundant node insertions. Like WITHIN1, the execution times of WITHIN2 for even values of R are generally lower than those for the adjacent odd values of R because of the effects of node aggregation.

A third algorithm, termed *WITHIN3* (Mason, 1987), uses an active border data structure to facilitate two passes over the nodes in the linear quadtree's node list. The goal is to avoid making redundant insertions. The first pass processes the node list in increasing order of locational codes. For each WHITE node, say W, it converts to BLACK all portions of W that are within a distance R of a BLACK node that has been encountered at a prior position in the list. The second pass is analogous except that the list is processed in reverse order. This approach is similar to that used to execute the WITHIN function on an array representation. However, the quadtree implementation requires a substantial amount of computation to determine and maintain the appropriate information about previously encountered BLACK nodes, and to split the WHITE nodes. Moreover, its execution time increases directly with the radius of expansion since the number of previously encountered BLACK nodes that must be examined increases.

4. The New Region Expansion Algorithm

Our new algorithm, termed WITHIN4, traverses the input quadtree in preorder and writes the result of the expansion to an output quadtree. It takes no action for large WHITE nodes. For large BLACK nodes and clusters of small nodes, the algorithm adjusts the vertex set (explained below) and performs a node expansion only when enough information has been collected.

WITHIN4's design stems from the observation that the execution times of algorithms that operate on linear quadtrees are dominated by the number of nodes that are inspected and inserted - in other words, by the I/O time required to locate and update the nodes in the node list. To improve the performance of such algorithms, we would like to reduce both the number of nodes requiring inspection (e.g., in WITHIN3 and in the distance transform computation step of WITHIN2), and the number of node insertions (the dominant factor in WITHIN1, and also important in WITHIN2). WITHIN4 achieves these goals by 1) reducing the number of input BLACK nodes that must be considered for expansion, and 2) reducing the number of output BLACK nodes that must be inserted as a result of the expansion. These reductions yield an algorithm whose execution time is relatively unaffected by the magnitude of the radius of expansion.

4.1. Reduction of the Number of Nodes to be Expanded

The keys to our new algorithm are the concepts of a *merging cluster* and a *vertex set*. These concepts allow us to consider a collection of BLACK nodes for expansion instead of expanding each BLACK node individually.

Given an input quadtree, Q, and a radius of expansion, R, let us look at a subtree rooted at an internal node S which represents the largest block whose width is less than or equal to $R+1$. Any WHITE leaf node in this subtree is within R pixels of a BLACK leaf node in the subtree and therefore will be changed to BLACK after expansion. S becomes a BLACK node as a result of the expansion. Therefore, we see that instead of changing each WHITE node in the subtree rooted at S to a BLACK node and then merging all these small BLACK nodes into one big BLACK node (as done by WITHIN2), we can just insert S as a single BLACK node. This motivates us to process collections of nodes, when possible, instead of examining the leaf nodes individually.

Define $w(R)$ to be the largest integer that is a power of 2 and is less than or equal to $R+1$, i.e., $w(R)=2^r \leq R+1<2^{r+1}$ for $r \geq 0$. If M is a non-leaf node in Q whose corresponding block is of width $w(R)$, then M is a *merging cluster* of width $w(R)$. Merging cluster M consists of the set of leaf nodes in the subtree rooted at M. In the rest of this paper, whenever we speak of the width of a node, we mean the width of the node's corresponding block.

Figure 4 is an example of a merging cluster with 13 leaf nodes when $R=3$. In this case, $w(R)$ is 4. In Figure 4, A, B, C, and D are the BLACK nodes in the merging cluster. The corners of their blocks, termed *vertices*, are designated by using the corresponding lower case letter with a subscript that designates the vertex. For example, a_{SW} is the SW vertex of node A.

The execution time of WITHIN2 can be reduced by observing that there is no need to individually insert each member of a merging cluster as a BLACK node. Instead, we just insert a single BLACK node of width $w(R)$. However, the main cost of WITHIN2 (i.e., processing those WHITE nodes of width greater than $(R+1)/2$) is not reduced by this method. This is because the determination of how much of the expansion will "spill over" the boundary of the BLACK node just created is much more difficult. We address this problem in the next section.

4.2. Computing the Vertex Set

Consider the expansion of the merging cluster in Figure 4 by 3 pixels. The result is given in Figure 5 from which we make the following observations:

(1) The node corresponding to the root of the merging cluster is now a BLACK node.

(2) The area expanded beyond the boundary of the merging cluster in the vertical and horizontal directions always forms a rectangle. The size of each rectangle is determined by the distance between the boundary and the nearest BLACK node. For example, expansion in the western direction results in a rectangle of size 3 by 4 since a BLACK node appears on the western border of the merging cluster. The sizes of the rectangles formed in the northern, eastern, and southern directions are 4 by 2, 3 by 4, and 4 by 3, respectively.

(3) The area of expansion of the merging cluster in a diagonal direction always forms a staircase-like region. The extreme points of the staircase are those points which can be obtained by translation of the vertices of some of the BLACK nodes in the merging cluster. For example, in Figure 5, the area of expansion in the NW direction is the staircase marked by the points a'_{NW}, b'_{NW}, and c'_{NW} which are obtained by translating the NW vertices a_{NW}, b_{NW}, and c_{NW} by the amount (–3,–3). Similarly, the NE vertices a_{NE} and d_{NE} are translated by (3,–3) into a'_{NE} and d'_{NE}, the SW vertex c_{SW} is translated by (–3,3) into c'_{SW}, and the SE vertex d_{SE} is translated by (3,3) into d'_{SE}. The expansion in the direction of a vertex is completely determined by a subset of the set of all vertices in that direction.

Based on observation (3), we now focus on how to compute efficiently the minimal subset, say V, of vertices of elements of the merging cluster. We know that the expansion from the merging cluster is totally determined by this set. For example, the merging cluster in Figure 4 has a minimal subset V={a_{NW} ,b_{NW} ,c_{NW} ,c_{SW} ,d_{SE}, d_{NE} ,a_{NE}}. The set V is termed the *vertex set* of the merging cluster. The purpose of the vertex set is to minimize the number of BLACK elements of the merging cluster requiring expansion.

Let d be in {NW, NE, SW, SE}. Let OPQUAD(d) denote the vertex direction opposite to d (e.g., OPQUAD(NW)=SE). The *vertex set* (VS) of a merging cluster M is defined to be the union of four vertex subsets VS_d. Given BLACK node P in M, vertex v of P is in VS_d if v is the d vertex of P and v is not in the closed OPQUAD(d) quadrant of any vertex of another BLACK node in M. For example, the vertex set V of the merging cluster in Figure 4 can be decomposed into VS_{NW}={a_{NW},b_{NW},c_{NW}}, VS_{NE}={a_{NE},d_{NE}}, VS_{SW}={c_{SW}} and VS_{SE}={d_{SE}}. Each subset VS_d has the property that the expansion from VS_d in direction d subsumes the expansion in direction d from all the BLACK nodes in the merging cluster. In other words, vertex subset VS_d completely determines the expansion from the merging cluster in direction d.

Now, let us consider expansion in direction D where D is in {N, W, S, E}. Let COMMON_EDGE($Q1$,$Q2$) indicate the boundary of the block containing quadrants $Q1$ and $Q2$ that is common to both of them; e.g., COMMON_EDGE(NW,NE)=N. For each D there exists a pair of vertex directions d_1 and d_2 such that COMMON_EDGE(d_1,d_2)=D. Given a merging cluster P, it can be shown that there exist two vertices v_1 and v_2 that are elements of VS_{d_1} and VS_{d_2}, respectively, such that v_1 and v_2 are at the same distance from the boundary of P's block in direction D. For example, in Figure 5, vertices a_{NW} in VS_{NW} and a_{NE} in VS_{NE} are both closest to the northern boundary of the merging cluster. We can always choose one of these two vertices to determine the extent of the expanded rectangle in direction D. Thus we see that the vertex set completely determines the expansion from the merging cluster in each of the eight directions.

It can be shown that the four vertex subsets are disjoint (Ang, 1988). This means that they can be constructed independently or even in parallel. This allows us to simplify the following discussion by describing how to construct a single vertex subset VS_{SE} of merging cluster M using a preorder traversal of the nodes comprising the merging

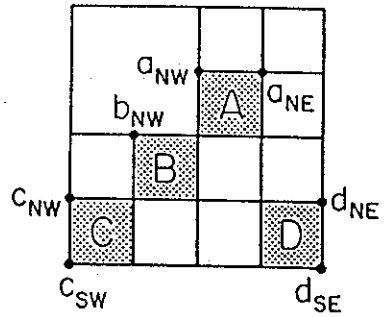


Figure 4. Example of a merging cluster.

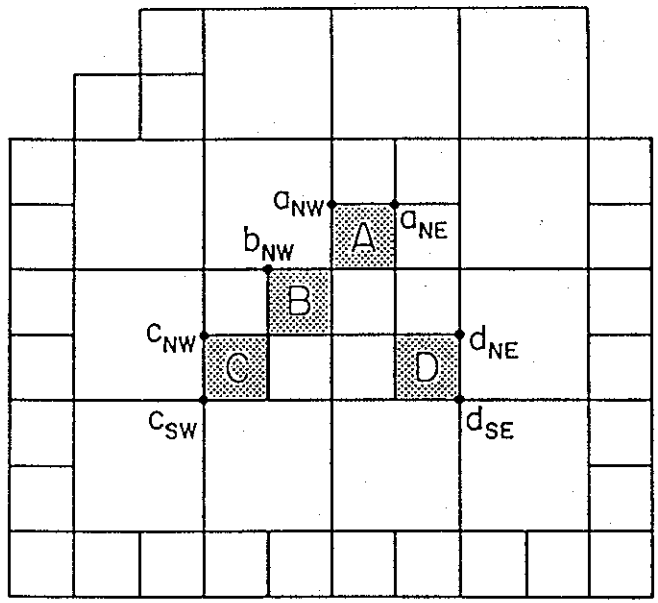


Figure 5. Result of expanding the merging cluster in Figure 4 by 3 pixels.

cluster. The other vertex sets are constructed in an analogous fashion.

Initially, VS_{SE} is empty. The SE vertex of the first BLACK node of M is inserted. For each subsequent BLACK node in M, say P, the position of the SE vertex of P, say v, is compared with the vertices currently in the set. If v is in the opposite quadrant (i.e., NW) with respect to any vertex in the set, say u, then the expansion of v in the SE direction is subsumed by u and hence v is excluded from VS_{SE}. Otherwise, insert v into VS_{SE} after removing all vertices currently in VS_{SE} that are in the NW quadrant of v.

Table 1 shows the changes to each vertex subset during the process of building the vertex set for the merging cluster in Figure 4. For example, the construction of VS_{SE} is as follows. Assume that the BLACK nodes are processed in the order A,B,C,D. When node A is processed, VS_{SE} is empty and hence a_{SE} is inserted into VS_{SE}. For node B, b_{SE} is inserted into VS_{SE} because b_{SE} is not in the NW quadrant of a_{SE} nor is a_{SE} (i.e., the current contents of VS_{SE}) in the NW quadrant of b_{SE}. Similarly, we insert c_{SE} into VS_{SE} for node C. However, node D results in the placement of d_{SE} in VS_{SE} and the removal of a_{SE}, b_{SE} and c_{SE} from VS_{SE} since they are in the NW quadrant of d_{SE}. At the end of the process, $VS_{SE} = \{d_{SE}\}$.

When the vertices in each VS_d are arranged in ascending order of the values of their x coordinates, the values of their corresponding y coordinates are in the following order: VS_{NW} is descending; VS_{NE} is ascending; VS_{SW} is ascending; VS_{SE} is descending;

It is easy to see that no two vertices in VS_d can have the same x coordinate value. Thus the number of vertices in any VS_d is at most $w(R)$. Therefore, the size of the vertex set is less than or equal to $4 \cdot w(R)$. In fact, we can tighten the bound as follows. Let us say that two vertex subsets VS_{d_1} and VS_{d_2} are *adjacent* if COMMON_EDGE$(d_1,d_2) \neq \Omega$.

Theorem: The size of the union of any two adjacent vertex subsets VS_{d_1} and VS_{d_2} is less than or equal to $w(R)+1$, and this bound is attainable.
Proof: The proof is simplified by looking at the different cases individually. For example, consider the case of VS_{NW} and VS_{NE}. Since all the points (pixels) in the SW

Table 1. Building vertex subsets for Figure 4.

node	VS_{NW}	VS_{NE}	VS_{SW}	VS_{SE}
A	$\{a_{NW}\}$	$\{a_{NE}\}$	$\{a_{SW}\}$	$\{a_{SE}\}$
B	$\{a_{NW},b_{NW}\}$	$\{a_{NE}\}$	$\{b_{SW}\}$	$\{a_{SE},b_{SE}\}$
C	$\{a_{NW},b_{NW},c_{NW}\}$	$\{a_{NE}\}$	$\{c_{SW}\}$	$\{a_{SE},b_{SE},c_{SE}\}$
D	$\{a_{NW},b_{NW},c_{NW}\}$	$\{a_{NE},d_{NE}\}$	$\{c_{SW}\}$	$\{d_{SE}\}$

quadrant with respect to the last vertex v in VS_{NW} will not be in VS_{NE}, the vertices in VS_{NE} can only be taken from the region containing points with x coordinate values greater than that of v. Therefore, all vertices in VS_{NW} and VS_{NE} have different x coordinate values. Within a block of width $w(R)$, there can be at most $w(R)+1$ different x coordinate values. Thus there can be no more than $w(R)+1$ vertices. Similar reasoning can be applied to any pair of adjacent vertex subsets VS_{d_1} and VS_{d_2}. The bound is attained when the only BLACK pixels in the merging cluster are those on either major diagonal of a merging cluster (e.g., Figure 6). •

If a node P is the only BLACK node in merging cluster M which is closest to the boundary of M in a given direction (e.g., nodes A, C and D in Figure 4), then P will contribute two vertices to the vertex set. Otherwise, a given node will contribute at most one vertex. Therefore, the number of the vertices in a vertex set is bounded by 4 plus the number of BLACK nodes in M. This bound is attainable as shown in Figure 7. Thus we have proved the following corollary:

Corollary: The size of the vertex set of merging cluster M is bounded by the minimum of $2 \cdot w(R)+2$ and 4 + the number of BLACK nodes in M. The bound is attainable.

4.3. Node Expansion

Now that we have defined the concepts of a merging cluster and a vertex set, we describe the generation of the region within R pixels of the vertex set. The expansions of a merging cluster in the eight directions can be decomposed into two groups, namely those that deal with directions {NW, NE, SW, SE} and those that deal with directions {N, W, S, E}. We shall describe one expansion from each group.

To expand in the NW direction from a merging cluster, only the elements of VS_{NW} need to be considered. The result of the expansion is a staircase-shaped region formed in the NW direction with the steps of the staircase marked by the vertices in VS_{NW} which have been translated by $(-R,-R)$; i.e., they are obtained by subtracting R from the coordinates of each vertex in VS_{NW}. To insert all the nodes that are components of the staircase, find the smallest quadtree block which covers the staircase. Blocks which are completely within (or outside) the staircase are inserted as BLACK (or WHITE). Blocks which partially overlap the staircase are decomposed into four equal-sized blocks which are processed recursively.

To expand in the W direction, we use the vertex v in VS_{NW} which is closest to the western boundary of the merging cluster since the expansion from v in direction W subsumes the expansions from all other vertices. Alternatively, we can also use the western-most vertex in VS_{SW} which has the same x coordinate value. Note that the result of the expansion in directions N, E, S, W which is outside the block, say B, corresponding to the merging cluster is a rectangle, say T. This is a direct consequence of the size of the merging cluster's block being limited to at most $R+1$. In the case of expansion in direction W, rectangle T is adjacent to the west side of B, has height equal to that of B, and width $R-d_W$ where d_W is the distance from v to the W edge of B.

So far we have described how to process the nodes of size $(R+1)/2$ or less by combining them into merging clusters. All WHITE nodes are ignored because they do not require any action. Our algorithm is completed by describing how to process large BLACK nodes (i.e., of size greater than $(R+1)/2$). For such a node B of width W, B will simply be expanded as in algorithm WITHIN1. In other words, the area corresponding to a square of size $W+2R$ centered on B is decomposed into quadtree nodes which are inserted into the output tree.

4.4. Further Reduction of Node Insertions

A further reduction in processing time can be achieved by taking advantage of the interactions between the blocks of neighboring merging clusters. For example, in Figure 8, a merging cluster P needs to be expanded in direction SW only if Q, the block that is adjacent to it only at its SW vertex, is WHITE and of size greater than or equal to $w(R)$. If Q is BLACK or Q is an element of a merging cluster, then P need not be expanded in the SW direction. Otherwise, those pixels of Q that are within R pixels of the vertex set of P are inserted and the nodes labeled Q_0, Q_1, and Q_2 may also need to be visited. Note that we check the colors of nodes Q, Q_0, Q_1, and Q_2 before P is expanded in direction SW. This check has to be done not only for efficiency reasons, but also to prevent the corruption of the output tree since the merging cluster of Q may have been processed before the merging cluster of P. By 'corrupt', we mean that we do not want to overwrite with the value BLACK a node which had another value in the original (multicolored) input tree.

5. Analysis

As Figure 3 demonstrates, the execution time of WITHIN4 generally increases at a much slower rate and even decreases with increasing R. WITHIN4 usually outperforms the other algorithms. This is a result of the interaction between two factors which have opposite effects. The first, and most important, factor is the number of merging clusters, while the second factor is the radius of expansion. As the radius of expansion increases, the size of the merging cluster also increases. This means that there are fewer merging clusters and thus the execution time should decrease since there is less need for expansion. On the other hand, as the radius of expansion increases, there is an increase in the number of nodes that must be inserted as a result of expanding from the merging cluster's vertex set. This has been explained in the discussion of WITHIN1 in Section 2. As the data in Figure 2 shows, these two competing factors tend to cancel each other out.

The data of Figure 3 also confirms the following more detailed analysis of the effect of the radius of expansion. Assume an image of size $2^n \times 2^n$. Clearly, when $R \geq 2^n-1$ only one node needs to be inserted into the output quadtree. In the more general case, as R increases from 2^r-2 to 2^r-1, $w(R)$ is doubled, thereby significantly reducing the number of nodes to be expanded, and the time required to expand them is less than that for $R = 2^r-2$. $w(R)$ is constant (i.e., 2^r) when R is between 2^r-1 and $2^{r+1}-2$, and the execution time increases slowly (but linearly - recall the analysis of

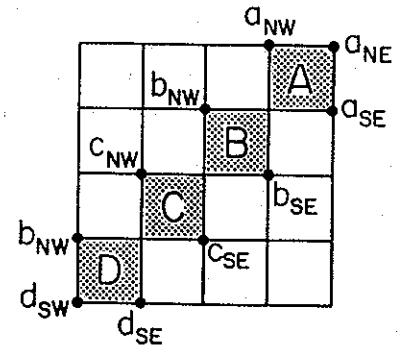


Figure 6. Example merging cluster where the upper bound on the size of two adjacent vertex subsets is attained.

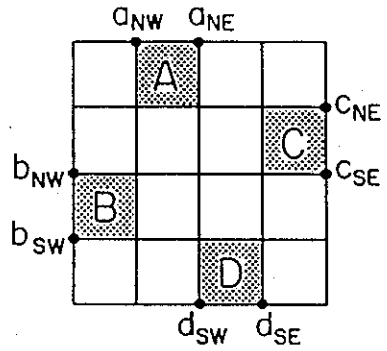


Figure 7. Example merging cluster where the upper bound on the size of the vertex set is attained.

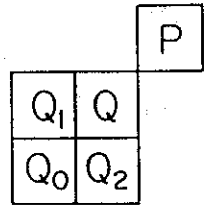


Figure 8. Example merging cluster configuration.

WITHIN1) as R increases since more nodes will be inserted. The execution times for odd values of R show a cyclical behavior which is characterized by a slow increase as R increases from 2^r-1 to 2^{r+1}-2, followed by a drop back to nearly the lowest execution time for 2^{r+1}-1. The behavior for even values of R generally follows the same pattern although, as mentioned in Section 2, the execution times are reduced due to the effects of node aggregation which results in fewer nodes.

The oscillatory pattern of execution times of WITHIN4 for odd and even values of R is analogous to those of WITHIN1 and WITHIN2. The only difference occurs when R increases from 2^{r+1}-3 to 2^{r+1}-1. In this case, the execution time of WITHIN1 increases as there are more nodes of size 1 to be inserted, whereas the execution times of WITHIN2 and WITHIN4 decrease. For WITHIN2, the decrease is caused by the doubling of the maximum size of the WHITE nodes which are automatically inserted as BLACK nodes. For WITHIN4, the analogous result is that $w(R)$ is doubled, thereby reducing the number of merging clusters. In particular, this means that at most four merging clusters P_1, P_2, P_3, and P_4 of size 2^r are merged into one merging cluster P of size 2^{r+1}. Therefore, at most eight expansions in the directions of a corner can be avoided depending on whether the surrounding nodes are WHITE or not. For example, if P_1 is a NW son of P, then it is possible to avoid the NE and SW expansions from P_1 if there are large WHITE nodes in those directions. Four insertions of P_1, P_2, P_3, and P_4 as BLACK nodes are now replaced by one insertion of P regardless of whether P_i is a merging cluster or a WHITE node. All these savings result in a reduction of the execution time. This is confirmed by the results shown in Table 2.

Table 2. Execution times(seconds) of WITHIN4 for large radii

Radius	Center	ACC
64	11.5	12.6
128	11.1	10.9
256	8.6	6.0
512	5.1	3.9

In general, the complexity of the region expansion process depends on the complexity of the image in the sense that as the number of nodes in the image (especially the expanded image) increases, so will the execution time for small radius values. However, as the radius of expansion increases past a certain value, for many complex images the expanded image will have a significantly smaller number of nodes due to merging. Thus the expansion algorithm should run faster for algorithms that process small nodes efficiently (e.g.,WITHIN2 and WITHIN4). This reduces the attractiveness of WITHIN1 and WITHIN3 as their execution times must increase when R increases regardless of node size. This is borne out by the Pebble image, where it is seen that the execution times of both WITHIN4 and WITHIN2 show a steady decrease with increasing R. In contrast, for the much smaller Center and ACC images, as R increases, the execution times of WITHIN4 are relatively constant while those for WITHIN2 increase. For these images the WHITE area is sufficiently large that the amount of merging has not yet started to dominate.

Although the complexity of the image is an important factor in determining the complexity of the region expansion process, it is overshadowed by the size of the merging

cluster. The effect of the size of the merging cluster on WITHIN4 has already been discussed. It also has an indirect effect on WITHIN2. Recall that WITHIN2's work lies in processing WHITE nodes. As R increases, we know that more WHITE nodes can automatically be inserted as BLACK nodes - in particular, all WHITE nodes of width less than or equal to $(R+1)/2$. For example, WITHIN2's execution times for the Center image exceed those for the ACC image by about 30% for the initial radius values. This correlates well with the ratio of their total node counts. Although initially Center has more nodes than ACC, for $R>16$, ACC contains more WHITE nodes of size $> 8\times8$ than Center and thus more work will be required to expand ACC than Center. This imbalance increases as R increases and is reflected by Figures 2a and 2b which show that the execution time of expanding ACC (79.3 sec.) exceeds that for Center (61.6 sec.) by about 30% for $R=32$.

6. Conclusions

In Section 1 we mentioned that a good yardstick for measuring the performance of a region expansion algorithm is the amount of time necessary to build the expanded quadtree from an array representation. Using the QUILT system, building quadtrees for the Center and Pebble images took approximately 16 and 110 seconds (Shaffer, 1987a), respectively. Recall that these images contain 4693 and 44950 nodes respectively. The building algorithm has the property that its execution time is directly proportional to the number of nodes in the image. For $R>15$, the expanded Center and Pebble images contain approximately 8000 and 46000 nodes, respectively. For $R=32$ expanding them by WITHIN4 took 12 and 76 seconds, respectively, which means that building them from an array representation is not as fast as expanding and building them simultaneously. By this measure, WITHIN1 and WITHIN3 are not attractive. Of course, a different implementation may yield different execution times (e.g., for WITHIN3), but we believe that our qualitative explanations of the algorithms are appropriate. Now, let us compare WITHIN2 and WITHIN4 more closely.

(1) In WITHIN2, every BLACK node in a merging cluster is individually inserted while in WITHIN4 only one insertion must be performed for the entire merging cluster. However, WITHIN2 can be modified to avoid this shortcoming.

(2) In WITHIN4, no repeated insertions are caused by the BLACK nodes within a merging cluster. Only a different merging cluster can cause a node to be inserted repeatedly. Moreover, a node can be inserted at most eight times since there are at most eight neighboring merging clusters. In WITHIN2, a node in the output tree, say B, may be repeatedly inserted as a result of the expansion of each of the BLACK nodes in the input tree which are within radius R of B.

(3) In WITHIN4, nodes are expanded using the merging cluster's block as the hub for the expansion process. This approach is similar to the expansion based on a BLACK node in WITHIN1. WITHIN4 stores the vertex set as an array sorted by the value of the x coordinate. Some of the operations on the vertex set require a sequential search. The data that we gathered revealed that, on the average, for radius values up to 32 each vertex subset VS_i contained about two vertices.

Thus, the performance of WITHIN4 is unlikely to be improved by using a more complex data structure to organize the vertex set.

(4) The ratio of the execution time of WITHIN2 to that of WITHIN4 increases with R. The magnitude of the ratio (6 for Center and ACC and 5 for Pebble) depends on the complexity of the expanded image. If the image does not have large WHITE nodes (e.g., Pebble), then the ratio is small since most of WITHIN2's time is devoted to processing large WHITE nodes.

References

Abel, D.J. and Smith, J.L., 1983, A data structure and algorithm based on a linear key for a rectangle retrieval problem, *Computer Vision, Graphics, and Image Processing 24*, 1(October 1983), 1-13.

Ang, C.H., Samet, H., and Shaffer, C.A., 1988, A fast polygon expansion algorithm for linear quadtrees, Computer Science Technical Report, University of Maryland, College Park, MD, to appear.

Gargantini, I., 1982, An effective way to represent quadtrees, *Communications of the ACM 25*, 12(December 1982), 905-910.

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

Hunter, G.M., 1979, Operations on images using quad trees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 1*, 2(April 1979), 145-153.

Klinger, A., 1971, Patterns and search statistics, in *Optimizing Methods in Statistics*, J.S. Rustagi, Ed., Academic Press, New York, 1971, 303-337.

Mason, D.C., 1987, Dilation algorithm for a linear quadtree, *Image and Vision Computing 5*, 1(February 1987), 11-20.

Nelson, R.C. and Samet, H., 1986, A consistent hierarchical representation for vector data, *Computer Graphics 20*, 4(August 1986), 197-206 (also *Proceedings of the SIGGRAPH'86 Conference*, Dallas, August 1986).

Rosenfeld, A. and Kak, A.C., 1982, *Digital Picture Processing*, Second Edition, Academic Press, New York, 1982.

Samet, H., 1981, Connected component labeling using quadtrees, *Journal of the ACM 28*, 3(July 1981), 487-501.

Samet, H., 1982, A distance transform for images represented by quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 4*, 3(May 1982), 298-303.

Samet, H., 1984a, *ACM Computing Surveys 16*, 2(June 1984), 187-260.

Samet, H., Rosenfeld, A., Shaffer, C.A., and Webber, R.E., 1984b, A geographic information system using quadtrees, *Pattern Recognition 17*, 6(1984), 647-656.

Samet, H., Rosenfeld, A., Shaffer, C.A., Nelson, R.C., and Huang, Y.G., 1984c, Application of hierarchical data structures to geographical information systems, Phase III, Computer Science TR-1457, University of Maryland, College Park, MD, November 1984.

Samet, H., Rosenfeld, A., Shaffer, C.A., Nelson, R.C., Huang, Y.G., and Fujimura, K., 1985a, Application of hierarchical data structures to geographical information systems : Phase IV, Computer Science TR-1578, University of Maryland, College Park, MD, December 1985.

Samet, H. and Webber, R.E., 1985b, Storing a collection of polygons using quadtrees, *ACM Transactions on Graphics 4*, 3(July 1985), 182-222.

Shaffer, C.A. and Samet, H., 1987a, Optimal quadtree construction algorithms, *Computer Vision, Graphics, and Image Processing 37*, 3(March 1987), 402-419.

Shaffer, C.A. and Samet, H., 1987b, An algorithm to expand regions represented by linear quadtrees, Computer Science TR-1849, University of Maryland, College Park, MD, May 1987, to appear in *Image and Vision Computing*.

Shaffer, C.A., Samet, H., and Nelson, R.C., 1987c, QUILT: A geographic information system based on quadtrees, Computer Science TR-1885, University of Maryland, College Park, MD, July 1987.

PROCEEDINGS

THIRD
INTERNATIONAL SYMPOSIUM ON
SPATIAL DATA HANDLING

August 17 - 19, 1988

Sydney, Australia