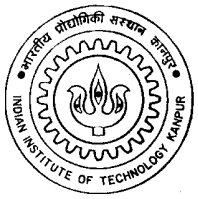
# Certain Quadtree Based Image Processing Algorithms

*A Thesis Submitted*
*in Partial Fulfillment of the Requirements*
*for the Degree of*
*Master of Technology*

*by*

**Burgu Praveen Kumar**




*to the*

**Department of Computer Science & Engineering**
Indian Institute of Technology, Kanpur

**May, 2001**

# Certificate

This is to certify that the work contained in the thesis entitled "*Certain Quadtree Based Image Processing Algorithms*", by *Burgu Praveen Kumar*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

May, 2001

(Dr. Phalguni Gupta)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

# Acknowledgements

I would like to take this opportunity to thank my thesis supervisor Dr. Phalguni Gupta for his expert guidance without which this thesis would not have been possible. He has been most understanding throughout my stay at I.I.T., Kanpur. Working with him has been a very enjoyable experience.

I thank all my class mates, batch mates and IKAN group for making my stay at I.I.T., Kanpur memorable. Especially, I would like to thank Prasanth, Sreekanth, Kiran, Vamsi, Sarath, Dharma Reddy, Chandu, Madhav, V S Rao and Krishna Reddy. I would also like to thank Vikrant Khanna. He has been most helpful.

Last but not the least, I thank all my family members for their constant love and support.

B Praveen Kumar

# Abstract

Recently, Image Processing has been found useful for wide applications in remote sensing. Often, in these applications, it is required to manipulate large images. Hence, efficient data structures and algorithms should be used for processing these images.

Quadtree is a popular hierarchical representation of images. Computing neighbors of a node is one of the basic problems in this representation. In this thesis, we propose a variation of quadtree to compute neighbors efficiently. We present algorithms for the construction and maintenance of this quadtree.

Perimeter computation of the image is one of the common image processing problems in remote sensing applications. A dynamic algorithm is presented for the maintenance of perimeter in quadtree representation of images. This algorithm updates the perimeter as the leaf nodes, in the quadtree, undergo changes.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation

Image processing plays an important role in remote sensing applications. Generally, the images used for these applications are of large size. Processing these images require high computational power and the algorithms designed should be efficient. These images are represented in many forms. Different representations are efficient for different applications. Many remote sensing applications require processing of a set of images where there is not much difference among the images. In particular, if we take binary images and operations on it, most of the operations on these images are same for each pixel. Instead of processing each pixel separately, if we group the pixels having same intensity then we can apply the operations on these groups. This reduces the overhead of redundant computations. Clearly, one needs efficient data structures and algorithms that are suited for remote sensing applications.

## 1.2 Goal of the Thesis

In this work, we focus our attention to two problems which generally occur while processing remotely sensed images. The first problem is to design a better data structure for representing binary images, that can be processed efficiently. Quadtree is one such data structure and is a popular hierarchical representation in which the binary image is decomposed into four quadrants. The basic operation which uses

this representation is to find neighbors of a region represented by a node. There exist variants of quadtree, which take much time to compute neighbors or takes extra space for maintaining links to neighbors. The objective here is to find a better data structure, which doesn't take extra space but finds the neighbors efficiently.

The second problem considered here deals with maintaining the perimeter when a homogeneous region in the quadtree representation of a binary image undergoes a change. The perimeter of an image is defined as the total number of pixels corresponding to the boundary of the regions in the image. The objective is to design efficient algorithm that updates the perimeter of the changed image.

## 1.3 Organization of the Thesis

Besides the first chapter which discusses the motivation of the work, the thesis contains four more chapters. The outline of our work is as follows:

**Chapter 2.** This chapter introduces the two representations: raster and hierarchical, with emphasis on the quadtrees. Existing variants of quadtree data structure are discussed. Algorithms for finding neighbors in quadtrees and, perimeter computation in quadtree representation are studied.

**Chapter 3.** A variant of quadtree data structure is proposed in this chapter. Algorithms for construction and maintenance of this quadtree have been presented.

**Chapter 4.** This chapter presents a new algorithm to maintain perimeter in quadtree representation of images.

**Chapter 5.** Concluding remarks are made in this chapter.

# Chapter 2

# Survey : Image Representation and Perimeter Computation

## 2.1 Digital Image Representation

Image representation plays a major role in Image Processing. An image data is essentially of spatial nature consisting of regions. The choice of representation of this data depends upon the applications in mind. The two popular representations are Array representation, and Hierarchical representation.

### 2.1.1 Array Representation

In Array representation image is represented in the form of a two dimensional array where each array element represents a distinct pixel in the image. The image is processed by scanning the array in particular order. There are a number of scanning orders possible for this representation but not all of them are efficient. The array representation is perhaps the most intuitive and easier to work with. But there are a number of applications where this representation is not that efficient. For example in an application if we are interested only in small portion of image, under this representation we have to scan unnecessary portions of the image. To overcome this limitation, we normally use hierarchical representation in such applications.

### 2.1.2 Hierarchical Representation

The hierarchical representation is based upon recursive decomposition of the image [9]. Here, we try to identify the homogeneous regions in the image and the criteria for homogeneity depends upon the application. Initially, the whole image is treated as one region. If the region does not satisfy the homogeneity criteria, it is split into different subregions and the process is repeated for each of the subregions. The way we split a non-homogeneous region into subregions of smaller size depends upon the kind of hierarchical representation we have in mind. We go on splitting the regions till we get a subregion which satisfies the homogeneity criteria. The advantages of hierarchical representation is that they allow focusing on subsets of data that are of interest without going through irrelevant data. This results in efficient image processing algorithms.

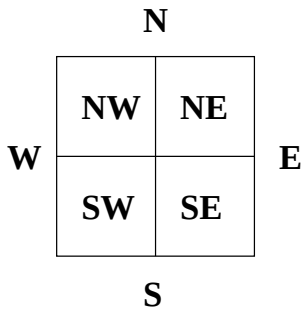



Figure 2.1: Relation between Quadrants and Boundaries.

Quadtree is a very common hierarchical representation. In the quadtree representation, if a region does not satisfy the homogeneity criteria, we divide it into four equal quadrants (also called as NW, NE, SW, SE quadrants), as shown in Figure 2.1. The sides of the region are also referred with the direction in which they lie; for example, the side adjacent to the NW and NE quadrants is called the N (or the northern) side. We stop splitting once a region satisfies the homogeneity criteria. A binary image and its corresponding quadtree is shown in Figure 2.2. The shaded region consists of the object pixels. The homogeneity criteria here is that the region must contain either only object pixels or only background pixels. If a region consists of only object

pixels, we call its corresponding node in the quadtree as BLACK node. If it consists of only background pixels, we call the node as WHITE node. Otherwise, the region consists of some object pixels and some background pixels and we call the node as GRAY node. In Figure 2.2, the GRAY nodes are represented as circles while the shaded squares represent the BLACK nodes and the white squares represent WHITE nodes.

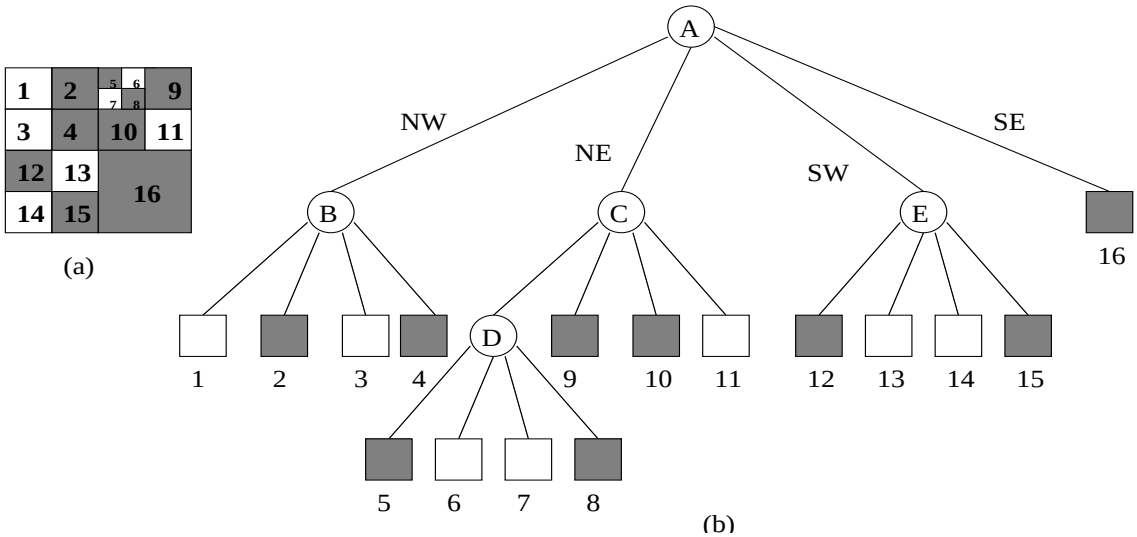



Figure 2.2: Binary Image and its Quadtree Representation.

## 2.2 Finding Neighbors of a Region

Many basic operations can be implemented as tree traversal of quadtree representation. The difference among them is the nature of computation performed at each node. Often these computations involve the examination of nodes whose corresponding blocks are *adjacent* to the block corresponding to the node being processed. We can term these adjacent nodes as *neighbors*.

Each node of a quadtree corresponds to a block in the image. We use the terms *blocks* and *node* interchangeably. The term used depends on whether we are referring to decomposition into blocks (i.e., Figure 2.2(a)) or a tree (i.e., Figure 2.2(b)). Each block has four sides and four corners. Sides and corners can be used as directions.

In this section, we discuss the techniques to find the adjacent neighbors in the horizontal or vertical directions. The basic idea is to ascend the quadtree until a common ancestor with the neighbor is located, and then descend back down the quadtree in search of the neighboring node. For example, to find eastern neighbor of node `2` in Figure 2.2 we follow these steps. The nearest common ancestor is the first ancestor node which is reached via its NW or SW son (i.e., the first ancestor node of which node `2` is not an eastern descendant. Next, we retrace the path used to locate the nearest common ancestor, except that we make mirror image moves about an axis formed by the common boundary between the nodes. In case of eastern neighbor, the mirror images of NW and SW are NE and SE, respectively. Therefore the eastern neighbor of node `2` is `D`. It is located by ascending the quadtree until the nearest common ancestor `A` has been located. This requires going through a NE link to reach node `B`, and a NW link to reach node `A`. Node `D` is subsequently located by backtracking along the previous path with the appropriate mirror image moves (i.e., by following NE link to reach node `C`, nad a NW link to reach node `D`. It is often the case that neighbors are of different size. In that case neighbor of a node may be of equal or greater size or many nodes with smaller size. If it is greater or equal size it returns that node. If the neighbors are many smaller nodes it returns a GRAY node of equal size if exists, otherwise the node adjacent to the border of the image and NULL is returned. The formal description of the procedure to find neighbors in horizontal or vertical direction is given in Algorithm 2.1.[4]

## 2.3 Other Methods for Neighbor Location

The computation performed at a node of a quadtree by the algorithms may involve the examination of its neighboring nodes. There exist algorithms for finding the neighbor of a node in horizontal or vertical direction [8, 4, 5]. These algorithms can be classified into two categories called top-down and bottom-up algorithms. The top-down algorithms pass the pointers to the neighbors from top to down as explicit parameters during the tree traversal [8]. The disadvantage of this type of algorithms is that the space complexity is more. Further, in some applications top-down algorithms cannot be used. The other class is bottom-up algorithms. These can further be

```
node procedure GTEQUAL_ADJ_NEIGHBOR(P,D)
/* Return the neighbor of node P in horizontal or vertical direction D which
   is greater than or equal in size to P. If such an node does not exist, then a
   GRAY node of equal size is returned. If this is also impossible, then the node
   is adjacent to the border of the image and NULL is returned. */
begin
   node P,Q;
   direction D;
   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
```

**Algorithm 2.1 :** Finding Neighbor of a node in Quadtree Representation

distinguished by their different approaches.

- Nearest common ancestor method : In this method the adjacent neighbors in the horizontal or vertical direction are located by ascending the quadtree upto the common ancestor of the node and its neighbor, and then descend back down the tree until it locates the neighbor. The disadvantage of this method is to traverse many nodes in between the adjacent nodes. The average number of nodes visited by this method to find a neighbor in horizontal or vertical direction is bounded by 5 [4].

- Method dependent of position and size of the node : In this method coordinate information and size of the image are used to locate an adjacent node in a given direction [5]. The disadvantage of this method is that it requires computation rather than pointer manipulation.

- Explicit linkage of neighbors : This method assumes explicit linkage of neighbors [1]. A *rope* is defined to be a link between two adjacent nodes of equal size where

at least one of them is a leaf node. For example, in Figure 2.2 the rope from node D points to node 2 in west direction. To find the neighbor in a direction, one checks whether a rope exists in that direction or not. If a rope exists, it leads to the desired neighbor. Otherwise, one ascends the quadtree until a rope exists in that direction. This will lead to the neighbor in that direction which is of greater size. A *net* is linked list whose elements are all the nodes, regardless of their relative sizes, that are adjacent along a given direction of node. For example, in Figure 2.2 the net in eastern direction of node 2 consists a linked list of nodes 5 and 7. Each leaf node in quadtree contains four `NET` entries, one for each direction, pointing to lists containing pointers to the adjacent nodes. Neighbors in a direction are located by accessing the appropriate `NET` field. The disadvantage of these trees is that it requires extra space for accommodating the `ROPE` and `NET` fields. For instance, each node in a rope quadtree requires four extra fields for storing ropes in four directions and each leaf node in a net quadtree requires four extra fields for storing nets in four directions and these nets in turn require space for accommodating linked lists consisting of neighbors.

## 2.4 Perimeter Computation in Quadtree Representation

The perimeter of a region is defined as the number of pixels corresponding to the boundary of the region. The perimeter of an image is defined as the total of the length of the perimeters of regions in the image. The problem of finding perimeter is defined as follows :

**Problem Definition :** Given an image $\mathcal{I}$, compute the total perimeter of regions in the image represented by quadtree.

The perimeter computation algorithm traverses the quadtree in postorder. For each `BLACK` terminal node, say `P`, visit all of the nodes whose corresponding blocks have a boundary in common with `P`'s block. These are termed as `P`'s northern, eastern, southern, western adjacencies. For each of the visited nodes that is `WHITE`, the length

```
procedure PERIMETER(P, LEVEL)
/* Find the perimeter of a region represented by a quadtree rooted
   at node p that spans a 2 ↑ LEVEL by 2 ↑ LEVEL space */
begin
   node P,Q;
   integer LEVEL,LEN;
   quadrant I;
   direction D;
   LEN ← 0
   if GRAY(P) then
      begin
         for I in {"NW","NE","SW,"SE"} do
               LEN ← LEN + PERIMETER(SON(P,I),LEVEL-1)
      end
   else if BLACK(P) then
      begin
         for D in {"N","E","W","S"} do
            begin
               Q ← GTEQUAL_ADJ_NEIGHBOR(P,D)
               LEN ← LEN + if NULL(Q) or WHITE(Q) then 2↑LEVEL
                           else if GRAY(Q) then
                              SUM_ADJACENT(Q,QUAD(OPSIDE(D),CSIDE(D)),
                                    QUAD(OPSIDE(D),CCSIDE(D)),LEVEL)
                           else 0
            end
      end
   return(LEN)
end
```

**Algorithm 2.2 :** Perimeter Computation in Quadtree Representation

of the common boundary is included in the value of the perimeter.

The main procedure is termed as `PERIMETER` [12] and is invoked with a pointer to the root of the quadtree representing the region and an integer corresponding to the resolution of the image (e.g., $n$ for a $2^n$ X $2^n$ image array). `PERIMETER` traverses the tree and controls the exploration of the adjacencies of each BLACK node. For each BLACK node, say `P`, `GTEQUAL_ADJ_NEIGHBOR` (given in Algorithm 2.1) locates a

```
integer procedure SUM_ADJACENT(P,Q1,Q2,LEVEL)
/* Find all WHITE leaves in quadrants Q1 and Q2 of the
   subquadtree rooted at node P of size 2 ↑LEVEL */
begin
   node P;
   quadrant Q1,Q2;
   integer LEVEL;
   return(if GRAY(P) then SUM_ADJACENT(SON(P,Q1),Q1,Q2,LEVEL-1)
                        + BTW_ADJACENT(SON(P,Q2),Q1,Q2,LEVEL-1)
          else if WHITE(P) then 2↑LEVEL
          else 0)
end
```

**Algorithm 2.3:** SUM_ADJACENT

neighboring node, say `Q`, of greater or equal size along a specified direction, say `D`. If `Q` is WHITE, then the contribution to the perimeter is the size of `P`. If `Q` is BLACK, then no contribution is made to the perimeter. If the boundary of `P` in direction `D` is on the border of the image, then no neighbor exists in the specified direction and NULL is returned. In such a case the contribution to the perimeter of the boundary of `P` in direction `D` is equal to the size of `P`. If the boundary of `P` in direction `D` is not on the border of the image, and no neighboring BLACK or WHITE node exists satisfying our size criteria, then a pointer to a GRAY node of equal size is returned. In such a case procedure `SUM_ADJACENT` continues the search by examining all nodes of smaller size that are adjacent to `P`'s boundary in direction `D` and accumulating the sizes of all such nodes that are WHITE. The formal description of procedure `PERIMETER` is given in Algorithm 2.2 and procedure `SUM_ADJACENT` is given in Algorithm 2.3.

# Chapter 3

# v-Quadtree

By observing that the data-structures presented in Section 2.3 takes much time to compute neighbor or takes extra space for maintaining links to neighbors, in this chapter we propose a new data-structure which doesn't take any extra space and finds the neighbors efficiently.

## 3.1 v-Quadtree : A data-structure

Like a quadtree, the `v-Quadtree` is represented by a tree with out-degree of four in which the root represents a block and the four sons represent in order the `NW, NE, SW, SE` quadrants. Here, we assume that each node is stored as a record containing seven fields. One field points to its parent. This field of the root node points to NIL. Four fields of a non-leaf node (`GRAY`) points to its four sons (of `NW, NE, SW, SE` quadrants). But in case of a leaf node (`BLACK` or `WHITE`) they points to its four neighbors (in `N, E, W, S` directions), provided they exist. The sixth field `NODETYPE` describes the contents of the block of the image which the node represents, i.e., `BLACK, WHITE` or `GRAY`. Using this field we can distinguish the intermediate nodes and leaf nodes. If `NODETYPE` is `GRAY` it is an intermediate node and it has links to the parent and four sons. Otherwise, it is a leaf node and it has links to the parent and four neighbors. The last field `SIZE` contains the size of the block, i.e., total number of pixels occupied by the corresponding region.

Figure 3.1(a) shows the structure for intermediate node and Figure 3.1(b) shows

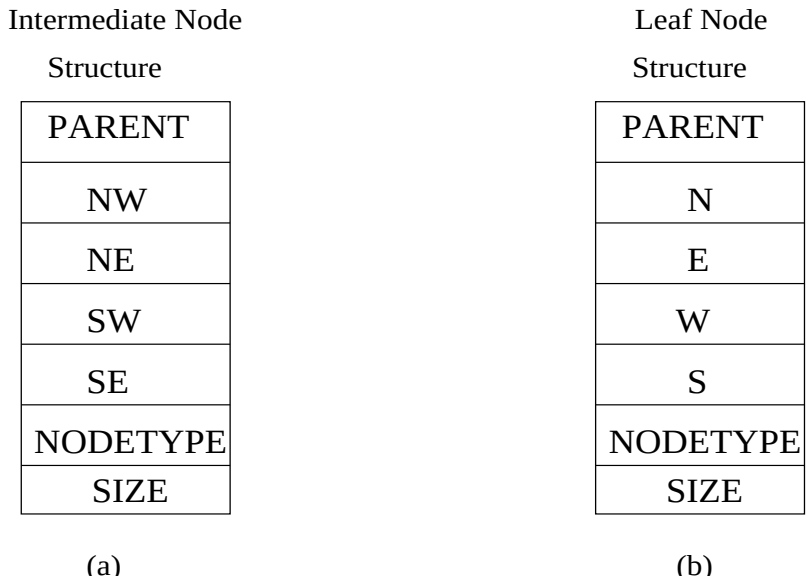



Figure 3.1: v-Quadtree Node Structure

the structure for leaf node. The fields `NW, NE, SW, SE` represents, pointers to the sons corresponding to `NW, NE, SW, SE` quadrants, respectively. For example, in Figure 2.2, the `PARENT` field of the node `B` contains pointer to node `A`, and the fields `NW, NE, SW, SE` contain pointers to the nodes `1, 2, 3, 4` respectively. The `NODETYPE` field contains `GRAY` and `SIZE` contains 16. For the node `10` in Figure 2.2, the `PARENT` field contains pointer to node `C`, the fields `N, E, W, S` contain pointers to the nodes `D, 11, 4, 16` respectively. The `NODETYPE` field contains `BLACK` and `SIZE` contains 4. Figure 3.2 shows `v-Quadtree` representation for the image shown in Figure 2.2(a). In this the solid lines represent the links to sons and dotted lines represent links to neighbors. The `NODETYPE G` represents `GRAY`, `B` represents `BLACK` and `W` represents `WHITE`. The links to sons are bi-directional as the sons also have link to parent. The field `'\'` represents a null pointer.

Let `P` be any node in the new quadtree, `Q` be any non-leaf node and `R` be a leaf node. Let `I` represent a quadrant which is one of `NW, NE, SW, SE` quadrants. Also, let `D` represent a direction which is either `N, E, W, S`. Now let `PARENT(P)` denote the parent of `P` which is equivalent to `P`$\rightarrow$`PARENT` and points to the parent of `P`. For example, in Figure 3.2, `PARENT(C)` denotes pointer to node `A`. Assume

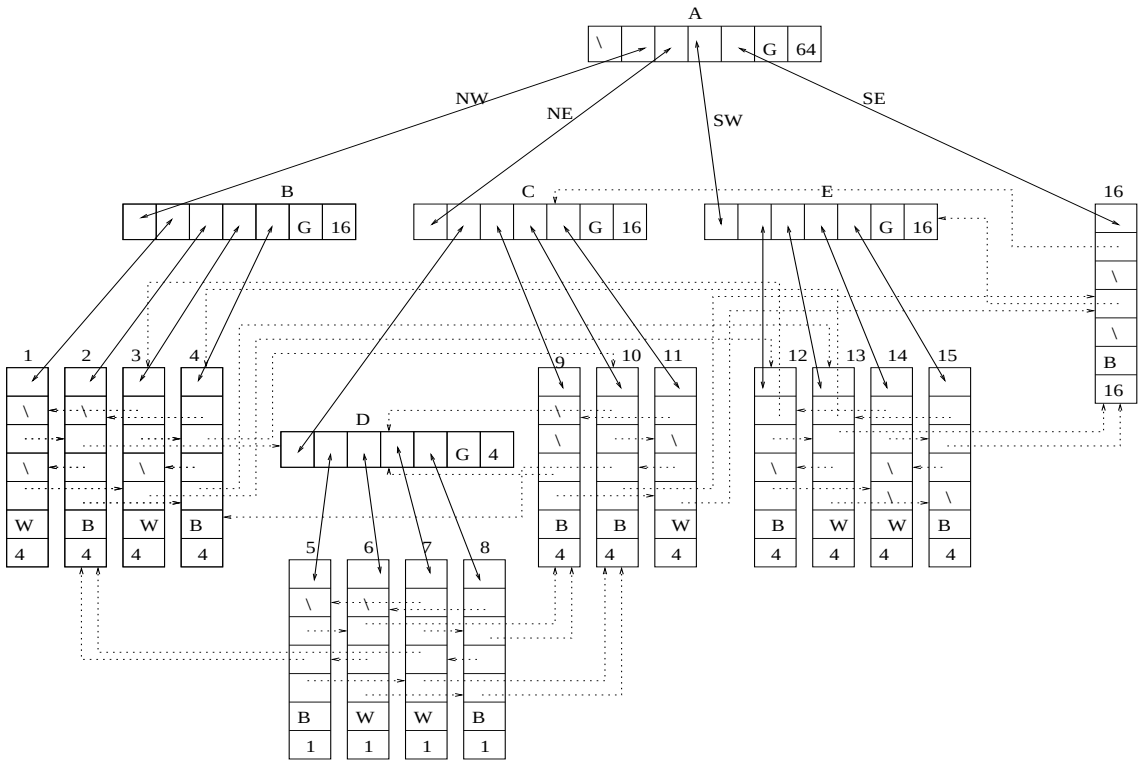



Figure 3.2: v-Quadtree Representation of Image shown in Fig 2.2(a).

`SON(Q,I)` denotes the son for a quadrant `I`, where `SON(Q,I)` is equivalent to `Q`→`I`. For example, in Figure 3.2, `SON(C,NW)` denotes a pointer to node `D`. Let `NEIGHBOR(R,D)` denote the neighbor in the direction `D`, which is equivalent to `R`→`D`. For example the `NEIGHBOR(7,S)` in figure 3.2 denotes pointer to node `10`. Note that `SON` and `NEIGHBOR` represent the same fields for a particular node.

Let `P` be a node and `I` represent a quadrant, the function `SONTYPE(P)` (where `SONTYPE (P) = I` iff `SON(PARENT(P),I) = P`) can be used to find the quadrant to which the node belongs. For example, in Figure 2.2, `SONTYPE(1)` is `NW`, since `SON(PARENT(1),NW) = 1`. We use the functions `OPSIDE, CSIDE, CCSIDE`, and `QUAD`, which are defined in [12], in the algorithms presented in Section 3.2. Let `B` be a side (`N, E, W, S`), then `OPSIDE(B)` is a side facing side `B`, e.g., `OPSIDE(N) = S`. `CSIDE(B)` and `CCSIDE(B)` correspond to the sides adjacent to side `B` in the clockwise and counterclockwise directions, respectively, e.g., `CSIDE(N) = E` and `CCSIDE(N) = W`. Let `S1`

and S2 be two sides, then QUAD(S1,S2) is the quadrant bounded by boundaries S1 and S2 (if S1 and S2 are not adjacent boundaries, then QUAD(S1,S2) is undefined), e.g., QUAD(S,E) = SE, while QUAD(S,N) is undefined.

## 3.2 Algorithm

In this section, we present algorithms for finding neighbors in v-Quadtree, constructing this quadtree from an image. The construction uses the top-down approach. Generally, the top-down algorithms are not efficient for constructing quadtrees from binary images. There exist efficient bottom-up algorithms for constructing quadtree from a binary image [7, 10]. We can use top-down approach for constructing quadtrees from gray-scale images. In this approach for every sub-region we scan the area and depending upon the threshold of this area we decide whether it is homogeneous or non-homogeneous region. Otherwise, we have to first convert the gray-scale image into binary image and then construct the quadtree from this binary image. We also propose an algorithm for maintaining v-Quadtree if some homogeneous region changes its membership.

### 3.2.1 Finding Neighbors in v-Quadtree

In the v-Quadtree structure, each leaf node has four links connected to its neighbors in horizontal and vertical directions. These neighbors can be reached by following a single link. For finding neighbor in corner directions we can use the algorithm given in [4] with a minor modification. This algorithm makes a call to function GTEQUAL_ADJ_NEIGHBOR, which finds the horizontal or vertical neighbor. In the modified algorithm, this call is replaced by an assignment to the horizontal or vertical neighbor as it has a direct link.

### 3.2.2 Constructing v-Quadtree

The algorithm for constructing v-Quadtree, termed as vquadtree, is based on the top-down approach. The algorithm initially starts by creating a node with create_node(), and passes the pointer to this node to vquadtree with size equal to the size of the image and parent with a null pointer. The procedure create_node()

```
procedure vquadtree(tree,size,parent)
/* This procedure constructs v-Quadtree of a region represented by
   root node tree. The region size and node's parent are given. */
begin
   node tree,parent,temp;
   integer size;
   quadrant I;
   direction D;
   Assign the size to tree.size, parent to tree→parent
   Find the type of the node (WHITE or BLACK or GRAY) store it in tree.nodetype
   Save tree node info. into temp node.
   if GRAY(tree) then
      begin
         CREATE_SONS(tree)
         PROCESS_GRAY(tree,temp)
         /* Recursively call quadtree for NW, NE, SW, SE sons with quarter
            the size of current node and parent as current node. */
         for I in {"NW","NE","SW,"SE"} do
               vquadtree(SON(tree,I),SIZE(tree)/4,tree)
      end
   else /* LEAF NODES.....*/
         PROCESS_LEAF(tree,temp)
   endif
end
```

**Algorithm 3.1 :** Constructing Quad Tree

creates a node with seven fields and returns a pointer to this node. The fields of the node are initialized to `NIL`. This is a recursive approach. In this approach we assume that unassigned pointers point to `NIL`. The algorithm terminates when the `NODETYPE` is either `BLACK` or `WHITE`, after processing the node. At any stage `vquadtree` is invoked with a pointer to a node of the quadtree, which represents a region, the size of the region and a pointer pointing to the parent. It finds the type of the node by scanning the region represented by this node. If this region is homogeneous, then the node is marked as either `WHITE` or `BLACK` depending on the threshold of the region. Otherwise the node is marked as `GRAY`. The information about the current node is saved in a temporary node (contains pointers to neighbors). If the current node is

```
procedure CREATE_SONS(tree)
/* This procedure creates four sons for the node tree to represent four quadrants of
   this region, SON(tree,NW), SON(tree,NE), SON(tree,SW), SON(tree,SE)
   represents these sons and assigns the pointers to the neighbors among them-
   selves. */
begin
    node tree;
        /*Create FOUR sons for the node tree */
    SON(tree,NW) ← create_node()
    SON(tree,NE) ← create_node()
    SON(tree,SW) ← create_node()
    SON(tree,SE) ← create_node()
        /* For NW son Eastern and Southern neighbors */
    NEIGHBOR(SON(tree,NW),E) ← SON(tree,NE)
    NEIGHBOR(SON(tree,NW),S) ← SON(tree,SW)
        /* For NE son Western and Southern neighbors */
    NEIGHBOR(SON(tree,NE),W) ← SON(tree,NW)
    NEIGHBOR(SON(tree,NE),S) ← SON(tree,SE)
        /* For SW son Northern and Eastern neighbors */
    NEIGHBOR(SON(tree,SW),N) ← SON(tree,NW)
    NEIGHBOR(SON(tree,SW),E) ← SON(tree,SE)
        /* For SE son Northern and Western neighbors */
    NEIGHBOR(SON(tree,SE),N) ← SON(tree,NE)
    NEIGHBOR(SON(tree,SE),W) ← SON(tree,SW)
end
```

**Algorithm 3.2 :** Creating sons for a node.

`GRAY` the procedure `CREATE_SONS`, with pointer to the current node, is invoked to create its sons and assign pointers to neighbors among themselves. At this stage, neighbors can be accessed through temporary node and sons through current node. Then the procedure `PROCESS_GRAY` with pointers to the current node and temporary node is invoked to obtain northern and western neighbors for the sons of `GRAY` node. The assignment of eastern and southern neighbors for the sons is postponed as they are not visited yet. These neighbors are assigned when corresponding neighboring leaf nodes are visited. Then the procedure `vquadtree` calls itself recursively for the sons of the `GRAY` node with quarter the size of the current node and pointer to current node. If the current node is leaf (`BLACK` or `WHITE`) the procedure `PROCESS_LEAF`

with a pointer to the current node is invoked to compute the eastern and southern neighbors for the sons of western and northern neighbors of the leaf node. The formal description of the procedure `vquadtree` is shown in Algorithm 3.1.

```
procedure PROCESS_GRAY(tree, temp)
/*This procedure computes northern and western
  neighbors for the sons of gray node. */
begin
  node tree,temp;
  direction D;
  /*The North and West neighbors of this node are already visited. Assign
    these nodes as the neighbors for the sons of the current node. */
  for D in {"N","W"} do
  if not NULL(NEIGHBOR(temp,D)) then
      if GRAY(NEIGHBOR(temp,D)) then
        begin
          /*The adjacent in direction D is a GRAY node. Assign the sons
            of this node as neighbors of the current node's two sons.*/
          NEIGHBOR(SON(tree,QUAD(D,CSIDE(D)),D)) ←
                    SON(NEIGHBOR(temp,D),QUAD(OPSIDE(D),CSIDE(D)))
          NEIGHBOR(SON(tree,QUAD(D,CCSIDE(D)),D) ←
                    SON(NEIGHBOR(temp,D),QUAD(OPSIDE(D),CCSIDE(D)))
        end
      else
        begin
          /*The adjacent in direction D is a LEAF node. Assign
            this node as neighbor of the current node's two sons.*/
          NEIGHBOR(SON(tree,QUAD(D,CSIDE)),D) ← NEIGHBOR(temp,D)
          NEIGHBOR(SON(tree,QUAD(D,CCSIDE)),D) ← NEIGHBOR(temp,D)
          /*If the size of the adjacent node is same as current node,
            assign the current node as the neighbor of adjacent node
            in opposite direction of D. */
          if SIZE(tree)=SIZE(NEIGHBOR(temp,D) then
                NEIGHBOR(NEIGHBOR(temp,D),OPSIDE(D)) ←tree
        end
      endif
  endif
end
```

**Algorithm 3.3 :** Processing a GRAY node

The procedure CREATE_SONS is invoked with a pointer to the node for which sons has to be created. It creates four sons to represent NW, NE, SW, SE quadrants of the region. It also assigns the neighbors among the sons. The NW son of the gray node is assigned as western neighbor for NE, and northern neighbor for SW sons of the node. For example during the construction of v-Quadtree shown in Figure 3.2, node B is assigned as western neighbor for node C, and northern neighbor for node E. Similarly NE, SW, SE sons are assigned as neighbors for the two adjacent nodes in the appropriate directions. The formal description of the procedure CREATE_SONS is shown in Algorithm 3.2.

```
procedure PROCESS_LEAF(tree)
/* This procedure computes the eastern and southern neighbors for
   the sons of western and northern neighbors of leaf node. */
begin
   node tree;
   direction D;
   /* At this stage the neighbors for the current node in North and West
      directions are already assigned. If the neighbor is GRAY node assign
      current node as the neighbor in opposite direction of D for the descendant
      leaf nodes of neighbor.e.g., SW,SE sons for North Direction. */
   for D in {"N","W"} do
   if not NULL(NEIGHBOR(tree,D) then
         if GRAY(NEIGHBOR(tree,D)) then
               RecursiveSet(NEIGHBOR(tree,D),QUAD(OPSIDE(D),CSIDE(D)),
                            QUAD(OPSIDE(D),CCSIDE(D)),OPSIDE(D),tree)
         else if SIZE(tree)=SIZE(NEIGHBOR(tree,D)) then
               /* If the neighbor is leaf node of same size assign current node
                  as the adjacent for this node in direction opposite to D. */
               NEIGHBOR(NEIGHBOR(tree,D),OPSIDE(D)) ← tree
   endif
end
```

**Algorithm 3.4 :** Processing a LEAF node

The procedure PROCESS_GRAY is invoked with pointers to the node to be processed and the temporary node which has neighbor information. It finds northern neighbors

```
procedure RecursiveSet(from,Q1,Q2,D,to)
/* This procedure assigns the node 'to' as neighbor in direction
   D for the descendant Q1,Q2 leaf nodes of node 'from'. */
begin
      node from, to;
      quadrant Q1,Q2;
      direction D;
      if GRAY(from) then
            RecursiveSet(SON(from,Q1),Q1,Q2,D,to)
            RecursiveSet(SON(from,Q2),Q1,Q2,D,to)
      else NEIGHBOR(from,D) ← to
end
```

**Algorithm 3.5 :** RecursiveSet

for NW and NE sons, western neighbors for NW and SW sons as they are already visited. The northern neighbors for NW and NE sons are located in the following way. If the northern neighbor does not exist, then the northern neighbors for NW, NE sons doesn't exist and they are points to NIL. If the northern neighbor exist then there are two cases. In the first case, if the type of northern neighbor is GRAY then SW son of neighbor is assigned as northern neighbor for NW son of current node and SE son of neighbor is assigned as northern neighbor for NE son of current node. For example, during the construction of v-Quadtree shown in Figure 3.2, the northern neighbor for node E is node B and it is GRAY. Assign the nodes 3 and 4 as the northern neighbors for for the nodes 12 and 13, respectively. In the second case, if the type of northern neighbor is BLACK or WHITE then assign this node as the northern neighbor of NW and NE sons, and if the neighbor is of same size then assign the current node as southern neighbor for this node. Similarly it finds western neighbors for NW and SW sons. For example, during the construction of v-Quadtree shown in Figure 3.2, the western neighbor for node D is node 2 and it is BLACK. Assign node 2 as the western neighbor for nodes, NW, SW sons of the current node i.e., nodes 5 and 6. Since the size of the node 2 is same as the node D, assign node D as the eastern neighbor for node 2. The formal description of the procedure PROCESS_GRAY is shown in Algorithm 3.3.

The procedure PROCESS_LEAF is invoked with a pointer to the node to be processed.

It finds the eastern and southern neighbors for the western and northern neighbors of a node, respectively. If the current node is of type `BLACK` or `WHITE` then no further division takes place. It is already pointed to its neighbors in four directions. But if the northern neighbor is of type `GRAY` then all its `SW` and `SE` descendant leaf nodes should point to the current node in south direction. This is done by invoking the procedure `RecursiveSet`. The formal description of the procedure `RecursiveSet` is shown in Algorithm 3.5. For example, during the construction of `v-Quadtree` shown in Figure 3.2, the northern neighbor for node `10` is node `D` and it is `GRAY`. So recursively assign the current node, i.e., node `10` as the southern neighbor for the descendant `SW, SE` leaf nodes of node `C`, i.e., node `7` and node `8`. If the northern neighbor is of type `WHITE` or `BLACK` and its size is same as the current node then current node is assigned as southern neighbor for this node. For example, the northern neighbor for node `12` is node `3` and it is `WHITE`, assign node `12` as the southern neighbor for node `3`. Similarly it performs these operations in western direction. The formal description of the procedure `PROCESS_LEAF` is shown in Algorithm 3.4.

### 3.2.3 Maintaining v-Quadtree

Suppose some homogeneous region in the image changes to non-homogeneous, then instead of constructing the quadtree from the scratch, we can modify the existing quadtree to reflect the changes. When a homogeneous node changes its membership, `v-Quadtree` can be maintained by applying the Algorithm 3.1 with little modification. The procedures `PROCESS_GRAY` and `PROCESS_LEAF`, besides looking into the northern and western neighbors, also look into the eastern and southern neighbors, provided if the type of these neighbors is known. Otherwise, these nodes are not visited already and the links will be set when those nodes are visited. The Algorithm 3.1 is invoked with a pointer to the transformed node.

Figure 3.3 shows the transformation of `BLACK` node to `GRAY` node. Figure 3.3(a) shows an image while Figure 3.3(c) shows its quadtree representation. Suppose the block represented by the node `10` in Figure 3.3(a) changes as shown in the Figure 3.3(b). Instead of constructing the whole quadtree, we modify the quadtree represented by Figure 3.3(c) to the quadtree represented by Figure 3.3(d) and still maintain

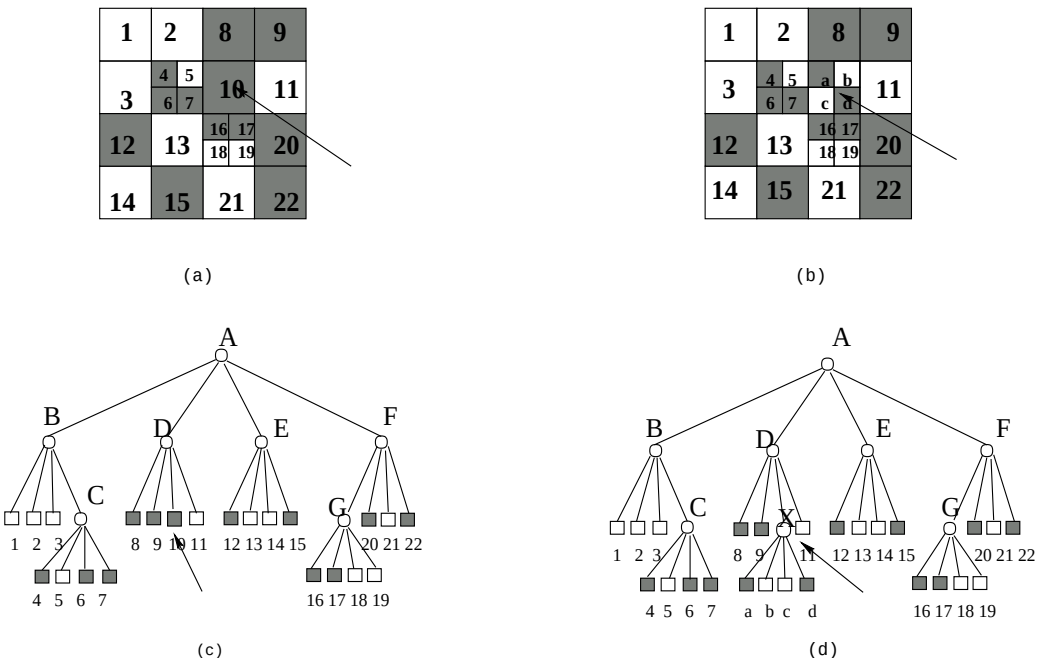



Figure 3.3: Transformation of a BLACK node to GRAY node.

the structure of the `v-Quadtree`. This is done by performing the following steps. Replace node `10` by node `X` and create four sons for this node namely `a, b, c, d` for `NW, NE, SW, SE` quadrants, respectively. Assign neighbors among themselves. The northern neighbor for node `10` is node `8` and it is `BLACK` node, assign this node as northern neighbor for nodes `a` and `b`. The eastern neighbor for node `10` is node `11` and it is `WHITE`, assign this node as eastern neighbor for nodes `b` and `d`. The western neighbor for node `10` is node `C` and it is `GRAY`, assign the `NE` son of this node, which is `5`, as the western neighbor for `NW` son of node `X` which is `a`. Similarly assign node `7` as the western neighbor for node `c`. The southern neighbor for node `10` is node `G` and it is `GRAY`, assign the `NW` son of this node, which is `16`, as the southern neighbor for `SW` son of node `X` which is `c`. Similarly assign node `17` as the southern neighbor for node `d`.

## 3.3 Comparison with regular Quadtree

The number of nodes that are traversed to find a neighbor in horizontal or vertical direction, in regular quadtree, on the average is bounded by 5 [4]. In v-Quadtree, the number of nodes that are traversed to find a neighbor for a leaf node is 1, as it has direct link to the neighbors in horizontal and vertical directions. For example, given the image shown in Figure 2.2(a) the regular quadtree traverses 156 nodes to find the neighbors in all four directions. If we use v-Quadtree this number is reduced to 64. If the number of levels in the quadtree increases the number of nodes traversed to find neighbors in regular quadtree increases very much, as it has to go many levels upward and traverse downwards to reach the neighbors. In case of v-Quadtree, even if the number of levels increase it does not affect the number of nodes traversed to find the neighbor, as it is not going to traverse many levels upwards and downwards.

## 3.4 Summary

In this chapter, we proposed a variation of quadtree data structure. This quadtree has links to four neighbors in horizontal and vertical directions. Hence neighbors are accessed by passing through the links, instead of traversing many nodes between neighboring nodes so as the case in regular quadtree. An algorithm is presented for construction of this quadtree. A technique is proposed for maintaining this quadtree when some homogeneous region in the image undergoes changes.

# Chapter 4

# Maintaining Perimeter of Image

Maintaining Perimeter is a problem encountered during the processing and analyzing remotely sensed data. As input, we normally have a set of $\mathtt{p}$ images $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_2, ..., \mathcal{I}_\mathtt{p}\}$ in which any pair of consecutive images $\mathcal{I}_\mathtt{j}$ and $\mathcal{I}_{\mathtt{j}+1}$ are very similar to each other and all the images have to undergo similar kind of processing. Traditionally, batch algorithms that process each image as separate entity have been employed. In order to find the perimeter of each image, the standard batch approach is to find the perimeter for each image separately. In contrast, dynamic approach is to update perimeter to represent the perimeter of the new image.

## 4.1 Dynamic Algorithm for Maintaining Perimeter in Quadtrees

The input to the Maintaining Perimeter problem is the set of images $\mathcal{I}$ and the hierarchical representation of the first image $\mathcal{I}_1$ in the form of quadtree data structure. Also, we assume that we are given the perimeter of the first image. As a preprocessing phase, we have to run an algorithm to find the perimeter on the quadtree representation of the image $\mathcal{I}_1$. For this we use the algorithm suggested by Samet [12] (details presented in Chapter 2).

The dynamic approach is presented to cater for unit changes in the quadtree, i.e., changes occurring in a particular homogeneous region only. The difference between

pair of images $\mathcal{I}_{\mathtt{j}}$ and $\mathcal{I}_{\mathtt{j+1}}$ can be decomposed into changes restricted to leaf nodes in the quadtree which represent homogeneous regions. Then each affected homogeneous region is treated separately by the dynamic approach. Henceforth, when we refer to a homogeneous region, we mean a region corresponding to a leaf node in the quadtree.

## 4.2 Region Transformations

From image $\mathcal{I}_j$ to $\mathcal{I}_{j+1}$, pixels may change their membership from $\mathcal{B}$ to $\mathcal{W}$ or vice-versa. In this work, we are presenting algorithms that handle changes concentrated in a particular homogeneous region represented by a leaf node in the quadtree. Effectively, all the changes can be grouped into those affecting a single region and then we can use our algorithms to handle the changes in every affected region. We assume that the changed pixels have already been grouped and we have identified the affected regions before we maintain perimeter in our quadtree. The region transformations can be of four types :

- $\mathcal{BTW}$ **Transformation** : In this transformation, all the BLACK pixels in the homogeneous region change to WHITE pixels. This means that the BLACK node will have to change its `NODETYPE` to WHITE to reflect this change, apart from updating the pixel information stored in the node.

- $\mathcal{WTB}$ **Transformation** : This transformation is exactly the opposite of $\mathcal{BTW}$ transformation. Here, all the WHITE pixels in the homogeneous region change to BLACK pixels. This means that the corresponding node in the quadtree will have to change its `NODETYPE` to BLACK.

- $\mathcal{BTG}$ **Transformation** : In this transformation, some BLACK pixels in a homogeneous region change to WHITE pixels. This makes the region non-homogeneous and has to be decomposed further till we get homogeneous regions. In the quadtree, it means that the corresponding BLACK node is transformed into a GRAY node. Apart from updating the BLACK node to GRAY node, we have to build the subquadtree with the affected node as the root. This could

be easily done using the algorithm suggested by Shaffer *et al* [13] and update the quadtree by modifying the pointers.

- $\mathcal{WTG}$ **Transformation** : In this transformation, some WHITE pixels in a homogeneous region change to BLACK pixels. As was the case in $\mathcal{BTG}$, we have to decompose the region further. In the quadtree, this requires changing the WHITE node to GRAY node and build the subquadtree with the affected node as the root. The subquadtree can be built in the same way as done in $\mathcal{BTG}$ transformation.

## 4.3 Algorithm

Suppose there is an image, which is represented by a quadtree and there is some change in the homogeneous region. The problem is to compute the perimeter of the changed image. As we have seen in the previous section, there are four types of possible transformations. Hence, we have to consider all the four cases for finding the perimeter. In this section we propose algorithms for maintaining perimeter of the image.

### 4.3.1 $\mathcal{BTW}$ Transformation

The $\mathcal{BTW}$ transformation is demonstrated with the help of of Figure 4.1. The original image and the modified image are presented in Figure 4.1(a) and (b) respectively. Their corresponding quadtree representations are given in Figure 4.1(c) and (d) respectively. Transformation of a node from BLACK type to WHITE type may cause merging of nodes as shown in the figure.

The algorithm updates perimeter under $\mathcal{BTW}$ transformation. It visits the neighbors in horizontal and vertical directions of transformed node in the original quadtree. For each of the visited neighbor that is BLACK, the length of the common boundary is added to the perimeter, and for each of the visited neighbor that is WHITE the length of the common boundary is subtracted from the perimeter. For example, given node `12` in Figure 4.1(c), nodes `7, 10, 13,` and `19` are visited. Of these nodes, `7` and `19` are BLACK and thus the lengths of the boundary segments between nodes

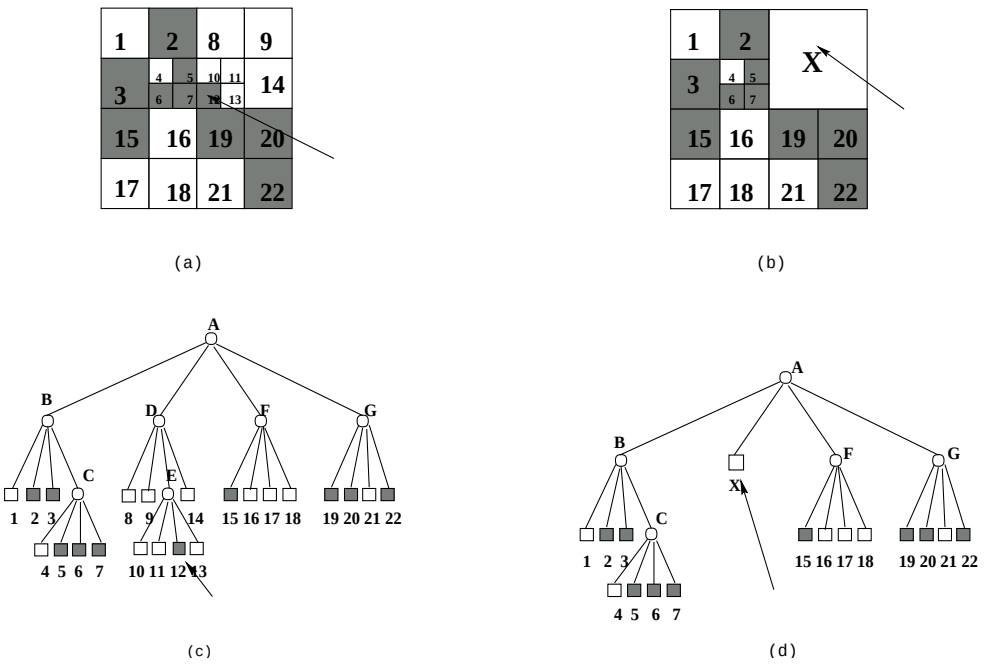



Figure 4.1: $\mathcal{BTW}$ Transformation.

(12, 7) and (12, 19) are added to the perimeter. And nodes 10 and 13 are WHITE and thus the lengths of the boundary segments between nodes (12, 10) and (12, 13) are subtracted from the perimeter. At last the quadtree is updated to represent the modified image.

The main procedure, BTW_PERIMETER, is given in Algorithm 4.1 and is invoked with a pointer to the transformed node representing the transformed region and integer corresponding to the level of the transformed node and the perimeter of the image before transformation. BTW_PERIMETER explores the adjacent neighbors for the transformed node. Let the transformed node be P, GTEQUAL_ADJ_NEIGHBOR locates a neighboring node, say Q, of greater or equal size along a specified direction, say D. If Q is BLACK, then the size of P is added to the perimeter. If Q is WHITE, then the size of P is subtracted from the perimeter. If the boundary of P in direction D is on the border of the image, then no neighbor exists in the specified direction and NULL is returned. In such a case the size of P is subtracted from the perimeter. If the boundary of P in direction D is not on the border of the image, and no neighboring

```
integer procedure BTW_PERIMETER(P,LEVEL,LEN)
/* This procedure updates the perimeter of the image under BTW transforma-
   tion. P is the transformed node that spans 2↑LEVEL by 2↑LEVEL space. LEN
   is the perimeter before transformation. This procedure returns the perimeter
   of the modified image.
begin
   node P;
   integer LEVEL,LEN;
   direction D;
   Let P be the transformed node
   for D in {"N","E","S","W"}
      begin
         Q ← GTEQUAL_ADJ_NEIGHBOR(P,D)
         LEN ← LEN + if BLACK(Q) then 2↑LEVEL
                     else if NULL(Q) or WHITE(Q) then -(2↑LEVEL)
                     else BTW_ADJACENT(Q,QUAD(OPSIDE(D),CSIDE(D)),
                                       QUAD(OPSIDE(D),CCSIDE(D)),LEVEL)
      end
   return(LEN)
end
```

**Algorithm 4.1 :** $\mathcal{B}TW$ Transformation

BLACK or WHITE node exists satisfying our size criteria, then a pointer to a GRAY node of equal size is returned. In such a case procedure `BTW_ADJACENT`, given in Algorithm 4.2, continues the search by examining all nodes of smaller size that are adjacent to `P`'s boundary in direction `D` and adding the sizes of all such nodes that are BLACK and subtracting the sizes of all such nodes that are WHITE.

### 4.3.2 $\mathcal{W}TB$ Transformation

The $\mathcal{WTB}$ transformation is demonstrated with the help of of Figure 4.2. The original image and the modified image are presented in Figure 4.2(a) and (b) respectively. Their corresponding quadtree representations are given in Figure 4.2(c) and (d) respectively. Transformation of a node from WHITE type to BLACK type may cause merging of nodes.

```
integer procedure BTW_ADJACENT(P,Q1,Q2,LEVEL)
/* Find all leaves in quadrants Q1 and Q2 of the sub-
   quadtree rooted at node P of size 2↑LEVEL */
begin
   node P;
   quadrant Q1,Q2;
   integer LEVEL;
   return(if GRAY(P) then BTW_ADJACENT(SON(P,Q1),Q1,Q2,LEVEL-1)
                        + BTW_ADJACENT(SON(P,Q2),Q1,Q2,LEVEL-1)
          else if BLACK(P) then 2↑LEVEL
          else -(2↑LEVEL))
end
```

**Algorithm 4.2 :** BTW_ADJACENT

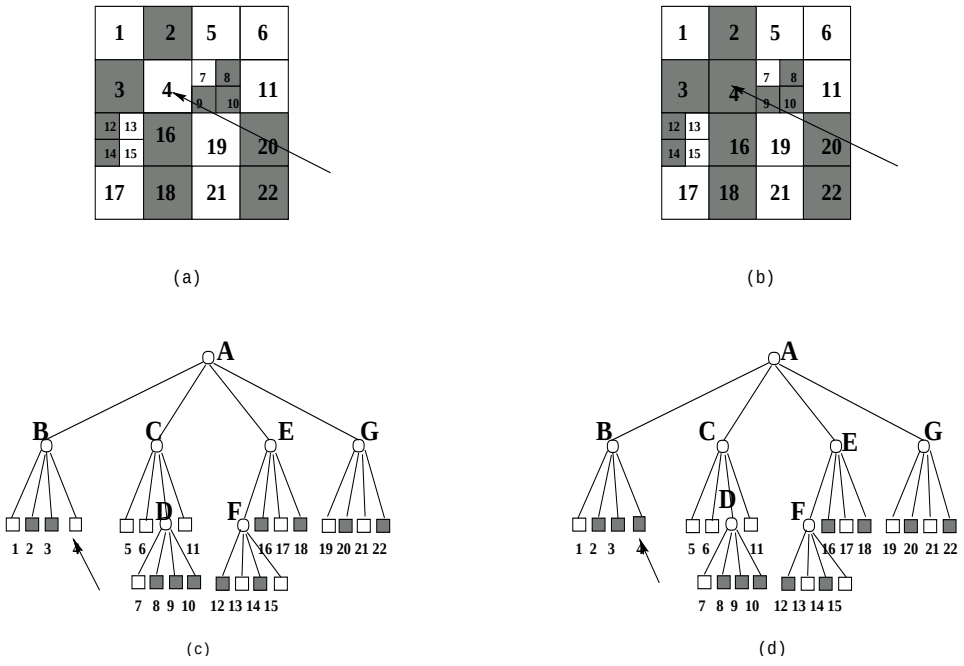



Figure 4.2: $\mathcal{WTB}$ Transformation.

The algorithm updates perimeter under $\mathcal{WTB}$ transformation. It visits the neighbors in horizontal and vertical directions of transformed node in the original quadtree. For each of the visited neighbor that is BLACK, the length of the common boundary

```
integer procedure WTB_PERIMETER(P,LEVEL,LEN)
/* This procedure updates the perimeter of the image under WTB transforma-
   tion. P is the transformed node that spans 2↑LEVEL by 2↑LEVEL space. LEN
   is the perimeter before transformation. This procedure returns the perimeter
   of the modified image.
begin
   node P;
   integer LEVEL;
   direction D;
   Let P be the transformed node
   for D in {"N","E","S","W"}
      begin
         Q ← GTEQUAL_ADJ_NEIGHBOR(P,D)
         LEN ← LEN + if BLACK(Q) then -(2↑LEVEL)
                     else if NULL(Q) or WHITE(Q) then 2↑LEVEL
                     else WTB_ADJACENT(Q,QUAD(OPSIDE(D),CSIDE(D)),
                                       QUAD(OPSIDE(D),CCSIDE(D)),LEVEL)
      end
   return(LEN)
end
```

**Algorithm 4.3 :** $\mathcal{W}TB$ Transformation

is subtracted from the perimeter, and for each of the visited neighbor that is WHITE the length of the common boundary is added to the perimeter. For example, given node 4 in Figure 4.2(c), nodes 2, 3, 7, 9 and 16 are visited. Of these nodes, 2, 3, 9 and 16 are BLACK and thus the lengths of the boundary segments between nodes (4, 2), (4, 3), (4, 9) and (4, 16) are subtracted from the perimeter. And node 7 is WHITE and thus the length of the boundary segment between nodes (4, 7) is added to the perimeter. At last the quadtree is updated to represent the modified image.

The main procedure, `WTB_PERIMETER`, is given in Algorithm 4.3 and is invoked with a pointer to the transformed node representing the transformed region and integer corresponding to the level of the transformed node and the perimeter of the image before transformation. `WTB_PERIMETER` explores the adjacent neighbors for the transformed node. Let the transformed node be `P`, `GTEQUAL_ADJ_NEIGHBOR` locates a neighboring node, say `Q`, of greater or equal size along a specified direction, say `D`. If

```
integer procedure WTB_ADJACENT(P,Q1,Q2,LEVEL)
/* Find all leaves in quadrants Q1 and Q2 of the sub-
   quadtree rooted at node P of size 2↑LEVEL */
begin
   node P;
   quadrant Q1,Q2;
   integer LEVEL;
   return(if GRAY(P) then WTB_ADJACENT(SON(P,Q1),Q1,Q2,LEVEL-1)
                        + WTB_ADJACENT(SON(P,Q2),Q1,Q2,LEVEL-1)
          else if BLACK(P) then -(2↑LEVEL)
          else 2↑LEVEL)
end
```

**Algorithm 4.4 :** WTB_ADJACENT

`Q` is BLACK, then the size of `P` is subtracted from the perimeter. If `Q` is WHITE, then the size of `P` is added to the perimeter. If the boundary of `P` in direction `D` is on the border of the image, then no neighbor exists in the specified direction and NULL is returned. In such a case the size of `P` is added to the perimeter. If the boundary of `P` in direction `D` is not on the border of the image, and no neighboring BLACK or WHITE node exists satisfying our size criteria, then a pointer to a GRAY node of equal size is returned. In such a case procedure `WTB_ADJACENT`, given in Algorithm 4.4, continues the search by examining all nodes of smaller size that are adjacent to `P`'s boundary in direction `D` and subtracting the sizes of all such nodes that are BLACK and adding the sizes of all such nodes that are WHITE.

### 4.3.3 $\mathcal{BTG}$ Transformation

The $\mathcal{BTG}$ transformation is demonstrated with the help of of Figure 4.3. The original image and the modified image are presented in Figure 4.3(a) and (b) respectively. Their corresponding quadtree representations are given in Figure 4.3(c) and (d) respectively. When a WHITE node changes to GRAY node, we have to scan the region and construct a subquadtree. Then we replace the leaf node representing that region by the subquadtree obtained.

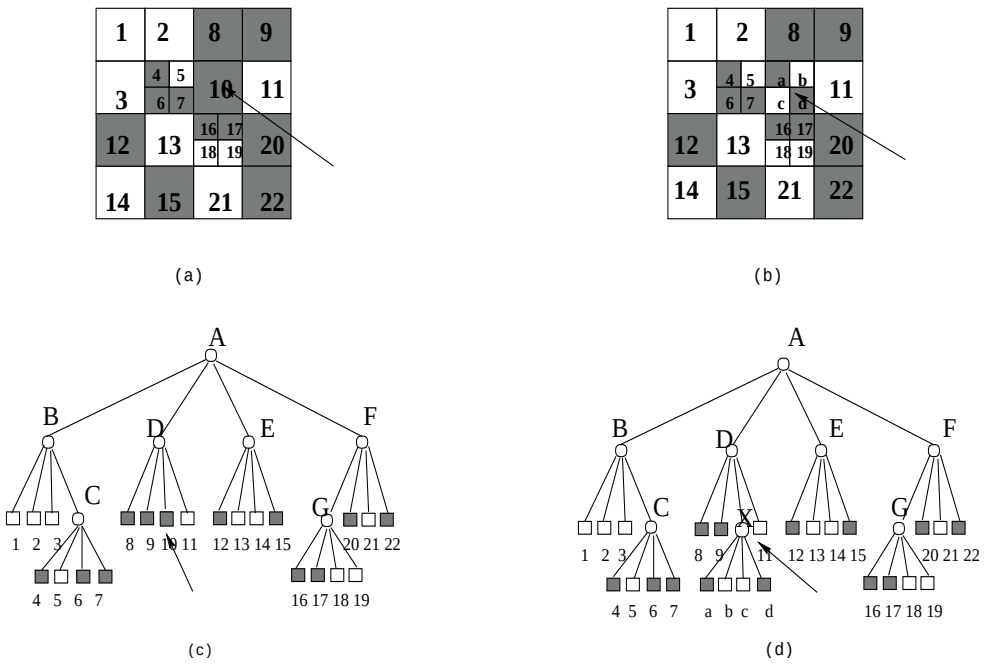



Figure 4.3: $\mathcal{BTG}$ Transformation.

The algorithm updates perimeter under $\mathcal{BTG}$ transformation. It visits the neighbors in horizontal and vertical directions of transformed node in the modified quadtree. For each of the visited neighbor that is WHITE, the length of the common boundary is subtracted from the perimeter, and for each of the visited neighbor that is BLACK, it visits neighbors towards the transformed node. If this neighbor is WHITE then the length of the common boundary is added to the perimeter. For example, given node `X` in Figure 4.3(d), nodes `5, 7, 8, 11, 16` and `17` are visited. Of these nodes, `5` and `11` are WHITE and thus the lengths of the boundary segments between nodes (`X`, `5`) and (`X`, `11`) are subtracted from the perimeter. And nodes `7, 8, 16` and `17` are BLACK and thus the lengths of the common boundary between these nodes and WHITE neighbors in the direction of transformed node (nodes `b` and `c`) are added to the perimeter. The perimeter of this transformed region, surrounded by other regions, is computed and added to the perimeter which yields the perimeter of the transformed image.

The main procedure, `BTG_PERIMETER`, is given in Algorithm 4.5 and is invoked

```
integer procedure BTG_PERIMETER(P,LEVEL)
/* This procedure updates the perimeter of the image under BTG transformation.
   P is the transformed node that spans 2↑LEVEL by 2↑LEVEL space. LEN is
   the perimeter before transformation. This procedure returns the perimeter of
   the modified image.
begin
   node P,Q,R;
   integer LEVEL;
   direction D;
   Let P be the transformed node
   for D in {"N","E","S","W"}
      begin
        Q ← GTEQUAL_ADJ_NEIGHBOR(P,D)
        if BLACK(Q) then
           begin
             R ← GTEQUAL_ADJ_NEIGHBOR(Q,OPSIDE(D))
             if WHITE(R) then LEN ← LEN + 2↑LEVEL
             else if GRAY(R) then
                       LEN ← LEN + BTG_BLK_ADJACENT(R,QUAD(D,CSIDE(D)),
                                                    QUAD(D,CCSIDE(D)),LEVEL)
           end
        else if WHITE(Q) or NULL(Q) then LEN = LEN - 2↑LEVEL
        else if GRAY(Q) then
                  LEN ← LEN + BTG_ADJACENT(Q,QUAD(OPSIDE(D),CSIDE(D)),
                                  QUAD(OPSIDE(D),CCSIDE(D)),OPSIDE(D),LEVEL)
      end
   PERIMETER(P,LEVEL)
end
```

**Algorithm 4.5 :** $\mathcal{B}TG$ Transformation

with a pointer to the transformed node representing the transformed region in the modified quadtree and integer corresponding to the level of the transformed node and the perimeter of the image before transformation. `BTG_PERIMETER` explores the adjacent neighbors for the transformed node. Let the transformed node be `P`, `GTEQUAL_ADJ_NEIGHBOR` locates a neighboring node, say `Q`, of greater or equal size along a specified direction, say `D`. If `Q` is WHITE, then the size of `P` is subtracted from the perimeter. If `Q` is BLACK, then it finds the neighbors in the opposite direction

```
integer procedure BTG_ADJACENT(P,Q1,Q2,D,LEVEL)
/* Find all leaves in quadrants Q1 and Q2 of the subquadtree rooted at
   node P of size 2↑LEVEL. It uses direction D to perform operations on
   the BLACK node. */
begin
   node P;
   quadrant Q1,Q2;
   integer LEVEL;
   direction D;
   if GRAY(P) then
         return(BTG_ADJACENT(SON(P,Q1),Q1,Q2,D,LEVEL-1)
                + BTG_ADJACENT(SON(P,Q2),Q1,Q2,D,LEVEL-1))
   else if BLACK(P) then
      begin
         R ← GTEQUAL_ADJ_NEIGHBOR(P,D)
         if WHITE(R) then return(2↑LEVEL)
         else if GRAY(R) then
               return(BTG_BLK_ADJACENT(R,QUAD(D,CSIDE(D)),
                                       QUAD(D,CCSIDE(D)),LEVEL))
      end
   else if WHITE(P) then return(- 2↑LEVEL)
end
```

**Algorithm 4.6 :** BTG_ADJACENT

of `D`, i.e., in the direction of transformed region. If this neighbor is WHITE then the length of the common boundary is added to the perimeter. If it is BLACK nothing is done, and if it is GRAY then procedure `BTG_BLK_ADJACENT`, given in Algorithm 4.7, continues the search by examining all nodes of smaller size adjacent to `Q` in the opposite direction of `D` and adding the sizes of all such nodes that are WHITE. If the boundary of `P` in direction `D` is on the border of the image, then no neighbor exists in the specified direction and NULL is returned. In such a case the size of `P` is subtracted from the perimeter. If the boundary of `P` in direction `D` is not on the border of the image, and no neighboring BLACK or WHITE node exists satisfying our size criteria, then a pointer to a GRAY node of equal size is returned. In such a case procedure `BTG_ADJACENT`, given in Algorithm 4.6, continues the search by examining all nodes of smaller size that are adjacent to `P`'s boundary in direction `D` and

```
integer procedure BTG_BLK_ADJACENT(P,Q1,Q2,LEVEL)
/* Find all WHITE leaves in quadrants Q1 and Q2 of the
   subquadtree rooted at node P of size 2↑LEVEL */
begin
   node P;
   quadrant Q1,Q2;
   integer LEVEL;
   return(if GRAY(P) then BTG_BLK_ADJACENT(SON(P,Q1),Q1,Q2,LEVEL-1)
                        + BTG_BLK_ADJACENT(SON(P,Q2),Q1,Q2,LEVEL-1)
          else if WHITE(P) then 2↑LEVEL)
end
```

**Algorithm 4.7 :** BTG_BLK_ADJACENT

performs operations as above. It then invokes `PERIMETER`, given in Algorithm 2.2, which finds the perimeter of the transformed region surrounded by other regions. It adds returned value to the length of the perimeter.

### 4.3.4 $\mathcal{WTG}$ Transformation

The $\mathcal{WTG}$ transformation is demonstrated with the help of of Figure 4.4. The original image and the modified image are presented in Figure 4.4(a) and (b) respectively. Their corresponding quadtree representations are given in Figure 4.4(c) and (d) respectively. Similar to the $\mathcal{BTG}$ transformation, we have to construct a subquadtree for the affected region and replace the leaf node with the obtained subquadtree.

The algorithm updates perimeter under $\mathcal{WTG}$ transformation. It visits the neighbors in horizontal and vertical directions of transformed node in the modified quadtree. For each of the visited neighbor that is BLACK, it visits neighbors towards the transformed node. If this neighbor is BLACK then the length of the common boundary is subtracted from the perimeter. For example, given node `X` in Figure 4.3(d), nodes `5, 7, 8, 11, 16` and `17` are visited. Of these nodes, `7, 16` and `17` are BLACK and thus the lengths of the common boundary between these nodes and BLACK neighbors in the direction of transformed node (nodes `b` and `c`) are subtracted from the perimeter. The perimeter of this transformed region, surrounded by other regions, is computed and added to the perimeter which yields the perimeter of the transformed image.

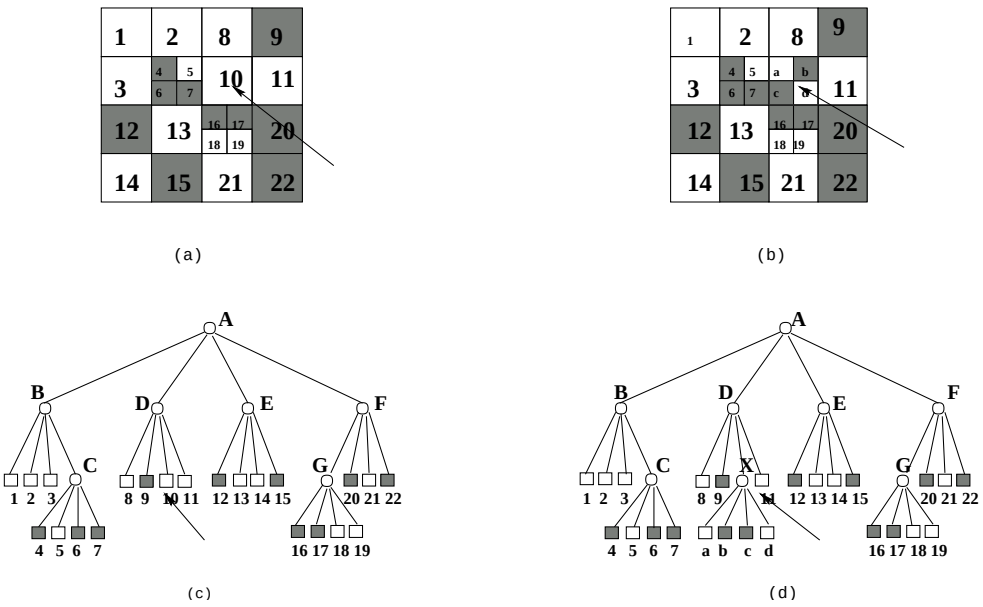



Figure 4.4: $\mathcal{WTG}$ Transformation.

The main procedure, `WTG_PERIMETER`, is given in Algorithm 4.8 and is invoked with a pointer to the transformed node representing the transformed region in the modified quadtree and integer corresponding to the level of the transformed node and the perimeter of the image before transformation. `WTG_PERIMETER` explores the adjacent neighbors for the transformed node. Let the transformed node be `P`, `GTEQUAL_ADJ_NEIGHBOR` locates a neighboring node, say `Q`, of greater or equal size along a specified direction, say `D`. If `Q` is BLACK, then it finds the neighbors in the opposite direction of `D`, i.e., in the direction of transformed region. If this neighbor is BLACK then the length of the common boundary is subtracted from the perimeter. If it is WHITE nothing is done, and if it is GRAY then procedure `WTG_BLK_ADJACENT`, given in Algorithm 4.10, continues the search by examining all nodes of smaller size adjacent to `Q` in the opposite direction of `D` and subtracting the sizes of all such nodes that are BLACK. If the boundary of `P` in direction `D` is on the border of the image, then no neighbor exists in the specified direction and NULL is returned. In such a case the size of `P` nothing is done as it will be added when `PERIMETER` applied on the transformed region. If the boundary of `P` in direction `D` is not on the border of the image, and no neighboring BLACK or WHITE node exists satisfying our size criteria,

```
integer procedure WTG_PERIMETER(P,LEVEL)
/* This procedure updates the perimeter of the image under WTG transforma-
   tion. P is the transformed node that spans 2↑LEVEL by 2↑LEVEL space. LEN
   is the perimeter before transformation. This procedure returns the perimeter
   of the modified image.
begin
   node P,Q,R;
   integer LEVEL;
   direction D;
   Let P be the transformed node
   for D in {"N","E","S","W"}
      begin
         Q ← GTEQUAL_ADJ_NEIGHBOR(P,D)
         if BLACK(Q) then
            begin
               R ← GTEQUAL_ADJ_NEIGHBOR(Q,OPSIDE(D))
               if BLACK(R) then LEN ← LEN - 2↑LEVEL
               else if GRAY(R) then
                        WTG_BLK_ADJACENT(R,QUAD(D,CSIDE(D)),
                                         QUAD(D,CCSIDE(D)),LEVEL)
            end
         else if GRAY(Q) then
               WTG_ADJACENT(Q,QUAD(OPSIDE(D),CSIDE(D)),
                            QUAD(OPSIDE(D),CCSIDE(D)),OPSIDE(D),LEVEL)
      end
   PERIMETER(P,LEVEL)
end
```

**Algorithm 4.8 :** $\mathcal{WTG}$ Transformation

then a pointer to a GRAY node of equal size is returned. In such a case procedure `WTG_ADJACENT`, given in Algorithm 4.9, continues the search by examining all nodes of smaller size that are adjacent to `P`'s boundary in direction `D` and performs operations as above. It then invokes `PERIMETER`, given in Algorithm 2.2, which finds the perimeter of the transformed region surrounded by other regions. It adds returned value to the length of the perimeter.

```
integer procedure WTG_ADJACENT(P,Q1,Q2,D,LEVEL)
/* Find all BLACK leaves in quadrants Q1 and Q2 of the subquadtree
   rooted at node P of size 2↑LEVEL. It uses direction D to perform oper-
   ations on these BLACK node. */
begin
   node P;
   integer LEVEL;
   quadrant Q1,Q2;
   direction D;
   if GRAY(P) then return(WTG_ADJACENT(SON(P,Q1),Q1,Q2,D,LEVEL-1)
                        + WTG_ADJACENT(SON(P,Q2),Q1,Q2,D,LEVEL-1))
   else if BLACK(P) then
      begin
           R ← GTEQUAL_ADJ_NEIGHBOR(P,D)
           if BLACK(R) then return(- 2↑LEVEL)
           else if GRAY(R) then
                     return(WTG_BLK_ADJACENT(R,QUAD(D,CSIDE(D)),
                                             QUAD(D,CCSIDE(D)),LEVEL))
      end)
end
```

**Algorithm 4.9 :** WTG_ADJACENT

```
integer procedure WTG_BLK_ADJACENT(P,Q1,Q2,LEVEL)
/* Find all BLACK leaves in quadrants Q1 and Q2 of the
   subquadtree rooted at node P of size 2↑LEVEL */
begin
   return(if GRAY(P) then WTG_BLK_ADJACENT(SON(P,Q1),Q1,Q2,LEVEL-1)
                        + WTG_BLK_ADJACENT(SON(P,Q2),Q1,Q2,LEVEL-1)
          else if BLACK(P) then -(2↑LEVEL))
end
```

**Algorithm 4.10 :** WTG_BLK_ADJACENT

## 4.4 Comparison with Batch Approach

Suppose we have an image set $\mathcal{I}$ in which consecutive images $\mathcal{I}_j$ and $\mathcal{I}_{j+1}$ are very similar to each other. In the batch approach for maintaining perimeter, we first build the quadtree and then compute the perimeter. This procedure is done for every

image in the set. Let us just concentrate on a pair of images $\mathcal{I}_j$ and $\mathcal{I}_{j+1}$. In order to build the quadtree of the image $\mathcal{I}_j$, we have to scan the whole image and using the algorithm suggested by Shaffer *et al* [13] in time bounded by $O(m)$ where $m$ is the number of nodes in the quadtree representation of that image. Let the quadtree has $\delta_O + \delta_B$ leaf nodes. Then the time required to construct the quadtree is bounded by $O(\delta_O + \delta_B)$ since the number of nodes in a quadtree is bounded by the number of leaf nodes it has. To find perimeter in the quadtree, the time complexity of the algorithm by Samet [12] is $O(\delta_O + \delta_B)$. This requires postorder traversal of the whole quadtree. We have to repeat the same proceedure for the next image $\mathcal{I}_{j+1}$. Creation of the quadtree of the modified image $\mathcal{I}_{j+1}$ is bounded by $O(\delta'_O + \delta'_B)$ if the quadtree has $\delta'_O + \delta'_B$ leaf nodes. Then finding perimeter in this quadtree takes $O(\delta'_O + \delta'_B)$ time. Hence the time complexity of the batch approach to update labels when a homomgeneous region undergoes change is $O(\delta_O + \delta_B + \delta'_O + \delta'_B)$.

In contrast, our approach for maintaining perimeter requires constructing the quadtree just once for the image set $\mathcal{I}$. We construct the quadtree for the first image and for successive images, we update the quadtree of the previous image. For this, we assume that instead of providing the whole images, we are just given the differences between an image and its predecessor image. So, we just have to update those portions of the quadtree that are undergoing changes, rather than building the whole quadtree from the scratch. Hence if the difference between consecutive images is not much, then only a minor portion of the quadtree need to be updated. The updation cost per unit change is bounded by $O(\delta_N)$ where $N$ is the number of leaf nodes in the updated portion of the quadtree.

Since the subquadtree with $\delta_N$ leaf nodes is a part of the quadtree of the modified image, $\delta_N \leq (\delta'_O + \delta'_B)$. Moreover, for large images, $\delta_N$ is very small compared to $(\delta_O + \delta_B)$. This means that the number of nodes visited by our algoithm has a bound lower than that of the batch method. If the number of changes involved is very few i.e. the consecutive images are very similar, our algorithm will not only save time in updating the quadtree, but also update the perimeter faster than the classical algorithm on an average. Theoretically, in the worst case, the updation of labels

in our algorithm for maintaining perimeter can be as bad as the classical algorithm employed in the batch approach.

## 4.5 Summary

In this chapter, we presented a new technique to maintain the perimeter in quadtree representation of the image as it undergoes change. There are four possibilities that a homogeneous region undergo changes. The transformations are, a BLACK node may change its membership to WHITE or GRAY, a WHITE node may change its membership to BLACK or GRAY. Algorithms to handle each of the four transformations are presented.

# Chapter 5

# Conclusions and Future Work

In this thesis, we have dealt with the problem of efficient data structure for image processing and maintaining perimeter as the image undergoes slight change. Though we have mainly concentrated on remote sensing applications the technique presented in this work can be used in any general image processing application.

We proposed a new data structure, v-Quadtree, which is a variation of regular quadtree. With this data structure the time taken in finding the neighbors of region is minimized without the need of extra storage. We presented algorithms for construction and maintenance of this quadtree.

Extension of the quadtree to represent three-dimensional objects is *octree*. The *octree* is an approach to object representation similar to quadtree, and is based on the successive subdivision of an object array into octants. The technique used in v-Quadtree can be extended to octree representation of volume data.

We presented a technique to update perimeter in quadtree representation of the image, as it undergoes changes. There are four possibilities a homogeneous region undergoes changes. Algorithms to handle each of the four cases are presented.

The technique used in the updation of perimeter in quadtree representation of image, can be extended to the updation of surface area in octree representation of the three dimensional objects.

# Bibliography


[1] Hunter, G.M., and Steiglitz, K. Linear transformation of pictures represented by quad trees. *Computer Graphics and Image Processing*, 10:289–296, 1979.

[2] Rosenfeld, A. and John L. Pfaltz. Sequential operations in digital picture processing. *J. ACM*, 12(4):471–494, Oct. 1966.

[3] Samet, H. Connected component labeling using quadtrees. *J. ACM*, 23(3):487–501, 1981.

[4] Samet, H. Neighbor finding techniques for image represented by quadtrees. *Computer Graphics and Image Processing*, 18:37–57, 1982.

[5] Samet, H. *Applications of Spatial Data Structures.* Addison-Wesley, 1990.

[6] Samet, H. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[7] Samet, H. An algorithm for converting rasters to quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-3(1):93–95, Jan 1981.

[8] Samet, H. A top-down quadtree traversal algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-7(1):94–98, January 1985.

[9] Samet, H. Quadtree and related hierarchial data structures. *Computing Surveys*, 16(2):187–260, June 1984.

[10] Samet, H. Region representation: Quadtrees from binary arrays. *Computer Graphics and Image Processing*, 13(1):90–93, May 1980.

[11] Samet, H. Distance transform for images represented by quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4(3):298–303, May 1982.

[12] Samet, H. Computing perimeters of regions in images represented by quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-3(6):683–687, November 1981.

[13] Shaffer, C.A., and Samet, H. Optimal quadtree construction algorithms. *Computer Vision, Graphics and Image Processing*, 37:402–419, 1987.

[14] Vikrant Khanna. New algorithms for some image processing problems in remote sensing applications. *M.Tech Thesis, IIT Kanpur*, May 2000.