CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

Canada

National Library of Canada / Bibliothèque nationale du Canada

Ottawa, Canada
K1A 0N4

40

TC –

0-315-23238-2

CANADIAN THESES ON MICROFICHE SERVICE – SERVICE DES THÈSES CANADIENNES SUR MICROFICHE

PERMISION TO MICROFILM – AUTORISATION DE MICROFILMER

• Please print or type – Écrire en lettres moulées ou dactylographier

AUTHOR – AUTEUR

Full Name of Author – Nom complet de l'auteur

XIAONING WANG

Date of Birth – Date de naissance

June 3, 1953

Canadian Citizen – Citoyen canadien

☐ Yes Oui ☐ No / Non

Country of Birth – Lieu de naissance

People's Republic of China

Permanent Address – Résidence fixe

Site 60, Box 47
RR#5 Edmonton
Alberta T5P 4B7

THESIS – THÈSE

Title of Thesis – Titre de la thèse

Some new approaches for linear equations

Degree for which thesis was presented
Grade pour lequel cette thèse fut présentée

M.Sc.

Year this degree conferred
Année d'obtention de ce grade

Fall 1985

University – Université

University of Alberta

Name of Supervisor – Nom du directeur de thèse

Dr. Wayne A. Davis

AUTHORIZATION – AUTORISATION

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to **microfilm** this thesis and to **lend** or **sell copies of the film.**

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de **microfilmer** cette thèse et de **prêter** ou de **vendre des exemplaires du film.**

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

ATTACH FORM TO THESIS – VEUILLEZ JOINDRE CE FORMULAIRE À LA THÈSE

Signature

Xiaoning Wang

Date

Sept. 1, 1985

NL-91 (r. 84/03)

Canada

THE UNIVERSITY OF ALBERTA

SOME NEW APPROACHES FOR LINEAR QUADTREES

by

© XIAONING WANG

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

August, 1985

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR XIAONING WANG

TITLE OF THESIS SOME NEW APPROACHES FOR LINEAR QUADTREES

DEGREE FOR WHICH THESIS WAS PRESENTED MASTER OF SCIENCE

YEAR THIS DEGREE GRANTED August, 1985

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(SIGNED) Xiaoning Wang

PERMANENT ADDRESS:

Site 16, Box #1

RR #5, Edmonton

Alberta T5P 4B7

DATED 16 Aug 1985

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled SOME NEW APPROACHES FOR LINEAR QUADTREES submitted by XIAONING WANG in partial fulfilment of the requirements for the degree of MASTER OF SCIENCE.

Wayne A. Davis
....
Supervisor

....

....

....

Date 16 Aug 85

IN MEMORY OF MY DEAR FATHER

XIA-BEI WANG

ABSTRACT

This thesis introduces a modified version of the linear quadtree for representing regions, and presents algorithms for the manipulation of regions using these modified linear quadtrees. For a 2^n by 2^n binary image and a region composed of N BLACK nodes represented as a modified linear quadtree, the following results will be presented: algorithms for labeling all connected components, computing the perimeter, finding the centroid, rotating by multiples of 90 degrees, and generating the background of the region can be executed in $O(n \cdot N)$ time. Computing the region's area can be done in $O(N)$ time. The intersection and union of two regions composed of N1 and N2 BLACK nodes takes time $O(N1+N2)$ and $O(n \cdot (N1+N2))$, respectively, in the worst case. Two other algorithms are described for converting a quadtree to a modified linear quadtree and vice versa in time proportional to the number of nodes in the quadtree. More importantly, these two algorithms can be used jointly in achieving an efficient condensation algorithm. It will also be shown that the proposed modified linear quadtree approach is more efficient than the linear quadtree in terms of both time and space complexities.

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to my supervisor, Dr. W.A. Davis, for introducing me to the field. I also thank him for his guidance, encouragement and assistance during the development of this research. Further thanks are due to the members of my examining committee, Drs. W.W. Armstrong, M.T. Ozsu and J.C. Muller, for their helpful comments.

I gratefully acknowledge the Department of Computing Science for the financial support in the form of teaching and research assistantships.

Finally, I am greatly indebted to my husband, De-Lei, without whose support and understanding, this thesis would never have been completed.

Table of Contents

List of Figures

CHAPTER 1

INTRODUCTION

Region representation is an important issue in image processing, cartography, computer graphics, geographic information systems, and robotics [31]. Currently there are a variety of methods used for representing regions. The success of an approach to the representation of regions depends on how easy it is to implement in terms of computational complexity, memory space requirements, and extensibility [19-22].

A method known as the chain code region representation specifies the boundary of a region by storing a sequence of elementary vectors [18]. This method is very compact in terms of the storage requirement, and makes it easy to detect the boundary of the region [29]. The disadvantage is the difficulty of performing set operations such as intersection and union for regions. A method called the medial axis transformation is less compact than a chain code, but it can perform both intersection and union operations more efficiently [30].

Recently, quadtrees have recieved increasing attention [5,10,14,26,34]. The quadtree representation is based on the principle of recursive decomposition of an image proposed by Klinger [12,13]. Its hierarchical nature makes it relatively compact, and well-suited to operations such as complement,

union and intersection [9], as well as for determining various other region properties [32]. Properties of quadtrees have been extensively studied, and many efficient algorithms have been developed, further supporting the quadtree concept [10,23,24,26,27].

Efforts at reducing the space requirements of quadtrees even more have led to the concept of linear quadtrees [6]. For a linear quadtree, the use of a locational code or key provides an unique identification of the BLACK node associated with that key, and the quadtree topology can be obtained by performing simple operations on the keys. As a result, a linear quadtree represents a quadtree as a sequence of terminal nodes in a specific order, while nonterminal nodes, or even WHITE nodes of the quadtree are omitted, achieving a significant space efficiency.

In this direction, the linear quadtree proposed by Mark and Lauzon [15] stores both BLACK and WHITE nodes in a compressed form where only the key of the last terminal node in a sequence of consecutive terminal nodes of the same color is stored.

The linear quadtree proposed by Gargantini stores only the BLACK nodes comprising the region, and inferring the WHITE nodes as required. This linear quadtree will be referred to as a quaternary-code linear quadtree (QLQ), since each BLACK node is associated with an n digit quaternary code whose digits reflect the successive decomposition of a 2^n by 2^n image. As reported in [6], a QLQ

introduces a saving of at least 66 percent of the memory space required by its standard quadtree counterpart. Algorithms for manipulating the QLQ have been proposed in [2,6-8]. Compared with the quadtree approach in terms of computational complexity, the QLQ approach is less efficient in supporting fast algorithms for the manipulation of regions. It is, therefore, desirable to use the linear quadtree approach to save memory space on one hand, and to have a more efficient representation to facilitate operations on the other hand.

To achieve such a goal, this thesis begins with an introduction to the problem of the representation of regions in Chapter 2. The recursive decomposition of images is briefly reviewed along with its quadtree and QLQ representations in preparation for the development and discussion of an alternative representation. A modified linear quadtree is then introduced in Chapter 3, and is shown to be more efficient spacewise than the QLQ. Chapter 4 investigates operations on images using the proposed structure. A number of algorithms are presented in detail to effectively support these operations. The proposed algorithms are shown to be more efficient than their corresponding QLQ counterpart in terms of time complexities. The last chapter of the thesis presents the conclusions and suggestions for further research.

CHAPTER 2

BACKGROUND

This chapter contains some basic definitions and terminology for region representations that are fundamental for the remainder of this thesis.

2.1 Definitions and notation

Definition 2.1: An **image** is a 2^n by 2^n array of unit square pixels each of which can assume one of 2^k values, where n is called the **resolution parameter** of the image.

Definition 2.2: An image is called a **binary image** when its pixels assume either 1 or 0 values. A pixel is BLACK if it has the value of 1, otherwise it is WHITE.

Without loss of generality, only binary images will be considered in this thesis, since all of the algorithms can be extended to nonbinary images.

Definition 2.3: The **region** of a binary image is composed of all BLACK pixels, and the **background** of the region is composed of all WHITE pixels.

Example: The region shown in Fig. 2.1 is represented by the 2^3 by 2^3 binary array in Fig. 2.2, where 1 and 0 correspond to BLACK and WHITE pixels, respectively.

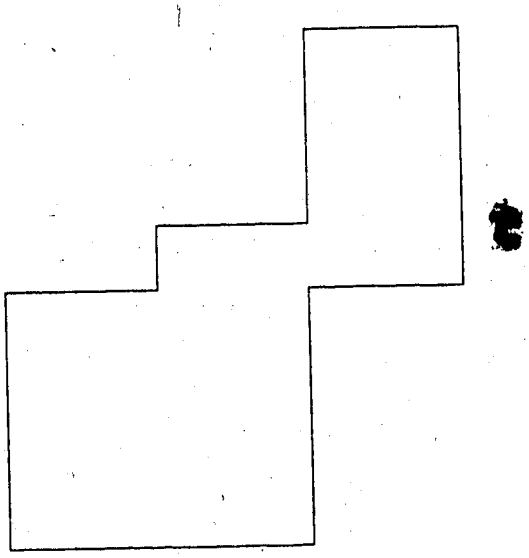


Fig. 2.1. A region.

7	0	0	0	0	1	1	0	0
6	0	0	0	0	1	1	0	0
5	0	0	0	0	1	1	0	0
4	0	0	1	1	1	1	0	0
3	1	1	1	1	0	0	0	0
2	1	1	1	1	0	0	0	0
1	1	1	1	1	0	0	0	0
0	1	1	1	1	0	0	0	0
	0	1	2	3	4	5	6	7

Fig. 2.2. The region in binary array representation.

Definition 2.4: Let (i,j) represent the location of a pixel p in a given image, where i and j are the column and row positions respectively. Then p has four horizontal and vertical neighbors located at: (i-1,j), (i,j-1), (i,j+1) and (i+1,j). These pixels are called the **4-neighbors** of p, and are said to be **4-adjacent** to p.

Definition 2.5: A **path** from pixel p to pixel q is a sequence of distinct pixels, $p = p_0, p_1, \ldots, p_n = q$, such that p_m is 4-adjacent to p_{m-1}, where $1 \leq m \leq n$.

Definition 2.6: For two BLACK pixels, p and q, of a region, p is said to be **connected** to q if there is a path from p to q consisting entirely of pixels of the region.

Definition 2.7: For any BLACK pixel p, the set of pixels connected to p is called a **connected component** of the region. If a region has only one component, then it is called "connected".

There has been considerable interest in region representations based on the principle of recursive decomposition, which is similar to the divide and conquer method [1]. An image is decomposed in the following manner to separate a region from its background, and to represent a set of pixels belonging to the same quadrant as a single block. If the region does not cover the entire binary array, the array will be subdivided into four equal-sized square blocks. This process will be applied recursively, until

blocks are obtained that are either totally contained in the region or totally disjoint from it. As an example, Fig. 2.3 is the decomposition of the region shown in Fig. 2.1. It can be seen that the region is decomposed into the maximal blocks A, B, C, D and E, thereby separated from the background.

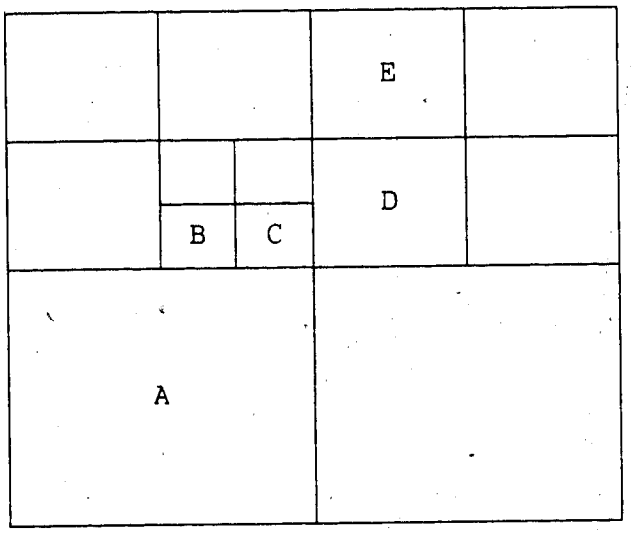


Fig. 2.3. Decomposition of the region in Fig. 2.1.

The recursive decomposition of an image produces blocks that must have standard sizes (powers of 2) and positions. Similar definitions can now be formulated in terms of blocks.

Definition 2.8: A block is said to be BLACK if it contains only BLACK pixels, WHITE if it contains only WHITE pixels, and GREY if it contains both BLACK and WHITE pixels.

The four sides of a block are referred as to its North, East, South and West sides, or N, E, S and W for short. And let **OPSIDE**(T) be the side opposite to T, e.g., **OPSIDE**(E)=W.

Definition 2.9: Two blocks P and Q are said to be 4-adjacent along the side T of P if the side T of P touches the side **OPSIDE**(T) of Q.

Definition 2.10: BLACK blocks P and Q are said to be connected if there exists a path consisting entirely of BLACK pixels from a pixel of P to a pixel of Q.

Clearly, representing a region as a union of maximal blocks is much more compact than as a union of individual pixels.

2.2 The quadtree approach

Hierarchical data structures are a natural way to represent the above recursive decomposition process. One such data structure, now known as quadtree, was proposed originally by Klinger [11,12], see also [33,35,36]. A quadtree is an ordered tree of out-degree four. The root represents the entire image, and each other node represents one of the four subblocks in order NW, NE, SW, SE of its father's block. No father node can have all its descendant terminal nodes with the same color. As an example, Fig. 2.4 demonstrates the quadtree for the region in Fig. 2.1, where the symbols O, □ and ■ represent GREY, WHITE and BLACK

nodes, respectively. Note that the terms block and node will be used interchangeably throughout.

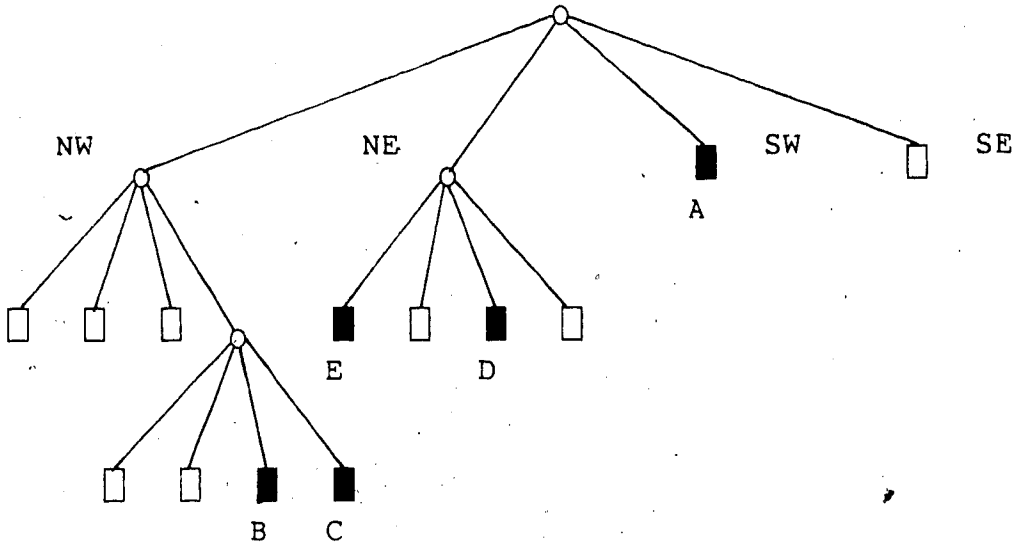


Fig. 2.4. The quadtree for the region in Fig. 2.1.

Each node in the quadtree holds six pieces of information. The first five are pointers to its father and its four sons. The sixth piece of information describes the color of the node. Terminal nodes have color BLACK or WHITE while nonterminal nodes have color GREY.

One problem with the quadtree representation of the recursive decomposition process is that it has a considerable amount of overhead associated with it, for example, in memory space. This argument is supported by the observation that only GREY nodes effectively use their four pointers to their four sons, while BLACK and WHITE nodes

have their pointers to empty records. In addition, the number of BLACK and WHITE nodes is nearly three times the number of GREY nodes as stated by the following lemma.

Lemma 2.1: The total number of GREY nodes in a quadtree having N BLACK nodes and W WHITE nodes is bounded by (N+W-1)/3.

Proof: Let G denote the number of GREY nodes. Given G GREY nodes and (N+W) terminal nodes, there are (G+N+W-1) edges. By the number of sons, there are 4G edges. Thus 4G=G+N+W-1, or G=(N+W-1)/3.

Q.E.D.

A simple solution may introduce two different kinds of records for terminal nodes and nonterminal nodes. This will greatly reduce the number of pointers per node from five to two on the average. The drawback of this solution is, however, that the resulting quadtree is no longer composed of the same record types and therefore operations must be performed differently when nodes differ.

Totally eliminating quadtree pointers together with removing all GREY and WHITE nodes may solve the problem. The latter is more important in two respects. Firstly, the resulting quadtree, which Gargentini calls a linear quadtree, consists only of BLACK nodes, and therefore operations performed at each node will be exactly the same. Secondly, the number of BLACK nodes comprising the region can be expected to be small relative to the total number of

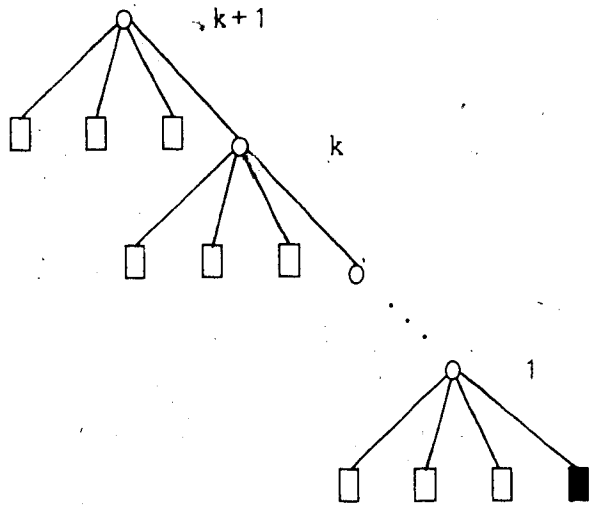
nodes in the quadtree. This is verified by the following two lemmas.

Lemma 2.2: The upper bound on the number of nodes in a quadtree having N BLACK nodes is 4n·N+1.

Proof: By induction on the resolution parameter n.

The basis, n=1, is trivial by observing that a quadtree with one BLACK node has 5 nodes, and any quadtree with N>1 BLACK nodes has less than 4N+1 nodes.

Now, assume the lemma is true for n=k≥1. Consider quadtree Q of k+1 levels having one BLACK node as shown below.



It is easy to verify that the number of nodes, T, in Q is 4(k+1)+1. It remains to show that any quadtree R of k+1 levels with N>1 BLACK nodes has less than 4(k+1)·N+1 nodes.

Without loss of generality, consider the following two primitive ways to construct R from Q.

Case 1. **R** is obtained by changing a WHITE node of **Q** into a BLACK one. Then the total number of nodes and the number of BLACK nodes in **R** are T and 2, respectively. Clearly,

$$T < 4(k+1)\cdot 2+1.$$

Case 2. **R** is obtained by replacing a WHITE node of **Q** by a quadtree **S** of k levels having m BLACK nodes. Invoking the inductive hypothesis and letting T_s be the number of nodes in **S**, then

$$T_s \leq 4k\cdot m+1.$$

Therefore,

$$T+T_s \leq 4(k+1)+4k\cdot m+2,$$

$$T+T_s-1 \leq 4(k+1)\cdot(m+1)+1-4m.$$

Thus,

$$T+T_s-1 < 4(k+1)\cdot(m+1)+1.$$

Note that $T+T_s-1$ and $m+1$ are the total number of nodes and the number of BLACK nodes in **R**, respectively.

The lemma is, therefore, proven.

Q.E.D.

Lemma 2.3: The lower bound on the number of nodes in a quadtree having N BLACK nodes is N+n+1.

Proof: Similar to the proof of Lemma 2.2.

Q.E.D.

It should be clear by now that the concept of linear quadtree provides the potential of a very efficient storage utilization.

2.3 The linear quadtree approach

The distinct feature of linear quadtrees is that they are pointerless and store only BLACK nodes. To capture the recursive decomposition process, a linear quadtree encodes its BLACK nodes in the following way: the NW quadrant is encoded with 0, the NE with 1, the SW with 2, and the SE with 3. Each BLACK pixel is then encoded as a sequence of n integers (i.e., a quaternary code):

$q_{n-1}, q_{n-2}, \cdots, q_0$, where $q_i \in \{0,1,2,3\}$ for $0 \le i < n$.

Each successive integer therefore represents the block decomposition from which it originates. Thus, q_{n-i}, for $1 \le i \le n$, identifies the subblock to which the pixel belongs at the ith decomposition. A BLACK node corresponding to a 2^k by 2^k block is encoded as $q_{n-1}, q_{n-2}, \cdots, q_0$, with $q_i = X$, $X \neq 0,1,2,3$, for $0 \le i < k$, where X is called a don't care sign. A linear quadtree is the sorted sequence of quaternary codes corresponding to the BLACK nodes comprising the region.

As an example, the linear quadtree representation for the region in Fig. 2.1 is the following sequence:

032, 033, 10X, 12X, 2XX

which correspond to the BLACK nodes B, C, E, D, and A, respectively, of Fig. 2.3. A number of algorithms for manipulating linear quadtrees have been reported in [2,6-8]. These algorithms suffer, in speed, from the quaternary encoding of the BLACK nodes.

In the next chapter, a modified linear quadtree (MLQ) is presented as an alternative to the linear quadtree.

Various algorithms will be developed and analyzed for the manipulation of regions using an MLQ.

CHAPTER 3

A MODIFIED LINEAR QUADTREE [3]

The purpose of this chapter is two-fold. The first is to introduce the MLQ, and the second is to establish a foundation upon which algorithms will be developed and analyzed in subsequent chapters. The computational model which will be used in the analysis of time requirements of the algorithms involves a RAM, or a random access machine [1], for which storage or retrieval of data from memory requires constant time. Time is measured in steps, or operations such as comparison and multiplication requiring constant time.

3.1 Definitions and notation

The following conventions will be adopted throughout this thesis.

Definition 3.1: For two integers I and J given by

$$I=\sum_{i=0}^{n-1}(I_i\cdot 2^i),\ \text{and}\ J=\sum_{i=0}^{n-1}(J_i\cdot 2^i),\ \text{where}\ I_i,\ J_i\in\{0,1\},$$

$$\textbf{SHUFFLE}(I,J)=\sum_{i=0}^{n-1}(I_i\cdot 2+J_i)\cdot 4^i.$$

Definition 3.2: For an integer $S = s_{2n-1} \cdots s_0$, where $s_i \in \{0,1\}$,

$$\textbf{EVEN}(S) = \sum_{i=0}^{n-1} s_{2i} \cdot 2^i \text{ and } \textbf{ODD}(S) = \sum_{i=0}^{n-1} s_{2i+1} \cdot 2^i. \quad -$$

The above two definitions have effectively specified how these three functions can be implemented in $O(n)$ steps. Hereafter, **SH**, **EV** and **OD** will be used as abbreviated versions of **SHUFFLE**, **EVEN** and **ODD**, respectively. The use of **SH**, **EV** and **OD** functions will become apparent later.

Given a 2^n by 2^n image, there are a number of ways to assign consecutive integers to the pixels. A method which is now known as a Morton matrix [16] is the best to capture the nature of the recursive decomposition described in the previous chapter.

Definition 3.3: A Morton matrix is an array of 2^n by 2^n pixels each of which is assigned an integer as follows. For a pixel p with coordinates (I,J), where I and J are the column and row position, respectively, the integer assigned to p is **SH**(I,J).

Example: Fig. 3.1 shows a 2^3 by 2^3 Morton matrix.

7	21	23	29	31	53	55	61	63
6	20	22	28	30	52	54	60	62
5	17	19	25	27	49	51	57	59
4	16	18	24	26	48	50	56	58
3	5	7	13	15	37	39	45	47
2	4	6	12	14	36	38	44	46
1	1	3	9	11	33	35	41	43
0	0	2	8	10	32	34	40	42
	0	1	2	3	4	5	6	7

Fig. 3.1. A 2^3 by 2^3 Morton matrix.

To represent a block obtained by the recursive decomposition method requires the following definition:

Definition 3.4: The **key** of a block or node with 2^s by 2^s pixels is the key of its left bottom pixel, where s is called the **resolution parameter** of the block.

It is now easy to show that the two-tuple <K,s> uniquely represents a block, where K and s are the key and resolution parameter of the block, respectively.

Definition 3.5: Given a block Q, its subblock with label i is called the ith subblock of Q and denoted by Q_i if Q_i is obtained by one subdivision of Q, where $i \in \{0,1,2,3\}$.

The quadrant labeling shown below will be assumed.

1	3
0	2

Example: For the image in Fig. 3.1, the entire image **E** is represented as <0,3>, and $\mathbf{E}_1$ as <16,2>.

The following two lemmas are useful in developing future algorithms.

Lemma 3.1: For any two nodes $<K_1,s_1>$ and $<K_2,s_2>$ where $<K_1,s_1> \epsilon Q_i$ and $<K_2,s_2> \epsilon Q_j$ for some node Q, if i<j then $K_1<K_2$.

Proof: Let Q be <K,s>. Then $i \cdot 4^{s-1} \leq K_1-K < (i+1) \cdot 4^{s-1}$ for $<K_1,s_1> \epsilon Q_i$; and $j \cdot 4^{s-1} \leq K_2-K < (j+1) \cdot 4^{s-1}$ for $<K_2,s_2> \epsilon Q_j$. Therefore the lemma follows, since i<j.

Q.E.D.

Lemma 3.2: For any two nodes $<K_1,s_1>$ and $<K_2,s_2>$ where $s_1>s_2$, then either $<K_1,s_1>$ contains $<K_2,s_2>$, or the intersection of $<K_1,s_1>$ and $<K_2,s_2>$ is empty.

Proof: The condition $s_1>s_2$ imposes two cases in which $<K_2,s_2>$ can be obtained.

Case 1: $<K_2,s_2>$ is obtained by successive subdivision of $<K_1,s_1>$ implying that $<K_2,s_2>$ is contained in $<K_1,s_1>$.

Case 2: $<K_2,s_2>$ is not obtained by successive subdivision of $<K_1,s_1>$ implying that $<K_2,s_2>$ and $<K_1,s_1>$ are disjoint.

Q.E.D.

3.2 An encoding scheme

Clearly, a BLACK node can now be encoded by a two-tuple <K,s>, where K uniquely identifies the position of the node in the image, and s specifies the resolution parameter of the node. A **modified linear quadtree** (MLQ) is defined to be a sequence of BLACK nodes in two-tuple form sorted in ascending key order. This differs from the linear quadtree in that, firstly, the key of the node is stored as a single integer rather than a n digit quaternary code, and, secondly, the resolution parameter of the node is given explicitly rather than implied by the number of don't care characters in the quaternary code.

This modification results in two advantages: space efficiency and improved execution time. The second advantage will become clear in Chapter 4. To see the first advantage necessiates a comparison between the storage requirements of the two encoding methods. As reported in [6], each BLACK node needs 3n bits. By contrast, the proposed encoding scheme requires (2n+log n) bits for each BLACK node, where 2n bits are used for storing the key and (log n) bits for the resolution parameter of the node. As an example, when n=12, the MLQ can save approximately 22 percent of the

memory space required by the linear quadtree.

3.3 Constructing an MLQ

In this thesis, two algorithms will be proposed for constructing an MLQ, given two different descriptions of a 2^n by 2^n image. An optimal algorithm will be presented in Chapter 4 for constructing an MLQ, given the quadtree description of the image. Another algorithm to be introduced in this section constructs an MLQ, given the array description of the image.

The first algorithm is useful because, as will be seen in Chapter 4, obtaining an MLQ from the quadtree description of the image is much more efficient both in time and space than generating an MLQ from the array description of the same image. When the quadtree description is not available, however, it becomes necessary to use the second algorithm.

The algorithm for generating an MLQ from the array description of a 2^n by 2^n binary image is presented below as a procedure termed **ARRAY-TO-MLQ**. It takes as input three parameters E, key and s, where E is a 2^s by 2^s binary array, key and s initially correspond to zero and n, respectively. The output of the algorithm is a global variable called LIST containing the desired MLQ representation.

```
Procedure ARRAY-TO-MLQ(E,key,s)
  begin
    if s=0 then
      if E is a BLACK pixel then
        begin
          add pair <key,s> to LIST;
          return(BLACK);
        end
        else return(NONBLACK)
      else
        begin
          for i:=0 to 3 do
            color[i]:=ARRAY-TO-MLQ(E_i,key+i*4**(s-1),s-1);
          if color[i] is BLACK, 0≤i≤3, then
            begin
              replace the last 4 pairs in LIST by <key,s>;
              return(BLACK);
            end
            else return(NONBLACK);
        end;
  end;
```

The algorithm examines each pixel value of the binary image in Morton sequence order. If a pixel is BLACK, then its two-tuple representation is formed and added to LIST which is initially empty. One of its important features is that the algorithm recursively merges the four small BLACK nodes corresponding to the last four two-tuples in LIST. This yields a bigger BLACK node whenever possible by first removing the four two-tuples and then adding the two-tuple representing the newly yielded bigger BLACK node to LIST. Upon termination of the algorithm, LIST contains all maximal BLACK nodes. This method is spacewise superior to one which first builds a list containing all BLACK pixels, and then attempts to merge the pixels in the list into maximal BLACK nodes [6].

As an application of the algorithm to the binary image in Fig. 2.2, the MLQ representing the region in Fig. 2.1 is

determined to be:

<0,2>, <24,0>, <26,0>, <48,1>, <52,1>,

which corresponds to the BLACK nodes A, B, C, D, E in Fig. 2.3, respectively.

Theorem 3.1: Procedure **ARRAY-TO-MLQ** constructs an MLQ, given the array description of the image, in time proportional to the number of pixels in the image.

Proof: Let $T(4^n)$ be the number of steps required by procedure **ARRAY-TO-MLQ** to generate an MLQ. Clearly, when $n=0$, $T(1)=1$. If $n>0$, $T(4^n)$ is the total number of steps used in the four calls of **ARRAY-TO-MLQ** on an array of size 4^{n-1}, plus approximately four steps in checking color[0] to color[3]. That is,

$$T(4^n) = \begin{cases} 4T(4^{n-1})+4 & n>0 \\ 1 & n=0. \end{cases}$$

The theorem follows by solving the recurrence.

Q.E.D.

3.4 Neighbor finding

Neighbor finding for regions represented by an MLQ is a fundamental operation. It is a cornerstone for many operations such as labeling connected components, computing perimeters, and others, which will be discussed in the following chapter. Neighbor finding using an MLQ involves essentially two steps. The first step generates the

two-tuple form of the desired neighboring block from the given block. The second step then determines the color of the neighboring block by consulting the MLQ under consideration. These two steps are now described more precisely.

3.4.1 Neighbor determination

Let each node in an MLQ be stored as a record containing two fields. The first field, named KEY, contains the key of the node. The second field, named RES, contains the resolution parameter of the node. The use of **OD** and **EV** functions on the key of the node will provide the coordinates of the node. This coordinate information with the resolution parameter will be sufficient to obtain the coordinates of the neighboring block in a specified direction. Then the two-tuple form of the neighboring block can be constructed by using the **SH** function on its coordinates.

Let **P** be a given node, its neighbor of equal size in a direction specified by side equal to {N, E, S, W} can be determined by the following procedure termed **EQ-NEIGHBOR** using the 4-adjacent criterion. Note that determining a neighbor of different size or using other adjacent criterion can be done similarly.

```
Procedure EQ-NEIGHBOR(P,side)
  begin
    I:=OD(P.KEY);
    J:=EV(P.KEY);
    case of side
      N: J:=J + 2**P.RES;
      E: I:=I + 2**P.RES;
      S: J:=J - 2**P.RES;
      W: I:=I - 2**P.RES;
    end;
    neighbor.KEY:=SH(I,J);
    neighbor.RES:=P.RES;
    return(neighbor);
  end;
```

Theorem 3.2: The time complexity of computing a neighboring node in any of the principal directions is $O(n)$.

Proof: The time complexities of the functions **SH**, **EV** and **OD** are all of $O(n)$.

Q.E.D.

3.4.2 Color determination

Since only BLACK nodes are explicitly stored in the MLQ, to determine the color of a neighboring node necessitates examining the MLQ against the neighboring node. Let **P** be the neighboring node which is to be compared with a BLACK node Q in the MLQ in the course of the examination. According to Lemma 3.2, the color of **P** can be found as one of the following cases:

1. If **P.KEY** and **P.RES** are identical to Q.KEY and Q.RES, respectively, then P=Q, and therefore **P** is BLACK.

2. If the following conditions are both satisfied, then **P** is contained in Q, and therefore **P** is BLACK:

a) P.RES < Q.RES,

b) Q.KEY ≤ P.KEY < Q.KEY + 4**Q.RES.

3. If the following conditions are both satisfied, then Q is contained in P, and therefore P is GREY:

a) P.RES > Q.RES,

b) P.KEY ≤ Q.KEY < P.KEY + 4**P.RES.

4. If none of the previous cases hold after searching the MLQ then it follows that P is WHITE.

Theorem 3.3: Determining the color of a given node can be done in $O(\log N)$ steps, where N is the number of BLACK nodes in the MLQ.

Proof: Binary search can be adopted to examine the MLQ with N BLACK nodes, since the MLQ is sorted in key order. Binary search takes $O(\log N)$ steps [1], and at each step the operation outlined above takes constant time.

Q.E.D.

It should be pointed out that with very simple hardware, operations such as **SH**, **EV**, and **OD** can all be performed in constant time, and hence **EQ-NEIGHBOR** takes time $O(1)$. This could improve the time complexity of neighbor finding from $O(n)$ to $O(\log N)$.

3.5 Conclusion

A modified linear quadtree has been introduced. The encoding scheme is based on the Morton matrix in conjunction with the concept of resolution parameters of the maximal

blocks being represented. This modification has resulted in a spacewise more efficient representation over the linear quadtree by Gargantini. The time efficiency will be considered in the next chapter.

CHAPTER 4

ALGORITHMS FOR MANIPULATING AN MLQ

This chapter contains a set of algorithms for the manipulation of an MLQ. In particular, an algorithm for labeling connected components is developed in Section 4.1. As a direct application of the algorithm, a technique for computing the perimeter of a region is described in the subsequent section. Algorithms for computing the area and centroid of a region are given in Sections 4.3 and 4.4 to show their simplicity. Section 4.5 contains two algorithms for the translation and rotation of a region, and shows that translation is an extremely costly operation. Three set-theoretical operations are presented in Section 4.6, where the importance of an efficient condensation method for the MLQ representation is demonstrated. Two algorithms are introduced to convert an MLQ to a quadtree and vice versa. More interestingly, these two algorithms can be used jointly in achieving an efficient condensation algorithm. An analysis of the performance of these algorithms will support the statement that the MLQ encoding scheme increases the speed of region operations.

4.1 Labeling connected components

The recognition of all connected components of a region [17,18] is a fundamental operation in image processing and geographic systems. Samet [23] presents an algorithm for labeling all connected components of a region represented by a quadtree, and shows that its average execution time is $O(T + N \cdot \log N)$, where T and N are the total number of nodes and the number of BLACK nodes in the quadtree, repectively. This algorithm outperforms the traditional method with an execution time proportional to the number of pixels of the image [18]. Gargantini [8] also describes an entirely different algorithm using a linear quadtree; however, the algorithm has limited power as it is only applicable to regions with special configurations, e.g., the region must consist of equal-sized BLACK nodes.

In this section, an algorithm adopting a novel approach for labeling all connected components of a region using an MLQ is presented. It is capable of handling regions with arbitrary configurations. Furthermore, the algorithm is of time complexity $O(n \cdot N)$, and hence compares favorably to Samet's algorithm [23].

4.1.1 Definitions and notation

Let each BLACK node in the MLQ be stored as a record consisting of three fields. The first two fields, termed KEY and RES, contain the key and the resolution parameter of the node, respectively. The third field, termed ID, identifies

the connected component containing the node. It is set as a result of the algorithm to be presented. An array **M** is used to represent the MLQ. As defined in the previous chapter, **M** has the property that for any i,j = (1,2,···,N), if i < j then **M**[i].KEY < **M**[j].KEY. The predicate **UNEXPLORED**(P,T) is true if and only if the side T of node **P** has not been marked "explored" in the progress of the algorithm. The predicate **LABEL**(P) is true if and only if **P**.ID has been assigned a value.

4.1.2 An observation

Given a node **P** in **M**, its four adjacent or neighboring nodes can be determined in $O(n)$ steps, by Theorem 3.2. Suppose **Q** is the adjacent node to **P** in the west direction. The color of **Q** can be determined as WHITE, BLACK or GREY in $O(\log N)$ time, by Theorem 3.3. The BLACK or WHITE color of **Q** provides the information regarding whether **Q** is connected to **P** or not. Very little knowledge, however, of what is happening between **P** and **Q** is known when the color of **Q** is GREY. Simply this is because there can either be no BLACK node or as many as up to 2^s BLACK nodes in **Q** adjacent to **P**, where s=**P**.RES, i.e., **P** is a block of 2^s by 2^s pixels. This implies that up to 2^s further searches on **M** must occur in order to exhaust all possible adjacencies. In fact, this is precisely what Samet's algorithm does. Assuming a random image in the sense that a node is equally likely to appear in any position and at any level in the quadtree, the

neighbor finding operation using a quadtree is so efficient that the average number of nodes visited is a constant [26]. Correspondingly, the neighbor finding operation using a linear quadtree is less efficient in that the average number of nodes visited is $O(\log N)$. Therefore, a connected component labeling algorithm using a linear quadtree cannot do the same thing as Samet's algorithm does. Gargantini's algorithm [8] imposes a special configuration on the region to avoid performing exhaustive search. As a result, the algorithm is not able to deal with regions with an arbitrary configuration. Clearly, it is a crucial step in achieving an efficient method that when Q, the adjacent node of **P**, turns to be GREY, how to preclude further searching on **M** without losing any information regarding the adjacencies.

It is this observation that leads to a new method, to be described in the next section, for labeling all connected components of a region using an MLQ.

4.1.3 An informal description of the algorithm

The connected component labeling algorithm has three phases. An array called MAP will be used mainly by the first phase. MAP is constructed from **M** such that for any two integers, $i,j = (1,2,\cdots,N)$, if $i < j$ then **M**[MAP[i]].RES $\leq$ **M**[MAP[j]].RES. In essence, the use of MAP provides the visit of the nodes in **M** in ascending size order, while traversing **M**.

The first phase explores all possible adjacencies between any pair of BLACK nodes in **M** and generates equivalence pairs. The second phase merges all the equivalence pairs generated during phase one into equivalence classes. Finally, the third phase assigns the same identifier (i.e., the label) to those BLACK nodes that belong to the same equivalence class to reflect a connected component.

In particular, phase one traverses **M** in ascending size order. For each BLACK node **P** in **M** being visited, and T in {N,E,S,W}, if the side T of **P** has not been previously marked, then the adjacency between node **P** and the BLACK node of greater or equal size along the side T of **P** needs be explored. If such a BLACK node indeed exists in **M**, say Q, then the side **OPSIDE**(T) of Q is marked "explored" and is assigned the same label as that of **P** to indicate that both **P** and Q belong to the same component. Depending on the configuration of the region under consideration, Q may already have been assigned a label different from that of **P**, in which case, an equivalence pair consisting of the two labels is generated. This equivalence pair will be used in the later stage of the algorithm to update the labels of **P** and Q so that eventually they will be assigned the same label. If the side T of **P** has already been marked "explored", then the exploration of the adjacency to the side T of **P** is no longer needed.

The consequence of this technique is not only to save one search on **M**, but rather to save the necessity of exhausting all possible adjancencies along the side T of **P**. The reason for this is as follows. The side T of **P** can be marked "explored" only at the time when that side of **P** was found to be connected to a BLACK node that was being visited by the algorithm. The size of this BLACK node cannot be bigger than **P** for otherwise it would not be visited before **P**. As a matter of fact, there could be as many such BLACK nodes as the size of **P** in **M**. Regardless how many BLACK nodes of this nature exist, the "explored" status of the side T of **P**, while **P** is being visited, simply indicates that the exploration of the adjacencies across the side T has been previously done.

The distinct feature of this algorithm is that phase one guarantees that, at most, one exploration of an adjacency along each side of every BLACK node in **M** is sufficient to discover all possible adjacencies between any pair of BLACK nodes. To see this, remember that phase one visits the nodes in **M** in ascending size order. Consider, for example, the image in Fig. 4.1, where the resolution parameter n is 3. By the time BLACK node **A** is visited,

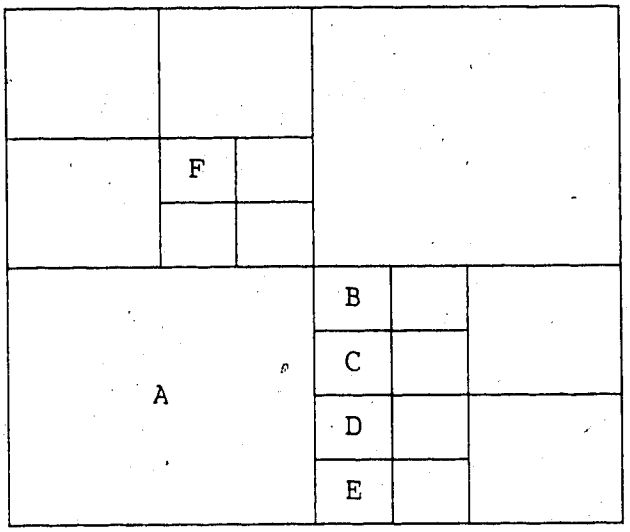


Fig. 4.1. An adjacency configuration.

its eastern adjacency needs not be reexplored, since BLACK nodes **E**, **D**, **C** and **B** have already been visited before **A**, and adjacencies between **A** and them were discovered then. Now, however, its northern adjacency must be explored, since that side of **A** cannot be marked "explored" although **F** was visited before **A**. As **A**'s northern neighbor of equal size is found to be the color of GREY, the algorithm immediately concludes that there does not exist any BLACK node adjacent to the northern side of **A** for otherwise the northern side of **A** would have been marked "explored". Therefore, no further search is necessary.

Phase two will merge the equivalence pairs generated during phase one into equivalence classes in such a way that each equivalence class contains all labels assigned to those BLACK nodes that form a connected component.

Finally, phase three updates the labels assigned to the BLACK nodes during phase one using the equivalence classes generated by phase two. Upon completion of phase three, all BLACK nodes of each connected component will have unique labels.

4.1.4 A formal statement of the algorithm

The connected component labeling algorithm will now be specified by the following procedures. Actually, only those procedures that correspond to phases one and three will be presented. Phase two can be achieved by using the well known **UNION-FIND** algorithm [1].

The main procedure is named **LABEL-CC** and invoked with an array **M** and an integer N corresponding to the number of BLACK nodes in **M**. Steps 1 and 2 construct the MAP and initialize a list called E-list which will contain the equivalence pairs as they are generated. Procedure **PROPAGATE** implements phase one. It visits the nodes in **M** in ascending size order through MAP, explores the adjacencies between pairs of BLACK nodes by invoking **EXPLORE**, assigns labels produced by **ID-GENERATOR**, and accumulates equivalence pairs in the E-list. Procedure **EQ-NEIGHBOR** used by **EXPLORE** is described in Section 3.4.1. The unspecified procedure **SEARCH(M,P)** works as follows: if **P** is a BLACK node then **SEARCH** returns an integer value k such that **P** is either equal to or contained in **M**[k]. However, if **P** is WHITE or GREY then **SEARCH** simply returns a zero. Unique labels are

generated by procedure **ID-GENERATOR**, and assigned to BLACK nodes by procedure **ASSIGN-LABEL**. Procedure **UPDATE** implements phase three by uniquely labeling each component while scanning **M**.

```
Procedure LABEL-CC(M,N)
  begin
  1  construct MAP;
  2  E-list:={φ};
  3  PROPAGATE(M,N);
  4  generate equivalence classes from E-list;
  5  UPDATE(M,N);
  end;


Procedure PROPAGATE(M,N)
  begin
    for i:=1 to N do
      begin
        j:=MAP[i];
        for side in {N,E,S,W} do
          if UNEXPLORED(M[j],side)
            then EXPLORE(M,j,side);
        if not LABEL(M[j]) then
          M[j].ID:=ID-GENERATOR;
      end;
  end;


Procedure EXPLORE(M,j,side)
  begin
    neighbor:=EQ-NEIGHBOR(M[j],side);
    k:=SEARCH(M,neighbor);
    if k>0 then
      begin
        mark OPSIDE(side) of M[k] "explored";
        ASSIGN-LABEL(M[j],M[k]);
      end;
  end;
```

```
Procedure ASSIGN-LABEL(node,adj)
  begin
    if LABEL(node) and LABEL(adj)
      then if node.ID ≠ adj.ID
             then add (node.ID,adj.ID) to E-list;
      else if LABEL(node)
             then adj.ID:=node.ID
             else if LABEL(adj)
                    then node.ID:=adj.ID
                    else node.ID:=adj.ID:=ID-GENERATOR;
  end;


Procedure UPDATE(M,N)
  begin
    for i:=1 to N do
      M[i].ID:=FIND(M[i]);
  end;
```

Example: As an example of the application of the algorithm, consider the region given in Fig. 4.2 whose block decomposition is given in Fig. 4.3. The BLACK nodes have been numbered in the order in which they were visited by phase one. Thus node 1 has been visited before nodes 2, 3, etc. The labels assigned to the two components by the first phase of the algorithm are shown in Fig. 4.4.

A short explanation about Fig. 4.4 is necessary at this point. When node 7 is visited, neither node 7 nor node 11, its eastern neighbor, has been labeled yet, thus label d is generated and assigned to both. When node 8 is visited, it has no label, but its northern neighbor, node 11, has already been assigned the label d, and thus node 8 is assigned the label d as well. Fig. 4.4 illustrates the status of the image at the conclusion of the first phase of the algorithm. It has four different labels, i.e., a, b, c

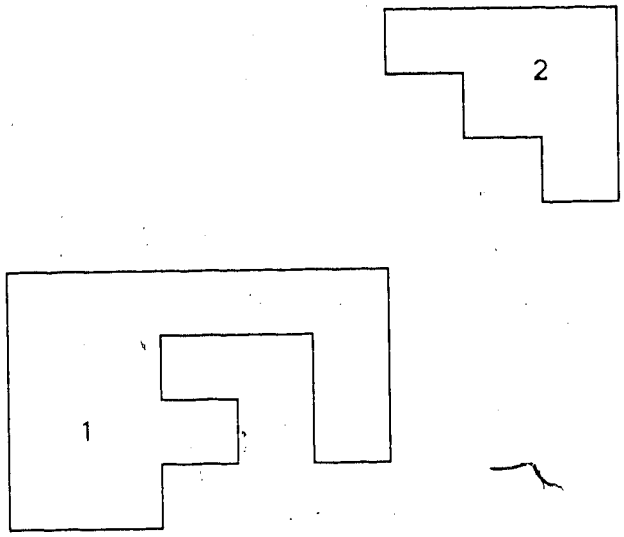


Fig. 4.2. A region with two components.

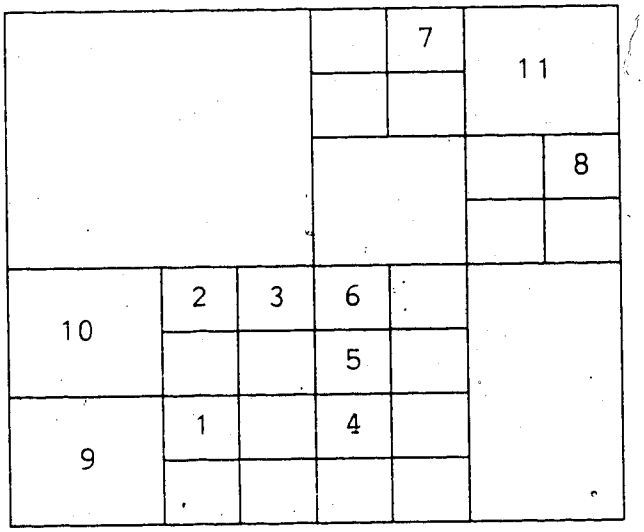


Fig. 4.3. Decomposition of the region in Fig. 4.2.

and d, with a equivalent to b, and b equivalent to c. The equivalence pair (a,b) was generated when node 9 was visited and its northern adjacency was explored. In essence, node 9 was labeled with a when node 1's western adjacency was explored, whereas node 10 was labeled with b when node 2's western adjacency was explored. Similarly, the equivalence pair (b,c) was generated when node 5 was visited.

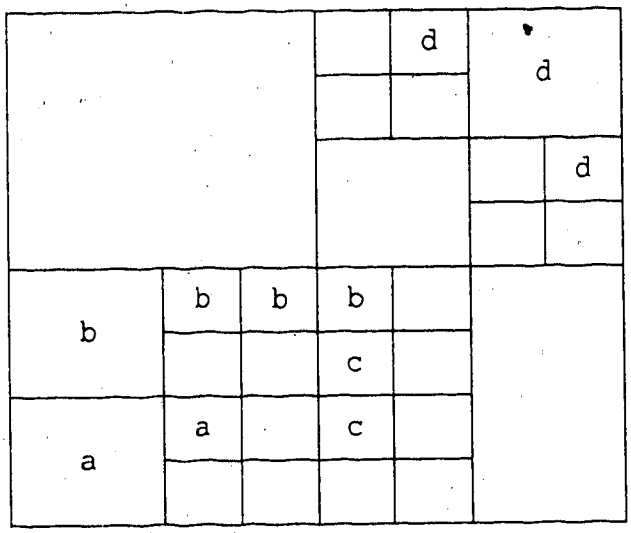


Fig. 4.4. Applying phase one to Fig. 4.3.

Applying the second phase of the algorithm to the generated equivalence pairs results in the following two equivalence classes:

{a,b,c} and {d}.

Fig. 4.5 shows the labels updated by the third phase of the algorithm.

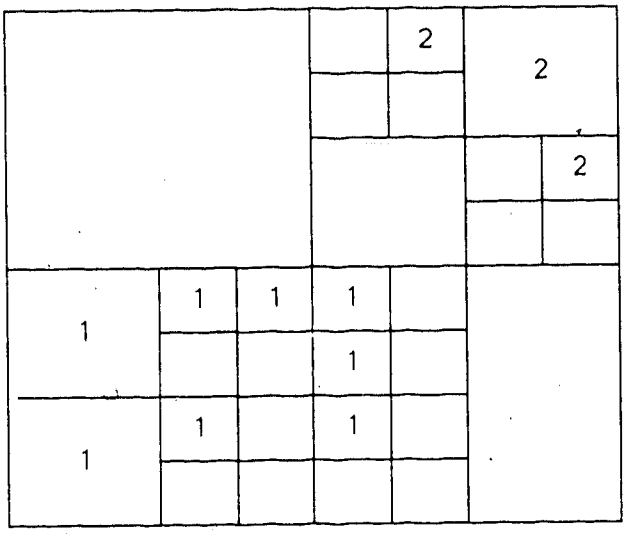


Fig. 4.5. Applying phase three to Fig. 4.4.

Theorem 4.1: The time complexity of the connected component labeling algorithm is $O(n \cdot N)$.

Proof: Constructing the MAP requires time $O(N \cdot \log N)$. Phase one calls procedure **EXPLORE** N times, and procedure **EXPLORE** requires time $O(n+\log N)$, where n and log N originates from the invoking of procedure **EQ=NEIGHBOR** and **SEARCH**, respectively. Therefore phase one takes time $O(n \cdot N + N \cdot \log N)$. Phase two requires time $O(N \cdot \log N)$ [1]. Phase three requires time $O(N)$. Since $\log N < 2n$, the time complexity of the algorithm is therefore $O(n \cdot N)$.

Q.E.D.

4.2 Computing the perimeter of a region

Perimeter computation is another basic operation in image processing. Algorithms computing the perimeter of a region in a binary image represented either by an array or by a chain code are contained in [18]. An algorithm for computing the perimeter of a region encoded as a quadtree has also been developed by Samet [24].

In what follows is a perimeter computation algorithm using an MLQ. It traverses the MLQ in ascending size order. For each node **P** in the MLQ being visited, the length of each of its four sides is first included in the value of the perimeter. Then the neighbor nodes of **P** which have not been previously visited need to be considered. For each adjacent node **Q** that is BLACK, twice the length of the common side is deducted from the value of the perimeter. This reflects the fact that the segment between **P** and **Q** does not belong to the boundary of the region. The factor 2 occurs because the adjacency between two BLACK nodes is explored once and only once due to the traversal strategy used. For example, given the BLACK node **D** in Fig. 4.7, the common segment between **D** and its southern neighbor **A** is explored by the time **D** is visited, but the same common segment is not considered when **A** is visited. Therefore the length of this segment **DA** has to be deducted in advance when **D** is visited.

The following procedure **PERIMETER** specifies the algorithm.

```
Procedure PERIMETER(M,N)
  begin
    construct MAP;
    perimeter:=0;
    for i:=1 to N
      begin
        j:=MAP[i];
        segment:=2**M[j].RES;
        perimeter:= perimeter + 4 * segment;
        for side in {N, E, S, W} do
          if UNEXPLORED(M[j],side) then
            begin
              neighbor:=EQ-NEIGHBOR(M[j],side);
              k:=SEARCH(M,neighbor);
              if k>0 then
                begin
                  perimeter:=perimeter - 2 * segment;
                  mark OPSIDE(side) of M[k] "explored";
                end;
            end;
      end;
    return(perimeter);
  end;
```

The key to this algorithm is that each node in the MLQ is visited once and, at most, its four neighbors need be explored. Such an advantage is achieved by traversing the MLQ in ascending size order. Otherwise, in the worst case, when the node being visited is of size 2^{n-1} by 2^{n-1}, 2^{n-1} nodes need be searched as in Samet's algorithm [24].

Example: As an example of the application of the algorithm, consider the region given in Fig. 4.6. The corresponding block decomposition is shown in Fig. 4.7. The MLQ contains six BLACK nodes representing blocks **A**, **B**, **C**, **D**, **F** and **G**. Assuming n=3, the perimeter is 30. Procedure **PERIMETER** visits the BLACK nodes in the order: **B**, **C**, **D**, **F**, **G** and **A**.

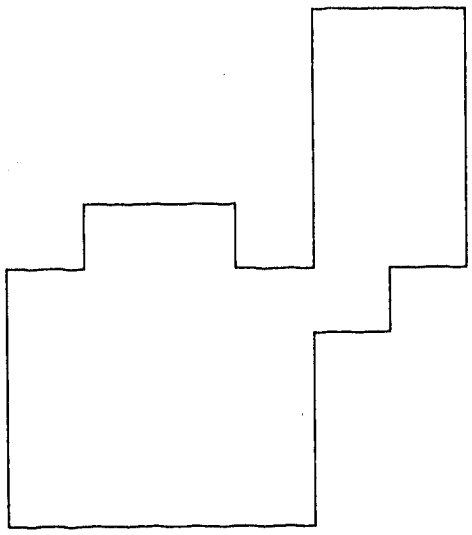


Fig. 4.6. A connected region.

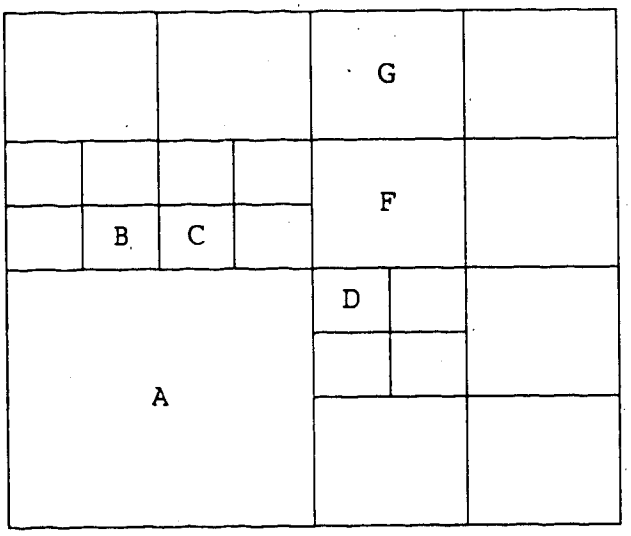


Fig. 4.7. Decomposition of the region in Fig. 4.6.

The following table contains a step-by-step trace through the perimeter computation algorithm for this example. The symbols ϕ and - stand for don't care and nonexistance, respectively.

node	side	neighbor	segment	contribut.	perim.
B				4	4
	N	-			4
	E	C	BC	-2	2
	S	A	BA	-2	0
	W	-			0
C				4	4
	N	-			4
	E	-			4
	S	A	CA	-2	2
	W	ϕ			2
D				4	6
	N	F	DF	-2	4
	E	-			4
	S	-			4
	W	A	DA	-2	2
F				8	10
	N	G	FG	-4	6
	E	-			6
	S	ϕ			6
	W	-			6
G				8	14
	N	-			14
	E	-			14
	S	ϕ			14
	W	-			14
A				16	30
	N	ϕ			30
	E	ϕ			30
	S	-			30
	W	-			30

Theorem 4.2: The time complexity of the algorithm **PERIMETER** is $O(n \cdot N)$.

Proof: Similar to the proof of Theorem 4.1.

Q.E.D.

Note that if the region is not connected, i.e., it contains more than one connected component, then the algorithm will return the sum of the perimeters of each connected component. It is, however, not difficult to compute the perimeter of every connected component of the region simultaneously in the same time complexity with a minor modification of the algorithm, provided that all connected components have been labeled.

4.3 Computing the area of a region

The following procedure **AREA** computes the area of a region represented by an MLQ, where the area is defined to be the total number of BLACK pixels comprising the region.

Input to the algorithm is **M**, an MLQ, and N, the number of BLACK nodes in the MLQ.

```
Procedure AREA(M,N)
  begin
    area:=0;
    for i:=1 to N do
      area:=area+4**M[i].RES;
    return(area);
  end;
```

Theorem 4.3: The procedure **AREA** takes time $O(N)$.

Proof: **AREA** visits each node exactly once, and the operation performed at each node takes constant time.

4.4 Computing the centroid of a region

The centroid of a region [32] is defined to be a pixel (x',y') such that

$$(x',y') = (\Sigma\, x_i/m,\ \Sigma\, y_i/m),$$

where m is the number of all BLACK pixels in the region, and (x_1,y_1), (x_2,y_2), ..., (x_m,y_m) are the coordinates of the BLACK pixels.

The number of all BLACK pixels in a binary image can easily be computed when it is represented as an MLQ. In what follows, a formula is derived to compute the sum of the X-coordinates and the sum of the Y-coordinates of all BLACK pixels of a BLACK node. With the aid of this formula, computing the centroid of a region is a simple process.

Let P be a BLACK node, and L be equal to 2**P.RES. The coordinates of the left bottom pixel of P is, therefore,

$$(x,y) = (\mathrm{OD}(\mathrm{P.KEY}),\mathrm{EV}(\mathrm{P.KEY})).$$

Consider the bottom row of P, there are L pixels. The Y-coordinates of all the L pixels are the same while their X-coordinates are:

$$x,\ x+1,\ \cdots,\ x+L-1.$$

The sum is therefore,

$$x\left(\sum_{i=1}^{L} 1\right) = x\cdot L\cdot(L+1)/2.$$

Since there are L rows in P, the sum of the X-coordinates of all the BLACK pixels of P is

$$x\cdot L^2\cdot(L+1)/2.$$

Similarly, the sum of the Y-coordinates of all the BLACK pixels of **P** is

$$y \cdot L^2 \cdot (L+1)/2.$$

The following procedure **CENTROID** calculates the centroid of a region. Input to the algorithm is **M**, an MLQ, and N, the number of the BLACK nodes in the MLQ.

```
Procedure CENTROID(M,N)
  begin
    sumx:=sumy:=0;
    m:=0;
    for i:=1 to N do
      begin
        L:=2**M[i].RES;
        S:=L*L*(L+1)/2;
        sumx:=sumx+OD(M[i].KEY)*S;
        sumy:=sumy+EV(M[i].KEY)*S;
        m:=m+4**M[i].RES;
      end;
    centx:=sumx/m;
    centy:=sumy/m;
    return(centx,centy);
  end;
```

Theorem 4.4: The procedure **CENTROID** takes time $O(n \cdot N)$.

Proof: The algorithm visits each BLACK node once and only once. And the operation performed at each BLACK nodes takes $O(n)$ steps due to the use of the **OD** and **EV** functions.

Q.E.D.

4.5 Transformation

Two algorithms, namely translation and rotation, for the transformation on regions represented as MLQs are developed in this section. The process of translating a region is complicated by the nature of to what extent each BLACK node should be decomposed in order to be translated

correctly and efficiently. A necessary and sufficient condition of the decomposition is given to guide the design of a translation algorithm in Section 4.5.1. Rotation by a multiple of 90 degree is a simple process, and an algorithm is described in Section 4.5.2.

4.5.1 Translation

The translation of a region involves translating each BLACK node of the region. It is assumed that the required translation leaves the region within the original 2^n by 2^n array [7]. Let **P** be a BLACK pixel to be translated S rows in the vertical direction and T columns in the horizontal direction. Let **Q** be the new BLACK pixel generated by the translation. Then Q can be easily computed as follows.

Q.KEY = **SH**(**OD**(P.KEY)+S, **EV**(P.KEY)+T), and

Q.RES = P.RES.

When P.RES>0, i.e., P is not a single pixel, however, the translation of **P** is not easy, since the corresponding BLACK node Q may not exist. For example, it is impossible to obtain a single BLACK node of the same size by translating the BLACK node A in Fig. 4.7 by one row in the vertical direction and one column in the horizontal direction. Therefore, the BLACK node **A** must be decomposed into 16 BLACK nodes of pixel size first, then each of the 16 BLACK nodes is translated as above individually. The decomposition of a BLACK node into pixels is, however, not always necessary depending on the size of the node and the

translation distance. For example, translating the BLACK node **A** by 4 columns in a horizontal direction can generate a single BLACK node.

In general, the following lemma gives a necessary and sufficient condition of decomposition.

Lemma 4.1: Let **P** be a BLACK node to be translated S rows and T columns, then **P** needs not be decomposed if and only if the Greatest Common Divisor (GCD) of S and T is a multiple of the size of **P**.

Proof: An elementary induction on the multiple.

This lemma is useful in two aspects. It can be used to test whether or not a BLACK node **P** can be translated as a whole. More importantly, if not, it can be used to minimize the decomposition of **P**.

Suppose the BLACK node **P** cannot be translated as a whole, and let GCD of S and T be $c \cdot 2^k$, where c is odd. Then **P** can be decomposed into 4^{s-k} BLACK nodes of size 2^k each, where s=P.RES. By Lemma 4.1, each of the smaller BLACK nodes can then be translated as a whole without being decomposed further.

Translating each of the 4^{s-k} BLACK nodes in exactly the same way requires approximately $3n \cdot 4^{s-k}$ steps, for functions **SH**, **EV** and **OD** are used. In fact, it can be done better. The following algorithm takes $n \cdot (4^{s-k}+2)$ steps to translate **P**.

```
Procedure DECOMP(P,S,T,k)
  begin
    I:=OD(P.KEY)+S;
    J:=EV(P.KEY)+T;
    if P.RES ≤ k
      then RES:=P.RES
      else RES:=k;
    for i:=I step 2**k to (I+2**P.RES-2**k) do
      for j:=J step 2**k to (J+2**P.RES-2**k) do
        add pair <SH(i,j),RES> to T-list;
  end;
```

Instead of applying functions **SH**, **EV** and **OD** to each of the 4^{s-k} decomposed BLACK nodes to be translated, the algorithm uses **EV** and **OD** only once to compute the coordinates (I,J) of the left bottom BLACK node after translation. The coordinates of the remaining $4^{s-k}-1$ BLACK nodes are calculated from I and J directly without using the functions **EV** and **OD**.

The following procedure **TRANSLATE** translates a region of N BLACK nodes by S rows in the vertical direction and T columns in the horizontal direction. The function **COMEXP**(S,T) used in the procedure **TRANSLATE** computes an integer k such that GCD of S and T is $c \cdot 2^k$, where c is odd.

```
Procedure TRANSLATE(M,N,S,T)
  begin
    k:=COMEXP(S,T);
    T-list:={φ};
    for i:=1 to N do
      DECOMP(M[i],S,T,k);
    sort the pairs in T-list;
    condense T-list;
    return(T-list);
  end;
```

Theorem 4.5: The procedure **TRANSLATE** takes time $O(n \cdot \Omega)$, where $\Omega = \sum_{i=1}^{N} Z_i$, and Z_i is 1 if **M**[i].RES $\leq$ k, and 4**(**M**[i].RES-k) otherwise.

Proof: The for loop is executed N times. During the i-th iteration, **DECOMP** is called to translate the BLACK node **M**[i]. When the resolution parameter of the BLACK node is not greater than k, **DECOMP** takes $O(n)$ steps. Otherwise, decomposition must be done. **DECOMP**, therefore, takes $O(n \cdot 4**(\text{M}[i].\text{RES}-k))$ steps. The number of pairs in T-list generated by the for loop is Ω. Sorting T-list takes $O(\Omega \cdot \log \Omega)$ steps. To condense T-list takes at most $O(n \cdot \Omega)$ steps using the algorithms to be given in Sections 4.7.1 and 4.7.2. The time complexity of procedure **TRANSLATE** is therefore $O(n \cdot \Omega)$, since $\log \Omega < 2n$.

- **Q.E.D.**

4.5.2 Rotation

The size of a BLACK node in an MLQ is invariant under rotation by a multiples of 90 degrees around the center of the image. Thus, decomposing BLACK nodes into smaller ones is not necessary. Let **P** be a BLACK node to be rotated by 90 degrees in a clockwise direction about the center of the image. Let **Q** be the new BLACK node generated by rotating **P**. Also **Q**.RES can be obtained easily since it is the same as **P**.RES. Now **Q**.KEY can be generated from **P**.KEY from the following.

By Definition 3.4, the coordinates of the left bottom pixel of **P** is (i,j)=(**OD**(**P**.KEY),**EV**(**P**.KEY)). After translating the origin to the center of the image, coordinates (i,j) become (x,y) given by

$$x = i - 2^{n-1},$$

$$y = j - 2^{n-1}.$$

Rotating a pixel (x,y) by 90 degrees in a clockwise direction will generate a new pixel (x',y'), where

$$x' = y,$$

$$y' = -x.$$

Translate the origin back, and let (I,J) be the new coordinates of the pixel (x',y'), then

$$I = x' + 2^{n-1},$$

$$J = y' + 2^{n-1}.$$

It is not difficult to see that the pixel (I,J) is, in fact, the left top pixel of **Q**. Since this pixel is 2^s-1 pixels apart from the left bottom pixel (I',J') of **Q** vertically, the key of **Q** can be easily calculated as follows:

$$I' = I$$
$$= x' + 2^{n-1}$$
$$= y + 2^{n-1}$$
$$= j, \qquad \text{and}$$

$$J' = J - 2^s + 1$$
$$= y' + 2^{n-1} - 2^s + 1$$
$$= -x + 2^{n-1} - 2^s + 1$$
$$= 2^n - 2^s - i + 1.$$

Thus, Q.KEY=**SH**(I',J').

The following procedure rotates a region of N BLACK nodes by 90 degrees in a clockwise direction around the center of the image. The input of the procedure is an array **M** and an integer N corresponding to the number of BLACK nodes in **M**.

```
Procedure ROTATE(M,N)
  begin
    R-list:={φ};
    for i:=1 to N do
      begin
        I:=EV(M[i].KEY);
        J:=2**n - 2**M[i].RES - OD(M[i].KEY) + 1;
        add pair <SH(I,J),M[i].RES> to R-list;
      end;
    sort R-list;
    return(R-list);
  end;
```

Theorem 4.6: The procedure **ROTATE** takes $O(n \cdot N)$ steps.

Proof: The for loop is of time complexity $O(n \cdot N)$ due to the invoking of functions **SH**, **EV** and **OD**. Sorting is of $O(N \cdot \log N)$. The total time is therefore $O(n \cdot N)$, since $\log N < 2n$.

Q.E.D.

4.6 Set-theoretic operations

Several algorithms for set-theoretic operations on regions represented by MLQs will be presented. In particular, algorithms for the operations of union, intersection and complement of regions are developed and analyzed in this section.

4.6.1 Complement

The complement of a region in a binary image is the background of the region. In an MLQ region representation, such an operation is useful because the WHITE nodes comprising the background are not explicitly stored.

This section is devoted to describing an $O(n \cdot N)$ method to complement a region in a 2^n by 2^n binary image represented by an MLQ. In what follows, the algorithm is capable of inferring all maximal WHITE nodes in ascending key order from two consecutive BLACK nodes. Therefore, no further sorting nor condensing is necessary.

The algorthm consists of four parts corresponding to four procedures called **GREYNODE**, **WHITE1**, **WHITE2** and **WHITE3**. Let **E** represent the entire image, i.e., **E**.KEY=0 and **E**.RES=n. Procedure **GREYNODE** takes, as input, two BLACK nodes **A**, **B** and the entire image **E** to recursively locate a GREY node Q such that $A \epsilon Q_i$, $B \epsilon Q_j$, for $i \neq j$, and returns the value of Q_i and Q_j to C and **D**, respectively, as output.

```
Procedure GREYNODE(A,B,E,C,D)
  begin
    determine i and j such that AϵE_i and BϵE_j;
    if i=j then GREYNODE(A,B,E_i,C,D)
           else begin
                  C=E_i;
                  D=E_j;
                end;
  end;
```

Procedure **WHITE1** takes, as input, a BLACK node **B** and a node Q, which is either GREY or BLACK, with $B \epsilon Q$, and recursively generates all maximal WHITE nodes within Q in

ascending key order except those whose keys are less than that of **B**.

```
Procedure WHITE1(B,Q)
  begin
    if B and Q are different
      then begin
             determine i such that B∈Q_i;
             WHITE1(B,Q_i);
             for k=i+1 to 3 do
               add Q_k to C-list;
           end;
   end;
```

Procedure **WHITE2** takes, as input, two nodes Q_i and Q_j for some Q, and generates, as output, WHITE nodes Q_k for all $i<k<j$ in ascending key order.

```
Procedure WHITE2(Q_i,Q_j)
  begin
    for k=i+1 to j-1 do
      add Q_k to C-list;
  end;
```

Procedure **WHITE3** takes, as input, a BLACK node **B** and a node Q, which is either GREY or BLACK, with $B \in Q$, and generates recursively all maximal WHITE nodes within Q in ascending key order except those whose keys are greater than that of **B**.

```
Procedure WHITE3(B,Q)
  begin
    if B and Q are different
      then begin
             determine j such that B∈Q_j;
             for k=0 to j-1 do
               add Q_k to C-list;
             WHITE3(B,Q_j);
           end;
   end;
```

The algorithm is given as procedure **WHITENODES**. The input of the algorithm is two BLACK nodes b1 and b2 with b1.KEY < b2.KEY. The output of the algorithm is in a list called C-list containing all maximal WHITE nodes generated by **WHITE1**, followed by those generated by **WHITE2**, followed by those generated by **WHITE3**. By Lemma 3.1, the obtained sequence of WHITE nodes between b1 and b2 is in ascending key order.

```
Procedure WHITENODES(b1,b2)
  begin
    GREYNODE(b1,b2,E,Q1,Q2);
    WHITE1(b1,Q1);
    WHITE2(Q1,Q2);
    WHITE3(b2,Q2);
  end.
```

Theorem 4.7: The time complexity of Algorithm **WHITENODES** is $O(n)$.

Proof: The purpose of procedure **GREYNODE** is to determine a GREY node **Q** such that b1 and b2 belong to two distinct quadrants of **Q**. Such a **Q** in conjunction with b1 and b2 will be sufficient to infer all maximal WHITE nodes between b1 and b2. The order in which **WHITE1**, **WHITE2** and **WHITE3** appear in the algorithm is the consequence of Lemma 3.1.

Let s_1 = b1.RES, s_2 = b2.RES, and $m = \max\{s_1, s_2\}$. Let n_o be the depth of recursion of **GREYNODE**, then

$$1 \leq n_o \leq n - m. \qquad (1)$$

The proof of (1) proceeds as follows: n_o is bounded from below by one is easy to see, since the **Q** obtained could be the entire image **E** itself. It remains to show that n_o is

bounded from above by n - m. Suppose $n_0 > n - m$ were true. Then $n - n_0 + 1 < m + 1$. Let $n - n_0$ be k, therefore, $2^{k+1} < 2^{m+1}$. Note that 2^{k+1} is nothing but the size of the GREY node Q obtained at the n_0-th recursion of **GREYNODE**. The inequlity $2^{k+1} < 2^{m+1}$ indicates that the size of Q cannot be larger than 2^m, the size of the bigger one of the BLACK nodes b1 and b2. Thus, b1 and b2 cannot both be contained in Q, the assumption, that $n_0 > n - m$, is false. Hence $n_0 \leq n - m$.

Let n_1 be the depth of recursion of **WHITE1**, then

$$n_1 = n - n_0 - s_1. \qquad (2)$$

The node Q1 obtained from **GREYNODE** is of size $2**(n-n_0)$, and when this Q1 is recursively subdivided by **WHITE1** into a node of size $2**s_1$, the recursion will terminate. This explains (2). Similarly, let n_3 be the depth of recursion of **WHITE3**, then

$$n_3 = n - n_0 - s_2. \qquad (3)$$

The total cost T, in terms of the depth of recursion required, is therefore $n_0 + n_1 + 1 + n_3$, where the 1 originates from **WHITE2**. Thus,

$$T = 2n - n_0 - (s_1+s_2) + 1. \qquad (4)$$

Combining (1) and (4) yields:

$$n + m - (s_1+s_2) + 1 \leq T \leq 2n - (s_1+s_2). \qquad (5)$$

The total number of recursions required is less than 2n according to (5), and each recursion takes constant time. Theorem 4.7 follows.

Q.E.D.

With algorithm **WHITENODES**, it is now possible to generate all maximal WHITE nodes comprising the background of the region in ascending key order when traversing the MLQ.

The following procedure termed **COMPLEMENT** takes, as input, **M** and N corresponding to the MLQ and number of BLACK nodes in the MLQ, respectively. The output of the algorithm is a list containing the maximal WHITE nodes in ascending key order.

In procedure **COMPLEMENT**, **WHITE3**(**M**[1],E) enumerates all WHITE nodes in E whose keys are less than **M**[1].KEY, while **WHITE1**(**M**[N],E) enumerates all WHITE nodes in E whose keys are greater than **M**[N].KEY. The for loop generates WHITE nodes between every two consecutive BLACK nodes in **M**.

```
Procedure COMPLEMENT(M,N)
  begin
    C-list:={φ};
    WHITE3(M[1],E);
    for i:=1 to N-1 do
      WHITENODES(M[i],M[i+1]);
    WHITE1(M[N],E);
    return(C-list);
  end;
```

The following result follows directly from Theorem 4.7.

Theorem 4.8: The time complexity of **COMPLEMENT** is $O(n \cdot N)$.

4.6.2 Intersection

The MLQ representation is especially efficient for performing the intersection of several MLQs. Each MLQ is essentially an ordered sequence of two-tuples. Hence, the

intersecion of two MLQs merely resembles a merge operation on two sorted lists of numbers.

Let M1 be an array of N1 BLACK nodes representing a region, and M2 an array of N2 BLACK nodes representing another region. The intersection of these two regions is obtained by the following procedure termed **INTERSECT**. For any given two BLACK nodes P and Q, the function **SWITCH**(P,Q) returns an integer value of 1 if P and Q are the same; 2 if Q contains P; 3 if P contains Q; 4 if P and Q are disjoint. The output of the procedure is a list called I-list containing the intersection of the two regions.

```
Procedure INTERSECT(M1,M2,N1,N2)
  begin
    I-list:={φ};
    i:=j:=1;
    while (i≤N1 and j≤N2) do
      case of SWITCH(M1[i],M2[j])
        1: begin
             add M1[i] to I-list;
             i:=i+1; j:=j+1;
           end
        2: begin
             add M1[i] to I-list;
             i:=i+1;
           end
        3: begin
             add M2[j] to I-list;
             j:=j+1;
           end
        4: if M1[i].KEY < M2[j].KEY
             then i:=i+1
             else j:=j+1;
    return(I-list);
  end;
```

Theorem 4.9: Procedure **INTERSECT** takes time $O(N1+N2)$ in the worst case.

Proof: The while loop can be repeated for at most N1+N2-1

times, and each iteration of the loop takes constant time.

4.6.3 Union

Union of two regions can be accomplished by a slight modification of the previously stated intersection algorithm. After a list containing the union of the two regions is obtained, however, it is necessary to make sure this list contains only maximal BLACK nodes. In other words, further condensation remains to be done.

To do this efficiently, the algorithms in the next section are used to convert the union of the two regions into a quadtree, and then convert it back to an MLQ.

The following procedure is termed **UNION** and invoked with M1, M2, N1 and N2. The output of the procedure is a list containing the union of the two regions.

```
Procedure UNION(M1,M2,N1,N2)
  begin
    U-list:={φ};
    i:=j:=1;
    while (i≤N1 and j≤N2) do
      begin
        case of SWITCH(M1[i],M2[j]
         1: begin
              add node M1[i] to U-list;
              i:=i+1; j:=j+1;
            end
         2: i:=i+1;
         3: j:=j+1;
         4: if M1[i].KEY < M2[j].KEY
              then begin
                     add M1[i] to U-list;
                     i:=i+1;
                   end
              else begin
                     add M2[j] to U-list;
                     j:=j+1;
                   end;
      end;
```

```
    if i≤N1
       then add the remainder of M1 to U-list
       else add the remainder of M2 to U-list;
    condense U-list;
    return(U-list);
end;
```

Theorem 4.10: Procedure **UNION** takes time $O(n\cdot(N1+N2))$ in the worst case.

Proof: Similar to the proof of Theorem 4.9.

4.7 Conversion between quadtrees and MLQs

Since methods of converting a quadtree region representation to other representations and vice versa have already existed [4,25,28], it is of interest to develop methods for converting from a quadtree representation to an MLQ representation and vice versa. Two algorithms that serve such purposes are presented. The algorithms will be shown to both have time complexities proportional to the number of nodes in the quadtree.

4.7.1 Converting from a quadtree to an MLQ

An algorithm is described for constructing an MLQ for a binary image given its quadtree description. The algorithm traverses the given quadtree in pretorder, i.e., visits the GREY node, and then traverses the SW subtree, the NW subtree, the SE subtree, the NE subtree. The order in which the quadtree is traversed corresponds to the ascending key order of the BLACK nodes in the MLQ to be constructed. Whenever a BLACK node is visited during the traversal, its

MLQ node representation is generated.

The following recursive procedure termed **QUAD-TO-MLQ** specifies the algorithm in detail. The procedure **QUAD-TO-MLQ** is invoked with three parameters: root - a pointer to the root of the quadtree, key - an integer variable initially set to zero, and s - an integer equal to n.

The algorithm is recursive in nature. It uses the two parameters, key and s to keep track of the position and the resolution parameter of the node being visited. Initially, the algorithm visits the root of the quadtree corresponding to the whole image, whereby <key,s> = <0,n> in terms of the MLQ representation. During the traversing of the quadtree, when a node **P** of the quadtree is visited by the algorithm, its color is examined first. If **P** is BLACK, then the pair <key,s>, which describes **P** in terms of the MLQ, is added to a list called M-list that is initially empty, and key is increased by 4^s so that it will correspond the key of the node to be visited next by the algorithm. Similarly, if **P** is WHITE, only key needs be increased, getting ready to visit the next node. When **P** is GREY, however, the algorithm calls itself four times to traverse the four sons of **P** in the order of SW, NW, SE and NE recursively. Since only a quadrant of **P** is to be visited each time, s is accordingly decreased by one. Upon termination of the algorithm, M-list contains a resulting MLQ.

```
Procedure QUAD-TO-MLQ(root,key,s)
  begin
    if root.COLOR is GREY
      then for son in {SW,NW,SE,NE} do
             QUAD-TO-MLQ(son,key,s-1)
      else begin
             if root.COLOR is BLACK
               then add <key,s> to M-list;
             key:=key + 4**s;
           end;
  end;
```

Theorem 4.11: The time required to convert from a quadtree to an MLQ is proportional to the number of nodes in the quadtree.

Proof: Each node of the quadtree is visited once and only once, and operations performed at each node take constant time.

Q.E.D.

4.7.2 Converting from an MLQ to a quadtree

To construct a quadtree from an MLQ is an inverse process of **QUAD-TO-MLQ**. The algorithm is shown as procedure **MLQ-TO-QUAD**. It is invoked with 6 parameters, where root is a pointer to the quadtree to be constructed, **M** is the given MLQ, N is the number of BLACK nodes, i is the index of **M** and equal to 1 initially, key and s are two integer variables equal to 0 and n initially. Procedure **MLQ-TO-QUAD** creates and colors the nodes of the quadtree to be constructed in preorder while scanning **M** in ascending key order.

```
Procedure MLQ-TO-QUAD(root,key,s,M,N,i)
  begin
    if (M[i].KEY≥(key+4**s)) or (i>N) then
      begin
        root.COLOR:=WHITE;
        key:=key+4**s;
      end
      else if (M[i].KEY=key) and (M[i].RES=s) then
            begin
              root.COLOR:=BLACK;
              key:=key+4**s;
              i:=i+1;
            end
            else
              begin
                root.COLOR:=GREY;
                create 4 sons SW,NW,SE,NE for root;
                for son in {SW,NW,SE,NE} do
                  MLQ-TO-QUAD(son,key,s-1,M,N,i);
              end;
  end;
```

Theorem 4.12: The time complexity of **MLQ-TO-QUAD** is $O(n+N)$ in the best case, and $O(n \cdot N)$ in the worst case.

Proof: Let the quadtree to be constructed have T nodes in total. The **MLQ-TO-QUAD** creates T nodes, and colors each of them exactly once. Therefore, $O(T)$ steps are required. Of the T nodes, there are N BLACK ones, and $N+n < T < 4n \cdot N$ by Lemmas 2.2 and 2.3. Therefore, the theorem follows.

It has been mentioned before that the combination of **MLQ-TO-QUAD** and **QUAD-TO-MLQ** can be used effectively as a condensation algorithm for an MLQ. In fact, procedure **MLQ-TO-QUAD** has to be slightly modified, for its input **M** may now contain such a sequence of BLACK nodes that the tree constructed from it satisfies the requirements of a quadtree, except there are cases of four identical BLACK sibling leaves. A quadtree may be formed by removing these

siblings and giving their color to their parent. Therefore, the following if-statement needs to be put into the place next to the for-statement of **MLQ-TO-QUAD:**

```
if the 4 sons of root are BLACK then
  begin
    remove the 4 sons of root;
    root.COLOR:=BLACK;
  end;
```

It should be pointed out that the time complexity of the modified **MLQ-TO-QUAD** remains the same as before.

4.8 Conclusion

Various techniques for the manipulation of regions using MLQs have been described in detail. The following algorithms have been presented: labeling connected components; computation of perimeter, area and controid; translation and rotation; set-theoretical operations of complement, union and intersection; conversion between an MLQ and a quadtree.

CHAPTER 5

CONCLUSIONS AND REMARKS

A new data structure called an MLQ for region representation has been presented in this thesis. Various operations on images using MLQs have been developed to support the proposed data structure. The analysis of the algorithms demonstrates that the manipulation of an MLQ is quite efficient. In particular, the algorithm for labeling connected components is superior to the one using a linear quadtree [8] in the sense that it is capable of handling regions with arbitrary configurations. By the same token, the perimeter algorithm shares the same advantage. The major credit of the $O(N)$ area algorithm is its simplicity and efficiency. Such merit cannot be achieved by either a quadtree or a linear quadtree [32]. As demonstrated by the algorithms **MLQ-TO-QUAD** and **QUAD-TO-MLQ**, the MLQ provides an intuitive and convenient way of performing conversion between a quadtree and an MLQ. The by-product of these two algorithms, however, results in an efficient condensation method which is extraordinarily important for linear quadtrees and MLQs. No condensation algorithm using linear quadtrees reported in the literature can achieve the same efficiency [6,7]. Translation of a region using a linear quadtree is an extremely costly operation, since it involves decomposing each BLACK node into pixels and then translating

each of the pixels [7]. The MLQ again introduces an intuitive approach leading to a necessary and sufficient condition for decomposition. The resulting translation algorithm using an MLQ could be more efficient. Since rotation requires neither decomposition nor condensation, algorithms using an MLQ or a linear quadtree achieve the same efficiency. Similarly, intersection of two regions can be done equally well no matter whether an MLQ or a linear quadtree is used. In some cases, the union algorithm using an MLQ outperforms the one using a linear quadtree [7] due to the concept of the MLQ supporting the fast condensation method. As reported in [6], an algorithm using a linear quadtree takes $O(n \cdot (N+W))$ steps to generate the region's background. By contrast, the algorithm using an MLQ requires $O(n \cdot N)$ steps.

In summary, the following table contains a comparison between manipulations for a linear quadtree (QLQ) and an MLQ. Np stands for the number of BLACK pixels in a binary image. For the definition of Ω, see page 50. The proposed algorithms using MLQs have time complexities ranging from $O(N)$ to $O(n \cdot N)$. This indicates that the MLQ representation supports the algorithms more efficiently or equally well as the quadtree does. The reason is that the quadtree algorithms could require $O(n \cdot N)$ steps to traverse the quadtree, regardless the operations performed at each node.

A problem of both practical and theoretically interesting is that when a quadtree or even an MLQ is too

big to be kept in main memory, the time complexity analysis adopted in the literature makes little sense, for the I/O operations to be performed can become dominant. In this regard, new complexity measures and suitable data structures may be obtained. All of these remains to be done.

	MLQ		QLQ
	Best case	Worst case	
Labeling	$O(n \cdot N)$		$O(n \cdot N)$
perimeter	$O(n \cdot N)$		-
Area	$O(N)$		-
centroid	$O(n \cdot N)$		-
Translation	$O(n \cdot \Omega)$		$O(n \cdot Np)$
Rotation	$O(n \cdot N)$		$O(n \cdot N)$
complement	$O(n \cdot N)$		$O(n(N+W))$
Intersect	$O(N1+N2)$		$O(N1+N2)$
Union	$O(N1+N2+n)$	$O(n(N1+N2))$	$O(n(N1+N2))$
MLQ-TO-QUAD	$O(n+N)$	$O(n \cdot N)$	-
QUAD-TO-MLQ	$O(T)$		

REFERENCES

1. Aho, A., Hopcroft, J. and Ullman, J.D. "The Design and Analysis of Computer Algorithms", *Reading, MA: Addison-Wesley*, 1974.

2. Bauer, M.A. "Note on Set Operations on Linear Quadtree", *Compt. Vision Graphics Image Process.*, Vol. 29, pp. 248-258, 1985.

3. Davis, W.A. and Wang, X. "A New Approach to Linear Quadtrees", *Proceedings of Graghic Interface '85*, pp. 195-202, Montreal, May 1985.

4. Dyer, C.R., Rosenfeld, A. and Samet, H. "Region Representation: Boundary Codes from Quadtrees", *Comm. ACM*, Vol. 23, pp. 171-179, March 1980.

5. Dyer, C.R. "The Space Efficiency of Quadtrees", *Comput. Graphics and Image Process.*, Vol. 19, pp. 335-348, 1982.

6. Gargantini, I. "An efficient way to represent properties of quadtrees", *Comm. ACM*, Vol. 25, pp. 905-910, Dec. 1982.

7. Gargantini, I. "Translation, rotation and superposition of linear quadtrees", *Int. J. Man-Mach. Stud.*, Vol. 18, pp. 253-263, March 1983.

8. Gargantini, I. "Detection of Connectivity for Regions Using Linear Quadtrees", *Comp. & Math. with Appl.*, Vol. 8, pp. 319-327, 1982.

9. Hunter, G.M. and Steiglitz, K. "Operations on Images Using Quad Trees", *IEEE Trans. Pattern Analy. & Mach. Intell.*, Vol. PAMI-1, pp. 145-153, 1979.

10. Hunter, G.M. and Steiglitz, K. "Linear Transformation of Pictures Represented by Quadtree", *Comput. Graphics and Image Process.*, Vol. 10, pp. 289-296, 1979.

11. Klinger, A. and Dyer, C.R. "Experiments in Picture Representation Using Regular Decomposition", *Comput. Graphics and Image Process.*, Vol. 5, pp. 68-105, 1976.

12. Klinger, A., Rhodes, M.L. and Omolayole, J. "Image Data Organization", *Proc. of San Diego Biomedical Symp.*, Vol. 5, New York: Academic press, pp. 175-180, 1976.

13. Klinger, A. and Rhodes, M.L. "Organization and Access of Image Data by Areas", *IEEE Trans. Pattern Analy. & Mach. Intell.*, Vol. PAMI-1, pp. 50-60, 1979.

14. Li, M., Grosky, W.I. and Jain, R. "Normalized Quadtrees with respect to Translations", *Comput. Graphics and Image Process.*, Vol. 20, pp. 72-81, 1982.

15. Mark, D.M. and Lauzon, J.P. "Linear Quadtrees for Geographic Information Systems", *Proc. of the International Symposium on Spatial Data Handling*, Vol. 2, pp. 412-430, Zurich, Switzerland, Aug. 1984.

16. Morton, G.M. "A Computer Oriented Geodetic Data Base, and a New Technique in File Sequencing", *IBM Canada Limited, unpublished report*, March 1, 1966.

17. Rosenfeld, A. "Connectivity in Digital Pictures", *J.ACM*, vol. 17, pp. 146-160, 1970.

18. Rosenfeld, A. and Kak, A.C. "Digital Pictures Processing", *Academic Press*, New York, 1976.

19. Rosenfeld, A. "Tree structures for Region Representation", *Comput. Graphics and Image Process.*, Vol. 11, pp. 137-150, 1980.

20. Rosenfeld, A. "Survey Picture Processing: 1981", *Comput. Graphics and Image Process.*, Vol. 19, pp. 35-75, 1982.

21. Rosenfeld, A. "Survey Picture Processing: 1982", *Comput. Graphics and Image Process.*, Vol. 22, pp. 339-387, 1983.

22. Rosenfeld, A. "Image Analysis: Progress, Problems and Prospects", *Proceedings of the 6th International*

Conference on Pattern Recognition, Vol. 1, pp. 7-15, Munich, Germany, Oct. 1982.

23. Samet, H. "Connected Component Labeling Using Quadtrees", *J.ACM*, Vol. 28, pp. 487-501, 1981.

24. Samet, H. "Computing Perimeters of Regions in Images Represented by Quadtrees", *IEEE Trans. Pattern Analy. & Mach. Intell.*, Vol. PAMI-3, pp. 683-687, 1981.

25. Samet, H. "An Algorithm for Converting Rasters to Quadtrees", *IEEE Trans. Pattern Analy. & Mach. Intell.*, Vol. PAMI-3, pp. 93-95, 1981.

26. Samet, H. "Neighbor Finding Techniques for Images Represented by Quadtrees", *Comput. Graphics and Image Process.*, Vol. 18, pp. 37-57, 1982.

27. Samet, H. "A Distance Transform for Images Represented by Quadtrees", *IEEE Trans. on Pattern Analy. & Mach. Intell.*, Vol. PAMI-4, pp. 298-303, 1982.

28. Samet, H. "Region Representation: Quadtrees from Binary Arrays", *Comput. Graphics and Image Process.*, Vol. 13, pp. 88-93, 1980.

29. Samet, H. "Region Representation: Quadtrees from Boundary Codes", *Comm. ACM*, Vol. 23, pp. 163-170, 1980.

30. Samet, H. "Quadtrees and Medial Axis Transforms," *Proceedings of the 6th International Conference on Pattern Recognition*, Vol. 1, pp. 184-187, Munich, Germany, Oct. 1982.

31. Samet, H. "The Quadtree and Related Hierarchical Data Structures", *Computing Survey*, Vol. 16, pp. 187-260, 1985.

32. Shneier, M. "Note: Calculations of Geometric Properties Using Quadtrees", *Comput. Graphics and Image Process.*, Vol. 16, pp. 296-302, 1981.

33. Shneier, M. "Two Hierarchical Linear Feature

Representations: Edge Pyramids and Edge Quadtrees", *Comput. Graphics and Image Process.*, Vol. 17, pp. 211-224, 1981.

34. Shneier, M. "Path Length Distances for Quadtrees", *Inform. Sci.*, Vol. 23, pp. 45-67, 1981.

35. Tanimoto, S.L. "Hierarchical Picture Indexing and Description", *Proc. of the Workshop on Picture Data Description and Management*, pp. 103-105, Aug. 1980.

36. Tanimoto, S.L. "A Comparison of Some Image Searching Methods", *Proc. 1978 IEEE Computer Soc. Conf. on Pattern Recognition and Image Processing*, pp. 280-286, June 1978.