Sādhanā, Vol. 18, Part 2, June 1993, pp. 279–300. © Printed in India.

Cellular logic array processing techniques for high-throughput image processing systems

E G RAJAN

Department of Electrical Engineering, Indian Institute of Technology, Kanpur 208 016, India

Abstract. This paper describes certain image processing techniques within the framework of *cellular automata* and *normal algorithms* for high-throughput data processing. The central idea on which these techniques have been developed is that a digital image can be treated as a *cellular automaton configuration*, and an image processing operation, as an *evolution* of the automaton due to an updating rule that describes a relational attribute among the pixel values in a specific neighbourhood. Filtering operations on digital images, like that of thinning, edge detection segmentation, erosion and dilation are modelled and realized using cellular automata.

Keywords. Pattern recognition; image processing; cellular automata; symbolic processing; computer vision; pattern-directed array processing; computer graphics.

1. Introduction

The literature on computer vision now has much work on advanced techniques for image processing and pattern recognition. Most of them are either spatial domain or transform domain techniques developed around the common idea that a digital image is represented as an array of numbers, and an operation on the image is carried out by applying a numerical algorithm to the array. In other words, an image-processing operation is a transformation $g(x, y) = T[f(x, y)]$, where $f(x, y)$ is the input image, $g(x, y)$ is the output image and T is the transformation operator on f, defined over some neighbourhood of (x, y). Regardless of the nature of operations, however, the general approach is to scan a given 2-D array by a window, and update all the pixel values with the computed ones, the computations involving the coefficients of the window and of the scanned image pixels.

But, there are occasions when one can carry out a required operation using a simple *pattern matching and updating* technique instead of computing update values from an expression involving the window coefficients and the corresponding image pixel values. For example, let us take the case of *edge detection* which is an operation of locating the pixels that form a transition between two regions of distinct gray level properties in an image. Numerous edge detection algorithms are available in the literature, and most of them could be seen to carry out the operation of edge

detection using arithmetic rules. Alternatively, the pattern-directed edge detection technique that is described in this paper makes use of a simple non-numerical operation of assigning the gray value 0 to a pixel if it along with its neighbours forms a particular homogeneous pattern inside a 3×3 window. By using this edge detection technique, we could achieve a considerable increase in the computational speed and the quality of the output image as compared to that observed in obtaining an edge map using various traditional techniques. A comparative study has been made, in our laboratory, of our edge detection technique with certain other conventional techniques, and the results are tabulated in § 5.2a(i).

All these show that pattern-directed array processing techniques play an important role in the image processing practice where *speed* and *quality of output* are of primary concern.

Combining the notions of Markov's algorithms and cellular automata, we have developed here, in our image processing laboratory, a formal logical framework, that we call *cellular logic array processing* (CLAP) *scheme* (Rajan 1990; Rajan & Sinha 1991), in which one can develop ways to construct fast algorithms for carrying out various pattern-directed array processing operations. With images treated as *symbolic array configurations* of cellular automata, our concern here is to consider the realization of various image processing operations in terms of pattern matching and substitution operations using what we call *generalized Markov algorithms* (GMA).

In the precise formulation of the concept of an algorithm, notions of recursive functions, Turing machines, Post's working hypothesis and Markov's normal algorithms play equivalent roles. While the first three of these occupy a well-established place in theoretical computer science and other related areas including that of signal and image processing, normal algorithms have received very little attention from the point of view of applications in these areas. The potentials for such applications would, however, seem to be enormous, considering the fact that normal algorithms play a key role in the works of Markov (1961), Shanin (1963, 1968) and others on constructive real numbers and analysis. Our choice of normal algorithms from amongst the equivalent notions just mentioned is based on these considerations.

In what follows, we review very briefly that part of the formalism which is essential for describing the concepts and the working principle of a normal algorithm, a generalized normal algorithm and a cellular automaton (Bobrow 1968; Kushner 1984; Wolfram 1986).

2. What is meant by a normal algorithm?

The notion of a *normal algorithm* was proposed by Markov (1961) with the idea of bringing out a precise definition of an *algorithm*. This notion plays a fundamental role in mathematics similar to that due to Kleene's *recursive functions* and *Turing machines*. We shall understand this notion of a normal algorithm in the following manner.

By *alphabet* we mean a finite, unordered list of primitive symbols known as letters. For example, $\mathscr{A} = \{\mathcal{C} \mid -\}$ denotes an alphabet $\mathscr{A}$ which contains the three letters $\mathcal{C}$, $\mid$ and $-$. Binary operations between alphabets are interpreted in exactly the same way as the set-theoretic operations such as *union*, *intersection* and *difference*. We shall use the same symbols $\cup$, $\cap$ and $\setminus$ to denote respectively the union, intersection and

difference operations between letters of the alphabet. Similarly all the relation symbols such as $\subseteq$, $\subset$, $\supseteq$, $\supset$ and other symbols like $\in$ and $\notin$ are also used which convey the same meanings they do in set theory.

By a *generic variable*, we mean a variable whose values are the letters taken from an alphabet. We shall use the symbols ξ, η and μ and their subscripted versions to denote generic variables, irrespective of the alphabet over which they range. But, whenever a generic variable is used, we shall define its alphabet range.

A string of letters drawn from an alphabet $\mathscr{A}$ and written one after another is called a *word* from the given alphabet. The word consisting of no letters is called *empty word* or the *null word*, which is denoted by the symbol Λ. The concatenation of two words from an alphabet is also a word from the same alphabet.

DEFINITION 2.1

A word U is called the *left factor* of a word P if the condition UV = P holds for P, where V is another word. In this case, V is called the *right factor* of P. In general, a word V is called a factor of another word P if the condition UVW = P holds for P where U and W are two other words (Rajan 1990).

In a word of the form UVW, the factors U and W are called the delimiters of the factor V.

Now let us consider an alphabet $\mathscr{A}$ and two other symbols $\rightarrow$ and $\cdot$ that are not in $\mathscr{A}$. Then words of the types: (i) $P \rightarrow Q$ and (ii) $P \rightarrow \cdot Q$ from the alphabet $\mathscr{A} \cup \{\cdot \rightarrow\}$ are called *substitution formulas*; the former is called a simple substitution formula and the latter a terminal substitution formula. P and Q are words from the alphabet $\mathscr{A}$, and are known as the left and the right parts of the corresponding formula.

Substitution formulas of the types (i) $\rightarrow Q$ (ii) $P \rightarrow$ and (iii) $\rightarrow \cdot$ whose blank parts are empty words, are admissible formulas, and those of the first type, we call *injection formulas*.

By application of a substitution formula to a word, we mean the replacement of the left part of the formula that occurs in the given word, by the right part of the formula.

An ordered list of simple and terminal substitution formulas is called a *scheme*. Let us denote such a scheme by the symbol $\mathfrak{S}$. Now, the ordered pair $\langle \mathscr{A}, \mathfrak{S} \rangle$ consisting of a specific alphabet $\mathscr{A}$ and a scheme $\mathfrak{S}$ is known as a *normal algorithm*, denoted in general by the symbol $\mathscr{N}$. By the operation of $\mathscr{N}$ on a word, say P, we mean the application of the formulas of its scheme to the word P in accordance with the following rules:

(i) If none of the left parts of the substitution formulas of $\mathscr{N}$ is a factor of P, then $\mathscr{N}$ is not applicable to P; symbolically we say $\neg !\mathscr{N}(P)$.

(ii) If at least one of the left parts of the substitution formulas of $\mathscr{N}$ is a factor of P, then $\mathscr{N}$ is definitely applicable to P (i.e., P is an input to $\mathscr{N}$) and we say $!\mathscr{N}(P)$.

(iii) If $!\mathscr{N}(P)$, then from the ordered list of $\mathscr{N}$, the first substitution formula that is applicable to P is identified and the right part of the identified formula is substituted for the first occurrence in P of this formula's left part. Let the result of this substitution be the word Q. If the applied formula is of the terminal type, the word resulting from the substitution is treated as the desired word transformed by $\mathscr{N}$ and we write $\mathscr{N}: P \vdash \cdot Q$. If the applied formula is of the simple type, then the word produced by the substitution is again subjected to $\mathscr{N}$ as outlined in the earlier steps. In this case, we express the one-step

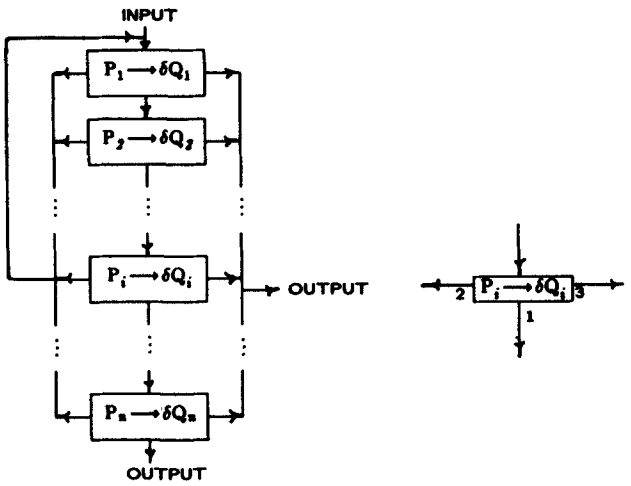

Figure 1. Functional block diagram of a normal algorithm.

transformation as $\mathcal{N}$: P $\vdash$ Q. The process of applying $\mathcal{N}$ can stop at some step either naturally (i.e., when no formula can be applied further) or by means of a terminal formula.

The rules are illustrated as shown in figure 1; here the symbol δ stands for the empty word Λ if the corresponding formula is simple, and for the symbol $\cdot$ if the formula is terminal. At a particular stage of computation, let us assume that the formula $P_i \rightarrow \delta Q_i$ is applicable to a current input word P. Then the first occurring factor P_i in P is replaced by Q_i and the resulting word is routed through channel 3 if δ stands for $\cdot$ or through channel 2 if δ stands for Λ.

3. The notion of a generalized Markov algorithm (GMA)

Since a normal algorithm over an alphabet $\mathcal{A}$, is a map of type $\mathcal{A}^* \rightarrow \mathcal{A}^*$, it is difficult to realize computable mappings of the type $(\mathcal{A}^*)^n \rightarrow \mathcal{A}^*$. $\mathcal{A}^*$ is the free monoid of the alphabet $\mathcal{A}$, and is defined as the set of all possible words from $\mathcal{A}$ together with the associative binary operation of concatenation. For example, it is inconvenient to process arrays of more than one-dimension consisting of symbols from an alphabet $\mathcal{A}$, using a simple normal algorithm over $\mathcal{A}$. However, efforts were made to solve this problem by actually providing an extension to the concept of Markov algorithms. The concept of a GMA is one such extension which was formulated by Caracciolo Di Fornio for the purpose of defining string processing languages (Bobrow 1968).

As described by Caracciolo (Bobrow 1968), the concept of generalized Markov algorithms is obtained by admitting certain higher order type substitution formulas, without altering the original sequencing scheme and functional rules.

There are two kinds of higher order substitution formulas, one representing algebraic expressions of the types $D(f_1 + f_2) \rightarrow D(f_1) + D(f_2)$ and $D(f_1 \cdot f_2) \rightarrow D(f_1) \cdot (f_2) + (f_1) \cdot D(f_2)$, and the other representing expressions of the type $XY \rightarrow X\phi[X, Y]$ where ϕ denotes a mapping from the set $A_X \cdot A_Y$ of all pairs of strings respectively belonging to X and Y into $\mathcal{A}^*$. Normal algorithms of the first kind are called *simple-generalized Markov algorithms* (S-GMA) and those of the second kind *conditional functional-*

generalized Markov algorithms (CF-GMA). In addition, a CF-GMA admits conditional clauses of the type *provided* $\beta(P)$, where $\beta(P)$ is a Boolean function depending on P which has to be satisfied in order to make the formula $P \rightarrow Q$ admissible.

It is to be noted here that both the concepts of S-GM and CF-GM algorithms were developed with an idea of providing a basic tool for defining the string processing languages PANON-1 and PANON-2. Though a PANON program is itself a string which is operated on by structural production rules, the rules from the argument string allow traditional numerical computations. Hence, it does not provide the capabilities of the required pattern-directed rewriting system in the true sense.

In order to overcome this difficulty, we consider here a particular type of generalization for Markov algorithms. As per this generalization, different factors located at various places in a string are together viewed as a pattern, that is, a subset of $(\mathscr{A} \cup \mathscr{E})^*$ and substituted by their values. For example, let us consider the string $P_1 P_2 \cdots P_i \cdots P_n \square Q_1 Q_2 \cdots Q_i \cdots Q_n$ from an alphabet $\mathscr{A} \cup \mathscr{E}$, where P_i, Q_i; $1 \leqslant i \leqslant n$ are words from $\mathscr{A}$ and $\square$ is a symbol from $\mathscr{E}$. Now, a formula of the type $\lfloor P_1 \rfloor \lfloor Q_1 \rfloor \rightarrow R$, where the symbols $\rfloor$ and $\lfloor$ denote respectively the right and the left parentheses, is an admissible substitution formula as per this generalization. We refer to this substitution formula as *conjoint substitution formula*, the details of which are given in Rajan & Ramprasad (1991). The result of applying this formula to the string $P_1 P_2 \cdots P_i \cdots P_n \square Q_1 Q_2 \cdots Q_i \cdots Q_n$ would be $RP_2 \cdots P_i \cdots P_n \square Q_2 \cdots Q_i \cdots Q_n$ or $P_2 \cdots P_i \cdots P_n \square RQ_2 \cdots Q_i \cdots Q_n$, and this depends on the type of rewriting chosen as to whether R has to replace P_1 or Q_1. So all computable mappings of the type $(\mathscr{A}^*)^n \rightarrow \mathscr{A}^*$ can be realized in terms of mappings of the type $(\mathscr{A} \cup \mathscr{E})^* \rightarrow \mathscr{A}^*$ using this concept of a generalized Markov algorithm, of course, with the help of simple pattern-directed *search* and *replace* procedures.

4. The notion of a cellular automaton

The concept of *cellular automata* was originally introduced by von Neumann and Ulam, under the name *cellular spaces*, as a possible idealization of biological systems, with a particular purpose of modelling biological self-reproduction (Toffoli & Margolus 1987). Later, this concept has been reinterpreted, for various purposes, under different names like *tesselation automata, homogeneous structures, cellular structures* and *iterative arrays.*

The term *cellular automaton*, in general, denotes a regular uniform lattice, usually infinite in extent, with a discrete variable at each site. The state of a cellular automaton, at a time instant, is determined by the corresponding *configuration*, that is, by the array of values (numerical) of these variables at that time instant. We are concerned here only with two-dimensional cellular automata, and we denote their configurations as *square array configurations.*

A cellular automaton evolves in discrete time steps, with the value of the variable at one site being affected by the values of the variables at sites in its neighbourhood on the pervious time step. By *neighbourhood of a site* we mean the site itself along with all or some of its immediately adjacent sites. The site values corresponding to a configuration at a particular time step, say t, are updated, all at once, based on the values in their neighbourhood, according to a definite set of updating rules. The resulting configuration at time step $t + 1$, is the evolved version of the one at time step t.

Updating rules are classified in the following manner. An updating rule which

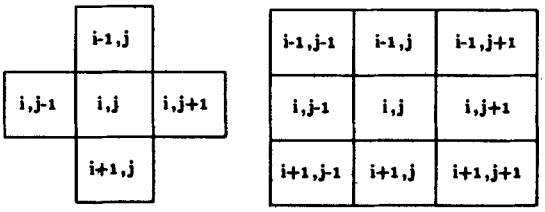

Figure 2. (**a**) 5- and (**b**) 9-neighbourhood structures.

involves the neighbourhood values of the just previous time step only, is referred as a *local rule* or a *first-order rule*, whereas the one which involves the neighbourhood values of the previous two time steps is called a *second-order rule*. All updating rules beyond second-order rules are called *higher order rules*. A cellular automaton which evolves purely according to a fixed set of local rules is called a *first-order cellular automaton*. Second and higher order cellular automata are to be understood in the same manner. This paper is concerned only with *first-order cellular automata*.

4.1 *Two-dimensional cellular automata*

As defined by Wolfram (1986), a two-dimensional N-ary valued cellular automaton $\mathfrak{C}$ consists of a 2-D array of cells. Each cell takes on any of the N given values, and all the values are simultaneously updated at a particular stage of evolution according to a rule involving the values contained in certain surrounding cells. Though one can think of several possible 2-D arrays and neighbourhood structures, we are concerned here only with square arrays with 5- and 9-neighbourhood structures as shown in figure 2. Let us consider a square array and let (i,j) be one of its cells. Then the value, $\xi_{i,j}^{(t)}$, held by the cell (i,j) at a stage t is an element from a set of numbers $X = \{0, 1, \ldots, N-1\}$. Now, the value held by the cell (i,j) could be updated in accordance with a 5-neighbourhood or a 9-neighbourhood rule. A 5-neighbourhood cellular automaton evolves according to a rule ϕ:

$$\xi_{i,j}^{(t+1)} = \phi[\xi_{i,j}^{(t)}, \xi_{i,j+1}^{(t)}, \xi_{i+1,j}^{(t)}, \xi_{i,j-1}^{(t)}, \xi_{i-1,j}^{(t)}].$$

Similarly, a 9-neighbourhood cellular automaton evolves according to a rule

$$\xi_{i,j}^{(t+1)} = \phi[\xi_{i,j}^{(t)}, \xi_{i,j+1}^{(t)}, \xi_{i+1,j+1}^{(t)}, \xi_{i+1,j}^{(t)}, \xi_{i+1,j-1}^{(t)}, \xi_{i,j-1}^{(t)}, \xi_{i-1,j-1}^{(t)}, \xi_{i-1,j}^{(t)}, \xi_{i-1,j+1}^{(t)}].$$

Table 1. Local rules of a 2D, 5-neighbourhood, binary valued cellular automaton.

Combinations	Local rules									
	ϕ_0	ϕ_1	ϕ_2	ϕ_3	ϕ_4	⋯	⋯	$\phi_{2^{32}-3}$	$\phi_{2^{32}-2}$	$\phi_{2^{32}-1}$
11111	0	0	0	0	0	⋯	⋯	1	1	1
11110	0	0	0	0	0	⋯	⋯	1	1	1
11101	0	0	0	0	0	⋯	⋯	1	1	1
11100	0	0	0	0	0	⋯	⋯	1	1	1
↓	↓	↓	↓	↓	↓	⋯	⋯	↓	↓	↓
00010	0	0	0	0	1	⋯	⋯	1	1	1
00001	0	0	1	1	0	⋯	⋯	0	1	1
00000	0	1	0	1	0	⋯	⋯	1	0	1

One can construct a total of N^{N^5} local rules for a five-neighbourhood N-ary valued two-dimensional cellular automaton, and N^{N^9} rules for a nine-neighbourhood N-ary valued two-dimensional cellular automaton.

For example, let us take the case of a binary-valued five-neighbourhood two-dimensional cellular automaton. A local rule, in this case, can be viewed as a 32-bit word as shown in the table 1. This table displays 2^{32} local rules $\phi_0, \phi_1, \phi_2, \ldots$. The column under ϕ_i is to be treated as the local rule ϕ_i.

In the same manner, an updating rule in the case of a nine-neighbourhood automaton, can be viewed as a 512-bit word, and there are 2^{512} such local rules for us to choose from.

Example 1: Let us consider a 2-D, 5-neighbourhood binary valued cellular automaton characterized by the rule $\phi_{222772222}$.

The 32-bit word corresponding to this rule is

00001101010001110011101111111110.

Let the configuration of this automaton at a particular stage of evolution t be

0 1 1 0 0
0 1 0 1 0
0 0 1 0 0
0 0 1 1 0
0 1 0 0 0.

Then the configuration of this automaton at the next stage of evolution $t + 1$ would be

0 0 0 0 0
0 1 0 1 0
0 1 0 1 0
0 1 0 1 0
0 0 0 0 0.

Example patterns generated after one hundred and twenty four evolutions of 2-D cellular automata starting from various seed values are shown in figure 3 (plate 1).

4.2 *The notion of cellular logic array processing*

With a view to reduce the space occupied by various cellular automata rules, and to provide a generalized mechanism of rule-writing, we introduce here the notion of cellular logic array processing.

As described earlier, an r-neighbourhood cellular automaton rule is nothing but a look-up table consisting of an ordered list of N^r patterns in one column with their corresponding N^r values in the other column. For a large value of N, specifying the individual entries of the corresponding look-up table is a tedious and error-prone job. Moreover, the look-up table would occupy considerable amount of space. Therefore, what we can do is to adhere to a structured approach to rule-writing, or to have a language for expressing a rule in whatever terms we find most suitable, and a mechanism for interpreting our needs and translating them into a look-up

table. In practice, we can use any extensible programming language for interpreting our needs but we cannot get them translated without a set of specialized primitives of the language for realizing them in a particular machine.

The notion of a generalized cellular logic scheme, on the other hand, does not require any specialized primitives of a high-level language. It mainly deals with the relational aspects of cell values in a specified neighbourhood, and pattern-directed symbol rewriting techniques. The term *pattern* is to be understood here as a group of cell values in a particular neighbourhood arranged in a manner described by a relational expression or a logical sentence. Any extensible programming language can be used for constructing logical sentences corresponding to various updating rules. More importantly, a logical sentence will not generate the look-up table of the corresponding rule. On the other hand, it will generate a minimal set of pattern-directed rules required for the processing of a cellular automaton configuration.

A simple example would illustrate this notion. Let us take the case of a two-dimensional binary-valued cellular automaton and a five-neighbourhood rule described by the following expression:

If $[\xi^{(t)}_{i,j}, \xi^{(t)}_{i,j+1}, \xi^{(t)}_{i+1,j}, \xi^{(t)}_{i,j-1}, \xi^{(t)}_{i-1,j}]_1$ is odd, then $\xi^{(t)}_{i,j}$ is 1; otherwise it is 0. The term $[\xi^{(t)}_{i,j}, \xi^{(t)}_{i,j+1}, \xi^{(t)}_{i+1,j}, \xi^{(t)}_{i,j-1}, \xi^{(t)}_{i-1,j}]_1$ refers to the number of 1s in a subarray under the five neighbourhood structure.

This expression refers to the following sixteen pattern-directed rules:

$$\begin{array}{llll} 00001 \rightarrow 1, & 00010 \rightarrow 1, & 00100 \rightarrow 1, & 00111 \rightarrow 1, \\ 01000 \rightarrow 1, & 01011 \rightarrow 1, & 01101 \rightarrow 1, & 01110 \rightarrow 1, \\ 10000 \rightarrow 1, & 10011 \rightarrow 1, & 10101 \rightarrow 1, & 10110 \rightarrow 1, \\ 11001 \rightarrow 1, & 11010 \rightarrow 1, & 11100 \rightarrow 1, & 11111 \rightarrow 1. \end{array}$$

The functional expression $\xi^{(t)}_{i,j} = \xi^{(t)}_{i,j} \oplus \xi^{(t)}_{i,j+1} \oplus \xi^{(t)}_{i+1,j} \oplus \xi^{(t)}_{i,j-1} \oplus \xi^{(t)}_{i-1,j}$ which is usually used to describe the updating rule discussed above will, on the other hand, generate the following look-up table with thirty two entries in it.

$$\begin{array}{llll} 00000 \rightarrow 0 & 01000 \rightarrow 1 & 10000 \rightarrow 1 & 11000 \rightarrow 0 \\ 00001 \rightarrow 1 & 01001 \rightarrow 0 & 10001 \rightarrow 0 & 11001 \rightarrow 1 \\ 00010 \rightarrow 1 & 01010 \rightarrow 0 & 10010 \rightarrow 0 & 11010 \rightarrow 1 \\ 00011 \rightarrow 0 & 01011 \rightarrow 1 & 10011 \rightarrow 1 & 11011 \rightarrow 0 \\ 00100 \rightarrow 1 & 01100 \rightarrow 0 & 10100 \rightarrow 0 & 11100 \rightarrow 1 \\ 00101 \rightarrow 0 & 01101 \rightarrow 1 & 10101 \rightarrow 1 & 11101 \rightarrow 0 \\ 00110 \rightarrow 0 & 01110 \rightarrow 1 & 10110 \rightarrow 1 & 11110 \rightarrow 0 \\ 00111 \rightarrow 1 & 01111 \rightarrow 0 & 10111 \rightarrow 0 & 11111 \rightarrow 1. \end{array}$$

It is clear from the above example, that relational pattern-directed approach to cellular automata realizations would be a convenient and more efficient means of doing array processing. We call this a *cellular logic array processing scheme.*

4.3 *The concept of growing configuration of a cellular automaton*

This concept has been introduced here with the purpose of reducing computation time involved in implementing especially 2-D and 3-D cellular automata which evolve from single-site seed configurations.

Usually, an initial configuration of a 2-D cellular automaton is treated as a square array of 0's, say of the size $N \times N$, but with a nonzero value at the centre of the array. This nonzero value $\xi^{(t)}_{N/2,N/2}$ acts as the seed to a 2-D cellular automaton, and it grows as per a 5- or 9-neighbourhood rule. Note that the first evolution is obtained only after $(N-2) \times (N-2)$ computations if we follow the traditional methods.

Here as per our method, an array of 0's, of size 5×5, with a nonzero value at the centre, is treated as the initial configuration of a 2-D cellular automaton. Thus the number of site-values to be processed turns out to be just 9. After the first evolution, the resulting array size is increased to 7×7 and it is treated as the input configuration for the second evolution. This technique of increasing the size of a configuration before causing an evolution is referred to here as *growing configuration.*

For example, let us consider the following type-1 square array configuration of size 11×11, with the seed value N at the centre.

O O O O O O O O O O O
O O O O O O O O O O O
O O O O O O O O O O O
O O O O O O O O O O O
O O O O O O O O O O O
O O O O O N O O O O O
O O O O O O O O O O O
O O O O O O O O O O O
O O O O O O O O O O O
O O O O O O O O O O O
O O O O O O O O O O O.

In this case, the first evolution is usually obtained, only after processing 81 site-values as per an updating rule. The maximum growth of the seed N that one could expect in the first evolution would be as shown below.

O O O O O O O O O O O
O O O O O O O O O O O
O O O O O O O O O O O
O O O O O O O O O O O
O O O O N N N O O O O
O O O O N N N O O O O
O O O O N N N O O O O
O O O O O O O O O O O
O O O O O O O O O O O
O O O O O O O O O O O
O O O O O O O O O O O.

As a matter of fact, this growth can be effectively obtained from the following square array configuration of size 5×5, with the seed value N at the centre, instead of from the 11×11 array considered above. The total number of site-values to be processed in this case is just 9.

O O O O O
O O O O O
O O N O O
O O O O O
O O O O O.

The configuration obtained after the first evolution would be of the form

O O O O O
O N N N O
O N N N O
O N N N O
O O O O O.

The grown configuration after the first evolution, which is used as input to the second evolution, would then be of the form

O O O O O O O
O O O O O O O
O O N N N O O
O O N N N O O
O O N N N O O
O O O O O O O
O O O O O O O.

In the same manner, the second evolution could be obtained from this grown configuration after processing 25 site-values as per the updating rule. The grown configuration with a maximum growth at this second stage of evolution would be as shown below.

O O O O O O O O O
O O O O O O O O O
O O N N N N N O O
O O N N N N N O O
O O N N N N N O O
O O N N N N N O O
O O N N N N N O O
O O O O O O O O O
O O O O O O O O O.

Thus, irrespective of the nature of the 2-D cellular automaton and the type of its updating rule, we have the initial square array configuration of size 5×5, and this configuration grows after every evolution.

5. Image processing using cellular logic principles

In our discussions so far, our attention has primarily been focussed on the role of cellular automata in the generation of patterns and pattern-like-images. We now look at their role in implementing image processing operations. In this role, the image to be processed forms the initial configuration from which a cellular automaton evolves as per a cellular logic array processing scheme and produces the desired processed image as its final configuration. In what follows, we provide high-throughput techniques for implementing *morphological operations*, and for implementing operations of *segmentation, binarization*, and *edge detection* and *extraction* on digital images.

Note that most of the traditional image processing operations involve numerical

computations, and so their implementation in a number-crunching digital computer purely in terms of pattern-directed algorithms, would lead to unwarranted complexity and consumption of more CPU time. Consequently, wherever it is necessary, we make use of these numerical operations as such in our fast operating pattern-directed algorithms, so that they form, together with a digital computer, a high throughput image-processing system. Needless to say, the operations of multiplication and division are totally avoided in these algorithms.

5.1 *Morphological operations on binary and gray level images*

The fundamental operations of mathematical morphology are the two dual operations of *dilation* and *erosion*. The operation of dilation is commutative, whereas that of erosion is not. These two operations have been defined separately for binary and gray level images.

5.1a *Dilation and erosion of binary valued images*: Let us consider a set of numbers E to denote the row and column positions of the pixels in *binary images* A and B. Let us treat the image B as the *structuring element*. Then the dilation of A by B is defined as the Minkowski addition, $A \oplus B = \{x|$ for some $a \in A$ and $b \in B$, $x = a + b\}$, and, the erosion of A by B is defined as the Minkowski subtraction, $A \ominus B = \{x|$ for every $b \in B$, $x + b \in A\}$.

5.1b *Dilation and erosion of gray level images*: For any two gray level digital images A and B, the dilation is defined as the Minkowski addition, $\mathscr{D}(A, B) = A \oplus B = \text{EXTSUP}_{(x,y)\in D_B}[A_{x,y} + B(x, y)]$, where D_B is the domain of the image B, and EXTSUP is an operation of *supremum over the union of the domains*. The operation of erosion is defined using the Minkowski subtraction, $A \ominus B = \text{INF}_{(x,y)\in D_B}[A_{x,y} + B(x, y)]$ as $\mathscr{E}(A, B) = \text{INF}_{(x,y)\in D_B}[A_{-x,-y} - B(x, y)]$, where D_B is the domain of the image B, and INF is an operation of *infimum over the intersection of the domains*. Following these definitions for both binary and gray level digital images (Giardina & Dougherty 1988), the morphological filtering operations of *closing* and *opening* are defined in the following manner.

The closing of A by B is represented as $A \circ B$ and defined as $A \circ B = (A \oplus B) \ominus B$. The opening of A by B is represented as $A \bullet B$ and defined as $A \bullet B = (A \ominus B) \oplus B$.

In the case of binary-valued images A and B, the morphological operations $A \oplus B = \{x|$ for some $a \in A$ and $b \in B$, $x = a + b\}$, and $A \ominus B = \{x|$ for every $b \in B$, $x + b \in A\}$ are simulated here, by a 2-dimensional, 9-neighbourhood binary-valued cellular automaton characterized by two updating rules, one for dilation and another for erosion.

The dilation rule has the code $0(1)^{511}$ and it consists of the following 512 conjoint substitution formulas:

Formula number	*Substitution formulas*
001	⌊000⌋⌊000⌋⌊000⌋→0
002	⌊001⌋⌊000⌋⌊000⌋→1
⋮	⋮
511	⌊010⌋⌊000⌋⌊000⌋→1
512	⌊111⌋⌊111⌋⌊111⌋→1.

The erosion rule has the code $(0)^{511}$ 1 and it consists of the following set of 512 substitution formulas:

Formula number	*Substitution formulas*
001	⌊000⌋⌊000⌋⌊000⌋→0
002	⌊001⌋⌊000⌋⌊000⌋→0
⋮	⋮⋮⋮⋮⋮
511	⌊110⌋⌊111⌋⌊111⌋→0
512	⌊111⌋⌊111⌋⌊111⌋→1.

5.1c *Fast algorithms for dilating/eroding binary images*: In view of the special form of the above two groups of substitution formulas their practical implementation is rather simple. In the case of dilation, if the left part of the first substitution formula is encountered, a 0 is substituted, and if it is not encountered a 1 is substituted. Likewise in the case of erosion, if the left part of the last substitution formula is encountered then a 1 is substituted, otherwise a 0 is substituted. Note that this algorithm does not involve even a single numerical operation.

Example 2: Figure 4a shows a random noise pattern of size 75 × 75. The image of figure 4b, of size 75 × 75, is masked by this noise pattern and the result is shown in figure 4c. Figure 4d shows the result of applying one pass of the opening operation implemented in terms of normal algorithmic cellular automata. The opening is done with a 3 × 3 square structure element consisting of 1's in all the cells.

5.1d *Fast algorithms for dilating/eroding gray level images*: Unlike what has been done in the case of binary dilation and erosion, the operations of gray level dilation and erosion cannot be simulated directly and exclusively by a 2-dimensional cellular automaton. The algorithm is described in a summarized form using the pseudo code given below. Here *pixel* is referred to as *pel*. Figure 5a (plate 2) shows an input image and a noise pattern, both of size 225 × 225. Figure 5b (plate 2) shows the image corrupted with the noise and the cleaned up image by opening it with a structuring element of size 3 × 3 consisting of the gray value 3 in all its cells.

Algorithm for erosion:
repeat sliding the structuring element over the image {
  subtract pels of structuring element from the corresponding

Figure 4. Morphological opening of a binary image.

pels of the image, find the minimum, k, among all of them
if all the structuring element pels are less than the corresponding
image pixels then replace the central pel in the image with k
else replace it with 0
} until the structuring element spans whole of the image.

Algorithm for dilation:
repeat sliding the structuring element over the image {
add pels of structuring element to the corresponding
pels of the image, find the maximum, k, among all of them,
if at least one of the image pels that are spanned by structuring
element is non-zero then replace the central pel in the image with k
else replace it with 0
} until the structuring element spans whole of the image.

5.2 *Fast algorithms for traditional image processing operations on binary and gray level images*

The algorithms for the basic image processing operations of *binarization, edge detection, segmentation,* and *feature extraction* are described here with an emphasis on that of edge detection.

5.2a *Binarization of gray level images*: The given digital image of size, say, $N \times N$ is scanned by a 3×3 window. On each move, the 3×3 sub-image covered by the window is examined to see whether the central pixel value lies in between a specified gray region including its lower and upper threshold values. If it is found to lie in that region, then the central cell is assigned a particular gray value. If not, it is assigned the value 0. This procedure is continued till the entire image is scanned. The overall effect is that the entire gray level image is transformed into an image having only two values.

5.2b *Edge detection of digital images*: Among the various types of edge detection algorithms that are available now, those which make use of *neighbourhood principles,* detect edges in a multiple gray level image by locating edge points at places where abrupt changes in gray levels occur.

We too make use of the neighbourhood principle, but with the difference that instead of locating abrupt changes in gray levels we locate boundaries of regions over which gray levels do not change. In order to understand this technique, we first take the case of detecting the edges of a digital image consisting of regions of uniform gray levels.

5.2b(i) *Algorithm for detecting edges of an image consisting of various uniform gray levels* – The given digital image of size, say, $N \times N$ is scanned by a 3×3 window. On each move, the 3×3 sub-image covered by the window is examined to see whether some or all of its pixels form a convex region of a single gray level. The smallest convex region inside a 3×3 window is that formed by pixels in a five-neighbourhood structure (see figure 2). Note that such regions can be formed in sixteen different ways as shown in figure 6. Of these, the sixteenth one is covered in the rest. To identify whether any of these regions has been encountered or not, it is therefore adequate to check for the sixteenth one alone.

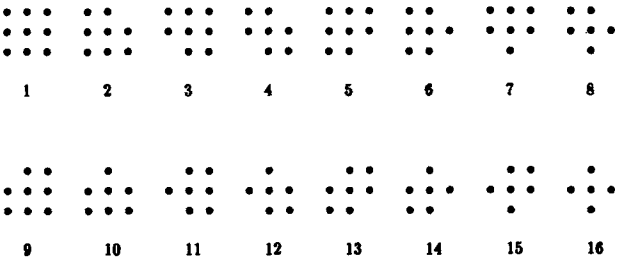

Figure 6. Sixteen convex patterns in a 3 × 3 window.

If the sub-image is found to contain the sixteenth pattern of figure 6, then its central cell is assigned the 0 gray level. This procedure is continued till the entire image is scanned. The overall effect is that the boundaries of various single gray level regions in the given image are retained and the interior parts are erased thus giving us the edge detected version of the original image (see figure 7, plate 2).

This algorithm is given below in the form of a pseudo code which can be easily translated into any computer language.

Pseudo code:

repeat sliding the 5-neighbourhood empty window over the image {
  if all the five pixel values of the sub-image covered by the
    scanning structure are identical then replace the central
      pixel value with 0, else slide the 5-neighbourhood window
        } until the structuring element spans whole of the image.

Figure 7 (plate 2) shows an image and its edge detected version.

5.2b(ii) *Algorithm for detecting edges of an image consisting of regions that appear to be uniform* – The technique of detecting the edges of various single-gray-level regions is easily extended to detecting the edges of regions that appear to be single-gray-level regions. A region that appears to have a single-gray-level may actually contain several adjacent gray levels. That they appear to be same is the result of visual quantization made by the observer. The process of partitioning a given gray level image into several regions, each of which appears to an observer to have a single-gray-level, is called *segmentation.* Hence, the central idea behind this extension is that the given image is first segmented using a threshold-based-quantization scheme and then the boundaries of the quantized regions are detected. We proceed as follows.

Similar to what is done in the previous case, the given digital image is scanned by the five-neighbourhood empty window. On each move, the 3 × 3 sub-image covered by this window is examined to see whether the gray-distance, say D, which is the difference between the maximum and the minimum gray values corresponding to that sub-image, is less than or equal to a threshold value, say T. If D is less than or equal to T, then the central cell is assigned the gray-value 0; otherwise the original value contained in the central cell is left as it is. This procedure is continued till the entire image is scanned. The overall effect is that the boundaries of various regions in the given image, that appear to be uniform, are retained, and their interior parts are erased, thus giving us the edge-detected version of the original image. The pseudo code of this algorithm is as follows:

Pseudo code:

repeat sliding the 5-neighbourhood empty window over the image { find the maximum gray-value

G_max; find the minimum gray value G_min; if the difference between these two values D is less than or equal to the specified threshold value T then replace the central pixel value with 0, else slide the 5-neighbourhood window } until the structuring element spans the whole of the image.

Figure 8b shows the edge-detected version of the image shown in figure 8a (plate 3).

The above algorithm has been tested in our image-processing laboratory, and a comparative study with various traditional edge detection algorithms has been made. The results of the study are given in table 2. The digital data corresponding to figure 8a was taken as the standard input to all the algorithms. This image is of size 256 × 256, and contains a maximum gray value of 255. All the operations were carried out in a root transputer INMOS T800 with an IBMPC/AT-386 used as a server.

The edge-enhancement operator given in table 2 is the most recent one reported in the literature (Galbiati 1990). The algorithm corresponding to this operator is given below. The given digital image is scanned by the nine-neighbourhood empty window. On each move, the maximum gray value contained in the 3 × 3 sub-image covered by this window is determined, the central cell value is subtracted from this maximum value, and the difference is assigned to the central cell. This procedure is continued till the entire image is scanned. The overall effect is that the boundaries of various regions in the given image, which appear to be uniform, are enhanced thus giving us the edge-detected version of the original image (see figure 9, plates 3 & 4).

Though this operator is somewhat similar to ours, it tampers with the original data, and causes difficulty in obtaining a contour map of a given image. In fact, contour information is completely lost in the process of edge enhancement. On the other hand, our algorithm does not alter the original pixel values, and provides precise contour information contained in a given digital image (see figure 8, plate 3).

5.2b(iii) *Algorithm for segmenting various regions of a given digital gray image* – Our approach to edge detection is easily extended to *segmentation*. The term segmentation

Table 2. Results of a comparative study.

Type of operator	Threshold	Processing time (milliseconds)	Remarks
Sobel	150	4670	See figure 9a. The original pixel values are altered
Robert	10	1395	See figure 9b. The original pixel values are altered
Edge-enhancement	—	3190	See figure 9c. The original pixel values are altered. Contour information is completely lost
Our algorithm	45	3670	See figure 8b. The original pixel values are not altered. Contour information is preserved

*Figures 8 & 9 are on plates 3 & 4.

refers to the process of partitioning a given image into regions each of which appears to an observer to have a single gray level. A region that appears to have a single gray level may actually contain several adjacent gray levels. That they appear to be same is the result of visual quantization done by the observer. Segmentation requires that we modify the pixel intensities of the given image in accordance with this quantization scheme. There are two ways of carrying out segmentation in a given image. First, the given image could be partitioned as a disjoint union of various known gray level distributions. Second, a global rule could be specified which would automatically partition the given image into various regions of statistically disjoint gray level distributions. Here, we provide an algorithm pertaining to the latter technique.

The given digital image is scanned by the five-neighbourhood empty window. On each move, the sub-image covered by this window is examined to see whether the gray-distance, say D, that is the difference between the maximum gray value $\bar{G}\_\max$ and the minimum gray value $\bar{G}\_\min$ corresponding to that sub-image, is less than or equal to a threshold value, say $\bar{T}$. If D is less than or equal to T, then the central cell is assigned the gray value $\bar{G}\_\min + (D/2)$; otherwise the original value contained in the central cell is left as it is. This procedure is continued till the entire image is scanned. The overall effect is that various regions of statistically disjoint gray level distributions in the image are partitioned thus giving us the segmented version of the original image.

5.2b(iv) *Solid object extraction in gray level images* – The algorithm for solid object extraction is somewhat similar to that of binarization. Here, the given digital image of size, say, $N \times N$ is scanned by a window of specified size. On each move, if the number of pixels, contained in the sub-image covered by the window that lie outside the specified gray range, is more than half the total pixels in the window, then the central pixel in the window is replaced with a zero, otherwise it is left untouched. In this way solid objects, bigger than the specified size and lying in the required gray range, are extracted. Figure 10 (plate 4) shows the image of an aircraft with a noisy background and a clean image extracted from it using a window of size 3×3 with the gray range between 18 and 31.

6. Conclusions

Speed and *quality of output* are the two major objectives of a high-throughput image processing system. But, more often than not, both these objectives are not simultaneously met. For example, in order to increase the processing speed we have to reduce the computational effort, that is, the number of local variables involved in the computation has to be reduced, which means loss of detail in a picture. Generally, spatial domain image processing techniques are used to obtain better results but they involve considerable time. On the other hand, transform domain image-processing techniques may not necessarily yield quality output but would certainly consume less processing time. It is in this complex situation that we found cellular logic array processing techniques to be of immense use in meeting both the objectives of *speed* and *quality of output*.

I express my sincere thanks to Prof B L Deekshatulu for his constant encouragement and support during the entire process of writing this paper. My thanks also to the anonymous referee for his constructive criticism without which this paper would not have taken this shape. I am grateful to my students and other research engineers who have been deeply involved in our group activities and have contributed their best towards the completion of this work.

References

Bobrow D G 1968 Symbol manipulation languages and techniques. *Proceedings of the IFIP Working Conference on Symbol Manipulation Languages* (Amsterdam: North-Holland)

Galbiati L J 1990 *Machine vision and digital image processing fundamentals* (Englewood Cliffs, NJ: Prentice Hall)

Giardina C R, Dougherty E R 1988 *Morphological methods in image and signal processing* (Englewood Cliffs, NJ: Prentice Hall)

Kushner B K 1984 *Lectures on constructive mathematical analysis* (Providence, Rhode Island: Am. Math. Soc.) vol. 60

Markov A A 1961 *Theory of algorithms: The Israel program for scientific translations*, Jerusalem

Rajan E G 1990 *Study of signals and systems in the framework of Markov's constructive mathematical logic*, Ph D thesis, Indian Institute of Technology, Kanpur

Rajan E G, Ramprasad V V 1991 Pattern-directed array processing, Technical Report, DSP-TR-91-5, Department of Electrical Engineering, Indian Institute of Technology, Kanpur

Rajan E G, Sinha V P 1991 Image processing using normal algorithmic cellular automata, Technical Report, DSP-TR-91-4, Department of Electrical Engineering, Indian Institute of Technology, Kanpur

Shanin N A 1963 On the constructive interpretation of mathematical judgements. *Am. Math. Soc., Transl.* 23:

Shanin N A 1968 *Constructive real numbers and constructive function spaces* (Providence, Rhode Island: Am. Math. Soc.)

Toffoli T, Margolus N 1987 *Cellular automata machines* (Cambridge, MA: MIT Press)

Wolfram S 1986 *Theory and applications of cellular automata* (Singapore: World Scientific)

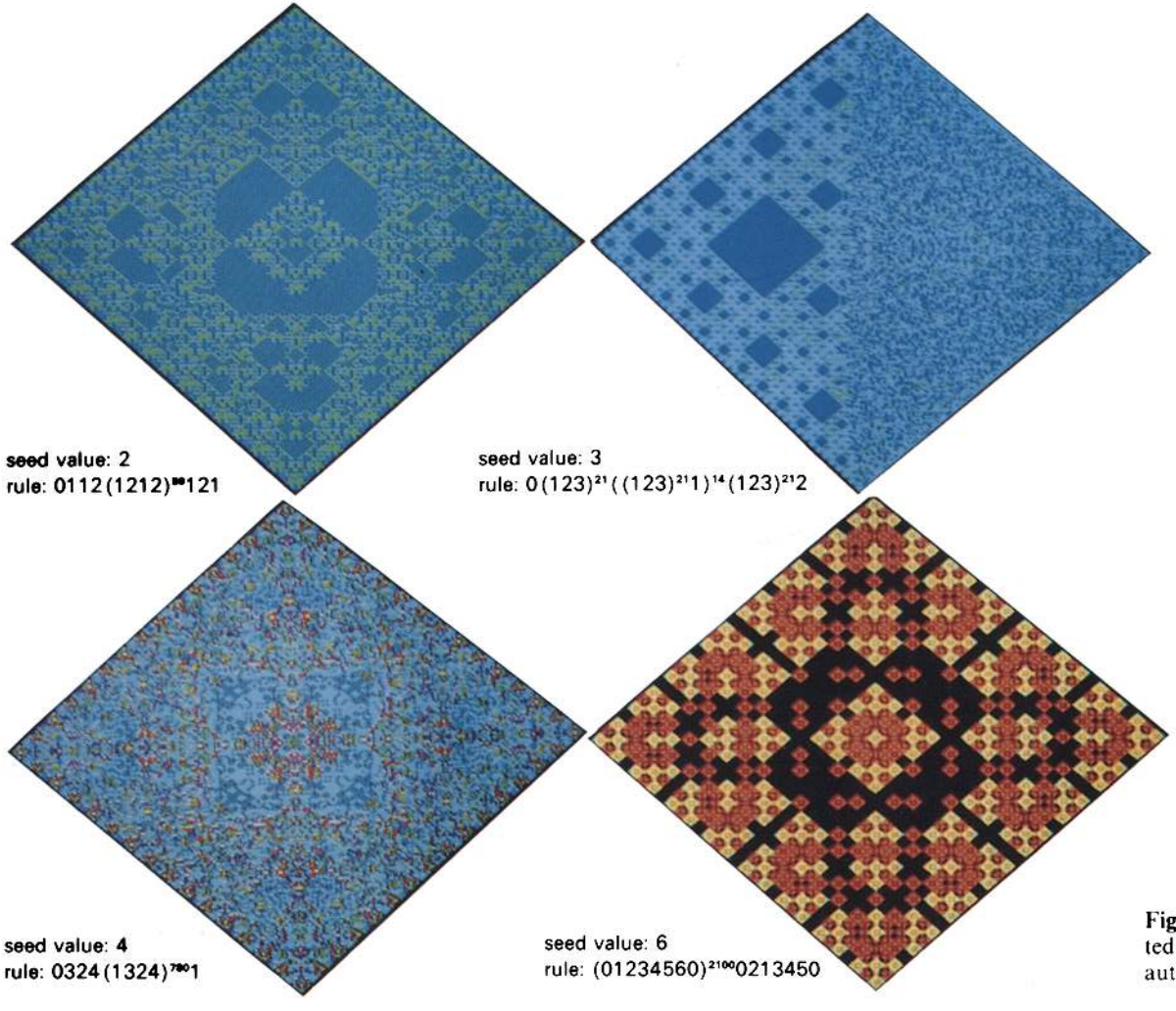

Figure 3. Example patterns generated by 2D, 5-neighbourhood cellular automata.

Plate 2

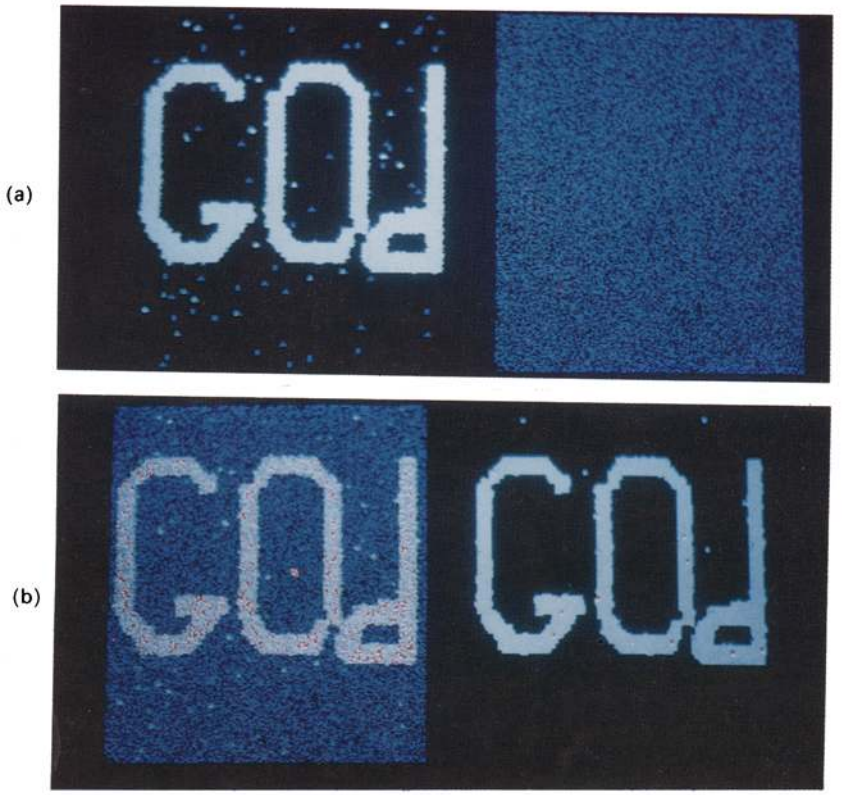

Figure 5. Morphological opening of a gray level image; (**a**) Input image and a noise pattern (size 225 × 225); (**b**) image corrupted with the noise and the cleaned up image.

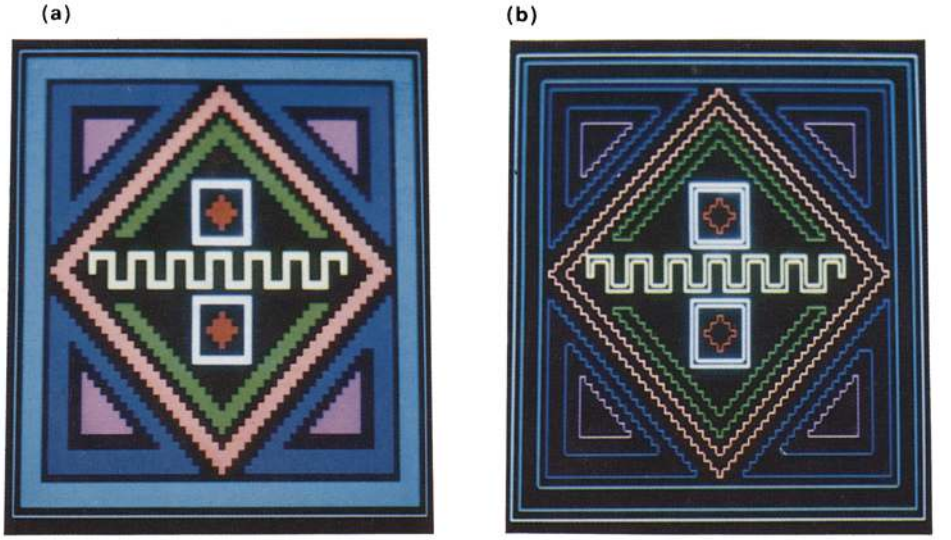

Figure 7. An image (**a**) and its edge detected version (**b**), consisting of various uniform gray level regions.

Plate 3

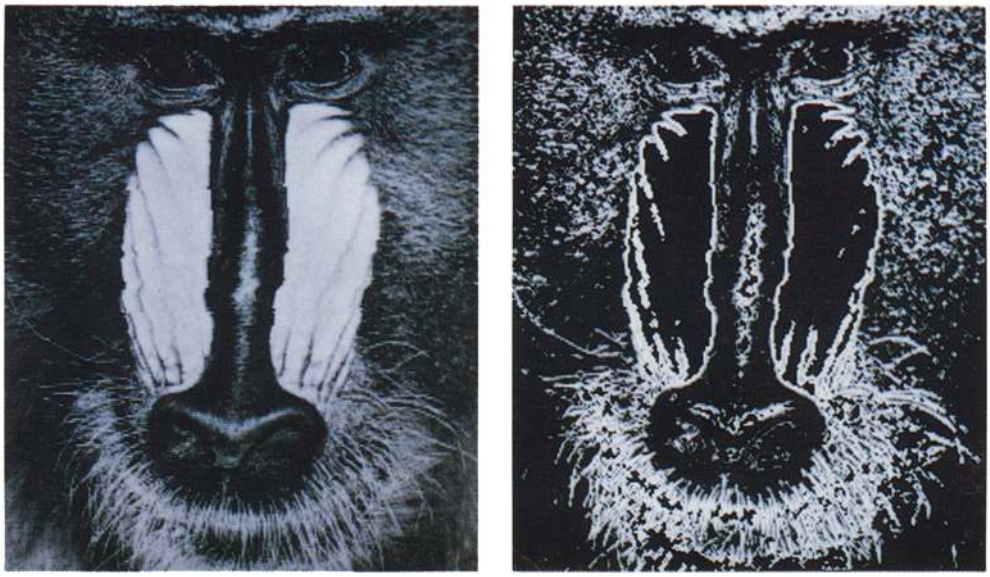

Figure 8. (**a**) Image consisting of regions that appear to be uniform, and (**b**) its edge detected version.

(a) (b)

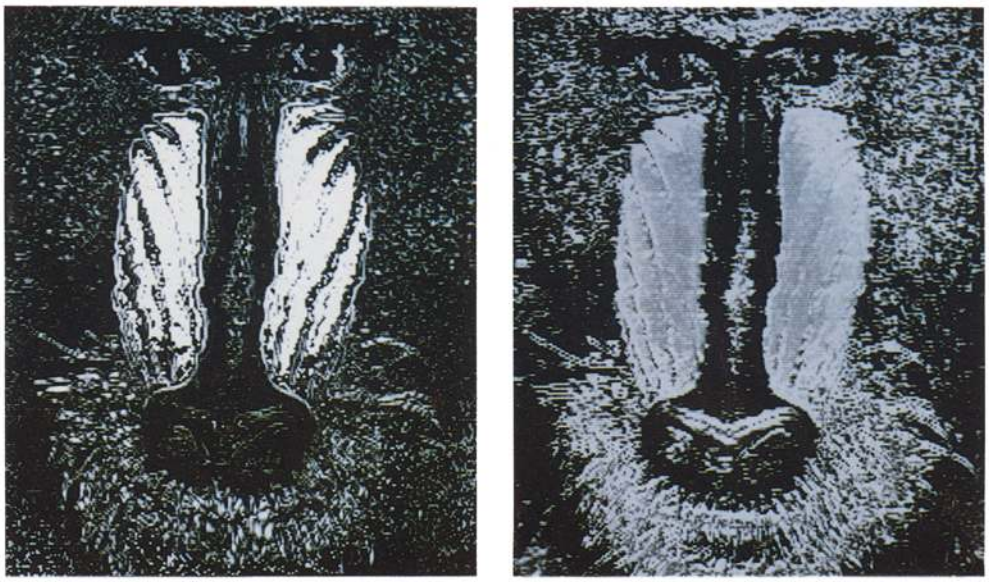

Figure 9. (**a**) & (**b**) (caption on p. 300).

Plate 4

Figure 9. (c)

Figure 9. Edge detection of a gray level image using known algorithms. Original pixel values are altered in (**a**) & (**b**), while in (**c**) contour information is also completely lost.

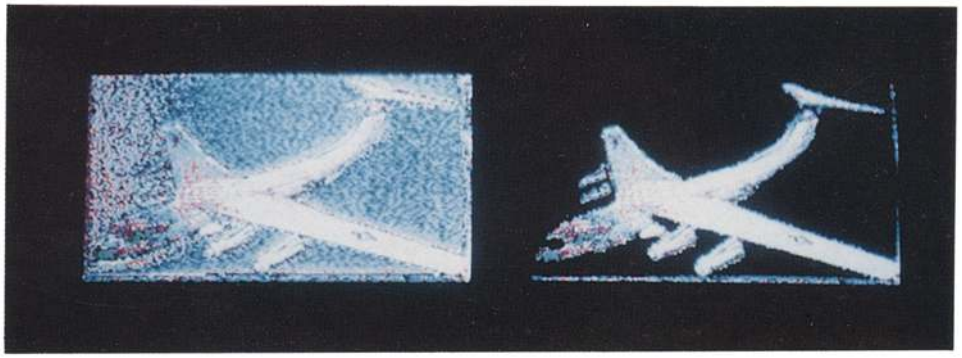

Figure 10. Solid object extraction from a noisy image.