REFERENCES

[1] J. W. Tukey, *Exploratory Data Analysis*. Reading, MA: Addison-Wesley, 1976, ch. 7, pp. 205–236.

[2] —, "The ninther, a robust estimate of location," in *Contributions to Survey Sampling and Applied Statistics*, H. A. David, Ed. New York: Academic, 1978, pp. 251–257.

[3] H. C. Andrews, "Monochrome digital image enhancement," *Appl. Optics*, vol. 15, pp. 495–503, Feb. 1976.

[4] R. W. Floyd and R. L. Rivest, "The algorithm SELECT–For finding the smallest N elements $M1$," *Commun. Ass. Comput. Mach.*, vol. 18, p. 173, Mar. 1975.

[5] T. S. Huang, G. J. Yang, and G. Y. Tang, "A fast two-dimensional median filtering algorithm," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-27, pp. 13–18, Feb. 1979.

[6] D. R. Morgan, "Analog sorting network," *Electron. Design*, vol. 21, pp. 72–73, Jan. 1973. Also, "Correction," *Electron. Design*, vol. 21, Aug. 1973.

[7] J. D. Gibbons, *Nonparametric Statistical Inference*. New York: McGraw-Hill, 1971, pp. 32–47.

[8] D. E. Knuth, *The Art of Computer Programming*, vol. 3. New York: Addison-Wesley, pp. 220–246, 1973.

[9] M. I. Shamos, "A robust picture processing operator and its circuit implementation," in *Proc. ARPA Image Understanding Workshop*, Nov. 1978, pp. 127–129.

[10] W. L. Eversole *et al.*, "Investigation of VLSI technologies for image processing," in *Proc. ARPA Image Understanding Workshop*, Nov. 1978, pp. 191–195.

Patrenahalli M. Narendra (S'73–M'76) was born in Jog Falls, India, on March 16, 1951. He received the B.Eng. degree in electronics from Bangalore University, Bangalore, India, in 1971 and the M.S. and Ph.D. degrees in electrical engineering from Purdue University, West Lafayette, IN, in 1973 and 1975, respectively.

He joined Honeywell Systems and Research Center in 1976, where he is presently a Staff Research Scientist in the Signal and Image Processing Section. His current research interests span dynamic scene analysis, symbolic image processing, and CCD/digital VLSI device architectures for implementing real-time image processing systems. He is the author of several publications in his areas of interest. In 1979, he won the H. W. Sweatt Award, the highest technical achievement award in Honeywell.

Dr. Narendra is a member of Phi Kappa Phi and the IEEE Computer, SMC, and ASSP Societies.

Parallel Image Processing by Memory-Augmented Cellular Automata

CHARLES R. DYER, MEMBER, IEEE, AND AZRIEL ROSENFELD, FELLOW, IEEE

Abstract–**This paper introduces a generalization of cellular automata in which each cell is a tape-bounded Turing machine rather than a finite-state machine. Fast algorithms are given for performing various basic image processing tasks by such automata. It is suggested that this model of parallel computation is a very suitable one for studying the advantages of parallelism in this domain.**

Index Terms–**Cellular automata, image processing, parallel processing, pattern recognition.**

I. INTRODUCTION

MANY basic image operations are local, and therefore can be implemented very efficiently on a parallel computer. These algorithms operate independently on each point of the image and its neighbors and do not make use of any results that may already have been obtained at previously processed points. For this reason, a "cellular" parallel processor array, in which each point of the image has a processor associated with it, can perform many image analysis tasks much faster than a conventional sequential processor which examines the picture points one at a time. See [1] for a review of the development of this type of parallel image processing hardware.

A bounded cellular array automaton (CA) is a convenient theoretical tool for analyzing parallel algorithms for image processing tasks that do not require more than a fixed amount of memory per processor. For many tasks, however, this is an unreasonable restriction. For example, in computing the medial axis of a region it would be most natural if each cell in this skeleton could store its associated distance from the background.

The bounded memory requirement of cellular automata is also too severe a restriction from a realistic point of view since real machines have bounded size. In particular, giving each cell an amount of storage sufficient to hold the address of an arbitrary cell in the array seems reasonable. Indeed, prototype parallel image processing computers such as ILLIAC III [2] (32×32 with 10 bits/cell), CLIP4 [3] (96×96 with 32 bits/cell), and MPP [4] (128×128 with 256 bits/cell) all contain much more memory per cell than the logarithm of the number of processors in the array. Furthermore, memory

Manuscript received July 13, 1979; revised December 27, 1979. This work was supported by the U.S. Air Force Office of Scientific Research under Grant AFOSR-77-3271.

C. R. Dyer was with the Computer Science Center, University of Maryland, College Park, MD 20742. He is now with Department of Information Engineering, University of Illinois, Chicago, IL 60680.

A. Rosenfeld is with the Computer Science Center, University of Maryland, College Park, MD 20742.

0162-8828/81/0100-0029$00.75

costs continue to drop, making larger memories in future computers more and more practical.

Lastly, the finite memory requirement associated with cellular automata was originally specified in part because the automaton was defined over an infinite space and also because the objective was to design a "minimal" structure which could reproduce itself [5]. In early work investigating the language recognition capabilities of cellular automata, e.g., [6], it was realized that an infinite cellular space is inappropriate; hence, the concept of a "bounded" cellular space was introduced to force the automaton to act finitely for any fixed size input. In view of the fact that memory bounds have been extensively used as a measure of sequential automaton computations and that there is a practical limit on the sizes of strings or arrays to be recognized, the historical precedent for a fixed amount of memory per cell is perhaps also too restrictive.

For these reasons, we introduce in this paper memory-augmented bounded cellular automata, in which each cell is now a tape-bounded Turing machine instead of a finite-state machine. In particular, we focus on the case where each cell has memory size proportional to the logarithm of the input size. The capabilities of log-space cellular automata for performing a variety of basic one- and two-dimensional image analysis tasks are described. Unlike ordinary CA's, each cell now has sufficient memory to store its own coordinates and to compute various arithmetic functions whose range is limited by the array size.

Section II defines memory-augmented bounded cellular acceptors and compares their language-accepting power to that of tape-bounded Turing acceptors. Section III presents algorithms for computing various gray level properties of a picture by log-space bounded cellular automata (i.e., each cell has memory size proportional to the logarithm of the picture size). In Sections IV and V we describe algorithms for converting between various representations of a picture subset and measuring geometrical properties of a picture subset by log-space bounded cellular automata.

II. Memory-Augmented Bounded Cellular Automata

This section generalizes the standard definition of a bounded cellular automaton, which specifies that each cell has a finite state set, to allow the memory size associated with each cell to be a function of the input size. That is, we define a cellular analog to the tape-bounded Turing machine.

A *memory-augmented* cell is a Turing machine without input tape, i.e., a 3-tuple $C = (Q, \Gamma, \delta)$, where Q is the finite, nonempty state set, Γ is the finite, nonempty storage tape alphabet, and δ: $(Q \times \Gamma)^3 \rightarrow 2^{(Q \times \Gamma \times \{-1,0,1\})}$ is the transition function if C is nondeterministic, δ: $(Q \times \Gamma)^3 \rightarrow Q \times \Gamma \times \{-1,0,1\}$ if C is deterministic.

A *memory-augmented bounded cellular automaton* is a 4-tuple $Z = (C, Q_I, \#, b)$, where $C = (Q, \Gamma, \delta)$ is a memory-augmented cell, one copy of which is assigned to each integer point on the real line; the copy at coordinate i is called cell i. $Q_I \subseteq Q$ is the set of input states, $\# \in Q_I$ is a special boundary state, and $b \in \Gamma$ is the blank storage tape symbol initially written on every square of the storage tapes. If Z is nondeterministic, the transition function for cell i maps the current states of cells $i-1$, i, and $i+1$, respectively, with their corresponding storage tape symbols currently being scanned, into a set of triples of possible new states of cell i's finite control, new symbols written at the storage tape square where i's head is currently positioned, and directions of movement of cell i's head. If Z is deterministic, then the mapping is into a single (state, symbol, direction of movement) triple. A *step* of computation consists of the simultaneous application of the transition function at each cell. A *configuration* of Z is a mapping from the integers into (Q, Γ^+, j) triples specifying the current state, tape contents, and head position of each cell in Z. For convenience, only those tape positions which the head has visited will be included in that tape's description since the remainder of the tape is known to be blank. The configuration prior to the first time step is called the *initial configuration.*

The boundary state # is used in the usual way to restrict a computation to a bounded number of contiguous cells. That is, an initial configuration of Z is of the form $(\#, b, 1)^\infty (q_1, b, 1)(q_2, b, 1) \cdots (q_n, b, 1)(\#, b, 1)^\infty$, where $q_1 q_2 \cdots q_n$ is a finite, nonnull string in $(Q_I - \{\#\})^+$, called the *input string.* Boundedness is now enforced by restricting the transition function δ to be both #-preserving and write-inhibited on storage tapes of cells in the boundary state. That is, $\delta(p, x, q, y, r, z) = (\#, w, d)$ implies $q = \#, w = y$, and $d = 0$, and $\delta(p, x, \#, y, r, z) = (\#, y, 0)$ for all p, r in Q and x, y, z in Γ. Because of these conditions we will assume without loss of generality that the string of cells is finite, and initially has the form $(\#, b, 1)(q_1, b, 1) \cdots (q_n, b, 1)(\#, b, 1)$.

Z is called an *$L(n)$-space bounded cellular automaton* if $L(n)$ is an upper bound on the number of storage tape squares visited by any cell in Z given any input string of length n and any valid sequence of steps of Z. In particular, if $L(n) = \log n$, then Z is called a *log-space bounded cellular automaton.*

An *$L(n)$-space bounded cellular acceptor* ($L(n)$-*space* CA) is a pair $M = (Z, Q_A)$, where Z is an $L(n)$-space bounded cellular automaton and $Q_A \subseteq Q$ is the set of accepting states. M's leftmost non-# cell is called the *accept cell* or cell 1. An input string $q_1 q_2 \cdots q_n$ is said to be *accepted by* M if given the initial configuration $(\#, b, 1), (q_1, b, 1), \cdots, (q_n, b, 1), (\#, b, 1)$, M's accept cell eventually enters an accepting state after some number of time steps. The set of strings accepted by M defines its *language.* Unless otherwise specified, we assume in the remainder of this paper that M is deterministic.

In two dimensions, an *$L(n)$-space bounded cellular array automaton* is defined by a straightforward extension of the one-dimensional definition. A memory-augmented cell is defined in the same way, except the transition function is now a function of a 5-tuple of (state, storage symbol) pairs. A copy of the cell is assigned to each point in I^2, where I is the set of integers. Cell (i, j)'s next state depends on the local configurations, i.e., (state, symbol) pairs, of itself and its four nearest neighbors, cells $(i-1, j)$, $(i+1, j)$, $(i, j-1)$, and $(i, j+1)$. An *$l \times m$ input array*, $n = lm$, defines the initial states of a rectangular block of cells which are surrounded by a border of #-cells. The accept cell in an $L(n)$-space bounded cellular acceptor is the upper-left corner non-# cell.

A language £ is said to be accepted by a (one- or two-dimensional) $L(n)$-space CA M in $O(f(l,m))$ time if there exists a constant c such that every $l \times m$ array, $n = lm$, in £ is accepted by M within $c \cdot f(l,m)$ time steps. In particular, if $f(l,m) = l+m$, then we say M accepts £ in *diameter time.* If $f(l,m) = lm$, then we say M accepts £ in *area time.* In the one-dimensional case $l = 1$, so diameter equals area time.

While this definition specifies the desired formal extension of bounded cellular acceptors to include augmented memory computations, it restricts storage access to only a single square of the storage tape at each time step. In order to simplify algorithm description and emphasize arithmetic rather than logical operations as primitive, we would like the transition function to "depend" on the entire contents of a cell's storage tape. We will limit this dependence by invoking a unit time cost for certain elementary operations that can be executed by an $L(n)$-space CA in time proportional to $L(n)$. In addition, multiplication will be considered a unit time step operation.

We will consider the storage tape to be divided into a finite number of tracks, called registers, each of length $L(n)$. The unit time criterion will be used for executing instructions such as: set the contents of register i to zero, increment the contents of register j, or copy the contents of register i into register j.

A. Relation to Tape-Bounded Turing Acceptors

In this section we establish the relationship between memory-augmented CA's and tape-bounded Turing acceptors. The following theorems are for the one-dimensional case; the generalization to two dimensions, where n is the array area, is straightforward.

Theorem 1: If a set is accepted by an $L(n)$-tape bounded Turing acceptor, then it is also accepted by an $\lceil L(n)/n \rceil$-space CA.

Proof: Given an $L(n)$-tape bounded Turing acceptor T, with state set Q, input tape alphabet Σ, and storage tape alphabet Γ, construct an $\lceil L(n)/n \rceil$-space CA M which simulates T as follows. M's input state set is $\Sigma \cup \{\#\}$ and tape alphabet is $\Gamma \cup \{b\}$, where # and b are two new symbols. Assume that the input string to T defines the initial states of M's cells. Then M passes a marker from cell to cell to keep track of the current position of T's input head and T's current state. Each cell's finite control permanently maintains the input symbol written at that position of T's input tape. The contents of T's storage tape are distributed evenly on M's n storage tapes using the following mapping. The symbol written on the kth tape square of T's storage tape is stored on M's $((k-1 \bmod n)+1)$th cell's storage tape at the $(\lceil k/n \rceil)$th position. This correspondence is shown in Fig. 1. All of M's storage heads stay in lock step at position $\lceil k/n \rceil$ and M's cells' finite controls pass another marker to indicate the $((k-1 \bmod n)+1)$th cell.

To simulate one transition of T, the two cells in M marking the positions of the input and storage heads initiate bidirectional signals containing the (state, input symbol) pair and storage symbol, respectively, located at these positions. After no more than $n/2$ time steps, the cell halfway between these cells receives all the information it needs to determine T's transition. This cell then returns (new state, direction of input head movement) and (new tape symbol, direction of storage head movement) pairs to the originating cells, and a direction of storage head movement to all cells in M.

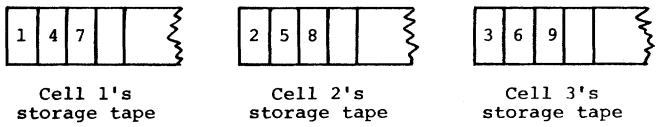


Fig. 1. Mapping from Turing acceptor's tape to CA's tape for $n = 3$.

At the next step the input head marker is moved left, right, or not at all according to the specified direction of movement for T's input head, and this cell enters a new state corresponding to the new state entered by T. Similarly, the cell containing the storage head marker writes a new symbol at the current position and the storage marker is passed to the appropriate cell (i.e., a move left implies the cell's left neighbor copies the mark; a move right implies the cell's right neighbor copies the mark; no move implies the mark remains at the current cell). The direction of storage head movement signal propagates to all cells in M; when a cell receives this signal it moves its storage head in the given direction.

Clearly, this algorithm simulates the action of T except when M's storage head marker attempts to move right from the rightmost cell or left from the leftmost cell. It is readily seen from the definition of the storage mapping function that in either case the marker should be placed at the cell at the opposite end of M. Thus, under these conditions, the rightmost (leftmost) cell sends the marker the length of the string to the leftmost (rightmost) cell. Notice that no special conditions are required for maintaining the proper storage head position, since all of M's cells' storage heads stay in lock step.

The mapping from T's storage tape to M's n storage tapes guarantees that if T scans no more than $L(n)$ storage tape squares, then each of M's cells scan no more than $\lceil L(n)/n \rceil$ storage tape squares. Thus, to simulate a single transition of T, M requires at most $n/2$ time steps to compute the transition, at most n steps to return this information to all of M's cells, and at most n more steps to record the new configuration of T. //

Theorem 2: If a set is accepted by an $L(n)$-space CA, $L(n) \geq 1$, then it is also accepted by an $(n \cdot L(n))$-tape bounded Turing acceptor.

Proof: Construct an $(n \cdot L(n))$-tape bounded Turing acceptor T to simulate an $L(n)$-space CA M as follows. Initially, T copies the input string containing M's cells' initial states, from its input tape to its storage tape. This block of n squares will keep track of the current states of M's cells. On each square of a second track of this length n block, a special tape symbol b and a storage head marker are written.

T will store M's storage tape contents in blocks of size n, so that the ith cell's storage tape contents are distributed on T's storage tape at positions i, $n+i$, $2n+i$, etc. On one of these squares a marker is placed to keep track of cell i's storage head. Initially, then, T recorded the fact that each of M's storage heads is on square 1 scanning the blank symbol.

To simulate one step of M, T systematically scans the first track on the first n squares of its storage tape from left to

right. At each position, T leaves a place holder at the current square i, moves its input head to the left end, and then moves its input and storage heads rightward at unit speed until the input head reaches the right end of the input string. T's storage head is now scanning the square storing the second tape symbol of M's ith cell's storage tape. If this cell contains the storage head marker, then T stores the symbol written on the square in its finite control and moves back to square i. Otherwise, T continues moving rightward in jumps of size n using the input tape as a yardstick until it eventually finds the head marker and returns.

T repeats this process from each of positions 1 to n. Say T's storage head has just returned to square i after finding the symbol currently scanned by cell i's storage head. T remembers the symbols printed at each of the last three storage head positions, so that with the three states stored at positions $i-2$, $i-1$, and i, T can compute the (new state, new storage symbol, direction of storage head movement) triple for cell $i-1$. T then moves left one square and prints the new state on track 1 (while still remembering the previous state of this cell in its finite control). Next, T searches for the storage marker for cell $i-1$ using the same technique given above. Once found, T prints the new storage symbol, erases the marker, and moves left or right n squares, or not all, corresponding to the specified direction of head movement. The marker is then placed on this square to complete the recording of the transition of M's cell $i-1$.

Readily, the interleaving of M's cells' storage tapes guarantees that T uses no more than $n \cdot L(n)$ tape squares. The time required to simulate a single step of M depends on the positions of the storage heads. If cells $i-1$, i, and $i+1$ have their storage heads at squares r, s, and t, respectively, then T's markers are at positions $((r-1)n+i-1)$, $((s-1)n+i)$, and $((t-1)n+i+1)$. To compute cell i's transition takes $2n(r+s+t-3)$ steps and at most another $2sn$ steps to update T's configuration to reflect this single transition of a single cell in M. //

From these two theorems we immediately have the following.

Theorem 3: The class of $L(n)$-space CA languages is equivalent to the class of $(n \cdot L(n))$-tape bounded Turing acceptor languages.

A deterministic, nonerasing stack automaton, introduced in [7], has a two-way input tape, a finite control, and a stack. The stack may be modified only by adding symbols at the top, i.e., no erasing of symbols is allowed. In addition, the stack head may move up or down the stack in a read-only mode.

Corollary 1: The class of languages accepted by log-space CA's is the same as the class of languages accepted by deterministic, nonerasing stack automata.

Proof: Hopcroft and Ullman [7] proved the equivalence of deterministic, nonerasing stack automata and deterministic (n log n)-tape bounded Turing machines. The corollary now follows from the equivalence of (n log n)-tape bounded Turing machines and log-space CA's shown in Theorem 3. //

Corollary 2: There exists a language accepted by a log-space CA, but not by any (constant-space) CA.

Proof: It is well known [8] that there exists a language accepted by an (n log n)-tape bounded Turing acceptor, but not by any n-tape bounded Turing acceptor. The corollary now follows from Theorem 3. //

III. Log-Space CA's for Picture Description

In this section we describe log-space CA algorithms for measuring various gray level properties of pictures. To ease the exposition, the algorithms are described for the one-dimensional case; all of the results generalize immediately to two dimensions. The presentation is informal, at the expense of some precision.

A. Local Property Counting

Smith [9] gives a diameter time procedure due to Meyer for recognizing the majority predicate, i.e., the set of all arrays over input state set $\{0, 1\}$ in which there are more 1's than 0's. That procedure uses a unary to binary conversion technique in order to count the occurrences of each type of input state. A log-space CA can use this same technique to count local properties, except that now only a single cell is needed as the accumulator.

A class of properties which are commonly used for picture description depend on the gray level distribution of the points. For example, statistical features based on the relative frequency with which various local gray level properties occur in a picture are often useful.

In order to compute these features, a picture must first be mapped into a "property space" of measurements taken at each picture point and which depend only on the gray levels of the point and its neighbors, not on the (global) spatial arrangement of gray levels. Since this class of gray level property density functions depend on a fixed number of neighbors and a quantized gray level range, we can assume that a finite range of property values is sufficient. Hence, the domain of the property space is bounded and the range grows with the picture size. The advantage of log-space CA's is obvious here where a fixed set of registers can accumulate the measured values from every point. Examples of such property spaces include histograms and cooccurrence matrices.

The general procedure for computing such features on a log-space CA is as follows. First, each cell computes in parallel its local property value in a bounded number of time steps. Afterwards, each cell routes its value to a designated cell which counts the number of occurrences of each property value. This requires diameter time steps. Finally, the designated cell computes a specified feature based on the distribution of property values in the property space.

The gray level *histogram* of a digital picture quantized to k levels is a vector H, where $H(z_i)$ is the number of points in the picture with gray level z_i. This gray level frequency vector is computed by the leftmost cell in a log-space CA as follows. At time step 1 cell 1 initializes k registers, $H_1, \cdots, H_k$, to 0. Beginning at time step 2 the entire picture shifts left at unit speed. If cell 1 receives gray level z_i, then it increments register H_i. After diameter time steps cell 1 contains the histogram of the picture.

Statistical properties of a picture's histogram are easily computed by a log-space CA. We will assume that the precision of these feature values grows at most linearly with the input size

so that they can be stored at a single cell. For example, techniques for finding valley bottoms (which are often reasonable points at which to threshold a picture) or p-tiles can be directly implemented by the cell storing the histogram. Details will not be given. The mean and variance of a picture's gray levels can also be computed directly from the histogram since the arithmetic operations involved require storing intermediate quantities which are bounded by a fixed multiple of the picture size.

Similarly, the gray level *cooccurrence matrix* of a picture, which measures how often each pair of gray levels occur at a specified relative displacement, can be computed. Cell 1 must now store k^2 registers, where again k is the number of gray levels. The algorithm only differs from the previous one in the initial property measurement, which requires a copy of the picture to be shifted the specified distance, but in the opposite direction, given by the relative displacement of pixels to be compared. Afterwards, these gray level pairs commence shifting leftward and are counted by the leftmost cell.

B. Moments

Another class of gray level properties do depend on the spatial arrangement of gray levels in a region or picture. Moments are examples of such properties. They are often useful as measures of location, shape, and for geometrical normalization. The ith moment of a one-dimensional picture f is defined to be $m_i = \Sigma_x x^i f(x)$.

The coordinate of a picture's centroid is given by m_1/m_0. The pixel closest to a picture's centroid can be marked by a log-space CA in diameter time as follows. In diameter steps the leftmost cell computes the sum of the gray levels in the picture using a modification of the histogramming procedure described earlier. After dividing this count by two, this value begins propagation rightward, each cell subtracting its input gray level from the current value as it passes by. The cell in which this value first becomes nonpositive is the centroid since half of the picture's sum of gray levels is on either side of this point.

If we use the centroid as the origin, the *second central moment* m_2 can be computed by a log-space CA in diameter time as follows. First, the picture's centroid is marked in diameter time. The centroid cell then initiates a signal sent to its left and right, along with an incrementing counter so that each cell can determine its coordinate with respect to the centroid. When a cell's coordinate is determined, it then computes $x^2 f(x)$ in two more steps, since multiplication takes unit time. This value next shifts back to the origin, where it is summed with the other cells' values.

C. Autocorrelation

The autocorrelation of a one-dimensional picture f is defined as $R_f(x') = \Sigma_x f(x)\, f(x - x')$. We now describe how a log-space CA can compute R_f in $O(\text{diameter}^2)$ time by shifting a copy of the picture with respect to the original, and summing the pairwise products of coincident gray levels at each relative displacement.

At step 1 each cell squares its input gray level. For the next diameter-1 steps these values shift leftward, and are summed by cell 1. (Since this sum is no larger than diameter $\cdot k^2$, where k is the number of gray levels, there is no problem storing it in a single register.) When cell 1 adds the value sent from the rightmost cell, its count contains the value of $R_f(0)$, so it initiates a firing squad which starts the computation of $R_f(1)$.

At the next step, a copy of the entire picture shifts right one position. We will assume that the picture is zero outside of its given domain, so at the next step cells 2 through n, where n is the length of the picture, multiply pointwise the input and shifted gray levels stored in each cell. For the next diameter-2 steps these values shift leftward and are summed by cell 2. This process continues until $R_f(n-1)$ is computed by cell n.

The computation of $R_f(i)$ requires one step to shift the picture to the new position, one step to pairwise multiply the cooccurring gray levels, and $n - i$ steps to shift and sum these values. We have not included the timing for the firing squad synchronization since in fact each cell has enough memory to store a clock which counts to $n - i + 2$ and therefore no firing squad is necessary. Thus, the algorithm takes $\Sigma_{i=0}^{n-1}\, n - i + 2 = (n(n+1))/2$ steps, i.e., $O(\text{diameter}^2)$ time steps, to compute R_f.

In two dimensions, R_f can be computed similarly; the computation of $R_f(i, j)$ takes one step to shift the picture to the new position, one step to pairwise multiply the cooccurring gray levels, and $l + m - i - j$ steps to accumulate these values in an $l \times m$ picture. Thus, a two-dimensional log-space CA can compute R_f in $O(\text{diameter} \cdot \text{area})$ time steps.

IV. Log-Space CA's for Region Representation

There are a variety of approaches to representing connected components, or regions, since each type of representation has its own advantages. For example, chain codes are very compact and make it easy to detect region boundary features, but are not useful for determining properties such as elongatedness. Thus, it becomes desirable to develop efficient methods of converting from one representation to another.

This section first considers methods of labeling and counting regions, and then presents two-dimensional log-space CA algorithms for transforming an array representation to several alternatives, and vice versa. In particular, we consider region representations using run length codes, chain codes, medial axis transforms, and quadtrees. First, we present some definitions and terminology concerning digital topology.

Given a point p in a digital picture, its four horizontal and vertical neighbors are called its *4-neighbors*, and are said to be *4-adjacent* to p. These four neighbors together with p's four diagonal neighbors are called p's *8-neighbors*, which are each *8-adjacent* to p. A *4-path* (8-path) from p to q is a sequence of picture points $p = p_0, p_1, \cdots, p_n = q$ such that p_i is 4-adjacent (8-adjacent) to p_{i-1} for all $1 \leqslant i \leqslant n$. A (4- or 8-) path is called a (4- or 8-) *geodesic* if no shorter path with the same endpoints exists. Given a picture subset S, we say that p is (4- or 8-) *connected in S* to q if there is a (4- or 8-) path from p to q consisting entirely of points in S. The equivalence classes under the equivalence relation "connected in S" are called the *connected components*, or *regions*, of S.

The subset of picture points not contained in a specified sub-

set S is denoted by $\bar{S}$. $\bar{S}$ can also be decomposed into connected components. If we assume that the picture is embedded in a larger picture consisting only of points in $\bar{S}$, then exactly one of $\bar{S}$'s components contains this set of "boundary points." This region is called the *background* of $\bar{S}$ and all other components of $\bar{S}$ are called *holes* in S. If S has no holes, it is called *simply connected*. The *border* of S is the set of points of S that have at least one neighbor in $\bar{S}$. If p is in the border of S, then we call p a *border point* of S. More specifically, those border points which are adjacent to the background are on the *outer border* of S, the other points are on *hole borders* of S. The set of points of S which are not on the border of S are called *interior points* of S.

A. Connected Component Counting and Labeling

Given a binary image, it is known [6], [10] that a CA can determine in diameter time if the 1's comprise a single connected component. The method shrinks each connected component of 0's or 1's in parallel to a single point, the upper-left corner of the component's upright framing rectangle, without disconnecting or merging components in the process. Readily, that algorithm can be converted to a log-space CA algorithm which counts connected components as follows. At the same time that a component disappears, the cell at that position detects this occurrence and creates a special marker which continues propagating to the upper-left corner cell in the array. Markers which meet during propagation continue as a single marker with an associated count indicating the sum of the number of components associated with each of the two merged markers.

We now consider the problem of assigning distinct labels to the connected components of a picture subset S, i.e., all pixels in the same component are given the same label, but no two pixels in different components may have the same label. Since there may be an unbounded number of components in a picture, a CA does not have enough states to distinctly label all of the components. (Consider the checkerboard picture which has O (area) 4-components of one pixel each. Each cell has a bounded number of states to store its label, but an unbounded number of labels are needed.) Thus, this problem is only meaningful on a log-space CA where enough storage is available for label names.

To solve the problem, we divide it into the following subtasks. First, a distinguished cell in each component is located. This cell generates a unique label for the component it represents, and then broadcasts the label back to the other cells in the component. We assume the input state set is $\{0, 1\}$ and we wish to label all 4-connected components of 1's in the picture.

Kosaraju has shown [11] that a CA can mark in diameter time a distinguished cell in each connected component using an algorithm based on fast multiplication of Boolean matrices. Each of these distinguished cells must now generate a unique label for the component it represents. One simple way is for each cell to use its matrix coordinates in the array as the label. This computation can be performed in parallel with the first phase as follows. The upper-left corner cell computes at step 1 its coordinates $(0, 0)$ in the array and then enters state p; all other cells are in state q. Beginning at the next step, if a cell c is in state q and either its left or upper neighbor is in state p, then c copies that neighbor's coordinates, increments the appropriate coordinate so that c's pair is now its matrix coordinates, and enters state p. Clearly, after twice diameter steps every cell (in particular, every distinguished cell) has computed its coordinates.

Each distinguished cell must now broadcast its label to all cells in the component. Kosaraju has also shown [11] that components can be labeled sequentially in diameter time each, but his method does not allow simultaneous labeling of all components. Alternatively, we now describe an algorithm which can be applied in parallel at every component.

The pixel labeling technique is similar to the coordinate computation algorithm, except that a label is passed only to neighbors which are part of the component. The ordered pairs of neighboring cells in which labels are passed defines a set of directed arcs from the distinguished cell to every cell in the component. Furthermore, these arcs define minimum length paths from the given cell to every other. Thus, in addition to the labeling process, subsequent procedures may have use for these minimum length intrinsic paths. We now formalize the definition and construction of this rooted minimum spanning tree (MST). This structure makes no demand on the augmented memory, and so can also be constructed by a CA.

Given a specified point c_0 in a 4-connected component S, define the *minimum spanning tree of S rooted at c_0* to be a rooted directed acyclic graph in which the vertices are the points of S and arcs exist between horizontally or vertically adjacent vertices such that the following properties are satisfied.

1) Vertex c_0 is the root, i.e., there are no arcs directed from c_0 to any of its neighbors.

2) Every vertex except the root has exactly one arc which is directed from it.

3) Every vertex u is connected to the root c_0 by a unique path $(v_1, v_2), (v_2, v_3), \cdots, (v_{n-1}, v_n)$, where $u = v_1, c_0 = v_n$, and (v_i, v_{i+1}) is the arc directed from v_i, for all $1 \leqslant i \leqslant n - 1$.

4) The path associated with each vertex u to the root is of minimum length, i.e., there does not exist another choice of arcs between horizontally or vertically adjacent vertices (of S) which satisfies properties 1)-3) and results in a shorter sequence of arcs from u to c_0.

Fig. 2 shows an example of a component and a rooted minimum spanning tree for one of its points.

We now show how a CA with a distinguished cell c_0 in a component S can construct its MST. The algorithm, based on a similar one by Beyer [6], takes O (intrinsic diameter of S) steps.[1] Initially, we assume each cell in S is in state 1 and its A register is set to zero, and all other cells are in state 0. The result of this algorithm will be to set each cell's A register to 0, 1, 2, 3, 4 according to whether there is no arc directed from this vertex, or an arc is directed to the vertex's upper, left, lower, or right neighbor, respectively.

At time step 1 cell c_0 enters state p for one time step and then enters state q. Every other cell in state 1 remains in state 1 until at least one of its neighbors enters state p. At the next time step this cell enters state p for one step before entering

[1] For the definition of intrinsic diameter, see Section V.

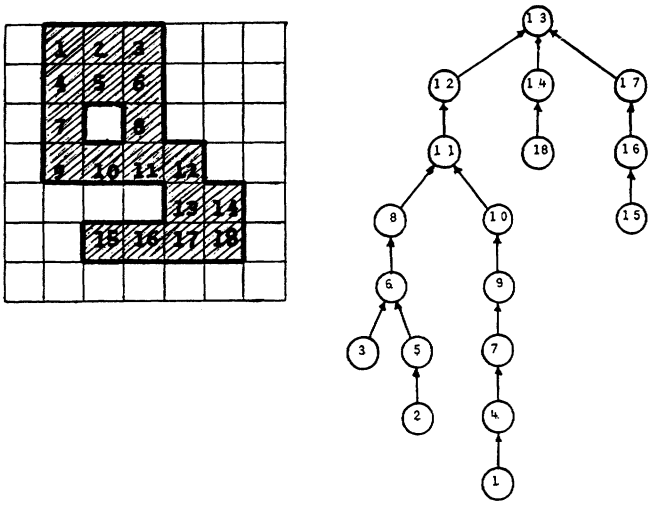


Fig. 2. A region and its rooted MST for point 13.

state q, setting its A register to either 1, 2, 3, or 4 depending on which neighbor was in state p (in case of multiple p-neighbors, choose one according to the precedence, say, upper, left, lower, right). In this way a signal p propagates from cell to cell through S, each cell in S entering state p after a number of time steps equal to its distance through cells in S from c_0. Thus, the chain of direction links, stored in each cell's A register, from a cell back to c_0 is a minimum length path.

If when a cell enters state q all of its four neighbors are either in state 0 or q, then the current cell entered state p at the same time as or later than all of its neighbors in S. Designate these cells which are a local maximum distance from c_0 as the *leaf* vertices in c_0's MST. At the time step after a leaf cell enters state p it enters state r. Each cell in state q remains in that state until all of its sons (i.e., neighbors with links directed back to it) enter state r; then it enters state r. Thus, the r signals are initiated by the leaf cells and propagate back up the tree signaling the completion of the algorithm.

In the remainder of this section we consider alternative representations for a given labeled connected component, determining how fast a log-space CA can transform an array representation to each alternative, and vice versa.

B. Run Length Codes

Given a component S, each row of the picture consists, in general, of runs of pixels in S separated by runs of pixels in $\bar{S}$. Thus, we can represent S by a list of run (starting position, length) pairs. If the runs, on the average, are sufficiently long, then this representation is more compact than, and yet an exact coding of, the original array representation of S. Since S is connected it cannot skip rows, and therefore we can shorten the code further as follows. The run length code of S consists of a starting row "header message," followed by run (starting column, length) pairs, with a punctuation bit separating runs on adjacent rows.

A log-space CA can output the run length code of a connected component S at a designated cell, say the accept cell, as follows. Assume cells in S are initially in state 1, all others are in state 0. At time step 1 cells at the left and right ends mark themselves. Beginning at time step 2 each right end cell initiates a signal which is sent to the left end of the run. In conjunction with this each left end cell increments a counter, initially set to zero, at each step until the signal arrives at the cell. Thus, when the signal arrives each run's length (minus one) is stored at the left end cell of the run.

Next, each row's runs are packed in order at the left end of the row. That is, if a row has k runs, then the ith run's length-count shifts left as far as it can, stopping at the ith cell from the far end. With each shift left, a coordinate counter is also incremented, so that when the shifting stops this counter is equal to $x - i$, where x is the column containing the leftmost point in the ith run.

This set of runs is now output in row-major order as follows. The leftmost run in the top row of S marks itself as the first run after detecting that the cell above it contains no run description. At the next step this run description begins shifting up the left column to the accept cell while incrementing a row-coordinate counter. Thus, when the first run is output, it also outputs the header message containing the starting row of S. The other runs in the top row shift left to the first column and then up to the output cell, following immediately behind the run descriptions ahead of them. Whenever a run description shifts left, its associated coordinate counter is incremented; thus when each run turns up the first column this counter contains the run's starting column.

The first run in every other row waits until the cell above it has shifted out all of its row's run descriptions. Then this run inserts a new row punctuation mark between it and the last run of the previous row before it commences to move up the left column to the output cell. In the worst case there can be O (area) runs in S, requiring area time to output this representation. (In such situations, however, run length coding would not be the appropriate representation anyway.)

Reconstruction of S from its run length code is straightforward. The first run of S, containing the starting row of S, shifts down the first column, the header-message counter being decremented and tested for zero by each cell that receives it. When this row counter is zero, the current cell marks itself as the starting row. The (starting column, length) pairs shift down the first column to the marked cell, then across the row to the run's starting column. The length counter continues shifting right from this cell; each cell that receives a nonzero counter marks itself in S, decrements the count, and sends it on to its right. When a punctuation bit travels down the left column and arrives at the marked current row, that cell's mark is erased and the mark rewritten at the cell below it.

C. Chain Codes

If a component is relatively compact, then its perimeter is proportional to the square root of its area. This implies that such regions can be efficiently stored by saving a description of their border points only. Trivially, a CA can mark a 4- (8-) component's border points in one (two) step(s).

Given a starting border point and an adjacent background point, we can traverse these border points by following the border while always "keeping our right hand" on S [12]. In fact, the direction of traversal of a given border through a given border point by this sequential border following rule is locally

computable in a 3 X 3 neighborhood of the point. Thus, each border point can determine, for each border passing through it, its predecessor and successor border points. If we represent each successor point of a border point by an octal code, where the correspondence between digits and neighbors is

```
321
4*0
567
```

then the border of S can be described exactly by specifying the coordinates of a starting point and an ordered sequence of octal digits. Furthermore, the "right-hand rule" implies that the inside of S's border is always determined by the link direction, so no other information is needed to represent S.

To output the chain code of a component S at a designated cell it is first necessary to find a distinguished starting point for each border of S. S and each connected component in $\bar{S}$ can locate such a point in diameter time using the method in [11]. Given a component in $\bar{S}$ with designated cell c on its border, specify one of c's 8-neighbors in S (if any exist), as the designated starting cell of this hole border of S.

Assume that each cell has previously computed its coordinates, and that associated with each starting border point's link are its coordinates. To output the chain code of S at the accept cell c_0, the starting cell of the outer border d first establishes an output path to c_0. Each cell along this directed path and the directed path of outer border cells now acts in "bucket-brigade" fashion, passing the chain code link by link clockwise around the border to d and then along the output path to c_0.

A hole border's chain code is passed counterclockwise around its border to its start cell. From this cell the links are passed up its column until they hit another border of S. Here the procession of links waits until that border's code has passed, and then follows it in the same direction around its border and eventually to the output cell c_0. Thus, hole border codes "bubble up" around other holes above them until they reach the outer border. Consequently, to output S's chain code requires O (perimeter of S) time steps, where perimeter is the number of border points in S. Notice that log-space is used only to store the border's starting coordinates, not the chain codes themselves.

To reconstruct S, the first link, containing the coordinates of S's outer border's starting point, establishes a path back to the starting cell. The remaining links of the chain code follow immediately behind and reidentify border cells from their link types after a counterclockwise traversal of the partially reconstructed border of S back to the link's successor cell. The direction of the link also defines which side is the interior of S. A link which starts a new hole border departs from the outer border at cell d and heads directly for its starting location. From there, the border is reconstructed in a clockwise order around the hole.

We can relabel the interior points of S since each border point knows which direction is the inside of S, and every run of S is bounded by a pair of border points. Hence, each border cell can initiate a signal to cells in the proper direction in its row, marking them as part of S until it meets another border point or an opposing relabeling signal. Since these signals are stopped by the opposing border point in its run, it is necessary to wait until the border has been completely reconstructed before commencing this process. If S has no holes, then the starting cell d sends a signal around to every border point of S after the final link has been reconstructed. Otherwise, each hole border's starting cell waits until its border has been completed, and then sends a completion message to the outer border's starting cell. When the outer border's starting cell has received completion messages from all of its hole borders, it commences the relabeling process as follows. The outer border points are signaled directly from the starting cell, which sends a message through this border. These cells then relabel points until they hit the opposing border. If the relabeling signal hits a hole border, then this initiates a signal through all cells of this border to begin the relabeling process. In this way runs of S which are bounded on both sides by hole borders are eventually filled in as desired.

D. Medial Axis Transformation

Given a connected component S of cells in state 1 and all other cells in state 0, we can associate with each point in S its distance to the closest point of $\bar{S}$. This *distance tranformation* of S is readily computed by a log-space CA as follows. Beginning at step 1 each cell increments a counter at each step as long as the cell is in state 1. At step 1 each cell in state 1 which has at least one of its neighbors in state 0, enters state 2. Subsequently, any cell in state 1 with at least one of its neighbors in state 2 enters state 2. Thus, at each step all border points of S are changed to 2's, so that S is successively "thinned" until it disappears. Each cell's counter records how long it takes for the point to be deleted from S, and thus contains its distance from $\bar{S}$.

The set of points in S whose distances from $\bar{S}$ are local maxima (i.e., no neighboring point has greater distance from $\bar{S}$) defines the *medial axis* of S. This set is easily computed from the distance transform in four times steps by comparing each cell's counter with its four neighbors' counters.

If we associate with each medial axis point its distance from $\bar{S}$, we obtain an exact representation of S called its *medial axis transformation*. Thus, diameter + 4 steps are required in the worst case to compute the medial axis transformation for all components in a picture.

Reconstruction of the array representation of S from its medial axis transformation is accomplished by reversing the skeletonization procedure so that border points are added at each step and S grows back to its original size.

E. Quadtrees

Quadtrees are an approach to region representation based on successive subdivision of a $2^n \times 2^n$ array containing a region into quadrants. If the component does not cover the entire array, we subdivide the array, and repeat this process for each quadrant, each subquadrant, etc. until we obtain blocks (possibly single pixels) that are either entirely contained in the component or entirely disjoint from it. This process is represented by a tree of out-degree four in which the root node corresponds to the entire array, the four sons of a node are its quadrants, and the leaf nodes correspond to those blocks

for which no further subdivision is necessary. The root node is said to be on level n; the level of an arbitrary node is one less than the level of its father node.

A log-space CA can compute the quadtree representation of a region S, storing each node at the center of its $2^k \times 2^k$ grid square, as follows. The construction of the quadtree will be bottom up, the center cell of each grid square on level k routing information about its base's contents diagonally across the array to the center of the $2^{k+1} \times 2^{k+1}$ square of which it is a quadrant. A node on level $k+1$ then computes whether or not it is in S's quadtree from the information passed from its four sons. If all four sons are leaf nodes of the same type, then the current node must delete its sons from the tree and insert itself as a leaf since its entire base is either in S or $\bar{S}$. Otherwise, the current node inserts itself as a nonleaf node in the tree since points in both S and $\bar{S}$ are in its base. This process continues until the root of the quadtree is determined after at most diameter steps.

More specifically, assume that the input picture is size $2^n \times 2^n$ and each cell has stored its matrix coordinates in the array. Since a cell must represent at most n nodes in the quadtree, this is easily stored as a bit vector at each cell. That is, bit i is equal to 1 iff this cell is representing a node on level i. Another vector of length n keeps track of the node type (i.e., leaf node in S, leaf node in $\bar{S}$, or nonleaf node). At step 1 each cell detects whether it is in S or $\bar{S}$ and creates a leaf node for this pixel. Next, each cell determines the direction of its father node from the least significant bits of its coordinates. That is, a cell's least significant column bit c_1 equals zero if it is in an even numbered column [remember, the upper-left cell is at position $(0, 0)$] so $c_1 = 0$ implies that the cell's father is in a column to its right. Similarly, testing the value of a cell's least significant row bit indicates whether its father is in a row above or below its own row. Thus, combining the information from these two bits indicates which of four possible diagonal directions to move in order to find one's father in the quadtree.

Beginning at the next time step, each cell sends in the indicated direction a signal specifying whether or not its node is in S. We define the center of a $2^k \times 2^k$ block of cells to be the upper-left corner cell in the 2×2 block of cells which surround the center point. Thus, at the end of two steps, the upper-left corner cell of each 2×2 block has received the signals from its sons.

If all four sons are either in S or $\bar{S}$, then the current cell becomes a leaf node on level 1 of the quadtree and returns a message to its four sons to delete themselves. Otherwise, only some of the current node's sons are in S, so the cell makes this node a nonleaf node in the quadtree.

Nodes on level $k+1$ are computed after each cell representing a node on level k routes a message about its node type to its father. Sons do not need to know the coordinates of their fathers, but only the proper direction, as determined by the kth bits of their own coordinates. 2^k times steps after the sons' signals have been sent, quadruples of signals collide at the cell which is their father node (Fig. 3). Each cell representing a node on level $k+1$ then computes its node type, and routes a copy of this information on towards its father. If this node is a leaf, then a delete message is returned to each son. In the worst case, the root of S's quadtree is on level n at cell $(2^{n-1} - 1, 2^{n-1} - 1)$. This node is computed 2^n steps after level 0 nodes are determined. Thus the quadtree representation of a region is computable in diameter time.

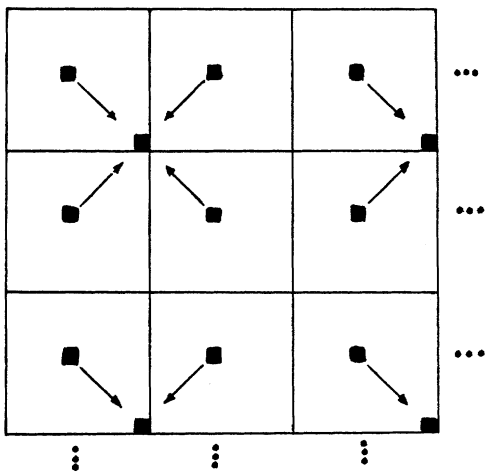


Fig. 3. Nodes routing signals to their fathers during computation of the quadtree representation of a region.

To reconstruct a region from its quadtree, traverse the tree top-down in parallel until a leaf node is reached. This node then relabels all of the cells in its base. The traversal and relabeling are most easily implemented if a cell first computes the coordinates of its sons and of the four corner cells in its base, respectively.

V. Log-Space CA's for Region Description

Section III described methods by which log-space CA's can measure certain gray level properties of one-dimensional pictures. Those techniques can be generalized to two dimensions and so will not be discussed further. In this section we investigate the ability of log-space CA's to measure geometrical properties of a region in a two-dimensional picture. Geometrical properties, unlike gray level properties, depend only on which points of the picture belong to the region, not on the gray levels of these points. In particular, we describe how log-space CA's can measure the geometrical properties area, perimeter, compactness, elongatedness, width, height, diameter, and convexity. We begin by reviewing some basic concepts of distance and diameter in digital pictures.

Given two points $p = (x, y)$ and $q = (u, v)$, define their *city-block distance* as $d_4(p, q) = |x - u| + |y - v|$, and their *chessboard distance* as $d_8(p, q) = \max(|x - u|, |y - v|)$. For simplicity, from now on we only present definitions which result from using city-block distance and 4-connectedness. An analogous set of definitions also exist using chessboard distance. Given a 4-connected component S and p, q in S, it can be shown [12] that $d_4(p, q)$ is just the length of a shortest 4-path from p to q through points in the picture. If we restrict the path of points to be entirely contained in S, then the length of a shortest such path is called the *intrinsic distance* between p and q.

The *4-diameter of S* is defined as the greatest city-block distance between any pair of points of S. By the *intrinsic 4-diameter of S* we mean the greatest intrinsic distance between

any pair of points of S. The *area of S* is the number of points in S; the *perimeter of S* is the number of border points of S, i.e., the number of points in S which have at least one neighbor in $\overline{S}$.

The time required to measure connected component properties will be specified in terms of the diameter, intrinsic diameter, and perimeter of a component. That is, if a log-space CA is shown to compute a given property of an arbitrary component S within $k \cdot$ (diameter of S) steps, for constant k, then we say that the property is computable in *region-diameter time.* Similarly, we say an algorithm takes *intrinsic-diameter time* if the property is computed in a number of steps proportional to the intrinsic diameter of the component. *Perimeter time* implies the number of steps is proportional to the component's perimeter.

A. Area

The *area* of a component in a digital picture is defined to be the number of pixels in the component. Smith [9] gives a diameter time CA algorithm for determining whether or not there are more 1's than 0's in a binary picture, in which 1's are counted in each row and then these row counts are summed. That algorithm is easily modified to compute the area of a component in region-diameter time.

In contrast, a log-space CA can compute the area of a component S in region-diameter time by modification of the Beyer/Levialdi CA algorithm [5], [11]. Briefly, each cell in S initializes a counter to 1 at time step 1, and each cell in $\overline{S}$ initializes a counter to 0. The shrinking algorithm starts at step 2. Whenever a point in S is deleted, one of its upper or left neighbors which is in S (and will not be removed by the connectedness criterion) adds the contents of the deleted cell's counter to its own counter. Clearly, when S is reduced to a single point, that cell's counter contains the number of pixels in S.

Both of the above algorithms make use of cells outside of S. However, this may cause problems if we want to simultaneously compute the areas of all components in a picture. Therefore, we now present an alternative, based on the minimum spanning tree of a component, which is restricted to those cells in S and hence can be used to compute in parallel all components' areas. Under this restriction, however, intrinsic-diameter time is required.

A log-space CA can compute the area of a component S and store it at a designated cell in intrinsic-diameter time as follows. Assuming a unique cell c_0 has been marked previously, the minimum spanning tree rooted at c_0 is constructed using the method described in Section IV-A. During this procedure each cell in S also initializes a counter to 1. Each leaf cell in the MST initiates a reply signal r which propagates back to c_0 along the path indicated by the direction links stored in the cells. Each nonleaf cell enters state r after adding the contents of all of its sons' counters to its own counter. Thus, its counter contains the number of cells in its subtree. In particular, after a number of time steps at most equal to the intrinsic diameter of S, c_0 enters state r and its counter contains the number of pixels in S.

B. Perimeter

The *perimeter* of a component S can be defined as the number of its border points, or as the total length of its borders' chain codes. In either case, Section IV-C showed how a CA can detect these points and compute the chain links in two time steps by looking in a 3×3 neighborhood around each point. The log-space CA algorithm which output the chain code at a designated cell can also be readily adapted to count the number of links or border points as they are output. (Note: if desired we can add $\sqrt{2}$ for diagonal links.) Since this process sequentially propagates the links around the border, it requires perimeter time to compute S's perimeter.

Alternatively, a modification of the Beyer/Levialdi CA algorithm similar to the one described for computing a component's area could be used. In this case, each cell initializes its counter to 1 if it is a border point of S; otherwise, the counter is set to 0. After region-diameter time, the component is reduced to a single cell whose counter contains the number of border points of S.

C. Compactness and Elongatedness

Various measures are used for quantifying the shape of a component. The *compactness* of a component is usually measured by A/P^2, where P is perimeter and A is area. We have just shown how a log-space CA can compute and store P and A in region-diameter time. If we assume multiplication and division take unit time, then two more steps are required to complete the computation.

The *elongatedness* of a component is measured by A/W^2, where W is the number of "shrinking" steps required to delete the component. A shrinking step consists of the deletion of all border points of the component. Thus, W is the maximum value in the distance transformation of the component. A log-space CA can compute W and store it at a designated cell of a component S as follows. First, the designated cell is located and the minimum spanning tree rooted at that cell is constructed. Simultaneously, the distance transformation of S is computed. Next, the leaf cells initiate reply signals which move back to the designated cell. When a cell receives the reply signal it compares its distance from $\overline{S}$ with the distances of all its sons, and stores the largest value as its new distance. Thus after a cell receives the reply signal, it stores the maximum distance of any cell in its subtree from $\overline{S}$. In particular, the designated cell receives the reply signal after intrinsic-diameter time. Again, if we assume multiplication and division are unit time operations, then elongatedness can be computed in two more time steps. Alternatively, we could again use the Beyer/Levialdi shrinking algorithm to find the maximum value in time proportional to the diameter of S.

D. Height and Width

The *height* and *width* of a component S are the distances between the highest and lowest rows, and the leftmost and rightmost columns, of the picture that contain S, respectively. Again, either the MST or Beyer/Levialdi method can be used to propagate and coalesce local information (in this

case four extremum coordinates) into a global region property measurement.

E. Diameter

The *diameter of a component S with respect to a point* c_0 is defined as the maximum distance between c_0 and any point of S. A log-space CA can compute the 4-diameter of a component S with respect to a point c_0 of S by using the minimum spanning tree of the picture rooted at c_0. At time step 1 cell c_0 initializes a counter to zero, begins counting at half speed and initiates the construction of its minimum spanning tree. Each border point of S initiates a signal which propagates back to the root at unit speed. If two signals arrive simultaneously at a node, then they both must have originated at an equal distance from this cell, so only a single signal continues. Since the tree grows at unit speed and a signal returns at unit speed, when c_0 receives the signal its counter contains the distance to the cell that originated the signal.

One of the four corners in the picture must be a maximum distance from c_0. Therefore, when each of these corner cells becomes part of the tree, it initiates a nondestructable signal back to c_0. By the time c_0 receives all four corner signals, after at most twice diameter steps, it must have received all of the signals sent by the border points of S. So the number stored in c_0's counter is the 4-diameter of S with respect to c_0.

Computing the diameter of S with a log-space CA appears more difficult because running the above minimum spanning tree algorithm simultaneously from every point of S would require each cell to store O(area of S) arcs, one for each tree rooted at a point of S. We now show that an alternative method can be used to obtain a region-diameter time solution to this problem. First, we derive some properties of the chessboard and city-block metrics which will be exploited to yield fast algorithms.

Theorem 4: The 8-diameter of a component is equal to the length of the longest side of the component's upright framing rectangle.

Proof: It is easily seen by the definition of chessboard distance that the maximum distance between any two points in an $m \times n$ upright rectangle is equal to $\max(m-1, n-1)$, i.e., the distance from the upper-left to the lower-right corner. In fact, this is the distance between any pair of points which are on opposite short sides of the rectangle. To see this, consider a rectangle m rows high and n columns wide, where $m \geq n$. The distance from an arbitrary point u in the top row to an arbitrary point v in the bottom row is equal to $d_8(u, v) = \max(m-1, |j-i|)$, where u is at coordinate $(1, i)$ and v is at (m, j). For all $1 \leq i, j \leq n$, $|j-i| < n \leq m$; hence $d_8(u, v) = m-1$.

Given a component S, its upright framing rectangle S' just contains S so there must be points of S in the top and bottom rows and the left and right columns of S'. Since $S \subseteq S'$, the 8-diameter of S must be no greater than the 8-diameter of S'. But the existence of points of S on each side of S' implies that the 8-diameter of S equals the 8-diameter of S', which is just the length of the longest side of S'. //

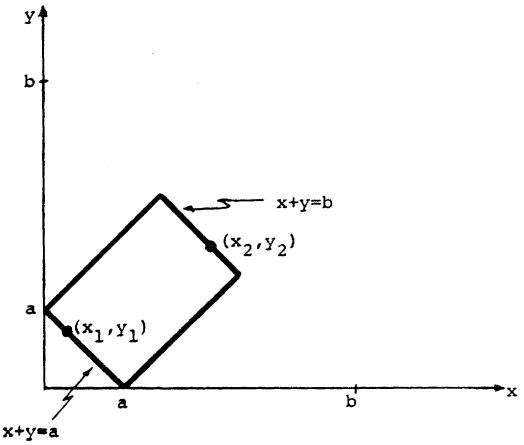


Fig. 4. $d_4((x_1, y_1), (x_2, y_2)) = b - a$ for all pairs of points on opposite short sides of a 45° tilted rectangle.

It can be shown [13] that a component's upright framing rectangle can be constructed by a CA in region-diameter time by repeatedly filling concave corners. Thus, a log-space CA can compute a component's 8-diameter from this rectangle in diameter-of-S more steps.

Similarly, using the city-block metric we have the following.

Theorem 5: The 4-diameter of a component is equal to the length of the longest side of the component's tilted framing rectangle.

Proof: A region's tilted framing rectangle is the smallest enclosing rectangle with sides inclined at ±45° to the picture's sides. The proof is analogous to the proof of Theorem 4, since again it can be shown, this time from the definition of city-block distance, that any pair of points on opposite short sides of any tilted rectangle are at the same distance from one another. To see this, consider Fig. 4 in which, without loss of generality, the short sides have slope −1. Let (x_1, y_1) be an arbitrary point on the side having y-intercept a and (x_2, y_2) an arbitrary point on the opposite side having y-intercept b. Since (x_1, y_1) and (x_2, y_2) are on the short sides of the tilted rectangle, it is readily seen that $x_2 > x_1$ and $y_2 > y_1$. This implies that $d_4((x_1, y_1), (x_2, y_2)) = (x_2 - x_1) + (y_2 - y_1) = (x_2 + y_2) - (x_1 + y_1)$. But the slopes of these sides imply that $x_1 + y_1 = a$ and $x_2 + y_2 = b$. Hence, $d_4((x_1, y_1), (x_2, y_2)) = b - a$, which is just the length of the longest side of the tilted rectangle. The theorem now follows from the fact that a component's tilted framing rectangle by definition contains points of the component on each of its four sides. //

Using methods similar to those used for constructing a component's upright framing rectangle, it can be shown that a component's tilted framing rectangle can be constructed by a CA in region-diameter time. Again, it is then a simple matter for a log-space CA to measure the dimensions of this rectangle to determine the 4-diameter of the component.

F. Convexity

A component S is called convex if every straight line that intersects S, intersects it in exactly one run of points of S. In digital pictures this definition requires some modification since a straight line will not in general pass through digital points.

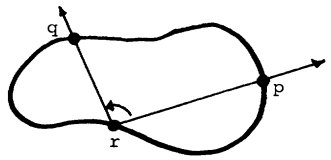


Fig. 5. Detecting convexity by testing that angle $\angle prq$ is not greater than 180° for all triples of border points.

Sklansky *et al.* [14] and Smith [9] have used the notion of a minimum perimeter polygon to recognize convexity, but this approach does not detect shallow concavities.

Another definition of convexity uses the notion of a line of support. A *line of support* of a subset of points S through a point p of S is a line through p such that S lies entirely in one of the closed half-planes bounded by this line. Then, a subset S is called *convex* if there exists a line of support through every border point of S. If S is a component in a digital picture, then we must modify this definition to allow for the discreteness of the data. Therefore, define a *digital line of support* of a component S through a point p of S to be a real line through p such that every border point of S is in or near one of the closed half planes bounded by this line. A digital point (i,j) is *near* the real point (x,y) if $\max(|x-i|, |y-j|) < 1$.

We now describe how a log-space CA M can decide whether or not a component S is convex in the above sense, in perimeter^2 time. M can check in region-diameter time whether or not S is simply-connected using the Beyer/Levialdi algorithm, a necessary condition for S to be convex. Therefore, we will assume that this process occurs in conjunction with, but does not interfere with, the action of M described below which assumes S is simply connected. If S is not simply connected, then M sends a reject signal to the distinguished cell of S.

Assume each cell's coordinates have been previously computed and a distinguished cell in S has been marked. During the first two steps each border point marks itself and determines its successor and predecessor border points. Next, each border point stores three copies of its coordinates in three separate "channels." Beginning at the next step the cyclic ordered list of coordinates in channel 1 shifts clockwise around the border at unit speed. That is, at each step each border point copies the coordinates stored in the first channel of its predecessor. By comparing the contents of its first and third channels at the end of each step, each cell can detect when its coordinates pass by every perimeter-of-S steps. At such times the coordinates in the second channel shift one position counterclockwise around the border. That is, each border point copies the coordinates stored in the second channel of its successor. In this way the coordinates of S's border points shift clockwise around the border at unit speed and simultaneously counterclockwise at 1/perimeter speed. Thus, at each step every point on the border stores the coordinates of three border points, one point p in channel 1, one point q in channel 2, and itself, point r, in channel 3. At each time step, each border point now determines the interior (i.e., counterclockwise) angle between $\vec{rp}$ and $\vec{rq}$, as shown in Fig. 5. If this angle is less than or equal to 180° or the distance from r to the line segment $\overline{pq}$ is less than one, then p and q lie on the inside of the tangent line through r. After perimeter^2 steps every triple of border points has been checked. At this time (when the distinguished cell's three points are all the same) the distinguished cell sends a signal around the border gathering the results of the perimeter^3 tests. If no concavities were detected by any border point, then the distinguished cell enters an accepting state.

TABLE I
SUMMARY OF LOG-SPACE CA COMPUTATION TIMES FOR A VARIETY OF IMAGE PROCESSING TASKS

Task	Time
Region representation	
Region labeling	region-diameter
Run length code	
construction	region-diameter
output	perimeter
Chain code	
construction	constant
output	perimeter
Medial axis transformation	region-diameter
Quadtree construction	region-diameter
Gray level property measurements	
Histogram construction	diameter
Cooccurrence matrix construction	diameter
Moments	diameter
Autocorrelation	diameter · area
Geometrical property measurements	
Area of a region	region-diameter
Perimeter	region-diameter
Compactness	region-diameter
Elongatedness	region-diameter
Height and Width	region-diameter
Diameter	region-diameter
Convexity	perimeter^2

VI. Discussion

In this paper we have investigated how augmenting bounded cellular automata with an amount of memory proportional to the logarithm of the picture size enhances the capabilities of this model and simplifies algorithm design. In addition, this memory-augmented model of parallel computation provides a more natural measure for evaluating the speed advantages of a variety of image processing operations. This in turn leads to a more realistic appraisal of the potential speedups with hardware architectures of this type. Table I summarizes our results, showing that log-space CA's can efficiently perform a variety of basic image processing tasks.

Acknowledgment

The authors gratefully acknowledge the help of K. Riley in preparing this paper.

References

[1] K. Preston, Jr., M. J. B. Duff, S. Levialdi, P. E. Norgren, and J. Toriwaki, "Basics of cellular logic with some applications in medical image processing," *Proc. IEEE*, vol. 67, pp. 826–856, 1979.

[2] B. H. McCormick, "The Illinois pattern recognition computer–ILLIAC III," *IEEE Trans. Comput.*, vol. C-12, pp. 791–813, 1963.

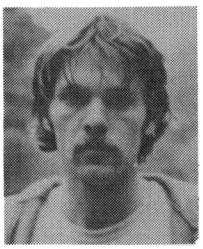
[3] M. J. B. Duff, "CLIP 4: A large scale integrated circuit array parallel processor," in *Proc. Int. Joint Conf. Pattern Recognition*, 1976, pp. 728–733.

[4] L. Fung, "A high-speed image processing computer," in *Proc. 17th Annu. ACM Tech. Symp.*, 1978, pp. 11–17.

[5] A. W. Burks, Ed., *Essays on Cellular Automata.* Urbana, IL: Univ. Illinois Press, 1970.

[6] W. T. Beyer, "Recognition of topological invariants by iterative arrays," Mass. Inst. Technol., Rep. MAC TR-66, Oct. 1969.

[7] J. E. Hopcroft and J. D. Ullman, "Nonerasing stack automata," *J. Comput. Syst. Sci.*, vol. 1, pp. 166-186, 1967.
[8] J. E. Hopcroft and J. D. Ullman, *Formal Languages and Their Relation to Automata.* Reading, MA: Addison-Wesley, 1969.
[9] A. R. Smith, III, "Two-dimensional formal languages and pattern recognition by cellular automata," in *Proc. IEEE 12th SWAT Symp.*, 1971, pp. 144-152.
[10] S. Levialdi, "On shrinking binary picture patterns," *Commun. Ass. Comput. Mach.*, vol. 15, pp. 7-10, 1972.
[11] S. R. Kosaraju, "Fast parallel processing array algorithms for some graph problems," in *Proc. 11th Annu. ACM Symp. Theory of Comput.*, 1979, pp. 231-236.
[12] A. Rosenfeld and A. C. Kak, *Digital Picture Processing.* New York: Academic, 1976.
[13] A. Rosenfeld, *Picture Languages.* New York: Academic, 1979.
[14] J. Sklansky, L. P. Cordella, and S. Levialdi, "Parallel detection of concavities in cellular blobs," *IEEE Trans. Comput.*, vol. C-25, pp. 187-196, 1976.



Charles R. Dyer (S'75-M'79) was born in Gilroy, CA, on March 14, 1951. He received the B.S. degree in mathematical sciences from Stanford University, Stanford, CA, in 1973, the M.S. degree in computer science from the University of California, Los Angeles, in 1974, and the Ph.D. degree in computer science from the University of Maryland, College Park, in 1979.

From 1975 to 1979 he was a member of the Computer Vision Laboratory, Computer Science Center, University of Maryland. Presently, he is an Assistant Professor in the Department of Information Engineering, University of Illinois, Chicago Circle.

Azriel Rosenfeld (M'60-F'72), for a photograph and biography, see this issue, p. 11.

Noah—A Bottom-Up Word Hypothesizer for Large-Vocabulary Speech Understanding Systems

A. RICHARD SMITH AND LEE D. ERMAN

Abstract—**Current high-accuracy speech understanding systems achieve their performance at the cost of highly constrained grammars over relatively small vocabularies. Less-constrained systems will need to compensate for their loss of top-down constraint by improving bottom-up performance. To do this, they will need to eliminate from consideration at each place in the utterance most words in their vocabularies solely on the basis of acoustic information and expected pronunciations of the words. Towards this goal, we present the design and performance of *Noah*, a bottom-up word hypothesizer which is capable of handling large vocabularies—more than 10 000 words. Noah takes (machine) segmented and labeled speech as input and produces word hypotheses.**

The primary concern of this work is the problem of word hypothesizing from large vocabularies. Particular attention has been paid to accuracy, knowledge representation, knowledge acquisition, and flexibility. In this paper we discuss the problem of word hypothesizing, describe how the design of Noah faces these problems, and present the performance of Noah as a function of the vocabulary size.

Index Terms—**Large vocabularies, speech understanding, and hypothesizing.**

Manuscript received November 29, 1978; revised January 2, 1980. This work was supported by the Defense Advanced Research Projects Agency under Grant F44620-73-C-0074 and monitored by the Air Force Office of Scientific Research.

A. R. Smith is with the ITT Defense Communications Division, San Diego, CA 92131.

L. D. Erman is with the Information Sciences Institute, University of Southern California, Marina del Rey, CA 90291.

I. The Problem

THE nature of speech is such that there is no direct mapping from acoustic information to a unique spoken word. The acoustic pattern of a word is embedded within the total pattern of the utterance and modified by it. This is called the *coarticulation* problem. A listener interprets an acoustic event not only by what actually occurs at the event, but also by the surrounding context and even by what he expects to hear. Environmental noise, differences among speakers, differences for the same speaker at different times, and variations in pronunciations also add to the difficulty of finding what words were spoken in an utterance. Another problem is carelessness by the speaker; it seems that a person often speaks just well enough to be understood (most of the time) by another human [16].