COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

2

14 TR-633
AFOSR-77-3271

11 Feb[redacted]978

6 PEBBLE, PUSHDOWN, AND PARALLEL-SEQUENTIAL PICTURE ACCEPTORS.

12 43P.

9 Technical rept.,

10 Azriel Rosenfeld
Computer Science Center
University of Maryland
College Park, MD 20742

18 AFOSR
19 TR-78-Φ423

15 ✓AFOSR-77-3271

16 23Φ4   17 A2

ABSTRACT

This report is a draft of Chapter 7 of a forthcoming book on "Picture Languages." The following chapters have also been issued in report form:

Chapter(s)	Subject	TR
2	Digital geometry	542
3-5	Sequential and parallel acceptors	613
6	Pyramid acceptors	544,596,616
8	Array grammars	629

Comments on the choice and treatment of the material are invited.

The support of the Directorate of Mathematical and Information Sciences, U.S. Air Force Office of Scientific Research, under Grant AFOSR-77-3271, is gratefully acknowledged, as is the help of Judy Myrick in preparing this paper.

403 018

JQB

CHAPTER 7

SPECIAL TYPES OF ACCEPTORS

Chapters 3-6 dealt with Turing and cellular acceptors for strings and arrays. This chapter discusses three specialized acceptor models:

a) Pebble acceptors, which cannot rewrite their input, but can make marks on it, with the restriction that only a bounded number of these marks can exist at any given time.

b) Pushdown acceptors, which cannot rewrite their input, but can store a string of symbols in such a way that only the last symbol in the string can be read (so that to retrieve a given symbol from the string, all the symbols beyond it must first be discarded).

c) Parallel/sequential acceptors, which are one-dimensional cellular acceptors that read one row of a rectangular input array at a time, and move up and down to scan the array.

1. Pebble acceptors

In Chapters 3 and 4 we discussed finite-state string and array acceptors, which can read and move around on their inputs, but cannot rewrite any input symbols. Their inability to write makes these acceptors quite weak. In this section we consider a class of acceptors, called "pebble acceptors", which have limited writing ability. Such an acceptor, A, can make marks on its input, but only a bounded number of these marks can exist at any given time. We can think of the marks as "pebbles" which A puts down in specified positions. If A wants to make a mark m_i that has already been made, it must first find and erase the existing instance of m_i before it can create a new instance. (Informally: if A has already put down pebble m_i, and wants to put it down elsewhere, A must first go to the position of m_i and "pick it up".)

Formally, we can define a k-pebble acceptor $A^{(k)}$ as a tape-bounded acceptor whose state set Q and vocabulary V are of the special forms

$$Q = Q' \times \{0,1\}^k$$

$$V = (V' \times \{0,1\}^k) \cup \{\#\}$$

and whose transition function δ satisfies the following restrictions: If state $(q_1,\alpha_{11},\ldots,\alpha_{1k})$ and symbol $(x_1,\beta_{11},\ldots,\beta_{1k})$ give rise to state $(q_2,\alpha_{21},\ldots,\alpha_{2k})$ and symbol $(x_2,\beta_{21},\ldots,\beta_{2k})$, where each α and β is 0 or 1, then

$x_2 = x_1$, and for each i, $1 \le i \le k$, only the following combinations of $(\alpha_{1i}, \beta_{1i}, \alpha_{2i}, \beta_{2i})$ are possible: (0,0,0,0), (0,1,0,1), (0,1,1,0), (1,0,1,0), and (1,0,0,1). These conditions can be interpreted as follows: the α's indicate which pebbles are currently being "carried" by $A^{(k)}$, while the β's indicate which of them have been put down at the position of the current symbol (1's indicate the presence of pebbles, and 0's indicate their absence). The five permissible combinations have the following interpretations:

$(\alpha_{1i}, \beta_{1i}, \alpha_{2i}, \beta_{2i})$	Meaning
(0,0,0,0)	$A^{(k)}$ was not carrying the ith pebble, and it was not at the current position; hence after the transition, $A^{(k)}$ cannot have picked it up or put it down.
(0,1,0,1)	The ith pebble was at the current position, but $A^{(k)}$ did not pick it up.
(0,1,1,0)	The ith pebble was at the current position, and $A^{(k)}$ did pick it up.
(1,0,1,0)	$A^{(k)}$ was carrying the ith pebble, and did not put it down.
(1,0,0,1)	$A^{(k)}$ was carrying the ith pebble, and did put it down.

We assume that the initial state is $(q_0, 1, \ldots, 1)$, meaning that $A^{(k)}$ is initially carrying all the pebbles. Readily, the above conditions guarantee that, in any given configuration, for each i, either $\alpha_i = 1$ or there exists exactly one position at which $\beta_i = 1$. The requirement that $x_2 = x_1$ means that $A^{(k)}$ cannot rewrite symbols except in the sense of picking up and putting down pebbles.

Pebble acceptors on strings and arrays have been studied by Blum et al. [1-2], Mylopoulos [3], and Shah [4]; the material in this section is based on these references.

1.1 Pebble string acceptors

A one-dimensional acceptor having a single pebble (k=1) is no stronger than a finite-state acceptor. This can be proved (see [1], Theorem 7) by an argument analogous, in part, to that used to show that in one dimension, one-way acceptors are as strong as two-way acceptors (see Section 3.3 of Chapter 3). The details will not be given here.

A two-pebble string acceptor, on the other hand, is stronger than an FSA. For example, a two-pebble acceptor $A^{(2)}$ can accept the set of strings $\{a^m b^m | m=1,2,\ldots\}$ by operating as follows: $A^{(2)}$ initially puts its pebbles at opposite ends of its input string σ, and verifies that the symbols at these ends are a and b, respectively. $A^{(2)}$ then moves repeatedly back and forth, and each time it reaches a pebble, it "pushes" that pebble one position closer to the center of σ (i.e., it picks up the pebble, moves one position closer, and puts the pebble down again), also verifying that the symbols in these positions are still a and b, respectively. If these conditions remain true until the pebbles reach adjacent positions (just on opposite sides of the center of σ), then σ must be of the desired form, and $A^{(2)}$ can accept it.

For any k, it can be shown that there exists an $\ell > k$ such that ℓ-pebble acceptors are strictly stronger than k-pebble acceptors (see, e.g., [1] and [5]).

It should be mentioned that, as an alternative to pebble acceptors, one can consider "multihead" acceptors which have a bounded number of independently movable, read-only heads. Readily, a k-pebble acceptor can simulate a k-head acceptor, by using the pebbles to mark the heads' positions, and internally representing their states; while a (k+1)-head acceptor can simulate a k-pebble acceptor by allowing all but one of its heads to behave like passive pebbles. Multihead acceptors will not be treated further here; see, e.g., [5-6].

1.2 Pebble rectangular-array acceptors

In two dimensions, a one-pebble acceptor is stronger than an FSA. For example, a deterministic, tape-bounded one-pebble FSA $A^{(1)}$ can determine whether the center symbol in a square, odd-side-length array Σ of 0's and 1's is 0 or 1, which cannot be done without the pebble (Theorem 3.4 of Chapter 4). To do this, $A^{(1)}$ starts at the upper left corner of Σ and moves down the main diagonal (rightward and downward). At each point, it puts down its pebble, memorizes the symbol at that position, and moves rightward and upward. If it reaches the upper right corner, and the memorized symbol is 1, it accepts. If it does not reach the upper right corner, it moves back (leftward and downward) to the main diagonal (which it can detect by the presence of the pebble), and resumes moving down the main diagonal.

Theorem 1.1. A deterministic, tape-bounded one-pebble acceptor can determine whether or not the set of 1's in a rectangular array Σ of 0's and 1's is connected.

Proof (see [1], Theorem 4): The acceptor A first checks whether or not the rows of Σ that contain 1's form a single run of consecutive rows. This does not require the use of a pebble; A need only scan Σ row by row, and determine whether there exists a run of rows that do contain 1's, followed by a run of rows that do not, followed by a row that does. If so, the 1's in Σ are evidently disconnected. If the rows with 1's do form a single run, A proceeds to the next step, which is to check whether the 1's in each row are all connected.

To check this for a given row, A scans the row from left to right. In general the row consists of runs of 1's separated by runs of 0's. It evidently suffices to check that any two consecutive runs of 1's are connected. Let P and Q be the last point of a run and the first point of the next run, respectively. If P and Q belong to the same connected component C of 1's, they must lie on the same border of C, since the same component of 0's is adjacent to both of them. Thus when A reaches P, it puts down the pebble and follows the border B defined by P and the 0 immediately succeeding it. At each move around B, if A is at a point Q' that has 0's on its left, it scans leftward until it hits a 1 (or reaches the border of Σ). If this 1 has the pebble on it, then Q' is Q, so that Q is connected to P, and A can pick up the pebble and go on to check the next run end pair on that row. Otherwise, A returns to Q' and resumes border following. If A succeeds in following the border completely around until it reaches the pebble at P again, it knows that Q is not connected to P, so that the 1's in Σ are not connected.

Suppose that A has found that the rows containing 1's form a single run, and that the 1's in each row are all connected. It then remains only to show that the 1's in each two consecutive rows r_1, r_2 are connected to each other.

If a run of 1's in r_1 is adjacent to a run of 1's in r_2, this is immediate; hence we may assume that these runs are interleaved. In particular, there exists a run ρ_1 of 1's in r_1, followed (or preceded) by a run of 0's which is adjacent to a run ρ_2 of 1's in r_2. Let P be the last point of ρ_1 and Q the first point of ρ_2; then just as in the preceding paragraph, if P and Q are connected, they must lie on the same border. A can thus put the pebble at P and follow this border; at each point Q' that has 0's on its left, it scans leftward through that run of 0's and checks whether any of them has the pebble above it. If so, Q' is Q, and A has verified that the 1's in r_1 are connected to those in r_2. If not, A returns to Q' and resumes border following. If it gets back to P without finding Q, it has verified that the 1's in r_1 are not connected to those in r_2, so that the 1's in Σ are not connected. If A succeeds in verifying that the 1's in each pair of consecutive rows are connected, it has confirmed that all the 1's in Σ are connected, and can accept Σ.//

It is not known whether an FSA can determine connectedness of the 1's in its input array.

Corollary 1.2. A one-pebble acceptor can determine whether or not the set of 1's in Σ is simply-connected.

Proof: A checks that the 1's are connected, and analogously checks whether the 0's are connected. (If the border of Σ

does not consist entirely of 0's, the latter check is carried out as if there were an additional row or column of 0's at each edge of Σ.) If the 0's are connected, the 1's have no holes.//

It is shown in [1] (Theorem 5) that a one-pebble acceptor cannot determine whether or not two simply-connected components of 1's C_1 and C_2 are congruent (i.e., differ only by a translation). On the other hand, a two-pebble acceptor can determine congruence even for two arbitrary components of 1's ([1], Theorem 6). [The proofs of these results will not be given here.] Hence two-pebble acceptors on rectangular arrays are strictly stronger than one-pebble acceptors. This also follows immediately from the one-dimensional result, if we consider one-row rectangular arrays.

1.3 Pebble connected-array acceptors

We saw in Section 4.2 of Chapter 4 that no FSA, A, can accept just those connected input arrays that fail to contain a given symbol; or, equivalently, that no such A can accept its input array Σ only after A has visited all of Σ. Our principal goal in this section is to show that an FSA $A^{(4)}$ with four pebbles can do these things. All acceptors in this section are deterministic and tape-bounded.

Proposition 1.3. A three-pebble acceptor $A^{(3)}$ can find the outer border of its input array Σ.

Proof: $A^{(3)}$ moves upward whenever possible. If it hits a #, say just above P, it drops the first pebble (α_1) at P and moves along the border B defined by P and the # above it, using α_2 to mark its current position, and moving α_3 back and forth along B to keep count of its net number n of upward or downward moves from P. If $n > 0$, and there is a non-# above $A^{(3)}$, it picks up the pebbles and moves upward again. If this never happens until $A^{(3)}$ returns to P, there is no point of B higher up than P. This implies that P is on the outer border, since if P were a hole border point with #s above it, there would have to exist a point higher than P on the same hole border.//

(This is essentially the same as the first part of the proof of Theorem 4.5 in Chapter 4. Compare the proof in [4], Lemma 3.1.1.)

Theorem 1.4 ([4], Theorem 6). A three-pebble acceptor $A^{(3)}$ can find the "upper left corner" (=leftmost of the uppermost points) of its input array Σ.

Proof: $A^{(3)}$ first finds the outer border of Σ as in Proposition 1.3. In the process of doing this, it finds an uppermost point of this border (whenever $n > 0$, take the current position as the new P). This point belongs to a run of uppermost point of Σ; let Q be the leftmost point of this run. $A^{(3)}$ now marks Q with α_1 and follows the border again, starting out leftward, and using α_2 to mark its current position and α_3 to keep count of its net number n of downward moves. Let Q' be the position of $A^{(3)}$ the first time that n=0. If Q'=Q, this is the only uppermost point of Σ, and we are done. Otherwise, if Q is not the upper left corner of Σ, it is easily seen that Q' must belong to a run of uppermost points of Σ that lies to the left of Q. To check this, $A^{(3)}$ marks Q' with α_2, returns to Q, picking up α_3 and α_1, and follows the border again from Q to Q', using the relative positions of α_1 and α_3 to determine its net number m of leftward and rightward moves. [This is done as follows: Each time α_1 is moved one step along the border, α_2 is moved

two steps, if α_1 made a rightward move

one step, if α_1 made neither a rightward nor a leftware move

no steps, if α_1 made a leftward move.

Thus the number of steps that α_3 is ahead of α_1 along the border is the number m of rightward moves made by α_1.] When α_1 reaches Q', if $m < 0$, Q' is to the left of Q. $A^{(3)}$ can then pick up the pebbles, move to the left end of the run of uppermost points containing Q', take this point as the new Q, and repeat the process. Eventually it must turn out that Q' is not to the left of Q, which means that Q was the upper left corner. In that case $A^{(3)}$ follows the border back from Q', using the pebbles to keep track of its net number n of downward moves; the first time that $n=0$, $A^{(3)}$ has returned to Q, which is the upper left corner.//
(This proof uses fewer marks than the last part of the proof of Theorem 4.5 in Chapter 4.)

Theorem 1.5 ([2]; compare [4], Theorem 7): A four-pebble acceptor $A^{(4)}$ can systematically scan its input array Σ, row by row.

Proof: We can assume by Theorem 1.4 that $A^{(4)}$ starts at a point on the uppermost row of Σ. We shall show below that if $A^{(4)}$ is on any given row r of Σ it can visit all the points of Σ on r. After having done so, $A^{(4)}$ can find a point of Σ on r that has a point of Σ below it, and thus move to the row below r and repeat the process; or $A^{(4)}$ can determine that no such point on r exists, in which case r is the lowest row of Σ and the scan is complete.

We shall now show that, starting at any $P \in \Sigma \cap r$, $A^{(4)}$ can visit the part of $\Sigma \cap r$ to the right of P (say). To do this,

$A^{(4)}$ moves rightward on r until it hits a #, say just to the right of Q. It then follows the border B defined by Q and this #, using pebble α_1 to mark its current position; the positions of pebbles α_2 and α_3 relative to α_1 to keep count of its net horizontal and vertical displacements m and n relative to Q; and the position of pebble α_4 relative to α_1 to keep track of the minimum positive value of m at those steps when n=0. (Q itself is not marked.)

The shifting of α_2 and α_3 along B is as described in the proof of Theorem 1.4. The first time α_3 reaches the same point of B as α_1 (i.e., n=0), if α_2 is also at that point (m=0), $A^{(4)}$ has returned to Q, which must thus be the sole point of r, so that the scan of r is complete. If $m < 0$, $A^{(4)}$ is to the left of Q, and we continue to move α_1 around b. If $m > 0$, we drop α_4 at the position of α_2, since this is the minimum positive value of m so far found, and during border following we maintain α_4 at a constant distance from α_1. At subsequent times when n=0,

a) If m=0, we are back at Q, so that α_4's distance from α_1 is the desired minimum positive m. We then follow B again, using α_1, α_2 and α_3 as before, but now keeping α_4 at a fixed distance from α_1. When we reach the point Q' at which n=0 and α_2 is at the same position as α_4, we know that we are at the first point of $\Sigma \cap r$ to the right of Q, and we can now resume the rightward scan.

b) If $m < 0$, we continue following B.

c) If $m > 0$, and α_2 is farther from α_1 than α_4 is, we continue following B, since the current m is not the minimum.

d) If $m > 0$, and α_4 is farther from α_1 than α_2 is, we move α_4 to the current position of α_2, since its old position was not the minimum; we then resume following B.

It is evident that this process is guaranteed to find the first point of $\Sigma \cap r$ to the right of Q. Thus by using this process repeatedly, we can scan all of $\Sigma \cap r$ to the right of P. If α_4 is never dropped, there are no points of $\Sigma \cap r$ to the right of Q, so that the rightward scan of $\Sigma \cap r$ is complete.//

<u>Corollary 1.6</u>. There exists a four-pebble acceptor that accepts Σ iff. Σ does not contain (or contains) a specified symbol.//

If Σ is known to be simply-connected, an FSA can scan Σ by operating as follows: A moves until it reaches a border B, which must be the outer border of Σ; it then follows B, and at each point P, moves rightward (if possible) until it hits a #, then back again until it hits a # (which must put it at P), then resumes following B. Since from every point of Σ, if we move left through non-#s, we must hit B, this process must visit every point of Σ. Note that A cannot know when the scan is complete, since it cannot know when it has finished following B. However, if we give A one pebble α, it can use α to mark its starting point on B, and accept when it reaches α again ([4], Theorem 1). These remarks imply

<u>Proposition 1.7</u>. If Σ is simply-connected, there exists an FSA that accepts Σ iff. it contains a specified symbol.

<u>Proposition 1.8</u>. If Σ is simply-connected, there exists a one-pebble acceptor that accepts Σ iff. it fails to contain a specified symbol.

<u>Theorem 1.9</u> ([4], Theorem 3). There exists a one-pebble acceptor $A^{(1)}$ that accepts Σ iff. it is not simply-connected.

<u>Proof</u>: It is easily seen that Σ is multiply connected iff., on each border B of Σ, there exists at least one point P, say at the end of a horizontal run ρ of non-#s, such that the other end Q of ρ is on a border of Σ other than B. Based on this fact, $A^{(1)}$ operates as follows: It moves (say) rightward until it hits a border B of Σ, say at P, and puts down α at P. $A^{(1)}$ now moves to the left end, P', of the horizontal run ρ of non-#s containing P; this P' is on some border B' of Σ (possibly the same as B). $A^{(1)}$ follows B'; at each step:

a) If $A^{(1)}$ finds α, then P and P' are on the same border (i.e., B=B'). In this case no conclusion can be reached as yet about the simple-connectedness of Σ. $A^{(1)}$ picks up α, moves along B, and repeats the entire process. [Note that if Σ is simply-connected, it will always be the case that B=B', so that $A^{(1)}$ will repeat the process indefinitely without ever accepting.] If $A^{(1)}$ is at a right end of a run of non-#s and does not find α, it continues following B'.

b) If $A^{(1)}$ is at the left end of a run of non-#s, it moves to the right end of the run. If it does not find α there, it moves back to the left end and continues following B'. If it finds α there, it has followed B' completely around and returned to P'. If this happens, $A^{(1)}$ cannot have found α while following B' (i.e., case (a) could not have occurred), so B' must be different from B. Thus $A^{(1)}$ now knows that Σ is not simply-connected, and can accept.

By the assertion in the first sentence of the proof, $A^{(1)}$ accepts iff. Σ is not simply connected.//

Theorem 1.10 ([4], Theorem 4). There exists a two-pebble acceptor $A^{(2)}$ that accepts Σ iff. it is simply-connected.

Proof: $A^{(2)}$ operates as in the proof of Theorem 1.9, except that it marks its starting point P on B with its other pebble β. If case (b) never occurs, $A^{(2)}$ will follow B completely around, verifying at each stage (where applicable) that B' is the same as B, until it finds β again. When this happens, $A^{(2)}$ knows that Σ is simply-connected, and can accept. If case (b) occurs, $A^{(2)}$ halts in a non-accepting state.//

By Theorem 4.3 of Chapter 4, there is no FSA that accepts just those Σ's that are not simply-connected. Indeed, such an FSA would accept, in particular, all the hollow rectangles; but as we have seen, any (tape-bounded) FSA that accepts these rectangles also accepts some sufficiently large rectangular spiral, which <u>is</u> simply-connected.

2. Pushdown acceptors

One-dimensional pushdown acceptors (PDA's) have been widely studied because of their well-known relationship to context-free grammars ([7], Chapter 5 and 12). Two-dimensional PDA's have not been widely studied; our main result about them is taken from [8] (Theorem 7.1).

A pushdown acceptor can be regarded as an "FSA" A with an infinite state set of the form $Q = Q' \times U^*$, where U is a finite set called the stack vocabulary, and U^* denotes the set of all strings (including the null string) composed of symbols in U. The transition function δ of A must satisfy the following conditions: if $(q_1, x_1 \ldots x_m)$ gives rise to $(q_2, y_1 \ldots y_n)$ under δ, where q_1, q_2 are in Q' and $x_1 \ldots x_m$, $y_1 \ldots y_n$ are in U^*, then one of (a-c) must hold:

a) $n=m+1$ and $y_1 \ldots y_{n-1} = x_1 \ldots x_m$. In this case we say that A has pushed symbol y_n onto the top of its stack.

b) $n=m$ and $y_1 \ldots y_n = x_1 \ldots x_m$. In this case the stack is unaffected by the transition.

c) $n=m-1$ and $y_1 \ldots y_n = x_1 \ldots x_{m-1}$. In this case A is said to have popped symbol x_m from the top of its stack.

Moreover, q_2 (and y_n, in case (a)), and the move made by A, are allowed to depend only on q_1, x_m (the topmost stack symbol), and the current input symbol (as well as on A's previous move); they cannot depend on $x_1,\ldots,x_{m-1}$. (A can also tell whether its stack is empty.) In other words, in making transitions, A has no access to the information contained below the top of its stack. [In the state $(q_1,x_1\ldots x_m)$, q_1 is called the internal state of A, and $x_1\ldots x_m$ constitutes A's stack contents.] In all of the above, A can be either one-way or two-way.

2.1 String PDA's

To illustrate the operation of PDA's in one dimension, we give a few simple examples of pushdown languages:

1) The set of strings $\{a^m b^m | m=1,2,...\}$ is accepted by a one-way deterministic PDA A that operates as follows: Starting at the left end of its input string σ, A moves rightward. When A reads an "a", it pushes a symbol u onto the top of its stack, so that after reading k a's, its stack contains the string u^k. When A reads a "b", it pops a u from the top of its stack. Unless A reads a succession of a's followed by a succession of b's, it halts in a non-accepting state. If it reaches the right end of σ without halting, and its stack has just then become empty, it knows that the number of b's is equal to the number of a's, and accepts σ. This example shows that one-way deterministic PDA's are strictly stronger than FSA's.

2) The set of symmetric strings ("palindromes") $\sigma=\omega\omega^R$, where ω is any string on {a,b} (say), and ω^R denotes the reversal of ω, is accepted by a one-way nondeterministic PDA A that operates as follows: Starting at the left end of σ, A moves rightward; when it reads an a or b, it pushes an a or b on top of its stack. At some point, nondeterministically chosen, A stops pushing symbols onto its stack. Instead, at each rightward

move, it pops the symbol from the top of its stack and compares it with the symbol of σ in its current position. If they differ, A halts in a non-accepting state. If A reaches the right end of σ without halting, and its stack has just then become empty, we know that the point at which it stopped pushing and started popping was the midpoint of σ, and that σ is symmetric. It can be shown ([7], p. 187, Problem 12.6) that this set of strings is not accepted by any one-way deterministic PDA; hence for one-way PDA's, nondeterministic are strictly stronger than deterministic. It is well-known ([7], pp. 74-78) that the one-way nondeterministic PDA's accept exactly the context-free languages (see Section 1.2 of Chapter 8).

3) Two-way deterministic PDA's can accept languages that are not context-free, and so are stronger than one-way nondeterministic PDA's. For example, consider the set of strings $\{a^m b^m c^m | m=1,2,\ldots\}$, which is not context-free ([7], p. 66, Problem 4.15). To accept this set, A first verifies that the numbers of a's and b's are equal, as in (1); it then moves back to the beginning of the b's, and verifies in the same way that the numbers of b's and c's are equal.

A two-way PDA can find the midpoint of its input string σ, e.g., by moving rightward from the left end of σ and pushing u's onto its stack at every second move, then moving leftward and popping u's from the stack at every move; when the stack is empty, the PDA is at the midpoint of σ. As an application of this, we show how a two-way PDA, A, can accept the set of repeated strings $\sigma=\omega\omega$, where ω is any string of a's and b's (say). A first finds the midpoint of σ, then moves leftward, pushing the symbols that it reads onto its stack; when it reaches the left end, its stack contains the first half of σ, with the first symbol on top. A then finds the midpoint again and moves rightward, popping the a's and b's from its stack and comparing them with the succession of symbols in the right half of σ. If a difference is found at any point, A halts and does not accept; if A reaches the right end of σ without halting, it accepts. An analogous construction can be used by a two-way deterministic PDA to recognize palindromes.

2.2 Array PDA's

The ability of PDA's to find midpoints, check symmetry and periodicity, etc. can be used in two dimensions to define PDA acceptors for a variety of rectangular array languages; the details are straightforward. Our main result in this section deals with deterministic PDA's on simply-connected arrays. [Array PDA's will always be assumed to be "4-way", i.e., able to move in any direction.] Specifically, we shall prove

Theorem 2.1. There exists a PDA that accepts any simply-connected input array Σ after it has scanned all of Σ.

Corollary 2.2. *If Σ is simply-connected, there exists a PDA* that accepts Σ iff. it fails to contain a specified symbol. (Compare Propositions 1.7-8.)

The proof of Theorem 2.1 is by induction on the number of points in Σ; it makes use of the fact that any simply-connected Σ contains simple points (see Proposition 5.4 of Chapter 2), and that the operation of the PDA remains valid when a simple point is added or deleted. The details of the proof will not be given here; they can be found in [8]. In the remainder of this section we present the definition of the desired PDA, and illustrate its operation by example.

A has six stack symbols, d, e, u, v, ℓ, and r, which are interpreted as follows:

Symbol	Interpretation
d	downward concavity detected while scanning a run of non-#s
e	downward concavity detected while following the border of Σ
u	upward concavity detected while scanning a run of non-#s
v	upward concavity detected while following the border of Σ
ℓ	left end of a run visited
r	right end of a run visited

A operates by following the border of Σ counterclockwise, starting at a left run end with an initially empty stack. Suppose A is at the left end of run ρ. It first scans the points of ρ from left to right and examines their upper neighbors. Every change of upper neighbors from non-# to # to non-# indicates that an upward concavity has been detected. Each time this happens, if the top symbol on A's stack is v, A pops the v; otherwise, A pushes a u. A then returns to the left and of ρ, and if there is an r on top of its stack it pops the r; otherwise, it pushes an ℓ. Next, A scans ρ from left to right again, examining lower neighbors. For each change from non-# to # to non-#, it pops an e from the top of its stack, or if there is none, it pushes a d. A then resumes following the border of Σ.

Whenever it reaches the right end of a run of non-#s, it pops an ℓ from the top of its stack, or if there is none, it pushes an r. Whenever it visits an upward concavity (moving downward from a left run end, then back upward to a right run end), it pops a u, or if there is none, pushes a v; and whenever it visits a downward concavity (moving upward from a right run end, then back downward to a left run end), it pops a d, or if there is none, pushes an e.

It can be shown that A's stack becomes empty again iff. Σ is simply-connected and A has returned to its starting point. On the other hand, if Σ is not simply-connected, the stack does not become empty when the border has been completely followed. These situations are illustrated by the following simple examples:

<u>Example 1</u>. Let Σ be

```
1→XXXX←6
2→X  X←5
3→XXXX←4
```

Point(s)	Current stack	Event
1	empty	ℓ
1	ℓ	d
2	ℓd	ℓ
3	ℓdℓ	u
3	ℓdℓu	ℓ
4	ℓdℓuℓ	r
5	ℓdℓu	r
6	ℓdℓur	r
1	ℓdℓurr	ℓ

In this case the stack is not empty when A returns to point 1. As A follows the border repeatedly, the string in the stack continues to grow and A never accepts.

<u>Example 2</u>. Let Σ be

```
                              1→XX←20
8→XXXXXXX←7                   2→XX←19
9→X←10 11→X←6 5→X←4           3→XX←18
       12→XXXXXXXXXXXXXXX←17
       13→XX←14              15→XX←16
```

The following table indicates the events that occur at successive points of the border of Σ (starting from point 1), or between pairs of border points.

Point(s)	Current stack	Event
1	empty	ℓ
2	ℓ	ℓ
3	ℓℓ	ℓ
(3,4)	ℓℓℓ	v
4	ℓℓℓv	r
5	ℓℓℓvr	ℓ
(5,6)	ℓℓℓv	v
6	ℓℓℓvv	r
7	ℓℓℓvvr	r
8	ℓℓℓvvrr	ℓ
8	ℓℓℓvvr	d
9	ℓℓℓvvrd	ℓ
10	ℓℓℓvvrdℓ	r
(10,11)	ℓℓℓvvrd	e
11	ℓℓℓvvr	ℓ
12	ℓℓℓvv	uu
12	ℓℓℓ	ℓ
12	ℓℓℓℓ	d
13	ℓℓℓℓd	ℓ
14	ℓℓℓℓdℓ	r
(14,15)	ℓℓℓℓd	e
15	ℓℓℓℓ	ℓ
16	ℓℓℓℓℓ	r
17	ℓℓℓℓ	r
18	ℓℓℓ	r
19	ℓℓ	r
20	ℓ	r
1	empty	ℓ

3. Parallel-sequential acceptors

This section describes a special type of rectangular array acceptor that represents a compromise between tape-bounded (sequential) and bounded cellular (parallel) acceptors. The former require large amounts of time (order (array area)) for acceptance, while the latter require large amounts of hardware (order (array area) cells). The proposed "parallel/sequential" acceptor is a one-dimensional cellular acceptor that reads one row of its rectangular input array at a time, and moves up and down to scan the array. It thus uses a greatly reduced amount of hardware (order (array width) cells),but is still able to perform some tasks quickly (in order (array diamater) time).

Acceptors of this type were first introduced in [9]; however, they violated several of the finiteness restrictions that are usually imposed on acceptors. In particular, each cell accepted inputs from all of the others, rather than from a neighborhood of bounded size. Moreover, acceptance was defined using a counter that summed the cell outputs on each row, and that could count modulo M, where M could grow with the input size. A more conventional definition of this type of acceptor was used in [10], on which most of the material in this section is based.

Formally, a parallel/sequential acceptor (for brevity: PSA) is a 9-tuple $A=(Q,q_0,Q_A,\#,V,\#_t,\#_b,\delta,\mu)$, where

Q is a finite, nonempty set of states

$q_0 \in Q$ is the initial state

$Q_A \subseteq Q$ is the set of accepting states

$\# \in Q$ is the blank state

V is a finite, nonempty set of symbols called the tape vocabulary

$\#_t$ and $\#_b$ are blank symbols in V

δ: $Q \times Q \times Q \times V \rightarrow 2^{Q \times V}$ is the state transition function

μ: $Q \times V \rightarrow 2^{\{-1,0,1\}}$ is the move function

The operation of A on a rectangular array Σ can be described as follows: A consists of a string of cells $A_1,\ldots,A_n$ whose length is equal to the width of Σ, together with two special "cells" A_0 and A_{n+1} that are regarded as permanently in the # state. Σ has a row of $\#_t$'s just above its top row and a row of $\#_b$'s just below its bottom row. Initially, A is on the top row of Σ with every cell in state q_0. At any given step, each cell A_i reads the symbol v in its position, senses the states q_1, q_2 of its neighbors $A_{i\pm 1}$, and can go into any new state q' and write any new symbol v' such that $(q',v') \in \delta(q,q_1,q_2,v)$, where q was the current state of A_i. The move function depends only on the (new) state of and symbol read by the distinguished cell A_1 (which is the only cell having a # on its left): $0 \in \mu$ means that A can stay where it is; $1 \in \mu$ means that A can move down; $-1 \in \mu$ means it can move up. It is required that $\mu(q,\#_t)=1$ and $\mu(q,\#_b)=-1$

for all q; in other words, A, bounces downward from $\#_t$ and upward from $\#_b$. (It is understood that $\#_t$ and $\#_b$ can never be rewritten: $(q',v') \in \delta(q,q_1,q_2,\#_{t \text{ or } b})$ implies $v'=\#_{t \text{ or } b}$, respectively.) If A_1 ever enters a state in q_A, we say that A has accepted Σ. [A formal description of the operation of A on Σ can be given by introducing the concept of a configuration; the details will not be given here.] The set of arrays accepted by A is called the language of A, and is denoted by L(A).

To illustrate acceptance of an array by a PSA, consider the set of arrays that contain a specific symbol x. An A that accepts this set is defined as follows: A starts on the top row of Σ and moves downward. If any cell A_i sees an x, it goes into state $q_A \in Q_A$. If any cell has its right neighbor in state q_A, if goes into state q_A. Evidently, if Σ contains an x (and only then), cell A_1 goes into state q_A after at most h+w time steps, where h and w are the height and width of Σ (h steps are needed for A to scan Σ from top to bottom, and at most w steps for the q_A signal to reach A_1).

<u>Theorem 3.1</u>. The set of languages accepted by PSA's is the same as that accepted by BCA's.

<u>Proof</u>: A BCA, C, can simulate a PSA, A, as follows: Initially, the cells of C are in the array of states Σ. At the first step, the top row of cells go into the states (q_0,v_i), where the v_i's were their initial states. The top row then

simulates the first transition of A; each cell goes into a state of the form (q,v,0). If this transition causes A to move, the leftmost cell of the row initiates a synchronization process (see Section 4.1 of Chapter 3) which causes all the cells to simultaneously change the third terms of their states to ±1 (+ for a downward move, -1 for an upward). At the next step, each such cell goes into state v, while the cell below it (for +1, or above it for -1) goes into state (q,u) (where u was its previous state). The next transition of A can now be simulated. If the leftmost cell of the "active" row (having pairs or triples as states) has first term in q_A, an acceptance signal is sent up the left column to the upper left cell of C, which then accepts. Note that this simulation requires on the order of w times real time (where w is the width of Σ), since O(w) steps are required for the synchronizations between simulations of transitions of A. [The simulation can also be done in O(w) times real time by a TBA, using Σ to record the states of A's cells.]

Conversely, a PSA, A, can simulate a BCA, C, by recording the states of C's cells on Σ. To simulate a transition of C, A scans Σ from top to bottom. On each row, A remembers the states recorded on the two previous rows, computes the new states of the cells on the row above (compare the proof of Theorem 4.5 in Chapter 3), and records them on that row. [The next transition is simulated using a scan from bottom to top, which shifts the recorded new states back to their proper positions.] When computing the new state of the upper left cell of C, if A_1 finds that C has accepted, it accepts.// Note that in Theorem 3.1, if A is deterministic, so is C, and vice versa.

In the remainder of this section we consider two restrictions on PSA's: not allowing them to rewrite the symbols in Σ, and not allowing them to move upward. We use the prefixes D, N, and O to denote "deterministic", "non-writing", and "one-way" PSA's, respectively. The set of languages accepted by (D)(N)(O)PSA's will be denoted by $L_{(D)(N)(O)PS}$. Thus Theorem 3.1 can be restated as: $L_{(D)PS}=L_{(D)BC}$.

We first show that non-writing PSA's are strictly stronger than FSA's, but strictly weaker than BCS's.

<u>Theorem 3.2</u>. $L_{(D)FS} \underset{\neq}{\subset} L_{(D)NPS} \underset{\neq}{\subset} L_{(D)BC}$

<u>Proof</u>: An NPSA, A, can simulate an FSA by having one of its cells, in a distinguished state, designate the position and state of the FSA. When the FSA moves right or left, the distinguished state is passed to the right or left by A; when the FSA moves up or down, A moves up or down (note that the special cell must send a signal to A_1 in order for A to do this). When the FSA accepts, the special cell sends a signal to A, which accepts. Thus NPSA's can accept all the FSA languages. Conversely, on a one-row array, an NPSA can evidently simulate a one-dimensional BCA, and so can accept

non-FS languages; thus FSA's cannot accept all the NPSA languages. These results remain valid if the acceptors in question are assumed to be deterministic.

A BCA can simulate an NPSA, as in the first part of the proof of Theorem 3.1; thus BCA's can accept all the NPSA languages. Conversely, on a one-column array, an NPSA can evidently be simulated by a one-dimensional FSA; thus NPSA's cannot accept all the (one-dimensional) BC languages. These results too remain valid for deterministic acceptors.//

For one-way PSA's, the ability to write provides no advantage (compare Section 3.3 of Chapter 3 on one-way TA's and TBA's). In the following two theorems we establish some proper inclusion relations on the classes of languages accepted by (deterministic) non-writing one-way PSA's (we omit the "N" prefix):

Theorem 3.3. $\mathcal{L}_{(D)OPS} \underset{\neq}{\subset} \mathcal{L}_{(D)PS}$.

Proof: Let L be the set of 2n(rows) by n(columns) arrays of a's and b's whose top and bottom halves are identical. [The fact that an array is twice as high as it is wide can be verified by a DOPSA, A, e.g., by moving a marker one step to the right whenever A moves downward, until it reaches A_n, then moving it one step to the left per downward move of A, and verifying that the marker returns to A_1 just as A reaches the bottom row.] To see that $L \in \mathcal{L}_{DPS}$, note that A can check that Σ has the defining property of L one column at a time by moving halfway down Σ (when the marker reaches A_n), storing the top half of the given column in its row of cells; A then moves the rest of the way down, comparing the stored top half with the bottom half of the same column. This can be done repeatedly until all columns have been checked.

We now show that $L \notin \mathcal{L}_{OPS}$. Suppose A were an OPSA that accepts L. For any top half Σ, let Q_Σ be the set of n-tuples of states that A could be in after scanning Σ, and that can lead to acceptance after the bottom half (Σ again) is scanned. Then if $\Sigma \neq \Sigma'$, we must have $Q_\Sigma \cap Q_{\Sigma'} = \emptyset$ -- for, if an n-tuple σ were in both, there would exist some bottom half Σ'' for which the initial n-tuple σ can lead to acceptance, and Σ'' must be the same as both Σ and Σ', contradiction. Thus each of the 2^{n^2} top halves must give rise to a nonempty set Q_Σ, and these Q's are pairwise disjoint. But the number of pairwise disjoint nonempty subsets of the $|Q|^n$ possible n-tuples of states cannot exceed $|Q|^n$, which is less than 2^{n^2}, contradiction.//

<u>Theorem 3.4</u>. $\mathcal{L}_{DOPS} \subsetneq \mathcal{L}_{OPS}$.

<u>Proof</u>: Let L' be the set of 2n by n arrays of a's and b's in which some row occurs in both the top and bottom halves. A nondeterministic OPS can accept L' by memorizing an arbitrary row in the top half and comparing it with an arbitrary row in the bottom half, and accepting if they are

equal. [It can tell the top half from the bottom half, and can check that Σ is 2n by n, by moving a marker as in the proof of Theorem 3.3.]

Suppose that L' were accepted by a DOPSA, A. When it leaves the top half of its input array, A must be in one of the $|Q|^n$ possible state combinations. Now the number possible sets of rows that can occur in the top half is $\binom{2^n}{1}+\binom{2^n}{2}+\ldots+\binom{2^n}{n}$ (depending on how many of these rows are distinct). For large n, this number is greater the $|Q|^n$, so that two of the sets, S and T, must yield the same state combination σ. Let ρ be a row in S but not in T (say), and let the bottom half of the array consist entirely of ρ's. When A scans n ρ's, starting in state combination σ, it either accepts or does not accept. In the former case, A accepts top half T (not containing ρ) followed by n ρ's, which is incorrect; in the latter case, A does not accept top half S (containing ρ) followed by n ρ's, which is incorrect.//

It is an open question whether $L_{DPS} \underset{\neq}{\subset} L_{PS}$. The relationships between $L_{(D)OPS}$ and $L_{(D)FS}$ are also open. We conclude this section by showing that a DOPSA can recognize connectedness (compare Theorem 1.1), which is an open question for FSA's.

<u>Theorem 3.5</u>. A DOPSA can determine whether or not the set of 1's in a rectangular array Σ of 0's and 1's is connected.

Proof: Let r_k denote the kth row, and R_k the top k rows, of Σ. We shall describe how a DOPSA, A, when it is on r_k, can construct a string of parentheses that indicates how the runs of 1's in r_k are connected in R_k. Specifically, for each component C of 1's in R_k, we put a left parenthesis at the leftmost point (if any) in which C meets r_k, and a right parenthesis at the rightmost point. (If C meets r_k in only one point, we put a pair of parentheses at that point.) This parenthesization is trivial to construct for k=1, since $R_k=r_k$; we simply put a left parenthesis at the left end of each run of 1's, and a right parenthesis at the right end.

We now show how, given the parenthesization for r_k, A can construct the parenthesization for r_{k+1}. Suppose that A can determine which runs in r_k belong to the same component of 1's in R_k. A can then move to r_{k+1}, having memorized r_k and its parenthesization, and examine the components of 1's in $r_k \cup r_{k+1}$; from this information A can easily determine which runs in r_{k+1} belong to the same component of 1's in R_{k+1}, and can thus construct the parenthesization of r_{k+1}.

For any run ρ of 1's in r_k, as we move leftward from ρ, let L_ρ be the first left parenthesis for which the count of lefts exceeds the count of rights. (If ρ has a left parenthesis at its left endpoint, this itself is L_ρ.) We claim that L_ρ is just the left parenthesis at the leftmost point of $C_\rho \cap r_k$, where C_ρ is the component of 1's in R_k that contains ρ. Indeed, suppose that some other component

D meets r_k between L_ρ and ρ (this is the only way that other parentheses can occur between L_ρ and ρ). If a pair of runs in $D \cap r_k$ separated a pair of runs in $C_\rho \cap r_k$, the paths of 1's in R_k joining these pairs would have to cross, contradicting the fact that $D \neq C_\rho$. Hence all of $D \cap r_k$ must lie between L_ρ and ρ, so that D contributes a pair of parentheses to the count. Thus the first excess left parenthesis found while moving left from ρ cannot come from any other component D; it must come from C_ρ itself.

It follows that, given any two runs ρ_1, ρ_2 of 1's in r_k, A can tell whether or not they belong to the same component in R_k by checking whether or not $L_{\rho_1} = L_{\rho_2}$. As indicated above, this allows A to construct the parenthesization of r_{k+1}, given that of r_k. In particular, A can tell whether a component of 1's that met r_k fails to meet r_{k+1}. A can also tell when a component of 1's reaches the bottom row of Σ. If exactly one of these events occurs, the 1's in Σ are connected; if more than one occurs, they are not.//

References

1. M. Blum and C. Hewitt, Automata on a 2-dimensional tape, Proc. 8th IEEE Conf. on Switching and Automata Theory, 1967, 155-160.

2. M. Blum and W. J. Sakoda, On the capability of finite automata in 2 and 3 dimensional space, Proc. 18th IEEE Conf. on Foundations of Computer Science, 1977, 147-161.

3. J. Mylopoulos, On the recognition of topological invariants by 4-way finite automata, Computer Graphics and Image Processing 1, 1972, 308-316.

4. A. N. Shah, Pebble automata on arrays, Computer Graphics and Image Processing 3, 1974, 236-246.

5. P. Hsia and R. T. Yeh, Marker automata, Information Sciences 8, 1975, 71-88.

6. A. N. Shah and A. Rosenfeld, Two heads are better than one, Information Sciences 10, 1976, 155-158.

7. J. E. Hopcroft and J. D. Ullman, Formal Languages and their Relation to Automata, Addison-Wesley, Reading, MA, 1969.

8. A. N. Shah, On traversing properties of array automata, Technical Report 274, Computer Science Center, University of Maryland, College Park, MD, November 1973.

9. S. M. Selkow, One-pass complexity of digital picture properties, J.ACM 19, 1972, 283-295.

10. A. Rosenfeld and D. L. Milgram, Parallel/sequential array automata, Information Processing Letters 2, 1973, 43-46.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 78-0423	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) PEBBLE, PUSHDOWN, AND PARALLEL-SEQUENTIAL PICTURE ACCEPTORS		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER TR-633
7. AUTHOR(s) Azriel Rosenfeld		8. CONTRACT OR GRANT NUMBER(s) AFOSR-77-3271
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Center University of Maryland College Park, MD 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Math. & Info. Sciences, AFOSR/NM Bolling AFB Washington, D. C. 20332		12. REPORT DATE February 1978
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16 DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19 KEY WORDS (Continue on reverse side if necessary and identify by block number)

Picture languages
Automata
Pebble automata.
Pushdown automata
Cellular automata

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report is a draft of Chapter 7 of a forthcoming book on "Picture Languages." The following chapters have also been issued in report form:

Chapter(s)	Subject	TR
2	Digital geometry	542
3-5	Sequential and parallel acceptors	613
6	Pyramid acceptors	544,596,616

(continued) 20.

Comments on the choice and treatment of the material are invited.