

This is a volume in
COMPUTER SCIENCE AND APPLIED MATHEMATICS
A Series of Monographs and Textbooks

Editor: **WERNER RHEINBOLDT**

A complete list of titles in this series appears at the end of this volume.

PICTURE LANGUAGES

Formal Models for Picture Recognition

AZRIEL ROSENFELD

*Computer Science Center
University of Maryland
College Park, Maryland*



ACADEMIC PRESS New York San Francisco London 1979

A Subsidiary of Harcourt Brace Jovanovich, Publishers

COPYRIGHT © 1979, BY ACADEMIC PRESS, INC.

ALL RIGHTS RESERVED.

**NO PART OF THIS PUBLICATION MAY BE REPRODUCED OR
TRANSMITTED IN ANY FORM OR BY ANY MEANS, ELECTRONIC
OR MECHANICAL, INCLUDING PHOTOCOPY, RECORDING, OR ANY
INFORMATION STORAGE AND RETRIEVAL SYSTEM, WITHOUT
PERMISSION IN WRITING FROM THE PUBLISHER.**

ACADEMIC PRESS, INC.

111 Fifth Avenue, New York, New York 10003

United Kingdom Edition published by

ACADEMIC PRESS, INC. (LONDON) LTD.

24/28 Oval Road, London NW1 7DX

Library of Congress Cataloging in Publication Data

Rosenfeld, Azriel, Date

Picture languages.

(Computer science and applied mathematics)

Includes bibliographical references and index.

1. Machine theory. 2. Optical Pattern recognition.

3. Formal languages. I. Title.

QA267.R67 629.8'91 78-11256

ISBN 0-12-597340-3

PRINTED IN THE UNITED STATES OF AMERICA

79 80 81 82 83 84 85 9 8 7 6 5 4 3 2 1

To my father

ACKNOWLEDGMENTS

The material in this book is based on the work of a number of authors; see the bibliographies at the ends of the chapters for specific attributions. In particular, Chapter 6, on cellular pyramids, is largely the work of C. R. Dyer.

The helpful comments of C. R. Dyer and D. L. Milgram, among others, on a draft of the book are greatly appreciated, as is the typing support that was ably provided by Shelly Rowe. Finally, the support received by the author from the Directorate of Mathematical and Information Sciences, U.S. Air Force Office of Scientific Research, is gratefully acknowledged.

AUTOMATON NOMENCLATURE

For the sake of compactness, abbreviations have usually been used for the names of the various types of automata and languages discussed in this book. The following is a summary of the notation used to construct these abbreviations.

TYPES OF AUTOMATA OR LANGUAGES

BC	bounded cellular
C	cellular
PC	pyramid cellular
TC	triangle cellular
PS	parallel/sequential
NWPS	nonwriting parallel/ sequential
FS	finite-state
PD	pushdown
#P	#-preserving
TB	tape-bounded
T	Turing

SPECIAL TYPES

DLP	diameter-limited perceptron
FRA	frontier-to-root tree acceptor

SUFFIXES

A	acceptor
L	language
M	machine

PREFIXES

D	deterministic
N	nondeterministic
OW	one-way
TW	three-way
U	bottom-up

USED WITH

all
all
FS,PD(#P,TB,T)
FS
PC,TC

EXAMPLES

NTBL	nondeterministic tape-bounded language
DOWFSA	deterministic one-way finite-state acceptor
UTCA	bottom-up triangle cellular acceptor

Chapter 1

INTRODUCTION

This book has two potential classes of readers. For those interested in pattern recognition, it introduces a class of formal approaches; for those interested in automata theory, it presents some basic results on two-dimensional languages. The first two sections of this introduction are addressed to these two groups of readers.

1.1 FOREWORD ON PICTURE RECOGNITION

Pictorial pattern recognition is a well-established and active field [1-3], with numerous applications in such areas as document processing (character recognition), industrial automation (assembly, inspection), medicine (cytology, radiology), and remote sensing, among many others. Efficient picture recognition systems are thus a matter of great practical interest. However, relatively little is known about the computational theory of picture recognition. Given a class of digital pictures (i.e., arrays of numbers representing gray levels), and a class of allowable operations on pictures, one would like to be able to answer such questions as:

- (a) What types of recognition tasks are feasible? For what properties is it possible to determine whether or not a given picture has the given property?
- (b) What is the minimum number of operations (of the given type) required to perform a given recognition task (for pictures of a given size)?

- (c) How can the minimum required computation time be reduced by allowing sets of operations to be performed in parallel?

This book treats these questions from the rather formal standpoint of automata theory. *Array automata* are simple processors that perform sequences of operations on arrays (e.g., pictures); their recognition power depends on which operations they are allowed to perform. *Cellular array automata* are arrays of processors that operate on pictures in a highly parallel fashion (one processor per picture element). The capabilities and relative efficiencies of various types of these automata with respect to various array recognition tasks are the main subject of this book.

Cellular arrays have been frequently proposed, and in a few cases, actually implemented, as fast picture-processing machines (e.g., [4–6]). The growing availability of integrated circuits makes their realization in hardware increasingly practical. Thus the material in this book on cellular recognition algorithms (Chapters 5 and 6, in particular) is of potential practical interest to designers and users of such hardware.

It must be admitted that automata are rather artificial models for picture processing and recognition. As ordinarily defined, an automaton has a bounded amount of internal memory (the condition of which, at a given time, is called the automaton's "state"), and this amount is the same no matter what the size of the input picture array. Thus an automaton cannot internally perform arithmetic operations on numbers of arbitrary size. It cannot even address an arbitrary element of the array, since specifying the coordinates of that element requires a number of bits that does grow (logarithmically) with the array size. Instead, it moves around on the array by making sequences of moves of bounded size—typically, from a given picture element to one of its neighbors. Similarly, the automata in a cellular array are each allowed only a bounded amount of internal memory; thus they cannot address arbitrarily specified cells in the array, but rather pass information from neighbor to neighbor.

In spite of (or perhaps because of) these restrictions, automaton models for picture recognition have been extensively studied, and many theoretical results about their capabilities have been established. This book provides a systematic introduction to this work.

Some work has also been done on other formal recognition models; a notable example is the monograph by Minsky and Papert on "Perceptrons" [7], in which linear threshold functions are used for picture property recognition (see Section 6.3.3). However, further work is needed on models that are less artificial. It is hoped that progress will be made in this direction during the coming years.

1.2 FOREWORD ON AUTOMATA THEORY

This book presents a collection of results on two-dimensional sequential and parallel (cellular) array acceptors (Chapters 4–7). For convenience, some of the analogous one-dimensional results are also reviewed (Chapter 3). Array grammars and their relation to acceptors are also treated (Chapter 8).

Array acceptors have been much less extensively studied than string acceptors, and are usually not treated in any detail in courses on automata and formal languages. However, they have a number of features that should make them of interest to automata theorists. Many of the standard one-dimensional results, e.g., the equivalence of nondeterministic and deterministic finite-state acceptors, break down in two dimensions. Certain array properties, such as connectedness, are trivially recognizable in one dimension, but are much less tractable in two. Array grammars must be defined in a special ("isometric") way in order to ensure that the application of rewriting rules does not have nonlocal effects on the array topology. In addition, many simple questions about array languages remain open. These and many other factors make the study of two-dimensional languages of special interest (aside from their relevance to picture recognition, as suggested in Section 1.1). It is hoped that this book will help to encourage their further study by automata theorists.

1.3 SCOPE OF THIS BOOK

Chapter 2 reviews a number of topics in *digital geometry*—the theory of geometrical properties of subsets of digital pictures. The

subjects covered include connectedness, surroundedness, arcs and curves, borders, perimeter, distance, diameter, and geodesics. This theory provides the foundations for a number of basic array automaton algorithms (e.g., border following, region traversal) that are developed in later chapters. From a more practical standpoint, it serves as a basis for defining and proving the correctness of various fundamental image analysis operations, such as object counting (i.e., connected component counting) and chain encoding [8] of region borders.

Chapter 3 reviews one-dimensional (string) automata and cellular automata, with emphasis on results whose two-dimensional analogs are treated in the subsequent chapters. Chapter 4 treats (sequential) array automata, and Chapter 5 deals with cellular array automata. Chapters 6 and 7 discuss a number of special automaton models, including "cellular pyramids," pebble and push-down automata, and parallel/sequential array automata. Chapter 8 is concerned with array grammars (including a brief review of string grammars), both sequential and parallel, and with their relationship to array automata. (Many other "grammar" models for picture languages have been proposed, and considerable work has been done on the use of syntactic methods in pattern recognition; see [9] for a review of this subject.)

The content of this book is basically mathematical, but an informal approach has been taken, especially in the proofs. Understandability has been favored over formal precision. To promote intelligibility, the most efficient algorithms possible have not always been given. It is hoped that the presentation will be found readable even by those having little or no previous exposure to automata theory.

REFERENCES

1. A. Rosenfeld and A. C. Kak, "Digital Picture Processing." Academic Press, New York, 1976.
2. R. C. Gonzalez and P. Wintz, "Digital Image Processing." Addison-Wesley, Reading, Massachusetts, 1977.
3. W. K. Pratt, "Digital Image Processing." Wiley, New York, 1978.
4. S. H. Unger, A computer oriented toward spatial problems. *Proc. IRE* **46**, 1744–1750 (1958).
5. B. H. McCormick, The Illinois pattern recognition computer—ILLIAC III. *IEEE Trans. Electronic Computers* **12**, 791–813 (1963).

6. M. J. B. Duff, A cellular logic array for image processing. *Pattern Recognition* **5**, 229–247 (1973).
7. M. Minsky and S. Papert, "Perceptrons." MIT Press, Cambridge, Massachusetts, 1969.
8. H. Freeman, Computer processing of line drawing images. *Computing Surveys* **6**, 57–97 (1974).
9. K. S. Fu, "Syntactic Methods in Pattern Recognition." Academic Press, New York, 1974.

Chapter 2

DIGITAL GEOMETRY

2.1 INTRODUCTION

In the process of analyzing a digital picture, one usually begins by extracting subsets of various types from the picture; this process is called "segmentation" (see e.g., [1, Chapter 8]). One then proceeds to measure various properties of the extracted subsets, including geometrical properties such as connectivity, area, and so on. A variety of algorithms for computing geometrical properties of digital picture subsets can be found in [1, Chapter 9] and in [2, Chapter 9].

By *digital geometry* we mean the mathematical study of geometrical properties of digital picture subsets. This chapter develops some basic parts of digital geometry, dealing with connected sets, arcs and curves, borders, area and perimeter, distance and diameter, and geodesics. Many other topics have been omitted, e.g., the theory of digital convexity; some references on these topics can be found in Section 2.10.

The material in this chapter provides a theoretical basis for many standard picture analysis algorithms, involving such tasks as object counting (i.e., counting the connected components of a set S) and border following (i.e., following the border defined by a connected component of a set S and a connected component of its complement $\bar{S}$). Some of these algorithms will play key roles in establishing the picture recognition capabilities of automata in later chapters.

In this chapter, a *digital picture* Σ is a finite rectangular array

whose elements are called "points" (or sometimes, "pixels"—short for "picture elements"). Each point P of Σ is defined by a pair of Cartesian coordinates (x, y) , which we may take to be integer valued.

2.2 BASIC CONCEPTS

A point $P = (x, y)$ in a digital picture Σ has two types of neighbors [3]:

- (a) its four horizontal and vertical neighbors (u, v) such that $|x - u| + |y - v| = 1$ and
- (b) its four diagonal neighbors (u, v) such that $|x - u| = |y - v| = 1$.

We shall refer to the neighbors of type (a) as the *4-neighbors* of P , and to the neighbors of both types, collectively, as the *8-neighbors* of P . The former neighbors are said to be *4-adjacent* to P , and the latter, *8-adjacent*. Note that if P is on the border of Σ , some of its neighbors may not exist.

By a *path* π of length n from P to Q in Σ we mean a sequence of points $P = P_0, P_1, \dots, P_n = Q$ of Σ such that P_i is adjacent to P_{i-1} , $1 \leq i \leq n$. Note that this comprises two definitions in one, namely, *4-path* and *8-path*, depending on whether we use the 4- or 8-definition of adjacency.

Let S be a nonempty subset of Σ . To avoid special cases, we shall usually assume that S does not meet the border rows and columns of Σ . We say that P and Q are *connected* in S if there exists a path from P to Q consisting entirely of points of S . Clearly, "connected in S " is an equivalence relation, since a path of length 0 is a path, the reversal of a path is a path, and the concatenation of two paths is a path. The equivalence classes under this relation are called the *connected components* of S . Again, these are all double-duty definitions.

We can also define connected components for the complement $\bar{S}$ of S . Evidently, exactly one of these contains the border of Σ ; we call it the *background* component of $\bar{S}$. All other components of $\bar{S}$, if any, are called *holes* in S . If S has no holes, it is called *simply-connected*.

One of the main results of this chapter (Section 2.6) will be that the components of S and of $\bar{S}$, under the adjacency relationship,

define a graph that is a tree. As a refinement of this result, we shall show that if a component of S and a component of $\bar{S}$ are adjacent, then one of them surrounds the other [4]. We shall also define "digital simple closed curve" and prove the digital Jordan curve theorem [5], as well as its converse [6]: If $\bar{S}$ has exactly two components and every point of S is adjacent to both of them, then S is a simple closed curve (Sections 2.4 and 2.5). In addition, we shall introduce natural metrics on digital pictures [7] and establish some of their properties, as well as developing some useful formulas for computing the genus of S (Sections 2.7 and 2.8).

Remark 1[§] The definition of connectedness given above is an "arcwise" definition, rather than a definition in terms of open and closed sets. We can, however [8], define a topology on Σ in which the standard notion of connectedness reduces to the definition just given. Let

$$\begin{aligned} M(P) &\equiv M(x, y) = \{P\}, & \text{if } x + y \text{ is odd} \\ &= \{P \text{ and its 4-neighbors}\}, & \text{if } x + y \text{ is even} \end{aligned}$$

It is not hard to see that the intersection of any two M s is either an M or empty. If we take the M s as a basis for the open sets, then a set is connected in the resulting topology iff it is 4-connected as defined above.

Proof: Let U be connected, $P \in U$, and U_P be the 4-component of U containing P . Clearly, $Q \in U_P$ implies $M(Q) \cap U \subseteq U_P$, and $Q \in (U - U_P)$ implies $M(Q) \cap U \subseteq U - U_P$, so that U_P and $U - U_P$ are both open in U . Since U is connected, $U - U_P$ must thus be empty, so that $U = U_P$ is 4-connected. Conversely, let V be 4-connected and suppose $V = X \cup Y$ where $X \cap Y = \emptyset$. Let $P = P_0, P_1, \dots, P_n = Q$ be a 4-path from $P \in X$ to $Q \in Y$, so that for some m we have $P \in X, P_{m+1} \in Y$; then the sum of coordinates of either P_m or P_{m+1} must be even, so that either $P_m \in M(P_{m+1})$ or $P_{m+1} \in M(P_m)$, which implies that either X or Y fails to be open, proving that V is connected. $\square$

Remark 2 If Σ were a hexagonal rather than a Cartesian array, there would be only one type ("6") of adjacency, connectedness, etc., and some of the results in this paper could be simplified. We can simulate a hexagonal array in a Cartesian array by omitting

[§]This remark assumes some knowledge of general topology; it can be skipped by nontopologists.

one pair of diagonal neighbors, e.g., defining the neighbors of (i, j) to be $(x \pm 1, y)$, $(x, y \pm 1)$, and

$$\begin{aligned} (x - 1, y \pm 1), & \quad \text{if } y \text{ is even} \\ (x + 1, y \pm 1), & \quad \text{if } y \text{ is odd} \end{aligned}$$

2.3 ARCS AND CURVES

Our goal in the next section is to prove a Jordan curve theorem, and a converse to it, for digital curves. Before defining digital curves, we define digital arcs and establish some of their properties.

An *arc* is a path $\alpha \equiv P_0, P_1, \dots, P_n$ such that

- (a) $P_i = P_j$ iff $i = j$, and
- (b) P_i is a neighbor of P_j iff $i = j \pm 1$.

These conditions imply that α neither crosses nor "touches" itself. P_0 and P_n are called the endpoints of α .

Since by (a) the set of points of α are all distinct, α can be regarded as a totally ordered set of points that satisfies (b). Conversely, if S is a set of points that can be ordered in such a way that (b) holds, we can speak of S as being an arc. Clearly, the reversal of an arc is an arc. We denote by $|S|$ the number of points in S , and we assume from now on that $|S| \geq 2$.

PROPOSITION 2.3.1 S is an arc iff

- (c) S is connected;
- (d) all but two points of S have exactly two neighbors in S , and those two have exactly one.

Proof: If S is an arc, it is certainly connected, and every point $P_k \neq P_0, P_n$ of S has exactly two neighbors $P_{k \pm 1}$ by (b), while P_0 and P_n have exactly one each. Conversely, if (d) holds, call one of the exceptional points P_0 , call its sole neighbor P_1 , call P_1 's remaining neighbor P_2 , and so on. This process must terminate (since S is finite), and it can only terminate at the other exceptional point. The resulting sequence $P_0, P_1, \dots, P_n$ is certainly an arc, and it remains to show that it is all of S . Let $Q \in S$ not be one of the P s; since S is connected, there is a path $Q = Q_0, \dots, Q_m = P_j$ in S from Q to some P_j . Let $Q_k = P_i$ be the first point of this path that is one of the P s; then Q_{k-1} is not one of the P s but is

a point of S that is a neighbor of P_i , which contradicts the construction of the P s. $\square$

Up to now we did not have to specify whether we were using the 4- or 8-definitions; our results hold for both. Now, however, we want to prove that an arc S is simply-connected, and this is only true if we use the proper definitions of connectedness for S and for its holes. For example, if we use the 4-definition for both, then the arc

$$\begin{array}{ccc} P & P & P \\ & P & P \\ P & & P \end{array}$$

has a hole. We shall see further examples of this type of behavior in later sections. *Let us assume from now on that we always use the opposite definitions for S and $\bar{S}$.* This certainly negates our counterexample; the 4-arc shown above has a 4-hole, but not an 8-hole. [An intuitive justification for the assumption is as follows: If S contains two diagonally adjacent points (the P s in $\begin{smallmatrix} PO \\ OP \end{smallmatrix}$) and we regard the P s as adjacent, then we cannot regard the O s ($\in \bar{S}$) as adjacent, and vice versa.]

Under the assumption of opposite definitions for S and $\bar{S}$, we can now prove

PROPOSITION 2.3.2 An arc is simply-connected.

Proof: This is clear if the arc S has length 0; suppose it true for length $n - 1$, and let $S = \{P_0, \dots, P_n\}$ be an arc of length n . By induction hypothesis, since $S - \{P_n\}$ is clearly an arc, any points Q, R in $\bar{S}$ are connected (in the $\bar{S}$ sense) in $\bar{S} \cup \{P_n\}$. But it is easily verified (by examining all possible cases) that any occurrence of P_n on a path from Q to R can be replaced by a sequence of points in $\bar{S}$. Thus Q and R are connected in $\bar{S}$ too, so that S is simply-connected. $\square$

In other words, a 4-arc has no 8-holes, while an 8-arc has no 4-holes, and a fortiori, no 8-holes.

EXERCISE 2.3.1 S is both a 4-arc and an 8-arc iff it is a horizontal or vertical straight line segment.

EXERCISE 2.3.2 A path from P_0 to P_n that is as short as possible is an arc.

A *curve* (short for "simple closed curve") is a path $\gamma \equiv P_0, P_1, \dots, P_n$ such that

- (a) $P_i = P_j$ iff $i = j$, and
- (b) P_i is a neighbor of P_j iff $i \equiv j \pm 1$ (modulo $n + 1$).

These conditions imply that γ never crosses or touches itself (except for P_0 touching P_n). As in the arc case, we can regard a curve as a cyclically ordered set, and can regard any such set that satisfies (b) as a curve. Clearly, any cyclic permutation of a curve is a curve.

The definition of "curve" just given allows a number of degenerate cases. For the 4-definition, the sets

$$P; \quad PP \quad \text{or} \quad \begin{matrix} P \\ P; \end{matrix} \quad \text{and} \quad \begin{matrix} PP \\ PP \end{matrix}$$

are all curves; the same is true for

$$\begin{matrix} P; & PP, & \begin{matrix} P \\ P, \end{matrix} & \begin{matrix} P \\ P, \end{matrix} & \text{or} & \begin{matrix} P \\ P; \end{matrix} \\ PP & PP & P & & & P \\ P, & P, & PP, & & \text{or} & PP \end{matrix}$$

by the 8-definition. We shall rule out these cases from now on.

EXERCISE 2.3.3 When we rule out these cases, a 4-curve must have at least eight points, and an 8-curve, four points. (In particular, show that there are no 4-curves with 5, 6, or 7 points.)

EXERCISE 2.3.4 No S can be both a 4-curve and an 8-curve.

In analogy with Proposition 2.3.1 we have

PROPOSITION 2.3.3 S is a curve iff

- (c) S is connected;
- (d) Every point of S has exactly two neighbors in S .

Proof: Analogous to that of Proposition 2.3.1. $\square$

It is not hard to see that if (d) alone holds, S is a union of disjoint curves, and conversely.

PROPOSITION 2.3.4 Deleting any point from a curve makes it an arc.

Proof: This follows immediately from the definitions. $\square$

COROLLARY 2.3.5 A curve has at most one hole.

Proof: It is easily seen that if $N(P)$ is the neighborhood of any point P of S , then $N(P) \cap \bar{S}$ has exactly two components. Thus deleting P from S can merge at most two components of $\bar{S}$. But by Proposition 2.3.4, $S - \{P\}$ has just one component; hence $\bar{S}$ has at most two. $\square$

What we would next like to prove is that a curve has exactly one hole; this is basically our Jordan curve theorem. However, here again we must be careful to use the right combinations of definitions for S and $\bar{S}$. For example, if we use the 4-definitions for both, then the 4-curve

$$\begin{array}{ccccc} P & P & P & & \\ & P & & P & P \\ & P & P & & P \\ & & P & P & P \end{array}$$

has two 4-holes; whereas if we use the 8-definitions for both, then the 8-curve

$$\begin{array}{ccc} & P & \\ P & & P \\ & P & \end{array}$$

has no 8-holes. However, if we use opposite definitions for S and $\bar{S}$, as already proposed, these counterexamples are negated.

2.4 THE JORDAN CURVE THEOREM

Our proof of the Jordan curve theorem is basically similar to a standard proof of the theorem for polygons; but there are some details of interest. In Section 2.5 we shall also prove a converse to the Jordan curve theorem.

For any point P , let h_P be the right half-line emanating from P —in other words, $h_{(x,y)} = \{(z,y) | z \geq x\}$. Let γ be a curve and let $P \notin \gamma$. Let $\rho \equiv P_r, \dots, P_s$ be a run of consecutive points of γ that lie on h_P , but such that P_{r-1} and P_{s+1} do not lie on h_P . Thus P_{r-1} and

P_{s+1} must lie on the row above or below h_P . If they are both above or both below, we say that h_P *touches* γ in ρ ; while if one is above and the other below, we say that h_P *crosses* γ in ρ . If h_P crosses γ an odd number of times, we say that P lies *inside* γ ; if an even number of times, we say that it lies *outside*.

PROPOSITION 2.4.1 Neighboring points of $\bar{\gamma}$ are either both inside or both outside γ .

Proof: If they are horizontal neighbors, this is immediate; suppose they are vertical or diagonal neighbors, say P above Q . Let $\rho \equiv P_r, \dots, P_s$ be a run of consecutive points of γ that lie on $h_P \cup h_Q$; thus P_{r-1} and P_{s+1} must each lie on the row above P or the row below Q . (When P, Q are diagonal neighbors, γ is a 4-curve, so it cannot leave $h_P \cup h_Q$ by slipping between P and Q .)

If they are both above (and similarly if both are below), let P_u and P_v be the first and last points of ρ that lie on h_Q . Then it is easily seen that h_P crosses γ in the subruns $P_r, \dots, P_{u-1}$ and $P_{v+1}, \dots, P_s$ of ρ , and h_Q does not cross γ in any subrun of ρ . If no points of ρ lie on h_Q , neither h_P nor h_Q crosses γ in any subrun of ρ . Thus in either case, the difference between the numbers of crossings is even (2 or 0).

If P_{r-1} is above and P_{s+1} below (or vice versa), let P_u be the first point of ρ that lies on h_Q , and P_v the last point of ρ that lies on h_P . Then it is easily seen that h_P crosses γ in the subrun $P_r, \dots, P_{u-1}$ of ρ , and h_Q crosses γ in the subrun $P_{v+1}, \dots, P_s$, but neither crosses γ in any other subrun of ρ ; thus here again the difference between the numbers of crossings is even.

Since these remarks hold for any run ρ , it follows that the difference between the total number of crossings of γ by h_P and h_Q is even, so that P and Q are either both inside or both outside γ . $\square$

COROLLARY 2.4.2 Points that are connected in $\bar{\gamma}$ are either both inside or both outside γ .

Proof: Suppose we had $P, Q \in \gamma$ with P inside and Q outside, and a path $P = P_0, P_1, \dots, P_n = Q$ in $\bar{\gamma}$ from P to Q . Then for some $1 \leq i \leq n$ we have P_{i-1} inside and P_i outside; but P_{i-1} and P_i are neighbors, contradicting Proposition 2.4.1. $\square$

PROPOSITION 2.4.3 The inside and outside of γ are both non-empty.

Proof: A point to the right of every point of γ (e.g., in the right-hand column of Σ) is evidently outside. On the other hand, let $\rho = P_r, \dots, P_s$ be a run of rightmost points of γ , so that P_{r-1} and P_{s+1} are in the column to the left of ρ . It follows readily from the definition of a curve that not all points of ρ can have their left neighbors in γ . Let P_i ($r \leq i \leq s$) have left neighbor P not in γ ; then it is easily seen that h_P crosses γ exactly once, so P is inside γ . $\square$

We thus have

THEOREM 2.4.4 A curve has exactly one hole.

Proof: It has at most one, by Corollary 2.3.5. On the other hand, the inside and outside of γ cannot meet the same component of $\tilde{\gamma}$ (Corollary 2.4.2), and both are nonempty; thus $\tilde{\gamma}$ has at least two components, giving γ at least one hole. $\square$

In fact, we have now shown that the background Z and hole W of a curve γ are just its outside and inside. *Proof:* The outside evidently meets Z ; hence the inside cannot meet Z . Thus the inside is contained in W , implying that the outside cannot meet W . Since every point of $\tilde{\gamma}$ is either inside or outside γ , and either in Z or in W , it follows that Z and W are exactly the inside and outside, respectively.

We can also show that every point P of γ is adjacent to both Z and W . [Here "adjacent" can mean either 4- or 8-adjacent, since if a component of S (here γ) and a component of $\tilde{S}$ are 8-adjacent, they are readily also 4-adjacent.[§]] In fact, suppose P were adjacent to only one of them, say W . We recall that $\gamma - \{P\}$ is connected (Proposition 2.3.4); hence for any $Q \in Z$, $R \in W$ we have a path $\rho: Q = Q_0, Q_1, \dots, Q_n = R$ in $\gamma - \{P\} = \tilde{\gamma} \cup \{P\}$ from Q to R . Clearly, π passes through P , since Q and R cannot be connected in $\tilde{\gamma}$. Let Q_i be a first occurrence of P on γ ; thus Q_{i-1} is a point of $\tilde{\gamma}$ adjacent to P and hence is in W . But then $Q_0, \dots, Q_{i-1}$ is a path in $\tilde{\gamma}$ from $Q \in Z$ to $Q_{i-1} \in W$ —a contradiction. (The proof is analogous if P is adjacent only to Z , taking Q_i to be a last occurrence of P on π and considering the path $Q_{i+1}, \dots, Q_n$ from $Q_{i+1} \in Z$ to $R \in W$.) In summary, we have established

[§]To see this, let $P \in \tilde{S}$, $Q \in S$, and consider $\begin{smallmatrix} PR' \\ RQ \end{smallmatrix}$; the R s must either be in S or in $\tilde{S}$.

PROPOSITION 2.4.5 The complement of a curve γ has exactly two components, namely the inside and the outside of γ , the latter being the background component. Moreover, every point of γ is adjacent to both of these components.

Our next goal is to prove that Proposition 2.4.5 actually characterizes curves. Specifically, we shall show that if S is any set such that S has exactly two components, and every point of S is adjacent to both of these components, then S is a curve. However, we must first prove a number of facts about sets whose complements have (at most) two components.

2.5 SIMPLE POINTS

Let P be a point of S , and let $N(P)$ denote the set of eight neighbors of P , excluding P itself.

PROPOSITION 2.5.1 The following statements are all equivalent:

- (a) $S \cap N(P)$ has the same number of components (in the S sense) as $S \cap [N(P) \cup \{P\}]$;
- (b) $\bar{S} \cap N(P)$ has the same number of components (in the $\bar{S}$ sense) as $[\bar{S} \cap N(P)] \cup \{P\}$;
- (c) $S \cap N(P)$ has just one component adjacent to P ("component" and "adjacent" in the S sense);
- (d) $\bar{S} \cap N(P)$ has just one component adjacent to P ("component" and "adjacent" in the $\bar{S}$ sense).

Proof: The equivalence of (c) and (d) can be easily verified by enumeration of cases. Clearly, (c) implies (a), since when we adjoin P , it becomes part of the component to which it was adjacent and has no effect on the other components (if any). Conversely, if $S \cap N(P)$ has two or more components adjacent to P , then when we adjoin P , these components merge and the number of components is reduced; thus (a) implies (c), and analogously, (b) is equivalent to (d). $\square$

A point of S that has the properties of Proposition 2.5.1 will be called *simple*. Evidently, neither an isolated point of S (i.e., a point of S with no neighbors in S) nor an interior point of S (all of whose

neighbors are in S) can be simple; whereas a point of S that has just one neighbor in S is always simple.

PROPOSITION 2.5.2 P is simple iff $S - \{P\}$ has the same number of components as S , and $\bar{S} \cup \{P\}$ has the same number of components as $\bar{S}$.

Proof: If P is simple, it is not an isolated point of S , so $S - \{P\}$ cannot have fewer components than S . Suppose it has more, i.e., there exist Q, R in $S - \{P\}$ that are connected in S but not in $S - \{P\}$; then there is a path π from Q to R in S , and P must be on π . The points just preceding and following P on π must be in components of $S \cap N(P)$ adjacent to P . By (c) of Proposition 2.5.1, there is only one such component, so that we can find a path between these points that lies in the component and avoids P —a contradiction.

Similarly, if P is simple, it is not an interior point of S , so $\bar{S} \cup \{P\}$ cannot have more components than $\bar{S}$; nor can it have fewer by an argument analogous to that in the preceding paragraph, using (d) of Proposition 2.5.1.

Conversely, if the conditions hold, P is certainly not an isolated or interior point. Let Q, R be points of S adjacent to P that belong to two different components of $N(P) \cap S$. Q, R must be connected in $S - \{P\}$, since it has no more components than S . Let π be a shortest possible path from Q to R in $S - \{P\}$; thus π is an arc (Exercise 2.3.2), and if we adjoin P to π , we get a curve γ . (If P is adjacent to some point of π other than Q and R , we can use that point instead of Q or R and obtain a shorter path.) By Proposition 2.4.5, P has neighbors P', P'' in $\bar{\gamma}$ that belong to the inside and the outside of γ , respectively, and we can readily pick P' and P'' to be in $\bar{S}$. Since P' and P'' are not connected in $\bar{\gamma}$, they are certainly not connected in $\bar{S}$; but they are connected in $\bar{S} \cup \{P\}$, so it must have fewer components than $\bar{S}$ —a contradiction. $\square$

Thus if P is not simple, exactly one of the following must be true: P is isolated or interior; $S - \{P\}$ has more components than S (i.e., P is a *cut point* of S); or $\bar{S} \cup \{P\}$ has fewer components than $\bar{S}$.

COROLLARY 2.5.3 If S is simply-connected and $P \in S$ is not an isolated, interior, or simple point, then $S - \{P\}$ is not connected, but consists of components that are simply-connected. If S is

connected and has exactly one hole, $S - \{P\}$ is either simply-connected or not connected.

Using these results, we can now prove

PROPOSITION 2.5.4 If S is simply-connected and $|S| > 1$, then S has at least two simple points.

Proof: This is clear if $|S| = 2$; we proceed by induction on $|S|$. Let P be the leftmost of the uppermost points of S , so that P 's neighborhood looks like

$$\begin{array}{ccc} O & O & O \\ O & P & P' \\ & P'' & Q \end{array}$$

where P' or P'' is in S and the O s are in $\hat{S}$. (This assumes that we use the 4-definitions for S ; the proof for the 8-definitions is analogous.)

If just one of them, say P' , is in S , P is simple, so that by Proposition 2.5.2, $S - \{P\}$ is still simply-connected. If $S - \{P\} = \{P'\}$ (i.e., $|S| = 2$), P and P' are both simple. If not, by the induction hypothesis, $S - \{P\}$ has at least two simple points. But it is not hard to see that any point $R \neq P'$ that is simple in $S - \{P\}$ is also simple in S , so that S has at least the two simple points P and R . Similarly, if P' , P'' , and Q are all in S , P is simple, so that $S - \{P\}$ is still simply-connected and hence has simple points, which readily must also be simple in S .

If P' , P'' are in S but $Q \notin S$, then P is not simple, so that by Corollary 2.5.3, $S - \{P\}$ is not connected. Thus P' and P'' must be in different components of $S - \{P\}$. If the component containing P' is $\{P'\}$, then P' is simple in S ; but if not, by the induction hypothesis it has at least two simple points, and any of these except possibly P' is also simple in S . A similar argument applies to the component containing P'' ; thus here again S has at least two simple points. $\square$

The result of Proposition 2.5.4 can be sharpened as follows: Define an "end" as a point having only one neighbor in S , and a "corner" as a point having just two neighbors in S , where the diagonal neighbor between them is also in S (see the case P' , P'' , Q all in S in the proof). Then S has at least two ends, or an end and two corners, or four corners. *Proof:* In our first case (just $P' \in S$), if

$S - \{P\} = \{P'\}$, there are two ends, P and P' . If not, $S - \{P\}$ has the desired points, of which at most P' can be an end in $S - \{P\}$ but not in S , and S has the additional end P . In our third case ($P', P'' \in S, Q \notin S$), the two components of $S - \{P\}$ have the desired points, and at most P' and P'' can be in these components but not in S . We can apply similar reasoning to the rightmost of the uppermost points of S , and to the leftmost and rightmost of the lowermost points of S , and if the analog of the first or third case holds for any of them, we are done. If not, the second case ($P', P'', Q \in S$) holds for all of them, so they are all corners. These remarks provide us with the following neat characterization of arcs.

THEOREM 2.5.5 S is an arc iff it is simply-connected and has exactly two simple points.

Proof: The "only if" is clear. Conversely, by the remarks in the previous paragraph, the two simple points must be ends; thus if $|S| = 2$, we are done. We proceed by induction on the number of points of S , and once again let P be the leftmost of the uppermost points of S . If P is an end, it can be seen (by enumeration of cases) that $S - \{P\}$'s simple points are all simple points of S , except possibly for P 's neighbor; hence $S - \{P\}$ can only have two ends, and so is an arc, which implies that S is also an arc, obtained by adjoining P to one end of $S - \{P\}$. If P is nonsimple, $S - \{P\}$ consists of two simply-connected components; by the same argument, each of them (if it has more than one point) has only two ends and so is an arc. S is then the arc obtained by joining these two arcs through the added point P . $\square$

Similarly, we can establish a neat characterization of curves:

THEOREM 2.5.6 S is a curve iff it is connected, has exactly one hole, and has no simple points.

Proof: The "only if" is clear. To prove the "if," note first that no $P \in S$ can have fewer than two neighbors in S (if it had none, we would have $S = \{P\}$, which has no holes; if it had one, it would be simple). It can also be shown that no point of S can have more than two neighbors in S . The argument is along the following lines: If S had interior points, then a border point adjacent to an interior point would have more than two neighbors; thus it suffices to show that a noninterior $P \in S$ cannot have more than two.

By Corollary 2.5.3, $S - \{P\}$ is either simply-connected or not connected. In the former case, $S - \{P\}$ has simple points; by enumeration of cases, some of these must be simple in S —a contradiction. In the latter case, at most one of the components of $S - \{P\}$ can have a hole (since S had only one hole); hence another one is simply-connected and has simple points, some of which (by enumeration of cases) must be simple in S —a contradiction. Thus every $P \in S$ has exactly two neighbors in S , which makes S a curve by Proposition 2.3.3. $\square$

Finally, we can use this last result to prove our desired converse to the Jordan curve theorem—more precisely, a converse to Proposition 2.4.5:

COROLLARY 2.5.7 Let S be connected, $\bar{S}$ have exactly two components, and any point of S be adjacent to both of these components; then S is a curve.

Proof: Since any $P \in S$ is adjacent to both components, $\bar{S} \cup \{P\}$ has only one component; hence no $P \in S$ is simple, making S a curve by Theorem 2.5.6. $\square$

Similarly, we have

COROLLARY 2.5.8 S is a curve iff it is connected and not simply-connected but $S - \{P\}$ is simply-connected for all $P \in S$.

Proof: It is not hard to see that S must have points P that are adjacent to only one hole (e.g., consider a rightmost point of S ; if it is adjacent to two holes, consider the point above or below it). Since $S - \{P\}$ must be simply-connected, S cannot have more than that one hole, which makes it a curve by Theorem 2.5.6. $\square$

The assumption in Corollary 2.5.7 that S is connected is redundant; we shall show at the end of Section 2.6 that if every point of S is adjacent to two components of $\bar{S}$, then S must be connected.

2.6 BORDERS, ADJACENCY, AND SURROUNDEDNESS

Let Γ be the graph whose nodes are the components of S and of $\bar{S}$, and in which two nodes are joined by an arc iff the two compo-

nents are adjacent. (We recall from Section 4 that this always implies 4-adjacency.) We call Γ the *adjacency graph* of S . Our main goal in this section is to show that Γ is a tree. Clearly, Γ is connected, since any two points P, Q of Σ can be joined by a path, and the sequence of components of S and $\bar{S}$ encountered by this path constitutes a path in Γ between the components containing P and Q . Thus we need only show that Γ can have no cycles. This is equivalent to showing that if T is a component of S , and U, V are components of $\bar{S}$ adjacent to T (or vice versa), then T separates U and V —i.e., any path (in the sense of $\bar{S}$'s connectedness) from U to V must meet T .

If we knew that U and V had to be in different components of $\bar{T}$, then T clearly would separate them. Suppose they were in the same component W of $\bar{T}$, and let W_T be the set of points of W that are 4-adjacent to T ; thus U and V would both meet W_T . Now $W_T \subseteq \bar{S}$, since a point adjacent to (but not in) the component T of S cannot be a point of S . Hence if we could prove that W_T is connected (in the $\bar{S}$ sense), U and V could not both meet it, since they are distinct components of $\bar{S}$. We shall therefore begin this section by showing how the results of Section 2.5 can be used to prove that W_T is connected. Specifically, we shall define an algorithm that visits a succession of points of W_T , show that the points it visits are all connected, and prove that it visits all of W_T .

A general version of this border-following algorithm, which we shall call BF, is as follows: Let X and Y be adjacent components of S and $\bar{S}$, respectively. We denote by X_Y the set of points of X that are 4-adjacent to Y , and call it the *Y-border* of X ; we analogously define the *X-border* Y_X of Y . Let P_0 and Q_0 be a pair of 4-adjacent points in X and Y , respectively, and suppose for concreteness that we are using 8-connectedness for S . We mark P_0 and Q_0 , and then proceed to determine the next pair P_1, Q_1 as follows: Let the points of $N(P_0)$ (the 8-neighborhood of P_0), in (say) clockwise order starting from Q_0 , be $Q_0 = R_1, R_2, \dots, R_8$. If none of the R s is in S , P_0 is an isolated point of S , and we are done. Otherwise, let R_i be the first of the R s that is in S . Take R_{i-1} as Q_1 , R_i as P_1 , and repeat the process with P_1, Q_1 replacing P_0, Q_0 . Note that Q_1 is in fact in $\bar{S}$ and is 4-adjacent to P_1 , by definition of the R s. BF stops when for some k we have $P_k = P_0$, provided one of the R s is Q_0 . It is clear that the successive P s found by BF are 8-adjacent points of S , so the set of P s found is 8-connected and is thus a subset of X_Y . Similarly, the set of Q s, together with their intermediate R s (e.g.,

$Q_0 = R_1, R_2, \dots, R_{i-1} = Q_1$), is a 4-connected subset of Y_X . If we were using 4-connectedness for S , we could use an exactly analogous algorithm to find a succession of pairs (Q_i, P_i) , starting from (Q_0, P_0) , by examining the neighborhoods of the Q s.

We must now prove that BF works, i.e., that it does stop and only after having visited all of X_Y (and Y_X). Note first that the points $R_1, \dots, R_i$ of Σ examined by BF at each step are all in X or Y . Thus the operation of BF remains the same if all points of $\tilde{S}$ except those in Y and in the background component Z are transferred from $\tilde{S}$ to S . When this is done, S has at most one hole, so its component X has at most one hole. (The points of a hole in X that are adjacent to X must be in $\tilde{S}$; if X had two holes, we would have points of $\tilde{S}$ that are not connected to each other or to the background in $\tilde{X}$, hence not connected in $\tilde{S}$, so that S would have two holes.) Thus it suffices to show that BF works if X has at most one hole.

If X is simply-connected, we can prove that BF works by induction on the number of points in X . There is no problem if X has only a few points. Suppose it has n points, and let P be a simple point of X (Proposition 2.5.4) so that $X - \{P\}$ is still simply-connected. By induction hypothesis, BF works for $X - \{P\}$; moreover, it is easily verified that BF's operation on X differs from its operation on $X - \{P\}$ only in the vicinity of P , and correctly reflects the changes in X_Y and Y_X caused by the addition of P . Thus BF works for all simply-connected X s.

We can easily show that BF works if X is a curve. Suppose X has exactly one hole but is not a curve; then by Theorem 2.5.6, it has a simple point P . Thus we can use an induction argument similar to that in the previous paragraph to show that BF works in this case too.

Applying these results to the problem at the beginning of the section, we see that indeed W_T must be connected, so that distinct components U, V of $\tilde{S}$ cannot be in the same component W of $\tilde{T}$, and so are separated by T . We have thus proved

THEOREM 2.6.1 For any $S \subseteq \Sigma$, the adjacency graph of S is a tree.

We can prove a little more about the adjacent components of S and $\tilde{S}$ by introducing the concept of surroundedness. Let U, V be any disjoint subsets of Σ ; we say that U *surrounds* V if any 4-path from V to the edge of Σ meets U . It is easily verified that "sur-

rounds" is a strict partial order relation, i.e., it is irreflexive, anti-symmetric, and transitive. Clearly, any $S \subseteq \Sigma$ surrounds its holes (if any)—indeed, any path (in the sense of $\tilde{S}$ connectedness) from a hole W to the background Z must meet S , since W and Z are different components of $\tilde{S}$. On the other hand, Z clearly surrounds S , and thus surrounds every component of S and every other component of $\tilde{S}$.

PROPOSITION 2.6.2 Let X, Y be adjacent components of $S, \tilde{S}$, respectively; then either X surrounds Y or vice versa. Moreover, exactly one component of $\tilde{S}$ surrounds any given component of S (and vice versa for nonbackground components of $\tilde{S}$).

Proof: Two Y s cannot be in the same component of $\tilde{X}$, as already shown. Thus at most one Y can be in the background component Z of $\tilde{X}$; all others are in holes in X and so are surrounded by X . Let P be a rightmost point of X , and Q be P 's right neighbor; thus $Q \in \tilde{S}$ is in some Y , call it Y_0 . Clearly, X does not surround Y_0 , so $Y_0 \subseteq Z$. Let π be a path from X to the edge of Σ , and let P_i be the last point of X on π . Then $P_{i+1} \in \tilde{S}$ is in some Y , and since π does not meet X again, this Y is not surrounded by X and hence must be Y_0 . Thus any path π from X to the edge of Σ meets Y_0 , so that Y_0 surrounds X . $\square$

These observations establish

PROPOSITION 2.6.3 The adjacency tree of S can be regarded as a directed tree under the relationship of surroundedness; the root of this tree is the background component Z of $\tilde{S}$.

EXERCISE 2.6.1 If S is connected, its holes (if any) are simply-connected.

As a further corollary of Theorem 2.6.1, we can prove the assertion made at the end of Section 2.5. Note first that if U, V are components of S and T, W are components of $\tilde{S}$, it is impossible that U and V are each adjacent to both W and T , since the adjacency tree would then contain a cycle. Now let S be a set such that every point of S is adjacent to two components of $\tilde{S}$; then S cannot have two components, by the previous sentence, and so must be connected. We can thus sharpen Corollary 2.5.7 as follows: Let $\tilde{S}$ have exactly two components and let every point of S be adjacent to both of these components; then S is a curve.

2.7 AREA, PERIMETER, AND GENUS

The *area* $|S|$ of $S \subseteq \Sigma$ is simply the number of points in S . The *perimeter* $\langle S \rangle$ of S can be defined in a number of ways; perhaps the simplest of these is the number of points of S that are 4-adjacent to $\bar{S}$. An alternative might be to define $\langle S \rangle$ as the number of steps required by BF (Section 2.6) to visit all the borders of all the components of S . Note that these two definitions can give rather different results; e.g., if S is

$$P P P P P P P P$$

then the first definition gives $\langle S \rangle = |S| = 8$, while the second definition gives $\langle S \rangle = 14$.

A standard measure of the *noncompactness* (or "complexity") of a simply-connected object T is $\langle T \rangle^2 / |T|$; this measure is commonly used in both perception psychology and pattern recognition. In ordinary geometry, for well-behaved objects T , the *isoperimetric inequality* guarantees that this ratio is never less than 4π , and achieves this minimum iff T is a circle. In digital geometry, on the other hand, smallness of this ratio does not correspond to circularity. For example, let O_{hk} be the "digital octagon" whose horizontal and vertical sides have h points each, and whose diagonal sides have k points each; e.g., O_{32} is the octagon

$$\begin{array}{c} P P P \\ P P P P P \\ P P P P P \\ P P P P P \\ P P P \end{array}$$

If $h = 1$, O_{hk} is a diamond; if $k = 1$, it is a square. It is readily verified that

$$\begin{aligned} |O_{hk}| &= [h + 2(k - 1)]^2 - 4k(k - 1)/2 \\ &= h^2 + 4h(k - 1) + 2(k - 1)(k - 2) \end{aligned}$$

and

$$\langle O_{hk} \rangle = 4(h - 1) + 4(k - 1)$$

In particular, for a diamond ($h = 1$), square ($k = 1$), and equal-sided octagon, we obtain the results shown in Table 2.1.

TABLE 2.1

Case	Area	Perimeter	(Perimeter) ² /area	Limit (for large figures)
$h = 1$	h^2	$4(h - 1)$	$16(h - 1)^2/h^2$	16
$k = 1$	$2k(k - 1) + 1$	$4(k - 1)$	$16(k - 1)^2/[2k(k - 1) + 1]$	8
$h = k$	$7h^2 - 10h + 4$	$8(h - 1)$	$64(h - 1)^2/[7h^2 - 10h + 4]$	$64/7 \doteq 9.14$

Thus (perimeter)²/area is smaller for a diamond than for either a square or an equal-sided octagon; the latter is certainly more circlelike than the diamond. In fact, for large digital approximations to circles, it can be verified empirically that the ratio is about 10.2, which is even worse than for the equal-sided octagons. It would seem that this ratio is not as good as a measure of noncompactness in digital geometry as it is in ordinary geometry. A characterization of S s for which the ratio is "minimal" can be found in [9]. We shall return to the topic of perimeter in the next section.

The *genus* $g(s)$ of $S \subseteq \Sigma$ can be defined as the number of S 's components minus the number of its holes. Thus $g(S) + g(\bar{S}) = 2$ for any S , where we are still using opposite types of connectedness for S and $\bar{S}$.

Let $v = |S|$ be the number of points in S ; let e the number of horizontally or vertically adjacent pairs of points in S $\left(\begin{smallmatrix} PP \\ P \end{smallmatrix}\right)$; and let f be the number of 2×2 blocks of points in S $\left(\begin{smallmatrix} PP \\ PP \end{smallmatrix}\right)$. Then we can prove

PROPOSITION 2.7.1 If we use 4-connectedness for S , then $g(S) = v - e + f$.

Proof: This is trivial if $|S| = 1$; we proceed by induction. Let P be the leftmost of the uppermost points of S , so that P 's neighborhood looks like

$$\begin{array}{ccc} O & O & O \\ O & P & P' \\ & P'' & Q \end{array}$$

where the O s are in $\bar{S}$. If P' and P'' are in $\bar{S}$, then P is an isolated point of S ; deleting P reduces v by 1, changes neither e nor f , and reduces g by 1. If P' is in S and P'' is in $\bar{S}$ or vice versa, deleting P

reduces v and e by 1 and does not change f ; nor does it change g since P is simple. If P', P'', Q are all in S , deleting P reduces v and f by 1 and e by 2, and leaves g unchanged since P is simple. Finally, if $P', P'' \in S$ but $Q \notin S$, deleting P reduces v by 1, e by 2, and does not change f ; it also increases g by 1 since P is nonsimple, so that its deletion either adds a component to S or subtracts one from $\hat{S}$ (see Section 2.5). Thus in all cases, if the proposition is true for $S - \{P\}$, it remains true for S . $\square$

Let d be the number of diagonally adjacent pairs of points in S $\binom{P}{P \text{ or } P}$, and t the number of right-triangle triples $\binom{P}{PP, PP, P \text{ or } PP}$. Then, analogously to Proposition 2.7.1, we can prove

PROPOSITION 2.7.2 If we use 8-connectedness for S , then $g(S) = v - e - d + t - f$.

Another approach to computing the genus can be found in [10].

EXERCISE 2.7.1 Let ρ be a run of points of S on some row of Σ , and let $n(\rho)$ be the number of runs on the row above that are adjacent to ρ . Then $g(S) = \sum_{\rho} (1 - n(\rho))$.

2.8 DISTANCE AND DIAMETER

We could define the distance between two points of Σ in the ordinary Euclidean way; but it seems more natural to use a discrete definition. For all $P = (x, y)$ and $Q = (u, v)$ in Σ , let us define the *city block distance* $d_4(P, Q)$ as $|x - u| + |y - v|$, and the *chessboard distance* $d_8(P, Q)$ as $\max(|x - u|, |y - v|)$. Note that the set of points within a given city block distance of P is just a diamond centered at P , while the points within a given chessboard distance of P form a square centered at P .[§] In particular, the points

[§]One can also define metrics for which the "disks" centered at P are octagons; for example [7],

$$\max(|x - u|, |y - v|, 2(|x - u| + |y - v| + 1)/3]$$

is such a metric.

Q for which $d_4(P, Q) = 1$ are just the 4-neighbors of P , while the Q s such that $d_8(P, Q) = 1$ are just P 's 8-neighbors.

It is readily verified that d_4 and d_8 are metrics on Σ —i.e., they are positive definite, symmetric, and satisfy the triangle inequality. They also have the property that for any $P \neq Q$ there exists an R such that $d(P, Q) = d(P, R) + d(R, Q)$, where both terms on the right are nonzero.

PROPOSITION 2.8.1 $d_4(P, Q)$ is just the length of a shortest 4-path in Σ from P to Q , and $d_8(P, Q)$ is just the length of a shortest such 8-path.

Proof: If the length is 1, i.e., the path is P, Q , then P and Q are 4-neighbors, so $d_4(P, Q) = 1$. We proceed by induction on the shortest path length. Let $P = P_0, P_1, \dots, P_n = Q$ be a shortest path; then it is easily seen that $P_0, P_1, \dots, P_{n-1}$ is a shortest path from P to P_{n-1} , so that $d_4(P, P_{n-1}) = n - 1$ by induction hypothesis. Since Q is a neighbor of P_{n-1} , we have $d_4(P_{n-1}, Q) = 1$; thus by the triangle inequality, $d_4(P, Q) \leq (n - 1) + 1 = n$. Conversely, if $d_4(P, Q) = m < n$, we can easily construct a path of length m from P to Q . For example, suppose that $P = (i, j)$ and $Q = (h, k)$, where $i \leq h$ and $j \leq k$ (the argument is similar if $i \geq h$ and/or $j \geq k$). Since $d_4(P, Q) = m$, we have $|i - h| + |j - k| = (i + h) + (j - k) = m$. But then, e.g., we have paths of length m from P to Q such as

$$\begin{aligned} P_0 &= P, & P_1 &= (i + 1, j), & P_2 &= (i + 2, j), & \dots, \\ & & P_{i-h} &= (h, j), & P_{i-h+1} &= (h, j + 1), \\ & & P_{i-h+2} &= (h, j + 2), & \dots, & P_{i-h+j-k} &= (h, k) = Q \end{aligned}$$

This contradicts the assumption that a shortest path from P to Q has length n . The proof for d_8 is analogous. $\square$

Let $P = P_0, P_1, \dots, P_n = Q$ be a shortest path from P to Q ; then for all $0 \leq i \leq j \leq n$, $P_i, \dots, P_j$ is a shortest path from P_i to P_j , and the distance from P_i to P_j is just $j - i$ (where we use 4-paths for d_4 and 8-paths for d_8).

As a simple application of the digital concept of distance, we use it to give a simple proof (based on [11]) that any connected $S \subseteq \Sigma$ contains a point whose deletion does not disconnect S (compare Proposition 2.5.4). Pick any $P \in S$ and let Q be a point of S whose distance from P is a local maximum—i.e., no neighbor R of Q that

belongs to S is farther from P than Q is. [Here "connected," "distance," and "neighbor" should all be used in the same (4- or 8-) sense.] Then $S - \{Q\}$ is still connected. Indeed, it suffices to show that for all $P' \in S - \{Q\}$ there exists a path in $S - \{Q\}$ from P to P' . (If $S - \{Q\}$ is empty, we have $S = \{Q\} = \{P\}$ and there is nothing to prove.) Let π be a path from P to P' in S ; if Q is not on π , we are done. Otherwise, let Q' be the point of π just after the last occurrence of Q , so that Q' is a neighbor of Q . Since Q' is no farther from P than Q is, a shortest path π' from P to Q' (in S) cannot contain Q , by the preceding paragraph. Thus if we adjoin to π' the part of π between Q' and P' , we obtain a path from P to P' in $S - \{Q\}$.

We can also define a concept of distance intrinsic to a given set S . Let the S -distance $d_S(P, Q)$ between two points P, Q of S be the length of the shortest path in S between P and Q , where we use the same type of path as for S 's connectedness. Clearly, $d_S(P, Q)$ cannot be less than the ordinary (extrinsic) distance (of the same type) from P to Q .

We conclude by establishing a simple relationship between the perimeter of a connected set S and its "intrinsic diameter" [12]. Let $p(S)$ be the number of steps taken by the BF algorithm in following all the borders of S . [This is a measure of the total perimeter of S , but it is not the same as the number $\langle S \rangle$ of border points of S , since different borders (where S is adjacent to different components of $\bar{S}$) may have points in common.] Let $d(S)$ be the greatest possible S -distance between two points of S ; we call such a pair of points *diametrically opposite*. Then we can prove

THEOREM 2.8.2 $d(S) \leq p(S)/2$.

Proof: We first show that this is true when S is simply-connected. It is trivial when $|S| \leq 2$; we proceed by induction. Let P, Q be diametrically opposite points of S . If these are S 's only simple points, then by Theorem 2.5.5, S is an arc. But it is not hard to see that for an arc we have $p(S) = 2|S| - 1$ and $d(S) = |S| - 1 \leq p(S)/2$, as desired. Otherwise, S has an end or corner R other than P and Q . Now deleting R from S cannot change the S -distance between any other two points, since a path through an end cannot be a shortest path, while a path through a corner can be diverted to avoid the corner without changing its length. Hence $d(S - \{R\}) = d(S) = d_S(P, Q)$. On the other hand,

$p(S - \{R\}) \leq p(S)$, since deleting an end shortens the perimeter, while deleting a corner does not change it. We thus have, using the induction hypothesis, $d(S) = d(S - \{R\}) \leq p(S - \{R\})/2 \leq p(S)/2$, completing the proof in the simply-connected case.

We now use induction on the number of holes in S . For any such hole W , we have $p(S) = p(S \cup W) + |S_W|$. (We recall that S_W denotes the set of points of S adjacent to W .) Let P, Q be diametrically opposite points of S , π be a shortest path between P and Q in $S \cup W$, and P', Q' be the first and last points in which π meets S_W . Clearly, there exists a path π' of length $\leq |S_W|/2$ from P' to Q' in S (in S_W , in fact) that we can construct by following the W -border of S the shorter way around. If we replace the part of π between P' and Q' by π' , we obtain a path π'' from P to Q in S whose length $\ell(\pi'')$ is at most $|S_W|/2$ greater than that of π . Hence $d(S) \leq \ell(\pi'') \leq \ell(\pi) + |S_W|/2$. But by the induction hypothesis, we know that $\ell(\pi) = d_{S \cup W}(P, Q) \leq d(S \cup W) \leq p(S \cup W)/2$. Thus $d(S) \leq p(S \cup W)/2 + |S_W|/2 = p(S)/2$, which completes the proof. $\square$

2.9 GEODESICS

A (4- or 8-) path is called a *geodesic* if no shorter path with the same endpoints exists. It is evident that a geodesic is an arc (see Exercise 2.3.2), and that any subarc of a geodesic is a geodesic. On the other hand, a geodesic need not resemble a straight line segment; for example,

$$\begin{array}{c} P \ P \ P \\ P \\ P \end{array}$$

is a 4-geodesic. In this section we give several characteristics of 4- and 8-geodesics [13].

PROPOSITION 2.9.1 The following properties of a 4-arc $\alpha = P_0, P_1, \dots, P_n = (x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ are equivalent:

- (a) α is a 4-geodesic;
- (b) α cannot turn right (or left) twice in succession, i.e., left and right turns must alternate;

- (c) $x_0 \leq x_1 \leq \cdots \leq x_n$ (or all $\geq$), and $y_0 \leq y_1 \leq \cdots \leq y_n$ (or all $\geq$);
- (d) $|x_0 - x_n| + |y_0 - y_n| = n$ —i.e., $d_4(P_0, P_n) = n$.

Proof: To see that (a) implies (b), suppose that α made two successive turns in the same direction, e.g.,

$$\begin{array}{ccc} P_{r+1} & P_{r+2} & \cdots & P_{s-1} \\ P_r & & & P_s \end{array} \cdots$$

(the argument for other cases is exactly analogous). Then the path along α from P_r to P_s has length $s - r$; but there is a horizontal path from P_r to P_s of length $s - r - 2$, so the subarc $P_r, \dots, P_s$ is not a geodesic, and thus neither is α .

We next show that (b) implies (c). Suppose that the initial direction of α is horizontal toward the right, and that the first turn (if any) is a left turn (the proofs in other cases are analogous). Let the turns be at $P_{n_1}, P_{n_2}, \dots$, where $0 < n_1 < n_2 < \cdots < n$. Then $x_0 < x_1 < \cdots < x_{n_1}$ and $y_0 = y_1 = \cdots = y_{n_1}$. After the first turn, α is headed vertically upward; thus $x_{n_1} = x_{n_1+1} = \cdots = x_{n_2}$ and $y_{n_1} < y_{n_1+1} < \cdots < y_{n_2}$. By (b), the second turn must be a right turn, after which α is again horizontal and headed rightward, so that $x_{n_2} < x_{n_2+1} < \cdots < x_{n_3}$ and $y_{n_2} = y_{n_2+1} = \cdots = y_{n_3}$; and so on, proving (c). Note that (c) means that if we take any point P_m of the arc as origin, then the subarcs $P_0, \dots, P_m$ and $P_m, \dots, P_n$ lie in a pair of opposite quadrants.

Next, we prove that (c) implies (d). At each step along a 4-arc, either x or y (but not both) changes by 1. Hence if (c) holds, say $x_0 \leq x_1 \leq \cdots \leq x_n$ and $y_0 \leq y_1 \leq \cdots \leq y_n$ (and similarly in the other cases), then the number of steps at which the x s increase and the number at which the y s increase add up to n , which implies (d).

Finally, we show that (d) implies (a). Any path from (x_0, y_0) to (x_n, y_n) must have length at least $|x_0 - x_n| + |y_0 - y_n|$, since a coordinate can only change by 1 at each step, and only one coordinate can change at a time. If (d) holds, this bound is n , and since α has length n , its length is the shortest possible, proving (a). $\square$

The situation in the 8-arc case is somewhat more complicated. An 8-arc can turn by either 45° or 90° , e.g.,

$$\begin{array}{ccc} & P & \\ & P & \\ P & P & P \end{array} \quad \text{or} \quad \begin{array}{ccc} & P & \\ & P & P \\ P & & P \end{array}$$

An 8-geodesic cannot make two successive right or left 90° turns; nor can it make two successive right or left 45° turns if they take it from horizontal to vertical, as in the arc

$$\begin{array}{c} P \\ P \\ P \\ P \\ P & P & P \end{array}$$

in which there is a shorter path between the endpoints (namely, a 45° straight line). However, an 8-geodesic *can* make two successive right or left 45° turns which take it from one diagonal direction to another, e.g.,

$$\begin{array}{ccc} & P & P & P \\ & P & & P \\ P & & & P \end{array}$$

In spite of this greater complexity, we can still prove an analog of Proposition 2.9.1:

PROPOSITION 2.9.2 The following properties of an 8-arc $\alpha = P_0, P_1, \dots, P_n$ are equivalent:

- (a) α is an 8-geodesic;
- (b) $x_0 < x_1 < \dots < x_n$ (or all $>$), or $y_0 < y_1 < \dots < y_n$ (or all $>$);
- (c) $|x_0 - x_n| = n$ or $|y_0 - y_n| = n$.[§]

Note that (b) can be interpreted as follows: If we take any point P_m of the arc as origin, then the subarcs $P_0, \dots, P_m$ and $P_m, \dots, P_n$,

[§]Since $|x_0 - x_n|$ and $|y_0 - y_n|$ must each be $\leq n$ (each coordinate can change by at most 1 at each step), this is equivalent to $\max(|x_0 - x_n|, |y_0 - y_n|) = n$; i.e., to $d_8(P_0, P_n) = n$.

P_n lie in a pair of opposite diagonal quadrants (quadrants bounded by the lines at $\pm 45^\circ$ through the origin).

Proof: The x or y coordinates can each change by at most 1 from point to point along an 8-arc. Hence $|x_0 - x_n| = n$ means that successive x_i s do differ by 1, and all in the same direction, in order to achieve a total difference of n ; thus $x_0 < x_1 < \dots < x_n$ (or all $>$). [In fact, we must have $x_i = x_0 + i$ for all i ($0 \leq i \leq n$), or $x_i = x_0 - i$ for all such i .] This proves that (c) implies (b). Conversely, $x_0 < x_1 < \dots < x_n$ means that successive x_i s differ by 1 in the same direction, so that $|x_0 - x_n| = n$; thus (b) implies (c).

On the other hand, since α has length n , we must have $|x_0 - x_n| \leq n$ and $|y_0 - y_n| \leq n$; an arc of length n cannot involve coordinate changes of more than n , since each coordinate changes by at most 1 per step. Now any 8-path from P_0 to P_n must have a length of at least $\max(|x_0 - x_n|, |y_0 - y_n|)$. If (c) holds, say $|x_0 - x_n| = n$, the maximum is n ; thus α , which has length n , is a shortest path, proving (a).

Conversely, if (c) fails, we have $|x_0 - x_n| < n$ and $|y_0 - y_n| < n$. Suppose that $x_0 \leq x_n$ and $y_0 \leq y_n$ and that $x_n - x_0 \leq y_n - y_0$ (the argument is analogous if any of these $\leq$ s is $\geq$). Then (x_0, y_0) , $(x_0 + 1, y_0 + 1)$, $\dots$, $(x_n, y_0 + (x_n - x_0))$, $(x_n, y_0 + (x_n - x_0) + 1)$, $\dots$, (x_n, y_n) is a path from (x_0, y_0) to (x_n, y_n) of length $y_n - y_0 < n$; thus α is not a shortest path, completing the proof. $\square$

For any digital arc $P_0, \dots, P_n$, we can define its *chord length* as the distance between its endpoints. In this definition, we shall use 4-distance for 4-arcs and 8-distance for 8-arcs. Thus the last parts of Propositions 2.9.1 and 2.9.2 can be restated as

PROPOSITION 2.9.3 An arc is a geodesic iff its arc length is equal to its chord length.

We can also give another characterization of geodesics in terms of arc and chord lengths as follows:

PROPOSITION 2.9.4 The following statements about the digital arc α are equivalent:

- (a) α is a geodesic;

- (b) Two subarcs of α have equal length iff their chords have equal length;
- (c) Equal chords (of subarcs of α) have equal arcs.

[Note that in the Euclidean plane, condition (b) is satisfied not only by straight line segments, but by arcs of circles. It has been shown by Herda that if S is a convex plane object satisfying the condition that equal arcs of S 's boundary have equal chords, then S must be a circle. Thus it is somewhat surprising that in a digital picture, condition (b) is equivalent to being a geodesic.]

Proof: If α is a geodesic, then as we have seen, every subarc of α is also a geodesic. Hence by Proposition 2.9.3, every subarc of α has arc length equal to its chord length, which certainly implies (b). Clearly, (b) implies (c).

Conversely, suppose α is not a geodesic; then the distance between the endpoints of α is less than its length n . Now α has subarcs of lengths $1, 2, \dots, n$, and their chord lengths cannot exceed their arc lengths; thus the possible chord lengths are all less than n (since the only subarc of length n , namely α itself, does not have chord length n). If (c) holds, unequal arcs must have unequal chords; but there are fewer than n possible chord lengths to assign to the n different arc lengths—a contradiction. $\square$

Interestingly, the other half of condition (b), i.e., the assertion that equal subarcs of α have equal chords, is *not* sufficient to make α a geodesic. For example, consider the 4-arc

$$\begin{array}{ccccccc} P_1 & P_2 & \cdots & P_{n-1} \\ & P_0 & & P_n \end{array}$$

This is certainly not a geodesic, but it can be verified that its equal subarcs have equal chords. However, it turns out that arcs with this property must “almost” be geodesics, as we shall see shortly.

Our results here regarding arcs and chords are all based on measuring chord length in terms of city block distance for 4-arcs, and chessboard distance for 8-arcs. What if we used Euclidean distance? It turns out that this would be very restrictive. In fact, suppose that for every subarc α' of α , the Euclidean chord length of α' is equal to its arc length (i.e., to one less than the number of points in α'). In particular, taking $\alpha' = P_i, P_{i+1}$ to have length 1, we see that P_i and P_{i+1} cannot be diagonal neighbors, since for such

neighbors the Euclidean chord length is $\sqrt{2}$. Similarly, taking $\alpha' = P_{i-1}, P_i, P_{i+1}$ to have length 2, we see that α cannot make any turns, since at a turn, the Euclidean distance between P_{i-1} and P_{i+1} is $\sqrt{2}$. Thus α can have no diagonal steps or turns, and so must be a horizontal or vertical straight line segment.[§]

If we require only that the Euclidean chord length of α itself be equal to its arc length, we obtain a more interesting result. This requirement can be written as

$$\sqrt{(x_0 - x_n)^2 + (y_0 - y_n)^2} = n$$

so that $|x_0 - x_n|$, $|y_0 - y_n|$, and n are a Pythagorean triple of non-negative integers. It is well known that all such Pythagorean triples (a, b, c) , where $a^2 + b^2 = c^2$, are of the form

$$a = k(u^2 - v^2), \quad b = 2kuv, \quad c = k(u^2 + v^2)$$

(or the same with a and b interchanged) for some positive integers k, u , and v , $u \geq v$. For example, the triple $(a, b, c) = (3, 4, 5)$ gives us 8-arcs such as

$$\begin{array}{c} P P \\ P \\ P \\ P \\ P \end{array}$$

Note that this arc is not an 8-geodesic.

On the other hand, suppose that α is a 4-arc satisfying the Pythagorean condition. The city block chord length of a 4-arc cannot exceed the arc length, since each step along the arc contributes at most 1 to the chord length. Hence we must have

$$|x_0 - x_n| + |y_0 - y_n| \leq n$$

If these numbers are a Pythagorean triple, we can write this condition (canceling k) as

$$u^2 - v^2 + 2uv \leq u^2 + v^2$$

which is equivalent to

$$2uv \leq 2v^2 \quad \text{or} \quad u \leq v$$

[§] If we instead measure arc length as the number of horizontal and vertical steps plus $\sqrt{2}$ times the number of diagonal steps, α can still make no turns and hence must be a horizontal, vertical, or diagonal line segment.

But since $u \geq v$, this implies $u = v$, so the triple is degenerate: It is of the form $(0, c, c)$ or $(c, 0, c)$. This means that $|x_0 - x_n| = 0$ and $|y_0 - y_n| = n$, or vice versa, so the arc is a horizontal or vertical line segment.

If the digital arc α has the property that every proper subarc of α is a geodesic, we call it a *near-geodesic*.

PROPOSITION 2.9.5 The following statements about the digital arc α are equivalent:

- (a) α is a near-geodesic;
- (b) if either endpoint of α is deleted, it becomes a geodesic;
- (c) equal subarcs of α have equal chords.

Proof: The equivalence of (a) and (b) is straightforward. Also, if (a) holds, then all proper subarcs of α are equal in length to their chords, so equal proper subarcs have equal chords; also, there is only one improper subarc (α itself), so (c) holds. We must now show that (c) implies (a).

We first give the proof for 4-arcs. Clearly, all arcs of length 1 are equal to their chords. Let the shortest proper subarc not equal to its chord have length k (if none such exists, we are done). Let $\alpha' \equiv P_0, \dots, P_k$ be such a subarc; thus $P_0, \dots, P_{k-1}$ is a 4-geodesic, and we have, e.g.,

$$x_0 \leq x_1 \leq \dots \leq x_{k-1} \quad \text{and} \quad y_0 \leq y_1 \leq \dots \leq y_{k-1}$$

but $x_{k-1} > x_k$ or $y_{k-1} > y_k$, i.e., $x_k = x_{k-1} - 1$ (and $y_{k-1} = y_k$) or $y_k = y_{k-1} - 1$ (and $x_{k-1} = x_k$). Since α' is proper, the original arc has another point P_{k+1} (if necessary, we can take α' in reverse order), which is a 4-neighbor of P_k , so that $x_{k+1} = x_k \pm 1$ (and $y_{k+1} = y_k$) or $y_{k+1} = y_k \pm 1$ (and $x_{k+1} = x_k$).

The city block distance $|x_0 - x_{k-1}| + |y_0 - y_{k-1}|$ from P_0 to P_{k-1} is $k - 1$; hence $d_4(P_0, P_k) = k - 2$. If $x_{k+1} = x_k - 1$ or $y_{k+1} = y_k - 1$, we have $d_4(P_0, P_{k+1}) = k - 3$, so that $d_4(P_1, P_{k+1}) \leq k - 4$. Thus the equal arcs $P_0, \dots, P_k$ and $P_1, \dots, P_{k+1}$ have unequal chords—a contradiction. Hence we must have $x_{k+1} = x_k + 1$ or $y_{k+1} = y_k + 1$. Clearly, if $x_k = x_{k-1} - 1$, we cannot have $x_{k+1} = x_k + 1$, since then (the y s being equal) we would have $P_{k+1} = P_{k-1}$; and similarly with the roles of the x s and y s interchanged. Thus there are only two possible cases:

- (1) $x_k = x_{k-1} - 1, y_k = y_{k-1}; x_{k+1} = x_k, y_{k+1} = y_k + 1$
- (2) $x_k = x_{k-1}, y_k = y_{k-1} - 1; x_{k+1} = x_k + 1, y_{k+1} = y_k$

In case (1), x_k must be equal to one of the earlier x s. To see this, recall that for $0 < i \leq k-1$ we have either $x_i = x_{i-1} + 1$ or $x_i = x_{i-1}$. Thus if $x_0 < x_{k+1}$, the sequence of x s must pass through the value $x_{k-1} - 1$ at some point. But if $x_0 = x_{k-1}$, the points $P_0, \dots, P_{k-1}$ are all collinear, and $P_0, \dots, P_k$ would be a geodesic, by Proposition 2.9.1. Let x_j be the last of the x s that is equal to $x_{k-1} - 1$; thus $0 \leq j < k-1$, and we have $x_j < x_{j+1} = \dots = x_{k-1}$. This implies that $y_j = y_{j+1} < \dots < y_{k-1}$, so that the $(k-j)$ -point subarc $P_j, \dots, P_k$ has chord length $(k-1) - (j+1) = k-j-2$. If $j > 0$, this arc has length less than k , which contradicts our assumption that the shortest proper subarc not equal to its chord has length k . It remains only to consider the case $j = 0$, where we have

$$\begin{aligned} x_0 &< x_1 = x_2 = \dots = x_{k-1} \\ y_0 &= y_1 < y_2 < \dots < y_{k-1} \end{aligned}$$

But in this case the city block distance from P_1 to P_{k+1} is $|x_{k+1} - x_1| + |y_{k+1} - y_1| = 1 + (k-1) = k$. Thus the equal arcs $P_0, \dots, P_k$ and $P_1, \dots, P_{k+1}$ have unequal chords—a contradiction. The proof in case (2) is analogous, as are the proofs for the other orderings of the x s and y s.

In the 8-arc case, let $\alpha' = P_0, \dots, P_k$ be defined analogously; thus $P_0, \dots, P_{k-1}$ is an 8-geodesic, so that, e.g.,

$$x_0 < x_1 < \dots < x_{k-1} \quad \text{but} \quad x_k \leq x_{k-1}$$

Suppose it were not true that $y_1 < y_2 < \dots < y_{k-1} < y_k$; i.e., suppose we had $|y_j - y_k| < |j - k|$ for some $1 \leq j \leq k$. Then $d_8(P_j, P_k) = \max(|x_j - x_k|, |y_j - y_k|) = \max[(k-j-1), |y_j - y_k|] < k-j$, contradicting the fact that the subarc $P_j, \dots, P_k$ has length $< k$ and so must be a geodesic by our choice of k . Thus we must have

$$y_1 < y_2 < \dots < y_k \quad \text{but} \quad y_0 \geq y_1$$

where the last inequality must hold because otherwise $P_0, \dots, P_k$ would be a geodesic. It follows that $d_8(P_0, P_k)$ is either $k-2$ (if $x_k < x_{k-1}$ and $y_0 > y_1$) or $k-1$ (otherwise).

As in the 4-case, α has another point P_{k+1} , and $d_8(P_1, P_{k+1})$ must be the same as $d_8(P_0, P_k)$. Suppose first that $x_k < x_{k-1}$ and $y_0 < y_1$, so that this distance is $k-2$. Then we cannot have $y_{k+1} \geq y_k$, since this would give us $y_{k+1} - y_1 > k-2$; thus $y_{k+1} = y_k - 1$. If $x_{k+1} = x_k + 1$, we would have $P_{k+1} = P_{k-1}$; if

$x_{k+1} = x_k$, P_{k+1} would be a neighbor of P_{k-1} ; and if $x_{k+1} = x_k - 1$, P_{k+1} would be a neighbor of P_{k-2} . Hence this case is impossible.

Next, suppose $x_k < x_{k-1}$ and $y_0 = y_1$, so that $d_s(P_0, P_k) = k - 1$. Here we cannot have $y_{k+1} > y_k$, since this would make $y_{k+1} - y_1 = k$, and we cannot have $y_{k+1} < y_k$ just as in the preceding paragraph; thus $y_{k+1} = y_k$. Here $x_{k+1} = x_k$ is impossible since $P_{k+1} \neq P_k$, and $x_{k+1} = x_k + 1$ is impossible since P_{k+1} is not a neighbor of P_{k-1} ; thus $x_{k+1} = x_k - 1$. But it is easily seen that this implies that $d_s(P_{k+1}, P_{k-2}) = 2$, contradicting our choice of k .

Finally, suppose that $x_k = x_{k-1}$. Here again we cannot have $y_{k+1} > y_k$, and we also cannot have $y_{k+1} \leq y_k$ since this would make P_{k+1} a neighbor of (or the same as) P_{k-1} . Thus this case too is impossible, and the proof is complete. $\square$

2.10 CONCLUDING REMARKS

Many other topics could have been included in this chapter; among these are digital "homotopy" [14], "dimension" [15], "elongatedness" [7], "straightness" [16–21], and "convexity" [22–29], as well as a variety of algorithms for connected component tracking and counting [1, 30, 31], adjacency tree construction [4], and border finding and following [1, 4, 32]. The interested reader is encouraged to investigate these topics and to study the capabilities of automata in recognizing these properties or implementing these algorithms.

REFERENCES

1. A. Rosenfeld and A. C. Kak, "Digital Picture Processing." New York, Academic Press, 1976.
2. R. O. Duda and P. E. Hart, "Pattern Classification and Scene Analysis." Wiley, New York, 1973.
3. A. Rosenfeld, Connectivity in digital pictures, *J. Assoc. Comput. Mach.* **17**, 146–160 (1970).
4. A. Rosenfeld, Adjacency in digital pictures, *Information and Control* **26**, 24–33 (1974).
5. A. Rosenfeld, Arcs and curves in digital pictures, *J. Assoc. Comput. Mach.* **20**, 81–87 (1973).
6. A. Rosenfeld, A converse to the Jordan Curve Theorem for digital curves, *Information and Control* **29**, 292–293 (1975).

7. A. Rosenfeld and J. L. Pfaltz, Distance functions on digital pictures, *Pattern Recognition* **1**, 33–61 (1968).
8. F. Wyse et al., Solution to Problem 5712, *Amer. Math. Monthly* **77**, 1119 (1970).
9. A. Rosenfeld, Compact figures in digital pictures, *IEEE Trans. Systems, Man, Cybernetics* **4**, 221–223 (1974).
10. S. B. Gray, Local properties of binary images in two dimensions, *IEEE Trans. Computers* **20**, 551–561 (1971).
11. N. Felsing, Solution to Problem E2018, *Amer. Math. Monthly* **76**, 827–828 (1969).
12. A. Rosenfeld, A note on perimeter and diameter in digital pictures, *Information and Control* **24**, 384–388 (1974).
13. A. Rosenfeld, Geodesics in digital pictures, *Information and Control* **36**, 74–84 (1978).
14. J. C. Alexander and A. Thaler, The boundary count of digital pictures, *J. Assoc. Comput. Mach.* **18**, 105–112 (1971).
15. J. P. Mylopoulos and T. Pavlidis, On the topological properties of quantized spaces, *J. Assoc. Comput. Mach.* **18**, 239–254 (1971).
16. R. Brons, Linguistic methods for the description of a straight line on a grid, *Computer Graphics and Image Processing* **3**, 48–62 (1974).
17. A. Rosenfeld, Digital straight line segments, *IEEE Trans. Computers* **23**, 1264–1269 (1974).
18. G. Bongiovanni, F. Luccio, and A. Zorat, The discrete equation of the straight line, *IEEE Trans. Computers* **24**, 310–313 (1975).
19. H. Klaasman, Some aspects of the accuracy of the approximated position of a straight line on a square grid, *Computer Graphics and Image Processing* **4**, 225–235 (1975).
20. C. Arcelli and A. Massarotti, Regular arcs in digital contours, *Computer Graphics and Image Processing* **4**, 339–360 (1975).
21. J. Rothstein and C. Weiman, Parallel and sequential specification of a context sensitive language for straight lines on grids, *Computer Graphics and Image Processing* **5**, 106–124 (1976).
22. J. Sklansky, Recognition of convex blobs, *Pattern Recognition* **2**, 3–10 (1970).
23. L. Hodes, Discrete approximation of continuous convex blobs, *SIAM J. Appl. Math.* **19**, 477–485 (1970).
24. J. Sklansky, R. L. Chazin, and B. J. Hansen, Minimum-perimeter polygons of digitized silhouettes, *IEEE Trans. Computers* **21**, 1233–1239 (1972).
25. J. Sklansky, Measuring concavity on a rectangular mosaic, *IEEE Trans. Computers* **21**, 1355–1364 (1972).
26. C. Arcelli and L. Cordella, Concavity point detection by iterative arrays, *Computer Graphics and Image Processing* **3**, 34–47 (1974).
27. D. Rutovitz, An algorithm for in-line generation of a convex cover, *Computer Graphics and Image Processing* **4**, 74–78 (1975).
28. J. Sklansky, On filling cellular concavities, *Computer Graphics and Image Processing* **4**, 236–237 (1975).
29. J. Sklansky, L. P. Cordella, and S. Levialdi, Parallel detection of concavities in cellular blobs, *IEEE Trans. Computers* **25**, 187–196 (1976).
30. C. B. Shelman, The application of list processing techniques to picture processing, *Pattern Recognition* **4**, 201–210 (1972).

31. S. Yokoi, J. Toriwaki, and T. Fukumura, An analysis of topological properties of digitized binary pictures using local features, *Computer Graphics and Image Processing* **4**, 63–73 (1975).
32. G. S. Sidhu and R. T. Boute, Property and encoding: application in binary picture encoding and boundary following, *IEEE Trans. Computers* **21**, 904–911 (1972).

Chapter 3

STRING ACCEPTORS

3.1 INTRODUCTION

The purpose of this and the following chapters is to define various types of formal machines—automata—that can recognize given classes of pictures. Before defining these machines for two-dimensional arrays (i.e., digital pictures), we devote this chapter to reviewing the analogous concepts in one dimension, where the machines recognize given classes of strings or “tapes.”

In order to make the analogy between the one- and two-dimensional cases more explicit, we have somewhat modified the usual definitions of the one-dimensional machines; but our definitions are readily equivalent to the usual ones. We also present a collection of propositions and theorems about one-dimensional machines; many of these are standard, while others are given as preparation for presenting their two-dimensional analogs. We give informal proofs of many of these results, but occasionally state them without proof and give references to standard texts in which proofs can be found.

The computational models described in this chapter are not all of equal utility; some of them are potentially of practical interest (e.g., bounded cellular automata), while others are much less so. However, even the impractical models are pedagogically useful, since they provide a standard framework for breaking computations down into small steps.

3.2 AUTOMATA AND ACCEPTORS

3.2.1 Transition Functions and Configurations

Informally, a (one-dimensional) *automaton* can be thought of as a "bug" that, at any given time, is in one of a set of *states* and is located at a given position on a *tape* that contains a sequence of *symbols*. The bug operates in a sequence of discrete time steps, at each of which it

- (a) reads the symbol in its current position, erases that symbol, and replaces it by a (possibly) new one;
- (b) changes to a (possibly) new state; and
- (c) moves to a neighboring position, or possibly does not move.

In general, the new state that the bug goes into, the new symbol that it writes, and the direction in which it moves all depend on its current state, on the current symbol, and on the direction in which the bug last moved. (The reason for making them dependent on the direction of last movement will be explained later.) Thus the bug's behavior is described by a *transition function* that maps (state, symbol, direction) triples into (state, symbol, direction) triples. We call the bug *deterministic* if each triple maps into a unique triple; otherwise, we call it *nondeterministic* and regard it as mapping triples into sets of triples.

On a more formal level, an *automaton* M is defined by specifying a triple (Q, V, δ) , where

Q is the set of *states*,

V is the set of *symbols* ("vocabulary"),

$\delta: Q \times V \times \Delta \rightarrow 2^{Q \times V \times \Delta}$ (or $\rightarrow Q \times V \times \Delta$, in the deterministic case) is the *transition function*, and

$\Delta \equiv \{L, R, N\}$ is the set of *move directions* ("left," "right," and "no move").

A *tape* is a mapping $\tau: I \rightarrow V$ from the integers into the vocabulary. A *configuration* of the automaton (and tape) is a quadruple (q, d, i, τ) , where q is M 's current state, d is the direction in which M has just moved, i is an integer denoting M 's position, and τ is the current tape. The transition function δ defines a mapping between configurations in the following way: Let $\tau_i \in V$ be the symbol in the i th position on τ . Let (q', v', d') be any triple in the image of $(q,$

τ_i, d) under δ , where d and d' are directions in δ ; in other words, suppose that when M is in state q , reads symbol τ_i , and has just moved in direction d , one of the possibilities for its behavior is that it rewrites τ_i as v' , moves in direction d' , and changes to state q' . Let τ' be the tape that is identical to τ except that τ_i has been changed to v' [i.e., $(\tau')_i = v'$]. Then the configuration (q', d', i', τ') is a possible successor to (q, d, i, τ) , where

$$\begin{aligned} i' &= i - 1, & \text{if } d' = L \\ &= i, & \text{if } d' = N \\ &= i + 1, & \text{if } d' = R \end{aligned}$$

This formalism corresponds to the intuitive description of M given in the first paragraph.

3.2.2 Finiteness Conditions

Up to now we have not imposed any restrictions on the sizes of the state set Q , the vocabulary V , or the tape τ . From now on we shall require that

- Q be a finite nonempty set containing a special "initial state" q_0 ;
- V be a finite nonempty set containing a special "blank symbol" $\#$; and
- τ be of the form $\#^\infty \sigma \#^\infty$, where σ is a finite nonnull string of non- $\#$ s.

We shall also require that the initial position of the automaton M be on a non- $\#$ symbol.

An M that satisfies these restrictions will be called a *Turing machine* (TM). We shall also consider three special types of TMs:

- (a) *#-preserving*: M neither creates nor destroys $\#$ s—in other words, for all $(q', v', d') \in \delta(q, v, d)$, we have $v = \#$ iff $v' = \#$.
- (b) *Tape-bounded*: M "bounces off" $\#$ s, so that it is essentially confined to the non- $\#$ portion of its input tape. Specifically, we require that if M has just moved in direction d and reads the symbol $\#$, it must not rewrite the $\#$ as anything else, and must move in direction $d' = d^{-1}$, where $L^{-1} = R$ and $R^{-1} = L$. (We cannot have had $d = N$, since initially M is on a non- $\#$, and it is easily shown by induction that whenever the move just made is N , the current symbol is

not $\#$.) Clearly, tape-bounded implies $\#$ -preserving. Note that if M is tape-bounded, we can require τ to be of the form $\#\sigma\#$, since M never visits any other part of τ .

- (c) *Finite-state*[§]: M never rewrites any symbols—i.e., $(q', v', d') \in \delta(q, v, d)$ implies $v' = v$. (In particular, such an M is $\#$ -preserving.)

The reason we required a transition of M to depend on the direction in which M has just moved was to allow us to define tape-boundedness as done here. If M 's current move did not depend on the previous move's direction, there would be no way to guarantee that M bounces off $\#$ s; M would not know which way to move when it reads a $\#$.

An alternative possibility for defining tape-boundedness would be to introduce two special " $\#$ " symbols, $\#_l$ and $\#_r$, and have $\#_l$ s to the left of the non- $\#$ segment of the tape and $\#_r$ s to its right. We could then stipulate that whenever M reads a $\#_l$ it moves right, and whenever it reads a $\#_r$ it moves left. This too would ensure that M bounces off the $\#$ s. To generalize it to rectangular arrays of non- $\#$ s in two dimensions (Section 4.3), we could introduce two further special " $\#$ " symbols, $\#_t$ and $\#_b$, at the top and bottom of the non- $\#$ s, and require that whenever M reads a $\#_t$ or $\#_b$ it moves down or up, respectively. However, this approach does not generalize to arbitrary (nonrectangular) two-dimensional arrays of non- $\#$ s (Section 3.4.4).

3.2.3 Acceptors and Their Languages

We now introduce the notion of "acceptance" of an input tape by an automaton. Formally, an *acceptor* A is a triple (M, q_0, Q_A) , where $M = (Q, V, \delta)$ is an automaton, $q_0 \in Q$ is M 's *initial state*, and $Q_A \subseteq Q$ is a set of special *accepting states*. We say that A accepts the tape τ_0 from position i_0 if repeated application of M 's transition function, starting from the configuration (q_0, N, i_0, τ_0) , leads to a

[§]A better term would be "non(re)writing," but we use "finite state" for historical reasons.

state set containing some $q \in Q_A$. If M is a Turing machine (and is $\#$ -preserving, tape-bounded, or finite-state), we call A a *Turing acceptor* (TA), *$\#$ -preserving acceptor* ($\#$ PA), *tape-bounded acceptor* (TBA), or *finite-state acceptor* (FSA), respectively.

As defined so far, acceptance depends on the initial position i_0 of A . In the following propositions, we show that this position can be standardized without changing the set of tapes that are accepted. From now on, A is a TA with initial tape $\#^\infty \sigma \#^\infty$, and we shall speak of A as accepting the string σ if it accepts the tape $\#^\infty \sigma \#^\infty$.

Let $L_l(A)$ be the set of strings σ that A accepts when its initial position is at the *left end* of σ . Let $L_\cup(A)$ be the set of σ s that A accepts from *some* initial position; in other words, for all $\sigma \in L_\cup(A)$, there exists an initial position (on a symbol in σ) from which A accepts σ . Let $L_\cap(A)$ be the set of σ s that A accepts from *any* initial position (on any symbol in σ).

PROPOSITION 3.2.1 For any A there exists an A' such that $L_\cap(A') = L_\cup(A') = L_l(A)$. Moreover, if A is $\#$ P, TB, or FS, so is A' , and if A is deterministic, so is A' .

Proof: Given A , we define A' as follows: From any initial position on σ , A' moves leftward, not rewriting any symbols, until it bounces off a $\#$. As soon as this happens, A' enters the initial state of A ; thereafter its behavior is identical to that of A . Since σ is not changed by the initial movements of A' , it is clear that A' accepts, from every initial position, exactly the same σ s that A accepts from their left ends. Moreover, the initial movements of A' do not violate $\#$ -preservation, tape-boundedness, or finite-statensness; hence if A has any of these properties, so has A' . Note also that if A is deterministic, so is A' . $\square$

PROPOSITION 3.2.2 For any A there exists an A'' such that $L_l(A'') = L_\cup(A)$. Moreover, if A is a $\#$ PA, TBA, or FSA, so is A'' .

Proof: Given A , we define A'' as follows: From its initial position at the left end of σ , A'' moves rightward, without rewriting any symbols. When it reads any non- $\#$ symbol, A'' can nondeterministically enter the initial state of A ; thereafter its behavior is identical to that of A . Since its initial behavior has no effect on σ , it is clear that A'' accepts σ iff there exists an initial position from

which A accepts σ . Clearly, if A is a $\#$ PA, TPA, or FSA, so is A'' , but note that A'' is nondeterministic even if A is deterministic. $\square$

PROPOSITION 3.2.3 For any A there exists an A''' such that $L(A''') = L_{\cap}(A)$. Moreover, if A is $\#$ P or TB, or is deterministic, so is A''' .

Proof: A''' rewrites each (non- $\#$) symbol x of σ as a pair (x, x) . Starting at the left end of σ , it then simulates A on the first terms of these pairs, leaving the second terms intact. If A rewrites a $\#$, say as y , the simulation creates a pair of the form $(y, \#)$. If the simulation accepts, A''' scans the non- $\#$ s and restores the pairs to their initial condition; i.e., it turns pairs of the form $(y, \#)$ into $\#$ s, and pairs of the form (y, x) into (x, x) . A''' then marks the left end of σ and moves to the next symbol. From this starting point, it again simulates A on the first terms of the pairs; if the simulation accepts, it restores the pairs, finds the first unmarked symbol, marks it, and moves to the next symbol. The process described in the preceding sentence is repeated until every symbol is marked; if this happens, A''' accepts. Clearly, this will happen iff σ is in $L_{\cap}(A)$. Evidently, A''' is $\#$ P, TB, or deterministic if A is. $\square$

Propositions 3.2.1–3.2.3 imply that in most cases it does not matter whether we define acceptance from a standard position (e.g., the left end), from some position, or from all positions. [Instead of the left end, we can use any specific position, provided A can locate that position; e.g., we would not want to use the midpoint of σ as a starting point if A is FS, since an FSA cannot locate midpoints (see, e.g., Section 3.3.2).] It will be convenient to use the left-end definition from now on. The set $L_{\cap}(A)$ will be called the *language* of A , and will be denoted by $L(A)$.

3.3 THE LANGUAGE HIERARCHY

A language accepted by a $\#$ PA, TBA, or FSA will be called a $\#$ -preserving, tape-bounded, or finite-state language ($\#$ PL, TBL, or FSL). The classes of all such languages will be denoted by $\mathcal{L}_{\#P}$, $\mathcal{L}_{TB}$, and $\mathcal{L}_{FS}$, respectively, and the class of all languages accepted by TAs, i.e., the class of TLs, will be denoted by $\mathcal{L}_T$. We will use the prefix D if the acceptors are required to be determin-

istic—e.g., $\mathcal{L}_{\text{DTB}}$ is the class of languages accepted by deterministic TBAs.

Determinism is not a restriction in the case of arbitrary TAs; it can be shown that $\mathcal{L}_{\text{DT}} = \mathcal{L}_{\text{T}}$. For the proof see [1, pp. 95–6]; given an arbitrary TA A , we can define a DTA A' that systematically simulates all possible sequences of transitions of A , so that A' accepts iff A accepts.

It can also be shown that $\mathcal{L}_{\text{DFS}} = \mathcal{L}_{\text{FS}}$; the proof depends on the fact that any FSA can be simulated by a “one-way” FSA (see Section 3.3.3) that moves only from left to right. Given any one-way FSA A , we can define a one-way DFSA A' whose states are sets of the states of A , and that keeps track of all the states in which A could possibly be at a given step (see [1, pp. 31–32] for the details). We define the accepting states of A' as those sets that contain accepting states of A ; thus A' accepts iff A does.

It is an open question whether $\mathcal{L}_{\text{DTB}} = \mathcal{L}_{\text{TB}}$ (and similarly for $\#P$).

In the remainder of this section, we establish some further inclusion relations among the classes of languages accepted by the various types of TAs.

3.3.1 $\mathcal{L}_{\#P} = \mathcal{L}_{\text{TB}}$

In this section we show that TBAs are as strong as $\#P$ As—i.e., they accept the same class of languages. We also show that tape-bounded FSAs are as strong as non-tape-bounded FSAs. The basic idea of the proofs is that when a $\#P$ A moves onto $\#s$, its returning state(s), if any, can be predicted by an acceptor that never leaves the non- $\#s$.

Let A be a $\#P$ A having $|Q|$ states, and suppose that A enters the $\#s$ by moving off the right-hand end of σ , while in some state $p \in Q$. If A is deterministic, then either it never returns or it returns in some specific state $q \in Q$; if A is nondeterministic, it either never returns or can return in any of a set of states $Q_R \subseteq Q$. We shall show that if A returns at all, it does so without ever getting farther away than $|Q|^2$ from σ ; in fact, if A can return in state q , it can return in that state without getting farther than $|Q|^2$ away from σ .

Let π be a legal path (i.e., a sequence of possible moves and state changes of A) on the $\#s$ that causes A to return to σ in state q .

Suppose that the farthest π gets away from σ is $n > |Q|^2 + 1$, where n is as small as possible for any such π , and where the number of times that π gets n away from σ is also as small as possible. Thus π passes through the n $\#$ s to the right of σ , which we shall call $\#_1, \dots, \#_n$. Suppose that π requires N time steps, which we shall denote by $t_1, \dots, t_N$.

It is easily seen that there must exist two time steps $t_{i_1} < t'_{i_1}$ at each of which A is on $\#_1$, and between which it is to the right of $\#_1$ and visits $\#_n$ at least once. Similarly, during the time interval (t_{i_1}, t'_{i_1}) , there must exist two time steps $t_{i_2} < t'_{i_2}$ at which A is on $\#_2$, and between which it is to the right of $\#_2$ and visits $\#_n$ at least once. Continuing this argument, we obtain a nested set of time step pairs

$$t_{i_1} < t_{i_2} < \dots < t_{i_{n-2}} < t_{i_{n-1}} < t'_{i_{n-1}} < t'_{i_{n-2}} < \dots < t'_{i_2} < t'_{i_1}$$

such that, for all $1 \leq j \leq n - 1$, A is on $\#_j$ at t_{i_j} and t'_{i_j} , is to the right of $\#_j$ between these times, and visits $\#_n$ at least once during that period.

Let q_j, q'_j be the states of A at time steps t_{i_j}, t'_{i_j} of π , $1 \leq j < n - 1$. Since $n - 1 > |Q|^2$, two of the state pairs (q_j, q'_j) must be the same, say $(q_j, q'_j) = (q_k, q'_k)$, where $j < k$. For any time steps t and t' , let $\pi(<t)$ denote the part of π prior to t , let $\pi(>t')$ denote the part subsequent to t' , and let $\pi(t, t')$ denote the part between t and t' (inclusive). Now consider the path π' obtained by concatenating $\pi(<t_{i_j})$, $\pi(t_{i_k}, t'_{i_k})$, and $\pi(>t'_{i_j})$. Since $q_j = q_k$, $q'_j = q'_k$, and A stays to the right of its starting point between these two states, we see that π' is a legal path for A on the $\#$ s and causes A to return to σ in state q after leaving it in state p . Moreover, during the part of π' corresponding to $\pi(t_{i_k}, t'_{i_k})$, A no longer reaches $\#_n$; in fact, it only gets $(n - k)$ to the right of $\#_j$, where $j + (n - k) < n$ since $j < k$. Thus π' is a legal path that either does not get as far as $\#_n$, or gets there fewer times than π did; and this contradicts our choice of π .

We have thus shown that if there exists any legal path for A that causes it to return to σ in state q after leaving it in state p , then there must exist such a path that never gets farther than $|Q|^2 + 1$ from σ . This observation allows us to prove

THEOREM 3.3.1 $\mathcal{L}_{\mathcal{P}} = \mathcal{L}_{\text{TB}}$.

Proof: Clearly, $\mathcal{L}_{\text{TB}} \subseteq \mathcal{L}_{\mathcal{P}}$, since TBAs are $\#$ PA's. Hence we

need only show that for any $\#^P A$ there exists a TBA A' that accepts exactly the same strings as A . Specifically, we define A' to simulate A as long as A remains on σ . If A leaves σ , say in state p , then by the above discussion, A' can determine whether A ever returns to σ in any given state q by internally simulating the behavior of A on the $|Q|^2 + 1$ $\#$ s adjacent to σ . [Paths that take A farther than $|Q|^2 + 1$ away from σ can be ignored, since all return possibilities for A occur on paths that stay within $|Q|^2 + 1$ of σ , and if there are no such paths, A never returns. If A accepts without returning to σ , we can replace it by an A'' that simulates it until it accepts, then moves back to σ and accepts; thus A'' accepts exactly the same strings as A , but never accepts while on $\#$ s, and so can be simulated by A' .] Since the $\#^P A$ (A or A'') need only be simulated on a string of $\#$ s of bounded length, its simulation by A' requires only a bounded amount of internal memory and can be carried out every time A' bounces off a $\#$. Thus A' can go into a nonaccepting absorbing state if A can never return, and into a returning state of A otherwise, after which A' continues the simulation of A (unless acceptance has occurred). Clearly, A' accepts iff A accepts. $\square$

In the proof of Theorem 3.3.1, if A is finite-state, so is A' ; we thus have

THEOREM 3.3.2 $\mathcal{L}_{\text{TDFS}} = \mathcal{L}_{\text{FS}}$.

In the deterministic case, we can give a somewhat different proof of Theorems 3.3.1 and 3.3.2, based on the fact that when a deterministic $\#$ -preserving automaton moves onto $\#$ s, it suffices to simulate its behavior for about $|Q|^2$ *time steps* (rather than out to *distance* $|Q|^2$). As we shall see in Chapter 4, this deterministic proof generalizes, in part, to two dimensions; we therefore present it here.

PROPOSITION 3.3.3 Let A be a deterministic FSA that has $|Q|$ states, and let the input tape of A be constant,[§] say z^∞ for some $z \in V$. Then after less than $|Q|$ steps, the sequence of states of A becomes periodic with period $m \leq |Q|$.

Proof: No matter how A moves, it always reads the same

[§]In this proposition, we temporarily relax the requirement that the input tape must be of the form $\#^\infty \sigma \#^\infty$.

symbol z ; hence its next state depends only on its current state. Since A has only $|Q|$ states, some state must occur twice during the first $|Q|$ steps of A 's operation. Suppose that the states at steps i and $i + m$ are the same, where $0 \leq i < i + m \leq |Q|$. Then the sequence of states at steps $i, i + 1, \dots, i + m - 1$ must be repeated at steps $i + m, i + m + 1, \dots, i + 2m - 1$, and the same sequence must then occur at steps $i + km, i + km + 1, \dots, i + (k + 1)m - 1$, for any $k \geq 1$, so that after time $i - 1 < |Q|$, A 's behavior is periodic, with period (at most) $m \leq |Q|$. $\square$

COROLLARY 3.3.4 Let A be a deterministic $\#PA$ that has $|Q|$ states. Then the behavior of A while it is on the $\#$ part of its input tape becomes periodic, with period $\leq |Q|$, after $< |Q|$ steps.

Proof: Since A cannot rewrite $\#s$, we can regard it as an FSA as long as it stays on $\#s$. $\square$

PROPOSITION 3.3.5 Let A be as in Corollary 3.3.4, and suppose that A moves onto the $\#$ part of its tape. Then either A returns to the non- $\#s$ within $|Q|^2 + |Q|$ steps, or it never returns.

Proof: Suppose A leaves the non- $\#$ string σ at its right end. If it returns during the nonperiodic part of its behavior (see Corollary 3.3.4), it has returned within $|Q|$ steps, and we are done. Otherwise, let its position at the beginning of its periodic behavior be k symbols to the right of σ ; clearly, $0 < k \leq |Q|$. During a single period of its periodic behavior, let A 's maximum leftward excursion from its starting position (at the beginning of the period) be h symbols, and let its net leftward excursion at the end of the period be n symbols, where evidently h and n each $\leq |Q|$. If $h \geq k$, then A returns to the non- $\#s$ during its first period, i.e., within time $\leq 2|Q|$. If not, and if $n \leq 0$, A never returns to the non- $\#s$. But if $n > 0$, A is closer to the non- $\#s$ at the end of each period than it was at the beginning, so that eventually it must return. In fact, it returns within at most j periods, where $k - jn \leq h$; this implies that j is at most $(k - h)/n \leq k \leq |Q|$. Thus the total time to return is at most $|Q|$ (the nonperiodic behavior) + $|Q|^2$ (at most $|Q|$ periods, each of length $\leq |Q|$). $\square$

Corollary 3.3.4 and Proposition 3.3.5 provide us with a proof that $\mathcal{L}_{D\#P} = \mathcal{L}_{DTB}$ and that $\mathcal{L}_{DFS} = \mathcal{L}_{DTBFS}$. Indeed, for any $D\#PA$, A , we can define a DTBA A' , that simulates A on σ . When A leaves σ ,

A' can immediately determine, by simulating a bounded number of transitions of A , whether A ever returns, and if so in what state, or whether A accepts without returning; A' can then continue the simulation of A or can accept. Clearly A' accepts iff A does, and if A is finite-state, so is A' .

3.3.2 $\mathcal{L}_T \supsetneq \mathcal{L}_{TB} \supsetneq \mathcal{L}_{FS}$

It is well known that $\mathcal{L}_T \supsetneq \mathcal{L}_{TB}$; in fact, one can prove a much more general proper inclusion result about a hierarchy of tape bounds (see [1, p. 149]). Thus we see that inability to create or destroy $\$$ s reduces the power of a TA. On the other hand, if we only prohibit creating $\$$ s, a TA's power is not reduced. In fact, given any TA A , we can define an A' that simulates A , except that whenever A creates a $\$$, A' creates a special symbol $\natural$, not in the vocabulary of A . For every transition of A when it reads a $\$$, A' has both that transition and another one with the $\$$ replaced by $\natural$. Evidently (by induction on the number of elapsed steps) the configurations of A and A' are the same except that for A' some $\$$ s have become $\natural$ s. Thus the sequences of states (or state sets) of A and A' are the same, so that A' accepts iff A accepts.

We can also show that $\mathcal{L}_{DTB} \supsetneq \mathcal{L}_{FS}$. For example, consider the set of strings σ of the form $x^n y^n$. A DTBA can accept this set by moving back and forth, alternately erasing x s and y s (changing them to z s, say) at the ends of the string, and verifying that the last y is erased just after the last x is erased. On the other hand, suppose this set were accepted by an FSA, call it A . We can assume without loss of generality that A starts at the left end of σ and accepts at the right end (given any A , simulate it by an A' that, as soon as A accepts, moves to the right end and then accepts). Thus there is a last step at which A crosses the boundary between the x s and the y s. At this step, A is in one of Q possible states, or $2^{|Q|}$ possible sets of states. But since there are infinitely many possible n s, the state (set) of A must be the same for two of them, say n_1 and n_2 . Since A never returns to the x s, its further states do not depend on the numbers of x s; hence if A accepts $x^{n_1} y^{n_1}$, it must also accept $x^{n_2} y^{n_1}$ —a contradiction.

3.3.3 One-Way Acceptors

In this section we consider TAs that are allowed to move only in one direction, say to the right. Since we have assumed that a TA always starts out at the left end of its input string σ , such a *one-way* TA (OWTA) sees all of σ . Moreover, an OWTA may as well accept as soon as it sees a $\$$, since from then on it will see nothing but $\$$ s, and can gain no further information about σ . (More precisely: let A be any OWTA, and let A' simulate A until it sees a $\$$. At that point, A' can decide whether A , starting in its current state and reading only $\$$ s, can ever enter an accepting state; if so, A' accepts, otherwise not.) Note that our OWTA thus always accepts, if at all, after $|\sigma|$ steps.

For OWTAs there is no restriction in being deterministic, tape-bounded, or even finite-state. In fact, given any OWTA A , we can define a deterministic OWTA A' whose states are the sets of states of A , and show that A' accepts iff A does (see the beginning of Section 3.3). The fact that a tape-bounded OWTA can simulate an arbitrary OWTA was pointed out in the previous paragraph. Finally, to see that a finite-state OWTA (A') can simulate an arbitrary one (A), note that the transitions of A cannot depend on the symbols that it writes, since it can never go back to read these symbols. Thus if we define A' to be the same as A except that it never rewrites any symbols, we see that the sequences of states of A and A' as they scan σ are the same, so that A' accepts iff A does.

It is well known that $\mathcal{L}_{\text{OWFS}} = \mathcal{L}_{\text{FS}}$; for the proof see [1, pp. 41–44]. Thus the class of languages accepted by one-way acceptors of any type is just the class of finite-state languages.

3.4 CELLULAR ACCEPTORS

Up to now we have considered automata that operate sequentially by moving around on an input tape and reading one symbol at a time. In this section we introduce a class of parallel automata that operate on the entire tape simultaneously. We shall show that these automata, when regarded as acceptors, are exactly as strong as TBAs. On the other hand, we shall show that they are generally much faster than TBAs—i.e., they can often accept in a much shorter time.

3.4.1 Cellular Automata and Acceptors

Informally, a *cellular automaton* is a string of “bugs” or *cells* each of which, at any given time, is in some *state*. The cells operate in a sequence of discrete time steps, at each of which every cell reads the states of its left and right neighbors and changes to a (possibly) new state. Thus, formally, a cellular automaton K is defined by specifying a pair (Q, δ) , where Q is the set of states and δ is the *transition function* that maps triples of states into sets of states (or into single states, if K is deterministic); i.e., $\delta: Q^3 \rightarrow 2^Q$ (or $\rightarrow Q$). Here, for each cell c of K , δ maps the triple (state of left neighbor, state of c , state of right neighbor) into the set of possible new states of c ; in the deterministic case, this set always consists of a single element. A *configuration* of K is simply a mapping from the integers into Q that specifies the state of each cell of K .

We shall assume from now on that Q is finite and that there is a special state $\# \in Q$ such that the initial configuration of K is of the form $\#^\infty \sigma \#^\infty$, where σ is a finite, nonnull string of non- $\#$ s. If, in addition, δ is $\#$ -preserving [i.e., $\# \in \delta(p, q, r)$ implies $q = \#$, and $\delta(p, \#, r) = \{\#\}$ for all p, r in Q], we call K a *bounded cellular automaton*.[§] Note that in the bounded case we may as well assume that the string of cells is finite and has the form $\# \sigma \#$, since the $\#$ s will always remain $\#$ s.

We now introduce the notion of acceptance of an input string (of states) by K . Formally, a *cellular acceptor* (CA) C is a triple (K, Q_I, Q_A) , where K is a cellular automaton, $Q_I \subseteq Q$ is a set of *initial states*, and $Q_A \subseteq Q$ is a set of *accepting states*, with $\# \in Q_I$. If K is bounded, we call C a *bounded cellular acceptor* (BCA). An input string is a configuration whose image is in Q_I , i.e., a string of the form $\#^\infty \sigma \#^\infty$, where $\sigma \in Q_I^+$. We say that C *accepts* this string (or for brevity, that C *accepts* σ) if repeated application of K 's transition function, starting from this configuration, can lead to an “accepting configuration.” Such a configuration can be defined in a number of ways (compare Section 3.2.3):

- (a) Every $c \in C$ has a state in Q_A .
- (b) Some $c \in C$ has a state in Q_A .
- (c) A particular $c_0 \in C$ —e.g., the one at the left end of σ —has a state in Q_A .

[§]As in Section 3.3.2, inability to create $\#$ s is not a restriction on K . We shall assume hereafter that no K ever creates $\#$ s.

Let $L_{\cap}(C)$, $L_{\cup}(C)$, and $L_{\ell}(C)$ be the sets of input strings that C accepts according to these three definitions. We shall assume here that accepting states cannot be destroyed, i.e., for all $p, r \in Q$ and all $q \in Q_A$, we have $\delta(p, q, r) = \{q\}$. We also assume that accepting states cannot cause $\#$ s to be rewritten, i.e., that if p or r is in Q_A , we have $\delta(p, \#, r) = \{\#\}$.

PROPOSITION 3.4.1 For any C there exists a C' such that $L_{\ell}(C') = L_{\cap}(C)$. Moreover, if C is bounded or deterministic, so is C' .

Proof: The noninitial states of the cells in C' are pairs of the form (x, q) , where q is a state of C . At the first time step, c_0 goes into state $(1, q)$, where q is its initial state, and all other (non- $\#$) cells go into states $(0, q')$, where the q' are their initial states. (This can be done since c_0 is the only cell whose left neighbor is $\#$.) C' then simulates C by applying the transition function of C to the second terms of its states. If a cell c that has a $\#$ as a neighbor enters a state whose second term is in Q_A , c sends out a signal (using the first items of the states) that spreads from cell to cell, provided that the cells involved have states whose second terms are in Q_A . If c_0 receives this signal from both sides, it accepts. Clearly, this can only happen if the initial string was in $L_{\cap}(C)$. Note that this proof depends on our requirements that accepting states are never rewritten and cannot cause $\#$ s to be rewritten. If not for the first requirement, c_0 could never discover that, at a given time step, every non- $\#$ cell happens to be in a Q_A state; and if not for the second, the signals from c_0 might never catch up with the (receding) $\#$ s. We have also made use of our assumption that C never creates $\#$ s; otherwise the signals could never know that they have seen all of the non- $\#$ cells. $\square$

PROPOSITION 3.4.2 For every C there exists a C'' such that $L_{\ell}(C'') = L_{\cup}(C)$. Moreover, if C is bounded or deterministic, so is C'' .

Proof: We use pairs for the states of C'' , as in the proof of Proposition 3.4.1, and c_0 marks itself uniquely at the first time step. C'' then simulates C using the second terms. If any cell's second term is in Q_A , that cell sends out signals (using the first terms) to the left and right; the propagation of these signals does not depend on the

second terms. If a signal reaches c_0 , it accepts. Clearly, this can only happen if the initial string was in $L_{\cup}(C)$. $\square$

PROPOSITION 3.4.3 For any C'' there exists a C such that $L_{\cup}(C) = L_{\ell}(C'')$. Moreover, if C'' is bounded or deterministic, so is C .

Proof: We again use pairs for the states of C , and c_0 marks itself uniquely (with first term 1) at the first step. Moreover, we define the accepting states of C as the pairs of the form $(1, q)$, where $q \in Q_A$; pairs whose first terms are 0 are never accepting states. C then simulates C'' on the second terms of its state pairs. The only cell of C that can ever enter an accepting state is c_0 ; thus $L_{\cup}(C)$ is trivially just $L_{\ell}(C'')$. $\square$

PROPOSITION 3.4.4 For any C' there exists a C such that $L_{\cap}(C) = L_{\ell}(C')$, and if C' is bounded or deterministic, so is C .

Proof: We first define an automaton $\bar{C}'$ that simulates C' at odd-numbered time steps and whose states do not change at even-numbered steps. This is done using pairs of the form (x, q) for the states of $\bar{C}'$, where $x = 0$ or 1 and q is a state of C' . At time step 0, each initial state q of a cell of C' is replaced by the state $(1, q)$. Subsequently, we have $(0, q') \in \delta((1, q_1), (1, q), (1, q_2))$ iff $q' \in \delta(q_1, q, q_2)$, and $\delta((0, q_1), (0, q), (0, q_2)) = \{(1, q)\}$ for all states q_1, q, q_2 of C' . Clearly, the sequence of configurations of $\bar{C}'$ at odd-numbered steps is the same as that of C' (at all steps) with $(1, q)$ s replacing qs . Thus if we define the accepting states of $\bar{C}'$ as the states $(1, q)$ for which $q \in Q'_A$ is an accepting state of C' , it is evident that the language of $\bar{C}'$, by any definition of acceptance, is the same as that of C' . Clearly, if C' is bounded or deterministic, so is $\bar{C}'$.

We now define C to have as noninitial states pairs of the form $(x, \bar{q})$, where $\bar{q}$ is a state of $\bar{C}'$, and where c_0 is uniquely marked, as in the previous proposition ($x = 0$ except for c_0 , which has $x = 1$). C simulates $\bar{C}'$ on the second terms of the pairs. We define the accepting states of C as all states of the form $(2, q)$, for any accepting state q of $\bar{C}'$. If c_0 enters a state of the form $(1, q)$, with q an accepting state of $\bar{C}'$, it immediately changes to state $(2, q)$, irrespective of its neighbors' states. The states whose first terms are 2 then propagate; e.g., if any neighbor of cell c has state $(2, q)$, then the

next state of c is $(2, q)$, irrespective of its own current state or its other neighbor's state. States whose first terms are 2 cannot enter into any other transitions, so that the simulation of $\bar{C}'$ stops at a given cell as soon as the propagation reaches that cell. Since the simulation of $\bar{C}'$ is quiescent at even-numbered time steps, the propagation must eventually catch up with it (even if $\bar{C}'$ is growing). Thus it is clear that if the simulation of $\bar{C}'$ accepts at c_0 , then every cell of C accepts, and conversely, so that $L_{\cap}(C) = L_{\cap}(\bar{C}') = L_{\cap}(C')$. Note that if C' is bounded or deterministic, so is C . $\square$

Propositions 3.4.1–3.4.4 show that, in all cases, the classes of languages accepted by every cell, by some cell, and by a specific cell are the same. [Instead of the cell at the left end, we could use any cell that can uniquely identify and mark itself at the start of the processing.] We shall use the left-end definition of acceptance from now on, and we shall always assume that c_0 is uniquely marked.

The set $L_{\cap}(C)$ will be called the *language* of C , and will be denoted from now on by $L(C)$. Note that $L(C)$ can contain strings of any length, depending on how many cells of C were initially in non- $\#$ states. [C is still considered to be the same CA, no matter what the length of σ may be, as long as C has the same transition function.] The class of all languages accepted by (D)(B)CAs will be denoted by $\mathcal{L}_{(D)(B)C}$.

It should be mentioned that one can also study CAs in which information can be transmitted only in one direction, e.g., left to right. In such a *one-way* CA (OWCA) the transition function maps pairs of states [i.e., (state of cell, state of left neighbor)] into states (or sets of states, in the nondeterministic case). It can be shown that OWCAs are equivalent to CAs; that nondeterministic OWBCAs are equivalent to CAs; and that deterministic OWBCAs can accept most of the BCA languages discussed in Section 3.4.3. (On OWBCAs see [2].)

Appendix to Section 3.4.1

The assumptions that accepting states cannot be destroyed and cannot cause $\#$ s to be rewritten, used in the proof of Proposition 3.4.1, can be eliminated, but at the cost of making the simulation very slow. The basic idea is as follows:

- (a) The simulation of C by C' proceeds one step at a time.
- (b) After each step, the simulation stops, and a signal is initiated from the ends of C that spreads through accepting states. If the signal reaches the distinguished cell c_0 from both sides, c_0 accepts; this can only happen if, at that step of the simulation, all non- $\#$ cells of C are in accepting states.
- (c) If a nonaccepting cell is found, the signal is modified to indicate this. When the modified signal reaches c_0 , it initiates a synchronization process that causes every non- $\#$ cell of C' to change state in a special way simultaneously. When this happens, another step of the simulation of C is performed.

The synchronization is necessary to insure that every non- $\#$ cell of C' does its step of the C simulation at the same time.

In order to simulate C one step at a time, we use pairs of the form (x, q) as states, where q is a state of C . For each transition $q' \in \delta(q_1, q, q_2)$ of C , we have a corresponding transition $(0, q') \in \delta(q_1, q, q_2)$ of the simulation. Thus initially, the first step of C is simulated, but the simulation stops at that point, since δ does not apply to pairs whose first terms are 0. We then carry out the signal propagation and synchronization steps, as described immediately below, using the first terms (x) of the states. At the end of this process, the cells simultaneously change into the states that are the second terms of their pairs [i.e., $(x, q) \rightarrow q$ for all x], so that the next step of the C simulation can take place simultaneously at every cell. This step turns the cells' states back into pairs $(0, q')$, so that the simulation of C stops again until another signal propagation and synchronization has been performed. This process is repeated until the signal informs c_0 that all non- $\#$ cells of the simulation are in accepting states (second terms in Q_A); c_0 can then accept.

The signal propagation process is straightforward: If a cell c has a $\#$ neighbor and is in a state of the form $(0, q)$, where $q \in Q_A$, then c goes into state $(1, q)$; while if $q \notin Q_A$, c goes into state $(2, q)$. Subsequently, any cell c whose state is $(0, q_1)$ and which has a neighbor in state $(2, q_2)$ goes into state $(2, q_1)$; while if c has a neighbor in state $(1, q_2)$, then c goes into state $(1, q_1)$ if $q_1 \in Q_A$, or into state $(2, q_1)$ if $q_1 \notin Q_A$. The distinguished cell c_0 is an exception: As long as either of its neighbors is in a state with first term 0, it does not change state. If both of its neighbors are in states

with the first term 1, and its own state is $(0, q)$ with $q \in Q_A$, it goes into state q (and thereby accepts). If one or both of its neighbors are in states with first term 2, or its own state is $(0, q)$ with $q \notin Q_A$, it goes into state $(3, q)$, which initiates the synchronization process to be described next.

Finally, the synchronization process operates as follows. Assume for simplicity that the distinguished cell c_0 is at the left end of the string σ of non- $\#$ s; if it is not, it can send a signal to the left end, which then acts as an initiator for the synchronization process. (See [3] for a synchronization construction that is somewhat faster and can proceed directly from an arbitrary initiation point.)

The left end sends two signals rightward, one of which travels three times as fast as the other; these signals are represented by special values of the first terms of the state pairs, say f and s . To transmit the fast signal, a cell whose left neighbor is $\#$ or f (we ignore the second terms from now on) becomes f , while an f becomes 0 at the next time step. To transmit the slow signals, a cell whose left neighbor is $\#$ or s becomes s'' ; an s'' becomes an s' , an s' becomes an s , and an s becomes 0 at the next time step. Thus the slow signal makes one move rightward at every third time step. The left end itself (3) changes to 0 as soon as it has initiated the signals, so that the signals are only sent once.

When the f signal reaches the right end of σ , it is transformed into a left-moving fast signal f' : A cell with f as its left neighbor and $\#$ as its right becomes f' ; a cell whose right neighbor is f' becomes f' ; and f' becomes 0 at the next time step. The fast signal moves leftward until it meets the slow signal at the middle of σ . Specifically, if $|\sigma| = 2k + 1$ is odd, then f' and s'' would both reach the center cell $(k + 1)$ at time step $3k$. Thus at step $3k - 1$, s is at cell k and f' at cell $k + 2$, so that cell $k + 1$ has s on its left and f' on its right. When this happens, we give cell $k + 1$ the value w . Similarly, if $|\sigma| = 2k$ is even, f' reaches cell $k + 1$ at step $3k - 2$, and s'' reaches k at step $3k - 3$; thus at step $3k - 2$, s' and f' are adjacent at cells k and $k + 1$. When this happens, we give both of these cells the value w . Thus when the signals meet, we have 0s at all cells of σ except the midpoint(s), where we have $w(s)$.

To illustrate the signal propagation process, we give two examples in Table 3.1, for the cases $|\sigma| = 5$ and 6.

We now carry out an analogous process in which each half of σ is bisected. A w sends fast and slow signals leftward and/or

TABLE 3.1

Step	$ \sigma = 5(k = 2)$	$ \sigma = 6(k = 3)$
0	f/s'' — — — —	f/s'' — — — — —
1	s' f — — — —	s' f — — — — —
2	s — f — — —	s — f — — — —
3	— s'' — f — —	— s'' — f — — —
4	— s' — — f' —	— s' — — f — —
5	— s — f' — —	— s — — — f' —
6	— — w — — —	— — s'' — f' — —
7		— — s' f' — — —
8		— — w w — — —

rightward through 0s; as soon as it has done so, it becomes a z , so that it does so once only. [If there are two w s, the left one sends signals leftward and the right one sends them rightward; if there is only one w , it sends them out on both sides.] The fast signals bounce off the ends of σ and meet the slow signals at the midpoints of their segments. Note that the two segments have the same length ($k + 1$, if $|\sigma| = 2k + 1$; k , if $|\sigma| = 2k$), so that this happens at the same time for both segments. When it happens, we again create w s; σ now has z (s) at its midpoint(s) and w s at its quarter and three-quarter points.

This process is repeated: The w s send out signals through 0s and become z s; the fast signals bounce off other z s or off the ends of σ , meet the slow signals, and create w s at the segment midpoints. Again, all these w s are created simultaneously. σ now has z s at its midpoint and quarter points, and w s at its $\frac{1}{8}$, $\frac{3}{8}$, $\frac{5}{8}$, and $\frac{7}{8}$ points, and we repeat the process again. The successive creations of w s are summarized in Table 3.2 for the cases $|\sigma| = 11$ and 12.

Eventually, the z s have only one or two 0s between them, so that when the signals meet, all the remaining 0s simultaneously turn into w s. Prior to this, no w or z ever had w s or z s on both sides of it; but now they *all* do. We can now use a transition that turns all

TABLE 3.2

Step	$ \sigma = 11$	Step	$ \sigma = 12$
15	— — — — — w — — — — —	17	— — — — — w w — — — — —
22	— — w — — z — — w — —	24	— — w — — z z — — w — —
24	w w z w w z w w z w w	26	w w z w w z z w w z w w

cells whose first terms are w or z , and which have ws , zs , or $\#s$ on both sides, into their second terms, so that another step of C can be simulated; note that this transition applies simultaneously to every cell in σ .

The number of time steps required to create a w at the midpoint of a string of length $2k + 1$ is $3k + 1$, and the time to create ws at the midpoints of a string of length $2k$ is $3k$. At each repetition of this bisection process, the length of the strings being bisected is less than half that at the previous repetition ($2k + 1$ becomes k ; $2k$ becomes $k - 1$). Thus the total time required for the synchronization procedure is less than $\frac{3}{2}|\sigma|[1 + \frac{1}{2} + \dots] = 3|\sigma|$. (This does not count the time needed for the chief cell to send a signal to the left end of σ to initiate the process, which is at most an additional $|\sigma|$ time steps.) As mentioned earlier, a more complicated synchronization procedure can be defined [3] that requires less than $2|\sigma|$ time steps; but in any case, the number of steps required is of order $|\sigma|$. Thus the simulation of a given C using this method takes order $|\sigma|$ time steps for each step of C , unlike the simulation in the proof of Proposition 3.4.1, which is nearly as fast as C itself, except for the propagation of the acceptance signal (which takes at most an additional $|\sigma|$ time steps).

3.4.2 Equivalence to Sequential Acceptors

THEOREM 3.4.5 Turing acceptors can simulate cellular acceptors.

Proof: Given a cellular acceptor C with transition function δ , we define a Turing acceptor A that simulates C as follows: The vocabulary of A is the state set of C . To simulate one transition of C , A systematically scans σ , the non- $\#$ part of its tape, say from left to right. (Since $\#s$ are never created, A can know when this scan is finished.) At each step of this scan, A remembers (by storing them internally) the tape symbols at the two previous positions; in other words, if A is at position i , it remembers the symbols q_{i-2} and q_{i-1} at positions $i - 2$ and $i - 1$. A then reads the symbol q_i at position i and rewrites it as $\delta(q_{i-2}, q_{i-1}, q_i)$ —or, in the nondeterministic case, as some q in the set $\delta(q_{i-2}, q_{i-1}, q_i)$. A can then forget q_{i-2} , memorize q_i , and move on to position $i + 1$. Thus when A has reached the right end of σ , say at position n , it has written in each position i a possible new state of the cell of C in position $i - 1$. At this point, A remembers q_{n-1} and q_n and reads the $\#$ at the right of

σ , so it can compute a possible new state for the n th cell of C ; it stores this state internally. [If C is not a BCA, A can write this state at position $n + 1$, where there was formerly a $\#$; and A can also write (in appropriate positions) the new states of the $\#$ s adjacent to σ , if C would have changed these to non- $\#$ s.] A now does a second scan, from right to left, and simulates the next transition of C . This time, A can write the new states in their proper positions, rather than shifted one over to the right; e.g., in position n , A reads the current state of the cell in position $n - 1$, and since A already knows the current states of the cells in positions $n + 1$ (i.e., $\#$) and n (stored internally), it can immediately write the new state for position n in that position. Thus in each scan of σ from left to right or from right to left, which takes $|\sigma|$ time steps, A can simulate a complete (parallel) transition of C . A accepts iff its simulation of C accepts. Note that if C is deterministic, so is A , and if C is bounded, A is tape-bounded. Note also that this simulation is the fastest possible, since A cannot modify a tape symbol without visiting that position, and $|\sigma|$ time steps are needed to visit all of σ . $\square$

THEOREM 3.4.6 Cellular acceptors can simulate Turing acceptors.

Proof: Given a Turing acceptor A with transition function δ , we define a cellular acceptor C that simulates A as follows: The noninitial state set of C is $V \times (Q \cup \{0\}) \times \Delta \times (\Delta \cup \{0\})$, where V and Q are the vocabulary and state set of A , and $\Delta = \{L, R, N\}$. At the first time step, each cell $c \in C$ except c_0 goes into a state of the form $(y, 0, N, 0)$, where $y \in V$ is c 's initial state. Cell c_0 goes into the state (x, q_0, N, d) , where $x \in V$ is c_0 's initial state, q_0 is A 's initial state, and $d \in \Delta$ is a direction in which A can move when it initially reads x [i.e., $\delta(x, q_0, N)$ contains a triple whose third term is d].

At subsequent time steps, cells whose states and neighbors' states are of the form $(y, 0, N, 0)$ do not change state. Let c' be a cell whose state is of the form (x, q, d_1, d_2) , and let (x', q', d_2) be a triple in $\delta(x, q, d_1)$. Then

- (a) If $d_2 = N$, the new state of c' is of the form (x', q', N, d') , where $d' \in \Delta$ is the third term of some triple in $\delta(x', q', N)$. In this case the neighbors of c' do not change state.
- (b) If $d_2 = L$, the new state of c' is $(x', 0, N, 0)$, and the new state of the left neighbor of c' is (y, q', L, d') , where $(y, 0, N, 0)$ was its old state, and where $d' \in \Delta$ is the third term

of some triple in $\delta(y, q', L)$. The right neighbor of c' does not change state.

- (c) If $d_2 = R$, the situation is analogous to that in (b), with R replacing L , and the roles of the left and right neighbors interchanged.

Evidently, at any time step there is exactly one cell c' whose state has the form (x, q, d_1, d_2) , and that cell is at a position in which A could have been at the given time step.

[The term d_2 is needed in the state of c' so that its neighbors know which of them, if any, is to become the new c' . If c' did not pick its own (simulated) direction of motion, and if $\delta(x, q, d_1)$ contained triples with more than one third term, then both neighbors of c' (or c' and one of its neighbors, or all three) might try to become the new c' . This problem does not arise if A is deterministic, since in that case $\delta(x, q, d_1)$ is always a singleton.]

This process is repeated until the cell c' enters a state (x, q, d_1, d_2) for which $q \in Q_A$, i.e., for which A would have accepted. At this point, c' sends an acceptance signal to c_0 , and c_0 accepts as soon as it receives this signal. Thus C accepts iff A does. Note that if A is deterministic, so is C , and if A is tape-bounded, we can regard C as bounded. [The latter requires special cases in the simulation to handle the steps in which A bounces off $\#$ s without C actually having to rewrite $\#$ s; the details will not be given here.] Note also that this simulation is the fastest possible, since C requires only one time step to simulate each time step of A . $\square$

Theorems 3.4.5 and 3.4.6 immediately imply

THEOREM 3.4.7 $\mathcal{L}_{(D)C} = \mathcal{L}_{(D)T}$; $\mathcal{L}_{(D)BC} = \mathcal{L}_{(D)TB}$.

Thus we have not created any new classes of languages by introducing CAs.

3.4.3 Speed Comparisons

As we have just seen, CAs can simulate TAs in real time; hence any language can be accepted by a (B)CA at least as fast as it can be accepted by a T(B)A. (This ignores the acceptance signal propagation process at the end of the simulation, which requires at most $|\sigma|$ time steps.) On the other hand, some languages can be ac-

cepted more quickly by (B)CAs than by T(B)As. In this section we discuss comparative acceptance speeds for sequential and parallel automata.

Let A (or C) be any acceptor, and suppose that there exists a function f , taking natural numbers into natural numbers, such that if A accepts the string σ at all, it accepts σ in at most $f(|\sigma|)$ time steps. We then say that A accepts *in time of order f* . For example, if f is linear, we say that A accepts in *linear time*; if f is the identity function, we say that A accepts in *real time*. We shall use the standard abbreviation $O(\)$ for "on the order of $(\)$."

It is easily seen that, except in certain special cases, no A or C can accept in faster than real time. Indeed, suppose A accepts some string σ in m time steps, where $m < |\sigma|$. Since A cannot make more than m rightward moves in m time steps, its state at time m cannot depend on the last $|\sigma| - m$ symbols of σ . Thus A accepts *any* input string whose first m symbols are the same as those of σ . Similarly, suppose C accepts σ in $m < |\sigma|$ steps. The state of c_0 at step m cannot depend on the initial states of cells farther than m away from c_0 ; hence C accepts *any* input string whose first m states are the same as those of σ . We may thus conclude that for "interesting" languages L , the time required to accept any $\sigma \in L$ must be at least $|\sigma|$.

There exist many simple languages that can, in fact, be accepted in real time. For example, let $L_{\bar{x}}$ be the set of strings in which a particular symbol x never appears. A deterministic FSA A that accepts $L_{\bar{x}}$ in real time can be defined as follows: A moves rightward; if it finds an x , it goes into a nonaccepting state and stops moving; if it reaches a $\#$ without finding an x , it accepts. Similarly, a BCA C that accepts $L_{\bar{x}}$ in real time is defined as follows: An acceptance signal is propagated leftward from the right end of σ ; if it meets an x , it stops; if it reaches c_0 without meeting an x , C accepts.

On the other hand, there are languages for which parallel acceptance is faster than sequential acceptance. An example is the set L_{sym} consisting of symmetrical strings of odd length made up of x s and y s, except for the center symbol, which is w . It can be shown [1, p. 145] that if A is any sequential acceptor whose language is L_{sym} , the time required for A to accept a string $\sigma \in L_{\text{sym}}$ of length $2n + 1$ is at least $O(n^2)$. For example, we can define such an A informally as follows:

- (a) A memorizes the leftmost symbol of σ , erases it (i.e., rewrites it as some new symbol z), moves to the right end of σ and compares the rightmost symbol with the memorized symbol. If they are different, A stops moving and goes into a nonaccepting state. If they are the same, A erases the rightmost symbol, memorizes the rightmost non- z symbol, moves to the left end of σ and compares the leftmost non- z symbol with the memorized symbol, and so on.
- (b) This process is repeated until a stage is reached at which the symbol to be memorized is w rather than x or y . At this stage, A checks that only z s remain on the other side of the w , and if so, it accepts.

Readily, acceptance by this A (which is a deterministic TBA) takes about $(2n + 1) + (2n - 1) + \cdots + 1 = (n + 1)^2$ time steps.

A deterministic BCA, C , whose language is L_{sym} can be defined informally as follows:

- (a) An x or y that has the w as right neighbor changes to u ; one that has the w as left neighbor changes to v . If the w has x as left neighbor and y as right neighbor, or vice versa, it changes to t ; otherwise, it remains w .
- (b) x s and y s can shift rightward through u s and leftward through v s.
- (c) A v that has $\#$ as its right neighbor becomes an acceptance signal z that can propagate leftward through v s. If z reaches the w , it changes to a new signal z' that can propagate leftward through u s.

It is not hard to see that C accepts iff its input string was in L_{sym} (otherwise the w would change to t at some point, or there would be excess x s or y s on one side of the w that did not change to u s or v s, and the acceptance signal could not reach c_0). The time required for acceptance is $O(n)$ (in fact, about $3n$), since it takes n steps for the first v to reach the right end, n steps for z to reach the w , and n steps for z' to reach c_0 .

We can consider upper as well as lower bounds on acceptance time. A nondeterministic acceptor can take arbitrarily long to accept a given string, since it can make an unbounded number of nondeterministic transitions into nonaccepting states before it finally makes one into an accepting state. Similarly, a non-(tape-)

bounded acceptor can take arbitrarily long to accept, since it can perform an arbitrarily long computation before it accepts.[§] On the other hand, a TBA, if it accepts σ at all, must do so in less than $|\sigma + 2||Q||V|^{|\sigma|}$ time steps. [*Proof:* there are only $|V|^{|\sigma|}$ possible tapes, on each of which the TBA can only be in $|\sigma + 2|$ different positions (including its positions when it bounces off $\#$ s) and in $|Q - Q_A|$ nonaccepting states. If one of these configurations repeats, the configurations have become periodic and the TBA can never accept.] Similarly, if a BCA accepts σ at all, it must do so in less than $|Q|^{|\sigma|}$ time steps. (Using Proposition 3.3.5, we can similarly give an upper bound on the number of time steps within which a $\#$ PA must accept σ if it ever does so.)

In the remainder of this section, we give several additional efficient algorithms for acceptance by CAS; all of these algorithms require time proportional to the string length $|\sigma|$.

(a) *Palindromes* [4]: Let L be the set of symmetric strings $\sigma = \omega\omega^R$, where ω is a nonnull string of x s and y s (say), and ω^R is the reversal of ω . To accept L , each cell of C sends a signal u leftward and rightward if its state is x , and a signal v leftward and rightward if its state is y . (These signals are erased when they reach the ends of the string.) If a pair of consecutive cells ever contain different signals (u and v or v and u), it cannot be the center pair of cells of a palindrome, and its cells mark themselves appropriately. Concurrently, the leftmost and rightmost cells send signals ℓ and r rightward and leftward. These signals meet at the center pair of cells. Evidently, σ is a palindrome iff this pair is unmarked when the ℓ and r signals meet there. [No other pair can remain unmarked, since for any other pair there is a $\#$ (whose occurrence is signaled by ℓ or r) located symmetrically to a non- $\#$.] The ℓ signal is then erased, and the r signal becomes an acceptance or rejection signal (r' or r'') that continues on to the left-hand cell; this cell then accepts or rejects, as appropriate.

(b) *Repetitions*: Let L be the set of strings $\sigma = \omega\omega$, where ω is a nonnull string of (say) x s and y s. To accept L , the rightmost cell of C sends out a signal u or v leftward, depending on whether its state is x or y . When this signal has passed it, the next-to-rightmost cell sends out a u or v signal leftward; when both of

[§]Indeed, given any time bound f , we can design our acceptor so that it first performs some operation that takes longer than $f(|\sigma|)$ time steps—e.g., it computes $f(|\sigma|)^2$ —before accepting.

these signals have passed it, the third-from-rightmost cell sends out a u or v ; and so on. Thus the right end of σ generates a string of leftward-moving us and vs representing the states of the cells in reverse order. Concurrently, the left end of σ sends out a signal ℓ rightward; this meets the string of us and vs in the middle of σ . When they meet, the ℓ stops and becomes an ℓ' . As each u or v passes the ℓ' , it becomes a u' or v' . When a cell in state x or y sees a u' or v' , its state is erased, and the u' or v' becomes a w or z , depending on whether or not it matched (u' matches x , v' matches y). Evidently, this process matches the signals from the cells in the right half of σ with the states of the cells in the left half of σ , rightmost first. If they all match, a string of ws is created; otherwise, some zs are also created. The cell c_0 at the left end receives this string; if it sees any zs , it rejects; if it sees no zs , and the ws stop coming, it accepts.

(c) *Run length*: Let L be the set of strings σ of the form (say) $x^n y^n$, where $n \geq 1$. To accept L , C sends signals ℓ and r rightward and leftward from its ends. If ℓ sees anything but xs , it changes to ℓ' ; if r sees anything but ys , it changes to r' . When ℓ and r meet, ℓ is erased and r turns into r'' , which continues leftward. If c_0 receives r'' , it accepts; if it receives r' , it rejects. [Similarly, C can accept, e.g., $\{x^m y^n \mid m \leq n\}$.]

Acceptance of strings of the form $x^n y^n z^n$ is slightly more complicated [4]. Here signals are sent rightward and leftward from the ends and are also sent rightward from xy pairs and leftward from yz pairs. If the left-end signal meets a yz signal at an xy boundary, and these signals have seen only xs with only ys , respectively, we know that σ begins with $x^h y^h$. Similarly, if the right-end signal meets an xy signal at a yz boundary, and they have seen only zs and only ys , respectively, σ ends with $y^k z^k$. The left- and right-end signals can then continue until they meet; if they have seen only ys since meeting the yz and xy signals, we know that σ is of the form $x^h y^{h=k} z^k$. The left-end signal can then be erased, and the right-end signal can continue to c_0 , which then accepts (or rejects, if one of the conditions failed to be met). We can similarly accept strings of the form $x^i y^j z^k$, where $j = i$ or $j = k$, by verifying that at least one of the equality conditions is met.

(d) *Parenthesis strings* [4]: Let L be the set of well-formed parenthesis strings σ (strings composed of several kinds of parentheses can be treated analogously). To accept L , each left parenthesis sends a signal u rightward, and each right parenthesis sends a

signal v leftward. The right end of σ also sends the signal r leftward. When a u meets a v , they cancel out. If a v reaches c_0 , it rejects; this means that some initial segment of σ had an excess of right parentheses over left parentheses. If a u meets r , it changes to r' , and if r' reaches c_0 , it rejects; this means that some final segment of σ had an excess of left over right parentheses. If r reaches c_0 , it accepts.

(e) *Counting* [5]: Let L be the set of strings of (say) x s and y s in which the numbers of x s and y s are equal (or there are more x s than y s, etc.). C can accept a language of this type by counting the numbers of x s and y s and comparing the counts.

To count x s, we shift the x s leftward, and the cells of C simulate a binary counter with the least significant bit at the left end. Specifically, the i th cell contains two bits (h_i, k_i) , representing sum and carry bits, which are initially both 0. When an x reaches the left end, it is erased, and we add 1 to h_1 (i.e., if $h_1 = 0$, we change it to 1; if $h_1 = 1$ we change it to 0 and change k_1 to 1). The carry bits propagate rightward according to the following rules: Whenever $k_i = 1$, we change it to 0 at the next time step. At the same time, if $h_{i+1} = 0$, we change it to 1; if $h_{i+1} = 1$, we change it to 0 and change k_{i+1} to 1. It is easily seen that after at most $|\sigma|$ steps, all the x s have been shifted leftward and added into this counter; and after at most an additional $\log_2 |\sigma|$ steps, the carry from the last addition has finished propagating.

To accept σ iff the numbers of x s and y s are equal, or unequal in a specified order, C counts the x s and y s simultaneously, using two independent counters C_x and C_y . When a signal from the right end of C has reached the left end and returned to the right end, the counting must be finished. A second signal can then be sent out from the right end, one that compares the contents of the two counters, most significant bit first. If this signal finds that C_x and C_y match up to some point, and at that point C_x has 1 but C_y has 0, we know there are more x s than y s; while if C_x and C_y match all the way to the left end of σ , we know that the numbers of x s and y s are equal. This entire process takes linear time, namely, $3|\sigma|$ time steps.

A simpler method of comparing the numbers of x s and y s is to simply shift them leftward, independently of one another, until they are "packed" (i.e., left-justified). The i th cell contains two bits (u_i, v_i) that denote the presence or absence of an x or a y . Initially, we set u_i to 1 if the i th cell contained an x , and v_i to 1 if it

contained a y . On subsequent steps, if $u_i = 0$ and $u_{i+1} = 1$, we set u_i to 1 and u_{i+1} to 0, and analogously for the v s. It is easily seen that after at most $|\sigma|$ steps the 1s in the u_i s and v_i s are all left-justified. When a signal from the left end has reached the right end, this process must be finished. The right end can then send out a signal that compares the lengths of the strings of 1s in the u and v bits, which are just the numbers of x s and y s in unary notation. Thus the entire process takes only $2|\sigma|$ time steps. We have given the more complicated construction involving binary counters because in two dimensions the unary construction would be much less efficient; see Section 5.5.1.

3.5 CLOSURE PROPERTIES

In this section, we show that the classes of languages defined in Section 3.3 are closed under various set-theoretic and geometric operations.

PROPOSITION 3.5.1 A finite intersection of TLs, TBLs, or FSLs is a language of the same type.

Proof: Let $A_1, \dots, A_n$ be acceptors for the languages $L_1, \dots, L_n$ of the given type. Define A as follows: It first simulates A_1 ; if the simulation accepts, it simulates A_2 ; and so on, until it finally accepts if all the simulations accept. For TLs or TBLs, before carrying out the simulation we first replace each symbol by a pair of identical symbols, and then perform the simulation of A_1 on the second terms of the pairs, leaving the first terms intact; if A_1 accepts, we restore the original second terms (by setting them equal to the first terms) and then simulate A_2 , and so on. Note that this construction preserves determinism. $\square$

PROPOSITION 3.5.2 A finite union of TLs, TBLs, or FSLs is a language of the same type.

Proof: Define A so it nondeterministically simulates one of the A_i s; thus A accepts iff one of the A_i s would have accepted. $\square$

For TLs and FSLs, the nondeterministic nature of this last proof is unimportant, since determinism is no restriction (see the beginning of Section 3.3). For TBLs, to handle the deterministic case, we

can use a different proof: Replace each symbol by an n -tuple of identical symbols, and then perform simulations of all the A_i s simultaneously, using the i th term of each n -tuple to record the symbol that A_i would have read at that position, and also to record the location of A_i . At each step of the simulation, A scans the non- $\#$ part of the tape, finds each A_i , and simulates its next move. A simulates the states of the A_i s internally, using n -tuples of states. If some simulation accepts, so does A . [Note that we can also use this proof for TLs, based on the fact that any TA can be simulated by a TA that never creates $\#$ s (Section 3.3.2), so that scanning the non- $\#$ part of the tape is still possible. We can also use n -tuples of states to simulate simultaneously all the A_i s in the FSA case; here we can use the fact that it suffices to simulate one-way FSAs, so that we need not keep track of their locations since they all move in step.] The same construction can be used to prove Proposition 3.5.1, except that A accepts iff all the simulations accept.

The complement of an FSL is also an FSL. This again follows from the fact that we need only consider one-way FSAs; given an OWFSA A that accepts L , we simulate it, and if it has not accepted at the end of its scan, we accept; thus we evidently accept exactly $\bar{L}$. The complement of a TL is not necessarily a TL; see [1, p. 123]. It is not known whether the complement of a TBL must be a TBL; see [1, p. 132].

PROPOSITION 3.5.3 Any singleton $\{\sigma\}$ is an FSL.

Proof: Given $\sigma = x_1 \dots x_n$, we can define a one-way deterministic FSA A that accepts σ . We do this as follows: If A is in its starting state q_0 and reads x_1 , it goes into state q_1 ; $\dots$; if it is in state q_i and reads x_{i+1} , it goes into state q_{i+1} ; $\dots$; state q_n is an accepting state. If A is in any state q_i ($i \geq 0$) and reads any symbol other than x_{i+1} , it goes into the nonaccepting absorbing state q_{n+1} . Clearly, A accepts iff its input is σ . $\square$

COROLLARY 3.5.4 Any finite set of strings is an FSL.

Proof: Propositions 3.5.2 and 3.5.3. $\square$

Let L_σ be the set of strings that contain a given substring σ ; then L_σ is an FSL. Indeed, we can define a DFSA A that scans a given string from left to right, and whenever it sees the initial

symbol x of σ_o , simulates an acceptor for $\{\sigma_o\}$. If this accepts, so does A ; if not, A returns to that instance of x (which can be located because it is at a bounded distance away) and resumes its scan. [It follows by the preceding paragraph that the complement of L_o , i.e., the set of strings that fail to contain a given σ_o , is also an FSL. In particular, the set of strings that contain or fail to contain a given symbol is an FSL.]

A subset of an FSL is not necessarily an FSL. For example, the set of strings $\{x^i y^j \mid i, j \geq 1\}$ is evidently an FSL. (A one-way DFSA accepts it by staying in state q_0 as long as it reads xs ; changing to state q_1 when it reads a y , and remaining in q_1 as long as it reads ys ; and accepting if it reaches the right end of σ while in state q_1 .) On the other hand, as we saw in Section 3.3.2; the subset $\{x^n y^n \mid n \geq 1\}$ of this language is not an FSL.

We now consider some geometric operations on languages. For any string σ , let σ^R denote the reversal of σ ; i.e., if $\sigma = x_1 \dots x_n$ then $\sigma^R = x_n \dots x_1$. For any language L , let $L^R = \{\sigma^R \mid \sigma \in L\}$; we call L^R the *reversal* of L .

PROPOSITION 3.5.5 The reversal of any TL, TBL, or FSL is a language of the same type.

Proof: Given an acceptor A for L , we can define an acceptor A' for L^R as follows: Starting at the left end of an input string σ , A' moves to the right end and then proceeds to simulate A with all move directions reversed; i.e., if A moves right, A' moves left, and vice versa. Clearly, the successive configurations of A' are exactly the reversals of those of A , so that the simulation accepts σ^R iff A accepts σ . Note that this construction preserves determinism, finite-statensness, and tape-boundedness. $\square$

For any string $\sigma = x_1 \dots x_n$, by a *cyclic shift* of σ is meant any string of the form $x_{k+1} \dots x_n x_1 \dots x_k$, for some $1 \leq k \leq n$. (Note that for $k = n$, this is σ itself.) For any language L , by the *cyclic closure* $\hat{L}$ of L is meant the set of all cyclic shifts of the strings in L .

PROPOSITION 3.5.6 If L is a TL, TBL, or FSL, so is $\hat{L}$.

Proof: We first consider the FSL case. Let A be a one-way DFSA that accepts L . Define an A' that starts at the left end of the given string σ in some state q of A and moves rightward. At each move,

A' computes two possible state changes for A , one corresponding to the symbol it actually sees, and the other corresponding to what would happen if A saw a $\#$. If the latter possibility yields a nonaccepting state of A , it is ignored. If it yields an accepting state, A' continues with two simulations, the original one and a new one in which A now enters its initial state q_0 . The new simulation simply continues until the right end of σ is reached; but the original simulation continues to compute two alternatives at each step, and to create another new simulation whenever the $\#$ alternative yields an accepting state. This may happen many times, but the number of simulations that need be tracked is bounded, since A can only be in a bounded number of states. Thus at any given stage, A' is tracking the original simulation (which may keep on branching) together with a set of new simulations (which do not branch). When A' reaches the right end of σ , if any one of these new simulations is in the original state q , A' accepts; in fact, if this happens, σ is a cyclic shift of a string σ' in L , and the starting point of σ' is a point at which A' began that particular new simulation. If none of the new simulations is in state q , A' can return to the left end of σ and begin the process again with a new starting state q' . If σ is in $\hat{L}$, some starting state of A' at the left end of L must eventually lead to acceptance in this way. Note that this construction preserves determinism; alternatively, we could pick the starting state (and the point at which A' switches to a new simulation) nondeterministically. Incidentally, we could have started A' at the left end simulating *every* state of A , and have A' keep track of a separate set of simulations for each starting state.

Next, let L be a TBL, and let A be a TBA for L . We define A' to move rightward until it marks some nondeterministically chosen point and begins to simulate A , except that

- (a) Whenever the simulation tries to pass the marked point from the left, or reaches the marked point from the right, it behaves as though A had reached a $\#$.
- (b) Whenever the simulation reaches an end of the string and moves onto a $\#$, it ignores the $\#$, moves to the other end of the string, and behaves as if it had moved onto that string endpoint.

Evidently, the successive configurations of this simulation [except while moving to another end of the string as in (b)] are essentially cyclic shifts of configurations of A on a string in L that ends at the

marked point. Thus the simulation accepts iff σ is a cyclic shift of a string in L .

If L is a TL, we must modify the construction in the preceding paragraph to allow simulation of rewriting $\#$ s. Specifically, we can modify (a) so that if the simulation rewrites a $\#$, A' shifts (say) the initial segment of σ (up to the marked point) one space to the left, symbol by symbol, thus creating a space just after the marked point. A' then writes the desired symbol in that space, moves the mark to that symbol, and resumes the simulation. [Alternatively, A' can simply create a new copy of σ in which the initial segment (up to the mark) now follows the final segment, and can then carry out the simulation on that copy.] $\square$

For any language L , let L^+ denote the set of all concatenations of strings in L , i.e., $L^+ = \{\sigma_1 \dots \sigma_n \mid n \geq 1; \sigma_i \in L, 1 \leq i \leq n\}$. We shall call L^+ the (concatenative) *closure* of L .

PROPOSITION 3.5.7 If L is a TL, TBL, or FSL, so is L^+ .

Proof: In the FSL case, we start at the left end of σ and simulate a one-way DFSA A for L , starting in its initial state q_0 . At each point of σ , we compute two possible state changes, as in the proof of Proposition 3.5.6. If the state change corresponding to a $\#$ does not yield an accepting state of A , we ignore it; but if it does, we continue with two simulations: the original one and a new one in which A reenters its initial state q_0 . For *both* of these simulations, we compute two possible state changes at each point, and whenever the change corresponding to a $\#$ yields an accepting state of A , a new simulation branches off. This may happen repeatedly, but the total number of simulations that need to be tracked remains bounded by the state set size of A . When A' reaches the right end of σ and reads a $\#$, if any of these simulations is in an accepting state, A' accepts. Evidently, this happens iff there exists a segmentation of σ into a concatenation of strings in L , in which the break-points are just those points at which the simulation had switched to q_0 . Note that this is a deterministic construction; nondeterministically, we could simply pick an arbitrary set of points at which the simulation is in an accepting state, and switch back to q_0 at these points.

In the TBL case, we define A' to segment σ nondeterministically by placing a set of marks on it, and then to simulate an acceptor A

for L successively on each of these segments σ_i , treating the marks as endpoints of σ_i [see (a) in the proof of Proposition 3.5.6]. If each of these simulations accepts, A' accepts. Evidently, this can happen iff $\sigma \in L^+$. In the TL case the proof is similar, except that if the simulation needs to rewrite a $\#$, A' shifts the initial or terminal part of σ in order to create a space at the appropriate segmentation point, as in the proof of Proposition 3.5.6. $\square$

By the *concatenation* of the languages $L_1, \dots, L_k$, we mean the set of strings $L_1 \dots L_k = \{\sigma_1 \dots \sigma_k \mid \sigma_i \in L_i, 1 \leq i \leq k\}$.

PROPOSITION 3.5.8. Any concatenation of TLs, TBLs, or FSLs is a language of the same type.

Proof: The proof in the FSL case is analogous to that of Proposition 3.5.7, using one-way DFSAs $A_1, \dots, A_k$ for the languages $L_1, \dots, L_k$. At each branching, we both continue the current simulation (of the A_i that accepted when it read a $\#$) and begin a new simulation of A_{i+1} , starting in its initial state; thus we only allow sequences of branchings that are k long. At any stage, we may be simulating all k A s, and for each of them, we may be in many different states; but the total number of cases to be tracked is still bounded (by the sum of the state set sizes of $A_1, \dots, A_k$). When we reach the right end of σ , if a simulation of A_k is in an accepting state, A' accepts; evidently, this happens iff $\sigma \in L_1 \dots L_k$. The proofs in the TBL and TL cases are just like those in Proposition 3.5.7, except that we segment σ into exactly k parts, and simulate the appropriate acceptor on each part. $\square$

REFERENCES

1. J. E. Hopcroft and J. D. Ullman, "Formal Languages and Their Relation to Automata." Addison-Wesley, Reading, Massachusetts, 1969.
2. C. R. Dyer, One-way bounded cellular acceptors, Tech. Rep. No. 680, Computer Science Center, Univ. of Maryland, College Park, Maryland, July 1978.
3. F. R. Moore and G. G. Langdon, A generalized firing squad problem. *Information and Control* **12**, 212–220 (1968).
4. A. R. Smith III, Cellular automata and formal languages. *Proc. SWAT*, 11th, pp. 216–224 (1970).
5. A. R. Smith III, Two-dimensional formal languages and pattern recognition by cellular automata. *Proc. SWAT*, 12th, pp. 144–152 (1971).

Chapter 4

SEQUENTIAL ARRAY ACCEPTORS

4.1 INTRODUCTION

Array acceptors are two-dimensional analogs of the automata studied in Chapter 3. Here the tape is an infinite two-dimensional array of symbols, and we shall usually require that all but a finite number of these symbols are $\#$ s. The non- $\#$ s may constitute a *rectangular* subarray, or we may only require them to be a *connected* subarray (see Section 2.2); both of these restrictions are generalizations of the requirement, in Section 3.2.2, that a one-dimensional tape must be of the form $\#^\infty \sigma \#^\infty$, so that the non- $\#$ s constitute a connected string. We shall first study array acceptors in the case of a rectangular subarray (Section 4.3), and shall then consider the generalization to an arbitrary connected subarray (Section 4.4). The rectangular case is more realistic since digital pictures are rectangular arrays, but the connected case poses some interesting research problems. Sequential acceptors will be treated in this chapter; cellular acceptors will be studied in Chapter 5.

4.2 ARRAY AUTOMATA AND ACCEPTORS

Informally, a two-dimensional automaton is a "bug" that, at any given time, is in one of a set of states and is located at a given position on an array of symbols. Just as in the one-dimensional case, the bug operates in a series of discrete time steps, at each of which it

- (a) reads the symbol in its current position, erases that symbol, and replaces it by a (possibly) new one;
- (b) changes to a (possibly) new state;
- (c) moves to a neighboring position, or possibly does not move.

Note that we could define “neighboring” here to mean one of the four horizontal or vertical neighbors, or we could allow it to include the four diagonal neighbors as well (see Section 2.2). For simplicity, we shall always use the first of these definitions in this chapter. Note that a diagonal move can be accomplished by making one horizontal move followed by one vertical move (or vice versa); thus an automaton that is allowed to make diagonal moves can be simulated, at half the speed, by one that is allowed to make only horizontal or vertical moves.

In general, the state into which the bug goes, the symbol that it writes, and the direction in which it moves all depend on its current state, on the current symbol, and on the direction in which it last moved (see Section 3.2). Thus the bug’s behavior is described, exactly as in the one-dimensional case, by a *transition function* that maps (state, symbol, direction) triples into (state, symbol, direction) triples; the only difference is that there are now five possible directions (left, right, up, down, and no move), rather than three (left, right, and no move). We call the bug *deterministic* if each triple maps into a unique triple; otherwise, we call it *nondeterministic* and regard it as mapping triples into sets of triples.

Formally, an *array automaton* M is a triple (Q, V, δ) , where Q, V, δ are defined exactly as in the one-dimensional case (Section 3.2.1), except that $\Delta = \{L, R, U, D, N\}$ (“left,” “right,” “up,” “down,” and “no move”). A *tape* is a mapping $\tau: I \times I \rightarrow V$ from the pairs of integers (coordinates of array points) into the vocabulary. A *configuration* is a quintuple (q, d, i, j, τ) , where q, d, τ are as in Section 3.1.1, and (i, j) are the coordinates of M ’s position. The mapping $\tilde{\delta}$ between configurations induced by the transition function δ is defined as follows: Let $\tau_{ij} \in V$ be the symbol in the (i, j) position on τ , let $(q', v', d') \in \delta(q, \tau_{ij}, d)$, and let $\tau' = \tau$, except that τ_{ij} has been replaced by v' ; then $(q', d', i', j', \tau') \in \tilde{\delta}(q, d, i, j, \tau)$, where

$$\begin{aligned} (i', j') &= (i - 1, j), & \text{if } d = L \\ &= (i + 1, j), & \text{if } d = R \\ &= (i, j - 1), & \text{if } d = D \end{aligned}$$

$$\begin{aligned}
 &= (i, j + 1), & \text{if } d = U \\
 &= (i, j), & \text{if } d = N
 \end{aligned}$$

As in Section 3.2.2, we shall require Q and V to be finite non-empty sets, where Q contains a special *initial state* q_0 , and V contains a special *blank symbol* $\#$. We shall also assume that only a finite number of tape symbols are non- $\#$ s, where $\# \in V$. (Further restrictions on the set of non- $\#$ s will be considered in a moment.) If these restrictions hold, we call M a *Turing machine* (TM). If M cannot create or destroy $\#$ s, we call it $\#$ -*preserving* ($\#P$); if M “bounces off” $\#$ s [i.e., $(q', v', d') \in \delta(q, \#, d)$ implies $v' = \#$ and $d' = d^{-1}$, where $L^{-1} = R$, $R^{-1} = L$, $U^{-1} = D$, $D^{-1} = U$], we call M *tape-bounded* (TB); and if M never rewrites any symbols, we call it *finite-state* (FS). There need be no confusion with the one-dimensional definitions, as long as we know which is meant.

As in Section 3.2.3, we define an *array acceptor* as a triple $A = (M, q_0, Q_A)$, where M is an array automaton, q_0 is M 's initial state, and Q_A is a set of *accepting states*. We say that A accepts the tape τ_0 from position (i_0, j_0) if repeated application of M 's transition function, starting from the configuration $(q_0, N, i_0, j_0, \tau_0)$, leads to a state set containing some $q \in Q_A$. The special types of acceptors (Turing, $\#P$, TB, FS) are also defined just as in Section 3.2.3. If Σ is the (finite) set of non- $\#$ s in τ_0 , we may also speak of A as accepting Σ .

In Chapter 3 we required that the initial tape τ of a Turing machine be of the form $\#^\infty \sigma \#^\infty$, where σ is a finite nonnull string of non- $\#$ s. As pointed out in Section 4.1, there are two possible generalizations of this requirement to two dimensions—namely, we can require that the set Σ of non- $\#$ s be rectangular in shape, or merely that Σ be connected. In Section 4.3 we shall assume that Σ is rectangular; the nonrectangular case will be considered in Section 4.4.

Before we begin our study of rectangular array acceptors, we show that an acceptor can verify that its non- $\#$ input array Σ (which we assume to be connected) is, in fact, rectangular. Indeed, we show that a deterministic, tape-bounded, finite-state array acceptor can do these things from any starting position; thus all of the stronger types can certainly do so.

PROPOSITION 4.2.1 There exists a DTBFSA A that accepts a connected array Σ of non- $\#$ s iff Σ is rectangular.

Proof: A moves leftward until it bounces off a $\#$, then upward until it again bounces off a $\#$; if Σ is rectangular, this puts A in the upper left corner of Σ . A now operates as follows:

- (a) Move down. If that point is $\#$, move back up and continue with step (b). If not, move left and check that a $\#$ is present. If it is not, reject; if it is, move right and up, and continue with step (b). (This verifies that the left border of Σ is locally straight at its upper end.)
- (b) Move up. If that point is not $\#$, reject; if it is $\#$, move back down, move right, and repeat the process. Continue until the rightward move hits a $\#$. (This verifies that the top border of Σ is straight.) Move back and continue with step (c).
- (c) Move down. If that point is $\#$, move back up and continue with step (d). If not, move right and check that a $\#$ is present. If it is not, reject; if it is, move left and up, and continue with step (e). (This verifies that the right border of Σ is locally straight at its upper end.)
- (d) Move left, move down, and check that a $\#$ is present. If it is not, reject; if it is, move back up and repeat the process. Continue until the leftward move hits a $\#$. (This verifies that the bottom border of Σ is straight.) If this step is ever reached, accept.
- (e) Move left until a $\#$ is hit. Move down. If that point is $\#$, move back up and continue with step (f). If not, move left and check that a $\#$ is present. If it is not, reject; if it is, move right and up, and continue with step (g). (This verifies local straightness of the left border.)
- (f) Move right, move down, and check that a $\#$ is present. If it is not, reject; if it is, move back up and repeat the process. Continue until the rightward move hits a $\#$. (This verifies that the bottom border is straight.) If this stage is ever reached, accept.
- (g) Move right until a $\#$ is hit. Move down. If that point is $\#$, move back up and continue with step (d). If not, move right and check that a $\#$ is present. If it is not, reject; if it is, move left and up, and continue with step (e). (This verifies local straightness of the right border.)

It is easily seen that A accepts Σ iff Σ is rectangular. Note also that

in the process of accepting a rectangular Σ , A does a raster scan of Σ , row by row, alternately left to right and right to left. $\square$

4.3 RECTANGULAR ARRAY ACCEPTORS

4.3.1 Rectangular Array Languages

If Σ is rectangular, it is easy to show, just as in Section 3.2.3, that in most cases the set of Σ s accepted by the acceptors of a given type is the same whether we require acceptance from all initial positions, from any initial position, or from some standardized initial position such as the upper left-hand corner of Σ . Specifically, given an acceptor A , let $L_{\cap}(A)$, $L_{\cup}(A)$, and $L_c(A)$ be the sets of Σ s accepted by A from all positions, from some position, and from the upper left corner of Σ , respectively. Then we can prove

PROPOSITION 4.3.1 For any A there exists an A' such that $L_{\cap}(A') = L_{\cup}(A') = L_c(A)$; if A is $\#P$, TB, or FS, or is deterministic, so is A' .

Proof: The proof is analogous to that of Proposition 3.2.1. From its initial position, A' moves leftward until it bounces off a $\#$, and then upward until it bounces off a $\#$, which puts it in the upper left corner of Σ ; it then simulates A . $\square$

PROPOSITION 4.3.2 For any A there exists an A'' such that $L_c(A'') = L_{\cup}(A)$; if A is $\#P$, TB, or FS, so is A'' .

Proof: The proof is analogous to that of Proposition 3.2.2. From its initial position in the upper left corner of Σ , A'' moves nondeterministically rightward and/or downward, and at any step (provided it is reading a non- $\#$ symbol), it can begin to simulate A . [Alternatively, A'' can do a systematic scan of Σ (see the proof of Proposition 4.2.1), and at any non- $\#$ step, can nondeterministically begin simulation of A .] $\square$

PROPOSITION 4.3.3 For any A there exists an A''' such that $L_c(A''') = L_{\cap}(A)$; if A is $\#P$, TB, or deterministic, so is A''' .

Proof: Here again the proof is analogous to that of Proposition 3.2.3: A''' does a systematic scan of Σ and simulates A (on first

terms of pairs) starting from each point of the scan. If the simulation accepts, A erases the traces of the simulation, marks the starting point, moves on to the next (unmarked) point of the scan, and repeats the process.[§] $\square$

By these propositions, it makes no difference, in most cases, whether we define acceptance from a standard point such as the upper left corner (or any point that A can locate), from all points, or from some point. We shall use the upper-left-corner definition from now on. The set $L_c(A)$ will be called the *language* of A , and will be denoted by $L(A)$. [Incidentally, as we shall see in Section 4.4, Propositions 4.3.2 and 4.3.3 hold for arbitrary connected Σ s, but Proposition 4.3.1 must be modified for such Σ s.]

4.3.2 The Language Hierarchy

Just as in the one-dimensional case, a rectangular array language accepted by a (deterministic) #PA, TBA, or FSA will be called a (D)#PL, (D)TBL, or (D)FSL, respectively. The classes of all such languages will be denoted by $\mathcal{L}_{(D)\#P}$, $\mathcal{L}_{(D)TB}$, or $\mathcal{L}_{(D)FS}$, and the class of all languages accepted by (deterministic) TAs will be denoted by $\mathcal{L}_{(D)T}$. In this section we shall establish some relationships among these classes of languages.

Determinism is not a restriction in the case of arbitrary TAs; in other words, $\mathcal{L}_{DT} = \mathcal{L}_T$. This can be shown in essentially the same way as in the one-dimensional case (see Section 3.3): Given an arbitrary TA A , we can construct a DTA A' that systematically simulates all possible sequences of transitions of A , so that A' accepts iff A does. It is an open question whether $\mathcal{L}_{DTB} = \mathcal{L}_{TB}$ (and similarly for #P), just as it is in one dimension.

In one dimension, we had $\mathcal{L}_{DFS} = \mathcal{L}_{FS}$, but this is false in two dimensions. The proof of the following theorem is based on [1]:

THEOREM 4.3.4 $\mathcal{L}_{FS} \supsetneq \mathcal{L}_{DFS}$.

Proof: Consider the set of square arrays Σ , having odd side length, composed of x s and y s. Note that a DFSA can recognize whether or not an array has these properties. [It can test for

[§]If A is not #P, erasing the traces of the simulation may not be trivial, since in the course of it, Σ may become non-rectangular. However, in Section 4.4.1 we shall show that a DTBA can in fact scan an arbitrary connected Σ .

squareness by moving diagonally—e.g., alternately down and right, beginning at the upper left corner, and verifying that it hits the bottom just at the lower right corner. It can test oddness of side length by moving along one side of the array and counting modulo 2, and can verify that the array consists of nothing but x s and y s by doing a raster scan.] We want to accept an array of the above type iff its center symbol is x .

A nondeterministic FSA can do this as follows: Starting at the upper left corner, move diagonally downward and to the right. At some point, nondeterministically chosen, memorize the symbol just read on the diagonal, and begin moving diagonally downward and to the left. If this reaches the lower left corner (i.e., it hits the bottom just at the corner), then the memorized symbol was at the center point of the array; if it is x , the FSA accepts. Evidently, acceptance occurs iff the center point is x .

To prove that a DFSA cannot do it, we first consider the behavior of such an FSA (call it A) relative to an $m \times m$ block (i.e., subarray) of x s and y s. A can enter the block at any one of $4m - 4$ positions, and can be in any one of $|Q|$ states when it enters. (It can also have just moved in either of two directions, if it enters at a corner; but this does not appreciably affect our argument.) Similarly, A can leave at $4m - 4$ places, in one of $|Q|$ states; or it can fail to leave, but we shall ignore this for the moment. The block thus defines a mapping from the $4(m - 1)|Q|$ incoming (position, state) pairs into the $4(m - 1)|Q|$ outgoing pairs. There are $[4(m - 1)|Q|]^{4(m - 1)|Q|}$ such mappings. On the other hand, there are $2^{(m^2)}$ possible blocks, and for sufficiently large m we have $[4(m - 1)|Q|]^{4(m - 1)|Q|} = 2^{4(m - 1)|Q| \log[4(m - 1)|Q|]} < 2^{(m^2)}$. Thus there must be two different blocks Σ'_1 and Σ'_2 that give rise to exactly the same mapping.

To complete the proof, suppose that we had an A that could accept the arrays whose center points are x s. We can assume that A does its accepting in a standard position, say at the lower right corner (given any A , we can simulate it until it accepts, then move to that corner and accept). Let (i, j) be a position in which $\Sigma'_1 = x$ and $\Sigma'_2 = y$ (or vice versa). Construct an array Σ_1 that contains Σ'_1 as a subarray, with its (i, j) point in the center of Σ_1 ; the rest of Σ_1 can be all x s (say). When A accepts Σ_1 , it starts and ends outside Σ'_1 . Hence A must also accept the array Σ_2 obtained from Σ_1 by replacing Σ'_1 by Σ'_2 ; but the center point of Σ_2 is y —a contradiction. $\square$

In one dimension (Section 3.3.1), we saw that $\mathcal{L}_{(D)\#P} = \mathcal{L}_{(D)TB}$ and $\mathcal{L}_{(D)TBFS} = \mathcal{L}_{(D)FS}$. We can show this in two dimensions for deterministic acceptors; the nondeterministic case is still open (see [2]).

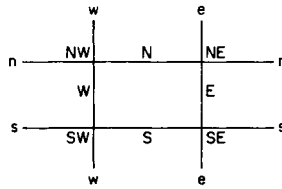
Note first that Proposition 3.3.3 and Corollary 3.3.4 hold in two dimensions as well as in one: The behavior of a DFSA on a constant input tape becomes periodic after at most $|Q|$ steps (where $|Q|$ is the number of states), with period at most $|Q|$, and the same is true for a D#PA while it is on the # part of its tape. We can also use these results to prove a two-dimensional analog of Proposition 3.3.5:

LEMMA 4.3.5 Let A be a D#PA that has $|Q|$ states, and suppose that A moves onto the # part of its tape, say from the right edge of the non-# array Σ . Let c be the column (of $I \times I$) that contains this edge of Σ . Then either A returns to c within $|Q|^2 + |Q|$ steps, or else it never returns.

We now now prove

THEOREM 4.3.6 $\mathcal{L}_{D\#P} = \mathcal{L}_{DTB}$; $\mathcal{L}_{DFS} = \mathcal{L}_{DTBFS}$.

Proof: Given any D#PA A , we can construct a DTBA A' that simulates A as long as it remains on Σ . When A leaves Σ , it must do so at one of the edges, and must enter one of the regions N, E, W, or S (see the accompanying figure), say region E. Let n, s be the



rows of $I \times I$ that contain the top and bottom of Σ , and let e, w be the columns of $I \times I$ that contain the right and left columns of Σ . Thus when A enters E, it does so from column e .

By Lemma 4.3.5, A either never returns to e , or it returns in at most $|Q|^2 + |Q|$ steps. In particular, if A ever gets farther than $\frac{1}{2}(|Q|^2 + |Q|)$ to the right of e , it can never return to e (and hence can never return to Σ). Thus A' can remain on e , starting at the

point where A left e , and keep track of the state changes of A using its own states, and of the left/right moves of A using an internal counter of capacity $\frac{1}{2}(|Q|^2 + |Q|)$. If this counter overflows without A having returned to e , A' can stop the simulation since A will never return. (Acceptance by A without returning will be discussed later.) At the same time, A' can simulate the up/down moves of A on the $\#$ s by moving up and down on e . As long as A remains in E , A' remains in Σ and no problems arise. If A returns to e (so that the counter becomes empty), A' is guaranteed to be in the proper state and at the point of e where A returns, and it can resume simulating A on Σ .

Suppose that A' leaves E , say by moving into NE . When this happens, A' crosses n , and A (simulating A' on e) reaches the northeast corner of Σ , where e and n intersect. At this point, A' can remain at the corner and simulate the up/down moves of A using a second internal counter of capacity $\frac{1}{2}(|Q|^2 + |Q|)$. By the analog of Lemma 4.3.5, if this counter ever overflows without A having returned to n , A will never return, and the simulation can stop. Thus as long as A remains in NE and does not move too far away from Σ to return, A' can stay at the corner and simulate all the moves of A using its two counters. If A leaves NE , it must enter either E or N , and A' detects this by noting that one of the counters is empty. If A enters E , the up/down counter is empty, and A' resumes simulating the up/down moves of A by moving up and down on e , as before. Similarly, if A enters N , the left/right counter is empty, and A' now simulates the left/right moves of A by moving left and right on n , as long as A remains in N .

The procedure is analogous if A enters any of the other $\#$ regions. Thus wherever A moves onto the $\#$ s, as long as it remains close enough to Σ so that return is still possible, A' can simulate A in such a way that if A does return, A' is in the proper position and in the proper state to resume simulating A on Σ .

If A accepts Σ without returning to it, this must happen within $|Q|$ steps, since the behavior of A on the $\#$ s is periodic with period at most $|Q|$. Thus this will always happen before the counters of A' have overflowed. If A' detects the fact that A has entered an accepting state, A' can accept immediately, without continuing the simulation. Thus in any case, A' accepts iff A does, which proves the first part of the theorem.

Note finally that in our simulation of A by A' , if A is FS, so is A' , which proves the second part of the theorem. $\square$

We can now also prove

THEOREM 4.3.7 $\mathcal{L}_{(D)T} \supsetneq \mathcal{L}_{TB}; \mathcal{L}_{DTB} \supsetneq \mathcal{L}_{DFS}.$

Proof: We know that these results are true in one dimension (see Section 3.3.2). Let L be a set of strings that is in $\mathcal{L}_T$ but not in $\mathcal{L}_{TB}$ (in one dimension). If we regard L as a set of $n \times 1$ arrays, then clearly no two-dimensional TBA can accept $\mathcal{L}$, since this TBA remains on one row and can thus be simulated by a one-dimensional TBA. On the other hand, a two-dimensional (D)TA can accept L by simulating a one-dimensional TA that accepts L .

Similarly, let L be a set of strings that is in $\mathcal{L}_{DTB}$ but not in $\mathcal{L}_{DFS}$ (in one dimension), and regard L as a set of $n \times 1$ arrays. If L were in $\mathcal{L}_{DFS}$ in two dimensions, it would be in $\mathcal{L}_{DTBFS}$ (Theorem 4.3.6), so that a two-dimensional DTBFSA would accept it; but this DTBFSA could be simulated by a one-dimensional DTBFSA—a contradiction. $\square$

4.3.3 Three-Way Acceptors

We can also define two-dimensional analogs of the one-way acceptors treated in Section 3.3.3. Specifically, let us require that A start at the upper left corner of its input array, and allow A to move right, left, and down, but not up. Such an A can still do a systematic scan of its array, but it can no longer verify that the array is rectangular. [In fact, A cannot even tell whether two consecutive rows have the same length; if it verifies that they line up at the left end, it can no longer do so at the right end.] We shall call such an A a *three-way acceptor* (TWA).

Evidently, on an $n \times 1$ array (confined to a single row), a TWA of a given type (Turing, #P, TB, FS) has the same power as a one-dimensional acceptor of the corresponding type and can be simulated by such an acceptor. (If the TWA ever leaves that row, it can never return to it; thus from then on it reads nothing but #s and can be simulated by a one-dimensional acceptor that remains on the row.) Hence any proper inclusion that holds for the languages of one-dimensional acceptors also holds for TWA languages.

On the other hand, on a $1 \times n$ array (confined to a single col-

umn), a TWA has the same power as a one-dimensional (D)FSA, by the same argument as in Section 3.3.3. It follows that a (four-way) two-dimensional acceptor of any type stronger than an FSA is strictly stronger than a three-way acceptor of the corresponding type. This is also true for FSAs, as we see from

PROPOSITION 4.3.8 $\mathcal{L}_{(D)FS} \supsetneq \mathcal{L}_{(D)TWFS}$.

Proof: Consider the set of $n \times 2$ (i.e., two-row) arrays, say of x s and y s; we want to accept such an array iff the two rows are identical. A DFSA can verify this by systematically checking that each symbol on the top row has an identical symbol below it. On the other hand, a TWFSFA A that has $|Q|$ states must leave the top row in one of $|Q|$ states at one of n positions. For a given top row r , let Q_r be the set of (state, position) pairs in which A leaves r . (If A is deterministic, Q_r is a singleton.) If the bottom row is r , at least one such pair (q, i) must result in acceptance (after A has examined the bottom row). Thus (q, i) cannot be in $Q_{r'}$ for any $r' \neq r$, so that each Q_r contains a pair that is in no other Q_r . But there are only $n|Q|$ pairs, and there are 2^n different rows, so this is impossible. $\square$

We can also show, as in one dimension, that $\mathcal{L}_{(D)TW\#P} = \mathcal{L}_{(D)TWTB}$ and $\mathcal{L}_{(D)TWFS} = \mathcal{L}_{(D)TWTBFS}$. In fact, let A be a $TW\#PA$, and let A' be a TWTBA that simulates A on each row. (On any given row, as long as A does not move down, A' can simulate it by the argument in Section 3.3.1.) If A moves down while on Σ , A' also moves down. Moreover, while simulating the moves of A off Σ , A' can determine in which states A can move down; if the simulation enters such a state, A' can move down to the end of the row below and continue the simulation. As long as A does not move below the bottom row of Σ , A' can still determine in which states A can return to Σ , as in Section 3.3.1. If A does move below the bottom row, A' can stop the simulation and determine whether A can accept while on $\#s$. In this simulation, if A is deterministic or finite-state, so is A' .

On the other hand, nondeterministic TWFSAs are strictly stronger than deterministic ones, and nondeterministic TWTBAs are strictly stronger than those that are deterministic:

PROPOSITION 4.3.9 $\mathcal{L}_{\text{TWFS}} \supsetneq \mathcal{L}_{\text{DTWFS}}; \mathcal{L}_{\text{TWTB}} \supsetneq \mathcal{L}_{\text{DTWTB}}.$

Proof: Consider the set of arrays of x s and y s that contain two vertically adjacent x s. A nondeterministic TWFS can accept this set by scanning its input array systematically and, when reading an x , nondeterministically moving down and accepting if it finds another x ; evidently, it accepts iff there exist two vertically adjacent x s. On the other hand, let A be a DTWTBA having $|Q|$ states. Then A must leave the top row of its input array at one of n positions and in one of $|Q|$ states. Since there are 2^n different top rows, there must exist two top rows r_1 and r_2 that A leaves in the same state and at the same point. Let j be a position in which r_1 and r_2 differ; say r_1 has an x in the j th position, while r_2 has a y . Consider the two-row array whose top row is r_1 or r_2 and whose bottom row is all y s except for an x in the j th position. Then if A accepts the first of these two-row arrays, it also accepts the second. $\square$

4.4 CONNECTED ARRAY ACCEPTORS

4.4.1 Traversal Theorems

If Σ is an arbitrary connected array, the dependence of acceptance on the initial position of the acceptor becomes a more complicated problem; the difficulty is that location of a standard initial position by the acceptor is no longer easy. In fact, it is not even obvious that an acceptor can visit all of Σ from a given starting point, so that the acceptor may not even be able to reach the standard position. We therefore begin this section by proving some results about the ability of array automata of various types to traverse their input arrays.

PROPOSITION 4.4.1 There exists a DTA that, from any starting point P on its input array Σ , can scan Σ completely and return to P with all marks made on Σ in the course of the scan erased.

Proof: Starting at P , the DTA A moves in a rectangular "spiral," e.g., in the sequence shown on top of the next page, and marks the points of the spiral, whether they are non- $\#$ or $\#$. Moving in a spiral is accomplished as follows: At any step, A is in a rightward, upward, leftward, or downward moving state. When

16	15	14	13	12
17	4	3	2	11
18	5	P	1	10
19	6	7	8	9
20	21	.	.	.

it is in a rightward moving state, at every step it checks to see whether the point above it is marked, and if so, moves right. If the point above it is unmarked, it stays at that point and goes into an upward moving state. The behavior of A while in the other three types of states is analogous: When it is in an upward (leftward, downward) moving state, it checks the point on its left (below it, on its right) at each step; if this point is marked, it moves up (left, down), but if not, it stays at that point and goes into a leftward (downward, rightward) moving state.

Since Σ is finite, there will eventually be a complete turn of the spiral that finds only $\#$ s. A detects this event using an internal counter that is initially set at 0 and can count up to 5. Whenever A changes direction and is at a $\#$, it adds 1 to the counter, but it resets the counter to 0 whenever it reads a non- $\#$. Thus the counter reaches 5 iff A has changed direction four consecutive times without seeing a non- $\#$. At this point, the marked points form a rectangle T that entirely contains Σ . (No point P' of Σ can be outside T , since there would have to be a path of non- $\#$ s between P' and P , and this path would have to cross the border of T .) Thus A has now seen all of Σ . If desired, A can now retrace the spiral in reverse and erase all the marks (or at least erase them from the non- $\#$ s, if A is not allowed to create $\#$ s). When A finishes retracing the spiral, it has returned to P . $\square$

Proposition 4.4.1 describes a relatively simple method of scanning Σ that can be implemented by a DTA because of its ability to move on $\#$ s and to mark them. A DTBA can also scan its input array Σ , but it must use a more complicated method, as we shall see next.

THEOREM 4.4.2 There exists a DTBA that, from any starting point on its input array Σ , can scan Σ completely and return to P with all marks made in the course of the scan erased.

Proof: The DTBA A can use a graph traversal algorithm to scan Σ (see [3] for the graph version). A uses five marks, u, v, x, y , and z , which are mutually exclusive but which do not interfere with the original symbols of Σ . Initially, it marks its starting point P_0 with u and then proceeds as follows:

- (a) If the current point P has an unmarked non- $\#$ neighbor, mark P with x and move to such a neighbor, say P' . If not, go to step (b).
 - (1) If P' has a neighbor marked u , mark P' with y , move back to P (it is the unique neighbor of P' marked x), mark P with u , and return to step (a).
 - (2) If P' has no neighbor marked u , mark P' with z , move back to P , erase all y marks from neighbors of P , mark P with u , move back to P' (it is the unique neighbor of P marked z), mark P' with u , and return to step (a).
- (b) If P has no unmarked non- $\#$ neighbor, erase all y marks from neighbors of P and mark P with v .
 - (1) If P has a neighbor marked u , go to such a neighbor and return to step (a).
 - (2) If not, stop.

We observe first that step (b2) must be reached eventually. In fact, the algorithm cannot stay in step (a) indefinitely, since at each pass through (a), a previously unmarked point P' of Σ is marked either y or u , and this can only happen a finite number of times. Moreover, at each entry into step (b), some point P is marked v , and this too can only happen a finite number of times.

Next we show that at each entry into step (a) the points marked u form an arc $P_1, \dots, P_k$ ($k \geq 1$) with $P_1 = P_0$ and $P_k = P$. [Recall (Section 2.3) that an arc does not cross or touch itself; i.e., P_j is a neighbor of P_i iff $j = i \pm 1$.] This is clearly true at the first entry into (a), since P_0 was initially marked u and there are as yet no other marks. Furthermore, if these properties are true at a given entry into (a), they are still true at the next entry. Indeed,

- (a1) If the next entry is via (a1), the set of u s is unchanged (P was initially marked u , (a) changed its mark to x , and (a1) changed it back to u).
- (a2) If the next entry is via (a2), a new point Q' marked u is added to the path; P' has no neighbor marked u (except

for P after its x is changed back to u), so the path still does not touch or cross itself.

- (b1) If the next entry is via (b1), the mark on P is changed to v , and the new P is a neighbor of P marked u (if any); by induction hypothesis, this P ($= P_{k-1}$) is unique, so the path has been shortened by one point and still does not touch or cross itself.

Evidently, (b2) can hold only if the path has length 1 ($P = P_k = P_1 = P_0$) since otherwise P does have a neighbor marked u ; thus when the algorithm stops, the current point is P_0 , and there are no more points marked u .

Finally, we show that when the algorithm stops, every point of Σ has been marked v . Note first that when it stops, there can be no points marked x , y , z , or u , and there is at least one point (P_0) marked v . Suppose there were an unmarked point; let P be such a point whose distance from P_0 , as measured by the length of a shortest path in Σ from P_0 to P , is as small as possible. Thus P has a neighbor P' (the previous point on the path) that is closer to P_0 than P , so that P' must be marked v . Let P'' be the neighbor of P that was last marked v by the algorithm. By (b), just before P'' was marked v , it had no unmarked neighbors. Since P cannot have been marked u (u s must eventually be changed to v s), it must have been marked y at that time. But by (a1), for P to have been marked y , it must have had a neighbor P''' other than the current point (P'') that is marked u . Thus when P'' was marked v , P still had a neighbor P''' marked u . This u eventually was changed to v by the algorithm, so that P'' was not the last neighbor of P to be marked v —a contradiction.

The foregoing discussion shows that the algorithm, starting at P_0 on an unmarked array Σ , stops at P_0 with every point of Σ marked with v . It follows that we can modify the algorithm, by interchanging the roles of points marked v and unmarked points, so that when it starts at P_0 on an array having every point marked v , it stops at P_0 with every point unmarked. Thus the algorithm and this modification of it define the desired DTBA, and the proof is complete. $\square$

Another DTBA traversal theorem will be given in Section 4.4.2 in the course of proving that a DTBA can distinguish the outer

border and the hole borders of its input array Σ , and can find, e.g., the leftmost of the uppermost points of the outer border.

A nondeterministic TBFSFA can traverse all of Σ by simply moving nondeterministically from neighbor to neighbor. However, it can never know when it has finished; i.e., it can never correctly make a state change that depends on its having seen all of Σ , and it cannot recognize its starting point if it moves too far away from that point. The proof of the following result is based on [4].

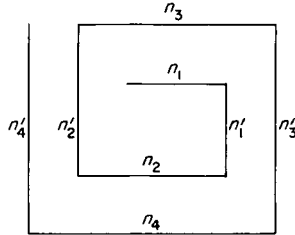
THEOREM 4.4.3 There exists no TBFSFA that, from any starting point on any input array Σ , accepts when and only when it has visited all of Σ .

Proof: Suppose A were such a TBFSFA, say having $|Q|$ states. Let us first consider the behavior of A on $n \times 1$ (i.e., one-row) arrays that are surrounded by $\#$ s except possibly at their ends. A can enter such an array at either end in any of $|Q|$ states, and it can leave at either end in any of $2^{|Q|}$ sets of states. Thus the number of possible mappings from entering end and state to leaving end and state set is $(2 \cdot 2^{|Q|})^{2|Q|}$. Since this number does not depend on n , there must be many n s that yield the same mapping, and in particular, there must be at least one infinite set of n s, say $N = \{n_1, n_2, n_3, \dots\}$, with $n_1 < n_2 < n_3 < \dots$, that yield the same mapping. A similar argument, using $1 \times n$ (one-column) arrays, shows that there exists an infinite set $N' = \{n'_1, n'_2, \dots\}$ of lengths all of which yield the same input/output mapping for A .

Since for any Σ , A accepts iff it has seen all of Σ , this is true in particular when Σ is a hollow rectangle one point thick. Let us choose such a rectangle T that has horizontal side length $n_i \in N$ and vertical side length $n'_j \in N'$. By the definitions of N and N' , the behavior of A on T is the same no matter which $n_i \in N$ and which $n'_j \in N'$ we use to define T , in the sense that if A starts in state q_0 at some point of T , the (sets of) states in which A enters and leaves the various sides of T are the same for all such T s. Since A accepts such an T in a finite time, A can only make a finite number m_1 of clockwise circumnavigations of T , and a finite number m_2 of counterclockwise circumnavigations, relative to its starting point, before accepting; these bounds are valid for all such T s.

Let S be the rectangular spiral constructed from a succession of horizontal and vertical line segments of lengths $n_1, n'_1, n_2, n'_2,$

$\dots, n_{4m}, n'_{4m}$, where $m = \max(m_1, m_2)$; e.g., the first two turns of S might look like the accompanying figure. (If two consecutive



n_i s or n'_j s differ by 1, we can skip one of them to insure that the turns of S never touch one another.) Suppose that A starts out in state q_0 from a position on the m th turn of S , i.e., from one of the segments $n_{2m-1}, n_{2m}, n'_{2m-1}$, or n'_{2m} . By construction of S , the (sets of) states in which A enters or leaves the various sides of S are the same as those for the T s as long as A never gets to an end of S . But in fact, A accepts the T s after having made at most $\max(m_1, m_2)$ turns around T , so that, in fact, A never reaches an end of S . Thus A accepts S without ever having seen its ends—a contradiction. $\square$

4.4.2 Connected Array Languages

We can apply the traversal results obtained in Section 4.4.1 to show that the sets of Σ s accepted by the arbitrary or the tape-bounded acceptors is usually the same whether we require acceptance from all initial positions, from any initial position, or from some standard initial position such as the "upper left corner" (leftmost of the uppermost points) of Σ . Let $L_\cap(A)$, $L_\cup(A)$, and $L_c(A)$ be defined as in Section 4.3.1. Then we can prove

PROPOSITION 4.4.4 For any A there exists an A' such that $L_\cap(A') = L_\cup(A') = L_c(A)$; if A is $\#P$, TB, or is deterministic, so is A' .

Proof: Suppose first that A is not $\#P$ or TB. From its initial position, A' marks a rectangular spiral that contains Σ , as in the proof of Proposition 4.4.1. By moving to the upper left corner of this spiral and then doing a row-by-row scan, A' can easily find the upper left corner of Σ . A' then simulates A , but ignoring the marks

on the points of the spiral. Clearly, A' accepts iff A does, and if A is deterministic, so is A' .

To handle the case where A is $\#P$ or TB, we must show that there exists a DTBA that can find the upper left corner of its input array. This is shown in the proof of Theorem 4.4.5 immediately below. $\square$

If A is FS, the proof of Proposition 4.4.4 breaks down; in fact, there does not exist a TBFSFA that can find the upper left corner of its input array. Indeed, suppose A' were a TBFSFA that accepted when it found this corner. Construct the hollow rectangle T and rectangular spiral S for A' as in the proof of Theorem 4.4.3, with the number of turns of S chosen to exceed the number of turns that A' makes on T before accepting at T 's upper left corner. Then if we start A' in the middle of S , it accepts at some northwest corner of S without ever having found the real upper left corner of S .

THEOREM 4.4.5 There exists a DTBA that, from any starting point on its input array Σ , can find the upper left corner of Σ .

Proof: The proof makes extensive use of the topological properties of arrays studied in Chapter 2, and in particular, of the border-following algorithm BF defined in Section 2.6. We first show that our DTBA A can find the *outer border* of Σ ; this is the border along which Σ is adjacent to the background component of $\bar{\Sigma}$, as distinguished from *hole borders* along which Σ is adjacent to other components of $\bar{\Sigma}$ (if any).

To find the outer border of Σ , A operates as follows:

- (a) A moves up until it bounces off a $\#$.
- (b) When this happens, A has hit a border of Σ . A now marks its position, say with the mark x , and proceeds to follow the border, using essentially the border-following algorithm BF of Section 2.6. (A can evidently check the neighbors of the x by making a sequence of local moves, and can determine which of these neighbors is the next border point specified by BF.) The point x may be on several borders of Σ , but the border that A follows is the one defined by the component of $\#$ s containing the $\#$ just above x .
- (c) During border following, A keeps track of its net up/down

moves by moving another marker y along the border. Specifically, whenever A moves down, it marks its current position on the border (say with z), follows the border back to y (or to x , if y has not yet been used), erases y , and rewrites it one step further along the border. A then returns to z , erases it, and resumes following the border. (Note that y can never catch up with z , since the number of downward moves made by A cannot exceed its total number of moves along the border.) Similarly, whenever A moves up, it marks its position with z , goes back to y , erases it, rewrites it one step further back along the border, returns to z , erases it, and resumes border following. A knows which $\#$ neighbor of the y or z point is the current $\#$ neighbor being used in the BF algorithm; note that the border may pass through some points twice, so that A must be able to distinguish between two positions along the border even if they are at the same point of Σ .

- (d) If y is in the same position as x when A comes to move it further back, or if A moves up before y is created, the current position of A on the border (as marked by z) is higher up than the point at which A hit the border (as marked by x). If this happens, A erases x and y , returns to z , erases z , and goes back to step (a).
- (e) At each entry into step (a), A is higher up than it was at the previous entry; hence steps (a)–(d) cannot keep repeating indefinitely, since Σ is finite. Thus eventually, while executing steps (b) and (c), A will get all the way around the border and back to x without case (d) occurring. This means that x is at a highest point of the border. Now the highest points of a hole border are evidently the points of Σ just above the highest points of the hole, and cannot have points of the hole above them. Since x does have above it a $\#$ belonging to the component of $\#$ s whose border A has been following [see steps (a) and (b)], this component cannot be a hole. Thus when case (d) fails to occur, x is on the outer border of Σ —indeed, on an uppermost point of this border.

To find the leftmost of the uppermost points on the outer border, A next moves left until it hits a $\#$, which puts it at a local “upper left corner” P on the top row of Σ . A marks P with x and

follows the outer border again, starting out leftward, using z to mark its current position and y to keep track of its net number n of downward moves, as in the first part of the proof. Let P' be the position of A the first time $n = 0$. If $P' = P$ (i.e., if x is there), P is the only uppermost point of Σ and we are done. Otherwise, if P is not the desired upper left corner of Σ , it is easily seen that P' must belong to a run of uppermost points of Σ that lies to the left of P . To check this, A marks P' with z , returns to P (erasing y), erases x , and follows the outer border again, this time using the relative positions of x and y to determine its net number m of left/right moves. [This is done as follows: Each time x is moved one step along the border (to mark the current position of A), y is moved

two steps, if x made a rightward move;
 one step, if x made neither a rightward nor a leftward move; or
 no steps, if x made a leftward move.

Thus the number of steps by which y is ahead of x along the border is the number of rightward moves made by x .] When x reaches P' , if $m < 0$, P' is to the left of P . A can then erase x and y , return to P' , erase z , and move left until it hits a $\#$, which puts it at another local upper left corner P'' on the top row of Σ that is to the left of P . A then takes P'' as the new P and repeats the process. Eventually (since Σ is finite) it must turn out that P'' is not to the left of P , so that P is the desired upper left corner. In that case, A moves back along the border from P' toward P , using y and z to keep track of its net number of downward moves n , as above. The first time that $n = 0$, A has returned to P , which is the upper left corner of Σ . $\square$

We shall show in Chapter 7 (Theorem 7.1.5), using similar methods, how to construct a DTBA that systematically scans its input array, row by row, using only four marks. A less parsimonious construction of such a DTBA is given in [5].

Returning to the relationship between $L_{\cap}$, $L_{\cup}$, and L_c , we next prove

PROPOSITION 4.4.6 For any A there exists an A'' such that $L_c(A'') = L_{\cup}(A)$, and if A is $\#P$, TB, or FS, so is A'' .

Proof: Analogous to the first proof of Proposition 4.3.2. $\square$

PROPOSITION 4.4.7 For any A there exists an A''' such that $L_c(A''') = L_c(A)$, and if A is $\#P$, TB, or deterministic, so is A''' .

Proof: Analogous to that of Proposition 4.3.3, using Proposition 4.4.1 or Theorem 4.4.2. $\square$

The analog of Proposition 4.4.7 for TBFSAs is false. Let L be the language consisting of all arrays Σ that fail to contain a particular symbol x . Evidently, we have $L_x = L_c(A)$, where A is the TBFSa that accepts from state q_0 if it reads any symbol other than x ; thus A accepts Σ from every starting position iff Σ contains no x s. On the other hand, suppose that we had $L_x = L_c(A''')$ for some TBFSa A''' . In analogy with the proof of Theorem 4.4.3, we can construct a double rectangular spiral S of the form shown in the accompanying figure on which A''' behaves in the same way as on a hollow



rectangle T . If T contains no x s and A''' starts at the upper left corner of T , A''' must eventually accept T , without making, say, more than m turns around T in either direction. If we choose the two spirals in S to have more than m turns, A''' must thus accept S without ever reaching its ends, so that it accepts S even if there are x s at its ends—a contradiction.

By Propositions 4.4.4, 4.4.6, and 4.4.7, it still makes no difference, in most cases, whether we define acceptance from a standard point, such as the upper left corner, from all points, or from some point. We shall use the upper-left-corner definition from now on. The set $L_c(A)$ will be called the *language* of A and will be denoted by $L(A)$.

4.4.3 The Language Hierarchy

We use the same terminology and notation for connected array languages as we did for rectangular array languages in Section 4.3.2.

Just as for rectangular arrays, we have $\mathcal{L}_{DT} = \mathcal{L}_T$ and $\mathcal{L}_{FS} \supsetneq \mathcal{L}_{DFS}$, while the properness of $\mathcal{L}_{TB} \supsetneq \mathcal{L}_{DTB}$ is an open question (and similarly for $\#P$).

We can still show for connected arrays that $\mathcal{L}_{D\#P} = \mathcal{L}_{DTB}$; but it is no longer true that $\mathcal{L}_{DFS} = \mathcal{L}_{DTBFS}$. In the remainder of this section, we prove these results. The proof of the former is based on [2]; that of the latter is taken from [6].

As in Section 4.3.2, the behavior of a $D\#PA$ A while on the $\#$ part of its tape becomes periodic after at most $|Q|$ time steps, with period at most $|Q|$. Let σ_0 be the string of U s, D s, L s, and R s that defines the nonperiodic part of A 's movement on the $\#$ s, and let σ be a string that similarly defines a single period of A 's periodic movement. Suppose that A leaves Σ at the point P , and let β be the border along which Σ meets the component of $\#$ s into which A has moved. If A ever reenters Σ , this must happen at some point P' of β .

If the net displacement of A over an entire period σ is zero, A must reenter Σ (if at all) either during σ_0 or during the first period σ , i.e., in less than $2|Q|$ time steps, so that the behavior of A can be simulated internally by a DTBA that remains on Σ . Suppose that the net displacement is $(m, n) \neq (0, 0)$. Then if A reenters during the k th period σ , k cannot exceed the length $|\beta|$ of β by more than a bounded amount, since the reentry point P' is on β . Thus A reenters in at most about $|Q||\beta|$ time steps, if it reenters at all.

It follows that the behavior of A while on the $\#$ s can be simulated by a DTBA A' that uses β as a tape. In particular, A' can use β to compute the position of A relative to P , as well as the state of A , at each of the $|Q||\beta|$ time steps. For each of these positions, A' can also follow the border β , compute the displacement of each border point relative to P (see the proof of Theorem 4.4.5), and check whether this displacement is the same as the given position of A . If so, A' can mark that border point, erase all other traces of its computations on β , go to the marked point, erase the mark, enter the appropriate state, and resume simulating A on Σ . If none of the positions of A during the first $|Q||\beta|$ time steps coincides with a point of β , A will never return to Σ . (If A accepts Σ without returning to it, this must happen within $2|Q|$ time steps, so that A' can ascertain this fact as soon as A leaves Σ .) This discussion has established

THEOREM 4.4.8 $\mathcal{L}_{DTB} = \mathcal{L}_{D\#P}$.

On the other hand, we shall now show that $\mathcal{L}_{DTBFS} \supsetneq \mathcal{L}_{DFS}$. To

this end, consider the set of thin (width 1) upright L-shaped arrays of, say, x s on a background of $\#$ s. We first show that this set is accepted by a DTBFSA A . In fact, A moves left until it bounces off a $\#$, then moves up until it bounces off a $\#$. A then verifies that it has $\#$ s on its left and right, moves down, and repeats the process. If at some stage A finds an x on its right, it verifies that there are $\#$ s on its left and below it and moves to the x on its right. Here A verifies that there are $\#$ s above and below, moves right, and repeats the process, until its rightward movement bounces off a $\#$. If all of these verifications are successful, A accepts; if any verification fails, A stops and does not accept. Evidently, A accepts iff its input array is as described above.

We next consider the set of such L-shaped arrays for which the arms have equal length, and show that this set is accepted by a DFSA A' . In fact, A' first verifies that its input is L-shaped, as in the preceding paragraph. It then moves "diagonally" across the $\#$ s; i.e., it moves alternately to the left and upward. If, on an upward move, A' hits an x , it verifies that this x is at the upper end of the L and accepts. Evidently, this happens only if the arms of the L are equal. Note that if the vertical arm is shorter than the horizontal arm, A' will move diagonally forever without accepting.

Finally, we show that no DTBFSA can accept just the equal-armed Ls. Suppose that A'' were such a DTBFSA. Consider the set of one-dimensional arrays of the form $u^m v u^n$, and let A''' be a DTBFSA that behaves on these arrays exactly as A'' behaves on the Ls, except that when A''' is to the left of the v , its moves are 90° rotations of those of A'' (so that when A'' moves up, A''' moves left, and so on). If A'' accepts just the equal-armed Ls, then A''' accepts just the strings $u^m v u^n$ for which $m = n$. We could thus define a one-dimensional FSA A''' that accepted exactly these strings; but this is impossible (see Section 3.3.2). We have thus proved

THEOREM 4.4.9 $\mathcal{L}_{\text{DTBFS}} \supsetneq \mathcal{L}_{\text{DFS}}$.

As in Section 4.3.2, we still have

THEOREM 4.4.10 $\mathcal{L}_{(\text{D})\text{T}} \supsetneq \mathcal{L}_{\text{TB}}; \mathcal{L}_{\text{DTB}} \supsetneq \mathcal{L}_{\text{DFS}}$.

The proof is essentially the same as that of Theorem 4.3.7, using $n \times 1$ arrays, on which DTBFSA's can simulate DFSA's.

“Three-way” acceptors (see Section 4.3.3) will not be studied for connected arrays; it is evident that they would be very weak.

4.5 CLOSURE PROPERTIES

In this section we consider two-dimensional generalizations of the closure properties discussed in Section 3.5.

A finite intersection of TLs or TBLs is a language of the same type; the proof is analogous to that of Proposition 3.5.1. Note that in this proof, after completing each simulation, the automaton A must restore the original array and return to its starting point (the upper left corner) before beginning the next simulation; this requires that A be able to scan the non- $\#$ s and to find the upper left corner, and we have established that TLs and TBLs have these capabilities. A finite intersection of rectangular FSLs is an FSL, as in Proposition 3.5.1, since no restoration of the original array is necessary, and it is trivial to find the starting point. We leave open the question of whether a finite intersection of connected FSLs is an FSL.

A finite union of TLs, TBLs, or FSLs is a language of the same type. As in the case of Proposition 3.5.2, we can prove this nondeterministically, or we can give a deterministic proof for (TLs or) TBLs: A scans Σ and updates the position and state of each simulation. We leave open the question of whether a finite union of DFSLs is a DFSL, as well as the question of whether the complement of an FSL is an FSL.

Any singleton $\{\Sigma\}$ is a DFSL, since we can always design a DFSA that goes through a specific sequence of moves and state changes and verifies that its input array is exactly Σ . By a similar argument, any finite set of arrays is a DFSL. A subset of an FSL is not necessarily an FSL; we can use the same example as in Section 3.5. The set of rectangular arrays that contain, or fail to contain, a given connected subarray Σ_0 is a DFSL, since we can design a DFSA that systematically scans the rectangle, and at each position checks whether Σ_0 is present. On the other hand, the set of connected arrays that fail to contain a given Σ_0 (or even a given symbol x) is not a DTBFSL, as shown at the end of Section 4.4.2. The set of connected arrays that contain Σ_0 is a nondeterministic TBFSL (the FSA moves around nondeterministically, and at each position, checks for Σ_0); we leave open the question of whether it is a D(TB)FSL.

Cyclic closure, closure, and concatenation operations cannot readily be defined for connected arrays, but we can define them for rectangular arrays in both the horizontal and vertical directions. It is clear that the row or column reversal of a TL, TBL, or FSL is a language of the same type, just as in Proposition 3.5.5, and the same is true when both rows and columns are reversed. By a similar argument, we can easily see that the set of reflections (in a horizontal or vertical axis, or both), or the set of 90° (or 180° or 270°) rotations of a given connected array language is a language of the same type.

The cyclic row closure or column closure (or both) of a TL or TBL is a language of the same type, as in Proposition 3.5.6. Here the TBA nondeterministically marks a row and/or a column of the given array Σ , and simulates an acceptor for the given language, using the marks to define a cyclic shift. Similarly, the TA dissects Σ at the marks and creates an “unshifted” copy of Σ on which to do the simulation. We leave open the question of whether cyclic closures of rectangular array FSLs are FSLs.

In defining the concatenative row (or column) closure of a rectangular array language, or the concatenation of a set of such languages, we must assume that the numbers of rows (or columns) all match, so that the resultant array is still rectangular. These operations preserve TLs or TBLs, as in Propositions 3.5.7 and 3.5.8; we can nondeterministically segment a given rectangular array by marking a set of columns (or rows), and then check that each segment is in the appropriate given language.

The FSLs are *not* closed under (row or column) concatenation or closure [7]. To prove this, we first make an observation about the behavior of an FSA A with respect to a given (horizontal or vertical) slice of a rectangular array Σ . (Compare the discussion of blocks in the proof of Theorem 4.3.4.) Let Σ' be a subarray of Σ consisting of, say, m consecutive rows where each row has length n . A can enter Σ' at its top row at any of n positions and in any of $|Q|$ states, and can leave Σ' at its bottom row at any of 2^n sets of positions and in any of $2^{|Q|}$ sets of states (or n and $|Q|$, if A is deterministic). Thus the number of mappings from (entering position, entering state) into (set of leaving positions, set of leaving states) is $(2^{n+|Q|})^{n|Q|} = 2^{(n+|Q|)n|Q|}$; note that this number is independent of m , the number of rows. On the other hand, the number of possible rows, say made up of two symbols x, y , is 2^n , and the number of possible nonempty sets of rows is $2^{(2^n)} - 1$. For large, n , 2^n is

greater than $(n + |Q|)n|Q|$, so that there are more sets of rows than there are mappings, and there must exist two subarrays having different sets of rows that give the same mapping for a given A . (Of course, these sets of rows may have to be quite large; i.e., m may have to be as large as 2^n .)

We can now prove

PROPOSITION 4.5.1 The set of rectangular FS languages is not closed under horizontal or vertical concatenation.

Proof: The set of all rectangular arrays on, say, x and y is obviously FS, and so is the set of all such arrays whose top and bottom rows are the same (A does a column-by-column scan and checks that the first and last elements of each column are the same). We shall now show that the vertical concatenation of these two languages is not FS. (An exactly analogous proof can be used for horizontal concatenation.) Note that this concatenation is just the set of arrays L in which the bottom row is the same as some nontop row. For any FSA A let Σ_1 and Σ_2 be rectangular arrays that have different sets of rows R_1 and R_2 , but that yield the same mapping for A , and let r be a row in R_1 but not in R_2 . Let Σ'_1 (Σ'_2) be Σ_1 (Σ_2) with some arbitrary row ($\neq r$) appended at the top and with r appended at the bottom. Thus Σ'_1 is in L but Σ'_2 is not. We can assume without loss of generality that A starts at the top row and moves down to the bottom row of its input array before accepting. Thus if A accepts Σ'_1 , it also accepts Σ'_2 , so A cannot accept exactly L . $\square$

A similar argument can be used to prove

PROPOSITION 4.5.2 The set of rectangular FS languages is not closed under horizontal or vertical closure.

Proof: Let Σ be an array obtained by vertically concatenating

- (a) an arbitrary array of x s and y s,
- (b) a row of z s, and
- (c) an array of x s and y s whose top and bottom rows are identical.

Clearly, the set L of all such arrays is FS. On the other hand, it is not hard to see that the vertical closure L^+ of L is not FS, by an

argument analogous to that used in the proof of Proposition 4.5.1: In the zones between the rows of zs , we must now verify that the top row is identical to some nonbottom row, which cannot be done by an FSA. $\square$

REFERENCES

1. M. Blum and C. Hewitt, Automata on a 2-dimensional tape, *Proc. SWAT, 8th* pp. 155–160 (1967).
2. D. L. Milgram, A region crossing problem for array-bounded automata, *Information and Control* **31**, 147–152 (1976).
3. D. L. Milgram, Web automata, *Information and Control* **29**, 162–184 (1975).
4. J. Mylopoulos, On the recognition of topological invariants by 4-way finite automata, *Comput. Graphics Image Processing* **1**, 308–316 (1972).
5. D. L. Milgram and A. Rosenfeld, Array automata and array grammars, "IFIP Congress 71." North-Holland, Amsterdam, pp. 69–74 (1972).
6. A. Rosenfeld, Some notes on finite-state picture languages, *Information and Control* **31**, 177–184 (1976).
7. K. Inoue, A. Nakamura, and I. Takanami, A note on two-dimensional finite automata, *Information Processing Lett.* (to be published).

Chapter 5

CELLULAR ARRAY ACCEPTORS

5.1 INTRODUCTION

Cellular array acceptors are two-dimensional analogs of the cellular acceptors studied in Section 3.4. They are arrays of cells, all but finitely many of which are in a special state $\#$. The non- $\#$ s may constitute a rectangular subarray or merely a connected subarray; as in Chapter 4, we shall consider both possibilities. As in the one-dimensional case, we shall show that these acceptors define no new classes of languages, but that they often make acceptance possible in a much shorter time.

5.2 CELLULAR ARRAY AUTOMATA AND ACCEPTORS

Informally, a two-dimensional *cellular automaton* is an array of "cells" each of which, at any given time, is in some *state*. The cells operate in a sequence of discrete time steps, at each of which every cell reads the states of its four horizontal and vertical neighbors and changes to a (possibly) new state. Thus, formally, a two-dimensional cellular automaton K is defined by specifying a pair (Q, δ) , where Q is the set of states and δ is the *transition function* that maps quintuples of states into sets of states (or into single states, if K is deterministic)—i.e., $\delta: Q^5 \rightarrow 2^Q$ (or $\rightarrow Q$). For each cell c of K , δ maps the quintuple (state of left neighbor, state of right neighbor, state of upper neighbor, state of lower neighbor, state of c) into the set of possible new states of c , which, in the deterministic case, always consists of a single element. A *configura-*

tion of K is simply a mapping from $I \times I$ (the pairs of integers) into Q that specifies the state of each $c \in K$.

5.2.1 Neighborhood Size

In the above definition, we used only the four horizontal and vertical neighbors of each cell c . We could have used other sets of neighbors; for example, we could have also allowed the states of the diagonal neighbors of c to affect c 's next state. However [1], a transition that depends on a larger set of neighbors can be simulated by a sequence of transitions involving only the four neighbors, so that a cellular automaton K that uses a larger neighborhood set can be simulated by a K' that uses only four neighbors, at a rate of several time steps of K' for each time step of K . For example, suppose that we want to simulate a transition that depends on the states of the eight (horizontal, vertical, and diagonal) neighbors of each cell. One way to do this is for each cell, at even-numbered time steps, to go into a state of the form $(q_1, \dots, q_5)$, where the q s are the previous states of the cell and its four neighbors. If we denote the state of the cell at position (i, j) at a given odd-numbered time step by q_{ij} , then at the following time step the cell states in the neighborhood of (i, j) are

$$\begin{array}{ccc}
 & (q_{i-1,j+1}, q_{i+1,j+1}, & \\
 & q_{i,j+2}, q_{ij}, q_{i,j+1}) & \\
 (q_{i-2,j}, q_{ij}, q_{i-1,j+1}, & (q_{i-1,j}, q_{i+1,j}, q_{i,j+1}, & (q_{ij}, q_{i+2,j}, q_{i+1,j+1}, \\
 q_{i-1,j-1}, q_{i-1,j}) & q_{i,j-1}, q_{ij}) & q_{i+1,j-1}, q_{i+1,j}) \\
 & (q_{i-1,j-1}, q_{i+1,j-1}, q_{ij}, & \\
 & q_{i,j-2}, q_{i,j-1}) &
 \end{array}$$

The cell at (i, j) now has available to it the states (at the previous step) of itself and its horizontal and vertical neighbors, stored in its own current state; it also has available the previous states of its diagonal neighbors (as well as the states of other cells at city block distance 2 from it), stored in the current states of its horizontal and vertical neighbors [e.g., the state $q_{i-1,j+1}$ of (i, j) 's upper left neighbor is available as the first term of its upper neighbor's state, and also as the third term of its left neighbor's state]. Thus at the next odd-numbered step, cell (i, j) can go into a new state that depends on the previous states of all eight of its neighbors. This

process can then be repeated at succeeding pairs of time steps. Note that it not only requires two time steps of the four-neighbor automaton to simulate each time step of an eight-neighbor automaton, but it also requires a much larger set of states (quintuples of the original states are needed at even-numbered time steps).

It can also be shown [1] that a cellular automaton whose transition function involves only *two* neighbors in noncollinear directions, say north and east, can simulate an automaton that has a larger neighborhood—e.g., the four horizontal and vertical neighbors. To see this, let us have each cell, at time steps that are not multiples of 4, go into a state that contains information about the previous states of its neighbors. Thus the cell at (i, j) , at time steps 1, 2, and 3 (modulo 4), contains information about the states of those cells shown in Table 5.1.

TABLE 5.1

Step	Cells
1	$(i, j), (i + 1, j), (i, j + 1)$
2	These together with $(i + 2, j), (i + 1, j + 1), (i, j + 2)$
3	These together with $(i + 3, j), (i + 2, j + 1), (i + 1, j + 2), (i, j + 3)$

Hence at step 3, cell (i, j) knows the states of cell $(i + 1, j + 1)$ and its four neighbors $(i + 2, j + 1), (i, j + 1), (i + 1, j + 2),$ and $(i + 1, j)$, so that at step 4, cell (i, j) can go into a new state that depends on the old states of cell $(i + 1, j + 1)$ and its four neighbors. The entire process is then repeated. Note that it requires four time steps of the two-neighbor automaton per time step of the four-neighbor automaton, as well as a much larger state set. Moreover, the two-neighbor simulation does not remain stationary; it shifts downward and to the left by one unit in each four time steps. This type of simulation would not be appropriate for bounded cellular automata (see Section 5.2.2). We shall consider only four-neighbor automata from now on.

5.2.2 Boundedness and Acceptance

We shall assume henceforth that the state set Q is finite and that there is a special state $\$ \in Q$ such that, in the initial configuration

of K , all but a finite number of the cells have state $\#$. In Section 3.4 we further required that the non- $\#$ cells form a string. There are two possible generalizations of this requirement to two dimensions: We can require that the array Σ of non- $\#$ cells be rectangular in shape, or merely that Σ be connected. We shall consider both of these possibilities in this chapter.

We say that the transition function δ of K is $\#$ -preserving if

- (a) $\# \in \delta(p, q, r, s, t)$ implies $t = \#$;
- (b) $\delta(p, q, r, s, \#) = \{\#\}$ for all p, q, r, s in Q

If δ has these properties, we call K a *bounded* cellular automaton. [As in Chapter 3, inability to create $\#$ s is not a restriction on K ; we shall therefore assume from now on that δ never creates $\#$ s.] Note that in the bounded case we may as well assume that the array of cells is finite, consisting of Σ surrounded by a border of $\#$ s, since the $\#$ s will never change.

The notion of acceptance of an input array by a cellular automaton K is defined exactly as in the one-dimensional case. Formally, a *cellular acceptor* (CA) C is a triple (K, Q_I, Q_A) , where K is a cellular automaton with state set Q , $Q_I \subseteq Q$ is a set of *initial states*, and $Q_A \subseteq Q$ is a set of *accepting states*, with $\# \in Q_I$. If K is bounded, we call C a *bounded cellular acceptor* (BCA). An *input array* is a configuration whose states are all in Q_I . We say that C accepts this array (or, for brevity, that C accepts the non- $\#$ part Σ of this array) if repeated application of K 's transition function, starting from this configuration, can lead to an "accepting configuration."

As in Section 3.4.1, we can define an accepting configuration in three ways:

- (a) Every $c \in C$ has its state in Q_A ;
- (b) some $c \in C$ has its state in Q_A ; or
- (c) a particular $c_0 \in C$ —e.g., the leftmost of the uppermost cells of Σ —has its state in Q_A .

We can show, using analogs of Propositions 3.4.1–3.4.4, that in most cases the classes of languages accepted in these three ways are the same. The proofs are much simpler for rectangular Σ s than for arbitrary connected Σ s; we shall treat the rectangular case in Section 5.3 and the connected case in Section 5.4.

5.3 RECTANGULAR CELLULAR ACCEPTORS

5.3.1 Rectangular Cellular Languages

The analogs of Propositions 3.4.2–3.4.4 for rectangular cellular arrays follow readily from the observation that the upper left corner cell c_0 (leftmost of the uppermost cells) can uniquely identify itself at the start since it is the only cell that has $\#$ s both above it and to its left. In Proposition 3.4.2, when any cell enters an accepting state in the simulation, it initiates an accepting signal that spreads in all directions (through non- $\#$ s); since Σ always remains connected ($\#$ s are never created), this signal eventually reaches c_0 , which then accepts. Proposition 3.4.3 is immediate: In the simulation, only c_0 can enter an accepting state. In Proposition 3.4.4, when c_0 accepts in the simulation, it initiates an accepting signal that spreads in all directions (through non- $\#$ s), so that this signal eventually reaches all of Σ and causes every cell to accept; by using a half-speed simulation, we can assure this even if the simulation is growing.

For Proposition 3.4.1, c_0 identifies itself and C' simulates C . If c_0 accepts in the simulation, it sends out a signal that spreads in all directions through accepting states. We shall show next how this signal can generate a reply that reaches c_0 iff every non- $\#$ cell is in an accepting state. (Here we use the assumptions that accepting states are never rewritten and cannot cause $\#$ s to be rewritten; we shall see later how these assumptions can be avoided at the cost of a much slower simulation.) Thus c_0 can accept iff every cell of the simulation has accepted.

If C is a BCA, the propagation of the reply signal can be done very simply, since the array Σ of non- $\#$ s remains rectangular. Let c_1 and c_2 be the cells in the upper right and lower right corners of Σ , respectively; they are uniquely identified by having $\#$ s as their right and upper (or lower) neighbors. When the acceptance signal α reaches c_2 , it initiates a signal x that can only spread leftward through accepting states; thus x reaches all cells of the bottom row of Σ iff they are all in accepting states. In turn, x initiates a signal y that can only spread upward through accepting states; thus y reaches the top of a column of Σ iff that column consists entirely of cells in accepting states. When y reaches c_1 , it initiates a signal z that can only spread leftward through cells that have received y . Thus z reaches c_0 iff y reached the top of every column of Σ , so that

every cell of Σ is in an accepting state, and c_0 can finally accept. Note that the time required for α to reach c_2 is $m + n - 2$, where m is the height and n the width of Σ ; the time required for z to reach c_0 , if every cell has accepted, is also only $m + n - 2$. [Indeed, x reaches the cells of the bottom row at times $0, 1, \dots, n - 1$; y reaches the cells of the top row at times $m - 1, 1 + (m - 1), \dots, (n - 1) + (m - 1)$; z is initiated at time $m - 1$ and reaches each cell of the top row just as y reaches it, so that it reaches c_0 at time $(n - 1) + (m - 1)$.]

If C is not a BCA, Σ may not be rectangular, but we can handle the propagation of the reply signal by generating a rectangle T containing Σ and transmitting the signals throughout that rectangle. (In Section 5.4 we shall describe a method of signaling that does not require going outside Σ even if Σ is nonrectangular.) In the following paragraphs we describe one method of doing this, derived from [2], which generates a square (oriented diagonally) centered at c_0 that contains Σ .

Let us assume that the simulation of C is done at half speed. Concurrently with this simulation (using first terms of state pairs), we propagate a signal x outward from c_0 ; i.e., any cell, say in state q (non- $\#$ or $\#$), not having x as first term, that sees first term x as one of its neighbors becomes a pair (x, q) . Moreover, we record the direction(s) from which x reaches each cell; e.g., we can do this by using eight versions of x , denoted by $x_N, x_S, x_E, x_W, x_{NE}, x_{NW}, x_{SE},$ and x_{SW} , to indicate whether x reached the given cell from its north, south, east, or west neighbor, or from both its north and east, $\dots$, or south and west neighbors (it is not hard to see that these are the only possibilities). This process generates an expanding wave of x s shaped like a square oriented diagonally, with center at c_0 . We shall assume that the propagation of the x s, like the simulation of C , takes place at half speed; thus the x s add a new layer to the expanding square at alternate time steps. (The reason for this assumption will become clear later.)

As the x s reach each new layer, they send back an "echo" signal y to c_0 . (This is done once only at each layer; to ensure this, we mark the newly created x s with primes that are erased at the next time step, and only the primed x s initiate the y signals.) The directional information contained in the x s is used to direct y so that it travels back to c_0 along the shortest possible paths. For example, if cell c , say in state (x, q) , has an $x'_N, x'_{NE},$ or x'_{NW} , or an $(x_N, y), (x_{NE}, y)$ or (x_{NW}, y) , in the first term of its south neighbor, it goes

into state $((x, y), q)$; and similarly for the other three directions. It is easily seen that the y signals from a given layer of x s generate a contracting wave of y s that occupies the successive layers of the square centered at c_0 , from outermost to innermost. If the y s start at a layer k steps away from c_0 , they reach c_0 (from all four sides) k time steps later. We shall assume that the y s propagate at full speed and that y s are erased as soon as they are created. It follows that the successive x layers create waves of y s that are spaced two units apart and do not interfere with each other.

Each y wave can carry information to c_0 about the x layer that initiated it and about the other layers through which it passed. Specifically, if a y is initiated by a cell whose state is of the form (x, q) , where $q \neq \#$, we shall denote it by y' . (A given y may be initiated by two x cells; we use y' if either of these cells has a non- $\#$ second term.) Moreover, if a y reaches a cell whose second term is neither $\#$ nor an accepting state of C , we change it to y' . When y signals reach a cell c from two or more of its neighbors at once, if any of them is a y' , we use y' at c also. These conditions imply that if any cell in the initiating x layer is non- $\#$, or if any cell within the square of x s is neither $\#$ nor accepting, then the y that reaches c_0 will be a y' . Conversely, if an unprimed y reaches c_0 (i.e., it receives y from all four neighbors, and its own state is $\#$ or accepting), we know that

- (a) every cell in the initiating layer is $\#$ and
- (b) every cell in the square is $\#$ or accepting.

Since the non- $\#$ s always remain connected, (a) and (b) imply that every non- $\#$ cell is in an accepting state. Thus when this happens, c_0 can accept. Moreover, if it ever does happen, this process will detect it, since accepting states do not change and do not cause $\#$ s to be rewritten; thus if every non- $\#$ is accepting, Σ stops growing, and the x propagation catches up with it and reaches a layer of all $\#$ s, so that a y wave is generated that satisfies (a) and (b).

In the construction just given, the x wave never stops propagating. In order to stop it, c_0 (upon acceptance) can send out a full-speed signal z that erases the x s and y s as it encounters them, and erases itself when it reaches the $\#$ s beyond the outermost x s. If this is done, the expanding square never grows beyond a bounded size. Note, however, that this size may be much greater than the radius of Σ ; e.g., Σ may stop growing long before its cells all go into accepting states, so that the square gets much bigger

than Σ before the y wave detects acceptance. Thus after every cell of Σ has accepted, the propagation of the acceptance signal y may take much longer than the diameter of Σ , which is undesirable.

We can reduce the time to be on the order of the diameter of Σ by modifying the construction as follows: The square of x s expands (at half speed) until its outermost layer consists of $\#$ s only, so that it entirely contains Σ . The y wave relays this information to c_0 , which sends out a z signal (at full speed) to stop the expansion of the x s. (Here the y wave must carry two independent pieces of information: one about $\#$ s in the outer layer, and the other about $\#$ s and accepting states in the interior. Also, the z signal must not erase x s and y s, but only "freeze" the outermost x s so they cannot expand. Note that by the time the y wave reaches c_0 and the z signal reaches the outer layer, the square has tripled in size; but its radius is still of the same order as that of Σ .) The outermost x s continue to send back y waves even though they have stopped expanding (these waves are now consecutive, but still do not interfere with one another), so that c_0 can continue to check whether the outer layer is still all $\#$ and whether the interior is all $\#$ or accepting. If these conditions are met, c_0 accepts. If non- $\#$ s reach the outer layer (due to continued expansion of Σ) before the cells of Σ have all accepted, c_0 discovers this from the y signal and sends a full-speed w signal to the outer layer. Arrival of this signal causes the outer layer to start expanding again (at half speed), and also to send a full-speed w' signal back to c_0 ; when c_0 receives w' , it sends a full-speed z signal to stop the expansion again. During this time, the square again triples in size. If the non- $\#$ s again overtake (or have already overtaken) its outer layer, the process can be repeated. If the cells of Σ ever all accept, Σ stops growing, so the square eventually expands beyond Σ , at which time the y wave detects the conditions for acceptance. In this construction, the radius of the square is never more than triple the radius of Σ ; hence the time for propagation of the acceptance signal Σ , after all cells have accepted, is on the order of the diameter of Σ .

Up to now we have assumed that accepting states never change and never cause $\#$ s to become non- $\#$ s. Even without these assumptions, we can still verify that every cell (of a simulation of C) has accepted, but the process is now much slower. The approach is analogous to that in the appendix to Section 4.3.1; it consists of the following steps:

- (a) We simulate C one step at a time.
- (b) After each step of the simulation, we construct a square centered at c_0 that contains Σ , and check whether every non- $\#$ cell in this square is simulating an accepting state of Σ . [Here C' is not a BCA!]
- (c) If so, c_0 accepts; if not, c_0 initiates a two-dimensional synchronization process that causes all the cells of the square to go into a special state, at which point the non- $\#$ cells simultaneously simulate another step of Σ .

This process is repeated as often as necessary.

The two-dimensional synchronization process is a straightforward extension of the one-dimensional process described earlier; see, e.g., [3]. To illustrate how it works, let us suppose for simplicity that we have an $m \times n$ upright rectangle Σ of cells with distinguished cell c_0 in the upper left corner. By the one-dimensional construction, c_0 can synchronize the left column of Σ , and this takes $O(m)$ (i.e., on the order of m) steps. Each cell in the left column can then synchronize its row, and this takes $O(n)$ steps. Thus in $O(m + n)$ steps, the entire rectangle can be synchronized.

The analogs of Propositions 3.4.1–3.4.4 show that the definitions of acceptance by some cells, all cells, and the upper left cell (or any cell that can uniquely identify itself) are all equivalent. We shall use the upper left definition from now on, and shall always assume that cell c_0 is uniquely marked.

The set of input arrays Σ accepted by C will be called the *language* of C , and will be denoted by $L(C)$. Note that $L(C)$ can contain arrays of any size, depending on how many cells of C are initially in non- $\#$ states. [C is regarded as the same CA no matter how big Σ is, as long as C has the same transition function.] The class of all languages accepted by (D)(B)CAs will be denoted by $\mathcal{L}_{(D)(B)C}$.

We conclude this section by showing how a DBCA can verify that its input Σ is indeed rectangular. Note first that a connected Σ is rectangular iff it has no concave corners, i.e., iff there are no 2×2 patterns in Σ exactly one of whose cells is $\#$ (this will be proved immediately below). Next, note that the presence of concave corners can be detected, in only two time steps, as follows:

- (a) Any cell that has a $\#$ as a north, south, east, or west neighbor is marked n , s , e , or w , respectively.
- (b) Any cell that has an s as east neighbor and an e as south

neighbor, or analogously for the pairs of directions s and w , n and e , and n and w , is marked x .

Evidently, a cell is marked x iff it belongs to a concave corner, e.g., xs
 $e\#$. Finally, cell c_0 can determine, in a number of time steps on the order of the diameter of Σ , whether or not Σ contains any xs ; a method of doing so will be described in Section 5.4. If Σ contains no xs , c_0 can accept.

PROPOSITION 5.3.1 A connected set Σ is an upright rectangle iff it has no concave corners.

Proof: Clearly, a rectangle has no concave corners. Conversely, consider a maximal set of pairwise adjacent runs in which consecutive rows of the array meet Σ . Since there are no concave corners, these runs must line up at both ends and none of them can merge or split; hence they constitute an upright rectangle, and no other run of points of Σ is adjacent to them. Since Σ is connected, they must thus be all of Σ . $\square$

Appendix to Section 5.3.1

In this appendix we show how to construct the upright rectangle $T(\Sigma)$ that just contains a given connected Σ [so that there are points of Σ in the top and bottom rows, and left and right columns, of $T(\Sigma)$]. We shall call $T(\Sigma)$ the *framing rectangle* of Σ and denote its extremal rows and columns by t , b , ℓ , and r , respectively.

Our construction is based on a propagation process that repeatedly fills concave corners; in other words, if a $\#$ cell has two non- $\#$ neighbors on adjacent sides of it (N and E, N and W, S and E, or S and W), we change it to a non- $\#$, say to $\natural$. We shall show that if this process is iterated until no further change takes place, the $\natural$ s together with Σ itself constitute exactly $T(\Sigma)$. Moreover, the process stops in a number of time steps at most equal to $m + n - 2$, where m and n are the height and width of $T(\Sigma)$.

Note first that when we fill concave corners, the framing rectangle is not enlarged. Indeed, a $\#$ above t , below b , to the right of r , or to the left of ℓ clearly cannot belong to a concave corner; hence no step of the propagation process can create a $\natural$ outside $T(\Sigma)$. Since $T(\Sigma)$ is finite, the process must stop eventually. On the other

hand, when the process stops, the resulting set T has no concave corners, and so by Proposition 5.3.1 must be an upright rectangle; clearly, the only such rectangle that meets t , b , ℓ , and r is $T(\Sigma)$ itself.

It remains only to show that this process of constructing $T(\Sigma)$ takes at most $m + n - 2$ time steps. We do this in several stages:

- (a) Let us first observe that $T(\Sigma)$ cannot contain a 4-path π of $\#$ s that has one end on t and the other on b . This is because Σ touches both ℓ and r , so that there is a path π' of non- $\#$ s in $\Sigma \subseteq T(\Sigma)$ from ℓ to r , and π' would have to cross π —a contradiction. Similarly, there can be no 4-path of $\#$ s in $T(\Sigma)$ that has one end on ℓ and the other on r .
- (b) Next, we show that for any point $P \in T(\Sigma)$, there is a quadrant in which Σ surrounds P ; in other words, there exists a pair of adjacent directions (N and E, N and W, S and E, or S and W) such that any 4-path starting at P that moves only in these directions must hit Σ . Indeed, suppose not; then in all four quadrants there are 4-paths from P that reach the border of $T(\Sigma)$ without meeting Σ . The path π'' in the NE quadrant, for example, must hit the border either on t or on r ; suppose it hits t . If the path in either the SE or SW quadrant hits the border on b , then by concatenating this path with π'' we would get a 4-path of $\#$ s from t to b (through P), contradicting (a). Hence the path in the SE quadrant must hit r , and that in the SW quadrant must hit ℓ , so that by concatenating these paths we get a 4-path of $\#$ s from ℓ to r (through P), contradicting (a). The argument is analogous if π'' hits r ; we would then consider the paths in the NW and SW quadrants. Thus we obtain a contradiction in every case, so that (b) must be true.
- (c) Let $P \in T(\Sigma)$, $\notin \Sigma$, and suppose that Σ surrounds P in the NE quadrant. Thus if we move, e.g., upward from P , we must hit Σ ; if we then move rightward (if possible), we must hit Σ again; if we then move upward, we hit Σ again; and so on. The path of $\#$ s (or $\natural$ s) along which we move is a 4-geodesic, since it makes left and right turns alternately (see Section 2.9); thus the k th point on the path is at city block distance k from P . Since Σ is finite, we cannot get farther than a finite city block distance from P ; hence this process of path construction must terminate. But it can

only terminate at a point that has Σ both above it and on its right, so that neither upward nor rightward motion is possible; thus the terminal point of the path is at a concave corner of Σ . Since the path always remains inside $T(\Sigma)$, its length is less than $m + n - 2$.

- (d) For any $P \in T(\Sigma)$, $P \notin \Sigma$, let h be the length of the longest geodesic from P through $\#$ s to a concave corner of Σ ; thus $h < m + n - 2$. When we fill the concave corners of Σ , this path is shortened. On the other hand, (a–c) evidently still hold, so that we can repeat this argument. At each iteration of concave corner filling, the longest geodesic length from P through $\#$ s to a concave corner of Σ decreases. Since initially this length was $< m + n - 2$, the number of iterations required to reduce it to zero is $< m + n - 2$. At this stage, P itself is at a concave corner of Σ , and at the next iteration Σ swallows up P . Since P was arbitrary, this means that Σ grows to fill all of $T(\Sigma)$ in at most $m + n - 2$ iterations.

5.3.2 Equivalence to Sequential Acceptors

In this section we prove that TAs and CAs can simulate each other, and that the same is true for TBAs and BCAs. This shows that, in introducing two-dimensional CAs, we have not created any new classes of languages. The proofs are analogous to those in Section 3.4.2. To show that T(B)As can simulate (B)CAs, we need to prove that a DT(B)A can systematically scan its array of non- $\#$ s, and at each point, compute what the transition of the (B)CA's cell at that point would have been. This has already been shown for DTAs in Proposition 4.4.1, and for DTBAs in Theorem 4.4.2. Note that even when the input array Σ is rectangular, we still need the traversal result of Proposition 4.4.1 for the TA/CA proof, since Σ may grow. On the other hand, for rectangular Σ s, we do not need Theorem 4.4.2 in the TBA/BCA proof, since Σ remains rectangular, and its traversal is straightforward, e.g., row by row as in the proof of Proposition 4.2.1.

THEOREM 5.3.2 (D)T(B)As can simulate (D)(B)CAs.

Proof: Given a CA C with transition function δ , we define a TA A that has a vocabulary consisting of states and pairs of states of C . At any given stage of the simulation of C by A , the non- $\#$ array Σ

has a pair of states at each point, one of which represents the current state of the corresponding cell of C , while the other represents the preceding state of that cell. (Initially, Σ consists of just the initial states of C 's cells.) To simulate one transition of C , A systematically scans Σ . At each point P of Σ , A also examines P 's neighbors, records the current states of the corresponding cells of C , returns to P , and computes a new state of the cell at P using δ . A then erases the previous-state information at C (if any) and replaces it by new-state information, but leaves the current-state information intact (since it may be needed to compute new states for neighbors of P that have not yet been processed). A also marks P to indicate that it has been processed and resumes scanning Σ . When the scan is complete, every point of Σ is marked and contains a pair of states, one of which is the former current state (now previous), while the other is the new state (now current) at that point. A now scans P again, erases all the marks, and returns to its starting point; it is now ready to simulate the next transition of C . A accepts iff its simulation of C accepts.

If C is not a BCA, A must also visit all the $\#$ s adjacent to Σ during its scan, check their neighbors, and rewrite them as non- $\#$ s if C would have done so; thus if C is not a BCA, A cannot be a TBA. On the other hand, if C is a BCA, the simulator A can be a TBA, and if C is deterministic, so is A . Note that a very large number of transitions of A is needed to simulate a single transition of C ; the simulation is very inefficient. (Compare the proof of Theorem 3.4.5, which provided the fastest possible simulation in the one-dimensional case.) $\square$

THEOREM 5.3.3 (D)(B)CAs can simulate (D)T(B)As.

Proof: Given a TA A with transition function δ , we define a CA C that simulates A just as in the proof of Theorem 3.4.6. Initially, the upper left cell c_0 goes into a special state representing the initial state q_0 of A , the symbol x at that position, and a direction in which A can move when it is in state q_0 and reads symbol x . [This is easy for rectangular input arrays, in which c_0 is unique, but it is not so easy for arbitrary connected arrays; see Section 5.4.] At subsequent transitions, the special state moves from cell to cell just as it did in the one-dimensional case, except that four move directions are now possible. At any step, exactly one cell is in the special state, corresponding to a position and state that A could have been in at that step. This process continues until the special cell's

state represents an accepting state of A . At this point, the special cell propagates an acceptance signal to c_0 , which can then accept, so that C accepts iff A does. Note that if A is deterministic, so is C , and if A is tape-bounded, so is C (if we modify the simulation to handle the steps in which A bounces off $\#$ s without requiring $\#$ cells of C to change state). As in Theorem 3.4.6, this simulation is the fastest possible, since C requires only one time step to simulate each time step of A after the first (except for the propagation of the acceptance signal at the end). $\square$

Theorems 5.3.2–5.3.3 immediately imply

THEOREM 5.3.4 $\mathcal{L}_{(D)C} = \mathcal{L}_{(D)T}$; $\mathcal{L}_{(D)BC} = \mathcal{L}_{(D)TB}$.

Note that we have proved these theorems not only for rectangular input arrays, but for arbitrary connected ones, except for the problem of c_0 initially identifying itself (in the proof of Theorem 5.3.3) in the arbitrary connected array case; this will be taken care of in Section 5.4.

5.4 CONNECTED CELLULAR ACCEPTORS

To extend the results of Section 5.3 to arbitrary connected arrays, we need to show that, in a DBCA,

- (a) the leftmost of the uppermost cells can uniquely identify itself, and
- (b) this cell can interrogate all the cells (e.g., as to whether or not they are in accepting states) and know when they have finished replying.

We shall show that these tasks can, in fact, be carried out in time proportional to the perimeter or diameter of the given array. The proof of (a) is based on [4].

Given that (a and b) can be done, it is straightforward to extend the results of Section 5.3 to connected arrays. The analogs of Propositions 3.4.2–3.4.4 require that (a) holds, while for Proposition 3.4.1 we require both (a) and (b) [in the non-BCA case, (b) is not needed, since we can use the expanding-square construction of Section 5.3.1]. The slower procedure based on simulation one step

at a time, in conjunction with scanning and synchronization, also applies in the connected case, provided that (a) holds. Note that we also need (b) to prove that a DBCA can tell whether its input array is rectangular, as pointed out in Section 5.3.1.

THEOREM 5.4.1 There exists a DBCA in which the leftmost of the uppermost cells eventually goes into a special state, while no other cell ever goes into that state.

Proof: Initially, each non- $\#$ cell c determines which of its neighbors (if any) is $\#$ and then copies the corresponding information from its non- $\#$ neighbors. This enables c to determine, if it is on a border of Σ , how the border-following algorithm BF (Section 2.6) would behave at it. In particular, c can tell which of its neighbors would precede or succeed it at each visit of BF to it. If c would be visited more than once by BF, it keeps the information about each visit separate by associating it (e.g., via a suitable subscript or position in a 4-tuple) with the particular $\#$ neighbor(s) that BF would examine at that visit. For example, if the neighborhood of c is

$$\begin{array}{ccc} a & \# & \# \\ b & c & d \\ \# & \# & \# \end{array}$$

and borders are always followed by keeping the $\#$ s on the left, then the predecessor of c with respect to its upper $\#$ neighbor is a , while its predecessor with respect to its lower $\#$ neighbor is d .

Based on this local analysis, c can tell in particular, for each visit of BF to it, whether its predecessor lies higher or lower than it; in the former case, c associates an x with that visit, and in the latter case, a y . In addition, if c is an upper left corner (i.e., its left and upper neighbors are both $\#$), it marks itself with z . All of this initial processing requires only a bounded number of time steps.

We now independently process the x s, y s, and z s on each border of Σ . As already pointed out, even if two borders pass through a given cell, their processes will not interfere, since the marks know with which $\#$ neighbor(s) of c they are associated. We can thus describe the processing for a single border β .

In this processing, the y s shift forward along β (i.e., in the direction that BF would take), and the x s shift backward (in the reverse direction). When an x and a y meet, they cancel out; and when an

x or y hits a z , that z is erased. We claim that when this process has gone to completion, the x s and y s will all have cancelled, and all the z s will be erased *except for those on the highest row of Σ (if any)*; this will be proved in the next two paragraphs. If β is the outer border of Σ , there will in fact exist z s on the highest row of β . On the other hand, if β is a hole border, it is clear that there cannot exist z s on its highest row, so that for such a border, all the z s will be erased. Thus when we process all the borders of Σ in this way, the only remaining z s will be those on the highest row of Σ .

To prove that all z s not on the highest row of β will be erased, consider a z_0 that is not a highest point of β ; then as we move forward along β from z_0 , we eventually reach a higher point. Let P be the first such point; then on the segment of β between z_0 and P , there are more x s than y s. Now consider what happens as the x s shift backward and the y s shift forward. If a y reaches z_0 from its other side, z_0 is erased; but if no such y reaches z_0 , the x at P eventually reaches z_0 (since only the y s between z_0 and P are available to cancel the x s) and erases it. (Note that by definition, P must also be higher than any other z s between z_0 and P , so that they too get erased.)

Conversely, let z_0 be on the highest row of β ; then as we move forward along β starting at z_0 , the net number of x s can never exceed the net number of y s, so that no backward-shifting x can ever reach z_0 . Similarly, as we move backward along β starting at z_0 , the net number of y s (which are upward moves when we go backward!) can never exceed the net number of x s, so that forward-shifting y s can never reach z_0 . We can think of the x s and y s between any two consecutive highest z s (or all around β , if there is only one highest z) as constituting a well-formed parenthesis string, which cancels itself out completely in the course of the shifting; see (d) at the end of Section 3.4.3.

As described up to now, the cancellation process leaves the highest z s intact, but these cells can never know that they are highest z s, since no matter how long they wait, they can never know that an x or y from a very distant higher point is not still on its way toward them. In order to allow these z s to discover that they are highest, we must make a small modification in the cancellation process. When an x and a y meet, instead of cancelling, they turn into "ghosts," denoted by x' and y' , which continue to shift in the same directions. When an x' and a y' meet, they simply pass

each other; but when an x' meets a y , the x' is erased (without affecting the y), and when a y' meets an x , the y' is erased. It is easily seen that, on any segment of β bounded by two highest z s, at least one x' and y' (namely, the last ones to be created) will not be erased, and these will reach the z s. Thus when a z has been hit by both an x' and a y' , it knows that no x or y can ever hit it, so that it must be a highest z . To illustrate this process, we show in Table 5.2 the successive shifting and cancellation steps for a simple example. (At subsequent steps, the x' moves left to z_1 , and the y' moves right to z_2 .)

Concurrently with and independently of all the above, any cell c that is a lower right corner (i.e., whose lower and right neighbors are $\#$ s) marks itself with w . By a process analogous to that described for the z s, the w s in the lowest row of β identify themselves. (Note that, like the z s, they can exist only on the outer border β_0 of Σ , not on hole borders.) When a w has identified itself, it sends out a signal t that travels clockwise around β_0 (i.e., with the $\#$ s on the left, starting with the $\#$ below the w). If t hits another w , it is erased. If it hits a z that has not yet been identified as highest, it waits until that z is either erased or found to be highest. In the former case, the signal continues to travel. In the latter case, that z must be the leftmost of the uppermost points of Σ , since it is the first of the highest z s that was reached by t starting from a lowest w . In other words, that z is the desired c_0 .

The entire process of uniquely identifying c_0 takes time proportional to the perimeter of Σ . In fact, the initial steps take a bounded amount of time; the identifications of the highest z s and lowest w s take times that are fractions of the perimeter; and the identification of c_0 using t takes time less than the outer perimeter. $\square$

TABLE 5.2

Step												
1	z_1	y	y	x	y	x	y	y	x	x	x	z_2
2			y	y'	x'	y'		y	x	x		
3				y	y'		y'	x'	x			
4					y	y'	x'	x				
5						y	x					
6						x'	y'					

In proving the analogs of Propositions 3.4.1–3.4.4 for non-BCAs, we can first identify c_0 , e.g., by creating a square that contains Σ (see Section 5.3.1), and scanning the square row by row until the leftmost uppermost non- $\#$ is found. We can then synchronize the square and initiate the simulation; the proofs now proceed as in Section 5.3.1. The time required, as pointed out in that section, is proportional to the (greatest) diameter of Σ .

For BCAs, the simulation and the process of identifying c_0 can proceed concurrently. (We can do this for non-BCAs also, if we do the identification of c_0 on a copy of the original Σ while allowing another copy of Σ to grow, if necessary, under the simulation.) In Proposition 3.4.1, when c_0 has identified itself and has also (either before or after identification) accepted in the simulation, it sends out an interrogatory signal that generates a reply, as in the theorem to be proved next (or, for non-BCAs, it can use the same method as in Section 5.3.1). In Proposition 3.4.2, any accepting cell sends out an acceptance signal; when it reaches c_0 (or when c_0 identifies itself and finds that this signal has already reached it), c_0 accepts. In Proposition 3.4.3, only c_0 can accept, and it can do so only after it has identified itself. In Proposition 3.4.4, when c_0 accepts (or identifies itself and finds it has accepted), it sends out an acceptance signal that causes every other cell to accept. The time required for transmission of these signals is at most the diameter of Σ ; as shown in Section 2.8, this is less than the perimeter of Σ .

We conclude this section by proving

THEOREM 5.4.2 There exists a DBCA with a distinguished cell c_0 such that c_0 goes into a special state after it has sent out a signal and received a reply from the cell farthest from it.

Proof: If c_0 is the upper left cell of a rectangular array, this is easy: c_0 sends out its signal, which propagates from cell to cell; i.e., each cell copies the signal from its neighbors. The lower right cell is uniquely identified by having $\#$ s below it and on its right, and this is the cell farthest from c_0 . When this cell receives the signal, it emits a reply, which propagates from cell to cell until it reaches c_0 . When c_0 receives the reply, it knows that the farthest cell has received its signal and replied, and can go into the special state.

For connected arrays, we could use the fact that DBCAs can simulate DTBAs, as was proved in Section 5.3.2. Cell c_0 could thus initiate a DTBA simulation that systematically scanned the array (see Theorems 4.4.2 and 4.4.5), and when finished, erased all its marks and returned to c_0 ; at this point, c_0 could accept. However, this approach is very slow (compare Section 5.5). Instead, we give a construction that makes extensive use of the DBCA's parallelism and requires only time proportional to the intrinsic diameter of the array.

First, c_0 sends out its signal x , which propagates from cell to cell. If a cell c receives x from exactly one of its neighbors, c stores the direction of that neighbor (and also copies x). If c receives x from two or more neighbors, it stores only one of their directions, according to an arbitrary preference ordering (e.g., L, R, U, D). Note that c receives x after a number of time steps equal to c 's city block distance from c_0 .

When a cell c receives x from all of its (non- $\emptyset$) neighbors, so that it has no neighbors that have not yet received x , the distance of c from c_0 must be a local maximum. Such c s initiate a reply signal y , which will propagate back to c_0 along the paths indicated by the directions stored in the cells; in other words, if c has received y , its neighbor c' accepts y from it only if c' is the neighbor of c in the direction that c had stored. Moreover, if c' has two (or more) neighbors, say c_1, c_2 , for which it is the stored-direction neighbor, it accepts y only after y has arrived at both c_1 and c_2 .

It is easily seen that the stored directions define a directed spanning tree of the array, rooted at c_0 . The reply signal y is initiated by the leaves of the tree and passes through a node of the tree only when it has been received by all sons of that node. Thus the time required for y to reach c_0 is equal to the longest distance from c_0 to any leaf, i.e., the longest distance from c_0 to any point of Σ , which is at most the diameter of Σ . $\square$

To apply this result to proving the analog of Proposition 3.4.1, we change the reply signal from y to y' if it ever encounters a nonaccepting cell. Thus if c_0 receives y , every cell of the simulation has accepted and c_0 can accept; while if c_0 receives y' , it can try again later (the simulation proceeds concurrently with the

signal and reply propagation). If every cell in the simulation eventually does accept, c_0 will eventually receive y (since accepting states never change) and will accept.[§]

The analogs of Propositions 3.4.1–3.4.4 show that the three definitions of acceptance are still equivalent even for connected arrays. We shall use the upper left definition from now on, and shall always assume that c_0 is uniquely marked. The *language* $L(C)$ of arrays accepted by C , and the class $\mathcal{L}_{(D)(B)C}$ of all languages accepted by (D)(B)CAs, are defined just as in the rectangular case (Section 5.3.1).

5.5 SPEED COMPARISONS

As we have just seen, CAs can simulate TAs in essentially real time, so that any language can be accepted by a CA at least as fast as it can be accepted by a TA (ignoring the acceptance propagation steps). On the other hand, some languages can be accepted by CAs much faster than by TAs. In this section, we discuss comparative acceptance speeds for two-dimensional TAs and CAs.

Let $|\Sigma|$ denote the number of elements in the array Σ , and let $\langle \Sigma \rangle$ denote the number of border points of Σ . If Σ is rectangular, say $m \times n$, we have $|\Sigma| = mn$ and $\langle \Sigma \rangle = 2(m + n - 2)$ (provided $m, n \geq 2$); it follows that $|\Sigma| < \langle \Sigma \rangle^2$. In the nonrectangular case, let $T(\Sigma)$ be the framing rectangle of Σ (see the appendix to Section 5.3.1), and suppose that $T(\Sigma)$ is m by n . Then Σ has at least two border points (run ends) in each row of $T(\Sigma)$, and at least two in each column of Σ , so that $\langle \Sigma \rangle \geq \max(2m, 2n) \geq m + n$. On the other hand, $|\Sigma| \leq mn$, so that here again we have $|\Sigma| < \langle \Sigma \rangle^2$.

For any acceptor A (or C), suppose that for all Σ there exists a function f , taking natural numbers into natural numbers, such that if A accepts Σ at all, it does so in at most $f(\langle \Sigma \rangle)$ time steps. We then say that A accepts in *perimeter time of order f* . For example, if f is linear, we say that A accepts in perimeter time; if f is quadratic, we say that A accepts in perimeter-squared time; and so on. Similarly, if there exists a function f such that A accepts any Σ , if at all, in at most $f(|\Sigma|)$ time steps, we say that A accepts in *area time of order f* . Note that by the remarks in the preceding paragraph, if A accepts

[§]The same argument is used to prove that a DBCA can verify its rectangularity: The reply y becomes y' if it encounters a concave corner; if c_0 receives y , there are no concave corners. Note that a DBCA could also verify rectangularity by simulating the proof of Proposition 4.2.1, but this would be very slow.

in area time, it accepts in perimeter-squared time. As usual, we abbreviate "on the order of ()" by " $O()$ ".

It is easily seen that, except in certain special cases, no TA can accept in faster than area time, and no CA can accept in faster than $O(\text{perimeter})$ time. Indeed, if A accepts Σ in $h < |\Sigma|$ time steps, it cannot have seen all of Σ , and so accepts any array that matches Σ in at most h positions, and is otherwise arbitrary. Similarly, if C accepts Σ in $h < \langle \Sigma \rangle / 2$ time steps, the state of c_0 at step h cannot depend on the initial states of cells farther than h (in city block distance) away. For an $m \times n$ rectangular array, the farthest point from c_0 (the lower-right-hand corner) is at distance $m + n - 2 = \frac{1}{2}\langle \Sigma \rangle$; hence the state of c_0 at step h cannot depend on this point. Thus C accepts any array that matches Σ out to city block distance h from c_0 , and is otherwise arbitrary.[§]

We may thus conclude that for "interesting" array languages L , the time required to accept any $\Sigma \in L$ must be at least $|\Sigma|$ for TAs, and at least $\frac{1}{2}\langle \Sigma \rangle$ for CAs [or $O(\langle \Sigma \rangle)$, in the nonrectangular case]. Note that the "speed limit" for acceptance by two-dimensional CAs is generally much lower than that for TAs, unlike the case in one dimension for which both speed limits were the same (namely, $|\sigma|$).

To illustrate acceptance in minimal time by TAs and CAs, we may again consider (as in Section 3.4.3) the language $L_{\bar{x}}$ consisting of all rectangular arrays in which a particular vocabulary symbol x never appears. A DTBFSFA can accept $L_{\bar{x}}$ in area time by systematically scanning Σ row by row; if it finds an x , it stops and enters a nonaccepting state; if it completes the scan without finding an x , it accepts. A BCA that accepts $L_{\bar{x}}$ is defined as follows: The bottom cell of each column initiates a signal y that propagates upward through non- x s, but is stopped by x s. The rightmost cell of the top row initiates a signal z that propagates leftward through y s. If z reaches c_0 (at the left end of the top row), it accepts. Evidently this

[§] For arbitrary connected arrays, the farthest point may be much less than $\langle \Sigma \rangle$ away from c_0 , so that nontrivial recognition of certain classes of such arrays in time of order less than $\langle \Sigma \rangle$ may be possible; but as the rectangular case shows, there do exist classes of arrays for which $O(\langle \Sigma \rangle)$ is a lower bound. On the other hand, as we saw in Section 2.8, the intrinsic diameter of any Σ (the greatest possible length of a shortest path in Σ between any two points of Σ) is at most half the total perimeter of Σ ; thus if acceptance time is $O(\langle \Sigma \rangle)$, the state of c_0 can depend on the state of any cell in Σ , however distant, so that there is no class of arrays that requires a lower bound higher than $O(\langle \Sigma \rangle)$.

happens iff no column of Σ contains an x . The time required for acceptance, on an $m \times n$ rectangular array, is just $n - 1$ (for the propagation of the y s up the columns) $+$ $(m - 1)$ (for the propagation of the z s across the top row), which is exactly $\frac{1}{2}(\Sigma)$. Thus $L_{\bar{x}}$ is an example in which CAs accept much faster than TAs.

Acceptance by CAs in perimeter time is of particular interest, since this is (on the order of) the fastest time possible. In the following subsections, we give a number of examples of perimeter-time acceptance algorithms for CAs. These algorithms illustrate the value of two-dimensional CAs as fast devices for array recognition.

It is straightforward to establish analogs of the upper bounds on acceptance time, discussed at the end of Section 3.4.3, for two-dimensional TBAs and BCAs; the details will not be given here.

5.5.1 Some Efficient Acceptance Algorithms for Rectangular Arrays

In this section we describe several perimeter-time (or, equivalently, diameter-time) algorithms for acceptance of specific rectangular array languages by DBCAs. Some perimeter-time algorithms for connected array languages will be presented in the next section.

(A) Row or Column Languages In Section 3.4.4 we gave a number of efficient algorithms for string language acceptance by one-dimensional DBCAs. These algorithms required times on the order of the string length. As an immediate application of these algorithms to rectangular arrays, consider the set of arrays each of whose rows belongs to a given string language, e.g., to the palindromes or well-formed parenthesis strings. We can accept such a set of arrays in $O(\text{perimeter})$ time as follows:

- (a) Each row simulates a one-dimensional DBCA, with distinguished cell at its left end, that accepts iff the row belongs to the given string language. This requires $O(\text{array width} = \text{string length})$ time.
- (b) The left-hand column simulates a one-dimensional DBCA, with distinguished cell at the top, that accepts iff the entire column consists of accepting states of the row DBCAs. This can easily be done in $O(\text{array height})$ time, once the rows have all accepted. The bottom left cell, when its row has ac-

cepted, initiates a signal that travels upward through acceptance-marked cells; if it reaches the top, the top cell accepts.

Thus the total acceptance time is $O(\text{width}) + O(\text{height}) = O(\text{diameter})$. An exactly analogous scheme can be used to accept arrays whose columns all belong to a given string language.

(B) Counting Symbols and Local Patterns We next consider languages defined in terms of the number(s) of occurrences of particular symbols. For example, let us consider arrays of x s and y s and consider the language consisting of all such arrays in which the number of x s is odd (or even), or in which the number of x s equals (or exceeds) the number of y s. In Section 3.4.3 we saw that the analogous one-dimensional languages could be accepted by converting the number of x s to a binary number, stored in the left-most $\log_2(\text{string length})$ cells, since we can then check whether the least significant bit is 0 or 1, to determine the parity of the number of x s. Similarly, if we simultaneously convert both the number of x s and the number of y s to binary numbers, both stored in the left-most cells (without interfering with one another), we can do a bit-by-bit comparison of these numbers to determine which of the numbers is greater, or whether they are equal. The binary conversion and comparison processes both take only $O(\text{length})$ time.

In two dimensions, we can similarly convert the number(s) of x s (and y s) on each row to a binary number. [We assume here that the rows are at least as long as the columns; this assures that the final binary numbers for the entire array, which are $O(\log_2(\text{area}))$ bits long, will fit on the rows, since $\text{row length} \geq (\text{area})^{1/2} > \log_2(\text{area})$. If the rows are shorter than the columns (which can be verified in $O(\text{perimeter})$ time; see the beginning of Section 5.5.2), we use an analogous process with the roles of columns and rows interchanged.] We must now add up these numbers to obtain a total sum, say on the top row. To do this, let us first mark the odd-numbered and even-numbered rows distinctively. [This takes $O(\text{height})$ time, using a signal that travels from bottom to top and alternates between two values; see the beginning of Section 5.5.2]. We now simultaneously add each even-numbered row to the odd-numbered row above it; this takes $O(\log \text{ width})$ time (see the next paragraph for the details). When the additions are complete, we shift the sums upward until they are packed into the upper half of the rows; this takes $(\text{height}/2)$ time. We now again

TABLE 5.3

Old h'_i	h'_i	New h_i	New k_i
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

simultaneously add even-numbered rows to the odd-numbered rows above them, which takes $O(\log \text{ width})$ time. The entire process is repeated, until finally the sums corresponding to the top and bottom halves of the array, in the top two rows, are added to yield the final sum. The sums never get longer than $\log(\text{area}) \leq \log(\text{width}^2) = 2 \log(\text{width})$, which is $O(\log \text{ width})$. The total shifting time is $\text{height}/2 + \text{height}/4 + \cdots = O(\text{height})$. The number of times additions are performed is $O(\log \text{ height})$ and the additions take $O(\log \text{ width})$; thus the total addition time is $O((\log \text{ height})(\log \text{ width})) \leq O(\log^2 \text{ width}) < O(\text{width})$. The entire processing time thus consists of $O(\text{width})$ for the row additions, $O(\text{height})$ for the upward shifting of rows, and less than $O(\text{width})$ for the additions of rows—a total of $O(\text{height}) + O(\text{width}) = O(\text{perimeter})$ time steps.

The addition of pairs of rows proceeds straightforwardly. Each cell (say the i th) on an odd-numbered row contains a pair of bits (h_i, k_i) ; initially h_i is the i th bit of the sum on that row, and $k_i = 0$. The cell below it on the even-numbered row also contains a pair of bits (h'_i, k'_i) with $k'_i = 0$. At the first step, the cells on the even rows go into "blank" states. The new states of the cells on the odd rows are determined as shown in Table 5.3. At subsequent steps, the new states of the odd-row cells are determined analogously, as shown in Table 5.4.

TABLE 5.4

Old h_i	Old k_{i-1}	New h_i	New k_i
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

To ensure that the additions are all done simultaneously as soon as the rows are packed, we can assume that the lowest nonblank row is specially marked; this is trivial to do initially, since the bottom row has $\#$ s below it, and the marks can be inherited each time the lowest nonblank row is added to the row above it. That row can initiate a signal that propagates upward through non-blanks. When the signal reaches the top row, it knows that the packing is complete and can initiate a synchronization process that causes all the pairs of rows to add at once. The signal propagation and synchronization steps, in preparation for the i th set of additions, take $O(\text{height}/2^{i-1})$ time, the same as the time for the preceding shifting and packing, so that our time order computation is not affected.

The new shifting and packing can begin as soon as the first step of the current addition is performed; but to insure that this addition is finished before the next one is initiated, we can proceed as follows: Assume that the cells containing bits of a sum are all marked, whether the bits are 1s or 0s; this can easily be done when the initial row sums are created, and can be preserved as part of the addition process. Concurrently with the addition, we can synchronize the marked cells on the lowest nonblank row; when the synchronization is done, we know that the addition is also (nearly) done, and we can initiate the upward signal that will indicate completion of the packing. (A simpler method of counting x s in a rectangular array is described in [5].)

As a generalization of counting x s, we can count occurrences of any given local pattern, e.g., $\begin{smallmatrix} xxx \\ x \end{smallmatrix}$. A uniquely defined cell (such as the upper left cell) of each such pattern first identifies itself; this can evidently be done by examining states of neighbors out to a bounded distance, and takes a bounded number of time steps. We then count the number of cells that have identified themselves in this way, and can then apply any desired tests or comparisons to these counts (e.g., parity, majority, etc.). We can also compute geometrical properties of the array that can be defined in terms of local patterns. For example, in an array of x s and y s, the set of x s is a union of nontouching rectangles iff it has no concave corners (compare Proposition 5.3.1), and this can be detected by marking concave corners and detecting the absence of a marked cell. Moreover, such a set of x s is a single rectangle iff it has exactly four

convex corners, and this too can be easily determined by detecting and counting the convex corners. To tell whether the rectangles are, e.g., squares, each northwest corner (i.e., each x that has xs as right and lower neighbors) can compare the numbers of upper and left-hand-edge points of its rectangle (we mark these points and shift the marks leftward or upward, respectively, through xs until they reach the corner; if they stop coming at the same time, that rectangle is a square). We can then mark the northwest corners of squares and subsequently count them.

As another example, the genus of an array of xs and ys (essentially the number of connected components of xs minus the number of components of ys ; see Section 2.7) is equal to $v - e + f$, where

v is the number of xs ,

e is the number of x^x s and xx s, and

f is the number of xx^{xx} s.

We can count these types of patterns (independently), and combine the counts (subtraction can be done analogously to addition) to compute the genus. We can thus, in particular, determine whether the array has some given genus, such as 0 or 1. All these processes take times proportional to the array perimeter or diameter. We can tell when the counting is complete by, e.g., sending a signal from the lower right corner; when it reaches the upper left corner, we know that diameter time (or some multiple of it, if we use a slow signal) has passed.

(C) Counting Connected Components It is more complicated to count the connected components in an array of xs and ys . In the following paragraphs, we describe a method, based on [6] and [7], of doing this in $O(\text{perimeter})$ time. The method makes use of a “shrinking” operation that reduces each component of xs or ys to a single point. We first describe this operation and then show how it can be used to count components of xs or ys —for example, to determine whether or not the xs are connected.

Let Ψ denote the operation that

- (a) changes a cell from x to y if its right and lower neighbors are both ys ;

- (b) changes a cell from y to x if its right, lower, and lower right diagonal neighbors are all x s;

and leaves all other cells unchanged. In other words, Ψ deletes upper left convex corners from the set S of x s, and fills upper left concave corners. If we are given the neighborhood

$$\begin{array}{cc} P & Q' \\ & Q'' \quad Q \end{array}$$

of the point P , then we change P from x to y iff $Q' = Q'' = Q = y$ and change it from y to x iff $Q' = Q'' = Q = x$. [Equivalently, if we represent x by 1 and y by 0, it is not hard to show that

$$\Psi(P) = 1 \quad \text{iff} \quad \begin{array}{l} P + Q' = 2, P + Q'' = 2, \\ \text{or} \quad Q' + Q'' + Q = 3. \end{array}$$

Note that $\Psi(P)$ depends only on two neighbors and one diagonal neighbor of P ; it requires two time steps to compute. We could have defined Ψ using other pairs of neighbors (right and upper, left and upper, left and lower) rather than the right and lower neighbors; analogous results can be obtained for these alternative definitions.

We shall next show that, in a sense to be defined more precisely below, applying Ψ to an array of x s and y s preserves the connectness of both the x s and the y s. Our definition of Ψ assumes that we use 4-connectedness for the x s and 8-connectedness for the y s. If we want to use the opposite types of connectedness, we should define Ψ to change P from x to y iff $Q' = Q'' = Q = y$, and to change P from y to x iff $Q' = Q'' = Q = x$. [It is not hard to see that this is equivalent to $\Psi(P) = 1$ iff $P + Q = 2$ or $P + Q' + Q'' \geq 2$, where 1 represents x and 0 represents y .]

For brevity, denote $\Psi(P)$ by P' . Let S and S' be the sets of x s before and after a given application of Ψ , and let $\bar{S}$ and $\bar{S}'$ similarly be the sets of y s. For any set T , let T'_x and T'_y be the sets of points of S' and of $\bar{S}'$, respectively, that either lie in T or have points of T as right and lower neighbors. We shall now prove that if U is a component of S , then U'_x is a component of S' , and if V is a component of $\bar{S}$, then V'_y is a component of $\bar{S}'$. (We assume here that U and V consist of more than one point; if not, it is clear that these points disappear under Ψ and that U'_x and V'_y are empty.)

Note first that if P' is in U'_x but not in U , we have $P' = x$ but $P = y$ (since P has points of U as right and lower neighbors, if it were

x , it too would be in U , since U is a component of xs). Hence P must have a neighborhood $\begin{smallmatrix} P & P_1 \\ P_2 & P_3 \end{smallmatrix}$ in which $P_1 = P_2 = P_3 = x$. Since $P_1 = x$ and $P_3 = x$, we must have $P'_1 = x$, and similarly $P'_2 = x$; thus P' has neighbors P'_1 and P'_2 that are in both U'_x and U . It follows that to prove the connectedness of U'_x , we need only show that any two points that are in both U and U'_x have a path joining them in U'_x .

Let Q_1, Q_2 be two such points; let π be a path joining them in U (π exists since U is connected) that is as short as possible; and let R be any point on π such that $R' = 0$. (If there is no such R , π is a path in U'_x and we are done.) Thus $R = x$ and $R' = y$, so that the neighborhood of R is of the form

$$\begin{array}{c} R_1 R_2 \\ R_3 R \quad y \\ y \end{array}$$

Since π is as short as possible, it must pass through both R_2 and R_3 ; hence R_2, R_3 are in U , so that $R_2 = R_3 = x$. It follows that $R'_1 = R'_2 = R'_3 = x$, so that R'_1, R'_2, R'_3 are all in U'_x . We can thus replace R by R_1 on π , and by doing this for every such R , we obtain a path π' in U'_x from Q_1 to Q_2 , as desired. Thus U'_x is connected.

We next show that U'_x is a component of S' —or equivalently, that any point of S' adjacent to U'_x must be in U'_x . Suppose first that the given point P is in both S and S' , i.e., $P = P' = x$. If P is adjacent to U , it is in U and hence in U'_x , and we are done. Otherwise, P must be adjacent to some $Q \in \hat{S}$ for which $Q' \in S'$ is in U'_x . The neighborhood of P must thus look like

$$\begin{array}{ccc} P & R & \\ Q & Q_1 & \\ Q_2 & Q_3 & \end{array} \quad \text{or} \quad \begin{array}{ccc} P & Q & Q_1 \\ R & Q_2 & Q_3 \end{array}$$

where Q_1, Q_2, Q_3 are in U (they are all in S since $Q = y$ but $Q' = x$; and Q_1, Q_2 are in U since Q' is in U'_x). Since $Q = y$ but $P' = x$, we cannot have $R = y$; but if $R = x$, P is connected to U and so is in U . Similarly, if P is in $\hat{S}$ and S' , i.e., $P = y$ but $P' = x$, then P has a neighborhood of the form

$$\begin{array}{ccc} & P_1 & P_2 \\ P_3 & P & Q_1 \\ P_4 & Q_2 & Q_3 \end{array}$$

with Q_1, Q_2, Q_3 in S . If Q_1 or Q_2 is in U'_x , then it is also in U , so that P is in U'_x . Otherwise, P_1 or P_3 must be in U'_x , and if $P \notin U'_x$, this means that P_2 or P_4 must be in U ; but then Q_1, Q_2, Q_3 are in U , so that P is in U'_x —a contradiction. Thus U'_x is a component.

Note finally that if U has more than one point, U'_x cannot be empty; for example, if P is the leftmost of the uppermost points of U , then $P' = x$ (since either its right or lower neighbor is in S), so that P is in U'_x . (By the same argument, if V is a component of $\tilde{S}$ and has more than one point, V'_y cannot be empty.)

We now turn to the proof that if V is a component of $\tilde{S}$, then V'_y is a component of $\tilde{S}'$. Let π be a path in V that is as short as possible, and let ρ be a run of points of π that are not in $\tilde{S}'$, but such that the points of π immediately preceding and following ρ are in $\tilde{S}'$. Then the neighborhood of ρ must be of the form, e.g.,

$$\begin{array}{ccccccc} & P_1 & P_2 & P_3 & & & \\ & & P_4 & Q_1 & x & & \\ P_5 & P_6 & Q_2 & x & x & & \\ P_7 & Q_3 & x & x & & & \\ P_8 & x & x & & & & \end{array}$$

where the Q s are in ρ . Thus P_4 and P_6 are in $\tilde{S}'$, so that if the point of π preceding ρ is P_5 or P_7 , and the point following ρ is P_1 or P_2 , we can replace ρ by the subpath P_6, P_4 , which lies in $\tilde{S}'$. On the other hand, if the preceding point is P_8 , we have $P'_7 \in \tilde{S}'$, and if the following point is P_3 , we have $P'_2 \in \tilde{S}'$, so that in these cases ρ can be replaced by P_7, P_6, P_4 , by P_6, P_4, P_2 , or by P_7, P_6, P_4, P_2 , all in $\tilde{S}'$. It follows that any (shortest) path in V can be converted into a path in V'_y . Thus any two points in V'_y are joined by a path in V'_y , so that V'_y is connected.

To show that V'_y is a component, we prove that any point of $\tilde{S}'$ adjacent to V'_y must be in it. Let the given point P be in S ; its neighborhood must then be

$$\begin{array}{ccc} & P_1 & \\ P_2 & P & Q_1 \\ & Q_2 & \end{array}$$

where Q_1, Q_2 are in $\tilde{S}$. If Q_1 or Q_2 is in V'_y , they are both in V , so that P is in V'_y . Otherwise, P_1 or P_2 , say P_1 , must be in V'_y , so that $P'_1 = y$; but since $P = x$, this means that $P_1 = y$, so that P_1 is in V , and this implies Q_1, Q_2 in V , proving $P \in V'_y$. On the other hand, let P be in $\tilde{S}$; then if it is adjacent to V , it is in V , hence in V'_y , and

we are done. Suppose it is adjacent to a point Q of V'_y that is not in V , so that its neighborhood looks like

$$\begin{array}{ccc} P & & P \ Q \ R_1 \\ Q \ R_1 & \text{or} & R_2 \\ R_2 & & \end{array}$$

where R_1 and R_2 are in V ; then P is also in V and hence in V'_y .

Finally, we show that if U is adjacent to V , then U'_x is adjacent to V'_y . We recall (Section 2.6) that U adjacent to V implies that U surrounds V or vice versa; suppose the former. Let P be the leftmost of the uppermost points of V , so that P 's upper neighbor Q is in U . Then it is not hard to see that $P' \in S'$ and $Q' \in \bar{S}'$, so that $P' \in U'_x$ and $Q' \in V'_y$.

For any component U of S , let i_u and j_u be the coordinates of the leftmost row and uppermost column that contain points of U . Let P be any point of U at maximum city block distance from (i_u, j_u) . Thus the right and lower neighbors of P cannot be in U and hence are in $\bar{S}$, so that $P' = 0$. Thus applying Ψ decreases the maximum city block distance of U from (i_u, j_u) .

No point of U'_x can lie to the left of i_u or above j_u , since a point to the left of i_u can neither be in U nor have its lower neighbor in U , and a point above j_u can neither be in U nor have its right neighbor in U . On the other hand, any leftmost uppermost or uppermost leftmost point of U must be in U'_x (as long as U has more than one point), since such a point must have either its right or lower neighbor in U . Thus i_u and j_u are still the leftmost and uppermost coordinates for U'_x , as they were for U .

The remarks in the last two paragraphs imply that when Ψ is repeatedly applied, the leftmost and uppermost coordinates (i_u, j_u) of any U remain unchanged, but the city block distance of U from (i_u, j_u) keeps decreasing. This means that U must eventually shrink to a single point, which must evidently be the point (i_u, j_u) , and this point then vanishes when Ψ is applied again. The number of steps required for this to happen is at most (twice[§]) the original maximum city block distance of U from (i_u, j_u) . Analogous remarks apply to any component V of $\bar{S}$. (The only exception is the background component, which is not surrounded by any component of U .) Note that the time required for a component to shrink to a

[§] Since Ψ takes two time steps to apply; we ignore this factor of 2 below.

point is at most the diameter of its framing rectangle, which is less than the diameter of Σ .

Using Ψ , we can design a DBCA that counts connected components of x s or y s. We apply Ψ repeatedly; this shrinks every connected component, except for the background component of y s, to a single point (which then vanishes), in time less than the diameter of Σ . When single-point components are created (or initially present), we can easily detect them, since they are x s all four of whose neighbors are y s, or y s all eight of whose neighbors are x s. Thus at the same time that Ψ annihilates them, we can replace them by special marks, say w and z , that can occupy a cell without interfering with the processing of the x s and y s by Ψ . We can now count these special marks, as described earlier in this section: Add them up in each row, then add the row sums. [We know that $O(\text{diameter})$ time steps suffice for the w s and z s to have all been created, shifted to the left ends of their rows, and counted there; thus after $O(\text{diameter})$ time steps we can safely initiate the process of adding the row sums.] This gives us counts of the numbers of components of x s and y s in the original Σ (except for the background component of y s). Thus we can accept Σ iff, e.g., these numbers have given values or given parities; e.g., if the x s are connected, or if there are no holes in the x s (nonbackground components of y s). Acceptance based on comparison of these numbers, e.g., more components than holes, can be done without using Ψ , since the difference between the number of components of x s and the number of nonbackground components of y s is just the genus, which can be computed more simply.

We can also use Ψ to define DBCA acceptors for various other languages. For example, to accept iff the x s consist of a single closed curve, we can first check the fact that every x has exactly two x s as neighbors, which implies that the x s are a union of non-touching simple closed curves (Section 2.3); and we can then check that the x s are connected (or equivalently, that they have exactly one hole).

5.5.2 Some Efficient Acceptance Algorithms for Connected Arrays

In this section we present some perimeter-time algorithms for DBCA acceptance of specific connected array languages. In particular, we may consider languages defined by the geometrical properties of the array Σ (of non- $\#$ s) itself. As one example, in Section

5.3 we showed how a DBCA can accept Σ iff it is rectangular. It is also easy to define DBCA acceptance algorithms for rectangles having particular properties, e.g., side lengths odd or even, height equal to (or greater than) width, etc. For example, to detect evenness of rectangle width, the upper right corner sends a signal to the upper left corner (c_0), and the signal alternates between two states, say u and v (i.e., if a cell's right neighbor is u , the cell becomes v , and vice versa), with the initial signal being u ; if c_0 receives v , it accepts. To detect the fact that height is greater than width, the lower right corner sends a signal to c_0 that travels diagonally, i.e., alternately upward and leftward (if a cell's lower neighbor is u , the cell becomes v ; if a cell's right-hand neighbor is v , the cell becomes u). If a u has a $\#$ on its left, it becomes a w ; this w moves upward one step and becomes z ; z moves upward until it reaches c_0 , which then accepts. If w reaches c_0 , the height and width are equal. Algorithms for acceptance of various nonrectangular array languages will now be described.

(A) *Simple-Connectedness* To determine whether or not Σ is simply-connected (i.e., has no holes), we can proceed as follows [4]: In the proof of Theorem 5.4.1 we showed that the distinguished cell c_0 of Σ can identify itself in time on the order of the outer perimeter of Σ . (The processes on inner borders may still be going on, but they will not lead to anything and can safely be ignored.) The outer border of Σ then marks itself by simulating a border-following process starting from c_0 ; this again takes outer perimeter time. [A cell c may be on both the outer border and an inner border, but this can be detected by checking whether the BF simulation has made use of all the $\#$ neighbors of c ; if not, the mark is modified to indicate that c is on more than one border.] After the marking is completed, each outer border cell c_1 sends out a signal that moves rightward along its row until it reaches another border cell c_2 and hits a $\#$; this must happen in a number of time steps at most equal to the width of Σ , which is evidently less than half the outer perimeter. If c_2 is marked (i.e., is on the outer border), the signal is erased. If c_2 is not marked, or has a modified mark, the signal generates a reply that moves back along the row until it reaches c_1 . (If Σ has a hole, this must happen. Indeed, let P be a hole point that is as far to the left as possible; thus P 's left neighbor Q is a hole border point. If we move leftward from Q

through non- $\#$ s until we hit a $\#$, the last non- $\#$ must be an outer border point, since there are no hole points to the left of P . Thus if Σ has a hole, some outer border point must have the property that when we move rightward from it through non- $\#$ s until we hit a $\#$, the last non- $\#$ must be a hole border point.) When this happens (or if c_1 had a modified mark to begin with), c_1 generates a signal that travels around the outer border to c_0 . If c_0 receives such a signal, it knows that Σ is not simply connected. If it fails to receive such a signal within (say) twice outer perimeter time after the marking of the outer border was completed (it can determine this time by sending a half-speed signal around the outer border), it knows that Σ is simply-connected, and can accept. The entire process takes $O(\text{outer perimeter})$ time.

(B) Arcs and Curves To determine whether Σ is an arc or curve (Section 2.3), c_0 identifies itself and marks the outer border. When this is finished, each outer border cell verifies that it has no unmarked neighbors, and at most two marked neighbors. If a cell fails to verify this, it sends a signal to c_0 , which then knows that Σ is not an arc or curve. If this signal does not arrive within outer perimeter time, c_0 knows that Σ is an arc or curve. If two border cells have only one neighbor, Σ is an arc; if all of them have two neighbors, Σ is a curve.

(C) Pattern Detection and Counting As we saw in the proof of Theorem 5.4.2, the distinguished cell c_0 of a DBCA can send out a signal and receive a reply from every cell in $O(\text{diameter})$ time and hence in $O(\text{perimeter})$ time. This reply can inform c_0 about the presence or absence of specific symbols or local patterns in Σ . We can also determine, e.g., the parity of the number of these symbols; in fact, each cell can receive parity information from its sons in the spanning tree and compute its own parity (i.e., the parity corresponding to the set of cells below it in the tree) for transmittal to its own father.

Counting specified symbols (or local patterns) is more difficult. For a non-BCA, there would be no problem; we could simply construct a containing rectangle and use the methods of Section 5.5.1 to count the various types of non- $\#$ s in that rectangle. For a BCA, we can use a different algorithm [8] that once again involves constructing a spanning tree. In such a tree, we know that each node has degree (number of non- $\#$ sons) ≤ 3 , since a node has only four neighbors and one of them must be its father. (The root node of

the tree has no father, but since it is the upper left corner of Σ , it has at most two non- $\#$ sons.) In the following paragraphs, we describe an algorithm in which each node of the tree successively outputs (to its father) the base-4 digits of the number of nodes in its subtree, least significant digit first. The root node, which has no father, can shift these digits down the tree so that they are stored along a longest path.[§] [Let h , the tree height, be the length of such a longest path; then it is not hard to see that the total number of nodes in the tree is at most $(1 + 3 + 3^2 + \dots + 3^h) < 3^{h+1}$, so that there is room to store this number along the path.]

The basic idea of the algorithm is as follows: We assume that the tree has been constructed, so that each cell knows which of its neighbors are its father and its sons. The states of a cell are triples of the form (x, y, z) , where

$x = 0, 1, 2$, or 3 represents a sum digit (to base 4);

$y = 0, 1, 2$, or 3 represents a carry digit. [Since a cell has at most three non- $\#$ sons, each of which is sending it a base-4 digit (≤ 3), and since its own digit (the carry digit from the previous addition) is also ≤ 3 , it can add these digits and obtain a sum $\leq 4 \cdot 3 = 12$; this sum can thus be represented using a pair of base-4 digits, $y \cdot 4^1 + x \cdot 4^0$.]

z indicates the type of tree node and the action to be taken with respect to the addition process; the significance of z will become clear in the course of describing the algorithm.

At the first step, each cell changes to state $(1, 0, z_1)$ if it is a leaf node, or to $(1, 0, z_2)$ if it is not. At each subsequent step,

- (1) A cell in state $(1, 0, z_2)$ remains unchanged unless all its sons are in states with third terms z_1 or z_4 ; it then changes to $(1, 0, z_3)$. Informally, the z_3 indicates that the cell is ready to add its sons' inputs; the z_4 indicates a cell that has just added its sons' inputs.
- (2) A cell in state $(x, 0, z_3)$ changes to state (x_1, x_2, z_4) , where $4x_1 + x_2 = x +$ (the sum of the digits shifted up from the cell's sons). It can be shown, by induction on tree height, that whenever a cell c is in a state with third term z_3 , its sons are all in states with third terms z_1 or z_4 , so that they all have sum digits that can be shifted up to c for addition. It

[§]This shifting is understood to be independent of, and not to interfere with, the upward shifting of the digits to be described below.

can also be shown that under these circumstances the second (carry digit) term of c 's state must be zero.

- (3) A leaf node cell [in state $(1, 0, z_1)$] shifts its sum digit (1) up whenever its father has third term z_3 . It can be shown that this happens only once.
- (4) A cell in state (x, y, z_4) remains unchanged unless its father is in a state with third term z_3 (or unless it is the root node); it then shifts x up to its father (or down along the storage path, if it is the root node), and changes to state $(y, 0, z_3)$.

Thus z_3 serves as a signal to a cell's sons that they can shift their sum digits up for addition (2)–(4). This ensures that all sons shift up at the same time, even though the information may have arrived at the sons at different times (because their subtrees have different heights). Note that the cell cannot be in a z_3 state until all its sons are ready (1). The z_4 symbol serves as a signal to a cell's father that the cell is ready to shift its sum digit up; at the same time, it serves as a signal to the cell's sons that the cell is not ready for them to shift their digits up to it. When a cell shifts a digit up to its father, its own state changes from z_4 to z_3 (4), indicating to its sons that it is ready to accept their digits again. Thus as long as digits remain to be shifted up to a cell, its state alternates between z_3 and z_4 .

It is not hard to prove [8] that a cell at height $h \geq 1$ in the tree remains in state $(1, 0, z_1)$ until step $z_1 + 1$, when it changes to state $(1, 0, z_3)$. At the end of that step, all the cell's sons' states have third terms z_1 or z_4 . (In general, whenever a cell's third term is z_3 , its sons' third terms are all z_1 or z_4 .) Thus the cell now adds its sons' outputs; by induction hypothesis, these are the least significant digits (to base 4) of the numbers of nodes in the sons' subtrees, so that the cell's sum digit is now the least significant digit of the sum of these sums, i.e., of the number of nodes in its own subtree. When the cell's father is ready, the cell outputs this digit, changes its carry digit to a sum digit, and becomes ready to accept further inputs from the sons. These can immediately output to it the next least significant digits of their subtrees' numbers of nodes. The cell adds these to its own sum digit, which yields the next least significant digit of its own subtree's number. This process continues until the sons have no further information to transmit. It can be shown that this takes $3h + 3$ time steps: $h + 1$ until the additions start, and two steps per digit (alternating

between z_3 and z_4 states) until they are finished. Thus the total time required for the sum to be computed for the entire tree is about three times the distance from the root cell to the (leaf) cell farthest from it, which is on the order of the diameter of Σ . It is easy for the root cell to know when this process has been completed (e.g., by using a signal that travels to a farthest cell at half speed and then back to the root at full speed; when the return signal arrives, the elapsed time is essentially $3h$).

It is trivial to modify this algorithm to count the number of cells having a particular state, rather than all the cells; or to count the number of occurrences of a particular local pattern (a uniquely defined cell in each such occurrence first goes into a special state, and these states are then counted).

REFERENCES

1. A. R. Smith III, Cellular automata complexity trade-offs, *Information and Control* **18**, 466–482 (1971).
2. H. B. Nguyen and V. C. Hamacher, Pattern synchronization in two-dimensional cellular spaces, *Information and Control* **26**, 12–23 (1974).
3. I. Shinahr, Two- and three-dimensional firing squad synchronization problems, *Information and Control* **24**, 163–180 (1974).
4. A. R. Smith III, Two-dimensional formal languages and pattern recognition by cellular automata, *Proc. SWAT*, 12th pp. 144–152, (1971).
5. S. R. Kosaraju, On some open problems in the theory of cellular automata, *IEEE Trans. Computers* **23**, 561–565 (1974).
6. W. T. Beyer, Recognition of topological invariants by iterative arrays, MAC TR-66, MIT, Cambridge, Massachusetts, 1969.
7. S. Levialdi, On shrinking binary picture patterns. *Comm. Assoc. Comput. Mach.* **15**, 7–10 (1972).
8. A. Wu, Cellular graph automata, 2, Tech. Rep. No. 621, Computer Science Center, Univ. of Maryland, College Park, Maryland, December 1977, Section 1.3.3.

Chapter 6

PYRAMID CELLULAR ACCEPTORS[§]

6.1 INTRODUCTION

This chapter introduces a special class of cellular acceptors, called pyramid acceptors, that can accept many languages in much less than $O(\text{diameter})$ time, at only a slight cost in additional hardware. A pyramid acceptor is a stack of two-dimensional cellular automata in which the bottom ("base") layer is $2^n \times 2^n$, the next layer is $2^{n-1} \times 2^{n-1}$, and so on, until the top layer consists of a single "apex cell." Each cell, in addition to being able to sense the states of its four neighbors (or "brother" cells), can also sense the states of four "son" cells in the layer below it, and of a "father" cell in the layer above it. The input to be accepted is stored in the base layer; the other layers are initially in a quiescent state. The input is accepted if the apex cell enters an accepting state. Note that the pyramid has $n + 1$ layers, so that the minimum number of time steps required to transmit information from any base cell to the apex cell is $n = \log_2(2^n)$, i.e., $O(\log \text{diameter})$ of the input.

In Section 6.2 we define pyramid acceptors, as well as their analogs, "triangle acceptors," which accept one-dimensional input. We also introduce a simplification of these definitions in which a cell is only allowed to sense the states of its sons (but not of its brothers or its father), so that information can only flow upward from the base to the apex. Section 6.3 presents $O(\log \text{diameter})$ -time algorithms for local property detection and counting. Section

[§] The material in this chapter is primarily the work of Charles R. Dyer, and originally appeared in [1,2].

6.4 compares bottom-up acceptors to finite-state acceptors, frontier-to-root tree acceptors, and diameter-limited perceptrons. Section 6.5 describes a number of other triangle and pyramid acceptance algorithms. In this chapter, all cellular acceptors are bounded; for brevity, we omit the prefix "B".

6.2 CELLULAR PYRAMIDS

In this section we define a pyramid cellular acceptor. In order to simplify the exposition of this model, we first present its lower-dimensional analog—the triangle cellular acceptor. We also introduce a simplification that restricts the neighbor relation to being "bottom-up."

6.2.1 Triangle Cellular Acceptors

A *triangle cellular acceptor* (TCA) is a triangular stack of strings of cells, where the bottom string is of length 2^n , the next 2^{n-1} , and so on. Surrounding the triangle is a border of cells in the boundary state $\#$. The transition function maps sextuples of states into states, $\delta: Q^6 \rightarrow Q$ (in the nondeterministic case, $\delta: Q^6 \rightarrow 2^Q$). As with the CA, we have the usual boundary state restrictions in order to confine the computation to the non- $\#$ cells: $\# \in \delta(q, \dots)$ iff $q = \#$.

Those cells whose states are the arguments of δ , for a given cell c , consist of c 's left and right neighbors (brothers) on its own level, two son cells of c on the level below it, and one father cell on the level above it. If c is the i th cell on its level, its sons are cells $2i - 1$ and $2i$ on their level, and its father is cell $\lceil i/2 \rceil$ on its level.

The apex cell of the triangle is called the *root*, and the bottom level of 2^n cells, the *base*. The *height* of the triangle is the length of a shortest path from the root to a cell in the base. Thus a triangle with a base of length 2^n has height n and $2^{n+1} - 1$ non- $\#$ cells.

The figure below illustrates a TCA with a base of length 8, where neighbor cells have been joined by line segments. The non- $\#$ cells are numbered in left-to-right order on each level. For example, the father of cell 3 on level 1 is cell 2 on level 2, its left and right brothers are cells 2 and 4 on its own level, and its left and right sons are cells 5 and 6 on level 0.

other in state y , it goes into state y . (The states of its brothers and father are irrelevant.) The accepting state is x . It is not hard to see that after k time steps, the cells on level k all change from state q to states x or y . It is easy to see by induction that if a cell goes into state x , the number of x s below it on the base (sons of sons . . . of its sons) is odd, while if it goes into state y , that number is even. [*Proof*: This is trivial for cells at level 0; and if it is true for level $k - 1$, it is also true for level k , since the sum of two odd or even numbers is even, while the sum of an odd and an even number is odd.] If we take y to be the accepting state, we see that the root cell accepts (or rejects, by going into state x) at time step n . Note that this acceptance time is equal to the logarithm of the length of the input. In contrast, recognition of this language by a BCA or TBA requires time proportional to the length of the input (i.e., linear time).

6.2.2 Pyramid Cellular Acceptors

A *pyramid cellular acceptor* (PCA) is a pyramidal stack of arrays of cells in which the bottom layer has size $2^n \times 2^n$, the next lowest $2^{n-1} \times 2^{n-1}$, and so forth, the $(n + 1)$ st layer consisting of a single cell, called the root. Each cell now has nine neighbors—four son cells in a 2×2 block on the level below, four brother cells on its own level, and one father cell on the level above. If the cell's coordinates in its layer are (i, j) , its sons' coordinates are $(2i - 1, 2j - 1)$, $(2i - 1, 2j)$, $(2i, 2j - 1)$, and $(2i, 2j)$, on the layer below; its brothers' coordinates are $(i - 1, j)$, $(i + 1, j)$, $(i, j - 1)$, and $(i, j + 1)$ on its own layer; and its father's coordinates are $(\lceil i/2 \rceil, \lceil j/2 \rceil)$ on the layer above.

The transition of function δ of a PCA maps 10-tuples of states into states—or sets of states, in the nondeterministic case. Otherwise, the conventions set up for the TCA still hold. The input array Σ consists of the initial states of the bottom layer, henceforth called the *base array*. The root is again the accepting cell.

Since the pyramid has only $n + 1$ layers, a PCA has a lower bound acceptance time proportional to the logarithm of the base array's diameter. Furthermore, the additional hardware cost required to achieve this potential time saving is moderate—less than a third more cells than in the base $(1 + \frac{1}{4} + (\frac{1}{4})^2 + \dots = 1/(1 - \frac{1}{4}) = \frac{4}{3})$.

6.2.3 Bottom-Up Acceptors

Alternative neighborhood definitions can be made that restrict information transmission through a PCA. In particular, we now define a simplification in which the only neighbors of a cell are its sons, so that state information can move only one way up the pyramid. A *bottom-up pyramid acceptor* (UPCA) is a PCA whose state transition function is modified to be $\delta: Q^5 \rightarrow Q$. In this case, the next state of a cell depends only on the current states of that cell and its four sons.

Similarly, we define a *bottom-up triangle acceptor* (UTCA) to be a TCA in which the transition function is defined as $\delta: Q^3 \rightarrow Q$ and the next state of a cell depends only on the current state of the cell and its two sons. In this chapter we establish some results regarding the acceptance power of PCAs and TCAs, and in particular of UPCAs and UTCAs.

6.2.4 PCAs Are Equivalent to CAs

PCAs are faster than CAs for some tasks, but they are not inherently more powerful. In fact, a CA can simulate a PCA, though the simulation is rather inefficient. For simplicity, we demonstrate this in the one-dimensional case, i.e., we show how a one-dimensional CA can simulate a TCA.

Since a TCA of base length 2^n contains $2^{n+1} - 1$ cells, we use for our simulation a CA of length 2^{n+1} ; or, equivalently, we use one of length 2^n , and have each cell simulate a pair of adjacent cells in a length- 2^{n+1} array. Each CA cell will simulate one TCA cell, with the correspondence defined by a breadth-first scan of the TCA; for example, if $n = 4$, the TCA cells

$$\begin{array}{ccccccc} & & & & 1 & & & & \\ & & & & 2 & 3 & & & \\ & & & 4 & 5 & 6 & 7 & & \\ 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \end{array}$$

are mapped into the CA cells

(0), 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15.

In this mapping, the cell in the i th position has its brother(s) in positions $i \pm 1$, its father in position $\lfloor i/2 \rfloor$, and its sons in positions $2i$ and $2i + 1$.

To simulate a single cycle of the TCA, each CA cell must have access to the states of its corresponding TCA cell's brothers, father, and sons (if they exist). It has immediate access to the brothers, since they are adjacent to it in the CA. If the TCA cell is at the end of a row, so that it has only one brother (or none, in the case of the apex cell), the CA can know this fact by specially marking those cells whose positions are powers of 2, corresponding to beginnings of TCA rows; the marks can then inhibit cells 2^i and $2^i - 1$ from treating each other as brothers. The marking can be done by a simple adaptation of the son-finding algorithm to be described immediately below.

To send information about its state to its father cell, a CA cell sends two signals leftward, one traveling at three times the speed of the other (i.e., one moving at every time step, the other waiting two time steps between moves). When the fast signal reaches the left end of the CA, it bounces back and moves rightward until it meets the slow signal, at which point the signals cancel out. It can be readily verified that this meeting takes place first for the signals from cell 1, then for those from cell 2, etc., so that all the cells can send their signals out simultaneously without danger of confusion. (Meetings of slow signals with fast signals that are still moving leftward are ignored.) Moreover, the pair of signals from cell i meets just at position $\lfloor i/2 \rfloor$. Thus the signals can inform cell $\lfloor i/2 \rfloor$ about the state of cell i . For cell 1, the meeting occurs at cell 0, which is not simulating a TCA cell since cell 1 has no father.

Similarly, to send information to its son cells, a CA cell sends a fast signal leftward and a slow signal rightward. The fast signal bounces off the left end and moves rightward until it overtakes the slow signal and they cancel. Again, this happens exactly in sequence for the signals from cells 1, 2, . . . , so that the signals can all be sent simultaneously. Moreover, the signals from cell i meet just at position $2i$, so that the signals can inform cell $2i$ (and its neighbor $2i + 1$) about the state of cell i . For cells 2^{n-1} , . . . , $2^n - 1$ the slow signal reaches the right end before being overtaken; in this case the signals can be allowed to die, since these TCA cells have no sons.

Note that this process takes time $2^{n+1} + 2^n = 3 \cdot 2^n$ (the time required for the messages from cell 15 and cell 7 to reach one another). In other words, simulating a single TCA time step takes the CA $O(\text{diameter})$ time.

PCAs can simulate CAs, since cells in the base array are identi-

fiable (only they have $\#$ cells as sons). Thus all other cells can remain in the quiescent state $\natural$ while the base array copies the transitions of the CA, using only state information from brother cells. If the CA's upper left corner cell ever enters an accepting state, the father of that cell in the PCA's base array can detect this and send an acceptance signal to the root. Therefore, PCAs are equivalent to CAs.

6.2.5 Nondeterministic UPCAs Are Equivalent to CAs

In this section we show that any nondeterministic (or deterministic) CA can be simulated by a nondeterministic UPCA (for brevity: NUPCA). This result was established by A. Nakamura and C. R. Dyer [3]. Given a CA M , we shall construct an NUPCA B that simulates a single step of M in $O(\text{diameter})$ steps of B . Conversely, a nondeterministic CA can simulate a NUPCA, as seen in Section 6.2.4; hence nondeterministic CAs and NUPCAs are equivalent.

Each base cell in B nondeterministically chooses a state from Q (the state set of M) at nondeterministically chosen time steps (call them B -steps), while also remembering the state it chose at the previous B -step. Thus at the end of B -step t each base cell c stores a pair of states (p, q) from Q , where p and q are the states chosen by c at B -steps t and $t - 1$, respectively.

To check whether or not the new configuration legally follows from the previous one, we must verify that the new state (q) of each cell c is in the range of δ given the previous states (ps) of c and of its four neighbors. In other words, we must verify, for each block of cells

$$\begin{array}{c} c_n \\ c_w \ c \ c_e \\ c_s \end{array}$$

that $q \in \delta(p, p_n, p_w, p_e, p_s)$. This involves verifying that

$$\begin{array}{ccccc} & (p_n, q_n) & & & \\ (p_w, q_w) & & (p, q) & & (p_e, q_e) \\ & (p_s, q_s) & & & \end{array}$$

is one of a finite number of legal patterns (in which q_n, q_w, q_e , and q_s are arbitrary).

We shall see in Section 6.3.1 that a deterministic UPCA can detect arbitrary local patterns in $O(\text{diameter})$ time steps. Using the

algorithm described there, the root can decide in (diameter + log diameter) time steps whether the base's configuration at B -step t was a legal successor to the configuration at B -step $t - 1$. The algorithm is easily modified to verify that all base cells choose their new states synchronously and at intervals $\geq$ (diameter + log diameter) time steps, so that the detection algorithm used to confirm the legality of B -step $t + 1$ does not commence before the legality of B -step t has been tested.

In addition, the root can check for each B -step whether or not the upper-left-corner base cell was in an accepting state. Hence if the CA goes through a sequence of configurations leading to acceptance, the NUPCA can nondeterministically guess the sequence, check its legality, and determine that an accepting state was entered. Note that it does not matter whether the CA is deterministic or nondeterministic.

In one dimension, we shall see in Section 6.3.1 that a deterministic UTCA can detect arbitrary local patterns in (log diameter) time steps. Hence the base cells in the NUTCA case can enter new states at each time step, so that the simulation is real time (following a log diameter start-up delay).

6.2.6 Deterministic UPCAs Are Periodic after Polynomial Time

If each cell in a UPCA changes state every r th time that its sons change state, and the cells in the base change state every r th time step, then the period of the cells in layer k is r^k , so that the apex cell has period $r^n = 2^{n \log r} = (2^n)^{\log r}$, which is polynomial in the base diameter.

We shall now show that the behavior of any UPCA must be periodic with period polynomial in the base diameter. To see this, note that the base cells are finite-state automata with constant input; hence their outputs are periodic, say with periods $m_1, \dots, m_{2^n}$ where each $m_i \leq |Q|$ (the number of states). Furthermore, given any cell whose inputs are periodic, say with periods i, j, u, v , its output must also be periodic with period $\leq |Q| \text{ LCM}(i, j, u, v)$, where LCM means "least common multiple". Thus any cell in the k th layer has periodic output with period

$$\begin{aligned}
&\leq |Q| \text{LCM}[|Q| \text{LCM}[|Q| \text{LCM}[\dots]]] \quad (k \text{ repetitions}) \\
&= |Q|^k \text{LCM}[\text{a set of } m_i s] \\
&\leq |Q|^k |Q|!
\end{aligned}$$

since the m_i s all $\leq |Q|$, so their LCM $\leq |Q|!$. In particular, the apex cell has period $\leq |Q|^n |Q|! = 2^{n \log |Q|} |Q|! = |Q|! (2^n)^{\log |Q|}$, which is polynomial in the base diameter.

It follows from these remarks that the complement of a UPCA language is always a UPCA language. Indeed, given a UPCA A that accepts the language L , suppose that the cells of A have $|Q|$ states. We can then construct another UPCA A' that simulates A and also "counts" (as in the first paragraph above) with apex cell period greater than $|Q|! (2^n)^{\log |Q|}$. When the apex cell completes one period, if the A simulation has not yet accepted its input, A' can accept.

6.2.7 "Up-Down" PCAs Are Equivalent to PCAs

We finally show that if information in a PCA is allowed to move both upward and downward but not sideways (i.e., the state of each cell depends on those of its father and son cells, but not on those of its brother cells), the resulting "up-down" PCA can simulate an arbitrary PCA. By Section 6.2.4, it suffices to show that such PCAs can simulate CAs. For simplicity, we show this in one dimension; the generalization to two dimensions is immediate.

Given a CA with input size m , let n be the smallest integer such that $2^n \geq m$. The input string will be left justified in the base of the TCA with the rightmost $2^n - m$ cells initialized to state $\#$.

To simulate a single step of the CA, each TCA base cell must have access to the states of its corresponding CA cell's two brother cells. Every two cells that are adjacent in the CA have a least common ancestor (LCA) in the TCA. Furthermore, the TCA cell that is the LCA for an adjacent pair in the CA cannot be the LCA for any other pair. Therefore every two adjacent cells in the base of the TCA can exchange state information as follows:

At time step 1, cells in layer 1 copy the pairs of states in their base segments. At time step 2, left (right) sons in layer 0 copy the right (left) states stored in their father cells. At the same time, cells in layer 2 store the ordered quadruples of states in their base segments by copying the state pairs stored in the layer 1 cells. Thus at

the end of step 1, $m/2$ LCAs have been found, and at the end of step 2, each base cell knows the state of one of its brothers. At step 3, left (right) sons in layer 1 copy the third (second) states stored in their fathers' quadruples of states and mark them ℓ (or r). Also, cells in layer 3 compute their own ordered quadruples of states by copying the first and fourth states from their two sons' quadruples. Thus at the end of step 2, $m/4$ more LCAs are found in layer 2. After step 3, the states of these $m/4$ adjacent cell pairs have been swapped by the sons of their LCAs. At subsequent steps, any right (left) son of a father with state marked ℓ (r) copies its father's state. In this way, the state information propagates back down the TCA to the proper brother cell.

Similarly, state quadruples are computed higher and higher in the TCA until the LCA of every adjacent pair of base cells is found. At time n , the root is left with a pair of states, which are the states of the leftmost and rightmost cells in the base. The brothers of these cells have state $\#$, and this information can be returned to the proper base cells in the same manner. Thus simulating one step of a CA takes $2n$ or $O(\log \text{ diameter})$ time for the up-down TCA.

6.2.8 Deterministic CAs Are Stronger than Deterministic UPCAs

In this subsection we show that deterministic CAs are stronger than deterministic UPCAs. By Section 6.2.5, it follows that deterministic UPCAs are strictly weaker than nondeterministic. For brevity, we show this first in the one-dimensional case, and then indicate how the proof extends to two dimensions.

It is known [4] that there exists a language L not accepted by any deterministic TBA, but accepted by a deterministic TA that uses an amount of tape equal to the square of the length of the input string. Let $L' = \{\sigma b^{|\sigma|^2 - |\sigma|} \mid \sigma \in L\}$, where b is a special "blank" symbol. Thus L' is accepted by a DTBA, since it can simulate the DTA that has $|\sigma|^2$ tape, and can also verify that the proper number of b s is present.

Suppose L' were accepted by a DUTCA C . We shall show in the following paragraphs that, because most of its input consists of bs , the action of C on an input string in L' of length $|\sigma|^2$ can be simulated by a DTBA using only $|\sigma|$ tape. In what follows, let 2^n be the smallest power of 2 that exceeds $|\sigma| - 1$.

In the initial configuration of C , the input string σ occupies (at most) the first 2^n cells of the base, and the remaining $2^{2n} - 2^n$ cells are bs . Thus many of the cells at higher levels have identical initial subtree configurations, and hence identical state sequences, since C is deterministic. In particular, there are at most $2^n + 1$ distinct level 0 cells, namely 2^n non- b cells and one b cell. At level k , $0 \leq k \leq n$, there are 2^{2n-k} cells of which 2^{n-k} have non- bs in their bases, while the others have identical all- b bases; thus there are at most $2^{n-k} + 1$ distinct cells at level k . At each of levels $n + 1$ through $2n$ there is exactly one cell with non- b input; hence there are only two distinct state sequences for cells at any one of these levels. Summing, we find that there are at most $2^{n+1} + 3n - 1$ distinct cell types in C . Thus, while C has $O(2^{2n})$ cells, only $O(2^n)$ of them, at most, have distinct state sequences.

Based on this, we show how a DTBA A having only tape of length 2^n can simulate C . As in Section 6.2.4, we can order the $2^{n+1} - 1$ cells with non- b bases breadth first and map them into the 2^n tape positions of A , two per cell. The remainder of C 's cells to be saved are of two types: $2n$ cells with all- b input, one each from levels $0, \dots, 2n - 1$; and n cells with non- b input, one at each of levels $n + 1, \dots, 2n$. Let us map the $2n$ all- b cells into the leftmost $2n$ positions of A 's tape, and let us store the n non- b cells in positions $n + 1, \dots, 2n$ of the tape.

A can easily mark off blocks of tape of size n by counting its input tape positions. Hence it can easily find those sections of tape that contain the extra $3n$ cells' states. At any time step, the states of both sons of an all- b cell are the same and are stored one position to the left of the given cell's state. Similarly, the states of the sons of non- b cells are located immediately to the left of the given cell's state. (There is one exception: the cell whose state is in position $n + 1$ has as its left son the root of the subtree containing the input string σ , and its state is located in position 1.) Thus A can simulate a transition of C by sequentially accessing the states of the sons for each of C 's $2^{n+1} + 3n - 1$ cells, as described above and in Section 6.2.4. In particular, the state of the root cell of C is stored at position $2n$, which A can find at the completion of each simulation step. If this cell is in an accepting state, A accepts.

Since A can simulate C , if L' were accepted by C , it could also be accepted by A . But this evidently implies that L is accepted by a DTBA—a contradiction. Thus L' is not accepted by C , and since it

is accepted by a DTBA, we see that DTBAs, or equivalently DCAs, are strictly stronger than DUTCAs.

It follows that nondeterministic UTCAs (which, by Section 6.2.5, can simulate CAs) are strictly stronger than DUTCAs.

The argument in two dimensions is analogous. We can show that a DTBA with a $2^n \times 2^n$ input tape can simulate a DUPCA with base $2^{2n} \times 2^{2n}$, making use of the fact that the latter's cells have at most $2^{2n+2}/3 + 3n$ distinct state sequences. The desired result now follows from the fact [4] that having $|\sigma|^4$ tape gives more power than having only $|\sigma|^2$.

6.3 LOCAL PROPERTY DETECTION AND COUNTING

In this section we describe some basic (U)PCA recognition algorithms. These include detecting the presence or absence of a specified local pattern in the input, and counting the occurrences of such local patterns. These algorithms require only $O(\log \text{diameter})$ time.

In describing the algorithms, we shall not use the terminology of states and transition functions; rather, we shall simply say that a cell transmits or outputs certain information to its father, or receives certain information from its sons. We assume as before that the size of the pyramid base is $2^n \times 2^n$, so that the height is n . The layers will be numbered $0, 1, \dots, n$, starting from the base.

6.3.1 Local Property Detection

We first consider the trivial problem of detecting the presence (or absence) of a specific value z in the input array. This can be done as follows. Each base cell compares its input array value to z , and if they match, the cell transmits a "match bit" to its father. In addition, every base cell transmits a "flag bit" to its father, to indicate that the matching has been performed. If a father receives a match bit from any of its sons, it transmits a match bit to its own father; and in any case, it transmits the flag bit that it has received. When the apex cell receives the flag bit, then if it also receives a match bit, it outputs a success signal; if not, it outputs a failure signal. Note that if we did not use the flag bit, the apex cell could not detect the absence of z s, since it would not know whether or

not the match information has reached it. The time required for this algorithm is just $(\log \text{diameter})$ steps.

Detecting arbitrary local patterns, rather than specific values, is best done by allowing sidewise transmission of information in the base. Each base cell can then decide whether it is the "head cell" (e.g., the leftmost of the uppermost cells) of the given pattern, by accepting appropriate information from its neighboring cells. Since the patterns are of bounded size, the amount of information needed is bounded and presents no special transmission problems. When a cell discovers that it is the head cell of a pattern, it can take on a special value, and the presence or absence of these values can then be detected as just described.

For a UPCA, detecting arbitrary local patterns is harder. The problem is that if the pattern is in a bad position, say exactly in the middle of the base, the cells at a bounded height above the base cannot see all of it and hence cannot detect it. This problem would not arise if the cells on each layer looked at overlapping neighborhoods on the layer below; but this would require a linearly, rather than exponentially, tapering pyramid, the height of which would be proportional to the base diameter rather than to its logarithm, so that $O(\log \text{diameter})$ -time operation would no longer be possible.

In one dimension, arbitrary local patterns can be detected by a UTCA using the following brute-force approach. Suppose that the desired pattern has length k . Each cell at height $\log k$ receives a copy of the entire portion B of the base below it, and decides which (if any) of the following conditions holds for that portion.

- (a) The entire pattern is present in B . We denote this condition by p (for "pattern match").
- (b) An initial segment of the pattern of length i is present at the right end of B , $1 \leq i < k$. This condition will be denoted by p_i .
- (c) A terminal segment of the pattern of length j is present at the left end of B , $1 \leq j < k$. We denote this condition by p'_j .

If none of these conditions holds, we denote that fact by q . Note that more than one of them can hold simultaneously. The set of those that do hold can be represented by a bit pattern of length $2k$. This bit pattern is transmitted to the cells at the next layer above, and these cells now operate as follows:

- (a) If p is received from either son, transmit p .
- (b) If p_i is received from the left son, and p'_{k-i} from the right son, $1 \leq i < k$, transmit p .
- (c) Otherwise, transmit the left son's p'_j information and the right son's p_i information (if none, transmit q).

Thus these cells too transmit only a bounded amount of information. The process is now repeated. It is clear that the apex cell will transmit p iff the desired pattern is present, and that the entire process takes $\log$ diameter time steps.

In the remainder of this section we describe a UPCA algorithm for detecting arbitrary $k \times k$ patterns in $\leq O(\text{diameter})$ time. The one-dimensional algorithm just given does not generalize to two dimensions, since pieces of the pattern can now occur in a large number of different positions around the borders of a given block B of the base, and these must be matched up properly with pieces on the borders of neighboring blocks.

Let c_i denote the leftmost $k - 1$ states in the i th row of the block B of the base below cell c . Similarly, let c'_i denote the rightmost $k - 1$ states in the i th row of B . We design the UPCA so that the cells in layer k count modulo 2^k and output alternating sequences of 0s and 1s of length $2^k - 1$. (See Section 6.2.6.)

Suppose cell c at height $\kappa = \lceil \log k \rceil$ repeatedly receives a copy of its block B and

- (a) decides whether or not the pattern is entirely contained in B ;
- (b) starting at time κ , outputs c_i and then c'_i , where $i = t - \kappa + 1 \pmod{2^\kappa}$.

Then a cell c in layer $\kappa + 1$ can use its modulo $2^{\kappa+1}$ counter and the outputs of its four sons to compute its own c_i and c'_i values (modulo $2^{\kappa+1}$), since

$$c_i = \begin{cases} [c(\text{NW})]_i, & 1 \leq i \leq 2^\kappa \\ [c(\text{SW})]_{i-2^\kappa}, & 2^\kappa < i \leq 2^{\kappa+1} \end{cases}$$

and

$$c'_i = \begin{cases} [c(\text{NE})]'_i, & 1 \leq i \leq 2^\kappa \\ [c(\text{SE})]'_{i-2^\kappa}, & 2^\kappa < i \leq 2^{\kappa+1} \end{cases}$$

where $c(\text{NW})$ is c 's northwest son, etc.

Simultaneously, c can decide whether or not the pattern appears anywhere within its block as follows. Since we need only check $k \times k$ squares that are not entirely contained in one of c 's sons' blocks, we need only reconstruct the $2^{\kappa+1} \times (k-1)$ vertical band centered on the middle column of c 's block in order to match across the vertical crack. In fact, only a $k \times 2(k-1)$ running window of the band needs to be stored by cell c at any time. Using its modulo $2^{\kappa+1}$ counter and its four sons' outputs, c can scan its vertical band, since the i th row of the band is computable at time $\kappa + i$ (modulo $2^{\kappa+1}$) by

$$[c(\text{NW})]_i' [c(\text{NE})]_i \quad \text{if } 1 \leq i \leq 2^\kappa$$

or

$$[c(\text{SW})]_{i-2^\kappa}' [c(\text{SE})]_{i-2^\kappa} \quad \text{if } 2^\kappa < i \leq 2^{\kappa+1}$$

After time step $k + \kappa$, c can begin checking whether the entire pattern is present anywhere in its $k \times 2(k-1)$ window.

In the same way, c can simultaneously scan the $2(k-1) \times 2^{\kappa+1}$ horizontal band centered on the middle row of its block in order to match across the horizontal crack. (This means that the cells at height κ will also have to output the top and bottom $k-1$ symbols in the i th column, $1 \leq i \leq 2^\kappa$, modulo 2^κ .) If at any time the pattern is found in either band, a success signal is immediately sent up to the apex cell.

Similarly, each cell in layer $m > \kappa + 1$ behaves like the cells in layer $\kappa + 1$, sequentially scanning their vertical and horizontal bands of length 2^m , starting at time m . Hence, if the pattern is detectable in layer m , then the apex cell will know this at a time no later than $m + 2^m + (n - m) = 2^m + n$. In the worst case, the pattern is detectable only by the apex cell and time $2^n + n$ is required.

If we assume that the $k \times k$ pattern occurs at least once and is equally likely to appear anywhere in the $2^n \times 2^n$ base, and $k \ll n$, the $O(\text{diameter})$ worst-case bound reduces to $O(\log \text{diameter})$ time in the average case. That is, there are $4^n/4^m$ cells in layer m , and each checks $2(k-1)2^m - 3(k-1)^2 k \times k$ blocks for the presence of the pattern. Thus in layer m ($m > \log k$) a total of $2^{2n}[2(k-1) \cdot 2^m - 3(k-1)^2/2^{2m}] k \times k$ blocks are inspected. It is easily verified that after processing layer $(\kappa + 2)$, over half of the 2^{2n} blocks have already been checked. Since the apex cell will know whether the pattern is detected in any of these layers by time step $4k + n$, the $O(\log \text{diameter})$ average time bound results.

6.3.2 Local Property Counting

We next show that a UTCA can count, in $2(\log \text{diameter})$ time steps, the number of occurrences of some particular symbol z in its input. Specifically, the apex cell will output, at time steps n to $2n$, the $n + 1$ bits in the binary representation of this number (which lies between 0 and 2^n), least significant bit first. The generalization to BPAs is straightforward.

The counting algorithm is defined inductively; we shall prove that at time steps m to $2m$, each cell in the m th layer above the base ($m = 0, 1, \dots, n$) outputs the number of z s in the portion of the base below it, least significant bit first. To initialize this, each base cell ($m = 0$) outputs, at the end of time step 0, a 1 if its input value is z , and a 0 otherwise. Now consider a cell c in layer $(m + 1)$. By induction hypothesis, it receives from its sons at the ends of time steps $m, \dots, 2m$ the numbers of z s in their base segments. Cell c can now function as a serial adder: At time step $m + 1$, it sums the two least significant bits of its sons' numbers and stores the sum and carry bits. At the end of step $m + 1$, it outputs the sum bit. At step $m + 2$, it receives the next least significant bits from its sons, adds them to the carry bit, stores the resultant carry bit, and outputs the sum bit. This process is repeated at steps $m + 3, \dots, 2m$. At the end of step $2m + 1$, c outputs the sum bit resulting from the last addition step, and at step $2m + 2$, it outputs the carry bit resulting from that step. Clearly, c 's outputs at steps $m + 1, \dots, 2m + 2 = 2(m + 1)$ are just the bits of the sum of its sons' outputs, i.e., the number of z 's in c 's base segment.

Note that the base cells must output their bits only at time step 0 if this algorithm is to work. To ensure this, the input data can be erased; or a flag bit can be set that inhibits further output. Once we know that the base cells produce no output after time step 0, it is easy to see that the cells in layer k produce no output after the end of time step $2k$. The total time required by the algorithm is $2n$, i.e., twice $(\log \text{diameter})$ time.

We next show that a UTCA can count the number of occurrences of an arbitrary local property in $2(\log \text{diameter})$ time steps. That is, the apex cell will output, at time steps n through $2n$, the $n + 1$ bits in the binary representation of this number, least significant bit first.

Suppose that the desired pattern has length $k = 2^*$. Each cell in layer κ receives, at time step κ , a copy of the block B of the base below it and outputs, at the end of step κ ,

- (a) 1 if B matches the pattern, and 0 otherwise;
- (b) B_k , the leftmost segment of B of length $k - 1$; and
- (c) B'_k , the rightmost segment of B of length k .

Now consider a cell c in layer $(m + 1)$. By the induction hypothesis, it receives from its sons at the ends of time steps $m, \dots, 2m$ the number of instances of the given pattern and the values of B_k and B'_k in their base segments. Let B_{k1}, B_{k2} be the values for c 's left son, and B'_{k1}, B'_{k2} the values for c 's right son. Then $B_k = B_{k1}$ and $B'_k = B'_{k2}$ can be output by c at the end of time step $m + 1$. In addition, $B_{k2}B'_{k1}$ is the segment of length $2(k - 1)$ centered on the middle of c 's base. At most $k - 1$ occurrences of the pattern can extend across c 's midbase cell, and all the information for checking these $k - 1$ positions is contained in the string $B_{2k}B_{k1}$. Hence at time step $m + 1$, c can compute and store the number of times h that the pattern is found crossing c 's midpoint and then add the least significant bit of h to the sum of the least significant bits of its sons' counts, which has just been computed. At step $m + 2$, c computes its next least significant bit by adding the next bits from its sons and the next bit of h . This process is repeated at steps $m + 3, \dots, m + \kappa$ since h is a κ -bit number. For the remaining steps $m + \kappa + 1, \dots, 2m + 2$, c just sums the remaining bits from its sons. Clearly, c 's outputs at steps $m + 1, \dots, 2m + 2 = 2(m + 1)$ are just the bits of the sum of its sons' counts and its own count, i.e., the number of times the given pattern occurs in c 's base block. The total number of time steps required by the algorithm is $2n$.

This algorithm does not generalize to two dimensions. However, as pointed out at the end of Section 6.3.1, by allowing sidewise transmission of information in the base, we can reduce the problem of counting local patterns to that of counting occurrences of a particular symbol; this can be done in $O(\log \text{diameter})$ time by an immediate generalization of the one-dimensional algorithm in which we count to base 4 rather than to base 2.

6.4 COMPARISONS WITH OTHER TYPES OF ACCEPTORS

In this section we compare UPCAs (or UTCAs) with three other computational models—finite-state acceptors (FSAs), frontier-to-root tree acceptors (FRAs), and diameter-limited perceptrons (DLPs). We show that

- (a) UTCAs are stronger than FSAs and can simulate them in log real time;
- (b) UPCAs are stronger than FRAs; and
- (c) UTCAs are stronger than DLPs, if we assume certain bounds on the precision of the coefficients and threshold used by the DLPs.

6.4.1 UPCAs and FSAs

We first show that UTCAs can simulate FSAs. It suffices to show this for an FSA, call it A , that is one-way and deterministic, since such A s can simulate two-way or nondeterministic FSAs. Strings whose lengths are not powers of 2 can be padded with $\#$ s at their right ends.

Let the state set of A be $Q = \{q_1, \dots, q_m\}$. Each base cell, say having input value v , constructs the vector $(q'_1, \dots, q'_m)$, where q'_i is the state into which A goes if it reads symbol v while in state q_i , $1 \leq i \leq m$. This vector defines a function from Q into Q . (It is understood that if A reads a $\#$, it can continue to move but does not change state.)

We now show that each cell c can compute a vector of states $(p_1, \dots, p_m)$, where p_i is the state in which A would end if it started in state q_i and scanned the segment of the base lying under c . This is true for the base cells, by the preceding paragraph. Suppose it is true for the sons c' , c'' of c , and let their vectors be $(p'_1, \dots, p'_m)$ and $(p''_1, \dots, p''_m)$, respectively. Then the vector $(p_1, \dots, p_m)$ of c can be computed as a "composition" of the vectors of c' and c'' . In fact, p_i is just $p''_{(p'_i)}$ —i.e., it is the ending state in the c'' vector that corresponds to beginning state p'_i (which in turn is the c' ending state resulting from q_i).

The apex cell's vector $(p_1, \dots, p_m)$ thus specifies the state in which A ends if it scans the entire base, starting in state q_i , $1 \leq i \leq m$. In particular, if A 's initial state gives rise in this way to a final or accepting state, the apex cell can accept the base, and otherwise not. This process requires $n + 1$ time steps, i.e., $O(\log \text{diameter})$ time; it is faster than the FSA itself, since that requires 2^n time steps to scan the base.

UTCAs can also accept many languages that cannot be accepted by FSAs. A simple example is $\{x^N y^N \mid n = 1, 2, \dots; N = 2^{n-1}\}$. The algorithm for accepting this is as follows: If a cell's sons are

both xs , it outputs x ; if both ys , it outputs y ; if x and y (in that order), it outputs z ; otherwise, it outputs w . If the apex cell outputs z , the UTCA accepts; otherwise, it rejects. A similar construction shows that a UTCA can accept languages that are not even context-free, e.g., $\{x^M y^M u^M v^M \mid n = 1, 2, \dots; M = 2^{n-2}\}$. Thus UTCAs are strictly stronger than FSAs.

It is not clear how to generalize this construction to two dimensions. The problem is that A can enter or leave a given segment B of the base at many places (i.e., anywhere along B 's perimeter), so that we can no longer specify A 's behavior relative to B by a vector of bounded length. Thus it is an open question whether a UPCA can simulate a two-dimensional FSA.

We shall now show that a UPCA can simulate a special type of two-dimensional FSA, namely one that can only do a fixed raster (row-by-row) scan of its input. We can assume that the given FSA A is deterministic, since for this type of FSA it is easy to show that nondeterminism gives it no additional power, as in the one-dimensional case. Our proof is based on the fact that each cell in layer k can be made to "count" in such a way that for the first 2^{k-1} steps it outputs 0, for the next 2^{k-1} it outputs 1, and so on, changing from 0 to 1 or 1 to 0 every 2^{k-1} steps.

At time step 1, each base cell in the UPCA constructs a state transition vector of length $|Q|$ based on the input value at the cell. Each nonbase cell c behaves as follows: When c 's counter is in state 0 (1) it constructs the composition of its upper left (lower left) and upper right (lower right) sons' transition vectors. Thus a cell in layer 1 alternately composes the transition vectors of its upper two and lower two sons. A cell in layer 2 composes vectors from its upper sons at the first two time steps, and composes its lower sons' vectors at the next two steps; and so on. As in the case of the one-dimensional FSA simulation at the beginning of this section, it follows that a cell in layer k computes left-to-right row scan transition vectors for the 2^k rows in its base in a top-to-bottom sequence at times $k, k + 1, \dots, k + 2^k - 1$; and this process repeats (modulo 2^k).

At the same time, each cell c in layer k composes these 2^k row scan transition vectors to get a complete raster scan transition vector for c 's base. This is accomplished by having c store a second cumulative transition vector. When c 's counter switches from state 1 to state 0, we must be computing the vector for c 's top row. Thus at this step we initialize the raster scan transition vector with the

first row's vector. At all other times, we just compose the current row r 's vector with the current cumulative vector to obtain a new cumulative vector describing A 's movement over the top r rows of c 's base. In particular, at time $n + 2^n - 1$, the apex cell's raster scan transition vector describes the state in which A ends after scanning the base in raster order when starting in any of its $|Q|$ states. Thus if A 's initial state gives rise to an accepting state in this vector, the UPCA can accept; otherwise it rejects its input.

The time $n + 2^n - 1$ required for this simulation is $O(\text{diameter})$. This is faster than A itself, which requires $O(\text{area})$ time to scan its input. However, a CA can also simulate a raster scan FSA in $O(\text{diameter})$ time.

Similarly, we can simulate other fixed scanning sequences of an FSA, e.g., snakelike or column-by-column scans, without significantly altering the construction given above. The critical knowledge that we have used here, which is not available in more general FSAs, is that the motion of A is fixed in advance and so it is not necessary to keep track of A 's position on the tape.

We now return to the general problem: How does a UPCA compare with an ordinary FSA, which can move to any of its four neighbors during a transition? The problem is that A can enter or leave a given block of base cells at any place along its boundary, and so the description of A 's behavior on a block grows with the size of the block. Consequently, we cannot specify A 's behavior relative to an unbounded size base segment by a state transition vector of bounded length.

We shall see in Section 6.5 that a UPCA can accept the vertically symmetric palindromes (i.e., the arrays each of whose rows is a palindrome). On the other hand, it can be shown, by an argument similar to that used to prove Theorem 4.3.4, that no FSA can accept this language. Thus the sets of languages accepted by UPCAs and FSAs are not the same.

6.4.2 UPCAs and FRAs

A *frontier-to-root tree acceptor* (FRA) can be defined [5] as having a finite-state automaton at each node of the given rooted tree. Initially, the automata at the leaf nodes read their inputs (i.e., the symbols at these nodes, if any) and generate output states. These states are sensed by the fathers of the leaf nodes, who use them (together with their own input symbols) to generate their output

states; and so on. A node does not compute its output until it has sensed the outputs of all its sons. The root node's output indicates whether or not the input tree has been accepted.

Suppose, in particular, that the tree is a complete 4-ary tree of depth n ; this corresponds exactly to the tree structure of a UPCA with base size $2^n \times 2^n$. If we assume that the input symbols are all $\mathfrak{h}$ except at the base, then the operation of the FRA can be simulated by a single "wave" of information passing up the UPCA from base to apex. This simulation requires exactly log diameter time (where computing the FRA's state transitions takes unit time).

On the other hand, there are some recognition tasks that a UPCA can perform that it cannot do in log diameter time; we shall show immediately below that recognition of palindromes is such a task. Evidently, such tasks cannot be done by an FRA, since whatever the FRA does at all must be done in log diameter time. Thus UPCAs are stronger than FRAs.

For simplicity, we treat palindrome recognition in the one-dimensional case, i.e., by a UTCA. We first show that it cannot be done in less than $O(\text{diameter})$ time. Indeed, acceptance by the apex cell depends only on the sequence of states of each of its sons, and there are $|Q|^t$ such sequences of length t . Now each half of the base has $|Q|^{(2^{n-1})}$ possible configurations. Thus if $t < 2^{n-1}$, two of these configurations, say B_1 and B_2 , must give rise to the same state sequence. This means that if the apex accepts $B_1 B_1^R$ in t time steps, where $t < 2^{n-1}$, it must also accept $B_2 B_1^R$, which is not a palindrome. Thus no UTCA can accept palindromes in less than $2^{n-1} = \text{half-diameter}$ time.

We shall now show that there does exist a UTCA that accepts the palindromes in $O(\text{diameter})$ time. As a preliminary to this, we show how the cells in the k th layer of a UTCA can be made to count modulo 2^k —i.e., to change state every 2^k time steps. We do this by outputting 1s from the base cells, repeatedly. A nonbase cell outputs 1s every other time that it receives 1s from its sons. It is not hard to see that this implies that the cells in layer k output 1s at time steps that differ by 2^k . We can regard such a cell as initially being in state 0; changing to state 1 when it receives 1s from its sons; changing back to state 0, and outputting 1, the next time it receives 1s from its sons; and so on.

Now suppose that whenever a cell is in state 0, it copies its left son's value, and when it is in state 1, it copies its right son's value. Thus a cell just above the base (in layer 1) copies its left and right

sons' values alternately; a cell in layer 2 copies the left and right sons of its left son, then the left and right sons of its right son, repeatedly; and so on. By induction, it follows that a son in layer k copies the values of the cells below it in the base, in left-to-right sequence, at times $k, k + 1, \dots, k + 2^k - 1$; and this process then repeats (modulo 2^k). In particular, the apex cell copies the entire base, in sequence, starting at time n .

Using a mirror image of this process, we can also get the apex cell to scan the base in right-to-left sequence. It can thus compare the two scans, cell by cell, and accept if their first halves match—i.e., if no mismatch has been found by the time the apex cell enters state 1. The total time required for this is $n + 2^{n-1}$ (the scans start at time n , when the values begin to reach the apex). This result is nearly optimal, since we saw above that 2^{n-1} is a lower bound. Two-dimensional palindrome recognition will be treated in Section 6.5.2.

6.4.3 (U)PCAs and DLPs

A *diameter-limited perceptron* (DLP) can be defined as operating in the following way [6]:

- (a) It detects the presence or absence of a given local property in a neighborhood of each point of its input; it is assumed that the neighborhoods are of some bounded diameter d .
- (b) It computes a sum of the form $\sum a_i p_i$ over the points of its input, where $p_i = 1$ or 0 , depending on whether or not the property is satisfied at the given point.
- (c) It compares the sum to a threshold t , and accepts the input iff the sum is at least t .

We have already indicated (see Section 6.2) that a CA (or a UTCA) can detect the presence or absence of a given local property in a fixed-size neighborhood of each of its base cells. Let us therefore assume that this has already been done, and that the cell has stored the value 1 if the property holds, and 0 if it does not (or if the property was not to be computed at that point). We assume that the number of properties to be computed in this way for each base cell is bounded, so that the cell can store a bit pattern of bounded size that indicates which of them are applicable and valid.

Let us further assume that the coefficients a_i in step (b) have bounded precision, so that they too can be stored in the base cells. Thus each

base cell can compute its portion of the sum $\sum a_i p_i$, since this sum includes only a bounded number of terms, each of bounded size.

Finally, we assume that the threshold t in step (c) has precision that grows at most linearly with the input size. This implies that t can be stored in the base of a UPCA as a unary number by storing 1s in a subset of the base cells.

Given all these assumptions, a UPCA can simulate the DLP by summing the 1s to compute t , and also summing the a_i s to compute $\sum a_i p_i$. The algorithm for summing the a_i s, which are numbers of bounded size, is a straightforward extension of the counting algorithm of Section 6.2; e.g., if the a_i s are in the range $[0, r]$, the cells can sum them by acting as base $(r + 1)$ adders. The digits of the sum will thus be output by the apex cell at times $n, \dots, 2n$. The digits of t will also be output at these times, and if we sum them too in base $(r + 1)$, we can compare digits serially to determine whether $\sum a_i p_i \geq t$.

In summary, we have shown that (U)PCAs can simulate a restricted class of diameter-limited perceptrons—namely, those for which the above three italicized assumptions hold. The simulation requires time $2n = O(\log \text{diameter})$. The DLP computes $\sum a_i p_i$ and compares it with t “instantaneously,” but this requires it to sum a very large number of inputs simultaneously, which is physically unrealistic.

UPCAs can also do many things that DLPs cannot do. As a simple example, it is known [6] that a DLP cannot determine the parity of the number of 1s (say) in its input. On the other hand, a UTCA can do this very simply, as we saw in Section 6.2.1.

6.5 OTHER UPCA LANGUAGES

6.5.1 Some Other UTCA and UPCA Algorithms

UTCAs can also recognize a variety of other input languages; the following are a few examples:

(a) *Connectedness*: In one dimension, the set of strings of x s and y s in which the x s are connected have at most one run of x s. Thus a UTCA can recognize this set by simulating an FSA, which checks that there is no xy pattern in the input to the left of a yx pattern. More generally, a UTCA can count connected components of x s by counting the number of (say) yx patterns and adding 1 to the

sum if there are xs at the left end of the input. It is an open question whether a UPCA can recognize connectedness of the set of xs in its two-dimensional input, or whether it can count connected components of xs .

(b) *Equality and majority*: A UPCA can easily check whether the number of xs in its input is equal to, or greater than, the number of ys , by simultaneously counting both of them and comparing the counts (see Section 6.3.2). Less trivially, a UTCA can check whether its input contains exactly two connected runs of xs both of which have the same length. It verifies that there are exactly two runs by simulating an FSA. At the same time, it counts the xs to the left of the leftmost xy pattern, by a straightforward extension of the counting algorithm to inhibit counting xs that lie to the right of an xy . (Specifically: besides transmitting numbers of xs , each cell also outputs a special signal z that indicates whether the part of the base below it contains the pattern xy . If a cell receives z from its left son, it copies the left son's count of xs and ignores the right son's count; otherwise, it adds the two counts. It is not hard to see that this results in the cell's counting the number of xs to the left of the leftmost xy under it in the base.) Concurrently, the UTCA counts the xs to the right of the rightmost yx pattern and compares these two counts, which allows it to determine whether the runs have equal length (or, for that matter, whether a particular one of them has greater length than the other).

(c) *Closure properties—Union, intersection, and reversal*: Trivially, any finite union (or intersection) of UPCA languages is a UPCA language, since we can construct a UPCA that simulates the acceptors of these languages simultaneously, and accepts iff one (or all) of them accept(s). The reversal of a UTCA language is a UTCA language, since it is accepted by a mirror-image of the given UTCA. We have seen in Section 6.2.5 that the complement of a UPCA language is also a UPCA language. A concatenation of two UTCA languages is a UTCA language, provided we concatenate only strings of the same length, since the left and right subtrees of the UTCA can then accept the two halves of the base, and the root (apex cell) can accept if its two sons have accepted. It is less easy to see whether concatenations involving strings of different lengths, or numbers of strings that are not powers of 2, can be accepted. This problem is closely related to that of whether UTCA languages are closed under translation—i.e., if L is a UTCA language, is there a UTCA that accepts $\#^k \sigma \#^{2^n - k - |\sigma|}$, for all $0 \leq k \leq$

$2^n - |\sigma|$, iff $\sigma \in L$? Shift invariance will be discussed further in Section 6.5.4.

(d) *Parenthesis strings*: We conclude this section by exhibiting a UTCA that accepts the set of well-formed parenthesis strings in $O(\log^2 \text{diameter})$ time.

Note first that, given any parenthesis string, we can reduce it to the form $)^i($ ^{j} , where $i \geq 0$ and $j \geq 0$, by repeatedly canceling adjacent pairs $()$. Given two strings whose reduced forms are $)^i($ ^{j} and $)^h($ ^{k} , the reduced form of their concatenation is

$$)^{i+h-j}(^k, \quad \text{if } j \leq h; \quad)^i(^{k+j-h}, \quad \text{if } j \geq h$$

A string is well formed iff its reduced form is the null string ($i = j = 0$). Thus a UTCA can recognize well-formed parenthesis strings if each cell can compute the reduced form of the parenthesis string in the part of the base below it; the UTCA accepts iff the apex cell's form is null.

Evidently, a cell just above the base can compute its reduced form; in fact, we have $i = j = 0$ if the base contains $()$; $i = 1, j = 1$ if the base is $)()$; $i = 2, j = 0$ if it is $))$; and $i = 0, j = 2$ if it is $(($. Suppose that each cell transmits its i and j counts to its father, as in the standard counting algorithm, using two separate "channels" to count i s and j s simultaneously. In order for the father to compute its own count, given the i, j and h, k counts of its sons, the father must first determine whether $j \leq h$ or $j \geq h$. It can do this as it receives the counts for the first time; it must then receive them a second time in order to do the necessary addition and subtraction ($i + h - j$ or $k + j - h$).

The timing of this process is as follows: Each cell receives the inputs from its sons and compares them to determine whether $j \leq h$. For a cell in layer m , these inputs have m bits, since the numbers are between 0 and 2^{m-1} , inclusive. The next time it receives them, it computes its own $(m + 1)$ -bit outputs. It then waits for the inputs to begin again, and computes its outputs again; this process is repeated. Each cell transmits a special flag bit whenever it begins outputting, so that its father can tell when its inputs begin.

It is not hard to see that a cell in layer m first begins to receive its m -bit inputs at time $m(m + 1)/2$. It may not begin to receive them again immediately, because of the possible waiting times mentioned above; but the delay, if any, is at most m time steps. Thus it can compute and transmit its $(m + 1)$ -bit outputs at times sepa-

rated by not more than $2m + 1$. In particular, the apex cell has computed its output by time at most $\frac{1}{2}n(n + 1) + 2n + 1 = O(n^2)$, and if this output is zero, it accepts the input string.

6.5.2 Two-Dimensional Palindromes

In two dimensions, the palindromes may be defined in alternative ways, depending on the axis of symmetry chosen. In this section we present a UPCA algorithm for recognizing vertically symmetric palindromes, i.e., arrays that are symmetrical about their central column.

Before presenting the algorithm, we first show how a cell in layer k of a BPA can be made to count modulo ab^{ck+d} for arbitrary natural numbers a, b, c, d . Since $ab^{ck+d} = a(b^d)(b^c)^k = ef^k$, we need only show how a cell in layer k can have a modulo ef^k counter for constants e and f . We have shown previously how such a counter is built when $e = 1$ and $f = 2$. In the general case, base cells are defined to output 1s every e time steps, and nonbase cells output 1s every f th time that their sons' counters output 1s. Thus layer 0 (base) cells are modulo $ef^0 = e$ counters. If cells in layer k are counting modulo ef^k , then it is not hard to see that layer $k + 1$ cells' f -step delay makes these cells modulo $f(ef^k) = ef^{k+1}$ counters.

Recognizing palindromes with a UPCA, as in the one-dimensional case, is based on performing a fixed sequential scan of the base. This is implemented by having cells in the k th layer count modulo 4^k as described above. The counters can be easily modified so as to output 0 for the first 4^{k-1} steps, 1 for the next 4^{k-1} steps, 2 for the next, and 3 for the last 4^{k-1} steps. If a father's counter is 0, it copies its NW son's value; if it is 1, it copies its NE son's value; if it is 2, it copies its SW son's value; and if it is 3, it copies its SE son's value. Thus a cell in layer 1 copies its sons' values at successive steps in the following order: $\begin{smallmatrix} 0 & 1 \\ 2 & 3 \end{smallmatrix}$. A cell in layer 2 copies

the values of its NW son for four steps, then of its NE son for four steps, etc., so that it repeatedly copies the values of its base cells in the following order:

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

By induction, it follows that a cell in layer k copies the values of its base cells at times $k, k + 1, \dots, k + 4^k - 1$; and this process repeats modulo 4^k .

We can obtain a mirror image of the above scan by simply changing the order of scanning the sons to $\begin{smallmatrix} 1 & 0 \\ 3 & 2 \end{smallmatrix}$. It is readily seen that running both scans simultaneously means the apex cell copies all pairs of symbols in base cells that are vertically symmetric about the central column of the base. If all of these pairs of symbols match, then the UPCA can accept its input. The total time required is $n + 4^n - 1$. This is nearly optimal, since it can be shown that the lower bound for UPCA recognition of palindromes is $\frac{1}{2}(4^n)$, by remarks similar to those in Section 6.4.2. Other types of two-dimensional palindromes, for example, using horizontal or central symmetry, can be handled by essentially the same method. A related question that is still open is whether the apex cell of a UPCA can copy the symbols in its base in raster order.

6.5.3 Rectangles and Squares

Let the language L_{RECT} be the set of input images that contain a single solid rectangle of xs (with sides parallel to the sides of the base) on a background of ys . We saw in Proposition 4.2.1 how an FSA on a two-dimensional tape can recognize L_{RECT} by checking each boundary point's local slope. Minsky and Papert [6] define a diameter-limited perceptron to recognize L_{RECT} by counting the local property "corner." That is, input image $\Sigma \in L_{\text{RECT}}$ iff Σ is an array of xs and ys and

$$\begin{aligned} & \left(\text{the number of occurrences of pattern } \begin{smallmatrix} x & y \\ y & \end{smallmatrix} \right) \\ & + \left(\text{the number of occurrences of pattern } \begin{smallmatrix} y & x \\ x & \end{smallmatrix} \right) \leq 4 \end{aligned}$$

where the patterns also include all 90° rotations of each.

A UPCA is not well suited either for simulating the moves of an FSA (see Section 6.4.1) or for counting local properties (Section 6.3.2). [If we allowed the UPCA to have sidewise transmission in its base, then the Minsky and Papert method could be implemented on a UPCA and L_{RECT} would require $O(\log \text{diameter})$ time to be recognized.] We now show how a UPCA can recognize L_{RECT}

by a third method, which compares consecutive row cross sections of the base.

Each row of an input image contained in L_{RECT} must be of the form $y^h x^i y^j$. We shall now prove that each cell c in layer k can output at time steps k to $2k$ the $k + 1$ bits in the binary representations of h, i , and j for the top row and h', i' , and j' for the bottom row in c 's base. Associated with the first (least significant) bit of each number will be a flag indicating whether or not the value is zero.

Clearly the cells in layer 1 can compute this information. Now consider a cell c in layer $k + 1$. By the induction hypothesis, it receives from its sons at the ends of time steps $k, \dots, 2k$ the values of h, i, j, h', i', j' in their base segments. Let the subscripts 1, 2, 3, 4 denote the northwest, northeast, southwest, and southeast sons of c , respectively. Then c 's base has the following form:

$$\begin{array}{ll} y^{h_1} x^{i_1} y^{j_1} & y^{h_2} x^{i_2} y^{j_2} & \text{row } 1 \\ y^{h_1} x^{i_1} y^{j_1} & y^{h_2} x^{i_2} y^{j_2} & \text{row } 2^k \\ y^{h_3} x^{i_3} y^{j_3} & y^{h_4} x^{i_4} y^{j_4} & \text{row } 2^k + 1 \\ y^{h_3} x^{i_3} y^{j_3} & y^{h_4} x^{i_4} y^{j_4} & \text{row } 2^{k+1} \end{array}$$

so that c computes its own h, i , and j values as follows:

- (1) if $i_1 = 0$ and $i_2 = 0$, then $h = h_1 + h_2, i = 0$, and $j = 0$;
- (2) if $i_1 \neq 0$ and $i_2 = 0$, then $h = h_1, i = i_1$, and $j = j_1 + h_2$;
- (3) if $i_1 = 0$ and $i_2 \neq 0$, then $h = h_1 + h_2, i = i_2$, and $j = j_2$;
- and
- (4) if $i_1 \neq 0, i_2 \neq 0, j_1 = 0$, and $h_2 = 0$, then $h = h_1, i = i_1 + i_2$, and $j = j_2$.

If none of these conditions holds, a failure signal is immediately propagated up to the root. The addition is performed as in the UTCA counting algorithm (Section 6.3.2); c functions as a bit serial adder. Similarly, c computes h', i' , and j' from $h'_3, i'_3, j'_3, h'_4, i'_4, j'_4$.

Simultaneously, c checks the two middle rows in its base, as specified by

$$\begin{array}{l} y^{h_1} x^{i_1} y^{j_1} y^{h_2} x^{i_2} y^{j_2} \\ y^{h_3} x^{i_3} y^{j_3} y^{h_4} x^{i_4} y^{j_4} \end{array}$$

to see whether they match, are of the form $y^h x^i y^j$, and are consistent with the top and bottom rows. Let these two middle rows be of the form $y^h x^i y^j$ and $y^{h'} x^{i'} y^{j'}$, respectively. If $i \neq 0$ and $i' \neq 0$, we must be considering two rows intersecting a potential rectangle;

and only if $h = h'$, $i = i'$, and $j = j'$ can both rows be part of a legal rectangle. If $i = i' = 0$, both rows are scanning only background; if only one of i and i' is nonzero, we are seeing either the top or bottom of the rectangle. In these cases, there is no matching to be performed; only a consistency check that prevents multiple rectangles from being accepted. Details will not be given.

By transitivity and the induction hypothesis, we now know that all nonzero rows in c 's base are connected and have the same description, since every consecutive pair of rows within c 's base has been matched. If a cell ever detects a completed rectangle, a success signal is propagated to the apex cell, which, if it receives exactly one such signal, accepts its input. The total time required by the algorithm is $2n = O(\log \text{diameter})$.

We now indicate how a UPCA can detect the fact that the rectangle of x s in its base is a square. Define L_{sq} to be the set of inputs containing a single solid square of x s on a background of y s. L_{sq} can be recognized by an FSA, which first checks that the input contains a single rectangle and then scans from this rectangle's upper left corner at 45° toward its opposite corner. If the opposite corner is reached, the rectangle is a square; otherwise it is not. L_{sq} cannot be recognized by a diameter-limited perception that measures local properties. However, there does exist an order-limited perceptron for recognizing squares that works by stratification—a process of sequentially enumerating and testing all possible starting positions and side lengths for a square in the image [6]. Again, neither of these methods is adaptable for use on a BPA.

A BPA can recognize L_{sq} in $O(\log \text{diameter})$ time by a modification of its own rectangle algorithm. Each cell c computes cross sections of its base segment's boundary rows and columns. In addition, if c 's base is nonzero, c computes a row and column cross section of the object contained in its base. This is done as follows: If c 's top and bottom rows have just been computed to be all zeros and c 's middle two rows are nonzero, then c 's base bounds the height of the object. If the object is a rectangle, c outputs one of its middle rows, which is a row cross section of the rectangle. All cells above c just copy this cross section while computing their own boundary cross sections. Similarly, cells copy a column cross section if a rectangle's width is bounded by their base segment.

When a cell detects a completed rectangle, it simultaneously compares the x counts in the horizontal and vertical cross sections through the rectangle. If they are equal, a success signal is transmitted to the apex cell, otherwise a reject signal is sent. The apex

cell accepts its input image if it receives exactly one accept signal and no reject signals at time step $2n$.

Counting the number of rectangles in an input image by a UPCA involves the same problem encountered in local property counting (Section 6.3.2). That is, there may be an unbounded number of rectangles that are detectable only by the apex cell.

The general technique of comparing consecutive row cross sections used in this subsection is adaptable for recognizing other fixed-orientation polygons whose boundary slopes are locally testable. For example, straight line segments with slopes that are multiples of 45° (with respect to the bottom edge of the image) are digitally realized as

$$\begin{array}{cccc}
 & & x & x & x \\
 x & x & x & & & & \\
 & x & & x & & x & \\
 & & x & & x & & x \\
 0^\circ & 45^\circ & 90^\circ & 135^\circ
 \end{array}$$

Therefore, a UPCA can recognize 45° right triangles with one side parallel to a side of the base, or diamonds (squares rotated 45°), by comparing triples of consecutive rows. (Computing the description of the top and bottom two rows of each $2^k \times 2^k$ block is sufficient for this test.) Similarly, straight lines at other fixed orientations relative to the base are characterized by periodic strings of x s of bounded length. A UPCA can be constructed to detect these slopes by examining a bounded number of cross sections above and below each input row.

It is an open question whether a UPCA can accept the set of arrays of x s and y s whose x s form a convex subset. However, several necessary conditions for a set to be convex can be tested by a UPCA. A set is row convex if it is connected and each row contains at most one connected component. This is a property that can be accepted by a UPCA, since it only requires verifying that there is at most one run of x s on each row, and that the runs of x s on any two adjacent rows overlap. Similarly, a UPCA can recognize column or diagonal convexity.

6.5.4 Size and Position

By definition, PCAs have bases whose dimensions are powers of 2, so that their inputs must always be of certain sizes. However,

we can use them to recognize inputs of arbitrary size by padding the input out (to the next higher power of 2) with $\#$ s. It is clear that this padding need not affect a (U)PCA's capability to count local events, to simulate an FSA (in one dimension), to perform linear threshold operations, or to recognize properties such as parity, equality, or majority on the non- $\#$ part of its input.

In general, if there exists a PCA C that accepts an input language L in some standard position (relative to the borders of the base), then we can construct a PCA that accepts L in any position, by simply shifting the input to the standard position and then simulating C . This assumes that the standard position is computable by the PCA.

The situation is more complicated as regards UPCAs, since they cannot shift their input. For languages that are inherently shift-invariant, such as parity, equality, majority, parenthesis strings, etc., there is no problem; but it is harder to see how a UTCA could accept, say, padded palindromes.

The UPCA is defined so that information can be transmitted only up the pyramid. If the cells in the base do not have any positional information in the initial configuration, then they clearly can never know their positions in the base. By induction, no cell higher in the UPCA can ever have positional information. Thus no cell can know the position of its subpyramid in the UPCA and, in particular, no cell can know that it is the apex cell. Consequently, if a language is accepted by a UPCA, then every subpyramid must also accept the language. This enforces strict adherence to the concept of local processing, since the computation on a given cell's base must be completed entirely within that cell's subpyramid. This implies that all UPCA algorithms must use a balanced divide-and-conquer method that composes solutions to four problems with base size $2^{n-1} \times 2^{n-1}$ into a single solution to the size $2^n \times 2^n$ problem.

REFERENCES

1. C. R. Dyer and A. Rosenfeld, Cellular pyramids for image analysis, Tech. Rep. No. 544, Computer Science Center, Univ. of Maryland, College Park, Maryland, May 1977.
2. C. R. Dyer, Cellular pyramids for image analysis, 2, Tech. Rep. No. 596, Computer Science Center, Univ. of Maryland, College Park, Maryland, November 1977.

3. A. Nakamura and C. R. Dyer, Bottom-up cellular pyramids for image analysis, *Proc. Int. Joint Conf. Pattern Recognition*, 4th, 1978.
4. R. E. Stearns, J. Hartmanis, and P. M. Lewis II, Hierarchies of memory limited computations, *Proc. Switching Automata Theory*, 6th, 179-190 (1965).
5. J. W. Thatcher, Tree automata: an informal survey, in "Currents in the Theory of Computing" (A. V. Aho, ed.), pp. 143-172. Prentice-Hall, Englewood Cliffs, New Jersey 1973.
6. M. Minsky and S. Papert, "Perceptrons." MIT Press, Cambridge, Massachusetts, 1969.

Chapter 7

SPECIAL TYPES OF ACCEPTORS

7.1 INTRODUCTION

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.

7.2 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* or *marking 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" that A puts down in specified positions. If A wants to make a mark z_i that has already been made, it must first find and erase the existing instance of z_i before it can create a new instance. (Informally: if A has already put down pebble z_i , and wants to put it down elsewhere, A must first go to the position of z_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 $(q_1, u_{11}, \dots, u_{1k})$ and symbol $(w_1, v_{11}, \dots, v_{1k})$ give rise to state $(q_2, u_{21}, \dots, u_{2k})$ and symbol $(w_2, v_{21}, \dots, v_{2k})$, where each u and v is 0 or 1, then $w_2 = w_1$, and for each i , $1 \leq i \leq k$, only the following combinations of $(u_{1i}, v_{1i}, u_{2i}, v_{2i})$ 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 u s indicate which pebbles are currently being "carried" by $A^{(k)}$, while the v s indicate which of them have been put down at the position of the current symbol (1s indicate the presence of pebbles and 0s indicate their absence). The five permissible combinations have the interpretations shown in Table 7.1. We assume that the initial state is $(q_0, 1, \dots, 1)$, meaning that $A^{(k)}$ is initially carrying all the pebbles. It is not hard to see that, as a consequence of the above

TABLE 7.1

$(u_{1i}, v_{1i}, u_{2i}, v_{2i})$	Meaning
$(0, 0, 0, 0)$	$A^{(k)}$ was not carrying the i th 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 i th pebble was at the current position, but $A^{(k)}$ did not pick it up.
$(0, 1, 1, 0)$	The i th pebble was at the current position, and $A^{(k)}$ did pick it up.
$(1, 0, 1, 0)$	$A^{(k)}$ was carrying the i th pebble, and did not put it down.
$(1, 0, 0, 1)$	$A^{(k)}$ was carrying the i th pebble, and did put it down.

conditions, in any given configuration, for each i , either $u_i = 1$ or there exists exactly one position at which $v_i = 1$. The requirement that $w_2 = w_1$ 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.

7.2.1 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 $\{x^n y^n | n = 1, 2, \dots\}$ 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 x and y , 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 x and y , 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 $h > k$ such that h -pebble acceptors are strictly stronger than k -pebble acceptors (see, e.g., [1] and [5]). Thus there is an infinite hierarchy of pebble languages, all containing the finite-state languages and contained in the tape-bounded languages.

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. Clearly, 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].

7.2.2 Pebble Rectangular-Array Acceptors

In two dimensions, a one-pebble acceptor *is* stronger than an FSA. For example, a deterministic, tape-bounded one-pebble acceptor $A^{(1)}$ can decide whether the center symbol in a square, odd-side-length array Σ of x s and y s is an x , which cannot be done without the pebble (Theorem 4.3.4). 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 x , 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 7.2.1 A deterministic, tape-bounded one-pebble acceptor can determine whether the set of x s in a rectangular array Σ of x s and y s is connected.

Proof (see [1], Theorem 4): The acceptor A first checks whether the rows of Σ that contain x 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 row that contains x s, followed by a run of rows that do not, followed by a row that does. If so, the x s in Σ are evidently disconnected. If the rows with x s do form a single run, A proceeds to the next step, which is to check whether the x 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 x s separated by runs of y s. It evidently suffices to check that any two *consecutive* runs of x 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 W of x s, they must lie on the same border of W , since the same component of y s is adjacent to both of them. Thus when A reaches P , it puts down the pebble and follows the border β defined by P and the y immediately succeeding it. At each move

around β , if A is at a point Q' that has ys on its left, it scans leftward until it hits an x (or reaches the border of Σ). If this x 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 xs in Σ are not connected.

Suppose that A has found that the rows containing xs form a single run, and that the xs in each row are all connected. It then remains only to show that the xs in each two consecutive rows r_1, r_2 are connected to each other. If a run of xs in r_1 is adjacent to a run of xs in r_2 , this is immediate; hence we may assume that these runs are interleaved. In particular, there exists a run ρ_1 of xs in r_1 , followed (or preceded) by a run of ys that is adjacent to a run ρ_2 of xs 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 ys on its left, it scans leftward through that run of ys and checks whether any of them has the pebble *above it*. If so, Q' is Q , and A has verified that the xs 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 xs in r_1 are not connected to those in r_2 , so that the xs in Σ are not connected. If A succeeds in verifying that the xs in each pair of consecutive rows are connected, it has confirmed that all the xs in Σ are connected, and can accept Σ . $\square$

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

COROLLARY 7.2.2 A one-pebble acceptor can determine whether the set of xs in Σ is simply-connected.

Proof: A checks that the xs are connected, and analogously checks whether the ys are connected. (If the border of Σ does not consist entirely of ys , the latter check is carried out as if there were an additional row or column of ys at each edge of Σ .) If the ys are connected, the xs have no holes. $\square$

It is shown in [1, Theorem 5] that a one-pebble acceptor cannot

determine whether two simply-connected components of xs W_1 and W_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 xs [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.

7.2.3 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 acceptor $A^{(4)}$ with four pebbles can do these things. All acceptors in this section are deterministic and tape-bounded.

PROPOSITION 7.2.3 A three-pebble acceptor $A^{(3)}$ can find the outer border of its input array Σ .

Proof: See the proof of the first part of Theorem 4.4.5, in which the marks x, y, z are used as pebbles (i.e., only one instance of each is used at any one time). $\square$ (Compare the proof in [4], Lemma 3.1.1.)

THEOREM 7.2.4 ([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: See the second part of the proof of Theorem 4.4.5. $\square$

THEOREM 7.2.5 ([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 7.2.4, 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 (analogously, $A^{(4)}$ can also visit the part of $\Sigma \cap r$ to the left of P). 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 β defined by Q and this $\#$, using pebble z_1 to mark its current position; the positions of pebbles z_2 and z_3 relative to z_1 to keep count of its net horizontal and vertical displacements m and n relative to Q ; and the position of pebble z_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 z_2 and z_3 along β is as described in the proof of Theorem 4.4.5. The first time z_3 reaches the same point of β as z_1 (i.e., $n = 0$), if z_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 z_1 around β . If $m > 0$, we drop z_4 at the position of z_2 , since this is the minimum positive value of m found so far, and during border following, we maintain z_4 at a constant distance from z_1 . At subsequent times when $n = 0$,

- (a) If $m = 0$, we are back at Q , so that z_4 's distance from z_1 is the desired minimum positive m . We then follow β again, using z_1 , z_2 , and z_3 as before, but now keeping z_4 at a fixed distance from z_1 . When we reach the point Q' at which $n = 0$ and z_2 is at the same position as z_4 , we know that we are at the first point of $\Sigma \cap r$ to the right of Q , and can now resume the rightward scan.
- (b) If $m < 0$, we continue following β .
- (c) If $m > 0$ and z_2 is farther from z_1 than z_4 is, we continue following β , since the current m is not the minimum.
- (d) If $m > 0$ and z_4 is farther from z_1 than z_2 is, we move z_4 to the current position of z_2 , since its old position was not the minimum; we then resume following β .

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 z_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. $\square$

COROLLARY 7.2.6 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 β , which must be the outer border of Σ ; it then follows β , 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 β . Since from every point of Σ , if we move left through non- $\#$ s, we must hit β , 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 β . However, if we give A one pebble z , it can use z to mark its starting point on β , and accept when it reaches z again [4, Theorem 1]. These remarks imply

PROPOSITION 7.2.7 If Σ is simply-connected, there exists an FSA that accepts Σ iff it contains a specified symbol.

PROPOSITION 7.2.8 If Σ is simply-connected, there exists a one-pebble acceptor that accepts Σ iff it fails to contain a specified symbol.

THEOREM 7.2.9 ([4, Theorem 3]) There exists a one-pebble acceptor $A^{(1)}$ that accepts E iff it is not simply-connected.

Proof: It is easily seen that Σ is multiply connected iff, on each border β 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 β . Based on this fact, $A^{(1)}$ operates as follows: It moves, say, rightward until it hits a border β of Σ , say at P , and puts down z 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 β' of Σ (possibly the same as β). $A^{(1)}$ follows β' ; at each step:

- (a) If $A^{(1)}$ finds z , then P and P' are on the same border (i.e., $\beta = \beta'$). In this case, no conclusion can be reached as yet about the simple-connectedness of Σ . $A^{(1)}$ picks up z , moves along β , and repeats the entire process. (Note that if Σ is simply-connected, it will always be the case that $\beta = \beta'$, 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 z , it continues following β' .
- (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 z there, it moves

back to the left end and continues following β' . If it finds z there, it has followed β' completely around and returned to P' . If this happens, $A^{(1)}$ cannot have found z while following β' [i.e., case (a) could not have occurred], so β' must be different from β . 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. $\square$

THEOREM 7.2.10 ([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 7.2.9, except that it marks its starting point P on β with its other pebble z' . If case (b) never occurs, $A^{(2)}$ will follow β completely around, verifying at each stage (where applicable) that β' is the same as β , until it finds z' again. When this happens, $A^{(2)}$ knows that Σ is simply-connected, and can accept. If case (b) occurs, $A^{(2)}$ halts in a nonaccepting state. $\square$

By Theorem 4.4.3, 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 is simply-connected.

7.3 PUSHDOWN ACCEPTORS

One-dimensional pushdown acceptors (PDAs) have been widely studied because of their well-known relationship to context-free grammars [7, Chapters 5 and 12]. Two-dimensional PDAs have not been widely studied; our main result concerning 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 $(q_1, u_1 \dots u_m)$ gives rise to $(q_2, v_1 \dots v_n)$ under δ , where q_1, q_2 are

in Q' and $u_1 \dots u_m, v_1 \dots v_n$ are in U^* , then one of (a–c) must hold:

- (a) $n = m + 1$ and $v_1 \dots v_{n-1} = u_1 \dots u_m$. In this case, we say that A has pushed symbol v_n onto the top of its stack.
- (b) $n = m$ and $v_1 \dots v_n = u_1 \dots u_m$. In this case, the stack is unaffected by the transition.
- (c) $n = m - 1$ and $v_1 \dots v_n = u_1 \dots u_{m-1}$. In this case, A is said to have popped symbol u_m from the top of its stack.

Moreover, q_2 [and v_n , in case (a)] and the move made by A are allowed to depend only on q_1, u_m (the topmost stack symbol), and the current input symbol (as well as on A 's previous move); they cannot depend on $u_1, \dots, u_{m-1}$. (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 $(q_1, u_1 \dots u_m)$, q_1 is called the *internal state* of A , and $u_1 \dots u_m$ constitutes A 's *stack contents*.] In all of the above, A can be either one-way or two-way.

7.3.1 String PDAs

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

- (1) The set of strings $\{x^n y^n \mid n = 1, 2, \dots\}$ 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 x , it pushes a symbol u onto the top of its stack, so that after reading k x s, its stack contains the string u^k . When A reads a y , it pops a u from the top of its stack. Unless A reads a succession of x s followed by a succession of y s, it halts in a nonaccepting 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 y s is equal to the number of x s, and accepts σ . This example shows that one-way deterministic PDAs are strictly stronger than FSAs.
- (2) The set of symmetric strings (palindromes) $\sigma = \omega\omega^R$, where ω is any string on, say, $\{x, y\}$ 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 x or y , it pushes an x or y 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 nonaccepting 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 PDAs, nondeterministic is strictly stronger than deterministic. It is well known [7, pp. 74–78] that the one-way nondeterministic PDAs accept exactly the context-free languages (see Section 8.1.2).

- (3) Two-way deterministic PDAs can accept languages that are not context-free, and so are stronger than one-way nondeterministic PDAs. For example, consider the set of strings $\{x^n y^n z^n \mid n = 1, 2, \dots\}$, which is not context-free [7, p. 66, Problem 4.15]. To accept this set, A first verifies that the numbers of x s and y s are equal, as in (1); it then moves back to the beginning of the y s and verifies in the same way that the numbers of y s and z 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, say, x s and y s. 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 x s and y 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 ac-

cepts. An analogous construction can be used by a two-way deterministic PDA to recognize palindromes.

7.3.2 Array PDAs

The ability of PDAs 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 PDAs on simply-connected arrays. (Array PDAs will always be assumed to be "four-way," i.e., able to move in any direction.) Specifically, we shall prove

THEOREM 7.3.1 There exists a PDA that accepts any simply-connected input array Σ after it has scanned all of Σ .

COROLLARY 7.3.2 If Σ is simply-connected, there exists a PDA that accepts Σ iff it fails to contain a specified symbol. (Compare Propositions 7.2.7 and 7.2.8.)

The proof of Theorem 7.3.1 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 2.5.4), 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 shown in Table 7.2.

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

TABLE 7.2

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

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 end 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:

Example 1. Let Σ be

$$\begin{aligned} 1 &\rightarrow P P P P \leftarrow 6 \\ 2 &\rightarrow P \quad P \leftarrow 5 \\ 3 &\rightarrow P P P P \leftarrow 4 \end{aligned}$$

Table 7.3 indicates the events that occur at successive points of the border of Σ (starting from point 1) or between pairs of border points.

TABLE 7.3

Point(s)	Current stack	Event
1	empty	ℓ
1	ℓ	d
2	ℓd	ℓ
3	$\ell d \ell$	u
3	$\ell d \ell u$	ℓ
4	$\ell d \ell u \ell$	r
5	$\ell d \ell u$	r
6	$\ell d \ell u r$	r
1	$\ell d \ell u r r$	ℓ

TABLE 7.4

Point(s)	Current stack	Event
1	empty	ℓ
2	ℓ	ℓ
3	$\ell\ell$	ℓ
3,4	$\ell\ell\ell$	v
4	$\ell\ell\ell v$	r
5	$\ell\ell\ell v r$	ℓ
5,6	$\ell\ell\ell v$	v
6	$\ell\ell\ell v v$	r
7	$\ell\ell\ell v v r$	r
8	$\ell\ell\ell v v r r$	ℓ
8	$\ell\ell\ell v v r$	d
9	$\ell\ell\ell v r d$	ℓ
10	$\ell\ell\ell v r d \ell$	r
10,11	$\ell\ell\ell v r d$	e
11	$\ell\ell\ell v r$	ℓ
12	$\ell\ell\ell v$	uu
12	$\ell\ell\ell$	ℓ
12	$\ell\ell\ell\ell$	d
13	$\ell\ell\ell\ell d$	ℓ
14	$\ell\ell\ell\ell d \ell$	r
14,15	$\ell\ell\ell\ell d$	e
15	$\ell\ell\ell\ell$	ℓ
16	$\ell\ell\ell\ell\ell$	r
17	$\ell\ell\ell\ell$	r
18	$\ell\ell\ell$	r
19	$\ell\ell$	r
20	ℓ	r
1	empty	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.

Example 2. Let Σ be

$$\begin{array}{ll}
 & 1 \rightarrow PP \leftarrow 20 \\
 8 \rightarrow P & P \leftarrow 7 \quad 2 \rightarrow PP \leftarrow 19 \\
 9 \rightarrow P \leftarrow 10 & 11 \rightarrow P \leftarrow 6 \quad 5 \rightarrow P \leftarrow 4 \quad 3 \rightarrow PP \leftarrow 18 \\
 & 12 \rightarrow P & P \leftarrow 17 \\
 & 13 \rightarrow PP \leftarrow 14 & 15 \rightarrow PP \leftarrow 16
 \end{array}$$

In this case the events are as shown in Table 7.4.

7.4 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, on the order of the array area, for acceptance, while the latter require large amounts of hardware, i.e., a number of cells on the order of the array area. 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 (a number of cells on the order of the array width), but is still able to perform some tasks quickly, in time on the order of the array diameter.

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 ,
- $\delta: Q \times Q \times Q \times V \rightarrow 2^{Q \times V}$ is the state transition function, and
- $\mu: Q \times V \rightarrow 2^{(-1,0,1)}$ 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 $c_1, \dots, c_n$ whose length is equal to the width of Σ , together with two special "cells" c_0 and c_{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 c_i reads the symbol v in its position,

senses the states q_1, q_2 of its neighbors $c_{i\pm 1}$, 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 is the current (old) state of c_i . The move function depends only on the (new) state of and symbol read by the distinguished cell c_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 $\mu(q, \#_l) = 1$ and $\mu(q, \#_b) = -1$ for all q ; in other words, A bounces downward from $\#_l$ and upward from $\#_b$. It is understood that $\#_l$ and $\#_b$ can never be rewritten: $(q', v') \in \delta(q, q_1, q_2, \#_l \text{ or } \#_b)$ implies $v' = \#_l$ or $\#_b$, respectively. If c_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 L_x 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 c_i sees an x , it goes into state $q_A \in Q_A$. If any cell has its right neighbor in state q_A , it goes into state q_A . Evidently, iff Σ contains an x , cell c_1 goes into state q_A after at most $m + n$ time steps, where m and n are the height and width of Σ (m steps are needed for A to scan Σ from top to bottom, and at most n steps for the q_A signal to reach c_1).

THEOREM 7.4.1 The set of languages accepted by PSAs is the same as that accepted by BCAs.

Proof: 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 are 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 the Appendix to Section 3.4.1) which causes all the cells simultaneously to change the third terms of their states to ± 1 (+1 for a downward move, -1 for an upward move). 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 is its previous state). The next transition of A can now be simulated. If the leftmost cell of the "active" row (having pairs of 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 n times real time (where n is the width of Σ), since $O(n)$ steps are required for the synchronizations between simulations of transitions of A . [The simulation can also be done in $O(n)$ 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 3.4.5), 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 c_1 finds that C has accepted, it accepts. Note that if A is deterministic, so is C , and vice versa. $\square$

In the remainder of this section, we consider two restrictions on PSAs: not allowing them to rewrite the symbols in Σ , and not allowing them to move upward. We use the prefixes D, NW, and OW to denote "deterministic," "nonwriting," and "one-way" PSAs, respectively. The set of languages accepted by (D)(NW)(OW)PSAs will be denoted by $\mathcal{L}_{(D)(NW)(OW)PS}$. Thus Theorem 7.4.1 can be restated as: $\mathcal{L}_{(D)PS} = \mathcal{L}_{(D)BC}$.

We first show that nonwriting PSAs are strictly stronger than FSAs, but strictly weaker than BCAs.

THEOREM 7.4.2 $\mathcal{L}_{(D)FS} \subsetneq \mathcal{L}_{(D)NWPS} \subsetneq \mathcal{L}_{(D)BC}$.

Proof: An NWPSA 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 c_1 in order for A to do this). When the FSA accepts, the special cell sends a signal to c_1 , which accepts. Thus NWPSAs can accept all the FSA languages. Conversely, on a one-row array, an NWPSA can evidently simulate a one-dimensional BCA, and so can accept non-FS languages; thus FSAs cannot accept all the NWPSA languages. These results remain valid if the acceptors in question are assumed to be deterministic.

A BCA can simulate an NWPSA, as in the first part of the proof of Theorem 7.4.1; thus BCAs can accept all the NWPSA languages. Conversely, on a one-column array, an NWPSA can evidently be simulated by a one-dimensional FSA; thus NWPSAs cannot accept all the (one-dimensional) BC languages. These results too remain valid for deterministic acceptors. $\square$

For one-way PSAs, the ability to write provides no advantage (compare Section 3.3.3 on one-way TAs and TBAs). In the following two theorems, we establish some proper inclusion relations on the classes of languages accepted by (deterministic) nonwriting one-way PSAs.

THEOREM 7.4.3 $\mathcal{L}_{(\text{D})\text{OWNWPS}} \subsetneq \mathcal{L}_{(\text{D})\text{NWPS}}$.

Proof: Let L be the set of $2n(\text{rows}) \times n(\text{columns})$ arrays of x s and y 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 DOWNWPSA A , e.g., by moving a marker one step to the right whenever A moves downward, until it reaches c_n , then moving it one step to the left per downward move of A , and verifying that the marker returns to c_1 just as A reaches the bottom row.) To see that $L \in \mathcal{L}_{\text{DNWPS}}$, 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 c_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 $L \notin \mathcal{L}_{\text{OWNWPS}}$. Suppose A were an OWNWPSA that accepts L . For any top half Σ' , let $Q_{\Sigma'}$ be the set of n -tuples of states in which A could be after scanning Σ' and which can lead to acceptance after the bottom half (Σ' again) is scanned. Then if $\Sigma'' \neq \Sigma'$, we must have $Q_{\Sigma''} \cap Q_{\Sigma'} = \emptyset$; indeed, if an n -tuple ν were in both, there would exist some bottom half Σ''' for which the initial n -tuple ν can lead to acceptance, and Σ''' must be the same as both Σ' and Σ'' —a contradiction. Thus each of the 2^n top halves must give rise to a nonempty set $Q_{\Sigma'}$, 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} —a contradiction. $\square$

THEOREM 7.4.4 $\mathcal{L}_{\text{DOWNWPS}} \subsetneq \mathcal{L}_{\text{OWNWPS}}$.

Proof: Let L' be the set of $2n \times n$ arrays of x s and y s in which some row occurs in both the top and bottom halves. A nondeterministic OWNWPS 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 $2n \times n$, by moving a marker as in the proof of Theorem 7.4.3.

Suppose that L' were accepted by a DOWNWPSA 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 of possible sets of rows that can occur in the top half is

$$\binom{2^n}{1} + \binom{2^n}{2} + \cdots + \binom{2^n}{n}$$

(depending on how many of these rows are distinct). For large n , this number is greater than $|Q|^n$, so that two of the sets, R_1 and R_2 , must yield the same state combination ν . Let r be a row in R_1 but not in R_2 (or vice versa), and let the bottom half of the array consist entirely of r s. When A scans n r s, starting in state combination ν , it either accepts or does not accept. In the former case, A accepts top half R_2 (not containing r) followed by n r s, which is incorrect; in the latter case, A does not accept top half R_1 (containing r) followed by n r s, which is incorrect. $\square$

It is an open question whether $\mathcal{L}_{\text{DNWPS}} \subsetneq \mathcal{L}_{\text{NWPS}}$. As regards the relationship between one-way PSAs and FSAs, we have

THEOREM 7.4.5 $\mathcal{L}_{(\text{D})\text{OWNWPS}}$ and $\mathcal{L}_{(\text{D})\text{FS}}$ are incomparable.[§]

Proof: On a one-row array, a DOWNWPSA can simulate a DBCA, which can accept non-FS languages such as $\{x^n y^n \mid n = 1, 2, \dots\}$. Conversely, consider the set L of rectangular arrays of x s, y s, and z s in which there is a 4-path consisting of x s and y s from the lower left to the lower right corner. We assume that, in L , each x always has exactly two x s or y s as 4-neighbors (except at the ends of a path) and that the 4-neighbors of each y are all x s. Thus the x s form a set of disjoint paths, and the y s occur only

[§] This result is due to C. R. Dyer.

where these paths cross. In following a path, when we move onto a y we always move off it in the same direction. Thus paths can be followed deterministically, so that a DFSA can accept L by moving to the lower left corner, following the path that begins there, and accepting if it reaches the lower right corner.

We now show that no OWNWPSA can accept L . Consider the set of $(2n) \times (4n + 1)$ arrays of x s, y s, and z s in which the bottom row is $zxzx \dots zxz$ and the array contains a set of inverted-U-shaped 4-paths of x s and y s that do not touch and that cross only at the y s, each of which connects a distinct pair of x s in the bottom row, as in the following example (where the z s are represented by blanks):

```

      x x x x x x x x
        x                      x
      x x x x y x x x x      x
        x      x      x      x
    x x y x x  x  x x y x x  x
    x  x  x  x  x  x  x  x  x

```

Thus there are n paths (one per pair of x s in the bottom row), and $2n$ rows (or fewer) suffice to construct these paths without touching. The number of possible pairings of x s in the bottom row is $\binom{2n}{2} \binom{2n-2}{2} \dots = (2n)!/2^n$; hence there are $(2n)!/2^n$ such arrays Σ_i that are topologically distinct.

Suppose A were a OWNWPSA that accepted L . For any two topologically distinct Σ_i and Σ_j we can easily construct a $4 \times (4n + 1)$ array Σ_{ij} such that the result is in L when Σ_{ij} is appended to the bottom of Σ_i , but not when it is appended to the bottom of Σ_j . Indeed, let x_h be paired with x_k in Σ_i but not in Σ_j ; then we simply construct a 4-row array in which there is a path of x s and y s from the lower left corner to x_h and from x_k to the lower right corner. It follows that the sets of $(4n + 1)$ -tuples of states in which A can be after scanning Σ_i and Σ_j , and which can lead to acceptance when the bottom is reached, must be disjoint. But there are only $|Q|^{4n+1}$ disjoint sets of $(4n + 1)$ -tuples of states, and this number is less than $(2n)!/2^n$ —a contradiction. $\square$

We conclude this section by showing that a DOWNWPSA can recognize connectedness (compare Theorem 7.2.1), which is an open question for FSAs.

THEOREM 7.4.6 A DOWNWPSA can determine whether or not the set of xs in a rectangular array Σ of xs and ys is connected.

Proof: Let r_k denote the k th row, and R_k the top k rows, of Σ . We shall describe how a DOWNWPSA A , when it is on r_k , can construct a string of parentheses that indicates how the runs of xs in r_k are connected in R_k . Specifically, for each component U of xs in R_k , we put a left parenthesis at the leftmost point (if any) in which U meets r_k , and a right parenthesis at the rightmost point. (If U 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 xs , 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 (as we shall show in the next two paragraphs) which runs in r_k belong to the same component of xs in R_k . A can then move to r_{k+1} , having memorized r_k and its parenthesization, and examine the components of xs 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 xs in R_{k+1} , and can thus construct the parenthesization of r_{k+1} .

For any run ρ of xs in r_k , as we move leftward from ρ , let $(_\rho$ 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 $(_\rho$.] We claim that $(_\rho$ is just the left parenthesis at the leftmost point of $U_\rho \cap r_k$, where U_ρ is the component of xs in R_k that contains ρ . Indeed, suppose that some other component V meets r_k between $(_\rho$ and ρ [this is the only way that other parentheses can occur between $(_\rho$ and ρ]. If a pair of runs in $V \cap r_k$ separated a pair of runs in $U_\rho \cap r_k$, the paths of xs in R_k joining these pairs would have to cross, contradicting the fact that $V \neq U_\rho$. Hence all of $V \cap r_k$ must lie between $(_\rho$ and ρ , so that V 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 V ; it must come from U_ρ itself.

It follows that, given any two runs ρ_1, ρ_2 of xs in r_k , A can tell whether or not they belong to the same component in R_k by checking whether or not $(_{\rho_1} = (_{\rho_2}$. 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 xs that met r_k fails to meet r_{k+1} . A can also tell when a component of xs reaches the

bottom row of Σ . If exactly one of these events occurs, the x s in Σ are connected; if more than one occurs, they are not. $\square$

REFERENCES

1. M. Blum and C. Hewitt, Automata on a 2-dimensional tape, *Proc. IEEE Conf. on Switching and Automata Theory*, 8th, pp. 155–160 (1967).
2. M. Blum and W. J. Sakoda, On the capability of finite automata in 2 and 3 dimensional space, *Proc. IEEE Conf. on Foundations of Computer Science*, 18th, pp. 147–161 (1977).
3. J. Mylopoulos, On the recognition of topological invariants by 4-way finite automata, *Comput. Graphics and Image Processing* **1**, 308–316 (1972).
4. A. N. Shah, Pebble automata on arrays, *Comput. Graphics and Image Processing* **3**, 236–246 (1974).
5. P. Hsia and R. T. Yeh, Marker automata, *Information Sci.* **8**, 71–88 (1975).
6. A. N. Shah and A. Rosenfeld, Two heads are better than one, *Information Sci.* **10**, 155–158 (1976).
7. J. E. Hopcroft and J. D. Ullman, "Formal Languages and their Relation to Automata." Addison-Wesley, Reading, Massachusetts, 1969.
8. A. N. Shah, On traversing properties of array automata, Tech. Rep. No. 274, Computer Science Center, Univ. of Maryland, College Park, Maryland. November 1973.
9. S. M. Selkow, One-pass complexity of digital picture properties, *J. Assoc. Comput. Mach.* **19**, 283–295 (1972).
10. A. Rosenfeld and D. L. Milgram, Parallel/sequential array automata, *Information Processing Lett.* **2**, 43–46 (1973).

Chapter 8

GRAMMARS

In the previous chapters, we have discussed various types of string and array *acceptors*. This chapter introduces *grammars* that can be used to either generate or accept (“parse”) sets of strings or arrays. It defines a hierarchy of such grammars, and also establishes equivalences between certain classes of grammars and of acceptors.

8.1 STRING GRAMMARS

We first consider grammars that generate or parse sets of strings. Sections 8.1.1 and 8.1.2 review the usual definitions of such grammars. Section 8.1.3 defines “isometric” grammars that rewrite #s, rather than simply extending the given string; this alternative approach will be important when we treat array grammars in Section 8.3. Section 8.1.4 discusses various definitions of “parallel” grammars and their relationships to conventional (sequential) grammars.

8.1.1 Grammars

Informally, a (string) *grammar* G is a mechanism that generates a set of strings by a process of repeatedly substituting one substring for another, starting with a standard initial string. The (*rewriting*) *rules* or *productions* of G specify which substrings can be replaced by which others. The *language* of G is the set of strings (usually required to consist entirely of symbols of a special type, called “ter-

minal" symbols) that can be produced in this way. We shall now define these ideas more precisely.

As we did in Chapter 3 for automata, we shall require that grammars satisfy certain finiteness and nontriviality conditions. Specifically, we shall assume that the *vocabulary*, or set of symbols, the subset of "terminal" symbols, and the set of rules are all finite nonempty sets. We shall also assume that the initial string consists of a single nonterminal symbol s . Under these assumptions, we can formally define a ("type 0") grammar G to be a quadruple (V, V_T, Π, s) , where

V is the vocabulary (a finite, nonempty set of symbols),

$V_T \subsetneq V$ is the terminal vocabulary ($\neq \emptyset$),

$s \in V - V_T$ is the initial symbol or "start symbol," and

Π is a finite nonempty set of pairs (λ, ζ) , where λ and ζ are non-null[§] strings of elements of V . These pairs are called the rules of G , and are usually written in the form $\lambda \rightarrow \zeta$ (denoting the fact that λ can be replaced by ζ).

In order to define the language of G , we must introduce the notion of a "derivation" in G . We say that the string σ' is *directly derived* from the string σ in G (notation: $\sigma \xRightarrow{G} \sigma'$) if there exists a rule $\lambda \rightarrow \zeta$ of G such that λ is a substring of σ , and τ is the result of replacing some occurrence of λ (as a substring of σ) by ζ . We say that σ' is *derived* from σ in G (notation: $\sigma \xRightarrow{*G} \sigma'$) if there exists a sequence of strings $\sigma = \sigma_0, \sigma_1, \dots, \sigma_n = \sigma'$, where $n \geq 0$, such that σ_i is directly derived from σ_{i-1} in G , $1 \leq i \leq n$. The set of strings on V_T that can be derived in G from the initial string s (consisting of the single symbol s) is called the *language* of G , and is denoted by $L(G)$. A sequence of strings $s = \sigma_0, \sigma_1, \dots, \sigma_n = \sigma' \in L(G)$ for which $\sigma_{i-1} \xRightarrow{G} \sigma_i$ is called a *derivation* of τ in G . (We shall omit "in G " from now on unless there is danger of confusion.)

Given any grammar G , there exists a grammar G' in which terminal symbols are never rewritten by any rule, and such that $L(G') = L(G)$. In fact, we can define G' to have nonterminal vocabulary $(V - V_T) \cup V'_T$, where V'_T is a set of primed copies of the

[§] We require ζ to be nonnull to prevent the null string from being generated by a grammar; this corresponds to our requirement, in earlier chapters, that the input strings of acceptors always be nonnull.

symbols in V_T . For each rule $\lambda \rightarrow \zeta$ of G , we have the rule $\lambda' \rightarrow \zeta'$ in G' , where the primes denote the fact that each symbol in V_T is replaced by the corresponding symbol in V'_T . In addition, G' has the rules $x' \rightarrow x$ for all $x \in V_T$. Evidently, a string on V_T is derivable in G' if it is derivable in G (since G' can derive the corresponding string on V'_T , which can then be changed into a V_T string using the $x' \rightarrow x$ rules), and conversely (since the $x' \rightarrow x$ rules are the only rules of G' that produce symbols in V_T).[§] We shall assume from now on that grammars never rewrite terminal symbols.

A grammar can function as an "acceptor" by performing derivations in reverse. Given a string σ on V_T , we say that G *parses* σ if there exists a sequence of strings $\sigma = \sigma_n, \sigma_{n-1}, \dots, \sigma_0 = s$ such that σ_{i-1} is the result of applying a rule $\lambda \rightarrow \zeta$ of G in reverse (i.e., replacing some ζ by λ) to σ_i , $1 \leq i \leq n$. It is clear that this is the same thing as saying that $\sigma_0, \dots, \sigma_n$ is a derivation; thus σ is parsed by G iff it is generated by G .

A less trivial equivalence between grammars and acceptors is provided by

THEOREM 8.1.1 The languages generated by grammars are the same as the languages accepted by Turing acceptors.

Proof: Let L be generated by G ; then we can define a Turing acceptor A that operates as follows: Given a string σ , A creates an s (separated from σ by a distinctive marker) and then (nondeterministically) applies a sequence of rules of G , starting with the s (and leaving σ intact), to produce a sequence of strings $\sigma_0 \equiv s, \sigma_1, \sigma_2, \dots$. At each step of this process, it compares σ_i with σ ; this can be done by moving back and forth from σ_i to σ and marking off corresponding symbols if they are the same. If at any stage A finds that $\sigma_i = \sigma$, it accepts. Evidently, this happens iff $\sigma \in L(G)$.

Conversely, let L be accepted by A . We shall define below a grammar G_A that generates two copies of an arbitrary string and simulates the operation of A on one copy. If the simulation accepts, G_A erases that copy and converts the other copy to a terminal string. Evidently, a terminal string σ is produced by G_A in this way iff $\sigma \in L(A)$.

[§] Note that if the $x' \rightarrow x$ rules are applied too soon, it may be impossible to complete a derivation, since a rule $\lambda' \rightarrow \zeta'$ may require the presence of some primed symbol that has already become unprimed.

The grammar G_A has the following sets of rules:

$$(1) \left. \begin{array}{l} s \rightarrow s'(x', x) \\ s' \rightarrow s'(x, x) \\ s' \rightarrow ((q_0, N)(x'', x)) \\ s \rightarrow ((q_0, N)(x''', x)) \end{array} \right\} \text{for all } x \text{ in the} \\ \text{vocabulary of } A.$$

These rules generate a string of the form $((q_0, N)(x''', x))$ or $((q_0, N)(x'_1, x_1))(x_2, x_2) \dots (x_{n-1}, x_{n-1})(x'_n, x_n)$ (where $n > 1$). Primed x s are always at the right end of the string; double-primed x s, at the left end; and triple-primed x s are always singletons. Here q_0 represents the initial state of A at the left end of its input string.

- (2) For all u, v, w, x, y, z ,
- (a) $((q_1, d)(w, x)) \rightarrow ((q_2, N)(t, x))$ for all $(q_2, t, N) \in \delta(q_1, w, d)$, where δ is the transition function of A , and where if w is singly, doubly, or triply primed, so is t . These rules simulate the case in which A changes state and rewrites a symbol but does not move.
 - (b) $(u, v)((q_1, d)(w, x)) \rightarrow ((q_2, L)(u, v))(t, x)$ for all $(q_2, t, L) \in \delta(q_1, w, d)$, where if w is primed, so is t . These rules handle the case in which A moves left, provided it is not at the left end of the string of non- $\#$ s.
 - (b') $\left. \begin{array}{l} ((q_1, d)(w'', x)) \rightarrow ((q_2, L)(\#'', \#))(t, x) \\ ((q_1, d)(w''', x)) \rightarrow ((q_2, L)(\#'', \#))(t', x) \end{array} \right\} \text{for all } (q_2, t, L) \in \delta(q_1, w, d).$ Rules (b') handle the case in which A is at the left end of the non- $\#$ s and moves left.
 - (c) $((q_1, d)(w, x))(y, z) \rightarrow (t, x)((q_2, R)(y, z))$ for all $(q_2, t, R) \in \delta(q_1, w, d)$, where if w is double-primed, so is t . These rules handle the case in which A is not at the right end of the string and moves right.
 - (c') $\left. \begin{array}{l} ((q_1, d)(w', x)) \rightarrow (t, x)((q_2, R)(\#', \#)) \\ ((q_1, d)(w''', x)) \rightarrow (t'', x)((q_2, R)(\#', \#)) \end{array} \right\} \text{for all } (q_2, t, R) \in \delta(q_1, w, d).$ Rules (c') handle the case in which A is at the right end of the non- $\#$ s and moves right.

These rules simulate A on the string, adding pairs of $\#$ s at its ends if necessary. Note that the conventions about single, double, and triple primes are preserved by these rules. Note also that the simu-

lation affects only the first terms of the pairs; the second terms are unchanged, except that $\#$ second terms may be added at the ends. We assume in (2) that q is not an accepting state of A .

- (3) For all w, x, y, z , where w may be double-primed and y may be primed, and for all accepting states q_A of A ,

$$((q_A, d)(w, x))(y, z) \rightarrow (w, x)((q_A, d)(y, z)).$$

When an accepting state is created, it moves rightward.

- (4) For all w, x, y, z , where w may be double-primed,

$$\begin{aligned} ((q_A, d)(y', z)) &\rightarrow ((y', z), q_A) && \text{if } z \neq \#, \\ (w, x)((q_A, d)(y', \#)) &\rightarrow ((w, x), q'_A), \\ ((q_A, d)(y''', z)) &\rightarrow z. \end{aligned}$$

When (q_A, d) reaches the right end of the string, it becomes q_A if the second term at the right end is non- $\#$, or erases the $\#$ and becomes q'_A if the second term at the right end is $\#$. If the string is a singleton, (q_A, d) turns it into a singleton terminal (the second term must be non- $\#$ in this case, since the original string of second terms was nonnull).

- (5) For all w, x, y , where w may be double-primed in the second case,

$$\begin{aligned} (w, \#)((y, \#), q'_A) &\rightarrow ((w, \#), q'_A), \\ (w, x)((y, \#), q'_A) &\rightarrow ((w, x), q_A) && \text{if } x \neq \#. \end{aligned}$$

The q'_A moves leftward, erasing $\#$ s; when it reaches a non- $\#$ second term (which must happen eventually, since the original string was nonnull), it becomes q_A .

- (6) For all w, x, y, z , where y may be primed,

$$\begin{aligned} (w, x)((y, z), q_A) &\rightarrow ((w, x), q_A)z && \text{if } x \neq \#, \\ ((y'', z), q_A) &\rightarrow z. \end{aligned}$$

The q_A moves left, turning pairs that have non- $\#$ second terms into their second terms; if it reaches the left end of the string, it erases itself, leaving only its pair's second term z . Note that z cannot be $\#$, by (4)–(6).

- (7) For all w, y, z , where y may be primed,

$$\begin{aligned} (w, \#)((y, z), q_A) &\rightarrow ((y, z), q_A), \\ (w'', \#)((y, z), q_A) &\rightarrow z. \end{aligned}$$

If the q_A reaches $\#$ second terms (which must lie to the left of

all the non- $\#$ second terms), it erases them; when it erases the last one, it also erases itself, leaving only its pair's second term z .

Evidently, this process generates a terminal string (namely, the original string of second terms; the terminal vocabulary of G_A is the same as the vocabulary of A) iff the simulation of A accepted that string (of first terms); in other words, G_A generates a terminal string σ iff $\sigma \in L(A)$. $\square$

8.1.2 Special Types of Grammars

A grammar G is called *context-sensitive* (or "type 1") if for each rule $\lambda \rightarrow \zeta$ of G there exist strings ξ , η , and ω on V , where ω is nonnull, and a symbol $X \in V - V_T$, such that $\lambda = \xi X \eta$ and $\zeta = \xi \omega \eta$. Thus any such rule rewrites a single nonterminal X , "in the context of" ξ and η , as a nonnull string ω . Note that in such a grammar, terminals are never rewritten.

We shall call a grammar G *monotonic* if for each rule $\lambda \rightarrow \zeta$ of G we have $|\lambda| \leq |\zeta|$.

THEOREM 8.1.2 The languages generated by monotonic grammars are the same as those generated by context-sensitive grammars.

Proof: Context-sensitive certainly implies monotonic, since ω nonnull implies $|\xi X \eta| \leq |\xi \omega \eta|$. To prove the converse, we show that any rule $\lambda \rightarrow \zeta$ with $|\lambda| \leq |\zeta|$ can be replaced by a set of context-sensitive rules without altering the language generated by the grammar. We can assume, as shown earlier, that the given grammar does not rewrite terminals, so that the symbols involved in the rule $\lambda \rightarrow \zeta$ are all nonterminals (unless the rule is of the form $x' \rightarrow x$, which is already context-sensitive).

Let $\lambda = X_1 \dots X_m$ and $\zeta = Z_1 \dots Z_n$, where $m \leq n$. Then we can replace $\lambda \rightarrow \zeta$ by the following sets of context-sensitive rules:

$$\begin{aligned} X_1 &\rightarrow X_{11} \\ X_{11}X_2 &\rightarrow X_{11}X_{22} \\ &\vdots \\ &\vdots \\ X_{m-1,m-1}X_m &\rightarrow X_{m-1,m-1}X_{mm} \end{aligned} \tag{1}$$

These rules successively change the symbols in λ to special forms that are understood to be unique to the given rule and to the given position in that rule, so that they can only be applied in the given sequence. If a rule is applied in the wrong place (only the beginning of λ is present, or a later part of λ is the same as its beginning), the rule sequence (1) cannot go to completion, and there is no way to get rid of the last-created special symbol (unless the rest of λ is later created; but if this happens, it cannot depend on the presence of the special symbols, and so could have happened earlier).

$$\begin{array}{l}
 X_{11}X_{22} \rightarrow Z_1X_{22} \\
 X_{22}X_{33} \rightarrow Z_2X_{33} \\
 \vdots \\
 X_{mm} \rightarrow Z_mZ_{m+1} \dots Z_n
 \end{array} \quad (2)$$

These rules change the special Xs to Zs after the next ones have been created (except in the case of the last one). As already pointed out, this process can go to completion only if all the special Xs are created, which can only happen if all of λ is present (or could have been present). Thus replacing $\lambda \rightarrow \zeta$ by (1) and (2) cannot change $L(G)$, since the only way (1) and (2) can create a string free of special symbols is to simulate a complete application of $\lambda \rightarrow \zeta$, and analogously for all the other rules of G . $\square$

THEOREM 8.1.3 The languages generated by monotonic grammars are the same as those accepted by tape-bounded acceptors.

Proof: When a terminal string σ is derived from s in a monotonic grammar, the successive strings in the derivation have non-decreasing lengths, so that none of them is longer than σ . Hence in the first part of the proof of Theorem 8.1.1, A can generate an s by turning some symbol x of σ into a pair (s, x) , and then simulating the application of a sequence of rules, creating further pairs as necessary, and shifting the pairs leftward or rightward if it comes to the end of σ . If the string of first terms generated in this way tries to become longer than σ , A cannot possibly be simulating a derivation of σ . Thus, in carrying out the proof of the first part of Theorem 8.1.1 in the case where G is context-sensitive, A need never move off the non- $\#s$.

Conversely, in the second part of the proof, note that the rules of G_A are all monotonic except for some rules in (4)–(7) which erase $\#$ s; but these rules are not needed if A is tape-bounded, since rules (2b') and (2c') are never used, so that pairs with $\#$ second terms are never created. [Since the string ends are marked (") and ('), the simulation of A need not involve $\#$ s at all, even when A bounces off them.] Thus if A is tape-bounded, G_A is monotonic. $\square$

A grammar G is called *context-free* (or "type 2") if in all rules $\lambda \rightarrow \zeta$ of G , λ is a single nonterminal symbol and ζ is nonnull. (Evidently, context-free implies context-sensitive.) G is called *finite-state* (or "type 3") if in all rules $\lambda \rightarrow \zeta$ of G , λ is a single nonterminal symbol and ζ either ends in a single nonterminal symbol, possibly preceded by terminal symbols, or else ζ is a nonnull string of terminal symbols.

THEOREM 8.1.4 The languages generated by finite-state grammars are the same as the languages accepted by finite-state acceptors.

Proof: Given G , we define an FSA A that operates on its input string σ as follows: A examines the right end of σ for matches to right-hand sides of rules of G . If $\sigma = \sigma_1 \zeta_1$, where $Z_1 \rightarrow \zeta_1$ is a rule of G , A memorizes Z_1 , moves to just past the beginning of ζ_1 (i.e., to the right end of σ_1), and looks for matches of the right end of $\sigma_1 Z_1$ to right-hand sides of rules of G . If it finds such a match, say to $Z_2 \rightarrow \zeta_2 Z_1$, where $\sigma_1 = \sigma_2 \zeta_2$, this process is repeated, with σ becoming shorter at each repetition. If at some state A finds that $\sigma_{n-1} = \zeta_n Z_{n-1}$ matches the right-hand side of a rule of G whose left-hand side is s , then A accepts σ . Evidently, this can happen iff σ can be derived in G , using the succession of rules $s \rightarrow \zeta_n Z_{n-1}$, $Z_{n-1} \rightarrow \zeta_{n-1} Z_{n-2}$, $\dots$, $Z_2 \rightarrow \zeta_2 Z_1$, $Z_1 \rightarrow \zeta_1$. Thus A accepts σ iff $\sigma \in L(G)$.

Conversely, suppose that L is accepted by a one-way FSA A . We can define a finite-state grammar G that generates L as follows:

- (1) $s \rightarrow (q_0, x)$ for all x ;
- (2) $(q, x) \rightarrow x(q', y)$ for all $q' \in \delta(q, x)$ and all y , where δ is A 's transition function, and q is a nonaccepting state of A ; and
- (3) $(q_A, x) \rightarrow x$ for all accepting states q_A of A .

Here the x s (the vocabulary of A) constitute the terminal vocabulary of G . Thus G generates a string and simulates A 's behavior on it; it generates an all-terminal string σ iff A accepts σ . $\square$

Theorems 8.1.1–8.1.4 tell us that the languages of (arbitrary, context-sensitive, finite-state) grammars are the same as the languages of (Turing, tape-bounded, finite-state) acceptors, respectively. [It can be shown (e.g., [1, pp. 74–78]) that the languages of context-free grammars are the same as the languages of one-way nondeterministic pushdown acceptors, but we shall not prove this here.] Thus introducing these types of grammars has not expanded our language hierarchy.

8.1.3 Isometric Grammars

In this section, we consider grammars in which the left- and right-hand sides of any rule $\lambda \rightarrow \zeta$ have the same length—i.e., $|\lambda| = |\zeta|$. In such “isometric” grammars (called “isotonic” in [2]), strings grow by rewriting $\#s$, and the initial string is $\#^\infty s \#^\infty$. We shall show that such grammars generate the same set of languages as do ordinary grammars, and that by suitably restricting them, we obtain the context-sensitive and finite-state languages. The restriction that, in any rule $\lambda \rightarrow \zeta$, λ and ζ must be of the same size will prove to be important when we define array grammars in Section 8.3.

Formally, an *isometric grammar* is a quintuple $G = (V, V_T, \Pi, s, \#)$, where F, V_T, Π , and s are defined just as in Section 8.1.1, except that, for all pairs $(\lambda, \zeta) \in \Pi$, we require that

- (a) $|\lambda| = |\zeta|$,
- (b) λ does not consist entirely of $\#s$, and
- (c) replacing λ by ζ cannot disconnect or eliminate the non- $\#s$.[§]

The symbol $\# \in V - V_T$ is called the blank symbol. Derivations are defined exactly as in Section 8.1.1. The language of G is the set of all (nonnull) strings σ on V_T such that $\#^\infty \sigma \#^\infty$ can be derived in G from the initial infinite string $\#^\infty s \#^\infty$.

THEOREM 8.1.5 The languages generated by isometric grammars are the same as those generated by ordinary grammars.

Proof: For any grammar G , we can define an isometric grammar G' that generates exactly $L(G)$ as follows:

- (1) For every rule $\lambda \rightarrow \zeta$ of G such that $|\lambda| \leq |\zeta|$, G' has the rule $\lambda \mathfrak{h}^{|\zeta|-|\lambda|} \rightarrow \zeta$, where $\mathfrak{h}$ is a special nonterminal symbol.

[§] It is not hard to see that condition (c) is equivalent to the following: The non- $\#s$ in ζ exist and are connected; if λ has a non- $\#$ at its left (right) end, so does ζ .

- (2) For every rule $\lambda \rightarrow \zeta$ of G such that $|\lambda| \geq |\zeta|$, G' has the rule $\lambda \rightarrow \zeta \#^{|\lambda| - |\zeta|}$.
- (3) In addition, G' has the rules $x\# \rightarrow x\#$, $x\# \rightarrow \#x$, and $\# \# x \rightarrow \# \# x$, for all non- $\#$ x . These rules allow $\#$ s to be changed into $\#$ s at the right end of the string of non- $\#$ s; $\#$ s to be shifted leftward; and $\#$ s to be changed back to $\#$ s at the left end of the string.

It is easy to see that G' can generate a terminal string σ (surrounded by $\#$ s) iff just enough $\#$ s are created, and these $\#$ s are shifted into just the right positions, to allow a derivation of σ in G to be simulated using rules in (1) and (2). (The terminal vocabulary of G' is the same as that of G .) Thus $L(G') = L(G)$.

Conversely, given an isometric grammar G' , we can define G such that, for every rule $\lambda \rightarrow \zeta$ of G' , G has the rule $\lambda' \rightarrow \zeta'$, where λ' and ζ' are the same as λ and ζ but with $\#$ s, if any, omitted. It is not hard to see that G can generate a terminal string σ iff G' generates $\#^\infty \sigma \#^\infty$. (We recall that under G' , the non- $\#$ s remain connected; thus derivations in G differ from those in G' only in that when G' destroys or creates $\#$ s at its ends, G simply grows or shrinks.) $\square$

THEOREM 8.1.6 The languages generated by isometric grammars that never create $\#$ s (i.e., for all rules $\lambda \rightarrow \zeta$, there can be $\#$ s in ζ only in positions corresponding to $\#$ s in λ) are the same as those generated by monotonic grammars.

Proof: In the first part of the proof of Theorem 8.1.5, if G is monotonic, there are no rules of type (2); hence we can omit the rules $\# \# x \rightarrow \# \# x$ and still guarantee that derivations in G can be simulated, by creating and shifting in just enough $\#$ s to allow the rules of type (1) to operate. Thus if G is monotonic, we can define G' so that it never creates $\#$ s. In the second part of the proof, if G' never creates $\#$ s, then for all rules $\lambda \rightarrow \zeta$ of G' we have $|\lambda'| \leq |\zeta'|$, so that G is monotonic. $\square$

There are several ways of defining a class of isometric grammars whose languages are the same as those of the finite-state grammars. It is evident that if the rules of G are all of the form $Z\#^m \rightarrow \zeta$, where $|\zeta| = m + 1$ and ζ is a string of terminals ending in at most one nonterminal, then $L(G)$ is finite-state; and conversely, if G' is finite-state, there is an isometric G of the above form such that $L(G') = L(G)$. We can also establish a stronger result [3]:

THEOREM 8.1.7 The languages generated by isometric grammars in which, for all rules $\lambda \rightarrow \zeta$, λ consists of a single nonterminal symbol together with $\#s$, and ζ consists of non- $\#s$,[§] are the same as the languages generated by finite-state grammars.

Proof: By the preceding paragraph, any language generated by a finite-state grammar is generated by an isometric grammar of the above form. Conversely, suppose that G is of the above form. Note first that if we have generated the string ω in G , and X is a symbol in the interior of ω , then X (if it is nonterminal) can only be rewritten by rules of the form $X \rightarrow Y$, since there are no $\#s$ adjacent to X to serve as context. Now eventually we must be able to turn ω into a string of terminals by applying rules of G ; if we combine these rules with the rules that generated ω , we obtain a set of strings ω' in which X has been replaced by all the terminals to which it can give rise, without changing $L(G)$. Thus we can assume that any string generated by G has nonterminals only at its ends. In particular, we can assume that the right-hand side of any rule of G has nonterminals only at its ends. Similarly, if X is a nonterminal at the right end of such a string (of length > 1), it can only be rewritten by rules of the form $X\#^m \rightarrow \zeta$ (as in the preceding paragraph), and analogously for an X at the left end.

Suppose that the first rule used in a derivation in G has a nonterminal only at one end, say the right. Then subsequent rules must all be of the form $X\#^m \rightarrow \zeta$, as explained above, so that the part of $L(G)$ generated by derivations that begin in this way is finite-state. The argument is analogous if the first rule has a nonterminal only at the left end, since it is easily seen that if we reverse the definition of a finite-state grammar (i.e., every rule right-hand side *begins* with at most one nonterminal), we still obtain exactly the finite-state languages.

If G has rules of the form $s \rightarrow X$, they can be eliminated by letting s go directly into the strings of length > 1 that are derivable from X , or into the singleton terminal symbols that are derivable from X . Note that in the latter case, these rules can contribute only finitely many strings to $L(G)$.

Finally, suppose that the first rule used in a derivation has nonterminal symbols at both ends, and has length > 1 . As we have already seen, the nonterminal at the right end can only be

[§] Note that rules of this form are analogous to *context-free* rules, rather than to finite-state rules.

rewritten by rules of the form $Z\#^m \rightarrow \zeta$, and analogously for the nonterminal at the left end. Thus the strings derivable in this way are essentially concatenations of pairs of strings, each belonging to a finite-state language, and the set of such concatenations is also a finite-state language (Proposition 3.5.8). Combining this with the results of the previous two paragraphs, we see that $L(G)$ is a finite union of finite-state languages, so that by Proposition 3.5.2 it too is finite-state. $\square$

8.1.4 Parallel Grammars

As defined in Sections 8.1.1 and 8.1.2, a grammar generates (or parses) a string by applying only one rule $\lambda \rightarrow \zeta$ at a time and replacing only one instance of λ by ζ when this rule is applied. In this section, we briefly discuss several approaches to defining *parallel grammars*, in which rules are applied in more than one place at a time.

We first consider an approach [2] in which we still apply only a single rule $\lambda \rightarrow \zeta$ at a given time, but apply this rule by replacing *every* instance of λ by ζ . One problem with this approach is that instances of λ can overlap. For example, if we apply the rule $XX \rightarrow ZZ$ to the string XXX , in parallel, the result is (presumably) $ZZZZ$, since XXX contains two instances of XX ; thus application of the isometric rule $XX \rightarrow ZZ$ changes the length of the string. (Indeed, if we apply $XXX \rightarrow ZZ$ to X^m in parallel, we get $Z^{2(m-2)}$, so that applying a length-decreasing rule can increase the string length.) In ordinary ("sequential") rule application, this could not happen; applying the rule $XX \rightarrow ZZ$ to XXX yields either ZZX or XZZ , to which the rule no longer applies.

Another problem with parallel rule application is that parsing and generating are no longer inverses of one another. For example, applying $XX \rightarrow ZZ$ to XXX in parallel yields $ZZZZ$, but applying $ZZ \rightarrow XX$ to $ZZZZ$ in parallel yields $XXXXXX$, since $ZZZZ$ contains three instances of ZZ . In fact (see [2]), it is not hard to exhibit a grammar G such that the sets of strings generated and parsed by G in parallel are disjoint from one another and from $L(G)$.

The *parallel language* $L_P(G)$ generated by a given grammar G is not the same as G 's "sequential language" $L(G)$ in general, but it is the same in certain important cases. For example, consider the finite-state grammars; it is evident that these have the property

that, at any stage of a derivation, at most one nonterminal symbol is present. (Grammars with this property are called "linear.") Since the rules of a finite-state grammar all have left-hand sides consisting of a single nonterminal symbol, it is clear that at any stage of a derivation, a given rule can apply in at most one place; thus it makes no difference whether a rule is applied sequentially (i.e., in one place at a time) or in parallel. In other words, if G is a finite-state (or more generally, a linear) grammar, we have $L_p(G) = L(G)$.

It can also be shown that the *class* of languages generated in parallel by grammars is the same as the class of languages generated in the usual way; and similarly for the classes of languages generated sequentially and in parallel by monotonic grammars. To prove that any (sequential) language is a parallel language, we show that for any grammar G , there exists a grammar G' with $L(G') = L(G)$, such that at any step of a derivation in G' , no rule applies at more than one place. Specifically, for each rule $X_1 \dots X_m \rightarrow Z_1 \dots Z_n$ of G , G' has the rule $\bar{X}_1 \bar{X}_2 \dots \bar{X}_m \rightarrow \bar{Z}_1 \bar{Z}_2 \dots \bar{Z}_n$, where the barred symbols are all nonterminals. We also give G' the rule $s \rightarrow \bar{s}'$ and the set of rules

$$\bar{X}'\bar{Y} \rightarrow \bar{X}\bar{Y}' \quad \text{and} \quad \bar{X}\bar{Y}' \rightarrow \bar{X}'\bar{Y}$$

for all pairs of nonterminals one of which is primed and the other is not. These rules initially create a prime and allow it to shift from symbol to symbol. Since the rules that correspond to those of G all involve the prime, it is clear that any rule applies at only one place (evidently, more than one prime never exists). Finally, we give G' the rules

$$\bar{x}'\bar{y} \rightarrow \bar{x}\bar{y}' \quad \text{and} \quad \bar{x}'\# \rightarrow x\#$$

for all $\bar{x}, \bar{y}$ corresponding to terminal symbols x, y of G (or, if we do not want to introduce the $\#$ symbol, we design G' so that the rightmost symbol of any string derivable in it is always uniquely marked). These last rules allow the prime to shift rightward through symbols that correspond to terminals, turning them into terminals as it goes, until it reaches the right end of the string and vanishes. It is evident that this process can result in a terminal string iff G' generated a string of symbols corresponding to terminals, by imitating a derivation in G that leads to a string of terminals; thus the resulting terminal string must be in $L(G)$. (If the last rules are used too soon, or are not started with the prime at the left

end of the string, a string consisting entirely of terminals will not be created.) Since in G' , no rule can apply in more than one place, we have $L_p(G') = L(G')$; thus $L(G) = L(G') = L_p(G')$, which proves that any language is a parallel language. Note also that if G is monotonic, so is G' .

Conversely, we can show that for any grammar G there exists a grammar G'' that, in effect, applies the rules of G in parallel; thus $L_p(G) = L(G'')$, so that any parallel language is a sequential language. Basically, we define G'' to simulate an automaton A (compare the proof of Theorem 8.1.1). Suppose that, at a given stage in the operation of A , we are ready to apply a rule of G in parallel to the current string σ (initially, this string is s). A picks a rule $\lambda \rightarrow \zeta$, scans σ , and marks every position in σ at which a match to λ begins. It then scans σ again and replaces each instance of λ by ζ . (If instances overlap, the part of σ from the start of one instance of λ to the start of the next is replaced by ζ .) When this process is complete, we are ready to apply another rule of G . It is not hard to see that if G is monotonic, so is G'' . (A detailed description of G'' can be found in [2].)

The remarks in the preceding three paragraphs show that any (arbitrary, context-sensitive, finite-state) language is an (arbitrary, context-sensitive, finite-state) parallel language, and conversely. The analogous result about context-free languages is false. For example, consider the context-free grammar whose rules are

$$s \rightarrow ss \quad \text{and} \quad s \rightarrow x.$$

It is not hard to see that the parallel language of this grammar is the set of strings $\{x^{2^n} \mid n = 0, 1, 2, \dots\}$; but it is well known that this is not a context-free language in the ordinary sense (see, e.g., [1, p. 57, Theorem 4.7]). Conversely, it can be shown that the parenthesis-string language, which is context-free [1, p. 67], is not parallel context-free. On the relationship between context-free and parallel context-free languages, see [4, 5].

Parallel rule application has a particularly convenient interpretation for rules that are in the context-sensitive form $\xi X \eta \rightarrow \xi \omega \eta$: For each instance of $\xi X \eta$ in the given string σ , we replace the X by an ω .[§] In this case, some of the problems mentioned at the begin-

[§] Note that this is not the same as replacing each instance of $\xi X \eta$ by $\xi \omega \eta$; for the rule $XX \rightarrow XZ$, applied to the string XXX , replacing XX s by XZ s gives $XZXZ$, whereas replacing X s by Z s when they have X s on their left gives XZZ .

ning of this section do not arise, since the substrings being replaced (i.e., the X s) cannot overlap, even if the $\xi X \eta$ s do overlap. Parsing and generation are still not always inverses of one another; e.g., if we apply $XX \rightarrow XZ$ to XXX , we obtain XZZ , whereas if we apply $XZ \rightarrow XX$ to XZZ , we obtain XXZ . On the other hand, string length can never decrease, since X s are being replaced by nonnull strings (ω s). The results in the preceding paragraphs all continue to hold for this modified concept of parallel rule application.

Another approach to defining parallelism for context-sensitive grammars is to apply rules in all possible positions at the same time. In other words, given a string $\sigma = X_1 \dots X_n$, for each X_i we choose a rule $\xi_i X_i \eta_i \rightarrow \xi_i \omega_i \eta_i$ of G that applies to X_i in σ (i.e., such that ξ_i precedes X_i in σ , and η_i follows it) and replace X_i by ω_i . (If no such rule exists, we leave X_i unchanged.) This is done in parallel for every symbol in σ ; the rule used is chosen independently for each X_i . Parallel rule-application systems of this kind are called *L-systems* [6–8]; if the rules are all context free, they are called *0L-systems*. L-systems have been extensively studied as models of biological growth; they will not be discussed here in detail.

In defining L-systems, it is customary to make no distinction between terminal and nonterminal vocabularies; the language is the set of all strings that can be derived from the initial string. Another possibility is to define the language as the set of all such strings that are stable under application of the rules; such strings correspond, from the biological growth standpoint, to “adult” organisms. It is not hard to show [9] that the “adult languages” of L-systems are just the context-sensitive languages. In fact, given an L-system G and a string σ , we can easily define a tape-bounded acceptor A that accepts σ iff it is in the adult language of G . [A first checks that σ is stable under G . It then nondeterministically generates a sequence of strings $\sigma_1, \sigma_2, \dots$ by applying the rules of G to the initial string σ_0 , and checks at each step whether σ_i matches σ . Since we have assumed that the rules of G are context-sensitive (i.e., the ω_i s are nonnull), we know that $|\sigma_0| \leq |\sigma_1| \leq |\sigma_2| \leq \dots$; thus if σ is derivable from σ_0 in G , no σ_i can be longer than σ , so A has room to store it, as in the proof of Theorem 8.1.3. Thus if σ is derivable in G , A can discover this fact and accept σ .] Conversely, given a context-sensitive grammar G' , we can define a modified G'' such that at any stage of a derivation in G'' ,

there is only one place where any rule applies, as shown earlier in this section. We can thus define an L-system G that simulates G'' . [Note that in the definition of G'' , as long as the prime remains, there is always some rule that applies; but the prime disappears only when all the symbols have been turned into terminals.[§]] Thus the only strings that are stable under the rules of G are the terminal strings of G'' , which are the same as the terminal strings of G' ; hence the adult language of G is just $L(G')$. If we modify the definition of an L-system to allow erasing rules (i.e., $\xi X \eta \rightarrow \xi \eta$), it can be shown that the adult languages are just the arbitrary (Turing-machine) languages (but allowing the null string to be in a language); the proof is analogous. (L-systems that do not allow erasing rules are called "propagating.") Similarly, it can be shown that the adult languages of 0L-systems are just the context-free languages. For the details of these proofs, see [9].

In analogy with Section 8.1.3, one can consider "isometric" parallel grammars (or L-systems) in which the rules are all of the form $\xi X \eta \rightarrow \xi Z \eta$, so that strings can grow only by rewriting $\#$ s at their ends. L-systems of this special type are essentially nondeterministic cellular automata, in which the transition function is defined by the rewriting rules: State X , in the context of states ξ on the left and η on the right, can change into state Z . If $\#$ s are never rewritten, they become bounded cellular automata. Note that from this point of view, we are regarding cellular automata as pattern generators rather than as acceptors.

8.2 MATRIX GRAMMARS

In this section, we discuss grammars that generate or parse sets of rectangular arrays. Grammars whose languages are sets of connected (not necessarily rectangular) arrays will be considered in Section 8.3.

A class of "matrix grammars" [10–12] that generate rectangular arrays can be informally defined as follows:

- (a) A string grammar G generates a string σ , which will become the top row of the array.

[§] This requires a slight modification of the definition of G'' so that the prime can only begin creating terminals when it is at the left end of a string; the details are straightforward.

- (b) The symbols in σ are initial symbols of a set of finite-state string grammars $G_1, \dots, G_n$. These grammars operate in parallel (compare Section 8.1.4) to generate the columns of the array. Their operation must be coordinated so that in every column, at any given time, a rule of the same length is applied, and so that the terminating rules are all applied at the same time.[§]

Formally, a *matrix grammar* M is a pair $(G, \bar{G})$, where G is a grammar and $\bar{G} = \{G_1, \dots, G_n\}$ is a set of finite-state grammars, such that the terminal vocabulary of G is the set $\{s_1, \dots, s_n\}$ of initial symbols of the G_i s. (We assume that the nonterminal vocabularies of the G_i s are disjoint.) M operates by generating a (horizontal) string σ of s_i s using the rules of G , then generating a rectangular array from the top row σ by applying rules of the G_i s in parallel, and finally terminating simultaneously.

Matrix grammars are of interest because they generate (or parse) rectangular arrays, but unfortunately the classes of array languages that they generate are not the same as the classes accepted by the various types of array acceptors. To see this, let us consider the following examples:

- (a) If G is finite-state, the language L of M is a finite-state array language. Indeed, we can define a finite-state array acceptor A that accepts L as follows: Given a rectangular array T , A scans the last column c_m of T , simulates acceptors for the finite-state column languages, and thus determines to which of these languages, if any, c_m belongs. A can thus verify whether a terminating rule of G could have produced any of the initial symbols s_m that began the generation of c_m . A then scans the next-to-last column c_{m-1} , determines which symbols s_{m-1} could have begun its generation, and verifies whether a rule of G could have produced any of these symbols together with a nonterminal that then resulted in s_m . This process is repeated until it has accounted for all of T . Conversely, however, there are (deterministic, tape-bounded) finite-state rectangular array languages that cannot be generated by any such M . For ex-

[§] We can assume without loss of generality that the rules of each G_i are all of the forms $X \rightarrow xZ$ and $X \rightarrow x$, where X, Z are nonterminals and x is a terminal. Under this assumption, it suffices to require that the terminating rules be applied in every column at the same time.

ample, consider the set L of square arrays of even side length whose rows are all of the form $x^n y^n$. No such M can generate L , since the string language $\{x^n y^n | n = 1, 2, \dots\}$ is not finite-state. On the other hand, a finite-state array acceptor can verify that its input array T is in L as follows: Verify squareness (and evenness of side length) by moving "diagonally;" then move at " $67\frac{1}{2}$ " (alternately two steps upward and one leftward, starting at the lower right corner) to find a middle column; finally, move down that column and verify that, in each row, there are x s on its left and y s on its right. Thus the class of matrix languages for which G is finite-state is a proper subset of the class of finite-state array languages. It is not hard to see that we can require A in the first part of the proof to be deterministic and tape-bounded; thus these matrix languages are a proper subset of the deterministic, tape-bounded finite-state array languages.

- (b) If G is context-free, the language of M is not finite-state. For example, the set L of rectangular arrays of even width whose rows are all of the form $x^n y^n$ can be generated by such an M (G generates $s_1^n s_2^n$; the rules of G_1 and G_2 are $s_1 \rightarrow xs_1$, $s_1 \rightarrow x$, and those of G_2 are $s_2 \rightarrow ys_2$, $s_2 \rightarrow y$), but L cannot be accepted by a finite-state array acceptor, since $\{x^n y^n | i = 1, 2, \dots\}$ is not a finite-state string language (consider the arrays of height 1!). Thus the matrix languages in general (even for context-free G s) are not a subset of the finite-state array languages. Conversely, there are (deterministic, tape-bounded) finite-state array languages that are not matrix languages, for any choice of G . For example, the set L consisting of all square arrays is not a matrix language, since the termination decision in generating the columns does not depend on the array width. (The step numbers at which column termination occurs depend on the set of s_i s that initiated the columns; there are only finitely many such sets and they cannot carry enough information to provide for a different termination step for every top row width.) Thus the matrix languages and the finite-state array languages are incomparable.
- (c) The matrix languages for which G is context-sensitive are a proper subset of the tape-bounded array languages. For ex-

ample, the arrays whose columns are all of the form $x^n y^n$ are a tape-bounded array language, but are not a matrix language (since $\{x^n y^n \mid n = 1, 2, \dots\}$ is not a finite-state string language). Conversely, a tape-bounded acceptor can scan the columns of a rectangular array, determine which s_i s could have initiated their generation, record these (sets of) s_i s in the top row, and then simulate a one-dimensional tape-bounded acceptor to verify whether some such set of s_i s could have been generated by G ; thus any such matrix language is a tape-bounded array language. On the other hand, there are matrix languages (having non-context-sensitive top rows and height 1) that are not tape-bounded array languages; thus the tape-bounded array languages and the matrix languages are incomparable. Finally, the matrix languages are a proper subset of the Turing rectangular array languages, by arguments similar to those already given.

It can be shown [13] that the languages of matrix grammars for which G is context-sensitive are the same as the languages of a special class of parallel/sequential array acceptors (Section 7.4)—namely, the class in which the cells (of the one-dimensional cellular acceptor) do not communicate with their neighbors until the acceptor has reached the last row of its input array. Other types of acceptors that are equivalent to matrix grammars are discussed in [9, 14].

A number of other grammarlike mechanisms can be used to generate rectangular array languages. In [11, 12], for example, a grammar generates a string that specifies a sequence of horizontal and vertical concatenations to be performed on rectangular arrays. It is understood that when two arrays are horizontally (vertically) concatenated, they must have the same numbers of rows (columns); hence any such sequence of concatenation operations generates a rectangular array. Another way to insure rectangularity is to require that all border symbols of a given type (north, south, east, or west) be rewritten simultaneously; note that the column generation process in a matrix grammar is a special case of this (rewriting all south border symbols at the same time). Such models have been studied in [15]. We shall not consider these approaches further here.

8.3 ARRAY GRAMMARS

This section introduces grammars that generate or parse sets of connected arrays, and investigates their relationship to array acceptors. As we shall see, several problems arise with array grammars that did not arise for string grammars.

An array grammar must operate by replacing subarrays by subarrays, just as a string grammar replaced substrings by substrings. However, if the two subarrays (Λ and Ξ , say) are not identical in size and shape, it is not clear how to replace Λ by Ξ . One can presumably shift the rows or columns of the host array relative to one another so as to make room for Ξ ; but this may cause changes in symbol adjacencies arbitrarily far away from Ξ , since adjacent rows or columns may shift by different amounts. Thus applying a local rewriting rule $\Lambda \rightarrow \Xi$ may cause nonlocal changes in the host array, which seems undesirable. To avoid this problem, we shall require array grammars to be isometric (see Section 8.1.3); this means that for any rule $\Lambda \rightarrow \Xi$, Λ and Ξ are geometrically identical, and the array grows (or shrinks) by rewriting (or creating) $\#$ s. For a treatment of nonisometric array grammars, see [16].

A second problem with array grammars is how to insure that they preserve connectedness (and nonemptiness) of the set of non- $\#$ s. Before formulating conditions for this, we first observe that in any rule $\Lambda \rightarrow \Xi$, Λ must contain non- $\#$ s, but they need not be connected. (In one dimension, as long as the non- $\#$ s remain connected, a rule in which λ has nonconnected non- $\#$ s could never apply; but in two dimensions, a globally connected array can contain locally nonconnected parts.) On the other hand, if Λ has more than one connected component of non- $\#$ s, every such component must touch the border of Λ . Our conditions on Ξ are then as follows:

- (1) If the non- $\#$ s of Λ do not touch the border of Λ , then the non- $\#$ s of Ξ must be connected (and nonempty).
- (2) Otherwise,
 - (a) every connected component of non- $\#$ s in Ξ must contain the intersection of some component of non- $\#$ s in Λ with the border of Λ ;
 - (b) conversely, every such intersection must be contained in some component of non- $\#$ s in Ξ .

PROPOSITION 8.3.1 If conditions (1) and (2) hold, applying the rule $\Lambda \rightarrow \Xi$ does not disconnect or eliminate the non- $\#$ s.

Proof: We assume that non- $\#$ s existed and were connected before $\Lambda \rightarrow \Xi$ was applied. In case (1), the non- $\#$ s of Λ must be the only non- $\#$ s that exist (there cannot be any outside Λ , since they would have to be connected to those in Λ , which would require that non- $\#$ s exist on the border of Λ); hence when we replace Λ by Ξ , the non- $\#$ s of Ξ are the only ones that exist, and they are connected by (1).

In case (2), we first show that any two non- $\#$ s P, Q not in Λ remain connected when Λ is replaced by Ξ . If there is a path of non- $\#$ s from P to Q that does not meet Λ , this is clear. Otherwise, note that each time the path enters and leaves Λ , it meets only one component of non- $\#$ s in Λ , and by (2b) the intersection of this component with the border of Λ is contained in a component of non- $\#$ s in Ξ ; hence each such segment of the path can be replaced by a segment in Ξ .

Similarly, if P is not in Λ and Q is in Ξ , then by (2a), the component of non- $\#$ s in Ξ that contains Q also contains the intersection of some component of non- $\#$ s in Λ with the border of Λ . Now P was connected (outside Λ) to this component and hence to the intersection; thus P is connected to Q .

Finally, if P and Q are both in Ξ , and are not connected, they are both in components of non- $\#$ s that, by (2a), contain intersections of components of non- $\#$ s in Λ with the border of Λ . But these components were connected outside Λ ; hence so were these intersections. $\square$

It is clear that if (1) did not hold, applying $\Lambda \rightarrow \Xi$ would disconnect (or annihilate) the non- $\#$ s. If (2a) did not hold, a component of non- $\#$ s in Ξ might be created that failed to be connected to the other non- $\#$ s; while if (2b) did not hold, some non- $\#$ s outside Ξ might be disconnected from the other non- $\#$ s.

8.3.1 Array Grammars and Languages

Formally, an (isometric) *array grammar* is a quintuple $G = (V, V_T, \Pi, s, \#)$, where all of these are defined just as in Section 8.1.2, except that Π is a set of pairs of connected arrays (Λ, Ξ) , for all of which

- (a) Λ and Ξ are geometrically identical,
- (b) Λ does not consist entirely of $\#$ s, and
- (c) Ξ satisfies conditions (1) and (2) of Proposition 8.3.1.

The *language* of G , $L(G)$, is the set of all (nonnull) connected arrays Σ on V_T such that Σ (embedded in an infinite array of $\#$ s) can be derived in G from the initial array, which consists of a single s (embedded in an infinite array of $\#$ s). Derivations are defined exactly as in Section 8.1, except that they involve replacement of one subarray by another, rather than of one substring by another. Parsing is also defined just as in Section 8.1. Note that by Proposition 8.3.1 and induction, the array resulting from any derivation always has its non- $\#$ s connected.

As in Section 8.1.1, we can (and shall from now on) assume that terminal symbols are never rewritten by any rule of G .

An array grammar is called *monotonic* if $\#$ s are never created by any rule—in other words, for all rules $\Lambda \rightarrow \Xi$ of G , there are $\#$ s in Ξ only in positions corresponding to $\#$ s in Λ . We can also define an array grammar G to be “context free” if, for all rules $\Lambda \rightarrow \Xi$ of G , Λ consists of a single nonterminal symbol and (possibly) of $\#$ s; note that such rules do use $\#$ s as “context.” Similarly, we can call G *linear* if G is “context-free” and, in addition, any Ξ contains at most one nonterminal symbol; note that at any stage (except the last) of a derivation in such a G , exactly one nonterminal exists. As we shall see later, there does not seem to be a good analog of “finite-state” for array grammars.

In some of the examples to be given later, it will be convenient to use compact notation for sets of array rewriting rules that differ by 90° rotations. We shall denote by $\Lambda \rightarrow \Xi$ (ROT) the set of four rules in which Λ and Ξ are both rotated by 0°, 90°, 180°, and 270°. For example,

$$XY \rightarrow WZ \text{ (ROT)}$$

denotes the four rules

$$\begin{array}{ll} XY \rightarrow WZ & \begin{array}{l} Y \rightarrow Z \\ X \rightarrow W \end{array} \\ YX \rightarrow ZW & \begin{array}{l} X \rightarrow W \\ Y \rightarrow Z \end{array} \end{array}$$

Alternatively, we may denote by $X(d)Y$ the pair of adjacent symbols such that Y is in direction d (U, D, L, or R) from X ; thus

$X(R)Y \equiv XY$, $X(L)Y \equiv YX$, $X(U)Y \equiv \frac{Y}{X}$, and $X(D)Y \equiv \frac{X}{Y}$. In this notation, we may write

$$X(d)Y \rightarrow W(d)Z \quad \text{for all } d \in \Delta \equiv \{U, D, L, R\}$$

instead of $XY \rightarrow WZ$ (ROT).

8.3.2 Equivalence to Array Acceptors

Our goal in this section is to prove that (arbitrary, monotonic) array grammars generate the same classes of languages as (Turing, tape-bounded) array acceptors. The proofs are based on [17]; see also [18], which proves an analog of Theorem 8.3.3 using a somewhat different definition of an array grammar. We also discuss the relationship between finite-state array grammars and array acceptors; this discussion is based on [19].

THEOREM 8.3.2 The languages generated by array grammars are the same as the languages accepted by Turing array acceptors.

Proof: Given an array grammar G , we define a TA A that accepts exactly $L(G)$ as follows: Given an array Σ of non- $\#$ s in the terminal vocabulary V_T of G , embedded in an infinite array of $\#$ s, A moves around nondeterministically, starting at some point of Σ , and rewrites each symbol x (possibly $\#$) that it comes to as $(x, \#)$.[§] Finally, at some step, A rewrites some x as (x, s) ; this can only happen once. After it happens, A begins to simulate rules of G on the second terms of pairs; pairs with no second term are regarded as having second term $\#$. Suppose that at some step, A decides to apply the rule $\Lambda \rightarrow \Xi$ of G . It then searches (nondeterministically) for one of the non- $\#$ symbols in Λ (as a second term), marks it, checks (deterministically) for the other symbols as second terms (these are in known positions relative to the given symbol), marks them, and when they have all been found, rewrites these second terms as the corresponding symbols in Ξ ; after this is done, A is ready to simulate another rule. [While searching, A continues to rewrite xs as $(x, \#)s$.] At any stage after applying a rule of G , A can systematically scan the array of non- $\#$ s (Proposition 4.4.1) and

[§] This is done in order to insure that A is always on a connected set of non- $\#$ s; if A moved onto the $\#$ s without rewriting them as pairs, it might be unable to find Σ again.

check whether (a) every non- $\#$ is a pair and (b) the second term of every pair is the same as its first term. [A can also check, during some such scan, that the non- $\#$ first terms are in fact all in V_T .] If this is found to be true, A accepts Σ ; otherwise, A can resume the simulation of G . Evidently, A accepts Σ iff a derivation in G exists that generates a copy of Σ ; thus A accepts exactly $L(G)$ (from some starting point).

Conversely, given A , we define a grammar G that generates exactly $L(A)$ as follows: Starting with an s on a background of $\#$ s, G begins by generating an arbitrary array Σ of triples of the form $(x, x, 0)$ or $(x, x, 1)$, where x is any non- $\#$ symbol in the vocabulary V of A , and where there is always exactly one 1. This can be done using the rules

- (1a) $s \rightarrow (x, x, 1)$,
- (1b) $(x, x, 1)\# \rightarrow (x, x, 0)(y, y, 1)$ (ROT),
- (1c) $(x, x, 1)(y, y, 0) \rightarrow (x, x, 0)(y, y, 1)$ (ROT), and
- (1d) $(x, x, 1) \rightarrow (x, x, (q_0, N))$

for all $x, y \neq \#$ in V . When (1d) is used, the 1 turns into a pair (q_0, N) representing the initial state and (fictitious) "previous move direction" of A .

Next, G simulates the operation of A on the array of second terms of the triples; the first terms remain unaffected. This is done using the rules (for all x, u, v in V)

- (2a) $(x, y, (q, d)) \rightarrow (x, z, (q', N))$ for all $(q', z, N) \in \delta(q, y, d)$;
- (2b) $(x, y, (q, d))(d')(u, v, 0) \rightarrow (x, z, 0)(d')(u, v, (q', d'))$ for all $(q', z, d') \in \delta(q, y, d)$; and
- (2c) $(x, y, (q, d))(d')\# \rightarrow (x, z, 0)(d')(\#, \#, (q', d'))$ (similarly),

where δ is the transition function of A , and provided that q is not an accepting state of A .

Finally, if the simulation of A enters an accepting state, the triples are turned into their first terms using the rules

- (3a) $(x, y, (q, d)) \rightarrow x$ for all $x, y \in V$ and all $q \in Q_A$, and
- (3b) $(u, v, 0)x \rightarrow ux$ (ROT) for all $u, v, x \in V$.

Since $V - \{\#\}$ is the terminal vocabulary of G , this means that G generates a terminal array of non- $\#$ s (the first terms of Σ) iff A accepts this array (from some starting point). $\square$

THEOREM 8.3.3 The languages generated by monotonic array grammars are the same as the languages accepted by tape-bounded array acceptors.

Proof: In the first part of the proof of Theorem 8.3.2, note that at any step of a derivation of Σ in the monotonic array grammar G , the non- $\#$ s are always a subset of Σ , since $\#$ s can never be created by G . Thus, we can simulate G using a tape-bounded A that always remains on Σ . [A problem arises with this simulation if a rule $\Lambda \rightarrow \Xi$ of G uses $\#$ s as context (i.e., there are $\#$ s in Λ that are also in the corresponding positions in Ξ); these $\#$ s may be outside Σ , but A must verify their presence in order to apply the rule. This can be done using the array scanning process described just prior to Proposition 4.4.6. Suppose that A is located at position (i, j) and wants to verify that there is a $\#$ in position (i', j') , where the differences $i - i'$ and $j - j'$ are known to A . To do this, A marks (i, j) , finds the outer border of Σ , and scans Σ row by row. When A reaches the row with the marked point, it returns to the outer border and follows it, using the positions of markers to keep track of its net up and down moves. If the outer border never reaches row i' , Σ cannot intersect that row, so that position (i', j') is certainly $\#$. Otherwise, A marks the points of the outer border that are on row i' and scans these rows of Σ , marking their points. Analogously, using a column-by-column scan, A finds the marked point and checks whether the outer border reaches column j' . If not, (i', j') is $\#$; if so, A marks the points of Σ that are on column j' . If these marks ever encounter the marks on row i' , (i', j') is not $\#$; otherwise, it is $\#$, and A can rescan Σ , erase the marks on row i' and column j' , rescan Σ , and return to the marked point (i, j) . In this way, A can verify that $\#$ s exist in given positions relative to some non- $\#$ point of Λ , and thus check that all of Λ is present, without having to leave Σ . Note that a similar process may be necessary for the rewriting of Λ , since the non- $\#$ s of Λ may be separated by $\#$ s, and A may have to compute the positions of these non- $\#$ s in order to locate them and rewrite them.] After any step of the simulation, A can scan Σ and check that the first and second terms are all the same (and all in V_T). If so, A accepts; otherwise, it continues the simulation. Thus A accepts exactly $L(G)$, as in the proof of Theorem 8.3.2.

In the second part of the proof, note that the only rules of G that can create $\#$ s are the terminating rules (3a) and (3b); and these

create $\#$ s only if triples with $\#$ first terms were created by the simulation rule (2c). But if A is tape-bounded, we can modify (2c) to have the form

$$(x, y, (q, d))(d')\# \rightarrow (x, z, (q'', d'^{-1}))(d')\#$$

where q'' is a possible state of A after it has bounced back off the $\#$. Thus when A is tape-bounded, G need never create $\#$ s, so that G can be monotonic. $\square$

It does not seem possible to define a natural class of array grammars whose languages are the same as those of the finite-state array acceptors. (A generator model that generates exactly the FSA languages is described in [20]. However, in this model, every time a given symbol is rewritten, it must be rewritten in the same way, even though the generation process cannot tell whether or not this is happening; thus this model requires some means, outside the generator itself, to reject arrays in which a rewriting inconsistency occurs.) Of course, one can always define a class of array grammars that are capable only of simulating FSAs [21]; but this is artificial.

In one dimension, the linear grammars (in which, at any step of a derivation except the last step, exactly one nonterminal symbol exists) generate more than the finite-state languages (e.g., $\{x^n y^n \mid n = 1, 2, \dots\}$ is a linear string language, having the grammar whose rules are $s \rightarrow sxy$, $s \rightarrow xy$); whereas the "right linear" grammars, in which (until the last step) the sole nonterminal symbol is always at the right end of the string, generate exactly the finite-state languages (Theorem 8.1.4). In two dimensions, there is no natural analog of "right linear." Moreover, we shall now show that the languages of linear array grammars are incomparable with the FSA languages.

We first exhibit an FSA language that cannot be generated by a linear array grammar. Consider the set of thin upright T-shaped arrays of x s on a background of $\#$ s. This set is accepted by a deterministic tape-bounded FSA that operates as follows: It moves up until it hits a $\#$, then moves left until it hits a $\#$ again. It then verifies that it is on a row of x s that have $\#$ s above and below them, with exactly one exception that has an x below it and that is not at either end of the row. Finally, it moves back to the exceptional x and moves downward, verifying that it is on a column of x s that

have $\#$ s on their left and right. When it reaches the bottom of this column, if all these verifications have been carried out successfully, it accepts.

We now show that the set of such Ts cannot be generated by a linear array grammar. Suppose G were such a grammar; let the greatest diameter of any right-hand side of a rule of G be k . Since the Ts can be arbitrarily large, it is clear that G must generate the ends of the arms of a large T at different steps in the derivation of that T. Let these arm ends be P_1 , P_2 , and P_3 . One of them, say P_1 , can be generated before the point P at the T junction is generated, but P_2 and P_3 must be generated afterwards. After P is generated, if the nonterminal moves sufficiently far ($> k$) down the P_2 arm, it can never generate the P_3 arm, since it cannot return down the P_2 arm (which now consists of terminals) or cross the $\#$ s (which, once rewritten, would destroy the shape of the desired thin T).

Conversely, we can exhibit a language that is generated by a linear array grammar but is not accepted by a tape-bounded FSA. Let G be the array grammar whose rules are

$$\begin{array}{ll} s\# \rightarrow xs, & s\# \rightarrow xW; \\ W \xrightarrow{x} & W \xrightarrow{x} \\ \# \rightarrow W', & \# \rightarrow X'; \\ \#X \rightarrow Xx, & \#X \rightarrow Yx; \\ \# \xrightarrow{Y} & Y \rightarrow x. \\ Y \rightarrow x', & \end{array}$$

Informally, the S moves to the right, leaving a trail of xs , until it changes to a W ; the W moves down, trailing xs , until it changes to an X ; the X moves left, leaving xs behind it, until it changes to a Y ; and the Y moves up, trailing xs , until it changes to an x . Thus the language of this grammar consists of four-sided upright rectangular arcs, composed of xs , that may touch themselves, but cannot cross themselves. In particular, all the hollow, thin upright rectangles of xs are in this language. But as we have seen (in the proof of Theorem 4.4.3), any tape-bounded FSA that accepts all such rectangles must also accept various sufficiently large rectangular spirals (having many more than four sides). Hence the language of G is not a tape-bounded FSA language.

Another possibility, discussed in [19, 21], is to allow a linear grammar to rewrite terminal symbols, but not to sense what symbols are present; for example, in a rule $\Lambda \rightarrow \Xi$, Λ would con-

sist of a single nonterminal Z together with a set of “don’t-care” symbols, and in applying the rule, we would rewrite whatever symbols lie in those positions relative to Z . (If we do allow the grammar to sense which symbols are present, it is at least as strong as an ordinary linear array grammar and hence can generate a class of patterns that is not an FSA language, as in the preceding paragraph.) If we do this, it becomes possible to generate thin T -shaped patterns (possibly including degenerate T s with arms missing); we omit the details here. On the other hand, such a “blind” grammar could not generate the set of arcs (composed of x s) that do not cross or touch themselves, since the nonterminal leaving the trail of x s could not know when it was about to cross its own path; but this set is evidently an FSA language (the FSA verifies that all but two of the non- $\#$ s have exactly two non- $\#$ neighbors). Conversely, a “blind” grammar can generate the set of all connected arrays of x s; but this is not an FSA language. [*Proof:* It contains the set of all thin, hollow rectangles of x s; but if A accepts such a rectangle, it also accepts a sufficiently large rectangular spiral without visiting its ends, so that it cannot tell whether or not there are symbols other than x at the ends.]

An acceptor model that accepts the languages generated by linear array grammars is described in [3]. However, in this model, acceptance requires not only entering an accepting state, but also marking the entire input array, even though the acceptor cannot tell whether this has been done; thus this model requires some means, outside the acceptor itself, to reject arrays that have not been completely marked.

8.3.3 Parallel Array Grammars

In Section 8.1.4, we introduced the idea of parallel rule application (rewrite all instances of Λ as Ξ s, rather than just a single instance), and showed that the classes of languages generated in parallel by (arbitrary, context-sensitive) grammars are the same as the classes of languages generated by those types of grammars in the ordinary way. The proofs apply also to isometric grammars; in fact, in the proof that any sequential language is a parallel language, if G is isometric, so is G' , while in the proof that any parallel language is a sequential language, we know that an isometric G'' can simulate an automaton of the desired type that applies the rules of G .

Analogous results can be established for (isometric) array grammars; the details are straightforward [22]. One can also define (isometric) array L-systems (the details will be omitted here); as pointed out in Section 8.1.4, they are essentially nondeterministic cellular automata.

REFERENCES

1. J. E. Hopcroft and J. D. Ullman, "Formal Languages and Their Relation to Automata." Addison-Wesley, Reading, Massachusetts, 1969.
2. A. Rosenfeld, Isotonic grammars, parallel grammars, and picture grammars, in "Machine Intelligence 6" (B. Meltzer and D. Michie, eds.), pp. 281–294 Edinburgh Univ. Press, 1971.
3. C. R. Cook and P. S. P. Wang, A Chomsky hierarchy of isotonic array grammars and languages, *Comput. Graphics and Image Processing* (in press).
4. R. Siromoney and K. Krithivasan, Parallel context-free languages, *Information and Control* **24**, 155–162 (1974).
5. S. Skyum, Parallel context-free languages, *Information and Control* **26**, 280–285 (1974).
6. G. T. Herman and G. Rozenberg, "Developmental Systems and Languages." North-Holland Publ., Amsterdam, 1974.
7. G. Rozenberg and A. Salomaa, "L-Systems." Springer-Verlag, Berlin and New York, 1974.
8. A. Lindenmayer and G. Rozenberg, eds., "Automata, Languages, Development." North-Holland Publ., Amsterdam, 1976.
9. A. Walker, Dynamically stable strings in developmental systems, *J. Comput. System Sci.* **11**, 252–261 (1975).
10. G. Siromoney, R. Siromoney, and K. Krithivasan, Abstract families of matrices and pictures languages, *Comput. Graphics and Image Processing* **1**, 284–307 (1972).
11. G. Siromoney, R. Siromoney, and K. Krithivasan, Picture languages with array rewriting rules, *Information and Control* **22**, 447–470 (1973).
12. G. Siromoney, R. Siromoney, and K. Krithivasan, Array grammars and Kolam, *Comput. Graphics and Image Processing* **3**, 63–82 (1974).
13. E. D. Feldman, Matrix grammars and parallel-sequential array automata, Tech. Rep. No. 250, Computer Science Center, Univ. of Maryland, College Park, Maryland, 1973.
14. P. S. P. Wang, Recognition of two-dimensional patterns. *Proc. Assoc. Comput. Mach. Nat. Conf.* pp. 484–489 (1977).
15. R. Siromoney and G. Siromoney, Extended controlled table L-arrays, *Information and Control* **35**, 119–138 (1977).
16. P. A. Ota, Mosaic grammars, *Pattern Recognition* **7**, 61–65 (1975).
17. D. L. Milgram and A. Rosenfeld, Array automata and array grammars, "Information Processing 71," pp. 69–74. North-Holland Publ., Amsterdam, 1972.
18. E. Yodokawa and N. Honda, On 2-dimensional pattern-generating grammars, *Systems-Computers-Control* **1**, 6–13 (1970).

19. A. Rosenfeld, Some notes on finite-state picture languages, *Information and Control* **31**, 177–184 (1976).
20. J. Mylopoulos, On the application of formal language and automata theory to pattern recognition, *Pattern Recognition* **4**, 37–51 (1972).
21. A. Rosenfeld and D. L. Milgram, Array automata and array grammars, 2, Tech. Rep. No. 171, Computer Science Center, Univ. of Maryland, College Park, Maryland, 1971.
22. A. Rosenfeld, Array grammar normal forms, *Information and Control* **23**, 173–182 (1973).

INDEX

A

Acceptor, 3.2.3
 array, 4.2
 cellular, *see* Cellular acceptor
 one-way, 3.3.3
 parallel/sequential, 7.4
 pebble, 7.2
 pushdown, 7.3
 three-way, 4.3.3, 4.4.3
 tree, 6.4.2
 Adjacency graph, 2.6
 Adjacent, 2.2
 Arc, 2.3, 2.9
 Area, 2.7
 Automaton, 3.2.1
 array, 4.2
 cellular, *see* Cellular acceptor

B

Background, 2.2
 BF (border-following algorithm), 2.6
 Blank (symbol or state, $\#$), 3.2.2, 4.2, 3.4.1, 8.1.3
 Border (following), 2.6

C

Cellular acceptor (automaton), 3.4.1, 5.2.2
 bottom-up, 6.2.3
 bounded, 3.4.1, 5.2.2
 neighborhood, 5.2.1

 pyramid, 6.2.2
 triangle, 6.2.1

Chord, 2.9

Closure (concatenative), 3.5, 4.5

Closure properties (of languages), 3.5, 4.5, 6.5.1

Complement, 3.5, 4.5, 6.2.5, 6.5.1

Concatenation, 3.5, 4.5, 6.5.1

Configuration, 3.2.1, 3.4.1, 4.2, 4.5

Connected (component), 2.2

Connectedness, 5.5.1, 5.5.2, 6.5.1, 7.2.2, 7.2.3, 7.4

Convexity, 2.10

Counting (equality, majority), 3.4.3, 5.5.1, 5.5.2, 6.5.1

Curve, 2.3

Cyclic closure, 3.5, 4.5

D

Derivation, 8.1.1

Deterministic, 3.2.1, 3.4.1, 4.2

Diameter, 2.8

Digital picture, 2.1

Distance, 2.8

 chessboard (d_8), 2.8

 city block (d_4), 2.8

 intrinsic (d_s), 2.8

E

Elongatedness, 2.10

F

Finite (language), 3.5, 4.5
 Finite-state, 3.2.2, 3.2.3, 4.2
 Frontier-to-root tree acceptor, 6.4.2

G

Genus, 2.7, 5.5.1
 Geodesic, 2.9
 Grammar, 8.1.1
 array, 8.3, 8.3.1
 context-free, 8.1.2, 8.3.1
 context-sensitive, 8.1.2
 finite-state, 8.1.2, 8.3.1
 isometric, 8.1.3
 linear, 8.3.1
 matrix, 8.2
 monotonic, 8.1.2, 8.3.1
 parallel, 8.1.4, 8.3.3

H

Hexagonal array, 2.2
 Hole, 2.2

I

Intersection, 3.5, 4.5, 6.5.1

J

Jordan curve theorem, 2.4

L

L-system, 8.1.4
 Language, 3.2.3, 3.4.1, 4.3.1, 4.4.2,
 5.3.1, 5.4, 8.1.1, 8.3.1
 adult, 8.1.4

N

Neighbors (4-, 8-), 2.2
 Nondeterministic, 3.2.1, 4.2

O

One-way acceptor, 3.3.3, 7.4

P

Palindromes, 3.4.3, 6.5.2, 7.3.1
 Parallel/sequential acceptor, 7.4
 Parenthesis strings, 3.4.3, 6.5.1
 Parsing, 8.1.1
 Path, 2.2
 Pebble acceptor, 7.2
 Perceptron, 6.4.3
 Perimeter, 2.7, 2.8
 Production, 8.1.1
 Pushdown acceptor, 7.3
 Pyramid cellular acceptor, 6

R

Rectangle, 4.2, 5.3.1, App. 5.3.1, 6.5.3
 Repetition, 3.4.3, 7.3.1
 Reversal, 3.5, 4.5, 6.5.1
 Rewriting rule, 8.1.1
 Run length, 3.4.3, 7.3.1

S

Separates, 2.6
 Shrinking, 5.5.1
 Simple point, 2.5
 Simply connected, 2.2
 Singleton, 3.5, 4.5
 State, 3.2.1, 3.4.1, 4.2, 5.2
 accepting, 3.2.3, 3.4.1, 5.2.2
 initial 3.2.2, 3.2.3, 3.4.1, 5.2.2
 Straightness, 2.10
 Surrounds, 2.6
 Symbol, *see* Vocabulary
 start (initial), 8.1.1
 Synchronization, App. 3.4.1, 5.3.1

T

Tape, 3.2.1, 4.2
 Tape-bounded, 3.2.2, 3.2.3, 4.2
 Three-way acceptor, 4.3.3, 4.4.3
 Time (for acceptance), 3.4.3, 5.5
 Transition function, 3.2.1, 3.4.1, 4.2,
 4.5
 Traversal, 4.4.1, 4.4.2, 7.2.3, 7.3.2
 Tree acceptor, 6.4.2
 Turing (machine), 3.2.2, 3.2.3, 4.2

U

Union, 3.5, 4.5, 6.5.1

V

Vocabulary, 3.2.1, 8.1.1
(Non)terminal, 8.1.1
Stack, 7.3

$\#$, *see* Blank
 $\#$ -preserving, 3.2.2, 3.2.3, 4.2

Computer Science and Applied Mathematics
A SERIES OF MONOGRAPHS AND TEXTBOOKS

Editor
Werner Rheinboldt
University of Maryland

HANS P. KÜNZI, H. G. TZSCHACH, and C. A. ZEHNDER. Numerical Methods of Mathematical Optimization: With ALGOL and FORTRAN Programs, Corrected and Augmented Edition

AZRIEL ROSENFELD. Picture Processing by Computer

JAMES ORTEGA AND WERNER RHEINOLDT. Iterative Solution of Nonlinear Equations in Several Variables

AZARIA PAZ. Introduction to Probabilistic Automata

DAVID YOUNG. Iterative Solution of Large Linear Systems

ANN YASUHARA. Recursive Function Theory and Logic

JAMES M. ORTEGA. Numerical Analysis: A Second Course

G. W. STEWART. Introduction to Matrix Computations

CHIN-LIANG CHANG AND RICHARD CHAR-TUNG LEE. Symbolic Logic and Mechanical Theorem Proving

C. C. GOTLIEB AND A. BORODIN. Social Issues in Computing

ERWIN ENGELER. Introduction to the Theory of Computation

F. W. J. OLVER. Asymptotics and Special Functions

DIONYSIOS C. TSICHRITZIS AND PHILIP A. BERNSTEIN. Operating Systems

ROBERT R. KORFHAGE. Discrete Computational Structures

PHILIP J. DAVIS AND PHILIP RABINOWITZ. Methods of Numerical Integration

A. T. BERZTISS. Data Structures: Theory and Practice, Second Edition

N. CHRISTOPHIDES. Graph Theory: An Algorithmic Approach

ALBERT NIJENHUIS AND HERBERT S. WILF. Combinatorial Algorithms

AZRIEL ROSENFELD AND AVINASH C. KAK. Digital Picture Processing

SAKTI P. GHOSH. Data Base Organization for Data Management

DIONYSIOS C. TSICHRITZIS AND FREDERICK H. LOCHOVSKY. Data Base Management Systems

WILLIAM F. AMES. Numerical Methods for Partial Differential Equations, Second Edition

ARNOLD O. ALLEN. Probability, Statistics, and Queueing Theory: With Computer Science Applications

ALBERT NIJENHUIS AND HERBERT S. WILF. Combinatorial Algorithms. Second edition.

JAMES S. VANDERGRAFT. Introduction to Numerical Computations

AZRIEL ROSENFELD. Picture Languages, Formal Models for Picture Recognition

In preparation

ISAAC FRIED. Numerical Solution of Differential Equations