

## Relationships Between Nondeterministic and Deterministic Tape Complexities\*

WALTER J. SAVITCH

*Department of Applied Physics and Information Science,  
University of California, San Diego, La Jolla, California 92037*

Received August 29, 1969

The amount of storage needed to simulate a nondeterministic tape bounded Turing machine on a deterministic Turing machine is investigated. Results include the following: *Theorem.* A nondeterministic  $L(n)$ -tape bounded Turing machine can be simulated by a deterministic  $[L(n)]^2$ -tape bounded Turing machine, provided  $L(n) \geq \log_2 n$ . Computations of nondeterministic machines are shown to correspond to threadings of certain mazes. This correspondence is used to produce a specific set, namely the set of all codings of threadable mazes, such that, if there is any set which distinguishes nondeterministic tape complexity classes from deterministic tape complexity classes, then this is one such set.

### INTRODUCTION

A well-known open problem in the theory of formal languages and computational complexity is to decide whether or not there exists a nondeterministic context-sensitive language, that is, to decide whether or not there is a set which is accepted by a nondeterministic linear bounded automaton, but is accepted by no such deterministic machine. In this paper, we give some partial answers to a natural generalization of this problem. We attempt to answer the following question: "Given a nondeterministic tape bounded Turing machine which accepts a set  $A$ , how much additional storage does a deterministic Turing machine require to recognize  $A$ ?"

More specifically, we show that any nondeterministic  $L(n)$ -tape bounded Turing machine can be simulated by a deterministic  $[L(n)]^2$ -tape bounded Turing machine, provided  $L(n) \geq \log_2 n$ . Thus, in particular, every context-sensitive language can be recognized within deterministic storage  $n^2$ , where  $n$  is the length of the input. As a corollary of the proof of this theorem, we get that if a context-sensitive language is

\* This work is based on a portion of the author's doctoral dissertation in the Department of Mathematics of the University of California, Berkeley. Most of the results were announced in [9]. Part of this research was done while the author was a National Science Foundation Graduate Fellow.

accepted by a nondeterministic linear bounded automaton within polynomial time, then the language is accepted by a deterministic Turing machine within storage  $n \log_2 n$ . It is also shown that if nondeterministic and deterministic tape complexity classes are equal for "small" functions, then they are equal for "large" functions.

The notion of a maze is formalized and computations of nondeterministic machines are related to threadings of certain mazes. This relation is used to obtain a specific set, namely the set of codings of threadable mazes, such that, if there is any set which distinguishes nondeterministic tape complexity classes from deterministic tape complexity classes, then this is one such set. That is, there is a nondeterministic Turing machine which accepts the codings of threadable mazes within storage  $\log_2 n$ . Also, if there is a deterministic Turing machine which accepts them within the same storage,  $\log_2 n$ , then, for every storage function  $L(n) \geq \log_2 n$ , any nondeterministic  $L(n)$ -tape bounded Turing machine can be stimulated by a deterministic  $L(n)$ -tape bounded Turing machine.

#### MACHINE MODEL

The device studied in this paper is the multitape Turing Machine. In the terminology of [3], it is a Turing machine with finitely many two-way infinite storage tapes and a read only input tape provided with end markers. We shall give only an informal definition of the device. A *multitape Turing machine* (hereafter called simply *Turing machine*) is a finite state control attached to a read only input tape and finitely many read/write storage tapes. The tapes are divided into squares. Each square of a storage tape is capable of holding one symbol from the finite storage tape alphabet  $\Gamma$ . The input tape will always contain a finite string  $w$  of symbols from the finite input alphabet  $\Sigma$ . The string  $w$  is separated by two symbols not in  $\Sigma$  called *left* and *right end markers*. Each tape has one head communicating with the finite control.

At any point in time, each head will scan one square on its tape and the control will be in one state. Depending on this state and the symbols scanned by the heads, the machine will, in one step, assume another state, overwrite a symbol on the scanned square of each of its storage tapes, and then shift some of its heads (including possibly the input head) either left or right one square. The finite control is designed so that the input head will never leave the segment of tape containing  $w$  and the end markers. The machine is said to be *deterministic* if there is only one possible action at each step. It is said to be *nondeterministic* if there are finitely many possible actions at each step.

Some states are distinguished and called *accepting states* and one state is distinguished and called the *initial state*. We say the deterministic machine  $Z$  *accepts* the input string  $w$  over  $\Sigma$  if  $Z$  enters an accepting state after finitely many steps, when started in the initial state, with all storage tapes blank,  $w$  on its input tape and its input head scanning the left end marker. The definition is the same in the case  $Z$  is nondeterministic

except that we merely require that there be a possible finite sequence of steps leading to an accepting state, starting from the initial configuration described above.

**DEFINITION.** Suppose  $L(n)$  is a function on the natural numbers,  $Z$  is a Turing machine (deterministic or nondeterministic), and  $A$  is a set of strings over the input alphabet of  $Z$ .  $Z$  is said to *accept* the set  $A$  *within storage*  $L(n)$  provided that

- (i) for each string  $w$  in  $A$ , there is at least one possible computation of  $Z$  which accepts  $w$  and in which each storage tape head scans at most  $L(|w|)$  squares, and
- (ii)  $Z$  accepts no string not in  $A$ .  $|w|$  is the length of the string,  $w$ .

In this paper we are considering only the growth rates of storage functions. That is, for our purposes, the storage functions  $L(n)$  and  $\epsilon L(n)$  are the same, for any constant  $\epsilon > 0$ . This identification of functions follows from the fact that our Turing machines are allowed arbitrary finite storage tape alphabets. Thus, any Turing machine which operates in storage  $L(n)$  can be modified to operate in storage  $\epsilon L(n)$  and still accept the same set of tapes. (See, for example, [3].) We assume the reader is familiar with such tape reduction techniques.

In what follows, it will sometimes be convenient to assume that the storage functions are “nice”. The niceness condition we want is that the functions be measurable [4].

**DEFINITION.** A function  $L(n)$  is said to be *measurable* if there is some Turing machine with just one storage tape such that, given any input of length  $n$ , the machine will halt after a computation in which the storage tape head scans exactly  $L(n)$  squares.

Apparently, all common storage functions  $L(n) \geq \log_2 n$  are measurable. In particular, any polynomial in  $n$  and  $\log_2 n$  is measurable.

#### DETERMINISTIC SIMULATION OF NONDETERMINISTIC TURING MACHINES

**THEOREM 1.** If a set,  $A$ , is accepted by a nondeterministic Turing machine,  $Z_N$ , within storage  $L(n) \geq \log_2 n$ , then  $A$  is accepted by some deterministic Turing machine,  $Z_D$ , within storage  $[L(n)]^2$ .

*Proof.* The theorem is proved by exhibiting an algorithm whereby  $Z_D$  can simulate the computations of  $Z_N$ . The algorithm is similar to that used by Lewis, Stearns, and Hartmanis [7] to show that every context-free language is accepted by a deterministic Turing machine within storage  $(\log_2 n)^2$ .

Suppose the Turing machine  $Z_N$  has  $k$  storage tapes, storage tape alphabet  $\Gamma$ , and internal state set  $Q$ . Let  $\Delta = Q \cup \Gamma \cup \{1, 2, *, \downarrow\}$ , where  $\downarrow$  and  $*$  are two new symbols. An *instantaneous description* (ID) of  $Z_N$  is a string over  $\Delta$  of the form

$pq * u_1 \downarrow v_1 * u_2 \downarrow v_2 * \dots * u_k \downarrow v_k$ , where  $p$  is a number in dyadic notation<sup>1</sup> indicating the position of the input head of  $Z_N$ ,  $q$  is an element of  $Q$  indicating that the finite control of  $Z_N$  is in state  $q$ , and  $u_i, v_i$  are elements of  $\Gamma^*$ ,  $i = 1, 2, \dots, k$ .  $u_i \downarrow v_i$  is interpreted to mean that  $u_i v_i$  is the contents of the  $i$ th storage tape of  $Z_N$  and the head on this tape is scanning the first symbol of string  $v_i$ . Thus, except for the contents of the input tape, an ID of  $Z_N$  is a complete description of a configuration of  $Z_N$ .

Since  $Z_N$  accepts the set  $A$  within storage  $L(n) \geq \log_2 n$ , it follows that for each string  $w$  of length  $n$  accepted by  $Z_N$ , there is a computation of  $Z_N$  accepting  $w$  in which every ID has length not exceeding  $cL(n)$ . Furthermore, since each ID is of length at most  $cL(n)$ , the computation need take no more than  $2^{cL(n)}$  steps.  $c$  and  $c'$  are constants depending only on  $Z_N$  and not on the input  $w$ .

We first give the algorithm for the case:  $L(n)$  is measurable.

Given an input of length  $n$ ,  $Z_D$  initializes its computation by dividing one of its storage tapes into  $\lceil cL(n) \rceil + 1$  blocks (called *registers*), each of length  $\lceil cL(n) \rceil$ , followed by  $\lceil cL(n) \rceil + 1$  blocks (called *tags*), each capable of storing either 0 or 1. This it can easily do if  $L(n)$  is measurable.  $\lfloor y \rfloor$  is the largest integer not exceeding  $y$ . Note that each register is capable of holding any ID of  $Z_N$  in which the nonblank portion of every storage tape is at most  $L(n)$ .  $Z_D$  has a second storage tape for scratch work.

The idea of the algorithm is as follows.  $Z_D$  has the string  $w$  as input. In the course of the algorithm,  $Z_D$  will consider two ID's  $I, I''$  stored in particular registers. For a prechosen  $m$ , it will need to check if, with input  $w$ ,  $Z_N$  can in  $2^m$  steps change its configuration from  $I$  to  $I''$ . Furthermore,  $Z_D$  will have to perform this task using only  $m$  registers.  $Z_D$  proceeds as follows. It runs through (in a systematic way) all ID's  $I'$  of  $Z_N$  and for each  $I'$ , checks to see if there is a computation of at most  $2^{m-1}$  steps from  $I$  to  $I'$ , and a computation of at most  $2^{m-1}$  steps from  $I'$  to  $I''$ .  $Z_D$  needs one register to keep track of the  $I'$ , leaving it with  $m - 1$  registers. So  $Z_D$  has reduced the task of simulating  $2^m$  moves of  $Z_N$  using  $m$  registers to simulating  $2^{m-1}$  moves of  $Z_N$  using  $m - 1$  registers.  $Z_D$  then reduces each computation of  $2^{m-1}$  steps to two computations of  $2^{m-2}$  steps.  $Z_D$  continues to reduce the length of the computations it need check until it need only check, for appropriate ID's  $I_1$  and  $I_2$ , whether, with input  $w$ ,  $Z_N$  can in one step go from  $I_1$  to  $I_2$ . This it can easily do using zero storage. Now if there is any accepting computation of  $Z_N$ , then there is one which accepts in  $2^{c'L(n)}$  steps. So  $Z_D$  can simulate this computation using  $c'L(n)$  registers or about  $[L(n)]^2$  squares of storage tape.

We now define some notation for a formal statement of the algorithm. Since ID's are strings over a finite alphabet,  $\Delta$ , they may be considered to be integers in  $m$ -adic notation, where  $m$  is the number of symbols in  $\Delta$ . More precisely, if  $\Delta = \{a_1, a_2, \dots, a_m\}$  then the string  $a_{i_k} a_{i_{k-1}} \dots a_{i_0}$  will represent the number  $\sum_{j=0}^k i_j m^j$ . We will not distinguish between a string and the number it represents.

<sup>1</sup> The string  $d_i d_{i-1} \dots d_0$  over the alphabet  $\{1, 2\}$  is the dyadic representation of the number  $\sum_{i=0}^i d_i 2^i$ .

DEFINITION. If  $I$  and  $I'$  are ID's of  $Z_N$  and if  $Z_N$  can in one step, with input  $w$ , pass from configuration  $I$  to configuration  $I'$ , or if  $I = I'$ , then we will write  $I \xrightarrow{w} I'$ . In the algorithm we will test if  $I \xrightarrow{w} I'$  is true for arbitrary strings  $I, I'$  over  $\Delta$ . If either  $I$  or  $I'$  is not an ID of  $Z_N$ , then the test fails.

In the following algorithm,  $x_i$  denotes the contents of the  $i$ -th register and  $T_i$  denotes the contents of the  $i$ -th tag [ $1 \leq i \leq c'L(n) + 1$ ].  $Z_D$  has a string  $w$  as input and all references to computations of  $Z_N$  mean computations of  $Z_N$  under the input  $w$  and in which the nonblank portion of each work tape remains at most  $L(|w|)$  squares long. At any point in the computation of  $Z_D$ , if  $T_i = 1$ , then there is a computation of  $Z_N$  from the start ID to  $x_i$ .

ALGORITHM FOR MACHINE  $Z_D$

Summary of Notation

$w$  is the input string.

$n$  is the length of  $w$ .

$r$  is the least integer such that  $r \geq c'L(n)$ .

$c'$  is such that, with  $w$  as input, if  $Z_N$  accepts  $w$  at all, then  $Z_N$  accepts  $w$  in  $2^{c'L(n)}$  steps.

$I_0$  is the initial ID of  $Z_N$ .

$N$  is the largest integer whose  $m$ -adic representation can be stored in a register.

$x_i$  is the contents of the  $i$ -th register.

$T_i$  is the contents of the  $i$ -th tag.

D0 (Initialize). Compute  $r$  and set up the  $r + 1$  registers and tags. For  $i = 1, 2, \dots, r$  set  $x_i \leftarrow 1$  and  $T_i \leftarrow 0$ . Set  $x_{r+1} \leftarrow I_0$  (the initial ID) and  $T_{r+1} \leftarrow 1$ .

D1 (Test). If each  $T_i = 1$ , go to D3. Otherwise, let  $j$  be the smallest index such that  $T_j = 0$ . Test: For some index  $h$ ,  $T_h = 1$  and  $x_h \xrightarrow{w} x_j$ .

Yes: Go to D2.

No: Go to D3.

D2 (Clear Storage) ( $j$  as in D1). If  $x_j$  is an accepting ID, ACCEPT. Otherwise, set  $T_j \leftarrow 1$  and for  $i = 1, 2, \dots, j - 1$  set  $x_i \leftarrow 1$  and  $T_i \leftarrow 0$ . Go to D1.

D3 (Back Track). If  $x_i = N$  for each  $i \leq r$ , REJECT. Otherwise, let  $k$  be the smallest index such that  $x_k \neq N$ . Set  $x_k \leftarrow x_k + 1$ ,  $T_k \leftarrow 0$ , and for  $i = 1, 2, \dots, k - 1$  set  $x_i \leftarrow 1$ ,  $T_i \leftarrow 0$ . Go to D1.

Clearly the algorithm works within storage  $c''[L(n)]^2$ , where  $c''$  is a constant depending only on  $Z_N$ . By standard techniques [3] the constant can be reduced to one. A simple

induction on the number of steps completed shows that if  $Z_D$  accepts an input  $w$ , then the nondeterministic machine  $Z_N$  does. Thus, it will suffice to show that  $Z_D$  accepts all strings that  $Z_N$  accepts. To show this, we need some preliminary lemmas. The lemmas are easier to state in case the machine does not stop when it accepts. So assume D2 (Clear Storage) has been modified to omit the first line ("If  $x_j$  is an accepting ID, ACCEPT"). We shall say  $Z_D$  is in *configuration*

$$\langle x_1, x_2, \dots, x_{r+1} \rangle, \quad \langle T_1, T_2, \dots, T_{r+1} \rangle$$

if it is about to execute D1, and the contents of the registers and tags are, respectively,  $x_1, x_2, \dots, x_{r+1}$  and  $T_1, T_2, \dots, T_{r+1}$ .

LEMMA 1. (For modified D2). For each index  $i$  ( $i = 1, 2, \dots, r$ ) and each integer  $x$  ( $x = 1, 2, \dots, N$ ) if at some time in a computation of  $Z_D$  the configuration of  $Z_D$  is

$$\langle \underbrace{1, 1, \dots, 1}_i, x_{i+1}, x_{i+2}, \dots, x_r, I_0 \rangle, \quad \langle \underbrace{0, 0, \dots, 0}_i, T_{i+1}, T_{i+2}, \dots, T_r, 1 \rangle \quad (1)$$

for some  $x_{i+1}, x_{i+2}, \dots, x_r$  and some  $T_{i+1}, T_{i+2}, \dots, T_r$ , then at some later time the configuration will be

$$\langle \underbrace{1, 1, \dots, 1}_{i-1}, x, x_{i+1}, x_{i+2}, \dots, x_r, I_0 \rangle, \quad \langle \underbrace{0, 0, \dots, 0}_i, T_{i+1}, T_{i+2}, \dots, T_r, 1 \rangle \quad (2)$$

or

$$\langle \underbrace{1, 1, \dots, 1}_{j-1}, x_j, x_{j+1}, \dots, x_r, I_0 \rangle, \quad \langle \underbrace{0, 0, \dots, 0}_{j-1}, 1, T_{j+1}, \dots, T_r, 1 \rangle, \quad (3)$$

where  $j$  is the least index in (1) such that  $j > i$  and  $T_j = 0$ . (If no such  $j$  exists, then (2) happens.)

The algorithm is trying to establish that there is a computation of  $Z_N$  from the start ID to  $x_j$ . It is trying to do this using only registers to the left of  $x_j$ . So the lemma says that  $Z_D$  can get any ID into any register, if it needs to.

*Proof.* A formal proof would use induction on  $i$  and  $x$ . We list some representative configurations in a computation. This should make the lemma clear. Note that since  $N$  is a string all of whose digits are the same,  $N$  is not an ID and, hence,  $x \neq N$  is not true for any integer  $x$ . Suppose  $Z_D$  is started in configuration (1) and that at no later

time does  $Z_D$  assume configurations (3).  $Z_D$  will then assume the following configurations:

$$\begin{aligned}
 &\langle \underbrace{1, 1, \dots, 1}_{i-1}, x-1, x_{i+1}, x_{i+2}, \dots, x_r, I_0 \rangle, && \langle \underbrace{0, 0, \dots, 0}_i, T_{i+1}, T_{i+2}, \dots, T_r, 1 \rangle \\
 &\langle \underbrace{1, 1, \dots, 1}_{i-2}, N, x-1, x_{i+1}, x_{i+2}, \dots, x_r, I_0 \rangle, && \langle \underbrace{0, 0, \dots, 0}_{i-1}, T_i, T_{i+1}, T_{i+2}, \dots, T_r, 1 \rangle \\
 & && \text{where } T_i = 0 \quad \text{or} \quad 1. \\
 &\langle \underbrace{N, N, \dots, N}_{i-1}, x-1, x_{i+1}, x_{i+2}, \dots, x_r, I_0 \rangle, && \langle \underbrace{0, 0, \dots, 0}_{i-1}, T_i, T_{i+1}, T_{i+2}, \dots, T_r, 1 \rangle
 \end{aligned}$$

Finally, via D3,  $Z_D$  assumes (2) as desired.

LEMMA 2. (For modified D2). Suppose that at some time in a computation the configuration of  $Z_D$  is

$$\langle \underbrace{1, 1, \dots, 1}_{j-1}, x, x_{j+1}, x_{j+2}, \dots, x_r, I_0 \rangle, \quad \langle \underbrace{0, 0, \dots, 0}_{j-1}, T_j, T_{j+1}, T_{j+2}, \dots, T_r, 1 \rangle \quad (4)$$

with  $T_j = 0$  and any values for the  $x$ 's and remaining  $T$ 's. If for some index  $q$ ,  $T_q = 1$  and there is a computation of  $Z_N$  of at most  $2^j$  steps from  $x_q$  to  $x$ , then at some later time the configuration of  $Z_D$  will be (4) with  $T_j = 1$ .

The lemma says that  $j$  registers are sufficient for  $Z_D$  to check a computation of  $2^j$  steps of  $Z_N$ .

LEMMA 3. (For modified D2). Suppose that at some time in a computation the configuration of  $Z_D$  is

$$\begin{aligned}
 &\langle \underbrace{1, 1, \dots, 1}_{j-1}, \underbrace{x_j, x_{j+1}, \dots, x_{j+p}, x, x_{j+p+2}, \dots, x_r, I_0}_{p+1} \rangle, \\
 &\langle \underbrace{0, 0, \dots, 0}_{j-1}, \underbrace{1, 1, \dots, 1, 0}_{p+1}, T_{j+p+2}, T_{j+p+3}, \dots, T_r, 1 \rangle
 \end{aligned} \quad (5)$$

for any values of the  $x$ 's and  $T$ 's. If for some index  $q$ ,  $T_q = 1$  and there is a computation of  $Z_N$  of at most  $2^j$  steps from  $x_q$  to  $x$ , then at some later time the configuration of  $Z_D$  will be

$$\begin{aligned}
 &\langle \underbrace{1, 1, \dots, 1}_{j+p}, x, x_{j+p+2}, x_{j+p+3}, \dots, x_r, I_0 \rangle, \\
 &\langle \underbrace{0, 0, \dots, 0}_{j+p}, 1, T_{j+p+2}, T_{j+p+3}, \dots, T_r, 1 \rangle.
 \end{aligned} \quad (6)$$

*Proof of Both Lemmas.* Lemmas 2 and 3 are basically a single lemma and are proven simultaneously by induction on  $j$ . The base case,  $j = 1$ , is clear. Assume both lemmas hold with  $j$  replaced by  $j - 1$ . We will show that Lemma 2 holds for  $j$ .

Suppose  $Z_D$  is in configuration (4) with  $T_j = 0$  and that for some index,  $q$ , there is a computation of  $Z_N$  from  $x_q$  to  $x$  of at most  $2^j$  steps. Decomposing this computation from  $x_q$  to  $x$ , we get that there is a  $y$  such that there are computations of at most  $2^{j-1}$  steps from  $x_q$  to  $y$  and from  $y$  to  $x$ . By Lemma 1, it follows that, at some later time, either the configuration of  $Z_D$  is

$$\begin{aligned} & \langle \underbrace{1, 1, \dots, 1}_{j-2}, y, x, x_{j+1}, x_{j+2}, \dots, x_r, I_0 \rangle, \\ & \langle 0, \underbrace{0, \dots, 0}_{j-2}, T_{j-1}, 0, T_{j+1}, T_{j+2}, \dots, T_r, 1 \rangle \end{aligned} \tag{7}$$

with  $T_{j-1} = 0$  or the configuration is (4) with  $T_j = 1$ . Configuration (4) with  $T_j = 1$  is the desired outcome. So assume  $Z_D$  assumed configuration (7) with  $T_{j-1} = 0$ . By induction hypothesis on Lemma 2, it follows that at some still later time the configuration of  $Z_D$  will be (7) with  $T_{j-1} = 1$ . If we then apply the induction hypothesis on Lemma 3, we get that at a still later time the configuration of  $Z_D$  will be (4) with  $T_j = 1$ . So Lemma 2 holds for  $j$ . Lemma 3 for  $j$  is established in a similar fashion. Thus, the induction hypothesis is established and the proof of both lemmas is completed.

Using Lemmas 1 and 2 it is easy to complete the proof of the theorem. Suppose the nondeterministic machine  $Z_N$  accepts  $w$ . We must show that  $Z_D$  accepts  $w$ . Since  $Z_N$  accepts  $w$ , there is an accepting ID,  $x$ , of  $Z_N$  such that there is a computation, with  $w$  as input, of at most  $2^r$  steps from  $I_0$  (the initial ID of  $Z_N$ ) to  $x$ . By Lemma 1, it follows that  $Z_D$  eventually assumes configuration

$$\langle \underbrace{1, 1, \dots, 1}_{r-1}, x, I_0 \rangle, \quad \langle 0, \underbrace{0, \dots, 0}_{r-1}, T_r, 1 \rangle \tag{8}$$

with  $T_r = 0$ . By Lemma 2, it follows that at some later time  $Z_D$  will assume configuration (8) with  $T_r = 1$ . All this was for the modified D2. The unmodified D2 will accept  $w$  rather than set  $T_r \leftarrow 1$ . This completes the proof of Theorem 1 in the case:  $L(n)$  is measurable.

If  $L(n)$  is not measurable, then the above algorithm must be altered, since  $Z_D$  cannot necessarily mark off blocks of length  $cL(n)$  within storage proportionate to  $[L(n)]^2$ . However, if  $Z_D$  were somehow given the value of  $L(n)$ , then by the above procedure,  $Z_D$  could determine whether or not  $Z_N$  accepts the input  $w$  within storage  $L(n)$ . So  $Z_D$  operates as follows.  $Z_D$  first assumes  $L(n) = 1$ . If  $Z_N$  accepts within storage  $L(n) = 1$ , then  $Z_D$  will accept. If  $Z_N$  does not accept within storage 1, then  $Z_D$  next



assumes  $L(n) = 2$ . If  $Z_N$  accepts within storage 2, then  $Z_D$  accepts. If not, then  $Z_D$  next assumes  $L(n) = 3$ .  $Z_D$  proceeds in this manner trying larger and larger values for  $L(n)$ . If  $Z_N$  accepts the input  $w$ , then  $Z_D$  will eventually find the correct value for  $L(n)$  and accept  $w$  within storage proportionate to  $[L(n)]^2$ . If  $Z_N$  does not accept  $w$ , then  $Z_D$  computes forever on input  $w$ . This completes the proof of Theorem 1.

The context-sensitive languages are precisely those sets accepted by nondeterministic Turing machines within storage  $L(n) = n$  (see, for example, [3]). So, setting  $L(n) = n$  in Theorem 1, we get,

**COROLLARY 1.** Every context-sensitive language is accepted by some deterministic Turing machine within storage  $n^2$ .

Theorem 1 can be generalized as follows.

**THEOREM 2.** If a set,  $A$ , is accepted by a nondeterministic Turing machine within storage  $L(n)$  and time  $T(n)$ , then  $A$  is accepted by some deterministic Turing machine within storage  $L(n) \log_2 T(n)$ .

**DEFINITION.** A Turing machine,  $Z$ , is said to accept a set,  $A$ , within storage  $L(n)$  and time  $T(n)$  provided that

- (i) for each  $w$  in  $A$ , there is at least one computation of  $Z$  which accepts  $w$ , takes at most  $T(n)$  steps, and in which each work tape head scans at most  $L(n)$  squares, where  $n$  is the length of  $w$ , and provided that
- (ii)  $Z$  accepts no string not in  $A$ .

*Proof of Theorem 2.* An analysis of the proof of Theorem 1 shows that the algorithm uses a bound on the storage of the nondeterministic machine to obtain a bound, namely exponential in the storage, on the time of the nondeterministic machine. It then marks off  $\lceil \log_2 T(n) \rceil$  blocks of storage, where  $T(n)$  is a bound on the time. Clearly, if the time bound were easy to compute, the algorithm could be modified to use it as the bound on the time. We would then get Theorem 2. If the time function is not easy to compute, then we use the same sort of trick as that used for nonmeasurable functions in the proof of Theorem 1.

**COROLLARY 2.** If a context-sensitive language is accepted by a nondeterministic Turing machine within linear storage and polynomial time, then it is accepted by some deterministic Turing machine within storage  $n \log_2 n$ .

The following theorem says that for any "well behaved" storage function  $L(n)$ , if every nondeterministic  $L(n)$ -tape bounded Turing machine can be simulated by a deterministic  $L(n)$ -tape bounded Turing machine, then for all larger storage functions nondeterministic and deterministic tape complexities are the same.

**THEOREM 3.** Suppose that  $L(n)$  and  $H(n)$  are measurable, that  $L(n)$  is monotone increasing and unbounded and that  $\log_2 n \leq L(n) \leq H(n)$ , for all  $n$ . Let the function  $k(n)$  be defined as follows. For each natural number  $n$ ,  $k(n)$  is the least natural number such that  $H(n) \leq L[k(n)]$ .

Suppose further that for any set,  $A$ , if  $A$  is accepted by a nondeterministic Turing machine within storage  $L(n)$ , then  $A$  is accepted by a deterministic Turing machine within storage  $L(n)$ . It then follows that for any set,  $B$ , if  $B$  is accepted by a nondeterministic Turing machine within storage  $H(n)$ , then  $B$  is accepted by a deterministic Turing machine within storage  $L[k(n)]$ .

The theorem is a bit stronger than it seems. For most common storage functions,  $L$ , there is a constant  $c$  such that  $cL[k(n)] \leq H(n) \leq L[k(n)]$ , for all  $H$  and  $n$ . Thus, in these cases, the conclusion can be strengthened to say that any set accepted within nondeterministic storage  $H(n)$  can be accepted within deterministic storage  $H(n)$ . In particular,

**COROLLARY 3.** Suppose  $L(n)$  is a polynomial in  $\log_2 n$ ,  $n$  and  $c^n$ , for some constant  $c$ . Suppose further that for any set,  $A$ , if  $A$  is accepted by a nondeterministic Turing machine within storage  $L(n)$ , then  $A$  is accepted by some deterministic Turing machine within storage  $L(n)$ .

It then follows that for any measurable function  $H(n) \geq L(n)$  and any set,  $B$ , if  $B$  is accepted by a nondeterministic Turing machine within storage  $H(n)$ , then  $B$  is accepted by some deterministic Turing machine within storage  $H(n)$ .

Among other things, the corollary says that if every context-sensitive language is accepted by a deterministic linear bounded automaton (i.e., deterministic Turing machine with storage bound  $n$ ), then nondeterministic and deterministic tape complexities are the same for all measurable tape bounds  $H(n) \geq n$ .

*Proof of Corollary 3.* It is easy to check that for any such polynomial  $L(n)$ , there is a constant  $c$  such that  $L(n)/L(n+1) \geq c$  for all  $n$ . So  $cL[k(n)] \leq L[k(n)-1]$ .  $k(n)$  is the function defined in the statement of the theorem. By definition of  $k(n)$ ,  $L[k(n)-1] \leq H(n)$ . So  $cL[k(n)] \leq H(n)$ .

Assume the hypothesis of the corollary and suppose  $B$  is accepted by a nondeterministic Turing machine within storage  $H(n)$ . By the Theorem 3,  $B$  is accepted by a deterministic Turing machine within storage  $L[k(n)]$ . So, by standard tape reduction techniques [3], it follows that  $B$  is accepted by a deterministic Turing machine within storage  $cL[k(n)]$ . Finally, since  $cL[k(n)] \leq H(n)$ , this means that  $B$  is accepted by a deterministic Turing machine within storage  $H(n)$ .

*Proof of Theorem 3.* Assume the hypothesis of the Theorem 3 and suppose  $B$  is a set accepted by a nondeterministic Turing machine,  $Z_N^H$ , within storage  $H(n)$ .

We will construct a deterministic machine,  $Z_D^H$ , which accepts  $B$  within storage  $L[k(n)]$ .

Let  $\beta$  be a new symbol. Let  $A = \{w\beta^h \mid w \text{ in } B \text{ and } |w| + h \geq k(|w|)\}$ . A non-deterministic Turing machine,  $Z_N^L$ , which accepts  $A$  within storage  $L(n)$  can be constructed as follows. Given input  $w\beta^h$ ,  $Z_N^L$  first marks off  $L(n)$  squares of storage, where  $n = |w\beta^h|$ . It then checks to see if  $L(n) \geq H(|w|)$ . If this is not the case, then  $n = |w| + h < k(|w|)$  and the computation halts. If  $L(n) \geq H(|w|)$ , then  $Z_N^L$  mimics  $Z_N^H$  to see if  $w$  is in  $B$ . If  $Z_N^H$  accepts, then it does. Since  $L(n) \geq H(|w|)$ , this machine will accept  $A$  within storage  $L(n)$ . So, by hypothesis, there is a deterministic machine,  $Z_D^L$ , which accepts  $A$  within storage  $L(n)$ . Furthermore, since  $L$  is measurable, we may assume that on every input of length  $n$ ,  $Z_D^L$  halts after a computation in which at most  $L(n)$  squares of storage tape are scanned. (Our definitions merely require this when the machine accepts the input.)

The deterministic machine,  $Z_D^H$ , which accepts  $B$  within storage  $L[k(n)]$  operates as follows. Given an input  $w$ ,  $Z_D^H$  mimics  $Z_D^L$  operating on  $w\beta^h$  for various values of  $h$ . It first tries  $h = 0$ . If  $Z_D^L$  accepts  $w$ , then  $Z_D^H$  does. If not, it then tries  $h = 1$ . If  $Z_D^L$  accepts  $w\beta$ , then  $Z_D^H$  does. If not, it then tries  $h = 2$ . If  $Z_D^L$  accepts  $w\beta^2$ , the  $Z_D^H$  does. If not, it then tries  $h = 3$ , and so forth. If  $w$  is in  $B$ , then  $Z_D^H$  will eventually find the least  $h$  such that  $|w| + h \geq k(|w|)$  and will accept  $w$  within storage  $L(|w\beta^h|) = L[k(|w|)]$ . This completes the proof of Theorem 3.

It is natural to ask if there is any storage function whose deterministic and non-deterministic complexity classes are equal. The answer was given by Manuel Blum and is "yes". Blum showed that there are arbitrarily large storage functions  $L(n)$  such that a set is accepted within deterministic storage  $L(n)$  if, and only if, it is accepted within nondeterministic storage  $L(n)$ . These functions  $L(n)$  are not, however, "well-behaved" and Theorem 3 does not apply to them. They are not measurable. A sketch of the proof of Blum's result appears in [8].

One may also ask if there is a storage function (large enough to be interesting) whose deterministic and nondeterministic complexity classes are different. The next section is devoted to this question. Only an incomplete answer is given, however, and the question is open.

### MAZES AND TURING MACHINES

Informally, a maze is a set of rooms connected by one-way corridors. Certain rooms are designated goal rooms and one room is designated the start room. Thus, a maze is a directed graph with certain nodes or rooms distinguished. The maze is threadable if there is a path from the start room to some goal room.

A nondeterministic Turing machine, with input  $w$ , gives rise in a natural way to a maze. The rooms of the maze are the instantaneous descriptions of the machine

and the corridors are given by the transition function of the machine. That is there is a corridor from room  $I_1$  to room  $I_2$  if, with input  $w$ , the machine can in one step change its configuration from  $I_1$  to  $I_2$ . The initial instantaneous description of the machine is designated the start room. Those instantaneous descriptions which include an accepting state are designated goal rooms. The machine will accept the input  $w$  if, and only if, the corresponding maze is threadable.

The algorithm for simulating a nondeterministic Turing machine, given in the last section, is really an efficient method for determining whether the corresponding maze is threadable. In this section we show that if an efficient enough method for "threading" mazes could be found, then deterministic machines could simulate nondeterministic machines in the same storage. More precisely, the set of threadable mazes, suitably coded, can be recognized by a nondeterministic Turing machine within storage  $\log_2 n$ . This set can be recognized by some deterministic Turing machine within storage  $\log_2 n$  if, and only if, deterministic  $L(n)$ -tape bounded Turing machines can simulate nondeterministic  $L(n)$ -tape bounded Turing machines, for all  $L(n) \geq \log_2 n$ .

**DEFINITION.** A *maze* over  $\Sigma$  (a finite alphabet) is a quadruple,  $\mathcal{M} = (X, R, s, G)$ , where  $X$  is a finite set of strings over  $\Sigma$  ( $X$  is the set of rooms),  $R$  is a binary relation on  $X$  (giving the corridors),  $s$  is an element of  $X$  ( $s$  is the start room), and  $G$  is a subset of  $X$  ( $G$  is the set of goal rooms).

**DEFINITION.** The maze  $\mathcal{M} = (X, R, s, G)$  is *threadable* if there is a sequence  $r_1, r_2, \dots, r_e$  of rooms such that  $r_1 = s$  (the start room),  $r_e$  is an element of  $G$  (the goal rooms), and  $R(r_i, r_{i+1})$  holds for  $i = 1, 2, \dots, e - 1$ .

**DEFINITION.** Let  $]$ ,  $[$ ,  $*$  be three new symbols. A *coding* of the maze  $\mathcal{M} = (X, R, s, G)$  is a string of the form

$$s[x_1 * y_1^1 * y_2^1 * \dots * y_{n(1)}^1][x_2 * y_1^2 * y_2^2 * \dots * y_{n(2)}^2] \dots [x_l * y_1^l * \dots * y_{n(l)}^l] u_1 * u_2 * \dots * u_g \tag{1}$$

where  $s$  is the start room of  $\mathcal{M}$ ;  $x_1, x_2, \dots, x_l$  is an enumeration without repetitions of the rooms in  $X$ ; for  $1 \leq i \leq l$ ,  $y_1^i, y_2^i, \dots, y_{n(i)}^i$  is an enumeration without repetitions of all  $y$  in  $X$  such that  $R(x_i, y)$  holds; and  $u_1, u_2, \dots, u_g$  is an enumeration without repetitions of the rooms in  $G$ .

*Notation.*  $M_\Sigma$  denotes the set of all codings of *threadable mazes* over  $\Sigma$ .

**THEOREM 4.** For any finite alphabet  $\Sigma$ , there is a nondeterministic Turing machine which accepts  $M_\Sigma$  within storage  $\log_2 n$ .

*Proof.* If the maze has length  $n$ , then it has at most  $n$  rooms. So each room can be coded by a number which can be stored in  $\log_2 n$  squares of tape. More specifically, if the maze is given by (1), then the room  $x_i$  is coded by the number  $i$  in dyadic notation. The algorithm is very simple. The machine writes down the code number of the initial room on its work tape, finds the possible next rooms, guesses at one of them, erases its work tape, and writes down a coding of its guess. It then finds the possible next rooms, guesses at one of them and so forth. After each guess it checks to see if it has reached a goal room. If it has, it accepts. Clearly the work tape will be bounded by  $c \log_2 n$ , for some constant  $c$ .

A straight forward crossing sequence argument, like that of Cobham [2] shows that the bound,  $\log_2 n$  in Theorem 4, is the best possible, up to a constant factor.

Applying Theorems 1 and 4 to  $M_{\Sigma}$  yields

**COROLLARY 4.** For any finite alphabet  $\Sigma$ , there is a deterministic Turing machine which accepts  $M_{\Sigma}$  within storage  $(\log_2 n)^2$ .

**LEMMA 4.** For any finite alphabets  $\Sigma$  and  $\Delta$  with at least two elements,  $M_{\Sigma}$  is accepted by some deterministic Turing machine within storage  $\log_2 n$  if, and only if,  $M_{\Delta}$  is.

*Proof.* For every coding,  $w$ , of a maze over  $\Sigma$ , there is a coding,  $\delta(w)$ , of a maze over  $\Delta$  which is isomorphic to  $w$ . So  $w$  is in  $M_{\Sigma}$  if, and only if,  $\delta(w)$  is in  $M_{\Delta}$ . More specifically, if  $w$  is

$$s[x_1 * y_1^1 * y_2^1 * \dots * y_{n(1)}^1][x_2 * y_1^2 * y_2^2 * \dots * y_{n(2)}^2] \\ \dots [x_i * y_1^i * \dots * y_{n(i)}^i] u_1 * u_2 * \dots * u_g$$

then  $\delta(w)$  may be taken to be

$$\delta(s)[\delta(x_1) * \delta(y_1^1) * \delta(y_2^1) * \dots * \delta(y_{n(1)}^1)][\delta(x_2) * \delta(y_1^2) * \delta(y_2^2) * \dots * \delta(y_{n(2)}^2)] \\ \dots [\delta(x_i) * \delta(y_1^i) * \dots * \delta(y_{n(i)}^i)] \delta(u_1) * \delta(u_2) * \dots * \delta(u_g)$$

where if  $k$  is the number of symbols in  $\Delta$ , then  $\delta(x_i)$  is the  $k$ -adic representation of the number  $i$  and so is an element of  $\Delta^*$ . Since the  $u$ 's and  $y$ 's and  $s$  are each equal to some  $x_i$ , this defines  $\delta$  on them as well.

Given a deterministic machine,  $Z_{\Delta}$ , which accepts  $M_{\Delta}$  within storage  $\log_2 n$ , we construct a deterministic machine,  $Z_{\Sigma}$ , which accepts  $M_{\Sigma}$  within storage  $\log_2 n$ , as follows. Given an input  $w$ ,  $Z_{\Sigma}$  operates by mimicking  $Z_{\Delta}$  operating in  $\delta(w)$ . More specifically, suppose  $Z_{\Sigma}$  has input  $w$  as in (1). When  $Z_{\Sigma}$  is simulating  $Z_{\Delta}$  with the input head of  $Z_{\Delta}$  someplace in  $\delta(z)$ , where  $z$  is some  $s, x, y$ , or  $u$  in (1), then  $Z_{\Sigma}$  will have its input head reading the first symbol of  $z$  and will have  $\delta(z)$  written on an auxiliary

storage tape.  $Z_E$  uses this auxiliary storage tape containing  $\delta(z)$  to mimic the input tape of  $Z_A$ . When, in the mimicking process, the input head of  $Z_A$  would leave the left (respectively, right) end of  $\delta(z)$ , then  $Z_E$  moves its input head to the  $s$ ,  $x$ ,  $y$ , or  $u$  to the left (respectively, right) of  $z$ , calculates  $\delta$  of this  $s$ ,  $x$ ,  $y$ , or  $u$  and replaces  $\delta(z)$  by  $\delta$  of this  $s$ ,  $x$ ,  $y$ , or  $u$ .  $\delta$  of this  $s$ ,  $x$ ,  $y$ , or  $u$  becomes the new  $\delta(z)$  and the simulation continues. To calculate the new  $\delta(z)$ ,  $Z_E$  need only compare the new  $z$  to each  $x_i$  until it finds the  $x_i$  which equals  $z$ . This it can do symbol by symbol and, thus, keep the amount of storage tape used less than  $c \log_2 n$ , for some constant  $c$ .

Since  $\Sigma$  and  $\Lambda$  are symmetric in the statement of the lemma, this is sufficient to prove the lemma.

**THEOREM 5.** For any finite alphabet,  $\Sigma$ , with at least two elements, the following statements are equivalent:

- (1)  $M_E$  is accepted by some deterministic Turing machine within storage  $\log_2 n$ .
- (2) For any finite alphabet  $\Gamma$ , any set,  $A$ , of strings over  $\Gamma$  and any function  $L(n) \geq \log_2 n$ , if  $A$  is accepted by a nondeterministic Turing machine within storage  $L(n)$ , then  $A$  is accepted by some deterministic Turing machine within storage  $L(n)$ .

*Proof.* Statement (1) follows from statement (2) by Theorem 4.

Assume (1) holds and that  $A$  is a set accepted by some nondeterministic Turing machine,  $Z_N$ , within storage  $L(n) \geq \log_2 n$ . We will construct a deterministic machine,  $Z_D$ , which accepts  $A$  within the same storage bound  $L(n)$ . Assume  $L(n)$  is measurable. We will later indicate how this restriction may be eliminated. With each string  $w$  over the input alphabet of  $Z_N$  associate the maze  $\mathcal{M}(w) = (X, R, s, G)$ , where  $X$  is the set of all ID's of  $Z_N$  in which the nonblank portion of each work tape has length at most  $L(|w|)$ ,  $s$  is the start ID of  $Z_N$ ,  $G$  is the set of ID's in  $X$  which include an accepting state, and  $R$  is defined by:  $R(x, y)$  holds if, and only if,  $x \xrightarrow{w} y$ . Clearly,  $Z_N$  accepts  $w$  if, and only if,  $\mathcal{M}(w)$  is threadable. Let  $m(w)$  denote a coding of the maze  $\mathcal{M}(w)$ . The deterministic machine  $Z_D$  which accepts  $A$  operates as follows. Given an input  $w$ ,  $Z_D$  constructs  $m(w)$  on one of its storage tapes, and then simulates the machine of statement (1) to determine whether  $m(w)$  is in  $M_E$ . If it is, then  $Z_D$  accepts  $w$ . By Lemma 4, we may assume  $\Sigma$  is large enough so that all ID's of  $Z_N$  are strings over  $\Sigma$ . So  $m(w)$  is in  $M_E$  if, and only if,  $Z_N$  accepts  $w$ . Thus,  $Z_D$  accepts precisely the set  $A$ .  $m(w)$  is of the length at most  $h^{L(n)}$ , where  $n = |w|$  and  $h$  is a constant depending only on  $Z_N$ . Except for the tape used to store  $m(w)$ ,  $Z_D$  operates in storage  $\log_2 h^{L(n)}$  which is proportionate to  $L(n)$ .

The machine  $Z_D$  cannot write down the entire string,  $m(w)$ , at once and still work within the allotted storage. However, all that is necessary in order to simulate the machine of statement (1) is that  $Z_D$  be able to compute one symbol at a time of the string  $m(w)$  and keep track of the symbols position in  $m(w)$ . This it could do provided it could, within storage  $L(n)$ , generate  $m(w)$ , from right to left, one symbol at a time.

Our remarks so far applied to any coding  $m(w)$  of the maze  $\mathcal{M}(w)$ . We now fix  $m(w)$  to be the following coding of the maze  $\mathcal{M}(w)$

$$s[x_1 * y_1^1 * y_2^1 * \dots * y_{n(1)}^1][x_2 * y_1^2 * y_2^2 * \dots * y_{n(2)}^2]$$

$$\dots [x_l * y_1^l * \dots * y_{n(l)}^l] u_1 * u_2 * \dots * u_g$$

where  $x_1 < x_2 < x_3 < \dots < x_l$  for each  $i = 1, 2, \dots, l$ ,  $y_1^i < y_2^i < y_3^i < \dots < y_{n(i)}^i$  and  $u_1 < u_2 < \dots < u_g$ .  $<$  is the usual "less than" ordering on the natural numbers. Recall that the  $x$ 's,  $y$ 's, and  $u$ 's are strings over a finite alphabet and, thus, may be regarded as numbers in  $k$ -adic notation, for some  $k$ . By definition of  $\mathcal{M}(w)$ , the  $x$ 's,  $y$ 's,  $u$ 's, and  $s$  are ID's of  $Z_N$  of length less than  $cL(n)$ , where  $c$  is a constant depending only on  $Z_N$ .  $Z_D$  can generate  $m(w)$  symbol by symbol, in the allotted storage, as follows.  $Z_D$  can easily calculate  $s$ , the initial ID of  $Z_N$ . Having generated  $s$ ,  $Z_D$  next runs through in numerical order, all strings of length less than  $cL(n)$  until it finds one which is an ID of  $Z_N$ . This will be  $x_1$ . So  $Z_D$  has generated  $x_1$ . Next it produces all  $y$  such that  $x_1 \leftarrow_w y$ . In this way, it generates the  $y_j^1$ . Next it runs through, in numerical order, all strings of length less than  $cL(n)$  until it finds the first such string which is an ID of  $Z_N$  and is greater than  $x_1$ . This is  $x_2$ . It now may erase  $x_1$ . The only thing it need keep in storage at this point is  $x_2$ .  $Z_D$  then proceeds to generate the  $y_j^2$ . It generates  $x_3$  from  $x_2$  in the same way it computed  $x_2$  from  $x_1$ . It proceeds in this way until it finds the largest  $x_i$ . Having generated the largest  $x_i$  and its associated  $y$ 's, it then generates the  $u_i$ 's as follows. It runs through all strings of length at most  $cL(n)$  and checks to see which are ID's which include an accepting state. Those that do are the  $u_i$ . This completes the generating process and the proof for the case:  $L(n)$  is measurable.

If  $L(n)$  is not measurable, then  $Z_D$  cannot necessarily generate precisely the strings of length less than  $cL(n)$  within storage proportionate to  $L(n)$  and so the above procedure will not work. However, if  $Z_D$  were somehow given the value of  $L(n)$ , then by the above procedure it could determine whether or not  $w$  is accepted by  $Z_N$  within storage  $L(n)$ .  $Z_D$  uses the same sort of trick as was used for nonmeasurable functions in the proof of Theorem 1. That is, first it assumes  $L(n) = 1$ . If  $Z_N$  accepts the input  $w$  within storage  $L(n) = 1$ , then  $Z_D$  accepts  $w$ . If not,  $Z_D$  next tries  $L(n) = 2$ . If  $Z_N$  accepts within storage  $L(n) = 2$ , then  $Z_D$  does. If not,  $Z_D$  tries  $L(n) = 3$  next, and so forth. If  $Z_N$  accepts the input  $w$ , then  $Z_D$  will eventually find the correct value for  $L(n)$  and accept  $w$  within storage proportionate to  $L(n)$ .

An unsolved problem in the theory of tape complexity is whether there is some set  $A$  of strings and some function  $L(n) \geq \log_2 n$  such that  $A$  is accepted by some non-deterministic Turing machine within storage  $L(n)$  but accepted by no deterministic Turing machine within storage  $L(n)$ . Theorem 5 shows that if any such  $A$  and  $L(n)$  exist, then  $A = M_T$  and  $L(n) = \log_2 n$  will do.

## ACKNOWLEDGMENT

The author thanks Stephen A. Cook for his help and encouragement in this work.

## REFERENCES

1. M. BLUM, A machine-independent theory of the complexity of recursive functions, *J. Assoc. Comput. Mach.* **14** (1967), 322-336.
2. A. COBHAM, "The Recognition Problem for the Set of Perfect Squares." *IEEE Conference Record of the Seventh Annual Symposium on Switching and Automata Theory*, Berkeley, Calif., 1966, pp. 78-87.
3. J. E. HOPCROFT AND J. D. ULLMAN, "Formal Languages and Their Relation to Automata." Addison-Wesley, Reading, Mass., 1969.
4. J. E. HOPCROFT AND J. D. ULLMAN, Relations between time and tape complexities. *J. Assoc. Comput. Mach.* **15** (1968), 414-427.
5. S. Y. KURODA, Classes of languages and linear-bound automata. *Information and Control* **7** (1964), 207-223. (The relevant material also appears in [3], pp. 115-119.)
6. P. S. LANDWEBER, Three theorems on phrase structure grammars of type 1. *Information and Control* **6** (1963), 131-136. (The relevant material also appears in [3], pp. 115-119.)
7. P. M. LEWIS II, R. E. STEARNS, AND J. HARTMANIS, "Memory Bounds for the Recognition of Context-Free and Context-Sensitive Languages." *IEEE Conference Record on Switching Circuit Theory and Logical Design*, Ann Arbor, Mich., 1965, pp. 191-202. (The relevant material also appears in [3], pp. 162-164.)
8. E. M. MCCREIGHT AND A. R. MEYER, "Classes of Computable Functions Defined by Bounds on Computation: Preliminary Report." *Conference Record of ACM Symposium on Theory of Computing*, Marina del Rey, Calif., May 1969, pp. 79-88.
9. W. J. SAVITCH, "Deterministic Simulation of Non-deterministic Turing Machines (Detailed Abstract). *Conference Record of ACM Symposium on Theory of Computing*, Marina del Rey, Calif., May 1969, pp. 247-248.