Alternation

ASHOK K. CHANDRA, DEXTER C. KOZEN, AND LARRY J. STOCKMEYER

IBM Thomas J. Watson Research Center, Yorktown Heights, New York

ABSTRACT. Alternation is a generalization of nondeterminism in which existential and universal quantifiers can alternate during the course of a computation, whereas in a nondeterministic computation there are only existential quantifiers. Alternating Turing machines are defined and shown to accept precisely the recursively enumerable sets. Complexity classes of languages accepted by time- (space-) bounded alternating Turing machines are characterized in terms of complexity classes of languages accepted by space- (time-) bounded deterministic Turing machines. In particular, alternating polynomial time is equivalent to deterministic polynomial space and alternating polynomial space is equivalent to deterministic exponential time. Subrecursive quantifier hierarchies are defined in terms of time- or space-bounded alternating Turing machines by bounding the number of alternations allowed during computations. Alternating finite-state automata are defined and shown to accept only regular languages, although, in general, 2^{2^k} states are necessary and sufficient to simulate a k-state alternating finite automaton deterministically. Finally, it is shown that alternating pushdown automata are strictly more powerful than nondeterministic pushdown automata.

KEY WORDS AND PHRASES: alternation, complexity

CR CATEGORIES: 5.22, 5.25, 5.26

1. *Introduction*

The concept of nondeterminism has played a number of important roles in the theory of computation. For example, in formal language theory, certain classes of languages have been characterized in terms of acceptance by nondeterministic automata [10]; in complexity theory, nondeterminism has provided the basis for the important notion of NP-completeness [1, 5, 11]. The purpose of this paper is to describe and investigate a generalization of nondeterminism which we call *alternation.*

In the case of nondeterministic machines, for example, nondeterministic Turing acceptors [10], the transition rules allow a single machine configuration α to reach several configurations $\beta_1, \ldots, \beta_k$ in one step; by definition, the configuration α leads to acceptance if and only if there exists a successor β_i which leads to acceptance. In addition to these "existential branches," an alternating Turing machine can also make "universal branches" and "negating moves." In a universal branch, a configuration α can again reach several configurations $\beta_1, \ldots, \beta_k$ in one step, but now α leads to acceptance if and only if *all* successors $\beta_1, \ldots, \beta_k$ lead to acceptance. In a negating move, a configuration α has only one successor β, and α leads to acceptance

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Preliminary versions of this paper appeared in the Proceedings of the 17th Annual IEEE Symposium on the Foundations of Computer Science, Houston, Texas, 1976 [4, 13].

The work of the second author was done in part at Cornell University and was supported by the National Science Foundation under Grant DCR 75-09433.

Authors' address: IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.

© 1981 ACM 0004-5411/81/0100-0114 $00.75

iff β leads to rejection. A formalization of this idea yields the definition of the *alternating Turing machine* (ATM), given in Section 2. Because of the possibility of nonterminating computation paths, the formal definition of acceptance for ATMs is slightly more involved than the informal definition outlined above. The basic definition is similar to least-fixed-point definitions of recursion [22], but it is shown that, viewing the computation of an ATM as a tree, acceptance could be defined equivalently as the existence of a finite accepting subtree. It is then immediate that ATMs accept precisely the recursively enumerable sets. We also define time and space complexity for ATMs and show that negating moves can be eliminated with no loss of efficiency—hence the term *alternation* referring to the alternation of universal and existential branches.

In Section 3 we characterize the complexity classes of languages accepted by time- (space-) bounded alternating Turing machines in terms of complexity classes of languages accepted by space- (time-) bounded deterministic Turing machines. It is shown that alternating time and deterministic space are equivalent to within a squaring of the resource bound, and that alternating space $S(n)$ is equivalent to (the union over constants $c > 0$ of) deterministic time $c^{S(n)}$. Thus the deterministic complexity hierarchy

$$\text{LOGSPACE} \subseteq \text{PTIME} \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{EXPSPACE} \subseteq \cdots$$

shifts by exactly one level when alternation is introduced.

In Section 4 we consider ATMs with the restriction that the number of alternations (of blocks of existential branches with blocks of universal branches) is bounded, and we discuss the relationship between these devices and subrecursive quantifier hierarchies such as the polynomial-time hierarchy [24]. The "zero-degree" of the polynomial-time hierarchy is deterministic polynomial time. The concept of bounded alternation facilitates natural definitions of quantifier hierarchies based on other zero-degrees, such as logarithmic space or exponential time. We also give a generalization of a result of Savitch [20] that nondeterministic space $S(n)$ is contained in deterministic space $S(n)^2$; namely, the bounded quantifier hierarchy based on space $S(n)$ is entirely contained in deterministic space $S(n)^2$.

Alternation can be applied to classes of automata other than Turing machines. In Section 5 we investigate the power of alternating finite automata and alternating pushdown automata. We show that although alternating finite automata accept only regular languages, in general 2^{2^k} states are necessary and sufficient to simulate a k-state alternating finite automaton deterministically. Finally, we show that any language which can be accepted by a deterministic Turing machine in time c^n for some constant c can also be accepted by some alternating pushdown automaton; therefore, alternating pushdown automata are strictly more powerful than nondeterministic pushdown automata.

Although the ultimate usefulness of alternation remains to be seen, several applications of alternation have already been made. One goal of complexity theory is to classify the computational difficulty of problems, either by establishing explicit upper and lower bounds on the time or space required to solve the problem or by showing that the problem is *complete* in some complexity class (see, e.g., [1, 11, 24, 26, 27]). If the problem of interest intrinsically involves alternating quantifiers, it may be easier to classify the problem in terms of time- or space-bounded alternating machines and then, using the characterizations of Section 3, translate the classification to one in terms of deterministic machines. Two types of problems which involve quantifiers are decision problems in logic and problems concerning the existence of winning strategies in combinatorial games. By exploiting the equivalence between alternating

linear space and deterministic exponential time, Fischer and Ladner [6] prove that the validity problem for propositional dynamic logic requires exponential time to solve. For certain two-person combinatorial games, Stockmeyer and Chandra [25] and Kasai, Adachi, and Iwata [12] show that it requires exponential time to determine which player has a winning strategy. By employing alternating Turing machines, Berman [2] classifies precisely the decision problems for the first-order theories of real addition and integer addition, and Kozen [14] classifies the elementary theory of Boolean algebras. Ruzzo [19] has shown that the class of languages reducible to context-free languages in logspace can be characterized by a class of alternating Turing machines.

An alternating machine can also be viewed as a machine with (unbounded) parallelism in which processes communicate only with their parent or offspring. When in a configuration α with several successors $\beta_1, \ldots, \beta_k$, the machine spawns k independent offspring which run to completion, report acceptance or rejection back to their parent α, and then die. The parent α combines the answers in a simple way (by ANDing the answers if α is universal or by ORing the answers if α is existential), the resulting answer is passed up to α's parent, and so on. Investigation of the power of a variety of parallel machine models [4, 7, 9, 17, 21, 23] has led to the formulation of a "parallel computation thesis" [4, 7], which states that time on a parallel machine is polynomially related to space on a serial (Turing) machine. Since one of the characterizations of Section 3 says that alternating time and deterministic space are polynomially related, one implication of this thesis is that an alternating Turing machine is, to within a polynomial, among the most powerful types of parallel machines.

To conclude, the notion of alternation impacts several topics in computer science, including time and space complexity, logic, games, and parallelism. Alternation leads to succinct representations in certain instances such as finite automata, and it adds power in others such as pushdown automata. Certain problems seem more convenient to program using the construct of alternation, but we do not know whether alternation will find its way into programming languages or have a role to play in structured programming. Such questions present themselves for further research.

2. *Alternating Turing Machines*

An alternating Turing machine is like a nondeterministic Turing machine [cf. 10] with the addition of a function associating one of the three Boolean functions $\wedge$ (and), $\vee$ (or), and $\neg$ (not) with each nonfinal state.

Definition 2.1. An *alternating Turing machine* (ATM) is a seven-tuple

$$M = (k, Q, \Sigma, \Gamma, \delta, q_0, g),$$

where

k is the number of work tapes,
Q is a finite set of *states*,
Σ is a finite *input alphabet* ($\not\vdash \notin \Sigma$ is an endmarker),
Γ is a finite *work tape alphabet* ($\# \in \Gamma$ is the blank symbol),
$\delta \subseteq (Q \times \Gamma^k \times (\Sigma \cup \{\not\vdash\})) \times (Q \times (\Gamma - \{\#\})^k \times \{\text{left, right}\}^{k+1})$ is the *next move relation*,
$q_0 \in Q$ is the *initial state*,
$g: Q \to \{\wedge, \vee, \neg, \textbf{accept}, \textbf{reject}\}$.

If $g(q) = \wedge$ (respectively, $\vee$, $\neg$, **accept**, **reject**), then q is said to be a *universal* (respectively, *existential*, *negating*, *accepting*, *rejecting*) state.

The machine has a read-only input tape with endmarkers and k work tapes, initially blank. A *step* of M consists of reading one symbol from each tape, writing a symbol on each of the work tapes, moving each of the heads left or right one tape cell, and entering a new state, in accordance with the transition relation δ. Note that the machine cannot write the blank symbol.

Definition 2.2. A *configuration* of an alternating Turing machine M is an element of

$$C_M = Q \times \Sigma^* \times ((\Gamma - \{\#\})^*)^k \times \mathbf{N}^{k+1}$$

representing the state of the finite control, the input, the nonblank contents of the k work tapes, and $k+1$ head positions; see, for example, [10]. If q is the state associated with configuration α, then α is said to be a *universal* (respectively, *existential, negating, accepting, rejecting*) *configuration* if q is a universal (respectively, existential, negating, accepting, rejecting) state. The *initial configuration of M on input x* is

$$\sigma_M(x) = (q_0, x, \underbrace{\lambda, \ldots, \lambda}_{k}, \underbrace{0, \ldots, 0}_{k+1}),$$

where λ is the null string.

A configuration represents an instantaneous description of M at some point in a computation.

Definition 2.3. Given M, we write $\alpha \vdash \beta$ and say β is a *successor* of α if configuration β follows from configuration α in one step, according to the transition rules δ. The relation $\vdash$ is not necessarily single valued, since δ is not. We do, however, require that δ is such that accepting and rejecting configurations have no successors, universal and existential configurations have at least one, and negating configurations have exactly one. The reflexive transitive closure of $\vdash$ is denoted $\vdash^*$.

A *computation path* of M on x is a sequence $\alpha_0 \vdash \alpha_1 \vdash \cdots \vdash \alpha_n$, where $\alpha_0 = \sigma_M(x)$.

We wish to define acceptance for such machines in a way which will capture the following idea. A single process is started in the initial configuration $\sigma_M(x)$. Subsequently, if a process is in an existential configuration α, and $\beta_1, \ldots, \beta_m$ are all the configurations following from α in one step, then the process spawns m distinct offspring processes which concurrently try to determine whether any of the β_i lead to acceptance. Some of the offspring computations may be infinite, but if at least one is accepting, this information is reported back to the parent process waiting at α, and it in turn reports back to its parent process that α leads to acceptance. If a process is in a universal configuration, it must determine whether *all* its offspring lead to acceptance. If a process is in a negating configuration, it must determine whether its unique offspring leads to rejection. A process in an accepting (rejecting) configuration just reports success (failure) to its parent.

We could formalize this idea in terms of a recursive procedure for labeling configurations as 1 (leading to acceptance) or 0 (leading to rejection) and say M accepts x iff the start configuration is ever labeled 1. The recursive procedure would be

$$f(\alpha) = \begin{cases} \bigwedge\limits_{\alpha \vdash \beta} f(\beta) & \text{if } \alpha \text{ is universal,} \\ \bigvee\limits_{\alpha \vdash \beta} f(\beta) & \text{if } \alpha \text{ is existential,} \\ \neg f(\beta) & \text{where } \alpha \vdash \beta, \text{ if } \alpha \text{ is negating,} \\ 1 & \text{if } \alpha \text{ is accepting,} \\ 0 & \text{if } \alpha \text{ is rejecting.} \end{cases}$$

However, the usual semantics will not suffice, since we wish to allow a process to return to its parent in cases where, although some offspring have not returned, it has enough information to determine what its labeling must be. For example, if $\alpha \vdash \beta$, $\alpha \vdash \gamma$, α is universal, and $f(\beta) = 0$, then we want $f(\alpha) = 0$, regardless of whether γ is ever labeled. In order to accomplish this, we extend the definition of the Boolean operations $\wedge$, $\vee$, $\neg$ to domain $\{0, \perp, 1\}$ according to the following tables:

$\wedge$:	1	$\perp$	0
1	1	$\perp$	0
$\perp$	$\perp$	$\perp$	0
0	0	0	0

$\vee$:	1	$\perp$	0
1	1	1	1
$\perp$	1	$\perp$	$\perp$
0	1	$\perp$	0

$\neg$:	
0	1
$\perp$	$\perp$
1	0

Thus $\vee$ gives least upper bound in the ordering $0 < \perp < 1$, and $\wedge$ gives greatest lower bound. Note that $\vee$ and $\wedge$ are associative. Intuitively, $\perp$ represents an incomplete computation; for example, note that $0 \wedge \perp = 0$, as desired.

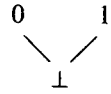
Let M be an alternating Turing machine, and let $x \in \Sigma^*$ be an input. A *labeling of configurations* is a map

$$l: C_M \to \{0, \perp, 1\}.$$

Let τ be the operator mapping labelings to labelings defined as follows:

$$\tau(l)(\alpha) = \begin{cases} \bigwedge_{\alpha \vdash \beta} l(\beta) & \text{if } \alpha \text{ is universal,} \\ \bigvee_{\alpha \vdash \beta} l(\beta) & \text{if } \alpha \text{ is existential,} \\ \neg l(\beta) & \text{if } \alpha \text{ is negating and } \alpha \vdash \beta, \\ 1 & \text{if } \alpha \text{ is accepting,} \\ 0 & \text{if } \alpha \text{ is rejecting.} \end{cases}$$

Let $\sqsubseteq$ be the partial ordering on labelings defined by extending the ordering



coordinatewise to functions from C_M to $\{0, \perp, 1\}$. It is easily checked that τ is monotone with respect to $\sqsubseteq$; that is, if $l \sqsubseteq l'$, then $\tau(l) \sqsubseteq \tau(l')$. Therefore τ has a least fixed point

$$l^* = \sup_{m \in \mathbf{N}} \tau^m(\Omega),$$

where the supremum is with respect to $\sqsubseteq$, where

$$\tau^0(l) = l,$$
$$\tau^{m+1}(l) = \tau(\tau^m(l)),$$

and where Ω is the $\sqsubseteq$-minimal labeling $\Omega(\alpha) = \perp$ for all α.

Definition 2.4. M *accepts* x iff $l^*(\sigma_M(x)) = 1$; M *rejects* x iff $l^*(\sigma_M(x)) = 0$; M *halts on* x iff it either accepts or rejects x; and

$$L(M) = \{x \in \Sigma^* \mid M \text{ accepts } x\}.$$

In the case that M has no negating states, other equivalent definitions of acceptance for alternating machines are given by Fischer and Ladner [6] and Ladner et al. [15].

We may consider a *nondeterministic Turing machine* to be an alternating machine with no universal or negating states, and a *deterministic Turing machine* to be a nondeterministic machine whose next move relation δ is single valued. The sets

accepted by nondeterministic Turing machines under this definition coincide with those of the classical definition [10], so alternating machines accept all recursively enumerable (r.e.) sets.

We wish to show that alternating machines accept only r.e. sets. To do this, we show that we can restrict our attention to labelings of finite subsets of C_M, similar to the labeling of finite subtrees of the computation tree as in [4], or limiting depth of recursion as in [13].

Let $C \subseteq C_M$, and let τ_C be the operator defined by

$$\tau_C(l)(\alpha) = \begin{cases} \tau(l)(\alpha) & \text{if } \alpha \in C, \\ \bot & \text{otherwise.} \end{cases}$$

Let

$$l_C^* = \sup_{m \in \mathbf{N}} \tau_C^m(\Omega).$$

τ_C is the same as τ but ignores configurations outside C. l_C^* exists, since τ_C is monotone with respect to $\sqsubseteq$, and is the least fixed point of τ_C.

C may be, for example, the set of configurations reachable from $\sigma_M(x)$ in t or fewer steps. In this case, l_C^* gives a labeling of the computation tree truncated to height t in the formalism of [4], or labelings of configurations when depth of recursion is limited to t in the formalism of [13].

Note that if C is finite, then l_C^* is computable. We can start with the restriction of Ω to C and then successively apply τ_C until no more changes occur. This must happen within $|C|$ steps, since τ_C is monotone. We need only look at configurations in C, since we know $\tau_C^m(\Omega)$ is always $\bot$ outside of C. This will be the basis of several simulation algorithms in later sections.

We need to establish some elementary properties of l_C^*.

LEMMA 2.1. *If $C \subseteq D$, then $l_C^* \sqsubseteq l_D^*$ and $\tau_C^m(\Omega) \sqsubseteq \tau_D^m(\Omega)$ for all m.*

PROOF. The proof is by induction on m, using the monotonicity of τ_C and τ_D. □

For $\alpha \in C_M$ and $m \in \mathbf{N}$ define

$$C(\alpha, m) = \{\beta \in C_M \mid \alpha \vdash^* \beta \text{ in } m \text{ or fewer steps}\}.$$

LEMMA 2.2. $l_{C(\alpha,m)}^*(\alpha) = \tau^{m+1}(\Omega)(\alpha)$.

PROOF. By preceding remarks, $\tau_{C(\alpha,m)}$ need be applied to Ω at most $m+1$ times before the fixed point is reached; that is,

$$l_{C(\alpha,m)}^*(\alpha) = \tau_{C(\alpha,m)}^{m+1}(\Omega)(\alpha).$$

By Lemma 2.1, $\tau_{C(\alpha,m)}^{m+1}(\Omega)(\alpha) \sqsubseteq \tau^{m+1}(\Omega)(\alpha)$. The intuition behind the other direction, $\tau^{m+1}(\Omega)(\alpha) \sqsubseteq \tau_{C(\alpha,m)}^{m+1}(\Omega)(\alpha)$, is that the value of $\tau^{m+1}(\Omega)(\alpha)$ is not affected by the labeling of configurations which cannot be reached from α within m or fewer steps. The formal proof, which we leave to the reader, is a straightforward induction on m. □

LEMMA 2.3. *For every α there is a finite set $C \subseteq C_M$ such that $l_C^*(\alpha) = l^*(\alpha)$.*

PROOF. Let $f(\alpha)$ be the least m such that $\tau^{m+1}(\Omega)(\alpha) = l^*(\alpha)$. By Lemma 2.2, $C(\alpha, f(\alpha))$ is the desired finite set. □

THEOREM 2.4. *The family of sets accepted by alternating Turing machines is exactly the family of r.e. sets.*

PROOF. Such machines accept all r.e. sets since every nondeterministic machine may be considered an alternating machine, by foregoing remarks.

For the converse, if M and x are given, enumerate finite subsets C of C_M, construct l^*_C, and accept if ever $l^*_C(\sigma_M(x)) = 1$. If such a C exists, then M accepts x by Lemma 2.1, and if M accepts x, then such a C exists by Lemma 2.3. □

The definition of time and space usage arises naturally.

Definition 2.5. Let M be an ATM, and let $s, t \in \mathbf{N}$. *M accepts x in time t* if $l^*_C(\sigma_M(x)) = 1$, where

$$C = C(\sigma_M(x), t) = \{\beta \mid \sigma_M(x) \vdash^* \beta \text{ in } t \text{ or fewer steps}\}.$$

M accepts x in space s if $l^*_D(\sigma_M(x)) = 1$, where

$$D = \{\alpha \mid \text{space}(\alpha) \leq s\},$$

where space(α) is the sum of the lengths of the nonblank work-tape contents in configuration α.

Let T and S be functions from $\mathbf{N}$ to the real numbers. *M accepts in time $T(n)$* (respectively, *space $S(n)$*) provided that for each $x \in L(M)$, M accepts x in time at most $T(|x|)$ (respectively, space at most $S(|x|)$), where $|x|$ denotes the length of x.

Again, the classical definitions of time- and space-bounded acceptance by deterministic and nondeterministic Turing machines are equivalent to ours when we consider a nondeterministic machine to be an ATM with no universal or negating states, and a deterministic machine to be a nondeterministic machine with a single-valued next move relation.

ASPACE($S(n)$) (respectively, DSPACE($S(n)$), NSPACE($S(n)$)) denotes the class of languages accepted by alternating (respectively, deterministic, nondeterministic) Turing machines which accept in space $S(n)$. The definitions of ATIME($T(n)$), DTIME($T(n)$), and NTIME($T(n)$) are analogous. In particular, let

$$\begin{aligned} \text{ALOGSPACE} &= \text{ASPACE}(\log n), \\ \text{APTIME} &= \bigcup \text{ATIME}(n^c), \\ \text{APSPACE} &= \bigcup \text{ASPACE}(n^c), \\ \text{AEXPTIME} &= \bigcup \text{ATIME}(2^{n^c}), \\ \text{AEXPSPACE} &= \bigcup \text{ASPACE}(2^{n^c}), \end{aligned}$$

where unions are over all constants $c > 0$. Similarly, the notations LOGSPACE, PTIME, PSPACE , etc., are defined with respect to deterministic complexity, and NLOGSPACE, NPTIME, etc., are defined with respect to nondeterministic complexity.

The following theorem illustrates why we have chosen to call these automata alternating Turing machines.

THEOREM 2.5. *For every ATM M there is an ATM N such that N has no negating states, $L(M) = L(N)$, and for all $x \in L(M)$ and all s, $t \geq 0$, if M accepts x in time t (space s), then N accepts x in time t (space s).*

PROOF. The simulating machine will remember in its finite control the parity of the number of negations that have been encountered. When the parity is odd, the machine will compute $\wedge$ instead of $\vee$, accept instead of reject, etc. In effect, the negations are pushed down to the final states by deMorgan's laws.

Let M be any alternating Turing machine. Let N be another machine whose finite control consists of two copies of the finite control of M,

$$Q^+ = \{q^+ \mid q \in Q\} \quad \text{and} \quad Q^- = \{q^- \mid q \in Q\}.$$

Q^+ and Q^- will be duals. If $q \in Q$ is a universal (existential, negating, accepting, rejecting) state of M, then q^+ is a universal (existential, existential, accepting, rejecting) state of N, and q^- is an existential (universal, universal, rejecting, accepting) state of N. The transition relation δ of N is defined so that for any input x and configurations $\alpha \vdash \beta$, if α is not a negating configuration, then

$$\alpha^+ \vdash \beta^+ \qquad \text{and} \qquad \alpha^- \vdash \beta^-;$$

otherwise,

$$\alpha^+ \vdash \beta^- \qquad \text{and} \qquad \alpha^- \vdash \beta^+,$$

where α^+ (respectively, α^-) is α with state q^+ (respectively, q^-) substituted for state q. The initial state of N is q_0^+.

For $C \subseteq C_M$, let

$$C' = \{\alpha^+, \alpha^- \mid \alpha \in C\}.$$

A straightforward inductive argument shows that for all m, C, and α,

$$\tau_C^m(\Omega)(\alpha) = \tau_{C'}^m(\Omega)(\alpha^+) = \neg\tau_{C'}^m(\Omega)(\alpha^-);$$

thus

$$l_C^*(\alpha) = l_{C'}^*(\alpha^+) = \neg l_{C'}^*(\alpha^-).$$

Taking $C = C_M$ and noting that $\sigma_N(x) = \sigma_M^+(x)$, it follows that $L(M) = L(N)$. To see that space is preserved, take $C = \{\alpha \mid \text{space}(\alpha) \leq s\}$. To see that time is preserved, take $C = C(\sigma_M(x), t)$. □

The following theorem states that for honest resource bounds (see [10, Sec. 10.6]) we can restrict attention to computations for which all computation paths terminate.

THEOREM 2.6

(a) If $T(n) \geq n$ is constructible in time $O(T(n))$, and if the ATM M accepts in time $T(n)$, then there is an ATM N such that $L(M) = L(N)$ and all computation paths of N on any input x are of length at most $O(T(|x|))$.

(b) If $S(n) \geq \log n$ is constructible in space $S(n)$, and if the ATM M accepts in space $S(n)$, then there is an ATM N such that $L(M) = L(N)$, all computation paths of N on any input x are of length at most $c^{S(|x|)}$ for some constant $c > 0$, and all configurations α reachable from $\sigma_N(x)$ satisfy space$(\alpha) \leq S(|x|)$.

PROOF

(a) By Theorem 2.5 we may assume without loss of generality that M has no negating states. N on input x with $n = |x|$ will first construct $T(n)$ on an extra tape, then simulate M, counting one on its extra tape for each simulated step of M. If the counter runs out before the simulation reaches a halting configuration of M, then N rejects. The restriction on the length of computation paths of N is clearly satisfied, so it remains to show that $L(N) = L(M)$.

After $T(n)$ has been marked off and while N is engaged in the simulation of M, the configurations of N may be represented as $\langle \alpha, t \rangle$, where $\alpha \in C_M$ and t represents the time left on the counter. By the definition of N, $\langle \alpha, t \rangle \vdash_N \langle \beta, t - 1 \rangle$ iff $\alpha \vdash_M \beta$ and $l^*(\langle \alpha, t \rangle) = 0$ for $t < 0$. Clearly N accepts x iff $l^*(\langle \sigma_M(x), T(n) \rangle) = 1$, since the path from $\sigma_N(x)$ to $\langle \sigma_M(x), T(n) \rangle$ is deterministic (it only involves constructing $T(n)$).

Let $\equiv$ be the equivalence relation on $\{0, \perp, 1\}$ generated by $0 \equiv \perp$. Note that $\wedge$ and $\vee$ preserve $\equiv$ (i.e., if $u \equiv v$ and $z \equiv w$, then $u \wedge z \equiv v \wedge w$ and $u \vee z \equiv v \vee w$).

Since M has no negating states, it should be evident that

$$l^*(\langle \sigma_M(x), T(n)\rangle) \equiv l^*_{C(\sigma_M(x),T(n))}(\sigma_M(x));$$

that is, N simulates M restricted to configurations in $C(\sigma_M(x), T(n))$ provided that we view 0 and $\perp$ as equivalent. But since M accepts in time $T(n)$, it follows that M accepts x iff N accepts x.

(b) Let c be a constant such that for all n, $c^{S(n)}$ is an upper bound on the number of configurations $\alpha \in C_M$ with space$(\alpha) \leq S(n)$. Recall that for any $C \subseteq C_M$, $l^*_C = \tau^m_C(\Omega)$ where m is the cardinality of C. It follows that M accepts in time $c^{S(n)}$.

N on input x will first construct $S(n)$ on a tape. On one track of that tape, N initializes a counter to $c^{S(n)}$; the counter is written in c-ary notation and occupies space $S(n)$. Now N simulates M using the $S(n)$ storage on another track of the tape. After each simulated step, N decrements the counter by one. If either the counter reaches zero or M attempts to use more than $S(n)$ tape squares, then N rejects. The proof that $L(N) = L(M)$ is similar to part (a). □

3. *Complexity*

In this section we study the complexity of alternating machines and establish fundamental relationships between alternating and deterministic complexity. In the last section we defined time and space for alternating machines and showed that without loss of efficiency in either time or space we could restrict our attention to machines with no negating states. We shall henceforth assume all machines are of this form.

The following four theorems are the main results of this section. They relate alternating time and space to deterministic time and space.

THEOREM 3.1. *If $S(n) \geq n$, then $NSPACE(S(n)) \subseteq \bigcup_{c>0} ATIME(c \cdot S(n)^2)$.*

THEOREM 3.2. *If $T(n) \geq n$, then $ATIME(T(n)) \subseteq DSPACE(T(n))$.*

THEOREM 3.3. *If $S(n) \geq \log n$, then $ASPACE(S(n)) \subseteq \bigcup_{c>0} DTIME(c^{S(n)})$.*

THEOREM 3.4. *If $T(n) \geq n$ and $c > 0$, then*

$$DTIME(T(n)) \subseteq ASPACE(c \cdot \log T(n)).$$

Theorems 3.3 and 3.4 can be combined into the following characterization.

COROLLARY 3.5. *If $S(n) \geq \log n$, then $ASPACE(S(n)) = \bigcup_{c>0} DTIME(c^{S(n)})$.*

The above results not only characterize the power of alternation but also reveal a striking relationship between Turing machine time and space.

COROLLARY 3.6

$$\begin{aligned} EXPSPACE &= AEXPTIME, \\ EXPTIME &= APSPACE, \\ PSPACE &= APTIME, \\ PTIME &= ALOGSPACE. \end{aligned}$$

That is, the deterministic hierarchy

$$LOGSPACE \subseteq PTIME \subseteq PSPACE \subseteq EXPTIME \subseteq EXPSPACE \subseteq \cdots$$

shifts by exactly one level when alternation is introduced.

PROOF OF THEOREM 3.1. The technique is similar to Savitch's [20] for the deterministic simulation of nondeterministic space-bounded computations. It is based on

the fact that if M is a nondeterministic Turing machine which accepts in space $S(n)$, then M accepts x iff there is a string of encodings of configurations of M on x, each of length at most $S(|x|)$, such that the first is the initial configuration, the last is an accepting configuration, and each intermediate configuration follows from its predecessor according to the transition rules of M. If the alphabet used to encode configurations is of cardinality c, and if such a string exists, there must be one of length at most $c^{S(n)}$.

We first prove the result for $S(n)$ constructible in deterministic time $O(S(n)^2)$. The alternating machine M' simulating M on input x, $|x| = n$, first marks off $S(n)$ tape in time $O(S(n)^2)$. It then writes down the initial configuration of M and guesses (using existential branching) an accepting configuration of M of length at most $S(n)$. It then writes down $c^{S(n)}$ in c-ary notation and calls a subroutine which takes inputs of the form (α, β, k), where α and β are configurations of M and k is a number in c-ary notation, $0 \leq k \leq c^{S(n)}$. The subroutine checks whether $\alpha \vdash^* \beta$ in k or fewer steps. If $k > 1$, it guesses a middle configuration γ (using existential branching) of length at most $S(n)$, then verifies in parallel (using universal branching) that $\alpha \vdash^* \gamma$ in $k/2$ or fewer steps and $\gamma \vdash^* \beta$ in $k/2$ or fewer steps, by calling itself recursively with inputs $(\alpha, \gamma, k/2)$ and $(\gamma, \beta, k/2)$. If $k \leq 1$, it checks whether $\alpha = \beta$ or $\alpha \vdash \beta$.

The entire computation requires $\log_2 c^{S(n)} = O(S(n))$ recursive calls, and each call takes $O(S(n))$ steps, or $O(S(n)^2)$ in all.

To remove the restriction of constructibility, have the simulating machine existentially guess whether $S(n) = 1, 2, \ldots$ and perform the above computation for the guessed value. If M accepts x, then the correct value of $S(n)$ will be guessed in time $S(n)$; thus M' will still accept in time $O(S(n)^2)$. If M does not accept x, then by Lemma 2.1, M' does not accept x for any guessed values of $S(n)$. □

PROOF OF THEOREM 3.2. Assume $T(n)$ is tape constructible, and let M be an alternating machine which accepts in time $T(n)$. A deterministic machine M' simulating M on input x first constructs $T(n)$ and writes down the initial configuration $\sigma_M(x)$. It then builds and traverses the tree with vertices labeled with

$$C = \{\beta \mid \sigma_M(x) \vdash^* \beta \text{ in } T(n) \text{ or fewer steps}\},$$

and edges $\vdash$, calculating $l_C^*(\beta)$ in postorder; recall that for $\alpha \in C$,

$$l_C^*(\alpha) = \begin{cases} \bigwedge\limits_{\alpha \vdash \beta} l_C^*(\beta) & \text{if } \alpha \text{ is universal,} \\ \bigvee\limits_{\alpha \vdash \beta} l_C^*(\beta) & \text{if } \alpha \text{ is existential,} \\ 1 & \text{if } \alpha \text{ is accepting,} \\ 0 & \text{if } \alpha \text{ is rejecting,} \end{cases}$$

since l_C^* is a fixed point of τ_C. Finally, M' accepts iff $l_C^*(\sigma_M(x)) = 1$.

At any point in the computation, if M' is visiting a node of the tree labeled α, then only α and a string representing the position of α in the tree need appear on the tape. The string may be a d-ary numeral, where d is the maximum outbranching of any configuration of M, denoting the unique path from the root to that vertex. Such a string need be of length at most $T(n)$.

To remove the assumption of constructibility, M' may iterate the computation above for successive values $T(n) = 1, 2, \ldots$. If M accepts x, then one such attempt (the one for the correct value of $T(n)$) results in acceptance. If M does not accept x, then no such attempt will, by Lemma 2.1. □

The restrictions $S(n) \geq n$ and $T(n) \geq n$ in the statements of Theorems 3.1 and 3.2

can be relaxed to $S(n) \geq \log n$ and $T(n) \geq \log n$ provided alternating machines are equipped with a device for reading input symbols without scanning the entire input tape. One way to do this is to allow alternating machines to write down a number i in binary, taking time $\log i$, and then enter a state requesting the ith input symbol. With this convention it is not difficult to see that the proofs of Theorems 3.1 and 3.2 still work in the cases $S(n) \geq \log n$ and $T(n) \geq \log n$, since the input x need not appear explicitly in configurations. The input appears only on the input tape and its symbols are accessed as needed.

PROOF OF THEOREM 3.3. Suppose M is an ATM which accepts in space $S(n)$, where $S(n)$ is constructible in time exponential in $S(n)$. A deterministic machine M' on input x first constructs $S(n)$, $n = |x|$, and then calculates $l_C^*(\sigma_M(x))$, where

$$C = \{\alpha \in C_M \mid \text{space}(\alpha) \leq S(n)\}.$$

l_C^* is calculated by writing down all configurations in C (there are at most $b^{S(n)}$ of them, for some constant b), labeling each configuration $\perp$ initially, then successively applying τ_C until no more changes occur. The resulting labeling is l_C^*. M' then accepts if $l_C^*(\sigma_M(x)) = 1$.

Since τ_C is monotone, it need be applied at most $b^{S(n)}$ times. Thus M' makes at most $b^{S(n)}$ passes over a tape of length at most $S(n) \cdot b^{S(n)}$, so M' runs in time $c^{S(n)}$ for some constant c.

The case in which $S(n)$ is not constructible in time exponential in $S(n)$ is handled as in the proof of Theorem 3.2. In this case, the running time of M' on inputs accepted by M is at most

$$\sum_{m=1}^{S(n)} c^m,$$

which is still at most $d^{S(n)}$ for some constant d. □

PROOF OF THEOREM 3.4. Let M be a deterministic Turing machine which accepts in time $T(n)$. By increasing the time to $T'(n) = d \cdot T(n)^2$ for some constant d, we can assume that (i) M has only one tape which is one-way infinite to the right, (ii) the input word is initially written left-justified on the otherwise blank tape, (iii) for all inputs x, M does not halt on x, and (iv) M *accepts* an input x by entering a designated state q_a at some time during the computation on x. (If x is accepted, q_a is entered on or before step $T'(|x|)$.)

Say that M has states Q and tape alphabet Γ, and let $\Delta = Q \cup \Gamma \cup \{\$\}$. Let x be an input, and let $n = |x|$. The computation of M on x is an (infinite) sequence of configurations $\alpha_0, \alpha_1, \alpha_2, \ldots$. We represent each configuration as an (infinite) word of the form $\$\mu q\nu\#\#\# \cdots$, where $q \in Q$ and $\mu\nu \in (\Gamma - \{\#\})^*$; the meaning is that $\mu\nu$ is written on the nonblank portion of the tape, and the machine is in state q scanning the first symbol of ν. For example, $\$q_0x\#\#\# \cdots$ represents the initial configuration α_0. For $t, j \geq 0$, let $\gamma_{t,j} \in \Delta$ be the jth symbol of the representation of α_t. Since M is deterministic, it is easy to see that there is a partial function $\text{Next}_M: \Delta^4 \to \Delta$ such that

$$\gamma_{t,j} = \text{Next}_M(\gamma_{t-1,j-1}, \gamma_{t-1,j}, \gamma_{t-1,j+1}, \gamma_{t-1,j+2})$$

for all $t, j \geq 1$. Next_M depends on M but not on x. Also, M accepts x iff there is a t and j with $1 \leq t, j \leq T'(n)$ such that $\gamma_{t,j} = q_a$.

The alternating machine M' which simulates M first guesses t and j using existential branching and then checks whether or not $\gamma_{t,j} = q_a$ by working backward through the computation of M. At each stage of the checking procedure, M' has integers t

and j and a symbol $z \in \Delta$, and the goal is to check whether or not $\gamma_{t,j} = z$ and accept or reject accordingly. To check that $\gamma_{t,j} = z$ is easy if either $j = 0$ (for then z must be \$) or $t = 0$ (because the jth symbol of $\$q_0x\#\#\# \cdots$ can be found directly). To check that $\gamma_{t,j} = z$ for $t, j \geq 1$, M' guesses four symbols $z_{-1}, z_0, z_1, z_2 \in \Delta$ using existential branching. If $z \neq \text{Next}_M(z_{-1}, z_0, z_1, z_2)$, then M' rejects. If $z = \text{Next}_M(z_{-1}, z_0, z_1, z_2)$, then M' chooses an integer k, $-1 \leq k \leq 2$, using universal branching and repeats the checking procedure to check whether or not $\gamma_{t-1,j+k} = z_k$.

The space required for this procedure is dominated by the space required to record the integers t and j. These integers are $O(T(n)^2)$. Hence for each $c > 0$ there is an integer b such that space $c \cdot \log(T(n))$ suffices when the integers are written in b-ary notation. □

4. *Hierarchies*

In this section we give a characterization of quantifier alternation hierarchies in terms of alternating machines.

Definition 4.1. Let M be an alternating Turing machine with no negating states, and let x be an input. We say M is *$A(n)$-alternation bounded* on x if whenever

$$\sigma_M(x) = \alpha_0 \vdash^* \alpha_1 \vdash^* \alpha_2 \vdash^* \cdots \vdash^* \alpha_m,$$

and α_i is a universal configuration iff α_{i+1} is an existential configuration for $0 \leq i \leq m - 1$, then $m < A(|x|)$.

In other words, any $\vdash$-path out of $\sigma_M(x)$ alternates universal and existential configurations at most $A(|x|) - 1$ times.

Definition 4.2. For $k \geq 1$, a Σ_k-machine (respectively, Π_k-machine) is a k-alternation bounded alternating machine M such that the initial state q_0 is existential (respectively, universal).

For example, a Σ_1-machine is a nondeterministic Turing machine. By convention, a Σ_0 or Π_0 machine is a deterministic machine.

Definition 4.3. $A\Sigma_k^p$ (respectively, $A\Pi_k^p$) is the class of sets accepted by Σ_k- (respectively, Π_k-) machines which accept in polynomial time.

For example,

$$\begin{aligned} A\Sigma_0^p &= A\Pi_0^p = \text{PTIME}, \\ A\Sigma_1^p &= \text{NPTIME}, \\ A\Pi_1^p &= \text{co-NPTIME}. \end{aligned}$$

Let Σ_k^p, Π_k^p denote the classes of the polynomial-time hierarchy, as defined by Stockmeyer [24]. The following theorem will perhaps aid in the placement of natural problems in this hierarchy.

THEOREM 4.1. *$\Sigma_k^p = A\Sigma_k^p$ and $\Pi_k^p = A\Pi_k^p$.*

PROOF. The proof is by induction on k. It is a straightforward application of well-known techniques (see, for example, [27]) and is left to the reader. □

The above theorem also gives us new complete problems for Σ_k^p and Π_k^p, namely,

$$\begin{aligned} S_k^p &= \{\bar{M}\$x\$^{3|\bar{M}|t} \mid M \text{ is a } \Sigma_k\text{-machine which accepts } x \text{ in time } t\}, \\ P_k^p &= \{\bar{M}\$x\$^{3|\bar{M}|t} \mid M \text{ is a } \Pi_k\text{-machine which accepts } x \text{ in time } t\}, \end{aligned}$$

where $\bar{M}$ is a suitable encoding of the alternating machine M. The construction is a

straightforward generalization of the case for Σ_1^p which appears in [8, Th. 6], once we observe that there is a universal alternating Turing machine which makes the same sequence of alternations as the machine it is simulating.

By changing the resource bounds on alternation bounded machines, we get other hierarchies. For example, the logspace hierarchy defined by

$$A\Sigma_k^{\log} = \{L(M) \mid M \text{ is a } \Sigma_k\text{-machine which accepts in space } \log n\},$$
$$A\Pi_k^{\log} = \{L(M) \mid M \text{ is a } \Pi_k\text{-machine which accepts in space } \log n\},$$

is analogous to the polynomial-time hierarchy in many ways. Some of its properties are listed below:

(1) $A\Sigma_k^{\log} \cup A\Pi_k^{\log} \subseteq A\Sigma_{k+1}^{\log} \cap A\Pi_{k+1}^{\log}$;
(2) $A\Sigma_k^{\log} \subseteq$ PTIME (since PTIME = ALOGSPACE);
(3) $S_k^{\log}$ (respectively, $P_k^{\log}$) $= \{\bar{M}\$x\$^s \mid M$ is a Σ_k- (respectively, Π_k-) machine which accepts x in space $\log(s)\}$ is complete for $A\Sigma_k^{\log}$ (respectively, $A\Pi_k^{\log}$); and
(4) $\bigcup_{k\geq 0} S_k^{\log}$ is complete for PTIME (in the same way that $\bigcup_{k\geq 0} S_k^p$ is complete for PSPACE).

The following theorem uses the result of Savitch [20] that NSPACE($S(n)$) $\subseteq$ DSPACE($S(n)^2$) and may be viewed as a strengthening of that result.

THEOREM 4.2. (A. BORODIN [3]). *If M is $S(n)$-space bounded and $A(n)$-alternation bounded with $S(n) \geq \log n$, then M can be simulated by a deterministic machine N of space complexity $A(n)S(n) + S(n)^2$.*

PROOF. Assume first that $S(n)$ is tape-constructible. We outline a recursive procedure MAIN for determining whether M accepts x. The procedure uses $S(n)^2$ global storage, plus $S(n)$ local storage for each instantiation, but the depth of recursion is limited to $A(n)$. Thus if the procedure is implemented on a Turing machine in the obvious way, that is, with a stack to preserve local storage across a call, then at most $A(n)S(n)$ space is needed for the local storage.

MAIN takes one input parameter, a configuration α of M with space(α) $\leq S(n)$, and it returns the value of $l_C^*(\alpha)$, where

$$C = \{\beta \mid \text{space}(\beta) \leq S(n)\}.$$

Thus to check whether M accepts x, N calls MAIN with parameter $\sigma_M(x)$ and accepts iff $l_C^*(\sigma_M(x)) = 1$.

First we describe a subroutine PATH which takes two parameters α, β, both configurations of M. If α is universal (respectively, existential), PATH determines whether there is a computation path from α to β such that all configurations appearing on the path (with the possible exception of β) are universal (respectively, existential) and lie in C. This can be done nondeterministically in space $S(n)$ just by guessing the path. By the above-mentioned result of Savitch, it can be done deterministically in space $S(n)^2$. PATH uses the $S(n)^2$ global storage for this purpose.

Now we describe the action of MAIN on input α. If α is an accepting or rejecting configuration or not in C, the procedure returns immediately with 1, 0, or $\perp$, respectively. If α is an existential configuration, $\alpha \in C$, note that

$$l_C^*(\alpha) = \bigvee l_C^*(\beta),$$

where the disjunction is taken over the set $\{\beta \mid \alpha \vdash^* \beta$ in such a way that all configurations along the path are existential and in C, and β is not existential$\}$. MAIN now writes down each nonexistential β in C successively and calls PATH to

check whether $\alpha \vdash^* \beta$ through only existential configurations, and if so, calls itself recursively with parameter β to determine $l_C^*(\beta)$. If no such β exists, then MAIN just returns the value $\perp$.

Similarly, if α is universal, $\alpha \in C$, then

$$l_C^*(\alpha) = \wedge\, l_C^*(\beta),$$

where the conjunction is taken over the set $\{\beta \mid \alpha \vdash^* \beta$ through only universal configurations in C, and β is not universal$\}$. In this case MAIN successively writes down all nonuniversal β, space$(\beta) \leq S(n)$, and calls PATH to check whether $\alpha \vdash^* \beta$ through only universal configurations in C; if so, it calls itself recursively with parameter β to determine $l_C^*(\beta)$. For universal α, MAIN must also check that no computation path out of α either (i) loops infinitely on $S(n)$ tape through only universal configurations, or (ii) reaches a configuration of space $S(n) + 1$ through only universal configurations. It does this by (i) calling PATH(α, β) and PATH(β, β) for all universal β of space $S(n)$; (ii) calling PATH(α, β) for all β of space $S(n) + 1$.

The depth of recursion is at most $A(n)$, since each recursive call of MAIN corresponds to another alternation, and if α is the parameter to a particular instantiation of MAIN, then only α need be preserved across recursive calls.

The case of $S(n)$ not tape-constructible is handled as in the proof of Theorem 3.2. That is, machine N successively iterates $s = 1, 2, \ldots$, and for each value s tries to verify that $l_C^*(\sigma_M(x)) = 1$, where $C = \{\beta \mid \text{space}(\beta) \leq s\}$. □

The following corollary generalizes the result of Savitch [20]; his result corresponds to the case $k = 1$.

COROLLARY 4.3. *For any k, if M is a k-alternation bounded alternating Turing machine which accepts in space $S(n) \geq \log n$, then $L(M) \in DSPACE(S(n)^2)$.*

Another corollary is that the entire logspace hierarchy is contained in DSPACE$((\log n)^2)$. This is perhaps surprising, in view of the fact that when the bound on the number of alternations is removed the resulting class is PTIME.

COROLLARY 4.4. $\bigcup_{k \geq 0} \Sigma_k^{log} \subseteq DSPACE((\log n)^2)$.

5. *Alternation in Other Automata*

5.1 ALTERNATING FINITE AUTOMATA. It is known that a k-state nondeterministic finite automaton can be simulated with a 2^k-state deterministic finite automaton [10, 18] and that 2^k states are necessary in certain cases [16]. We define alternating finite automata and show that they accept only regular sets. Furthermore, 2^{2^k} states are sufficient in general to simulate a k-state alternating finite automaton deterministically, and there are cases for which 2^{2^k} states are necessary.

Definition 5.1. An *alternating finite automaton* is a five-tuple,

$$P = (Q, \Sigma, q_1, F, g),$$

where

Q is a finite set of *states* $\{q_1, \ldots, q_k\}$,
Σ is a finite *input alphabet*,
$q_1 \in Q$ is the *start state*,
$F \subseteq Q$ are the *final states*, and
$g: Q \to (\Sigma \times B^k \to B)$,

where B denotes the set $\{0, 1\}$.

The function g associates with each state q_i a Boolean valued function $g(q_i)$, or g_i for short,

$$g_i : \Sigma \times B^k \to B.$$

One can think of g_i as a function which, given some input symbol and a Boolean value associated with each of the k states, computes a new Boolean value to be associated with the state q_i.

Let $\mathbf{u}$ denote a k-tuple of Boolean values $\langle u_1, u_2, \ldots, u_k \rangle$. Let π_i be the ith projection, $\pi_i(\mathbf{u}) = u_i$.

Let $\mathbf{f}$ denote the characteristic vector of F, that is,

$$\pi_i(\mathbf{f}) = \begin{cases} 1 & \text{if } q_i \in F, \\ 0 & \text{if } q_i \notin F. \end{cases}$$

Define $H_i : \Sigma^* \to (B^k \to B)$, $1 \le i \le k$, inductively as follows:

$$H_i(\lambda) = \pi_i,$$
$$H_i(ax)(\mathbf{u}) = g_i(a, H_1(x)(\mathbf{u}), \ldots, H_k(x)(\mathbf{u})),$$

where $a \in \Sigma$, $x \in \Sigma^*$, and $\lambda \in \Sigma^*$ is the null string.

$H_i(x)(\mathbf{f})$ is meant to correspond to the $l^*(\alpha)$ of the previous sections. That is, $H_i(\lambda)(\mathbf{f}) = 1$ iff q_i is a final state, and if ax is the input remaining, a process in state q_i scans a and splits into k independent parallel processes which run to completion, determining the values of $H_j(x)(\mathbf{f})$, $1 \le j \le k$; then $g_i(a, \cdot)$ is applied to these values to get $H_i(ax)(\mathbf{f})$. The following definition is then the natural analog of $l^*(\sigma_M(x)) = 1$.

Definition 5.2 *P accepts x* iff $H_1(x)(\mathbf{f}) = 1$.

The H_i are defined recursively "inside out." For technical reasons we wish to define a similar function "outside in." Let $G_i : \Sigma^* \to (B^k \to B)$, $1 \le i \le k$, be defined by

$$G_i(\lambda) = \pi_i,$$
$$G_i(xa)(\mathbf{u}) = G_i(x)(g_1(a, \mathbf{u}), \ldots, g_k(a, \mathbf{u})).$$

LEMMA 5.1. $G_i = H_i$, $1 \le i \le k$.

PROOF. By definition we have that $G_i(\lambda) = H_i(\lambda) = \pi_i$, and for all $x, y \in \Sigma^*$, $a \in \Sigma$, $\mathbf{u} \in B^k$,

$$\begin{aligned} &G_i(xa)(H_1(y)(\mathbf{u}), \ldots, H_k(y)(\mathbf{u})) \\ &\quad = G_i(x)(g_1(a, H_1(y)(\mathbf{u}), \ldots, H_k(y)(\mathbf{u})), \ldots, g_k(a, H_1(y)(\mathbf{u}), \ldots, H_k(y)(\mathbf{u}))) \\ &\quad = G_i(x)(H_1(ay)(\mathbf{u}), \ldots, H_k(ay)(\mathbf{u})). \end{aligned} \tag{1}$$

But for any $x \in \Sigma^*$, $\mathbf{u} \in B^k$,

$$G_i(x)(\mathbf{u}) = G_i(x)(H_1(\lambda)(\mathbf{u}), \ldots, H_k(\lambda)(\mathbf{u})),$$

and applying (1) $|x|$ times we get

$$\begin{aligned} G_i(x)(\mathbf{u}) &= G_i(\lambda)(H_1(x)(\mathbf{u}), \ldots, H_k(x)(\mathbf{u})) \\ &= \pi_i(H_1(x)(\mathbf{u}), \ldots, H_k(x)(\mathbf{u})) \\ &= H_i(x)(\mathbf{u}). \end{aligned}$$

□

THEOREM 5.2 *Any alternating finite automaton P accepts a regular set; moreover, if P has k states, then there is a deterministic finite automaton equivalent to P with at most* 2^{2^k} *states.*

PROOF. Let $L(P)$ denote the set of strings accepted by P. Define $x \approx y$ iff $G_1(x) = G_1(y)$. Then $\approx$ is an equivalence relation of index at most 2^{2^k}, since there are 2^{2^k} functions $B^k \to B$. Also, $\approx$ is right invariant, since $G_1(x) = G_1(y)$ implies

$$\begin{aligned} G_1(xa)(\mathbf{u}) &= G_1(x)(g_1(a, \mathbf{u}), \ldots, g_k(a, \mathbf{u})) \\ &= G_1(y)(g_1(a, \mathbf{u}), \ldots, g_k(a, \mathbf{u})) \\ &= G_1(ya)(\mathbf{u}). \end{aligned}$$

It is immediate from the definition of acceptance and Lemma 5.1 that $L(P)$ is a union of $\approx$-classes, so the states of the deterministic automaton can be the $\approx$-classes. □

The next theorem shows that the state bound 2^{2^k} cannot be improved in general.

THEOREM 5.3 *For each $k \geq 1$, there is a k-state alternating finite automaton with a three-letter input alphabet such that the smallest deterministic finite automaton accepting the same set has 2^{2^k} states.*

PROOF. Let $\Sigma = \{a, b, c\}$. We construct

$$P = (Q, \Sigma, q_1, F, g)$$

with k states and arbitrary F. Define, for $1 \leq i \leq k$,

$$g_i(a, \mathbf{u}) = \pi_i((\mathbf{u} - 1) \bmod 2^k),$$

where here and subsequently, a Boolean k-tuple $\langle u_1, \ldots, u_k \rangle$ denotes that integer between 0 and $2^k - 1$ whose k-digit binary representation is $u_1 u_2 \cdots u_k$. Note that

$$(g_1(a, \mathbf{u}), \ldots, g_k(a, \mathbf{u})) = (\mathbf{u} - 1) \bmod 2^k. \tag{2}$$

Let $\sim$ be the equivalence relation defined by

$$x \sim y \quad \text{iff} \quad (\forall w)[xw \in L(P) \text{ iff } yw \in L(P)].$$

Then the $\sim$-classes give the minimal deterministic finite automaton accepting the same set as P, so $x \approx y$ implies $x \sim y$ for any $x, y \in \Sigma^*$. We claim that the action of g_i on a forces the converse, that is, $x \sim y$ implies $x \approx y$. To see this, assume $x \sim y$, let $\mathbf{u}$ be arbitrary, and let $\mathbf{v} = (\mathbf{f} - \mathbf{u}) \bmod 2^k$. Then

$$\begin{aligned} G_1(xa^{\mathbf{v}})(\mathbf{f}) &= G_1(xa^{\mathbf{v}-1})(g_1(a, \mathbf{f}), \ldots, g_k(a, \mathbf{f})) \\ &= G_1(xa^{\mathbf{v}-1})((\mathbf{f} - 1) \bmod 2^k) \\ &\vdots \\ &= G_1(x)((\mathbf{f} - \mathbf{v}) \bmod 2^k) \\ &= G_1(x)(\mathbf{u}), \end{aligned}$$

and similarly, $G_1(ya^{\mathbf{v}})(\mathbf{f}) = G_1(y)(\mathbf{u})$. Since $xa^{\mathbf{v}} \in L(P)$ iff $ya^{\mathbf{v}} \in L(P)$, we have $G_1(x)(\mathbf{u}) = G_1(y)(\mathbf{u})$. Since $\mathbf{u}$ was arbitrary, we have $x \approx y$.

It remains to construct the rest of g so that all potential $\approx$-classes are nonempty; that is, for each Boolean function $h: B^k \to B$ there is an input $x \in \Sigma^*$ such that $G_1(x) = h$. Then the minimal automaton, given by the $\approx$-classes, will have 2^{2^k} states.

Let

$$\begin{aligned} &g_i(b, \mathbf{u}) = g_i(c, \mathbf{u}) = \pi_i(\mathbf{u}) \qquad \text{if} \quad i \neq 1 \quad \text{or} \quad \mathbf{u} \neq 0 \bmod 2^{k-1}, \\ &g_1(b, 0) = g_1(c, 0) = 1, \\ &g_1(b, 2^{k-1}) = 1, \\ &g_1(c, 2^{k-1}) = 0. \end{aligned}$$

Thus,

$$(g_1(b, \mathbf{u}), \ldots, g_k(b, \mathbf{u})) = \begin{cases} \mathbf{u} & \text{if } \mathbf{u} \neq 0, \\ 2^{k-1} & \text{if } \mathbf{u} = 0. \end{cases} \tag{3}$$

$$(g_1(c, \mathbf{u}), \ldots, g_k(c, \mathbf{u})) = \begin{cases} \mathbf{u} & \text{if } \mathbf{u} \neq 0 \bmod 2^{k-1}, \\ 2^{k-1} & \text{if } \mathbf{u} = 0, \\ 0 & \text{if } \mathbf{u} = 2^{k-1}. \end{cases} \tag{4}$$

For $2^{k-1} \leq \mathbf{v} < 2^k$, let

$$x_{\mathbf{v}} = \begin{cases} a & \text{if } h(\mathbf{v}) = 1, \\ ac & \text{if } h(\mathbf{v}) = 0, \end{cases}$$

and for $0 \leq \mathbf{u} < 2^{k-1}$, let

$$x_{\mathbf{u}} = \begin{cases} a & \text{if } h(\mathbf{u}) \neq h(\mathbf{u} + 2^{k-1}), \\ ab & \text{if } h(\mathbf{u}) = h(\mathbf{u} + 2^{k-1}). \end{cases}$$

Let $n = 2^k - 1$, and let $x = x_n x_{n-1} \cdots x_1 x_0$.

We claim that $G_1(x) = h$. First we note that for $\mathbf{v} \neq 0 \bmod 2^{k-1}$ and any y, $\mathbf{u}$,

$$G_1(yx_{\mathbf{u}})(\mathbf{v}) = G_1(y)((\mathbf{v} - 1) \bmod 2^k). \tag{5}$$

This is because $x_{\mathbf{u}}$ is one of a, ab, ac, and

$$\begin{aligned} G_1(yab)(\mathbf{v}) &= G_1(ya)(g_1(b, \mathbf{v}), \ldots, g_k(b, \mathbf{v})) \\ &= G_1(ya)(\mathbf{v}) \qquad \text{by (3)} \\ &= G_1(y)((\mathbf{v} - 1) \bmod 2^k) \qquad \text{by (2).} \end{aligned}$$

Similarly,

$$G_1(yac)(\mathbf{v}) = G_1(ya)(\mathbf{v}) = G_1(y)((\mathbf{v} - 1) \bmod 2^k).$$

Let $0 \leq \mathbf{u} < 2^{k-1}$ and $\mathbf{v} = \mathbf{u} + 2^{k-1}$. Then

$$\begin{aligned} G_1(x)(\mathbf{u}) &= G_1(x_n \cdots x_{\mathbf{v}} \cdots x_{\mathbf{u}} \cdots x_1 x_0)(\mathbf{u}) \\ &= G_1(x_n \cdots x_{\mathbf{v}} \cdots x_{\mathbf{u}})(0) && \text{by (5)} \\ &= \begin{cases} G_1(x_n \cdots x_{\mathbf{v}} \cdots x_{\mathbf{u}+1})(2^k - 1) & \text{if } h(\mathbf{u}) \neq h(\mathbf{v}) \\ G_1(x_n \cdots x_{\mathbf{v}} \cdots x_{\mathbf{u}+1})(2^{k-1} - 1) & \text{if } h(\mathbf{u}) = h(\mathbf{v}) \end{cases} \\ & && \text{by (3) and (2)} \\ &= \begin{cases} G_1(x_n \cdots x_{\mathbf{v}})(2^{k-1}) & \text{if } h(\mathbf{u}) \neq h(\mathbf{v}) \\ G_1(x_n \cdots x_{\mathbf{v}})(0) & \text{if } h(\mathbf{u}) = h(\mathbf{v}) \end{cases} \\ & && \text{by (5) and (2)} \\ &= \begin{cases} G_1(x_n \cdots x_{\mathbf{v}+1})(2^{k-1} - 1) & \text{if } h(\mathbf{u}) \neq h(\mathbf{v}) \text{ and } h(\mathbf{v}) = 1 \\ G_1(x_n \cdots x_{\mathbf{v}+1})(2^k - 1) & \text{if } h(\mathbf{u}) \neq h(\mathbf{v}) \text{ and } h(\mathbf{v}) = 0 \\ G_1(x_n \cdots x_{\mathbf{v}+1})(2^k - 1) & \text{if } h(\mathbf{u}) = h(\mathbf{v}) \text{ and } h(\mathbf{v}) = 1 \\ G_1(x_n \cdots x_{\mathbf{v}+1})(2^{k-1} - 1) & \text{if } h(\mathbf{u}) = h(\mathbf{v}) \text{ and } h(\mathbf{v}) = 0 \end{cases} \\ & && \text{by (4) and (2)} \\ &= \begin{cases} G_1(x_n \cdots x_{\mathbf{v}+1})(2^k - 1) & \text{if } h(\mathbf{u}) = 1 \\ G_1(x_n \cdots x_{\mathbf{v}+1})(2^{k-1} - 1) & \text{if } h(\mathbf{u}) = 0 \end{cases} \\ &= \begin{cases} G_1(\lambda)(\mathbf{v}) & \text{if } h(\mathbf{u}) = 1 \\ G_1(\lambda)(\mathbf{u}) & \text{if } h(\mathbf{u}) = 0 \end{cases} && \text{by (5)} \\ &= h(\mathbf{u}). \end{aligned}$$

Similarly,

$$\begin{aligned}
G_1(x)(\mathbf{v}) &= G_1(x_n \cdots x_{\mathbf{v}} \cdots x_{\mathbf{u}} \cdots x_0)(\mathbf{v}) \\
&= G_1(x_n \cdots x_{\mathbf{v}} \cdots x_{\mathbf{u}})(2^{k-1}) && \text{by (5)} \\
&= G_1(x_n \cdots x_{\mathbf{v}} \cdots x_{\mathbf{u}+1})(2^{k-1} - 1) && \text{by (3) and (2)} \\
&= G_1(x_n \cdots x_{\mathbf{v}})(0) && \text{by (5)} \\
&= \begin{cases} G_1(x_n \cdots x_{\mathbf{v}+1})(2^k - 1) & \text{if } h(\mathbf{v}) = 1 \\ G_1(x_n \cdots x_{\mathbf{v}+1})(2^{k-1} - 1) & \text{if } h(\mathbf{v}) = 0 \end{cases} \\
&= \begin{cases} G_1(\lambda)(\mathbf{v}) & \text{if } h(\mathbf{v}) = 1 \\ G_1(\lambda)(\mathbf{u}) & \text{if } h(\mathbf{v}) = 0 \end{cases} && \text{by (5)} \\
&= h(\mathbf{v}).
\end{aligned}$$

This completes the proof. □

It is interesting that the reverse of any $L(P)$,

$$(L(P))^{\mathrm{R}} = \{x^{\mathrm{R}} \mid x \in L(P)\},$$

where x^{R} is x written backward, can be accepted deterministically with only 2^k states, by taking

$$x \approx y \quad \text{iff} \quad H_i(x)(\mathbf{f}) = H_i(y)(\mathbf{f}), \qquad 1 \le i \le k.$$

5.2 Alternating Pushdown Automata. We now turn to pushdown automata and show that for these devices alternation does enlarge the class of accepted languages. An alternating PDA is similar to a nondeterministic PDA (see [10, Ch. 5]) except that there is a function mapping states to {∧, ∨, **accept**, **reject**} and the input is supplied with a right endmarker which can be sensed by the machine; the input head is one-way. A *configuration* of an alternating PDA consists of the state, the input word, the position of the input head, and the contents of the pushdown store. The definition of acceptance for alternating PDAs follows the definition given in Section 2 for ATMs simply by replacing ATM configurations by PDA configurations in the definition. Let ALT-PDA denote the class of languages accepted by alternating PDAs.

Theorem 5.4. $\bigcup_{c>0} DTIME(c^n) \subseteq ALT\text{-}PDA$.

Proof. By Theorem 3.4 it suffices to prove that

$$\mathrm{ASPACE}(n) \subseteq \mathrm{ALT\text{-}PDA}.$$

Let M be an ATM which accepts in space n. We first modify M so that M has only one tape which is one-way infinite to the right, M has no negating states, and, when given an input of length n, at most $n + 1$ tape squares are visited along any computation path (cf. Theorems 2.5 and 2.6). It is also convenient to assume that the transition relation is given by a partial function

$$\delta: Q \times \Gamma \to (Q \times \Gamma \times \{\text{left, right}\})^2,$$

where Q is the set of states and Γ is the tape alphabet; that is, when in a nonfinal state q scanning the symbol γ, M has exactly two moves described by the two components of $\delta(q, \gamma)$. As in the proof of Theorem 3.4, a *configuration* of M is viewed as a word $\mu q \nu$ where $\mu\nu \in \Gamma^*$, $|\mu\nu| = n + 1$, and $q \in Q$. The initial configuration on input x is $q_0 x \#$. For configurations α and β write $\alpha \vdash_1 \beta$ ($\alpha \vdash_2 \beta$) iff α can reach β in one step according to the first (second) component of δ. The alternating PDA M' which simulates M will "choose" a computation path of M and push it onto the

pushdown store. The path is represented by a string of the form

$$\alpha_0^R m_0 \alpha_1^R m_1 \alpha_2^R \cdots,$$

where $\alpha_0, \alpha_1, \alpha_2, \ldots$ are configurations of M (α^R denotes the reverse of α), $m_i \in \{1, 2\}$ for $i \geq 0$ (we assume $1, 2 \notin Q \cup \Gamma$), $\alpha_0 = q_0 x \not\#$, and $\alpha_i \vdash_{m_i} \alpha_{i+1}$ for $i \geq 0$. The symbol m_i is chosen by a universal (existential) branch of M' if α_i is a universal (existential) configuration of M. The strings α_i are chosen by existential branching. Each time the guess of some α_{i+1}^R is terminated, M' enters a universal state to choose one of three further actions. One action is to continue choosing the computation path; another action is to check that α_{i+1} is the correct length $n + 2$; and another is to check that $\alpha_i \vdash_{m_i} \alpha_{i+1}$. This latter check is, in essence, that of checking that two words match position by position, where "position" is actually a block of four symbols and "match" means match according to the transition function of M and the value of m_i. The matching is facilitated by universal branching, that is, by universally choosing a position in α_{i+1} to make the match, and the input head of M' is used to measure the distance (roughly) n between the chosen block of α_{i+1} and the corresponding block of α_i.

Notice that the alternation of M' is used in two ways. First, in choosing the m_i it is used to simulate the alternation of M. Second, universal branching is used to perform several actions in parallel, such as checking that two configurations match in all positions. We now describe the procedures of M' more carefully. The *and*'s ($\wedge$) and *or*'s ($\vee$) in these procedures are implemented using alternation. For example, $A \wedge B$ means to enter a universal state to choose which one of A or B to perform.

NEW:	Using existential branching, push some word in $\Gamma^* \cdot Q \cdot \Gamma^*$ onto the pushdown store; at the point where the state symbol is guessed, remember in the finite-state control whether it is accepting, rejecting, universal, or existential. If this is the first invocation of NEW, then call INIT-TOP; otherwise, call TOP.
INIT-TOP:	CONTINUE $\wedge$ INIT.
TOP:	CONTINUE $\wedge$ LENGTH $\wedge$ MATCH.
CONTINUE:	If the top configuration is accepting, then accept. If the top configuration is rejecting, then reject. If the top configuration is universal, then (push 1 $\wedge$ push 2), and then call NEW. If the top configuration is existential, then (push 1 $\vee$ push 2), and then call NEW.
INIT:	Check that $(q_0 x \not\#)^R$ is written on the pushdown store and accept or reject accordingly; this is done by popping the store while comparing it with the input.
LENGTH:	Using the input head to count up to n, check that the top of the store contains a string of $n + 2$ symbols in $(Q \cup \Gamma)^*$ followed by a symbol in $\{1, 2\}$, and accept or reject accordingly.
MATCH:	Using universal branching, choose a position in the top configuration to match against the corresponding position in the next-to-top configuration. The distance (roughly) n between the two positions is measured by using the input head of M' as discussed above. If the two positions match, then accept; otherwise reject.

We let reader verify that these procedures correctly simulate M and that they can be implemented on an alternating PDA. Since the input head of M' is one-way, it is important to note that the input is read only once along any computation path of M'. □

In fact

$$\bigcup_{c>0} \text{DTIME}(c^n) = \text{ALT-PDA}.$$

As part of their study of alternating auxiliary pushdown automata, Ladner, Lipton, and Stockmeyer [15] prove that ALT-PDA $\subseteq \bigcup_{c>0} \text{DTIME}(c^n)$; moreover, this is true if the alternating PDA has a two-way input head.

REFERENCES

1. AHO, A.V., HOPCROFT, J.E., AND ULLMAN, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. BERMAN, L. Precise bounds for Presburger arithmetic and the reals with addition. Proc. 18th IEEE Symp. on Foundations of Computer Science, Providence, R.I., 1977, pp. 95–99.
3. BORODIN, A. Personal communication.
4. CHANDRA, A.K., AND STOCKMEYER, L.J. Alternation. Proc. 17th IEEE Symp. on Foundations of Computer Science, Houston, Texas, 1976, pp. 98–108.
5. COOK, S.A. The complexity of theorem proving procedures. Proc. 3rd ACM Symp. on Theory of Computing, Shaker Heights, Ohio, 1971, pp. 151–158.
6. FISCHER, M.J., AND LADNER, R.E. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci. 18*, 2 (1979), 194–211.
7. GOLDSCHLAGER, L.M. A unified approach to models of synchronous parallel machines. Proc. 10th ACM Symp. on Theory of Computing, San Diego, Calif. 1978, pp. 89–94.
8. HARTMANIS, J., AND HUNT, H.B. III. The LBA problem and its importance in the theory of computing. In *Complexity of Computation*, R.M. Karp, Ed., American Mathematical Society, Providence, R.I., 1974, pp. 1–26.
9. HARTMANIS, J., AND SIMON, J. On the power of multiplication in random access machines. Proc. 15th IEEE Symp. on Switching and Automata Theory, New Orleans, La., 1974, pp. 13–23.
10. HOPCROFT, J.E., AND ULLMAN, J.D. *Formal Languages and their Relation to Automata.* Addison-Wesley, Reading, Mass., 1969.
11. KARP, R.M. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, Eds., Plenum Press, New York, 1972, pp. 85–104.
12. KASAI, T., ADACHI, A., AND IWATA, S. Classes of pebble games and complete problems. *SIAM J. Comput. 8* (1979), 574–586.
13. KOZEN, D. On parallelism in Turing machines. Proc. 17th IEEE Symp. on Foundations of Computer Science, Houston, Texas, 1976, pp. 89–97.
14. KOZEN, D. Complexity of Boolean algebras. *Theoret. Comput. Sci. 10* (1980), 221–247.
15. LADNER, R.E., LIPTON, R.J., AND STOCKMEYER, L.J. Alternating pushdown automata. Proc. 19th IEEE Symp. on Foundations of Computer Science, Ann Arbor, Mich., 1978.
16. MEYER, A.R., AND FISCHER, M.J. Economy of description of automata, grammars, and formal systems. Proc. 12th IEEE Symp. on Switching and Automata Theory, East Lansing, Mich., 1971, pp. 188–191.
17. PRATT, V.R., AND STOCKMEYER, L.J. A characterization of the power of vector machines. *J. Comput. Syst. Sci. 12* (1976), 198–221.
18. RABIN, M.O., AND SCOTT, D. Finite automata and their decision problems. *IBM J. Res. Dev. 3* (1959), 115–125.
19. RUZZO, W.L. General context-free language recognition. Ph.D. Diss., Computer Science Division, Univ. of California, Berkeley, Calif., 1978.
20. SAVITCH, W.J. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci 4* (1970), 177–192.
21. SAVITCH, W.J., AND STIMSON, M.J. The complexity of time bounded recursive computations. Proc. 1976 Johns Hopkins Conf. on Information Sciences and Systems, Baltimore, Md., pp. 42–46.
22. SCOTT, D. Outline of a mathematical theory of computation. Proc. 4th Ann. Princeton Conf. on Information Sciences and Systems, Princeton, N. J. 1970, pp. 169–176.
23. SIMON, J. On feasible numbers. Proc. 9th ACM Symp. on Theory of Computing, Boulder, Colo., 1977, pp. 195–207.
24. STOCKMEYER, L.J. The polynomial-time hierarchy. *Theoret. Comput. Sci. 3* (1977), 1–22.
25. STOCKMEYER, L.J., AND CHANDRA, A.K. Provably difficult combinatorial games. *SIAM J. Comput. 8* (1979), 151–174.
26. STOCKMEYER, L.J., AND MEYER, A.R. Word problems requiring exponential time: Preliminary report. Proc. 5th ACM Symp. on Theory of Computing, Austin, Texas, 1973, pp. 1–9.
27. WRATHALL, C. Complete sets and the polynomial-time hierarchy. *Theoret. Comput. Sci. 3* (1977), 23–34.

RECEIVED JANUARY 1979; REVISED JANUARY 1980; ACCEPTED FEBRUARY 1980