Math. Systems Theory 29, 191–226 (1996)

Mathematical Systems Theory

Two-Way Automata and Length-Preserving Homomorphisms*

J.-C. Birget

Department of Computer Science and Engineering, University of Nebraska, Lincoln, NE 68588-0115, USA
birget@cse.unl.edu

Abstract. Closure under *length-preserving homomorphisms* is interesting because of its similarity to *nondeterminism*. We give a characterization of NP in terms of length-preserving homomorphisms and present related complexity results. However, we mostly study the case of two-way finite automata: Let l.p.hom[n state 2DFA] denote the class of languages that are length-preserving homomorphic images of languages recognized by n-state 2DFAs. We give a machine characterization of this class. We show that any language accepted by an n-state two-way *alternating* finite automaton (2AFA), or by a 1-pebble 2NFA, belongs to l.p.hom[$O(n^2)$ state 2DFA]. Moreover, there are languages in l.p.hom[n state 2DFA] whose smallest accepting 2NFA has at least c^n states (for some constant $c > 1$). So for two-way finite automata, the closure under length-preserving homomorphisms is much more powerful than nondeterminism. We disprove two conjectures (of Meyer and Fischer, and of Chrobak) about the state-complexity of unary languages. Finally, we show that the equivalence problems for 2AFAs (resp. 1-pebble 2NFAs) are in PSPACE, and that the equivalence problem for 1-pebble 2AFAs is in EXPSPACE (thus answering a question of Jiang and Ravikumar); it was known that these problems are hard in these two classes. We also give a new proof that alternating 1-pebble machines recognize only regular languages (which was first proved by Goralčík *et al.*).

* This research was supported in part by N.S.F. Grant DMS 8702019.

1. Introduction

This paper was initially motivated by two issues:

(1) The similarities (and differences) between nondeterminism and length-preserving homomorphisms.
(2) The Open Problem (see below) of Sakoda and Sipser, and similar open problems.

We do not solve the Sakoda–Sipser conjecture; but we do obtain many results about length-preserving homomorphisms and we use this approach in order to solve other problems (e.g., to show that the equivalence problem for two-way alternating finite automata is in PSPACE). In Section 2 we also give some complexity results involving length-preserving homomorphisms. Further results on the state-complexity of finite automata, proved in similar ways, appear in [B4].

Notation. 1DFA (resp. 2DFA) means one-way (resp. two-way) *deterministic* finite automaton; 1NFA (resp. 2NFA) means one-way (resp. two-way) *nondeterministic* finite automaton (see [HU] for definitions); 1AFA (resp. 2AFA) means one-way (resp. two-way) *alternating* finite automaton (see [CKS], [CS], [Ko], [LLS], and [BL]).

Open Problem ([SS] and Seiferas). If a language L is recognized by a 2NFA, or a 1NFA, with n states, how many states does a 2DFA need (in the worst case) to recognize L?

In [SS] it is conjectured that the required number of states of the 2DFA is *larger than any polynomial in n*.

Sipser [Si1] proved that a *sweeping* 2DFA (i.e., a 2DFA which can only make reversals at the ends of its input tape) needs 2^n states to recognize certain languages which are accepted by an n-state 1NFA. On the other hand, Chrobak [Ch] proved that each *unary* n-state 1NFA can be simulated by a 2DFA with $O(n^2)$ states. The Sakoda–Sipser conjecture is probably very hard. In this paper we study and solve the following problem which is implied by (is easier than) the conjecture.

Problem (Length-Preserving Homomorphisms). If the language L ($\subseteq \Sigma^*$) can be written as $h(L_0)$, where h is a length-preserving homomorphism from Δ^* to Σ^*, and $L_0 \subseteq \Delta^*$ is recognized by an n-state 2DFA, then how many states does a 2DFA need to recognize L?

We prove a worst-case lower bound of c^n (for some constant $c > 1$).

We mention a few other open problems about the state-complexity of various finite automata; some of these problems are probably harder than the Sakoda–Sipser conjecture (in the sense that an affirmative answer to the question implies the Sakoda–Sipser conjecture). We encounter some of these questions in this paper, but for different finite-state devices.

Halting Two-Way Automata. If a language L is recognized by a 2NFA with n states, how many states does a 2NFA with the additional property that all its computations halt,

need to recognize L? Is the increase in the number of states larger than any polynomial in n?

Note that all computations of a 2NFA halt iff, for every input and every position on that input, no state is ever repeated at the same position. In the deterministic case Sipser [Si2] solved the corresponding problem (negatively): every 2DFA with n states is equivalent to a 2DFA which always halts, and which has $\leq c_1 \cdot n^2$ states (and also $\leq c_2 \cdot |\Sigma| \cdot n$ states), where c_1 and c_2 are constants, and Σ is the input alphabet. He also mentions the above problem.

Complementation. If a language L is recognized by a 2NFA (or a 1NFA) with n states, how many states does a 2NFA need to recognize $\overline{L}$ (the complement of L)? Again, does a superpolynomial lower bound exist?

Sakoda and Sipser [SS] solved this problem (affirmatively) in the "1NFA $\rightarrow$ 1NFA" case (and *a fortiori* for "2NFA $\rightarrow$ 1NFA"): for every n there is a language L_n which is recognized by an n-state 1NFA, but such that any 1NFA accepting $\overline{L}_n$ needs $\geq 2^n$ states; another proof of this result is given in [B5].

1-Pebble 2DFA. If a language L is recognized by a 2NFA (or a 1NFA) with n states, how many states does a 1-pebble 2DFA need to recognize L? Does a superpolynomial lower bound exist?

A 1-pebble 2DFA is a 2DFA which has one pebble which it can leave on the input tape, and retrieve, and put down elsewhere, etc. Blum and Hewitt [BH] proved that 1-pebble 2DFAs can only recognize regular languages; see also Exercise 3.19 on p. 73 of [HU]. In Section 5 we show that a 1-pebble 2DFA can have exponentially fewer states than any equivalent 2NFA, for some languages. In Section 6 we give a new proof that 1-pebble 2AFAs (alternating 1-pebble machines) recognize only regular languages (this result was first proved in [GGK] by very different methods).

2. Length-Preserving Homomorphisms

Homomorphisms are a familiar notion in formal language theory, especially regarding regular languages (see [HU]). By definition, a homomorphism $h\colon \Delta^* \rightarrow \Sigma^*$ (where Δ and Σ are finite alphabets) is a function satisfying $h(\varepsilon) = \varepsilon$ (where ε is the empty word), and $h(u \cdot v) = h(u) \cdot h(v)$, for all $u, v \in \Delta^*$. A homomorphism h is length-preserving iff $|h(u)| = |u|$, for all $u \in \Delta^*$. From now on we write l.p.hom. for "length-preserving homomorphism." An l.p.hom. h is determined by its restriction $h|_\Delta\colon \Delta \rightarrow \Sigma$ (restriction to the subset Δ of the domain Δ^*); l.p.hom.'s are also called "letter-to-letter homomorphisms." For a class $\mathbf{C}$ of languages, we denote by l.p.hom($\mathbf{C}$) the closure of $\mathbf{C}$ under l.p.hom.'s; that is,

$$\text{l.p.hom}(\mathbf{C}) = \{\varphi(L)/L \in \mathbf{C} \text{ and } \varphi \text{ is an l.p.hom. whose domain alphabet is the alphabet of } L\}.$$

One motivation for studying l.p.hom.'s (and, in particular, the l.p.hom.-closure of a class of languages defined by machines) is the similarity with nondeterminism: both nondeterminism and l.p.hom.-closure involve the existential quantifier.

Note that when we speak of a *class* of languages we never fix a particular alphabet; all finite alphabets are allowed. This is necessary to make the l.p.hom.-closure a nontrivial operation. We may assume however that all our alphabets are finite subsets of a fixed countable set; this assumption never constrains us, and it avoids possible set-theoretic difficulties.

For *one-way* devices, nondeterminism and l.p.hom.-closure of determinism are often equivalent. For example:

(1) l.p.hom(DCFL) = CFL.
(2) l.p.hom[n state 1DFA] = [n state 1NFA].

Notation. (D)CFL is the class of (deterministic) context-free languages; [n state 1DFA] (resp. [n state 1NFA]) is the class of all languages recognized by some n-state 1DFA (resp. 1NFA); again, the alphabet is not fixed.

For *two-way* machines however there is usually no close relation between nondeterminism and l.p.hom.-closure of determinism. A few facts are known:

(1) It is easy to see that if $S(n) \geq n$ (for all n), then l.p.hom(DSPACE(S)) = DSPACE(S) and l.p.hom(NSPACE(S)) = NSPACE(S).
(2) However, Ibarra and Ravikumar [IR] proved that, for every function $S \in o(\log) \cap \Omega(\log\log)$, DSPACE($S$) $\neq$ l.p.hom(DSPACE(S)). More strongly, they proved that the class $\bigcup_s$ DSPACE(S), where S ranges over all of $o(\log) \cap \Omega(\log\log)$, is not closed under l.p.hom.'s.
(3) The Graph Accessibility Problem is in l.p.hom(DSPACE(log)) (due to [IR]).
(4) CFL $\subseteq$ l.p.hom(DSPACE(log)) [Sp].
(5) It can easily be proved that if $T(n) \geq n$ (for all n), then l.p.hom(NTIME(T)) = NTIME(T).
(6) The class NTIME($O(n)$) (nondeterministic linear time) is equal to the closure, under l.p.hom's, of finite intersections of context-free languages [BG].

Moreover, since CFL is in AC^1 [R1], and AC^1 is closed under intersection, this implies NTIME($O(n)$) $\subseteq$ l.p.hom(AC^1). In Theorem 2.3 we improve the latter to NTIME($O(n)$) $\subseteq$ l.p.hom(NC^1).

When we talk about these circuit-based parallel-complexity classes we always mean *logspace-uniform* AC^1, AC^0, NC^1, etc., as defined in the chapter by D. Johnson, pp. 138–143, in [vL]; see also [W], [Co], and [R2]; stronger uniformity conditions would also work, but uniformity is not the main issue here. The reader should consult the references above (as well as the chapter by Boppana and Sipser in [vL]) for the definition of AC^k and NC^k ($k \geq 0$); we just give a summarized definition here: A function f belongs to AC^k iff there is a family (C_n: $n \geq 0$) of combinational circuits such that C_n contains a polynomially bounded number of gates and has depth $O((\log n)^k)$, and C_n computes $f(w)$ for any word w of length n. The family (C_n: $n \geq 0$) is log-space uniform iff a function which outputs C_n can be computed in deterministic log-space (when n is given

as an input in *unary*, i.e., as a string of length n). The class NC^k is defined in a similar way to AC^k; the only difference is that for NC^k all the gates in the circuits are restricted to having bounded fan-in.

We have the following new results:

Theorem 2.1. *The* CNF-Satisfiability *problem can be written as* $h(L_0)$, *where* h *is an* l.p.hom. *and* L_0 *is in* AC^0. *Thus, the class* l.p.hom(AC^0) ($\subseteq$ l.p.hom(DSPACE(log)) $\subseteq$ NP) *contains some* NP*-complete problems.*

Proof. It is straightforward to check (see also (5) above) that NP is closed under l.p.hom.'s, and, hence, contains l.p.hom(DSPACE(log)) and l.p.hom(AC^0). We show that the Satisfiability problem for CNFs (*conjunctive normal forms*, see [HU] and the references to Cook's work therein) is an l.p.hom. image of a language in uniform AC^0. We represent the boolean variables of CNFs by nonempty words over a two-letter alphabet $\{a, b\}$; the other symbols that occur are $\vee$ (or), $\wedge$ (and), $\neg$ (not), and $(\!($, $)\!)$ (parenthesis symbols). This representation of CNFs is simple and natural; representations could also be devised that are more compact or use a smaller alphabet. A word w over this seven-letter alphabet represents a *satisfiable* CNF if and only if w is an l.p.hom. image of a syntactically well-formed CNF w' which is *marked* by a truth-value assignment that evaluates to 1 (= TRUE); the truth-value assignments to the boolean variables will be written under the leftmost letter of each boolean variable. The l.p.hom. just erases the truth-value assignment. The alphabet has now the additional four letters $\underset{0}{a}$, $\underset{1}{a}$, $\underset{0}{b}$, and $\underset{1}{b}$.

For example, the satisfiable CNF $w = (\!(aa \vee \neg ab \vee \neg b)\!) \wedge (\!(\neg aa \vee b)\!) \wedge (\!(bb \vee \neg aa)\!)$ is the image of the marked CNF $w' = (\!(\underset{0}{a}\, a \vee \neg \underset{1}{a}\, \underset{0}{b} \vee \neg \underset{0}{b})\!) \wedge (\!(\neg \underset{0}{a}\, a \vee \underset{0}{b})\!) \wedge (\!(\underset{1}{b}\, b \vee \neg \underset{0}{a}\, a)\!)$ which evaluates to 1 for the marked assignment (here the boolean variables and their truth-value assignments are $aa := 0$, $ab := 1$, $b := 0$, $bb := 1$).

Let $L_0 \subseteq \{a, b, \vee, \wedge, \neg, (\!(,)\!), \underset{0}{a}, \underset{1}{a}, \underset{0}{b}, \underset{1}{b}\}^*$ be the language of CNFs, each marked by a truth-value assignment that evaluates to 1. We want to show that L_0 is in AC^0. First, a counter-free finite automaton can check that an expression over the eleven-letter alphabet is a CNF and that the truth-value assignment evaluates to 1. (Counter-free finite automata can be simulated by AC^0 circuits, see [BT] and [BCST].) The only hard thing to check is that the truth-value assignment is *consistent* (i.e., the same boolean variable is assigned the same truth-value wherever it occurs in the CNF). To check this, all pairs $\underset{x}{c}\, w_1, \underset{y}{d}\, w_2$ (where $c, d \in \{a, b\}$, $x, y \in \{0, 1\}$, and $w_1, w_2 \in \{a, b\}^*$) of variables marked by truth values that appear in the CNF are picked; all these pairs are handled in parallel. The two members of such a pair are compared as follows: if $cw_1 = dw_2$ and $x \neq y$, then the truth-value assignment is *in*consistent; if this does not happen for any pair, the truth-value assignment of the CNF is consistent. So, the problem reduces to checking the equality of strings; this can be done in AC^0 (straightforward exercise, see also [Sa] and [W]). □

Theorem 2.2 (A Characterization of NP). *A language* L *is in* NP *iff there is a polynomia length padding of* L *which belongs to* l.p.hom(NC^1).

Definition. The $p(\cdot)$-length padding of L is the language $\{w\$^{p(|w|)}/w \in L\}$; here $p(\cdot)$ is a polynomial, and \$ is a letter that does not belong to the alphabet of L.

Proof. If a polynomial-length padding of L is in l.p.hom(NC^1) (or, more generally, in NP), then L is clearly in NP. Conversely, if L is in NP, then L can be reduced to the CNF-Satisfiability problem via a many-to-one NC^1-computable reduction (Cook's theorem, see [HU]); the argument in [HU], showing that the reduction is in log-space, also shows that the reduction is in NC^1. Let f be the reduction function used: For every word w, $f(w)$ is a boolean formula (represented as a string as in the proof of Theorem 2.1) which is satisfiable iff w belongs to L; moreover, $f(w)$ is computable in NC^1, and we can assume that $|(\!(f(w))\!)| = p(|w|)$, for some polynomial $p(\cdot)$. Consider now the language

$$L_1 = \{wB/w \in L \text{ and } B \text{ is any marked boolean formula (as in the proof of Theorem 2.1) obtained from } f(w) \text{ by marking it with a truth-value assignment that evaluates to 1}\}.$$

We assume that the alphabet of L, and the alphabet used for representing marked boolean formulas, are disjoint.

The $p(\cdot)$-length padding of L is then an l.p.hom. image of L_1. Moreover, L_1 is in AC^0: Given a string z, we first factor z as wB by looking for the first occurrence of the left-parenthesis symbol "(" (if "(" does not occur we reject); we do this by factoring z as wB in all possible ways (there are $|z| + 1$ possibilities) in parallel, and retaining only the factorization where B starts with "(" and w is over the alphabet of L; this is an AC^0-algorithm. Second, we check that B belongs to L_0, as in the proof of Theorem 2.1; this problem is in AC^0, as we saw. If we have not rejected z so far, B is the marked form of a satisfiable boolean formula β. Finally, we can compute $f(w)$ in NC^1 and check that β and $f(w)$ are the same. □

We would like to thank one of the anonymous referees for suggesting a result similar to Theorem 2.2, as an extension of Theorem 2.1. Also, if CNF-Satisfiability can be proved to be NP-complete for many-to-one reductions that are more restrictive than NC^1 (but no less restrictive than AC^0), then Theorem 2.2 is automatically strengthened.

The following theorem generalizes (4) above, since $NC^1 \subseteq$ DSPACE(log) and CFL $\subseteq$ NTIME($O(n)$); compare also with (5). Theorem 2.2 could also be derived from Theorem 2.3, since the $T(\cdot)$-padding of any language in NTIME(T) belongs to NTIME($O(n)$).

Theorem 2.3. NTIME($O(n)$) $\subseteq$ l.p.hom(NC^1).

Proof. Let $L \subseteq \Sigma^*$ be accepted by a k-tape nondeterministic Turing machine M in time $c \cdot n$ (where c is a constant). The machine M has the form $(Q, \Sigma, \Gamma, \Delta, q_0, F)$, where Γ is the work-tape alphabet (including a blank symbol), q_0 is the start state, F is the set of accept states, and $\Delta \subseteq Q \times \Gamma^k \times Q \times \Gamma^k \times \{-1, +1\}^k$ is the transition relation. We call any element of Δ a *transition*; so, a transition has the form

$(q, a_1, \ldots, a_k, q', b_1, \ldots, b_k, i_1, \ldots, i_k)$; we also write

$$(q, a_1, \ldots, a_k) \to (q', b_1, \ldots, b_k, i_1, \ldots, i_k),$$

and we call $(q, a_1, \ldots, a_k)$ the left side of the transition, and $(q', b_1, \ldots, b_k, i_1, \ldots, i_k)$ the right side; we call a_i (resp. b_i) the ith *tape-coordinate* of $(q, a_1, \ldots, a_i, \ldots, a_k)$ (resp. $(q', b_1, \ldots, b_i, \ldots, b_k, j_1, \ldots, j_k)$).

We write every word $w = a_1 \cdots a_n \in L$ as an l.p.hom. image of a word obtained by writing down w and uniformly interleaving w with a sequence of $c \cdot n$ transitions (provided that this sequence of transitions describes an accepting computation for w). More precisely, we introduce the alphabet $\Sigma \times \Delta^c$, which is finite and consists of elements of the form $(a, \delta_{i_1} \cdots \delta_{i_c})$. We consider the l.p.hom. h: $(\Sigma \times \Delta^c)^* \to \Sigma^*$ which just erases the second coordinate; so $h((a, \delta_{i_1} \cdots \delta_{i_c})) = a$. We define the following language L' such that $L = h(L')$:

$$L' = \{(a_1, \delta_1 \cdots \delta_c)(a_2, \delta_{c+1} \cdots \delta_{2c}) \cdots (a_n, \delta_{cn-c+1} \cdots \delta_{cn}) \in (\Sigma \times \Delta^c)^* / n \geq 0, a_1 \cdots a_n \in L, \text{ and } \delta_1 \cdots \delta_c \delta_{c+1} \cdots \delta_{2c} \cdots \cdots \delta_{cn-c+1} \cdots \delta_{cn} \text{ describes an accepting computation of } M \text{ on input } a_1 \cdots a_n\}.$$

We want to show that L' is in NC^1, i.e., we want an NC^1 algorithm which, given w (with $|w| = n$) and a sequence $\delta_1 \cdots \delta_{cn}$ of transitions, decides whether this sequence of transitions describes a well-formed accepting computation of M on input w. The following observation is important: For any given time t $(0 \leq t \leq cn)$, the position $p_i(t)$ of the head on the ith tape $(1 \leq i \leq k)$ can be computed by an NC^1 algorithm. This can be done by simply summing the direction-of-movement coordinate over the sequence $\delta_1 \cdots \delta_t$ (and it is known that the sum of a linear number of bounded-size integers can be computed in NC^1; see [Sa] and [W]). Moreover, since t is linearly bounded, we can compute *all* the positions $p_i(t)$ (for all t) in parallel, in NC^1. Thanks to this observation, we can talk about the positions of the heads during the hypothetical computation.

We have the following criterion.

The sequence $\delta_1 \cdots \delta_{cn}$ of transitions of M on input w describes an accepting computation **iff** *all the following consistency conditions hold*:

States—the state on the left side of the first transition δ_1 is the start state q_0, and the state on the right side of the last transition δ_{cn} belongs to the set of accept states F; moreover, for each t $(1 \leq t < cn)$, the state on the right side of δ_t is the same as the state on the left side of δ_{t+1}.

*Input—for all i, when the head on the i*th *tape visits a position p for the first time (suppose this happens at time τ) it sees the p*th *input letter a_p (if $i = 1$ and $p \leq n$) or the blank symbol; more precisely, the left side of transition δ_τ has a_p, respectively the blank, as the i*th *tape-coordinate.*

*Read-write consistency—for all i, if the head on the i*th *tape visits a position p at times σ and τ $(> \sigma)$ (without visiting p inbetween), then the head reads at time τ what it wrote at time σ; more precisely, the right side of δ_σ and the left side of δ_τ have the same i*th *tape-coordinate.*

This criterion readily leads to the desired NC^1 algorithm. The condition about the states can be checked by a finite automaton, and hence by an NC^1 circuit.

The two conditions about the input and read-write consistency are combined. For all t ($1 \leq t \leq cn$) in parallel, we compute the positions $p_1(t), \ldots, p_k(t)$ of the k heads at time t (using the earlier observation that this can be done in NC^1).

We check for all $t' < t$ (in parallel) whether $p_i(t') \neq p_i(t)$; if so, the position $p_i(t)$ has never been visited before on the ith tape, so we check that the ith tape-coordinate of the left side of δ_t is a_p (with $p = p_i(t)$), respectively the blank symbol (input condition). This is done for each i. The operation of comparing ($<$, $\leq$, $\neq$) two integers can be done in NC^1 (see [Sa] and [W]).

On the other hand, if $p_i(t') = p_i(t)$ for some t' ($< t$), then we are not in a situation where the input condition needs to be checked at time t. For this time t we compute the next time t'' when position $p_i(t)$ is visited again; this is done by considering all τ ($> t$) in parallel and taking t'' to be the smallest τ such that $p_i(\tau) = p_i(t)$; this "minimum" operation can be done in NC^1 (since even the sorting problem can be solved in NC^1; again, see [Sa] and [W]). Now we check that on the right side of δ_t and on the left side of $\delta_{t''}$, the ith tape-coordinates are the same (read-write consistency). If no t'' exists, then the condition is vacuous and does not need to be checked.

Finally, the results of these $O(n^2)$ parallel checks are combined into a boolean "and" (which, again, can be done in NC^1). This completes the proof that the language L' is in NC^1. □

For $S(n) \in o(n)$ little seems to be known about how l.p.hom(DSPACES), DSPACES, and NSPACES are related. In this paper we consider the case where S is a constant and show that here l.p.hom. is exponentially more powerful than nondeterminism, and at least as powerful as alternation (up to squaring the number of states). In [B4] we show that l.p.hom. is also exponentially more powerful than alternation.

If instead of l.p.hom.'s *all* homomorphisms are taken, then the following is obtained (see [Sp]): the homomorphism closure of DSPACE(log) is the class of all recursively enumerable languages. (Indeed, in log-space the validity of any Turing machine calculation can be checked; to obtain the language accepted by the Turing machine, the noninput configurations of the computations are erased, by applying a homomorphism.)

Remark. A standard wrong argument for "proving" that

$$\text{l.p.hom}[n \text{ state 2DFA}] \subseteq [O(n) \text{ state 2NFA}]$$

(which is also a *wrong assertion*, see Theorem 4.3) needs to be addressed. Suppose $L = h(L_0)$, $L_0 \in [n \text{ state 2DFA}]$. In the wrong argument a 2NFA A would, each time it reads a letter a on its tape, guess a letter $b \in h^{-1}(a)$ and feed b to the 2DFA recognizing L_0. This works correctly for one-way devices, but to be correct for two-way automata, A would have to guess the *same* $b \in h^{-1}(a)$ each time it revisits a at a given position on the tape.

3. A Machine Model Related to l.p.hom[n state 2DFA]

Here [n state 2DFA] denotes the class of all languages (over all possible finite alphabets) that are recognized by n-state 2DFAs.

A machine model that is closely related to the class l.p.hom[n state 2DFA] is the *nondeterministic single-tape n-state Turing machine*, with the two additional restrictions:

(1) The read-write head *cannot leave the input portion* of the tape, which is surrounded by end markers "⟨" (at the left) and "⟩" (at the right).
(2) For every word that is accepted an accepting computation exists during which every position of the tape is *visited at most k times* (for some constant k which is independent of the input). We say in that case that "*there are $\leq k$ visits per position.*"

We assume that the machine moves right or halts when it reads "⟨", and that it moves left or halts when it reads "⟩." In a starting configuration, the state is the start state and the head is on the leftmost letter of the input (just right of "⟨." In an accepting configuration the state is an accept state and the head is on "⟩" (so, we use the same acceptance rule as for two-way finite automata).

We call such machines **Hennie machines**, after Hennie [He] who proved (among other results):

(1) If a language L is recognized by a Hennie machine, then L is regular.
(2) Every one-tape *deterministic* Turing machine with linear-time complexity and which never leaves the input portion of the tape, is actually a deterministic Hennie machine (for emphasis: such a Turing machine is not only equivalent to a Hennie machine, it *is* a Hennie machine; it can visit each tape position only a bounded number of times during accepting computations).
(3) Any deterministic (resp. nondeterministic) one-tape linear-time Turing machine is equivalent to a deterministic (resp. nondeterministic) one-tape linear-time Turing machine which never leaves the input portion of the tape.

It is remarkable that the above result (2) of Hennie does not extend to *non*deterministic linear time: indeed, nondeterministic one-tape linear-time Turing machines exist which accept NP-complete languages [M], and, thus, such Turing machines are not equivalent to nondeterministic Hennie machines (which only accept regular languages). So, one-tape linear-time Turing machines have a striking property: When these machines are extended from determinism to nondeterminism, the languages accepted pass from *regular* to a subclass of NP which includes NP-*complete* languages.

We often use the following observation: If in a Hennie machine there are $\leq k$ visits per position (during a certain computation), then, obviously, no state can occur more than k times at the same position. A slightly weaker converse also holds: If in an n-state Hennie machine no state occurs more than k_1 times at the same position, then there are $\leq nk_1$ visits at the same position (indeed, suppose there were more than nk_1 visits at a position; then, since there are only n states, some state would have to occur more than k_1 times at that position, by the Pigeon-Hole Principle).

Theorem 3.1.

(a) *If* $L \in$ l.p.hom[*n state* 2DFA], *then* L *is recognized by a nondeterministic* n*-state Hennie machine, in which no state occurs more than once at the same tape position during any accepting computation .*

(b) *Suppose* L *is recognized by an* n*-state Hennie machine such that for every word accepted an accepting computation exists during which no state occurs more than* k_1 *times at the same position of the tape (where* k_1 *is a constant); then* L *belongs to* l.p.hom[$n \cdot k_1$ *state* 2DFA].

Corollary 3.2 (Machine Characterization of l.p.hom[n state 2DFA]). *The following thr conditions are equivalent for any language* L*:*

(1) L *belongs to* l.p.hom[*n state* 2DFA].

(2) L *is recognized by a nondeterministic* n*-state Hennie machine such that, in* **every accepting computation**, *no state occurs more than once at the same tape position.*

(3) L *is recognized by a nondeterministic* n*-state Hennie machine such that,* **for every accepted word an accepting computation exists** *in which no state occurs more than once at the same position.*

Proof of Theorem 3.1. (a) Suppose $L = h(L_0) \subseteq \Sigma^*$, where $L_0 \subseteq \Delta^*$ is recognized by an n-state 2DFA, Σ and Δ are finite alphabets, and h: $\Sigma^* \to \Delta^*$ is an l.p.hom. We may assume that $\Sigma \cap \Delta = \varnothing$ (if necessary, we make a new copy of Δ; this does not change L, nor the number of states needed to accept L_0 by a 2DFA).

A nondeterministic Hennie machine accepting L ($\subseteq \Sigma^*$) simulates the 2DFA of L_0 in the following way: The Hennie machine has the same state set as the 2DFA, and its work alphabet is $\Sigma \cup \Delta$. When the Hennie machine reads a letter $\sigma \in \Sigma$ in state q, it guesses a letter $\delta \in h^{-1}(\sigma)$ and prints δ (overwriting σ), and then simulates how the 2DFA would move on δ in state q; all this constitutes one nondeterministic Hennie machine step. When the Hennie machine reads a letter $\delta \in \Delta$ (printed earlier), it just simulates the 2DFA. This uses the fact that $\Sigma \cap \Delta = \varnothing$. No new states are introduced. (Recall also that, by definition, we only require that the Hennie machine makes a bounded number of visits per position on at least one *accepting* computation for each *accepted* word; some accepting computations might make more visits; rejecting computations can do any number of visits. So we need not assume that our 2DFA for L_0 halts on all inputs.)

(b) Suppose $L \subseteq \Sigma^*$ is recognized by a nondeterministic Hennie machine H with n states, and for every word in L an accepting computation exists during which no state occurs more than k_1 times at the same position. Let $k = n \cdot k_1$; by the observation made before Theorem 3.1, H makes $\leq k$ visits per position (during the computations under consideration).

We write $L = h(L_c)$, where L_c is the "k-track computation language" of H (described also in [B3]): L_c consists of all accepting computations of H, encoded into k parallel rows, each as long as the input. See Figure 1 for an example. Then L_c is recognized by a 2DFA A which takes such a k-track picture as an input and checks whether the computation described by the picture is valid and accepting. The state set of A is $Q \times \{1, \ldots, k_1\}$ (where Q is the state set of H); A remembers the state in which H

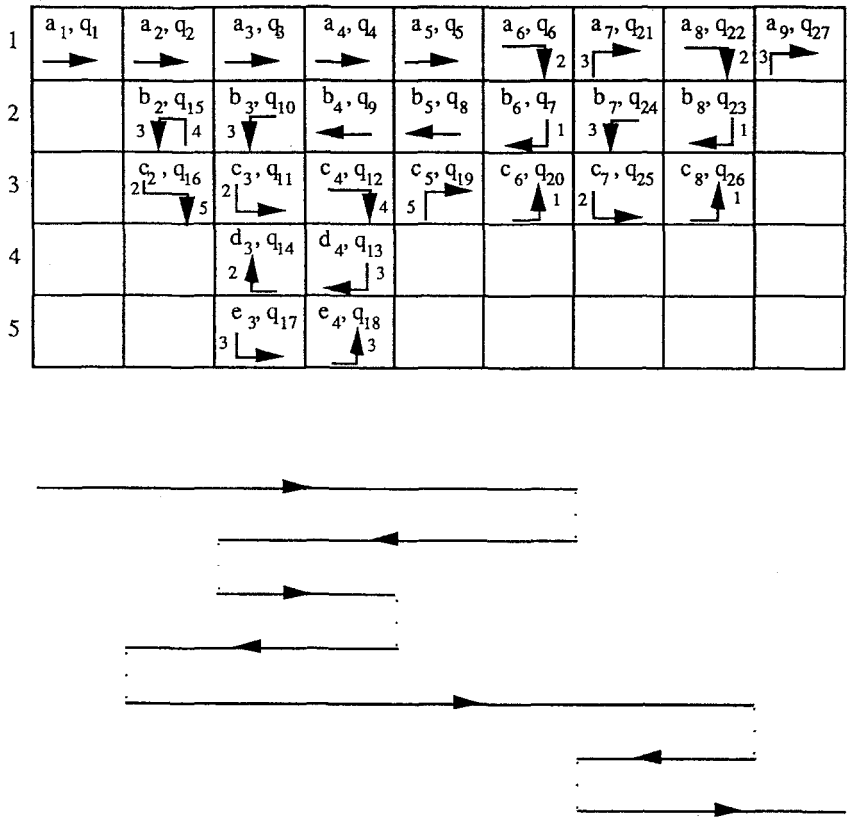


Fig. 1. The k-track picture (here $k = 5$) of a computation of H, and the corresponding movement of the head.

would be, and also a number i ($1 \leq i \leq k_1$) chosen so that the state together with i will enable A to know which track it is on (since $k = n \cdot k_1$, no more than k_1 choices for i are required). Each time H makes a reversal the k-track picture shows the continuation of the computation on another track. The idea is similar to the hint in Exercise 3.19 on p. 73 of [HU] (where 1-pebble automata are discussed).

The alphabet Δ of L_c and A is the cartesian product

$$(\Sigma \times Q \times D) \times (\Gamma \times Q \times D \cup \{\varepsilon\})^{k-1},$$

where Γ is the work alphabet of H, and

$$D = \{\rightarrow, \leftarrow, \downarrow_i, \leftarrow_i, \rightarrow_i, \downarrow_i, {}^i\uparrow, {}^i\rightarrow, \uparrow^i, \leftarrow^i, {}_i\uparrow^j, {}^j\uparrow_i, {}^i\downarrow_j, {}_j\downarrow^i,$$
$${}_i\downarrow_j, {}_i\downarrow_j, {}^i\uparrow^j, {}^i\uparrow^j / 1 \leq i \leq k, 1 \leq j \leq k\}.$$

A letter of Δ is thus a k-track column (each cell of which contains an element of $\Gamma \times Q \times D$ or the empty word ε).

We define the l.p.hom. h: $\Delta^* \to \Sigma^*$ by mapping every element of Δ to its Σ-coordinate of the first cell. □

Corollary 3.3. l.p.hom[*n state* 2DFA] = l.p.hom[*n state* 2NFA].

Proof. Theorem 3.1(b) applies here; an n-state 2NFA is a special case of a nondeterministic Hennie machine with $k_1 = 1$. □

Corollary 3.4. l.p.hom[*n state* 2DFA] = l.d.hom[*n state* 2DFA].

Here l.d.hom *stands for "length-decreasing homomorphism," i.e., homomorphisms h satisfying $|h(x)| \leq |x|$ (equivalently, every letter is mapped to a letter or to the empty string).*

Proof. By Theorem 3.1 it is sufficient to show: If $L = h(L_0) \subseteq \Sigma^*$ (where $L_0 \subseteq \Delta^*$ is recognized by a 2DFA with n states, and h: $\Delta^* \to \Sigma^*$ is an l.d.hom.), then L is recognized by a nondeterministic Hennie machine H with n states in which no state occurs more than once at the same position. We can assume again that $\Sigma \cap \Delta = \varnothing$. On input $a_1 \cdots a_k \in \Sigma^*$, H works as follows: Each time H reads a letter a_i it guesses a string $x_i \in \Delta^*$ such that $h(x_i) = a_i$, and prints $[x_i]$, overwriting a_i ($[x_i]$ is the quadruple of global state-transition functions, or Shepherdson tables, with respect to the 2DFA of L_0; see the Background paragraph at the end of the proof of Theorem 4.1(c) of this paper, or [B1], [B3], and the Appendix of [B4]). Next H goes to the state determined by the current state and $[x_i]$. When H reads an $[x_j]$ (printed earlier) it just applies the appropriate global transition function of $[x_j]$ to the current state. □

In the previous theorems, large alphabets appear; the following fact puts an upper bound on the required size of alphabets.

Fact 3.5 (Alphabet Sizes). *If $L \subseteq \Sigma^*$ belongs to* l.p.hom[*n state* 2DFA], *then $L = h(L_0)$ for some $L_0 \in$* [*n state* 2DFA], *where the alphabet of L_0 has size $\leq |\Sigma| \cdot n^n$.*

Proof. The proof of this is similar to the proof of Corollary 3.4: if L_0 is originally over an arbitrarily large alphabet Δ and $L = h(L_0)$, we replace Δ by the alphabet $\{(h(\delta), [\delta])/\delta \in \Delta\}$; again, $[\delta]$ is the quadruple of global state-transitions, as above; h is replaced by the l.p.hom. θ defined by $(h(\delta), [\delta]) \to h(\delta)$. □

As a consequence, there are only finitely many languages over a given alphabet Σ that belong to l.p.hom[n state 2DFA] for a fixed n; and there are only finitely many languages over a given alphabet Σ that are accepted by n-state k-visiting nondeterministic Hennie machines, for fixed n and k (but irrespectively of the size of the work alphabet); for the latter one uses Theorem 3.1(b).

The Hennie machines associated with l.p.hom[n state 2DFA] (by Theorem 3.1), and the ones we will encounter in Section 4 (in relation with n-state 1-pebble machines or alternating finite automata), all have a number of visits per position which is polynomially bounded by the number of states of the Hennie machine. The rest of this section is a

digression which shows that in general, however, the number of visits per position of a deterministic Hennie machine need not be recursively bounded by the number of states of the machine.

Theorem 3.6. *There is a family of languages $L_n \subseteq \{a\}^*$ ($n \in \mathbb{N}$), where L_n is recognized by a deterministic Hennie machine with n states, but:*

(1) *Every 2AFA recognizing L_n has a number of states $> \Sigma(n - c_1)$, where $\Sigma(\cdot)$ is the Busy Beaver function and c_1 is a constant.*
(2) *Every nondeterministic Hennie machine with s states and $\leq k$ visits per position, which recognizes L_n, satisfies $s \cdot k > \Sigma(n - c_2)$, for some constant c_2.*

(It is well known that the Busy Beaver function $\Sigma(\cdot)$ is eventually larger than any recursive function; see, e.g., [DDQ].)

Proof. Let B_n be an n-state Busy Beaver; B_n is a deterministic Turing machine with one two-sided infinite tape, with work alphabet $\{0, 1\}$. When B_n is started on a blank tape it eventually halts, with $\Sigma(n)$ 1's on the tape (there may be 0's as well). Let $\mathbf{t}(n)$ be the length of the 0–1 string on the tape when B_n halts; so $\mathbf{t}(n) \geq \Sigma(n)$.

Let $L_n = \{a^{\mathbf{t}(n-c)}\} \subseteq \{a\}^*$ (L_n consists of a single word); c is a constant to be determined later.

We first prove that L_n is recognized by a deterministic Hennie machine H with n states. On input a^m, H simulates B_{n-c}, treating a and the two endmarkers of H like the blank symbol. If B_{n-c} wants to write on the right endmarker, H halts and rejects. If B_{n-c} runs out of space on the left (i.e., it wants to write on the left endmarker of H), then H will shift the whole 0–1 string one space to the right (to do this H needs a fixed number of states); if in this process H would need to write on the right endmarker, it halts and rejects. Finally, when B_{n-c} halts (and if H has not rejected the input yet), the tape content is of the form $xa^{m-\mathbf{t}(n-c)}$, where x is a 0–1 string of length $\mathbf{t}(n - c)$. Now H will accept iff at this point the tape actually contains no a's (i.e., iff $m = \mathbf{t}(n - c)$). Then H has n states, if the constant c is appropriately chosen.

Proof of parts (1) and (2) of the theorem:

Since a minimum-state 1NFA accepting $\{a^{\mathbf{t}(n-c)}\}$ has $\mathbf{t}(n - c) + 1$ states (proof by Pumping Lemma), any 2AFA with s states must satisfy $2^{s2^s} > \mathbf{t}(n - c) \geq \Sigma(n - c)$ (see [LLS] or [B4]). Since $\Sigma(\cdot)$ grows extremely fast, this implies that $s > \Sigma(n - c_1)$, for a large enough constant c_1. Similarly, any Hennie machine with s states and $\leq k$ visits per position is equivalent to a 1NFA with $(sk)^{sk}$ states (by Theorem 3.1(b), and by the fact that any 2DFA with r states is equivalent to a 1NFA with $\leq r^r$ states, see [HU] and [B4]). Thus $(sk)^{sk} > \Sigma(n - c)$. Since $\Sigma(\cdot)$ grows so fast, we obtain $sk > \Sigma(n - c_2)$, for a constant c_2. □

4. The Power of l.p.hom[n state 2DFA]

We have seen already (Corollary 3.3) that [n state 2NFA] is contained in l.p.hom[n state 2DFA]. The next theorem says that, up to squaring the number of states,

l.p.hom[n state 2DFA] is at least as powerful as *alternating two-way finite automata* (2AFA), or *nondeterministic 1-pebble two-way automata* (1-pebble 2NFA). The 2AFAs that we use here are allowed to have any boolean functions (with the states as boolean variables) attached to their state transitions (see [Ko] for 1AFAs; our model of 2AFAs is more general than [LLS], where only $\exists$- and $\forall$-configurations are admitted and where the head can move only when the configuration is deterministic).

Theorem 4.1.

(a) *Every n-state* **2AFA** *can be simulated by a deterministic Hennie machine with $O(n)$ states, with no state occurring more than $O(n)$ times at the same position. Thus (by Theorem* 3.1(b)*), for some constant c:*

$$[\mathbf{n} \textit{ state } \mathbf{2AFA}] \subseteq \text{l.p.hom}[\mathbf{c} \cdot \mathbf{n^2} \textit{ state } \mathbf{2DFA}].$$

(b) *Every language recognized by an n-state* **1AFA** *or, more generally, by a halting n-state* 2AFA *(i.e., which eventually halts on every computation path), belongs to* l.p.hom[$\mathbf{c} \cdot \mathbf{n}$ *state* **2DFA**]*, for some constant c.*

(c) *Every n-state* **1-pebble 2DFA** *(resp.* **1-pebble 2NFA***) can be simulated by a deterministic (resp. nondeterministic) Hennie machine with $O(n^2)$ states, such that every accepted word has an accepting computation in which no state occurs more than once at the same position. Thus (by Theorem* 3.1(b)*) the language recognized belongs to* l.p.hom[$\mathbf{c} \cdot \mathbf{n^2}$ *state* **2DFA**]*, for some constant c.*

For nondeterministic 1-pebble machines we also have: every n-state **1-pebble 2NFA** *can be simulated by a* **deterministic** *Hennie machine with $O(n^2)$ states, in which $O(n)$ states occur at most $O(n)$ times at the same position, and each of the remaining $O(n^2)$ states occurs just once at the same position.*

The following theorem is about alternating 1-pebble machines; in particular, it implies that such devices recognize only regular languages (which was first proved by Goralčík *et al.* [GGK] in a very different way).

(d) *Every* **n**-*state* **1-pebble 2AFA** *can be simulated by a deterministic Hennie machine with $O(n2^n)$ states; $O(n)$ of these states occur at most $O(n)$ times at the same position, and the remaining $O(n2^n)$ states occur just once at the same position. Thus (by Theorem* 3.1(b)*):*

$$[\mathbf{n} \textit{ state } \mathbf{1\text{-}pebble\ 2AFA}] \subseteq \text{l.p.hom}[\mathbf{c} \cdot \mathbf{n2^n} \textit{ state } \mathbf{2DFA}],$$

for some constant c.

(The proofs of Theorems 4.1(a)–(d) are given at the end of this section.)
Combining Theorem 4.1(d) with Corollary 4.4 of [B4] we obtain:

$$[\mathbf{n} \text{ state } \mathbf{1\text{-}pebble\ 2AFA}] \subseteq [2^{\mathbf{cn2^n}} \text{ state } \mathbf{1NFA}] \qquad \text{for some constant } c.$$

This is similar to a result of [GGK]: [n state 1-pebble 2AFA] $\subseteq [|\Sigma| \cdot d(n)^{3n+1} + d(n)$ state 1NFA], when a fixed alphabet Σ is used. Here $d(n)$ ($\leq 2^{2^n}$) is the size of the free distributive (0, 1)-lattice on n variables. The definition of alternation in [GGK] is

more restrictive than the one used here (they only allow $\exists$ and $\forall$ on the states, instead of arbitrary boolean functions), and their bound depends on the alphabet size.

Questions analogous to the open problems in Section 1 can be asked about Hennie machines. The problem about *halting* (for deterministic or nondeterministic Hennie machines) can be easily solved (affirmatively), since a nondeterministic Hennie machine could write a number ($\leq k$, and equal to 1 at the first visit) at every position and increment it each time it visits the position (and halt when the number becomes $> k$). This also solves (affirmatively) the *complementation* problem for *deterministic* Hennie machines. Some of the other problems are solved in [B4]. In summary we have:

Fact 4.2.

(1) *Every nondeterministic (resp. deterministic) Hennie machine with n states and $\leq k$ visits per position, is equivalent to a nondeterministic (resp. deterministic) Hennie machine with n states and $\leq k$ visits per position, in which every computation eventually* **halts**. *The same is true when "$\leq k$ visits per position" is replaced everywhere by "with every state occurring $\leq k$ times at the same position."*

(2) *If a language L is recognized by a deterministic Hennie machine H_1 with n states and $\leq k$ visits per position, then $\overline{L}$ (the* **complement** *of L) is recognized by a deterministic Hennie machine H_2 with $\leq n+1$ states and $\leq k+1$ visits per position. Similarly: If L is recognized by a deterministic Hennie machine H_1 with n states and with every state occurring $\leq k$ times at the same position, then $\overline{L}$ is recognized by a deterministic Hennie machine H_2 with $\leq n+1$ states and with every state occurring $\leq k+1$ times at the same position.*

In (2), the reason why n and k might have to be increase to $n+1$ and $k+1$ is that we want H_2 to end up at the right end of the tape in an accepting computation.

The next theorem gives exponential lower bounds for the recognition of certain languages in l.p.hom[n state 2DFA] by 2NFAs.

Theorem 4.3. *There is a unary language L in* l.p.hom[n *state 2DFA*] *such that any 2NFA recognizing L needs $> c^n$ states; here c is a constant > 1.414 $(= \sqrt{2} - \varepsilon)$.*

Proof Outline (see Section 5 for more details). We show in Section 5 that the one-word language $\{a^{2^n}\}$ can be recognized by a deterministic Hennie machine with $2n+e$ states (where e is a constant), with no state occurring more than once at the same position; and we prove that a 2NFA needs more than 2^n states to recognize $\{a^{2^n}\}$. Now let $L = \{a^{2^{(n-e)/2}}\}$, which belongs to l.p.hom[n state 2DFA]; then a 2NFA or a 1-pebble 2DFA recognizing L needs $> c^n$ states, where $c = \sqrt{2} - \varepsilon$ (where ε is a positive real number, which becomes arbitrarily small when n becomes large). □

Meyer and Fischer [MF] introduced the language $\{a^{2^n}\}$ and they strongly conjectured that a 1-pebble 2DFA needs $\geq 2^n$ states to recognize it; this is false, however: we show in Section 5 that a 1-pebble 2DFA needs only $O(n^2/\log n)$ states to recognize $\{a^{2^n}\}$ (and the same holds for Π_2-1AFAs).

Proof of Theorem 4.1(a) *and* (b). We first construct a **complete language A_n for n-state 2AFAs**. The alphabet of the language A_n is the set of all tripartite directed graphs of $3n$ vertices, with n vertices in each of the three partitions; we view the three partitions as three parallel vertical columns of n vertices each; the only edges are directed edges that go from the right column and from the left column to the middle column; each vertex in the middle column is labeled by a boolean function (and is viewed as a gate implementing this boolean function). We call the letters of this alphabet *circuit-slices*. A word is a sequence of slices. However, when we draw a word we let neighboring letters overlap half-way; i.e., we superpose neighboring halves of neighboring letters. This does not lead to any loss of information, because edges in a letter (circuit-slice) always point to the middle column only. In this picture the word appears as a circuit; we call this the circuit representation of the word. Note that the circuit representation of a word of length k has $k + 2$ columns. See Figure 2(a) for an example.

The complete language A_n for n-state 2AFAs consists of all the words whose circuit representation has the following property: When the n vertices in the rightmost column of the circuit all carry a truth value 1, and all other sources (vertices without in-edges) in the circuit carry a truth value 0, then this truth value assignment "*forces* a 1" to appear on the top vertex in the leftmost column of the circuit. See Figure 2(b) for an example.

Definition of "forcing a 1." A truth value 1 is *forced* at a vertex i (in the leftmost column of the circuit) iff, in every truth-value assignment which is consistent (with the gates of the circuit and with the 1's at the right end and the 0's at the other sources), 1 appears at node i.

The complete language for n-state 2AFAs is analogous to the complete languages for n-state 2NFAs or 1NFAs of Sakoda and Sipser [SS] (where all the nodes can be viewed as OR-gates). Note that here we direct the edges in the way one would expect in a circuit (the input wires to a gate point to that gate, and the output wires point out); but the movement of the 2AFA head is actually opposite to the direction of these edges (see [SS], where the opposite convention is more natural).

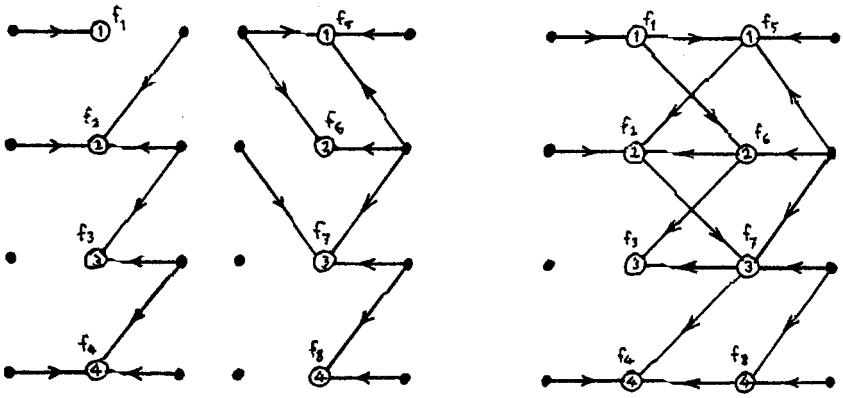
In this paper we use the following definition of *reduction* (due to [SS]) which is particularly suited for regular languages and state-complexity.

Definition of "reduction." Let $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Delta^*$, where Σ and Δ are alphabets; L_1 reduces to L_2 iff an l.p.hom. φ: $\Sigma^* \to \Delta^*$ exists, and two words $x, y \in \Delta^*$ exist such that, for all $w \in \Sigma^*$, $w \in L_1 \Leftrightarrow x \cdot \varphi(w) \cdot y \in L_2$.

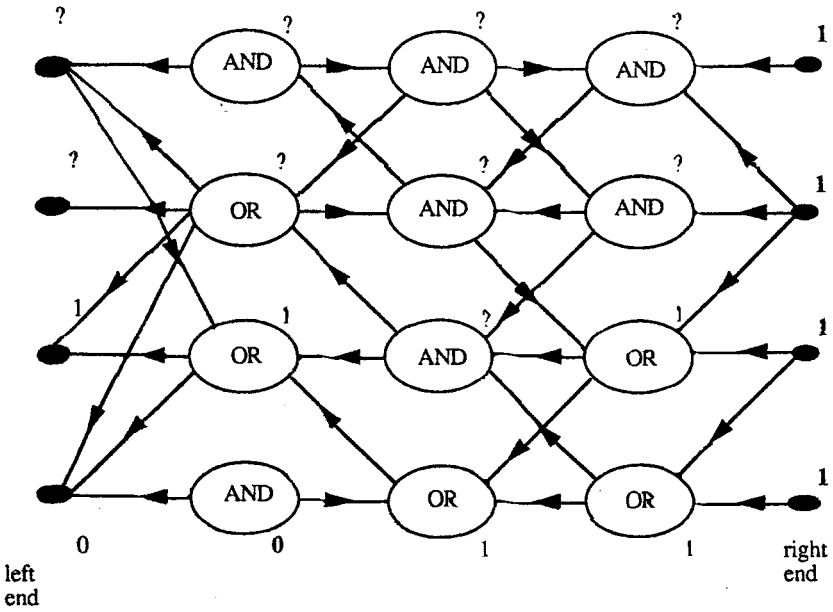
Thanks to the next lemma we only need to consider the complete laguage A_n in order to prove Theorem 4.1(a) and (b).

Lemma 4.4.

(1) *The language $L \subseteq \Sigma^*$ is accepted by an n-state 2AFA iff L reduces to A_n (the complete language for n-state 2AFAs).*



(a)



(b)

Fig. 2. (a) Two circuit-slices, and the circuit obtained by letting the two slices overlap half-way; here $f_1, f_2, \ldots, f_8$, denote boolean functions. (b) A word (of length 4) of circuit-slices (with neighboring column letters overlapping). The truth values in bold are directly given; the other values are forced; the "?" indicates that the truth value on that vertex is not forced.

(2) *If* L_1 *reduces to* L_2, *and* L_2 *is in* l.p.hom[n *state 2DFA*], *then* L_1 *is in* l.p.hom[$n + 4$ *state 2DFA*].

Proof. (1) Suppose L is recognized by a 2AFA $(Q, \Sigma, \delta, q_0, F)$; the state-transition function is δ: $Q \times \Sigma \to$ "set of boolean functions over the set of boolean variables $Q \times \{-1, +1\}$." Here $Q \times \{-1\}$ represents the states reachable by left-movement from state q on input a; and $Q \times \{+1\}$ represents the states reachable by right-movement on q and a. It is easier to think first of a 2NFA (which is a special case of a 2AFA in which all states carry the OR function, applied to a subset of $Q \times \{-1, +1\}$); then $\delta(q, a)$ is the OR of a subset of the set of variables $Q \times \{-1, +1\}$; equivalently, $\delta(q, a)$ is viewed as a subset of $Q \times \{-1, +1\}$ (then $Q \times \{-1\} \cap \delta(q, a)$ represents the set of states reachable from q and a by a left-move, and $Q \times \{+1\} \cap \delta(q, a)$ does the same for right-moves). In the case of general 2AFAs, we have arbitrary boolean functions attached to the states, and $\delta(q, a)$ is such a function over a subset of the set of variables $Q \times \{-1, +1\}$. Note also that in a general 2AFA, a different boolean function can be attached to the same state for different input letters; so it is more correct to say that boolean functions (over a subset of the set of variables $Q \times \{-1, +1\}$) are attached to the *transitions*. The 2AFA has n states, so we can assume $Q = \{1, \ldots, n\}$, with $q_0 = 1$. We also have endmarkers, as usual.

To reduce L to A_n we use the following l.p.hom. φ and the following two words x and y: φ: $a \in \Sigma \to \varphi(a) =$ "circuit-slice of a."

The "circuit slice of a" is defined as follows: Each one of the three columns of the circuit-slice of a has n vertices; moreover, for each vertex i in the middle column of the slice ($1 \leq i \leq n$), we have:

The vertex i is labeled by the boolean function attached to the state i, with respect to the input letter a.

If a state p appears as $(p, -1)$ in the boolean expression for $\delta(i, a)$, then an edge points from p (in the left column) to i (in the middle column).

If a state q appears as $(q, +1)$ in the boolean expression for $\delta(i, a)$, then an edge points from q (in the right column) to i (in the middle column); no other edges exist in the circuit-slice of a.

The word x is simply the circuit-slice of the left endmarker "⟨" (which is defined in the same way as the circuit-slice of a letter $a \in \Sigma$).

The word y is obtained from the circuit-slice of the right endmarker "⟩" by modifying its boolean functions as follows: All edges from the right column that are incident to accept states ($\in F$) are set to a boolean value 1, and the other edges from the right column are set to 0 (this modification replaces the boolean functions on "⟩" by constant functions); all edges incident with the right column of the modified circuit-slice are then removed.

It follows immediately from the definitions that the 2AFA accepts a word w iff $x \cdot \varphi(w) \cdot y \in A_n$.

Proof of the *converse*. First, it is straightforward to check that A_n is accepted by an n-state 2AFA. Next, assume that $L_1 \subseteq \Sigma^*$ reduces to $L_2 \subseteq \Delta^*$ (i.e., an l.p.hom. φ: $\Sigma^* \to \Delta^*$ exists, and two words $x, y \in \Delta^*$ exist such that, for all $w \in \Sigma^*$, $w \in$

$L_1 \Leftrightarrow x \cdot \varphi(w) \cdot y \in L_2$); and assume L_2 is accepted by an n-state 2AFA A_2. We construct a 2AFA A_1 accepting L_1. The state set of A_1 is the same as the state set of A_2; the two machines also have the same start state q_0, and the endmarkers $\langle$ and $\rangle$. On a tape $\langle w \rangle$ (with $w \in \Sigma^*$) the machine A_1 does the following: A_1 behaves on a letter $a \in \Sigma$ in the same way as A_2 behaves on the letter $\varphi(a) \in \Delta$. When A_1 reads $\langle$, it makes a transition to the same boolean function as the one that A_2 produces as a result of processing the word $\langle x$. Similarly, when A_1 reads $\rangle$, it makes a transition to the same boolean function as the one that A_2 produces as a result of processing the word $y\rangle$. In this way, A_1 accept w iff A_2 accepts $x \cdot \varphi(w) \cdot y$.

(2) Assume $L_1 \subseteq \Sigma^*$ reduces to $L_2 \subseteq \Delta^*$ (i.e., an l.p.hom. φ: $\Sigma^* \to \Delta^*$ exists, and two words $x, y \in \Delta^*$ exist such that, for all $w \in \Sigma^*$, $w \in L_1 \Leftrightarrow x \cdot \varphi(w) \cdot y \in L_2$). Assume L_2 is accepted by an n-state nondeterministic Hennie machine H_2, which visits every position of the tape at most once in the same state (recall Theorem 3.1); let Γ_2 be the total tape alphabet of H_2. We may assume that $\Sigma \cap \Gamma_2 = \varnothing$.

We construct a nondeterministic Hennie machine H_1 accepting L_1, with $n+4$ states, which visits every position of the tape at most once in the same state; Lemma 4.4(2) then follows, by Theorem 3.1. The state set of H_1 is the same as the state set of H_2, except that a new start state q_n, and another three new states, q_{n+1}, q_{n+2}, q_{n+3}, are added. The two machines have endmarkers "$\langle$" and "$\rangle$." The tape alphabet of H_1 is $\Gamma_1 = \Sigma \cup \Gamma_2 \cup \{\S x\} \times \Gamma_2 \cup \Gamma_2 \times \{y\S\}$.

In summary, H_1 first replaces its input $w = a_1 a_2 \cdots a_{m-1} a_m \in \Sigma^*$ by the word $(\S x, \varphi(a_1))\varphi(a_2) \cdots \varphi(a_{m-1})(\varphi(a_m), y\S) \in \Gamma_1^*$; then H_1 simulates H_2 on this new word, except for a little adjustment regarding $(\S x, \varphi(a_1))$ and $(\varphi(a_m), y\S)$; the symbol § is introduced to distinguish between left and right. It is important to note that no state of H_2 has any transition defined on a letter in Σ (since $\Sigma \cap \Gamma_2 = \varnothing$) nor on any of the other new letters, so we can introduce such transitions into H_1 without disturbing the simulation of H_2.

In more detail, on tape $\langle a_1, a_2 \cdots a_{m-1} a_m \rangle$ the machine H_1 does the following: It starts in state q_n on the letter a_1, replaces a_1 by $(\S x, \varphi(a_1))$, moves right, and goes to state q_{n+1}. It keeps moving right in state q_{n+1} while changing the tape into

$$\langle (\S x, \varphi(a_1))\varphi(a_2) \cdots \varphi(a_{m-1})\varphi(a_m) \rangle.$$

When $\rangle$ is encountered, H_2 moves left on $\rangle$ and goes to state q_{n+2}. In this state it changes $\varphi(a_m)$ to $(\varphi(a_m), y\S)$, moves left and goes to state q_{n+3}. In this state H_1 moves left (without changing the tape) until it reads $(\S x, \varphi(a_1))$. Thereupon it goes to the old start state q_0 of H_2 and starts the simulation of H_2 on tape $\langle (\S x, \varphi(a_1))\varphi(a_2) \cdots \varphi(a_{m-1})(\varphi(a_m), y\S) \rangle$. For this simulation, H_2 is modified as follows: When $(\S x, \varphi(a_1))$ is read in some state q, H_1 goes to the same state(s) that H_2 reaches when it processes the word $\langle x\varphi(a_1)$ (starting in state q with the head on the letter $\varphi(a_1)$). Suppose that in this process H_2 replaces the word $\langle x\varphi(a_1)$ by $\langle x'b$ (with $|x'b| = |x\varphi(a_1)|$), and leaves on the right (or never leaves this portion of tape); then H_1 replaces the letter $(\S x, \varphi(a_1))$ by the letter $(\S x', b)$ and moves right (respectively, halts without making a transition). In a similar way, H_1 simulates H_2 on other letters of the form $(\S x'', c)$ (with $|x''| = |x|$, $x'' \in \Gamma_2^*$, $c \in \Gamma_2$), near the left end of the tape. On the right end of the tape on letters of the form $(d, y''\S)$ (with $|y''| = |y|$, $y'' \in \Gamma_2^*$, $d \in \Gamma_2$) the simulation is also similar, with left and

right interchanged; here, if H_2 never leaves the tape portion $dy''\rangle$, but halts on $\rangle$ in an accept state, then H_1 also moves to $\rangle$ and goes into the same accept state. This way, H_1 accept w iff H_2 accepts $x \cdot \varphi(w) \cdot y$. □

We first prove Theorem 4.1(b).

Halting 2AFAs, and in particular 1AFAs, correspond to circuits (in the complete language A_n) that are *acyclic*.

Every such circuit can be written as an l.p.hom. image of a circuit with truth values already drawn on all the gates. The l.p.hom. just consists in dropping the truth values (thus returning the circuit itself). An $O(n)$-state 2DFA can check that:

(1) At the right end of the circuit all the vertices have a truth-value 1 and that the top vertex at the left end has a 1.
(2) The *truth-value assignment on all the gates is consistent*[1] *with the circuit* (i.e., a gate has a 1 in the assignment iff a 1 is returned when the boolean function of the gate is applied to the boolean values assigned to the vertices that point to the gate).
(3) The nodes with in-degree 0 have a boolean value 0.
(4) For any two neighboring letters (circuit-slices) a and b of the circuit word, the middle column of a and the left column of b have identical truth-value assignments; and also, the right column of a and the middle column of b have the same truth-value assignment.

To check these properties the 2DFA needs only $O(n)$ states: Property (1) is easy to check (with a constant number of states). Properties (2) and (3) are just restrictions on the letters that may appear in accepted words, so no new states are needed to check them. To check property (4), a 2DFA makes $2n$ sweeps over the circuit-word with its truth-value assignment (alternating left-to-right and right-to-left sweeps, n times). In the ith ($1 \leq i \leq n$) left-to-right sweep, the 2DFA checks that in *row i* of the circuit, the middle column of a and the left column of b have the same truth value (this is done for all neighboring letters a, b); and in the ith ($1 \leq i \leq n$) right-to-left sweep the 2DFA checks that in *row i* of the circuit, the right column of a and the middle column of b have the same truth value. Each sweep uses a constant number of states, so $O(n)$ states are used in total.

We now prove Theorem 4.1(a).

When the circuit contains cycles, properties (1), (3), and (4) above need no changes; but, regarding property (2), it is no longer enough to check whether the truth values on the gates are consistent with the circuit; we want the 1's at the right end to *force* a 1 to appear at the top vertex at the left end (not just to be consistent with it). In the cyclic case, consistency does not imply forcing (for example, recall the *RS*-latch, which is bistable; it has two consistent truth-value assignments when the input is $R = 0$, $S = 0$; no output value is forced in that case).

To check whether a 1 is forced at the top vertex in the leftmost column, a deterministic Hennie machine does *depth-first search* in the circuit. The search process is

[1] "Consistent" is the term used in Logic; in Circuit Theory the term "stable" is used instead.

greatly simplified by the next lemma, which implies that we may assume that all the boolean functions in our circuit words are monotone. (By definition, a boolean function is *monotone* iff it can be represented by a boolean expression using AND and OR only, without negation.)

Lemma 4.5 (Monotonicity). *Every general n-state 2AFA is equivalent to a 2AFA with 2n states, whose boolean functions are all monotone.*

Proof. We use a standard idea for the elimination of negations (which has also been used for alternating Turing machines, see p. 90 of [Ko] and p. 120 of [CKS]). Suppose in the original 2AFA (with state set $Q, |Q| = n$) the boolean function $\beta(q_1, \ldots, q_n, p_1, \ldots, p_n)$ is attached to $(q, \sigma) \in Q \times \{-1, +1\}$. The new 2AFA will have state set $\{+, -\} \times Q$; we attach the following monotone boolean functions to $((+, q), \sigma)$ (resp. $((-, q), \sigma)$):

To $((+, q), \sigma)$ we attach

$$\beta'((+, q_1), (-, q_1), \ldots, (+, q_n), (-, q_n), (+, p_1), (-, p_1), \ldots, (+, p_n), (-, p_n)),$$

which is obtained from β by writing β in disjunctive normal form and then replacing each negated variable $\overline{v}$ by $(-, v)$, and each nonnegated variable v by $(+, v)$. To $((-, q), \sigma)$ we attach β'' (over the same $4n$ variables as β'), obtained from $\overline{\beta}$ (the negation of β, put into disjuntive normal form) in the same way as β' is obtained from β. □

The deterministic Hennie machine carries out a *depth-first search* in the circuit as follows: Initially, the only vertices of the circuit that carry a truth value are the vertices in the rightmost column (which carry a 1) and the other sources (vertices without in-edges, which carry a 0). All other vertices carry a value "?". The search starts at the top vertex in the leftmost column of the circuit word.

At an arbitrary instant during the search, suppose a vertex v with value "?" has been reached; consider the values ($\in \{0, 1, ?\}$) of the vertices that point to v via the in-edges of v.

Case 1: *Finding and Backtracking.* If these values determine the truth value ($\in \{0, 1\}$) on v (by application of the boolean function attached to v), then this truth value is written on v (e.g., if v is an AND gate and some in-edge of v carries a 0, then 0 will be written on v). Now the search continues by backtracking along a colored edge (see Case 2 for the coloring actions) pointing away from v; at this moment we also erase all the in-edges of v. Note that during backtracking, the edges are traversed in the direction in which they point.

Case 2: *Forward Search.* If the values applied to v by its in-edges do not determine a truth value ($\in \{0, 1\}$) on v, then an in-edge of v carrying the value "?" is chosen and colored and the corresponding incident vertex v' is examined next. So, in the forward search, edges are traversed in the reverse of the direction in which they point. The only nonstraightforward part is the handling of *cycles*: Suppose that during depth-first search a directed cycle is detected (the Hennie machine detects this when, along its search path, it sees a colored edge pointing out of the vertex v' that is examined next in the forward

search). Then the machine continues the search along an in-edge of v other than the one(s) that would close a directed cycle. If all in-edges of v would close cycles, then the truth value 0 is written on v. Indeed, in a *monotonic* circuit, the existence of such cycles implies that the truth value at vertex v is not forced, and hence it cannot help force the truth value of other gates. In a monotonic circuit, this is equivalent (regarding the forcing of truth values on other vertices) to giving a 0 to this gate.

This Hennie machine needs only $O(n)$ states (to remember which vertex it is visiting within an n-vertex column). Each vertex is visited only $O(n)$ times (since each vertex has degree $O(n)$). This proves 4.1(a). □

Proof of Theorem 4.1(c). Blum and Hewitt [BH] give a proof (due to Albert Meyer) that the language recognized by an n-state 1-pebble 2NFA is regular, by constructing an equivalent deterministic Hennie machine with $O(2^{n^2})$ states. By slightly modifying their proof we obtain a nondeterministic (resp. deterministic) Hennie machine with $4n^2 + n + O(1)$ states, which is equivalent to the given nondeterministic (resp. deterministic) n-state 1-pebble machine; we proceed as follows:

The Hennie machine first makes two passes over the input tape. In the first pass (from left to right), it uses the 2NFA (or 2DFA) which is obtained from the 1-pebble machine when no pebble is used; the equivalent 1DFA of the 2NFA (or 2DFA) as constructed by Shepherdson [Sh] is considered, and the *Shepherdson tables* are written on a second track of the tape. If $u \in \Sigma^*$ is a prefix of the input, we denote the corresponding Shepherdson table by $[\langle u^{\leftarrows}_{\rightarrow}]$ (see [B1], or the Background and Notation section at the end of this proof). In the second pass (from right to left), the reverse is done: for every suffix v of the input, the table $[^{\rightarrow}_{\leftarrow}v\rangle]$ (also defined in [B1] and below) is written on a third track of the tape.

So far the procedure (as in [BH]) is deterministic, and requires $\leq 2^{n^2}$ states (but this number will be reduced shortly): in the first pass there are 2^{n^2} possible tables of the form $[\langle u^{\leftarrows}_{\rightarrow}]$; a table $[\langle u^{\leftarrows}_{\rightarrow}]$ is remembered in the state while the table $[\langle ua^{\leftarrows}_{\rightarrow}]$ of the neighboring cell (with input letter a) is being printed. The second pass is handled similarly.

However, we can modify the procedure so that $[\langle ua^{\leftarrows}_{\rightarrow}]$ can be generated from $[\langle u^{\leftarrows}_{\rightarrow}]$ without remembering all of $[\langle u^{\leftarrows}_{\rightarrow}]$ in the state. First $[\langle u^{\leftarrows}_{\rightarrow}]$ (which is a set of pairs of states) is copied over to the neighboring cell on the right *one pair of states at a time*, in lexical order; e.g., this takes $\leq 2n^2 + O(1)$ states for the Hennie machine. (A different "color" is used to print this new copy of $[\langle u^{\leftarrows}_{\rightarrow}]$.) Second, once $[\langle u^{\leftarrows}_{\rightarrow}]$ and a are together in this cell, $[\langle ua^{\leftarrows}_{\rightarrow}]$ can be directly printed. None of these $\leq 2n^2 + O(1)$ states occurs more than once at the same position. The second pass is handled similarly, and another $\leq 2n^2 + O(1)$ states are introduced.

We now directly simulate a 2NFA N_2 (or a 2DFA), whose head permanently carries the pebble; N_2 has n states. (We do not need the remaining three passes of [BH] then.) When the 1-pebble machine drops the pebble at a certain position, the two Shepherdson tables on the tape of the Hennie machine (at that position) contain the information needed to know in which state the 1-pebble machine is when it comes back to this position. (See the hint of Exercise 3.19, p. 73 of [HU].) It takes $\leq n$ states to simulate this 2NFA or 2DFA. No state occurs twice at the same position during an accepting computation. Note that now the Hennie machine becomes nondeterministic if the original 1-pebble machine was nondeterministic.

Proof of the second paragraph of Theorem 4.1(c): In order to simulate a 1-pebble 2NFA (nondeterministic) by a *deterministic* Hennie machine we use the same first two (deterministic) passes as above (and the Shepherdson tables $[\langle u^{\leftrightarrows}_{\rightarrow}]$, $[^{\rightarrow}_{\leftarrow} v\rangle]$ are printed on the tape); this uses $\leq 4n^2 + O(1)$ states, and no state occurs twice at the same tape position. Next, instead of directly simulating the 2NFA N_2 above, we simulate the deterministic Hennie machine which is equivalent to N_2 (i.e., we use the proof of Theorem 4.1(a) in the special case of a 2NFA). This deterministic Hennie machine has $O(n)$ states, with no state occurring more than $O(n)$ times at the same tape position. Thus, overall the n-state 1-pebble 2NFA is simulated by a deterministic Hennie machine with $O(n^2)$ states; $O(n)$ of these states occur at most $O(n)$ times at the same position, and the remaining $O(n^2)$ states occur just once at the same position.

Background and Notation for the Proof of 4.1(c)

Definition. The *Shepherdson tables* (or *global state transitions*) of a 2NFA N on a word $u \in (\Sigma \cup \{\langle, \rangle\})^+$ are the following four relations on the state set Q (see [B1], where however a slightly different kind of 2NFA was used).

The relation $[\rightarrow u \rightarrow] \subseteq Q \times Q$ is defined as follows: $(q_1, q_2) \in [\rightarrow u \rightarrow]$ iff there is a computation of N starting at the left end of u in state q_1, and during this computation the reading head of N stays on u, and eventually leaves u on the right end in state q_2.

The relation $[^{\rightarrow}_{\leftarrow} u] \subseteq Q \times Q$ is defined as follows: $(q_1, q_2) \in [^{\rightarrow}_{\leftarrow} u]$ iff there is a computation of N starting at the left end of u in state q_1, and during this computation the reading head of N stays on u, and eventually leaves u on the left end in state q_2.

The relation $[u^{\leftrightarrows}_{\rightarrow}] \subseteq Q \times Q$ is defined like $[^{\rightarrow}_{\leftarrow} u]$, exchanging left and right; $[\leftarrow u \leftarrow] \subseteq Q \times Q$ is defined like $[\rightarrow u \rightarrow]$, exchanging left and right.

We have the following fact (see [B1]), from which it follows that $[u^{\leftrightarrows}_{\rightarrow}]$ (resp. $[^{\rightarrow}_{\leftarrow} v]$) and $a \in \Sigma$ determine $[ua^{\leftrightarrows}_{\rightarrow}]$ (resp. $[^{\rightarrow}_{\leftarrow} av]$).

Fact. *If* $u, v \in (\Sigma \cup \{\langle, \rangle\})^+$, *then we have for the concatenation* uv:

$$[\rightarrow uv \rightarrow] = [\rightarrow u \rightarrow]([^{\rightarrow}_{\leftarrow} v][u^{\leftrightarrows}_{\rightarrow}])^*[\rightarrow v \rightarrow],$$

$$[^{\rightarrow}_{\leftarrow} uv] = [^{\rightarrow}_{\leftarrow} u] \cup [\rightarrow u \rightarrow]([^{\rightarrow}_{\leftarrow} v][u^{\leftrightarrows}_{\rightarrow}])^*[^{\rightarrow}_{\leftarrow} v][\leftarrow u \leftarrow],$$

$$[uv^{\leftrightarrows}_{\rightarrow}] = [v^{\leftrightarrows}_{\rightarrow}] \cup [\leftarrow v \leftarrow]([u^{\leftrightarrows}_{\rightarrow}][^{\rightarrow}_{\leftarrow} v])^*[u^{\leftrightarrows}_{\rightarrow}][\rightarrow v \rightarrow],$$

$$[\leftarrow uv \leftarrow] = [\leftarrow v \leftarrow]([u^{\leftrightarrows}_{\rightarrow}][^{\rightarrow}_{\leftarrow} v])^*[\leftarrow u \leftarrow].$$

Notation: Juxtaposition of relations denotes composition of relations (defined in the usual way). The star *, *applied to a relation, denotes reflexive-transitive closure.* □

Let $[u] = ([\rightarrow u \rightarrow], [^{\rightarrow}_{\leftarrow} u], [u^{\leftrightarrows}_{\rightarrow}], [\leftarrow u \leftarrow])$ (a quadruple of Shepherdson tables). By the above fact, the knowledge of $[u]$ and $[v]$ determines $[uv]$; thus we can define a product of quadruples by $[u][v] = [uv]$; this product is associative.

Proof of Theorem 4.1(d). The proof is very similar to the proof of Theorem 4.1(c). However, since we now deal with an n-state 1-pebble 2AFA A, the more complicated Shepherdson tables $[\langle w \rightarrow]$, $[\leftarrow w\rangle]$ are used; they associate a boolean function on n boolean variables with each state of A. (For more details on the Shepherdson tables of a 2AFA see Background and Notation, below; these tables are also used implicitly in [LLS] for a more special kind of 2AFAs.) There are 2^{2^n} possible boolean-valued functions on n boolean variables; so there will be at most $(2^{2^n})^n = 2^{n2^n}$ different Shepherdson tables for a given n-state 2AFA (of the form $(q)[\langle u \rightarrow]$ or $(q)[\leftarrow v\rangle]$, as defined in the Background and Notation at the end of this proof), and thus at most $2^{n2^n} \cdot 2^{n2^n} = 2^{2n2^n}$ pairs of such tables. The deterministic Hennie machine needs at most $O(n2^n)$ states (with no state occurring twice at the same position) in order to copy one such table from one tape-cell to a neighboring tape-cell (this is done just as in the proof of Theorem 4.1(c), but now $O(n^2)$ is replaced by $O(n2^n)$).

On this tape (with the pair of Shepherdson tables written down at every position) we can simulate the n-state 1-pebble 2AFA by a 2AFA A_2 with n states, whose head permanently carries the pebble. When the 1-pebble 2AFA drops the pebble at a certain position, the Shepherdson tables that are written on the tape (at that position) tell A_2 which boolean function of states the 1-pebble 2AFA will be in when its head comes back to this position.

Next, we simulate the new 2AFA A_2 by a deterministic Hennie machine (as described in the proof of Theorem 4.1(a); this machine has $O(n)$ states with no state occurring more than $O(n)$ times at the same tape position.

Thus, overall the n-state 1-pebble 2AFA is simulated by a deterministic Hennie machine with $O(n2^n)$ states; $O(n)$ of these $O(n2^n)$ states occur at most $O(n)$ times at the same position, and the remaining $O(n2^n)$ states occur just once at the same position.

Background and Notation for the Proof of 4.1(d). The Shepherdson tables $[w \rightarrow]$ and $[\leftarrow w]$ of $w \in (\Sigma \cup \{\langle, \rangle\})^+$ with respect to a 2AFA A (with state set Q and alphabet Σ) are functions from Q into the set of all boolean-valued functions over the set of boolean variables $Q \times \{-1, +1\}$.

The function $[w \rightarrow]$ is defined as follows: First we consider the circuit of w, as introduced in the proof of Lemma 4.4(1). Then for $q \in Q$ we define $(q)[w \rightarrow]$ to be the boolean-valued function (over the the set of boolean variables $Q \times \{-1, +1\}$) which is computed by the circuit if the vertex q in the middle column of the rightmost circuit-slice is chosen as the output port. There are $2n$ input ports in this circuit: the n vertices in the left column of the leftmost slice (these vertices are labeled by the set of boolean variables $Q \times \{+1\}$, corresponding to a right-movement of the 2AFA); and the n vertices in the right column of the rightmost slice (these vertices are labeled by the set of boolean variables $Q \times \{-1\}$).

In a similar way we define $(q)[w \rightarrow]$. Here the output port is vertex q in the middle column of the leftmost circuit-slice. It can easily be seen $[w \rightarrow]$ (resp. $[\leftarrow w]$) and $a \in \Sigma$ determine $[wa \rightarrow]$ (resp. $[\leftarrow aw]$).

It can also be observed that $(q)[\langle u \rightarrow]$ really only depends on the set of n boolean variables $Q \times \{+1\}$, as the 2AFA cannot make left-moves on $\langle$; and $(q)[\leftarrow w\rangle]$ is only a function of the set of n variables $Q \times \{-1\}$. □

5. The Language $\{a^m\}$ (for Fixed $m > 1$)

It turned out that one-word languages over a one-letter alphabet are sufficient to prove Theorem 4.3; here we give a more detailed proof of Theorem 4.3. In addition, we give other results about such languages, and in particular we disprove a conjecture of Meyer and Fischer, and a conjecture of Chrobak. The study of $\{a^m\}$ is a study of the number m, so it is not surprising that number theory is useful here.

Fact 5.1. *The unary one-word language $\{a^m\}$ is recognized by a deterministic Hennie machine H with $\leq 2\lfloor \log_2 m \rfloor + O(1)$ states, with no state occurring more than once at the same tape position.*

Proof. For the exposition, we first consider the case $m = 2^n$. The machine H uses an alphabet $\{a, b\}$. For $\{a^{2^n}\}$ it first counts mod 2 and replaces every second a by b (until the right endmarker $\rangle$ is reached). Next H goes left while replacing every oddly placed b by an a (and leaving the other b's alone), until the left endmarker $\langle$ is reached; this requires a second mod-2-counter. Now, going right again, it again replaces every oddly placed b by a (and leaves the other b's), using a third mod-2 counter. So, H performs $n - 1$ sweeps. Altogether it uses n different mod-2 counters (plus another state to move left all the way to the right at the end, if n is even). So $2n + O(1)$ states are used, and no state occurs twice at the same position. Finally, we accept if after these n (or $n + 1$) sweeps we end up with a tape containing one b at the right (just before $\rangle$) and a elsewhere.

To accept $\{a^m\}$ for arbitrary $m > 1$, essentially the same construction is used: There are $\lfloor \log_2 m \rfloor + 1$ mod-2 counters; the kth counter checks the kth bit in the binary representation of m; when the head begins the next pass (after the kth pass) it marks ("erases") all the letters until a b is found; it leaves this b and the remaining letters alone, and continues moving. In each pass only the nonmarked ("nonerased") letters are read. □

Fact 5.2. *Any 2NFA recognizing $\{a^m\}$ (for any fixed $m \geq 1$) requires more than m states.*

Proof. We consider a 2NFA (as defined in [HU]), with state set Q. By contradiction, assume $|Q| \leq m$; by adapting the Pumping Lemma for Regular Languages (see [HU]) we show that some other word a^M (with $M > m$) is then also accepted by the 2NFA.

For the input a^m we consider some (fixed) accepting computation of the 2NFA.

A *left-to-right traversal* is a subsegment of this computation, in which the head starts at the left endmarker $\langle$ and reaches the right endmarker $\rangle$, and no endmarker is visited inbetween. Similarly, one defines *right-to-left traversal.* An accepting computation must contain at least one traversal.

A *left-to-left nontraversal* is a maximal subsegment of the above computation, in which the head starts and ends at the left endmarker $\langle$, and during which the right endmarker $\rangle$ is never visited. A *right-to-right nontraversal* is similarly defined.

An accepting computation can be factored uniquely into a sequence of traversals and nontraversals.

We now apply a "pumping" argument to the traversals. In every left-to-right traversal we consider the first time that each position on a^m is visited during this traversal; this corresponds to m state-transitions, in each of which a is read from left to right; $m+1$ states occur during these m transitions, and so some state q must appear twice (since $|Q| \leq m$). Also, since we look only at *first* visits in the traversal, the second occurrence of q must be at a position strictly to the right of the first occurrence of q.

Similarly, in every right-to-left traversal, a state p occurs twice, with the second occurrence of p strictly to the left of the first occurrence.

Suppose there are k traversals. In the ith traversal let d_i $(1 \leq i \leq k)$ be the distance between the two occurrences of the repeated state considered above.

Let $M = m + \prod_{i=1}^{k} d_i$. We claim that the 2NFA accepts a^M.

Indeed, the 2NFA has an accepting computation of a^M, constructed from the above accepting computation of a^m, as follows: The nontraversals of a^M are those of a^m. The traversals of a^M are obtained from those of a^m by "pumping" between the chosen repeated states (the pumping is possible because $\Pi_i d_i$ is a multiple of each d_i); thus the beginning and ending states of each traversal are the same in a^M and a^m. □

The argument in this theorem is similar to the proof of Theorem 2.1 in [LSH]. Fact 5.2 was also given by Ibarra and Sahni [IS], but their 2NFAs do not have endmarkers on the tape and, thus, are weaker.

In the above "pumping" argument it is crucial that the input is over a one-letter alphabet, because we "pump" subwords that are at different positions for different traversals. Suppose the alphabet contains two letters a, b (or more). Then a singleton-language $\{w\}$, with $|w| = m$, could possibly be recognized by a 2DFA with far fewer states than m. For example, $\{(ab)^{m/2}\}$ (for m even) is recognized by a 2DFA with $m/2 + O(1)$ states. (The 2DFA first goes right and checks if the input belongs to $(ab)^*$, using a mod-2 counter; then it comes back to the left; all this uses $O(1)$ states. Next it counts the number of a's using $m/2$ states.) More generally, the singleton language $\{(ab^{k-1})^{m/k}\}$ (where k divides m) is recognized by a 2DFA with $m/k + k + O(1)$ states. (The minimum number of states obtained this way is $2\sqrt{m} + O(1)$, when $k = \sqrt{m} + O(1)$, assuming m is a square.)

The proof of Fact 5.1 (that $\{a^m\}$ is recognized by a deterministic Hennie machine with $2 \cdot \lfloor \log_2 m \rfloor + O(1)$ states) also provides us with a word w (over an alphabet with $\lfloor \log_2 m \rfloor$ letters) such that $|w| = m$, but $\{w\}$ is recognized by a 2DFA with $2\lfloor \log_2 m \rfloor + O(1)$ states.

Corollary of Fact 5.2. *For every $n > 0$ there is a language (for example, $\{a^{n-1}\}$) which is recognized by a 2NFA with n states but with no less than n states. The same holds true for 2DFAs (and, as is well known, for 1NFAs and 1DFAs). In other words, there is no gap in the state-complexity hierarchy for these types of automata.*

Fact 5.3. *The language $\{a^m\}$ (for any fixed $m \geq 1$) is recognized by a 1-pebble 2DFA, and also by a $\forall\exists$-1AFA (i.e., of type Π_2), with $\leq (\ln m)^2 / \ln \ln m + O(1)$ states (where "ln" is the natural logarithm).*

Fact 5.3 disproves a "strong conjecture" of Meyer and Fischer [MF, p. 190], according to which a 1-pebble 2DFA would have needed $\geq 2^n$ states to recognize $\{a^{2^n}\}$. The fact also disproves a conjecture of Chrobak (Final Remarks in [Ch]), according to which unary 1AFAs and unary 2DFAs would have been polynomially equivalent regarding state-complexity.

In [Le1] Leiss proved (using a construction of [BL]):

If a language L is recognized by a 1DFA *with n states, then L^{rev} is recognized by a* 1AFA *with $\leq \lceil \log_2 n \rceil$ states; in particular, for any unary language L: if L has an n-state* 1DFA, *then L has a* 1AFA *with $\leq \lceil \log_2 n \rceil$ states.*

("L^{rev}" is the reverse, or mirror image, of L; see [L2] for a proof of the necessity for the reverse.)

Thus Leiss' result is stronger than Fact 5.3, for general 1AFAs. The 1AFA of Fact 5.3 has the advantage of being of type Π_2 (whereas the 1AFAs of [L1] and [BL] not only use an unbounded number of alternations, but arbitrary boolean functions have to be attached to the transitions in addition to the usual AND, OR, NOT). Fact 5.3 and Leiss' results are surprising because it was thought that a pumping argument (like for 2NFAs in Fact 5.2) should work for 1-pebble 2DFAs and for 1AFAs (especially for type Π_2); but it cannot.

Proof of Fact 5.3. The following classical facts from number theory are used:

(1) The Chinese Remainder Theorem (see, e.g., p. 117 of [A]).

(2) The Prime Number Theorem [A, pp. 74, 79–80], in the following two equivalent forms:

$$p_k = k \ln k(1 + \varepsilon(k)) \qquad \text{for some function } \varepsilon(\cdot) \text{ with } \lim_{k\to\infty} \varepsilon(k) = 0;$$

$$\prod_{k=1}^{n} p_k = e^{n \ln n(1+\varepsilon(n))} \qquad \text{for some } \varepsilon(\cdot) \text{ with } \lim_{n\to\infty} \varepsilon(n) = 0.$$

Here p_k is the kth prime number.

(3) From the Prime Number Theorem it follows that $\sum_{k=1}^{n} p_k = \frac{1}{2} n^2 \ln n(1+\varepsilon(n))$, with $\varepsilon(n) \to 0$.

Definition. For any integer $m > 0$, we let $N(m)$ be the smallest integer n such that $m \leq \prod_{k=1}^{n} p_k$.

By the above, $N(m) = (\ln m / \ln \ln m)(1 + \varepsilon(m))$, where $\varepsilon(m) \to 0$. Also

$$\sum_{k=1}^{N(m)} p_k = \tfrac{1}{2} N(m)^2 \ln N(m)(1 + \varepsilon(m))$$

$$= \frac{1}{2} \frac{(\ln m)^2}{\ln \ln m}(1 + \varepsilon_1(m)) \qquad \text{where} \quad \varepsilon_1(m) \to 0.$$

For each k ($1 \leq k \leq N(m)$), let r_k be the unique integer such that $m \equiv r_k (\mathrm{mod}\ p_k)$ and $0 \leq r_k < p_k$. By the Chinese Remainder Theorem and by the definition of $N(m)$, m will be the (unique) smallest nonzero solution of the system of congruences $\{x \equiv r_k \bmod p_k / k = 1, \ldots, N(m)\}$.

We construct a 1-pebble 2DFA (and a 1AFA) whose number of states will be

$$2 \sum_{k=1}^{N(m)} p_k + O(1) = \frac{(\ln m)^2}{\ln \ln m}(1 + \varepsilon(m)).$$

These automata, on input a^x, first check that x is a solution to the system of congruences and, second, check that no smaller number ($\neq 0$) is a solution.

Construction of the 1-pebble 2DFA for $\{a^m\}$. On input a^x the machine does the following computation:

In the first phase the pebble stays on the left endmarker, and the head makes $N(m)$ passes over the input a^x; in the kth pass it counts mod p_k to check if $x \equiv r_k \bmod p_k$. If during one pass the congruence is not verified, the machine halts and rejects. At the end of the first phase the head goes left, to the left endmarker.

Next (second phase), the pebble is moved right one step; the input between the pebble and the right endmarker will now be counted mod p_k (for $k := 1$ to $N(m)$) until the number does *not* satisfy some congruence. If the number of a's (between the pebble and the right endmarker) satisfies all the congruences, then the machine halts and rejects. If some congruence is encountered that is not satisfied, then the head moves left to the pebble, moves it one step to the right, and starts phase 2 again.

This goes on until the pebble reaches the right endmarker, at which point the machine accepts the input. Overall, $2 \sum_{k=1}^{N(m)} p_k + O(1)$ states are used (the same mod-p_k counters are reused in all the iterations of phase 2).

Construction of a 1AFA for $\{a^m\}$. For one-way devices (and for 1AFAs in particular) no endmarkers are used on the tape; however, for an input to be accepted it is necessary that it be entirely read (for some accepting computation).

Our 1AFA starts at the left of the input a^x in a $\forall$-state; there are $N(m) + 1$ branches out of this state. The first $N(m)$ branches lead to the parallel (deterministic) execution of mod-p_k-counters ($1 \leq k \leq N(m)$). The second branch leads to another $\forall$-state; from this $\forall$ one checks, at every position (from position 2 onward), that the length of the remaining word is *not* a solution to our system of congruences (i.e., that a congruence that is *not* satisfied exists); this uses a $\exists$-state which branches into another set of mod-p_k counters ($1 \leq k \leq N(m)$). The total number of states is $2 \sum_{k=1}^{N(m)} p_k + 3$. The resulting 1AFA is of type Π_2 (i.e., $\forall\exists$). □

The details are best understood on an example. See Figure 3, which gives the state diagram for a 1AFA recognizing the language $\{a^{128}\}$ ($= \{a^{2^7}\}$); since $2 \cdot 3 \cdot 5 < 128 \leq 2 \cdot 3 \cdot 5 \cdot 7$ we have $N(128) = 4$; thus 128 is the unique minimum solution of the system of congruences $\{x \equiv 0 \bmod 2, x \equiv 2 \bmod 3, x \equiv 3 \bmod 5, x \equiv 2 \bmod 7\}$. The 1AFA for $\{a^{128}\}$ has $2 \cdot (2 + 3 + 5 + 7) + 3 = 37$ states.

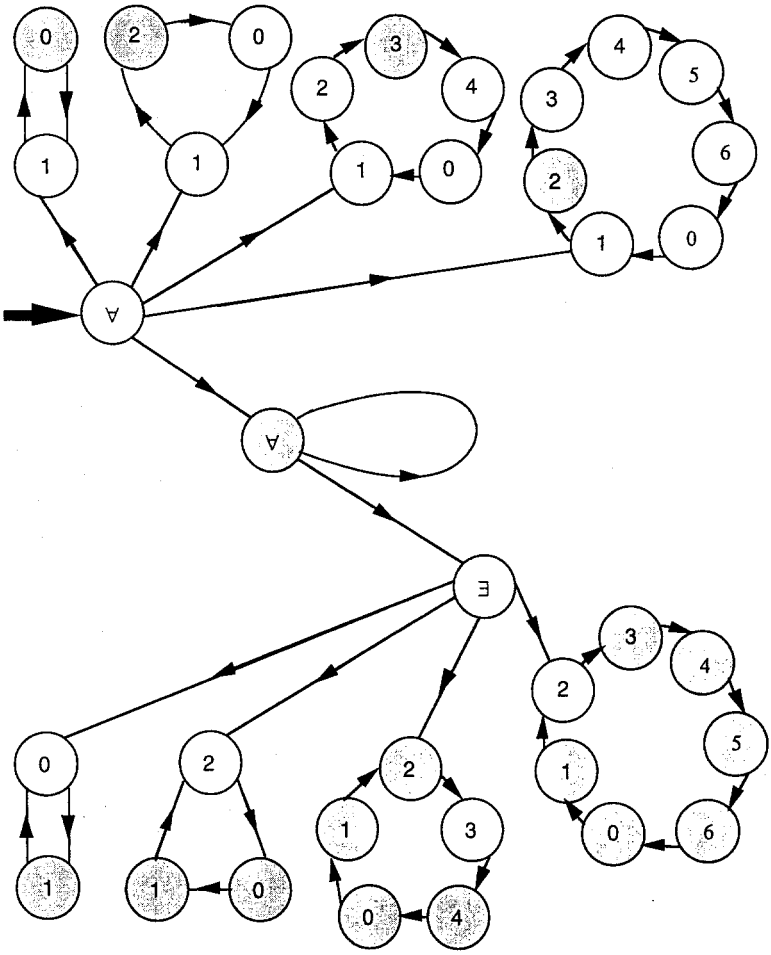


Fig. 3. A 37-state ∀∃-1AFA recognizing $\{a^{128}\}$. The start state is pointed to by a bold arrow, the accept states are shaded. The label a on all the edges has been omitted.

6. The Equivalence Problem for Certain Finite-State Devices

This section uses the results of Sections 3 and 4 to solve three problems of Jiang and Ravikumar [JR]:

(1) Is the equivalence problem for n-state 2AFAs in PSPACE?
(2) Is the equivalence problem for n-state 1-pebble 2NFAs in PSPACE?
(3) Is the equivalence problem for n-state 1-pebble 2AFAs in EXPSPACE?

We show that the answers are all "yes."

The *equivalence problem* for 2AFAs is the following: given two 2AFAs, decide whether the two 2AFAs recognize the same language. Here the input alphabet of the 2AFAs is not fixed, but is part of the input of the problem. The equivalence problem for

the other kinds of machines are stated in a similar way. In these problems we assume that an n-state 2AFA with input alphabet Σ (and similarly for 1-pebble 2NFAs or 1-pebble 2AFAs) is described by its n-by-$|\Sigma|$ state-transition table. If each of the n states (and $|\Sigma|$ letters) is expressed as a string of length $\lfloor \log_2 n \rfloor + 1$ (resp. $\lfloor \log_2 |\Sigma| \rfloor + 1$), then the input size for each of the above three problems is $\geq cn|\Sigma| \log n \log |\Sigma|$, for some constant $c \geq 1$.

The equivalence problem for 1NFAs is PSPACE-complete (see [AHU]), so the first two problems are PSPACE-hard; in [JR] it is proved that the third problem is EXPSPACE-hard. Thus, we can now conclude that these problems are *complete* in these classes.

To prove that these problems are in PSPACE (or in EXPSPACE) we reduce them (in three steps) to problems which are known to belong to PSPACE. We use many-to-one polynomial-space reductions for the first two problems, and many-to-one exponential-space reductions for the third problem. It is well known that the composition of two many-to-one polynomial-space (resp. exponential-space) reductions yields again a many-to-one polynomial-space (resp. exponential-space) reduction. Moreover, if a language L reduces to a language M by a many-to-one polynomial-space reduction, and if M is in PSPACE, then L is also in PSPACE (see [HU]); similarly, if a language L reduces to a language M by a many-to-one exponential-space reduction, and if M is in PSPACE or in EXPSPACE, then L is in EXPSPACE.

Theorem 6.1. *The equivalence problem for* **2AFAs** *is in* PSPACE.

Proof. Three successive many-to-one polynomial-space reductions are given in Claims A–C; in Claim D the last problem is shown to be in PSPACE.

Claim A. *The equivalence problem for n-state 2AFAs reduces to the equivalence problem for $O(n)$-state deterministic Hennie machines, in which no state occurs more than $O(n)$ times at the same position, and which always halt.*

Proof of Claim A. By Theorem 4.1(a), for every n-state 2AFA A we can construct an equivalent $O(n)$-state deterministic Hennie machine H in which no state occurs more than $O(n)$ times at the same position, and (by Fact 4.2) we can assume that this machine always halts. This reduces the equivalence problem for 2AFAs to the equivalence problem for Hennie machines of the above type. We have to check that the construction of H, given A, can be done using polynomial work-space.

In the first step of the proof of Theorem 4.1(a), A is replaced by a monotone 2AFA with $2n$ states; this step can be done in linear time. We assume now that A is monotone.

Let Σ be the alphabet of A, which is thus also the input alphabet of H. The work alphabet of H consists of the circuit-slices corresponding to Σ (so $|\Sigma|$ such slices are used), in which some edges may be colored or erased. Since a circuit-slice has $\leq n^2$ edges, this leads to $\leq c^{n^2}$ possible modified slices, for some constant $c \geq 2$; thus the work alphabet of H has size $\leq |\Sigma| c^{n^2}$. Each letter of H can thus be represented in space $O(n^2 + \log |\Sigma|)$. We can compute the (exponentially large) transition table of H, using only polynomial work-space, as follows: We consider all pairs (state, letter) of H, in lexical order; for each such pair, we can compute the next state and the letter printed and the head-direction, in polynomial time (as in the proof of Theorem 4.1(a)).

We then output the state, letter, next state, letter printed, and direction. Since we reuse the work-space to do this calculation for each pair, only polynomial work-space is used (although the time is exponential).

Finally, the construction in Fact 4.2, which makes the Hennie machine halt, takes only polynomial time (in particular, the work alphabet of the Hennie machine only grows polynomially in size). □

Claim B. *The equivalence problem for m-state deterministic Hennie machines in which no state occurs more than k_1 times at the same position and which always halt, reduces to the* emptiness problem *for Hennie machines of this same type (with m and k_1 now replaced by cm and ck_1 for some constant $c \geq 1$). (In the emptiness problem, the question is whether the language accepted by the given machine is empty.)*

Proof of Claim B. Given two such Hennie machines H_1 and H_2, a new Hennie machine H of the same type (except that m and k_1 are now replaced by cm and ck_1), recognizing the *symmetric difference* of the two languages accepted by H_1 and H_2, can be constructed. H_1 and H_2 (which always halt) are simply executed one after the other; also, by the halting property, the new machine H can check whether H_1 and H_2 accept or reject the input. □

Claim C. *The emptiness problem for m-state deterministic Hennie machines in which no state occurs more than k_1 times at the same position and which always halt, reduces to the emptiness problem for mk_1-state 2DFAs.*

Proof of Claim C. By Theorem 3.1(b), given a Hennie machine H of the above type, we can construct an l.p.hom. h and a 2DFA D_2 (with mk_1 states) such that $L_H = h(L_D)$; here L_H and L_D are the languages accepted by H and D_2, respectively. Thus, L_H is empty iff L_D is empty.

We must still show that the construction of D_2, given H, only takes polynomial work-space. Let Γ be the total alphabet of H. Then the alphabet of D_2 has size $\leq (|\Gamma| \cdot m \cdot mk_1)^k$, where $k = mk_1$ (see the proof of Theorem 3.1(b)); each letter can be represented using space $cmk_1(\log|\Gamma| + \log mk_1)$. We compute the transition table of D_2, using only polynomial work-space, as follows: We consider all pairs (state, letter) of D_2, in lexical order; for each such pair, we can compute the next state and the head-direction, in polynomial time (as in the proof of Theorem 3.1(b)); we then output the state, letter, next state, and direction. Since we reuse the work-space to do this calculation for each pair, only polynomial work-space is used (although the time is exponential). □

Claim D [Hu]. *The emptiness problem for 2DFAs is in* PSPACE.

Proof of Claim D. To make the paper more self-contained we give a proof here. It is sufficient to show that the nonemptiness problem for 2DFAs is decided by a nondeterministic polynomial-space algorithm (since PSPACE is closed under complement and does not change when nondeterminism is used, by Savitch's theorem).

Let $L \subseteq \Sigma^*$ be the language accepted by a 2DFA $D = (Q, \Sigma, q_0, \delta, F)$ with n states. Then L is nonempty iff $w \in \Sigma^*$ exists such that w is accepted by D. Moreover, w is accepted by D iff $(q_0)[\rightarrow\ \langle w\rangle\ \rightarrow] \in F$ (where $[\rightarrow\ \langle w\rangle\ \rightarrow]$ is the one of the Shepherdson tables defined at the end of the proof of Theorem 4.1(c)). Let $[u]$ denote the quadruple of the four Shepherdson tables of u (see the end of the proof of Theorem 4.1(c)). To check nondeterministically whether L is empty, we guess (in polynomial space) a quadruple $[\langle w\rangle]$ and then check (in polynomial time) if $(q_0)[\rightarrow\ \langle w\rangle\ \rightarrow] \in F$. To guess $[\langle w\rangle]$ we proceed as follows: We compute $[\langle]$ from the transition table of D (in polynomial time), then we successively guess more letters; suppose $[\langle a_1 \cdots a_i]$ has been obtained and remembered so far; we guess a letter a_{i+1} and we compute $[\langle a_1 \cdots a_i a_{i+1}] = [\langle a_1 \cdots a_i][a_{i+1}]$ (according to the fact at the end of the proof of Theorem 4.1(c)), and then erase $[\langle a_1 \cdots a_i]$ and a_{i+1}. For each letter guessed we reuse the same polynomial space (although the number of letters guessed is not necessarily polynomial). Finally, we guess that the guessing of $[\langle w]$ is complete; we then compute $[\langle w\rangle] = [\langle w]\rangle]$. □

This completes the proof that the equivalence problem for 2AFAs is in PSPACE. □

Theorem 6.2. *The equivalence problem for* **1-pebble 2NFAs** *is in* PSPACE.

Proof. We reduce the problem (in three reductions) to a problem which is in PSPACE. Only the first step (Claim A′) is different from the proof of Theorem 6.1; the rest is identical.

Claim A′. *The equivalence problem for n-state 1-pebble 2NFAs reduces to the equivalence problem for $O(n^2)$-state* deterministic *Hennie machines in which $O(n)$ states occur $O(n)$ times at the same position (whereas each of the remaining $O(n^2)$ states occurs at most once at the same position), and which always halt.*

Proof of Claim A′. By Theorem 4.1(c), for every n-state 1-pebble 2NFA P we can construct an equivalent $O(n^2)$-state deterministic Hennie machine H in which $O(n)$ states occur $O(n)$ times at the same position (whereas the remaining $O(n^2)$ states occur at most once at the same position). By Fact 4.2 we can assume that this machine always halts. This reduces the equivalence problem for 1-pebble 2NFAs to the equivalence problem for Hennie machines of the above type. We still have to check that the construction of H, given P, can be done using polynomial work-space.

Let Σ be the alphabet of P, which is thus also the input alphabet of H. The work alphabet of H consists of:

(1) The alphabet Δ of the 2DFA N_2. These are letters of Σ that come together with pairs of Shepherdson tables, and incompletely constructed pairs of Shepherdson tables (see the first two passes in the proof of Theorem 4.1(c)). There are $|\Delta| \leq |\Sigma| b^{n^2}$ such letters, where b is a constant ≥ 2.

(2) The circuit-slices (with coloring and erasing of some edges) associated with the alphabet Δ of N_2 (when N_2 is being simulated by a deterministic Hennie machine according to Theorem 4.1(a)). There are $\leq |\Delta| d^{n^2} = |\Sigma| c^{n^2}$ such slices, where d and c are constants ≥ 2.

Thus the work alphabet of H has size $\leq |\Sigma| c^{n^2}$. Each letter of H can thus be represented in space $O(n^2 + \log|\Sigma|)$. We can compute the (exponentially large) transition table of H, using only polynomial work-space, as follows: We consider all pairs (state, letter) of H, in lexical order; for each such pair, we can compute the next state and the letter printed and the head-direction, in polynomial time (as in the proof of Theorems 4.1(a) and (c)). We then output the state, letter, next state, letter printed, and direction. Since we reuse the work-space to do this calculation for each pair, only polynomial work-space is used (although the time is exponential).

Finally, the construction in Fact 4.2, which makes the Hennie machine halt, takes only polynomial time (in particular, the work alphabet of the Hennie machine only grows polynomially in size). □

The remainder of the proof is identical to the proof of Theorem 6.1. □

Theorem 6.3. *The equivalence problem for* **1-pebble 2AFAs** *is in* EXPSPACE.

Proof. We reduce the problem (in three reductions) to a problem which is in EXPSPACE. Only the first step (Claim A″) is different from the proof of Theorems 6.1 and 6.2; the rest is identical.

Claim A″. *The equivalence problem for n-state 1-pebble 2AFAs reduces (via a many-to-one exponential-space reduction) to the equivalence problem for $O(n2^n)$-state* deterministic *Hennie machines in which $O(n)$ states occur $O(n)$ times at the same position (whereas each of the remaining $O(n2^n)$ states occurs at most once at the same position), and which always halt.*

Proof of Claim A″. By Theorem 4.1(d), for every n-state 1-pebble 2AFA P we can construct an equivalent $O(n2^n)$-state deterministic Hennie machine H in which $O(n)$ states occur $O(n)$ times at the same position (whereas the remaining $O(n2^n)$ states occur at most once at the same position). By Fact 4.2 we can assume that this machine always halts. This reduces the equivalence problem for 1-pebble 2AFAs to the equivalence problem for Hennie machines of the above type. We have to check that the construction of H, given P, can be done using exponential work-space.

Let Σ be the alphabet of P, which is thus also the input alphabet of H. The work alphabet of H consists of:

(1) The alphabet Δ of the 2AFA A_2. These are letters of Σ that come together with pairs of Shepherdson tables (with respect to 2AFAs), and incompletely constructed pairs of Shepherdson tables (see the first two passes in the proof of Theorem 4.1(d)). There are $|\Delta| \leq |\Sigma| b^{n2^n}$ such letters, for some constant $b \geq 2$.

(2) The circuit-slices (with coloring and erasing of some edges) associated with the alphabet Δ of N_2 (when N_2 is being simulated by a deterministic Hennie machine according to Theorem 4.1(a)). There are $\leq |\Delta| d^{n^2} = |\Sigma| c^{n2^n}$ such slices, where d and c are constants ≥ 2.

Thus the work alphabet of H has size $\leq |\Sigma| c^{n2^n}$. Each letter of H can thus be represented in space $O(n2^n + \log|\Sigma|)$. We can compute the (doubly exponentially large) transition table of H, using only exponential work-space, as follows: We consider all pairs (state, letter) of H, in lexical order; for each such pair, we can compute the next state and the letter printed and the head-direction, in polynomial time (as in the proof of Theorems 4.1(a) and (d)). We then output the state, letter, next state, letter printed, and direction. Since we reuse the work-space to do this calculation for each pair, only exponential work-space is used (although the time is doubly exponential).

Finally, the construction in Fact 4.2, which makes the Hennie machine halt, takes only polynomial time (in particular, the work alphabet of the Hennie machine only grows polynomially in size in this step). □

Claims B–D of the proof of Theorem 6.1 are now used without any change. So our problem reduces to a PSPACE-complete problem via exponential-space reductions and thus can be solved in exponential space. □

Acknowledgments

In addition to the motivation of the Sakoda–Sipser conjecture, this paper was inspired by a discussion about Exercise 3.19 of [HU] that I had with Douglas Albert in Berkeley many years ago. I would also like to thank the anonymous referees for their useful reports; in particular, the characterization of NP (Theorem 2.2) was suggested by one of the referees as an extension of Theorem 2.1.

References

[A] T. Apostol, *Introduction to Analytic Number Theory*, Springer-Verlag, New York, 1976.

[AHU] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[B1] J.-C. Birget, Concatenation of inputs in a two-way automaton, *Theoretical Computer Science* **63** (1989), 141–156.

[B2] J.-C. Birget, Basic techniques for two-way finite automata, in: J. E. Pin (ed.), *Formal Properties of Finite Automata and Applications*, Lecture Notes in Computer Science, Vol. 386, Springer-Verlag, Berlin, 1989, pp. 56–64.

[B3] J.-C. Birget, Positional simulation of two-way automata: proof of a conjecture of R. Kannan, and generalizations, *Journal of Computer and System Sciences* **45** (1992), 154–179 (special issue on *STOC* 89).

[B4] J.-C. Birget, State-complexity of finite-state devices, state-compressibility and incompressibility, *Mathematical Systems Theory* **26** (1993), 237–269.

[B5] J.-C. Birget, Partial orders on words, minimal elements in regular languages, and state complexity, *Theoretical Computer Science*, **119** (1993), 267–291.

[B6] J.-C. Birget, The minimum automaton for certain languages, in preparation.

[BG] R. Book and S. Greibach, Quasi-realtime languages, *Mathematical Systems Theory* **4** (1970), 97–111.

[BH] M. Blum and C. Hewitt, Automata on a 2-dimensional tape, *Proceedings of the 8th IEEE Symposium on Switching and Automata Theory*, 1965, pp. 155–160.

[BCST] D. A. M. Barrington, K. Compton, H. Straubing, and D. Thérien, Regular languages in NC^1, *Journal of Computer and System Sciences*, to appear.

[BL] J. Brzozowski and E. Leiss, On equations for regular languages, finite automata, and sequential networks, *Theoretical Computer Science* **10** (1980), 19–35.

[BT] D. A. M. Barrington and D. Thérien, Finite monoids and the fine structure of NC^1, *Journal of the Association for Computing Machinery* **35** (1988), 941–952.

[Ch] M. Chrobak, Finite automata and unary languages, *Theoretical Computer Science* **47** (1986), 149–158.

[CKS] A. Chandra, D. Kozen, and L. Stockmeyer, Alternation, *Journal of the Association for Computing Machinery* **28** (1981), 114–133.

[Co] S. Cook, A taxonomy of problems with fast parallel algorithms, *Information and Control* **64** (1985), 2–22.

[CS] A. Chandra and L. Stockmeyer, Alternation, *Proceedings of the 17th IEEE Annual Symposium on Foundations of Computer Science*, 1976, pp. 98–108.

[DDQ] P. Denning, J. Dennis, and J. Qualitz, *Machines, Languages, and Computation*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[GGK] P. Goralčík, A. Goralčíková , and V. Koubek, Alternation with a pebble, *Information Processing Letters* **38** (1991), 7–13.

[He] F. C. Hennie, One-tape off-line Turing machine computations, *Information and Control* **8** (1965), 553–578.

[Hu] H. B. Hunt III, On the time and tape complexity of languages, I, *Proceedings of the 5th Annual ACM Symposium on Theory of Computing*, 1973, pp. 10–19.

[HU] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, MA, 1979.

[IR] O. Ibarra and B. Ravikumar: Sublogarithmic-space Turing machines, nonuniform space complexity, and closure properties, *Mathematical Systems Theory* **21** (1988), 1–17.

[IS] O. Ibarra and S. Sahni, Hierarchies of Turing machines with restricted tape alphabet size, *Journal of Computer and System Sciences* **11** (1975), 56–67.

[JR] T. Jiang and B. Ravikumar, A Note on the Space Complexity of Some Decision Problems for Finite Automata, *Information Processing Letters* **40** (1991), 25–31.

[Ka] R. Kannan, Alternation and the power of non-determinism, *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, 1983, pp. 344–346.

[Ko] D. Kozen, On parallelism in Turing machines, *Proceedings of the 17th IEEE Annual Symposium on Foundations of Computer Science*, 1976, pp. 89–97.

[L1] E. Leiss, Succinct representation of regular languages by boolean automata, *Theoretical Computer Science* **13** (1981), 323–330.

[L2] E. Leiss, Succinct representation of regular languages by boolean automata, II, *Theoretical Computer Science* **38** (1985), 133–136.

[LLS] R. Ladner, R. Lipton, and L. Stockmeyer, Alternating pushdown automata, *Proceedings of the 19th IEEE Symposium on Foundations of Computer Science*, 1978, pp. 92–106, and *SIAM Journal of Computing*, **13**(1) (1984), 135–155.

[LSH] P. M. Lewis, R. Stearns, and J. Hartmanis, Memory bounds for recognition of context-free and context-sensitive languages, *Proceedings of the 6th IEEE Symposium on Switching Circuit Theory and Logical Design*, 1965, pp. 191–202.

[M] P. Michel, An NP-complete language accepted in linear time by a one-tape Turing machine, *Theoretical Computer Science* **85** (1991), 205–212.

[MF] A. R. Meyer and M. J. Fischer, Economy of description by automata, grammars, and formal systems, *Proceedings of the 21st IEEE Symposium on Switching and Automata Theory*, 1971, pp. 188–191.

[MP] D. E. Muller and F. P. Preparata, Bounds to complexities of networks for sorting and switching, *Journal of the Association for Computing Machinery* **22** (1975), 97–111.

[R1] W. L. Ruzzo, Tree-size bounded alternation, *Journal of Computer and System Sciences* **21** (1980), 218–235.

[R2] W. L. Ruzzo, On uniform circuit complexity, *Journal of Computer and System Sciences* **22** (1981), 365–383.

[Sa] John E. Savage, *The Complexity of Computing*, Wiley, New York, 1976, reprint by Krieger, 1987.

[Sh] J. C. Shepherdson, The reduction of two-way automata to one-way automata, *IBM Journal of Research and Development* (1959), 198–200; also in E. F. Moore (ed.), *Sequential Machines: Selected Papers*, Addison-Wesley, Reading, MA, 1964.

[Si1] M. Sipser, Lower bounds on the size of sweeping machines, *Proceedings of the 11th ACM Symposium on Theory of Computing*, 1979, pp. 360–364; expanded in *Journal of Computer and System Sciences* **21** (1980), 195–202.

[Si2] M. Sipser, Halting space-bounded computations, *Theoretical Computer Science* **10** (1980), 335–338. (See also *Proc. IEEE FOCS*, 1978.)

[Sp] F. N. Springsteel, "On the pre-AFL of [log n] space, and related families of languages, *Theoretical Computer Science* **2** (1976), 295–304.

[SS] W. Sakoda and M. Sipser, Non-determinism and the size of two-way automata, *Proceedings of the 10th ACM Symposium on Theory of Computing*, 1978, pp. 275–286.

[vL] J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Vol. A, MIT Press, Cambridge, MA, and Elsevier, Amsterdam, 1990.

[W] I. Wegener, *The Complexity of Boolean Functions*, Wiley–Teubner Series in Computer Science, Wiley, New York, 1987.

Received December 24, 1990, *and in revised form August* 15, 1990, *and in final form January* 20, 1994.