INFORMATION AND COMPUTATION 103, 299–347 (1993)

Inference of Finite Automata Using Homing Sequences*

RONALD L. RIVEST AND ROBERT E. SCHAPIRE†

MIT Laboratory for Computer Science, Cambridge, Massachusetts 02139

We present new algorithms for inferring an unknown finite-state automaton from its input/output behavior, even in the absence of a means of resetting the machine to a start state. A key technique used is inference of a *homing sequence* for the unknown automaton. Our inference procedures experiment with the unknown machine, and from time to time require a teacher to supply counterexamples to incorrect conjectures about the structure of the unknown automaton. In this setting, we describe a learning algorithm that, with probability $1-\delta$, outputs a correct description of the unknown machine in time polynomial in the automaton's size, the length of the longest counterexample, and $\log(1/\delta)$. We present an analogous algorithm that makes use of a diversity-based representation of the finite-state system. Our algorithms are the first which are provably effective for these problems, in the absence of a "reset." We also present probabilistic algorithms for permutation automata which do not require a teacher to supply counterexamples. For inferring a permutation automaton of diversity D, we improve the best previous time bound by roughly a factor of $D^3/\log D$.

1. INTRODUCTION

Imagine a simple, autonomous robot placed in an unfamiliar environment. Typically, such a robot would be equipped with some sensors (a camera, sonar, a microphone, etc.) that provide the robot limited information about the state of its environment. Being autonomous, the robot would also have some simple actions that it has the option of executing (step ahead, turn left, lift arm, etc.).

For instance, the robot might be in the simple toy environment of Fig. 1. In this environment, the robot can sense its local environment (whether the "room" it occupies is shaded or not), and can traverse one of the out-going edges by executing action "x" or action "y."

* This paper is a significantly revised version of a preliminary report which appeared in the "Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing," 1989.

† Current address: AT&T Bell Laboratories, 600 Mountain Avenue, Room 2A-424, Murray Hill, New Jersey 07974.

0890-5401/93 $5.00

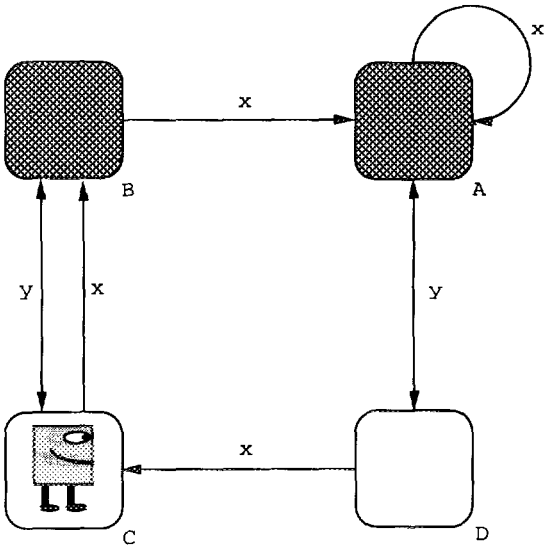

FIG. 1. An example robot environment.

A priori, the robot may not be aware of the "meaning" of its actions, nor of the sense data it is receiving. It may also have little or no knowledge beforehand about the "structure" of its environment.

This problem motivates the research presented in this paper: How can the robot infer on its own from experience a good model of its world? Specifically, such a model should explain and predict how the robot's actions affect the sense data received.

Certainly, once such a model has been inferred, the robot can function more effectively in the learned environment. However, programming the robot with a complete model of a fairly complex environment would be prohibitively difficult; what is more, even if feasible, a robot with a pre-programmed world model is entirely lacking in flexibility and would likely have a hard time coping in environments other than the one for which it was programmed. Thus, the development of effective learning methods would both simplify the job of the programmer, and make for a more versatile robot.

This problem of learning about a new environment from experience has been addressed by a number of researchers using a variety of approaches: Drescher (1986) explores learning in quite rich environments using an approach based on Piaget's theories of early childhood development. Wilson (1985) studies so-called genetic algorithms for learning by "animats" in unfamiliar environments. Kuipers and Byun (1988) advocate a "qualitative" approach to the related problem of learning a map of a

mobile robot's environment. The map-learning problem is also studied by Mataric (1990).

In this paper, we take an initial step toward a general, algorithmic solution to the robot's learning problem. Specifically, we give a thorough treatment to the problem of inferring the structure of an environment that is known *a priori* to be *deterministic* and *finite state*. Such an environment can be naturally modeled as a deterministic finite-state automaton: the robot's actions then are the inputs to the automaton, and the automaton's output is just the sense data the robot receives from the environment. Our goal then is to infer the unknown automaton by observing its input–output behavior.

This problem has been well studied by the theoretical community, and it continues to generate new interest. (See Pitt's paper (1989) for an excellent survey.) Virtually all previous research, however, has assumed that the learner has a means of "resetting" the automaton to some start state. Such an assumption is quite unnatural, given our motivation; as in real life, we expect the robot to learn about its environment in one continuous experiment. The main result of this paper is the first set of provably effective algorithms for inferring finite-state automata in the absence of a reset.

Here is a brief history of some of the previous, theoretical work on inference of automata. The most important lesson of this research has been that a combination of active experimentation and passive observation is both necessary and sufficient to learn an unknown automaton.

First, concerning inference of an unknown automaton from passively received data, Angluin (1978) and Gold (1978) show that it is NP-complete to find the smallest automaton consistent with a given sample of input–output pairs. Pitt and Warmuth (1993) show that merely finding an approximate solution is intractable (assuming $P \neq NP$). In Valiant's (1984) so-called "probably approximately correct" model, Kearns and Valiant (1989) consider the problem of predicting the output of the automaton on a randomly chosen input, based on a random sample of the machine's behavior. Extending the work of Pitt and Warmuth (1990), they show that this problem is intractable, assuming the security of various cryptographic schemes. Thus, learning by passively observing the behavior of the unknown machine is apparently infeasible.

What about learning by actively *experimenting* with it? Angluin (1981) shows that this problem is also hard. She describes a family of automata that cannot be identified in less than exponential time when the learner can only observe the behavior of the machine on inputs of the learner's own choosing. The difficulty here is in accessing certain hard-to-reach states.

In spite of these negative results, Angluin (1987), elaborating on Gold's results (1972), shows that a *combination* of active and passive learning

is feasible. Her inference procedure is able to actively experiment with the unknown automaton, and in addition, in response to each incorrect conjecture of the automaton's identity, her algorithm passively receives a counterexample, a string that is misclassified by the conjectured automaton. (Such counterexamples are considered passive data because they are not freely chosen by the learner.) Angluin's algorithm exactly identifies the unknown automaton in time polynomial in the automaton's size and the length of the longest counterexample.

As mentioned above, a serious limitation of Angluin's procedure is its critical dependence on a means of *resetting* the automaton to a fixed start state. Thus, the learner can never really "get lost" or lose track of its current state since it can always reset the machine to its start state. In this paper, we extend Angluin's algorithm, demonstrating that an unknown automaton can be inferred even when the learner is not provided with a reset.

This paper also includes an improved version of Angluin's algorithm in the case that a reset *is* available; this improved algorithm significantly reduces the number of experiments that must be performed by the learner.

The generality of our results allows us to handle any "directed-graph environment," such as the one in Fig. 1. This means that we can handle many special cases as well, such as undirected graphs, planar graphs, and environments with special spatial relations. However, our procedures do not take advantage of such special properties of these environments, some of which could probably be handled more effectively. For example, we have found that permutation automata are generally easier to handle than non-permutation automata.

In previous papers (Rivest and Schapire, 1987; Rivest and Schapire, 1990; Schapire, 1988), we introduced the "diversity-based" representation of finite automata, and we argued that, in some situations, this "egocentric" representation is far more natural and compact than the usual state-based representation. We also described an algorithm that was proved to be effective for permutation automata, even in the absence of a reset. Some general techniques for handling non-permutation automata were also discussed; although not provably effective, these seemed to work well in practice for a variety of simple environments.

In this paper, we generalize these results, demonstrating probabilistic inference procedures that are provably effective for both permutation and non-permutation automata. More generally, we present new inference procedures for the usual global state representation, as well as for the diversity-based representation.

Like Angluin, we assume that the inference procedures have an unspecified source of counterexamples to incorrectly conjectured models of the automaton. This differs from our previous work where the learning

model incorporated no such source of counterexamples; as already mentioned, this limitation makes learning of finite automata infeasible in the general case. For a robot trying to infer the structure of its environment, a counterexample is discovered whenever the robot's current model makes an incorrect prediction. For the special class of permutation automata, we show that an artificial source of counterexamples is unnecessary.

Our algorithms use powerful new techniques based on the inference of *homing sequences*. Informally, a homing sequence is a sequence of inputs that, when fed to the machine, is guaranteed to "orient" the learner: the outputs produced in executing the homing sequence completely determine the state reached by the automaton at the end of the homing sequence. (This should not be confused with a *distinguishing sequence*, described later, whose output determines the *starting* state from which the sequence was executed.) Every finite-state machine has a homing sequence. For each inference problem, we show how a homing sequence can be used to infer the unknown machine, and how a homing sequence can be inferred as part of the overall inference procedure.

In sum, the main results of this paper are four-fold: We describe efficient algorithms for inference of general finite automata using both the state-based and the diversity-based representations; both of these algorithms require a means of experimenting with the automaton and a source of counterexamples. Then, for permutation automata, we give efficient algorithms for both representations that do not require an external source of counterexamples. The time of the diversity-based algorithm for permutation automata beats the best previous bound by roughly a factor of $D^3/\log D$, where D is the size of the automaton using the diversity-based representation. In the other three cases, our procedures are the first provably effective polynomial-time algorithms.

2. Two Representations of Finite Automata

2.1. *The Global State-Space or Standard Representation*

An *environment* or *finite-state automaton* $\mathcal{E}$ is a tuple $(Q, B, \delta, q_0, \gamma)$ where:

- Q is a finite nonempty set of *states*,
- B is a finite nonempty set of *input symbols* or *basic actions*,
- δ is the *next-state* or *transition function*, which maps $Q \times B$ into Q,
- q_0, a member of Q, is the *initial state*, and
- γ is the *output function*, which maps Q into $\{0, 1\}$.

This is the standard, or state-based, representation. Note that, although we prove all our results for two-value output functions γ, the generalization of these results to multiple-value output functions is straightforward.

For example, the graph of Fig. 1 depicts the global state representation of an automaton whose states are the vertices of the graph (A, B, C, D), whose transition function is given by the edges, and whose output function is given by the shading of the vertices (i.e., $\gamma(q) = 1$ if and only if q is shaded).

We denote the set of all finitely long action sequences by $A = B^*$, and we extend the domain of the function $\delta(q, \cdot)$ to A in the usual way: $\delta(q, \lambda) = q$, and $\delta(q, ab) = \delta(\delta(q, a), b)$ for all $q \in Q$, $a \in A$, $b \in B$. Here, λ denotes the empty or null string. Thus, $\delta(q, a)$ denotes the state reached by executing sequence a from state q; for short, we often write qa to denote this state.

We say that $\mathcal{E}$ is a *permutation automaton* if for every action b, the function $\delta(\cdot, b)$ is a permutation of Q.

We refer to the sequence of outputs produced by executing a sequence of actions $a = b_1 b_2 \cdots b_r$ from a state q as the *output of a at q*, denoted $q\langle a \rangle$:

$$q\langle a \rangle = \langle \gamma(q), \gamma(qb_1), \gamma(qb_1b_2), \ldots, \gamma(qb_1b_2 \cdots b_r) \rangle.$$

For instance, if the robot in Fig. 1 executes action $a = \mathrm{xy}$ from its current state $q = \mathrm{C}$, then it will observe the sequence of outputs

$$q\langle a \rangle = \mathrm{C}\langle \mathrm{xy} \rangle = 010.$$

(Do not confuse $\gamma(qa)$ and $q\langle a \rangle$. The former is a single value, the output of the state reached by executing a from q; for instance, $\gamma(qa) = 0$ in the example above. In contrast, $q\langle a \rangle$ is a $(|a| + 1)$-tuple consisting of the sequence of outputs produced by executing a from state q.)

Finally, for $a \in A$, we denote by $Q\langle a \rangle$ the set of possible outputs on input a:

$$Q\langle a \rangle = \{q\langle a \rangle : q \in Q\}.$$

Clearly, $|Q\langle a \rangle| \leq |Q|$ for any a.

Action sequence a is said to *distinguish* two states q_1 and q_2 if $q_1\langle a \rangle \neq q_2\langle a \rangle$. For instance xy distinguishes states C and D of the environment of Fig. 1, but not states A and B. We assume that $\mathcal{E}$ is *reduced* in the sense that, for every pair of distinct states, there is some action sequence that distinguishes them.

2.2. *The Diversity-Based Representation*

In this section, we describe the second of our representations. Our interest in this representation derives from the fact that, for a variety of environments, the diversity-based representation is far more compact and

natural than the usual state-based representation. (An example is given below.) See our previous papers (Rivest and Schapire, 1987; Rivest and Schapire, 1990; Schapire, 1988) for further background and detail.

The diversity-based representation is based on the notion of *tests* and *test equivalence*. A *test* is an action sequence. (This definition differs slightly from that given in previous papers where the automata considered had multiple outputs (or "sensations") at each state.) The *value* of a test t at state q is $\gamma(qt)$, the output of the state reached by executing t from q.

Two tests t_1 and t_2 are *equivalent*, written $t_1 \equiv t_2$, if the tests have the same value at every state. For instance, in the environment of Fig. 1, tests `yxx` and `xx` are equivalent since the value of each is 1 in every state; similarly, tests `yy` and λ are equivalent since each has value 1 in states `A` and `B`, and 0 in states `C` and `D`.

It is easy to verify that "$\equiv$" defines an equivalence relation on the set of tests. We write $[t]$ to denote the *equivalence class* of t, the set of tests equivalent to t. The value of $[t]$ at q is well defined as $\gamma(qt)$. The *diversity* of the environment, $D(\mathcal{E})$, is the number of equivalence classes of the automaton: $D(\mathcal{E}) = |\{[t]: t \in A\}|$. We will see that $D(\mathcal{E})$ is a natural measure of the size of the diversity-based representation. It can be shown that $\lg(|Q|) \leq D(\mathcal{E}) \leq 2^{|Q|}$, so the diversity of a finite automaton is always finite (Rivest and Schapire, 1987; Schapire, 1988).

As described in our previous papers, there are natural environments with small diversity but an enormous number of states. For example, this is the case in the "Register World." In this environment, the robot observes the leftmost bit of an r-bit register. The robot's actions allow it to flip the value of this bit, or to rotate the contents of the entire register left or right (with wrap-around). It can be shown that the diversity of this environment is only $2r$: it turns out that for each of the register's bits, this environment has one test equivalence class corresponding to the bit, and one corresponding to the complement of the bit. In contrast, the usual global-state representation for this environment is enormous—there is one state for each of the 2^r possible configurations of the register's r bits.

The equivalence classes can be viewed as *state variables* whose values entirely describe the state of the environment. This is true because two states are equal (in a reduced sense) if and only if every test has the same value in both states.

It is often convenient to arrange the equivalence classes in an *update graph* such as the one in Fig. 2 for the environment of Fig. 1. Each vertex in the graph is an equivalence class so the size of the graph is $D(\mathcal{E})$. An edge labeled $b \in B$ is directed from vertex $[t_1]$ to $[t_2]$ if and only if $t_1 \equiv bt_2$. These edges are well defined since if $t \equiv t'$ then $bt \equiv bt'$. Note that, by transitivity of the equivalence relation, each vertex has exactly one in-going edge labeled with each of the basic actions.

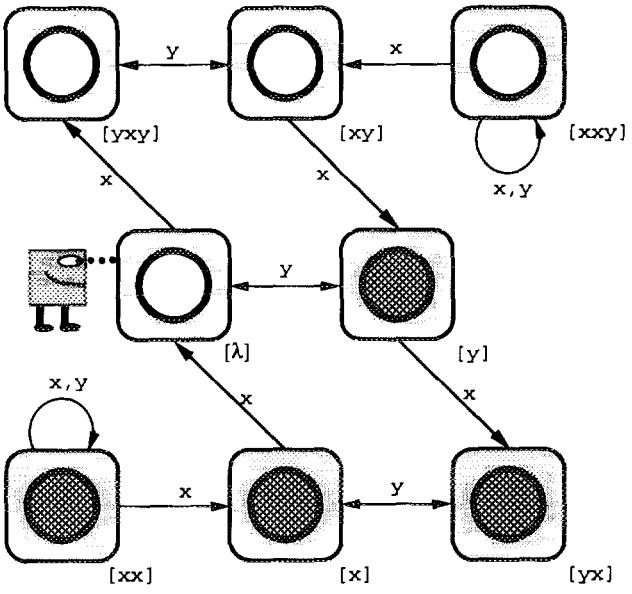

FIG. 2. The update graph for the environment of Fig. 1.

We associate with each vertex $[t]$ the value of t in the current state q. In the figure, we have used shading to indicate the value of each vertex in the robot's current state. The output of the current state is given by vertex $[\lambda]$, so this is the only vertex whose value can be directly observed. When an action b is executed from q, the value of each vertex $[t]$ is replaced by the old value of $[bt]$, the vertex at the tail of $[t]$'s (unique) in-going b-edge. That is, in the new state qb, equivalence class $[t]$ takes on the old value of $[bt]$ in the starting state q. This follows from the fact that $\gamma((qb)t) = \gamma(q(bt))$. For instance, if action y is executed in the environment of Figs. 1 and 2 from the current state C, then the value of $[\lambda]$ in the new state is 1, the old value of [y]; the new value of [yxy] is 0, the old value of [xy]. One can verify from Fig. 1 that these are indeed the correct values of these tests in the new state Cy = B.

Thus, the value of each equivalence class in the state reached by executing any action can be determined easily using the update graph. Thus, the update graph can be used to simulate the environment.

2.2.1. *Simple-Assignment Automata*

The update graph can be viewed more abstractly as a special kind of automaton: A *simple-assignment automaton* $\mathcal{S}$ is a tuple (V, B, Y, v_0, ω), where:

- V is a finite nonempty set of *variables*,
- B is a finite nonempty set of *input symbols* or *basic actions*,
- Υ is the *update function*, which maps $V \times B$ into V,
- v_0, a member of V, is the *output variable*, and
- ω is the *initial-value function* which maps V into $\{0, 1\}$.

Here, we interpret V as a vector of state variables whose values determine the state of $\mathcal{S}$. The initial values of these variables are given by ω, and the output of the machine is the current value of the special variable v_0. When an action $b \in B$ is executed, each variable v is updated in the new state with the old value of variable $\Upsilon(v, b)$.

The function Υ can be extended to the domain $V \times A$ by defining $\Upsilon(v, \lambda) = v$ and $\Upsilon(v, ba) = \Upsilon(\Upsilon(v, a), b)$ for $v \in V$, $a \in A$ and $b \in B$. Note that, unlike the more usual extension of this kind, Υ processes action sequences from right to left, rather than from left to right. We define Υ in this fashion to give it the desired property that when action sequence a is executed, variable v is updated with the old value of variable $\Upsilon(v, a)$. In particular, this means that the output of $\mathcal{S}$ after executing $a \in A$ from the initial state is $\omega(\Upsilon(v_0, a))$.

Thus, the update graph is itself a simple-assignment automaton. In this case, the set V is the set of equivalence classes $\{[t]: t \in A\}$; the update function is defined by the rule $\Upsilon([t], b) = [bt]$; the output variable is $v_0 = [\lambda]$; and $\omega([t])$ is the value of $[t]$ in the initial state, $\gamma(q_0 t)$. With these definitions, it is straightforward to verify that $\Upsilon(v_0, t) = [t]$, and so $\omega(\Upsilon(v_0, t)) = \gamma(\delta(q_0, t))$ for all $t \in A$.

On first blush, the structures of simple-assignment automata (such as the update graph of Fig. 2) and of ordinary finite-state automata (such as the one given by the transition diagram of Fig. 1) appear to be quite similar. In fact, their interpretations are very different. In the global-state representation, the robot moves from state to state while the output values of the states remain unchanged. On the other hand, in the diversity-based (or simple-assignment) representation, the robot remains stationary, only observing the output of a single variable ($[\lambda]$), and causing with its actions the values of the variables to move around. Thus, the diversity-based representation is more egocentric—the world is represented *relative* to the robot. In contrast, in the state-based representation, the world is represented by its *global* structure.

3. Homing Sequences

Henceforth, we set $D = D(\mathcal{S})$, $n = |Q|$, $k = |B|$.

A *homing sequence* is an action sequence h for which the state reached by

executing h is uniquely determined by the output produced: thus, h is a homing sequence if and only if

$$(\forall q_1 \in Q)(\forall q_2 \in Q)\, q_1\langle h\rangle = q_2\langle h\rangle \Rightarrow q_1 h = q_2 h.$$

For example, the string consisting of the single action "x" is a homing sequence for the environment of Fig. 1. If $q\langle \mathtt{x}\rangle = 00$, then $q\mathtt{x} = \mathtt{C}$; if $q\langle \mathtt{x}\rangle = 01$, then $q\mathtt{x} = \mathtt{B}$; and, if $q\langle \mathtt{x}\rangle = 11$ then $q\mathtt{x} = \mathtt{A}$.

It is easy at first to confuse a homing sequence with a *distinguishing sequence*, a sequence a whose output uniquely determines the *starting* state from which the sequence was executed (i.e., if $q_1\langle a\rangle = q_2\langle a\rangle$ then $q_1 = q_2$). Every distinguishing sequence is a homing sequence, but the converse is false. For example, although x is a homing sequence for the environment of Fig. 1, it is not a distinguishing sequence since if $q\langle \mathtt{x}\rangle$ is observed to be 11, then it is impossible to determine if the starting state q was A or B. In fact, this environment has *no* distinguishing sequence. In contrast, every environment has a homing sequence. Distinguishing sequences will play an important role in the permutation-environment algorithms of Sections 6 and 7.

It is perhaps also helpful to contrast the notion of a homing sequence with that of a *synchronizing sequence*. A synchronizing sequence is a sequence a whose execution brings the automaton into the same state, regardless of the starting state. That is, $q_1 a = q_2 a$ for all states q_1 and q_2. For example, xxx is a synchronizing sequence for the environment of Fig. 1 since $q\mathtt{xxx} = \mathtt{A}$ for all states q. Every synchronizing sequence is a homing sequence (trivially), but the converse is false. (For example, in Fig. 1, x is not a synchronizing sequence.) Also, not every environment has a synchronizing sequence. (For example, no non-trivial permutation automaton has a synchronizing sequence.)

Kohavi (1978) gives a complete discussion of homing sequences (as well as distinguishing and synchronizing sequences). He distinguishes between *preset* and *adaptive* homing sequences. Initially, we make use only of the former because they are simpler; later, we show that our inference procedures can be improved using adaptive homing sequences.

We outline next two techniques for constructing homing sequences. These will play central roles in all of the inference algorithms that follow.

Given full knowledge of the structure of $\mathcal{E}$, a homing sequence h can be constructed easily as shown in Fig. 3. Initially, $h = \lambda$. On each iteration of the loop, a new extension x is appended to the end of h so that h now distinguishes two states not previously distinguished. Thus, $|Q\langle h\rangle| < |Q\langle hx\rangle| \leqslant n$, and therefore the program will terminate after at most $n-1$ iterations. Further, since each extension need only have length $n-1$ (see, for instance, Kohavi (1978, Theorem 10-2)), we have shown how to construct a homing sequence of length at most $(n-1)^2$.

```
Input:  E - a finite-state automaton
Output: h - a homing sequence
Procedure:
1 h ← λ
2 while q1⟨h⟩ = q2⟨h⟩ but q1h ≠ q2h for some q1, q2 ∈ Q do
3     let x ∈ A distinguish q1h and q2h
4     h ← hx
5 end
```

FIG. 3. A state-based algorithm for constructing a homing sequence.

In Sections 5 and 7, we will be interested in a special kind of homing sequence that is formulated in terms of the diversity-based framework. Recall that a homing sequence h has the defining property that the final state reached after executing h from some starting state q can be determined from the observed output sequence $q\langle h\rangle$. Recall also that the current state of the environment is given by the values of all the tests $t \in A$. Thus, h is a homing sequence if the value of every test t can be determined at qh, the final state reached, given the observed output sequence $q\langle h\rangle$.

An important special case occurs when the value of any test t at qh is simply equal to one of the components of $q\langle h\rangle$, for every state q. That is, for some prefix p of h, $\gamma(qht) = \gamma(qp)$ for all $q \in Q$, which of course means, by definition, that $p \equiv ht$.

More formally, we define a *diversity-based homing sequence* to be an action sequence h satisfying the property described above, namely, that for all tests $t \in A$ there exists a prefix p of h that is equivalent to ht.

As outlined above, every diversity-based homing sequence h is indeed a homing sequence. For suppose $q_1h \neq q_2h$. Then there is some t for which $\gamma(q_1ht) \neq \gamma(q_2ht)$. Since ht is equivalent to some prefix p of h, we have $\gamma(q_1p) \neq \gamma(q_2p)$. Thus, $q_1\langle h\rangle \neq q_2\langle h\rangle$.

As an example, it can be shown that $h = \mathrm{xxyx}$ is a diversity-based homing sequence for the environment represented in Figs. 1 and 2. For instance, if $t = \mathrm{yxy}$ then $ht = \mathrm{xxyxyxy} \equiv \mathrm{xxy}$.

Figure 4 shows an algorithm for constructing a diversity-based homing sequence h. Again, h is built up from λ by appending extensions x. On each iteration, the cardinality of the set $\{[p]: p \text{ prefix of } h\}$ increases by at least

```
Input:  E - a finite-state automaton
Output: h - a diversity-based homing sequence
Procedure:
1 h ← λ
2 while (∃x ∈ A)(∀p prefix of h) p ≢ hx do
3     h ← hx
4 end
```

FIG. 4. A diversity-based algorithm for constructing a homing sequence.

one; since the cardinality of this set is clearly bounded by D, there can be at most $D-1$ iterations. Also, each extension need be no longer than $D-1$. (For if $|x| \geq D$, then x has at least $D+1$ suffixes, at least two of which must be equivalent. Thus, for some p, r, s, $x = prs$, $rs \equiv s$ and $r \neq \lambda$; therefore, ps is a shorter extension of h than x for which hps is inequivalent to every prefix of h.) Thus, we can find a diversity-based homing sequence of length at most $(D-1)^2$.

Some other remarks about the length of homing sequences: First, the homing sequences constructed by the preceding algorithms are the best possible in the sense that there exist environments whose shortest homing sequence has length $\Omega(n^2)$ (or $\Omega(D^2)$). However, given a state-based (or a diversity-based) description of a finite-state machine, it is NP-complete to find the shortest homing sequence for the automaton. (This can be shown, for instance, by a reduction from exact 3-set cover.)

4. A State-Based Algorithm for General Automata

In this and the next sections, we describe general algorithms for inferring the structure of an unknown environment $\mathcal{E}$.

We say that the learner has a *perfect model* of its environment if it can predict perfectly the output of the environment given any sequence of actions. The goal of our inference procedures is to construct a perfect model.

We assume that the learner is given access to $\mathcal{E}$, in other words, that the learner can observe the output of the environment when actions of its own choosing are executed. We also assume that there is a "teacher" who provides the learner with counterexamples to incorrectly conjectured models of the environment. A counterexample is a sequence of actions whose true output from the current state differs from that predicted by the learner's model. Typically, there will be many sequences of actions that are counterexamples to a given conjecture, and by choosing an especially long or short counterexample, the teacher can significantly affect the running time of the procedure. This fact is reflected in our running times which depend on the length of the counterexamples provided.

The notion of a teacher who provides counterexamples may at first seem unnatural in the context of a robot exploring its environment. However, in practice, it may often be possible for the robot to find counterexamples on its own without the aid of an outside agent. For instance, we might imagine the robot, upon completion of a model of the environment which it believes to be correct, using that model to make predictions of the output of the environment's next state until an incorrect prediction is made. In this situation, the sequence of actions leading up to the error is the needed

counterexample. Thus, the number of counterexamples needed by our algorithms can be viewed as the number of times the robot's conjectured world model must be revised due to incorrect predictions.

Alternatively, to find a counterexample, the robot might simply execute a random sequence of actions. Again, a counterexample is obtained if the random walk causes the robot's model to make an incorrect prediction. The use of random walks in lieu of counterexamples is a central theme of the algorithms for permutation automata presented in Sections 6 and 7. It is an open problem to characterize generally those environments in which random walks are an adequate substitute for an external source of counterexamples.

We assume that the unknown automaton is *strongly connected*, that is, every state can be reached from every other state:

$$(\forall q_1 \in Q)(\forall q_2 \in Q)(\exists a \in A)(q_1 a = q_2).$$

We make this assumption with little loss of generality: if $\mathcal{E}$ is not strongly connected, then an experimenting inference procedure, having no reset operation, will sooner or later fall into a strongly connected component of the state space from which it cannot escape, and so will have to be content thereafter learning only about that component.

This section focuses on an algorithm based on the global state representation for inferring an arbitrary unknown automaton.

4.1. *Angluin's L* Algorithm*

Our procedure is based closely on Angluin's L^* algorithm for learning regular sets (1987). Angluin shows how to efficiently infer the structure of any finite-state machine in the presence of what she calls a *minimally adequate teacher*. Such a teacher can answer two kinds of queries: On a *membership query*, the learner asks whether a given input string w is in the unknown language U, i.e., whether the string is accepted by the unknown machine. (In our framework, a string w is accepted if $\gamma(q_0 w) = 1$.) On an *equivalence query*, the learner conjectures that the unknown machine is isomorphic to one it has constructed. The teacher replies that the conjecture is either correct or incorrect, and in the latter case provides a counterexample w, a string accepted by one machine but not the other.

The idea of Angluin's algorithm is to maintain an *observation table* (S, E, T). Here, S is a prefix-closed set of strings, and E is a suffix-closed set of strings. We can think of S as a set of strings that lead from the start state to the states of the automaton, and E as experiments which are executed from these states. The last variable T is a two-dimensional table whose rows are given by $S \cup SB$, and whose columns are given by E. Each

entry $T(se)$, where $s \in S \cup SB$ and $e \in E$, records whether the string se is in the unknown language. For fixed s, Angluin denotes by row(s) the vector of entries $T(se)$ for varying $e \in E$. Her algorithm extends S and E based on the results of queries, and ultimately outputs the correct automaton based on an equivalence between the states of the unknown machine and the distinct rows of the table T. We denote by N_M and N_E the number of membership and equivalence queries made by L^*. These variables are implicit functions of n, k and m, where m is the length of the longest counterexample received. For Angluin's procedure L^*, we have $N_M = O(kmn^2)$ and $N_E = n - 1$. However, in Section 4.5 below, we show how N_M can be improved to $O(kn^2 + n \log m)$.

In our framework, the learner could easily simulate Angluin's algorithm L^* if it were given a reset: to perform a membership query on w, the learner resets the environment, and executes the actions of w, observing the output of the last state reached. To perform an equivalence query on $\mathcal{E}'$, the learner resets the automaton and conjectures that $\mathcal{E}'$ is a perfect model of the environment. The teacher returns an action sequence w on which the conjectured model fails; this is the counterexample needed by L^*.

4.2. *Using a Homing Sequence in Lieu of a Reset*

Of course, in our model the learner is not provided with a reset. The main idea of our algorithm is to replace the reset with a homing sequence. In many respects, a homing sequence behaves like a reset: by executing the homing sequence, the learner discovers "where it is," what state it is at in the environment. However, unlike a reset, the final state is not fixed, and the learner does not know beforehand what state it will end up in. (As mentioned above, an automaton need not possess a synchronizing sequence, a sequence that forces the automaton into a given state independent of its starting state. So we use homing sequences instead.)

We begin by supposing that the learner has been provided with a true homing sequence h. Later, we show how to remove this assumption.

Suppose we execute h from the current state q, producing output $\sigma = q\langle h \rangle$. If we ever repeat this experiment from state q' and find $q'\langle h \rangle = \sigma$, then, because h is a homing sequence, the states where we finished must have been the same in both cases: $qh = q'h$. If we could guarantee that the output of h would continue to come up σ with good regularity, then we could simply infer $\mathcal{E}$ by simulating Angluin's algorithm, treating qh as the initial state. When L^* demands a reset, we execute h: if the output comes up σ, then we must be at qh, and our "reset" has succeeded; otherwise, try again. Unfortunately, in the general case, it may be very difficult to make h produce σ regularly.

Instead, we simulate an independent copy L^*_σ of L^* for each possible

```
Input:  access to 𝓔, a finite-state automaton
        h - a homing sequence for 𝓔
Output: a perfect model of 𝓔
Procedure:
 1 repeat
 2     execute h, producing output σ
 3     if it does not already exist, create L*_σ, a new copy of L*
 4     simulate the next query of L*_σ:
 5         if L*_σ queries the membership of action sequence a then
 6             execute a and supply L*_σ with the output of the final state reached
 7         if L*_σ makes an equivalence query then
 8             if the conjectured model 𝓔' is correct then
 9                 stop and output 𝓔'
10             else
11                 obtain a counterexample and supply it to L*_σ
12 end
```

FIG. 5. A state-based algorithm for inferring $\mathscr{E}$ given a correct homing sequence.

output σ of executing h, as shown in Fig. 5. Since $|Q\langle h\rangle| \leqslant n$, no more than n copies of L^* will be created and simulated. Furthermore, on each iteration of the loop, at least one copy makes one query and so makes progress towards inference of $\mathscr{E}$. Thus, this algorithm will succeed in inferring $\mathscr{E}$ after no more than $n(N_M + N_E)$ iterations.

4.3. *Constructing a Homing Sequence*

We now describe how to combine construction of the homing sequence h with the inference of $\mathscr{E}$. We maintain throughout the algorithm a sequence h which we presume is a true homing sequence. When evidence arises indicating that this is not the case, we will see how h can be extended and improved, eventually leading to the construction of a correct homing sequence. Initially, we take $h = \lambda$.

We use our presumably correct homing sequence h as described above and in Fig. 5. If h is indeed a true homing sequence, we will of course succeed in inferring $\mathscr{E}$.

On the other hand, if h is incorrect, we may discover *inconsistent behavior* in the course of simulating some copy of L^*: suppose on two different iterations of the loop in Fig. 5, we begin in states q_1 and q_2, execute h, produce output $q_1\langle h\rangle = q_2\langle h\rangle = \sigma$, and, as part of the simulation of L^*_σ, execute action sequence x. If h were a homing sequence, then x's output would have to be the same on both iterations since $q_1 h$ and $q_2 h$ must be equal.

However, if h is not a homing sequence, then it may happen that $q_1 h\langle x\rangle \neq q_2 h\langle x\rangle$. That is, we have discovered that x distinguishes $q_1 h$ and $q_2 h$, and so, just as was done in the algorithm of Fig. 3, we replace h

Input: access to $\mathcal{E}$, a finite-state automaton
n - the number of states of $\mathcal{E}$
Output: a perfect model of $\mathcal{E}$
Procedure:
1 $h \leftarrow \lambda$
2 **repeat**
3 execute h, producing output σ
4 if it does not already exist, create L_σ^*, a new copy of L^*
5 **if** $|\{row(s) : s \in S_\sigma\}| \leq n$ **then**
6 simulate the next query of L_σ^* as in Figure 5 (and check for inconsistency)
7 **else**
8 let $\{s_1, \ldots, s_{n+1}\} \subset S_\sigma$ be such that $row(s_i) \neq row(s_j)$
9 randomly choose a pair s_i, s_j from this set
10 let $e \in E_\sigma$ be such that $T_\sigma(s_i e) \neq T_\sigma(s_j e)$
11 with equal probability, re-execute either $s_i e$ or $s_j e$ (and check for inconsistency)
12 if inconsistency found executing some string x **then**
13 discard all existing copies of L^*
14 $h \leftarrow hx$
15 **until** a correct conjecture is made

FIG. 6. A state-based algorithm for inferring $\mathcal{E}$.

with hx, producing in a sense a "better" approximation to a homing sequence. At this point, the existing copies of L^* are discarded, and the algorithm begins from scratch (except for resetting h, of course). Since h can only be extended in this fashion $n-1$ times, this only means a slowdown by at most a factor of n, compared to the algorithm of Fig. 5.

Figure 6 shows how we have implemented these ideas. Here we have assumed n, the number of global states, has been provided to the learner. In fact, this assumption is entirely unnecessary. Although we omit the details, we can show that the stated bounds below hold (up to a constant) for a slightly modified algorithm which does not require that the learner be explicitly provided with the value of n. The trick is the usual one of repeatedly doubling our estimate of n.

Recall that L^* requires maintenance of an observation table (S, E, T). Let $(S_\sigma, E_\sigma, T_\sigma)$ denote the observation table of L_σ^*. Of course, T_σ can only record output produced when executing an action sequence from what is only *presumed* to be a fixed initial state.

Angluin's analysis implies that if L_σ^* makes more than $N_M + N_E$ queries, then the number of distinct rows will exceed n. This can only happen if h is not a homing sequence, but how do we know how to correctly extend h if we have not actually seen an inconsistency? We show that if an inconsistency has not been found by the time the number of rows exceeds n, then we can use a probabilistic strategy to find one quickly with high probability.

Suppose we execute h from state q, with output σ, and we find that for

L_σ^*, there are more than n distinct rows. Then, as in Fig. 6, there exist strings $s_1, \ldots, s_{n+1}$ in S_σ whose rows are all distinct. By the pigeon-hole principle, there is at least one pair of distinct rows s_i, s_j such that $qhs_i = qhs_j$. Further, since $\text{row}(s_i) \neq \text{row}(s_j)$, there is some $e \in E_\sigma$ for which $T_\sigma(s_i e) \neq T_\sigma(s_j e)$. However, $\gamma(qhs_i e) = \gamma(qhs_j e)$. Therefore, either $\gamma(qhs_i e) \neq T_\sigma(s_i e)$ or $\gamma(qhs_j e) \neq T_\sigma(s_j e)$, and so re-executing $s_i e$ (or $s_j e$, respectively) from the current state qh will produce the desired inconsistency. (Recall that T_σ records the results of previous executions of these strings.) If such an inconsistency is detected, then h is extended as usual with the appropriate sequence ($s_i e$ or $s_j e$).

So the chance of randomly choosing the correct pair s_i, s_j as above is at least $\binom{n+1}{2}^{-1}$, and the chance of then choosing the correct experiment to re-run of $s_i e$ or $s_j e$ is at least $1/2$. Thus, the probability of finding an inconsistency using the technique of Fig. 6 in this situation is at least $1/n(n+1)$. Repeating this technique $n(n+1)\ln(1/\delta)$ times gives a probability of at least $1-\delta$ of finding an inconsistency.

Since h is extended at most $n-1$ times, and since at most n copies of L^* can be in existence at any one time, there can be at most $n(n-1)$ copies of L^* ever created by the procedure. Thus, the total number of counterexamples required is at most $n(n-1)N_E$. Also, since each of the $n-1$ extensions of h has length $O(n+m)$ (a bound on the length of any query required by L^*), the total length of h cannot exceed $O(n^2+nm)$. Finally, since the probabilistic procedure described above for finding inconsistencies may need to be applied to each of the $n(n-1)$ copies of L^*, we replace δ with δ/n^2 to ensure an overall probability of success of $1-\delta$.

Putting these facts together, we have proved:

THEOREM 1. *Given $\delta > 0$, the algorithm described in Fig. 6 halts and outputs a perfect model with probability at least $1-\delta$ in time polynomial in n, m, k and $\log(1/\delta)$, and after executing*

$$O(n^3(n+m)(n^2\log(n/\delta) + N_M + N_E))$$

actions. Also, the total number of counterexamples required is at most $O(n^2 N_E)$.

If we assume $m = O(n)$ and $k = O(1)$ and use the previously given bounds on N_M and N_E, then the number of actions executed by the procedure (and the running time as well) simplifies to $O(n^6 \log(n/\delta))$.

The procedure can be modified, replacing the preset homing sequence which we have been using with an adaptive one whose input at each step depends on the output seen up to that point. This modification shaves a factor of n off the bound given above, and is described in greater detail in the next section.

It is an open question whether this bound can be significantly tightened. It seems likely that an algorithm which combines the many copies of L^* into one would have a superior running time, but we have not been successful in implementing this intuition.

4.4. *Adaptive Homing Sequences*

The algorithm of Fig. 6 is certainly quite wasteful in that, when h is discovered not to be a homing sequence, everything is thrown away and the algorithm starts over from scratch. As a result, up to n copies of L^* are discarded each time h is extended. Since h can be extended up to $n-1$ times, this means as many as n^2 copies of L^* may eventually be simulated by the algorithm.

In this section, we describe a way of modifying the procedure so that only one copy of L^* is discarded when h is extended, leading to an $O(n)$ bound on the total number of copies of L^* that are simulated.

As mentioned above, the idea of the modification is to replace our preset homing sequence (the kind described up to this point) with an adaptive one. In many ways, preset homing sequences are rather inefficient tools. For example, it may be that, starting from some states, executing only half the sequence is sufficient to reach a state uniquely determined by the observed output. An adaptive homing sequence is a much more intelligent kind of homing sequence. It is like a preset homing sequence in that the output observed can be used to determine the state reached. However, the difference is that the action executed at each step may depend on the output observed up to that point.

Despite its name, an *adaptive sequence* a is not a sequence at all but a decision tree with the following properties: The root node of a is labeled λ, and each of the other nodes in the tree is labeled with one of the basic actions in B. Every node has at most one 0-child, and at most one 1-child. An example adaptive sequence is given in Fig. 7.

An adaptive sequence is executed in a natural manner: We begin at the

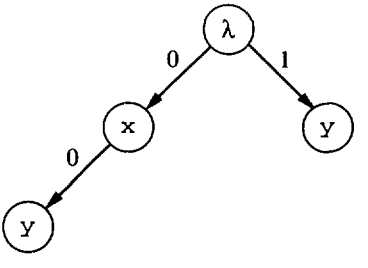

FIG. 7. An example adaptive homing sequence.

root node. If the output of the current state is 0 (or 1) then we proceed to the 0-child (1-child) of the root. The basic action labeling the node reached is then executed, and, based on the resulting output, we proceed down the tree in the same fashion, at each step branching to the 0- or 1-child depending on the output observed. This continues until we "fall off" the tree, i.e., until it is necessary to move to a node that does not exist in the tree.

For example, if the tree of Fig. 7 is executed from the current state of Fig. 1, then the action sequence "x" will be executed producing output 01; if the sequence is executed from state D, then "xy" will be executed with output 001.

As with ordinary sequences, we write qa to denote the state reached by executing adaptive sequence a from state q, and we write $q\langle a\rangle$ to denote the output produced by executing a from state q. Thus, as for preset homing sequences, an *adaptive homing sequence* is an adaptive sequence h for which $q_1\langle h\rangle = q_2\langle h\rangle$ implies $q_1 h = q_2 h$ for all $q_1, q_2 \in Q$.

4.4.1. *Modifying the Algorithm*

We are now ready to describe how the algorithm of Fig. 6 can be modified to use adaptive rather than preset homing sequences. The structure of the algorithm is not changed at all. Nor is the simulation of queries, the handling of over-sized copies of L^*, etc. Only the construction of the adaptive homing sequence h is modified.

Initially, h is chosen to be the adaptive sequence consisting just of a root node λ. As before, h is repeatedly executed; each time, its output selects a copy of L^*. A query of the selected copy is then simulated, just as before. Now, however, a detected inconsistency is handled differently: Suppose an inconsistency is found executing x. More precisely, suppose that on two different iterations, we began in states q_1 and q_2, executed h, and observed output $q_1\langle h\rangle = q_2\langle h\rangle = \sigma$. Further, when x was then executed, it was discovered that $q_1 h\langle x\rangle \neq q_2 h\langle x\rangle$. This latter fact implies that $q_1 h \neq q_2 h$, and so h cannot be an adaptive homing sequence. As before, we would like to use x to repair h: we would like to "graft" x onto tree h so that the resulting tree h' distinguishes q_1 and q_2.

In fact, this can be done quite easily: Let $x = b_1 \cdots b_r$, where $b_i \in B$, and let v_0 be the last node of h visited when h was executed from q_1 and q_2 (it must be the same node in both cases since $q_1\langle h\rangle = q_2\langle h\rangle$). In other words, when h was executed from q_1, our traversal of tree h must have ended at v_0. For the traversal to have terminated at this point, it must be the case that v_0 has no child labeled with the output of the final state reached by executing h from q_1; that is, v_0 must have no $\gamma(q_1 h)$-child. The grafted tree h' is the same as h except that in h', node v_0 has a $\gamma(q_1 h)$-child which is the root of a linear subtree corresponding to the execution of x.

More precisely, in h', each node v_{i-1} has a $\gamma(q_1 h b_1 \cdots b_{i-1})$-child v_i labeled b_i, for $1 \leqslant i \leqslant r$.

It can be verified that $q_1\langle h' \rangle \neq q_2\langle h' \rangle$ since $q_1 h\langle x \rangle \neq q_2 h\langle x \rangle$. Thus $|Q\langle h \rangle| < |Q\langle h' \rangle| \leqslant n$, and so h will be grafted in this fashion at most $n-1$ times.

So line 14 in Fig. 6 is replaced by a call to a grafting subroutine as described above. Further, since h and h' are the same except for node v_0, it is no longer necessary to discard at line 13 all copies of L^*—it is sufficient to discard only L^*_σ, the copy on which an inconsistency was discovered. Thus, since $|Q\langle h \rangle|$ increases each time a single copy of L^* is discarded, at most $n-1$ copies are ever discarded throughout the execution of the algorithm. Since the number of copies in existence at any one time is also bounded by n, it follows that at most $2n-1$ copies of L^* are simulated by this modified procedure. Thus, this improves the bounds given in Theorem 1 by a factor of $\theta(n)$, both on the number of actions executed and on the number of counterexamples required.

4.5. *Improving Angluin's L* Algorithm*

In this section, we describe a variant of Angluin's L^* algorithm that significantly improves the worst-case number of membership queries made by the inference procedure. This, in turn, leads to immediate improvements in the performance of our homing sequence algorithms.

As mentioned above, Angluin's algorithm maintains an observation table (S, E, T). The function or table T records the value $T(x) = \gamma(q_0 x)$ for each string $x \in (S \cup SB)E$. (Here, q_0 is $\mathcal{M}$'s initial state to which, in Angluin's model, the automaton can always be reset.) The entries of T are filled in using membership queries, and it follows that the number of queries needed is just the cardinality of $(S \cup SB)E$. For Angluin's algorithm, $|S|$ is bounded by $O(mn)$, and $|E|$ by $O(n)$. Our algorithm improves on Angluin's by limiting $|S|$ to just n; however, to achieve this bound on $|S|$, $n \lg m$ additional queries will be needed, giving an overall bound of $O(kn^2 + n \log m)$ on the required number of membership queries.

As mentioned earlier, S is a prefix-closed set of strings representing states of $\mathcal{M}$. Unlike Angluin's algorithm, ours maintains the condition that for all $s_1, s_2 \in S$, if $s_1 \neq s_2$ then $q_0 s_1 \neq q_0 s_2$. Thus, $|S| \leqslant n$ at all times. Also, S only grows in size (strings are never deleted from S). The set E represents a set of experiments which distinguish the states of S (i.e., the states $q_0 s$ for $s \in S$).

Here is an outline of our algorithm, which is very similar to Angluin's. Initially, S and E are initialized to the set $\{\lambda\}$. Using membership queries, fill in the entries of table T, and make (S, E, T) closed (discussed below). Then, from (S, E, T), construct and conjecture machine $\mathcal{M}'$. If the

conjecture is correct, quit. Otherwise, update the set E using the returned counterexample, and repeat until a correct conjecture is made.

We say observation table (S, E, T) is *closed* if for all $s \in SB$ there exists $s' \in S$ such that $\text{row}(s) = \text{row}(s')$. (Recall that $\text{row}(s)$ is that function $f: E \to \{0, 1\}$ for which $f(e) = T(se)$.) If $s \in SB$ witnesses that (S, E, T) is not closed, then s is simply added to S (and T updated using membership queries). Note that this maintains the condition that all rows of S are distinct (and thus, the states to which they lead from q_0 are also distinct).

(Angluin's algorithm also requires that the observation table be *consistent*, that is, that $\text{row}(s_1 b) = \text{row}(s_2 b)$ whenever $\text{row}(s_1) = \text{row}(s_2)$ for $s_1, s_2 \in S$ and $b \in B$. However, since our algorithm maintains the condition that $\text{row}(s_1) \neq \text{row}(s_2)$ for $s_1 \neq s_2$, this condition is always trivially satisfied.)

The conjectured machine $\mathscr{E}' = (Q', B, \delta', q_0', \gamma')$ is constructed in a natural manner: its state set is $Q' = S$ with initial state $q_0' = \lambda$; its output function is defined by $\gamma'(s) = T(s)$; and its transition function is given by $\delta'(s, b) = s'$, where s' is that unique member of S for which $\text{row}(sb) = \text{row}(s')$.

Finally, if $\mathscr{E}'$ is different from $\mathscr{E}$, a counterexample z is obtained, and the set E must be updated. Our algorithm adds only a single string to E using z. However, to find this string, the procedure makes up to $\lg |z|$ membership queries.

The key property that must be satisfied by the new experiment e (which will be added to E) is the following: for some $s, s' \in S$ and $b \in B$ for which $\text{row}(s) = \text{row}(s'b)$, it must be that $\gamma(q_0 se) \neq \gamma(q_0 s'be)$. That is, experiment e must witness that $q_0 s$ and $q_0 s'b$ are different states. If this property is satisfied, then adding e to E will cause $|S|$ to increase by at least one (to maintain closure) so that the total number of equivalence queries is bounded by $n - 1$.

We now describe how such an experiment can be found. For $0 \leqslant i \leqslant |z|$, let p_i, r_i be such that $z = p_i r_i$, and $|p_i| = i$. Let $s_i = \delta'(\lambda, p_i)$ be the state reached in $\mathscr{E}'$ after the first i symbols of z have been executed. Recall that s_i is both a state of $\mathscr{E}'$ and a string of A.

On input z, machine $\mathscr{E}$ reaches a state outputting the value $\gamma(q_0 z) = \gamma(q_0 s_0 r_0)$. (Assume this value is 0.) On the other hand, on input z, machine $\mathscr{E}'$ reaches a state outputting the value $\gamma'(\delta'(\lambda, z)) = \gamma'(s_{|z|}) = T(s_{|z|}) = \gamma(q_0 s_{|z|} r_{|z|})$. Since z is a counterexample, this value must be 1.

Let $\alpha_i = \gamma(q_0 s_i r_i)$. Roughly speaking, α_i is the output obtained by processing the first i symbols of z using $\mathscr{E}'$, and the rest using $\mathscr{E}$. Note that, by simulating $\mathscr{E}'$, s_i can be computed, and so α_i can be determined with a membership query for any i. From the comments above, we have that $\alpha_0 = 0$ and $\alpha_{|z|} = 1$. Using a kind of binary search, we can find some i such that $\alpha_i \neq \alpha_{i+1}$ (such an i clearly must exist): first we query $\alpha_{|z|/2}$; if the result

is 1, then query $\alpha_{|z|/4}$; otherwise, query $\alpha_{3|z|/4}$, etc. In this manner, such an i can be found in $\lg |z|$ queries.

We claim then that r_{i+1} is the desired experiment: Let b be the first symbol of r_i. Then $\gamma(q_0 s_i b r_{i+1}) = \gamma(q_0 s_i r_i) = \alpha_i \neq \alpha_{i+1} = \gamma(q_0 s_{i+1} r_{i+1})$. However, by definition of s_i, we have $s_{i+1} = \delta'(s_i, b)$ and so $\text{row}(s_{i+1}) = \text{row}(s_i b)$. Thus, as argued above, adding r_{i+1} to E causes $|S|$ to increase. It follows that at most $n-1$ equivalence queries are required by the algorithm. For each equivalence query, $\lg m$ membership queries are needed to find the right experiment to add to E. Also, since $|E| \leqslant n$ and $|S| \leqslant n$, at most $|(S \cup SB)E| \leqslant (k+1)n^2$ membership queries are needed to record the entries of T. Finally, it can be seen that each membership query has length at most $n+m$. The procedure is clearly polynomial time, and its correctness follows from arguments given above and by Angluin.

In a quite naive implementation of the algorithm, the rows of S are filled in first, and, once a row of some string in SB has been completed, it is compared in $O(n^2)$ time to every other row of S until an identical row is discovered, or until it is determined that there is no other identical row in S (in which case, (S, E, T) is not closed). For even such a naive implementation, it can be verified that each query requires processing time that is at worst proportional to the bound of $O(n^2 + nm)$ on the length of h in the algorithm of Fig. 6.

Combining this improvement to L^* with the adaptive homing sequence ideas described in Section 4.4, we thus have shown:

THEOREM 2. *There exists an algorithm that halts and outputs a perfect model of any finite-state environment $\mathcal{E}$ with probability at least $1-\delta$. The algorithm's running time, and the number of actions executed are both bounded by*

$$O(n^3(n+m)(n \log(n/\delta) + kn + \log m)).$$

Also, the total number of counterexamples required is at most $O(n^2)$.

5. A DIVERSITY-BASED ALGORITHM FOR GENERAL AUTOMATA

In this section, we describe a diversity-based algorithm for inferring finite automata in the general case.

In the presence of a reset, it can be shown that Angluin's L^* algorithm can be modified to infer the unknown automaton in time polynomial in the diversity D (rather than the number of states n). This follows from an observation made independently by Angluin and Young that is described in Section 2.7 of Schapire's master's thesis (1988). Briefly, this observation

states that the update graph captures, in some sense, the "reverse behavior" of the unknown automaton $\mathscr{E}$. More specifically, by reversing the direction of all edges in $\mathscr{E}$'s update graph, we obtain a structure that is essentially equivalent to another automaton $\mathscr{E}'$. Angluin and Young's observation says that a string w is accepted by $\mathscr{E}$ (i.e., $\gamma(q_0 w) = 1$) if and only if w^R, the reverse of w, is accepted by $\mathscr{E}'$. Thus, a structure that is basically isomorphic to the update graph (modulo the reversal of all edge directions) can be inferred using L^* (assuming a reset) by appropriately "reversing" all queries made by the algorithm.

In the absence of a reset, the problem of inferring a diversity-based representation becomes considerably more involved. As far as we know, there is no direct application of L^* that solves this problem, as was the case in Section 4.

The main idea of the algorithm described in this section is to construct a simple-assignment automaton that is equivalent to the update graph. More specifically, the algorithm builds a set T of tests which eventually contains exactly one representative of each equivalence class. These tests become the variables of the constructed simple-assignment automaton, with the obvious correspondence of tests $t \in T$ to nodes $[t]$ of the update graph.

Recall that, in the update graph, each node $[t]$ has a single in-going b-edge which is directed from node $[bt]$. Therefore, to construct an appropriate update function, our algorithm tries to determine for each test $t \in T$ and basic action $b \in B$, that test $t' \in T$ which is equivalent to bt. Setting $\Upsilon(t, b) = t'$, this clearly yields a structure that is isomorphic to the update graph.

So to summarize, our algorithm constructs a simple-assignment automaton by constructing a set T of tests representing all of the equivalence classes, and by determining which test in T is equivalent to x, for all $x \in BT$.

Initially, the set T is the singleton $\{\lambda\}$. A test t is added to T only after it has been determined that t is inequivalent to every test already in T. Thus, at all times, $|T| \leqslant D$, and, as desired, each test of T represents a different test-equivalence class (i.e., a node of the update graph); eventually, we would like for all of the equivalence classes to be represented.

As described above, for each test $x \in BT$, we want to determine that test in T which is equivalent to x. To this end, a function or table $r: BT \to 2^T$ is maintained with the interpretation that $r(x)$ represents those tests in T which are plausibly equivalent to x. Initially, $r(x) = T$ since at the start we have no evidence that any test $t \in T$ is or is not equivalent to x. When evidence arises that $t \not\equiv x$, the test t is removed from $r(x)$.

Note that if $|T| = D$ (so that every equivalence class is represented in T), and if, for all $x \in BT$, $r(x)$ is a singleton $\{s_x\}$ for some $s_x \in T$, then x must

be equivalent to s_x, and, as outlined above, a simple-assignment automaton isomorphic to the update graph can be easily constructed: its variable set is T, its output variable is λ, and its update function Υ is defined by $\Upsilon(t, b) = s_{bt}$. (The initial values function ω is handled below.)

The simple-assignment automata conjectured by our algorithm are constructed from T and r in a very similar manner. We choose $V = T$ and $v_0 = \lambda$. However, in general, it may not be the case that $|r(x)| = 1$ for all $x \in BT$. Therefore, we choose $\Upsilon(t, b)$ to be an *arbitrary* element of $r(bt)$. If one or more of our choices is incorrect, then we can use the provided counterexample to correct our error. More precisely, we show below that, using experiments and counterexamples to this conjectured automaton, we can find $t \in T$ and $b \in B$ such that $\Upsilon(t, b) \not\equiv bt$. When this happens, our choice for $\Upsilon(t, b)$ can be removed from $r(bt)$. Thus, for some $x \in BT$, $r(x)$ is reduced in size.

Also, note that if $r(x)$ is reduced to the empty set, then x is inequivalent to every member of T, and so can itself be added to T. The table r is then updated appropriately. Since $|T| \leqslant D$, since $r(x) \subset T$, and since some $r(x)$ shrinks on each iteration, it follows that this simplified algorithm converges to a perfect model after at most $(k+1)D^2$ iterations.

5.1. *An Algorithm That Uses a Provided Homing Sequence*

As in Section 4, we assume initially that a diversity-based homing sequence h is given. Later, we show how h can be constructed.

Let t be any test. Then ht is equivalent to some prefix of h. It will be important, for selected tests t in A, to determine specifically that prefix which is equivalent to t. For this reason, we maintain *candidate sets* $C(t) \subset \{0, ..., |h|\}$ representing the prefixes of h which are plausibly equivalent to ht. Let h_i denote that prefix of h of length i. Initially, $C(t) = \{0, ..., |h|\}$, and, when it has been determined that $h_i \not\equiv ht$, index i is removed from the set. Note that when ht is executed from some state q, both of the outputs $\gamma(qh_i)$ and $\gamma(qht)$ are observed since h_i is a prefix of ht. Thus, if we find that these outputs differ, then clearly $h_i \not\equiv ht$ and so i can be deleted from $C(t)$.

Suppose h has been executed from some state q producing output $\sigma = \langle \sigma_0, ..., \sigma_{|h|} \rangle$. We say that a set $X \subset \{0, ..., |h|\}$ is *coherent* (with respect to σ) if $\sigma_i = \sigma_j$ for $i, j \in X$. If X is coherent, then the common value of all σ_i with $i \in X$ is called X's *selected value* (with respect to σ), and it is denoted $\sigma[X]$.

Note that, if $C(t)$ is coherent, then the value of t in the current state qh is known—it is just $C(t)$'s selected value (since ht must be equivalent to some h_i with $i \in C(t)$). On the other hand, if candidate set $C(t)$ is incoherent, then if t is executed, at least one element of $C(t)$ will be eliminated.

What's more, if i is eliminated from $C(t)$, then every other index j for which $h_i \equiv h_j$ is also removed since the two tests have the same value in every state. That is, $|\{[h_i]: i \in C(t)\}|$ decreases by at least one. Thus, $C(t)$ can be reduced in this fashion at most $D-1$ times.

Also, if we find for tests t_1 and t_2 that $C(t_1)$ and $C(t_2)$ are disjoint, then ht_1 and ht_2 cannot possibly belong to the same equivalence class. Moreover, if for any $a \in A$ we find that $C(at_1)$ and $C(at_2)$ are disjoint, then $hat_1 \not\equiv hat_2$ and therefore $t_1 \not\equiv t_2$. This is the primary technique used by our procedure for determining inequivalence of tests (and thus for the elimination of tests from $r(x)$).

Our algorithm maintains a candidate set for each $t \in T$. If all of these candidate sets are coherent (after h has been executed from some state q), then the value of every test $t \in T$ is known in the current state qh; these values are used then to determine the function ω in the conjectured automaton. Specifically, if all the candidate sets for the tests in T are coherent, then a conjecture may be made in which V, v_0, and Υ are as described above, and $\omega(t)$ is taken to be the selected value of $C(t)$ (which is, from the preceding remarks, the value of t in the current state).

We describe next how a counterexample z to such a conjecture $\mathscr{S}$ is handled. The technique is similar in some ways to that described in Section 4.5. Let $z = p_i s_i$ where $|p_i| = i$ for $0 \leqslant i \leqslant |z|$. Let $t_i = \Upsilon(\lambda, s_i)$. Finally, let $u_i = p_i t_i$. We maintain henceforth a candidate set for each test u_i. Our hope is that these candidate sets will be reduced to the point that, for some i, $C(u_i) \cap C(u_{i+1}) = \varnothing$. For if this happens, then we can conclude that $u_i \not\equiv u_{i+1}$. Noting that $u_i = p_i t_i$ and $u_{i+1} = p_i b t_{i+1}$ where b is the last symbol of p_{i+1}, this implies that $t_i \not\equiv b t_{i+1}$. Since $t_i = \Upsilon(t_{i+1}, b) \in r(bt_{i+1})$, it follows that t_i can be deleted from $r(bt_{i+1})$ as desired.

We show that $C(u_0)$ and $C(u_{|z|})$ are disjoint. This will allow us, eventually, to reduce the candidate sets $C(u_i)$ sufficiently so that two consecutive sets $C(u_i)$ and $C(u_{i+1})$ will be made disjoint as needed. Note that the conjectured automaton $\mathscr{S}$ predicted that the value of z in the current state qh is $\omega(\Upsilon(\lambda, z)) = \omega(t_0)$. Assume this value is 0. Then, by ω's definition, $C(u_0) = C(t_0) \subset \sigma^{-1}(0)$, where $\sigma^{-1}(x) = \{0 \leqslant i \leqslant |h|: \sigma_i = x\}$. On the other hand, since z is a counterexample, $\gamma(qhz) = 1$. Thus, if z is executed from the current state, then $C(u_{|z|}) = C(z)$ will be included in $\sigma^{-1}(1)$, and, as claimed $C(u_0) \cap C(u_{|z|})$ will be empty.

Unfortunately, to continually reduce the sets $C(u_i)$, these sets must continually be found incoherent. This may be a problem because they may very well all be found to be coherent without any consecutive pair being disjoint. To handle this situation, our algorithm makes a new conjecture that leaves V, v_0 and Υ alone, but which chooses ω appropriately as described above. This gives a new sequence of tests u_i for which candidate sets must also be maintained. We show below that no more than $D-1$

Input: access to $\mathcal{E}$, a finite-state automaton
h - a diversity-based homing sequence for $\mathcal{E}$
Output: a perfect model of $\mathcal{E}$
Procedure:

```
1  T ← {λ}; C(λ) ← {0, ..., |h|}
2  r(b) ← T, ϒ(λ, b) ← λ for b ∈ B
3  ℓ ← 0
4  repeat
5      execute h, producing output σ
6      if C(t) is incoherent for some t ∈ T then
7          execute t and update C(t)
8      else if K(i, j) is incoherent for some 1 ≤ i ≤ ℓ, 0 ≤ j ≤ m_i then
9          choose the smallest i for which K(i, j) is incoherent for some 0 ≤ j ≤ m_i
10         execute u_ij and update K(i, j)
11     else
12         ω(t) ← σ[C(t)] for t ∈ T
13         conjecture S = (T, B, ϒ, λ, ω)
14         if S is a perfect model then
15             stop and output S
16         else
17             obtain counterexample z
18             ℓ ← ℓ + 1; m_ℓ ← |z|
19             for 0 ≤ j ≤ m_ℓ:
20                 u_ℓj ← p_j · ϒ(λ, s_j) where z = p_j s_j and |p_j| = j
21                 K(ℓ, j) ← {0, ..., |h|}
22             K(ℓ, 0) ← σ^{-1}(ω(u_ℓ0))
23             execute z = u_ℓm_ℓ and update K(ℓ, m_ℓ)
24     if K(i, j) ∩ K(i, j + 1) = ∅ for some 1 ≤ i ≤ ℓ, 0 ≤ j < m_i then
25         x ← b_0 t_0 where u_{i,j+1} = p b_0 t_0, |p| = j and b_0 ∈ B [this implies u_ij = p · ϒ(t_0, b_0)]
26         r(x) ← r(x) − {ϒ(t_0, b_0)}
27         if r(x) = ∅ then
28             r(t) ← r(t) ∪ {x} for t ∈ BT − T
29             T ← T ∪ {x}; C(x) ← {0, ..., |h|}
30             r(bx) ← T for b ∈ B
31         ϒ(t, b) ← any member of r(bt) for b ∈ B, t ∈ T
32         ℓ ← 0
33 end
```

FIG. 8. A diversity-based algorithm for inferring $\mathcal{E}$ given a diversity-based homing sequence.

such sequences need ever be started by the algorithm before one of the sets $r(x)$ is reduced.

The complete algorithm is shown in Fig. 8. In the figure, when a counterexample z is received, a sequence of tests $u_{l0}, \ldots, u_{lm_l}$ is constructed as described above; variable l counts the number of such counterexamples received for the same choice of Y. The set $K(i, j)$ is a candidate set for test u_{ij}. Note that the same test may have several candidate sets, not necessarily the same: even if $u_{ij} = u_{i'j'}$, it may be that $K(i, j) \neq K(i', j')$ if $\langle i, j \rangle \neq \langle i', j' \rangle$. Although this may seem inefficient, it appears to be necessary for proving the algorithm's correctness.

Also, candidate sets are updated in the obvious way: if $t \in T$ is executed leading to a state outputting the value x, then $C(t) \leftarrow C(t) \cap \sigma^{-1}(x)$ (and similarly for sets $K(i, j)$). Note that only the specified candidate set is modified.

THEOREM 3. *The algorithm described in Fig. 8 halts in polynomial time after executing at most*

$$O(kmD^4(|h| + D + m))$$

actions, and outputs a perfect model.

Proof. If the algorithm halts, then it outputs a perfect model. Therefore, it suffices to prove that it halts having executed only the stated number of actions.

Most of the arguments needed to prove this theorem were given above. Here, we try to pull those arguments together, filling in missing details. Below, we say that a property holds *on each iteration* if it holds at the beginning each iteration of the main loop.

First, on each iteration, if $i \notin C(t)$ then $h_i \not\equiv ht$ for any t. This follows from the manner in which C is updated. Also, the contrapositive implies that $C(t)$ is non-empty on each iteration since $h_i \equiv ht$ for some i since h is a diversity-based homing sequence. These statements hold also for candidate sets $K(i, j)$. (At line 22, this follows from the fact that, by ω's definition, $C(u_{i0}) \subset \sigma^{-1}(\omega(u_{i0}))$.)

These facts imply that if $K(i, j) \cap K(i, j+1) = \varnothing$ as at line 24, then $u_{ij} \not\equiv u_{i,j+1}$, which implies that $x = b_0 t_0 \not\equiv \Upsilon(t_0, b_0)$, where b_0, t_0 and x are as defined at line 25. Therefore, we can conclude generally that, on each iteration, $t \notin r(x)$ only if $x \not\equiv t$, for $t \in T$, $x \in BT$. Note also that $r(x)$ is non-empty on each iteration, due to the code at lines 27–30.

Thus, if the last element of $r(x)$ is eliminated, then $x \not\equiv t$ for all $t \in T$, and so it follows that the tests in T are pairwise independent. Thus, by definition of diversity, $|T| \leqslant D$ on each iteration, and so lines 25–32 are executed at most $(k+1)D^2$ times; in particular, this implies that l is reset to zero at most this many times.

We say a set x *respects* set y if either $x \subset y$ or $x \cap y = \varnothing$.

By definition of equivalence, and also because of the manner in which C is updated, the set $\{0 \leqslant i \leqslant |h| : h_i \equiv x\}$ respects $C(t)$ for any tests t and x, on each iteration. (That is, if $h_i \equiv h_j$ and i is removed from $C(t)$, then so is j.) Thus, $C(t)$ can be reduced in size at most $D-1$ times (and similarly for $K(i, j)$). Combined with the fact that $|T| \leqslant D$, this implies that the condition at line 6 is satisfied at most $D(D-1)$ times.

We show below that $l \leqslant D-1$ on each iteration. This will complete the theorem: Since l is reset to zero at most $(k+1)D^2$ times, the condition at

line 8 can be satisfied at most $(k+1)mD^2(D-1)^2$ times (where, as usual, m is the length of the longest counterexample so that each $m_i \leqslant m$). Also, since l is incremented at line 18, the conditions at lines 6 and 8 can fail to be satisfied at most $(k+1)D^2(D-1)$ times. This gives us an overall bound on the total number of iterations of the main loop, and, since at most $|h|+m+D-1$ actions are executed on each iteration, the result follows.

Thus, to complete the proof, we show that $l \leqslant D-1$ on each iteration.

LEMMA 4. $l \leqslant D-1$.

Proof. First, note that on each iteration, $K(i,j) \cap K(i,j+1) \neq \varnothing$ for $1 \leqslant i \leqslant l$ and $0 \leqslant j < m_i$. This follows from the fact that if the condition at line 24 is satisfied, then l is reset to zero.

We claim that on each iteration $K(i',j')$ respects $K(i,j)$ for $1 \leqslant i' < i \leqslant l$, $0 \leqslant j \leqslant m_i$, and $0 \leqslant j' \leqslant m_{i'}$. To see that this is so, note that $K(i,j)$ can only be reduced at lines 10, 22, and 23. If $K(i,j)$ is reduced at line 10, it must be that $K(i',j')$ is coherent since the *smallest* i for which $K(i,j)$ is incoherent is chosen at line 9. Thus, if one element of $K(i',j')$ is removed from $K(i,j)$, then so are all the others, and so $K(i',j')$ will respect $K(i,j)$ following an update at line 10. Similarly, if $K(i,j)$ is updated at lines 22 or 23, then it must be that $K(i',j')$ is coherent (since the condition at line 8 did not hold). Thus, as before, $K(i',j')$ respects $K(i,j)$ following an update at these lines. Therefore, as claimed, $K(i',j')$ respects $K(i,j)$ on each iteration for $i' < i$.

To prove $l \leqslant D-1$, we define a sequence of undirected graphs $G_0, \ldots, G_l$. The vertex set of each graph is the set $\{0, \ldots, |h|\}$. In G_i, an edge connects two vertices r and s if and only if $h_r \equiv h_s$ or $\{r,s\} \subset K(i',j)$ for some $1 \leqslant i' \leqslant i$, $0 \leqslant j \leqslant m_{i'}$.

We are interested in counting the number of connected components of each graph G_i. First, note that G_0 has at most D connected components by definition of diversity. We show below that each graph G_{i-1} has at least one more connected component than G_i. Since every (non-empty) graph has at least one connected component, this implies that $l \leqslant D-1$.

Since the edge set of G_{i-1} is a subset of the edge set of G_i, it suffices to find a single pair of vertices which are connected in G_i, but not in G_{i-1}.

As argued above in discussing the handling of counterexamples, the sets $K(i,0)$ and $K(i,m_i)$ are disjoint. Let r and s be respective members of these sets. Then r and s are connected in G_i because, as remarked above, $K(i,j) \cap K(i,j+1) \neq \varnothing$ on each iteration, for $0 \leqslant j < m_i$.

We claim that r and s are not connected in G_{i-1}. For if they were, then since r but not s is contained in $K(i,0)$, there must be adjacent vertices r' and s' on the path from r to s for which r' but not s' is contained in $K(i,0)$. Since r' and s' are adjacent, either $h_{r'} \equiv h_{s'}$ or $\{r',s'\} \subset K(i',j)$ for some

$i' < i$. However, as already argued, either case implies that $\{r', s'\}$ respects $K(i, 0)$, a contradiction.

This completes the proof of the lemma, as well as the proof of Theorem 3. ∎

So Theorem 3 shows that an effective diversity-based algorithm exists for inferring a finite-state environment, assuming a diversity-based homing sequence has been provided. We turn next to the problem of extending this algorithm to handle environments when such a sequence is not available.

5.2. *Constructing a Homing Sequence*

As was done in the algorithm for learning with a state-based representation (Section 4), we presume that some sequence h is a true diversity-based homing sequence until it becomes necessary to extend and improve h. Our algorithm constructs h in a manner similar to that outlined in Fig. 4. Initially, $h = \lambda$. If for some test x, candidate set $C(x)$ is reduced to the empty set, then clearly h cannot be a diversity-based homing sequence since this implies that hx is inequivalent to every prefix of h. We therefore replace h with hx as is done in Fig. 4. Since more equivalence classes are represented by the prefixes of hx than by those of h, it follows that h must converge to a correct homing sequence if extended in this fashion at most $D - 1$ times.

Our extended algorithm is quite similar to the one given in Fig. 8. As before, we maintain a set T and function r, which together record inequivalences determined among the tests. Now, however, the problem of determining that two tests are inequivalent becomes more difficult: we saw earlier that if h is a diversity-based homing sequence and $C(x) \cap C(y) = \varnothing$ for two tests x and y, then $x \not\equiv y$. However, if h is *not* a diversity-based homing sequence, this conclusion may be false since it may be that x and y are equivalent to one another, but that hx and hy are not equivalent to any prefix of h.

Nevertheless, we show that if x and y are in fact equivalent, then by re-running these tests repeatedly in an appropriate manner, we can with high probability eliminate all the elements of one of the candidate sets, thus yielding an extension to h as described above.

Suppose that, having executed h from state q, we find that $C(x)$ and $C(y)$ are coherent, and furthermore, that their selected values are different. If $x \equiv y$ then, by definition of equivalence, the true values of the two tests in the current state qh are equal. Thus, the selected value of one of the candidate sets must disagree with the common value of the two tests in the current state. If this is the case for x (say), and x is executed, then $C(x)$ will be reduced to the empty set (and hx can replace h). In general, if the

algorithm *randomly* chooses which of x or y to execute, then with probability 1/2, the candidate set of the chosen test is emptied. Of course, by repeating such an experiment many times, we can lift our confidence to arbitrarily high levels.

This then is the approach used by our extended algorithm (Fig. 9) in determining test inequivalence. The algorithm proceeds just as before. Now, however, when the candidate sets of two tests are found to be disjoint (line 27), the procedure does not immediately conclude that the tests are inequivalent. Rather, it keeps the two candidate sets around and, when given the opportunity, re-runs the two tests as described above. Only after the tests have been re-run many times with neither of the candidate sets emptying does the algorithm conclude that the tests are inequivalent.

THEOREM 5. *The algorithm of Fig. 9, with probability at least* $1-\delta$, *halts and outputs a perfect model. The algorithm's running time, and the number of actions executed are both bounded by*

$$O(kD^4(m+D)(mD+\log(kD/\delta))).$$

Also, the total number of counterexamples required by the algorithm is at most $O(kD^3)$.

Proof. The proof of this theorem is quite similar to the proof of Theorem 3. As before, we need only show that the algorithm halts in the stated number of steps since it only halts when a perfect model has been found.

As before, $i \notin C(t)$ only if $h_i \not\equiv ht$ for any test t and $0 \leqslant i \leqslant |h|$. Similarly for $K(i, j)$.

In the algorithm, the variable s_i serves two purposes: When the sequence of tests $u_{i0}, \ldots, u_{im_i}$ is first created (lines 21–22), s_i is set to -1. Variable s_i remains negative until the candidate sets of two consecutive tests in this sequence are reduced to the point that they are disjoint. At this point, s_i become a (non-negative) counter indicating how many times the tests u_{i0} and u_{i1} have been re-run without either candidate set emptying. It is easily verified that, on each iteration, $K(i, j) \cap K(i, j+1) \neq \varnothing$ for $0 \leqslant j < m_i$ if $s_i < 0$, and $m_i = 1$ and $K(i, 0) \cap K(i, 1) = \varnothing$ if $s_i \geqslant 0$. (This is because of the modifications to these data structures that take place at line 29.) Note that this implies at line 8 that $\bigcup_j K(i, j)$ is coherent if and only if every $K(i, j)$ is coherent and $K(i, 0)$ and $K(i, 1)$'s selected values agree.

The algorithm is randomized, and can only be shown to behave correctly when certain low probability events do not occur. Therefore, to simplify the analysis, we will assume that the algorithm has a *good run*—specifically, that if $u_{i0} \equiv u_{i1}$ then s_i does not exceed $\lg(1/\delta_0)$, for $1 \leqslant i \leqslant l$. Later, we will show that a good run occurs with probability at least $1-\delta$.

Input: access to $\mathcal{E}$, a finite-state automaton
D - the diversity of $\mathcal{E}$
δ - desired confidence
Output: a perfect model of $\mathcal{E}$
Procedure:
1 $h \leftarrow \lambda$
2 $\delta_0 \leftarrow \delta/((D-1)((k+1)D^2 + D - 1))$
3 initialize T, r, Υ, C and ℓ as in Figure 8 (lines 1–3)
4 **repeat**
5 execute h, producing output σ
6 **if** $C(t)$ is incoherent for some $t \in T$ **then**
7 execute t and update $C(t)$
8 **else if** $\bigcup_{j=0}^{m_i} K(i,j)$ is incoherent for some $1 \leq i \leq \ell$ **then**
9 choose the smallest i for which this is so
10 **if** $K(i,j)$ is incoherent for some $0 \leq j \leq m_i$ **then**
11 execute u_{ij} and update $K(i,j)$
12 **else** [$m_i = 1$, $K(i,0) \cap K(i,1) = \emptyset$ and $\sigma[K(i,0)] \neq \sigma[K(i,1)]$]
13 choose $j \in \{0,1\}$ randomly
14 execute u_{ij} and update $K(i,j)$
15 **if** $K(i,j) \neq \emptyset$ **then**
16 $s_i \leftarrow s_i + 1$
17 **if** $s_i > \lg(1/\delta_0)$ **then**
18 conclude $u_{i0} \not\equiv u_{i1}$: update r, T, C, Υ as in Figure 8 (lines 25–31)
19 $\ell \leftarrow 0$
20 **else**
21 make conjecture; handle returned counterexample as in Figure 8 (lines 12–23)
22 $s_\ell \leftarrow -1$
23 **if** $K(i,j) = \emptyset$ for some $1 \leq i \leq \ell$, $0 \leq j \leq m_i$ **then**
24 $h \leftarrow hu_{ij}$
25 $C(t) \leftarrow \{0, \ldots, |h|\}$ **for** $t \in T$
26 $\ell \leftarrow 0$
27 **else if** $K(i,j) \cap K(i,j+1) = \emptyset$ and $s_i < 0$ for some $1 \leq i \leq \ell$, $0 \leq j < m_i$ **then**
28 $s_i \leftarrow 0$
29 $u_{i0} \leftarrow u_{ij}$; $u_{i1} \leftarrow u_{i,j+1}$; $K(i,0) \leftarrow K(i,j)$; $K(i,1) \leftarrow K(i,j+1)$; $m_i \leftarrow 1$
30 **end**

FIG. 9. A diversity-based algorithm for inferring $\mathcal{E}$.

Assuming then that a good run occurs, it is clear that $t \notin r(x)$ only if $x \not\equiv t$ for $t \in T$, $x \in BT$. Thus, all pairs of tests in T are inequivalent, and $|T| \leq D$. Further, this shows that lines 18–19 are executed no more than $(k+1)D^2$ times.

As argued above, h cannot be extended more than $D-1$ times, implying that lines 24–26 are executed at most $D-1$ times. Thus, variable l is reset to zero no more than $R = (k+1)D^2 + D - 1$ times. Later we will again argue that $l \leq D-1$ on each iteration; assume for now that this is the case. Then since $s_i \leq \lg(1/\delta_0)$ on each iteration, lines 13–19 are executed at most $R(D-1)\lg(1/\delta_0)$ times.

It can be verified, as in Theorem 3, that the set $\{0 \leq i \leq |h|: h_i \equiv x\}$

respects $C(t)$ for any tests t and x, on each iteration (and likewise for $K(i, j)$). Applying our bound on l and on the number of times l is reset, this implies line 11 is executed at most $R(D-1)^2 m$ times, and so the condition at line 8 is satisfied at most $R(D-1)((D-1)m + \lg(1/\delta_0))$ times.

The sets $C(t)$ for $t \in T$ are reset to $\{0, ..., |h|\}$ at most $D-1$ times (i.e., only when h is extended). Thus, the condition at line 6 is satisfied at most $D(D-1)^2$ times.

Finally, since l is bounded and is incremented at line 21, the conditions at lines 6 and 8 fail to be satisfied at most $R(D-1)$ times, yielding the stated bound on the number of counterexamples needed. Thus, the number of iterations of the outer loop can be computed, and the bound on the number of actions executed follows from the fact that $|h| \leqslant (D-1)(m+D-1)$. Note that this also gives the stated bound on the number of counterexamples needed.

The proof that $l \leqslant D-1$ is quite similar to that given in Lemma 4. As before, we define graphs $G_0, ..., G_l$ on vertex set $\{0, ..., |h|\}$. We let $\{r, s\}$ be an edge of G_i if and only if $h_r \equiv h_s$ or $\{r, s\} \subset \bigcup_{j=0}^{m_{i'}} K(i', j)$ for some $1 \leqslant i' \leqslant i$. Then G_0 has at most D connected components. It can be argued as before that $\bigcup_j K(i', j)$ respects $K(i, j')$ if $i' < i$. Also, $K(i, 0)$ and $K(i, m_i)$ are disjoint. Therefore, if r is in $K(i, 0)$ and s is in $K(i, m_i)$ then r and s are connected in G_i but not in G_{i-1} by the argument given in the proof of Theorem 3. Thus, G_{i-1} has at least one more connected component than G_i, and $l \leqslant D-1$.

Thus, we have proved that the stated bound on the number of actions executed holds on a good run. It remains then only to show that a good run occurs with probability at least $1-\delta$.

As argued above, if $K(i, 0)$ and $K(i, 1)$ are coherent with different selected values, and if $u_{i0} \equiv u_{i1}$, then the probability is 1/2 that $K(i, j)$ is empty after u_{ij} is executed, for j chosen randomly from $\{0, 1\}$. Thus, if $u_{i0} \equiv u_{i1}$, then the probability that s_i exceeds k is less than 2^{-k}. In particular, s_i exceeds $\lg(1/\delta_0)$ with probability less than δ_0.

We argued above that, on a good run, lines 21–22 are executed at most $R(D-1)$ times; that is, at most this many pairs u_{i0}, u_{i1} are created. The chance that s_i exceeds $\lg(1/\delta_0)$ when $u_{i0} \equiv u_{i1}$ for any of these pairs is thus bounded by δ. Thus, δ bounds the probability of a bad run, completing the proof of the action execution bound.

It is clear that this algorithm runs in polynomial time. It is not so obvious, however, how it can be implemented to run in time proportional to the bound on the number of actions executed. We discuss techniques that can be used to achieve such a time bound.

Perhaps the most time consuming task performed by the algorithm is in checking the coherence of the many candidate sets. In a naive implementation, determining the coherence of a subset of $\{0, ..., |h|\}$ takes $O(|h|)$ time.

Thus, for instance, checking the $|T|$ candidate sets $C(t)$ at line 6 takes up to $O(D\,|h|)$ time; since only $O(|h|+D+m)$ actions are executed on each iteration, this gives a time bound that exceeds the action execution bound by at least a factor of D.

We show instead how the coherence of any candidate set can be checked in $O(D)$ time using a different representation: We maintain a partition π of the set $\{0, ..., |h|\}$ with the interpretation that i and j are in the same block of π if and only if every time that h was previously executed (line 5), it was observed that $\sigma_i = \sigma_j$ (where σ was the observed output sequence, as usual). In particular, if $h_i \equiv h_j$, then i and j are always in the same block of π. Thus $|\pi| \leqslant D$.

Note that if such a partition is maintained, then on each iteration, each block of π respects each candidate set $C(t)$ or $K(i, j)$. It therefore makes sense to represent each candidate set as a set of pointers to the blocks of π that it includes. If this is done, then each candidate set contains at most D pointers, and each set's coherence can be determined in $O(D)$ time (it is only necessary to examine the value of one member of each block since all the other members have the same value).

It is quite easy to see how the partition π can be maintained: Initially, and each time h is extended, π is set to $\{\{0, ..., |h|\}\}$. After h is executed with output σ at line 5, the coherence of each block of π is determined. Since each index $0, ..., |h|$ occurs in only one block of π, this only takes $O(|h|)$ time. If any block s is incoherent, then it is split into two new blocks $s \cap \sigma^{-1}(0)$ and $s \cap \sigma^{-1}(1)$. Since $|\pi| \leqslant D$, this can happen at most $D-1$ times. Naturally, when it does happen, all of the candidate sets must be changed so that their members point to blocks of the new partition. This takes $O(D)$ time for each of the $O(mD)$ candidate sets. Thus, since h can be extended at most $D-1$ times, the algorithm spends at most $O(mD^4)$ time updating candidate sets in this fashion. (This time is negligible compared to the number of actions executed.)

This still does not give the desired time bound because, even with this modification, naively computing l unions, each of up to m candidate sets as at line 8, can take $O(mD^2)$ time. Instead of the naive approach, we therefore maintain a counter $e(i, s)$ for each $1 \leqslant i \leqslant l$ and $s \in \pi$. This counter indicates the number of candidate sets $K(i, j)$ which include s: $e(i, s) = |\{0 \leqslant j \leqslant m_i : s \subset K(i, j)\}|$. It is straightforward how such a counter can be efficiently maintained, and the union $\bigcup_j K(i, j)$ can now be easily computed in $O(D)$ time as the union of those blocks $s \in \pi$ for which $e(i, s) > 0$.

Finally, lines 23 and 27, which appear to require a great deal of search, actually do not because only a small number of values i, j (those for which $K(i, j)$ was modified) need actually be checked.

With these ideas, it can now be fairly easily verified that the algorithm halts within the stated time bound. ▮

6. A State-Based Algorithm for Permutation Automata

In this and the next section, we present algorithms for inferring permutation automata. Unlike the procedures described up to this point, these procedures do *not* rely on a means of discovering counterexamples; the procedures actively experiment with the unknown environment, and output a perfect model with arbitrarily high probability.

As before, we describe both a state-based and a diversity-based procedure. In both cases, we describe deterministic procedures that, given a (diversity-based) homing sequence h, output a perfect model of the environment in time polynomial in n (or D) and $|h|$. To construct the needed homing sequence, we show that any sufficiently long random sequence of actions is likely to be a homing sequence.

We begin in this section with the state-based case. Consider first the simpler problem of inferring a *visible* automaton, i.e., one in which the identity of each state is readily observable. For instance, suppose each state, instead of outputting 0 or 1, outputs its own name. In this situation, inference of the automaton is almost trivial. From the current state q, we can immediately learn the value of $\delta(q, b)$ by simply executing b and observing the state reached. If $\delta(q, b)$ is already known for all the basic actions, then either we can find a path based on what is already known about δ to a state for which this is not the case, or we have finished exploring the automaton. It is not hard to see that $O(kn^2)$ actions are executed in total by this procedure.

Now suppose that the unknown environment $\mathcal{E}$ is a permutation automaton and that a homing sequence h has been provided. Because $\mathcal{E}$ is a permutation environment, we can easily show that h is also a *distinguishing sequence*; that is, h distinguishes every pair of unequal states of $\mathcal{E}$. Put another way, $q_1\langle h\rangle = q_2\langle h\rangle$ if and only if $q_1 = q_2$. (For if $q_1\langle h\rangle = q_2\langle h\rangle$ then, since h is a homing sequence, $q_1 h = q_2 h$. This implies $q_1 = q_2$ since $\mathcal{E}$ is a permutation environment.) Thus, the identity of any state is uniquely given by the output of h at that state; its identity is almost directly observable.

To infer the environment, we therefore use the inference procedure sketched above for visible automata. Each state q is named or represented by $q\langle h\rangle$, the output of h at that state. To identify the current state, simply execute h and observe the output produced.

Although executing h is helpful in identifying the state from which the sequence was executed, doing so is also likely to leave us in a state at the end of the sequence whose identity is unknown. This is a problem because the visible-automaton inference procedure requires that we be able to find a state whose identity is known even without executing h. We can overcome this problem, however, by maintaining a table u which records

the fact that if $\sigma = q\langle h\rangle$ was just observed as the output of executing h, then the output of h if executed from the current state qh is given by $u(\sigma)$.

Thus, we can reach a state whose identity is known (without executing h from it), we can execute an experiment as dictated by the visible-automaton inference procedure, and we can identify the last state reached by executing h. This can of course be repeated as many times as necessary.

Our procedure is given in Fig. 10. As mentioned, each state q is represented by $q\langle h\rangle$, the output of h at q. For $\sigma \in Q\langle h\rangle$, we write q_σ to denote that state for which $q_\sigma\langle h\rangle = \sigma$. This state is well-defined since h is a distinguishing sequence. A function or table $u: Q\langle h\rangle \to Q\langle h\rangle$ is maintained for which $u(\sigma) = q_\sigma h\langle h\rangle$. That is, if h was just executed with output σ, then the current state is $q_{u(\sigma)}$.

The transition function is represented by the program variable $d: Q\langle h\rangle \times B \to Q\langle h\rangle$. For notational purposes, the function d can be extended in the usual manner to the domain $Q\langle h\rangle \times A$. The variable d is used to store and compute the output of h in future states. Given $\sigma \in Q\langle h\rangle$ and $b \in B$, $d(\sigma, b)$ denotes the output of h in state $q_\sigma b$. That is, if properly constructed, $d(\sigma, b) = q_\sigma b\langle h\rangle$.

```
Input:  access to E, a permutation automaton
        h - homing sequence
Output: a perfect model of E
Procedure:
 1 d, u are initially undefined everywhere
 2 execute h, producing output σ
 3 repeat
 4     if u(σ) is not defined then
 5         execute h, producing output τ
 6         u(σ) ← τ
 7         σ ← τ
 8     else if (∃a ∈ A, b ∈ B) d(u(σ), a) is defined, but d(u(σ), ab) is undefined then
 9         choose the shortest such ab
10         α ← d(u(σ), a)
11         execute ab
12         execute h, producing output τ
13         d(α, b) ← τ
14         σ ← τ
15     else
16         exit loop
17 end
18 let q be the current state
19 output the following prediction rule (model of E):
20     on input a ∈ A,
21         α ← d(u(σ), a)
22         predict γ(qa) = α0    [where α0 is the first symbol of α]
```

FIG. 10. A state-based algorithm for inferring permutation environment $\mathcal{E}$.

THEOREM 6. *The algorithm of Fig.* 10 *halts and outputs a perfect model of* $\mathcal{E}$ *after executing at most* $O(kn(|h|+n))$ *actions, and in time* $O(kn(|h|+kn))$.

Proof. Clearly, $|Q\langle h\rangle| \leqslant n$, so that after at most $n+kn$ iterations, the procedure will halt since every entry of u and d will be defined.

We can view d as defining a directed graph whose vertices are the elements of $Q\langle h\rangle$, and whose edges are of the form $\sigma \to d(\sigma, b)$ whenever $\sigma \in Q\langle h\rangle$, $b \in B$, and $d(\sigma, b)$ is defined. Then the problem of finding an experiment ab as in the figure can be treated as that of finding a path in the graph from $u(\sigma)$ to another vertex α whose out-degree is less than k. This is easily done in $O(kn)$ time (for instance, using breadth-first search), and the resulting experiment ab has length at most n, the size of the graph. This proves the upper bound on the number of actions executed.

The remaining steps of the loop can be achieved in $O(|h|)$ time, for instance, if we store the elements of $Q\langle h\rangle$ at the leaves of a depth $(|h|+1)$ binary tree. It remains then only to show that the prediction rule output by the algorithm is a perfect model of $\mathcal{E}$.

We prove this by showing that the following invariants hold between each iteration of the main loop:

1. If $\sigma \in Q\langle h\rangle$ and $u(\sigma)$ is defined, then $u(\sigma) = q_\sigma h\langle h\rangle$.
2. If $\sigma \in Q\langle h\rangle$, $b \in B$ and $d(\sigma, b)$ is defined then $d(\sigma, b) = q_\sigma b\langle h\rangle$.

Initially, these invariants hold vacuously since u and d are undefined everywhere. Suppose at the top of an iteration of the loop that h was just executed from some state q with output σ. Then $q = q_\sigma$, and the current state is $q_\sigma h$. If $u(\sigma)$ is undefined, then h is executed from the current state with output τ. Thus, we learn that $\tau = q_\sigma h\langle h\rangle$. Setting $u(\sigma)$ to τ, invariant 1 is maintained.

On the other hand, if $u(\sigma)$ is defined, then the current state is $q_{u(\sigma)}$. If an experiment ab is found as shown in the figure, then invariant 2, together with an easy induction argument on the length of a, shows that $\alpha = d(u(\sigma), a) = q_{u(\sigma)} a\langle h\rangle$. The state we reach by executing a then is just q_α. Executing b, and then h with output τ, we learn that $\tau = q_\alpha b\langle h\rangle$. Setting $d(\alpha, b) = \tau$, invariant 2 is maintained.

With these invariants, it is not hard to see why, after the loop is exited, the output prediction rule is correct. The current state q is just $q_{u(\sigma)}$ as before. Given $a \in A$, we have $qa\langle h\rangle = d(u(\sigma), a)$. Therefore, the first element of $d(u(\sigma), a)$ is $\gamma(qa)$. ∎

Finally, we must consider how to construct h. In fact, any sufficiently long random sequence of actions is likely to be a homing sequence:

THEOREM 7. *Let $\delta > 0$, and let h be a random sequence of length $8kn^5 \cdot \ln(n) \cdot (n + \ln(1/\delta))$. Then h is a homing sequence with probability at least $1 - \delta$.*

Proof. The idea is to randomly construct the homing sequence in the manner described in Fig. 3. On each iteration, an appropriate extension x which distinguishes some pair of states as needed by the algorithm is likely to be given by any sufficiently long random walk. This follows from previous results on random walks in permutation automata. Specifically, we use the following result:

LEMMA 8. *Let q_1 and q_2 be two distinct states and let x be a random sequence of length $2kn^4 \ln(n)$ of the following form: At each step, with equal probability, we either do nothing, or we execute a uniformly and randomly chosen basic action from B. Then the probability that $\gamma(q_1 x) \neq \gamma(q_2 x)$ is at least $1/2n$.*

Essentially, the same result is proved in Schapire's master's thesis (1988) using results of Fiedler (1972) on the eigenvalues of doubly stochastic matrices, in addition to certain properties of point-symmetric graphs. There, the result was proved for update graphs, but, because of the "dual" relationship between update graphs and finite automata, the results hold as stated as well.

Let $x_1, \ldots, x_r$ be a sequence of random strings, each of length $2kn^4 \ln(n)$. Let $y_i = x_1 x_2 \cdots x_i$. We wish to show that y_r is a homing sequence with high probability. Consider a sequence of trials in which "success" on the ith trial means that either y_{i-1} is a homing sequence (so that y_i is as well) or $|Q\langle y_i \rangle| > |Q\langle y_{i-1} \rangle|$. Clearly, if n of the trials succeed, then y_r is a homing sequence.

For any choice of y_{i-1}, Lemma 8 shows that the probability of success on the ith trial is at least $1/2n$. To probabilistically lower bound the total number of successes, we use the following form of Chernoff bounds due to Angluin and Valiant (1979):

LEMMA 9. *Let $X_1, \ldots, X_m$ be a sequence of m independent Bernoulli trials, each succeeding with probability p so that $\mathbf{E}[X_i] = p$. Let $S = X_1 + \cdots + X_m$ be the random variable describing the total number of successes. Then for $0 \leqslant \gamma \leqslant 1$, the following hold:*

- $\mathbf{Pr}[S > (1+\gamma)\, pm] \leqslant e^{-\gamma^2 mp/3}$, *and*
- $\mathbf{Pr}[S < (1-\gamma)\, pm] \leqslant e^{-\gamma^2 mp/2}$.

Thus, applying this lemma, we see that the probability of fewer than n successes in r trials is at most δ if $r \geqslant 4n(n + \ln(1/\delta))$. This proves the theorem. ∎

These theorems give our inference procedure a running time of $O(k^2n^6 \log(n) \cdot (n + \log(1/\delta)))$.

7. A Diversity-Based Algorithm for Permutation Automata

We can show in a similar manner how a permutation environment can be inferred using a diversity-based representation. As before, we reduce the problem to that of inferring a visible automaton—in this case, one for which all of the test-equivalence classes are known, and for which the value of each test class is observable in every state. The problem of inferring such automata is solved in Chapter 4 of Schapire's master's thesis (1988); the solution is based on the careful planning of experiments, and on the maintenance of candidate sets similar to those described in Section 5.

Let h be a given diversity-based homing sequence for the unknown permutation environment $\mathcal{E}$. As before, to simulate the inference algorithm for visible automata, it suffices to show that the state of the automaton (i.e., the values of the test classes) can be observed by executing h, and further that it is possible to reach a state whose identity is known even without executing h. Since $\mathcal{E}$ is a permutation environment, we can show that every test class is represented by some prefix of h. Therefore, at the current state q, the values of all the test classes can be observed simply by executing h.

If, having executed h from some state q, we find that candidate set $C(h_i)$ is coherent, then the value of test h_i in the current state qh is just the selected value of $C(h_i)$. (As before, h_i is the prefix of h of length i.) Thus, if all the candidate sets are coherent, then $qh\langle h\rangle$, the output of the entire sequence, is known in the current state. On the other hand, if one of the candidate sets is incoherent, then by re-executing h we are guaranteed to reduce one of the candidate sets. Thus, we can quickly reach a state in which the output of h is known without actually executing it.

We say action sequence a is a *diversity-based distinguishing sequence* if every test is equivalent to some prefix of a. Such a sequence is clearly a distinguishing sequence, since if $q_1 \neq q_2$ then there exists a test t distinguishing the two states; since $t \equiv p$ for some prefix p of a, $\gamma(q_1 p) \neq \gamma(q_2 p)$ and so $q_1\langle a\rangle \neq q_2\langle a\rangle$.

A diversity-based distinguishing sequence is also a diversity-based homing sequence, as is obvious from their definitions. In permutation environments (but not in general), the converse holds: Suppose h is a diversity-based homing sequence. Let $[t_1]$, $[t_2]$, ..., $[t_D]$ be the equivalence classes of $\mathcal{E}$. Then there exist prefixes p_1, p_2, ..., p_D of h such that $p_i \equiv ht_i$. Since $\mathcal{E}$ is a permutation environment, if $t_i \not\equiv t_j$ then $ht_i \not\equiv ht_j$. Therefore, the D prefixes p_i are pairwise inequivalent, and so every equivalence class is

represented by some prefix of h. Thus, h is a diversity-based distinguishing sequence.

As in the last section, we assume a diversity-based homing sequence h has been given, and show later how such a sequence can be randomly constructed.

Our procedure is given in Fig. 11. As mentioned above, the algorithm maintains various kinds of candidate sets. First, for each $0 \leqslant i \leqslant |h|$, a set $G(i)$ is maintained with the interpretation that j is in $G(i)$ if h_j could plausibly be equivalent to hh_i, i.e., if it has not yet been determined that $h_j \not\equiv hh_i$. (Thus, $G(i) = C(h_i)$ in the notation of Section 5.) As described above, such candidate sets are useful for reaching a state in which the output of h is known prior to its execution from that state.

The algorithm also maintains sets $U(i, b)$ for $0 \leqslant i \leqslant |h|$ and $b \in B$; these sets consist of indices j for which h_j is plausibly equivalent to bh_i. To see why such sets might be useful, suppose h has been executed from some state q with output σ. As seen before, if $G(i)$ is coherent for all i, then $\beta = qh\langle h \rangle$ is known, the output of h if executed from the current state qh. If, moreover, $U(i, b)$ is coherent with respect to β, then $\gamma(qhbh_i)$ is known;

Input: access to $\mathcal{E}$, a permutation automaton
h - a diversity-based homing sequence
Output: a perfect model of $\mathcal{E}$
Procedure:
1 $G(i), U(i, b) \leftarrow \{0, \ldots, |h|\}$ **for** $i \in \{0, \ldots, |h|\}, b \in B$
2 execute h, producing output σ
3 **repeat**
4 **if** $G(i)$ is incoherent for some $0 \leq i \leq |h|$ **then**
5 execute h, producing output τ
6 $G(i) \leftarrow G(i) \cap \sigma^{-1}(\tau_i)$ **for** $i \in \{0, \ldots, |h|\}$
7 $\sigma \leftarrow \tau$
8 **else**
9 $\beta_i \leftarrow \sigma[G(i)]$ **for** $i \in \{0, \ldots, |h|\}$
10 **if** PLAN-EXP can find a shortest useful experiment ab **then**
11 $\alpha_i \leftarrow \beta[U(i, a)]$ **for** $i \in \{0, \ldots, |h|\}$
12 execute ab
13 execute h, producing output τ
14 $U(i, b) \leftarrow U(i, b) \cap \alpha^{-1}(\tau_i)$ **for** $i \in \{0, \ldots, |h|\}$
15 $\sigma \leftarrow \tau$
16 **else**
17 exit loop
20 **end**
21 let q be the current state
22 $\beta_i \leftarrow \sigma[G(i)]$ **for** $i \in \{0, \ldots, |h|\}$
23 **output** the following prediction rule (model of $\mathcal{E}$):
24 on input $a \in A$, predict $\gamma(qa) = \beta[U(0, a)]$

FIG. 11. A diversity-based algorithm for inferring permutation environment $\mathcal{E}$.

thus, if this is the case for all i, then $qhb\langle h\rangle$ can be determined, the output of h from the state reached if b were executed.

The function U can be extended in a natural manner to the domain $\{0, ..., |h|\} \times A$ by the rule $U(i, \lambda) = \{i\}$ and $U(i, ab) = \bigcup_{j \in U(i,a)} U(j, b)$ for $i \in \{0, ..., |h|\}$, $a \in A$ and $b \in B$. Then the above statements also hold if b is replaced by any action $a \in A$.

Our algorithm works by trying to reduce the candidate sets $U(i, b)$ as much as possible until $U(i, a)$ is coherent for all i and all $a \in A$; at this point, from the preceding comments, a perfect model has been attained.

Let σ, β, and q be as above, assuming all $G(i)$'s are coherent with respect to σ. If $U(i, b)$ is incoherent (with respect to β), then executing b and then h will clearly cause some candidate set $U(i, b)$ to shrink. In this case, b is called an *immediately useful experiment*. However, it may be the case that there is no immediately useful experiment (all the sets $U(i, b)$ are coherent) but, nevertheless, some set $U(i, a)$ is incoherent for $a \in A$ so that a perfect model has not been achieved. In this case, it is possible to find a *useful experiment*; this is an experiment in which a "set-up" action $a \in A$ is first executed, leading to a state in which an immediately useful experiment can be executed.

More precisely, a sequence ab, where $a \in A$ and $b \in B$, is a *useful experiment* if, for some $0 \leqslant i \leqslant |h|$, $U(i, ab)$ is incoherent, but $U(j, a)$ is coherent for $j \in U(i, b)$. Note that the *shortest* useful experiment has the additional property that $U(j, a)$ is coherent for all j, $0 \leqslant j \leqslant |h|$ (otherwise, a prefix of a would be a shorter useful experiment). A procedure for finding a shortest useful experiment, called PLAN-EXP, was described in Schapire's master's thesis (1988), and is treated here as a "black-box" subroutine. (The inputs required by PLAN-EXP are omitted from Fig. 11, but are described fully below.)

Thus, at a high level, our algorithm is simple: execute h; if some $G(i)$ is incoherent, then re-execute h and update G; otherwise, find and execute a shortest useful experiment, and update U. If no useful experiment exists, then a perfect model has been found.

Below, $\alpha(\cdot, \cdot)$ is an inverse of Ackermann's function (Tarjan, 1975); α is an extremely slow growing function.

THEOREM 10. *The algorithm described in Fig.* 11 *halts and outputs a perfect model of* $\mathcal{E}$ *after executing at most* $O(kD(|h| + D))$ *actions, and in time* $O(kD(|h| + D^2 + kD \cdot \alpha(kD, D)))$.

Proof. First, note that because of the manner in which G is updated, an index j is removed from $G(i)$ only if $h_j \not\equiv hh_i$. Thus, since h is a diversity-based homing sequence, $G(i)$ is never empty. Also, if j is removed from $G(i)$, then every other index j' for which $h_j \equiv h_{j'}$ must also be removed since equivalent tests have the same value in every state.

In addition, every index j appears in some set $G(i)$; i.e., $\bigcup_i G(i) = \{0, ..., |h|\}$. To see that this is so, note that, because h is a diversity-based distinguishing sequence, every equivalence class is represented by the prefixes of h, that is, $|\{[h_i]: 0 \leq i \leq |h|\}| = D$. Since $\mathcal{E}$ is a permutation environment, $h_i \equiv h_j$ if and only if $hh_i \equiv hh_j$. Thus, $|\{[hh_i]: 0 \leq i \leq |h|\}| = D$. Therefore, the test h_j is equivalent to some hh_i, implying $j \in G(i)$.

For the analysis, it is important to note that the set $\{G(i): 0 \leq i \leq |h|\}$ is a partition of $\{0, ..., |h|\}$. This can be proved by an inductive argument. For suppose, prior to the execution of line 6, that $G(i)$ and $G(j)$ are equal or disjoint, for some i, j. Then if $G(i) = G(j)$ and $\tau_i = \tau_j$, then clearly $G(i) \cap \sigma^{-1}(\tau_i) = G(j) \cap \sigma^{-1}(\tau_j)$. On the other hand, if $G(i)$ and $G(j)$ are disjoint or if $\tau_i \neq \tau_j$, then $G(i) \cap \sigma^{-1}(\tau_i)$ and $G(j) \cap \sigma^{-1}(\tau_j)$ must also be disjoint. In either case, the new sets following the execution of line 6 will be equal or disjoint.

Thus, since the set $\{0 \leq i \leq |h|: h_i \equiv x\}$ respects $G(i)$ for any test x on each iteration, it follows that $|\{G(i): 0 \leq i \leq |h|\}| \leq D$ on each iteration. Since, some $G(i)$ shrinks each time that lines 5–7 are executed, it follows that this block is executed at most $D - 1$ times.

We would like to give a similar argument showing that lines 9–17 are executed at most $k(D-1)$ times. We first give an inductive proof that $j \notin U(i, b)$ only if $h_j \not\equiv bh_i$:

Suppose that h has been executed from q with output σ. Suppose also that each $G(i)$ is coherent with respect to σ. Then there is some j for which $h_j \equiv hh_i$, and which is therefore in $G(i)$. Thus $\gamma(qhh_i) = \gamma(qh_j) = \sigma_j = \sigma[G(i)]$, and so $qh\langle h\rangle = \beta$ where $\beta_i = \sigma[G(i)]$, as in the figure.

Suppose that PLAN-EXP returns an experiment ab. Then $U(i, a)$ is coherent (with respect to β) for all $0 \leq i \leq |h|$, but, for some i, $U(i, ab)$ is not. Since h is a diversity-based distinguishing sequence, there exists j for which $h_j \equiv ah_i$. By inductive hypothesis, $j \in U(i, a)$. Since $U(i, a)$ is coherent, we have $\alpha_i = \beta[U(i, a)] = \gamma(qhh_j) = \gamma(qhah_i)$. Thus, $\alpha = qha\langle h\rangle$.

It can now be verified that j is removed from $U(i, b)$ only if $h_j \not\equiv bh_i$, completing the induction. As before, this implies that each $U(i, b)$ is non-empty on each iteration, and that $\bigcup_i U(i, b) = \{0, ..., |h|\}$ on each iteration for each $b \in B$. Also, having argued that $\alpha = qha\langle h\rangle$ at this point in the program, it can now be argued as before that the set $\{i: h_i \equiv x\}$ respects each $U(i, b)$, and that the set $\{U(i, b): 0 \leq i \leq |h|\}$ is a partition consisting of at most D blocks for each $b \in B$.

Since ab is a useful experiment, some set $U(i, b)$ must shrink at line 14. Thus, by the preceding arguments, lines 9–17 are executed at most $k(D-1)$ times.

We argue later that the returned useful experiment has length at most D. This will then complete the proof of the action execution bound.

Given the above arguments, it is quite easy to prove the correctness of

the output prediction rule: On exiting the main loop, each set $G(i)$ or $U(i, a)$ is coherent (with respect to σ and β, respectively, as in the figure) for all i and $a \in A$. As argued above, in the current state q, this implies that $q\langle h\rangle = \beta$. Also, given $a \in A$, we showed above that $\gamma(qah_i) = \beta[U(i, a)]$. Thus, $\gamma(qa) = \beta[U(0, a)]$, and the output rule is a perfect model.

Finally, we turn to efficiency considerations. If naively implemented, the running time of the procedure may be quite poor. However, using similar techniques to those described in Section 5, we can derive a time bound comparable to the action execution bound.

In particular, we maintain a partition π over the set $\{0, ..., |h|\}$ with the condition that i and j belong to the same block of π if and only if the values of h_i and h_j have never differed on any execution of h (so that the two tests are plausibly equivalent). As before, if $h_i \equiv h_j$, then i and j must be in the same block of π. Thus, $|\pi| \leq D$.

It is easily verified that, on each iteration, if i and i' are in the same block of π, then $G(i) = G(i')$, and $\{i, i'\}$ respects each set $G(j)$. Similarly, for $b \in B$, $U(i, b) = U(i', b)$ and $\{i, i'\}$ respects each set $U(j, b)$. Thus, with respect to the data structures G and U, the two indices i and i' are entirely indistinguishable. Therefore, we can represent these structures more efficiently in terms of the blocks of π.

In particular, as was done in Section 5, we can represent each candidate set as a list of pointers to those blocks of π which it includes. Thus, the representation of such a set has size at most D. Also, since $G(i) = G(j)$ if i and j are in the same block, we only need maintain a candidate set for a single member of each block (say, the minimum element). That is, we maintain a candidate set $G(i)$ or $U(i, b)$ (explicitly represented as described above) if and only if i is the smallest member of its block; the other candidate sets are only implicitly maintained, based on the equalities among candidate sets described above.

With such a representation, lines 4, 6, and 14 take only time $O(D^2)$. Using the fact (to be proved) that $|ab| \leq D$, we can also show that line 11 takes time $O(D^2 + |h|)$: computing $\alpha_i = \beta[U(i, a)]$ for a single value of i takes $O(D)$ time since $|a|$ is bounded, and since $U(i, a)$ is known to be coherent. Thus, computing α_i for each $i \in \{\min(s): s \in \pi\}$ takes $O(D^2)$ time. Finally, all the other values of α_i can be computed by setting $\alpha_i \leftarrow \alpha_{\min(s)}$ for $s \in \pi$, $i \in s$ in $O(|h|)$ time.

The partition π is easily maintained in the same manner described in Section 5: Each time that h is executed, the coherence of each block of π is checked in $O(|h|)$ time. If any block is incoherent, then the structures G and U must be updated; this takes $O(kD^2)$ time. Since π can be partitioned at most D times, this adds $O(kD^3)$ to the total running time of the procedure.

It remains then only to show how the running time of PLAN-EXP can

be bounded. The procedure PLAN-EXP takes as input a set V of variables; a set of candidate sets for each $v \in V$, $b \in B$; and an assignment to the variables in V. It returns a shortest useful experiment ab (or reports that none exists) in time $O(k|V| \cdot \alpha(k|V|, |V|))$. The length of the returned experiment ab is bounded by $|V|$.

Thus, if we use $\{0, ..., |h|\}$ as our variable set in our call to PLAN-EXP, then the procedure may take too long, and could plausibly return an experiment far longer than D. Instead, we will use the blocks of π as our variable set. The candidate sets are then defined naturally by the rule $U'(s, b) = \{s' \in \pi: s' \subset U(\min(s), b)\}$ for $s \in \pi$ and $b \in B$. The assignment $\beta'(s)$ is similarly defined to be $\beta(\min(s))$.

Note that our representation scheme for U is essentially equivalent to the structure U', and the structure β' is easily computed in $O(D)$ time. Also, since $|\pi| \leqslant D$, PLAN-EXP runs in time $O(kD \cdot \alpha(kD, D))$, and returns an experiment of length at most D.

It can be argued by induction on the length of a that $U'(s, a) = \{s' \in \pi: s' \subset U(i, a)\}$ for $s \in \pi$ and $a \in A$, assuming $i \in s$. With this fact, it can be seen that $U'(s, a)$ is coherent with respect to β' if and only if $U(i, a)$ is coherent with respect to β.

In particular, this shows that if PLAN-EXP when called in this manner returns an experiment ab, then $U(i, a)$ is coherent (with respect to β) for all $0 \leqslant i \leqslant |h|$, but, for some i, $U(i, ab)$ is not; that is, ab is indeed a shortest useful experiment. Likewise, if PLAN-EXP fails to find a useful experiment, then each $U(i, a)$ is coherent for all i and all $a \in A$.

This completes the proof. ∎

As in the state-based case, we can construct a diversity-based homing sequence by choosing a sufficiently long sequence of actions. Below, $H_n = \sum_{i=1}^{n} (1/i)$ is the nth harmonic number. It is well known that $H_n = \theta(\log n)$.

THEOREM 11. *Let $\delta > 0$, and let h be a random sequence of length $2kD^3H_D \cdot \ln(D) \cdot \ln(D/\delta)$. Then h is a diversity-based homing sequence with probability at least $1 - \delta$.*

Proof. We follow the algorithm of Fig. 4 for constructing a diversity-based homing sequence. On each iteration, we need to find an extension x to h for which hx is inequivalent to every prefix of h. That is, if v equivalence classes are represented by the prefixes of h, and $[t_1]$, $[t_2]$, ..., $[t_{D-v}]$ are the equivalence classes *not* represented, then we wish to find x such that $hx \equiv t_i$ for some i. Equivalently, we want $x \equiv h^{-1}t_i$. (Here, h^{-1} is a sequence of actions for which $h^{-1}h$ is the "identity" action; i.e., $qh^{-1}h = q$ for all $q \in Q$. The existence of h^{-1} is guaranteed by the fact that $\mathscr{E}$ is a permutation environment.)

Based on the results on random walks given in Schapire's master's thesis (1988), it is easy to conclude the following:

LEMMA 12. *Let t be any test, and let x be a random sequence of length $kD^2 \ln(D)$ of the following form: At each step, with equal probability, either we do nothing, or we execute a uniformly and randomly chosen basic action from B. Then the probability that $t \equiv x$ is at least $1/2D$.*

Thus, the probability that an extension x as described above is equivalent to any $h^{-1}t_i$ is at least $(D-v)/2D$. Extending h in this manner $(2D/(D-v)) \cdot \ln(1/\delta)$ times gives a probability of at least $1-\delta$ of successfully increasing the number of equivalence classes represented by the prefixes of h. Replacing δ with δ/D, we can conclude that h is a homing sequence with probability at least $1-\delta$ if its length is at least

$$\sum_{v=0}^{D-1} kD^2 \ln(D) \cdot \frac{2D}{D-v} \cdot \ln(D/\delta)$$

as claimed. (This sequence may be longer than strictly necessary since v may increase by more than one with each extension; also, many of the "actions" required by the lemma are actually "no-ops." This, however, does not affect the argument since a homing sequence remains one even if suffixed or prefixed.) ▮

Thus, our inference procedure runs in time $O(k^2D^4 \log^2(D) \cdot \log(D/\delta))$. This improves our previously best-known bound (Rivest and Schapire, 1987; Schapire, 1988) of $O(k^2D^7 \log(D) \cdot \log(kD/\delta))$ by roughly a factor of $D^3/\log(D)$.

8. EXPERIMENTAL RESULTS

The algorithm described in Section 4 has been implemented and tested on several simple robot environments.

In the "Random Graph" environment, the robot is placed on a randomly generated directed graph. The graph has n vertices, and each vertex has one out-going edge labeled with each of the k basic actions. For each vertex i, one edge (chosen at random) is directed to vertex $i+1 \bmod n$; this ensures that the graph contains a Hamiltonian cycle, and so is strongly connected. The other edges point to randomly chosen vertices, and the output of each vertex is also chosen at random.

In the "Knight Moves" environment, the robot is placed on a square checker-board, and can make any of the legal moves of a chess knight. However, if the robot attempts to move off the board, its action fails and no movement occurs. The robot can only sense the color of the square it

occupies. Thus, when away from the walls, every action simply inverts the robot's current sensation: any move from a white square takes the robot to a black square, and vice versa. This makes it difficult for the robot to orient itself in this environment.

Finally, in the "Crossword Puzzle" environment, the robot is on a crossword puzzle grid such as the one in Fig. 12. The robot has three actions available to it: it can step ahead one square, or it can turn left or right by 90°. The robot can only occupy the white squares of the crossword puzzle; an attempt to move onto a black square is a "no-op." Attempting to step beyond the boundaries of the puzzle is also a no-op. Each of the four "walls" of the puzzle has been painted a different color. The robot looks as far ahead as possible in the direction it faces: if its view is obstructed by a black square, then it sees "black"; otherwise, it sees the color of the wall it is facing. Thus, the robot has five possible sensations. Since this environment is essentially a maze, it may contain regions which are difficult to reach or difficult to get out of.

In the current implementation, we have used an adaptive homing sequence or homing tree as described in Section 4.4. We have also used the modified version of L^* described in Section 4.5. Finally, we have implemented a heuristic that attempts to focus effort on copies of L^* that have already made the most progress: if the homing sequence is executed and the L^* copy reached is not very far along, then the procedure is likely to

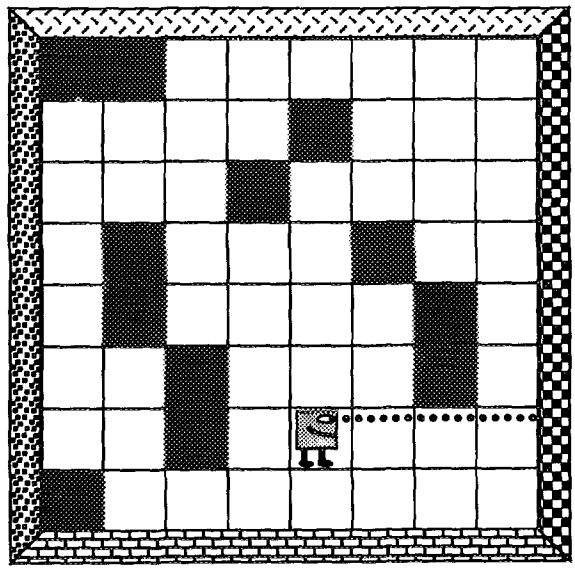

FIG. 12. A crossword puzzle environment.

re-execute the homing sequence to find one that is closer to completion. The idea of the heuristic is not to waste time on copies that have a long way to go. The heuristic seems to improve the running time for these three environments by as much as a factor of six.

For the "Random Graph" and "Crossword Puzzle" environments, the inference procedure was provided in some experiments with an oracle which would return the shortest counterexample to an incorrect conjecture. All three environments were also tested with no external source of counterexamples; to find a counterexample, the robot would instead execute random actions until its model of the environment made an incorrect prediction of the output of some state.

Table 1 summarizes how our procedure handled each environment. In the table, "Source" refers to the robot's source of counterexamples: "S" indicates that the robot had access to the shortest counterexample, and "R" indicates that it had to rely on random walks. The column labeled "$|\text{ran}(\gamma)|$" gives the number of possible sensations which might be experienced by the robot. (Extending our algorithms to the case that the range of γ consists of more than two elements is trivial.) "Copies" is the number of copies of L^* which were active when a correct conjecture was

TABLE 1

Experimental Results

Environment	Size	n	k	$\lvert\text{ran}(\gamma)\rvert$	Source	Copies	Queries	Actions	Time
Random graph	25	25	3	2	S	20	1,108	10,504	:01.0
					R	21	1,670	17,901	:01.2
	50	50	3	2	S	37	5,251	69,861	:06.0
					R	33	4,581	61,325	:03.6
	100	100	3	2	S	68	14,788	279,276	:24.1
					R	64	17,221	342,450	:18.1
	200	200	3	2	S	137	34,182	1,100,244	1:31.9
					R	136	29,796	1,012,279	:47.5
	400	400	3	2	S	275	72,027	3,010,377	4:52.0
					R	258	33,388	1,757,720	1:19.5
Knight moves	4	16	8	2	R	10	2,082	19,621	:01.4
	8	64	8	2	R	50	17,818	385,678	:19.4
	12	144	8	2	R	88	22,208	780,595	:36.3
	16	256	8	2	R	124	63,476	3,855,520	2:41.9
	20	400	8	2	R	157	129,407	8,329,257	5:58.9
Crossword puzzle	4	48	3	5	S	41	2,424	30,285	:02.5
					R	41	2,817	55,749	:04.1
	8	208	3	5	S	97	18,523	839,087	:52.9
					R	104	16,643	1,049,466	:51.0
	12	416	3	5	S	188	68,793	5,564,299	5:15.6
					R	193	58,222	8,850,079	7:12.5

made, "Queries" is the total number of membership and equivalence queries which were simulated, "Actions" is the total number of actions executed by the robot, and "Time" is elapsed cpu time in minutes and seconds. The procedure was implemented in C on a DEC MicroVax III. For example, inferring the 8×8 "Knight Moves" environment using randomly generated counterexamples required about 400,000 moves and 19 seconds of cpu time.

Note that for the "Random Graph" environment, the learning procedure sometimes did better with randomly generated counterexamples than with an oracle providing the shortest counterexample. It is not clear why this is so, although it seems plausible that in some way the random walk sequences give more information about the environment. For example, the counterexamples often become subsequences of the homing sequence, and it may be that random walk counterexamples make for better, more distinguishing homing sequences.

In sum, the running times given are quite fast, and the number of moves taken is far less than allowed for by the theoretical worst-case bounds. Nevertheless, it is also true that the number of actions executed is still somewhat large, much too great to be practical for a real robot. There are probably many ways in which our algorithm might be improved—both in a theoretical sense, and in terms of heuristics which might improve the performance in practice. We leave these questions as open problems.

9. Conclusions and Open Questions

We have shown how to infer an unknown automaton, in the absence of a reset, by experimentation and with counterexamples. For the class of permutation automata, we have shown that the source of counterexamples is unnecessary. We have described polynomial-time algorithms which are both state-based and diversity-based.

As discussed in the introduction, these results represent only modest progress toward our ultimate goal, the development of a robot capable of inferring a usable model of its real-world environment. It is not clear how to get there from where we are now. To begin with, we need algorithms that are even more efficient than the ones described here. Perhaps more importantly, we need techniques for handling more realistic environments. These would include environments with infinitely many states, and also environments exhibiting various kinds of randomness or uncertainty. Some progress on this latter problem has recently been made by Dean *et al.* (1992) who extended some of the results described in this paper to handle automata with stochastic output functions.

For truly realistic environments, inference of a perfect model will almost

certainly be out of the question. What then is the best we can hope for? What are the skills most needed for the robot to function in its environment, and how can those skills be learned?

ACKNOWLEDGMENTS

This research was supported by ARO Grant DAAL03-86-K-0171, DARPA Contract N00014-89-J-1988, NSF Grants CCR-8914428 and DCR-8607494, and a grant from the Siemens Corporation. In addition, part of this research was done while the second author was visiting GTE Laboratories in Waltham, Massachusetts.

RECEIVED March 14, 1991; FINAL MANUSCRIPT RECEIVED August 7, 1992

REFERENCES

ANGLUIN, D. (1978), On the complexity of minimum inference of regular sets, *Inform. and Control* **39**, 337–350.

ANGLUIN, D. (1981), A note on the number of queries needed to identify regular languages, *Inform. and Control* **51**, 76–87.

ANGLUIN, D. (1987), Learning regular sets from queries and counterexamples, *Inform. Comput.* **75**, 87–106.

ANGLUIN, D., AND VALIANT, L. G. (1979), Fast probabilistic algorithms for Hamiltonian circuits and matchings, *J. Comput. System Sci.* **18**, 155–193.

DEAN, T., ANGLUIN, D., BASYE, K., ENGELSON, S., KAELBLING, L., KOKKEVIS, E., AND MARON, O. (1992), Inferring finite automata with stochastic output functions and an application to map learning, *in* "Proceedings of the Tenth National Conference on Artificial Intelligence."

DRESCHER, G. L. (1986), "Genetic AI—Translating Piaget into Lisp," Technical Report 890, MIT Artificial Intelligence Laboratory.

FIEDLER, M. (1972), Bounds for eigenvalues of doubly stochastic matrices, *Linear Algebra Appl.* **5**, 299–310.

GOLD, E. M. (1972), System identification via state characterization, *Automatica* **8**, 621–636.

GOLD, E. M. (1978), Complexity of automaton identification from given data, *Inform. and Control* **37**, 302–320.

KEARNS, M., AND VALIANT, L. G. (1989), Cryptographic limitations on learning Boolean formulae and finite automata, *in* "Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing," pp. 433–444.

KOHAVI, Z. (1978), "Switching and Finite Automata Theory," second ed., McGraw–Hill, Englewood Cliffs, NJ.

KUIPERS, B. J., AND BYUN, Y.-T. (1988), A robust, qualitative approach to a spatial learning mobile robot, *in* "SPIE Advances in Intelligent Robotics Systems."

MATARIC, M. J. (1990), "A Distributed Model for Mobile Robot Environment-Learning and Navigation," Master's thesis, Massachusetts Institute of Technology, Technical Report AI-TR 1228, MIT Artificial Intelligence Laboratory.

PITT, L. (1989), "Inductive Inference, DFAs, and Computational Complexity," Technical Report UIUCDCS-R-89-1530, University of Illinois at Urbana–Champaign, Department of Computer Science; also appears *in* "Proceedings of the 1989 International Workshop on

Analogical and Inductive Inference," Lecture Notes in Computer Science, Springer-Verlag, Berlin/New York.

PITT, L., AND WARMUTH, M. K. (1990), Prediction-preserving reducibility, *J. Comput. System Sci.* **41**, 430–467.

PITT, L., AND WARMUTH, M. K. (1993), The minimum consistent DFA problem cannot be approximated within any polynomial, *J. Assoc. Comput. Mach.* **40**, 95–142.

RIVEST, R. L., AND SCHAPIRE, R. E. (1987), Diversity-based inference of finite automata, *in* "28th Annual Symposium on Foundations of Computer Science," pp. 78–87; *J. Assoc. Comput. Mach.*, to appear.

RIVEST, R. L., AND SCHAPIRE, R. E. (1990), A new approach to unsupervised learning in deterministic environments, *in* "Machine Learning: An Artificial Intelligence Approach" (Y. Kodratoff and R. Michalski, Eds.), Vol. III, pp. 670–684, Morgan Kaufmann.

SCHAPIRE, R. E. (1988), "Diversity-Based Inference of Finite Automata," Master's Thesis, Massachusetts Institute of Technology, supervised by Ronald L. Rivest; Technical Report MIT/LCS/TR-413, MIT Laboratory for Computer Science.

TARJAN, R. E. (1975), Efficiency of a good but not linear set union algorithm, *J. Assoc. Comput. Mach.* **22**, 215–225.

VALIANT, L. G. (1984), A theory of the learnable, *Comm. Assoc. Comput. Mach.* **27**, 1134–1142.

WILSON, S. W. (1985), Knowledge growth in an artificial animal, *in* "Proceedings of an International Conference on Genetic Algorithms and Their Applications," pp. 16–23.