# 8

# Learning Finite Automata by Experimentation

## 8.1 Active and Passive Learning

Although our early investigation of PAC learning revealed a number of natural but simple classes (such as boolean conjunctions, decision lists, and some geometric concepts) that are efficiently PAC learnable, the results given in Chapters 6 and 7 present rather daunting negative evidence regarding the efficient learnability of more complex classes such as boolean formula and finite automata. These intractability results must lead us to question, at least in some of its details, the model of learning under consideration. For instance, are there sources of information about the target concept that are more powerful than random examples but are still somehow natural, and that we should make available to the learning algorithm? Might our failure to model such sources partially account for the chasm between the hope that efficient learning should be possible and the intractability results we have derived?

Perhaps the most obvious source of information that we have failed to model is *experimentation*. The PAC model is a passive model of learning, in the sense that the learning algorithm has absolutely no control over the sample of labeled examples drawn. However, it is easy to imagine that the ability to experiment with the target concept might be extremely helpful to the learner. This is what we shall demonstrate in this chapter. We model experimentation by giving the learner the ability to make *membership queries*: the learner, when learning the target concept $c$, is given access to an oracle that on any input $x$ returns the correct target label $c(x)$. Thus the learning algorithm may choose particular inputs and see their target classification rather than only passively receiving random labeled inputs.

One setting where membership queries are natural is when the learner is assisted by a teacher. Nature, as modeled by the target distribution in the PAC model, is indifferent to the learner and provides only random examples of the target concept. Particular questions that the learner may have are answered only insofar as the random training sample happens to answer them. The teacher, on the other hand, knows the target concept (or perhaps has already learned an accurate approximation to it), and is sufficiently patient to classify inputs of the learner's choice as positive or negative examples of the target.

In this chapter we show that allowing membership queries can have a significant impact on the complexity of learning problems. In particular, we show we can learn deterministic finite automata in polynomial time in the augmented PAC model where the learning algorithm is given access to an oracle for membership queries in addition to the usual oracle for random examples. This result will in fact follow from an efficient algorithm for learning finite automata in a more demanding model: *exact learning from membership and equivalence queries*, which we define in the next section. Combining this positive result with the hardness results of Chapters 6 and 7, we conclude under the Discrete Cube Root Assumption, membership queries provably make the difference between intractability and efficient learning for finite automata.

In the latter part of the chapter, we generalize our learning algorithm for finite automata to solve another natural learning problem. Imagine that the learner is actually a robot wandering in an unknown environment which consists of $s$ distinct sites. At each step the robot can move from its current site to a neighboring site by performing one of a small set of primitive operations (for example, by moving one step forward, or to the left). Suppose that each site contains some information that can help the robot orient itself in the environment. An example of such information could be the color of the current site. We can actually assume without loss of generality that there is only a single bit of information at each site, because we can modify an arbitrary environment into an equivalent binary environment by replacing each site of the original environment by a "corridor" of binary-valued sites in the new environment that encode the value at the original site.

The robot's goal is to derive a complete model of the observable behavior of its environment. More precisely, the model should predict the exact sequence of bits the robot would observe on any sequence of moves starting from its current position. A natural model of the environment is that of a deterministic finite state automaton. The states of the automaton correspond to the sites in the environment, and transitions correspond to the primitive moves. Each state of the automaton has a single bit of information associated with it (namely, whether it is an accepting state or a rejecting state). This bit represents the bit of information at the corresponding site in the environment.

We give an efficient algorithm for creating an exact model of any such deterministic finite state environment. The algorithm is a refinement of the algorithm for learning finite automata from membership queries. Let us briefly sketch the essential difference between these two automata learning problems. While the robot can actively experiment with its environment (the target automaton), it cannot reset the automaton to a definite state (like the start state). However, this is precisely the ability that is conferred upon the learner by membership queries, since a membership query may be regarded as a reset to the start state followed by an execution of the query string. To prove the robot learning result we show how to simulate a weak reset that is effective enough to help us simulate the previous learning algorithm.

## 8.2 Exact Learning Using Queries

We now introduce a model of learning called **exact learning from membership and equivalence queries**. As usual, the learning algorithm is attempting to learn an unknown target concept chosen from some known concept class $\mathcal{C}$. Unlike in the PAC model, where we were satisfied with a close approximation to the target concept, we will insist that the learning algorithm output the representation of a concept that is exactly equivalent to the target concept. Instead of random examples as in the PAC model, however, the learner now has access to oracles answering the following two types of queries:

- **Membership Queries:** On a membership query, the learning algorithm may select any instance $x$ and receive the correct classification $c(x)$.
- **Equivalence Queries:** On an equivalence query, the learning algorithm submits a hypothesis concept $h \in \mathcal{C}$. If $h(x) = c(x)$ for all $x$ then the learner has succeeded in exactly identifying the target. Otherwise, in response to the query the learner receives an instance $x$ such that $h(x) \neq c(x)$. Such an instance is called a **counterexample**. We make no assumptions on the process generating the counterexamples. For instance, they may be chosen in a manner designed to be confusing to the learning algorithm.

**Definition 17** *We say that the representation class $\mathcal{C}$ is* **efficiently exactly learnable from membership and equivalence queries** *if there is a fixed polynomial $p(\cdot,\cdot)$ and an algorithm $L$ with access to membership and equivalence query oracles such that for any target concept $c \in \mathcal{C}_n$, $L$ outputs in time $p(size(c), n)$ a concept $h \in \mathcal{C}$ such that $h(x) = c(x)$ for all instances $x$.*

Note that we have assumed that concepts are defined only over instances of a single common length $n$ (such as in the case of boolean formulae over $\{0,1\}^n$). This is clearly not the case for a finite automaton, which may accept strings of any length. To apply the definition of exact learning from queries to finite automata, we could simply restrict our attention to finite automata accepting strings of only a single length, as was done in deriving the hardness results in Chapter 7 (such a restriction only makes the hardness result stronger). But it turns out we can give an efficient algorithm without this restriction, provided we make a minor but necessary modification to the definition. For finite automata, if the counterexamples given by the equivalence oracle can be arbitrarily long, it is natural that for our new definition should allow the running time of the learning algorithm to depend on the length of these counterexamples (since we certainly should give the algorithm enough time to read the counterexamples). Thus, to generalize our definition to handle the exact learning of finite automata, in Definition 17 we simply assume that rather than being the exact length of all examples, the parameter $n$ is a given a priori bound on the length of the *longest* counterexample that will be given to $L$ in response to any equivalence query. (In Exercise 8.2 we show that for any equivalence query there always exists a counterexample whose length is at most polynomial in the number of states of the target automaton, and moreover the shortest counterexample can be efficiently computed given the target automaton. Thus, by providing sufficiently short counterexamples, a cooperative teacher can induce the exact learning algorithm for finite automata to run in time polynomial in the number of target states.)

At first glance, it might appear that equivalence queries are an unrealistically strong source of information to provide to the learner. However, it can be shown (see Exercise 8.1) that any representation class that is efficiently exactly learnable from membership and equivalence queries is also **efficiently PAC learnable with membership queries**. By this we mean that it is efficiently learnable in the PAC model, provided the learning algorithm is provided with membership queries in addition to the usual oracle $EX(c, \mathcal{D})$ for random examples. All other aspects of the PAC model, including the success criterion of finding a hypothesis with error less than $\epsilon$ with respect to the target concept and distribution, remain intact.

## 8.3 Exact Learning of Finite Automata

Over the next several sections, we will gradually develop and analyze an algorithm for efficiently exactly learning deterministic finite automata from membership and equivalence queries. We will keep the development at a fairly high level to emphasize the intuition behind the algorithm, but will eventually provide a complete and precise description of the algorithm in Section 8.3.5.

Let $M$ be the target automaton, and assume without loss of generality that $M$ is minimized (that is, it has the fewest states among all automata accepting the same language). We define $size(M)$ to be the number of states of $M$.

The key idea of the algorithm is to attempt to continually discover new states of $M$. By new states we mean states exhibiting behavior that is demonstrably different from the states discovered so far. The algorithm runs in phases. In each phase, the algorithm constructs a tentative hypothesis automaton $\hat{M}$ whose states are the currently discovered states of $M$. It then makes an equivalence query on $\hat{M}$. The counterexample from this equivalence query allows the algorithm to use membership queries to discover a new state of $M$. When all the states of $M$ have been discovered, we will have $\hat{M} = M$.

### 8.3.1 Access Strings and Distinguishing Strings

How can the learning algorithm discover information about the states of $M$? The algorithm will maintain a set $S$ consisting of at most $size(M)$ **state access strings**, and a set $D$ of **distinguishing strings**:

- (*Access*) Each string $s \in S$, when executed from the start state of $M$, leads to a unique state of $M$ that we denote $M[s]$.
- (*Distinguishability*) For each pair of strings $s, s' \in S$ such that $s \neq s'$, there is a distinguishing string $d \in D$ such that one of $sd$ and $s'd$ reaches an accepting state of $M$, and the other reaches a rejecting state of $M$. (That is, exactly one of $M[sd]$ and $M[s'd]$ is an accepting state.)

We shall refer to the states $\{M[s] : s \in S\}$ as the *known* states of $M$, since we know how to access them from the start state. Notice that all these known states must be distinct. This is because for each pair of strings $s, s' \in S$, there is a string $d$ in $D$ that witnesses the fact that starting from states $M[s]$ and $M[s']$ and executing $d$ leads to different final states. The goal of the learning algorithm is to discover all the states of $M$ by finding *size*$(M)$ access strings, together a with distinguishing string for every pair of access strings. The task of reconstructing the actual transitions of $M$ from this information is quite straightforward (as we shall see).

In the algorithm, the current sets $S$ and $D$ of access and distinguishing strings will be maintained in a convenient data structure, a binary **classification tree**. Each internal node is labeled by a string in $D$, and each leaf is labeled by a string in $S$. The tree is constructed by placing at the root any string $d$ from $D$ that distinguishes two strings in $S$, and placing in the left subtree of the root all strings $s \in S$ such that $sd$ is rejected by $M$, and in the right subtree all $s \in S$ such that $sd$ is accepted. This induces a nontrivial partition of $S$ (since $d$ distinguishes some pair of strings in $S$), and we simply recurse at each subtree until each string in $S$ is at its own leaf. Then any pair of access strings $s, s' \in S$ are distinguished by the string labeling their least common ancestor in the classification tree. Our algorithm will dynamically maintain a classification tree representation of $S$ and $D$.

Our algorithm will make sure that the distinguishing string that labels the root of the classification tree is always the empty string $\lambda$. This will

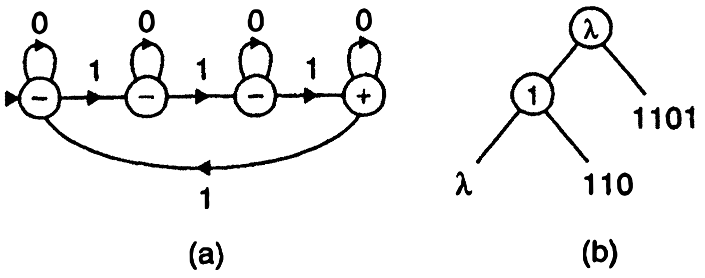

Figure 8.1: (a) *Finite automaton counting the number of* 1*'s in the input* 3 *mod* 4. (b) *A classification tree for this automaton.*

ensure that all the access strings to accepting states will lie in the right subtree and the access strings to rejecting states in the left subtree. The algorithm will also arrange that $\lambda$ is one of the access strings. This ensures that we can access the start state of the automaton.

Figure 8.1(a) shows a finite automaton that will form the basis of a running example. This automaton accepts an input string if and only if the number of 1's in the string is 3 modulo 4. Figure 8.1(b) shows a classification tree for this automaton, with access strings $\{\lambda, 110, 1101\}$ and distinguishing strings $\{\lambda, 1\}$.

## 8.3.2 An Efficiently Computable State Partition

Now suppose we are given a new string $s'$ that is not in the current access string set $S$, but that $M[s'] = M[s]$ for some access string $s \in S$. Then we can efficiently determine $s$ from $s'$ by **sifting** $s'$ down our classification tree using membership queries: starting at the root, if we are at an internal node labeled by the distinguishing string $d$, we make a membership query on the string $s'd$ and go to the left or right subtree as indicated by the query answer (left on reject, right on accept). We continue in this manner to reach a leaf, which must be labeled by $s$.

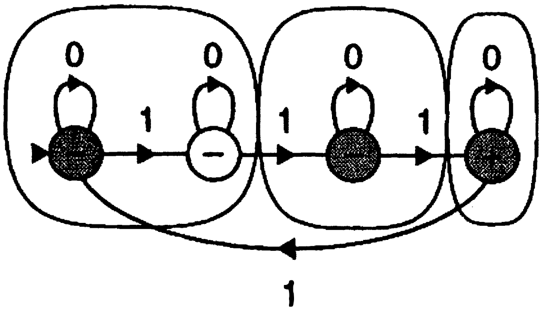

Figure 8.2: *Partition induced by the classification tree in Figure 8.1.*

More importantly, even if $M[s'] \neq M[s]$ for all $s \in S$, sifting $s'$ down the classification tree still defines a path to a leaf, and this path depends only on $M[s']$. In other words, for any strings $s'$ and $s''$, if $M[s'] = M[s'']$ then sifting $s'$ and $s''$ defines exactly the same path down the classification tree. Thus, the classification tree induces a partition on the states of $M$, and each equivalence class of this partition contains exactly one state $M[s]$ such that $s \in S$, which we will consider the representative element for the equivalence class.

Sifting can be efficiently implemented, and the number of membership queries for a sift operation is bounded by the depth of the classification tree.

Figure 8.2 shows the partition of the automaton of Figure 8.1(a) that is induced by the classification tree of Figure 8.1(b). The known or representative state in each equivalence class has been shaded. Note that the access string for a known state may not be the shortest string reaching that state. For example, in Figures 8.1 and 8.2, we have the access string 110 even though the shorter string 11 accesses the same state.

## 8.3.3 The Tentative Hypothesis $\hat{M}$

We are now in a position to describe the construction of a hypothesis automaton $\hat{M}$, whose states can be thought of as the known states of $M$ (that is, states for which there are access strings in the leaves of the current classification tree). If all the states of $M$ have been discovered then it will turn out that $\hat{M} = M$. Otherwise, the counterexample from the equivalence query on $\hat{M}$ will be used to discover a new state (that is, access string) of $M$.

We first define $\hat{M}$ algorithmically and then provide some insight into its structure. Given the classification tree, it is easy to construct $\hat{M}$ using equivalence queries. We identify (label) the states of $\hat{M}$ with the access strings in the classification tree. For each access string (state) $s$ and symbol $b$, the destination state of the $b$-transition out of state $s$ is just the access string that results from sifting $sb$ down the classification tree.

$\hat{M}$ can be thought of as an automaton whose states are a subset of the states of $M$, but with transitions that are possibly quite different than those of $M$. Imagine a state diagram of $M$ in which the transitions are represented by dashed lines, and the states are grouped by the equivalence classes defined by the current classification tree. (See Figure 8.3(a), in which $M$ is the four-state automaton shown, with its transitions represented by dashed lines. The states of $M$ are partitioned into two classes of two states each.) Now let us shade each known state $M[s]$ for $s \in S$. (The shaded states of $M$ in Figure 8.3(a) are the known states.) Then $\hat{M}$ will be defined only on those states of $M$ that are shaded, and each equivalence class of $M$ has exactly one such shaded state. The transitions of $\hat{M}$, which will be represented by bold lines, are defined as follows: for $b \in \{0, 1\}$, the bold $b$-transition leaving the shaded state $M[s]$ is obtained simply by taking $M$'s dashed $b$-transition leaving $M[s]$ and redirecting it from its current destination state to the unique shaded state of the equivalence class of the destination state in $M$. (See Figure 8.3(a).) For example, in Figure 8.3(a), the dashed 0-transition of the left

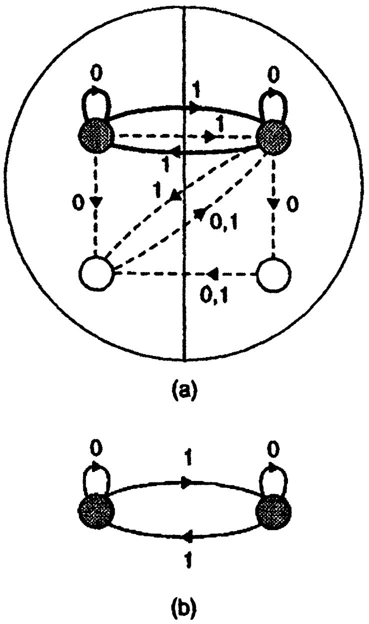

Figure 8.3: (a) *Embedded hypothesis defined by a partition of a target automaton into two equivalence classes. Transitions of the target automaton $M$ are dashed, transitions of the hypothesis $\hat{M}$ defined by the partition and the shaded known states are bold.* (b) *The resulting hypothesis $\hat{M}$ extracted.*

shaded state stays in the same equivalence class of states; thus, the bold 0-transition of the left shaded state becomes a self-loop. Similarly, the dashed 1-transition of the right shaded state goes to the left equivalence class; thus, the bold 1-transition of the right shaded state also goes to the left equivalence class, but is redirected to the left shaded state. Notice that in the case when all the states of $M$ are shaded, $\hat{M} = M$.

We should point out a common point of confusion about $\hat{M}$: $\hat{M}$ might look very different from $M$ and it might accept a totally different language than that accepted by $M$. So it is a mistake to think of $\hat{M}$ as an approximation to the target automaton $M$ (unless they have the same number of states, in which case $M = \hat{M}$). A related point is that the learning algorithm makes progress by increasing the number of access strings or leaves in the classification tree. The tentative hypothesis automaton $\hat{M}$ facilitates this increase in the number of leaves in the classification tree.

## 8.3.4 Using a Counterexample

We now show how we can use a string $\gamma$ that is a counterexample to the equivalence of $M$ and $\hat{M}$ in order to discover a new state of $M$, thus allowing the classification tree to be updated. The conceptual idea is to simulate the behavior of both $M$ and $\hat{M}$ in parallel on the string $\gamma$ (that is, follow both the dashed trajectory and the bold trajectory dictated by $\gamma$) in order to discover the first point at which the two trajectories diverge to different equivalence classes of states. At this point of divergence, the dashed and bold transitions must take place from two different states in the same equivalence class, thus providing us with access to a new state in this equivalence class.

To make this precise, we first recall our assumption that the root of the classification tree is labeled by the empty string $\lambda$, and that one of the access strings is $\lambda$ (both of these conditions will be easily arranged by our algorithm in its initialization step). The first condition implies that no equivalence class of $M$ contains both an accepting and a rejecting state. The second condition implies that in the embedding of $\hat{M}$ in $M$, the start states of the two automata coincide, and thus the machines are "synchronized" at the start of any string. So the dashed and bold trajectories determined by the counterexample $\gamma$ begin in a common equivalence class (in fact, in the same state) and end up in different equivalence classes (since exactly one of $M$ and $\hat{M}$ accepts $\gamma$).

Let $\hat{M}[s]$ denote the state reached by following the transitions of $\hat{M}$ on string $s$; this is just the final destination of the bold trajectory determined by $s$. Let $\gamma_i$ denote the $i^{th}$ symbol of $\gamma$ and let $\gamma[i]$ denote the prefix of $\gamma$

of length $i$, that is $\gamma[i] = \gamma_1 \cdots \gamma_i$. Let $1 \leq j \leq |\gamma|$ be the first index such that the equivalence class of $M[\gamma[j]]$ differs from that of $\hat{M}[\gamma[j]]$ (thus, the two trajectories have diverged for the first time). See Figure 8.4.

By the choice of $j$, we know that $M[\gamma[j-1]]$ and $\hat{M}[\gamma[j-1]]$ are in the same equivalence class, yet the dashed transition from $M[\gamma[j-1]]$ and the bold transition from $\hat{M}[\gamma[j-1]]$ on the symbol $\gamma_j$ led to different equivalence classes. This means that $M[\gamma[j-1]]$ and $\hat{M}[\gamma[j-1]]$ are actually different states in the same equivalence class. Since the only shaded (known) state in this class is $\hat{M}[\gamma[j-1]]$, and recalling that the access strings discovered so far reach only the shaded states, $M[\gamma[j-1]]$ is a new state with access string $\gamma[j-1]$.

To distinguish $M[\gamma[j-1]]$ from all previously discovered states (that is, to place this state in its own equivalence class), we only need to distinguish $M[\gamma[j-1]]$ and $\hat{M}[\gamma[j-1]]$ from each other (that is, to "split" the current equivalence class to which they both belong). The correct distinguishing string simply expresses the fact that the $\gamma_j$ transitions from $M[\gamma[j-1]]$ in $M$ and from $\hat{M}[\gamma[j-1]]$ in $\hat{M}$ lead to different equivalence classes, namely, the equivalence classes of $M[\gamma[j]]$ and $\hat{M}[\gamma[j]]$. If $d$ is the string distinguishing the equivalence classes of $M[\gamma[j]]$ and $\hat{M}[\gamma[j]]$, then the correct distinguishing string for $M[\gamma[j-1]]$ and $\hat{M}[\gamma[j-1]]$ is $\gamma_j d$.

It should be clear that the task of updating the classification tree by processing a counterexample string can be carried out efficiently using membership queries. This involves determining the equivalence class of each prefix of the counterexample string by sifting it down the current classification tree, as well as tracing its path in the hypothesis automaton $\hat{M}$, which is known explicitly.

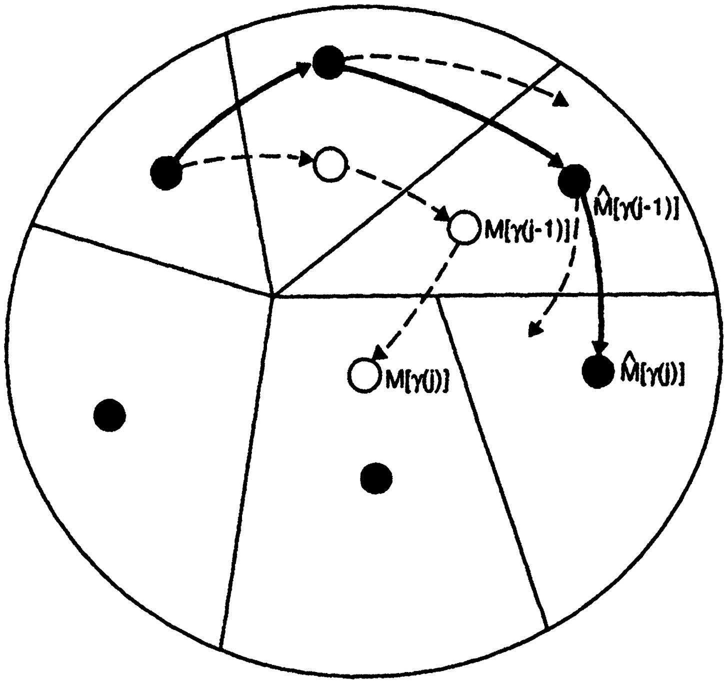

Figure 8.4: ***The trajectories in the target automaton (dashed transitions, unshaded states) and the hypothesis automaton (bold transitions, shaded states) traced by a counterexample. From each shaded state on the bold trajectory, the dashed transition of the target automaton (which may be different from the bold transition) is shown for completeness.***

To sum up, as long as the number of leaves of the classification tree is smaller than $size(M)$, the hypothesis automaton $\hat{M}$ is necessarily different from $M$. Therefore an equivalence query must return some counterexample string $\gamma$ which we can use to update the classification tree by adding a new leaf node. Eventually the classification tree will have $size(M)$ leaf nodes, each accessing a different state of $M$, and at this point $\hat{M} = M$.

## 8.3.5 The Algorithm for Learning Finite Automata

We can now describe our algorithm for learning finite automata in some detail. We start by describing the subroutine **Sift**. This subroutine takes as input a string $s$ and the current classification tree $T$, and outputs the access string in $T$ of the equivalence class of $M[s]$, the state of $M$ accessed by $s$.

**Procedure Sift**$(s, T)$:

- Initialization: set the current node to be the root node of $T$.
- Main Loop:
  - Let $d$ be the distinguishing string at the current node in the tree.
  - Make a membership query on $sd$. If $sd$ is accepted by $M$, update the current node to be the right child of the current node. Otherwise, update the current node to be the left child of the current node.
  - If the current node is a leaf node, then return the access string stored at this leaf. Otherwise, repeat the Main Loop.

Next, we describe the procedure for constructing the hypothesis automaton $\hat{M}$ that is defined by the current classification tree $T$.

**Procedure Tentative-Hypothesis**$(T)$:

- For each access string (leaf) of $T$, create a state in $\hat{M}$ that is labeled by that access string. Let the start state of $\hat{M}$ be the state $\lambda$.
- For each access state $s$ of $\hat{M}$ and each $b \in \{0, 1\}$, compute the $b$-transition out of state $s$ in $\hat{M}$ as follows:

    - $s' \leftarrow$ **Sift**$(sb, T)$.
    - Direct the $b$-transition out of state $s$ to state $s'$.

- Return $\hat{M}$.

Next we describe the procedure **Update-Tree**, which takes as arguments the current classification tree $T$ and a counterexample string $\gamma$ to the hypothesis automaton $\hat{M}$ defined by $T$. The procedure finds a new access string, and updates $T$ by adding a new leaf node labeled with the new access string.

**Procedure Update-Tree$(\gamma, T)$:**

- For each prefix $\gamma[i]$ of $\gamma$:
    - $s_i \leftarrow$ **Sift**$(\gamma[i], T)$.
    - Let $\hat{s}_i = \hat{M}[\gamma[i]]$.
- Let $j$ be the least $i$ such that $s_i \neq \hat{s}_i$.
- Replace the node labeled with the access string $s_{j-1}$ in $T$ with an internal node with two leaf nodes. One leaf node is labeled with the access string $s_{j-1}$ and the other with the new access string $\gamma[j-1]$. The newly created internal node is labeled with the distinguishing string $\gamma_j d$, where $d$ is the correct distinguishing string for $s_j$ and $\hat{s}_j$ ($d$ can be obtained from $T$).

We are now ready to describe the overall algorithm for learning finite automata:

**Algorithm Learn-Automaton:**

- Initialization:
    - Do a membership query on the string $\lambda$ to determine whether the start state of $M$ is accepting or rejecting.

  - Construct a hypothesis automaton that consists simply of this single (accepting or rejecting) state with self-loops for both the 0 and 1 transitions.
  - Perform an equivalence query on this automaton; let the counterexample string be $\gamma$.
  - Initialize the classification tree $T$ to have a root labeled with the distinguishing string $\lambda$ and two leaves labeled with access strings $\lambda$ and $\gamma$.

- Main Loop:
  - Let $T$ be the current classification tree.
  - $\hat{M} \leftarrow$ **Tentative-Hypothesis**($T$).
  - Make an equivalence query on $\hat{M}$. If it is equivalent to the target then output $\hat{M}$ and halt. Otherwise, let $\gamma$ be the counterexample string.
  - **Update-Tree**($T, \gamma$).
  - Repeat Main Loop.

In Figure 8.5, we show the evolution of the hypothesis $\hat{M}$ and the classification tree as the algorithm is executed on the target automaton first shown in Figure 8.1.

### 8.3.6 Running Time Analysis

The number of times the Main Loop of algorithm **Learn-Automaton** is executed is exactly $size(M)$. This is because, as we have already argued, each iteration discovers a new state of $M$ in the form of an access string, and when all states are discovered then $\hat{M} = M$. Each execution of the Main Loop of **Learn-Automaton** makes a call to procedure **Tentative-Hypothesis** to compute $\hat{M}$, and each such call invokes $O(size(M))$ sifting operations. Also, each execution of the Main Loop of

**Learn-Automaton** requires the processing of a single counterexample by procedure **Update-Tree**. A counterexample of length $n$ requires at most $n$ sifting operations. Therefore, we have $size(M)$ Main Loop executions, each of which requires $O(size(M) + n)$ sifting operations, where $n$ is the length of the longest counterexample. It is easy to see that the running time of our algorithm is dominated by the sifting operations, and that sifting is a $O(size(M))$ operation. We have thus derived the first of the two main results of this chapter:

**Theorem 8.1** *The representation class of deterministic finite automata is efficiently exactly learnable from membership and equivalence queries.*

It is worth noting that as a corollary to our analysis of the learning algorithm, we can give an alternative derivation of the well-known Myhill-Nerode theorem, which states that for any regular language $L$ there is a unique automaton of minimum size accepting $L$. First observe that the learning algorithm only gets information about the language $L$ accepted by the target automaton $M$, and so if two different target automata $M_1$ and $M_2$ accept the same language $L$ then the learning algorithm must produce the same output automaton $\hat{M}$. On the other hand, we showed that output automaton $\hat{M}$ is identical to the target automaton, *assuming only* that the target automaton is a minimum state automaton. It follows that the minimum state automaton accepting $L$ is unique.

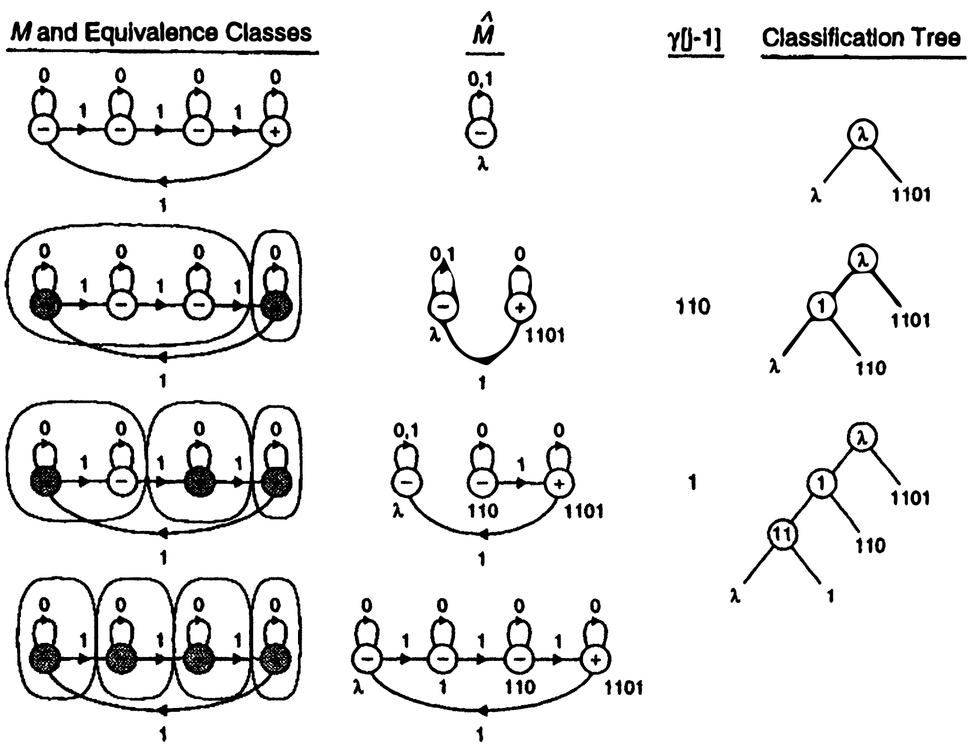

Figure 8.5: *Sample execution of algorithm* **Learn-Automaton** *on the* 3 *mod* 4 *counter target automaton. In the first column, we show the target automaton with the partition defined by the classification tree of the previous row, along with the shaded known states. The second column shows the hypothesis* $\hat{M}$ *defined by the partition to its left; the states of* $\hat{M}$ *are also labeled by their access string. Every equivalence query on* $\hat{M}$ *is answered by the same repeated counterexample* $\langle 1101, 1\rangle$ *until* $\hat{M} = M$. *The third column shows the prefix* $\gamma[j-1]$ *of* $\gamma = 1101$ *on which a difference of equivalence classes is first detected, and the fourth column shows the classification tree at each step.*

## 8.4 Learning without a Reset

In this section, we strengthen the result from the previous section and give an efficient algorithm for learning deterministic finite automata from equivalence queries and membership queries **without resets**. By this we mean that the membership oracle does not reset $M$ to its start state before each membership query; instead it simply starts processing the next query string from its current state. Thus the answer to the query string $\gamma^i$ which follows a sequence of queries $\gamma^1, \ldots, \gamma^{i-1}$ indicates whether $M[\gamma^1 \cdots \gamma^{i-1}\gamma^i]$ is an accepting or rejecting state. As stated in the introduction, we will assume without loss of generality that each of the query strings $\gamma_i$ is only a single-bit query.

We need to be a little careful in specifying the goal of the learner in this new setting. The problem arises from the fact that the target automaton $M$ may contain components from which the learning algorithm can never escape once they are entered, and thus might not be able to explore the rest of the automaton. For simplicity, we shall finesse this problem by simply assuming that $M$ is **strongly connected**: that is, there is a directed path between every pair of states in $M$. In the more general case, the automaton would eventually get trapped in a strongly connected component. In this case the learning algorithm would end up with an accurate model of this strongly connected component.

In keeping with the idea that the learner's goal is to model its environment from its current position, we shall also modify the oracle for equivalence queries. Whenever the learning algorithm makes an equivalence query on hypothesis automaton $\hat{M}$, this query is interpreted from the learning algorithm's current position. This means that if $\hat{M}$ is equivalent to $M$ when we define the start state of $M$ to be the current position of the learning algorithm in $M$, then learning is complete, and if $\hat{M}$ is not equivalent to $M$ from the current position, a counterexample from the current position is provided. Thus, a counterexample to $\hat{M}$ provides the learning algorithm with a sequence of moves $\gamma$ such that if we execute $\gamma$ from the current position in $M$, and if we execute $\gamma$ from the start state of $\hat{M}$, different outputs are obtained. For brevity, we shall refer to this learning model for finite automata with the modified membership and equivalence queries as the **no-reset model** of exactly learning deterministic finite automata from membership and equivalence queries, and to the original model as the **reset model**. Note that the no-reset model only makes sense in the context of learning the particular representation class of finite automata, whereas the original model is of more general interest.

It is not difficult to see that a learning algorithm in the no-reset model can be simulated by an algorithm in the reset model, by making a membership query for each prefix of the string describing the movements of the no-reset learner. On the other hand, the no-reset learner does not seem to have the power of membership queries with resets, since it may not know how to return to the start state from its current position in the target automaton.

We will use the notion of *homing sequences* to effect a kind of simulation of resets in the no-reset model, and this will allow us to modify our algorithm **Learn-Automaton** for the reset model into an efficient *randomized* algorithm for learning automata in the no-reset model. The overview of the development is as follows. We begin in Section 8.4.1 by defining a homing sequence, and showing how we can learn in the no-reset model if we are given a short homing sequence for the target automaton. In Section 8.4.2 we prove the existence of short homing sequences, and we analyze the key idea of our new algorithm: simulating many copies of our algorithm **Learn-Automaton** using a possibly faulty homing sequence. We show that the failure of such a simulation allows us to improve our proposed homing sequence and restart the entire simulation. In Section 8.4.3 we give a detailed description of our algorithm and its analysis, and in Section 8.4.4 we return to address an assumption made during the development about the many simulated copies of **Learn-Automaton**.

## 8.4.1 Using a Homing Sequence to Learn

Let $M$ be the target automaton. As before we will assume that $M$ is a minimum state automaton, and let $size(M)$ be the number of states of $M$. To begin with, without loss of generality we will assume that our learning algorithm knows the value of $size(M)$; it is a simple exercise to eliminate this assumption.

For any string $h$, we denote by $output(q, h)$ the output (that is, the complete sequence of accept/reject bits) observed by executing $h$ from state $q$ of $M$, and by $state(q, h)$ the state of $M$ reached by executing $h$ from $q$. For any sequence $h$, we define

$$output(h) = \{output(q, h) : q \in M\}.$$

This is just the set of all possible outputs observed by executing $h$ as we range over all possible starting states $q$ of $M$. Notice that if $M$ has $size(M)$ states, $|output(h)| \leq size(M)$ for any sequence $h$.

A **homing sequence** for a finite automaton $M$ is a sequence $h$ such that for any state $q$ of $M$, $output(q, h)$ uniquely determines $state(q, h)$: that is, if $output(q, h) = output(q', h)$ then $state(q, h) = state(q', h)$. Note that we do not demand that $q = q'$; a homing sequence simply ensures that identical output sequences imply the same destination state, not the same origin.

Let us first show the existence of short homing sequences for any automaton, and how a homing sequence can be used to learn in the no-reset model, and defer the problem of finding such a sequence in the no-reset model to Section 8.4.2. The main idea is that any sequence $h$ that is not already a homing sequence can be extended to a sequence $hx$ such that $|output(hx)| > |output(h)|$ for some string $x$ of length at most $size(M)$. Since $|output(h)| \leq size(M)$ for every string $h$, we will have the desired homing sequence after at most $size(M)$ such extensions.

First note that for any $h$ and any $x$, $|output(hx)| \geq |output(h)|$. Now if $h$ is not a homing sequence, there exist two different states $q$

and $q'$ of $M$ such that $output(q, h) = output(q', h)$, but $state(q, h) \neq state(q', h)$. However, there must be a distinguishing sequence $d$ for the destination states $state(q, h)$ and $state(q', h)$. So now we get two distinct output sequences $output(q, hd) \neq output(q', hd)$ in place of the single output sequence $output(q, h) = output(q', h)$, and thus $|output(hx)| > |output(h)|$.

Returning to our learning problem, note that a homing sequence $h$ provides a kind of "weak" reset for $M$. Although executing $h$ does not always return us to the same fixed state of $M$, it does "orient" us within $M$, in the sense that the output observed upon executing $h$ uniquely determines the resulting state. Given the homing sequence $h$, we can imagine simulating our learning algorithm **Learn-Automaton** for the reset model in the following way: each time **Learn-Automaton** requests a reset (that is, makes a membership query), we temporarily suspend its execution and repeatedly execute $h$ until some execution results in the specific output sequence $\sigma$. We then resume simulation of **Learn-Automaton** and in this way, before every membership query of **Learn-Automaton** we return to the same fixed state of $M$, which we may consider the "start state".

Unfortunately, we have no way of bounding the amount of time we may have to wait before executing $h$ gives rise to the specific output sequence $\sigma$. This will be addressed by simulating many copies $L_\sigma$ of **Learn-Automation**, one for each output sequence $\sigma$ that we have observed upon executing $h$ (that is, one for each $\sigma \in output(h)$ that we have seen so far). At any time, at most one copy $L_\sigma$ will be awake. When this copy makes a membership query, we suspend its execution, execute $h$ and obtain some output $\sigma'$, and then awaken (that is, resume execution of) the copy $L_{\sigma'}$. There are at most $|output(h)| \leq size(M)$ copies, and any copy that terminates has exactly learned $M$. Each copy does at most as much computation as an execution of **Learn-Automaton** in the reset model, and thus the total amount of computation performed is at most $size(M)$ times that of **Learn-Automaton** (plus a small overhead cost for the executions of $h$). Thus we have shown:

**Lemma 8.2** *There is an efficient algorithm for exactly learning deterministic finite automata in the no-reset model of membership and equivalence queries, provided the algorithm is also given a homing sequence for the target automaton as input.*

The main difficulty with the above proposal is that we must first somehow find a homing sequence. We now address this issue.

## 8.4.2 Building a Homing Sequence Using Oversized Generalized Classification Trees

The overall idea for finding a homing sequence will be to run the multi-copy simulation suggested above using a sequence $h$ which in fact may not be a homing sequence. If this simulation fails to learn $M$, we will be able to extend $h$ to a sequence $hx$ that is "closer" to being a homing sequence.

For any sequence $h$, let us denote by $reset(h, \sigma)$ the set of possible states of $M$ we could be in if the string $\sigma$ has just been observed as the output while executing the string $h$. Thus,

$$reset(h, \sigma) = \{r \in M : (\exists q \in M) state(q, h) = r, output(q, h) = \sigma\}.$$

Suppose that we use a sequence $h$ which is not a homing sequence, and awaken the copy $L_\sigma$ only when we have just executed $h$ and observed the ouput sequence $\sigma$. Then every time that $L_\sigma$ is awakened, $M$ will be in some state in $reset(h, \sigma)$.

As we have mentioned, our hope is to iteratively update $h$ from failed attempts to learn $M$ using the copies $L_\sigma$, until we end up with a homing sequence, at which point we have already argued the correctness of our multi-copy simulation (Lemma 8.2). The correctness of this scheme will rely on the following important property of each $L_\sigma$, whose proof we shall defer until a later section: if we use a sequence $h$ which is not a homing sequence to run the copies $L_\sigma$, then each $L_\sigma$ either halts and outputs an automaton equivalent to $M$, or it successively constructs a series of larger and larger *generalized classification trees.*

Structurally, a generalized classification tree looks just like the classification tree of algorithm **Learn-Automaton** in the reset model. The key property of a generalized classification tree $T$ is that for any access string (leaf) of $T$, and any distinguishing string (internal node) $d$ of $T$ that is on the path from the root to $s$, there is some state $q \in reset(h, \sigma)$ that "witnesses" the claimed behavior of $M$ on these strings. More precisely, we say that $T$ is a **generalized classification tree** with respect to $h$ and $\sigma$ if and only if for any access string $s$ and distinguishing string $d$ on the path from the root to $s$ in $T$, if $s$ is in the right (left, respectively) subtree of $d$, there is a $q \in reset(h, \sigma)$ such that $state(q, sd)$ is accepting (rejecting, respectively). Note that a classification tree is just a generalized classification tree in which $|reset(h, \sigma)| = 1$.

Assuming for now that each copy $L_\sigma$ can only either halt with a correct hypothesis automaton or construct successively larger generalized classification trees, the only way in which our simulation can fail when using an $h$ that is not a homing sequence is that some copy $L_\sigma$ constructs a generalized classification tree $T_\sigma$ with $size(M) + 1$ leaves. We now propose and analyze a randomized scheme for using such an oversized $T_\sigma$ to find a string $x$ such that $hx$ is closer to being a homing sequence, in the sense that $|output(hx)| > |output(h)|$.

Let $r \in reset(h, \sigma)$. Thus $r$ is one of perhaps many states that $M$ could be in when $L_\sigma$ is restarted when using the sequence $h$. Since $M$ has only $size(M)$ states and $T_\sigma$ contains $size(M)+1$ access strings, there must exist access strings $s_i$ and $s_j$ of $T_\sigma$ such that $state(r, s_i) = state(r, s_j)$ by the Pigeonhole Principle. Let $d$ be the distinguishing string for $s_i$ and $s_j$ in $T_\sigma$. Then since $s_i$ and $s_j$ lead to the same state of $M$ from $r$, we must also have $state(r, s_i d) = state(r, s_j d)$; assume without loss of generality that this is an accepting state. On the other hand, since $d$ is a distinguishing string for $s_i$ and $s_j$ in $T_\sigma$, there must also exist states $r_i, r_j \in reset(h, \sigma)$ such that exactly one of $state(r_i, s_i d)$ and $state(r_j, s_j d)$

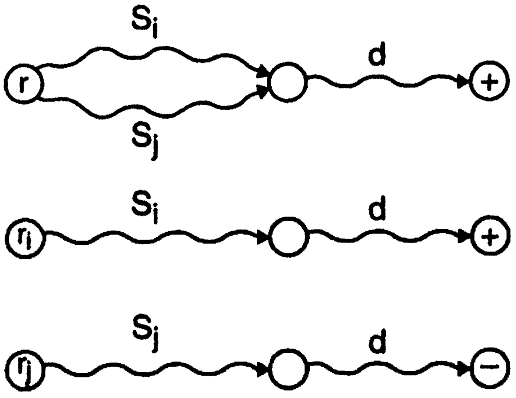

Figure 8.6: *Homing sequence update.*

is an accepting state, say $state(r_i, s_i d)$. Now $hs_j d$ is closer to being a homing sequence than $h$, because on output $\sigma$, $h$ might have led us to either of $r$ and $r_j$, but now $s_j d$ distinguishes between $r$ and $r_j$ (see Figure 8.6).

Of course, we have no way of determining just by looking at $T_\sigma$ which access strings $s_i$ and $s_j$ have the above property. Instead, we use a randomized scheme that chooses two leaves $s_i$ and $s_j$ of $T_\sigma$ at random, and updates the proposed homing sequence $h$ to be $hs_j d$, where $d$ is the distinguishing string (least common ancestor) for $s_i$ and $s_j$ in $T_\sigma$. We then restart the entire multi-copy simulation of algorithm **Learn-Automaton.**

Since we know that there is some pair $s_i$ and $s_j$ in $T_\sigma$ that can be used to improve $h$, the probability that we actually make an improvement is at least $1/(size(M))^2$. Note that even if we fail to make an improvement to $h$, we certainly cannot make it worse because we always have $|output(hs_j d)| \geq |output(h)|$.

## 8.4.3 The No-Reset Algorithm

We are now prepared to give a detailed description of our algorithm for the no-reset model. We then provide its analysis under the assumption that all copies $L_\sigma$ of **Learn-Automaton** always maintain a generalized classification tree, and then return to validate this assumption in the following section.

**Algorithm No-Reset-Learn-Automaton:**

- $h \leftarrow \lambda$.
- Main Loop:
  - Execute (that is, make a membership query on) the current proposed homing sequence $h$, and let $\sigma$ be the output sequence observed.
  - If the output sequence $\sigma$ has not previously been observed after executing the current $h$, initialize a copy $L_\sigma$ of algorithm **Learn-Automaton**.
  - Awaken copy $L_\sigma$ and simulate its next membership query and all subsequent computation up to (but not including) the next membership query:
    * Any time $L_\sigma$ makes a equivalence query $\hat{M}$, give this query to the equivalence query oracle. If it is successful, halt and output $\hat{M}$ (learning is complete). If it is unsuccessful return the counterexample $\gamma$ to $L_\sigma$.
    * If the generalized classification tree $T_\sigma$ of copy $L_\sigma$ ever has size of $size(M) + 1$ leaves, then choose leaves $s_i$ and $s_j$ of $T_\sigma$ at random, and perform the update $h \leftarrow hs_jd$, where $d$ is the least common ancestor of $s_i$ and $s_j$ in $T_\sigma$. Delete all copies of algorithm **Learn-Automaton** and restart the entire simulation by returning to the Main Loop.

For the analysis, note that **No-Reset-Learn-Automaton** halts only if learning is complete. Thus we only need to bound the running time. First, we observe that the number of times the tentative homing sequence can be improved is at most $size(M)$; this is because as we already argued, each improvement increases the size of the set $output(h)$ up to a maximum of $size(M)$. Improvements in the tentative homing sequence happen with probability at least $1/(size(M))^2$ each time the algorithm discovers an oversized classification tree. Since the simulation is simply restarted after each such modification to the tentative homing sequence, the running time of the algorithm is bounded by $size(M)^3$ multiplied by the time required to build an oversized tree starting with a new tentative homing sequence. From Section 8.4.1, the latter quantity is at most $(size(M))$ times the running time of **Learn-Automaton** in the reset model. Therefore the expected running time of the new algorithm is at most $(size(M))^4$ times the running time of **Learn-Automaton** in the reset model.

### 8.4.4 Making Sure $L_\sigma$ Builds Generalized Classification Trees

We now must return to an issue that we had deferred earlier: we still need to show that each copy $L_\sigma$ has the property that even if it is awakened when we observe output $\sigma$ upon executing a string $h$ that is not a homing sequence, $L_\sigma$ either halts and outputs an automaton equivalent to $M$, or it successively constructs a series of larger and larger generalized classification trees $T_\sigma$. The issue here is that if $h$ is not a homing sequence then each reset of $L_\sigma$ puts $M$ in an arbitrary state $q \in reset(h, \sigma)$.

First, let us assume that the lack of a consistent reset state does not ever cause the copy $L_\sigma$ to abort. We will momentarily come back and address this assumption. In this case, the only way $L_\sigma$ halts is if it made a successful equivalence query, and therefore discovered an automaton equivalent to $M$. On the other hand, if it does not halt, then it works in phases, and in each phase it adds a new leaf node to its current tree $T_\sigma$, which we now argue is a generalized classification tree.

We thus have to verify that if $d$ is a distinguishing string on the path from the root to the leaf $s$ in the current tree $T_\sigma$, then if $s$ is in the right subtree of $d$ there is some reset state $q \in reset(h, \sigma)$ such that the $state(q, sd)$ is an accepting state, and if $s$ is in the left subtree of $d$ there is some reset state $q \in reset(h, \sigma)$ such that the $state(q, sd)$ is a rejecting state. This fact is established by proving that for every such $(s, d)$ pair there is a *witness* in the membership query history of $L_\sigma$ — that is, $L_\sigma$ must have at some point performed the membership query $sd$, and that the current tree $T_\sigma$ is consistent with the answer given to that membership query.

Recall that $T_\sigma$ is modified only by a call to **Update-Tree**$(T_\sigma, \gamma)$ for some counterexample string $\gamma$. Let us denote the updated tree by $T'_\sigma$. We will show that if all $(s, d)$ pairs of $T_\sigma$ were witnessed, then this continues to be true of $T'_\sigma$. Since we update $T_\sigma$ by adding a single access string $\gamma[j-1]$ and a single distinguishing string $\gamma_j d$, we must only verify that there are witnesses for pairs that involve one of these two strings.

There are only two access strings in $T'_\sigma$ whose path from the root passes through the new internal node labeled $\gamma_j d$ — namely, $\gamma[j-1]$ and $s$, where $s$ is the access string reached by sifting $\gamma[j-1]$ down $T_\sigma$ (see Section 8.3.4). Of these, the pair $(\gamma[j-1], \gamma_j d)$ was witnessed by the membership query $\gamma[j-1]\gamma_j d = \gamma[j]d$ which was made while doing a sift operation on the string $\gamma[j]$ (while processing the counterexample $\gamma$). The pair $(s, \gamma_j d)$ was witnessed by the membership query $s\gamma_j d$ which was performed to determine the destination state for the $\gamma_j$-transition out of the state $s$ in the tentative hypothesis automaton $\hat{M}$. To see this more clearly, recall that determining this transition involved sifting $s\gamma_j$ down $T_\sigma$, and that $d$ is one of the distinguishing strings on the path from the root to access string $s$ in $T_\sigma$.

Lastly, we must witness every remaining new pair $(\gamma[j-1], d')$ of $T'_\sigma$ for all of the distinguishing strings $d' \neq \gamma - jd$ on the path from the root to $\gamma[j-1]$. Note that all such $d'$ were present in the tree $T_\sigma$. All these pairs were witnessed while sifting the string $\gamma[j-1]$ down $T_\sigma$ to determine its equivalence class.

As our final detail, we have to consider the possibility that $L_\sigma$ may abort since the answers to the membership queries can be inconsistent because there is no consistent reset state. We will show that (with one small exception which is easily fixed) $L_\sigma$ never checks the answers to membership queries for consistency. First observe that $L_\sigma$ makes membership queries in two places: one is to fill in the transitions of the hypothesis automaton $\hat{M}$. Notice that even if the answers to all these queries were arbitrary, they would not cause $L_\sigma$ to abort, they would just result in incorrect transitions for $\hat{M}$. The other place where the algorithm makes membership queries is while processing the counterexample string $\gamma$. Once again incorrect answers to membership queries do not cause the algorithm to abort, with one small exception. Let the length of the counterexample string $\gamma$ be $m$. Suppose that all the prefixes of $\gamma$ up to $\gamma[m-1]$ reveal no difference between the equivalence class in $\hat{M}$ and the equivalence class in $M$. When the algorithm goes on to compute the equivalence class of $\gamma[m] = \gamma$ in $M$ (using membership queries), it must not turn out to be equal to the equivalence class of $\gamma$ in $\hat{M}$, otherwise the algorithm as stated would abort. This situation is easily fixed by changing the algorithm so that if it gets this far it does not try to compute the equivalence class of $\gamma$, but instead uses the information that $\gamma$ was a counterexample to directly update the generalized classification tree as follows: the new access string is $\gamma[m-1]$ and the new distinguishing string is $\gamma_m$. The correctness of the generalized classification tree is unchanged except for the fact that the correctness of the pair $s = \gamma[m-1]$ and $d = \gamma_m$ relies on the fact $\gamma$ was a counterexample string after a reset operation, and therefore there must be some reset state $q \in \mathit{reset}(h, \sigma)$ such that $\mathit{state}(q, \gamma)$ is an accepting or rejecting string as claimed by the counterexample.

We have finally shown:

**Theorem 8.3** *There is a randomized algorithm that halts in expected polynomial time and exactly learns the representation class of deterministic finite automata in the no-reset learning model.*

It is easy to argue that we can alternatively state this result by saying that there is a randomized algorithm that takes as input $0 < \delta \leq 1$, and that with probability at least $1 - \delta$, exactly learns any deterministic finite automata $c$ in the no-reset model in time polynomial in $\log(1/\delta)$, $n$ and $size(c)$. Here $n$ is again a bound on the length of the longest counterexample to any equivalence query.

## 8.5 Exercises

8.1. Show that for any representation class $\mathcal{C}$, if $\mathcal{C}$ is efficiently exactly learnable from membership and equivalence queries, then $\mathcal{C}$ is efficiently learnable in the PAC model with membership queries.

8.2. Show that properties of the classification trees constructed by our algorithm for learning finite automata in the reset model imply that any two inequivalent states in any deterministic finite automata $M$ of $s$ states have a distinguishing string of length at most $s$. Show that for any equivalence query $\hat{M}$ of our algorithm, if $\hat{M} \neq M$ then there is a counterexample of length $2s$ which can be found efficiently on input $\hat{M}$ and $M$.

8.3. Let $\mathcal{C}_n$ be the class of monotone DNF formulae over $x_1, \ldots, x_n$, and let $\mathcal{C} = \cup_{n \geq 1} \mathcal{C}_n$. Give an algorithm for efficiently exactly learning $\mathcal{C}$ from membership and equivalence queries.

8.4. Consider modifying our algorithm for finite automata in the no-reset model so that the copy $L_\sigma$ is halted only when its generalized classification tree $T_\sigma$ has $2s$ leaves rather than just $s + 1$, where $s$ is the number of states in the target automaton. Note that this increases the running time of the algorithm by only a constant. Show that this modification increases the probability that we improve our candidate homing sequence from $1/s^2$ to $1/s$.

## 8.6 Bibliographic Notes

The model of exact learning with membership and equivalence queries, and the algorithm given here for learning finite automata, is due to Angluin [5]. Her seminal paper inspired a tremendous amount of subsequent research in the model, and has yielded many positive results. These include efficient algorithms for learning the class of decision trees, due to Bshouty [25]; for learning conjunctions of Horn clauses, a restricted form of DNF formulae, due to Angluin, Frazier and Pitt [7]; for learning a subclass of context-free languages accepted by counter machines, due to Berman and Roos [15]; for learning read-once boolean formulae, due to Angluin, Hellerstein and Karpinski [8]; for learning sparse multivariate polynomials, due to Schapire and Sellie [83]; and for many other concept classes. The algorithm for learning monotone DNF that is the subject of Exercise 8.3 is due to Angluin [6]; this paper also provides many general resource bounds for query learning (also see the work of Kannan [54]). The monotone DNF algorithm was subsequently extended by Angluin and Slonim [11] to tolerate certain types of errors in the query responses.

However, there are still limitations: Angluin and Kharitonov [9] demonstrate that for the class of DNF formulae, membership queries provide no additional power to the learner over the PAC model for some input distributions (under certain cryptographic assumptions), and subsequently Kharitonov [62] greatly strengthened the hardness results we derived in Chapter 6 when he proved that boolean formulae cannot be efficiently learned from random examples and membership queries, even when the input distribution is uniform (again under cryptographic assumptions).

The extension of Angluin's algorithm to the problem of learning finite automata without a reset mechanism is due to Rivest and Schapire [80], who also study learning algorithms using an alternative representation for finite automata based on a quantity called the diversity [79]. A recent paper of Freund et al. [37] gives algorithms for learning finite automata on the basis of a single long walk in an average-case setting.

There is actually a huge literature on finite automata learning problems that predates the computational learning theory work. While there was less explicit emphasis in this previous work on efficiency considerations, there are still many efficient algorithms and other fundamental results in the older literature. It is far too large to survey here, but the book of Trakhtenbrot and Barzdin' [90] provides a thorough investigation.