SIAM J. COMPUT.
Vol. 21, No. 2, pp. 240–266, April 1992

LEARNING INTEGER LATTICES*

DAVID HELMBOLD†, ROBERT SLOAN‡, AND MANFRED K. WARMUTH†

Abstract. The problem of learning an integer lattice of $\mathbf{Z}^k$ in an on-line fashion is considered. That is, the learning algorithm is given a sequence of k-tuples of integers and predicts for each tuple in the sequence whether it lies in a hidden target lattice of $\mathbf{Z}^k$. The goal of the algorithm is to minimize the number of prediction mistakes. An efficient learning algorithm with an absolute mistake bound of $k + \lfloor k \log(n\sqrt{k}) \rfloor$ is given, where n is the maximum component of any tuple seen. It is shown that this bound is approximately a $\log\log n$ factor larger than the lower bound on the worst case number of mistakes given by the VC dimension of lattices that are restricted to $\{-n, \cdots, 0, \cdots, n\}^k$.

This algorithm is used to learn rational lattices, cosets of lattices, an on-line word problem for abelian groups, and a subclass of the commutative regular languages. Furthermore, by adapting the results of [D. Helmbold, R. Sloan, and M. K. Warmuth, *Machine Learning*, 5(1990), pp. 165–196], one can efficiently learn nested differences of each of the above classes (e.g., concepts of the form $c_1 - (c_2 - (c_3 - (c_4 - c_5)))$, where each c_i is the coset of a lattice).

Key words. lattices, concept learning, mistake bounds

AMS(MOS) subject classifications. 68T05, 68Q25, 06B99

1. Introduction.

Integer lattices are one of the most basic combinatorial structures. An integer lattice is a nonempty set of k-tuples from $\mathbf{Z}^k$ that is closed under addition and subtraction. Let $\mathcal{L}^k$ be the concept class consisting of all integer lattices in $\mathbf{Z}^k$. In this paper we present an algorithm for learning $\mathcal{L}^k$, and prove that its performance is within a log log factor of optimal in the on-line model of learning, using the worst case number of mistakes as the performance criterion. Note that any learning algorithm with a good on-line mistake bound can be used as a subroutine to construct a PAC learning algorithm [17], [4], but some PAC learning algorithms have very poor mistake bounds.

Although the learning algorithm is of interest itself, this paper's major technical contributions are analyzing the learning performance of that algorithm and computing the VC dimension for the class of integer lattices (thus giving a lower bound on the worst case number of mistakes made by any learning algorithm). In particular, we prove that our learning algorithm has an absolute mistake bound of $k(1+\log(n\sqrt{k}))$,[1] where n is an upper bound on the absolute value of any component of any instance seen, and that no learning algorithm can have a mistake bound of less than $(1-\epsilon)k \ln n / \ln\ln n$ for any $\epsilon > 0$. Thus we achieve nearly optimal learning performance in a very strict model.[2]

* Received by the editors July 30, 1990; accepted for publication January 15, 1991.

† Department of Computer and Information Sciences, University of California, Santa Cruz, California 95060. The research of the first author was supported in part by a Regents' Junior Faculty fellowship and by Office of Naval Research grant N00014-85-K-0554. The research of the third author was supported by Office of Naval Research grant N00014-86-K-0454. Part of this work was done while the third author was visiting Aiken Computation Laboratory, Harvard University, and was supported by Office of Naval Research grant N00014-85-K-0554.

‡ Electrical Engineering and Computer Science Department, Box 4348, University of Illinois, Chicago, Illinois 60680. Much of this research was done while this author was visiting the Department of Computer and Information Sciences, University of California, Santa Cruz, and was supported by Office of Naval Research grant N00014-86-K-0454. This author's research was also supported by United States Army Research Office grant DAAL 03-86-K-0171 and by a National Science Foundation Graduate Fellowship.

[1] Throughout, log represents the base two logarithm, and ln represents the natural logarithm.

[2] Abe [1] considers learning the harder class of semilinear sets. Using different parameters, he presents a learning algorithm for semilinear sets when $k = 2$.

The algorithm we present keeps a basis for the "smallest" lattice containing all positive examples seen so far and predicts on new instances according to whether they are in this lattice or not. Although any reasonable basis can be used, our algorithm keeps a special basis which facilitates prediction and updates while storing only a small number of derived examples.

Our algorithm is similar to the algorithms of Kannan and Bachem [15] and Chou and Collins [25] for converting an integer basis into a special integer basis called a Hermite normal form (HNF). However, in our application we are not given all the vectors (positive instances) in advance, so we must convert the HNF algorithm into an on-line algorithm. Significant adaptations of the HNF algorithm and its analysis were required for our application, since we want to keep a HNF-like basis while efficiently processing new examples.

We also present several applications of the learning algorithm for $\mathcal{L}^k$. For example, it can be used to learn rational lattices, cosets of lattices, and an on-line word problem for abelian groups. Furthermore, we can use it to learn the subclass of commutative regular languages accepted by DFAs whose single final state reaches the start state. This subclass includes the class of "counter languages." The last result is surprising, because the counter languages are precisely the regular languages used to prove that the minimum consistent DFA problem for regular languages cannot be approximated within any polynomial [21] (see §6.5 for details). Finally, adapting the results from a companion paper [14], our algorithm can be applied to efficiently learn nested differences of all the above learnable classes.

2. Methods and outline.

The setting we will be concerned with is that of on-line learning. Formally, *concepts* are subsets of some *instance space* X from which instances are drawn and a *concept class* is a subset of 2^X, the power set over X. The instances are labeled *consistently* with a fixed *target concept* c which is in the *concept class* C to be learned; i.e., an instance is labeled "+" if it lies in the target concept and "−" otherwise. Labeled instances are called *examples.* An *on-line learning algorithm* interactively participates in a series of *trials.* On each trial, the algorithm gets an instance and predicts what its label is. After predicting, the on-line algorithm is informed of the instance's true label. A *mistake* is a trial where the on-line algorithm makes an incorrect prediction. Between trials, the algorithm's *hypothesis* is the subset of the instance space where the algorithm predicts "+."

2.1. Example: Learning vector subspaces.

Imagine for the time being that our instance space is an arbitrary vector space $\mathcal{S}$, and that our concept class is the set of all vector subspaces of $\mathcal{S}$. A natural on-line learning algorithm for this problem, shown to us by Haussler and inspired by a similar algorithm of Shvaytser [23] is described in Fig. 1.

2.2. Algorithm V is a closure algorithm.

The class of all subspaces of a vector space is intersection-closed, and Algorithm V is in fact a special case of an algorithm for learning intersection-closed concept classes called the "closure algorithm" [20], [6], [14].

DEFINITION. A concept class is *intersection-closed* if for any finite set contained in some concept, the intersection of all concepts containing the finite set is also a concept in the class.

DEFINITION. Fix an intersection-closed concept class $\mathcal{C}$ on some instance space. Let S be a set of positive examples of some concept from $\mathcal{C}$. The *closure* of S with respect to the concept class $\mathcal{C}$ (written CLOSURE(S) when $\mathcal{C}$ is understood) is the

Algorithm V

Predict that the zero vector is positive and all other vectors are negative until a mistake is made.
Whenever a mistake is made predicting the label of some instance, add that instance to a hypothesized basis set for the target subspace.
In general, predict that any instance in the span of the hypothesized basis is positive, and all other instances are negative.

FIG. 1. *An algorithm for learning subspaces of a vector space.*

intersection of all concepts in $\mathcal{C}$ containing the instances of S.

Thus the closure of a set of instances is the smallest concept containing all those instances. The *closure algorithm* is a generic on-line learning algorithm whose hypothesis is CLOSURE(POS) where POS is the set of positive examples seen so far. If the instance space is a vector space, and the concept class is the set of all subspaces, then the closure of a set of positive examples (vectors) is their span. Thus Algorithm V is a closure algorithm.

2.3. How good is closure algorithm V for learning subspaces? This paper uses the mistake-based performance criteria of Littlestone [16] to measure the performance of on-line learning algorithms. For any learning algorithm Q and target concept c define $M_Q(c)$ to be the maximum number of mistakes that Q makes on any possible sequence of instances labeled according to c. For any nonempty concept class C, define $M_Q(C) = \max_{c \in C} M_Q(c)$. Any bound on $M_Q(C)$ is called an (*absolute*) *mistake bound* for algorithm Q applied to class C. The *optimal mistake bound* for concept class C, $opt(C)$, is the minimum over all learning algorithms Q of $M_Q(C)$.

One lower bound on the mistake bound of any learning algorithm for a concept class is given by a very useful combinatorial parameter, the Vapnik–Chervonenkis (VC) dimension [24].

DEFINITION. A set of instances S is *shattered* (by the concept class C) if for each subset $S' \subseteq S$, there is a concept $c \in C$ that contains all of S', but none of the instances in $S - S'$. The *Vapnik–Chervonenkis dimension* of concept class C, denoted by VCdim(C), is the cardinality of the largest set shattered by the concept class.

Littlestone [16] has given the following relationship between the VCdim(C) and the optimal mistake bound.

THEOREM 2.1. *For every concept class C,* VCdim(C) $\leq opt(C)$.

It is fairly easy to see that the closure algorithm V, has an absolute mistake bound of the vector space dimension of $\mathcal{S}$ when learning subspaces of vector space $\mathcal{S}$. Each time a mistake is made, a new vector is added to the basis set Algorithm V maintains, and this can happen at most as many times as the vector space dimension of $\mathcal{S}$. The VC dimension of this concept class is also the vector space dimension of $\mathcal{S}$, so in this case the closure algorithm is optimal.

2.4. The closure algorithm is not always good. Even if a concept class is not intersection-closed, it can always be embedded in one that is. Thus the closure algorithm applies to any concept class. However, there are a number of potential problems:

1. There might not be an efficient algorithm for computing the closure of a set of instances.

2. Given a representation of the closure, it might be difficult to predict whether a new instance is in the closure.
3. The closure algorithm might not have a good mistake bound.

For example, regular sets are intersection-closed. The closure of a finite set of words equals the set itself and thus the closure algorithm makes a mistake on each new positive instance. So here the first two problems do not arise, but the third problem does. If we restrict the concept class to regular sets representable by DFAs with at most s states, then the class is no longer intersection-closed.

Theorem 2.1 implies that any algorithm makes at least as many mistakes as the VC dimension. However, even when the concept class is intersection-closed and has a small VC dimension, the closure algorithm can still make a large number of mistakes. For example, if the instance space is $0, \cdots, n$ and the concepts are initial segments (i.e., the intervals $[0, i]$ for $1 \leq i \leq n$), then the mistake bound of the closure algorithm is n (consider increasing sequences of positive examples) even though the concept class has VC dimension 1.

2.5. Applying the closure algorithm to integer lattices. As in the case of vector spaces, the concept class $\mathcal{L}^k$ of integer lattices of $\mathbf{Z}^k$ is intersection-closed. For any set $S \subseteq \mathbf{Z}^k$, the closure of S is the set of all k-tuples produced by summing integer multiples of elements in S. Thus the generic closure algorithm can be used to learn $\mathcal{L}^k$. However, lattices are significantly more complicated than vector spaces. Any implementation of the closure algorithm must take into account the "holes" in the lattice. For example, consider the lattice generated by the basis $(2, 2)$ and $(0, 2)$. Although the point $(1, 2)$ can be written as $\frac{1}{2}(2, 2) + \frac{1}{2}(0, 2)$, it is not in the lattice, as it is not an *integer* linear combination of the basis vectors. Determining if a point is in a lattice involves solving a set of linear Diophantine equations rather than inverting a matrix.

We give a particular implementation of the closure algorithm, called Algorithm A, which overcomes the three potential problems stated in the previous section.

1. Algorithm A (presented in the Appendix) efficiently computes closures with respect to $\mathcal{L}^k$.
2. Algorithm A represents closures so that prediction can be done efficiently.
3. We prove a good mistake bound for the closure algorithm when applied to lattices.

We show in §3 below that VCdim($\mathcal{L}^k$) is infinite. Thus we know by Theorem 2.1 that the mistake bound of any algorithm must be infinite. To overcome this difficulty, we restrict our attention to $\mathcal{L}^k(n)$, which we define to be the class $\mathcal{L}^k$ where instances are restricted to $\{-n, \cdots, 0, \cdots, n\}^k$. The concept class $\mathcal{L}^k(n)$ is also intersection-closed, as is the restriction of any intersection-closed class to some subset of its domain. Theorem 3.3 below shows that VCdim($\mathcal{L}^k(n)$) is roughly $k \ln n / \ln \ln n$.

In this paper we prove that the absolute mistake bound of the closure algorithm when learning $\mathcal{L}^k(n)$, $M_{\text{CLOSURE}}(\mathcal{L}^k(n))$, is at most $k + \lfloor k \log(n\sqrt{k}) \rfloor$. This bound is only a factor of roughly $\log \log n$ larger than the lower bound on the worst case number of mistakes given by the VC dimension of $\mathcal{L}^k(n)$.

Remark. We will not actually limit the size of the input to the closure algorithm; we will simply be describing its performance as a function of the size of its inputs.

2.6. Our algorithm and its advantages. Algorithm A (fully described in the Appendix) for learning $\mathcal{L}^k(n)$ is a particular implementation of the closure algorithm since its hypothesis is always the closure of the positive examples seen so far. Thus Algorithm A has the same mistake bound as the generic closure algorithm. Our algorithm, however, has some space and computational advantages over the obvious implementations of the closure algorithm.

One obvious implementation of the closure algorithm saves all previous mistakes. After each mistake it recomputes a lower triangular basis for the smallest lattice containing the positive examples using the HNF algorithms of [15], [25]. Prediction is done by back substitution in this lower triangular matrix. Note that the number of mistakes can be at least as large as the VC dimension. As we shall see in §3, the VC dimension is bounded below by approximately $k \ln n / \ln \ln n$, thus this obvious implementation of the closure algorithm requires storing about $k \ln n / \ln \ln n$ positive examples.

In contrast, Algorithm A stores only the lower triangular basis (k derived positive examples). Rather than recomputing the basis from scratch after each mistake, it is updated on-line. The analysis of Algorithm A is nontrivial and is included in the Appendix. One of the more difficult parts is bounding the number of bits required to represent a derived example. The other contribution of this paper is the application of Algorithm A to the problems described in §6.

2.7. Outline of the paper. In the following section we compute the VC dimension of $\mathcal{L}^1(n)$, lattices in one dimension, and bound the VC dimension of $\mathcal{L}^k(n)$. In §4 we compute lower bounds on $opt(\mathcal{L}^k(n))$, the minimum mistake bound any algorithm can achieve when learning $\mathcal{L}^k(n)$. The closure algorithm's mistake bound is then analyzed in §5. At the end of the section, we show that a modified version of the halving algorithm [5], [19], [4] has a nearly optimal mistake bound when learning $\mathcal{L}^1(n)$. Section 6 shows how Algorithm A can be extended to learn rational lattices, cosets of lattices, and a word problem for abelian groups. We conclude §6 by showing how Algorithm A can be applied to learning a subclass of commutative regular languages. In §7 we discuss how Algorithm A can be used in conjunction with a master algorithm for learning nested differences. Our master algorithm is a modification of a similar algorithm presented in a companion paper [14]. It leads to efficient learning algorithms for nested differences of any of the concept classes that we learn using Algorithm A. A short summary of our results is given in the concluding section. In the Appendix we formally state Algorithm A and prove bounds on its resource requirements.

3. VC dimension of integer lattices. This section contains bounds on the VC dimension of the concept class $\mathcal{L}^k(n)$. The VC dimension provides a lower bound on both the number of examples stored by the standard implementation of the closure algorithm [14], and on the mistake bound of any learning algorithm.

We begin by exactly calculating the VC dimension of $\mathcal{L}^1(n)$. Note that each concept in $\mathcal{L}^1(n)$ can be represented by an integer between 0 and n. The concept represented by j contains the subset of $-n, \cdots, n$ whose members are multiples of j.

THEOREM 3.1. *For all $n \geq 1$,*

$$\mathrm{VCdim}(\mathcal{L}^1(n)) = \max\{r \mid 2 \cdot 3 \cdot 5 \cdots p_r \leq 2n\},$$

where p_i is the ith prime.

We start with a definition we will need in our proof.

DEFINITION. Let S be any shattered set; let $T \subseteq S$. We call a concept c such that $T = c \cap S$ a *witness* for T.

Proof. Let r be maximum such that $P = \prod_{i=1}^{r} p_i \leq 2n$ and $d = \mathrm{VCdim}(\mathcal{L}^1(n))$. Now we show $d \geq r$ by exhibiting set S with $|S| = r$ that is shattered: S is all products of all but one of the first r primes. In symbols, $S = \{P/p_i \mid 1 \leq i \leq r\}$. Since $P/p \leq n$ for all primes p, every element of S is in the instance space. For every $x = P/p_i$ in S, define $\hat{x} = p_i$. It is easy to see that S is shattered: The witness for any nonempty $T \subseteq S$ is $P/\prod_{x \in T} \hat{x} = \prod_{x \notin T} \hat{x}$. The witness for S is 1, and the witness for the empty set is 0.

Hence $d \geq r$. Now we show that $d \leq r$.

Let $S = \{x_1, x_2, \cdots, x_d\}$ be a largest shattered set. First we argue that we may assume that there is no $s > 1$ that divides all elements of S. If there is such an s, we can work instead with $S' = \{x_1/s, x_2/s, \cdots, x_d/s\}$ and divide each witness by s.

Call any subset of $d-1$ elements of S a *minor*. Set S has d minors, S_1 through S_d (where S_i is the minor *not* containing x_i). Let t_i be a witness for S_i.

Now no t_i can be 1, since $t_i \not| x_i$. (Note that $0 \notin S$, because 0 is a positive example of every concept and S is shattered.) Furthermore, $\gcd(t_i, t_j) = 1$ for all $i \neq j$, since $\gcd(t_i, t_j) \mid x$ for every $x \in S$, and we assumed that 1 is the only number with this property. Thus it must be that for each i there is a prime p_i such that $p_i \mid t_i$ but $p_i \not| t_j$ for any $j \neq i$.

Pick any odd $x \in S$. Element x is in $d-1$ minors of S so $d-1$ different t_i's divide x, and x is a multiple of at least $d-1$ different p_i's. Since $x \leq n$ and $2 \not| x$, there are d distinct primes ("2" plus the $d-1$ primes dividing x) whose product is at most $2n$, thus $d \leq r$. □

In order to obtain numerical bounds on the VC dimension, we recall some facts from number theory (see, e.g., Hardy and Wright [13, pp. 262–263]):

1. Let $f(n)$ be the maximum number of consecutive primes such that $\prod_{i=1}^{f(n)} p_i \leq n$. Then for every $\epsilon > 0$, for all sufficiently large n we have

$$f(n) > \frac{(1-\epsilon)\ln n}{\ln\ln n}. \tag{1}$$

2. For all $\epsilon > 0$, for all sufficiently large m, we have

$$\log(\tau(m)) < \frac{(1+\epsilon)\ln m}{\ln\ln m}, \tag{2}$$

where $\tau(m)$ is the number of positive divisors of m.

COROLLARY 3.2. *For all ϵ, for sufficiently large n,*

$$\frac{(1-\epsilon)\ln n}{\ln\ln n} < \mathrm{VCdim}(\mathcal{L}^1(n)) < \frac{(1+\epsilon)\ln n}{\ln\ln n}.$$

Proof. The left inequality follows directly from Theorem 3.1 together with (1), once we note that $\ln n/\ln\ln n < \ln 2n/\ln\ln 2n$.

For the right inequality, we need to bound $f(2n)$, where f is the function specified in (1). Let m be a particular integer of the form $m = 2\cdot 3\cdot 5\cdots p_r$. Note that for such an m we have $\log(\tau(m)) = f(m)$, and thus $\max_{m\leq n}\log(\tau(m)) \geq f(n)$. Thus we have an upper bound on $f(n)$ in terms of the log of the number of divisors of any $m \leq n$, and can apply (2) to get the desired result. □

Remark. The preceding should make it clear that $\mathrm{VCdim}(\mathcal{L}^1(n))$ can be made arbitrarily large by choosing a suitable value for n, and thus that $\mathrm{VCdim}(\mathcal{L}^1)$ is infinite.

We now get a lower bound on the VC dimension of the more general concept class $\mathcal{L}^k(n)$.

THEOREM 3.3.

$$k + \left\lfloor k \log(n\sqrt{k}) \right\rfloor \geq \mathrm{VCdim}(\mathcal{L}^k(n)) \geq k\mathrm{VCdim}(\mathcal{L}^1(n)).$$

Proof. The first inequality is Corollary 5.3, proven later. The proof of the second inequality is essentially a particular case of a bound of Dudley's on the VC dimension of cross products of those concept classes, among them $\mathcal{L}^1(n)$, that have a certain property he calls "being bordered" [9, Thm. 9.2.14].

Let S be the exhibited set of numbers shattered in the proof of Theorem 3.1. Let U be the set of size $k\,|S|$ consisting of all k-tuples of integers containing $k-1$ zeros and one value from S. In this case witnesses are sets of k-tuples. To shatter any $T \subseteq U$, first write $T = T_1 \cup T_2 \cup \cdots \cup T_k$, where all elements of T_i have nonzero values only in position i. For each T_i, let t_i be the number that is the witness for the set of all nonzero components of all elements of T_i (viewed as a concept from $\mathcal{L}^1(n)$ as in the proof of Theorem 3.1). The witness for T is then the set of all integer combinations of the k tuples that have t_i in the ith position and 0's elsewhere. □

The lower bound in Theorem 3.3 is not tight. For example, $\mathrm{VCdim}(\mathcal{L}^1(2)) = 1$, but $\mathrm{VCdim}(\mathcal{L}^2(2)) = 3$ since $(1,2)$, $(2,1)$, and $(2,2)$ are shattered. Determining tight bounds on $\mathrm{VCdim}(\mathcal{L}^k(n))$ remains an open problem.

4. Lower bounds on learning $\mathcal{L}^k(n)$.

We know from Theorem 2.1 that $opt(\mathcal{L}^k(n)) \geq \mathrm{VCdim}(\mathcal{L}^k(n))$, so from Theorem 3.3 together with Corollary 3.2, we get a first lower bound for $opt(\mathcal{L}^k(n))$.

COROLLARY 4.1. *For all $\epsilon > 0$, for sufficiently large n,*

$$opt(\mathcal{L}^k(n)) \geq k\,(1-\epsilon)\,\frac{\ln n}{\ln\ln n}.$$

For sufficiently large n we can get a slightly better lower bound on $opt(\mathcal{L}^1(n))$ than $\mathrm{VCdim}(\mathcal{L}^1(n))$ using an adversary argument.[3]

THEOREM 4.2.

$$opt(\mathcal{L}^1(n)) \geq \max_{m \leq n} \sum_{i=1}^{s} \lfloor \log(e_i + 1) \rfloor$$

where $m = \prod_{i=1}^{s} p_i^{e_i}$.

Proof. Let $m = \prod p_i^{e_i}$ be a number less than or equal to n with a maximal number of divisors. Our adversary begins by first giving m as a positive instance. The adversary then makes the algorithm perform a search similar to binary search for the value of each exponent as follows. The next group of instances begin with $p_1^{\lfloor e_1/2 \rfloor} \prod_{i \geq 2} p_i^{e_i}$, and continues with various exponents for p_1 (times $\prod_{i\geq 2} p_i^{e_i}$). The algorithm is forced to make $\lfloor \log(e_1+1) \rfloor$ mistakes since there are $e_1 + 1$ choices for the exponent of p_1. The following group of instances all have the exponent of p_1 set to its correct value, the exponents of each p_i for $i \geq 3$ set to e_i, and force the algorithm to search for the value of the exponent of p_2. Next comes a group of instances forcing the algorithm to search for the exponent of p_3, and so on. □

[3] To be precise, the adversary argument gives a stronger bound on $opt(\mathcal{L}^k(n))$ for all $n \geq 2^7 \cdot 3^3 \cdot 5 \cdot 7 \cdot 11 \cdots 31 \approx 8.9 \times 10^{13}$. After this point, $\max_{m \leq n} \sum_{i=1}^{s} \lfloor \log(e_i + 1) \rfloor$ is strictly greater than $\max\{r \mid 2 \cdot 3 \cdot 5 \cdots p_r \leq 2n\}$.

COROLLARY 4.3.

$$opt(\mathcal{L}^k(n)) \geq k \left(\max_{m \leq n} \sum_{i=1}^{s} \lfloor \log(e_i + 1) \rfloor \right)$$

where $m = \prod_{i=1}^{s} p_i^{e_i}$.

Proof. The method used by the adversary in the proof of Theorem 4.2 can be easily extended to the general case. There are k rounds; in the ith round the adversary gives instances with all components but the ith set to 0. The values of the ith component are chosen according to the adversary strategy in the proof of Theorem 4.2. □

5. Mistake bounds. This subsection calculates $M_{\text{CLOSURE}}(\mathcal{L}^k(n))$, the mistake bound of the closure algorithm, and considers how close it is to $opt(\mathcal{L}^k(n))$. We will bound $M_A(\mathcal{L}^k(n)) = M_{\text{CLOSURE}}(\mathcal{L}^k(n))$ by noticing that every time the algorithm makes a mistake, its new hypothesis is a strict superset of its old hypothesis. At the end of the section we analyze a modified halving algorithm for the special case when $k = 1$.

Before stating the main theorem, we recall some facts about lattices. The standard definition of lattices in $\Re^k$ insists that the lattice be generated by k linearly independent basis vectors. Our definition allows less than k basis vectors, thus our lattices need not have full "rank."

DEFINITION. The *rank* of lattice $\Lambda \subseteq \mathbf{Z}^k$ is the minimum of the ranks of all vector subspaces of $\Re^k$ that contain Λ.

Thus any lattice of rank r can be written as $\{\sum_{i=1}^{r} z_i x_i \mid z_i \in \mathbf{Z}\}$ for some r basis vectors $x_i \in \mathbf{Z}^k$.

It is easy to see that any integer lattice in $\mathbf{Z}^k$ with rank $r < k$ can be rotated into $\Re^r$ (i.e., the last $k - r$ coordinates of every point in the rotated lattice are zeros). For every set of r basis vectors that determine the same lattice, the volume of the r-dimensional parallelepiped that they generate is the same. Furthermore, rotating the lattice into $\Re^r$ preserves volume. The volume of the parallelepiped is called the *determinant* of the lattice [7]. We write $\det \Lambda$ to denote the determinant of lattice Λ. If $\Lambda \subseteq \mathbf{Z}^k$, then $\det \Lambda = 0$ only if Λ is O, the null lattice containing only the origin. Otherwise, $\det \Lambda$ is a positive integer.

THEOREM 5.1. *Let* $O = \Lambda_0 \subset \Lambda_1 \subset \Lambda_2 \subset \cdots \subset \Lambda_m$ *be a sequence of distinct lattices of* $\mathbf{Z}^k$ *where each* Λ_i, *for* $1 \leq i \leq m$, *is the closure of* Λ_{i-1} *plus some* $x_i \in (\mathbf{Z}^k \setminus \Lambda_{i-1})$ *and every component of every* x_i *has absolute value at most* n. *Then*

$$m \leq k + \left\lfloor k \log(n\sqrt{k}) \right\rfloor .$$

Proof. There are at most k values of i for which Λ_{i+1} can have greater rank than Λ_i.

Consider now the case where Λ_i and Λ_{i+1} have the same rank. Since Λ_i is a sublattice of Λ_{i+1}, we have $t \det \Lambda_{i+1} = \det \Lambda_i$ for some integer $t \geq 2$ [7].[4] Thus every time the rank of the lattice stays the same, we decrease the determinant by at least a factor of 2.

On the other hand, when the rank of Λ_{i+1} is greater than the rank of Λ_i, then the determinant can increase. The first lattice, Λ_1, is simply all integer multiples of some particular $x_1 \in \mathbf{Z}^k$, so its volume is $\|x_1\| \leq n\sqrt{k}$, where $\|x\|$ denotes the

[4] In geometry of numbers, t is called the index of Λ_i in Λ_{i+1}.

Euclidean norm of vector x. In general, when Λ_i has rank r and Λ_{i+1} has rank $r+1$, one set of basis vectors for Λ_{i+1} will be x_{i+1} together with the basis vectors of Λ_i. The $(r+1)$-dimensional volume of the parallelepiped associated with Λ_{i+1} is $\det \Lambda_i$ times the distance from x_{i+1} to the hyperplane containing Λ_i. Hence we have

$$\begin{aligned} \det \Lambda_{i+1} &\leq \|x_{i+1}\| \det \Lambda_i \\ &\leq n\sqrt{k} \det \Lambda_i . \end{aligned}$$

Thus $\det \Lambda_1 \leq n\sqrt{k}$, and the determinant is multiplied by no more than $n\sqrt{k}$ in at most $k-1$ other steps. In all other steps the determinant is divided by at least 2, and $\det \Lambda_m \geq 1$. Therefore $m \leq k + \lfloor k \log(n\sqrt{k}) \rfloor$. □

COROLLARY 5.2. $M_{\text{CLOSURE}}(\mathcal{L}^k(n)) \leq k + \lfloor k \log(n\sqrt{k}) \rfloor$.

Comparing Corollary 5.2 to Corollary 4.1 we see that the mistake bound achieved by the closure algorithm is indeed within a $\log \log n$ factor of optimal (assuming $n \gg k$).

By giving example sequences for which the closure algorithm makes the maximum number of mistakes we next show that for infinitely many choices of k and n, the bounds of Theorem 5.1 and Corollary 5.2 are tight. These sequences are constructed using Hadamard matrices. A *Hadamard matrix* is a square matrix where the entries are ± 1 and the columns are orthogonal to each other. Using n times the columns of a $k \times k$ Hadamard matrix, one can force the closure algorithm to make k mistakes while the volume of the closure algorithm's hypothesis grows to $(\sqrt{kn^2})^k$. Let n be a power of 2 and k a power of 4, so this volume is a power of 2. Consider now a $k \times k$ lower-triangular matrix whose columns form a basis for the hypothesized lattice. Since the determinant of this matrix is the product of the diagonal elements and equals the volume of the lattice, each of the diagonal elements is a power of 2. In particular, the element in the lower-right corner is some $\pm 2^l$. We can reduce this element by a factor of two (without changing the other diagonal elements) by giving the closure algorithm the positive example $(0, 0, \cdots, 0, 2^{l-1})$. After the corner element is reduced to ± 1, we can start reducing the diagonal element on the second-to-last row, and so on. The number of additional mistakes made by the closure algorithm is $\log(\sqrt{kn^2})^k = k \log(n\sqrt{k})$. Thus for infinitely many choices of k and n, the bound of Theorem 5.1 and Corollary 5.2 can be achieved.

Surprisingly, the mistake bound of the closure algorithm (together with Theorem 2.1) gives us our tightest upper bound on the VC dimension of $\mathcal{L}^k(n)$.

COROLLARY 5.3. $\text{VCdim}(\mathcal{L}^k(n)) \leq k + \lfloor k \log(n\sqrt{k}) \rfloor$.

For the case $k = 1$, when instances are integers and concepts are sets of multiples, instead of using Algorithm A, we can implement a modified version of the halving algorithm [5], [19], [4]. This modified halving algorithm has a basically optimal mistake bound in this case, but requires more computation than the closure algorithm. The modified halving algorithm predicts "−" on every instance except 0 until it makes a mistake on some instance m. The target concept is then known to be all multiples of one of the divisors of m. The modified halving algorithm predicts so that each time it makes a mistake, the number of possible target concepts is cut by at least half (i.e., it predicts as the majority of the remaining target concepts do). Note that the modified halving algorithm factors m, but that is one factorization for the whole run of the algorithm, rather than one per mistake.

THEOREM 5.4. *On instances of absolute value at most n, the modified halving algorithm achieves a mistake bound of* $1 + \max_{m \leq n} \lfloor \log \tau(m) \rfloor$ *where $\tau(m)$ is the number of positive divisors of m.*

Proof. The algorithm makes one mistake on the first positive example it sees. Call this example m. Without loss of generality, $0 \le m \le n$. Once m has been seen, the divisors of m are the only candidates for being the target concept, and the modified halving algorithm cuts the number of candidates by at least half with every mistake. Hence the mistake bound is, as claimed, $1 + \max_{m \le n} \lfloor \log \tau(m) \rfloor$. $\square$

Note that the bound of the theorem can be rewritten as

$$1 + \max_{m \le n} \left\lfloor \sum_{i=1}^{s} \log(e_i + 1) \right\rfloor ,$$

where $m = \prod_{i=1}^{s} p_i^{e_i}$. Thus the mistake bound of the modified halving algorithm is just slightly above Theorem 4.2's lower bound of $\max_{m \le n} \sum_{i=1}^{s} \lfloor \log(e_i + 1) \rfloor$.

6. Generalizations and applications of Algorithm A.

6.1. Arbitrary Euclidean domains. We need not limit ourselves to lattices of $\mathbf{Z}^k$. Algorithm A can in fact learn submodules of a free D-module for any Euclidean domain D. Careful examination of the algorithm shows that it does not rely on any properties of $\mathbf{Z}$ not possessed by all Euclidean domains. This generalization gives us learning algorithms for various exotic instance spaces such as k-tuples of Gaussian integers. Also, if we take D to be a field, then Algorithm A becomes Algorithm V for learning vector subspaces.

Unfortunately, however, the analysis of the mistake bound given in Theorem 5.1 no longer carries through. The difficulty is that we can no longer find a bound on any quantity analogous to the determinant of a lattice when we go from one module to a supermodule of higher rank.

6.2. Rational lattices. A slight modification of Algorithm A learns rational lattices (where the basis vectors consist of rationals rather than integers). After the derived examples are written with a common denominator, Algorithm A can be used whenever a mistake is made. By the argument of Theorem 5.1, the modified Algorithm A has a mistake bound of k plus the maximum number of times the determinant (volume) of the hypothesized lattice can be divided by an integer greater than 1. The determinant of the hypothesis is upper bounded by $(\sqrt{kn^2})^k$ where n is the largest component seen by the algorithm and lower bounded by v, the determinant of the target lattice. Therefore, the algorithm makes at most $k + \lfloor k \log(n\sqrt{k}) - \log v \rfloor$ mistakes. However, finding a common denominator and operating on the consequently larger numerators makes the modified algorithm computationally more expensive than Algorithm A applied to integer lattices.

6.3. Cosets of lattices and Algorithm A^+. A simple trick allows us to generalize the class of concepts we can learn from lattices to arbitrary cosets of lattices (viewing the lattice as an abelian group). The algorithm still responds "Negative" until it makes a mistake on some positive instance x. We now run the basic Algorithm A given in the Appendix, with the addition that x is subtracted from every instance. For the remainder of the paper, we use "A^+" to denote this modification of Algorithm A. Note that the mistake bound of Algorithm A^+ when learning cosets of $\mathcal{L}^k$ restricted to the domain $\{-n, \cdots, 0, \cdots, n\}^k$ is at most 1 plus the mistake bound of Algorithm A on $\mathcal{L}^k(2n)$ (the subtraction doubles the bounds on the size of the components). This leads to a mistake bound of

$$1 + k + \left\lfloor k \log(2n\sqrt{k}) \right\rfloor = 1 + 2k + \left\lfloor k \log(n\sqrt{k}) \right\rfloor$$

for Algorithm A^+ which is $k+1$ larger than Algorithm A's mistake bound when learning $\mathcal{L}^k(n)$.

6.4. Abelian groups. Consider the following on-line word problem for groups over a set of k generators:

> Given a sequence of words using the generators and their inverses as letters, predict for each word whether it is equal to some fixed element with respect to a hidden target group over the generators.

The goal is to minimize the number of mistakes.

In the case of abelian groups, words over the generators and their inverses can be represented as k-tuples. All words that are equal to some particular element with respect to a hidden abelian group form the coset of an integer lattice of $\mathbf{Z}^k$. Thus Algorithm A^+ leads to an efficient solution to this learning problem for abelian groups with a mistake bound of $1 + 2k + \lfloor k \log(n\sqrt{k}) \rfloor$, where n is a bound on the maximum word length of all instances seen. Since the word problem for general groups is undecidable, it is unlikely that there is an efficient learning algorithm for nonabelian groups. (See [10] for related work on permutation groups.)

6.5. Learning some commutative regular languages. We call a language L over alphabet Σ *commutative* if whenever some word w is in L, then all permutations of w are also in L. When learning such languages we can represent words as k-tuples of nonnegative integers, where $k = |\Sigma|$. Each component of the k-tuple equals the number of times the corresponding letter occurs. We use π to denote this mapping from words to tuples. If language L is commutative and $\pi(w) = \pi(w')$, then word $w' \in L$ if and only if $w \in L$. Therefore learning a commutative language is similar to learning a set of tuples that have nonnegative components.

For any subset S of $\mathbf{Z}^k$, we call the set of all tuples in S that have only nonnegative components the *nonnegative restriction* of S. If we have a class of languages where, for each language in the class, the image under π of the words in the language is the nonnegative restriction of a coset of a lattice, then Algorithm A^+ can be used to efficiently learn that class of languages. Simply use π to convert each word into a tuple and learn the tuples. The hypothesis of Algorithm A^+ may include tuples with negative components, however, no mistakes will be made on these tuples as they are not in the domain. The bulk of this subsection is devoted to showing that the images under π of a certain subclass of commutative regular languages (defined below) are the nonnegative restrictions of cosets of lattices, and thus can be efficiently learned by Algorithm $A^+$with a mistake bound of $1 + 2|\Sigma| + \lfloor |\Sigma| \log(n\sqrt{|\Sigma|}) \rfloor$.[5]

Note that many commutative nonregular languages can also be learned using Algorithm A. For example, the language over a, b containing all words with one more a than b is commutative but not regular, and its image under π is the coset of a lattice.

We now present several definitions concerning DFAs.

DEFINITION. A DFA is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$ where $q_0 \in Q$, $F \subset Q$, and δ is a partial function from $Q \times \Sigma$ to Q.[6] The elements of Q are called *states*; δ is called the *transition function*; q_0 is called the *start* state; and the states in F are called

[5] Abe has recently [2] discovered a learning algorithm (using different parameters) for arbitrary commutative regular languages. The mistake bound of his algorithm grows exponentially in $|\Sigma|$.

[6] We extend the transition function to words in the obvious way; if $a \in \Sigma$ and $w = aw'$, then $\delta(q, w) = \delta(\delta(q, a), w')$.

final states. The language *accepted* by this DFA is the set of all words w such that $\delta(q_0, w) \in F$.

DEFINITION. A DFA $M = (Q, \Sigma, \delta, q_0, F)$ is *closed* if for each final state $q_f \in F$ there is some word w_f such that $\delta(q_f, w_f) = q_0$.

DEFINITION. We say a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is *canonical* if it has the following properties:

1. Every state can reach a final state.
2. If the language accepted by M is commutative, then we also require that for all states q and all words w and w', $\delta(q, ww') = \delta(q, w'w)$.

Given a closed DFA, one can create (by deleting and merging states) a closed canonical DFA that accepts the same language.

In the remainder of this section Σ denotes the set of useful letters—those on which the transition function is defined for some state, and k denotes the cardinality of Σ.

DEFINITION. A DFA $M = (Q, \Sigma, \delta, q_0, F)$ is *invertible*[7] if for each letter in Σ, every state in Q has both an incoming and an outgoing transition for that letter.

LEMMA 6.1. *Any closed canonical* DFA *accepting a commutative regular language is invertible.*

Proof. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a closed canonical DFA accepting a commutative language L. It suffices to show that every state in Q has at least one incoming transition for every letter of Σ. Since each state has at most one outgoing transition for every letter of Σ, this implies by the pigeonhole principle that for each state there is exactly one incoming and exactly one outgoing transition for every letter.

Let a be an arbitrary letter in Σ and q be an arbitrary state in Q. Since every state reaches a final state in F there is a word u_1au_2 that is accepted, i.e., $\delta(q_0, u_1au_2) = q_f \in F$. Let v be a word leading from the start state to q, and w be a word leading from q_f to the start state. Now u_1au_2wv leads to q and since u_1u_2wva is a permutation of the latter word it leads to q as well. Thus for the state $q' = \delta(q_0, u_1u_2wv)$, we have $\delta(q', a) = q$. □

Remark. Note that exactly one incoming and one outgoing transition per letter property does not hold if we remove the condition that the start state be reachable from the final states. For instance, the canonical DFA for $L_a = \{a\}$ has no outgoing transitions from its final state, even though L_a is commutative.

With an invertible DFA, one can talk about the inverses of letters. For the alphabet $\Sigma = \{\sigma_1, \cdots, \sigma_k\}$ we define the inverse alphabet Σ^{-1} to be the set of new symbols $\{\sigma_1^{-1}, \cdots, \sigma_k^{-1}\}$. The *extended alphabet* is $\Sigma \cup \Sigma^{-1}$.

DEFINITION. The *extension* of an invertible DFA $M = (Q, \Sigma, \delta, q_0, F)$ is the DFA $M' = (Q, \Sigma \cup \Sigma^{-1}, \delta', q_0, F)$, where for each $a \in \Sigma$ and $q \in Q$:

$$\delta'(q, a) = \delta(q, a),$$
$$\delta'(q, a^{-1}) = r \text{ such that } \delta(r, a) = q.$$

LEMMA 6.2. *For any closed canonical* DFA M *accepting a commutative language* L*, the language accepted by the extension of* M *is commutative.*

[7] Invertible DFAs with a single final state are *zero-reversible.* Angluin [3] presents an algorithm for learning the languages accepted by zero-reversible DFAs (as well as for the more general class of r-reversible DFAs). However, those algorithms learn in the limit and no bound was given on the number of mistakes made.

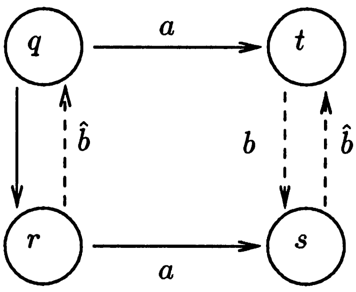

FIG. 2. *Detail of proof of Lemma* 6.2: $\delta(q,ab) = \delta(q,ba)$. *Dashed lines indicate the deduced transitions.*

Proof. Let $M = (Q, \Sigma, \delta, q_0, F)$ be such a DFA. By Lemma 6.1, each state has exactly one incoming and one outgoing transition for each letter in Σ. This means that M' is a DFA with transitions defined at each state for every letter in $\Sigma \cup \Sigma^{-1}$.

Now we argue that the language accepted by M' is commutative. It is sufficient to show that for all $q \in Q$, $\delta'(q,ab) = \delta'(q,ba)$ for arbitrary $a, b \in \Sigma \cup \Sigma^{-1}$. There are three cases:

1. Both $a, b \in \Sigma$. In this case, δ' is the same as δ.
2. Both $a, b \in \Sigma^{-1}$. Say $a = \hat{a}^{-1}$ and $b = \hat{b}^{-1}$, where $\hat{a}, \hat{b} \in \Sigma$. Now $\delta'(q,ab)$ must be some state r such that $\delta(r, \hat{b}\hat{a}) = q$. Since M is a canonical machine for a commutative language, $\delta(r, \hat{a}\hat{b}) = \delta(r, \hat{b}\hat{a}) = q$. Therefore $\delta'(q,ba) = r$ as well.
3. Let the letter $a \in \Sigma$, and the letter $b \in \Sigma^{-1}$. Again, say $b = \hat{b}^{-1}$ where $\hat{b} \in \Sigma$. By Lemma 6.1, every state has an entering and exiting transition for each letter in Σ, and hence for every letter in Σ^{-1} as well. Now let $r = \delta'(q,b)$, let $s = \delta'(r,a)$, and let $t = \delta'(q,a)$. $\delta'(q,ba)$ is obviously s. Now we need to calculate $\delta'(q,ab) = \delta'(t,b)$.
   By the commutativity of δ', $\delta(r, a\hat{b}) = \delta(r, \hat{b}a)$. Now since $r = \delta'(q,b)$, it follows that $\delta(r,\hat{b}) = q$. Therefore $t = \delta(r, \hat{b}a) = \delta(r, a\hat{b}) = \delta(s, \hat{b})$. Thus, as desired, $s = \delta'(t,b)$. (See Fig. 2.) □

Remark. Again, the property that the start state is reachable from the final states is crucial. Even if each state in a commutative canonical DFA has at most a single incoming transition, the extension of the DFA may not be commutative. Consider the two-state extended DFA for the commutative language $\{a\}$. The string $aa^{-1}a$ is accepted by that DFA, but the string $a^{-1}aa$ is not, because there is no transition out of the start state for a^{-1}.

When the extended DFA is commutative, it makes sense to talk about its behavior given simply letter counts from the extended alphabet, rather than specific words. We extend π as follows:

$$\pi(w) = ((\text{number of } \sigma_1 \in w) - (\text{number of } \sigma_1^{-1}), \cdots,$$
$$(\text{number of } \sigma_k \in w) - (\text{number of } \sigma_k^{-1} \in w)).$$

If a word w over the extended alphabet contains more occurrences of a letter σ_i^{-1} than σ_i, then the ith component of $\pi(w)$ is negative. Note that when learning a language L, neither the language nor the input include words with characters in Σ^{-1}. Although the hypothesis of Algorithm A^+ may contain these words, they are not in the domain, and thus no mistakes will be made on them. The inverse alphabet is used only as a tool in the proofs.

We will show that closed canonical DFAs that accept a commutative language are essentially Cayley graphs [8], [11], [18].

DEFINITION. A directed multigraph (possibly with self-loops) where the edges are labeled with elements from an alphabet Σ is a *Cayley graph* if it has the following properties:

1. For each letter in Σ, every vertex has exactly one incoming and one outgoing edge labeled with that letter. This means that for each vertex, each word over the extended alphabet $\Sigma \cup \Sigma^{-1}$ describes an undirected path starting at that vertex (if the next letter in the word is some $\sigma \in \Sigma$ then follow the edge labeled with σ leaving the current vertex and if the next letter is some $\sigma^{-1} \in \Sigma^{-1}$ go to the tail of the edge labeled with σ entering the current vertex).
2. If a word over the extended alphabet describes a closed path starting at some vertex, then that word describes a closed path starting at every vertex in the graph.

DFAs naturally define a directed graph: The states are the vertices and the transitions correspond to directed edges.

LEMMA 6.3. *The directed graph defined by a closed canonical* DFA *accepting a commutative regular language is a Cayley graph.*

Proof. Let $M = (Q, \Sigma, \delta, q_0, F)$ be such a DFA and $M' = (Q, \Sigma \cup \Sigma^{-1}, \delta', q_0, F)$ be the extension of M. The first property follows from the fact that M is invertible (Lemma 6.1). For the proof of the second property, let state $q \in Q$ and word w over the extended alphabet be such that $\delta'(q, w) = q$. We need to show that for all states $r \in Q$, $\delta'(r, w) = r$ holds. By the definition of closed canonical DFAs there is a string $x \in \Sigma^*$, such that $\delta'(q, x) = r$. By Lemma 6.2, M' is commutative and thus $r = \delta'(q, x) = \delta'(q, wx) = \delta'(q, xw)$. We conclude that w also leads from r to r. □

Cayley graphs over the alphabet Σ correspond to groups over the generator set Σ as follows [18]: The vertices are the elements of the group and after fixing a start state (which becomes the neutral element) the words leading from the start state to a particular vertex are the words over the generators and their inverses that equal the group element corresponding to that vertex. Also, for each group generated by Σ there is a Cayley graph based on the above correspondence [18].

This implies that any closed canonical DFA (with alphabet Σ) accepting a commutative language corresponds to an abelian group generated by Σ. The language accepted by the DFA consists of all words over Σ that are equal to one of the group elements whose corresponding state is a final state. The abelian group defines a lattice and the language is the union of positive restrictions of cosets of that lattice, one coset for each final state. This proves the main theorem of this section.

THEOREM 6.4. *Let M be a* DFA *accepting a commutative language that is closed and has a single final state. Then the image under π of the language accepted by M is the positive restriction of a coset of a lattice.*

COROLLARY 6.5. *The class of commutative regular languages accepted by closed* DFA*s with one final state can be learned with a mistake bound of*

$$1 + 2|\Sigma| + \left\lfloor |\Sigma| \log(n\sqrt{|\Sigma|}) \right\rfloor,$$

where Σ is the alphabet and n is the length of the longest word seen.

Remark. A set of tuples in $\mathbf{Z}^k$ is the coset of a lattice if and only if whenever x, y, and z are in the set, then the tuple $x-y+z$ is also in the set. Unfortunately, it is not true that a subset of $\mathbf{N}^k$ is the nonnegative restriction of a lattice coset if and only if whenever x, y, and z are in the subset, then $x-y+z$ is either in the subset or has a negative component. Consider the set $\{(1,1,1),(3,0,0),(0,3,0)\}$. Every $x-y+z$ combination of these three vectors either has a negative component or is already in the set, but any lattice coset containing these three vectors also contains $(0,0,3)$. This is the simplest counterexample since for $k \leq 2$ the above characterization of nonnegative restrictions of cosets of lattices does hold.[8]

Corollary 6.5 is somewhat surprising in light of the results of Pitt and Warmuth [21]. They identify a small subclass $\mathcal{C}_k$ of the closed commutative regular languages over k letters, called counter languages, and show that for any $k \geq 2$ and any polynomial Q, the problem: "given a set of examples (from some $L \in \mathcal{C}_k$ accepted by a DFA of s states), find a DFA or NFA with fewer than $Q(s)$ states consistent with the examples" is NP-hard [21].[9] Algorithm A^+ bypasses that hardness result by representing its hypothesis as a coset of a submodule rather than as a DFA. The resulting algorithm for learning $\mathcal{C}_k$ makes at most $1+2k+\lfloor k\log(n\sqrt{k})\rfloor$ mistakes, where n is the length of the longest word seen.

We now define a number of subclasses of regular languages and give lower bounds on their VC dimensions (and hence the number of mistakes made on them by any learning algorithm). By this method we will show that the mistake bound of Corollary 6.5 is within a $\log\log n$ factor of optimal.

DEFINITION. Let $\mathrm{CCS}_{k,n}$ be the class of commutative regular languages over alphabets of size k accepted by closed DFAs having a single final state and restricted words of length at most n ($\mathrm{CCS}_{k,n}$ is the class of Corollary 6.5). Let $\mathrm{REG}_{k,n}$ and $\mathrm{CREG}_{k,n}$ be the class of all regular languages and all commutative regular languages, respectively, over alphabets of size k restricted to words of length at most n.

LEMMA 6.6.

1. $\mathrm{VCdim}(\mathrm{CCS}_{k,n}) \leq 1+k+\lfloor k\log(n\sqrt{k})\rfloor$ *and for every* $\epsilon > 0$ *and for all sufficiently large* n, $\mathrm{VCdim}(\mathrm{CCS}_{k,n}) \geq k(1-\epsilon)\ln n/\ln\ln n$.
2. $\mathrm{VCdim}(\mathrm{REG}_{k,n}) = \frac{k^{n+1}-1}{k-1}$ *if* $k>1$ *and* $n+1$ *if* $k=1$.
3. $\mathrm{VCdim}(\mathrm{CREG}_{k,n}) = \binom{n+k}{k}$.

Proof. The upper bound of part 1 of Lemma 6.6 follows from the mistake bound of the algorithm for learning $\mathrm{CCS}_{k,n}$ given in Corollary 6.5. For the lower bound first observe that the commutative languages over the single letter σ accepted by closed DFAs with a single final state are all languages of the form $\sigma^i(\sigma^j)^*$. Thus the letter counts of these languages are the positive restrictions of shifted one-dimensional lattices. By Corollary 3.2, one-dimensional lattices (ignoring possible shifts) restricted to $\{-n,\cdots,0,\cdots,n\}$ have VC dimension larger than $(1-\epsilon)\ln n/\ln\ln n$, for every $\epsilon>0$ and for all sufficiently large n. The shattered set used to prove the lower bound for one-dimensional lattices (proof of Theorem 3.1) consisted only of positive numbers. Thus the same lower bound applies for $\mathrm{CCS}_{1,n}$.

Positive restrictions of one-dimensional lattices correspond to languages of the

[8] This example corresponds to the language L containing aaa, bbb, and all permutations of abc. Language L is commutative and zero-reversible [3] and thus Algorithm A^+ is unable to learn all of the commutative zero-reversible languages over three or more letters.

[9] If k is an input to the problem then it is even NP-hard to produce a consistent NFA of superpolynomial size: for any $0<\epsilon<1$ it is NP-hard to find a consistent NFA of size $s^{(1-\epsilon)\log\log s}$ [22].

form $(\sigma^j)^*$, i.e., the canonical DFAs accepting these languages have a single final state that equals the start state. Let C denote the subclass of these languages over one letter. To complete the proof of part 1 it suffices to show that $\text{VCdim}(\text{CCS}_{k,n}) \geq k\text{VCdim}(C)$. The proof of this is similar to the proof of Theorem 3.3.

Let S_1 be a set of words over the same letter shattered by C. Let $S = \bigcup_{i=1}^{k} S_{\sigma_i}$, where S_{σ_i} is a copy of S_1 using σ_i as the single letter. Clearly, $|S| = k\,|S_1|$. To show that S is shattered by $\text{CCS}_{k,n}$, let T be an arbitrary subset of S and $T_i = T \cap {\sigma_i}^*$. Now for each T_i there is a DFA M_i over the letter σ_i accepting T_i. Using a standard cross product construction it is easy to build from $M_1, \cdots, M_k$ a commutative DFA over the alphabet $\sigma_1, \cdots, \sigma_k$ accepting T. Since for each M_i the start state equals the single final state, the same holds for the new DFA. Thus this DFA witnesses the fact that T is in $\text{CCS}_{k,n}$.

For the proof of part 2 of Lemma 6.6, observe that the class $\text{REG}_{k,n}$ shatters its entire domain of all words of length at most n, since all subsets of the domain are in the class. The size of the domain is $\sum_{i=0}^{n} k^i$.

In the commutative case (part 3), the class $\text{CREG}_{k,n}$ also shatters its entire domain, which can be characterized by the set of all k-tuples of nonnegative integers whose components sum to at most n. The size of this domain is $\binom{n+k}{k}$. □

6.6. Discussion. Part 1 of Lemma 6.6 shows that the mistake bound of Algorithm $A^+$when used to learn $\text{CCS}_{k,n}$ is within a $\log \log n$ factor of optimal and parts 2 and 3 indicate that it is much harder to learn arbitrary regular languages or even arbitrary commutative regular languages.

7. Nested differences.

Using the results obtained in a companion paper [14], we can apply Algorithm A in the construction of a number of master algorithms that learn nested differences of lattices. Let $\text{DIFF}(\mathcal{L}^k)$ be the class of concepts of the form $\Lambda_1 - (\Lambda_2 - (\Lambda_3 - \cdots - (\Lambda_{p-1} - \Lambda_p))\cdots)$, where each $\Lambda_i \in \mathcal{L}^k$. Thus each concept in $\text{DIFF}(\mathcal{L}^k)$ is a nested difference of lattices. We call p the *depth* of the concept. The master algorithms learn the class $\text{DIFF}(\mathcal{L}^k)$ with a mistake bound that is p times the bound for single lattices. The master algorithms can be used to learn nested differences of any intersection-closed class.[10]

In this section we sketch only a single master algorithm for learning $\text{DIFF}(C)$, where C is any intersection-closed class, and discuss how it can be adapted to learn nested differences of those concept classes where Algorithm A (or its coset modification) was applied. Thus this modified master algorithm can be used to learn nested differences of cosets of lattices, nonhomogeneous vector spaces, commutative regular languages accepted by DFAs whose single final state reaches the start state, etc. The mistake bound, the efficiency, and (generally) the VC dimension grow linearly with the depth of the nested difference. Thus the master algorithm efficiently learns these classes with good mistake bounds.

The master algorithm[11] for $\text{DIFF}(C)$ keeps track of a finite sequence of closures. The depth of the master algorithms hypothesis is the number of closures in the sequence. In [14], each closure is represented by a minimal set of instances that defines the closure. When given a new instance x, the master predicts on x by the following rule. Let the lth closure be the first one that does not contain x. (If x is in all of

[10] In [14] we also give master algorithms for the case when each concept Λ_i is in the union of several concept classes, each of which is intersection-closed, and for the case where the innermost concept Λ_p is from a concept class that is not necessarily intersection-closed.

[11] This is the "space efficient master algorithm" of [14] for learning $\text{DIFF}(C)$.

the closures, then let l be 1 + the depth of the hypothesis.) If l is even, then predict "+" and if l is odd, predict "−." When a mistake is made, x is added to the lth closure. If l is one larger than the depth of the current hypothesis, then the depth of the hypothesis is increased by initializing the lth closure to the closure of $\{x\}$.

The above algorithm applies to learning nested differences of lattices, since they are intersection-closed. A different copy of Algorithm A can be used to efficiently compute each closure of the sequence. It is shown in [14] that the mistake bound of the master algorithm for learning concepts in DIFF(C) of depth p is at most p times the mistake bound of the closure algorithm applied to C.

In §6.3 we gave a simple trick for learning cosets of lattices. A slight modification of the master algorithm lets this trick be used at each position in the sequence. Let x_i be the first mistake made at each position i in the sequence of closures. Example x_i is not directly used to form the ith closure, but rather is remembered as the *shift* at position i. When predicting on a new instance x, l is now the first closure that does not contain $x - x_l$, the appropriately shifted example. (If $x - x_i$ is in the ith closure for all i between 1 and the depth of the hypothesis, then set l to 1 + the depth of the hypothesis.) When a mistake is made, the lth closure is adjusted to include $x - x_l$. If l is one larger than the depth of the hypothesis, then x_l is set to x, i.e., x becomes the shift of the new lth level and the depth of the hypothesis is increased.

This modified master algorithm learns nested differences of cosets of lattices, abelian groups, and commutative regular languages accepted by DFAs whose single final state reaches the start state. Its mistake bound is at most p times the mistake bound for cosets of lattices (which is $1 + k$ larger than the bound for lattices), where p is the depth of the target.

8. Conclusions.

This paper contains a nontrivial algorithm that efficiently learns the basic combinatorial class of integer lattices. The algorithm leads to efficient learning algorithms for a large number of other classes. The mistake bounds of this algorithm, and most of its applications presented, are provably within roughly a $\log\log n$ factor of general lower bounds derived from the VC dimension.

9. Appendix: An implementation of the closure algorithm.

This appendix gives a precise description of Algorithm A for on-line learning of the concept class $\mathcal{L}^k$ of integer lattices, and analyzes its running time. This algorithm is an implementation of the closure algorithm and was derived from the algorithm of Kannan and Bachem [15] for putting a matrix in Hermite normal form (HNF). Basically, we keep track of a basis for the smallest lattice containing all of the positive instances seen so far. Whenever a mistake is made, a new basis for the smallest lattice containing both the old lattice and the example on which the mistake was made must be found. Since Algorithm A is an implementation of the closure algorithm, it never makes mistakes on negative examples.

We first present the on-line Algorithm A and show that it computes the appropriate basis. In order to bound the size of the entries stored by Algorithm A, we present a batch algorithm A', which gets all of the examples at once. By an analysis similar to that used for HNF algorithms [25], [15], we bound the size of entries stored by A'. Finally, we argue that the values stored by Algorithm A after it has made its ith mistake are a subset of the values stored by Algorithm A'. Therefore the bound on the entries stored by Algorithm A' carries over to Algorithm A.

Algorithm A keeps a k by k lower triangular matrix M whose column span represents the current hypothesis. Matrix M is initially all 0, and gradually has nonzero columns added to it as positive examples are seen. Algorithm A will occasionally exchange rows in M. This operation corresponds to changing the order of the components in examples. At any point, the permutation π reflects the row exchanges made by A, and the necessary adjustment to the components of new examples. The algorithm's current hypothesis is all (permuted) examples in CLOSURE(M), the (integer) column space of M. Algorithm A makes a mistake only when it gets a new positive example x where $\pi(x)$ is not in the column space of M. When this happens, Algorithm A updates M (and possibly π) so that the column space contains $\pi(x)$ in addition to the (possibly permuted) column space of the original matrix.

ALGORITHM *A*. Matrix M is initially the $k \times k$ matrix of 0's. Permutation π is initialized to the identity permutation on k elements. The variable z is initialized to 1 and contains the index of the leftmost all-zero column in M. Throughout, m_{ij} denotes the entry in the ith row and jth column of M. The following procedure is executed for each instance x.

1. Permutation:
   $x := \pi(x)$. The component ordering of x now agrees with the row ordering in M.
2. Prediction:
   Determine whether x can be written as an integer combination of the columns in the matrix. (Since M is lower triangular, this can be done by back-substitution with $O(k^2)$ arithmetic operations.) If so, predict "Positive," since it is certain that x is in the target lattice. Otherwise, predict "Negative." (If M is the zero matrix, then a positive prediction is made only on the zero vector.)
3. Update:
   If a mistake is made, then x replaces the all-zero column z in M.[12]
   To return M to normal form, perform the following operation, rather similar to Gaussian elimination:
   (a) For $i := 1$ to $z-1$ do: if m_{iz} is not already 0, force it to 0 by the following:
      i. Use the extended GCD algorithm to find a, b, and g such that $am_{ii} + bm_{iz} = g = \gcd(m_{ii}, m_{iz})$.
      ii. Simultaneously update columns i and z of M. Replace column i with a times column i plus b times column z (and thus m_{ii} becomes g) and replace column z of M with m_{ii}/g times column z minus m_{iz}/g times the old value of column i. This "zeros out" the entry m_{iz}.
   (b) If column z now contains a nonzero entry then x is not linearly dependent on the previous examples. Let j be the first row containing a nonzero entry in column z. If this nonzero entry is negative, multiply column z by -1. Swap rows j and z in M, and swap the zth and jth elements in π. Finally, set z to $z+1$, as the number of nonzero columns in M has increased.
   (c) Call REDUCE($z-1$, M) to ensure that each element to the left of a nonzero diagonal element is less than that diagonal element.
4. Get a new example and go to step 1.

Procedure REDUCE(i, M) [25]. This procedure performs elementary column operations to ensure that each below-diagonal element in both the first i columns and the first i rows is nonnegative and smaller than the (positive) diagonal element on the same row. The off-diagonal elements are examined from column $i-1$ down to column 1, and from row $c+1$ to row i within each column c.

```
for $c := i - 1$ down to 1 do
    for $r := c + 1$ to $i$ do
        if $(m_{rc} < 0)$ or $(m_{rc} \geq m_{rr})$ then
            subtract $\lfloor m_{rc}/m_{rr} \rfloor$ times column $r$ from column $c$
            making $0 \leq m_{rc} < m_{rr}$
        end if;
    end for $r$;
end for $c$;
end REDUCE
```

[12] If M already contains k nonzero columns, then a temporary column $z = k+1$ is created to hold x. The temporary column is deleted after it is "zeroed out" by the following operations.

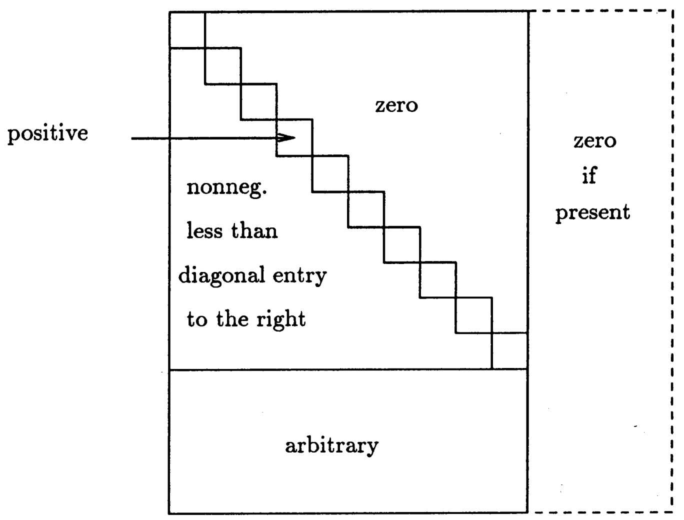

FIG. 3. *Pseudo-Hermite normal form.*

Throughout, M is kept in a pseudo-Hermite normal form (pseudo-HNF).[13]

DEFINITION. A matrix M_1 is a *pseudo-Hermite normal form* (see Fig. 3) of matrix M_2 if both:

- $M_1 = PM_2U$ where P is a permutation matrix and U is unimodular. Thus M_1 can be derived from M_2 by a series of row permutations and elementary column operations [15].
- M_1 is a lower-triangular matrix where in every column with any nonzero entry, the diagonal element is positive and each element to its left is nonnegative and less than that diagonal element.

Although these properties are not needed until we prove that the entries of M remain small, they are the motivation behind several of the operations in Algorithm A.

The strange order used in the REDUCE procedure (see Fig. 4) ensures that only previously reduced elements are added to (or subtracted from) the elements which have not yet been reduced (see [25]).

LEMMA 9.1. *Algorithm A correctly implements the closure algorithm.*

Proof. The row exchange in step 3(b) is reflected by an update to permutation π, which is applied to all future examples. Thus it suffices to show that whenever a prediction mistake is made, matrix M is updated so that its column span (ignoring the row permutation) includes x and no points not in the column span of $M \cup \{x\}$. This occurs when x is inserted into M at the beginning of the update step. It remains to show that the other operations on M do not change its (permuted) column span.

The operations in procedure REDUCE consist of adding a multiple of one column to another, and thus do not change the column span of M. Similarly, the multiplication of a column by -1 in step 3(b) does not change the column span. Finally, we

[13] The notion of pseudo-Hermite normal form is a generalization of Hermite normal form defined for nonsingular (integer) matrices.

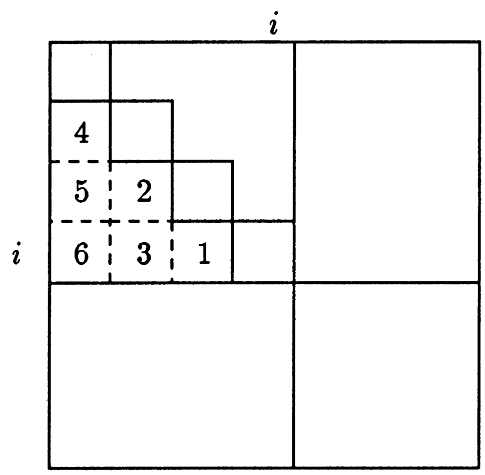

FIG. 4. *Illustration of the REDUCE procedure.*

show that the simultaneous update in step 3(a)ii does not change the column span of M.

It is easy to see that the new columns are in the lattice generated by the old columns. Furthermore, the old column z is a times the new column z plus m_{rz}/g times the new column r, and the old column r is $-b$ times the new column z plus m_{rr}/g times the new column r. Therefore, the lattice generated by M is not changed by the simultaneous updates. □

Although we now know that Algorithm A is correct, we have yet to show that it is efficient. The time taken by the GCD computations performed by Algorithm A, as well as the amount of space used by Algorithm A, depends on the size of the numbers stored in the $k \times k$ matrix M. To bound these entries we study a slightly different algorithm, A'. Algorithm A' uses unimodular column operations and row swaps to convert a $t \times t$ nonsingular matrix of padded examples, M', into pseudo-Hermite normal form. Using the techniques in [25], [15], we bound the entries of this matrix after each column is processed. We will also show that every nonzero entry stored in matrix M by Algorithm A is stored in M' by Algorithm A'.

To create the matrix M', fix the sequence of examples on which the closure algorithm makes t prediction mistakes and let e_i be the example on which the closure algorithm makes its ith mistake. (Thus e_1 will be the first nonzero example.) We assume that $t \geq k$ and that the e_i's have full row rank,[14] and pad the examples out to length t as follows. First, for $1 \leq i \leq t$, define $r(i)$ to be the row rank of the $k \times i$ matrix with columns $e_1, \cdots, e_i$. Note that $i - r(i)$ is the number of linearly dependent columns in this matrix and is a nondecreasing function of i. We now extend each column e_i to an e'_i of length t as follows. If $i = 1$ or $r(i) > r(i-1)$, then vector e'_i is e_i followed by $t - k$ "0"s. If $r(i) = r(i-1)$, then e'_i is e_i followed by $i - 1 - r(i)$ "0"s, a single "1," and another $t - k - i + r(i)$ "0"s. The ith column in matrix M' is e'_i, for $1 \leq i \leq t$. Since the e'_i's are linearly independent and of length t, matrix M', consisting of $e'_1, \cdots, e'_t$, is square and nonsingular. The last $t - k$ rows of M' are the *padding rows*, and any column with a nonzero entry in a padding row is a *padded column* (see Fig. 5).

[14] Additional examples where the closure algorithm would make mistakes can be appended to $\{e_1, \cdots, e_t\}$, making the assumptions true.

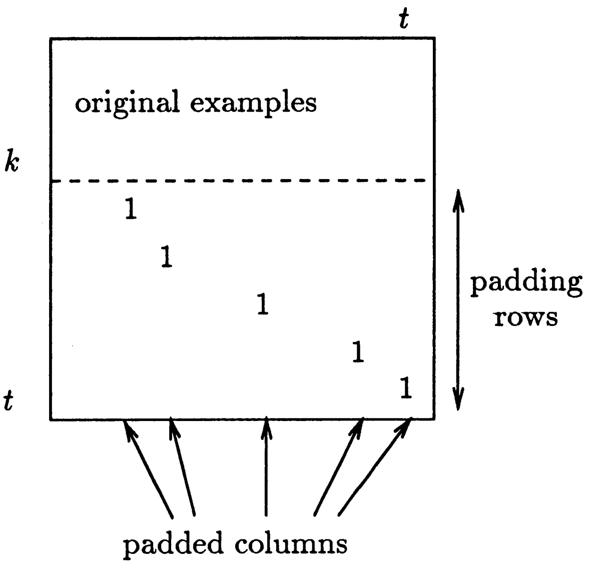

FIG. 5. *Initial structure of M'.*

Like the Hermite normal form (HNF) algorithm of Kannan and Bachem [15], our algorithm iteratively processes the columns from left to right, placing the principal minors of M' into (pseudo-) Hermite normal form.

That algorithm first preconditions the matrix using column permutations so that all of the principle minors in the resulting matrix are nonsingular. In our application we are given one example (column) at a time, so we replace the preconditioning step by on-the-fly row swaps. These row swaps will ensure that after processing column i, the $i \times i$ principal minor is nonsingular. Furthermore, we segregate the padding rows and padded columns from the normal rows and columns. Whenever a $c \times c$ principal minor has been placed into pseudo-HNF, there will be some number, say c', of processed nonpadded columns. These nonpadded columns will be in columns one through c', and the processed padded columns will be in columns $c'+1$ through c. Similarly, rows one through c' will be nonpadding rows and rows $c'+1$ through c will be padding rows (see Fig. 6).

To maintain this organization, at each iteration the new column is moved left to between columns c' and $c'+1$. Similarly, some row below row $c-1$ will be moved up between rows c' and $c'+1$.

ALGORITHM A'. *This algorithm modifies matrix M', placing it into pseudo-*HNF.

- Initialize c' to 0. Variable c' counts the number of nonpadded columns which have been processed.
- For $c := 1$ to t put the $c \times c$ minor into pseudo-HNF as follows:
  1. For $i := 1$ to c' do: if m'_{ic} is not already 0, force it to 0 by the following:
     (a) Use the extended GCD algorithm to find a, b, and g such that $am'_{ii} + bm'_{ic} = g = \gcd(m'_{ii}, m'_{ic})$.
     (b) *Make m'_{ic} zero:* Simultaneously update columns i and c of M'. Replace column i with a times column i plus b times column c (and thus m'_{ii} becomes g) and replace column c of M to m'_{ii}/g times column c minus m'_{ic}/g times the old value of column i. This "zeros out" the entry m'_{ic}.
  2. If $m'_{cc} = 0$ then permute rows to make it nonzero. Let m'_{rc} be the topmost nonzero entry in column c below row c. Move row r to row c, shifting each row from row c through row $r-1$ down one position. The column c is a padded column exactly

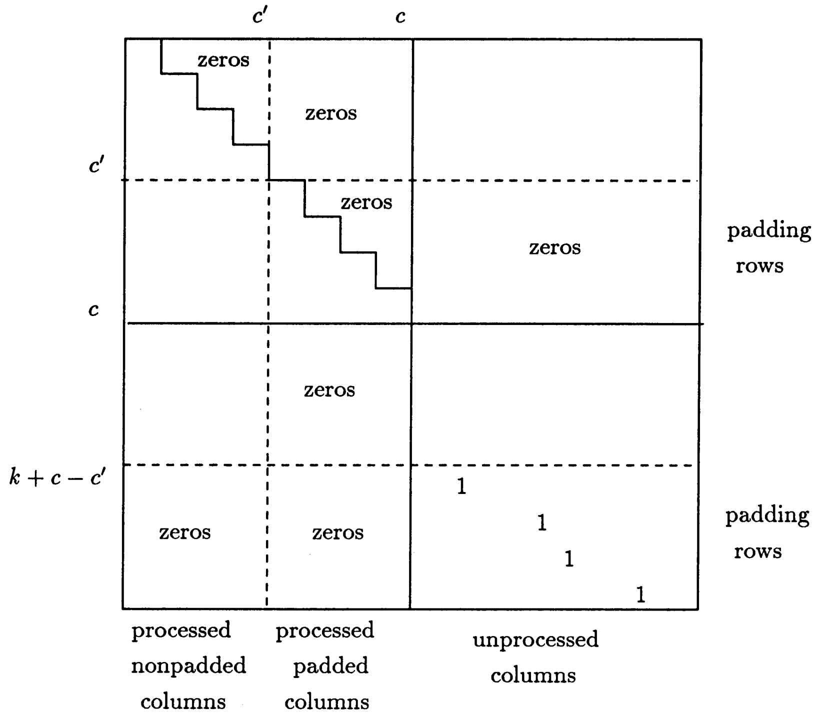

FIG. 6. *Structure of M' after c iterations.*

```
      when $r \geq k + c - c'$.
3. If $m'_{cc} < 0$ then multiply column $c$ by $-1$.
4. Permute the rows and columns to segregate the padding rows and padded column.
   (See Fig. 7.)
   (a) Move column $c$ to column $c'+1$, shifting columns $c'+1$ through $c-1$ one column
       to the right.
   (b) Move row $c$ to row $c'+1$ shifting rows $c'+1$ through $c-1$ one row down.
5. If column $c$ was a nonpadded column then $c' := c' + 1$.
6. Call REDUCE($c$, $M'$) to ensure that the $c \times c$ principal minor of $M'$ is in pseudo-HNF.
```

Note that Algorithm A' uses only columns 1 through c while processing the first c columns and placing the $c \times c$ principal minor into pseudo-HNF. Although the simultaneous transformations made in step 1(b) can affect the values of padding rows, we adopt the convention that this step never changes the padded/nonpadded status of a column. Also, the padded/nonpadded status of columns and the padding/nonpadding status of rows is carried along with them when rows and columns are permuted.

The simultaneous transformations of Step 1(b) are unimodular since the matrix

$$\begin{bmatrix} a & -m'_{rc}/g \\ b & m'_{rr}/g \end{bmatrix}$$

has determinant $am'_{rr}/g + bm'_{rc}/g = 1$. Note also that this transformation is identical to the one used in Algorithm A.

In some sense, the padded columns are used in only one iteration of the "for c"

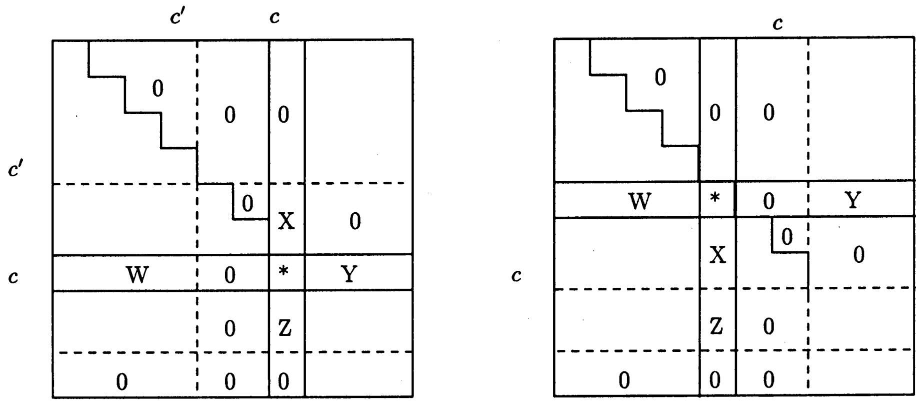

FIG. 7. *Illustration of step* 4.

loop of Algorithm A'. A padded column is created from an example that is a linear combination of the previous examples. Therefore, immediately after a padded column has been processed its only nonzero entries will be in padding rows.

It is not obvious that a processed padded column will always have zero entries in nonpadding rows. The padded column might be spoiled during step 1 while processing a future column, or during the REDUCE procedure if a nonpadded column is used to reduce one of its entries.

LEMMA 9.2. *A processed padded column always has a* 0 *in every nonpadding row.*

Proof. The proof is by induction on the number of processed padded columns. Let column c be the first padded column in the initial matrix M'. The lemma holds trivially until iteration c, when this padded column is processed. Column c was created from example e_c, which can be expressed as a linear combination of the previous examples. Columns 1 through $c-1$ represent a basis for the vector space generated by the previous examples, so all entries in column c on normal rows will be "zeroed out" during step 1. Thereafter, the row and column permutations guarantee that column c remains the last column in the pseudo-HNF principal minor. The only further references to column c are shifting it right a column, permuting its entries in steps 2 and 4(b), and in the procedure REDUCE where it is subtracted from other columns. None of these operations can create a nonzero entry in a nonpadding row.

Let there be $c-c' > 0$ processed padded columns before iteration c where column c is a padded column. The nonpadding rows of the first c' columns represent a basis for the vector space generated by the first c examples. During step 1, all entries in column c representing components of e_c will be zeroed out. The only other (nonpermutation) modifications to the column occur during the REDUCE procedure, where multiples of other padded columns may be subtracted from it. However, by the inductive hypothesis, the nonzero entries of these other columns occur only in padded rows. Therefore, the column's entries on nonpadding rows remain 0. □

COROLLARY 9.3. *Entries in a nonpadding row are not affected by adding multiples of padded columns during the* REDUCE *procedure.*

LEMMA 9.4. *At the end of iteration c, the first c columns of M' are converted to* *pseudo*-HNF *form.*

Proof. Since the algorithm performs only row permutations and unimodular column operations, the first condition for a pseudo-HNF form is met. The algorithm ensures that the diagonal elements are positive (step 3), and the REDUCE procedure ensures that the entries to the left of the diagonal have the proper values. The remainder follows by induction on c.

If the newly processed column is a nonpadded column then, after step one, the first $c-1$ entries of c are 0. Each padding row is either in its original position or is above row c, so the first nonzero entry in column c is in a nonpadding row. Thus after step 2, row c will be a nonpadding row. By Lemma 9.2 that row contains a 0 in each processed padded column. Therefore, if the first $c-1$ columns were lower-triangular before step 4, then the first c columns are lower-triangular after that step.

If the newly processed column is a padded column, then at step 2 the only nonzero element in column c below row c is some multiple[15] of the original "1" in the padded example for that column. Before iteration c, this column was the only column containing a nonzero entry on that padding row, which becomes row c after step 2. Thus, the previously processed padded columns still have zero entries in the (new) row c, and if the first $c-1$ columns were lower-triangular before step 4, then the first c columns are lower-triangular after that step. □

LEMMA 9.5. *Let M^i be the modified matrix M' after i iterations of the main loop of Algorithm A'. The entries of M^i are no larger than $tn^{k+1}k^{k/2}$ for $1 \le i \le t$.*

Proof. Algorithm A' ensures that, at the end of iteration i, the $i \times i$ principal minor of M^i is a nonsingular, lower-triangular matrix with each entry between 0 and the value of the diagonal element to its right. Furthermore, each M^i is formed from M' by a series of row permutations and unimodular column operations, thus $M^i = P^i M' U^i$. Since the column operations involve only the first i columns,

$$U^i = \begin{bmatrix} U_i^i & 0 \\ 0 & I \end{bmatrix},$$

where U_i^i is an $i \times i$ unimodular matrix. Let M_i^i and M_i' denote the $i \times i$ principal minors of M^i and $P^i M'$, respectively. Thus, $M_i^i = M_i' U_i$ and[16]

$$U_i = (M_i')^{-1} M_i^i = \frac{\text{adj } M_i'}{\det M_i'} M_i^i.$$

Let adjmax(M_i') be the largest absolute value of any entry in adj M_i'. Now when u_{jk} and m_{jk}^i are the entries in row j and column k of U_i and M_i^i, respectively,

$$|u_{jk}| \le \sum_{l=1}^{i} |m_{lk}^i| \frac{\text{adjmax}(M_i')}{|\det M_i'|}.$$

The sum of the entries in any column contains at most one diagonal element. Since M_i^i is in Hermite normal form, every entry is nonnegative and each nondiagonal entry is less than the diagonal entry to its right. Therefore,

$$\sum_{l=1}^{i} |m_{lk}^i| \le (m_{11}^i - 1) + (m_{22}^i - 1) + \cdots + (m_{ii}^i - 1) + 1$$

[15] In step 1 column c is repeatedly replaced by m_{ii}'/g times column c plus a multiple of some already processed column.

[16] adj M represents the adjoint of M.

$$\leq \prod_{l=1}^{i} m_{ll}^{i} = \det M^{i} = \det(M_i'U_i) = |\det M_i'|,$$

and so for each u_{jk},

$$|u_{jk}| \leq \text{adjmax}(M_i').$$

Every element of adj M_i' is the determinant of an $(i-1) \times (i-1)$ minor of M_i'. Recall that the last $t-k$ rows of M' each contain a single 1 and $t-1$ 0's. Therefore, by cofactor expansion, the determinant of any large submatrix of M' is at most the determinant of a $k \times k$ matrix whose columns are in $\{e_1, \cdots, e_t\}$. Note that each entry in this smaller matrix is at most n, so its determinant contains $k!$ terms each at most n^k. Applying Hadamard's inequality (see, for example, [12]) shows that the determinant of any $k \times k$ matrix with entries bounded by n, and thus each u_{jk}, has absolute value at most $k^{k/2}n^k$.

As $M^i = P^i M' U^i$, each entry of M' has absolute value at most n, and each entry of U has absolute value at most $k^{k/2}n^k$; the absolute value of each entry of M^i is at most $tnk^{k/2}n^k$. □

In particular, $t \leq k + k\log(n\sqrt{k})$, the mistake bound of the closure algorithm. This gives us the following corollary.

COROLLARY 9.6. *After each iteration of the "for c" loop of Algorithm A', the largest entry in M^c is at most $k^{k/2}n^{k+1}(k + k\log(n\sqrt{k}))$, which can be written in $O(k\log(nk))$ bits.*

Using the above, we bound the size of numbers *during* iterations of Algorithm A'.

LEMMA 9.7. *During the computation of Algorithm A', each entry of M' requires at most $O\left(k\log^2(nk)\right)$ bits.*

Proof. The sizes of entries are changed only in the REDUCE procedure and step 1. Let $m = k^{k/2}n^{k+1}(k + k\log(n\sqrt{k}))$ be a bound on the largest absolute value of an entry in M' at the beginning of step 1. Each iteration of this step increases the entries in column c by a factor of at most $2m$, as both a and b found in Step 1(a) are at most m [15]. Since there are at most k iterations, the largest entry in column c at the end of step 1 is at most $m(2m)^k$. The entries in the other columns are bounded by the same expression.

The REDUCE procedure can also produce large intermediate results. Consider what happens as we reduce some column. The entries in the previously reduced columns are bounded by m. Each time an entry in the column is reduced, the unreduced entries in that column are increased by at most m times the value of the entry being reduced (reduced entries are never greater than m). Since there are t entries in each column that are originally bounded by $m(2m)^k$, the entries of the column never get larger than $m(2m)^k(1+m)^t$.

The maximum number of bits needed to represent an entry is roughly $t\log(m+1)+k+(k+1)\log m$. Plugging in the bounds on m and t gives us that $O\left(k^2\log^2(nk)\right)$ bits suffice to represent each entry (note that the hidden constants are small). □

Note that one of the key contributions of [25] is the clever order used by the REDUCE procedure. This lets them show that the maximum entry size *during* an iteration of their HNF algorithm is within a constant factor of the between-iteration bound. It appears likely that their techniques could achieve better bounds on the maximum entry size for Algorithm A' than the simple ideas used in the proof of Lemma 9.7.

We now formally state the relationship between Algorithm A and Algorithm A'.

LEMMA 9.8. *After Algorithm A has processed the cth mistake and Algorithm A' has processed the first c columns of M', each nonzero entry in M also appears in M'.*

Proof. It is easy to show by induction on c that the first c columns of M' are identical to the nonzero columns in M after the padded columns and padding rows are deleted. □

This allows us to apply Corollary 9.6 and Lemma 9.7 to Algorithm A.

THEOREM 9.9. *Using the uniform*[17] *cost measure, Algorithm A requires time $O\left(k^2\right)$ to make a prediction, and time $O\left(k^3 + k\log(nk)\right)$ to perform an update, where n is the largest absolute value of any component of any instance seen.*

Proof. Prediction time: The prediction time is just the time for back substitution, $O\left(k^2\right)$.

Update time: In step 3(a), updating matrix M can in general require k extended GCD operations to be performed, and the running time for extended GCD is proportional to the logarithm of the smaller of the two numbers. In our case, one of the two numbers will always be a diagonal element stored between iterations. By Lemma 9.5, the saved diagonal elements are at most $(k + k\log(n\sqrt{k}))k^{k/2}n^{k+1}$ and each extended GCD computation can be done in $O\left(k\log(kn)\right)$ time.

Finally, each iteration of the nested for loop of procedure REDUCE requires $O(k)$ subtractions, so this step takes time $O(k^3)$ altogether. □

Acknowledgment. We would like to thank David Haussler for insightful discussions.

REFERENCES

[1] N. ABE, *Polynomial learnability of semilinear sets*, in Second Workshop on Computational Learning Theory, Santa Cruz, CA, July 1989, Morgan Kaufmann, Los Altos, CA, pp. 24–40.

[2] ———, *Learning commutative deterministic finite state automata in polynomial time*, New Generation Computing, 8 (1991), pp. 319–336.

[3] D. ANGLUIN, *Inference of reversible languages*, J. Assoc. Comput. Mach., 29 (1982), pp. 741–765.

[4] ———, *Queries and concept learning*, Machine Learning, 2 (1987), pp. 319–342.

[5] J. M. BARZDIN AND R. V. FREIVALD, *On the prediction of general recursive functions*, Soviet Math. Dokl., 13 (1972), pp. 1224–1228.

[6] S. BOUCHERON, *Learnability from positive examples in the Valiant framework*, unpublished manuscript, 1988.

[7] J. W. S. CASSELS, *An Introduction to the Geometry of Numbers*, Springer-Verlag, Berlin, New York, 1959.

[8] H. S. M. COXETER AND W. O. J. MOSER, *Generators and Relations for Discrete Groups*, Third Edition, Springer-Verlag, New York, 1972.

[9] R. M. DUDLEY, *A course on empirical processes*, in Lecture Notes in Mathematics No. 1097, Springer-Verlag, Berlin, New York, 1984.

[10] A. FIAT, S. MOSES, A. SHAMIR, I. SHIMSHONI, AND G. TARDOS, *Planning and learning in permutation groups*, in 30th Annual IEEE Symposium on Foundations of Computer Science, 1989, pp. 274–279.

[11] I. GROSSMAN AND W. MAGNUS, *Groups and Their Graphs*, Vol. 14, New Mathematical Library, Mathematical Association of America, Washington, 1964.

[12] G. H. HARDY, J. E. LITTLEWOOD, AND G. PÓLYA, *Inequalities*, Second Edition, Cambridge University Press, Cambridge, U.K., 1952.

[13] G. H. HARDY AND E. M. WRIGHT, *An Introduction to the Theory of Numbers*, Fourth Edition, Oxford University Press, Oxford, U.K., 1960.

[14] D. HELMBOLD, R. SLOAN, AND M. K. WARMUTH, *Learning nested differences of intersection-closed concept classes*, Machine Learning, 5 (1990), pp. 165–196.

[17] This is reasonable since we have just shown that the numbers involved remain reasonably small.

[15] R. Kannan and A. Bachem, *Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix*, SIAM J. Comput., 8 (1979), pp. 499–507.

[16] N. Littlestone, *Learning when irrelevant attributes abound: A new linear-threshold algorithm*, Machine Learning, 2 (1988), pp. 285–318.

[17] N. Littlestone, *From on-line to batch learning*, in Second Workshop on Computational Learning Theory, Santa Cruz, CA, July 1989, Morgan Kaufmann, Los Altos, CA, pp. 269–284.

[18] W. Magnus, A. Karrass, and D. Solitar, *Combinatorial Group Theory: Presentation of Groups in Terms of Generators and Relations*, John Wiley & Sons, New York, 1966.

[19] T. Mitchell, *Generalization as search*, Artificial Intelligence, 18 (1982), pp. 203–226.

[20] B. K. Natarajan, *Machine Learning: A Theoretical Approach*, Morgan Kaufman, San Mateo, CA, 1991.

[21] L. Pitt and M. K. Warmuth, *The minimum* DFA *consistency problem cannot be approximated within any polynomial*, Tech. Report UIUCDCS-R-89-1499, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, Feb. 1989.

[22] ———, *The minimum* DFA *consistency problem cannot be approximated within any polynomial*, J. Assoc. Comput. Mach., to appear.

[23] H. Shvaytser, *Linear manifolds are learnable from positive examples*, unpublished manuscript, 1988.

[24] V. N. Vapnik and A. Y. Chervonenkis, *On the uniform convergence of relative frequencies of events to their probabilities*, Theory Probab. Appl., 16 (1971), pp. 264–280.

[25] T. Wu J. Chou and G. E. Collins, *Algorithms for the solution of systems of linear Diophantine equations*, SIAM J. Comput., 11 (1982), pp. 687–708.