# A Composition Theorem for Learning Algorithms with Applications to Geometric Concept Classes

Shai Ben-David* Nader H. Bshouty† Eyal Kushilevitz‡

**Abstract**

This paper solves the open problem of exact learning geometric objects bounded by hyperplanes (and more generally by any constant degree algebraic surfaces) in the constant dimensional space from equivalence queries only (i.e., in the on-line learning model).

We present a novel approach that allows, under certain conditions, the composition of learning algorithms for simple classes into an algorithm for a more complicated class. Informally speaking, it shows that if a class of concepts $C$ is learnable in time $t$ using a small space then $C^\star$, the class of all functions of the form $f(g_1, \ldots, g_m)$ with $g_1, \ldots, g_m \in C$ and *any* $f$, is learnable in polynomial time in $t$ and $m$. We then show that the class of halfspaces in a fixed dimension space is learnable with a small space.

## 1 Introduction

Littlestone's *on-line learning model* [L88, L89] is one of the major models of learning. Learnability in this model implies learnability in Valiant's PAC model [Val84], and is equivalent to learnability in Angluin's equivalence-queries model [A88]. Many efficient algorithms were developed for learning various concept classes in the on-line setting. For example, [L88, L89] show how to learn classes such as $k$-term-DNF and present the WINNOW algorithm for learning a (single) hyperplane. [HSW90, Sim95] prove the learnability of decision-lists in the on-line model.

One type of concept classes which attracted considerable attention (in the on-line model as well as in other models) is that of geometric concept classes. In this case we consider a discretized domain of $R^d$ (i.e., the set of points of the form $\{0, 1, \ldots, n\}^d$, for some $n$) and the concepts considered are of geometric nature such as axis-parallel boxes (e.g., [MT89, CM92, Aue93]). A very important open problem in this area is whether the class of objects in a fixed dimension (i.e., in $R^d$ for $d = O(1)$) which are bounded by halfspaces can be learned efficiently (i.e., in $poly(\log n)$ time and queries). The widest geometric concept-class of this sort that was known to be efficiently learnable in the on-line setting is that is of geometric objects in a fixed dimension which are bounded by halfspaces whose slopes are known [BCH94].

In this paper we present a powerful tool for the design of efficient on-line learning algorithms. This tool allows taking an efficient learning algorithm $A$, satisfying a certain "simplicity" constraint, that

*Department of Computer Science, Technion, Haifa 32000, Israel. E-mail: shai@cs.technion.ac.il.

†Department of Computer Science, University of Calgary, Calgary, Alberta, Canada. E-mail: bshouty@cpsc.ucalgary.ca. http://www.cpsc.ucalgary.ca/~bshouty/home.html. and Department of Computer Science, Technion, Haifa 32000, Israel. E-mail: bshouty@cs.technion.ac.il.

‡Department of Computer Science, Technion, Haifa 32000, Israel. E-mail: eyalk@cs.technion.ac.il. http://www.cs.technion.ac.il/~eyalk. This research was supported by Technion V.P.R. Fund 120-872 and by Japan Technion Society Research Fund.

learns a "simple" concept class $\mathcal{C}$ and construct a new efficient algorithm $A^\star$ that can learn the concept class $\mathcal{C}^\star$ of all compositions of functions from $\mathcal{C}$ (i.e., $\mathcal{C}^\star$ includes all the functions of the form $f(g_1, \ldots, g_m)$ where every $g_i$ is a function in $\mathcal{C}$ and $f$ is an *arbitrary* boolean function; the "size" of such a function is defined as the sum of "sizes" of the $g_i$'s). The "simplicity" requirements on $A$ and $\mathcal{C}$ are as follows. First, we require that $A$ will be a space-bounded algorithm; informally speaking, at any time during the learning, $A$ is allowed to store in its memory at most $k = O(1)$ examples and $O(\log\log n)$ additional bits. (The notion of space-bounded learning appears, for example, in [AFHM93, Ame94, Ame95, Flo89, FW95].) In addition the set of hypotheses used by $A$ must have a VC-dimension of $O(1)$ (this implies in particular that VC-dim$(\mathcal{C}) = O(1)$).

While the above "simplicity" constraints may seem very restrictive it turns out that several interesting concept classes $\mathcal{C}$ can be learned in this manner. [Ame94] shows such a "simple" algorithm for half-planes (in $R^2$); we generalize his result to any fixed dimension (i.e., we present an efficient algorithm which requires $O(d^2)$ space for learning a half-space in $R^d$; hence, for $d = O(1)$ this gives a "simple" algorithm). More generally, we show how to learn in this manner any constant-degree algebraic surface in a fixed dimension.

Applying our composition theorem to these algorithms we get an efficient algorithms for learning the corresponding concept classes $\mathcal{C}^\star$. In particular, this gives an efficient algorithm for learning any function which can be defined by halfspaces (in a fixed dimension), solving the abovementioned open problem. (E.g., for $d = 2$ this class includes all the functions that can be defined by a collection of $m$ lines in the plane and assigning each "region" created by these lines one of the values 0 or 1; the complexity of the algorithm depends on $m$.) Similarly, we can efficiently learn the class of constant-degree semi-algebraic functions (that is, the functions which are defined using a collection of constant-degree algebraic surfaces) in a fixed dimension. Results of this strength were known before only in the PAC model [BGMST96].

**Organization:** In section 2 we provide some preliminary definition and facts. In Section 3 we present the composition theorem. Finally, in Section 4 we present space-bounded ("simple") algorithms for several concept classes. In Appendix A we prove that every class $\mathcal{C}$ with VC-dim$(\mathcal{C}) = 1$ is space-bounded learnable (not necessarily in an efficient way).

# 2 Preliminaries

In this section we give some definitions and preliminary results.

## 2.1 Dual Class and VC Dimension

Let $X = \{x_1, \ldots, x_m\}$ be a set and $\mathcal{H} \subseteq \{0,1\}^X$ be a class of boolean functions over $X$. Define the *dual* class $\mathcal{H}^\perp \subseteq \{0,1\}^{\mathcal{H}}$ the class of boolean functions $x_i^\perp$ where for every $h \in \mathcal{H}$ we have $x_i^\perp(h) = h(x_i)$.

It is convenient to think of a class $\mathcal{H}$ as a matrix $M$. Each row of $M$ corresponds to a function $h \in \mathcal{H}$ and each of its columns corresponds to a function $x^\perp \in \mathcal{H}^\perp$. The class $\mathcal{H}^\perp$ is the class represented by the transposed matrix $M^T$. The VC-dimension of $M$, denoted VC-dim$(\mathcal{H})$, is the maximal number $d$ of columns in which all the $2^d$ combinations of 0's and 1's appear. The following claim relates the VC-dim of $\mathcal{H}$ and $\mathcal{H}^\perp$.

**Claim 1** *For every class $\mathcal{H}$,*

$$\textit{VC-dim}(\mathcal{H}) \geq \lfloor \log \textit{VC-dim}(\mathcal{H}^\perp) \rfloor .$$

**Proof:** Denote $d = \text{VC-dim}(\mathcal{H}^\perp)$. That is, the matrix $M$ corresponding to $\mathcal{H}$ has a set $B$ of $d$ rows in which all the $2^d$ combinations appear. Assume for convenience that $d = 2^k$ (i.e., $d$ is a power of two). We now show that $M$ contains $\log d = k$ columns in which all $2^{\log d} = d$ combinations appear. This implies the claim.

Enumerate the $d$ rows in $B$ as $0, 1, \ldots, d-1$, and define $k$ vectors $v_i$ $(i = 1, \ldots, k)$ as follows: The vector $v_i$ contains the $i$-th bit in the binary representations of the $d$ numbers. Since the $d$ rows contain all $2^d$ combinations then, in particular, there are $k$ rows which contain (in the entries corresponding to $B$) the $k$ vectors $v_1, \ldots, v_k$. This implies that in this $k$ columns all the $2^k = d$ combinations of 0s and 1s appear, as needed. $\square$

**Remark:** Note that this is the best possible connection between $\text{VC-dim}(\mathcal{H}^\perp)$ and $\text{VC-dim}(\mathcal{H})$. To see this, consider a matrix of dimensions $d \times 2^d$ in which each combination in $\{0,1\}^d$ appears in one of the columns. It follows that $\text{VC-dim}(\mathcal{H}^\perp) = d$ while $\text{VC-dim}(\mathcal{H}) = \lfloor \log d \rfloor$.

## 2.2 The Learning Model

The model we consider in this paper is the on-line learning model or equivalently exact learning with equivalence queries only. Let $X_n$ be a sequence of *domains*. Given a class of boolean function $\mathcal{C} \subseteq \{0,1\}^{X_n}$ and a class of hypotheses $\mathcal{H} \subseteq \{0,1\}^{X_n}$. In the *exact learning model* the learner uses *equivalence queries* to identify the target function $f \in \mathcal{C}$. An equivalence query receives a function $h \in \mathcal{H}$ and returns an answer which is either YES, indicating that $h$ is logically equivalent to $f$, or (NO,$x$), indicating that $h$ is not logically equivalent to $f$ and $f(x) \neq h(x)$. In the latter case we call $x$ a *counterexample* to $h$.

We say that the class $\mathcal{C}$ is *learnable* by $\mathcal{H}$ in polynomial time using equivalence queries if there exists a learner, whose time and query complexity are polynomial in $\log |X_n|$ and the size of $f$, that for every target function $f \in \mathcal{C}$ finds a hypothesis $h$ that is logically equivalent to $f$.

An equivalence queries algorithm has the following structure: it works by a sequence of phases, each phase starts when the algorithm has some information in its memory, it then performs some deterministic computation which results in producing some hypothesis $h \in \mathcal{H}$. Finally, it asks an equivalence query on $h$. If $h$ is not equivalent to the target function then the algorithm receives a counterexample to his hypothesis based on which he can update the information in its memory. An algorithm is called a $(k_1, k_2)$ space bounded algorithm if the information it keeps in its memory from phase to phase is restricted to an array of $k_1$ labeled examples it has seen so far, and an additional array of $k_2$ bits (any other information it computes in order to produce $h$ is erased whenever the equivalence query is asked). While this definition might seem artificial, we emphasize that we use it only as a tool to construct algorithms in the standard (non-restricted) sense.

## 2.3 Halfspaces and Algebraic Surfaces

For an integer $d$ we call the domain $X_n = \{0, 1, \ldots, n\}^d$ the *d-dimensional discretized space*. We consider boolean functions in $\{0,1\}^{X_n}$. In particular, the class of *Halfspaces* over $X_n$ is the set of functions of the form

$$f(x) = \begin{cases} 1 & c_1x_1 + \cdots + c_dx_d \geq b \\ 0 & c_1x_1 + \cdots + c_dx_d < b \end{cases}$$

where $c_1, \ldots, c_d, b$ are integers.

Let $C$ be the class of all halfspaces over $\{0, 1, \ldots, n\}^d$. We define $C^r$ the class of all functions $f(g_1, \ldots, g_r)$ where $f$ is any boolean function and $g_1, \ldots, g_r \in C$. Notice that any geometric object that is bounded by $r$ hyperplanes is in $C^r$. Denote $C^\star = \cup_r C^r$. The size of a geometric object $G$ in $C^*$ will be the minimal $r$ such that $G \in C^r$, i.e., the minimal number of hyperplanes that bound $G$.

An algebraic surface of degree $k$ is defined by a degree-$k$ multivariate polynomial in $x_1, \ldots, x_d$. If $P(x)$ is such a polynomial then the corresponding function $f$ gives the value 1 to every $x$ such that $P(x) \geq 0$ and the value 0 otherwise. (Obviously a halfspace is a special case of such a surface with degree $k = 1$.) Let $\mathcal{C}$ be the class of all degree-$k$ algebraic surfaces over $\{0, 1, \ldots, n\}^d$. The class $\mathcal{C}^\star$ is called degree-$k$ semi-algebraic functions (over $R^d$).

# 3 The Composition Theorem

In this section we present our main tool. This is a reduction that allows using any "simple" learning algorithm for any "simple" concept class to construct a (non–"simple") algorithm that learns any combination of concepts in the above class.

Let $\mathcal{C}$ be a class of boolean functions. Define the class $\mathcal{C}^\star$ to be the set of all boolean functions that can be represented as $f(g_1, \ldots, g_m)$ where $f$ is any boolean function, $m \geq 0$ and $g_i \in \mathcal{C}$ for $i = 1, \ldots, m$. If $s$ is the size measure function used on $\mathcal{C}$ then we define the size $s^\star$ of $f(g_1, \ldots, g_m)$ to be $s(g_1) + \cdots + s(g_m)$. (Note that for the definition of size we do not care what $f$ is and that different function in the class $\mathcal{C}^\star$ might be composed from a different number of function, $m$.)

**Theorem 1** *Let $\mathcal{C}$ and $\mathcal{H}$ be classes of boolean functions over domain $X$. If the concept class $\mathcal{C}$ is $(k_1, k_2)$-space bounded learnable by (the hypotheses class) $\mathcal{H}$ using $q$ equivalence queries then the concept class $\mathcal{C}^\star$ is learnable by (the hypotheses class) $\mathcal{H}^\star$ using*

$$(2qm + 1)^{k_1 v + 1} \cdot 2^{k_2 v}$$

*equivalence queries where $v = \text{VC-dim}(\mathcal{H}^\perp) \leq 2^{\text{VC-dim}(\mathcal{H})}$, and $m$ is the number of functions in $\mathcal{C}$ on which the target function is based.*

**Proof:** Let $ALG$ be a $(k_1, k_2)$ space-bounded algorithm that learns the concept class $\mathcal{C}$ by $\mathcal{H}$ and uses $q$ equivalence queries. By the definition of $ALG$ being space bounded, each of which can be considered as applying a (deterministic) procedure $Alg$ on the content of the memory ($k_1$ labeled examples plus $k_2$ bits) to produce some hypothesis $h \in \mathcal{H}$ (this procedure in particular can modify the content of the memory). It then asks an equivalence query on $h$. If $h$ is not equivalent to the target function then the algorithm $ALG$ receives a counterexample to his hypothesis which is stored in a special position in the memory.

In what follows we present our algorithm that, based on the procedure $Alg$, can learn the concept class $C^\star$. Let $f(g_1, \ldots, g_m)$ be the target function. Before providing the details, we wish to give an informal overview of what the algorithm does. The algorithm maintains a collection $A$ of labeled examples. For each subset of $A$ of size at most $k$ and every choice of $k_2$ bits, our algorithm uses $Alg$ to generate a hypothesis (Step 4). Next, we combine the resulted hypotheses $h_1, \ldots, h_t$ into a single hypothesis $H$ as follows. If a point $x$ has a "similar" point $x' \in A$, meaning that all the hypotheses agree on $x$ and $x'$, then $H(x)$ is defined as the label of $x'$. For every other point we define arbitrarily $H(x) = 0$ (Steps 5 and 6). We then ask an equivalence query on $H$. Either it turns out that $H$ is equivalent to the target function or we get a counterexample $z$. If $z$ is such that we had no "similar" point to $z$ (and hence $H(z)$ was given an arbitrary value) we update the hypothesis such that every point similar to $z$ will get the same value as $z$ (Step 8). (Using the condition on VC-dim$(\mathcal{H}^\perp)$, we will show that the number of times that this can happen is not too large.) If $z$ is a point for which we did have a "similar" point this implies that we can find a counterexample to one of the $h_i$'s. In this case we add the corresponding examples to $A$ (Step 9) and start again.

1. $A \leftarrow \emptyset$.
2. Let $\mathcal{S} = \{S_1, \ldots, S_t\}$ be the collection of all possible memory contents; that is, each $S_i$ consists of an array of $k_1$ examples from $A$ and additional $k_2$ bits.
3. $Z \leftarrow \emptyset$.
4. for $i = 1$ to $t$ do $h_i \leftarrow Alg(S_i)$.
5. Define a table $M(y_1, \ldots, y_t)$, $y_i \in \{0, 1\}$

$$M(y_1, \ldots, y_t) = \begin{cases} \xi & \text{if } (x, \xi) \in A \text{ and } y_i = h_i(x) \text{ for all } i \\ \star & \text{otherwise} \end{cases}.$$

   (* The size of $M$ is $2^t$. However (as we will prove) most of its entries contain the value $\star$ hence we will actually hold in the memory only the non-$\star$ entries which we will construct by going over the elements of $A$. *)

6. Define a hypothesis

$$H(h_1(x), \ldots, h_t(x)) = \begin{cases} M(y_1, \ldots, y_t) & \text{if } M(y_1, \ldots, y_t) \neq \star \\ 0 & \text{otherwise} \end{cases}.$$

7. Ask $EQ(H(x)) \rightarrow z$. If the answer is "yes" then return $H(x)$ and Halt.
8. If $M(h_1(z), \ldots, h_t(z)) = \star$ then set $M(h_1(z), \ldots, h_t(z))$ to 1, set $Z \leftarrow Z \cup \{z\}$, and goto 6.
9. If $M(h_1(z), \ldots, h_t(z)) \neq \star$ then there exists $z' \in Z$ such that

$$(h_1(z), \ldots, h_t(z)) = (h_1(z'), \ldots, h_t(z')).$$

10. $A \leftarrow A \cup \{z, z'\}$.
11. goto 2.

The following sequence of claims yields the theorem. The first claim bounds the number of non-$\star$ entries in $M$.

**Claim 2** *For every $t$ as above (i.e., $t$ is the number of hypotheses $h_i$), the number of entries $y \in \{0,1\}^t$ such that $M(y) \neq \star$ is at most $t^v$.*

**Proof:** Notice that $M(y) \neq \star$ implies that $y = (h_1(x), \ldots, h_t(x))$ for some $x$ (in $A$ or in $Z$). Therefore, the number of non-star entries of $M$ is at most the number of different values of $(h_1(x), \ldots, h_t(x))$ over all $x$. These are simply vectors in the space $\mathcal{H}^{\perp}$. Since $v = \text{VC-dim}(\mathcal{H}^{\perp})$ then using Sauer lemma the result follows. □

The next claim shows that if in the collection of our hypotheses we have the correct functions $g_1, \ldots, g_m$ then the protocol will halt.

**Claim 3** *If $g_1, \ldots, g_m \in \{h_1, \ldots, h_t\}$ then the algorithm will only execute steps 6-9 several times (at most $t^v$) and then will get the answer "yes" to one of its equivalence queries (in which case the algorithm halts).*

**Proof:** We will show that if $g_1, \ldots, g_m \in \{h_1, \ldots, h_t\}$ then for every counterexample $z$ we have $M(h_1(z), \ldots, h_t(z)) = \star$ (hence, in this case, the condition in step 8 holds and the condition in step 9 does not). This implies that the algorithm will execute only steps 6-9 (see the condition in step 9). Suppose $M(h_1(z), \ldots, h_t(z)) \neq \star$. Then (by the description of the algorithm) there is an assignment $z' \in A \cup Z$ such that $(h_1(z), \ldots, h_t(z)) = (h_1(z'), \ldots, h_t(z'))$. On the other hand since $z$ is a counterexample then (by the way $H$ is defined) $f(g_1(z), \ldots, g_m(z)) \neq f(g_1(z'), \ldots, g_m(z'))$. However, since we have $h_i(z) = h_i(z')$ for all $i$ and because $g_1, \ldots, g_m \in \{h_1, \ldots, h_t\}$ then in particular $g_i(z) = g_i(z')$ for every $i$. Therefore $f(g_1(z), \ldots, g_m(z)) = f(g_1(z'), \ldots, g_m(z'))$. A contradiction. □

The next claim shows that the number of times that the main loop (steps 2-11) is performed is at most $qm$.

**Claim 4** *We have $|A| \leq 2qm$.*

**Proof:** We will show that after executing steps 9-10 $r$ times we have: for every $g_i$ there is a hypothesis $h_{w(i)}$ that is equivalent to the hypothesis that we would get from running algorithm $ALG$ for $r_i$ phases for the target $g_i$ and $r = r_1 + \cdots + r_m$. By "step" we mean the running of the algorithm until a new equivalence query is asked. We show the above by showing that executing steps 9-10 is equivalent to running the algorithm $ALG$ one more step for some $g_i$. This will implies that steps 9-10 are executed at most $qm$ times and therefore $|A| \leq 2qm$.

To show the above, let $M(h_1(z), \ldots, h_t(z)) = \xi \neq \star$. There is $z'$ such that $(h_1(z), \ldots, h_t(z)) = (h_1(z'), \ldots, h_t(z'))$ and $f(g_1(z), \ldots, g_m(z)) \neq f(g_1(z'), \ldots, g_m(z'))$. From the latter we must have $g_i(z) \neq g_i(z')$ for some $i$. Now because $h_{w(i)}(z) = h_{w(i)}(z')$ we have that either $z$ or $z'$ is a counterexample for $h_{w(i)}$. Therefore using this counterexample will generate the hypothesis generated in $ALG$ in step $r_i + 1$. □

Now, since $|A| \leq 2qm$ then in each iteration (i.e., whenever we go through step 2) we have $t \leq (2qm+1)^{k_1} \cdot 2^{k_2}$. By Claim 2, the number of non-$\star$ entries in $M$ (in each iteration) is at most $t^v \leq (2qm+1)^{k_1 v} \cdot 2^{k_2 v}$. In particular the number of equivalence queries asked by the algorithm is at most $t^v$ in each iteration and over all at most $t^v \cdot qm \leq (2qm+1)^{k_1 v+1} \cdot 2^{k_2 v}$. In addition, if $B$ is a bound on the running time of $Alg$ then the running time of our algorithm is $O((2qm+1)^{k_1 v+1} \cdot 2^{k_2 v} \cdot B)$. This completes the proof of the theorem. □

**Remark:** The bound obtained by the above theorem can be somewhat improved for certain classes by using the notion of "consistent sign assignments" (see [W68, GJ95]). This can be done by noticing that Claim 2 uses only a bound on the number of such assignments (which is always bounded by $t^v$).

A particularly interesting case of the above theorem is when we allow the learning algorithm only a constant number of examples (i.e., $k_1 = O(1)$), additional $O(\log\log|X|)$ bits, and the VC-dim of the hypothesis class $\mathcal{H}$ is also bounded by a constant (this also implies that VC-dim($\mathcal{C}$) and VC-dim($\mathcal{H}^\perp$) are $O(1)$). In this case we get:

**Corollary 2** *Let $\mathcal{C}$ and $\mathcal{H}$ be classes of boolean functions over domain $X$, where VC-dim($\mathcal{H}$) $= O(1)$. If the concept class $\mathcal{C}$ is $(O(1), O(\log\log|X|))$-space bounded learnable by $\mathcal{H}$ using $q$ equivalence queries then the concept class $\mathcal{C}^\star$ is learnable by $\mathcal{H}^\star$ using $poly(q, m, \log|X|)$ equivalence queries where $m$ is the number of functions in $\mathcal{C}$ on which the target function is based.*

# 4 Space-Bounded Learning of Half-Spaces

In this section we present a constant-space algorithm to learn efficiently a halfspace over the $d$-dimensional grid $\{0, 1, \ldots, n\}^d$ (where $d = O(1)$), that uses hypotheses from a class $\mathcal{H}$ of constant VC-dim (more precisely, the class of halfspaces).

First, if $f$ is non-constant then (by Claim 5 below) there exists a corner of the grid (i.e., a point in $\{0, n\}^d$) such that the halfspace gives this corner one value (0 or 1) and to the "neighbors" of the opposite corner the halfspace gives the opposite value. We will present our algorithm, $ALG$, under the assumption that we know what is this corner and what is its value. With this assumption we will show how to learn the halfspace in a constant-space manner. In addition, if the assumption is false, then $ALG$ will return FAIL. Based on $ALG$, we can easily construct a modified algorithm, $ALG2$ that goes over all possible corners and signs one-by-one (there are $2^d \cdot 2$ possibilities) and for each of them executes $ALG$. If $ALG$ succeeds we are done; otherwise we get the value FAIL and try the next possibility. All together the running time (and query complexity) grows by a constant multiplicative factor ($2^{d+1}$), while the space is increased by a constant $(d + 1)$ bits.

Without loss of generality, we assume that $\vec{0}$ is a corner as above and that it is a positive point (i.e., $f(\vec{0}) = 1$). Denote $y^P = \vec{0}$ and $y_i^N = \vec{n} - e_i$, where $1 \le i \le d$, $e_i$ denotes the $i$-th unit vector of length $d$, $\vec{0} = (0, \ldots, 0)$ and $\vec{n} = (n \ldots, n)$. Since $f(y^P) = 1$ and $f(y_i^N) = 0$ we know that the hyperplane that we try to learn passes through the interval $[y^P, y_i^N]$. If we will find the $d$ intersection points, then (by Claim 6 below) this determines the required hyperplane and identifies the target function.

The idea behind the algorithm is as follows. We will perform a so-call "nested binary search" for the intersection points of the hyperplane with the $d$ intervals. For this, we will maintain on each interval $[y^P, y_i^N]$ a sub-interval $[a_i^P, a_i^N]$ and a point $a_i$ in this sub-interval that satisfy the following invariant: if for every $j < i$ the intersection point of the hyperplane with $[y^P, y_j^N]$ is $a_i$ then the intersection point of the hyperplane with $[y^P, y_i^N]$ is in $[a_i^P, a_i^N]$. Roughly speaking, in a "nested binary search" if on each interval $[y^P, y_i^N]$ there are $\Delta$ possibilities to choose from and there are $d$ intervals then the search will end within $(\log \Delta)^d$ steps. One crucial point is that in order to meet the space constraints we cannot store in our memory the points $\{a_i^P, a_i^N \ : \ 1 \le i \le d\}$. Instead, we will store for each of these points a set of examples that we have seen (to be denoted $P_i$ and $N_i$, for $1 \le i \le d$) that determines the points (in a sense that will be defined below). The size of each of $P_i$ and $N_i$ is $3^{d-i}$ (also see Remark 4 below). With the above description in mind here is a detailed description of the algorithm:

1. **Define Hypothesis:**
   For $i = 1, \ldots, d$ do:
   Let $a_i^P$ be the closest point to $y^P$ in the interval $[y^P, y_i^N]$ such that there exists a hyperplane that passes through $a_1, \ldots, a_{i-1}, a_i^P$ and is consistent with the examples in $P_i$.[1] (Claim 9 below shows that this point can be efficiently computed.)
   Let $a_i^N$ be the closest point to $y_i^N$ in the interval $[y^P, y_i^N]$ such that there exists a hyperplane that passes through $a_1, \ldots, a_{i-1}, a_i^N$ and is consistent with the examples in $N_i$.
   Let $a_i$ be the middle point of $a_i^P$ and $a_i^N$; that is, $a_i = \frac{1}{2}(a_i^P + a_i^N)$.
2. Let $h$ be a hyperplane that passes through $a_1, \ldots, a_d$ (by Claim 6 below $h$ is well defined).
3. Ask $EQ(h) \to z$
4. If answer is YES, halt with output $h$. Otherwise, let $z$ be the counterexample received as an answer. Denote $P_{d+1} = \{z\}$ and $N_{d+1} = \emptyset$.

[1] Initially $P_i, N_i$ are empty. The interpretation of an empty set is as if it contains the points $\{y^P, y_1^N, \ldots, y_d^N\}$.

5. Recompute $a_1, \ldots, a_d$ as in step 1 (due to the algorithm being space-bounded it cannot store this information that was computed before the equivalence query).

6. **Memory Update:**
   For $i = d, \ldots, 1$ (i.e., backwards) do:
   If there is a hyperplane consistent with $a_1, \ldots, a_i$ and with all the examples in $S = (\bigcup_{i+1 \leq j \leq d+1} P_j) \cup (\bigcup_{i+1 \leq j \leq d+1} N_j)$ then memory update is over; goto step 1.
   If there is no such a consistent hyperplane (at least for $i = d$ this is the case as $z$ is a counterexample) then either (1) every hyperplane that passes through $a_1, \ldots, a_{i-1}$ and is consistent with $S$ passes through points in $[a_i^P, a_i)$ in which case we replace $N_i$ by $S$ (note that $|S| \leq 3^{d-i}$); or (2) every hyperplane that passes through $a_1, \ldots, a_{i-1}$ and is consistent with $S$ passes through points in $(a_i, a_i^N]$ in which case we replace $P_i$ by $S$; or (3) there is no hyperplane that passes through $a_1, \ldots, a_{i-1}$ that is consistent with $S$ and passes through $[y^P, y_i^N]$. In this case we do nothing and proceed to the next $i$ (for which again there will be no consistent hyperplane). If we reach case (3) for the first interval ($i = 1$) stop and output FAIL (this indicates a wrong choice of corner).

We now turn to the proof of correctness and complexity of the above algorithm. While doing so we will also clarify some points which are not completely specified by the above description. The first claim proves the existence of a corner as assumed above. That is, a point $y \in \{0, n\}^d$ such that the halfspace gives $y$ a value $\sigma \in \{0, 1\}$ and to the neighbors of the opposite corner the halfspace gives the opposite value.

**Claim 5** *Consider the grid $\{0, 1, \ldots, n\}^d$ ($n \geq 2$). Let $f(x)$ be a boolean function of the form "$c_1 x_1 + \ldots + c_d x_d \geq b$" which is not-constant over $\{0, 1, \ldots, n\}^d$. Then, there exist $y \in \{0, n\}^d$ and $\sigma \in \{0, 1\}$ such that $f(y) = \sigma$ and for each $y'$, a neighbor of $\vec{n} - y$ (in the $d$-dimensional cube) $f(y') = 1 - \sigma$.*

**Proof:** First, observe that we can assume without loss of generality that $c_i \geq 0$ for all $i$ (otherwise we can map the cube to itself by transforming $x_i$ to $n - x_i$ and look at the function $f'$ which has the same coefficients as $f$ except that $c'_i = -c_i$ and $b' = b - c_i n$. Since $f'$ has fewer negative coefficients, we can first find a corner as desired for $f'$, and then transform it back by applying again the transformation $x_i$ to $n - x_i$). Now, observe that $f(\vec{0})$ must be 0 (since $\vec{c} \cdot \vec{0} = 0$ and so if $f(\vec{0}) = 1$ then $b < 0$ and the function is constantly 1) and that $f(\vec{n})$ must be 1 (since if $f(\vec{n}) = 0$ then $b > n \cdot \sum c_i$ and the function is constantly 0). Assume, without loss of generality, that $c_1 \leq c_2 \leq \ldots \leq c_d$. Consider two cases: (1) If $c_1 \leq \ldots \leq c_d < b$ then $f(e_i) = 0$ for all $i$ (as $\vec{c} \cdot e_i = c_i < b$). In this case $y = \vec{n}$ is an appropriate corner. (2) If $c_1 \leq \ldots \leq c_r < b$ and $c_{r+1} = \ldots = c_d = b$ (larger coefficients can be reduced without changing the function) then $f(\vec{n} - e_i) = 1$ for all $i$ (as $\vec{c} \cdot (\vec{n} - e_i) \geq n c_d - c_i \geq b$). In this case $y = \vec{0}$ is an appropriate corner. □

The next claim argues that after finding the points $a_1, \ldots, a_d$ there exists a unique hyperplane that passes through these points (here we build on the particular choice of $d$ intervals $[y^P, y_i^n]$ that we made). Note that since $y^P$ is a corner, the hyperplane can always be "pushed away" from this corner without changing the classification of any point. This implies that $a_i^N$ is always different than $y^P$ and hence $a_i \in (y^P, y_i^N]$.

**Claim 6** *For any points $a_1, \ldots, a_d$ such that $a_i \in (y^P, y_i^N]$, there exists a unique hyperplane $h$ that passes through these points*

**Proof:** As mentioned, we assume that $y^P = \vec{0}$ and $y_i^N = \vec{n} - e_i$ (otherwise we make the appropriate transformation of the cube). Since $a_i$ is a point on the interval $(y^P, y_i^N]$ we can write $a_i = \lambda_i(\vec{n} - e_i)$ $(0 < \lambda_i \leq 1)$. To show the uniqueness of the hyperplane it suffices to show that the matrix

$$\begin{pmatrix} a_1 \\ \vdots \\ a_d \\ \vec{1} \end{pmatrix},$$

where $\vec{1} = (1, \ldots, 1)$, is non-singular. For this we write

$$\det \begin{pmatrix} \lambda_1(\vec{n} - e_1) \\ \vdots \\ \lambda_d(\vec{n} - e_d) \\ \vec{1} \end{pmatrix} = \det \begin{pmatrix} -\lambda_1 e_1 \\ \vdots \\ -\lambda_d e_d \\ \vec{1} \end{pmatrix} = (-1)^d \lambda_1 \cdots \lambda_d \neq 0.$$

□

The next two claims establish the complexity of the algorithm. The first says that we always make a significant progress on one of the intervals (without hurting any of the previous intervals). The second bounds the number of steps the binary search needs to go through on each interval. Note that although the points of interest to us are grid points, the points that we compute during the algorithm are not necessarily grid points.

**Claim 7** *Assume that during the Memory-Update (Step 6) the memory corresponding to intervals $j, j+1, \ldots, d$ was updated (and that no change was made for $1, \ldots, j-1$). Then $[a_i^P, a_i^N]$, for $i \leq j-1$ remains unchanged and $[a_j^P, a_j^N]$ is reduced by a factor of at least 2 relative to its previous size. (There is no claim regarding $[a_i^P, a_i^N]$, for $i > j$.)*

**Proof:** By the description of the algorithm, if the corresponding $P_i, N_i$ are not changed then so is the interval $[a_i^P, a_i^N]$. Hence for $i < j$ the claim follows. As for the $j$-th interval, by convexity if there are halfspaces that pass through $a_1, \ldots, a_{j-1}$, intersect $[a_i^P, a_i)$ and are consistent with $S$, and there are also halfspaces that pass through $a_1, \ldots, a_{j-1}$, intersect $(a_i, a_i^N]$ and are consistent with $S$ then there must also exist a halfspace that passes through $a_1, \ldots, a_{j-1}, a_j$ and is consistent with $S$. This (by the description of the algorithm) contradicts the fact the memory corresponding to the $j$-th interval was updated. Therefore, only one of (1) or (2) can happen in Step 6. Say, (1) is the case. This implies that $P_j$ remains unchanged (and therefore so is $a_j^P$) while $N_i$ is replaced by $S$ which implies that next time $a_i^N$ will be in the current interval $[a_i^P, a_i)$. Since $a_i$ is the middle point of the current interval then the size of the interval is reduced by at least a factor of 2. □

**Claim 8** *All the points computed by the algorithm on the $i$-th interval are of the form $(\frac{\alpha_1}{\alpha}, \ldots, \frac{\alpha_d}{\alpha})$ for integers $\alpha_1, \ldots, \alpha_d, \alpha$ where $\alpha \leq M_i = (dn)^{d^{O(i)}}$.*

**Proof:** The hyperplane that we search for is defined by grid points (since we are only interested in the value of the function on the grid then even if the hyperplane is not defined by grid points we can slightly "rotate" it to be so). It is a well known fact [Mur71] that the equation of such a hyperplane can always be written as "$c_1x_1 + \ldots + c_dx_d \geq b$" for $c_1, \ldots, c_d, b$ which are all integers bounded in absolute value by $(dn)^d$. Suppose we are looking for an intersection point of such a hyperplane (consistent with a given set of examples) with the first interval $[y^P, y_1^N]$. This intersection point $u$ can be expressed as $\lambda y^P + (1-\lambda)y_1^N$ for $0 \leq \lambda \leq 1$. Since $u$ is an intersection point, $b = \vec{c}u = \lambda\vec{c}y^P + (1-\lambda)\vec{c}y_1^N$, which gives

$$\lambda = \frac{b - \vec{c}y_1^N}{\vec{c}y^P - \vec{c}y_1^N}.$$

The denominator of this expression is bounded by $2n(dn)^d$ (and the nominator is obviously an integer). This implies that both $a_1^P, a_1^N$ produced by our algorithm are vectors of rational numbers the nominator of each is an integer and the denominator is bounded by $2n(dn)^d$. For computing $a_i$ we take the average of the two numbers whose denominator is at most $2(2n(dn)^d)^2 = (dn)^{O(d)} \leq M_1$.

To see what happens with respect to the second interval we need to be a bit more careful. The reason is that on top of what we have for the first interval we also have the requirement that the hyperplane passes through the point $a_1$ which is of the form $(\frac{\alpha_1}{\alpha}, \ldots, \frac{\alpha_d}{\alpha})$ for $\alpha \leq M_1$. In order to be able to use the same line of proof as above, we simply multiply (for the purpose of analysis only) all our numbers by $\alpha$. This makes all our numbers into integers on a new grid with $n' = n\alpha$ points. A similar argument to the one above implies that we can get numbers with denominator $(dn')^{O(d)}$. To go back to numbers in the original scale we divide by $\alpha$ and so the denominator can grow to $(dn')^{O(d)}\alpha \leq (d(dnM_1)^{O(d)})^{O(d)} \cdot M_1 \leq M_2$. Continuing this way the claim follows. $\square$

The next claim shows that all the computations done during the protocol can be done efficiently.

**Claim 9** *The rational values $a_i^P, a_i^N$ as described in the protocol (and in Claim 8 above) can be efficiently computed.*

**Proof:** Again, an hyperplane is defined by an equation of the form "$c_1x_1 + \ldots + c_dx_d = b$". The hyperplane passes through a point $x$ if $\vec{c} \cdot x = b$; the point $x$ is a positive example if $\vec{c} \cdot x \geq b$, and it is a negative example if $\vec{c} \cdot x < b$; the hyperplane passes through an interval $[x, x']$ if $\vec{c} \cdot (\lambda x + (1-\lambda)x') = b$ for some $0 \leq \lambda \leq 1$. Therefore, deciding whether there exists a hyperplane consistent with a set of examples and passes through a set of points is just deciding whether a polytope defined by the corresponding linear constraints is not empty. By the theory of linear programming (see, for example, [Kar91]), to find the extreme points of this polytope it suffices to choose out of the above constraints (whose number is essentially $d$ plus the size of the sets $P_i, N_i$) all subsets of $d+1$ constraints (there are exponential in $d$ many subsets) and solve them with equality; the solution for each subset can be done in polynomial time. All together, this takes time polynomial in $\log n$ and exponential in $d$. $\square$

We can now state the results that we get:

**Theorem 3** *Let $\mathcal{C}$ be the class of halfspaces over $\{0, 1, \ldots, n\}^d$. The above algorithm ALG2 learns a target halfspace in $\mathcal{C}$ using $(3^d, d+1)$-space and its time and query complexity are polynomial in $\log n$ and exponential in $d$. Furthermore, the hypotheses used by the algorithm are from the class of $d$-dimensional halfspaces whose VC-dim is $d+1$.*

**Proof:** For the search on the $i$-th interval we start from the whole interval whose length is at most $\sqrt{nd}$, whenever we make a progress on an interval we cut it by a factor of at least 2 and when the interval is of size smaller than $1/M_i$ then by Claim 8 there is a single intersection point of the form found by the algorithm. Therefore, at most $\log(\sqrt{nd}M_i) = d^{O(i)} \log n$ steps are required to search on the $i$-th interval. Therefore, overall the nested binary search needs $(d^{O(d)} \log n)^d$ steps.

The space used by the algorithm are all the sets $P_i, N_i$ used by $ALG1$ and in addition the corner information required by $ALG2$. This information is of the required size. $\square$

**Remark 4** *The sizes of the sets $P_i, N_i$ can always be reduced to $d+1$ points hence the total space to $O(d^2)$. The reason is that in every set of labeled examples of size larger than $d+1$ there is a subset of size $d+1$ that defines the same halfspaces.*

**Corollary 5** *There exists an algorithm that learns any function in $\mathcal{C}^\star$; that is, any function over $\{0,1,\ldots,n\}^d$ which is defined by a combination of halfspaces. The time and query complexity is polynomial in $\log n$ and $m$ (the number of halfspaces that define the target) and exponential in $d$.*

**Proof:** Combine Theorem 3 with Corollary 2. $\square$

# 5 Extensions

The results of the previous section can be extended in a very natural way to deal with constant degree semi-algebraic functions (as defined in Section 2.3). This follows from the existence of a simple reduction from degree-$k$ surfaces to halfspaces (in a larger dimension). This reduction works as follows: we translate each term of the form $x_1^{\alpha_1} \cdot x_2^{\alpha_2} \cdots x_d^{\alpha_d}$, where $0 \le \alpha_i \le k$, into a new variable $y_i$. The number of new variables is therefore $d' = (k+1)^d$. Since each variable $x_j$ gets values which are integers in the range $\{0,\ldots,n\}$ then the $y_i$'s get values in the range $\{0,\ldots,n'\}$ for $n' = n^{kd}$. With this notation there is a one-to-one correspondence between degree $k$ polynomials in $x_1,\ldots,x_d$ and hyperplanes on $y_1,\ldots,y_{d'}$. Therefore, to learn a semi-algebraic function $f(x)$ we use our algorithm from Section 4 to learn the corresponding hyperplane $f'(y)$. Whenever we get an example $x$ we transform it into an example $y$ by the above transformation. Whenever the algorithm asks an equivalence query $h'(y)$ (where $h'$ is a halfspace) we translate it into $h(x)$ (where $h$ is a degree $k$ polynomial). It is well known that the VC-dim of the class of hypotheses (i.e., the class of degree $k$ polynomials over $x_1,\ldots,x_d$) is at most $d'+1$. Hence by using the results of Section 4 and our composition theorem we get:

**Theorem 6** *For any two constants $k, d$, there exists an algorithm that learns the class of degree-$k$ semi-algebraic functions over $\{0,1,\ldots,n\}^d$. The time and query complexity is polynomial in $\log n$ and $m$ (the number of surfaces that define the target).*

# References


[AFHM93] F. Ameur, P. Fischer, K. U. Hoffgen, and F. Meyer auf der Heide, "Trial and Error: A New Approach to Space-Bounded Learning", *Proc. of 1st EuroCOLT*, pp.133-144, 1993.

[Ame94] F. Ameur, "Space-Bounded Identification of Halfplanes in the Discrete Grid", *Proc. of AAAI Fall Symp.*, pp. 5-8, 1994.

[Ame95] F. Ameur, "A Space Bounded Learning Algorithm for Axis-Parallel Rectangles", *Proc. of 2nd EuroCOLT*, LNAI, Vol. 904, pp. 313-321, 1995.

[A88] D. Angluin. Queries and Concept Learning. *Machine Learning*, pp 319–342, 2, 4, 1988.

[Aue93] P. Auer. "On-line Learning of Rectangles in Noisy Environments", *Proc. of 6th Annu. ACM Workshop on Comput. Learning Theory*, pages 253–261, 1993.

[BCH94] N. H. Bshouty, Z. Chen, and S. Homer. "On Learning Discretized Geometric Concepts", *Proc. of 35th Annu. Symp. on Foundations of Computer Science*, pages 54–63, 1994.

[BGMST96] N. H. Bshouty, S. A. Goldman, H. D. Mathias, S. Suri, and H. Tamaki. "Noise-Tolerant Distribution-Free Learning of General Geometric Concepts" *Proc. of the 28th Annu. ACM Symp. on Theory of Computing*, pages 151–160, 1996.

[CM92] Z. Chen and W. Maass. "On-line Learning of Rectangles", In *Proc. of 5th Annu. ACM Workshop on Comput. Learning Theory*, 1992.

[Flo89] S. Floyd, "On Space-Bounded Learning and the Vapnik-Chervonenkis Dimension", TR 89-061, ICSI, Berkeley, 1989.

[FW95] S. Floyd, and M. K. Warmuth, "Sample Compression, Learnability, and the Vapnik-Chervonenkis Dimension", *Machine Learning*, 1995.

[GJ95] P. W. Goldberg, and M. R. Jerrum, "Bounding the Vapnik-Chervonenkis Dimension of Concept Classes Parameterized by Real Numbers", *Machine Learning*, Vol. 18, pp. 131-148, 1995. Early version COLT93.

[HSW90] D. Helmbold, R. Sloan, and M. K. Warmuth, "Learning Nested Differences of Intersection-Closed Concept Classes", *Machine Learning*, Vol 5, pp. 165-196, 1990.

[Kar91] H. Karloff, "Linear Programming", Birkhauser, 1991.

[L88] N. Littlestone, "Learning when Irrelevant Attributes Abound: A New Linear-Threshold Algorithm" *Machine Learning*, Vol. 2, pp. 285–318, 1988.

[L89] N. Littlestone, "Mistake bounds and logarithmic linear-threshold learning algorithms" PhD thesis, U.C. Santa Cruz, March 1989.

[MT89] W. Maass and G. Turan. "On the Complexity of Learning from Counterexamples", *Proc. of 30th Annu. Symp. on Foundations of Computer Science*, pages 262–273, 1989.

[Mur71] S. Muroga, "Threshold Logic and its Applications", Wiley Interscience, 1971.

[Sim95] H. U. Simon, "Learning Decision Lists and Trees with Equivalence-Queries", *Proc. of 2nd EuroCOLT*, LNAI, Vol. 904, pp. 322-336, 1995.

[Val84] L. G. Valiant. "A Theory of the Learnable", *Communications of the ACM*, 27(11):1134–1142, 1984.

[W68] H. E. Warren, "Lower Bounds for Approximation by Non-linear Manifolds", *Transactions of the AMS*, 133, pp. 167-178, 1968.

[WW87] E. Welzl, and G. Woeginger, "On Vapnik-Chervonenkis Dimension One", unpublished 1987.

## A Space-Bounded Learning of Classes with VC-dim $= 1$

In this section we show that every class of boolean functions over a finite domain $X$ whose VC-dim is 1 can be learned in a space-bounded manner, using $poly(\log |X|)$ equivalence queries (though, the algorithm may not be computationally efficient). For this, we take the following view on classes of VC-dim $=$ 1 based on [WW87]. Let $\mathcal{C}$ be a concept-class over a finite domain $\mathcal{X}$. Without loss of generality, assume that $\mathcal{C}$ is non-redundant; that is, there are no two points $x, y \in \mathcal{X}$ such that

$\forall f:\ f(x) = f(y)$ or $\forall f:\ f(x) \neq f(y)$. Next, we pick some (arbitrary) function $g \in \mathcal{C}$ and define for all $x, y \in \mathcal{X}$

$$x \leq_{\mathcal{C}} y \qquad \text{if} \qquad \forall f \in \mathcal{C} \ \ (f(x) = g(x) \ \Rightarrow \ f(y) = g(y)).$$

We observe that $\leq_{\mathcal{C}}$ is a partial order. First, if $x \leq_{\mathcal{C}} y$ and $y \leq_{\mathcal{C}} z$ then by definition $x \leq_{\mathcal{C}} z$ (i.e., $\leq_{\mathcal{C}}$ is transitive). Second, if both $x \leq_{\mathcal{C}} y$ and $y \leq_{\mathcal{C}} x$ then, by the definition, $\forall\ f \in \mathcal{C}$, $(f(x) = g(x) \iff f(y) = g(y))$, which by the assumption that $\mathcal{C}$ is non-redundant implies that $x = y$ (i.e., $\leq_{\mathcal{C}}$ is a-symmetric). Since $\leq_{\mathcal{C}}$ is a partial order, we can organize the elements of $\mathcal{X}$ in a tree in a way that $x$ is an ancestor of $y$ if and only if $x \leq_{\mathcal{C}} y$ (we assume for simplicity that $\leq_{\mathcal{C}}$ has a minimal element; if this is not the case we put in the root of the tree a dummy element).

Now, we claim that if VC-dim$(\mathcal{C}) = 1$ then every function $f \in \mathcal{C}$ corresponds to a branch from the root of the tree to some node of the tree. By this we mean that, for every $x$ which is not on this branch, $f(x) = g(x)$ while for every $x$ on this branch $f(x) = 1 - g(x)$. (The other direction, that such a class $\mathcal{C}$ has VC-dim$(\mathcal{C}) = 1$ is simple.) To see that each function in $\mathcal{C}$ corresponds to a branch, first note that if for some $y$ we have $f(y) \neq g(y)$ then for each ancestor $x$ of $y$ (by the definition of $\leq_{\mathcal{C}}$) we also have $f(x) \neq g(x)$. Also note that if for some incomparable $x$ and $y$ there exists a function for which $f(x) \neq g(x)$ and $f(y) \neq g(y)$ then the class $\mathcal{C}$ shatters the set $\{x, y\}$. This is because $g$ gets some values on $x$ and $y$, $f$ gets the opposite values, and by the fact that $x$ and $y$ are incomparable according to $\leq_{\mathcal{C}}$ it follows that there exist functions $f_1, f_2$ such that $f_1(x) = g(x), f_1(y) \neq g(y)$ and $f_2(x) \neq g(x), f_2(y) = g(y)$; this implies that VC-dim$(\mathcal{C}) \geq 2$ which is a contradiction.
Now, using this structure we show the following:

**Theorem 7** *Let $\mathcal{C}$ be a concept class over a finite domain $\mathcal{X}$ such that VC-dim$(\mathcal{C}) = 1$. Then, there exists a $(2,0)$-space bounded algorithm that learns $\mathcal{C}$ using $(\log|\mathcal{X}|)^2$ equivalence queries, from a class $\mathcal{H}$ with VC-dim$(\mathcal{H}) = 1$.*

**Proof:** First, by the above discussion, if we have in our memory a point $x_+$ such that $f(x_+) \neq g(x_+)$ then this gives the correct classification for all points which are not in the subtree defined by $x_+$: for all points $x$ on the branch from the root to $x_+$, we have $f(x) \neq g(x)$, while for all points $x$ which are not in the subtree of $x_+$ but also not on the branch leading to $x_+$, we have $f(x) = g(x)$. In some cases the algorithm will also have another point $x_-$ such that $f(x_-) = g(x_-)$ and $x_+$ is an ancestor of $x_-$.

Our algorithm works as follows. First it obtains a point $x_+$ by asking $EQ(g)$ and getting a counterexample (or, if the target function is $g$, then we are done). Now the algorithm works in iterations (until identifying the target function) where each iteration is as follows :

1. If the algorithm has in its memory only one point $x_+$ it looks at the subtree of $x_+$, say it has $t$ nodes, and finds a node $y_1$ in the subtree such that the number of nodes in the subtree of $x_+$ which are not in the subtree of $y_1$ is at most $t/2$ and the number of nodes in the subtree of each child of $y_1$ is also at most $t/2$ (the number of children may be large). To see that such a node exists, start with node $x_+$, and at each step proceed to the child of the current node with maximal number of nodes in its subtree. Once this number is smaller than $t/2$ stop and the current node is the desired $y_1$. Now the algorithm defines a hypothesis $h$ using the branch leading to $y_1$ and asks an equivalence query $EQ(h)$. If the answer to the query is "YES" it stops with output $h$; otherwise, it gets a counterexample $z$. If $f(z) \neq g(z)$ (i.e., $z$ is either in the subtree of $y_1$ or in the subtree of $x_+$ but not on the path to $y_1$) then the algorithm sets $x_+ \leftarrow z$ while if $f(z) = g(z)$ (i.e., $z$ is on the path connecting $x_+$ and $y_1$) then the algorithm sets $x_- \leftarrow z$.

2. If the algorithm has in its memory two points $x_+, x_-$ then the algorithm performs a binary search on the path connecting $x_+$ and $x_-$ as follows: the algorithm picks the node $y_2$ which is the middle point on the path connecting $x_+$ and $x_-$; the algorithm defines an hypothesis $h$ using the branch leading to $y_2$ and asks an equivalence query $EQ(h)$. If the answer to the query is "YES" it stops with output $h$. Otherwise, if the counterexample $z$ is a point on the path connecting $x_+$ and $x_-$ satisfying $f(z) \neq h(z)$ (i.e., a point below $y_2$) the algorithm sets $x_+ \leftarrow z$; if the counterexample $z$ is a point on the path connecting $x_+$ and $x_-$ satisfying $f(z) = h(z)$ (i.e., a point above $y_2$) the algorithm sets $x_- \leftarrow z$; if $z$ is not at the path at all (in this case $f(z) \neq h(z)$) the algorithm sets $x_+ \leftarrow z$ and removes $x_-$ (that is, in the next iteration it will perform Step 1).

Since Step 2 performs a binary search on the path connecting $x_+$ and $x_-$ then each sequence of iterations that perform Step 2 is of length at most the logarithm of the length of this path which is at most $\log|\mathcal{X}|$. Such a sequence ends with either identifying $f$ or with a new $x_+$ outside the path. The number of times that Step 1 is performed is $\log|\mathcal{X}|$, because at each step we either find a new $x_+$ which by the definition of $y_1$ has in its subtree at most $1/2$ of the number of nodes that we had before, or we start a sequence of iterations of Step 2 in which (unless we identify $f$ before) we find a new $x_+$ outside the path to $x_-$ (and $y_1$) in which again, by the definition of $y_1$, leaves us with at most half the size of the tree. All together the number of iterations is bounded by $(\log|\mathcal{X}|)^2$ and the space is bounded by 2 points. $\square$

**Corollary 8** *For every class $\mathcal{C}$ as above, the class $\mathcal{C}^\star$ is learnable using number of queries which is polynomial in $\log \mathcal{X}$ and $m$.*