CSC 2416 Project: Machine Learning & the Automatizability of Proof Systems

Alexander & Philipp Hertel

December 16th, 2005

Contents

1 Introduction

Proof complexity and machine learning theory seem to overlap in a very interesting way. Namely, automatizability in proof complexity appears to be very closely related to learnability in machine learning theory [ABF$^+$04].

Intuitively, a proof in proof system P can be viewed as a circuit from the corresponding underlying circuit class $C(P)$. Therefore, if an efficient proper PAC learning algorithm exists for $C(P)$ then perhaps this algorithm can used to automatize P, since any algorithm which properly learns $C(S)$ efficiently is capable of efficiently finding a specific poly-sized circuit / proof.

If this connection can be made formal, it has obvious negative implications; namely if a proof system S is not automatizable, then $C(S)$ can not be properly PAC learned efficiently. A result such as the main theorem from [AR02], which shows that under the assumption that $\mathcal{W}[\mathcal{P}]$ is not tractable Resolution is not automatizable, would strongly suggest that there is no efficient PAC learning algorithm for DNF formulas, Resolution's apparent underlying circuit class.

However, the negative implications would work in the other direction as well. Consistency model algorithms are very simple, and can nearly always lead to PAC learning algorithms given a large enough sample set. Therefore, if a circuit class $C(P)$ is not PAC learnable, then that may rule out a large class of algorithms as candidates for automatizing P. In fact, it may be the case that an inability to PAC learn $C(P)$ efficiently even implies the non-automatizability of P. Efficient PAC learning and automatizability may be totally equivalent.

The purpose of this paper is to provide a survey of the literature pertaining to learnability and automatizability with an emphasis on the learnability of intersections of halfspaces and the automatizability of the Cutting Planes proof system.

This survey is laid out as follows: In Section 2 we discuss various proof systems whose underlying circuit classes are of interest to machine learning. In Section 3 we discuss what is known about the automatizability of these proof systems. Next, in Section 4 we discuss the connections which have been observed between proof systems and their corresponding circuit classes. In Section 5, we shall discuss what is known about the learnability of these circuit classes. Finally, in Section 6 we relate learnability to the automatizability of proof systems and suggest avenues of future research.

2 Proof Systems

We shall use the standard Cook-Reckhow definition for proof systems:

Definition 1. [CR74, Rec76, CR79] *A **proof system** for a language $L \subseteq \Sigma^*$ is a poly-time computable onto function $f : \Sigma_p^* \to L$ where Σ_p is some alphabet.*

Intuitively, Σ_p is an alphabet of 'proof symbols', and f maps proofs to the elements in L which they prove. The words 'poly-time computable' enforce our intuitive notion that when given a proof, we should be able to tell (compute) what exactly it is that the proof is trying to prove within a feasible amount of time.

Proof systems must be sound; for example, a proof system for tautologies should not be able to prove a non-tautology. Similarly, proof systems must be complete; for example, a proof system for unsatisfiability must be capable of proving the unsatisfiability of every possible unsatisfiable formula. Another characteristic of the Cook-Reckhow definition is that it implicitly demands soundness and completeness. Soundness is guaranteed because in practice f is also total, since syntactically incorrect proofs are usually all mapped to a dummy element in L. In other words, no proof is mapped outside of L. The fact that f is onto guarantees completeness.

A central concept in the area of proof complexity is that of p-simulation, since it allows us to objectively compare the strength two proof systems:

Definition 2. *Let $f_1 : \Sigma_1^* \to L$ and $f_2 : \Sigma_2^* \to L$ be proof systems for L. If for all $x_1 \in \Sigma_1^*$, there exists an $x_2 \in \Sigma_2^*$ such that $f_1(x_1) = f_2(x_2)$ where $|x_2| \leq p(|x_1|)$ for some polynomial p, and if there exists a polytime computable function $t : \Sigma_1^* \to \Sigma_2^*$ such that for all $x \in \Sigma_1^*$, $f_1(x) = f_2(t(x))$, then we say that f_2 **p-simulates** f_1.*

*Proof systems that p-simulate each other are said to be **p-equivalent**.*

We will focus on the Resolution, Cutting Planes, Bounded-depth Frege, and Polynomial Calculus proof systems, and use the concept of p-simulation to tie them together.

2.1 Resolution

Resolution in all of its many forms is perhaps the best-studied propositional proof system, and is used as a refutation system for unsatisfiable CNF formulas. The resolution rule is quite simple: given two clauses $(A \vee x)$ and $(B \vee \overline{x})$, we resolve on the variable x to derive the new clause $(A \vee B)$. A resolution refutation of a contradictory set of clauses consists of the application of a sequence of resolution steps until the empty clause $\emptyset$ is derived. This proof can be written as a sequence of clauses in which each is either an initial clause or follows by the resolution rule from two previous ones. It is also possible to represent a resolution proof as a directed graph instead of as a sequence of clauses. The vertices of these graphs are labeled with clauses. Initial clauses have in-degree 0, whereas all other clauses have in-degree 2, with the edges indicating which clauses they were derived from.

Tree Resolution

If the underlying structure of a resolution proof forms a tree, then we say that the proof is 'tree-like'. Intuitively, this implies that each clause may only be resolved on once, so if a clause needs to be used again, it must be re-derived. Tree Resolution (TRES) is Resolution in its weakest form, and its exponential lower bounds are easily understood.

The algorithmic form of TRES is called DLL and is a centrally important algorithm in the field of automated theorem proving.

General Resolution

General Resolution (RES) is resolution in which the underlying structure of the proof forms a directed acyclic graph (DAG). Exponential lower bounds for RES were proved in [Hak85] by Haken, who used the pigeonhole principle (PHP). The PHP is a combinatorial principle which states that it is impossible to put n pigeons into $n-1$ holes such that no two pigeons share the same hole. The formula encoding the negated PHP is called PHP^n_{n-1}, and since the PHP is clearly true, the formula PHP^n_{n-1} must be unsatisfiable. Haken showed that any RES proof of PHP^n_{n-1} requires exponentially many clauses.

2.2 Cutting Planes

Motivated by integer programming, Cutting Planes (CP) is a proof system that is used to refute a set of linear inequalities. A linear inequality is of the form $\Sigma_i a_i x_i \geq k$, where a_i and k are integers, and the underlying variables are x_i.

Formally, CP is defined as follows:

Axioms

1. $x_i \geq 0$
2. 1 - $x_i \geq 0$

Rules

1. **Addition of Inequalities**

$$\sum_{i=1}^{n} a_i x_i \geq k, \sum_{i=1}^{n} b_i x_i \geq k^{'} \Rightarrow \sum_{i=1}^{n} (a_i + b_i) x_i \geq k + k^{'}$$

2. **Multiplication by a Constant**

$$\sum_{i=1}^{n} a_i x_i \geq k \Rightarrow \sum_{i=1}^{n} c \cdot a_i x_i \geq c \cdot k$$

3. **Division with Ceiling**

$$\sum_{i=1}^{n} t \cdot a_i x_i \geq k \Rightarrow \sum_{i=1}^{n} a_i x_i \geq \left\lceil \frac{k}{t} \right\rceil$$

For Example:

$2x_1 + 2x_2 \geq 3 \Rightarrow x_1 + x_2 \geq 2$

A CP refutation of a set of linear inequalities $L = \{C_1, C_2, \ldots, C_q\}$ is a sequence of inequalities, S_1, $S_2, \ldots, S_m$ where each S_i is either from L or is an axiom or follows from two previous inequalities by a valid rule, and the final inequality $S_m = (0 \geq 1)$. A CP proof may equivalently be viewed as a DAG. A more restricted form of CP is tree-like CP (TCP) in which the underlying DAG of the proof is required to be a tree. The leaves of the tree each correspond to one of the initial linear inequalities, the root corresponds to the final inequality S_m, and the edges correspond to inference rules. An example of a tree-like CP proof is shown below in Figure 1.

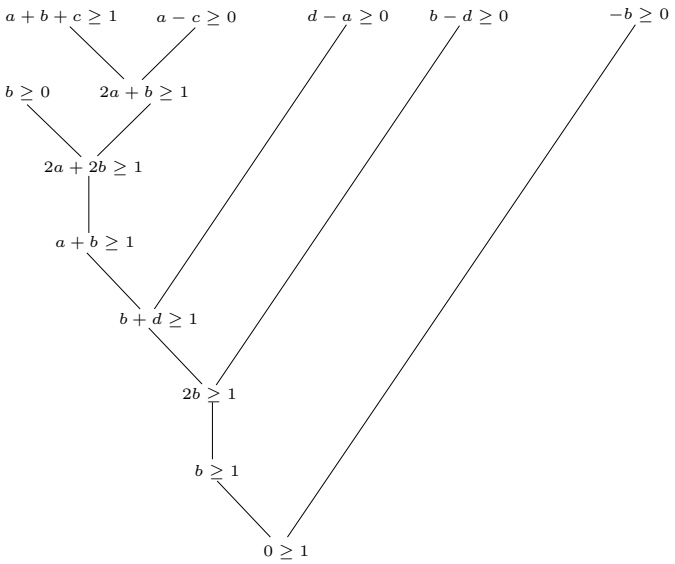

Figure 1: An Example of a Cutting Planes Refutation

In [BPR97], the authors describe a restricted form of TCP which we shall call TCP*. It is identical to TCP except that coefficients must be bounded by a polynomial.

2.3 Bounded-Depth Frege Systems

Frege systems (Frege), also called Hilbert-style systems, are a group of robust proof systems that are of particular interest in the area of proof complexity. Any sound and complete proof system that includes a finite number of axiom schemes instead of axioms is considered to be a Frege system. Axiom schemes are axiom templates that allow for a simultaneous and uniform substitution of any arbitrary formulas for sentence letters. For example, if a Frege system has an axiom scheme of the form $p \rightarrow p$, then a substitution of the formula $(q \vee r)$ for p yields the formula $(q \vee r) \rightarrow (q \vee r)$, which can be introduced at any point in a proof. In effect, Frege systems have an infinite number of axioms. Tree-like and DAG-like Frege systems are p-equivalent [CK01, p.372].

When we place a restriction on Frege systems requiring that all formulas can have a depth of at most a constant d, we create a new class of proof systems called Bounded-Depth Frege systems ($\mathsf{AC}^0-\mathsf{Frege}$).

Definition 3. *The* depth *of a formula F is the minimum height among all unbounded fan-in circuits equivalent to F.*

For example, all CNF and all DNF formulas have depth 2.

2.4 Polynomial Calculus & Nullstellensatz

Nullstellensatz (NS) [CK01, p.308] and Polynomial Calculus (PC) [CK01, p.316] are algebraic refutation systems. PC is strictly stronger than NS [CK01, p.316], so we shall focus on it. PC is a refutation system for unsatisfiable sets of polynomial equations.

Formally, PC is defined in the following way [CK01, p.317]: Fix a field F and let $P \subseteq F[x_1, ..., x_n]$ be a finite set of multivariate polynomials over F. The axioms and rules of PC are as follows:

Axiom

1. $x_i^2 - x_i$ for $1 \leq i \leq n$

Rules

1. **Multiplication by a Variable**

   From polynomial p, infer $x_i \cdot p$ where $1 \leq i \leq n$.

2. **Linear Combination**

   From polynomials p and p', infer $a \cdot p + b \cdot p'$ where $a, b \in F$.

A PC refutation of a polynomial P is a derivation of 1 from P.

2.5 Relationships Between Proof Systems

Some relationships between proof systems are immediately obvious. For example, $\mathsf{AC}^0-\mathsf{Frege}$ systems p-simulate RES, which in turn p-simulates TRES. Similarly, CP obviously p-simulates TCP, which p-simulates TCP^*. It is also not hard to see that CP p-simulates RES (and that TCP p-simulates TRES) [CK01, p.345]. Furthermore, CP has poly-sized refutations of PHP formulas, which shows that even $\mathsf{AC}^0-\mathsf{Frege}$ cannot p-simulate it [CK01, p348].

Perhaps not as obvious is that TCP does not p-simulate resolution [CK01, p.365]. Furthermore, $\mathsf{AC}^0-\mathsf{Frege}$ and TCP^* do not p-simulate each other [BPR97]. As mentioned before, PC p-simulates NS, but the algebraic proof systems seem to be incomparable with the others, since $\mathsf{AC}^0-\mathsf{Frege}$ systems cannot p-simulate NS, and NS cannot even p-simulate TRES [BP01].

This hierarchy of p-simulation is shown below in Figure 2.

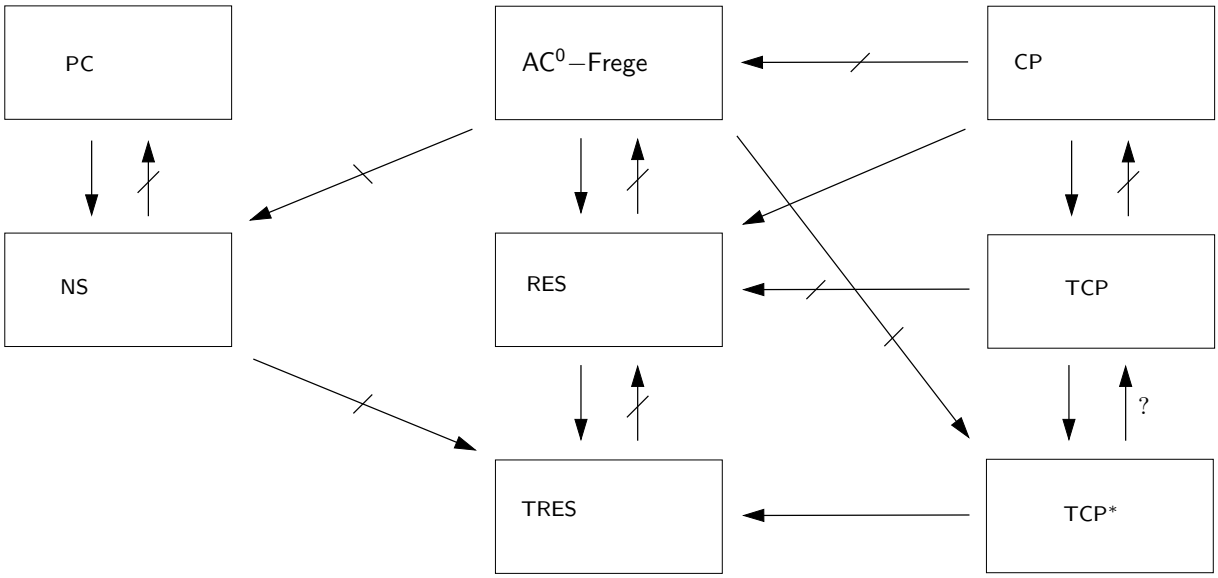

Figure 2: The Relationship Between The Various Proof Systems Being Discussed

It is not hard to see that since $\mathsf{AC}^0-\mathsf{Frege}$ systems cannot p-simulate TCP^* or NS, none of the proof systems in the middle column can p-simulate any of the proof systems in the left or right columns.

3 Automatizability

First formalized in [BPR97], automatizability is an important concept in proof complexity. Formally,

Definition 4. [BPR97] *A propositional proof system S is **automatizable** if there exists a deterministic algorithm A such that for every CNF formula F, A returns an S-proof of F in time polynomial in the size of the smallest S-proof of F. If A is as above except that it returns a proof of F that is not an S-proof, then S is said to be **weakly automatizable**.*

Intuitively, automatizability tells us if a proof system is useful in any practical sense. If a proof system is not automatizable, then it is impossible to implement a proof search strategy that is guaranteed to find proofs that are in any way comparable in length to the shortest possible proof. In other words, even if we have a powerful proof system that has very short proofs for certain formulas, if it is not automatizable, then its strengths are inaccessible from the point of view of automated theorem proving.

Note that if a proof system A p-simulates proof system B but B does not p-simulate A, then that does not let us conclude anything about the automatizability of A. Only when A and B are p-equivalent does that tell us anything about automatizability; ie. if A and B are p-equivalent then A is automatizable if and only of B is automatizable.

However, the situation is different with weak automatizability; if proof system A p-simulates proof system B, B does not p-simulate A, and A is automatizable, then B is weakly automatizable by the following algorithm: take any input x for B and translate it into $f(x)$, an input for A. Then run A's automatization algorithm. The resulting proof is guaranteed to be only polynomially larger than the smallest B proof of x since it is only polynomially larger than the smallest A proof, and we said that A p-simulates B.

3.1 Positive Results

The only proof systems known to be automatizable are NS and PC [BPR97], but to say that they are automatizable is not quite correct; there is a caveat. For example, PC is automatizable using the Groebner basis algorithm [CEI96], which is an $O(n^{3d})$ algorithm, where d is the largest degree of all equations in the input set. Since d could be as large as n, PC is only automatizable for certain inputs.

3.2 Negative Results

As for negative results, under various cryptographic assumptions, a variety of proof systems are known to be non-automatizable:

Theorem 5. [AR02] *If $W[P]$ is not tractable, then neither* TRES *nor* RES *are automatizable.*

Theorem 6. [CK01, p.393] *If $\forall \epsilon > 0$ the Diffie-Hellman cryptographic function cannot be computed by circuits of size 2^{n^ϵ}, then* AC^0-Frege *systems are not automatizable.*

Theorem 7. [BPR00] *If factoring Blum integers is hard, then* Frege *systems are not automatizable.*

Theorem 8. [CK01, p.393] *If RSA is secure, then* ExtendedFrege *systems are not automatizable.*

Each of these is considered to be good evidence of true non-automatizability since these cryptographic assumptions are widely believed to be true.

It is worth noting that most of these results were proved by first showing that under their respective cryptographic assumptions, the various proof systems have no feasible interpolation. Since automatizability implies feasible interpolation [CK01, p.298], the non-automatizability results followed. This is important because CP in fact does have feasible interpolation [CK01, p.362], which means that we cannot rule out its automatizability. In addition, a non-automatizability result for CP, if possible, would have to be proved using a different strategy.

3.3 Positive Results For $f(s,n)$-Automatizability

The best algorithms which automatize TRES and RES do not run in polynomial time [BKPS02]. Therefore, these algorithms cannot technically be said to automatize TRES or RES. Given the negative results above (notably [AR02]) it seems unlikely that polytime automatizability algorithms will be found. It is therefore reasonable to have a more general definition of automatizability to distinguish between the (inevitably weak) powers of different proof search algorithms.

Definition 9. *A propositional proof system S is $f(n,s)$-**automatizable** if there exists a deterministic algorithm A such that for every unsatisfiable CNF formula F, A returns an S-proof of F in time bounded by the function $f(n,s)$ where n is the size of F and s is the length of the shortest S-proof of F. If A is as above except that it returns a proof of F that is not an S-proof, then S is said to be **weakly** $f(n,s)$-**automatizable**.*

The traditional notion of automatizability therefore corresponds to $(s \cdot n)^k$-automatizability for k a constant. We will briefly discuss the algorithms presented in [BKPS02] which $O(n^{\log s})$-automatize TRES and $O(n^{\sqrt{n \log s}})$-automatize RES. These algorithms feature prominently in the comparison between learnability and automatizability [ABF+04]. The high-level ideas of these algorithms are given below.

3.3.1 TRES Algorithm

Theorem 10. [BKPS02] TRES *is $O(n^{\log s})$-automatizable.*

Given an unsatisfiable CNF formula F, the TRES algorithm works recursively to produce a refutation of F that is not too large compared to the smallest TRES refutation of F. The TRES algorithm begins at level one of its recursion by building $2n$ decision trees in parallel, one for each literal of F. We know that if a decision tree of size s exists for F, then for some literal l_1 there must be a decision tree of size less than or equal to $s/2$ for $F \restriction_{l_1}$. Since we do not know which literal l_1 this is true for, we try them all in parallel. Then at the next level of recursion we know that if there is some decision tree for $F \restriction_{l_1}$ of size $s/2$, then there must be some literal l_2 such that there is decision tree of size less than or equal to $s/4$ for $F \restriction_{l_1,l_2}$. So we continue in parallel and each branch from level one is extended in all $2n-1$ ways etc. Since we are trying all possible branches in parallel we are guaranteed to have some branch which includes l_1 and l_2. The recursion bottoms out at depth i when a branch finally falsifies F (or potentially satisfies F if we do not know that F is unsatisfiable). In this case, the current leafs $n-i$ competing branches are terminated and its sibling tree is computed. Computing the sibling tree has the weakest guarantees. Just because we know that we are guaranteed to find a tree of size at most $s/2^i$, it does not guarantee that its sibling will also have small size. We may actually find a left subtree that is smaller than the optimal's left subtree, but when we branch in the other direction on the literal that led the small left subtree, we may be stuck computing a right subtree that is quite a bit larger than the optimal's right subtree. All that we know down this second branch is that we must recompute the entire problem, with one variable fewer. An analysis along these lines yields Theorem 10, above.

3.3.2 RES Algorithm

Theorem 11. [BKPS02] RES *is* $O(n^{\sqrt{n \log s}})$*-automatizable.*

Given an unsatisfiable CNF formula F and some width w (for which a RES proof for any formula of that width can be computed in an acceptable amount of time), the RES algorithm begins at level one of its recursion by building $2n$ decision trees in parallel, just like the algorithm for decision trees. Unlike the decision tree algorithm, the recursion bottoms out when the residual formula at a node in one of the trees at depth i has width less than or equal to w. When this occurs, the algorithm continues just like the decision tree algorithm except that continues to terminate branches using the width condition. In the end, a RES proof of F is generated that looks tree-like near to where the empty clause is derived and looks like a series of trivial width w brute-force RES proofs at the tree-like proof's leaves. There must always be some literal that is in the average number of clauses of width greater than w and since we branch in all possible ways until some leaf is reached, in the worst case we will at least branch on such a literal at each level of the tree. Like in the decision tree algorithm, the problematic side is calculating the sibling of the shortest tree at each level. The problem is that just because one literal occurs often enough in $F \restriction_\alpha$, it does not mean that its negation does. So instead of removing a large number of wide clauses, we simply reduce their widths by one. An analysis along these lines yields Theorem 11, above.

4 Circuit Classes

Although the literature is replete with references to the correspondence of proof systems and circuit classes, this notion is not particularly clear. For example TRES is associated with circuits that form decision trees, since every TRES proof is a decision tree. On the other hand, $\mathsf{AC}^0-\mathsf{Frege}$ is associated with the circuit class AC^0 not because the underlying structure of every $\mathsf{AC}^0-\mathsf{Frege}$ proof forms a circuit from this class (in fact they form DAGs which can have non-constant depth), but rather because every line of an $\mathsf{AC}^0-\mathsf{Frege}$ proof is a formula that has an AC^0 circuit representing it.

In other words, the literature contains different analogies: one is that the structure of the proofs indicate what circuit class the relevant proof system is associated with, and the other is that the lines of the proof indicate what circuit class it is associated with. Neither one of these analogies is entirely acceptable, since each one works with some proof systems, but not with others.

Proof System	Associated Circuit Class	Analogy Being Used
$TreeResolution$	Decision Trees	Structural
$RegularResolution$	Read-once Branching Programs	Structural
$Resolution$	CNFs	Search
$CuttingPlanes$	Intersections of Halfspaces	Search
AC^0Frege	AC^0	Line
$Frege$	NC^1	Line

Table 1: Correspondence between proof systems and circuit classes.

In fact, the analogies break down even further; for example, RES is associated with the class of circuits for CNF formulas, but RES proofs are not structured like CNF formulas, so the first analogy does not work. Similarly, every line of a RES proof is a clause, not a CNF formula, so the second analogy also does not work. CP is similar, since it is associated with circuits that form intersections of halfspaces, but neither are its proofs structured like intersections of halfspaces, nor is every line of a CP proof a set of linear inequalities; rather, each line is a single linear inequality. In effect, we need a third analogy: these proof systems are associated with the circuit classes which represent the set of inputs to the proof or alternatively to the output of the corresponding search problem.

We shall call the first analogy the 'structural analogy', the second analogy the 'line analogy', and the third one the 'search analogy'. Table 1 lists the proof systems that are being discussed along with their associated complexity classes, and which analogy is typically used to connect the two.

5 Learning

5.1 PAC Learning

In the PAC learning model, the learner's goal is to infer an unknown target concept c from some known concept class $C = \cup_{n=1}^{\infty} C_n$, where each set C_n is a subset of all boolean functions over $\{0,1\}^n$. The learner is provided with two constants $0 \leq \delta, \epsilon \leq 1$ and has access to a set S of examples drawn randomly according to a distribution $\mathcal{D}$, which are each labeled according to c. It is the learner's goal to output with probability at least $1-\delta$ a hypothesis h from a class of functions $\mathcal{H}$ which has probability at most ϵ of disagreeing with c on an example x randomly drawn from $\mathcal{D}$, that is $Pr(h(x) \neq c(x)) < \epsilon$. δ is typically referred to as the confidence parameter and ϵ is typically refered to as the error parameter. Intuitively, we can have confidence δ that the learner will make less than an ϵ-proportion of errors once trained. The formal definition of PAC learning follows:

Definition 12. [KV94, p.10] *We say that a concept class $\mathcal{C}$ is **PAC-learnable** if there exists an algorithm L with the following property: for every concept $c \in \mathcal{C}$, for every distribution $\mathcal{D}$, and for all $0 < \epsilon < 1/2$ and $0 < \delta < 1/2$, if L is given access to an oracle $EX(c, \mathcal{D})$ and inputs ϵ and δ, then with probability at least $1-\delta$, L outputs a hypothesis concept $h \in \mathcal{H}$ such that the probability that $c(x) \neq h(x) \leq \epsilon$. If the above conditions hold and $\mathcal{H} = \mathcal{C}$ then we say that $\mathcal{C}$ is **proper PAC-learnable**.*

We consider a learning algorithm efficient if the number of examples drawn and the overall runtime is polynomial in $1/\delta$, $1/\epsilon$, and n. Analogous to the concept of $f(s,n)$-automatizability we can define the notions of $f(n, 1/\delta, 1/\epsilon)$**-proper PAC-learnable** and $f(n, 1/\delta, 1/\epsilon)$**-PAC-learnable** which take into

account a (not necessarily polynomial) time bound of $f(n, 1/\delta, 1/\epsilon)$ on the complexity of the learning algorithm L.

5.2 Membership Queries

The PAC model can be augmented with *membership queries*. A membership query posed by a learning algorithm is a point $x \in \{0,1\}^n$. The query is posed to an oracle for c which responds either positively or negatively depending on whether $c(x) = 1$ or $c(x) = 0$. [KP98]

5.3 Preliminary Definitions For Hardness Results

Some of the hardness results presented below require the following definitions.

Definition 13. *A **Probabilistic Turing Machine** (PTM) is a non-deterministic Turing machine which randomly chooses between the available transitions at each point during its computation with equal probability.*

Definition 14. $\mathsf{RP} = \{\ L \mid L$ *is a language such that there exists a PTM* M *which accepts any* $w \in L$ *with probability* $\geq \frac{1}{2}$ *and rejects any* $w \notin L$ *with probability* $1\ \}$

5.4 Individual Halfspaces

Definition 15. *A **Linear Threshold Function** (LTF) is a function of the form* $h(x_1 \cdots x_n) = sign(\sum_{i=1}^{n} \alpha_i x_i - \theta)$. *Each LTF induces a **Halfspace** and the terms can be used interchangeably.*

Learning a single halfspace was one of the earliest problems in machine learning theory and has been widely addressed. They are of interest for numerous reasons [KP98]. Learning individual halfspaces are polytime learnable in most models of learnability including the PAC model. Conceptually the easiest PAC algorithm for learning a halfspace h from a set of labeled examples S involves converting the labeled examples in linear inequalities and then solving for the unknown coefficients of h. Many of the following algorithms for learning more complicated concept classes use learning individual halfspaces as a basic tool.

5.5 Polynomial Threshold Functions

Definition 16. *A **Polynomial Threshold Function** (PTF) is a function* $f : \{0,1\}^n \longrightarrow \{+1, -1\}$ *such that* $f(x_1 \cdots x_n) = sign(p(x_1 \cdots x_n))$ *where* $p(x_1 \cdots x_n)$ *is a real-valued multivariate polynomial. The **degree** of a polynomial threshold function is the degree of the polynomial* p.

In [KS01], Klivans and Servedio show how learning any polynomial threshold function of degree d over n variables can be reduced to learning a single halfspace over n^d variables. Their observation is quite simple. Let p be a polynomial over n variables and let t be any term of p with degree $1 < d' \leq d$. Clearly any x_i can only take values in $\{0,1\}$, so $x_i^{d'} = x_i$. This means that the degrees of the individual variables in t are irrelevant to its value. So p is really equivalent to a multivariate monomial where each term can be the product of at most d variables. This means that any polynomial threshold function of degree d over n variables must choose its terms (not including the coefficients) from a pool of at most $\sum_{i=1}^{d} \binom{n}{i} \leq n^d$ choices. We can therefore replace each term with one of n^d correspondingly set variables to form a linear threshold function that is equivalent to p. We can then use the polytime algorithm for learning individual halfspaces described in Section 5.4 to learn the coefficients of p in time $n^{O(d)}$. Since we will output a hypothesis which is a degree d polynomial, this algorithm properly PAC learns.

5.6 Intersections of Halfspaces

Definition 17. *Let $H = \{h_1(x), h_2(x), \ldots, h_s(x)\}$ be a set of linear threshold functions where $x \in \{0,1\}^n$. The intersection f of the halfspaces in H is the function $f(x) = h_1(x) \wedge h_2(x) \wedge \cdots \wedge h_s(x)$.*

5.6.1 Hardness Results

A number of hardness results are known with respect to learning intersections of halfspaces. The problem is known to be NP-complete, even when restricted to proper learning the intersection of two halfspaces that are consistent a sample of points in $\{0,1\}^n$ [BR89]. More recently the following result was proved.

Theorem 18. [ABF+04] *If $\mathsf{NP} \neq \mathsf{RP}$ then there is no algorithm A such that for every c in the concept class of intersections of two halfspaces and for every distribution $\mathcal{D}$, A runs in polytime in n, $|c|$, and $1/\epsilon$ and with probability $2/3$ outputs a hypothesis h from the class of intersections of any constant number of halfspaces such that $error_{\mathcal{D}}(h,c) \leq \epsilon$.*

This expands on the previous result because it shows that it is unlikely that pairs of halfspaces are PAC learnable, even when non-proper learning is permitted.

Numerous negative results are known with respect to learning intersections of a larger number of halfspaces, even if the learning algorithm does not have to be proper. One notable hardness result comes from Long and Warmuth in [LW94], who show that learning convex polyhedra in the continuous domain is as hard as learning poly-sized circuits. Blum and Rivest also show that if a polytime algorithm exists to learn n^{ε} halfspaces with hypotheses taken from the class of all sets of $n^{1-\varepsilon}$ halfspaces, then $\mathsf{NP} = \mathsf{RP}$.

5.6.2 Upper Bounds

Proper PAC Learning Algorithm with Membership Queries [KP98]

In [KP98], Kwek and Pitt introduce an algorithm, which they name *POLLY*, for learning s intersections of halfspaces in the PAC model augmented with membership queries. They prove the following guarantee on the performance of *POLLY*.

Theorem 19. *Let the target concept, c, be an intersection of s halfspaces in $[0,1]^n$, defining a polytope P with volume V. Let $\epsilon, \delta > 0$ and let S be an initially obtained random sample of*

$$m = max\left\{ \frac{4}{\epsilon}\log\frac{4}{\delta}\ ,\ \frac{16(n+1)s\log(3s)}{\epsilon}\log\frac{26}{\delta} \right\}$$

examples. Let $LP(m,n)$ be the time needed to solve a linear programming problem with m constraints and n unknowns. The in time (brace yourself) $\tilde{O}((\frac{1}{V}\ln\frac{1}{\delta} + n\log\frac{1}{d^-})(m\log\frac{ms}{V} + mn\log\frac{n}{d^-} + \frac{m}{V}\ln\frac{1}{\delta} + sLP(m,n)))$ (where d^- is the minimum distance between and of the halfspaces in c and any negative example in S) and with sample complexity m (excluding membership queries), the probability that POLLY fails to output a collection of s halfspaces with error at most ϵ is at most δ.

POLLY is defined as a consistency model algorithm augmented with membership queries. Since Occam's Razor holds even when membership queries are admitted, *POLLY* is easily shown to be PAC compliant when run on a large enough sample size. *POLLY* is learning in the consistency model and so it therefore must separate all of the m positive examples from the negative examples using exactly s halfspaces, where $m >> s$. *POLLY* does so by reducing the problem of finding an intersection of s halfspaces to s instances of finding individual halfspaces, which is easily solved using linear programming (see Section 5.4).

Each positive example must be included by all s halfspaces in c, whereas each negative example must have some halfspace in c which excludes it. Let N be the set of negative examples. If *POLLY*

could output a hypothesis consisting of as many as $|N|$ halfspaces, *POLLY* could just solve $|N|$ linear programming instances, each finding a halfspace that separates a single negative example from all of the positive examples. But since $|N|$ could be as large as m and $m >> s$, *POLLY* must match numerous negative examples to the halfspaces which it finds using linear programming.

Let $a_i \rightarrow p$ denote a line segment from negative example a_i to a positive example p. Ideally, *POLLY* tries to match numerous negative examples to a single halfspace f by seeing if the line segment $a_i \rightarrow p$ intersects the portion of f which forms one of the faces of the polytope P. If the negative examples can all be matched to unique faces (and therefore halfspaces) in this way, then s individual linear programs can be solved, one for each face of P which results in a half space that separates all of the positive examples from the negative examples associated with the face. If two negative examples, a_i and a_j are very close to P, then it can be determined that they are not on the same side of a face by checking to see if the midpoint between a_i and a_j is in P. *POLLY* therefore does a binary search using membership queries at points $a_i^{'}$ and $a_j^{'}$ along $a_i \rightarrow p$ and $a_j \rightarrow p$, and whenever both $a_i^{'}$ and $a_j^{'}$ are outside of P, it checks the midpoint using another membership query. The full details are available in [KP98].

PAC Learning Algorithm for the Uniform Distribution [KOS04]

In [KOS04], Klivans, O'Donnell, and Servedio introduce two algorithms for learning a wide range of functions of halfspaces. They prove the following result.

Theorem 20. [KOS04] *Let $g : \{+1,-1\}^k \longrightarrow \{+1,-1\}$ be any Boolean function on k bits and let $h_1, \ldots, h_k$ be arbitrary halfspaces on $\{+1,-1\}^n$. The class of functions $\{g(h_1(x), \ldots, h_k(x))\}$ can be learned under the uniform distribution to accuracy ϵ in time $n^{O(k^2/\epsilon^2)}$, assuming $\epsilon < 1/k^2$.*

If g is restricted to an intersection of **read-once** halfspaces, (i.e. each variable x_i appears as input to at most one halfspace) then the bound is improved to the following:

Theorem 21. [KOS04] *Let $h_1, \ldots, h_k$ be arbitrary halfspaces on $\{+1,-1\}^n$ which depend on disjoint sets of variables. The class of functions $\{h_1(x) \wedge \cdots \wedge h_k(x)\}$ of read-once intersections of k halfspaces can be learned under the uniform distribution to accuracy ϵ in time $n^{\tilde{O}((\log k/\epsilon)^2)}$, assuming $\epsilon < 1/\log k$.*

Klivans, O'Donnell, and Servedio really prove that these classes of functions have low Fourier concentration bounds which allows them to apply the sampling algorithm of Linial, Mansour, and Nisan [LMN93] which learns a class C over the uniform distribution in time $n^{O(deg(n,\varepsilon))}$ where $deg(n,\varepsilon)$ is the Fourier concentration bound of C. That is, for every $c \in C_n$, all but an ϵ fraction of the Fourier Spectrum of c is concentrated on degree up to $deg(n,\varepsilon)$. The algorithm works by deriving good approximations for the Fourier coefficients of c (good because the Fourier concentration bound of c low) and then uses the approximate coefficients to predict the value of c at future points. In [LMN93] there is also an analysis that shows that any function expressible by AC^0 circuits on n inputs has a concentration bound of $deg(n,\varepsilon) = O(polylog(n/\varepsilon))$, which means that they are learnable in time $n^{polylog(n/\varepsilon)}$. Klivans, O'Donnell, and Servedio improve these bounds for Boolean functions involving halfspaces. This approach does not proper PAC learn and works only for the uniform distribution.

PAC Learning Algorithm for an Arbitrary Distribution [KOS04]

Klivans, O'Donnell, and Servedio's second algorithm learns intersections of k halfspaces under an arbitrary Distribution in the PAC model. Again, a new algorithm is not given, rather they show that various functions of halfspaces, including the intersection, can be represented by polynomial threshold functions of low degree. This allows the learning of these functions of halfspaces using the algorithm described in [KS01] which learns the class of degree d polynomial threshold functions in time $n^{O(d)}$ (see Section 5.5). The following results are given.

Theorem 22. [KOS04] *Let $h_1, \ldots, h_k$ be arbitrary weight w halfspaces on $\{+1, -1\}^n$. The class of functions $\{h_1(x) \wedge \cdots \wedge h_k(x)\}$ of intersections of k weight w halfspaces can be learned under any distribution to accuracy ϵ in time $n^{k \log k \log w}/\epsilon$.*

When k is large the previous bound is superpolynomial. Klivans, O'Donnell, and Servedio therefore show a trade-off in the analysis between the number of halfspaces k and the weight of the halfspaces w, which allows them to give the following bound which is better when k is large.

Theorem 23. [KOS04] *Let $h_1, \ldots, h_k$ be arbitrary weight w halfspaces on $\{+1, -1\}^n$. The class of functions $\{h_1(x) \wedge \cdots \wedge h_k(x)\}$ of intersections of k weight w halfspaces can be learned under any distribution to accuracy ϵ in time $n^{\sqrt{w} \log k}/\epsilon$.*

PAC Learning Algorithm Using A Margin [KS04]

In [KS04], Klivans and Servedio give an algorithm for intersections of halfspaces under the assumption that no example lies too close to any separating hyperplane. Given m examples which are known to each have minimum distance ρ to any halfspace of the target concept c, their algorithm proceeds as follows. It first projects the m points in $\mathbf{R}^n$ into $\mathbf{R}^{O(\log m/\epsilon^2)}$, which keeps the distances between any two of the m points within a $1 \pm \epsilon$ factor. This allows them to learn in a low dimension space compared to $\mathbf{R}^n$ and still be guaranteed a margin of $\rho/2$. From this point forward they concentrate on learning a low degree polynomial threshold function for c as in [KOS04]. They then use the Kernel Perceptron Algorithm to learn c. This is done similarly to Section 5.5, by considering the polynomial threshold function representing c as a linear threshold function over n^d variables. Knowing the margin of c allows one to bound the number of mistakes which the Kernel Perceptron Algorithm will make when learning the linear threshold function. The following theorems are proved using different constructions for the polynomial threshold functions representing c:

Theorem 24. [KOS04] *There is an algorithm which PAC learns any ρ-margin intersection of t halfspaces over $\boldsymbol{R}^n$ in at most $\frac{n}{\epsilon}\left(\frac{t}{\rho}\log\frac{1}{\delta\epsilon}\right)^{O(t \log t \log(1/\rho))}$ time steps.*

Theorem 25. [KOS04] *There is an algorithm which PAC learns any ρ-margin intersection of t halfspaces over $\boldsymbol{R}^n$ in at most $\frac{n}{\epsilon}\left(\frac{t}{\rho}\log\frac{1}{\delta\epsilon}\right)^{O(t^2 \log(1/\rho))}$ time steps.*

When t is constant, the previous two algorithms have quasipolynomial runtime in ρ.

Theorem 26. [KOS04] *There is an algorithm which PAC learns any ρ-margin intersection of t halfspaces over $\boldsymbol{R}^n$ in at most $\frac{n}{\epsilon}\left(\frac{\log t}{\rho}\log\frac{1}{\delta\epsilon}\right)^{O(\sqrt{1/\rho}\log t)}$ time steps.*

When ρ is constant, the preceding algorithm has quasipolynomial runtime in t.

5.7 DNFs and Decision Trees

In [ABF+04], the authors observed that the $f(n, s)$-automatizability algorithms (and accompanying upper bounds) for RES and TRES discussed in Sections 3.3.2 & 3.3.1 are also PAC learning algorithms for learning, respectively, the classes of DNF formulas and decision trees. When considered as an algorithm which learns decision trees from a set of labeled examples the algorithm discussed in Section 3.3.1 yields the following guarantees:

Theorem 27. [BKPS02] *Decision trees are $O(n^{\log s})$-proper PAC-learnable.*

Similarly, the algorithm discussed in Section 3.3.2 for automatizing RES can be viewed as a learning algorithm for DNF formulas.

Theorem 28. [BKPS02] *DNFs are $O(n^{\sqrt{n \log s}})$-proper PAC-learnable.*

5.7.1 Hardness Results

Many of the techniques used in proving hardness results for learning DNFs and decision trees are inspired by techniques in [AR02] used to show that automatizing Resolution is difficult [ABF+04]. This mirrors, in a somewhat weaker way, the link between automatizability and learnability displayed by the positive results. The following hardness results are the best known with respect to learning decision trees:

Theorem 29. [HJLT95] *If* $\mathsf{NP} \not\subseteq \mathsf{RTIME}(2^{\log^{O(1)} n})$*, then decision trees of size* s *over* n *variables are not learnable by size* $s \cdot 2^{\log^{1-\varepsilon} n}$ *decision trees in polytime, for some* $\varepsilon < 1$*.*

Theorem 30. [ABF+04] *If* $\mathsf{NP} \not\subseteq \mathsf{DTIME}(2^{n^{\varepsilon}})$*, then decision trees are not properly PAC learnable in polytime, for some* $\varepsilon < 1$*.*

Theorem 31. [ABF+04] *If* $\mathsf{NP} \neq \mathsf{RP}$*, then 2-term DNF formulas are not PAC learnable by the hypothesis class of* k*-term DNF formulas for any constant* $k > 0$*.*

Theorem 32. [ABF+04] *If* $\mathsf{NP} \neq \mathsf{RP}$ *then there is no algorithm* A *such that for every* c *in the concept class of DNF formulas and for every distribution* $\mathcal{D}$*,* A *runs in polytime in* n*,* $|c|$*, and* $1/\epsilon$ *and with probability* $2/3$ *outputs a hypothesis* h *from the class of OR-of-threshold-formulas such that* $error_{\mathcal{D}}(h, c) \leq \epsilon$*.*

5.7.2 Upper Bounds

The RES automatizability algorithm is not the best known algorithm for learning DNF formulas. In [KS01], Klivans and Servedio give an algorithm which learns DNF formulas in time $2^{\tilde{O}(n^{1/3})}$. Like their algorithm for learning the intersection of halfspaces under any arbitrary distribution (see Section 5.6.2), this algorithm reduces the problem of learning a DNF formula to learning a low degree polynomial threshold function. Their main results are:

Theorem 33. [KS01] *Any* s*-term DNF over* $\{0,1\}^n$ *in which each term is of size at most* t *can be expressed as a polynomial threshold function of degree* $O(\sqrt{t} \log s)$*.*

Theorem 34. [KS01] *Any* s*-term DNF over* $\{0,1\}^n$ *can be expressed as a polynomial threshold function of degree* $O(n^{1/3} \log s)$*.*

5.8 Discussion Relating Learning to Automatizability

Some successful initial connections between automatizability and learnability were made in [ABF+04].

1. Both automatizability and learnability are concerned with search problems. Automatizability concerns itself with finding refutations within a given proof system, while learnability concerns itself with finding a concept from a given concept class.

2. When learnability seems difficult and only super-polynomial upper bounds are known, the concept classes under consideration correspond to objects which have prominent roles in specific proof system. For example decision trees correspond to TRES and intersections of halfspaces correspond to CP.

3. There are specific algorithms known which have the same guarantees with respect to automatizing a proof system and to learning a circuit from a circuit class which closely corresponds to the proof system (RES and DNF formulas, TRES and decision trees).

4. Some ideas used in hardness results for automatizability of proof systems have been used successfully to prove hardness results for learning the corresponding circuit class.

Despite these apparent similarities numerous difficulties exist in the comparison. The two algorithms, which $O(n^{logs})$-automatize TRES and $O(n^{\sqrt{n}\log s})$-automatize RES and learn the correspond circuit classes, rely on different analogies to make the connection between learnability and proof complexity. In the case of learning decision trees the analogy is structural, meaning that the object being learned has the same form as the underlying structure of the proof, whereas in the case of learning DNF formulas the object being learned corresponds to the result of the search problem being computed by a RES proof. In fact, this analogy is even weaker because RES proofs actually find groups of clauses (CNF formulas). It is not clear how to make these differing interpretations agree. The lack of a uniform treatment suggests that the connections between the algorithms may be coincidental (though it may just mean that the connection is of a deeper nature). Furthermore, more efficient learning algorithms than the RES algorithm for learning the class of DNF formulas are known [KS01] and it is not clear how they yield search algorithms for RES.

Relating the automatizability of PC to known learning algorithms for objects which seem related to PC proofs is also challenging. The one notable candidate for a connection is Klivans and Servedio's learning algorithm for polynomial threshold functions. But it is unclear whether the connections are anything more than superficial. There may be a deeper connection between the Groebner Basis algorithm used to $n^{O(d)}$-automatize PC and the $n^{O(d)}$-learning algorithm of [KS01], which reduces a polynomial threshold functions to linear programming, but the connections (other than the similarity of the upperbounds) are not readily apparent.

Finally, we were most interested in looking at algorithms which learn intersections of halfspaces in order to see if they could give any insight into the automatizability of CP or even TCP. As mentioned earlier, CP has feasible interpolation and is therefore still a candidate for automatizability. Unfortunately, we were unable to see an immediate connection between any of the learning algorithms studied and proof search for CP.

Though we are expressing some skepticism regarding a connection between automatizability and learnability, there do seem to be some uncanny similarities in the approaches that have worked for both. The hope is still therefore that a general theorem can be proved that would allow automatizability results for proof systems to be translated into learnability results for their corresponding circuit classes and vice versa. It seems clear that the first step should be to understand and present a consistent treatment for what is meant when we say that a proof system P corresponds to a circuit class $C(P)$.

Since this project is our first view into this area, our survey of the relevant literature has not been complete. There is a little more reading which we would like to do before tackling open problems. There are a few algorithms which we did not have the chance to investigate, most notably Haussler and Ehrenfeucht's decision tree learning algorithm which has the same runtime as the automatizability/learning algorithm which was discussed, and Jackson's DNF learning algorithm which makes use of membership queries. Learning algorithms which make use of membership queries may still provide insight into the problem of automatizability since we have unrestricted access to a formula when searching for a refutation of it.

We would also liked to have studied *Lovász, Schrijver Systems* which are automatizable and resemble the algebraic systems CP and PC. There are also other hardness result pertaining to learning DNF formulas which we were not able to study. Supposing a real link between learning DNF formulas and automatizing RES, close investigation into these techniques may yield insight into improving the result of [AR02].

6 Open Problems

In surveying the literature related to automatizability and learning concepts which seem related to proof systems, we identified the following potential avenues of investigation which could further the goal of discovering whether there is a concrete relationship between learning and automatizability.

- ***What does it mean for a circuit class to correspond to a proof system?***

This seems to be the most important question regarding the relationship between learning and automatizability. Ultimately we would like a result which shows something akin to: *You can $f(s,n)$-automatize proof system P if and only if you can $f'(n, 1/\delta, 1/\epsilon)$-learn $C(P)$, where $C(P)$ is the set of circuits corresponding to proofs in P.* In order for such a statement to be made the notion of correspondence must be clarified.

- ***If proof system A p-simulates proof system B, and A is automatizable, then is B automatizable?***

  Though some work has been done on the relationship between the p-simulation and automatizability (notably by Bonet and Atserias), we have not been able to find an answer to the above question. If A p-simulates B, then a proof search algorithm for A must be searching through a richer domain than the set of B proofs. It therefore seems reasonable to hypothesize that proof search algorithms which find A-proofs in time proportional to the shortest A proof might be modified to find B proofs in time that is proportional to the shortest B-proof.

- ***If proof system A p-simulates proof system B, and B is not automatizable, then is A also not automatizable?***

  This is a similar question to the previous one. If it is difficult to find proofs in a smaller domain, then it would seem counterintuitive to think that one would have an easier time finding proofs in a richer domain, especially considering that the proofs in A might be considerably shorter than those in B.

- ***Find other examples of similarities amongst learning algorithms and automatizability algorithms.***

  Though not a very specific goal, finding ways to adapt known automatizability algorithms in order to learn a related class of concepts or finding ways to adapt known learning algorithms in order to automatize proof systems would help guide the search for a general result connecting the two notions. Even this is very difficult, since it is unclear how any of the learning algorithms studied could help automatize CP. It is not even clear how the $2^{n^{1/3}}$ algorithm for learning DNF formulas can be adapted to speed up proof search for RES. This is despite the fact that there have already been links established between learning DNF formulas and automatizing RES.

References

[ABF+04] M. Alekhovich, M. Braverman, V. Feldman, A. Klivas, and T. Pitassi. Learnability and Automatizability. *Proceedings of the 45^{th} Annual IEEE Symposium on Foundations of Computer Science*, 2004.

[AR02] M. Alekhovich and A. Razborov. Resolution is Not Automatizable Unless $\mathcal{W}[\mathcal{P}]$ is Tractable. *Proceedings of the 42^{nd} Annual IEEE Symposium on Foundations of Computer Science*, 2002.

[BKPS02] P. Beame, R. Karp, T. Pitassi, and M. Saks. The Efficiency of Resolution and Davis-Putnam Procedures. *SIAM Journal of Computing*, Vol. 31, No. 4, 2002.

[BP01] P. Beame and T. Pitassi. Propositional proof complexity: Past, present, and future. In G. Paun, G. Rozenberg, and A. Salomaa, editors, *Current Trends in Theoretical Computer Science: Entering the 21st Century*, pages 42 – 70. World Scientific Publishing, 2001.

[BPR97] M. Bonet, T. Pitassi, and R. Raz. Lower Bounds for Cutting Planes Proofs with Small Coefficients. *Journal of Symbolic Logic*, Vol. 62, No. 3, 1997.

[BPR00] M. Bonet, T. Pitassi, and R. Raz. On Interpolation and Automatization For Frege Systems. *SIAM Journal of Computing*, Vol. 29, No. 6, 2000.

[BR89] A. Blum and R. Rivest. Training a 3-node neural net is NP-complete. In D. S. Touretzky, editor, *Advances in Neural Network Information Processing Systems I*, pages 494 – 501. Morgan Kaufman, 1989.

[CEI96] M. Clegg, J. Edmons, and R. Impagliazzo. Using the Groebner Basis Algorithm to Find Proofs of Unsatisfiability. *Proceedings of the* 28^{th} *Annual ACM Symposium on Theory of Computing*, pages 174 – 183, 1996.

[CK01] P. Clote and E. Kranakis. *Boolean Functions and Computation Models.* Springer-Verlag, Berlin, 2001.

[CR74] S.A. Cook and R. A. Reckhow. On the Lengths of Proofs in the Propositional Calculus. *Proceedings of the Sixth Annual ACM Symposium on the Theory of Computing*, pages 135 – 148, 1974.

[CR79] S.A. Cook and R. A. Reckhow. The Relative Efficiency of Propositional Proof Systems. *The Journal of Symbolic Logic*, 44:36 – 50, 1979.

[Hak85] A. Haken. The Intractability of Resolution. *Theoretical Computer Science*, 39:297 – 308, 1985.

[HJLT95] T. Hancock, T. Jiang, M. Li, and J. Tromp. Lower Bounds On Learning Decision Lists and Trees. *Proceedings of the* 12^{th} *Annual Symposium on Theoretical Aspects of Computer Science*, pages 527 – 538, 1995.

[KOS04] A. Klivans, R. O'Donell, and R. Servedio. Learning Intersections and Thresholds of Halfspaces. *Joural of Computer and System Sciences*, pages 808 – 840, 2004.

[KP98] S. Kwek and L. Pitt. PAC Learning Intersections of Halfspaces With Membership Queries. *Algorithmica*, Vol. 22, Issue 1:53 – 75, 1998.

[KS01] A. Klivans and R. Servedio. Learning DNF in time $2^{\tilde{O}(n^{1/3})}$. *Proceedings of the* 33^{rd} *Annual ACM Symposium on Theory of Computing*, pages 258 – 265, 2001.

[KS04] A. Klivans and R. Servedio. Learning Intersections of Halfspaces with a Margin. *Computational Learning Theory*, pages 348 – 362, 2004.

[KV94] M. Kearns and U. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, Cambridge, Massachusetts, 1994.

[LMN93] N. Linial, Y. Mansour, and N. Nisan. Constant Depth Circuits, Fourier Transform, and Learnability. *Journal of the Association for Computing Machinery*, Vol. 40, Issue 3:607 – 620, 1993.

[LW94] P. Long and M. Warmuth. Composite Geometric Concepts and Polynomial Predictability. *Information and Computation*, Vol. 113, Issue 2:230 – 252, 1994.

[Rec76] R. A. Reckhow. *On the Lengths of Proofs in the Propositional Calculus.* PhD thesis, University of Toronto, 1976.