Clause Learning Can Effectively P-Simulate General Propositional Resolution

Tracking number: ????

Abstract

Currently, the most effective complete SAT solvers are based on the DPLL algorithm augmented by clause learning. These solvers can handle many real-world problems from application areas like verification, diagnosis, planning, and design. Without clause learning, however, DPLL loses most of its effectiveness on real world problems. Recently there has been some work on obtaining a deeper understanding of the technique of clause learning. In this paper we introduce the idea of effective p-simulation, which is a new way of comparing clause learning with general resolution and other proof systems. We then show that pool proofs, a previously used characterization of clause learning, can effectively p-simulate general resolution. Furthermore, this result holds even for the more restrictive class of *greedy, unit propagating*, pool proofs, which more accurately characterize clause learning as it is used in practice. This result is significantly stronger than previous results, indicating that clause learning is more powerful than was previously known. Finally, we show that this power also has some negative implications for the prospects of efficient proof search.

Introduction

DPLL (Davis, Logemann, & Loveland 1962) is a depth-first backtracking search procedure for solving SAT developed in 1960's. However, it is only recently that SAT solving using DPLL has been effective enough for practical application. This has mainly been due to algorithmic improvements to the basic procedure. The most important improvement, and the one that has revolutionized DPLL SAT solvers, has been the technique of clause learning. Clause learning involves learning information (new clauses) when paths in the search tree lead to failure, akin to explanation based learning (Stallman & Sussman 1977). These clauses are then utilized to reject, or refute, nodes in the future search thus avoiding having to search the subtree below these nodes. This yields considerable reductions in the size of the search tree.

How much can the search tree and the run time of DPLL be reduced by clause learning? Can this reduction be quantified? Are there "easy" inputs in which clause learning remains ineffective? These are the kinds of questions that arise as we try to achieve a better understanding of the power and limitations of clause learning. In this paper we study the formal properties of clause learning, addressing questions like those just posed. We provide a number of new insights into this important technique extending previous work in this direction (Beame, Kautz, & Sabharwal 2004; Van Gelder 2005).

In particular, we argue that previous formalizations of clause learning have some critical differences from practical implementations of clause learning algorithms which make them unrealistically powerful. Thus, the formal results obtained from these characterizations are not as informative about practice as one would like. We identify some very natural constraints to a previous formalization that allow us to more accurately capture clause learning as it is implemented in practice.

Our main result shows that clause learning is more powerful than it was previously known to be. Surprisingly, this even holds for our natural, constrained form. We show that it is essentially as powerful as the full resolution proof system, in a sense made precise in the paper. Since our constraints were specifically introduced to make our system more similar to practical clause learning algorithms, we argue that our result is highly relevant to practice.

We temper this good news about clause learning with a negative result which shows that for every deterministic implementation of a clause learning algorithm A, there will always be formulas which some other implementation of clause learning could solve quickly, but A cannot, unless a widely-believed complexity assumption is false. This shows that clause learning's power is a double-edged sword.

As with previous work we analyze clause learning by formalizing it as a proof system. We therefore first present the formalism of proof systems in Section , and explain how proof systems are compared. Then in Section we explain our model of clause learning and how it generates proofs. This results in a characterization of clause learning as a proof system, and thus allows us to compare it with resolution. In Section we show that our formalism of clause learning is essentially as powerful as resolution. Next, in Section we show how this power leads to our negative result, namely that no deterministic implementation of clause learning can ever realize the power of our formalizations, despite its natural and constrained nature. It is important to note that since our system is a constrained version of previous clause learning formalizations, both our positive and negative results hold for them also. Finally, in Section we give some concluding remarks and present some related open problems.

Proof Systems and Resolution

For this paper, we are concerned with *resolution* proofs about propositional formulas, primarily proofs of *unsatisfiability*, called **refutations**. A proof system S is a language for expressing proofs, such that a proposed S-proof π, based on a formula F, can be verified (or found to be faulty) in time that is polynomial in the lengths of π and F (Cook & Reckhow 1979). Proofs may be exponentially longer than the underlying formulas. Although the original definition casts proofs as strings, the system which we are concerned with in this paper (resolution) has proofs which can intuitively be viewed as *directed acyclic graphs* (DAGs).

Definition 1 lays the groundwork for comparing the power of different proof systems.

Definition 1 *The **complexity** of an unsatisfiable formula F in proof system S is the size of the shortest proof of F in S. We denote this by $|S(F)|$. Size is usually measured by the number of applications of the rules of inference (for our purposes, resolution steps). For proof systems S and T we say S **p-simulates** T if, for all formulas F, the shortest S-proof for any formula F is at most polynomially longer than the shortest T proof of F.*

The power of a proof system S generally relates to the space of formulas for which short S proofs exist. To this effect, if proof system S p-simulates proof system T, then S is viewed to be at least as powerful as T.

Resolution (res) is a proof system that permits unrestricted use of the resolution rule, its only rule of inference. All the proof systems discussed in this paper are **refinements** of general resolution; that is, they also use only the resolution rule, but have various constraints on its application, hoping to achieve greater efficiency in practice. Clearly, res p-simulates all of its various refinements.

Resolution operates on formulas in **conjunctive normal form** (CNF), i.e., a conjunction of a set of **clauses**. Each clause is a disjunction of a set of **literals**.

Definition 2 *We define the **resolution rule** as a function which takes as input two clauses A and B and a variable x, and produces a single clause.*

- *If $x \in A$ and $\neg x \in B$, then $Res(A, B, x) = A' \vee B'$ where A' and B' are exactly A and B respectively, except that any mention of x has been removed from them.*
- *If, on the other hand, $x \notin A$ and $x \in B$, then $Res(A, B, x) = A$. We call this the degenerate resolution rule.*
- *Finally, if $x \notin A$ and $x \notin B$, then $Res(A, B, x)$ can be either clause, but for the convenience of making the rule deterministic, we will always pick the shorter clause.*

*We refer to either of the last two cases as the **degenerate resolution rule**.*

Our definition of the resolution rule is slightly different than the standard definition because we allow degenerate resolutions. It is important to note that the degenerate resolution rule adds no power to res. It is easily shown that every minimally sized res proof will contain no degenerate resolutions (Van Gelder 2005), even if it is permitted to use them.

We define a res-proof based on some formula F to be a rooted directed acyclic graph (DAG), where the root is the unique source. Each node v of the DAG is labeled by a clause $clause(v)$. Internal nodes (with out-degree two) are also labeled with a **clashing literal**, denoted by $clash(v)$. Nodes with out-degree zero (sink nodes, leaves) are labeled with a clause of F, called an **initial clause**, and have no clashing literal. The clause label of an internal node is the (unique) clause obtained by appling the resolution rule on the clause-labels of its two children, on its clashing literal. The two edges leaving v, are labeled with $clash(v)$ and $\neg clash(v)$ in the way that is consistent with the resolution step.

If $clause(root) = \emptyset$ (the empty clause), then the DAG is a **refutation** of F (a proof of F's unsatisfiability); otherwise it is a proof that F logically implies $clause(root)$. The size of a proof in this format is the number of nodes in the DAG, which corresponds to our notion of complexity from Definition 1. We also require the notion of a **restriction** of a CNF formula.

The set of traces of a decision algorithm's computations on its inputs naturally produces a proof system. The traces of an algorithm $\mathcal{A}$ correspond to the proofs of a proof system S, then we say that $\mathcal{A}$ is an S-based algorithm. The traces of many SAT-solvers naturally correspond to resolution proofs. A lowerbound on the size of resolution proofs for some class of formulas implies a lowerbound on the performance of such resolution-based algorithms on that class. An upperbound on the size of resolution proofs shows that some resolution-based algorithm exists which can solve that class efficiently. The upperbound situation is made more complicated because refuting a formula with a deterministic resolution-based SAT-solver requires that 1) a short resolution proof exists for that formula, and 2) such a short proof can be found quickly. Our two results basically show that clause learning is almost as good as resolution at producing short proofs, but as a consequence, it is just as hard to search for them as it is to search for resolution proofs.

Tree-like Resolution (tree) and Regular Resolution (reg) are two well-known refinements of res. A res proof π is a tree-proof iff the induced subgraph of π in which all initial nodes have been removed is a tree. A path in π is **regular** if every internal node along the path is labeled by a distinct clashing variable; the proof is regular if all its paths are. Irregularities in a tree-proof can be removed without increasing the size of the proof. tree-proofs may therefore always be considered to be a refinement of reg. tree-proofs correspond very naturally to the traces of DPLL algorithms.

A relatively new refinement of res that is useful for analyzing clause learning is **pool resolution** (pool) (Van Gelder 2005). Briefly, a res proof π is a pool resolution proof iff there is some depth-first search of π (starting at the root)

Algorithm 1: A1 (DPLL with Clause Learning). The objective of $\mathsf{A1}(\rho)$ is to return an initial or soundly derived clause that is falsified by the partial assignment ρ.

1 *// The current set of clauses F is a global data structure*;
2 $\mathsf{A1}(\rho)$ **begin**
3   **if** *All clauses of F are satisfied by ρ* **then**
4   **Exit** (ρ is a satisfying assignment);
5   **if** ($F|_\rho$ *contains a falsified clause*) $\wedge$
6   (*GREEDY* $\vee$ **choose**(*Accept*))
7   **then**
8   **choose** *a falsified clause C of $F|_\rho$*;
9   **return** C (*without ρ applied*);
10   **choose** *A literal ℓ such that neither ℓ nor $\neg\ell$ is in ρ*;
11     $D_1 = \mathsf{A1}(\langle\rho, \ell\rangle, F)$;
12     $D_2 = \mathsf{A1}(\langle\rho, \neg\ell\rangle, F)$;
13     $D_3 = Res(D_1, D_2, \ell)$;
14   **if choose** *Learn* **then**
15   Add D_3 to F;
16   **return** *(D_3)*;
17 **end**

such that every path traversed by the depth-first search is a regular path. One can think of pool proofs as being *depth-first regular*.

It is known that reg is exponentially stronger than tree (Buresh-Oppenheim & Pitassi 2003) (i.e. reg p-simulates tree but tree cannot p-simulate reg). We observe that reg is a refinement of pool—every depth-first search of a reg proof is regular, so clearly pool p-simulates reg. However, it is known that reg cannot p-simulate pool (Van Gelder 2005).

It is also known that reg cannot p-simulate res (Alekhnovich *et al.* 2002), but it is unknown whether pool can p-simulate res. Although we have not proved that pool p-simulates res, our main result shows that even a restricted form of pool, which we call CL, is "effectively" as powerful as res in a sense we make precise below.

The Clause Learning Algorithm

Algorithm 1 shows an abstracted version of a clause learning algorithm which we will use to present our CL proof system. It operates with a background set of clauses F initialized to the initial CNF we are trying to solve. As clauses are learned (line 15) F is augmented. The algorithm builds up a partial assignment ρ (a set of literals made true) as it descends through its recursive calls. If ρ satisfies all the clauses of F we have a **solution** and the algorithm can exit (line 4). At various points ρ might falsify some clauses of F. (The terminology can be confusing: this event is called a conflict, but the falsified clause is not a conflict clause.) If the algorithm is operating in **greedy mode**, i.e., the *GREEDY* flag is true, it must accept a falsified clause (line 8) if one is present in the possibly augmented F. In **nongreedy mode**, the algorithm can **choose** to accept or ignore the falsified clause. (It must accept if all variables have been assigned by ρ.)

If neither a solution nor an accepted falsified clause has been found, the algorithm continues to line 10 where a new unassigned literal ℓ is chosen and branched upon, thus building up ρ. The common terminology is that the global **decision level** starts at zero and increases by one when ℓ is *not* in a unit clause; in this case ℓ is called a **decision literal**. The decision level remains unchanged when ℓ *is* in a unit clause. The procedure continues with two recursive calls, one for each setting of ℓ's underlying variable.

Either of these recursions could return a clause not containing the branch variable ℓ, in which case the value of D_3 is governed by the degenerate resolution rule. If the first recursive call (line 11) learned D_1 before returning, and D_1 does *not* contain $\neg\ell$, then the second recursive call (line 12) may immediately return D_1 (from line 9), as well. In this case, $D_3 = D_1$. If D_1 *does* contain $\neg\ell$, but D_2 does *not* contain ℓ, then $D_3 = D_2$. However, it might be important that D_1 was learned in the recursive call, as it can be utilized later.

In the normal case, the two returned clauses clash on ℓ, and a (standard) resolution step derives a clause D_3 not containing ℓ. In some cases this is what is known as a conflict clause in the clause-learning literature, but more often it is intermediate, on the way to deriving a conflict clause. Backtracking then continues. If F is UNSAT, eventually some recursive call will derive the empty clause as a conflict clause, which will cause backtracks that unwind all recursions (line 9 or 16 will constantly be executed until the initial recursive call is terminated).

There are some differences between this abstracted version of clause learning and standard implementations.

- Standard implementations are *greedy* (*GREEDY* is set to true), backtracking as soon as a conflict is detected.
- Standard implementations employ mandatory *unit propagation*. The abstracted version permits this as a policy for **choose** literal (line 10), but does not force it.
- Standard implementations generally employ **far backtracking** after deriving a 1-UIP clause (possibly backtracking multiple decision levels). That is, when a 1-UIP conflict clause D_3 is learned the program backtracks to the decision level closest to the root such that $D_3|_\rho$ (for ρ at that level) is a unit clause. In the abstracted algorithm, a 1-UIP conflict clause causes the search to backtrack just one decision level.

Algorithm 1 represents a class of different algorithms, where each algorithm in this class resolves in some particular way the non-deterministic choices shown above. Included in this class are algorithms that are essentially the same as standard implementations operating without far backtracking. In particular, algorithms operating in greedy mode are among the deterministic realizations of Algorithm 1 as are those that perform unit propagation.

The deterministic realizations of Algorithm 1 do support the derivation of 1-UIP clauses, but they lack far backtracking, which comes into play following such a derivation. Far backtracking is a form of partial restarts, and some search nodes might have only one child, which puts them outside the framework of Algorithm 1. In fact, when a unit 1-UIP clause is learned, far backtracking causes a total restart.

However, our main result on the power of Algorithm 1 indicates that far backtracking does not add any extra theoretical power, even though it may be useful in practice. Incorporating far backtracking into Algorithm 1 would also make its analysis considerably more complex. For these two reasons we will assume in the rest of the paper that far backtracking is not being employed and instead take Algorithm 1 to be a informative abstract model of clause learning.

Proofs Generated by Algorithm 1

As discussed above, to study the properties of Algorithm 1, we view it in terms of the proofs of UNSAT it is capable of generating. If given an UNSAT formula, it can be seen that Algorithm 1 will generate a DAG proof containing resolution steps (added when line 13 is executed). These proofs are a refinement of res proofs that have a special structure.

The results of (Van Gelder 2005) show that the proofs generated by the most general form of Algorithm 1, in which neither mandatory unit propagation nor greedy caching are enabled, are pool proofs. Van Gelder also shows that every pool proof can be generated by the same form of Algorithm 1. Hence, Algorithm 1 exactly characterizes the proof system we call pool—a system in which all proofs are depth-first regular resolution proofs. Intuitively, this characterization arises from the fact that Algorithm 1 explores a DPLL tree and the paths of that tree never branch on the same variable twice. The DPLL tree is embedded in the pool DAG as the regular depth-first traversal, each *cross-edge* of which corresponds to a cache hit of the algorithm.

Most standard implementations of clause learning include mandatory unit propagation and greedy caching. As a consequence, the space of pool DAGs is much richer than the space of clause learning computations. In order to better understand the theoretical power of clause learning, we therefore define a new refinement of pool that we call CL, which is much closer to what is generally meant by clause learning in practice.

Definition 3 *The **Clause Learning** proof system (CL) is the refinement of pool that consists of refutations that can be generated by Algorithm 1 with greedy mode and unit propagation enabled.*

In earlier work, Beame et al. (Beame, Kautz, & Sabharwal 2004) also studied clause learning as a proof system, giving a more complex characterization. They formalized clause learning as a system containing any proof that could be produced by running DPLL with unit propagation and some learning scheme $\mathcal{S}$. The learning scheme $\mathcal{S}$ is able to learn conflict clauses that are cuts in a conflict graph (Marques-Silva & Sakallah 1996). We call this previous idea of clause learning **conflict clause resolution** (CNFL).

The difficulty with CNFL is that it allows very general, even non-deterministic, learning schemes for extracting conflict clauses from the implication graph. For example, the learning scheme $\mathcal{S}$ can extract a conflict clause from a path in a manner that is unrelated to the order in which the variables along the path have been set. Furthermore, it can extract such conflict clauses and backtrack without inserting a literal into the DPLL path. This last point is particularly important since, if no literal is inserted into the DPLL path, DPLL is then free to branch again on that variable: i.e., this introduces a new source of irregularity into CNFL-proofs. The CNFL formalism is therefore very hard to analyze in its full generality. In particular, a proof separating res from CNFL would require one to prove that *for all* learning schemes $\mathcal{S}$, there is no CNFL proof utilizing $\mathcal{S}$ which can match the performance of res on some infinite family of formulas. Nor is it clear that the full generality of CNFL offered by its choice of learning schemes could ever be profitably exploited in any real implementation.

Furthermore, for all of the more specific clause learning schemes mentioned in (Beame, Kautz, & Sabharwal 2004), including first-cut, RelSat (Bayardo & Schrag 1997), 1-UIP (Marques-Silva & Sakallah 1996), and all-decision (Zhang *et al.* 2001), Van Gelder has shown that CNFL using any of these schemes produces proofs that are in pool (Van Gelder 2005). (As mentioned, restarts and far backtracking from 1-UIP clauses do not fall into this framework.) Unfortunately, the form of clause learning characterized by pool operates in a nongreedy mode, giving it much more power than real implementations of clause learning have. CL eliminates much of this extra power, but we are still able to show that it is essential as powerful as res.

CL is Effectively as Powerful as res

In this section we present our main result, which is that CL is effectively as powerful as res. To accomplish this, we introduce a new method for comparing proof systems, called *effective p-simulation*.

Definition 4 *The following definition is adapted from (Hertel, Hertel, & Urquhart 2007).*

- *We say that $R(F,m)$ is a **polytime truth-preserving** CNF transformation if for all CNF formulas F, F is UNSAT iff $R(F,m)$ is UNSAT, and $R(F,m)$ runs in time polynomial in $|F|+m$, where $|F|$ is the length of F, and m is some auxiliary parameter.*
- *For proof systems S and T we say S **effectively p-simulates** T if there is a polytime truth-preserving CNF transformation $R(F,m)$ such that for all UNSAT formulas F, if m is* at least *the length of the shortest T-refutation of F, then $R(F,m)$ has an S-refutation whose size is polynomial in $|F|+m$.*

Theorem 1 *CL effectively p-simulates res.*

The proof is omitted due to lack of space, but is submitted in an appendix. In its place we provide the simpler proof that the stronger pool proof system effectively p-simulates res, which contains many of the same ideas.

Theorem 2 *pool effectively p-simulates res.*

Proof: We first give the algorithm for the polytime SAT encoding R. Let F be a CNF formula over n variables

$x_1, \ldots, x_n$, and let k be a positive number. We construct $R(F,k)$ as follows. Create k new variables $y_1, \ldots, y_k$, and $2n \cdot k$ new clauses $(y_i \vee x_j)$ and $(y_i \vee \neg x_j)$ for each $i \in [1,k]$ and each $j \in [1,n]$. Add each of these new clauses to F to form $R(F,k)$. If we assign every variable $y_i = 1$ in $R(F,k)$ it becomes F. Therefore, $F \in UNSAT$ iff $R(F,k) \in UNSAT$.

We now show that a pool proof of $R(F,m)$ exists that is not much larger than the smallest res proof of F. We will describe the pool proof's DAG structure. To do so, we must first consider the Resolution DAG π. Since π is a DAG, we can topologically sort it to produce a linear order O of the clauses which label its nodes, $C_1, C_2, \ldots, C_{|\pi|}$, such that if the derived clause C_k is the result of resolving clauses C_i and C_j, then $i < k$ and $j < k$. Consider the ordering O', formed by removing all of the initial clauses from O, $D_1, \ldots, D_m$, such that $m < |\pi|$. Clearly, $C_{|\pi|} = D_m = \emptyset$.

We set up the proof of $R(F,|\pi|)$ so that it derives each of the clauses O' in the same order that they appear in O'. Our proof is composed of two main parts: a "backbone," and a series of "side chains." Recall that $R(F,m)$ contains exactly as many new y_i variables as O' contains clauses. The "backbone" of the proof is a long sequence of nodes whose edges are labeled with each successive positive literal of y_i from $i = 1$ to $i = m$. The negative branch at each of these nodes leads to a "side chain". We learn a single clause of O down every side chain of our proof. They are learned in the same order as they appear in O, so the side chain which starts by setting $y_i = 0$ contains the node labeled by D_i.

We will now show how we derive successive clauses of the res proof down the corresponding side chains of our pool proof. Suppose for example that $D_1 = (x_1 \vee x_2 \vee \neg x_3)$, and is derived from initial clauses $C_a = (x_1 \vee x_2 \vee \neg x_4)$ and $C_b = (x_1 \vee \neg x_3 \vee x_4)$ in π. We want to learn D_1 down the first side chain. The chain therefore begins by setting $y_1 = 0$ and then falsifies both C_a and C_b to force a conflict in which we will derive their resolvent. The side chain therefore branches on x_1 at its next node.

The positive side of this branch is labeled by the initial clause $(y_1 \vee \neg x_1)$. The negative side extends the side chain. The next node branches on x_2. Again, the positive side is labeled by an initial clause, $(y_1 \vee \neg x_2)$, while the negative side extends the side chain. The next node branches on x_3 which appears negatively in the clause we wish to learn. So this time the negative side is labeled with an initial clause and we extend the side chain down the positive side. The side chain's final branch is on the variable which the res proof resolved on to produce D_1, namely x_4. The positive side is labeled with C_a and the negative side is labeled with C_b. These are resolved to label their parent. This is where we learn the resolvent of C_a and C_b which is D_1, as desired. The rest of the side chain is labeled in the obvious way, all the way back up to the backbone, where it branches positively on y_1. Clearly, this process works for any initially derived clause D_1 of a res proof and the example is for clarity's sake.

The second side chain branches negatively on y_2 to learn D_2 in the same way, except now D_2 might be the resolvent of two initial clauses or perhaps an initial clause and D_1. Clearly the structure outlined above works just as well when D_2 is the resolvent of two initial clauses as when one is D_1. If one is D_1 then we just connect the last node of the side chain back to the node labeled with D_1.

In future side chains, D_i could be the resolvent of two previously derived clauses. Inductively, in these cases, both derived clauses already label nodes in previous side chains. So the proof continues like this, building side chain after side chain to derive each D_i in the order they appear in O'. Eventually the empty clause, D_m, is derived at a node in the m^{th} side chain. By the degenerate resolution rule, we can label every node up the y_m side chain and then the whole backbone of the proof with the empty clause.

Clearly the backbone of the pool proof has size m and each side chain has size at most n. This proof is therefore only a polynomial factor larger than the original res proof. Furthermore, the resulting proof is pool, because the depth-first seach which traverses down each side chain successively is clearly regular, with each reuse of a D_i clause occurring across a cross-edge.

Note that the same size $O(mn)$ pool proof exists for any formula $R(F,m')$, where $m' \geq m$. $\square$

This theorem means that if F has a short res refutation, say of size $m = |\text{res}(F)|$, then we can transform F to a new equivalently satisfiable formula F' in time polynomial in m. Algorithm 1 can then be used to refute F' with a proof whose size is within a polynomial in m, the size of the original short res-refutation of F. (Its run time will be polynomially related to the size of the refutation it generates). That is, by first applying the simple transformation R to its inputs we can make Algorithm 1 p-simulate res.

Note that in this theorem Algorithm 1 is operating with the standard features of clause learning implementations, except for far backtracking. In fact, the construction used in the proof has no opportunities to use far backtracking, anyway. Therefore, the theorem is very applicable to standard clause learning implementations.

This theorem shows that CL is effectively as powerful as res. Though the result is weaker than the standard notion of p-simulation, it shows that a simple polytime encoding is all that is required to bridge the gap between res and CL.

Finding Short CL Proofs Is Hard

As stated earlier, the definition of a propositional proof system is agnostic about the difficulty of proof *search*, as is the definition of the power of a proof system. However, it turns out that proof search is generally harder in more powerful proof systems, since they have combinatorially larger spaces of proofs which must be searched. It is believed to be very hard to search for res proofs. People have devised algorithms like DPLL and CL in order to do some sort of (hopefully fast) systematic search of the space of res proofs. But these algorithms end up only searching small portions of the overall space of res proofs, which is why they correspond to refinements of res rather than to res itself.

To formalize the difficulty of searching for proofs, (Bonet, Pitassi, & Raz 2000) devised *automatizability*.

Definition 5 *(Bonet, Pitassi, & Raz 2000) A proof system S is automatizable if there is some S-based algorithm $\mathcal{A}$ which can find a proof of F in time only a polynomial factor longer than the best possible run of $\mathcal{A}$ on F.*

Showing that a proof system is not automatizable shows that there can be no algorithm based on it that works relatively quickly for all formulas.

In (Alekhnovich & Razborov 2001), it was proved that res is *not* automatizable unless a widely believed complexity assumption is false. We show that it is just as hard to search for CL proofs. This result is easily adapted to apply to more general formalisms for clause learning such as pool.

Theorem 3 *If CL is automatizable then so is res.*

Proof Sketch: Suppose we have some algorithm $\mathcal{A}$ which automatizes CL, i.e. given an arbitrary unsatisfiable formula CNF F over n variables, $\mathcal{A}$ produces a CL proof of F in time $g(m)$, where m is $|\mathsf{CL}(F)|$, the size of the smallest CL proof of F, and g is some polynomial.

We can use $\mathcal{A}$ to build an algorithm for automatizing res, i.e. for finding a res proof of an arbitrary unsatisfiable CNF F in polytime in $|\mathsf{res}(F)|$. We do so as follows: We first modify $\mathcal{A}$ so that it takes a time bound t as a parameter. If $\mathcal{A}(F, t)$ has not found a CL proof of F within time t, it rejects. By Theorem 1, if we know the number of derived clauses, m in the smallest res proof π of F, then we can construct a polytime truth-preserving encoding of F, $R(F, m)$, such that $R(F, m)$ has a CL proof, π', of size $O(n \cdot m)$, which we could find in polytime using $\mathcal{A}(R(F, m), g(n^2m))$.

Of course, we do not know m. We will therefore use successive guesses to determine a reasonably close upper bound on the true m. From the details of the proof of Theorem 1, it is clear that $R(F, s)$, for any $s \geq m$, has the same CL proof of size $O(n \cdot m)$ as $R(F, m)$. Therefore, the only way $\mathcal{A}$ could fail to find a proof after $g(n^2s)$ units of time (for a large enough n), is if s is too small a guess for m. If this occurs, we can just double s and try again, each time running $\mathcal{A}$ for no more than $g(n^2s)$ steps. After at most $\log m + 1$ iterations, we will have found an s such that $m \leq s \leq 2m$ and $\mathcal{A}$ will successfully find the CL proof of $R(F, s)$ within $g(n^2m))$ steps. The running time of this procedure is clearly $O(\log m \cdot g(n^2m))$, which is polytime in the size of π.

Since every CL proof is a res proof, we have obtained a res proof of $R(F, s)$ in polytime in the size of the smallest res proof of F. But what we really want is a res proof of F, not a res proof of $R(F, s)$. Since res is closed under restrictions, we can just hit the final proof with a restriction which satisfies every variable added by R, transforming the res proof of $R(F, s)$ into a res proof of F which is no longer than the original, thereby automatizing res. □

Conclusions and Open Problems

References

Alekhnovich, M., and Razborov, A. A. 2001. Resolution is Not Automatizable Unless W[P] is Tractable. In *IEEE Symposium on Foundations of Computer Science*, 210–219.

Alekhnovich, M.; Johannsen, J.; Pitassi, T.; and Urquhart, A. 2002. An Exponential Separation Between Regular And General Resolution. *Proceedings of 34th ACM Symposium on the Theory of Computing* 448 – 456.

Bayardo, R. J. J., and Schrag, R. C. 1997. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, 203–208.

Beame, P.; Kautz, H.; and Sabharwal, A. 2004. Towards Understanding and Harnessing the Potential of Clause Learning. *Journal of Artificial Intelligence Research* 22:319 – 351.

Bonet, M. L.; Pitassi, T.; and Raz, R. 2000. On interpolation and automatization for Frege systems. *SIAM Journal on Computing* 29(6):1939–1967.

Buresh-Oppenheim, J., and Pitassi, T. 2003. The Complexity of Resolution Refinements. *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science* 138 – 147.

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

Davis, M.; Logemann, G.; and Loveland, D. 1962. A Machine Program For Theorem Proving. *Communications of the ACM* 5:394 – 397.

Hertel, A.; Hertel, P.; and Urquhart, A. 2007. Formalizing dangerous sat encodings. In *SAT*, 159–172.

Marques-Silva, J. P., and Sakallah, K. A. 1996. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, 220–227.

Stallman, R., and Sussman, G. J. 1977. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence* 9:135–196.

Van Gelder, A. 2005. Pool resolution and its relation to regular resolution and DPLL with clause learning. In *LPAR*, 580–594.

Zhang, L.; Madigan, C. F.; Moskewicz, M. W.; and Malik, S. 2001. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, 279–285.