Clause Learning Can Effectively P-Simulate General Propositional Resolution

Philipp Hertel and **Fahiem Bacchus** and **Toniann Pitassi**
Department of Computer Science,
University of Toronto,
Toronto ON M5S 3G4, Canada
[philipp|fbacchus|toni] at cs.toronto.edu

Allen Van Gelder
University of California,
Santa Cruz,
CA, 95060, USA
avg at cs.ucsc.edu

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 utilize 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 surprising and indicates that clause learning is significantly more powerful than was previously known.

Introduction

DPLL (Davis, Logemann, & Loveland 1962) is a depth-first backtracking search procedure for solving SAT developed in the 1960s. 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(Marques-Silva & Sakallah 1996). 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 2005b).

In particular, we argue that previous formalizations have characterized clause learning as a system that is far more general than any practical implementation. Specifically, these formalizations have allowed clause learning to utilize extra degrees of non-determinism that would be very hard to exploit in practice. Hence, the formal results obtained from these characterizations are not as informative about practice as one would like. We identify some very natural constraints on a previous formalization that allows us to more accurately capture clause learning as it is implemented in practice.

Previous work has shown relatively little about the relationship between clause learning and general resolution. In this paper we show the surprising result that the formalization of clause learning we present here is essentially as powerful as general resolution, despite it being more constrained than previous formalizations. This immediately implies that previous formalizations are also essentially as powerful as general resolution, in a sense made precise in the paper. However, more importantly, since our formalization is much closer to practical clause learning algorithms, this indicates that clause learning as it is used in practice adds more power to DPLL than previously thought.

Our second result shows that as a consequence of this power no clause learning implementation can ever efficiently solve all inputs.[1] That is, for any deterministic implementation $\mathcal{A}$ there will always be formulas that some alternative clause learning implementation can solve super-polynomially faster than $\mathcal{A}$. This means, e.g., that there can be no universally effective branching heuristic for clause learning; any heuristic will be seriously sub-optimal on some inputs. This is in stark contrast to DPLL without clause learning, for which an algorithm exists that always runs in time quasi-polynomial in the size of the shortest DPLL refutation (in technical terms DPLL is quasi-automatizable). In sum, our two results show that clause learning is more powerful than previously known, but that finding short proofs with a clause learning algorithm is

[1]Unless a widely-believed complexity assumption is false.

harder than previously known.

As with previous work we analyze clause learning by formalizing it as a proof system. We therefore first present the formalism of proof systems and explain how proof systems are compared. We then explain our model of clause learning and how it generates proofs. This results in a characterization of clause learning as a proof system, and allows us to compare it with resolution. We show that our formalism of clause learning is essentially as powerful as resolution. Next we show how this power leads to our negative result, namely that no deterministic implementation of clause learning can ever fully realize the power of our clause learning.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.

Proof Systems and Resolution

For this paper, we are concerned with *resolution* proofs of 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, of 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 with which we are concerned 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 $|S(F)|$. 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.*

If the proof system S p-simulates the proof system T, then all formulas that have a short T proof also have a short S proof. Thus the range of formulas with short S proofs is at least as large as the range of formulas with short T proofs, and 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. Clearly, res p-simulates all of its 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 A and B with any mention of x and $\neg x$ removed.*
- *If, on the other hand, $x \notin A$ and $x \in B$, then $Res(A, B, x) = A$.*
- *Finally, if $x \notin A$ and $x \notin B$, then $Res(A, B, x)$ can be either clause, but in the interest of making the rule deterministic, we will always pick the shorter clause.*

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

Our definition of the resolution rule is slightly different from the standard definition because we allow degenerate resolutions. It is important to note that degenerate resolutions add no power to res. It is easily shown that a minimally sized res proof contains no degenerate resolutions (Van Gelder 2005a), even if it is permitted to use them. Use of the degenerate resolution rule can alter the structure of a proof and might therefore affect the power of resolution *refinements* (though it can never make them exceed the power of res). In particular, since clause learning based SAT-solvers cannot anticipate whether the variable branched on at a node will actually appear in the clauses learned at either of its children, they naturally generate proofs containing degenerate resolutions. We therefore obtain a more accurate formalization of standard clause learning implementations by extending resolution in this way. Previous formalizations of clause learning, notably (Beame, Kautz, & Sabharwal 2004; Van Gelder 2005b) also extend resolution in this way.

We define a res-proof of a 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 (leaves in the DAG) 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 applying the resolution rule to the clause-labels of its two children and 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.

The set of traces of a decision algorithm's computations on its inputs naturally produces a proof system. If 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 lower bound on the size of resolution proofs for some class of formulas implies a lower bound on the performance of such resolution-based algorithms on that class. The situation is made more complicated because efficiently refuting a formula with a resolution-based SAT-solver requires that 1) a short resolution proof *exists* for that formula, and that 2) such a short proof can be *found* quickly. Our two results basically show that clause learning is almost as good as res at producing short proofs, but as a consequence, it is just as hard to search for them as it is to search for short res 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 leaf 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, hence tree-proofs may always be considered to be a refinement of reg.

tree-proofs correspond very naturally to the traces of DPLL algorithms, but this correspondence breaks down once clause learning is employed. A relatively new refinement of res that is useful for analyzing clause learning is **pool resolution** (pool) (Van Gelder 2005b). Briefly, a res proof π is a pool resolution proof iff there is some depth-first search of π (starting at the root) 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, reg cannot p-simulate pool (Van Gelder 2005b).

It is also known that reg cannot p-simulate res (Alekhnovich *et al.* 2002; Urquhart 2008), but it is unknown whether pool can p-simulate res. Although that question remains open, 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. [2]

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 with **greedy conflict detection**, 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.

[2] We use "effective" in its natural meaning, not in the technical sense of "computable" as seen in the computability literature.

Algorithm 1: A1 (DPLL with Clause Learning). The objective of A1(ρ) 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  A1($\rho$) begin
3      if All clauses of $F$ are satisfied by $\rho$ then
4          Exit ($\rho$ 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 $\rho$ applied);
10     choose A literal $\ell$ such that neither $\ell$ nor $\neg\ell$ is in $\rho$;
11     $D_1 =$ A1($\langle\rho, \ell\rangle$);
12     $D_2 =$ A1($\langle\rho, \neg\ell\rangle$);
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
```

Either of these recursions could return a clause not containing the branch variable ℓ, in which case the value of D_3 is set 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. (D_1 is falsified by ρ so it is also falsified in the second recursive call). 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).

Algorithm 1 abstractly represents a class of algorithms, where each algorithm in this class resolves in some particular way the non-deterministic choices shown in the pseudocode. Algorithms that are essentially the same as standard deterministic implementations of DPLL with clause learning (but without far backtracking). In particular, algorithms that use greedy conflict detection (*GREEDY* = *true*) and algorithms that perform unit propagation are among the deterministic realizations of Algorithm 1.

Notice that we can ensure that unit propagation is performed by requiring that the non-determinism of variable selection (line 10) is resolved in such a way that it always selects a literal appearing in a unit clause of $F|_\rho$ if one exists.

Thus a decision literal is chosen only after unit propagation is exhausted.

The deterministic realizations of Algorithm 1 include versions that support the derivation of 1-UIP clauses, but lack far backtracking. Far backtracking is a form of partial restarts that comes into play following a 1-UIP derivation. In fact, when a *unit* 1-UIP clause is learned, far backtracking backtracks to the root of the search tree, causing a total restart. However, our main result on the power of Algorithm 1 shows that far backtracking does not add any extra theoretical power, even though it may be useful in practice. For this reason we assume in the rest of the paper that far backtracking is not being employed and take Algorithm 1 to be our 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 2005b) show that the proofs generated by the most general form of Algorithm 1, in which neither mandatory unit propagation nor greedy conflict detection are enabled, are pool proofs. Van Gelder also shows that every pool proof can be generated by 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 the falsification in the algorithm of an earlier-derived clause.

Most standard implementations of clause learning include unit propagation and greedy conflict detection. As a consequence, the space of pool proofs is much richer than the space of clause learning computations. To more accurately characterize the theoretical power of clause learning, we therefore define a new refinement of pool that we call CL, which is much closer to clause learning as it is used 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 conflict detection 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** (CL^+).

The difficulty with CL^+ 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 CL^+ proofs. The CL^+ formalism is therefore very hard to analyze in its full generality. In particular, a proof separating res from CL^+ would require one to prove that *for all* learning schemes $\mathcal{S}$, there is no CL^+ proof utilizing $\mathcal{S}$ than can match the performance of res on some infinite family of formulas. Nor is it clear that the full generality of CL^+ 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 CL^+ using any of these schemes produces proofs that are in pool (Van Gelder 2005b). (As mentioned, restarts and far backtracking from 1-UIP clauses do not fall into this framework.) Hence, pool can reasonably be said to subsume CL^+ for any practical clause learning scheme. CL further refines pool to make it an even better formalization of clause learning.

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 use a recent method for comparing proof systems, called *effective p-simulation*, proposed by (Hertel, Hertel, & Urquhart 2007).

Definition 4 *Let S and T be two proof systems. We say that S* ***effectively p-simulates*** *T if there is an algorithm R that takes as input a CNF formula F, and outputs a CNF formula $R(F)$ such that*

- *$F \in$ sat iff $R(F) \in$ sat,*
- *the run time of R is polynomial in $|F|$,*
- *and if F has a T proof of size s then $R(F)$ has an S proof whose size is polynomial in s.*

Theorem 1 *CL effectively p-simulates res.*

This theorem means that if F has a short res refutation, then we can transform F to a new equivalently satisfiable formula F' in time polynomial in $|F|$. Algorithm 1 can then be used to refute F' with a proof whose size is polynomial in $|\mathsf{res}(F)|$, the size of the short res-refutation of F. That is, by first applying the transformation R, described in Def. 6, to its inputs we can make Algorithm 1 p-simulate res.

Note that the theorem states that this can be accomplished by Algorithm 1 operating with the standard features of clause learning implementations, except for far backtracking. That is, we are restricting Algorithm 1 so that it generates CL proofs. Furthermore, the construction used in the proof does not use far backtracking. Therefore, since CL with far backtracking is still a subsystem of res, the theorem also shows that far backtracking adds no extra theoretical power to clause learning. It is possible, however, that far backtracking might allow CL to p-simulate res—i.e., allow CL to generate as short proofs as res without first having to transform the input to $R(F)$.

Though the result that CL effectively p-simulates res is weaker than the standard notion of p-simulation, it shows that an encoding that can be performed in deterministic polynomial time is all that is required to bridge the gap between res and CL. Note that $R(F)$ does not need to *know* anything about the proofs of F.

We provide an outline of the proof of Theorem 1 in the appendix. Using the same reduction which provides us with our effective p-simulation, we also prove the following theorem. A similar theorem was first proved in (Beame, Kautz, & Sabharwal 2004). This theorem allows us to conclude that most other refinements of res, including Regular Resolution, are weaker than CL.

Corollary 2 *Every resolution refinement S, that is closed under taking restrictions and cannot p-simulate res also cannot p-simulate CL.*

Proof: Let R be the polytime reduction which allows CL to effectively p-simulate res, as described above. If S cannot p-simulate res, then there must be some infinite family of unsatisfiable formulas, Γ, whose smallest S proofs are super-polynomially larger than its smallest res proofs. Now consider the family of formulas $\Gamma' = \{R(F) : F \in \Gamma\}$. Since S is closed under taking restrictions and the restriction that sets $y_0 = 0, a = 0$, and $b = 0$ transforms $R(F)$ back into F (see Def. 6), S must still have high proof complexity on Γ', but the proof complexity of CL on Γ' will be within a polynomial factor of res's. So S cannot p-simulate CL because of Γ' (technically we say Γ' super-polynomially separates CL from S). □

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 deterministic implementations of CL 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 $f(n,s)$-**automatizable** if for all formulas F, an S proof of F can be found in time $f(n,s)$ where n is the size of F and s is the size of the smallest S proof of F. A proof system S is **automatizable** if it is $f(n,s)$-automatizable, for some polynomial function $f(n,s)$. A proof system S is **quasi-automatizable** S is $f(n,s)$-automatizable, for some quasi-polynomial function $f(n,s)$. (Quasi-polynomial means $O(n^{\log^c n})$ for some constant c.)*

Showing that a proof system S is not automatizable shows that there can be no algorithm based on it that works relatively quickly for all formulas. That is, any algorithm for generating S proofs must sometimes generate a proof that is super-polynomially longer than necessary (and thus take super-polynomially more time than needed). In (Alekhnovich & Razborov 2001), it was proved that res is *not* automatizable unless the widely believed complexity assumption that $W[p]$ *is intractable* is false. A reviewer pointed out that the proof in (Alekhnovich & Razborov 2001), also implies that CL is not automatizable. But that proof leaves open the possibility that CL is quasi-automatizable while res is not. Currently, the best known lower bound on the automatizability of res is quasi-polynomial, while the best known upper bound is exponential. Before our work, it was unknown whether clause learning might be as easy to automatize as Tree Resolution, which has both quasi-polynomial upper and lower bounds, or as hard as res. We have shown that it is as hard to automatize CL as it is to automatize res. This result is easily adapted to apply to more general formalisms for clause learning such as pool.

Theorem 3 *If CL is $f(n,s)$-automatizable (resp. quasi-automatizable) then res is $g(n,s)$-automatizable, where $g(n,s)$ is within a polynomial factor of $f(n,s)$.*

Proof Sketch: We show that we can build a res solver that is only a polynomial factor slower than an assumed polytime CL solver. If some algorithm $\mathcal{A}(F)$ produces a CL proof of F in time polynomial (resp. quasi-polynomial) in the size of the smallest CL proof of F, then run $\mathcal{A}(R(F))$ (Def. 6) obtaining a CL proof π of $R(F)$. We can then transform this into a res proof of F in polynomial time. This is possible because res is closed under taking restrictions, so we can restrict both π and $R(F)$ by the assignment $y_0 = 0, a = 0, b = 0$, which transforms $R(F)$ back to F and transforms π into a refutation of F. The full proof is included in a thesis (Hertel 2008). □

Conclusions and Open Problems

In this paper, we have studied the power of clause learning and the closely connected question of its automatizability. Clause learning sits in the middle between DPLL and general resolution, and was hoped to have the best of both worlds. DPLL, on the one hand, is tractable from an implementation point of view because the only nondeterminism that has to be resolved is in the branching heuristic. This limited nondeterminism is tractable in the sense that there is an algorithm that searches for DPLL proofs and that runs in time almost polynomial in the size of the shortest DPLL proof. (In technical terms, DPLL is quasi-automatizable.) In contrast, the nondeterminism of general resolution is provably intractable. That is, we know that general resolution is not automatizable and it is widely believed that in general it requires exponential time find the shortest general resolution proofs.

Clause learning, while known to be more powerful than DPLL, has the same amount of nondeterminism, since the only source of nondeterminism comes from the branching heuristic. Thus one might hope that it should also be automatizable. We show somewhat surprisingly that this isn't the case. An explanation lies in the difference in "budget": in order to automatize clause learning, we must run in time polynomial in the size of the shortest CL proof, which can be significantly smaller than the size of the smallest DPLL proof. The practical implication is that no implementation generating CL proofs can be effective on all formulas: some formulas with short CL proofs will run very inefficiently. On the positive side, we prove that CL is more powerful than expected (it can effectively p-simulate resolution).

Acknowledgments

We thank the anonymous reviewers for numerous helpful comments.

Appendix: Proof of Theorem 1

In this appendix we provide an outline of the proof of our main result, Theorem 1. Showing that proof system S p-simulates proof system T requires one to demonstrate that, for every unsatisfiable CNF formula F, the smallest S proof of F is at most a polynomial factor larger than the smallest T proof of F. This usually involves describing a function which maps S proofs to similarly sized T proofs. An effective p-simulation is a little more complicated. It requires one to demonstrate that there exists a deterministic, polytime computable function R which maps formulas onto formulas, such that for every unsatisfiable CNF formula F, $F \in \mathsf{sat}$ iff $R(F) \in \mathsf{sat}$, and the smallest S-proof of $R(F)$ is at most a polynomial factor larger than the smallest T-proof of F.

This proof therefore has a number of subparts. First we start with a specification of the transformation function R that is applied to the input formula. We then prove that $F \in \mathsf{sat}$ iff $R(F) \in \mathsf{sat}$. We then describe how to transform a res proof π of F into a CL proof of $R(F)$, which is only a polynomial factor larger than π. We begin with the encoding:

Definition 6 *Let CNF formula F consist of clauses $C_1, \ldots, C_m$ and have variables $x_1, \ldots, x_n$. The notation $\pm v$ means that we include separate clauses with each combination of v and $\bar{v}$. The encoding $R(F)$ is defined as follows:*

- *Introduce new variables, $a, b, c, d, e,$ $y_0, y_1, \ldots, y_n$, and $z_1, \ldots, z_n$.*

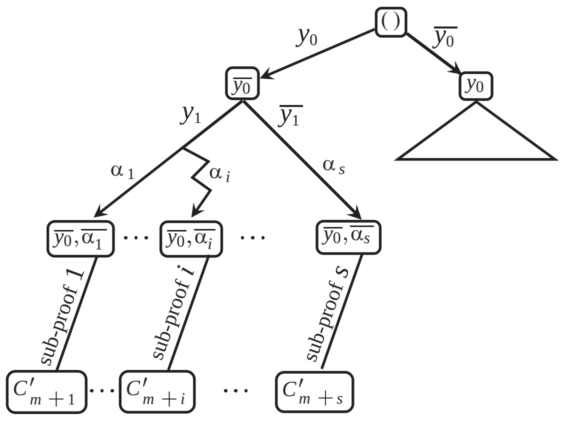

Figure 1: Overview of the construction for Theorem 1, showing a balanced binary tree with s leaves that branches on $y_1, y_2, \ldots$ to the depth necessary.

- *Introduce $O(n^3)$ new* ***Type I*** *clauses $(\bar{y}_0, p, \pm y_j, z_i, \pm e)$, for all $1 \leq j \leq n$ and all $j \leq i \leq n$ and all $p \in \{\bar{a}, \bar{b}, \pm x_1, \ldots, \pm x_n\}$.*
- *Introduce $O(n)$ new* ***Type II*** *clauses $(\bar{y}_0, p, \bar{z}_1, \ldots, \bar{z}_n, \pm e)$ for all $p \in \{\bar{a}, \bar{b}, \pm x_1, \ldots, \pm x_n\}$.*
- *Introduce 24 new* ***Type III*** *clauses $(\bar{a}, \bar{b}, \pm c, \pm d, \pm y_0)$, $(\bar{a}, b, \pm c, \pm d, \pm y_0)$, $(a, \bar{b}, \pm c, \pm d, \pm y_0)$.*
- *Finally, we transform every clause C_j of F into $C'_j = (C_j, a, b)$.*

We now prove that $F \in \mathsf{sat}$ iff $R(F) \in \mathsf{sat}$. An assignment of 0 to y_0, a, and b transforms $R(F)$ back into F, so any assignment which satisfies F can be extended to an assignment which satisfies $R(F)$. But every extension of every assignment which falsifies F also falsifies some clause of $R(F)$. To prove this, consider any falsifying assignment of F. Applying it to $R(F)$ transforms every C'_i clause into (a, b). We can resolve this clause with the Type III clauses to derive the empty clause. So $F \in \mathsf{sat}$ iff $R(F) \in \mathsf{sat}$.

We now describe the main ideas of the simulation by describing the construction of the CL refutation corresponding to an arbitrary res refutation. The main ideas of the simulation are shown in the figures. Due to space constraints, we do not provide a proof of correctness for our construction. The full proof appears in (Hertel 2008).

Let $\pi = C_1, C_2, \ldots, C_m, C_{m+1}, \ldots, C_{m+s}$ be a res refutation of F. We will now show that there is a size $O(s \cdot n^2)$ CL refutation of $R(F)$. Note that clauses $C_{m+1}, \ldots, C_{m+s}$ are the derived clauses of π. We will produce a CL proof which derives these clauses in the same order as they appear in π. To be precise, we will not derive the exact same clauses. Since we added a and b to every initial clause of F to form $R(F)$, we will actually derive $C_i \vee a \vee b$ in the CL proof whenever we would derive C_i in π. Just as with the initial clauses, we call such a derived clause C'_i. Since π ends by deriving $C_{m+s} = []$, we will end up by deriving $C'_{m+s} = (a, b)$. We will then use this

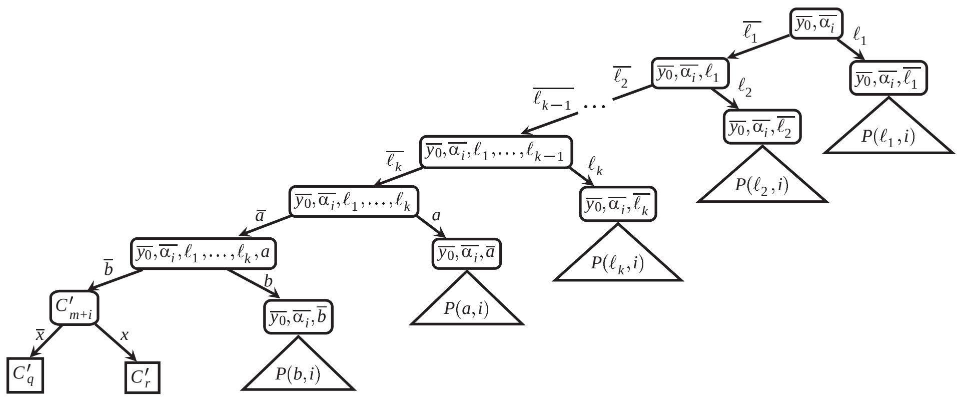
$\overline{y_0},\overline{\alpha_i}$
$\overline{\ell_1}$
ℓ_1
$\overline{y_0},\overline{\alpha_i},\ell_1$
$\overline{y_0},\overline{\alpha_i},\overline{\ell_1}$
$\overline{\ell_2}$
ℓ_2
$\overline{\ell_{k-1}}$ …
$\overline{y_0},\overline{\alpha_i},\overline{\ell_2}$
$P(\ell_1,i)$
$\overline{y_0},\overline{\alpha_i},\ell_1,\ldots,\ell_{k-1}$
$\overline{\ell_k}$
ℓ_k
$P(\ell_2,i)$
$\overline{y_0},\overline{\alpha_i},\ell_1,\ldots,\ell_k$
$\overline{y_0},\overline{\alpha_i},\overline{\ell_k}$
$\overline{a}$
a
$\overline{y_0},\overline{\alpha_i},\ell_1,\ldots,\ell_k,a$
$\overline{y_0},\overline{\alpha_i},\overline{a}$
$P(\ell_k,i)$
$\overline{b}$
b
C'_{m+i}
$\overline{y_0},\overline{\alpha_i},\overline{b}$
$P(a,i)$
$\overline{x}$
x
C'_q
C'_r
$P(b,i)$

Figure 2: Sub-proof i of the simulation.

clause along with the 12 clauses of Type III which contain the positive literal of y_0 to derive [].

We will describe how to construct the refutation from the root downwards. We begin our refutation by branching on y_0. On the positive side of this branch, we will first derive each successive clause from $C'_{m+1},\ldots,C'_{m+s}$. Once we have derived C'_{m+s}, we will use it to derive [] on the negative side of y_0. The general form of the proof can be seen in Figure 1. On the positive side of the root the refutation forms a binary tree with exactly s leaves, which we name node 1 through to node s. Each of the clauses $C'_{m+1},\ldots,C'_{m+s}$ will be derived in a subproof rooted at one of these leaves, with C'_{m+i} being derived in the subproof rooted at node i. The tree is composed entirely by branching on y_i variables, where the variable's subscript corresponds to its depth in the tree. Furthermore, the tree is constructed so that paths to the "left" are never shorter than paths to the "right". Clearly, no path will ever be repeated twice. We introduce two pieces of notation: $\overline{\alpha_i}$ refers to the clause which contains exactly the literals falsified by α_i, and y^i_j refers to the literal of variable y_j falsified by α_i.

We will now show how to construct subproof i, the subproof rooted at node i, at the end of α_i. Clause C'_{m+i} is derived deep within subproof i. Suppose that clause $C_{m+i} = (l_1 \vee \cdots \vee l_k)$ is derived from clauses C_q and C_r ($q, r < m+i$) by resolving on variable x in π. In our refutation, we will therefore derive clause $C'_{m+i} = (l_1 \vee \cdots \vee l_k \vee a \vee b)$ from clauses C'_q and C'_r by resolving on variable x in subproof i.

The idea is to falsify every literal in C'_{m+i} down the backbone of subproof i. This falsifies every literal in C'_q and C'_r except for x. Unit propagating x will then allow us to derive C'_{m+1} by resolving the already existent C'_q and C'_r on x. Figure 2 shows subproof i. Each node is labeled with the clause derived at it, and each edge is labeled with literal set when crossing the edge. A copy of the proof $P(p,i)$ is rooted at the negative branch of each node in the backbone of subproof i, where p is the variable of C'_{m+i} branched on at that node of the backbone. We will describe the construction of $P(p,i)$ in the next paragraph. For now, the most important fact about each $P(p,i)$ proof is that the clause labeling its root contains $\overline{\alpha_i}$. Every node of subproof i's backbone will therefore also be labeled with $\overline{\alpha_i}$. This will avoid unwanted cache hits. The literals of C'_{m+i} can be branched on down the backbone in any order (but we can set a lexicographical ordering for concreteness), except that a must be negated second last and then b must be negated last. Call the node at the end of this backbone v. This is the node that we will label with C'_{m+i}. At v, we branch on x. Setting $x = 0$ falsifies C'_q, while setting $x = 1$ falsifies C'_r. If C'_q or C'_r is an initial clause of F, then we add a sink node labeled with the appropriate clause as a child to v. If one is a derived clause, then we add a cross edge from v to the node of the earlier subproof where that clause was derived.

We now explain how to construct each $P(p,i)$ proof. The structure of $P(p,i)$ depends on the length of α_i. The root of $P(p,i)$ is labeled by the clause $(\bar{p} \vee \bar{y}_0 \vee \overline{\alpha_i})$. $P(p,i)$ immediately branches on the variable e. On both sides of this branch, $P(p,i)$ is composed of two linear chains of unit resolutions. When e is set in either direction, every z_j, $1 \leq j \leq n$ is unit propagated because of a Type I clause. Accordingly, each chain is built by branching on each z_j from $j = 1$ to $j = |\alpha_i|$. The negative side of each branch immediately falsifies the initial clause of Type I which includes z_j, p, and y^i_j. After positively setting $z_{|\alpha_i|}$, we continue to unit propagate z_j from $j = |\alpha_i| + 1$ to $j = n$, except that the falsified Type I initial clause on the negative side now contains p and $y^i_{|\alpha|}$ rather than y^i_j. We must therefore branch on y_j there. Finally, setting $z_n = 1$ falsifies the Type II clause which includes p.

The proof's depth-first traversal first branches positively on y_0 and visits every subproof in order from $i = 1$ to $i = n$. Within each subproof, the traversal travels down the entire backbone before entering each $P(p,i)$ on the way back up. We do not provide the full proof of correctness here, but will give a high level intuition. We argue that the traversal will not violate the greedy unit propagation or cache hit conditions, so the only cache hits in the entire proof are those which hit one of the derived clauses from the original res refutation. For the most part this is true, but some greedy cache hits do occur from some $P(p,i)$ proofs to earlier ones. We point out under which conditions this can occur and then argue that these cache hits do not interfere with the simulation. For all the details, see (Hertel 2008).

References

Alekhnovich, M., and Razborov, A. A. 2001. Resolution is Not Automatizable Unless W[P] is Tractable. In *FOCS*, 210–219.

Alekhnovich, M.; Johannsen, J.; Pitassi, T.; and Urquhart, A. 2002. An Exponential Separation Between Regular And General Resolution. *STOC* 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. *JAIR* 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. *LICS* 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.

Hertel, P. 2008. *Ph.D. Thesis*. Ph.D. Dissertation, Department of Computer Science, University of Toronto.

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.

Urquhart, A. 2008. Regular and general resolution: An improved separation. In *SAT*, 277–290.

Van Gelder, A. 2005a. Input distance and lower bounds for propositional resolution proof length. In *SAT*.

Van Gelder, A. 2005b. 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.