Journal of Artificial Intelligence Research 53 (2015) 497-540 Submitted 01/15; published 07/15

Satisfiability and Systematicity

Matthew L. Ginsberg
Connected Signals, Inc.
355 Goodpasture Island Road, Suite 200
Eugene, Oregon 97401

Abstract

We introduce a new notion of systematicity for satisfiability algorithms with restarts, saying that an algorithm is *strongly* systematic if it is systematic independent of restart policy but *weakly* systematic if it is systematic for some restart policies but not others. We show that existing satisfiability engines are generally only weakly systematic, and describe FLEX, a strongly systematic algorithm that uses an amount of memory polynomial in the size of the problem. On large number factoring problems, FLEX appears to outperform weakly systematic approaches.

1. Introduction

Once upon a time, DPLL (Davis, Logemann, & Loveland, 1962; Davis & Putnam, 1960) was the algorithm of choice for solving Boolean satisfiability problems, or SAT. There were three reasons for this.

First, DPLL was *systematic* in that if a particular problem had a solution, DPLL would eventually find it. If a problem had no solution, DPLL would identify it as unsatisfiable. Such properties are essential if we want to simply invoke a solver, allow it as long as necessary to solve a problem, and be assured that some answer will always result.

Second, DPLL used an amount of memory *polynomial* in the size of the problem. The amount of time used was, of course, exponential given the NP-complete nature of SAT. But the memory used was polynomial, and therefore only logarithmic in the running time.

And finally, DPLL *worked*. Its success was only a harbinger of the uses to which later SAT engines would be put, but the range of problems to which DPLL could practically be applied exceeded that of any previous general-purpose NP algorithm. Today that range is wider still, including microprocessor verification (Kaivola, Ghughal, Narasimhan, Telfer, Whittemore, Pandav, Slobodov, Taylor, Frolov, Reeber, & Naik, 2009), device driver validation (Moura & Bjørner, 2010) and many others (Biere, Heule, van Maaren, & Walsh, 2009).

Two changes led to significant algorithmic improvements. The first was the recognition that a DPLL backtrack was best thought of not as a move in a tree-like search space, but instead as a resolution-based inference step. This idea appeared first in Stallman and Sussman's (1977) work on dependency-directed backtracking and Doyle's subsequent (1979) work on truth maintenance. All of these earlier authors, however, described algorithms that might accumulate an exponential number of resolution consequences as the search proceeded.

Ginsberg's (1993) work on dynamic backtracking was the first to show that systematic nonchronological inference methods could be constructed using only a polynomial amount of memory, describing the algorithm as an extension of backjumping (Gaschnig,

©2015 AI Access Foundation. All rights reserved.

1979). Bayardo and Schrag (1997) continued to work on the approach, renaming it (sensibly) relevance-bounded learning. These ideas are currently known as *conflict driven clause learning*, or CDCL, a name that was introduced by Ryan (2002) and then popularized by Marques-Silva, Lynce and Malik (2009).

The work through 1997 involved using a scheme for retaining learned clauses that guaranteed that only a polynomial number of such clauses would be kept at any particular point in the search. With GRASP (Marques-Silva & Sakallah, 1999), this requirement was dropped. Virtually all modern SAT solvers use CDCL, generally modified in a way that either allows an unlimited number of clauses to be collected (as in TINISAT, Huang, 2006; or Quest0.5, Lynce, Baptista, & Marques-Silva, 2001) or, more commonly, in a way that no longer guarantees systematicity (zCHAFF, Moskewicz, Madigan, Zhao, Zhang, & Malik, 2001; RSAT, Pipatsrisawat & Darwiche, 2007; MINISAT, Sörensson & Een, 2005, and many others).

The abandonment of systematicity was linked to the inclusion of *restarts* in satisfiability algorithms. This appears to have been due to Gomes, Selman and colleagues (1998); the basic idea is to escape from areas where the solver is “thrashing” (searching nearly endlessly without ever identifying the real source of a problem) by simply restarting from scratch. Put somewhat differently (but equivalently), restarts provide an out when the solver makes a mistake near the root of the search tree, what Harvey (1995) refers to as an “early mistake.”

In the setting provided by DPLL, restarting the prover will inevitably result in nonsystematicity. This is because crucial information about the state of the search is stored in the position in the search tree, and restarting the search abandons this information.

The work on restarts has matured in two separate directions. First, there has been a considerable body of work examining the theoretical properties of search with restarts in the CDCL case where all of the clauses are retained as the search proceeds. Such a search is obviously systematic, since only a limited number of new clauses can be learned before either a solution is found or the empty clause is derived (proving the original problem to be unsatisfiable).

There are more interesting conclusions to be drawn, however, The fact that restarts can prevent thrashing appears to be related to the fact that they appear to allow foxr more flexible inference schema than previous methods. This work was begun by Beame and colleagues (2004), continued by Buss et al. (2008) and Hertel et al. (2008), and extended further by Pipatsrisawat and Darwiche (2011), who showed that CDCL with restarts is equivalent to general resolution. CDCL without restarts is not known to be generally able to produce proofs exponentially shorter than tree-based resolution, and general resolution can lead to exponentially shorter proofs than tree-based resolution in many instances (Ben-Sasson, Impagliazzo, & Wigderson, 2000).[1]

In addition to this theoretical work, practical work was proceeding as well, focusing in large measure on the development of strategies identifying points at which the search should be restarted. It was realized fairly quickly that virtually any restart policy was an improvement on not restarting at all. Luby et al. (1993) had developed a restart policy that they showed was within a logarithmic factor of optimal on a wide range of problems. The

1. It is also not known that tree-based resolution *can't* polynomially simulate general resolution (Bonet & Johannsen, 2014).

Luby restart policy was shown to outperform a variety of alternatives in the SAT domain by Huang (2007).

The Luby restart policy involves gradually (but not monotonically) increasing the number of backtracks between restarts; roughly speaking, the $2^{k\text{th}}$ restart is the first that allows 2^k backtracks before restarting once again. We will refer to the effort undertaken between consecutive restarts as a *probe*; the number of backtracks between the restarts will be called the *size* of the probe.

The Luby approach has two useful consequences. The first is that the total number of backtracks grows quadratically with the number of restarts because of the infrequency of probes of large size.

The second consequence of the Luby policy is that the search is once again systematic, assuming that the original procedure, without restarts, was itself systematic. The reason is that the Luby restart policy will eventually involve probes of sufficiently large size that systematicity is guaranteed.

This is a distinction worth formalizing. Given an algorithm A that involves a restart policy, we will say that A is *strongly systematic* if it is systematic independent of the restart policy chosen. We will say that A is *weakly systematic* if it is systematic for some restart policies but not for others, and we will say that A is *nonsystematic* if it is not systematic for any restart policy.

The connection between systematicity and the amount of memory used is sufficiently important that we present the following classification of some SAT solvers:

	Strongly Systematic	**Weakly Systematic**	**Nonsystematic**
Exp-space	TINISAT	—	—
Polyspace	?	MINISAT	WSAT

By exp-space, we mean algorithms that demonstrably may take an amount of space exponential in the size of the problem being solved; "polyspace" algorithms are those that remove learned clauses and thus could, at least in theory, use only polynomial amounts of space as a result.

Memory is cheap these days; perhaps exp-space is not as much of a drawback as it once was. But this turns out not to be the case, for the following reason.

Most of the CPU time consumed by SAT engines is spent in the unit propagation procedure, which finds obvious consequences of the variable assignments made at any particular point in the search. Although there have been a variety of engineering improvements that speed this process considerably (Moskewicz et al., 2001), the overall time can still be linear in the size of the accumulated clausal database. As this database grows exponentially in size, the time needed for a single inference can therefore grow exponentially as well.

Given TINISAT's existence in the upper left, there has been little interest in an exp-space solver with weaker systematicity properties. But TINISAT cannot really be considered a production solver; on difficult problems, the number of stored clauses grows impractically and the solver itself essentially grinds to a halt.

We have not previously discussed the WSAT family of algorithms (Selman, Kautz, & Cohen, 1993), nor shall we. These algorithms are very different in spirit from the ones that we have described thus far, working with complete variable assignments and using hill

climbing to attempt to gradually convert these assignments to solutions of the problems being investigated. While competitive in the 1990's, the progress on systematic algorithms has been such that these fundamentally nonsystematic approaches are currently given relatively little attention, although recent work has shown that the connection between these algorithms and the CDCL approach is closer than one might think (Goultiaeva & Bacchus, 2012).

Most modern SAT solvers join MINISAT in the middle of the bottom row, weakly systematic but using only a relatively modest amount of memory to get the job done. Removing some clauses is essential in any practical solver and there simply has been no known strongly systematic algorithm capable of doing so. The general view has been that although not strongly systematic, these systems work exceptionally well in practice, including managing to produce proofs of unsatisfiability when necessary.

It should be noted, however, that this loss of systematicity is more than a simple theoretical concession. It is obviously important, for example, that distinct probes begin their search from distinct locations in the search space. Should those starting locations be "close" in some sense, it is also important that the corresponding probes overlap as little as possible. We believe that the experimental results in Section 4 support this intuition.

As we have remarked, our work in dynamic backtracking introduced the first nonchronological, polyspace, weakly systematic SAT engine. Our goal in the current paper is to describe a new CDCL-based SAT engine FLEX that is both polyspace and *strongly* systematic. The outline of the paper is as follows. In Section 2, we introduce the necessary notation and describe a standard CDCL algorithm. We also describe the choices typically made in implementing such algorithms.

Section 3 describes the FLEX algorithm; the proof that it is both systematic and polyspace is deferred to an appendix. We discuss the additional data structures needed to guarantee systematicity, and the restrictions that must be placed on some standard choices as a result. Experimental results are presented in Section 4 and concluding remarks are in Section 5.

2. Background, Notation and Existing Work

By a *satisfiability problem*, we will mean a collection of clauses over Boolean variables such as:

$$\begin{gathered} a \vee b \\ a \vee \neg b \\ \neg a \vee b \\ \neg a \vee \neg b \vee \neg c \end{gathered}$$

Each of the clauses is a disjunction, and the goal is to value all of the variables so that each clause is satisfied. In this particular example, we have to make a and b true, and c false.

Definition 2.1 *A* variable *is a letter, such as a, b, c and so on. A* literal *is a variable or the negation of a variable. A* clause *is a disjunction of literals; the (unsatisfiable) empty clause will be denoted* $\perp$*. A* theory *is a conjunction of clauses.*

An interpretation *or* bias *is an assignment of true or false to each variable in a theory. An interpretation will be said to be a* model *of a theory T if every clause in T is satisfied by the interpretation.*

We will often call theories SAT *instances* or *satisfiability problems*, and will think of clauses simply as sets of the literals they contain, writing, for example, that $a \in a \vee b$. We will similarly think of theories as sets of clauses, and will think of the size of a theory T as simply the number of clauses in T, denoting it by $|T|$.[2]

Modern satisfiability algorithms work by manipulating partial assignments of values to variables; the idea is to either extend the partial assignment to value more variables (and eventually to a model of the theory in question) or to show that a particular partial assignment cannot be extended to a model.

Definition 2.2 *A* partial assignment *is a sequence $P = \langle l_1, l_2, \ldots, l_n \rangle$ of literals such that if $i \neq j$, then $l_i \neq l_j$ and $l_i \neq \neg l_j$. A variable v will be called* valued *by P if $v \in P$ or $\neg v \in P$. A literal l will be called* satisfied *by P if $l \in P$, and will be called* falsified *by P if $\neg l \in P$. A clause c will be called* falsified *by P if every literal $l \in c$ is falsified by P.*

In practice, the variables assigned values in a partial assignment are typically annotated with reasons of some sort. A variable value may be the result of a choice made by the search engine, or the obvious result of previous choices:

Definition 2.3 *An* annotated partial assignment *is a sequence $\langle (l_1, c_1), (l_2, c_2), \ldots, (l_n, c_n) \rangle$ of pairs, where each pair contains a literal l_i and a clause c_i. The clause c_i will be called the* reason *for l_i, and the set $\{c_1, \ldots, c_n\} - \{\bot\}$ will be denoted $R(P)$ and called the* reasons *of the annotated partial assignment.*

Given an annotated partial assignment P, the ith pair $(l, c) \in P$ will be called justified *if either $c = \bot$ or $l \in c$ and every literal in c other than l is $\neg l_j$ for some $j < i$. An annotated partial assignment P will be called* justified *if every $(l, c) \in P$ is justified.*

Note that a justified partial assignment can always contain pairs of the form $(l, \bot)$, since the justification condition is trivially satisfied. We will use this to indicate that l is a choice made by the search algorithm:

Definition 2.4 *If P is an annotated partial assignment and $(l, \bot) \in P$, we will say that l is a* choice *in P. The set of choices in P will be denoted by $C(P)$. The set of negations of choices in P will be denoted by $\neg C(P)$.*

Similarly, a justified partial assignment can always contain pairs of the form (l, l), which we will use to indicate that l itself has been proven to be a consequence of the theory being investigated.

In general, all of the partial assignments that we consider will be justified.

2. From a complexity perspective, it might seem more natural to think of the size of a theory T as the total number of literals in the clauses in T. Our choice is a bit more convenient notationally and for a theory involving v variables, the total number of literals in T is obviously at most $v|T|$.

Definition 2.5 *Let P be an annotated partial assignment. Then if (l,c) is the ith element of P, we will say that the* assertion level *of l is*

$$|\{j \leq i | c_j = \perp\}|$$

For a clause α, we will say that the assertion level *of α is the maximum of the assertion levels of the literals in α.*

In other words, the assertion level of a literal is the number of choice variables that precede it in P.

Definition 2.6 *Let P be an annotated partial assignment for a theory T. A clause $\alpha = l_1 \vee \cdots \vee l_n$ such that no literal in α is satisfied by P, and exactly one literal is unvalued by P, will be called* unit*.*

If a single literal $l \in \alpha$ is unvalued by P for some unsatisfied clause α, then any extension of P that is a model of T must include l, and P will remain a justified partial assignment if we add (l,α) to its end. This is called *unit propagation*:

Definition 2.7 Unit propagation *is a procedure* UNIT *that accepts as input a theory T and a justified partial assignment P. It returns a pair (P',c) where P' is a maximal justified partial assignment that extends P, so that $P \subseteq P'$, and either $c =$ `true` or $c \in T$ is falsified by P'.*

If unit propagation returns $c =$ `true`, unit propagation "succeeded". If it returns a partial assignment P' and a clause c falsified by P', a contradiction has been found and c is the reason.

Definition 2.8 *We will say that the* backtrack level *of α is zero if every literal in α has the same assertion level, or the second greatest of the assertion levels of literals in α otherwise.*

Given an annotated partial assignment P and clause α, we define the result of backtracking *to α, to be denoted* BACKTRACK(P,α)*, to be the result of removing from P any literal with assertion level greater than the backtrack level of α.*

If α is a clause with backtrack level $n \neq 0$, there is clearly some literal $l \in \alpha$ such that the assertion level of l is greater than n. If this literal l is unique, then backtracking to the backtrack level of α will leave every element of α other than l unchanged, so that α itself (which was a contradiction prior to the backtrack) will be a unit clause after the backtrack is complete.

Most SAT implementations spend the bulk of their time in the UNIT procedure; this is a consequence of the fact that they must search the theory T for unit clauses. Moskewicz et al.'s (2001) introduction of so-called *watched literals* in ZCHAFF sped this process considerably, but it continues to consume the bulk of the computational resources in implemented systems.

Eventually, unit propagation either terminates without finding a contradiction (presumably because no unit clauses remain in T), or returns a clause $c \in T$ that is falsified by the current justified partial assignment P. In the latter case, the satisfiability engine needs

to respond to this new information and modify P in some way. DPLL, for example, would modify P simply by returning to the most recent choice point and then resuming the search.

CDCL implementations work differently. They take P and c and produce a new clause that is entailed by T and that captures, as best possible, the reason for what went wrong.

As an example, suppose that P gives $\alpha = a \vee b \vee c$ as the reason for c (so that P contains $\neg a$, $\neg b$ and therefore c), and that $\beta = a \vee \neg c$ appears in T. This clause is falsified by P. We can now resolve α and β to get the new clause $a \vee b$, which is also falsified by P and suggests (correctly) that either a or b is the source of the problem even if many other variable choices appear subsequently in P itself. This may enable a much deeper backtrack than simply returning to the most recent choice point.

Definition 2.9 *Let P be a justified partial assignment, and c a clause falsified by P. By a* (P, c)-conflict *we will mean any clause γ that is a logical consequence of c together with the reasons in P and that is also falsified by P, so that $c \wedge R(P) \models \gamma$ and $P \models \neg\gamma$.*

As shown above, once unit propagation identifies a contradiction, we can use resolution to derive a (P, c)-conflict where P is the current partial assignment and c is the conflicting clause in T.

But we can also sometimes do more. Consider the following theory:

$$\neg a \vee b \tag{1}$$

$$\neg b \vee \neg e \vee c \tag{2}$$

$$\neg a \vee \neg e \vee d \tag{3}$$

$$\neg d \vee \neg f \vee \neg c \tag{4}$$

and imagine that after e and f are true, we choose to make a true as well. Now unit propagation allows us to conclude b and then c from b (and e), and also d and then $\neg c$ from d (and e and f). Now a single resolution of (2) (the reason for c) and (4) (the reason for $\neg c$) will allow us to conclude

$$\neg b \vee \neg d \vee \neg e \vee \neg f \tag{5}$$

But we can we go further, resolving (5) with the reason (1) for b to get $\neg a \vee \neg d \vee \neg e \vee \neg f$ and then also resolving with the reason (3) for d to get simply $\neg a \vee \neg e \vee \neg f$. This last conclusion correctly identifies a itself as the source of the problem once e and f are both true.

We see from this example that a single (P, c) pair may have many conflicts and we will generally need to identify a single such conflict to continue the search.

In general, we will want to select a conflict that has a single literal at its assertion level. That will allow the algorithm to redraw the conclusion in question after backtracking to the backtrack level of the conflict. Continuing with our example, suppose that the levels of the choice literals involved were 1 for e and f and 4 for the choice literal a. As we explained, all of the following are possible learned clauses in this case:

clause	**assertion level**	**literals at assertion level**	**backtrack level**
$\neg b \vee \neg d \vee \neg e \vee \neg f$	4	$\neg b, \neg d$	1
$\neg a \vee \neg d \vee \neg e \vee \neg f$	4	$\neg a, \neg d$	1
$\neg a \vee \neg e \vee \neg f$	4	$\neg a$	1

It is only if we learn the last of these clauses that we will be able to draw a unit conclusion ($\neg a$) after backtracking to level one.

Definition 2.10 *Given a justified partial assignment P and a clause γ, let n be the assertion level of γ. Then γ will be called a* unique implication point*, or* UIP*, if it has a unique literal whose assertion level is n.*

UIP's were introduced in GRASP (Marques-Silva & Sakallah, 1999), and experience has shown that part of the strategy for selecting a conflict clause should be to always select a UIP. This can be done by resolving away reasons for any multiple literals at the assertion level of γ.

We can now describe a CDCL-based inference engine. As we do so, recall from Definition 2.1 that a *bias* is a complete assignment of values to variables. If B is a bias and v a variable, we will denote by $B(v)$ the value specified by the bias for v, so that $B(v) \in B$ and $B(v) = v$ or $B(v) = \neg v$. We also define:

Definition 2.11 *For a bias B and set of literals S, we define the result of* restricting B to S*, to be denoted $B|_S$, to be $B|_S = B \cup S - \{\neg l | l \in S\}$.*

Informally, the result of restricting a bias is to label all of the literals in S as true, removing any negations that appeared in the bias before the restriction was performed.

The bias is intended to capture our current guess as to the most likely values in a model of T. So when we extend our partial assignment to take a value at a new variable, we choose the variable and then use the value suggested by the bias. A specific example of this idea was introduced by Pipatsrisawat and Darwiche (2007) in RSAT and was called *phase saving* in that work.[3]

3. A similar idea was introduced slightly earlier by Selmann and Ansótegul (2006); earlier still, Beck (2005) had introduced the idea of saving *some* preferred values to use in the next portion of the search.

Procedure 2.12 (CDCL) *Let T be a theory, Γ a set of clauses such that $T \models \Gamma$, and B a bias for T. Then to compute* CDCL(T, Γ, B, n), *one of:* $\bot$ *if T is shown to be unsatisfiable, a model P of T if one is found, or* UNKNOWN *if no solution is found after n steps:*

1 $i \leftarrow 0$;
2 **while** $i < n$ **do**
3 | $(P, c) \leftarrow$ UNIT$(T \cup \Gamma, P)$;
4 | **while** $c =$ `true` **do**
5 | | **if** P *assigns a value to every variable in* T **then return** P;
6 | | $v \leftarrow$ *a variable unvalued by* P; `// choice`
7 | | $(P, c) \leftarrow$ UNIT$(T \cup \Gamma, \langle P, (B(v), \bot) \rangle)$;
8 | $\gamma \leftarrow$ *any* (P, c)*-conflict that is a* UIP; `// choice`
9 | $\Gamma \leftarrow \Gamma \cup \{\gamma\}$;
10 | $B \leftarrow B|_P$;
11 | **if** $\bot \in \Gamma$ **then return** $\bot$;
12 | $P \leftarrow$ BACKTRACK(P, γ);
13 | $\Gamma \leftarrow \Gamma -$ DISCARD(Γ, P); `// choice`
14 | $i \leftarrow i + 1$
15 **return** UNKNOWN

This algorithm is hardly original with us, but let us give an explanation in any event. As the algorithm proceeds, P is the justified partial assignment being considered and Γ is the collection of learned clauses.

The algorithm works by extending the partial assignment P until either a solution is found (line 5) or a new clause is learned (line 7 with $c \neq$ `true`). In the latter case, we identify a new conflict-driven UIP γ (line 8), add it to Γ, backtrack until γ is unit, and repeat. We update the bias as the search proceeds; we present in line 10 the RSAT choice of updating the bias every time a variable is set by unit propagation. As we will see in Section 3, other choices are also possible.[4]

After the new clause is discovered and added to Γ on line 9, we discard some elements of Γ (line 13); it is here that we can potentially shrink Γ so that only a relatively small (and potentially polynomial) number of clauses is retained. Then we either give up if we have encountered n backtracks, or continue searching.

Recall that for an annotated partial assignment P, $R(P)$ is the set of reasons for literals in P. If (P, c) is a value such as is returned by unit propagation, we will abuse notation somewhat and use $R(P, c)$ to denote simply $R(P)$. We now have:

Lemma 2.13 *Suppose that* $R($UNIT$(T \cup \Gamma, P)) \cap$ DISCARD$(\Gamma) = \emptyset$ *for all* Γ*, so that we never discard reasons or unit clauses that are about to become reasons. Then throughout the execution of Procedure 2.12, we will have* $T \models \Gamma$ *and* P *will be a justified partial assignment of* $T \cup \Gamma$*. In addition:*

4. The attentive reader may wonder why the test on line 11 does not appear until after γ is added to Γ on line 9 and the bias is updated on line 10. The reason is that FLEX may perform additional inference at this point.

1. *If Procedure 2.12 returns $\perp$, then T is unsatisfiable. If Procedure 2.12 returns P for some partial assignment P, then the literals in P are a model of T.*

2. *For a theory with v variables and $n \geq 2^v$, Procedure 2.12 will not return* UNKNOWN*, independent of the further nature of* DISCARD*.*

Proof. We first show the invariants mentioned above. That P is a justified partial assignment follows from the fact that it is returned by UNIT; any justified partial assignment will obviously remain so during a backtrack such as appears on line 12. That $T \models \Gamma$ follows from the fact that T entails any (P, c)-conflict if P is a justified partial assignment and $c \in T$.

Now consider the various points at which Procedure 2.12 returns. On line 5, P is a model of T. And on line 11, unit propagation and resolution have collectively derived $\perp$ as a (P, c)-conflict, so that $T \models \perp$ and T is unsatisfiable.

That Procedure 2.12 does not return UNKNOWN is a bit more subtle. To see this, suppose that we have a partial assignment P. Now we can associate to P a sequence of positive integers $\langle n_0, n_1, \ldots, n_k \rangle$, where n_j is the number of literals in P with assertion level j. We will call this sequence the *size* of P and denote it by $\textbf{size}(P)$.

Now let $n = \langle n_0, n_1, \ldots, n_k \rangle$ and $m = \langle m_0, m_1, \ldots, m_l \rangle$ be two sizes. We will say that m is *smaller* than n, writing $m < n$, if one of the following two conditions holds:

1. There is some j with $m_j \neq n_j$ and, for the first such j, $m_j > n_j$.

2. $m_i = n_i$ for all $i \leq k$, and there is a j with $k < j \leq l$ and $m_j > 1$.

Informally, if the size of a partial assignment gets smaller, we have made "progress" in that we have derived more consequences earlier in the search tree, and the unexplored region of the search space has therefore gotten smaller. We formalize this via the following two claims:

1. Denote by P_i the partial assignment after line 3 in Procedure 2.12 for a given value of i. Then $\textbf{size}(P_{i+1}) < \textbf{size}(P_i)$.

2. If $s_1 > s_2 > \cdots > s_z$ is a properly descending sequence of sizes, then $z \leq 2^v$, where v is the number of variables in the theory in question.

Note that these two results suffice to complete the proof, since it will follow that the loop in Procedure 2.12 can be repeated no more than 2^v times.

For the first claim, note that the new clause γ is unit after the backtrack on line 12. Since it is unit, it will be a reason in UNIT$(T \cup \Gamma, P)$ and therefore cannot be discarded by DISCARD. Now let j be the assertion level of γ. It is immediate that after the unit propagation on line 12, there will be at least one new variable assigned a value at level j.

Now suppose that $\textbf{size}(P_i) = \langle n_0, n_1, \ldots, n_k \rangle$, and $\textbf{size}(P_{i+1}) = \langle m_0, m_1, \ldots, m_j \rangle$. (The number of assertion levels in P_{i+1} is necessarily equal to j.) P_i and P_{i+1} agree before the point of the backtrack, so $m_k = n_k$ for $k < j$. Now if $j \leq k$ (so that the backtrack brought us into P_i), then $m_j > n_j$ and $\textbf{size}(P_{i+1}) < \textbf{size}(P_i)$. If, on the other hand, $j > k$, then the

sizes agree up to k and $m_j > 1$ because the conclusion of γ is a unit consequence at this level. This proves the first claim.

For the second, note first that the ordering conditions are unchanged if we append to the end of any size $\langle n_0, n_1, \ldots, n_k \rangle$ enough 1's so that $\sum_i n_i = v$. This essentially pretends that we have added values for all subsequent variables, but there have been no unit propagations from those variables. If we write $\overline{n}$ for the result of extending a sequence in this way, it is easy to see that $n < m$ if and only if $\overline{n} < \overline{m}$.

Now let n be a size. We claim that any descending sequence of length at least 2^{v-n_0} steps beginning with n and ending in a size m will have $m_0 > n_0$. The proof is by induction on $v - n_0$.

If $v - n_0 = 1$, then we have $n_0 = v - 1$ and must have $n = \langle v-1, 1 \rangle$. But now the only way to make it smaller at all is to have $m = \langle v \rangle$, and $m_0 > n_0$.

For the inductive step, suppose that $n = \langle n_0, 1, \ldots \rangle$. Now after at most 2^{v-n_0-1} steps, we will have gotten to $\langle n_0, 2, \ldots \rangle$ by the inductive hypothesis. After at most 2^{v-n_0-2} steps, we will have gotten to $\langle n_0, 3, \ldots \rangle$, and so on. We will therefore arrive at $\langle n_0, v - n_0 \rangle$ after at most

$$\sum_{i=1}^{v-n_0-1} 2^{v-n_0-i} = 2^{v-n_0} - 2$$

steps. One more step, a total of $2^{v-n_0} - 1$, causes n_0 to increment. 2^{v-n_0} steps thus cause n_0 to increment as well. This completes the induction.

Now imagine that we add a new variable w to our theory, where w appears in no clauses but is evaluated first when the search begins. For our new theory with $v+1$ variables, given that the size of any partial assignment must have its first component incremented after at most $2^{(v+1)-1}$ steps, we will either have learned something about w (an impossibility) or derived the empty clause $\perp$. Thus the second claim follows and the lemma is proved.

This proof is very similar in spirit to the proof of Theorem 3.10, our main theoretical result. ■

In general, Lemma 2.13 appears to be "common knowledge" in the SAT community, although I am unaware of any other published proof of the second claim and it is not completely clear if the SAT community knows it to be true.[5]

Until this point, we have not yet included restarts in our algorithm, but doing so is straightforward. We define:

Definition 2.14 *A* restart policy *is a mapping $r : \mathbb{N} \to \mathbb{N}$, where $\mathbb{N}$ is the set of positive integers.*

The restart policy simply gives the number of backtracks permitted as a function of probe.

5. A similar result, with a similar proof but based on the assumption that no clauses are ever deleted, does appear in Section 4.2 of Zhang's (2003) Ph.D. thesis.

Procedure 2.15 (CDCL with restarts) *Let T be a theory and r a restart policy. Then to compute* $\textsc{sat}(T, n)$*, either the empty clause $\bot$ if T is unsatisfiable or a model of T:*

```
1 Γ ← ∅;
2 B ← any bias for T;                                   // choice
3 i ← 1;
4 while true do
5 |  x ← CDCL(T, Γ, B, r(i));
6 |  if x ≠ UNKNOWN then return x;
7 |  i ← i + 1;
```

2.1 Theoretical Results

We can now show the following:

Proposition 2.16 *Let T be a theory involving v variables, and r a restart policy. Then if Procedure 2.15 returns $\bot$, T is unsatisfiable. If Procedure 2.15 returns a partial assignment P, then the literals in P are a model of T. In addition:*

1. *If $R(\textsc{unit}(T \cup \Gamma, P)) \cap \textsc{discard}(\Gamma, P) = \emptyset$ and if there is some i such that $r(i) \geq 2^v$, then Procedure 2.15 will always terminate.*

2. *If $\textsc{discard}(\Gamma, P) = \Gamma - R(\textsc{unit}(T \cup \Gamma, P))$, then $|\Gamma| \leq v$ as Procedure 2.15 is executed.*

3. *If $\textsc{discard}(\Gamma, P) = \emptyset$, then the size of Γ may grow to be exponential in v, independent of the restart policy r.*

Proof. The proof of Lemma 2.13 also shows that Procedure 2.15 is correct.

Claim 1 of Proposition 2.16 follows immediately from claim 2 of Lemma 2.13. Claim 2 follows from the fact that each variable can have at most one reason in $R(P)$. Claim 3 follows from the fact that the amount of memory used by Procedure 2.15 is linear in the run time if no clauses are ever discarded, and there are known to be many problems for which the shortest resolution proof is of exponential length (Bonet, Pitassi, & Raz, 1997; Haken, 1985, 1995; Krajiček, 1997; Pudlak, 1997; Tseitin, 1970). ■

Corollary 2.17 *There are weakly systematic, polyspace instances of Procedure 2.15. There are strongly systematic, exp-space instances of Procedure 2.15.*

Proof. Take $\textsc{discard}(\Gamma, P) = \Gamma - R(\textsc{unit}(T \cup \Gamma, P))$ to get a weakly systematic polyspace instance by virtue of the first two claims of Proposition 2.16. If $\textsc{discard}(\Gamma) = \emptyset$, the resulting instance is strongly systematic because any instantiation will eventually run out of new clauses to learn, and may use an exponential amount of memory because of the third claim of Proposition 2.16. ■

Tinisat is an exp-space, strongly systematic instance. Minisat is a (potentially) polyspace, weakly systematic instance.

2.2 Practical Concerns

In addition to the selection of restart policy, there are three further points where an explicit choice must be made in Procedure 2.12, and one where a choice must be made in Procedure 2.15. In line 6 of Procedure 2.12, a variable is selected for addition to P. In line 8, a particular conflict clause is selected for addition to Γ. In line 13, the set Γ is reduced, presumably in order to conserve memory (and thereby speed unit propagation). And in line 2 of Procedure 2.15, an initial bias is selected.

Given that much work on SAT currently focuses on one implementation of Procedure 2.15 (and therefore of Procedure 2.12) or another, there is a considerable volume of literature on each of these choices. We will summarize that work here, providing additional details in areas where our ideas will constrain the allowed options.

Variable and bias selection Variables are generally selected in a CDCL prover using a heuristic called VSIDS ("variable state independent decaying sum") that was introduced in ZCHAFF (Moskewicz et al., 2001). We continue to use VSIDS in our work and do not describe it further here.

We also need to specify an initial bias, before search or choice has valued any of the variables. There is no real reason to prefer one initial bias over another, especially given that the values are likely to be overwritten as the search moves forward. We will follow MINISAT's example and set every variable to false in our initial bias.[6]

Conflict clause selection We have already remarked both that a particular (P, c) pair may have many associated conflicts, and that it is generally wise to select one that is a UIP. That will allow the algorithm to redraw the conclusion in question after backtracking to the backtrack level of the conflict, and is central to the proof of Lemma 2.13.

But we can sometimes do more. Recall that in our running example, we derived the UIP $\neg a \vee \neg e \vee \neg f$. Suppose also that the actual choice literal at level one was g, with e and f following from $\neg g \vee e$ and $\neg g \vee f$ respectively. Now we can continue the resolution process, using

$$\neg g \vee \neg a$$

as the learned clause instead of $\neg a \vee \neg e \vee \neg f$. Should we do so?

Definition 2.18 *A* UIP γ *will be called a* decision UIP, *or* DUIP, *if* $C(P) \models \neg\gamma$, *so that every literal in* γ *is the negation of a choice in* P.

Proposition 2.19 *Let* P *be a justified partial assignment and* γ *a clause falsified by* P. *Then* (P, γ) *has a unique* DUIP.

Proof. The only way to construct a DUIP is to resolve literals away until only decisions remain. It is not hard to see that the set of decisions so obtained is independent of the order of such resolutions. ∎

Should we always work with a DUIP as our algorithm proceeds? Surprisingly, the answer appears to be no; the extra resolutions seem to move the conflict away from the "real problem" more often than not. But one thing we should do is to remove any literals that

6. On naturally occurring problems, the effectiveness of biasing every variable to be false is perhaps rooted in the so-called *closed world assumption* (Clark, 1978; Reiter, 1978), but that will not concern us here.

are simply unnecessary. So if in our running example the reason for f is $\neg e \vee f$, it is obviously wise to resolve this into the conflict clause being learned, replacing $\neg a \vee \neg e \vee \neg f$ with the properly stronger $\neg a \vee \neg e$.

Zhang et al. define a *first* UIP as follows.[7] The basic idea is that we want to define a "first" UIP as a UIP that is both minimal and as close to the eventual conflict as possible.

Definition 2.20 *For an annotated partial assignment P and conflict c, a (P, c)-conflict γ that is a* UIP *will be called* reduced *if no proper subset of γ is a (P, c)-conflict.*

By "close to the eventual conflict," Zhang et al. actually mean that the UIP in question is as *late* in the conflict graph as possible:

Definition 2.21 *Let P be an annotated partial assignment and suppose that γ_1 and γ_2 are clauses falsified by P. We will write $\gamma_1 <_P \gamma_2$ if every literal in γ_1 either appears in γ_2 or has a reason that includes at least one literal in γ_2. A (P, c)-conflict that is reduced and minimal under the transitive closure of $<_P$ will be called a* first UIP*, or* FUIP*.*

As remarked above, $\gamma_1 <_P \gamma_2$ if γ_1 is closer to the eventual contradiction (and therefore later in the implication graph) than is γ_2.

Proposition 2.22 (Zhang, Madigan, & Moskewicz, 2001) *Let P be a justified partial assignment and γ a clause falsified by P. Then (P, γ) has a unique* FUIP*.*

The basic idea of the proof is that we can construct an FUIP by resolving just enough to ensure that we are working with a UIP and then removing any literals that can be subsumed by other literals in γ. Additional resolutions are not needed and produce new clauses that are not minimal under $<_P$. ■

CDCL-based provers can work with any UIP, but performance is generally best when FUIP's are used (Zhang et al., 2001).

The learned clauses interact with the bias; in general, we want to learn a clause that is in conflict with the bias (and therefore encourages us to adjust the bias in a useful way).

Definition 2.23 *Given a bias B, a* UIP *γ will be called a* bias UIP for *B, or* BUIP *for B, if $B \models \neg\gamma$. A* BUIP *that is minimal under the transitive closure of $<_p$ as in Definition 2.21 will be called a* first BUIP*, or* FBUIP*.*

In the case where the bias B contains negations of all of the literals in P (as in Procedure 2.12 as written), any UIP will be a BUIP, since the literals in P will always conflict with the values in B. So in this case, the FUIP will be an FBUIP as well.

We also have:

Definition 2.24 *Let P be an annotated partial assignment and B a bias. We will say that P is* supported by *B if $C(P) \subseteq B$, so that every choice literal in P appears in B.*

7. Zhang's description of an FUIP is a bit less formal; Definitions 2.20 and 2.21 are simply a formalization of their ideas.

In other words, a bias supports an annotated partial assignment if all of the choices made in P were as suggested by the bias.

Lemma 2.25 *Let P be a justified partial assignment and c a clause falsified by P, and suppose that B is a bias that supports P. Then any* DUIP *for (P, c) is also a* BUIP. ■

It follows from this and Proposition 2.19 that if the bias supports the annotated partial assignment, it will have at least one BUIP, and therefore at least one FBUIP. It is not difficult to see that this FBUIP is unique (basically, we stop resolving as soon as the necessary conditions are met). FBUIP's share the crucial property that they modify the learned clause γ minimally subject to the restriction that γ contradicts the bias and is a UIP.

Discard strategy Clause reduction generally proceeds by estimating the likely future value of each clause in Γ in the subsequent search. The least valuable clauses are periodically removed as the search proceeds. We will be free to follow an identical strategy, although some $\gamma \in \Gamma$ will be marked as essential to preserving systematicity. Those clauses will need to be retained independent of their estimated future value. If our procedure is to use polynomial amounts of memory, the number of such clauses must be guaranteed to grow at most polynomially with the size of the problem.

3. FLEX

Consider now the search for a polyspace, strongly systematic satisfiability algorithm. We immediately have:

Observation 3.1 *Strongly systematic, polyspace satisfiability algorithms exist.*

The reason for this is obvious. Imagine any implementation of CDCL with restarts, and suppose that we maintain, from one restart to another, a separate partial assignment P_D as constructed by DPLL. This additional data structure requires polynomial memory (at most v clauses, each of which is of size at most v), and if we simply incorporate the clauses in P_D into Γ, the resulting algorithm is clearly systematic.

In practice, however, this is of essentially no value. The systematicity guaranteed by DPLL is completely "separate" from the inference going on in the individual probes, so it is unlikely that the additional clauses in Γ will do much to constrain the search. In addition, most restart policies restart relatively infrequently, so while a guarantee that no more than 2^v restarts will be required in satisfying from a theoretical perspective, it may be of little or no value in practice.

3.1 Pure FLEX

Far more interesting would be to find an instance of Procedure 2.15 itself that was strongly systematic. Showing that to be possible is our main theoretical result in this paper.

The basic idea behind our method is that we will manage the set of learned clauses so as to retain enough information to ensure that the solver is making progress in a lexicographic sense, where we identify some variables as being lexicographically earlier than others. The lexicographic ordering is quite weak, and is weakened further when possible as the search

proceeds. So, for example, if we make progress by eliminating one possible value for the variable a, then we can drop any restrictions on the relative ordering of variables that follow a (although we do need to remember that they do indeed follow a).

To get a somewhat better feel for this, imagine that we have ordered the variables in our theory in some way, so that the m variables are ordered as $v_1 < v_2 < \cdots < v_m$.

We will now view it as "progress" if we eliminate a value for the variable v_i without introducing new possible values for any v_j with $j < i$. If we can repeatedly do this, we will eventually manage to eliminate a value for v_1, and that value will remain eliminated as the search proceeds. Then we will eventually eliminate one of the two values for v_2, and so on. We will say that we are making "lexicographic progress" when this happens.

In fact, we don't need to maintain a complete ordering on the variables in question. When we make progress by eliminating a value for v_i, all we really need to do is to remember which variables precede v_i in the ordering (since their eliminated values must remain eliminated). Variables following v_i in the ordering can be reordered if the subsequent search suggests doing so.

To formalize this, we maintain a partial order $\leq$ on the variables appearing in the theory as Procedure 2.12 is executed.[8] When we learn something new about a particular variable x, we should, as described above, be free to discard information regarding the relative ordering of variables that follow x in the partial order. Since we have eliminated a value for the variable x, this further ordering information is no longer needed. We formalize this as follows:

Definition 3.2 *If $\leq$ is a partial order and x is a point, we define the* weakening *of $\leq$ at x, to be denoted $\leq_x$, to be the partial order given by $y \leq_x z$ if and only if $y \leq z$ and either $y = z$ or $x \not< y$.*

In other words, the weakened partial order is the same as the original but "forgets" that $y < z$ if $x < y < z$, so that both y and z follow x in the partial order.

Proposition 3.3 *$\leq_x$ is a partial order.*

Proof. Reflexivity is immediate. For transitivity, suppose that $u \leq_x v \leq_x w$. Now if $u = v$, $u \leq_x w$ since $v \leq_x w$. If $u \neq v$, then $x \not< u$ since $u \leq_x v$. But we also have $u \leq v \leq w$ so $u \leq w$. Thus $u \leq_x w$. Anti-symmetry of $\leq_x$ follows immediately from the anti-symmetry of $\leq$. ∎

As a specific example, if the initial partial order $<$ is given by $a < b < c < d$, then weakening this partial order at b produces $a <_b b$, $b <_b c$ and $b <_b d$. c and d are incomparable under $\leq_b$, since they both followed b under the original partial order.

Now imagine a justified partial assignment P. If (l, γ) is a pair in P, it is reasonable to think of γ as "implying" l. In other words, we think of the disjunction $\gamma = \bigvee_j l_j$ as

$$\neg l_1 \wedge \cdots \wedge \neg l_{i-1} \wedge \neg l_{i+1} \wedge \cdots \wedge \neg l_k \rightarrow l_i$$

8. We note in passing that it is possible, with some difficulty, to modify all of our work to deal with the case where $\leq$ is a preorder instead of a partial order. This provides some additional flexibility in the manipulation of the bias, but it is not obvious that that additional flexibility is of practical value. We discuss only the partial order case in the interest of maintaining the clarity of the exposition.

where $l_i = l$ is the literal in the (l, γ) pair. While we do not retain the partial assignment from one search probe to another, we will retain the partial ordering $\leq$ and use it to split the antecedent from the conclusion in any particular clause. As a specific example, if c follows a and b in the ordering, so $a \leq c$ and $b \leq c$, the clause $\neg a \vee \neg b \vee c$ will be interpreted as $a \wedge b \rightarrow c$.

In general, if γ is a clause and $\leq$ a partial order, we will think of the $\leq$-maximal elements of γ as its conclusion, and the rest of γ as its antecedent:

Definition 3.4 *Let γ be a clause and $\leq$ a partial order on the variables it contains. The* conclusion *of γ, to be denoted $\gamma^{\leq}$, is the disjunction*

$$\gamma^{\leq} = \{l \in \gamma | \nexists m \in \gamma, m > l\}$$

The antecedent *of γ, to be denoted $\gamma_{\leq}$, is $\gamma_{\leq} = \neg(\gamma - \gamma^{\leq})$.*

The conclusion of the learned clause γ is the disjunction of the maximal literals in γ, and the antecedent is the negation of what's left.

Again, an example may help. Suppose that $\gamma = a \vee b \vee c \vee d$ and the partial order has $a \leq c$ and $b \leq d$. Now c and d are the maximal elements of γ, so the conclusion $\gamma^{\leq}$ is the disjunction $c \vee d$. The antecedent is

$$\begin{aligned} \gamma_{\leq} &= \neg(\gamma - \gamma^{\leq}) \\ &= \neg(\{a, b, c, d\} - \{c, d\}) \\ &= \neg\{a, b\} \\ &= \neg(a \vee b) \\ &= \neg a \wedge \neg b \end{aligned}$$

as one would expect. The learned clause has effectively been rewritten as

$$\neg a \wedge \neg b \rightarrow c \vee d$$

We have been cavalier in our use of notation, switching freely between representing γ as a disjunction and as a set.

Definition 3.5 *If Γ is a set of learned clauses and $\leq$ a partial order, we will denote by $\Gamma_{\leq}$ the conjunction of the antecedents $\gamma_{\leq}$ for all $\gamma \in \Gamma$. We will denote by $\Gamma^{\leq}$ the conjunction of the conclusions $\gamma^{\leq}$ for all $\gamma \in \Gamma$.*

We are finally in a position to describe the procedure by which we add a new clause γ to Γ. In Procedure 2.12, this was via the simple $\Gamma \leftarrow \Gamma \cup \{\gamma\}$ on line 9, but here it is a bit more complex. The addition of γ to Γ can also both modify the partial order $\leq$ and replace γ with a lexicographically stronger clause in some circumstances:

Procedure 3.6 *To compute* ADD$(\gamma, \Gamma, \leq)$*:*

1 **if** $\gamma = \bot$ **then return** $(\gamma, \Gamma \cup \{\bot\}, \leq)$;
2 *select* $l \in \gamma^{\leq}$; `// l is the intended conclusion`
3 **if** *there is a* $\rho \in \Gamma$ *with* $\rho^{\leq} = \neg l$ **then**
4 | **return** ADD(RESOLVE$(\gamma, \rho), \Gamma, \leq)$
5 *add* $x \leq l$ *to* $\leq$ *for each* $x \in \gamma$;
6 $\leq \leftarrow \leq_l$;
7 *remove from* Γ *any* c *with either* $c_{\leq} \cap \{l, \neg l\} \neq \emptyset$ *or both* $c \cap \{x | l < x\} \neq \emptyset$ *and* $|c^{\leq}| > 1$;
8 *return* $(\gamma, \Gamma \cup \{\gamma\}, \leq)$

Line 1 handles the edge case where we have actually derived a contradiction. Line 4 calls for a resolution if both l and $\neg l$ appear as conclusions of clauses in Γ, and line 7 says that clauses should be deleted from Γ if they include either l or $\neg l$ in their antecedent, or if they have ambiguous conclusions based on variables that follow l in the partial order. This is in keeping with our overall "lexicographic" approach: Once we make progress on a particular literal l, we can forget everything about following literals in the partial order, removing both their relative orderings (as per the weakening on line 6 of Procedure 3.6) and clauses that contain them (as per the removal on line 7).

Proposition 3.7 *Suppose that* Γ *contains* n *clauses involving* v *variables. Then Procedure 3.6 can be executed in* $O(v^3 + nv^2)$ *time.*

Proof. Except for the recursion, the most expensive steps are line 6, which can potentially take time $O(v^2)$ and line 7, which can take time $O(nv)$. This assumes that we can determine whether $x < y$ in time $O(1)$, but whatever the data structure used to retain the partial order, we can always compute the entire partial order at each iteration of the loop, which will take time at most $O(v^2)$. The total time taken without the recursion is thus bounded by $O(v^2 + nv)$.

The maximum number of recursive calls is $O(v)$, since each variable will be resolved upon at most once. ■

Just as the $o(v^2)$ unit propagation procedure remains practical on problems containing millions of variables, so we expect the ADD procedure 3.6 to do so as well. (And for roughly the same reasons: Clauses are not of length v, literals generally appear in only a relative handful of clauses, and so on.)

Procedure 3.6 is the primary difference between a standard CDCL algorithm and FLEX; we will replace the act of adding γ to Γ with the more involved Procedure 3.6. When we do so, note that Procedure 3.6 may actually derive the empty clause in line 4; we then return a theory containing the empty clause on line 1, leading to a failure on line 11 of Procedure 2.12. We have:

Procedure 3.8 (Pure FLEX) *Let T be a theory, Γ a set of clauses such that $T \models \Gamma$, $\leq$ a partial order on the variables in T, and B a bias for T. Then to compute* FLEX$(T, \Gamma, \leq, B, n)$, *one of: $\bot$ if T is shown to be unsatisfiable, a model P of T if one is found, or* UNKNOWN *if no solution is found after n steps:*

1 $i \leftarrow 0$;
2 **while** $i < n$ **do**
3 $\quad (P, c) \leftarrow$ UNIT$(T \cup \Gamma, P)$;
4 $\quad$ **while** $c =$ `true` **do**
5 $\quad\quad$ **if** P *assigns a value to every variable in* T **then return** P;
6 $\quad\quad v \leftarrow$ *a variable unvalued by* P;
7 $\quad\quad (P, c) \leftarrow$ UNIT$(T \cup \Gamma, \langle P, (B(v), \bot)\rangle)$;
8 $\quad \gamma \leftarrow$ *any* (P, c)*-conflict that is a* UIP *with* $B \models \neg\gamma$;
9 $\quad (\gamma, \Gamma, \leq) \leftarrow$ ADD$(\gamma, \Gamma, \leq)$;
10 $\quad B \leftarrow B|_{\gamma\leq}$;
11 $\quad$ **if** $\bot \in \Gamma$ **then return** $\bot$;
12 $\quad P \leftarrow$ BACKTRACK(P, γ);
13 $\quad i \leftarrow i + 1$
14 **return** UNKNOWN

Note that we require the selection of a BUIP on line 8 of Procedure 3.8 (since we require $B \models \neg\gamma$), and that we adjust the bias only to cater to the conclusion of the new clause γ on line 10. Both of these observations will be important as we discuss practical implementation of our methods.

The only modification needed to Procedure 2.15 is to initialize and then use the partial order $\leq$:

Procedure 3.9 (Pure FLEX with restarts) *Let T be a theory and r a restart policy. Then to compute* SAT(T, n)*, either the empty clause $\bot$ if T is unsatisfiable or a model of T:*

1 $\Gamma \leftarrow \emptyset$;
2 $\leq \leftarrow \emptyset$;
3 $B \leftarrow$ *any bias for* T;
4 $i \leftarrow 1$;
5 **while** *true* **do**
6 $\quad x \leftarrow$ FLEX$(T, \Gamma, \leq, B, r(i))$;
7 $\quad$ **if** $x \neq$ UNKNOWN **then return** x;
8 $\quad i \leftarrow i + 1$;

We finally have:

Theorem 3.10 *Suppose that T is a theory with n clauses involving v variables. Then:*

1. *Each loop of Procedure 3.8 can be executed in time $O(v^3 + nv^2)$,*

2. *$|\Gamma| \leq v$ as Procedure 3.9 is executed, and*

3. *For any n with $\sum_{i=0}^{n} r(i) \geq 2^v$, Procedure 3.9 will finish before completing iteration n.*

Corollary 3.11 *Procedure 3.9 is strongly systematic and polyspace.* ■

The proof of Theorem 3.10 is lengthy and appears in the appendix.

Note that our contribution here is not to show that it is possible to ensure systematicity by retaining a sufficient number of learned clauses. TINISAT has already shown this, and Lecoutre et al. have shown (2007) that retaining a single nogood per restart can suffice. But in this other work, the number of restarts grows polynomially with the number of backtracks, and therefore potentially exponentially with problem size. All of these earlier results thus may require keeping an exponential number of learned clauses; our contribution is to show that a *polynomial* number of nogoods can suffice.

3.2 Hybrid FLEX

While Procedures 3.8 and 3.9 achieve the basic theoretical goal set in this paper, a direct implementation of these procedures as written encounters two practical difficulties.

First, the ADD procedure is expensive. Executing it at every backtrack introduces an overall computational burden greater than that of unit propagation.

Second, the search in Procedure 3.8 is guided by the bias B, and that bias is modified not to match every unit propagation, but only to match the conclusions of newly learned clauses. Experimentation shows that the RSAT bias, matching as it does the result of every unit propagation, is more effective in practice. We cannot value newly selected variables as in RSAT, since if the bias doesn't support the nogood, there may be no BUIP on line 8 of Procedure 3.8. In fact, experimentation on number factoring problems shows that approximately 40% of the time, *none* of the clauses learned in any particular probe can be resolved to a BUIP.

Nevertheless, there is a straightforward way to deal with these issues and to still preserve the intuition underlying our approach. In each probe, we follow our bias only until a new clause is learned. That new clause will have a BUIP (because the bias was followed in constructing it), and we then incorporate that new clause into Γ as usual. We then complete the probe using the RSAT bias. By doing this, we follow the RSAT bias almost all of the time and only call ADD once per probe, eliminating the substantial computational overhead that this procedure might introduce.

We will refer to this compound procedure as *hybrid* FLEX, or simply FLEX. In order to implement it, we need to maintain two biases, which we refer to as B (for the FLEX component) and R (for the RSAT component), and two sets of learned clauses, which we refer to as Γ (for the FLEX component) and Φ (for the conventional component). We finally have:

Procedure 3.12 (FLEX) *Let T be a theory, Γ and Φ sets of clauses such that $T \models \Gamma \wedge \Phi$, $\leq$ a partial order on the variables in T, and B and R biases for T. Then to compute* $\text{FLEX}(T, \Gamma, \leq, B, \Phi, R, n)$*, one of: $\perp$ if T is shown to be unsatisfiable, a model P of T if one is found, or* UNKNOWN *if no solution is found after n steps:*

1. $i \leftarrow 0$;
2. **while** $i < n$ **do**
3. $\quad (P, c) \leftarrow \text{UNIT}(T \cup \Gamma \cup \Phi, P)$;
4. $\quad$ **while** $c = \texttt{true}$ **do**
5. $\quad\quad$ **if** P *assigns a value to every variable in* T **then return** P;
6. $\quad\quad v \leftarrow$ *a variable unvalued by* P;
7. $\quad\quad$ **if** $i = 0$ **then** $l \leftarrow B(v)$;
8. $\quad\quad$ **else** $l \leftarrow R(v)$;
9. $\quad\quad (P, c) \leftarrow \text{UNIT}(T \cup \Gamma \cup \Phi, \langle P, (l, \perp)\rangle)$;
10. $\quad$ **if** $i = 0$ **then**
11. $\quad\quad \gamma \leftarrow$ *any* (P, c)*-conflict that is a* UIP *with* $B \models \neg\gamma$;
12. $\quad\quad (\gamma, \Gamma, \leq) \leftarrow \text{ADD}(\gamma, \Gamma, \leq)$;
13. $\quad\quad B \leftarrow B|_{\gamma\leq}$
14. $\quad$ **else**
    $\quad\quad \gamma \leftarrow$ *any* (P, c)*-conflict that is a* UIP;
15. $\quad\quad \Phi \leftarrow \Phi \cup \{\gamma\}$;
16. $\quad\quad R \leftarrow R|_P$
17. $\quad$ **if** $\perp \in \Gamma \cup \Phi$ **then return** $\perp$;
18. $\quad P \leftarrow \text{BACKTRACK}(P, \gamma)$;
19. $\quad \Phi \leftarrow \Phi - \text{DISCARD}(\Phi)$;
20. $\quad i \leftarrow i + 1$
21. **return** UNKNOWN

Procedure 3.13 (FLEX with restarts) *Let T be a theory and r a restart policy. Then to compute* $\text{SAT}(T, n)$*, either the empty clause $\perp$ if T is unsatisfiable or a model of T:*

1. $\Gamma \leftarrow \Phi \leftarrow \emptyset$;
2. $\leq \leftarrow \emptyset$;
3. $B \leftarrow R \leftarrow$ *any bias for* T;
4. $i \leftarrow 1$;
5. **while** *true* **do**
6. $\quad x \leftarrow \text{FLEX}(T, \Gamma, \leq, B, \Phi, R, r(i))$;
7. $\quad$ **if** $x \neq$ UNKNOWN **then return** x;
8. $\quad i \leftarrow i + 1$;

As in Theorem 3.10 and Corollary 3.11, we have:

Corollary 3.14 *Procedure 3.13 is strongly systematic and polyspace for any* DISCARD *procedure that ensures that Φ is polyspace.* ■

4. Experimental Results

Before we present experimental results based on our methods, let us describe the setting in which these experiments were performed.

First, we expect the value of our results to vary with problem size. We have one major advantage over existing methods, in that the strong systematicity of our approach will cause probes late in the search to examine areas of the search space that were unexplored earlier.

This advantage is also a drawback, however. Especially early in the search, where the probes are small, the rsat bias is known to be extremely effective and we are likely to see a performance degradation from the fact that we cannot follow this bias on the first set of choices in any particular probe.

So for easy problems, we would expect our methods to perform worse than conventional ones because of our inability to begin probes by following the rsat bias. For difficult problems, we would expect our methods to perform better than conventional ones, as we manage to reach new areas of the search space while older methods are reexamining previously explored regions. What we don't know is where the transition will be between these two general problem types.

In order to evaluate this, we used two separate sets of problems in our experiments. The first was the set of 1030 problems in the sat 2013 benchmarks. These problems were solved with a 5000 second cutoff in order to ensure that all processes terminated.

The second set of problems used were number factoring problems (Bebel & Yuen, 2013). Although subexponential algorithms for number factoring are known to exist (Buhler, Lenstra, & Pomerance, 1993, and a great deal of subsequent work), there is no reason to believe that the techniques involved will bear significantly on the sat encodings of such problems.

One advantage of using number factoring is that problems of gradually increasing difficulty can be run to completion in every case, avoiding the fact that a simple timeout is less informative regarding overall computational efficacy than is the time needed to actually solve a problem. Problems of gradually increasing difficulty can be generated by simply increasing the magnitudes of the numbers being factored.

The number factoring problems were generated by using an iterated Miller-Rabin test to generate pairs of primes and then using the Purdom/Sabry (2005) sat generator to produce a sat encoding of the problem of factoring the product of these primes. In order to avoid situations where the factoring problem might be too easy, the prime factors were selected so that the sizes of their binary representations were within three bits of one another. For any given problem size (number of bits in the number being factored), fifty instances were generated randomly.

As a baseline solver, we used minisat 2.2 (Sörensson & Een, 2005). But there were in fact three separate sets of changes that we needed to make to produce flex:

1. First, we converted minisat from C to C++ and changed it to use the C++ standard library in many locations instead of the variety of specialized techniques included in minisat. We will refer to this solver as stlsat.

2. Second, flex can be expected to be more efficient as the number of restarts increases. In minisat and similar solvers, the number of backtracks before restarting is not given

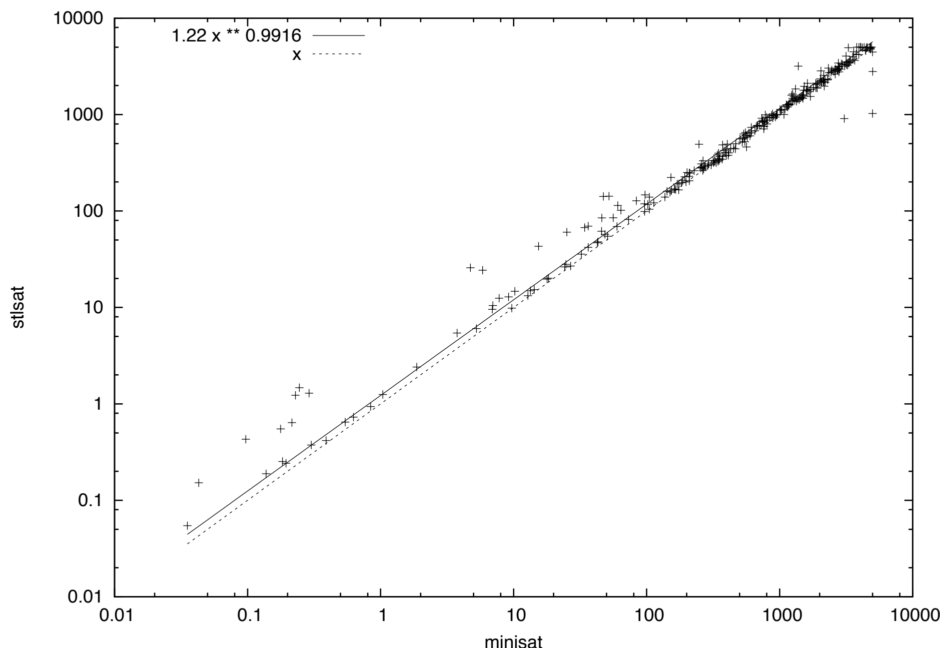


Figure 1: MINISAT vs. STLSAT, SAT competition problems (CPU time in seconds)

as specified by the original Luby policy (Luby et al., 1993), but instead by multiplying the Luby numbers by some constant (generally taken to be 100, so that the first probe involves 100 backtracks). In order to increase the number of probes, we reduced the multiplicative factor to four (so that the first probe involves only four backtracks). We will refer to this solver as STLSAT_4.

3. Finally, we modified STLSAT to produce FLEX, also with an initial probe size of four.

We present results for the SAT 2013 benchmarks first, followed by results for number factoring. We compare MINISAT to STLSAT, STLSAT to STLSAT_4, and then STLSAT_4 to FLEX. All experiments were run on a 24-core Intel Xeon machine running at 2.2GHz with 256K of L2 cache per processor, 24GB of main memory, and 75GB of swap space. 24 problems were solved at a time, and the swap space was needed to deal with some of the larger SAT competition instances, since many were running in parallel.[9]

The software used in these experiments is available as an online appendix to this paper. stlsat.zip contains the source for STLSAT, and flex.zip contains the source for FLEX. (Stlsat is just FLEX with the various extensions removed.)

4.1 Sat Competition Problems

Figure 1 compares MINISAT to STLSAT for the SAT competition problems. The solver that is closer to original MINISAT will always be on the x-axis, with the modification on the y-axis,

9. We did not evaluate the impact of the use of swap on the running times of larger problems, although it obviously might be substantial. In practice, however, once significant amounts of swap were used, none of the systems being evaluated was able to solve the problem in question. In addition, running the problems serially would simply have been impractical given the available hardware.

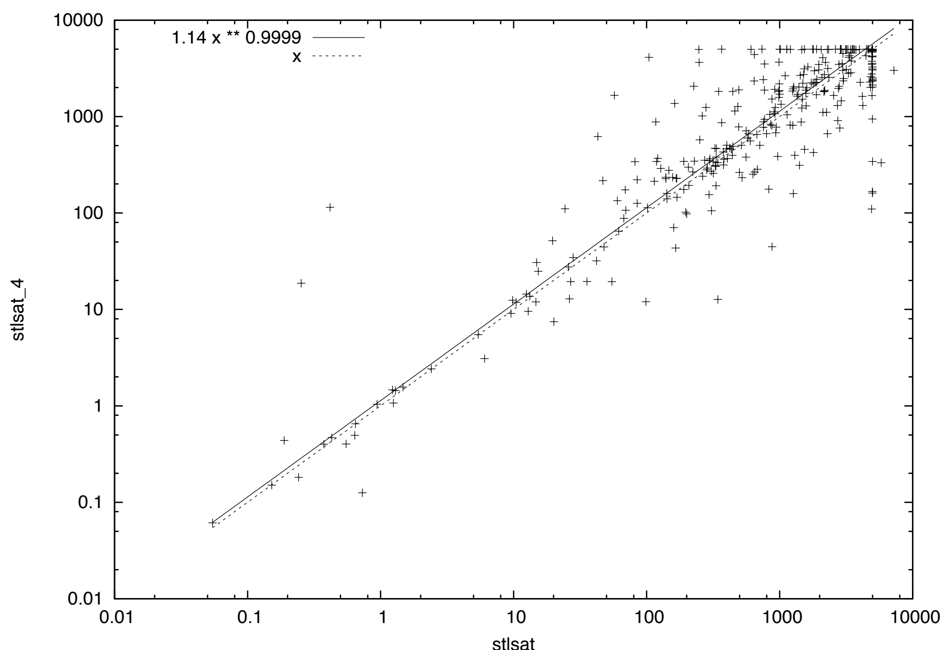


Figure 2: STLSAT vs. STLSAT_4, SAT competition problems (CPU time in seconds)

with both axes plotted using a log scale and a dashed line giving the $y = x$ baseline. One point is plotted for each problem solved by either of the two methods. A point above $x = y$ means that the old solver is better; a point below $x = y$, that the new solver is better.

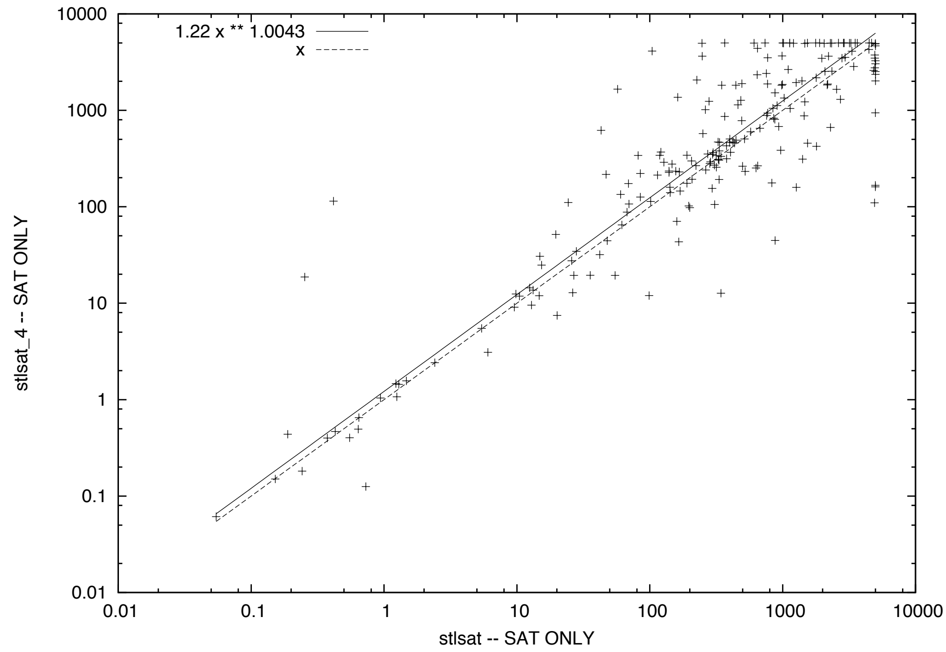
In many cases, we will also give the best polynomial fit of the form $y = ax^b$ for fixed a and b. So in Figure 1, STLSAT is seen to be 22% worse than MINISAT at the outset, but the exponent slightly below one indicates that the disparity is shrinking as the problems become more difficult. (Indeed, this is apparent from looking at the graph itself.)

To find the polynomial fit, we did not use the standard technique of minimizing the squared vertical distance between a point and the line in question. The reason is that if we flipped the axes, we would then be minimizing the squared *horizontal* distance to the line, and a potentially different fit would be found. To ensure that the same fit was found independent of the axis selection, we minimized the sum of the squares of the actual (perpendicular) distances between the data points and the line in question.

In some cases, we will split the data depending on whether the problem instances being solved were or were not satisfiable. This is not an interesting distinction here.

These results are as one might expect. Without the fine-tuned data structures of MINISAT, STLSAT performs slightly worse. This is more significant on easy problems, and the two solvers appear to be virtually identical on more difficult instances.

Figures 2 and 3 show the impact of reducing the number of backtracks per probe. Overall (Figure 2), there is a 14% cost to making the change, and the cost is nearly independent of problem size. Somewhat curiously, however, the cost is significantly different for satisfiable instances (a 22% cost, top of Figure 3) when compared to unsatisfiable ones (a 7% *savings*, bottom of Figure 3).



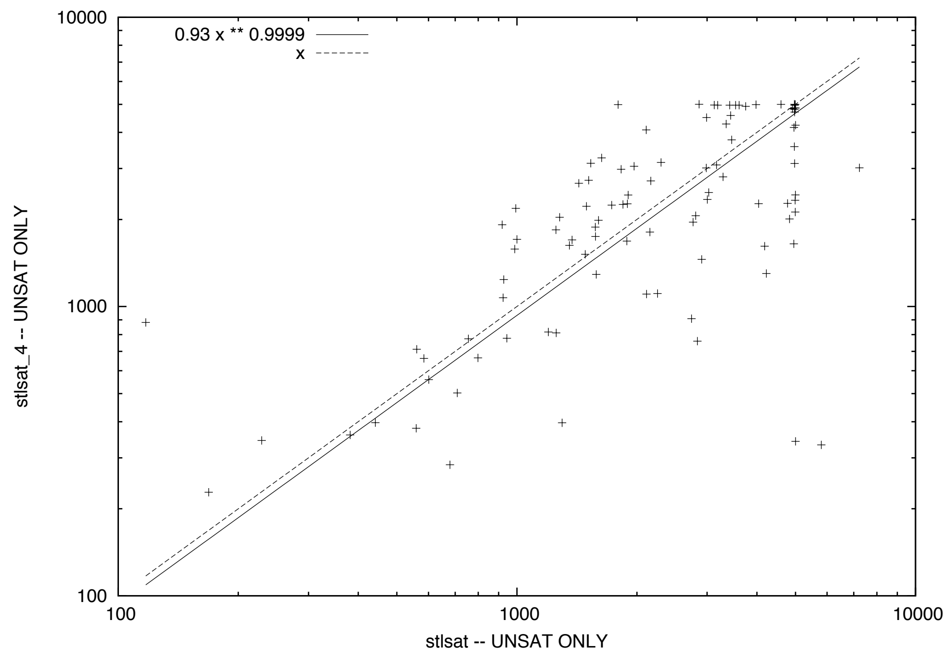


Figure 3: STLSAT vs. STLSAT_4, SAT competition problems (CPU time in seconds)

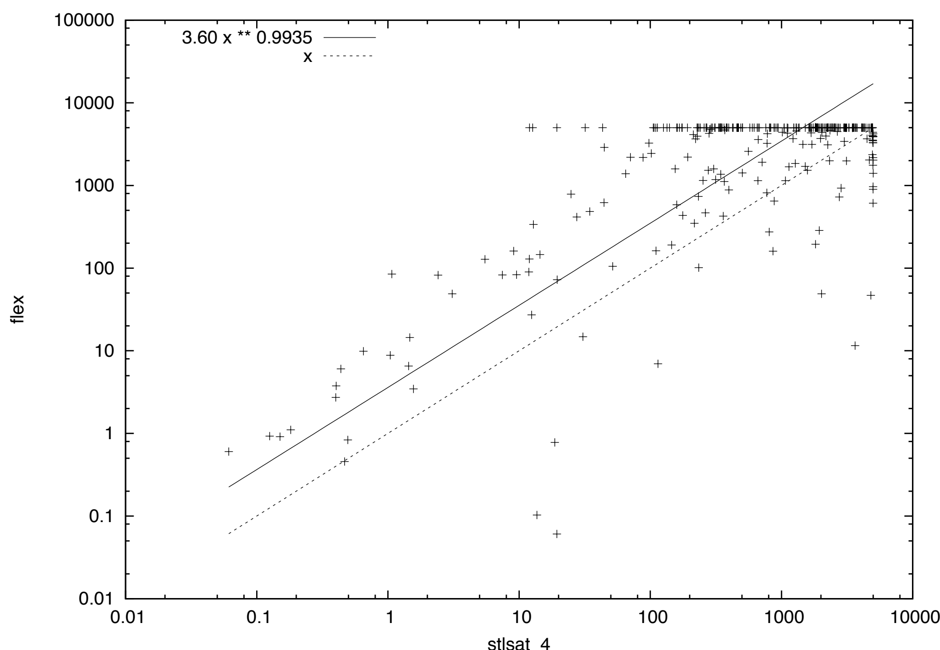


Figure 4: STLSAT$_4$ vs. FLEX, SAT competition problems (CPU time in seconds)

Note that the scatter in these cases is much more significant than that of Figure 1. This is as expected – STLSAT is intended to be essentially unchanged from MINISAT from an algorithmic perspective, while the change in restart frequency can be expected to have fairly dramatic effects. Note also the collection of points with either $x = 5000$ or $y = 5000$, indicating that the problem was solved by one method but timed out using the other.

Finally, Figure 4 compares STLSAT$_4$ to FLEX on the SAT competition problems, and FLEX is about a factor of 3.6 slower (4.53 times slower for satisfiable instances; 2.01 times slower for unsatisfiable ones).[10] Many more problems time out for FLEX than for the modified version of MINISAT.

This is shown clearly in Figure 5, where we compare the number of problems being solved by STLSAT$_4$ and by FLEX on the SAT competition problems. For any specific time cutoff, STLSAT$_4$ solves about 2.5 times as many problems as does FLEX.

It does appear to be the case, however, that the performance gap is narrowing as the problems become more difficult; for example, FLEX solves 36 problems in between 3000 and 5000 seconds; STLSAT$_4$ solves 42 problems in that time. It is for this reason that we examined number factoring problems as well.[11]

10. The graphs for satisfiable and unsatisfiable instances are not shown; they look basically the same as Figure 4.

11. Note that we cannot simply increase the timeout limit for the SAT competition problems; doing so caused only a relative handful of new problems to be solved. If we are to generate an interesting number of more difficult problems, and want the instances to be known to be satisfiable, number factoring seems to be the simplest way to go.

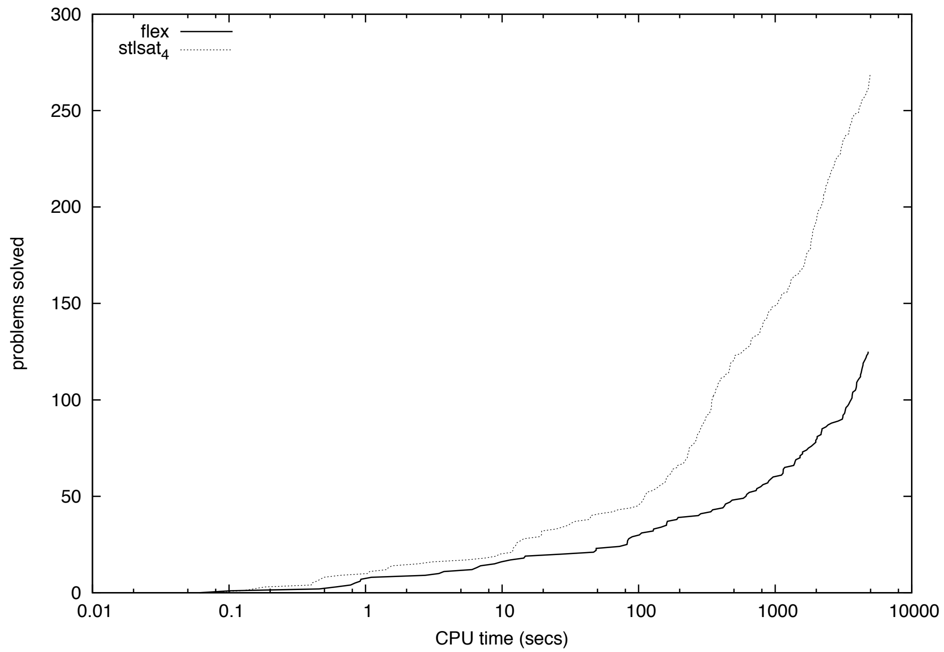


Figure 5: STLSAT$_4$ vs. FLEX, SAT competition problems (5000 second timeout)

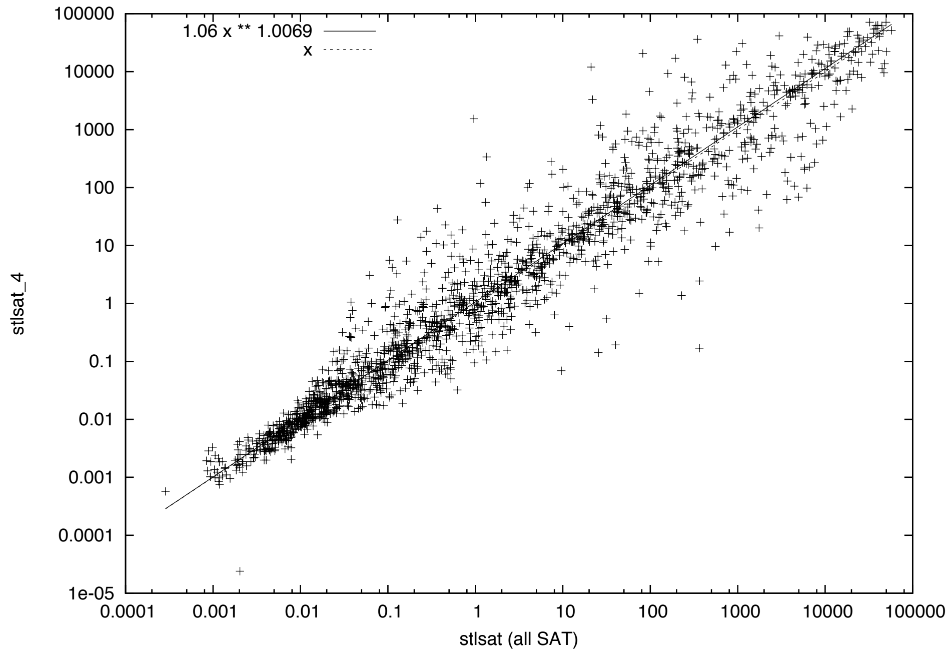


Figure 6: STLSAT vs. STLSAT$_4$, factoring problems (CPU time in seconds)

4.2 Number Factoring

For number factoring, we generated 50 factoring problems of each size from five to 45 bits. These problems were then run through STLSAT, STLSAT$_4$ and FLEX.

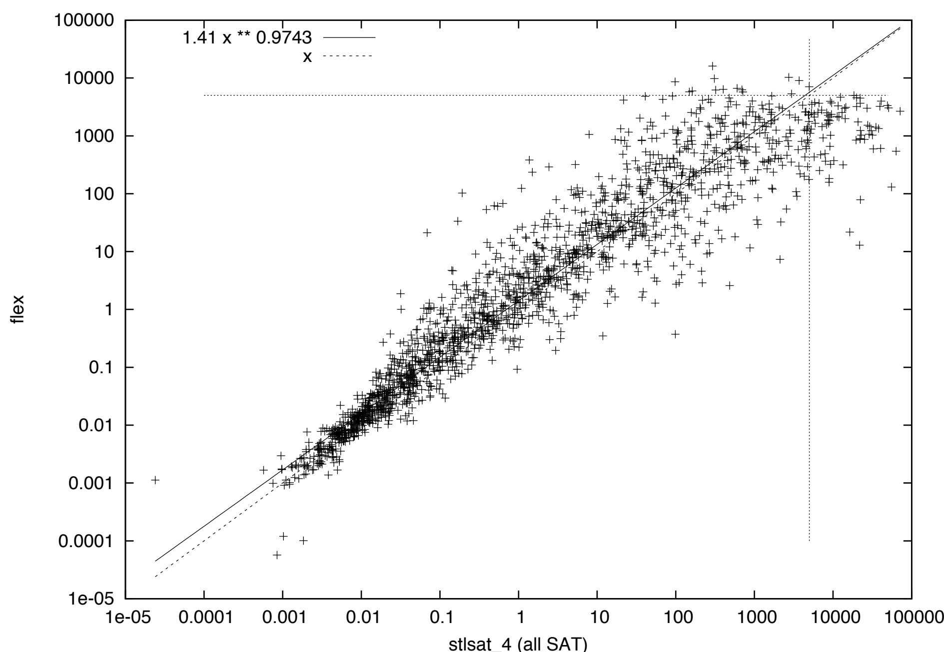


Figure 7: STLSAT$_4$ vs. FLEX, factoring problems (CPU time in seconds; 5000 second timeout)

The (relatively uninteresting) result of comparing STLSAT and STLSAT$_4$ is shown in Figure 6. As for the SAT competition problems, reducing the size of the first probe to four backtracks leads to a small degradation in performance on satisfiable problems. We include this primarily to ensure that the change in probe size is not responsible for the results in the following figures.

FLEX and STLSAT$_4$ are compared in Figure 7. Here, we show only problems that one method or the other was able to solve in 5000 seconds or less, although exact times to solution are used for both solvers. The figure also includes dotted lines indicating the 5000 second timeout for either solver.

The overall results are similar to those of the previous section, although FLEX's performance is improved somewhat. On SAT competition problems, it was 3.6 times slower overall and 4.53 times slower for satisfiable problems specifically. On number factoring problems, FLEX is only 41% slower than is STLSAT$_4$.

The curve fit is being dominated by the large number of easy problems in this case. In addition, it *appears* that FLEX is actually doing better than STLSAT$_4$ as the problems become more difficult; many more problems appear to be "timing out" with STLSAT$_4$ (points to the right of $x = 5000$ seconds in Figure 7) than with FLEX (points above $y = 5000$ in the figure). This suggests that instead of looking at the easiest factoring problems, we look at the hardest ones.

This is done in Figure 8, where we consider only problems for which one of the two solvers required at least 5000 seconds to reach a solution. The results are strikingly different. A glance shows that FLEX is in general solving problems more quickly than is STLSAT$_4$; in fact,

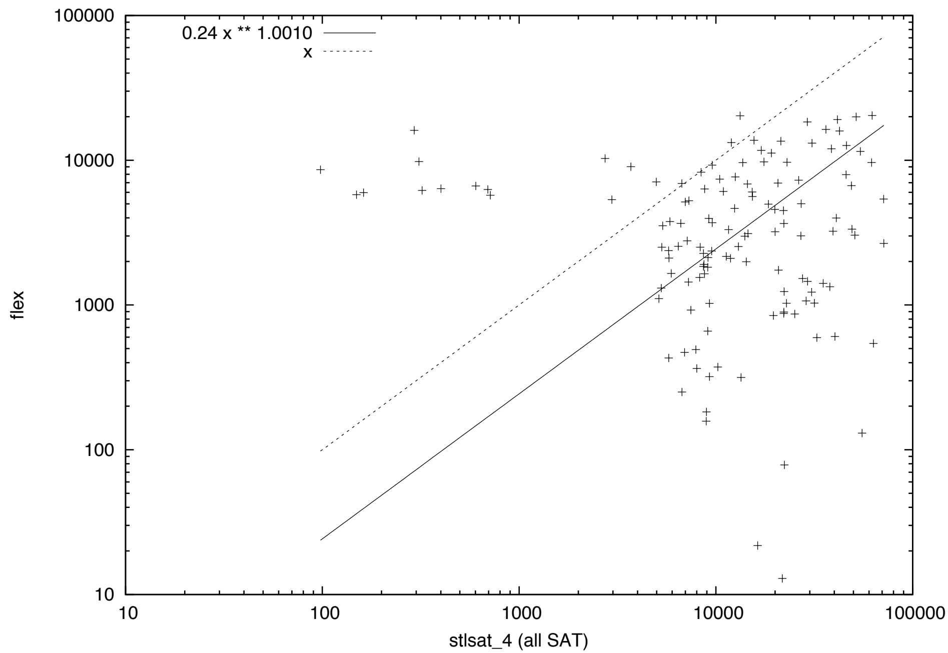


Figure 8: STLSAT$_4$ vs. FLEX, factoring problems (CPU time in seconds; 5000 seconds or harder)

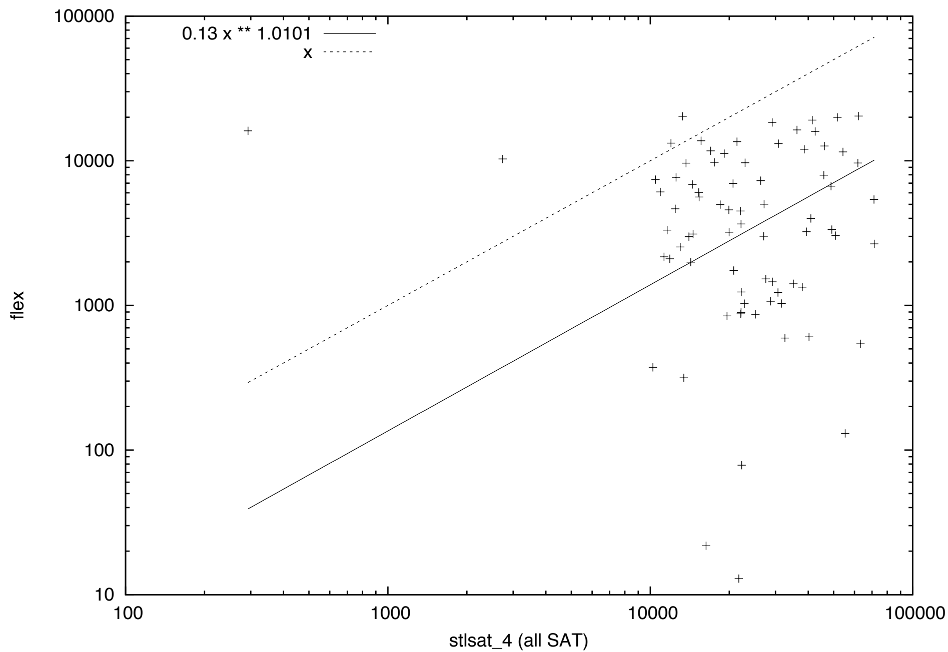


Figure 9: STLSAT$_4$ vs. FLEX, factoring problems (CPU time in seconds; 10,000 seconds or harder)

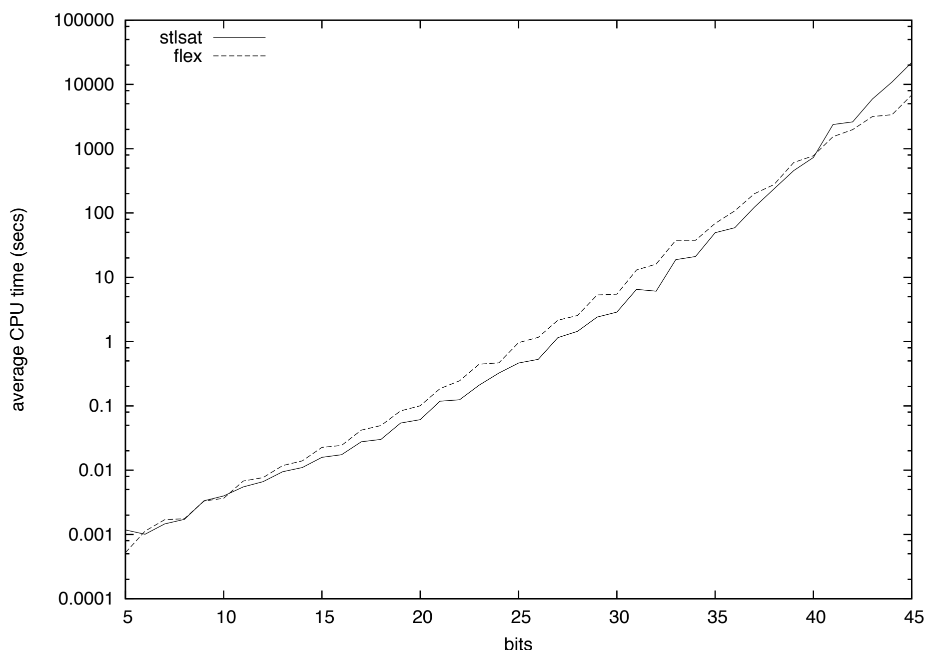


Figure 10: STLSAT$_4$ vs. FLEX, average time to factor an n-bit number

the line of best fit shows that FLEX is about four times as fast is as STLSAT$_4$. Restricting to problems that took at least 10,000 seconds makes the distinction sharper still; now (Figure 9) FLEX is eight times faster than is STLSAT$_4$.

Similar results appear in Figure 10, which shows the average running time needed by FLEX or by STLSAT to factor a number of n bits. FLEX is slower for smaller numbers but faster for larger ones. It is unfortunately impractical to extend this graph much further, since the average runtime for STLSAT on a single 45-bit factoring problem is already approximately six hours.

As previously, we show the number of difficult problems (40 bits or more) solved by the solvers as a function of time in Figure 11. With a cutoff of less than 1000 seconds, STLSAT$_4$ performs best; as the problems get harder, this is reversed.

5. Conclusion and Future Work

The fundamental claim that we have made in this paper is that the SAT community did not need to abandon the systematicity of its methods when it switched from DPLL-style provers to CDCL-based ones. We showed that it was possible to simultaneously guarantee systematicity, maintain a polynomially-sized set of learned clauses, and restart the prover as frequently as desired. Furthermore, we showed that while a hybrid between a "pure" systematic prover and a more typical CDCL engine incurs significant computational cost in achieving these goals, that cost and more is recovered and run time appears to be reduced for difficult problems.

Looking forward, there are two obvious ways in which this work can be extended.

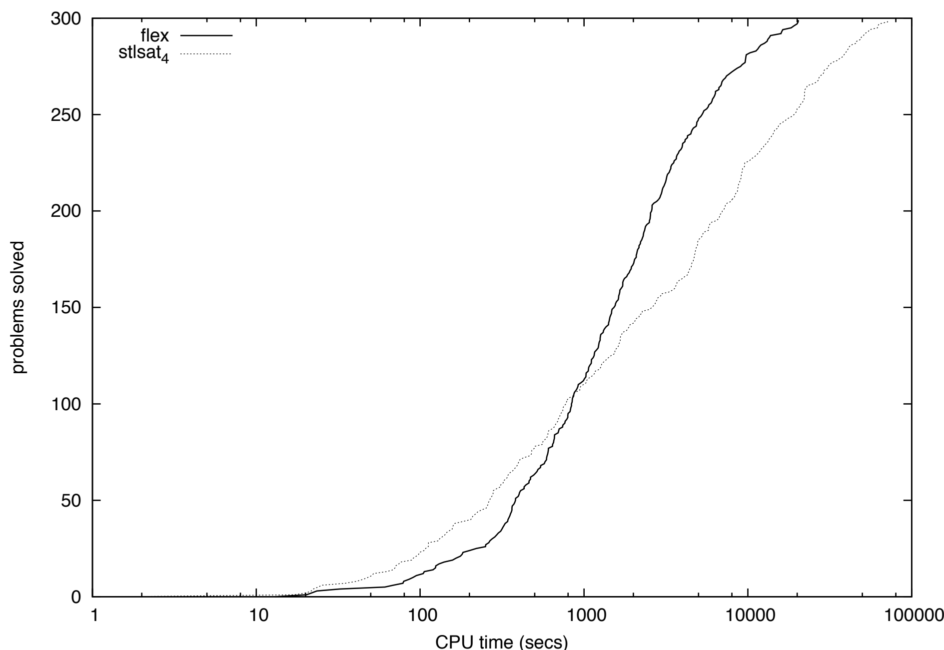


Figure 11: STLSAT$_4$ vs. FLEX, factoring problems (40 bits or larger)

First, we have seen that the value of our methods increases as the number of restarts increases. One of the principal reasons to limit the number of restarts in a CDCL-based prover is that restarting the prover is expensive. Recent work by Ramos et al. (2011), however, suggests that a significant part of this expense can be avoided. The basic idea is that any particular probe may well begin its search by repeating many of the literal choices and unit propagations from the previous probe. In such a case, there is no reason to backtrack past the point at which the two probes first diverge. Incorporating this idea into our methods should improve their value further by enabling a larger number of restarts and thus making the systematic component that we offer more effective.

Second, we hope that the overall approach that we have proposed – guaranteeing systematicity by retaining a partial order and an appropriate set of learned clauses – finally puts to rest the notion that SAT engines achieve systematicity only by careful management of a search space defined by partial assignments. It is instead the learned clausal database that guarantees systematicity through semantic methods.

This general conclusion should lead to a variety of new algorithmic choices. As an example, the values represented by the bias are presumably best thought of as probabilistic estimates regarding the likelihood of any particular literal appearing in a model of the theory in question. This probabilistic approach gives a very different flavor to satisfiability in general, and suggests that recent results from Monte Carlo Tree Search, or MCTS (Browne, Powley, Whitehouse, Lucas, Cowling, Rohlfshagen, Tavener, Perez, Samothrakis, & Colton, 2012; Kocsis & Szepesvári, 2006, and others) may well find a place in SAT as well.

Acknowledgments

I would like to thank my Connected Signals and On Time Systems coworkers, especially Aran Clauson and Heidi Dixon, for useful technical advice and assistance. I would also like to thank the anonymous referees for their many carefully considered comments and suggestions, which improved this paper enormously. Finally, I would like to thank David McAllester for many contributions to this work over the years; the work presented here draws greatly from McAllester's original work (1993) on partial-order dynamic backtracking.

Appendix A. Proof of Theorem 3.10

The proof of Theorem 3.10 will depend on certain invariants that are maintained as Procedure 3.8 is executed. We begin by discussing those invariants.

A.1 Learned Clauses

To understand them, suppose that a set of learned clauses Γ has been derived during the course of a systematic search. What conditions would we expect Γ to satisfy?

To answer this, imagine that we have a partial assignment P and that we have derived a new learned clause γ that we are about to add to Γ.

We certainly expect P to satisfy the existing learned clauses. We will also assume that P satisfies the antecedents of those learned clauses; this is in keeping with the idea that the learned clauses should continue to be "relevant" to the portion of the search space being explored.

What about the new clause γ? First, it will turn out that γ should have a *unique* conclusion under the partial order $\leq$. In addition, the new learned clause γ should be new knowledge in the sense that it eliminates a portion of the search space that is still admissible under Γ. In other words, we expect the conclusion $\gamma^{\leq}$ to be distinct from the conclusions $\Gamma^{\leq}$ of other elements of Γ.

Formally, we have:

Definition A.1 *We will say that a partial order* $\leq$ parses *a learned clause* γ *if* $|\gamma^{\leq}| = 1$.

We will say that a set of learned clauses Γ *is* acceptable *for a partial order* $\leq$ *if the following conditions hold:*

1. *For any* $\gamma \in \Gamma$, $\leq$ *parses* γ,
2. *For any* $\gamma_1, \gamma_2 \in \Gamma$, *if* $\gamma_1^{\leq} = \gamma_2^{\leq}$, *then* $\gamma_1 = \gamma_2$, *and*
3. $\Gamma_{\leq} \not\models \neg\Gamma$.

The first condition repeats the requirement that the conclusions $\gamma^{\leq}$ are all individual literals. The second condition says that Γ does not contain two distinct learned clauses with the same conclusion. The third says that it is at least *possible* to find a partial assignment P as described above, since the collection of learned clauses Γ is consistent with the antecedents $\Gamma_{\leq}$ of those learned clauses.

Lemma A.2 *If* Γ *is acceptable for a partial order* $\leq$, *then* Γ *contains at most one learned clause with any particular variable* x *in its conclusion.*

Proof. If Γ contains clauses with both x and $\neg x$ as conclusions, it will be impossible to have both Γ and $\Gamma_{\leq}$, contradicting the requirement of acceptability. ∎

Proposition A.3 *If Γ is a set of clauses involving v variables and is acceptable for a partial order $\leq$, then $|\Gamma| \leq v$.* ∎

A.2 Search Space Size

Before considering the general way in which the new clause γ allows us to improve the bias, and the way in which the new clause is incorporated into Γ, let us examine a simple special case where $\leq$ is in fact a total order on the variables in question. This means that any particular learned clause γ is always interpreted as forcing the value of the single variable it contains that occurs latest under the ordering $\leq$.

Now given a set of learned clauses Γ, we extend P until we either solve the problem or derive a new learned clause γ. Given the bias B, we are assuming that $B \models \neg\gamma$ so that the learned clause allows us to improve the bias, and we have already remarked that we assume that B is consistent with $\Gamma \wedge \Gamma_{\leq}$. It follows that $\neg\gamma$ is consistent with $\Gamma \wedge \Gamma_{\leq}$ as well. If l is the conclusion of γ, how are we to update Γ?

Note first that there can be no $\alpha \in \Gamma$ with l as its conclusion. If there were, then we would have $\Gamma \wedge \Gamma_{\leq} \models l$ and therefore $\Gamma \wedge \Gamma_{\leq} \models \gamma$ also. There are now two cases: Γ may contain a learned clause with conclusion $\neg l$, or Γ may contain no such learned clause.

If Γ does not contain a learned clause with conclusion $\neg l$, we would like to simply add γ to Γ. But doing so may cause Γ to become unacceptable, since Γ may contain a learned clause with $\neg l$ in its antecedent. If we simply add γ to Γ, the result will be unacceptable with respect to $\leq$ because it will be impossible to satisfy all of the elements of Γ while simultaneously satisfying their antecedents.

As an example, suppose that Γ contains the learned clause $a \wedge c \rightarrow d$, the antecedent of which is $a \wedge c$. If we learn $a \rightarrow \neg c$ and simply add it to Γ, then $\Gamma \wedge \Gamma_{\leq}$ includes both $a \wedge c$ and $a \rightarrow \neg c$, a contradiction.

What we can do in this case is to remove from Γ every learned clause that has $\neg l$ in its antecedent; call the result Γ'. We now take $\textsc{add}(\gamma, \Gamma)$ to be $\Gamma' \cup \{\gamma\}$. $\leq$ parses γ and obviously continues to parse all of the learned clauses remaining in Γ'. $\Gamma' \cup \{\gamma\}$ is therefore acceptable with respect to $\leq$, since $\neg l$ can no longer appear in the antecedent of any learned clause in Γ'.

Given that we have discarded some of the learned clauses in Γ, do we have any basis for concluding that the procedure will eventually terminate? We do. As mentioned in the main body of the text, the key observation is that we have made *lexicographic* progress in the search space. While it is true that the allowed domains of all of the variables following l may have gotten bigger when we replaced Γ with Γ', the domain for l itself has gotten smaller. Since l precedes these other variables in the total order, we have made lexicographic progress and systematicity remains guaranteed.

Consider now the second case, where Γ does contain a learned clause with conclusion $\neg l$. Note first that since Γ is acceptable, the learned clause concluding $\neg l$ must be unique. We can resolve this learned clause with γ to obtain a new learned clause ρ. Every variable in ρ precedes l according to the ordering $\leq$, so the conclusion of ρ must precede l as well. We can thus continue to make lexicographic progress if we define $\textsc{add}(\gamma, \Gamma)$ to be $\textsc{add}(\rho, \Gamma)$, where

we have essentially chosen to incorporate the lexicographically more powerful resolvent ρ instead of the original learned clause γ.

To ensure that these ideas are clear, suppose that our original learned clause set Γ is given by the following, where the ordering $\leq$ ranks the variables alphabetically:

$$\begin{aligned} a \wedge c &\rightarrow d \\ c \wedge d &\rightarrow e \end{aligned} \tag{6}$$

Now if the new learned clause is $a \wedge b \rightarrow f$, we simply add it to Γ. No learned clause need be removed because the conclusion f does not appear in the antecedent of any learned clause in Γ.

Having added $a \wedge b \rightarrow f$, suppose that we now learn $a \rightarrow \neg c$. As in the example above, we add the learned clause to Γ and remove every learned clause whose antecedent includes c, which means that both of the original learned clauses are removed from Γ, although $a \wedge b \rightarrow f$ is retained. The new Γ set is:

$$\begin{aligned} a \wedge b &\rightarrow f \\ a &\rightarrow \neg c \end{aligned} \tag{7}$$

We have made lexicographic progress. If a is true, the original learned clause set (6) allowed c to be either true or false; the new learned clause set (7) forces c to be false. Reducing the number of possible values for c counts as progress, whatever happens to the possible values for subsequent variables.

Suppose that instead of learning $a \rightarrow \neg c$, we had learned $c \rightarrow \neg f$. This conflicts with $a \wedge b \rightarrow f$, so we resolve the two together to obtain $\neg a \vee \neg b \vee \neg c$, which the total order causes us to interpret as $a \wedge b \rightarrow \neg c$. We add this to Γ as in the previous paragraph; once again, lexicographic progress has been made.

This restricted example contains the essence of the ideas we will use in the more general case. The general case is more complex because, just as we must ensure that Γ remains acceptable, we would also like to ensure that the partial order $\leq$ remains as flexible as possible. If $\leq$ were to approach a total order, it would tend to force specific interpretations of newly learned clauses, reducing our ability to reverse decisions that were made early in the search. This would mean that we would have relatively little flexibility in modifying the bias; we would always have to simply flip the bias variable that was most recently valued even as evidence accumulated that this variable was valued correctly.

We begin by formalizing the notion of lexicographic progress.

Definition A.4 *If Γ is a set of learned clauses and $\leq$ is a total order, we will denote by $|\Gamma|_x$ the number of elements of Γ that have conclusion either x or $\neg x$. We will denote by $|{>}x|$ the number of y with $y > x$.*

If v is the number of variables being considered, we now define the total order size of Γ under $\leq$ *to be*

$$\mathbf{size_t}(\Gamma, \leq) = 2^v - \sum_x 2^{|>x|} |\Gamma|_x \tag{8}$$

Proposition A.5 *If Γ is acceptable for a total order $\leq$, then $2^v \geq \mathbf{size_t}(\Gamma, \leq) > 0$.*

Proof. The first inequality is immediate. For the second, since $|\Gamma|_x \leq 1$,

$$\mathbf{size_t}(\Gamma, \leq) \geq 2^v - \sum_x 2^{|>x|}$$

But consider the summation in isolation, where we have:

$$\sum_x 2^{|>x|} \leq \sum_{i=1}^{v} 2^{i-1} = 2^v - 1.$$

Thus $\mathbf{size_t}(\Gamma, \leq) > 0$. ■

Before moving on, let us explain the intuition underlying Definition A.4. The basic idea is that the expression in (8) should reflect the size of the search space remaining to be considered, because

$$\sum_x 2^{|>x|} |\Gamma|_x$$

counts the number of potential models eliminated by the learned clauses in Γ.

That it does should be fairly clear. The unique learned clause in $|\Gamma|_x$ eliminates a value for x. Each such elimination corresponds to the removal of a subtree in the overall search space. The size of the subtree is given by $2^{|>x|}$, since $|{>x}|$ is the number of variables properly following x in the ordering.

Proposition A.6 *Fix a total order $\leq$, and let Γ and Γ' be two sets of learned clauses. Assume that there is some x with $|\Gamma|_x < |\Gamma'|_x$ but $|\Gamma|_y = |\Gamma'|_y$ for all $y < x$.*

Now if $\leq'$ is a total order that agrees with $\leq$ when restricted to the set $\{y, z | y, z \leq x\}$, then $\mathbf{size_t}(\Gamma', \leq') < \mathbf{size_t}(\Gamma, \leq)$.

Proof. This is simply a formalization of the lexicographic argument made earlier. To show $\mathbf{size_t}(\Gamma, \leq) - \mathbf{size_t}(\Gamma', \leq') > 0$, we have

$$\mathbf{size_t}(\Gamma, \leq) - \mathbf{size_t}(\Gamma', \leq') = \sum_{y'} 2^{|>'y|} |\Gamma'|_y - \sum_y 2^{|>y|} |\Gamma|_y \tag{9}$$

We will show:

1. For any $y < x$,
$$2^{|>'y|} |\Gamma'|_y = 2^{|>y|} |\Gamma|_y \tag{10}$$

2. For $y = x$,
$$2^{|>'x|} |\Gamma'|_x - 2^{|>x|} |\Gamma|_x \geq 2^{|>x|} \tag{11}$$

3. For $y > x$,
$$\sum_y 2^{|>y|} |\Gamma|_y < 2^{|>x|} \tag{12}$$

Collectively, these suffice. The terms (10) with $y < x$ have no effect on the difference in (9). The amount contributed by the term (11) with $y = x$ is greater than the amount subtracted by the terms (12) with $y > x$. Thus the total sum is positive and the result follows.

(10) is a consequence of the fact that for $y \leq x$, $|{>'}y| = |{>}y|$ and $|\Gamma'|_y = |\Gamma|_y$. For (11), since $|\Gamma'|_x > |\Gamma|_x$, we get

$$\begin{aligned} 2^{|>x|}|\Gamma|_x &\leq 2^{|>x|}(|\Gamma'|_x - 1) \\ &= 2^{|>'x|}|\Gamma'|_x - 2^{|>x|} \end{aligned}$$

For (12), the sum is essentially identical to that appearing at the end of the previous proof. ■

Definition A.7 *A partial order* $\leq'$ *will be called a* refinement *of a partial order* $\leq$ *if* $x \leq y \rightarrow x \leq' y$. *A refinement of* $\leq$ *that is a total order will be called a* total refinement *of* $\leq$.

Definition A.8 *For a set of learned clauses* Γ *and partial order* $\leq$, *we define the* size of Γ under $\leq$, *denoted* $\mathbf{size}(\Gamma, \leq)$, *to be the maximum of* $\mathbf{size_t}(\Gamma, \leq')$ *over all total refinements* $\leq'$ *of* $\leq$.

The following is an immediate consequence of Proposition A.5:

Corollary A.9 *If* Γ *is acceptable for a partial order* $\leq$, *then* $2^v \geq \mathbf{size}(\Gamma, \leq) > 0$. ■

The point of all this is that we can now state our desiderata for ADD a bit more precisely. We need:

1. **Polynomial space:** If Γ is acceptable with respect to $\leq$ and $(\gamma', \Gamma', \leq') = \text{ADD}(\gamma, \Gamma, \leq)$, then we want Γ' to be acceptable with respect to $\leq'$. This will allow us to use Proposition A.3 to conclude that Γ can be stored in polynomial space as the algorithm proceeds. $\leq$ and the bias B can obviously be stored in polynomial space as well.

2. **Systematicity:** If $(\gamma', \Gamma', \leq') = \text{ADD}(\gamma, \Gamma, \leq)$, then we want $\mathbf{size}(\Gamma', \leq') < \mathbf{size}(\Gamma, \leq)$. This will allow us to use Corollary A.9 to conclude that ADD can be invoked at most 2^v times before the entire search space is eliminated.

A.3 Systematicity

Recall:
Procedure 3.6 *To compute* $\text{ADD}(\gamma, \Gamma, \leq)$:

1. **if** $\gamma = \bot$ **then return** $(\gamma, \Gamma \cup \{\bot\}, \leq)$;
2. *select* $l \in \gamma^{\leq}$; // l is the intended conclusion
3. **if** *there is a* $\rho \in \Gamma$ *with* $\rho^{\leq} = \neg l$ **then**
4. | **return** $\text{ADD}(\text{RESOLVE}(\gamma, \rho), \Gamma, \leq)$
5. *add* $x \leq l$ *to* $\leq$ *for each* $x \in \gamma$;
6. $\leq \leftarrow \leq_l$;
7. *remove from* Γ *any* c *with either* $c_{\leq} \cap \{l, \neg l\} \neq \emptyset$ *or both* $c \cap \{x | l < x\} \neq \emptyset$ *and* $|c^{\leq}| > 1$;
8. *return* $(\gamma, \Gamma \cup \{\gamma\}, \leq)$

Proposition A.10 *Suppose that* Γ *is acceptable with respect to* $\leq$*, and that* $\Gamma \wedge \Gamma_{\leq} \not\models \gamma$*. Then if* $(\gamma', \Gamma', \leq') = \text{ADD}(\Gamma, \gamma, \leq)$*,* Γ' *is acceptable with respect to* $\leq'$*.*

Proof. Note first that the procedure terminates. The only possible source of nontermination is the recursive call on line 4, but the conclusion of the resolvent is necessarily $<$ the conclusion of γ. The number of recursive calls is therefore bounded, and the procedure terminates.

To show that Γ' is acceptable, we consider first the nonrecursive case where there is no $\rho \in \Gamma$ with conclusion $\neg l$. For the three conditions of Definition A.1, we have:

1. To show that $\leq'$ parses c for any $c \in \Gamma'$, there are two cases.

   If c is the learned clause γ, we know by construction that $\leq'$ includes $y \leq x$ for every y appearing in c.

   For the case where $c \in \Gamma$, we need the following lemma:

   Lemma A.11 *Suppose that* $x \leq u$ *but* $x \not\leq' u$*. Then* $l <' x$ *and* $l < x$*.*

   Proof. Suppose that we denote by $\leq^+$ the partial order constructed in line 5 of the procedure, where additional arcs are added to the original $\leq$. Since $\leq^+$ is a refinement of $\leq$, it is clear that if $x \leq u$, we will have $x \leq^+ u$ as well.

   We now have $\leq' = \leq^+_l$, so the only way to have $x \leq^+ u$ but not $x \not\leq' u$ is if $l <^+ x$, so that $l <' x$ as well.

   To show that $l < x$, note that if $l <' x$, then $l <^+ x$ because arcs are removed in the construction of $<'$ from $<^+$. But if $l <^+ x$ we must also have $l < x$, since the arcs added in the construction of $<^+$ are all of the form $z < l$. ■

   Returning to the proof of the proposition, suppose that $y, z \in c^{\leq'}$ with y and z incomparable under $\leq'$. Now if either $l <' y$ or $l <' z$, the clause in question would have been eliminated by line 7 of the ADD construction, so it follows that $l \not<' y$ and $l \not<' z$.

   We claim that there is no $u \in c$ with $y < u$. If there were such a u, then we would have $y \leq' u$ by virtue of the lemma, so $y \notin c^{\leq'}$. Similarly there is no $u \in c$ with $z < u$. Thus $y, z \in c^{\leq}$ and therefore $y = z$, since $\leq$ parses c.

2. To show that no two elements of Γ' have the same conclusion, note first that no two elements of Γ' have the same conclusion under the original partial order $\leq$. This is true by construction for two elements of Γ. If the new learned clause γ were to have the same conclusion as an element of Γ, we would have had $\Gamma \wedge \Gamma_{\leq} \models \gamma$.

   The only way for the conclusion of $c \in \Gamma'$ under $\leq'$ to be different than the conclusion under $\leq$ is if there are $y, z \in c$ with $y < z$ but $z <' y$. But then by the lemma we must have $l < y$ and $l <' y$. Since $l < y < z$, $l < z$ and therefore $l <' z$. But now the weakening in line 6 causes y and z to be incomparable under $<'$.

3. Finally, we must show that $\Gamma'_{\leq'} \not\models \neg\Gamma'$, or that $\Gamma'_{\leq'}$ and Γ' are consistent.

   To see this, we can use the fact that $\Gamma \wedge \Gamma_{\leq} \not\models \gamma$ to find a complete assignment P of values to variables that satisfies Γ, $\Gamma_{\leq}$ and $\neg\gamma$. Note that P must both satisfy the antecedent of γ and include $\neg l$ in order to falsify γ.

   We now claim that $P|_l$ satisfies both $\Gamma'_{\leq'}$ and Γ'. $P|_l$ continues to satisfy the antecedent $\gamma_{\leq}$, since $l \notin \gamma_{\leq}$, and $P|_l$ satisfies γ since it contains l. We therefore need only consider $c \in \Gamma'$ that appear in Γ as well.

   Since P satisfies both c and $c_{\leq}$, the only way that $P|_l$ might not satisfy c and $c_{\leq'}$ is if $P|_l$ fails to satisfy c because the conclusion of c is no longer satisfied, or $P|_l$ fails to satisfy $c_{\leq'}$ because one of the terms in the antecedent is no longer satisfied. The first of these cannot happen because $P|_l$ differs from P only at l, and l cannot be c's conclusion. The second cannot happen because if $\neg l$ appears in the antecedent of c, c would have been removed in the construction of Γ'.

This concludes the proof that Γ remains acceptable if a nonrecursive path is taken through ADD; to see that the recursive path remains acceptable as well, we need to show that the fundamental requirement that $\Gamma \wedge \Gamma_{\leq} \not\models \gamma$ holds for the recursive call. In other words, the proof will be complete if we can show that $\Gamma \wedge \Gamma_{\leq} \not\models \text{RESOLVE}(\gamma, \rho)$ in line 4.

Suppose that $\Gamma \wedge \Gamma_{\leq} \models \text{RESOLVE}(\gamma, \rho)$, and note that

$$\text{RESOLVE}(\gamma, \rho) = \neg\gamma_{\leq} \vee \neg\rho_{\leq}$$

Now $\rho_{\leq} \in \Gamma_{\leq}$, so we must therefore have

$$\Gamma \wedge \Gamma_{\leq} \models \neg\gamma_{\leq}$$

But this would imply $\Gamma \wedge \Gamma_{\leq} \models \gamma$, contradicting the assumptions of the proposition. The proof is complete. ∎

Proposition A.12 *Suppose that Γ is acceptable with respect to $\leq$, and that $\Gamma \wedge \Gamma_{\leq} \not\models \gamma$. Then if $(\gamma', \Gamma', \leq') = \text{ADD}(\Gamma, \gamma, \leq)$, $\mathbf{size}(\Gamma', \leq') < \mathbf{size}(\Gamma, \leq)$.*

Proof. We need the following lemma:

Lemma A.13 *Let $\leq$ be a partial order and x a point. Then if $\leq_x^T$ is a total refinement of $\leq_x$, there is a total refinement $\leq^T$ of $\leq$ such that $\leq_x^T$ and $\leq^T$ agree when restricted to $\{y, z | y, z \leq_x^T x\}$.*

Proof. Define $y \leq' z$ if and only if any of the following three conditions hold:

1. $y, z \leq_x^T x$ and $y \leq_x^T z$,
2. $x <_x^T y, z$ and $y \leq z$, or
3. $y \leq_x^T x <_x^T z$.

We claim that $\leq'$ is a *partial* order satisfying the requirements of the lemma. (Informally, $\leq'$ is the partial order that matches $\leq$ above x and $\leq_x^T$ below x.)

That $\leq'$ is reflexive is clear. For transitivity, suppose that $u \leq' v \leq' w$. If $x <_x^T u$, the second clause of the definition is the only one that can support $u \leq' v$, so we must have $x <_x^T v$ and similarly $x <_x^T w$ together with $u \leq v \leq w$, so that $u \leq' w$. If $w \leq_x^T x$, an analogous argument about applicability of the first clause requires $v \leq_x^T x$ and similarly $u \leq_x^T x$; now the transitivity of $\leq_x^T$ gives $u \leq_x^T w$ so $u \leq' w$ also. Finally, if $u \leq_x^T x <_x^T w$, we get $u \leq' w$ directly.

To see that $\leq'$ is anti-symmetric, suppose $u \leq' v \leq' u$ with $u \neq v$. We cannot have both $u \leq_x^T x \leq_x^T v$ and $v \leq_x^T x \leq_x^T u$ because $\leq_x^T$ is a partial order, so either the first or the second clause of the definition of $\leq'$ must apply. Either way, we must have $u = v$ because both $\leq_x^T$ and $\leq$ are partial orders.

To see that $\leq_x^T$ and $\leq'$ agree when restricted to points y and z with $y, z \leq_x^T x$, note that it is only the first clause in the definition that ever asserts $y \leq' x$ for such y, and this clause says that $\leq_x^T$ and $\leq'$ match.

It remains only to show that $\leq'$ is a refinement of $\leq$, so that if $y \leq z$, then $y \leq' z$.

So suppose that $y \leq z$. Now if $x \leq y$, then $x \leq_x y$ and therefore $x \leq_x^T y$. Similarly, $x \leq_x^T z$. Thus case 2 of the definition applies, and $y \leq' z$.

If, on the other hand, $x \not\leq y$, then $y \leq_x z$ by the definition of the weakening $\leq_x$. Thus $y \leq_x^T z$ as well. Now there are two cases, depending on the $\leq_x^T$ relationship between x and y:

1. If $x \leq_x^T y$, then $x \leq_x^T z$ as well and case 2 of the definition implies $y \leq' z$.
2. If $y \leq_x^T x$ (recall that $\leq_x^T$ is total), there are two subcases:
   (a) If $x \leq_x^T z$, then case 3 of the definition applies and $y \leq' z$.
   (b) If $z \leq_x^T x$, then case 1 applies.

In all cases, we can conclude $y \leq' z$ from $y \leq z$.

This shows that $\leq'$ is a partial order satisfying the conditions of the lemma. Taking $\leq^T$ to be any total refinement of $\leq'$ completes the proof. ■

We can now return to the proof of Proposition A.12. As earlier, we denote by $\leq^+$ the partial order obtained in line 5 of the ADD procedure, where we have added new constraints based on the selection of l as the conclusion of γ. We will show that

$$\mathbf{size}(\Gamma', \leq') < \mathbf{size}(\Gamma, \leq^+) \leq \mathbf{size}(\Gamma, \leq) \tag{13}$$

The second inequality is easy; since $\leq^+$ is a refinement of $\leq$, there are fewer further refinements to consider in Definition A.4 of **size**. For the first inequality, if $\leq_1$ is any total order used to evaluate $\mathbf{size}(\Gamma', \leq')$, we must show that there is *some* total order $\leq_2$ used to evaluate $\mathbf{size}(\Gamma, \leq^+)$ for which $\mathbf{size_t}(\Gamma', \leq_1) < \mathbf{size_t}(\Gamma, \leq_2)$.

Recall that lines 5 and 6 of Procedure 3.6 constructed $\leq'$ as $\leq_l^+$. It follows that $\leq_1$ is a total refinement of $\leq_l^+$. We therefore know from the lemma that there is some $\leq_2$ that is a total refinement of $\leq^+$ such that $\leq_1$ and $\leq_2$ agree for $\{x, y | x, y \leq_1 l\}$. Since $\gamma \in \Gamma'$, $|\Gamma'|_l > |\Gamma|_l$ and we can thus apply Proposition A.6 to conclude

$$\mathbf{size_t}(\Gamma', \leq_1) < \mathbf{size_t}(\Gamma, \leq_2)$$

It follows that the maximum over the $\leq_1$'s is less than the maximum over the $\leq_2$'s, so that

$$\mathbf{size}(\Gamma', \leq') < \mathbf{size}(\Gamma, \leq^+)$$

as needed in (13). The proof is complete. ■

Procedure 3.8 *Let T be a theory, Γ a set of clauses such that $T \models \Gamma$, $\leq$ a partial order on the variables in T, and B a bias for T. Then to compute* FLEX$(T, \Gamma, \leq, B, n)$, *one of: $\bot$ if T is shown to be unsatisfiable, a model P of T if one is found, or* UNKNOWN *if no solution is found after n steps:*

```
1  i ← 0;
2  while i < n do
3  |  (P, c) ← UNIT(T ∪ Γ, P);
4  |  while c = true do
5  |  |  if P assigns a value to every variable in T then return P;
6  |  |  v ← a variable unvalued by P;
7  |  |  (P, c) ← UNIT(T ∪ Γ, ⟨P, (B(v), ⊥)⟩);
8  |  γ ← any (P, c)-conflict that is a UIP with B ⊨ ¬γ;
9  |  (γ, Γ, ≤) ← ADD(γ, Γ, ≤);
10 |  B ← B|γ≤ ;
11 |  if ⊥ ∈ Γ then return ⊥;
12 |  P ← BACKTRACK(P, γ);
13 |  i ← i + 1
14 return UNKNOWN
```

Proposition A.14 *At every iteration of the loop in Procedure 3.8, the following conditions hold:*

1. $B \models \Gamma \wedge \Gamma_{\leq} \wedge C(P)$.
2. $\Gamma \wedge \Gamma_{\leq} \not\models \neg C(P)$.
3. $T \models \Gamma$.
4. Γ *is acceptable for* $\leq$.

Proof. To see that $B \models C(P)$, note that this is obviously true at the start, where $C(P) = \emptyset$. Then there are three places where it could fail: line 7 where a new variable assignment is made and unit propagated, line 3 where P is extended by unit propagation, and line 10, where B is modified to incorporate the conclusion of γ.

Line 7 is consistent with B by construction. Line 3 does not change $C(P)$, since no new choices are made. On line 10, γ is always unit after the backtrack on line 12, so adding the conclusion of γ to B will not conflict with any choice that survives that backtrack.

That $B \models \Gamma \wedge \Gamma_{\leq}$ throughout is a bit more subtle; we begin by showing that $B \models \gamma \wedge \gamma_{\leq}$ specifically. Clearly $B \models \gamma$, since the bias has been modified to satisfy the conclusion $\gamma^{\leq}$. To see that $B \models \gamma_{\leq}$, note that prior to the adjustment on line 10, $B \models \neg\gamma$ by construction so that $B \models \gamma_{\leq}$. Negating the conclusion $\gamma^{\leq}$ will not change this.

Next, we must show that for any other $\beta \in \Gamma$, $B \models \beta \wedge \beta_{\leq}$ after line 10 is complete.

In order for B to stop entailing a particular learned clause β, the conclusion of β must be $\neg\gamma^{\leq}$, since only $\gamma^{\leq}$ is modified in line 10. But in this case, a resolution will have been performed by ADD and a different γ returned. Thus $B \models \beta$.

In order for B to stop entailing a particular antecedent $\beta_{\leq}$, $\neg\gamma^{\leq}$ must have been included in that antecedent. But in this case, the learned clause in question will have been removed by the ADD procedure. Thus $B \models \beta_{\leq}$ as well. This concludes the proof of the first claim of Proposition A.14.

The second claim follows from this; if $\Gamma \wedge \Gamma_{\leq} \models \neg C(P)$, then B would entail both $\neg C(P)$ and $C(P)$ and would be inconsistent as a result.

To see that $T \models \Gamma$, note first that $T \models \gamma$ in line 8 by the construction of a BUIP. The ADD procedure 3.6 adds to Γ either γ or the result of resolving γ with existing elements of Γ; either way, the clause being added is necessarily entailed by T. The fact that various learned clauses may be removed from Γ clearly does not affect the conclusion that $T \models \Gamma$.

Finally, $\Gamma \wedge \Gamma_{\leq} \not\models \gamma$ by virtue of the facts that $B \models \Gamma \wedge \Gamma_{\leq}$ and $B \models \neg\gamma$. Thus Proposition A.10 can be used to conclude that Γ remains acceptable for $\leq$. ■

Theorem 3.10 *Suppose that T is a theory with n clauses involving v variables. Then:*

1. *Each loop of Procedure 3.8 can be executed in time $O(v^3 + nv^2)$,*
2. *$|\Gamma| \leq v$ as Procedure 3.9 is executed, and*
3. *For any n with $\sum_{i=0}^{n} r(i) \geq 2^v$, Procedure 3.9 will finish before completing iteration n.*

Proof. Since Γ is acceptable with respect to $\leq$, the second claim follows from Proposition A.3. The first claim now follows from Proposition 3.7. The third claim follows from Proposition A.5, which bounds $\mathbf{size}(\Gamma, \leq)$ to be between 0 and 2^v, and Proposition A.12, which ensures that the size decreases on each pass through the loop in Procedure 3.8. ■

References

Bayardo, R. J., & 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*, pp. 203–208.

Beame, P., Kautz, H., & Sabharwal, A. (2004). Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, *22*, 319–351.

Bebel, J., & Yuen, H. (2013). Hard SAT instances based on factoring. In *Proceedings of SAT Competition 2013: Solver and Benchmark Description*, p. 102, Helsinki, Finland.

Beck, J. C. (2005). Multi-point constructive search. In *Proc. of the Eleventh Int'l. Conf. on Principles and Practice of Constraint Programming (CP05)*, pp. 737–741.

Ben-Sasson, E., Impagliazzo, R., & Wigderson, A. (2000). Near-optimal separation of tree-like and general resolution. Tech. rep., Electronic Colloquium in Computation Complexity.

Biere, A., Heule, M. J. H., van Maaren, H., & Walsh, T. (Eds.). (2009). *Handbook of Satisfiability*, Vol. 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Bonet, M. L., & Johannsen, J. (2014). Improved separations of regular resolution from clause learning proof systems. *Journal of Artificial Intelligence Research*, *49*, 669–703.

Bonet, M. L., Pitassi, T., & Raz, R. (1997). Lower bounds for cutting planes proofs with small coefficients. *Journal of Symbolic Logic*, *62*(3), 708–728.

Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., & Colton, S. (2012). A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(1), 1–49.

Buhler, J., Lenstra, H., & Pomerance, C. (1993). *Factoring integers with the number field sieve*, Vol. 1554 of *Lecture Notes in Mathematics*, pp. 50–94. Springer-Verlag.

Buss, S. R., Hoffmann, J., & Johannsen, J. (2008). Resolution trees with lemmas: Resolution refinements that characterize DLL algorithms with clause learning. *Logical Methods in Computer Science*, *4*.

Clark, K. L. (1978). Negation as failure. In Gallaire, H., & Minker, J. (Eds.), *Logic and Data Bases*, pp. 293–322. Plenum, New York.

Davis, M., & Putnam, H. (1960). A computing procedure for quantification theory. *J. Assoc. Comput. Mach.*, *7*, 201–215.

Davis, M., Logemann, G., & Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, *5*(7), 394–397.

Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence*, *12*, 231–272.

Gaschnig, J. (1979). *Performance Measurement and Analysis of Certain Search Algorithms*. Ph.D. thesis, Carnegie-Mellon University.

Ginsberg, M. L. (1993). Dynamic backtracking. *Journal of Artificial Intelligence Research*, *1*, 25–46.

Gomes, C. P., Selman, B., & Kautz, H. (1998). Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, pp. 431–437, Madison, Wisconsin.

Goultiaeva, A., & Bacchus, F. (2012). Off the trail: Re-examining the CDCL algorithm. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT-2012)*, pp. 30–43.

Haken, A. (1985). The intractability of resolution. *Theoretical Computer Science*, *39*, 297–308.

Haken, A. (1995). Counting bottlenecks to show monotone $P \neq NP$. In *Proceedings 36th Annual IEEE Symp. on Foundations of Computer Science (FOCS-95)*, pp. 36–40, Milwaukee, MN. IEEE.

Harvey, W. D. (1995). *Nonsystematic Backtracking Search*. Ph.D. thesis, Stanford University, Stanford, CA.

Hertel, P., Bacchus, F., & Pitassi, T. (2008). Clause learning can effectively P-simulate general propositional resolution. In *Proceedings of the Twenty-Third National Conference on Artificial Intelligence*, pp. 283–290.

Huang, J. (2006). TINISAT in SAT-race 2006..

Huang, J. (2007). The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, pp. 2318–2323, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Kaivola, R., Ghughal, R., Narasimhan, N., Telfer, A., Whittemore, J., Pandav, S., Slobodov, A., Taylor, C., Frolov, V., Reeber, E., & Naik, A. (2009). Replacing testing with formal verification in Intel® Core i7 processor execution engine validation. In Bouajjani, A., & Maler, O. (Eds.), *Computer Aided Verification*, Vol. 5643 of *Lecture Notes in Computer Science*, pp. 414–429. Springer Berlin Heidelberg.

Kocsis, L., & Szepesvári, C. (2006). Bandit based Monte-Carlo planning. In *In: ECML-06. Number 4212 in LNCS*, pp. 282–293. Springer.

Krajiček, J. (1997). Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *J. Symb. Logic*, *62*(2), 457–486.

Lecoutre, C., Saïs, L., Tabary, S., & Vidal, V. (2007). Recording and minimizing nogoods from restarts. *J. on Satisfiability, Boolean Modeling and Computation (JSAT)*, *1*, 147–167.

Luby, M., Sinclair, A., & Zuckerman, D. (1993). Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, *47*, 173–180.

Lynce, I., Baptista, L., & Marques-Silva, J. (2001). Stochastic systematic search algorithms for satisfiability. In *In LICS Workshop on Theory and Applications of Satis Testing*, pp. 1–7.

Marques-Silva, J. P., & Sakallah, K. A. (1999). GRASP – A search algorithm for propositional satisfiability. *Computers*, *48*(5), 506–521.

Marques-Silva, J., Lynce, I., & Malik, S. (2009). *Conflict-Driven Clause Learning SAT Solvers*, Vol. 185 of *Frontiers in Artificial Intelligence and Applications*, pp. 131–153. IOS Press.

McAllester, D. A. (1993). Partial order backtracking. Unpublished.

Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference*.

Moura, L., & Bjørner, N. (2010). Bugs, moles and skeletons: Symbolic reasoning for software development. In Giesl, J., & Hähnle, R. (Eds.), *Automated Reasoning*, Vol. 6173 of *Lecture Notes in Computer Science*, pp. 400–411. Springer Berlin Heidelberg.

Pipatsrisawat, K., & Darwiche, A. (2007). A lightweight component caching scheme for satisfiability solvers. In *Proceedings of 10th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 294–299.

Pipatsrisawat, K., & Darwiche, A. (2011). On the power of clause-learning SAT solvers as resolution engines. *Artif. Intell.*, *175*(2), 512–525.

Pudlak, P. (1997). Lower bounds for resolution and cutting planes proofs and monotone computations. *J. Symbolic Logic*, *62*(3), 981–998.

Purdom, P., & Sabry, A. (2005). CNF generator for factoring problems.. `cgi.cs.indiana.edu/~sabry/cnf.html`.

Ramos, A., Tak, P., & Heule, M. (2011). Between restarts and backjumps. In Sakallah, K., & Simon, L. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2011*, Vol. 6695 of *Lecture Notes in Computer Science*, pp. 216–229. Springer.

Reiter, R. (1978). On closed world data bases. In Gallaire, H., & Minker, J. (Eds.), *Logic and Data Bases*, pp. 119–140. Plenum, New York.

Ryan, L. (2002). Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University.

Sellmann, M., & Ansótegui, C. (2006). Disco – Novo – GoGo: Integrating local search and complete search with restarts. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, pp. 1051–1056.

Selman, B., Kautz, H. A., & Cohen, B. (1993). Local search strategies for satisfiability testing. In *Proceedings 1993 DIMACS Workshop on Maximum Clique, Graph Coloring, and Satisfiability*.

Sörensson, N., & Een, N. (2005). Minisat v1.13 - a SAT solver with conflict-clause minimization. 2005. SAT-2005 poster. Tech. rep..

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

Tseitin, G. (1970). On the complexity of derivation in propositional calculus. In Slisenko, A. (Ed.), *Studies in Constructive Mathematics and Mathematical Logic, Part 2*, pp. 466–483. Consultants Bureau.

Zhang, L. (2003). *Searching for Truth: Techniques for Satisfiability of Boolean Formulas*. Ph.D. thesis, Princeton University, Princeton, NJ.

Zhang, L., Madigan, C. F., & Moskewicz, M. H. (2001). Efficient conflict driven learning in a boolean satisfiability solver. In *In ICCAD*, pp. 279–285.