Extracting (Easily) Checkable Proofs from a Satisfiability Solver that Employs both Preorder and Postorder Resolution

Allen Van Gelder
237 B.E., University of California, Santa Cruz, CA 95064
E-mail avg@cs.ucsc.edu.

October 1, 2001

Abstract

In many applications the desired outcome of satisfiability checking is that the formula is unsatisfiable: A satisfying assignment essentially exhibits a bug and unsatisfiability implies a lack of bugs, at least for the property being verified. Current high-performance satisfiability checkers are unable to provide proof of unsatisfiability. Since bugs have been discovered in many solvers long after being put into service, an uncheckable decision poses a significant problem if important economic or safety decisions are to be based upon it. Current tableau-based systems that are able to produce proofs are unable to process propositional formulas of practical size.

This paper describes modifications of the classical backtracking-search satisfiability algorithm of Davis, Putnam, Loveland and Logemann (DPLL) that are designed to extract checkable proofs of practical length when the formula is believed to be unsatisfiable. It is known that the purely postorder resolution proofs extractable from standard DPLL (called "tree resolution" proofs) are exponentially longer than nontree proofs in the worst case. Experience also shows them to be of impractical length. This paper describes an efficient method to integrate postorder resolution with preorder reasoning methods, including binary-clause reasoning, equivalent-literal identification, and variable-elimination resolution, to produce nontree proofs.

Preliminary experiments show that the resulting proofs are much shorter, but that memory has to be managed extremely carefully and the time is usually longer compared to the best "complete" programs that do not produce proofs.

1 Introduction

We assume the reader is familiar with the *satisfiability problem*, which seeks to determine if any assignment to the propositional variables of a Boolean formula causes it to evaluate to *true*. In recent research, planning problems, hardware and software verification problems and others have been encoded as satisfiability problems. There is a substantial difference among these types of problems, however. For planning problems, the successful outcome is a satisfying assignment, which describes the plan, and is easily checkable. For verification problems, the successful outcome is the *lack of* a satisfying assignment, which is not easily checkable.

Nearly all complete satisfiability solvers are in the DPLL family (for Davis, Putnam, Loveland, and Logemann [DLL62]). They search for a satisfying assignment by fixing variables one by one and backtracking when an assignment forces the formula to be *false*. The procedure is not very effective in its original form, but it has been enhanced with various techniques to reduce the search space. Techniques to choose the branch variable are a separate topic, not treated here. Reasoning techniques can be broadly classified as *preorder* and *postorder*. Preorder techniques are applied as the search goes forward, and include binary-clause reasoning, equivalent-literal identification, and other efficient reasoning steps whose goal is to show that certain variable bindings cannot lead to a satisfying assignment [BS92, Pre95, VGT96, Li00]. (The omission of the unit-clause rule is intentional, as explained in Section 4.) Postorder techniques are applied when the search is about to backtrack, because a "conflict" has been discovered [SS96, Zha97, BS97, MMZ$^+$01]. Postorder techniques are variously called non-chronological backtracking, conflict-directed back-jumping, and learning. These techniques are compared in an upcoming paper [ZMMM01]. In none of these works have preorder and postorder techniques been combined.

This paper describes how preorder and postorder techniques can be combined and presents the results of a preliminary implementation. The difficulty to be overcome is that derived clauses depend on the assumptions (guessed assignments) in a nontransparent way. If the only reasoning is unit resolution, then each derived clause corresponds to one original clause, and the relevant assumptions can be traced through this association [LP92, SS96, VGO99]. With non-unit resolution, a derived clause may be associated with an arbitrary number of original clauses. Our program, called "`2clVER` with CBJ" or simply "`2cl`", constructs a directed acyclic graph to maintain the association.

Unlike any other satisfiability solver based on DPLL, `2cl` maintains the information necessary to output a resolution refutation for an unsatisfiable formula. A key to understanding the correctness of the procedure is that the DPLL algorithm can be viewed dually as a procedure to construct a resolution refutation. The refutation is constructed in a post-order fashion. Conflict-directed back-jumping (CBJ) falls out naturally from this dual view. This view is then enhanced with resolutions performed during the search.

Implementation of the method in C posed challenges in memory management. The auxiliary data structure for the directed acyclic graph can be built with only a constant amount of overhead per operation, but during backtracking large amounts of the structure become garbage. Preliminary experimental results are reported.[1]

2 Notation

In CNF, the formula is a conjunction of clauses and each clause is a disjunction of literals; each literal is a propositional variable x or its negation $\neg x$. We denote a clause as $[q_1, q_2, \ldots, q_k]$ and a formula as $\{C_1, C_2, \ldots, C_m\}$. An empty formula is *true* and [], the empty clause, is *false*. We also define the *tautologous clause* $\top$, which is true under any assignment.

Definition 2.1: (strengthened formula) Let $\mathcal{A}$ be a partial assignment for formula $\mathcal{F}$. The clause $C|\mathcal{A}$, read "C strengthened by $\mathcal{A}$", and the formula $\mathcal{F}|\mathcal{A}$, read "$\mathcal{F}$ strengthened by $\mathcal{A}$", are defined as follows.

1. $C|\mathcal{A} = \top$, if C contains any literal that occurs in $\mathcal{A}$.
2. $C|\mathcal{A} = C - \{q \mid q \in C \text{ and } \neg q \in \mathcal{A}\}$, if C does not contain any literal that occurs in $\mathcal{A}$. This might be the empty clause.
3. $\mathcal{F}|\mathcal{A} = \big\{C|\mathcal{A} \bigm| C \in \mathcal{F}\big\}$; i.e., apply strengthening to each clause in $\mathcal{F}$.

   Usually, occurrences of $\top$ (produced by part (1)) are deleted in $\mathcal{F}|\mathcal{A}$.

The operation $\mathcal{F}|p$ (i.e., $\mathcal{F}|\{[p]\}$) is sometimes called "unit simplification". $\square$

The resolution operation is denoted by res(q, C_0, C_1); it returns the resolvent of clauses C_0 and C_1 with clashing literal q. The resolvent is $(C_0 - [q]) \cup (C_1 - [\neg q])$.

[1] A preliminary stage of this work was presented in the satisfiability workshop at LICS 2001, for which there are no published proceedings.

3 Easily Checkable Proofs

If an important decision is to be taken based on claims that certain statements have been formally verified, there is a need to be able to verify the verifier. It is probably impractical to prove that the solver is bug-free, but if it produces an easily checkable proof, then that *proof* can be verified without addressing the issue of whether the program is bug-free.

Propositional resolution proofs are very easy to check, independently of the program that produced the proof. The proof can be presented as a sequence of "records". The m input clauses are indexed 1 through m in this sequence. After that, each record consists of its index in the sequence, the clashing literal, the two operand clauses (i.e., their indexes), and the resolvent clause. The correctness of the proof can be established merely by applying the definition of the resolution operation to each record in isolation. In theoretical terms, the checking problem is in logspace, a very easy complexity class.

These considerations motivate our study of the problems connected with extracting proofs from the runs of satisfiability solvers.

4 DPLL as Construction of a Refutation

Normally, the classical branching algorithm of Davis, Putnam, Loveland, and Logemann (DPLL) [DP60, DLL62] is viewed as a backtracking search for a satisfying assignment for a Boolean CNF formula. It can be sketched as a recursive procedure with parameters $\mathcal{F}$ and $\mathcal{A}$, the formula and the assumptions (guessed assignments, represented as a set of literals):

DPLL($\mathcal{F}$, $\mathcal{A}$)

If $\mathcal{F}|\mathcal{A}$ has no clauses:
    output "sat by $\mathcal{A}$" and **terminate**.
If $\mathcal{F}|\mathcal{A}$ contains an empty clause:
    return "unsat".

(Otherwise) Choose a splitting literal q.
Call DPLL($\mathcal{F}$, $\{\mathcal{A}, \neg q\}$).
Call DPLL($\mathcal{F}$, $\{\mathcal{A}, q\}$).
Return "unsat".

The top-level call is DPLL($\mathcal{F}$, $\emptyset$). For implementation efficiency, the first parameter is normally $\mathcal{F}|\mathcal{A}$, rather than $\mathcal{F}$. Two important observations are:

1. *Unit-Clause Rule*: Note that the unit-clause rule is incorporated in the above sketch by choosing q to be a unit clause if $\mathcal{F}|\mathcal{A}$ contains any such.

2. *Pure-Literal Rule*: Note that the pure-literal rule is incorporated in the above sketch by *not* choosing q if it is a pure literal (unless all remaining variables are pure literals).

Thus the procedure incorporates all of DPLL.

If the formula is satisfiable, the pseudocode outputs and terminates, rather than returning back out of the recursion. This is not recommended for actual implementation, but it simplifies the presentation to focus on the processing of unsatisfiable formulas.

A dual view of this procedure is that it is constructing a resolution refutation. The procedure returns a *resolution tree* whose *root* contains the clause derived by the tree and whose *leaves* are clauses in $\mathcal{F}$. Resolution trees are denoted by P_0 and P_1. Each internal node contains the resolvent clause of its two children.

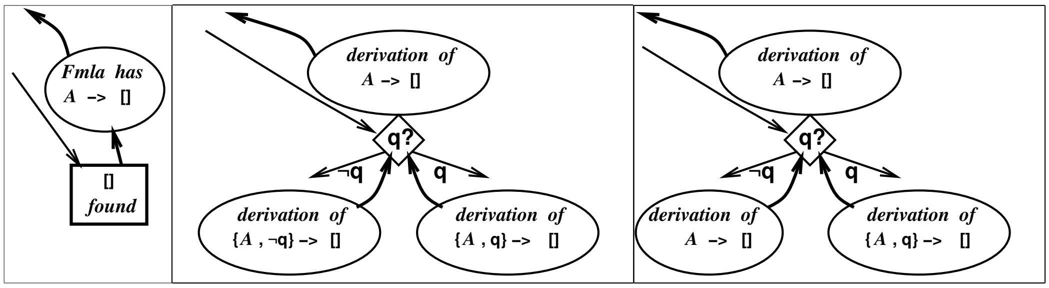


Figure 1: Extracting a resolution refutation from DPLL (as Refute). Note that the implication $\mathcal{A} \rightarrow []$ is equivalent to a disjunctive clause. In general, the antecedents are *subsets* of A, $\{A, \neg q\}$, and $\{A, q\}$, rather than the entire sets. Left panel is the base case; middle panel shows resolution, the usual case; right panel applies when a missing clashing literal prevents resolution.

Refute($\mathcal{F}$, $\mathcal{A}$)

If $\mathcal{F}|\mathcal{A}$ has no clauses:
  output "sat by $\mathcal{A}$" and **terminate**.
If $\mathcal{F}|\mathcal{A}$ contains an empty clause:
  return a clause of $\mathcal{F}$ that became empty.

(Otherwise) Choose a splitting literal q.
$P_0 =$ Refute($\mathcal{F}$, $\{\mathcal{A}, \neg q\}$).
$P_1 =$ Refute($\mathcal{F}$, $\{\mathcal{A}, q\}$).
Return the tree for res(q, *root*(P_0), *root*(P_1)).

We confine this discussion to $\mathcal{F}$ being unsatisfiable. The objective of Refute($\mathcal{F}$, $\mathcal{A}$) is to return the derivation of a clause C such that $\neg C \subseteq \mathcal{A}$. (We are being loose about notation, regarding $\neg C$ as a set of literals and using $\{\mathcal{A}, q\}$ to denote the addition of q to the set $\mathcal{A}$.) If Refute($\mathcal{F}$, $\mathcal{A}$) always achieves its objective, then, since $\mathcal{A}$ is empty in the top level call, the value returned to top level is a derivation of the empty clause.

It is clear that Refute($\mathcal{F}$, $\mathcal{A}$) *does* achieve its objective in the nonrecursive case, where it encounters an empty clause. Otherwise, if both recursive calls meet *their* objectives, then $\neg$*root*$(P_0) \subseteq \{\mathcal{A}, \neg q\}$ and $\neg$*root*$(P_1) \subseteq \{\mathcal{A}, q\}$. By the definition of resolution, $\neg$ res(q, *root*(P_0), *root*(P_1)) $\subseteq \mathcal{A}$.

The only gap in the above argument is that possibly *root*(P_0) does not contain q, so that resolution with q as the clashing literal is not defined. But then $\neg$*root*$(P_1) \subseteq \mathcal{A}$, so Refute($\mathcal{F}$, $\mathcal{A}$) can simply return P_0 and meet its objective. Similarly, if *root*(P_1) does not contain $\neg q$, then Refute($\mathcal{F}$, $\mathcal{A}$) can return P_1. With these added details the algorithm is correct. Figure 1 illustrates the ideas.

Now we observe that if *root*(P_0) does not contain q, then the call Refute($\mathcal{F}$, $\{\mathcal{A}, q\}$) is unnecessary. The right half of the refutation tree can be pruned and the left half becomes the whole tree. Thus conflict-directed back-jumping is built into this algorithm!

In other words, $\neg$*root*(P_0) is a conflict set for $\mathcal{F}| \{\mathcal{A}, \neg q\}$. If *root*$(P_0)$ does not contain q, then $\neg$*root*(P_0) is also a conflict set for $\mathcal{F}|\mathcal{A}$, and the assumption $\neg q$ was irrelevant to the former formula being unsatisfiable.

5 Incorporating Resolution on the Way Down

We now consider an enhanced version of Refute($\mathcal{F}$, $\mathcal{A}$) in which some resolution steps may be carried out prior to the recursive call. The new procedure is Refute($\mathcal{F}$, $\mathcal{D}$, $\mathcal{A}$), where $\mathcal{D}$ denotes a set of derived clauses. Also, Δ will denote a set of clauses derived after Refute($\mathcal{F}$, $\mathcal{D}$, $\mathcal{A}$) begins.

Refute($\mathcal{F}$, $\mathcal{D}$, $\mathcal{A}$)

If $\mathcal{F}|\mathcal{A}$ has no clauses:
    output "sat by $\mathcal{A}$" and **terminate**.
If $\mathcal{F}|\mathcal{A}$ contains an empty clause:
    return a clause of $\mathcal{F}$ that became empty.
If $\mathcal{D}|\mathcal{A}$ contains an empty clause:
    return a proof tree for a clause of $\mathcal{D}$ that became empty.
(Otherwise) Derive additional clauses Δ.
If $\Delta|\mathcal{A}$ contains an empty clause:
    return a proof for a clause of Δ that became empty.

(Otherwise) Choose a splitting literal q.
P_0 = Refute($\mathcal{F}$, $\{\mathcal{D}, \Delta\}$, $\{\mathcal{A}, \neg q\}$).
If q is not in *root*(P_0):
    return P_0.
(Otherwise) P_1 = Refute($\mathcal{F}$, $\{\mathcal{D}, \Delta\}$, $\{\mathcal{A}, q\}$).
If $\neg q$ is not in *root*(P_1):
    return P_1.
(Otherwise) Return the tree for res(q, *root*(P_0), *root*(P_1)).

As long as each clause in $\mathcal{D}$ or Δ is derived by resolution, its derivation can be represented by a resolution DAG (directed acyclic graph) whose edges point to earlier-derived clauses or original clauses. Such DAGs are returned in the nonrecursive case of Refute($\mathcal{F}$, $\mathcal{D}$, $\mathcal{A}$). Thus the structure returned is still a valid resolution proof, but it is not necessarily a tree.

The program `2cl` has extensive preorder reasoning, much of it based on binary clauses, as previously reported [VGT96]. Its performance has been improved recently through more efficient data structures. The next section describes a significant new feature that has been incorporated since the cited paper.

6 Variable-Elimination Resolution

The operation that we call Variable-Elimination Resolution (VER) was used in the Davis-Putnam *resolution* procedure, where it was called "elimination of an atom" [DP60]. If x is the variable to be eliminated, all resolutions involving clauses containing x or $\neg x$ are performed; then all clauses containing x or $\neg x$ are eliminated from the new formula. It is easy to show that $\mathcal{F}$ is satisfiable if and only if VER$(x, \mathcal{F})$ is satisfiable, for any choice of x. Note that the pure literal rule is a special case of VER in which the set of resolvents is empty.

The program `2clVER` is `2cl` enhanced with VER. VER is applied according to an heuristic formula that evaluates the trade-off between reducing the number of variables and increasing the overall formula length. At the point where a variable would be selected to branch on, `2clVER` considers both branching and performing VER. Both choices lead to eliminating one free variable from the formula. Branching results in two subproblems with shorter formulas in each. VER results in one subproblem, often with a longer formula.

The trade-off is estimated with a simple cost model. For formulas with the same number of variables, we assume the cost of solution is proportional to $e^{\alpha L}$, where L is formula length. The estimate uses these parameters, where q is the literal that would be selected for branching, and x is the literal that would be selected for VER.

R = increase in formula length after VER on x;
P = decrease in formula length after binding q to be positive;
N = decrease in formula length after binding q to be negative.

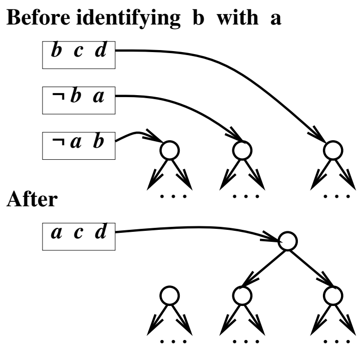


Figure 2: Resolution-DAG nodes need to be separate from their clauses because the clauses can disappear.

Increases and decreases are estimations and R might be negative. VER should be more efficient than branching when $e^{\alpha(L+R)} < e^{\alpha(L-P)} + e^{\alpha(L-N)}$. After dividing out $e^{\alpha L}$, we assume the exponents are small and use one term of Taylor expansion, rather than evaluate transcendental functions: $1 + \alpha R < 1 - \alpha P + 1 - \alpha N$. Finally, the rule is:

Select VER when $R < (1/\alpha) - (P + N)$.

Based on experience, the default value of α is 0.0039 and $1/\alpha = 256$. This means the program chooses to let the formula grow by some 200 literals rather than split it into two subproblems. The program user can change the value on the command line.

7 Finding Conflict Sets in Practice

Section 5 gives us the theoretical basis for constructing a resolution refutation of a given formula $\mathcal{F}$. Each derived clause that is returned by Refute($\mathcal{F}$, $\mathcal{D}$, $\mathcal{A}$) is a conflict set for $\{\mathcal{F}, \mathcal{D}\}\,|\mathcal{A}$. These conflict sets can be used to prune unnecessary backtracking. A conflict set $\neg P$ can be represented very nicely as a descending-order list of the depths at which the literals of $\neg P$ were assumed or guessed. Once conflict sets are materialized, they can be combined in time that is linear in their combined length.

The main problem is materializing the conflict set for the resolution DAG returned in the nonrecursive case. Notice that this is a relatively simple matter for Refute($\mathcal{F}$, $\mathcal{A}$) because the returned value is always the trivial DAG consisting of a clause of $\mathcal{F}$, say C, such that $\neg C \subseteq \mathcal{A}$. For Refute($\mathcal{F}$, $\mathcal{D}$, $\mathcal{A}$) it would not be practical to materialize every derived clause in case it happened to be useful. Instead, if D is a conceptually derived clause, only $D|\mathcal{A}$ is materialized and two pointers are maintained to the ancestor clauses that were resolved to produce $D|\mathcal{A}$.

Let us describe the *dependency DAG*, which is the skeleton of the resolution DAG, in more detail. Every assumed literal has a DAG node that is a source, and the literal is stored in this DAG node. Source nodes have no ancestors. There is also a *true source*. Clauses of the original formula have the true source as their DAG node. Each derived clause has a DAG node with two ancestors, which identify how this clause was derived; i.e., the ancestors are the DAG nodes of the clauses that were resolved to obtain this clause. In the data structures pointers are from clauses to DAG nodes and from DAG nodes to their parents; i.e., toward the sources. There is no way to discover what clause a nonsource DAG node corresponds to by following pointers. However, following pointers from a certain clause *does* lead to the assumed literals that were used to derive that clause. The idea is illustrated in Figure 2.

Suppose $D|\mathcal{A}$ is the empty clause. To reconstruct D it suffices to traverse the DAG rooted at $D|\mathcal{A}$ and collect all the reachable assumptions. This is accomplished efficiently with depth-first search. (Efficiently does not mean inexpensively.) The justification is that every literal that occurs in a clause (not an assumption) that participated in the derivation of D either survives in D or was resolved away during the derivation. If $\neg q$ is in $\mathcal{A}$ and is reached in the DAG, then q must be in an ancestor-clause of D and also in D. In fact, the ancestors collected comprise a conflict set for $\{\mathcal{F}, \mathcal{D}\} |\mathcal{A}$.

To include equivalent-literal identification in the reasoning scheme we proceed as follows. To replace b by a after $a = b$ has been derived, an old clause, say $[b, c, d]$ becomes a new clause $[a, c, d]$. The two pointers for $[a, c, d]$ point to $[b, c, d]$ and $[\neg b, a]$, as shown in Figure 2. The latter must exist because we have no way to derive $a = b$ without having that clause *if we limit our reasoning operations to resolution*. This is the method we implemented. If more sophisticated equivalent-literal identification schemes are used, then more complicated data structures might be needed to track the relevant assumptions.

8 Asserting Postorder Lemmas

If S is a conflict set derived as in the previous sections (regarded as a conjunction of literals), then $\neg S$ is a clause that has been derived from $\mathcal{F}$ and may be added to $\mathcal{F}$ without changing the set of satisfying assignments for $\mathcal{F}$. We call $\neg S$ a *postorder lemma* to distinguish it from clauses derived while the search is "going forward". In principle, conflict sets can be used for conflict-directed back-jumping whether or not their clause is asserted permanently; whether it is practical to do so might depend on the implementation.

The reported satisfiability solvers that use CBJ and have been successful on large problems all have the capability to assert postorder lemmas [SS96, Zha97, BS97, MMZ$^+$01]. The policy for asserting and retracting these lemmas is an heuristic. Usually the user selects values (or accepts the defaults) for several parameters.

The program reported in this paper follows a simple policy because its primary goal is to extract a short proof. It asserts all postorder lemmas that were used at least once for proof construction and were not immediately subsumed in their parent node. (Note that "trivial" postorder lemmas consisting of the negations of all the assumed literals must be subsumed by it parent.) Once asserted, postorder lemmas are never retracted.

The overhead of managing the postorder lemmas is kept reasonable by using an extension of the *variable-watching* strategy introduced in `Chaff` [MMZ$^+$01]. Due to the presence of equivalent-literal replacements, if v is a literal that is being watched in postorder lemma L, then L must appear on the list for any literals that transitively replaced v. Details on how to accomplish this efficiently are reported elsewhere, due to lack of space.

9 Experimental Results

We tested a preliminary implementation of the method described in this paper on some large formulas from planning applications, pigeon-hole formulas, and random 3CNF formulas. The random and pigeon-hole formulas provide families of formulas with similar characteristics and varying sizes, to measure how program performance scales.

The basic program (`2cl`) [VGT96] is a DPLL-style solver with 2-closure reasoning, which means that all binary and/or unit clauses that can be derived from other binary and/or unit clauses are materialized. Equivalent-literal identification is also carried out, and variable-elimination resolution (VER) is used according to the heuristic described in Section 6. Our main question is how long are the refutations found by the program for unsatisfiable formulas. Timing is also measured.

Since no other program of which we are aware maintains the information needed to construct a proof, we cannot offer meaningful comparisons of proof size. The output produced by the copy of `Chaff2` obtained from the web page does not report the number of unit clauses, so we lack information to estimate the size of a proof associated with its computation. We observe that `2clVER` does 2 to 10 times less branching than `Chaff2` on the unsatisfiable Satplan

Table 1: Proof sizes of `2cl` on three unsatisfiable Satplan formulas. Formula sizes are after simplification.

Problem	Deadline	Vars	Clauses	Lits	`2clVER` Proof Size	Sol'n time (secs.)
logistics.c	12	1482	6974	18,244	16,747	4
bw_large.c	13	4405	29280	66,547	14,706	207
bw_large.d	17	8545	82320	184,180	1,048,465	36,493

Table 2: Proof sizes on pigeon-hole formulas.

n	fmla	Resolution steps in Refutation	
pigeons	size	`2clVER`	ideal `DPLL`
4	48	48	42
5	100	175	188
6	180	951	970
7	294	5,876	5,862
8	448	58,516	41,090
9	648	474,015	328,792
10	900	4,000,947	2,959,218

formulas studied. However, this may not be closely correlated with proof size, especially since `Chaff2` does random restarts. Other programs have similar difficulties.

Three unsatisfiable planning formulas from the Satplan family [KS96] are shown in Table 1. The proofs are rather small for the sizes of the formulas. The program took an inordinate amount of time to find them, however: `Chaff2` solved `bw_large.d-17` in only 54 seconds. All CPU times are based on a Sun Ultra 10 with 440 MHz clock.

For the family of Pigeon-Hole formulas, we have a theoretical model against which to compare `2clVER`. An ideal DPLL constructs a "tree resolution" refutation with t_n steps for the formula for n pigeons and $(n-1)$ holes, where

$$t_n = (n-1)! \left(\frac{2}{(n-2)!} + 3 \sum_{j=0}^{n-3} \frac{1}{j!} \right) \approx 3e(n-1)!$$

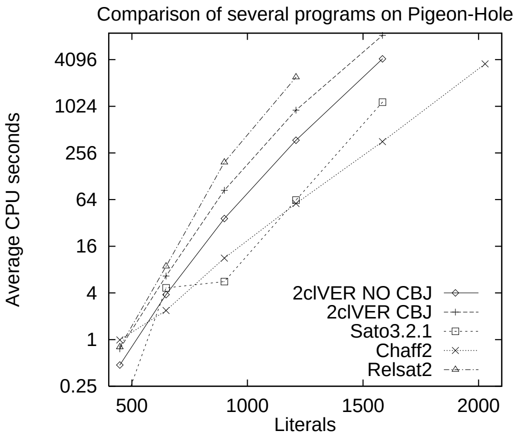
(That is, the ideal DPLL search can be translated into such a refutation, as described in Section 4.) Table 2 shows the results for `2clVER` and the ideal for DPLL. The result is quite disappointing in that `2clVER` is apparently wandering around with its preorder reasoning, finding circuitous proofs that lengthen, rather than shorten, the refutation.

Table 3 shows the average proof sizes for unsatisfiable random 3CNF formulas with clause-variable ratio of 4.27, which is believed to be close to the hardest ratio. Fifty samples were tested for each size and the satisfiable ones were discarded, since they obviously do not a refutation. The growth rate is about $2^{0.008L}$ in this range.

For reference purposes we also tested several other programs that use CBJ: Sato3.2.1 [Zha97], Relsat2 [BS97],

Table 3: Proof sizes on unsatisfiable random 3CNF formulas.

vars	proof length	ln(pf len)	fmla len (L)	samples
141	40,033	10.60	1806	21
168	113,226	11.64	2151	27
200	560,013	13.24	2562	29
238	2,531,890	14.74	3048	28



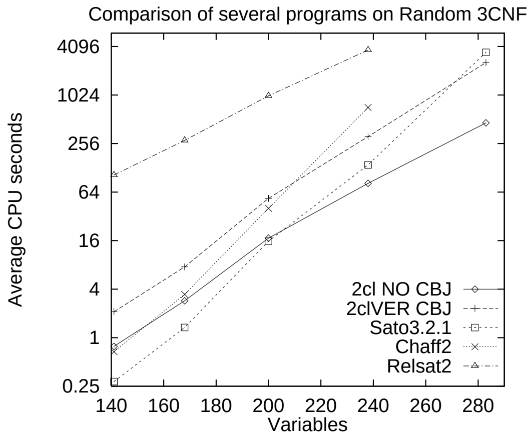


Figure 3: Growth rates for CPU time on various programs with CBJ: pigeon-hole formulas for 8–13 pigeons (left) and random 3CNF formulas at ratio 4.27 (right).

and Chaff2 [MMZ$^+$01]. For sato we used the default settings; for relsat we used learn-order 4 and fudge-factor 0.9, as recommended; for chaff we used the default settings, except on the random formulas, where we used alternative settings suggested by one of the program authors. The somewhat older version of Grasp [SS96] that we have ran considerably slower. All of the programs have several runtime parameters and it is possible that different parameter setting would produce much different results. Results are summarized in Figure 3. On semi-log scales, straight lines with slope α represent functions proportional to 2^α.

On random formulas we observe that both sato and chaff are much faster for small formula sizes but grow considerably faster than 2cl and relsat. On pigeon-hole formulas, both sato and relsat are growing both sato3.2.1 and relsat2 are growing somewhat faster than 2cl. However, chaff is growing much slower than the other programs, unlike its behavior on random formulas. Chaff is the only program that finished 13 pigeons in the time available.

10 Conclusion

We have described a method to combine preorder and postorder resolution-based reasoning, as enhancements to the basic DPLL procedure. The method includes equivalent-literal identification and variable-elimination resolution, besides binary-clause resolution. Postorder lemmas can be asserted and are tracked with an extension of the watched-literal technique. Careful bookkeeping permits a refutation to be extracted when the formula is unsatisfiable. Preliminary experiments indicate that the idea is feasible, but much work remains to be done. We believe this is the first program that has the possibility to deliver a proof of unsatisfiability in an independently checkable format.

Acknowledgments

This work was supported in part by NSF grants CCR-9505036 and CCR-9503830.

References

[BS92] A. Billionnet and A. Sutter. An efficient algorithm for the 3-satisfiability problem. *Operations Research Letters*, 12:29–36, July 1992.

[BS97] R. J. Bayardo, Jr. and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 203–208, 1997.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[KS96] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *13th National Conference on Artificial Intelligence*, pages 1194–1201, 1996.

[Li00] C. M. Li. Integrating equivalency reasoning into davis-putnam procedure. In *AAAI*, 2000.

[LP92] S.-J. Lee and D. A. Plaisted. Eliminating duplication with the hyper-linking strategy. *Journal of Automated Reasoning*, 9(1):25–42, August 1992.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference*, June 2001.

[Pre95] D. Pretolani. Efficiency and stability of hypergraph SAT algorithms. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1995.

[SS96] J. P. Silva and K. A. Sakallah. GRASP–a new search algorithm for satisfiability. In *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design*, pages 220–227. IEEE Comput. Soc. Press, 1996.

[VGO99] A. Van Gelder and F. Okushi. Lemma and cut strategies for propositional model elimination. *Annals of Mathematics and Artificial Intelligence*, 26(1–4):113–132, 1999.

[VGT96] A. Van Gelder and Y. K. Tsuji. Satisfiability testing with more reasoning and less guessing. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1996. (also at `ftp://ftp.cse.ucsc.edu/pub/avg/kclose-tr.ps.Z`).

[Zha97] H. Zhang. SATO: An efficient propositional prover. In *14th International Conference on Automated Deduction*, pages 272–275, 1997.

[ZMMM01] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, Nov. 2001.