Persistent and Quasi-Persistent Lemmas in Propositional Model Elimination

Fumiaki Okushi (okushi@acm.org)
Electronics for Imaging, 303 Velocity Way, Foster City, CA 94404, U.S.A.

Allen Van Gelder (avg@cse.ucsc.edu)*
Department of Computer Science, SOE, University of California, Santa Cruz, CA 95064, U.S.A.

Abstract. Model elimination is a back-chaining strategy to search for and construct resolution refutations. Recent extensions to model elimination, implemented in *Modoc*, have made it a practical tool for satisfiability checking, particularly for problems with known goals. Many formulas can be refuted more succinctly by recording certain derived clauses, called *lemmas*. Lemmas can be used where a clause of the original formula would normally be required. However, recording too many lemmas overwhelms the proof search. Lemma management has a significant effect on the performance of Modoc. Earlier research studied pure persistent (global) strategies, and pure unit-lemma (local) strategies. This paper describes and evaluates a hybrid strategy to control the lifetime of lemmas, as well as a new technique for deriving certain lemmas efficiently, using a lazy strategy. Unit lemmas are recorded locally as in previous practice, but certain lemmas that are considered valuable are asserted globally. A range of functions for estimating value is studied experimentally. Criteria are reported that appear to be suitable for a wide range of application-derived formulas.

Keywords: Modoc, Model Elimination, satisfiability, resolution, lemmas, refutation.

AMS: 68T15, 03B35

1. Introduction

Model Elimination is a goal-sensitive (or back-chaining) strategy to search for and construct resolution refutations [16, 17]. Classical Model Elimination has a major performance problem in the propositional domain due to search redundancy [23]. The recent *autarky pruning* extension to propositional Model Elimination produces exponentially better performance in theory and improvements by factors exceeding 10^6 in practice, even on small formulas, such as 20 variables and 90 clauses [26, 27]. Further improvements in lemma mechanisms have made Model Elimination with autarky pruning a practical tool for satisfiability checking, particularly for formulas with known "goals",

* Corresponding author.

and have been implemented in a program named Modoc [28, 29]. Some familiarity with the terminology of resolution refutation is assumed.

It is known that many formulas can be refuted more succinctly in the Model Elimination framework if the procedure is able to record certain derived clauses, then use them where an input clause (a clause of the original formula) would normally be required [14]; such recorded clauses are called *lemmas*. However, recording too many lemmas overwhelms the proof search in practice.

The first reported use of lemmas in Model Elimination, by Fleisig *et al.*, was to assert them permanently and possibly use them in *extension* (input resolution) operations, although only one literal in each clause might be used as the clashing literal [8]. This paper and all early papers on Model Elimination studied it on first-order formulas. Fleisig *et al.* write about the lack of selection rules and the increase in the number of eligible extension clauses that need to be tried. Overall, they found lemmas to be "detrimental". Shostak writes that lemma clauses tend to be highly redundant because they are often subsumed by other lemma clauses and/or clauses from the formula [24].

However, later studies showed that by either limiting the types of lemmas asserted, or limiting the lifetime of lemmas, lemmas could be beneficial to the search [24, 14, 1]. One method was Shostak's C-literal strategy [24]. Instead of asserting the lemma permanently, it embeds the consequent of the lemma in the appropriate place in the derivation, where all literals except the last have been resolved away; the surviving literal is called the *C-literal*, and the other literals of the derived lemma clause are called *dependency literals*. Recently, additional methods specialized to propositional formulas were introduced, and found to be beneficial [28].

This paper examines two extensions to earlier reported lemma strategies. One strategy is concerned with lemma propagation. When one lemma is derived, how can it be used to derive additional lemmas? The strategy uses the idea of unit propagation to derive the additional lemmas efficiently. However, it takes a deferred approach in that only the effect of the lemma is recorded and the computation of the exact lemma, namely the dependency literals, is deferred until it becomes absolutely necessary. This amounts to savings in computation cost.

The other strategy examines the effect of converting certain C-literals (with their dependency literals) into *persistent lemmas*, that is, asserting them permanently. This permits a lemma to be available even after Shostak's C-literal mechanism would discard it. When a subrefutation is successful, Modoc records the fact by asserting a lemma in the form of a unit clause (Shostak's C-literal strategy), conditioned on the premise in which it was successful. When the premise no longer holds (because the search has moved back out of the premise), the lemmas that pass through a "filter" will be converted to a persistent lemma. We have implemented these ideas in Modoc and report on their effect on the search procedure.

A different technique for deriving lemmas, which are also persistent, has been examined by Sakallah and Silva [25], and by Zhang [30] for model-searching satisfiability testers. (Both papers use the terminology of intelligent or non-chronological backtracking, rather than lemmas.) We believe the current paper is the first to examine persistent lemmas in the context of propositional refutation search.

1.1. Motivation for Back-Chaining

Backward-chaining search, also called goal-sensitive search, proceeds from a clause (or a set of clauses) that represents a conjectured theorem, attempting to prove that theorem. In the resolution context, the clause or clauses actually encode the negation of the conjectured theorem, and a refutation is sought.

Many real-world problems can be viewed as theorem-proving problems. A formula that is derived from such a problem consists of *axioms*, which are obviously consistent, and the negated conjectured theorem (as *theorem clauses*), which is the possible source of inconsistency. Backward-chaining search attempts to focus on the possible causes of inconsistency by starting refutation attempts from such theorem clauses. This capability is expected to be important as formulas generated automatically from real-world problems will have a large number of irrelevant clauses.

Modoc is the only program for propositional formulas that uses back-chaining, as far as we are aware, that has been reported in the literature. Most reported methods are some version of the DPLL method, due to Davis, Putnam, Loveland, and Logemann [5, 4], and no way to make this goal sensitive has been reported.

1.2. Terminology and Notation

Our terminology and notation follow that commonly used in the satisfiability literature. We list below some of the possible diversions and concepts that may be nonstandard.

DEFINITION 1. Formulas are in conjunction normal form (CNF), so clauses are disjunctions of literals and formulas are conjunctions of clauses. Formulas and clauses are expressed using set notation with some similarities to Prolog notation. For the sake of clarity, square brackets ('[' and ']') are used to enclose the literals of a clause, while the clauses of a formula are grouped with curly braces ('{' and '}'). Double negations of literals cancel.

In some cases, it is convenient to use "←", which can be read "if" in clauses. The notation is referred to as the *implication form.* As in Prolog, the literals to the right of "←" are considered to be conjunctively joined antecedants of an implicational formula. If a clause contains no "←", it is considered to be at the extreme right. Let a_i and c_j be literals (not just variables). Then

$$[c_1, \ldots, c_m \leftarrow a_1, \ldots, a_n]$$

denotes the formula $(c_1 \vee \cdots \vee c_m \vee \neg a_1 \vee \cdots \vee \neg a_n)$.

Literals to the left of "←" are called *active* literals and those to the right are called *antecedant* literals. □

EXAMPLE 2. The following boolean formula in conjunctive normal form

$$(u \vee v) \wedge (u \vee \neg v) \wedge (\neg u \vee w) \wedge (\neg u \vee \neg w) \wedge (\neg u \vee \neg v \vee x) \wedge (\neg u \vee v \vee \neg x)$$

can be expressed as

$$\{[u, v], [u, \neg v], [\neg u, w], [\neg u, \neg w], [\neg u, \neg v, x], [\neg u, v, \neg x]\}.$$

To remove v as an active literal throughout this formula, it can be converted into the antecedant literal $\neg v$, rewriting the formula as:

$$\{[u \leftarrow \neg v], [u, \neg v \leftarrow], [\neg u, w \leftarrow], [\neg u, \neg w \leftarrow], [\neg u, \neg v, x \leftarrow], [\neg u, \neg x \leftarrow \neg v]\}.$$

Notice that $\neg v$ continues to appear as an active literal in some clauses. □

EXAMPLE 3. If a particular literal appears as an active literal in one clause and an antecedant literal in another clause, the two clauses can be resolved, eliminating both occurrences of that literal. For example, $[a \leftarrow \neg b]$ and $[a, \neg b \leftarrow c, d]$ resolve to produce $[a \leftarrow c, d]$. □

DEFINITION 4. A *partial truth assignment* is a partial function from propositional variables into $\{\mathsf{true}, \mathsf{false}\}$. It is expressed using set notation, as the exact set of literals that are true under the assignment. □

EXAMPLE 5. The set $\{a, \neg b, c\}$ represents the partial truth assignment ν that is formally defined by:

x	a	b	c	otherwise
$\nu(x)$	true	false	true	undefined

□

DEFINITION 6. A *satisfying truth assignment* of a formula is a partial truth assignment for which each clause in the formula has at least one literal that is true under the assignment. Note that we do not require a satisfying truth assignment to be total. □

1.3. Organization of the Paper

Section 2 briefly reviews Modoc. Section 3 describes secondary eager lemmas and Section 4 describes persistent lemmas and the filtering strategy used. Experimental results are summarized in Section 5. The paper concludes with Section 6.

2. The Modoc Algorithm—An Overview

The Modoc algorithm is based on propositional Model Elimination. As such, the aim of Modoc is to find a refutation, demonstrating that the formula is inconsistent. However, if no refutation exists, Modoc develops a partial assignment that is witness to that fact. This section reviews the main ideas.

2.1. Propositional Derivation Trees

In Modoc, a refutation is embodied in a *refutation tree*, and the progress of the construction is described by a *propositional derivation tree* (PDT). The tree format for Model Elimination was mentioned by Loveland [16] and was further developed by Minker and Zanon [18]. Letz, Mayr, and Goller recognized the relationship to tableaux and further developed the tree format [14]. Modoc specializes the tree format for propositional reasoning [26].

Modoc (without lemmas) tries to construct a refutation tree using two operations: *PDT extension* and *PDT reduction*. As with other Model Elimination procedures, Modoc employs lemma strategies to help it avoid repeating certain refutations. However, using only the two basic operations, a subgoal or clause is *refuted* when it is the root of a tree or subtree that has no subgoals as leaves.

EXAMPLE 7. Figure 1 shows a formula on the left and an example propositional derivation tree (PDT) that might be developed on the right. The root of the PDT is called the *verum* and the top clause is $[a, b]$, with subgoals a and b. The clause $[\neg a, c]$ is used for *extension* at the left subgoal a, creating a new subgoal c. Then $[\neg a, \neg c]$ is used for extension at c.

Input Clauses
a, b
$a, \neg b, c$
$\neg a, c$
$\neg a, \neg c$
$\neg b, \neg c$

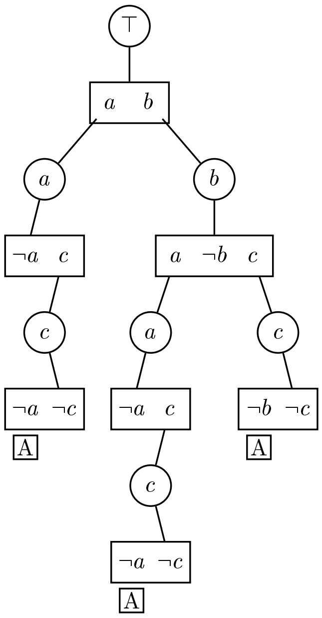


Figure 1. Propositional derivation tree for Example 7.

The notation $\boxed{\text{A}}$ under the $\neg a$ of this clause indicates that $\neg a$, which would otherwise be a new subgoal, is removed by a *reduction* operation. Reduction occurs when the complement of that literal is an ancestor goal node. (The literal that is the complement of the immediate parent is not marked.)

Similarly the right subtree of the top clause is built by means of four extensions and two reductions. This tree is also a refutation tree for the formula because there are no leaf subgoals. □

2.2. Search Trees

To build a refutation tree, Modoc *selects* a subgoal leaf for extension, then *chooses* a clause with which to perform the extension. After attaching that clause as the child of the subgoal, Modoc applies any possible reductions to literals of the chosen clause; any remaining literals become new subgoals. The distinction between selection and choice is that selection of the subgoal need not be backtracked over, whereas unsuccessful choices of a clause for extension must be backtracked over to achieve completeness.

We can associate a *search tree* with Modoc's actual computation, which is different from the PDT. In fact, the nodes of the search tree are themselves PDTs, representing a state of the refutation process. The edges represent the choices for extension at the selected subgoal. If there are no subgoal leaves to select, the search node is a success leaf. If there are no possible extensions for the selected subgoal, the search node is a failure leaf.

It is helpful to remember that for Modoc the search fails and needs to backtrack when it cannot construct a *refutation*, whereas most satisfiability methods need to backtrack when they cannot construct a *model*. However, in this paper we loosely refer to all of a program's processing time as "search time".

Additionally, Modoc employs autarky pruning, which prunes certain branches of the search tree (not the PDT) that cannot lead to a successful refutation. An *autarky* for a formula $\mathcal{F}$ is a partial assignment M and a subset $\mathcal{A}$ of the clauses of $\mathcal{F}$ such that M satisfies $\mathcal{A}$ and does not affect the clauses $\mathcal{F} - \mathcal{A}$ [20]. *Autarky pruning* consists of removing $\mathcal{A}$ from consideration for constructing a refutation of $\mathcal{F}$. That is, any potential edges of the search tree that represent extension by some clause in $\mathcal{A}$ are eliminated. Autarky pruning significantly reduces the search time for propositional Model Elimination [26, 27]; however, a deep understanding of this topic is not necessary to understand the technical contents of this paper.

2.3. Quasi-Persistent Lemmas

The basic lemma mechanism of Modoc is called the *quasi-persistent lemma* strategy. It is based on the C-literal strategy [24], adapted and enhanced for use in the propositional domain. The *ancestors* of subgoal p are those subgoals on the path in the PDT from the top clause to p, and include p; the *proper ancestors* exclude p. When a subgoal p has been refuted in a PDT, a lemma of the form

$$[\neg p \leftarrow a_1, \ldots, a_m]$$

has been derived, where $a_1, \ldots, a_m$ are precisely those proper ancestors of p that were used for reductions in the subtree rooted at p. The literal $\neg p$ is called a *C-literal*. Consider the subtree rooted at the lowest of these ancestors (i.e., furthest from the root), say a_r. The key idea is that everywhere in this subtree, the derived lemma functions as a unit clause. If p becomes a subgoal, it can be extended by the lemma, then subgoals $\neg a_1, \ldots, \neg a_m$ are immediately reduced. The net effect is indistinguishable from applying reduction to p, so this operation is called *C-reduction* [24]. The C-literal strategy consists of "attaching" the lemma to the subgoal a_r, using $\neg p$ as though it were an additional ancestor within the subtree of a_r, and ignoring it outside that subtree.

The quasi-persistent strategy consists of preserving this lemma even if computation backtracks from the search node in which the lemma was derived, as long as the node backtracked into also has a_r as an ancestor [26].

A difference between Shostak's C-literal strategy and Modoc's quasi-persistent lemma strategy is in the latter's possible persistence after failed subrefutations. The quasi-persistent lemma strategy retains the lemma even if the (attempted) subrefutation in which the lemma was derived is abandoned, provided that the goal to which the lemma is *attached* is not abandoned. For example, if the lemma is derived at depth 50 and is attached at depth 30 (because all ancestor nodes relevant to the derivation of the lemma occurred at depth 30 or less), and later the attempted subrefutation at depth 40 has to be abandoned, then the lemma persists. Previous practice, geared toward first-order clause sets, was to abandon the lemma in this situation. This paper investigates under what conditions it seems to be profitable to retain the lemmas permanently.

2.4. Pre-Reduction and Eager Lemmas

To focus on appropriate clauses for extensions, as well as to help implement certain lemma strategies, Modoc implements *pre-reduction* as soon as a subgoal leaf, say p, is selected for extension. Pre-reduction can be thought of as a bookkeeping operation that converts all active occurrences of $\neg p$ throughout the formula into antecedant occurrences of p. The *ancestors* of subgoal p are those subgoals on the path in the PDT from the top clause to p, and include p. Since all of these ancestors were selected earlier, pre-reduction has occurred with all of them.

DEFINITION 8. A clause is *active* if none of its active literals is an ancestor subgoal. Otherwise, the clause is *inactive*. A clause is *eligible* for extension at the selected subgoal p if it is active and contains p as an *antecedant* literal. The properties of being active, inactive, or eligible are relative to current selected subgoal. □

Modoc also implements *C-pre-reduction* when a C-literal q is attached. This is like ancestor pre-reduction, except that active occurrences of $\neg q$ are transferred to antecedant occurrences of q. However, C-literals do not affect eligibility for extension, for technical reasons explained elsewhere [28].

In previously reported work, C-pre-reduction was not used to its fullest capability, because an efficient implementation was not known. This paper extends the capability of C-pre-reduction, which in turn increases the power of the eager lemma strategy, reviewed next.

Because of pre-reduction we know that if a clause is chosen for extension, then exactly its active literals will become subgoals; all others will be subject to reduction (or C-reduction), since they are complements of ancestors (or C-literals).

EXAMPLE 9. Suppose Modoc is in the middle of a refutation and the current form of some clause is $[b, \neg d, \neg e \leftarrow \neg a]$. This indicates that $\neg a$ is already an ancestor or C-literal. Now, if subgoal d is selected, then d is added to the set of ancestor literals, and pre-reduction changes the implication form of this clause to $[b, \neg e \leftarrow \neg a, d]$. $\square$

Pre-reduction, as we have described it, is somewhat similar to unit resolution often used in model-search procedures, if we associate the antecedant literals with the unit clauses that were assumed or derived. Just as unit resolution can be propagated when new unit clauses are derived, pre-reduction can be propagated when clauses are transformed to have only one remaining active literal.

DEFINITION 10. If a clause has one active literal after pre-reduction as a result of some subgoal selection, the one remaining active literal is called an *E-literal* and the clause is called an *eager lemma*. In addition, if a clause has one active literal after pre-reduction as a result of creating another E-literal, the one remaining active literal is called an *E-literal* and the clause is called an *eager lemma*. E-literals and C-literals are collectively called *lemma literals*. $\square$

Effectively, an E-literal can be added to the set of ancestors and used for further pre-reductions. The key observation is that an appropriate clause *could* be derived by resolution to have the same active literals as the clause that is pre-reduced by the E-literal. An example makes this more concrete.

EXAMPLE 11. Suppose $[q, \neg p \leftarrow a, b]$ and $[r, \neg q \leftarrow c]$ are active in the formula when p is selected as a subgoal. Literals a, b, and c are already ancestors. Pre-reduction by p produces $[q \leftarrow a, b, p]$. Now q is an E-literal and pre-reduction by q produces $[r \leftarrow c, q]$.

Although not all antecedant literals of the latter clause are ancestors, it is possible to *derive* a clause in which r is the only active literal and all the antecedant literals are ancestors. Simply resolve $[q \leftarrow a, b, p]$ and $[r \leftarrow c, q]$ to yield $[r \leftarrow a, b, p, c]$. Thus the eager lemma $[r \leftarrow c, q]$ can be thought of as an indirect representation of the derived clause $[r \leftarrow a, b, p, c]$. $\square$

As with C-literals, pre-reduction with an E-literal shifts literals from active to antecedant, but does not influence eligibility. Eager lemmas are attached to the current selected subgoal in the PDT.

2.5. Additional Lemma Strategies

Apart from the quasi-persistent and eager lemma strategies, Modoc incorporates several other mechanisms to derive additional lemmas.

1. When there are two active clauses whose respective implication forms are $[a \leftarrow b_1, \ldots, b_m]$ and $[\neg a \leftarrow c_1, \ldots, c_n]$, then the ancestor to which the lower of the lemma literals a and $\neg a$ are attached is refuted. This is called the *lemma-induced cut* rule.

2. When the attachment of a lemma literal causes the implication form of a clause to change to $[\leftarrow b_1, \ldots, b_n]$, then the lowest ancestor in $\{b_1, \ldots, b_n\}$, or the lowest ancestor to which a lemma literal in $\{b_1, \ldots, b_n\}$ is attached, whichever is lower, is refuted. This is called the *C-reduction-induced cut* rule.

The logical basis for the lemma strategies reviewed in Sections 2.3 through 2.5, including proofs, can be found elsewhere [28, 21].

This paper explores two additional lemma strategies. One is the derivation of additional lemmas by propagating the effects of C-pre-reduction, after a C-literal is attached to the PDT. These lemmas are called *secondary eager lemmas* and are described in Section 3. Another is the retention of certain lemmas permanently, called the *persistent lemma* strategy, and is described in Section 4.

3. Secondary Eager Lemmas

The lemma strategies reviewed in the previous section do not cover all opportunities for lemma derivation. One case that had not been pursued was the case in which an E-literal is derived *after* a C-literal is attached to the PDT.

DEFINITION 12. When a C-literal is attached to the PDT and causes an active clause's implication form to change to $[p \leftarrow a_1, \ldots, a_n]$ for some literals $p, a_1, \ldots, a_n$, where $n \geq 1$, a *secondary eager lemma* is derived, and p is called the *secondary E-literal.*

As with E-literals, the clause $[p \leftarrow a_1, \ldots, a_n]$ is an indirect representation of the lemma. The actual lemma clause is defined by tracing back through any antecedant literals that are not ancestors to the ancestor literals.

When necessary to distinguish the secondary eager-lemma strategy from the original eager lemma strategy, we will refer to the original eager lemma strategy as the *primary* eager-lemma strategy. Secondary E-literals, C-literals and primary E-literals are collectively called *lemma literals.* □

Unlike the E-literals found by the primary eager-lemma strategy, the attachment point of a secondary E-literal is not immediately known. This is because the pre-reductions that caused a secondary E-literal to be derived can also come from below the ancestor to which the C-literal is attached. Contrast this to the case of a primary E-literal, where the pre-reduction can only come from either the same depth or above. Thus, to determine the correct attachment point of a secondary E-literal, all the dependency literals must be considered, similar to the case when a quasi-persistent lemma is derived.

EXAMPLE 13. Figure 2 shows an example of a secondary eager lemma derivation. To avoid clutter, parts of the derivation that are not germane to the discussion are omitted.

The figure shows that ancestor subgoal e (dashed circle) has just been refuted and C-literal $\neg e$ (thick square) has been attached to ancestor subgoal $\neg k$. The C-literal $\neg e$ starts a chain of pre-reductions (solid-line arrows) deriving the secondary E-literals g, h, $\neg f$, $\neg i$, c, and $\neg d$ (double-line squares). The attachment points of these secondary E-literals are discussed in Example 15. □

THEOREM 14. *Suppose the selected subgoal q has just been refuted and a secondary eager lemma is derived during the subsequent propagation of C-pre-reductions. That is, an active clause's implication form is now $[a \leftarrow b_1, \ldots, b_n]$ for some literals $a, b_1, \ldots, b_n$, where $n \geq 1$. The attachment point of the secondary E-literal a is the lowest ancestor in $\{b_1, \ldots, b_n\}$, or the lowest ancestor to which some lemma literal in $\{b_1, \ldots, b_n\}$ is attached, whichever is lower.*

Proof. We order all the lemmas attached to ancestors of q, first by depth, then within depth by the sequence in which they were derived. Then the antecedant literals in each lemma clause are either ancestors (at the same or lesser depth as the lemma clause) or they are active literals (C-, E- or secondary E-literals) of lemmas earlier in the order. Beginning with the clause $[a \leftarrow b_1, \ldots, b_n]$, repeatedly select an antecedant literal that is latest in the order just defined, say b_i, and resolve the current clause with the lemma whose active literal is b_i. Continue until all antecedants of the resolvent are ancestor literals. The lowest of these antecedants is a valid attachment point.

EXAMPLE 15. This example continues from Example 13. The attachment point of the secondary E-literal h is the lowest among the C-literal $\neg e$ and the ancestor subgoal $\neg k$. Thus, h is attached to ancestor $\neg k$.

The attachment point of the secondary E-literal $\neg i$ is the lowest among the the C-literal $\neg e$ and the lemma literal $\neg m$, which might

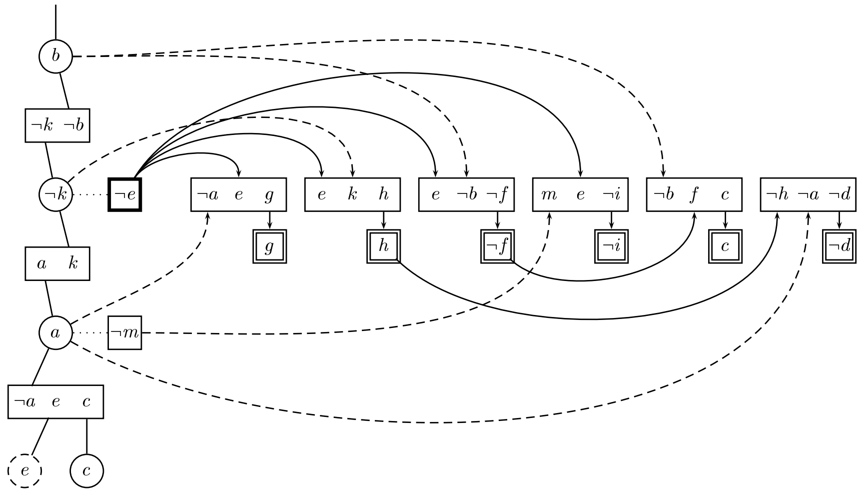


Figure 2. An example of a secondary eager lemma derivation. We assume that subgoal e (dashed circle) has just been refuted and that the C-literal $\neg e$ (thick square) is attached to ancestor subgoal $\neg k$. Dashed-line arrows indicate earlier pre-reductions. The secondary eager-lemma strategy performs a chain of pre-reductions (solid-line arrows), deriving the secondary E-literals g, h, $\neg f$, $\neg i$, c, and $\neg d$ (double-line squares).

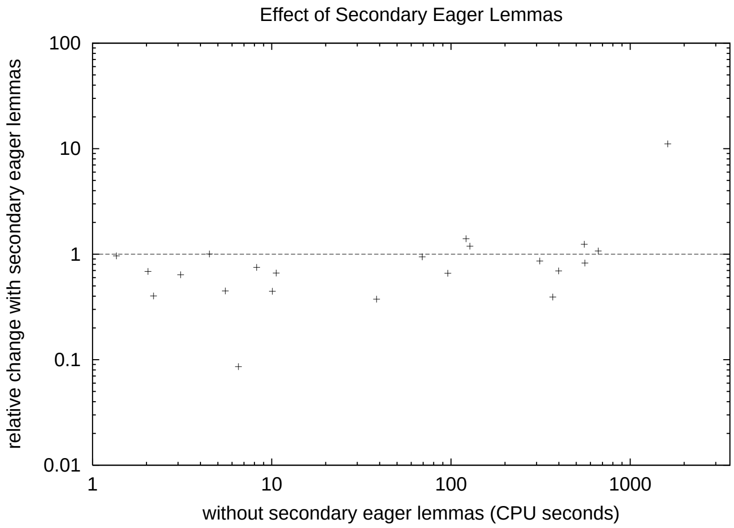
be a C-literal, E-literal, or secondary E-literal. Thus, $\neg i$ is attached to ancestor a. Similarly, $\neg d$ is attached to ancestor a.

Notice that all secondary eager lemmas derived as a result of the C-literal $\neg e$ must attach to ancestors in the range from a (the parent of the refuted subgoal) to $\neg k$ (the attachment point of the C-literal $\neg e$ that triggered the chain of C-pre-reductions). □

To assess the benefit of secondary eager-lemma (2el) strategy, we ran `modoc` (an implementation of Modoc) on several sets of formulas. More information on these formulas can be found in Section 5.

The first formula set was composed of 141 Medic formulas, using a 30-minute time limit. There were 22 "interesting" formulas—those that took at least one second and were finished by at least one of the two runs (i.e., one run with secondary eager lemmas and one without) within 30 CPU minutes. Six were rejected because of dual timeouts and 113 were rejected for taking less than one second. With today's computer speeds, taking less than one second does not mean the formula is trivial; up to 300,000 Modoc operations might be performed in one CPU second. Still, we think people don't care which fraction of a second a program takes on one formula when the next one might take hours.

Population: Medic planning formulas



Times are CPU seconds on an UltraSparc workstation (440MHz). The horizontal line in the middle shows the point of "no change". Plots below the line indicate formulas for which the secondary eager-lemma strategy had a beneficial effect. Note that both axes use log scale.

Figure 3. Effect of the secondary eager-lemma (2el) strategy on `modoc` search times.

Figure 3 shows the ratio of search times for `modoc`, with and without the secondary eager-lemma strategy, for the interesting formulas. This graph shows that secondary eager lemmas were usually beneficial, but considerably detrimental in a few cases.

As Figure 3 shows, the difference in performance without and with secondary eager lemmas is highly variable. To compare the performances more formally, we need a method that provides for programs to time out on some formulas.

DEFINITION 16. A program's *CPU time per solution* for a sample is the total time taken on *all* formulas in the sample divided by the number of formulas *solved*. That is, formulas on which the program times out or otherwise fails do not contribute to the denominator. If the program solves all formulas in the sample, this time is the same as the average. □

With this sample of 22, we obtained the following statistics:

Population: BMC formulas

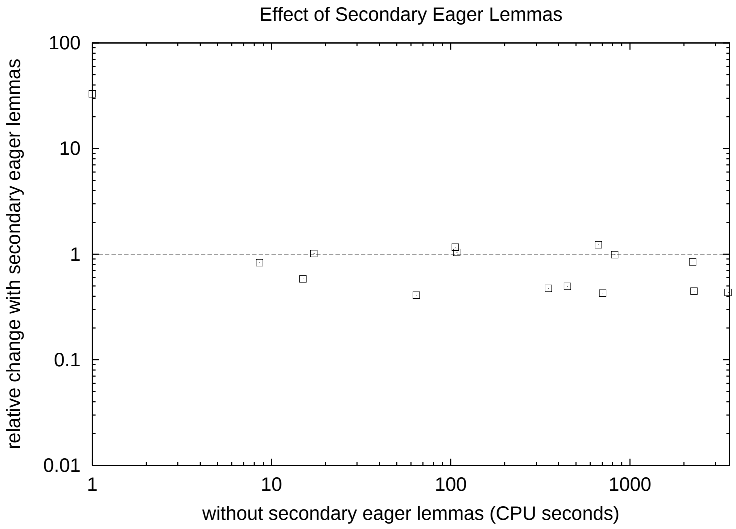


Figure 4. Effect of the secondary eager-lemma (2el) strategy on `modoc` search times. Further explanation may be found in Figure 3.

	without 2el	with 2el	difference
cpu secs. per solution	226	231	–5
standard error	80	88	17

The program with secondary eager lemmas timed out on one formula in this sample, so its CPU time per solution is the total time taken on all 22 formulas, divided by 21. The "without 2el" program finished all 22, so it has 22 in the denominator. For this sample, the difference in CPU time per solution is not statistically significant, being only about 1/3 of the standard error of the difference.

The second formula set was composed of 34 bounded-model-checker (BMC) formulas, using a one-hour time limit. There were 15 "interesting" formulas—those that took at least one second and were finished by at least one run within one hour (see Figure 4). Eleven were rejected because of dual timeouts and eight were rejected for taking less than one second. With this sample of 15 we obtained the following statistics:

	without 2el	with 2el	difference
cpu secs. per solution	756	472	284
standard error	276	156	150

Population: Circuit formulas

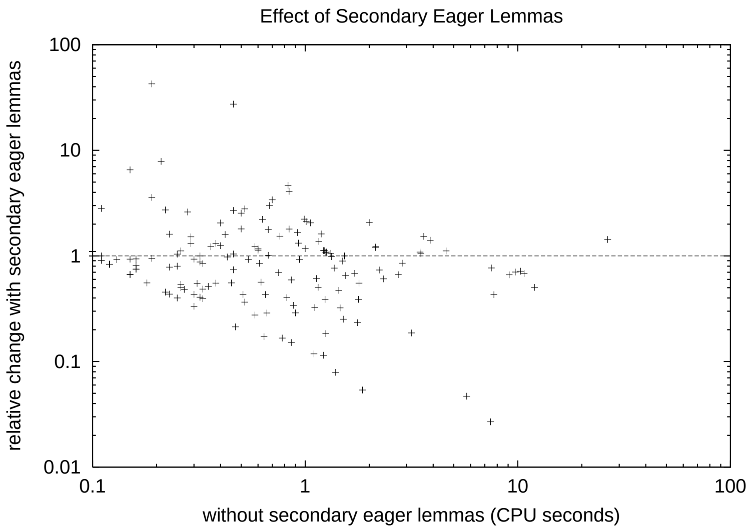


Figure 5. Effect of the secondary eager-lemma (2el) strategy on `modoc` search times. Further explanation may be found in Figure 3.

Although the average savings for 2el (about 38 percent), looks impressive, the high values of the standard error (the same as the standard deviation of the mean) show that this difference might be explained by random variations. However, the standard error of the formula-by-formula *difference* in CPU times is considerably less than the average difference. Despite the small sample, the difference is significant at the 95-percent level by the Student pairs t-test.

The strategy was also tested on three additional formula sets, namely, the circuit formula set, the 141-variable random formula set, and the 200-variable random formula set. All problems were solved. The results show two extreme pictures. On the circuit formula set (Figure 5), the ratios are scattered over a wide range and clearly trend downward as formula difficulty increases. On the other hand, the random formula sets (Figures 6 and 7) show the extreme opposite. A much more consistent improvement is evident and it does not vary with formula difficulty. The statistics are summarized below. Circuit formulas that were solved in less than one second by both programs were discarded before compiling the statistics.

Population: Random 141-variable formulas

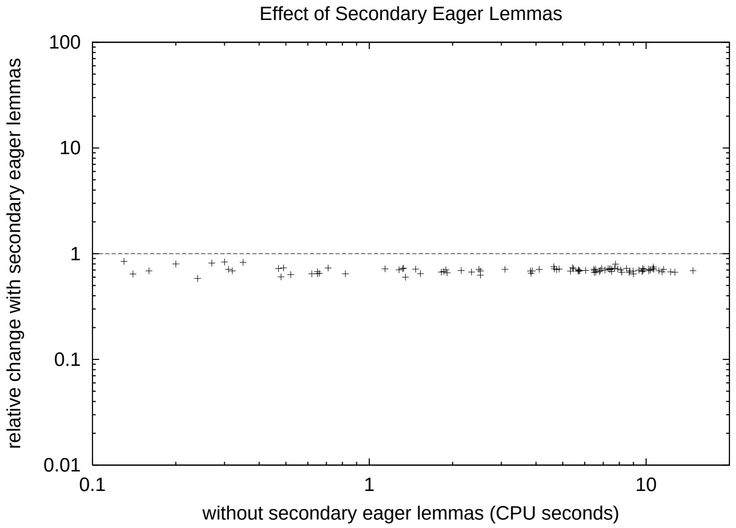


Figure 6. Effect of the secondary eager-lemma (2el) strategy on `modoc` search times. Further explanation may be found in Figure 3.

Population: Random 200-variable formulas

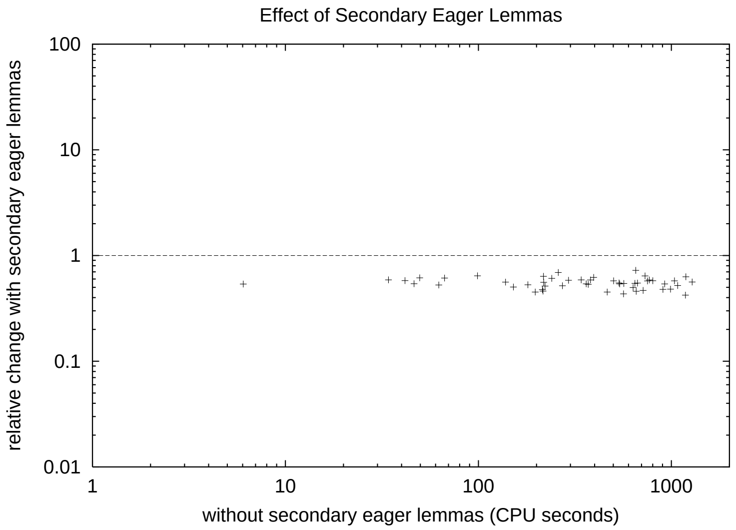


Figure 7. Effect of the secondary eager-lemma (2el) strategy on `modoc` search times. Further explanation may be found in Figure 3.

formula set	sample size	cpu secs. per soln. without 2el	with 2el	difference	standard error of difference
circuit	71	1.78	1.82	−0.04	0.34
141-var. random	100	4.86	3.40	1.46	0.12
200-var. random	50	477	260	217	24

On the circuit formula set, there is no significant difference in using secondary eager lemmas, whereas on the random formulas there are sizable gains and the difference is statistically significant at the 99 percent level.

In conclusion, the benefit of secondary eager lemmas appears to depend strongly on the characteristics of the formula. For random formulas, it appears to guarantee significant improvement. For the circuit formula set it appears neutral on average. For other formula sets, the benefit appears to differ from one formula to another with wild variations. At this time we do not have any insight about what conditions cause this, but we conjecture that it is an interaction of some kind with the way that eligible clauses are ordered for possible extensions. In the next section we find an interesting connection between secondary eager lemmas and persistent lemmas.

4. Persistent Lemmas and Filtering Function

The original policy of asserting lemmas permanently resulted in an unfavorable evaluation of the effectiveness of lemmas [8]. This has led to other studies where the extent of lemmas are controlled either by their kind or by their lifetime. The C-literal strategy controls the lifetime of lemmas, as does our quasi-persistent lemma strategy.

This section describes a strategy for converting certain C-literals and their associated quasi-persistent lemmas into persistent lemmas. When a C-literal is about to be discarded, because the computation backs up from the goal node to which the C-literal is attached, a filtering function is applied to the associated quasi-persistent lemma. If the lemma passes through the filter, it is converted into clause form and asserted permanently.

A technique that is equivalent to deriving and recording lemmas has been described by Sakallah and Silva [25], in the context of a model-searching satisfiability program, such as that of Davis, Putnam, Loveland and Logemann (DPLL). They do not use the term “lemma” but their description indicates that they are recording lemmas that are derivable from the DPLL search. Their method, called *Grasp*, incorporates a mechanism described in terms of non-chronological backtracking [25]. (This is also called intelligent backtracking, dependency-directed backtracking, conflict-directed back-jumping, etc.) The same idea was broached by Lee and Plaisted much earlier, but with few details [13]. The main idea is that a *conflict set* is a set of literals $\{q_1, \ldots, q_k\}$ that cannot all be true simultaneously in any possible model of the formula. In this case, the clause $[\leftarrow q_1, \ldots, q_k]$ is a lemma. Note that this clause is expressed in implicational form, entirely with antecedant literals (see Definition 1).

Further empirical results on the Grasp idea are reported by Zhang [30] with the program *Sato3*, which also incorporates a very efficient subsumption algorithm. In both cases, the programs simply retain all derived clauses up to a user-specified length. The default length is 20, based on empirical optimization by Sakallah and Silva on a subset of the circuit formulas mentioned in Section 5.

4.1. A Model of Lemma Value

All work of which we are aware simply uses some constant cutoff value for the length of an acceptable lemma (whether it is called a lemma, a conflict set or other term). For example, in Grasp and Sato3 the cutoff is set to 20, meaning that lemmas with more than 20 literals are rejected; the cutoff is user adjustable, but little is known about how to choose a good value. This section proposes a simple model for estimating the value of a lemma, based on both its length and the actual cost of deriving it. The analysis uses many gross approximations and simplifying assumptions.

We model the computation as a DPLL-style binary tree. (Modoc is more complicated, but we need a very simple model.) At each node some variable is chosen as the branching literal. Along the edge to the left child that literal is fixed to true and other variables may get fixed by unit propagation and other reasoning. Along the edge to the right child the branching literal is fixed to false and other variables may get fixed similarly. All variables fixed on edges are considered to be ancestors for this discussion, not just the branching literals. To make the model tractable, we assume that a total of α variables get fixed on each edge and that the tree is balanced with depth n/α, where the formula has n variables. (This is an overestimate of the depth, but we will see that the result is not very sensitive to this assumption.) We also assume that all free variables are equally likely to get fixed on a given edge.

Suppose we have a formula $\mathcal{F}$ and some tree T that refutes it, as just outlined. Consider some clause with L literals, $Q = [\leftarrow q_1, \ldots, q_L]$, written in implication form, so Q contains only antecedant literals (see Definition 1). Suppose that the literals $q_1, \ldots, q_L$ occur along some branch of T, not necessarily in that order. Then we call Q a lemma and we define the *cost* of deriving Q to be the number of nodes in a maximal subtree S such that all of Q's antecedant literals are proper ancestors of S. (If there is more than one candidate for S, choose the leftmost.)

Without worrying about where Q came from, we can ask how many branches of T can be expected to contain all of $q_1, \ldots, q_L$. We say the lemma Q is *enabled* on such branches. On each such branch, if Q were an additional known clause, then T could be pruned of the subtree in which all of the antecedant literals of Q are proper ancestors. We can also ask how many nodes can be expected to be pruned in this way.

The answers to the questions in the previous paragraph can be obtained (at least approximately) with standard methods of probability and recurrent events. Since our only purpose is to arrive at a filtering function, not to prove any properties of proofs or algorithms, we omit the details. The tree T was assumed to have about $2^{n/\alpha}$ nodes and edges. The expected number of branches on which Q is enabled is

$$\left(\frac{\alpha}{ne^L}\right) 2^{n/\alpha},$$

where e is the natural log base, 2.718.... The expected number of nodes that can be pruned due to Q having no active literals is $\left(b^L\right) 2^{n/\alpha}$, where $b = (e-1)/e \approx 0.632$.

We estimate the cost of carrying an extra clause Q with L literals as

$$A\left(\frac{\alpha L}{n}\right) 2^{n/\alpha}$$

on the reasoning that the chance that a specific literal is processed on a random edge is α/n and Q has L literals. Here A is some constant that depends on the implementation. Carrying Q corresponds to making it persistent in Modoc.

Suppose we have just derived the lemma Q at cost C (the number of nodes in the subtree S mentioned above, which the algorithm has counted). If we assume that each time Q is enabled the resulting subtree will have C nodes, then the total expected cost of constructing these subtrees (without carrying Q) would be

$$\left(\frac{C\alpha}{ne^L}\right) 2^{n/\alpha}$$

whereas if we carry Q these subtrees cost zero. With this assumption, Q is worth carrying if

$$C > ALe^L.$$

An alternative assumption is that each time Q is enabled the resulting subtree will have a random size as predicted by the model, and the total expected size of all such subtrees would be the cost without carrying Q. Then we should carry Q if its estimated carrying cost is lower, that is,

$$A\left(\frac{\alpha L}{n}\right)2^{n/\alpha} < \left(b^L\right)2^{n/\alpha}.$$

This holds if $L\log(1/b) + \log(L) < \log(n) - \log(A\alpha)$. Any convenient base can be used for the logs and $1/b \approx 1.582$. Under this assumption the cost of deriving Q is irrelevant to the decision on whether to carry it; only the length matters. Moreover, the best cutoff length varies about logarithmically with the number of variables in the formula, so a simple constant cutoff should be suitable for a group of "similar" formulas.

We experimented with several filtering strategies motivated by this very approximate analysis. These are described next, in Section 4.2.

4.2. Filtering Strategies

As filtering strategies for Modoc, we considered several strategies that try to determine the "worthiness" of the lemma. The first strategy takes into consideration the "cost" (to be explained later) of deriving the lemma, and the estimated cost of retaining it, as discussed in Section 4.1, leading to the filter function $f(L) = A\,L\,e^L$ (see Equation 1 in Section 5.2). The quasi-persistent lemma is converted into a persistent lemma if (and only if) its derivation cost is at least $f(L)$.

In this study, the cost was defined as the number of goal nodes extended in the refutation subtree that derived the quasi-persistent lemma. Intuitively, this is roughly proportional to the size of the proof. Note that this is different from the number of goal nodes extended during the proof *search*, as the latter includes goal nodes whose refutation attempt later turned out to fail.

The second strategy is simply to keep lemmas up to some constant length, as was done in the cited papers. Although it would be easy to make the cutoff vary with the log of the number of variables, as suggested by the alternative assumption in Section 4.1, regression analysis of relative improvement vs. number of variables with various cutoffs did not indicate that this would be an improvement.

For the first strategy, we experimented with varying values of A; for the second we tried cutoffs ranging from two to six and observed that performance was degrading sharply by this time. We also investigated combinations of the two strategies. We report on our findings in Section 5.2.

4.3. Incorporation of Persistent Lemmas into Proof Search

Modoc exploits persistent lemmas in two ways. The obvious way is to use them for extensions, like additional input clauses. However, there might be some concern that these extra clauses will just provide more alternatives to bog down the search, as they have done in earlier reports [8]. A more restricted use is also possible in Modoc, which is to use the persistent lemmas only if they complete a refutation, by losing all of their active literals through pre-reduction.

For Modoc, an argument against using persistent lemma clauses in extension operations would be to keep the bookkeeping cost down, as well as to keep the branching factor of the search tree down. An argument for using persistent lemma clauses would be that since these clauses embody successful subproofs, their use may shorten the overall proof. There is no apparent way to evaluate this trade-off except through empirical studies. Our experience has been that the bookkeeping cost is quite minor, in the neighborhood of a few percent. We have also observed that using persistent lemmas in extensions normally results in substantially fewer operations overall.

We believe that an implementation choice in `modoc` prevents weak clauses (those with numerous literals) from bogging down the search in most cases. Such clauses are often subsumed by shorter clauses. Recall that propositional clause C subsumes clause D if $C \subseteq D$. (A notable difference between Grasp and Sato3 is that Sato3 has a very efficient implementation of subsumption tests.) In `modoc`, eligible clauses are ordered by increasing length at each subgoal. So if C subsumes D and both are eligible, C will be tried in an extension first. If using C leads to a refutation, D will not be tried. If, on the other hand, C *fails* to be refuted, it produces an autarky literal that is one of the literals in C, hence in D. A property exploited in autarky pruning is that any clause that contains an autarky literal cannot lead to a successful refutation [26]. Thus, although `modoc` has no explicit test for subsumption, autarky pruning ensures that clause D will not be tried in this case, either.

5. Experimental Results

To assess the benefit of the strategy described in Section 4, we added it to `modoc` (which denotes a specific implementation of Modoc), and tested it by means of experiments. The experiments involved running `modoc` with the feature either turned on or off on a large collection of formulas from different domains.

Obtaining statistically significant experimental results for lemmas is difficult. Lemmas seem to become important as formulas get difficult. We found that conclusions reached by studying easy formulas did not necessarily scale up to more difficult formulas. For example, the popular "DIMACS circuit formula" benchmarks, used by numerous researchers, including ourselves, proved to be too easy. They behaved much differently with respect to lemmas than harder formulas, so we were obliged to construct a set of benchmarks with formulas ranging up to 100 times more difficult than any of the circuit formulas. The difficulty level of the benchmark suite limits the amount of experimentation that is possible.

5.1. Formula Sets

Table I summarizes the formula sets used in the experimental study. Further detail is provided in the remainder of this subsection.

Three formula sets were generated from various planning problems. A planning formula encodes a particular planning problem for a particular deadline and has the property that it is satisfiable if and only if there is a plan (i.e., sequence of actions) that achieves the goals of the problem on or before the deadline. An optimal plan is one that completes within the tightest deadline; thus it is necessary to detect unsatisfiability to prove optimality. The goals can be thought of as the negation of a conjectured theorem that the problem is *not* solvable, and their clauses are called *theorem clauses* [29]. The back-chaining aspect of Modoc comes into play by using the theorem clauses as top clauses.

The first formula set is a collection of 154 formulas generated using a SAT compiler called *Medic* [6]. A SAT compiler takes a description of a planning problem that requires a sequence of actions and generates the corresponding formula for a specific deadline. Medic takes a planning problem described in STRIPS [7] and can generate a formula in a number of different encodings. We used the encodings named ecse, efst, and erpe, which are among the more efficient encodings. The problems include block-world, logistics, towers of Hanoi, monkey and banana, flat tire, and fridge fixing.

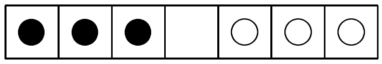
Seven formulas were eliminated for being orders of magnitude harder than the rest; 12 others were eliminated due to encoding errors or

Table I. Formula sets used in the experiments. Further details are given in Section 5.

formula set	no. of fmlas	application	description, citation, URL
Medic	154	planning	Generated from SAT compiler *Medic* [6]. Problems include block-world, logistics, towers of Hanoi, monkey and banana, flat tire, and fridge fixing.
		ftp://ftp.cs.washington.edu/pub/ai/medic.tar.gz	
		ftp://ftp.cs.washington.edu/pub/ai/domains.tar.gz	
Satplan	6	planning	Generated from SAT compiler *Satplan* [10] (modified to identify goal clauses). Problems include logistics and block world.
		ftp://ftp.research.att.com/dist/ai/	
Checker interchange	4	planning	Encodes a one-player version of the Chinese Checker game.
		ftp://ftp.cse.ucsc.edu/pub/avg/Modoc	
BMC	34	model checking	Unsatisfiable formulas generated by the bounded model checker *BMC* [3]. Formulas verify assertions about barrel shifter, long multiplier, and queue invariance.
	http://www.cs.cmu.edu/∼modelcheck/bmc/bmc-benchmarks.html		
circuit	325	ATPG	Automated test-pattern generation (ATPG) formulas generated from ISCAS-85 benchmark circuits [11]. Formulas simulate “single stuck-at” and “bridge” faults.
		ftp://ftp.cse.ucsc.edu/pub/avg/Modoc/cnf-bf,ssa	
Simplified versions			Formulas were simplified in a manner that preserves top clauses [29]. These versions, and the simplifier, are available at
		ftp://ftp.cse.ucsc.edu/pub/avg/Modoc	
		ftp://ftp.cse.ucsc.edu/pub/avg/2cl_simpDir/	
random	100	none	3-CNF formulas generated at the hard ratio of 4.27 with 141 or 200 variables.

simplifying to the empty clause. The number of variables ranged from 30 to 14025, with the average at 1908. The number of clauses ranged from 30 to 102551, with the average at 3231. Eventually, another 107 were dropped for compiling statistics because they always took less than one CPU second, leaving a core of 28 “interesting” formulas.

The second formula set is called the Satplan formula set and is a collection of six formulas generated using a SAT compiler called *Satplan*



The goal of the checker-interchange problem is to interchange the positions of the white and black checkers through a series of moves and jumps, somewhat like Chinese checkers. White checkers may move left, or jump left over a black checker, into an empty space. Black checkers have the opposite capabilities.

Figure 8. Checker-interchange problem

[10]. Unlike Medic, Satplan takes a planning problem described in a language special to the program. The sizes of the formulas are shown in Table III, with run-time data.

The third formula set is called the checker-interchange formula set and is a collection of four formulas. Checker-interchange formulas encode a single-player game, which is illustrated in Figure 8 for the 3-checker case. The problem is interesting in that is is believed to have only one solution (by always starting with the black checker to break the symmetry), regardless of the number of checkers. These formulas were generated using a special-purpose script. The size of the formulas are shown in Table IV, with run-time data. This family of problems has some similarities to the Towers-of-Hanoi family (in the Medic set with problems up to three disks), in that there are typically two choices of move in each position, and only one successful sequence. However, the length of the plan increases only polynomially in the number of checkers, whereas it increases exponentially in the number of disks for Towers-of-Hanoi. But both families quickly exceed the capabilities of today's solvers.

Another formula set was generated by a bounded model checker and contains three series of increasingly large formulas. After simplification, the number of variables ranges from 22 to 7310, averaging 2008, the number of clauses ranges from 71 to 31323, averaging 8185, and there are up to 88564 literals. One formula simplified to the empty clause. Each formula constitutes the negation of a verification theorem. For example, `longmult03` is the negation of a theorem that asserts that bit 3 of the output of a given circuit has the correct value for all possible inputs, where the function of the circuit is to multiply its inputs; showing that `longmult03` is unsatisfiable proves that bit 3 is correct for all inputs. There is a separate theorem for each bit of the output.

Another formula set is called the circuit formula set and is generated from a hardware application called *automated test-pattern generation* (ATPG) [11]. Each formula has the property that it is satisfiable if and only if there is a setting of input bits that causes the correct and faulty circuits to exhibit different outputs. We used a collection of 325 formulas that simulate either "single stuck-at" or "bridge faults". The circuits were taken from the ISCAS-85 benchmark suite. The number of variables ranged from 421 to 10410, with the average at 1830. The number literals ranged from 2278 to 87423, with the average at 13213.

The last formula set is called the random formula set and is a collection of 100 randomly generated 3-CNF formulas with 141 variables, and another 50 with 200 variables, both generated at the clauses-to-variables ratio of 4.27. This ratio is believed to generate the hardest random 3SAT formulas [12, 19].

Both the circuit formula set and the random formula set are relatively easy and were used to study the run-time behavior of `modoc` over different choices of parameters.

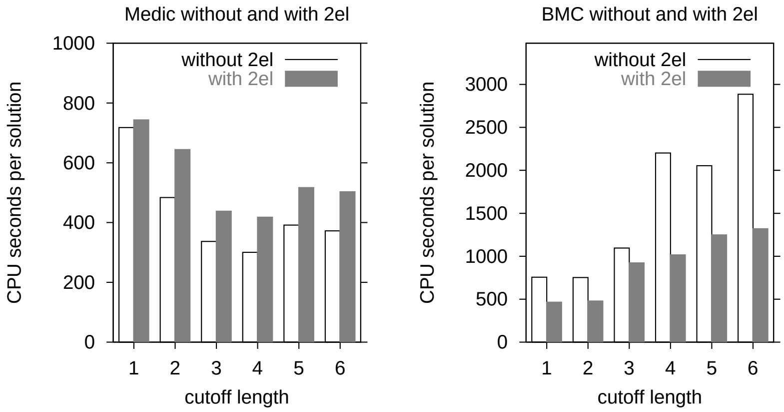
Section 3 summarized the experimental assessment of the effectiveness of secondary eager lemmas. Section 5.2 reports on some of the data obtained while we were seeking a good choice of parameters for the filtering function. The policy of using persistent lemmas in extension operations is also evaluated. Section 5.3 evaluates the use of persistent lemmas on the Satplan and checker-interchange formulas described above, which comprise the hardest problems studied. To assess the competitiveness of `modoc` we also ran several other state-of-the-art SAT testers on these formulas.

5.2. Finding Good Parameters for the Filtering Function

Before experimenting with various filtering functions, we first tested the effect of making lemmas persistent based on a simple length cutoff. Following Sakallah and Silva [25], we decided that we will always discard lemmas whose length is more than 20 as being too weak. Also, lemmas of length 1 are always made persistent, in effect, by being attached to the root of the search tree.

Figure 9 shows search times of `modoc` on the Medic and BMC formula sets using various constant cutoff values. A cutoff value of k means that all quasi-persistent lemmas with k or fewer literals are made persistent. We observe a dramatically different behavior by `modoc`. Figure 10 shows additional runs of `modoc` on the circuit formula set and the two random formula sets.

These runs showed several surprising results. First, and most obviously, persistent lemmas were bad for the BMC formulas in all cases. Secondly, we noticed that the improvement produced by persistent lemmas was significantly greater in the absence of secondary eager lemmas, for the Medic formulas. In all cases, we observe the performance degrading when the cutoff is five or six.



A quasi-persistent lemma that contains no more literals than the cutoff value is made a persistent lemma. Cutoff 1 is the same as no persistent lemmas. Statistics are based on 28 Medic formulas and 15 BMC formulas. Times are for an UltraSparc 440 MHz workstation.

Figure 9. Solution times for `modoc` on two formula sets.

Based on the results, we decided to look for a (non-constant) filtering function that would (nearly) always make lemmas of size 3 or less persistent, but would sharply curtail retention of lemmas with more than six literals. We tried to find a better filtering function that could improve upon what was shown in Figures 9 and 10.

Motivated by Section 4.1, we considered a filtering function of the form

$$f(L) \;=\; A\,L\,e^{L} \tag{1}$$

where A is a parameter and L is the number of literals of the lemma in clause form. Note that $A = 0$ degrades to the case where all the lemmas through length 20 are made persistent.

Figure 11 shows how the total search time changes as the parameter A varied on the set of 28 interesting Medic formulas, i.e., those taking at least one CPU second; the same formula set was used in Figure 9. Since this set performed better without secondary eager lemmas, we ran these tests in that mode. However, the robustness of the function can be seen from tests *with* secondary eager lemmas in effect, which are reported in Section 5.3.

We observe that the filtering function did a little better than the simple cutoff at four over a range of parameter values for A. The best choice based on this data is $A = 0.50$ with all lemmas of length four or less also being retained; this is the filtering function used for runs reported later. The CPU time per solution was 266 seconds, as opposed

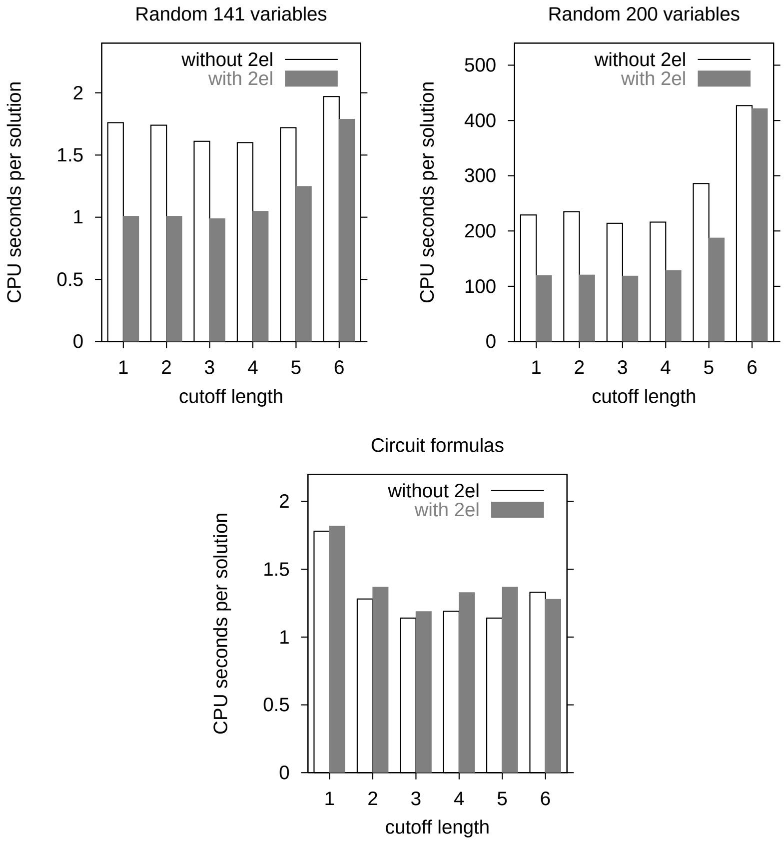
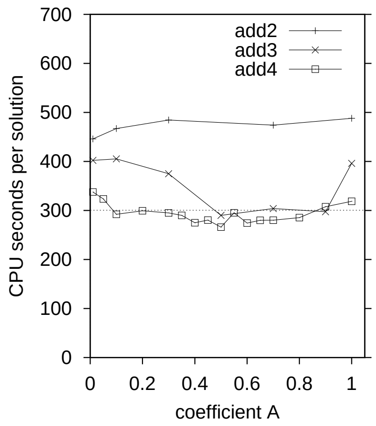


Figure 10. Solution times for `modoc` on 71 Circuit formulas, 100 random 141-variable formulas, and 50 random 200-variable formulas, using the constant cutoff policy in Figure 9. Times are for an UltraSparc 440 MHz workstation.

to 300 seconds with cutoff four. The difference was about the same as its standard error, so it is not statistically significant by itself. However, the fact that other nearby parameter settings also improved on the simple cutoff at four reinforces our confidence in this filtering function.

Note that with the decrease in A, more lemmas are turned into persistent lemmas. However, this does not automatically mean that the search will become faster. One problem is the increase in overhead to manage the increased number of clauses. Another potential problem is the decrease in quality of the persistent lemmas. We have observed that not only do the search times start to increase, but in some cases, so do the number of extensions.

Population: Medic planning formulas



The filtering function is $A\,L\,e^{L}$, where L is the length of the lemma. Secondary eager lemmas were *not* used. The function is combined with various cutoffs, labeled add2, add3, and add4, meaning that lemma clauses of these lengths or less were always made persistent. The line at 300 represents the time for the constant cutoff of 4. Times are CPU seconds on an UltraSparc 440 MHz workstation.

Figure 11. Variation in `modoc` search times for different coefficients in the filtering function.

5.3. Effect of Selective Persistent Lemmas

This section examines the improvement in terms of `modoc` search times on various hard planning formulas. The filtering strategy labeled C4 was:

1. Never discard lemmas whose length is 4 or less.

2. Always discard lemmas whose length is greater than 20.

3. Otherwise, compute $0.5\,Le^{L}$ for the lemma clause length L (actually, look it up). If the lemma clause cost is less than this value, discard it; otherwise, let it through.

Alternative strategies using simple cutoffs were also evaluated and are labeled "$\leq$ 3" and "$\leq$ 4". Throughout this section secondary eager lemmas *are* in effect.

Table II gives the formula sizes for tests in this section. Tables III and IV compare the search times of `modoc` against several state-of-the-art SAT testers, `satz`, `relsat`, `sato3`, and `heerhugo`. The first three testers are complete model-search procedures based on the DPLL algorithm [5, 4]. The last uses a breadth-first procedure to generate lemmas, which is quite different.

Table II. Satplan and Checker-Interchange formula sizes after simplification.

problem	dead-line	no. of vars	no. of literals
logistics.c	12	787	18,244
	13	897	21,412
bw_large.c	13	1,935	66,547
	14	2,222	78,146
bw_large.d	17	4,275	184,180
	18	4,714	205,559

no. of checkers	dead-line	no. of vars	no. of literals
3	14	261	16,794
	15	282	18,294
4	23	570	53,252
	24	597	55,934

The program `satz` incorporates a sophisticated branch-variable selection heuristic that uses unit propagation to score variables, and it incorporates failed-literal tests [15].

The program `relsat` incorporates a two-step branch-variable selection heuristic that makes stochastic choices, as well as a mechanism to record (and discard) *nogoods*, which are information derived from failed searches [2]. "Learn orders" of both 3 and 4 were used, as recommended by the author. However, because "learn order" of 4 did better in almost all cases, only the search times from those runs are reported. (Please see the cited paper for explanation of "learn order".)

The program `sato3` is essentially an enhanced version of `sato2` [30]. (The author reports that `sato3.2.1` corrects some bugs in version 3.0.) It adopts a more sophisticated rule for choosing the splitting variable and adds its own implementation of the Grasp lemma mechanism [25], which permanently asserts lemmas of lengths up to 20. We also experimented with `grasp` [25] but found that it was generally slower than other programs, which agrees with other reports [30]. Therefore, no times are reported for `grasp`.

The program `heerhugo` uses a new forward-chaining refutation procedure [9]. It derives lemmas by successively deeper resolution proofs in a breadth-first manner.

Table III compares the search times on the Satplan formulas. For each problem, two formulas were generated with different deadlines. The first formula has the deadline that is one less than the shortest plan possible, making it unsatisfiable. The second formula has the deadline that is the shortest plan possible, making it satisfiable.

The formulas were simplified before running the SAT testers. Since Modoc takes a backward-chaining approach, formulas must have theorem clauses intact for it to work efficiently. Standard formula simplification procedures do not preserve theorem clauses, so a special goal-sensitive simplifier [29] was used as a preprocessor for Modoc.

Table III. Search times of SAT testers on planning formulas generated by Satplan. For each problem, the first formula (earlier deadline) is unsatisfiable, and the second formula (later deadline) is satisfiable. Times are CPU seconds on an UltraSparc 440 MHz workstation. '>54000' indicates that the run was terminated after 15 hours.

problem	deadline	satz 41	relsat 2.00	sato 3.2.1	heerhugo 0.3	modoc (persistence strategy) none	modoc C4	modoc ≤ 3	modoc ≤ 4
logistics.c	12	493	1	5	155	1857	1118	737	1044
	13	84	1	1	485	1	1	1	1
bw_large.c	13	3	9	3	4625	444	387	378	422
	14	1	18	8	51,471	1606	1141	1238	1165
bw_large.d	17	524	226	358	>54000	>54000	>54000	>54000	>54000
	18	847	149	472	>54000	5	5	5	5

However, since the other programs do not care about preserving theorem clauses, the formulas were further simplified as preprocessing for those programs. Thus the other programs were running on formulas slightly smaller than the version Modoc processed. The formula sizes in the tables refer to the smaller sizes. The simplification operations included unit implication, unit propagation, pure-literal elimination, and equivalent-literal elimination. Note that these operations are all quick and simple. They are intended only to eliminate the trivial subconstructs within the formulas, leaving mostly the "hardness" of the formulas to challenge the SAT testers. simplification times are not included in the search times reported later.

The worse performance of `modoc` on `bw_large.c`, compared to other SAT testers, is because it started with a "bad" top clause. Currently, `modoc` has no mechanism to prioritize candidate top clauses, and thus, it simply uses them in the order given. In the case of the satisfiable variant of `bw_large.c` (i.e., one with deadline 14), a solution could be found in 1 second had `modoc` started the search starting from the 7th clause, while starting from many of the other top clauses will cause it to take at least 10 minutes to finish. Variability of search times depending on the top clause has been observed on other formulas, although it is not as prominent on random formulas. While this is a problem when running a single Modoc to solve the problem, it can be exploited when multiple Modocs are run in parallel. Okushi investigates a SAT tester in which multiple Modoc "agents" are run, each using a different top

Table IV. Search times of SAT testers on the checker-interchange formulas. For each problem, the first formula (earlier deadline) is unsatisfiable, and the second formula (later deadline) is satisfiable. Times are CPU seconds on an UltraSparc 440 MHz workstation. ">54000" indicates that the run was terminated after 15 hours.

		program / version								
no. of	dead-	`satz`	`relsat`	`sato`	`heerhugo`	`modoc` (persistence strategy)				
checkers	line	41	2.00	3.2.1	0.3	none	C4	≤ 3	≤ 4	≤ 5
3	14	19	17	3	418	16	4	9	5	6
	15	1	23	9	254	6	3	4	3	3
4	23	22,480	7961	>54000	10,411	22,214	4522	5555	4631	6292
	24	2561	2184	>54000	25,623	1079	292	383	309	338

clause, and the agents share certain autarkies and lemmas as they are derived [22, 21].

Table IV compares the search times on the checker-interchange formulas. As in Table III, two formulas were generated for each instance of the game—one unsatisfiable and one satisfiable. The formulas were simplified as before. Notice the relatively modest numbers of variables and literals, compared to the earlier application-derived formulas. Nevertheless, the problems have proven to be much harder.

Most programs had little trouble with the three-checker formulas, but results varied widely on the four-checker formulas.

The persistent lemmas were most effective on this set, cutting the run times by factors of 2 to 4. This reinforces our tentative conclusion that persistent lemmas become more important as formulas become harder.

Historically, `modoc` with the ≤ 3 persistence strategy was the first program to solve the four-checker problem with deadline 23; however, with today's faster processors, several programs can solve it within several hours, including `modoc` without persistent lemmas. The results on this problem tentatively indicate that persistent lemmas are a promising technique for solving extremely hard problems generated from an application.

6. Conclusions and Future Work

Judicious use of lemmas in refutation search has a great potential to improve the search efficiency. Secondary eager lemmas constitute a form of lemma propagation using unit-clause propagation. We also presented a strategy that converts certain quasi-persistent lemmas to be asserted permanently. The decision on whether to convert a quasi-persistent lemma into a persistent one is based on the length of the lemma clause and the cost to derive it. We have demonstrated their benefit by implementing them in a propositional theorem prover called Modoc. Experimental results show that secondary eager lemmas are beneficial to about the same degree across a wide range of formula difficulty. The use of persistent lemmas appears to be increasingly beneficial as formulas get harder. We also observed that the filtering function we adopted in our study appears to be suitable across a wide range of application-derived formulas.

Whereas the "Grasp" strategy combined with DPLL was found to benefit by retaining clauses of up to 20 literals, we did not find much benefit in Modoc by retaining clauses of more than three to four literals. Perhaps this is because Modoc's quasi-persistent lemmas were already fairly effective, whereas DPLL started from nothing. There is no length limit on quasi-persistent lemmas.

Future work should proceed along several directions. Continued work on exploiting lemmas while preserving search efficiency is a must. There is still much work to be done in finding better ordering heuristics in general. Extension to first-order theorem proving deserves investigation.

Acknowledgements

This work was supported in part by NSF grant CCR-95-03830. Many of the formulas used in this research were obtained from other researchers. Bounded-model-checker formulas were obtained from Ed Clarke's group (Carnegie-Mellon University). Circuit formulas were obtained from Tracy Larrabee (University of California, Santa Cruz). Planning formulas, other than the checker-interchange formulas, were generated using two "SAT compilers"—Medic, developed by Michael Ernst, Todd Millstein, and Daniel Weld (University of Washington), and Satplan, developed by Henry Kautz and Bart Selman (AT&T Bell Laboratories).

Source code of SAT testers other than `modoc` was provided by their respective authors—`satz`, by Chu Min Li (Université de Picardie Jules Verne), `relsat`, by Roberto Bayardo and Robert Schrag (University of Texas at Austin), `sato3`, by Hantao Zhang (University of Iowa), `grasp`, by João Silva and Karem Sakallah (University of Michigan), and `heerhugo`, by Jan Groote (Eindhoven University of Technology) and Joost Warners (Technical University of Delft).

References

1. Astrachan, O. L. and D. W. Loveland: 1997, 'The Use of Lemmas in the Model Elimination Procedure'. *Journal of Automated Reasoning* **19**, 117–141.
2. Bayardo, Jr., R. J. and R. C. Schrag: 1997, 'Using CSP Look-Back Techniques to Solve Real-World SAT Instances'. In: *Proceedings Fourteenth National Conference on Artificial Intelligence (AAAI-97)*. pp. 203–208.
3. Biere, A., A. Cimatti, E. M. Clarke, and M. Fujita: 1999, 'Symbolic Model Checking using SAT Procedures instead of BDDs'. In: *Proc. Design Automation Conference*.
4. Davis, M., G. Logemann, and D. Loveland: 1962, 'A Machine Program for Theorem-Proving'. *Communications of the ACM* **5**, 394–397.
5. Davis, M. and H. Putnam: 1960, 'A Computing Procedure for Quantification Theory'. *Journal of the Association for Computing Machinery* **7**, 201–215.
6. Ernst, M. D., T. D. Millstein, and D. S. Weld: 1997, 'Automatic SAT-Compilation of Planning Problems'. In: *15th International Joint Conference on Artificial Intelligence*. pp. 1169–1176.
7. Fikes, R. E. and N. J. Nilsson: 1971, 'STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving'. *Artificial Intelligence* **2**(3/4), 189–208.
8. Fleisig, S., D. W. Loveland, A. K. Smiley, and D. L. Yarmush: 1974, 'An implementation of the model elimination proof procedure'. *JACM* **21**(1), 124–139.
9. Groote, J. F. and J. P. Warners: 2000, 'The propositional formula checker HeerHugo'. *Journal of Automated Reasoning* **24**(1), 101–125.
10. Kautz, H. and B. Selman: 1996, 'Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search'. In: *13th National Conference on Artificial Intelligence*. pp. 1194–1201.
11. Larrabee, T.: 1992, 'Test Pattern Generation Using Boolean Satisfiability'. *IEEE Transactions on Computer-Aided Design* **11**(1), 6–22.
12. Larrabee, T. and Y. Tsuji: 1992, 'Evidence for a Satisfiability Threshold for Random 3CNF Formulas'. Technical Report UCSC–CRL–92–42, UC Santa Cruz, Santa Cruz, CA.
13. Lee, S.-J. and D. A. Plaisted: 1992, 'Eliminating duplication with the hyper-linking strategy'. *Journal of Automated Reasoning* **9**(1), 25–42.
14. Letz, R., K. Mayr, and C. Goller: 1994, 'Controlled integration of the cut rule into connection tableau calculi'. *Journal of Automated Reasoning* **13**(3), 297–337.
15. Li, C. M. and Anbulagan: 1997, 'Heuristics Based on Unit Propagation for Satisfiability Problem'. In: *Proceedings International Joint Conference on Artificial Intelligence*. pp. 366–371.
16. Loveland, D. W.: 1968, 'Mechanical Theorem-Proving by Model Elimination'. **15**(2), 236–251.
17. Loveland, D. W.: 1969, 'A simplified format for the model elimination theorem-proving procedure'. *Journal of the Association for Computing Machinery* **16**(3), 349–363.
18. Minker, J. and G. Zanon: 1982, 'An extension to linear resolution with selection function'. *Information Processing Letters* **14**(3), 191–194.
19. Mitchell, D., B. Selman, and H. Levesque: 1992, 'Hard and Easy Distributions of SAT Problems'. In: *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92), San Jose, CA*. pp. 459–465.

20. Monien, B. and E. Speckenmeyer: 1985, 'Solving Satisfiability in less than 2^n Steps'. *Discrete Applied Mathematics* **10**, 287–295.
21. Okushi, F.: 1998, 'Propositional Theorem Proving: Advanced Lemma Strategies and Multi-Agent Search'. Ph.D. thesis, University of California, Santa Cruz.
22. Okushi, F.: 1999, 'Parallel Cooperative Propositional Theorem Proving'. *Annals of Mathematics and Artificial Intelligence* **26**(1–4), 59–85.
23. Plaisted, D. A.: 1994, 'The search efficiency of theorem proving strategies'. In: *12th International Conference on Automated Deduction*. pp. 57–71.
24. Shostak, R. E.: 1976, 'Refutation graphs'. *Artificial Intelligence* **7**(1), 51–64.
25. Silva, J. P. and K. A. Sakallah: 1996, 'GRASP–A new search algorithm for satisfiability'. In: *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design*. pp. 220–227.
26. Van Gelder, A.: 1999a, 'Autarky Pruning in Propositional Model Elimination Reduces Failure Redundancy'. *Journal of Automated Reasoning* **23**(2), 137–193. (preprint at `ftp://ftp.cse.ucsc.edu/pub/avg/JAR/aut-jar-dist.ps.Z`).
27. Van Gelder, A.: 1999b, 'Complexity Analysis of Propositional Resolution with Autarky Pruning'. *Discrete Applied Mathematics* **96–97**, 195–221.
28. Van Gelder, A. and F. Okushi: 1999a, 'Lemma and Cut Strategies for Propositional Model Elimination'. *Annals of Mathematics and Artificial Intelligence* **26**(1–4), 113–132.
29. Van Gelder, A. and F. Okushi: 1999b, 'A Propositional Theorem Prover to Solve Planning and Other Problems'. *Annals of Mathematics and Artificial Intelligence* **26**(1–4), 87–112.
30. Zhang, H.: 1997, 'SATO: An Efficient Propositional Prover'. In: *14th International Conference on Automated Deduction*. pp. 272–275.

Address for Offprints:
Allen Van Gelder
Department of Computer Science, SOE
University of California, Santa Cruz, CA 95064
U.S.A.