Lemma and Cut Strategies for Propositional Model Elimination

Allen Van Gelder [a,*] Fumiaki Okushi [b]

[a] *Computer Science Dept., 225 AS, University of California, Santa Cruz, CA 95064, U.S.A.*
E-mail: avg@cs.ucsc.edu

[b] *Computer Science Dept., California State University, Bakersfield, CA 93311, U.S.A.*
E-mail: okushi@cs.csubak.edu

This paper describes new "lemma" and "cut" strategies that are efficient to apply in the setting of propositional Model Elimination. Previous strategies for managing lemmas and C-literals in Model Elimination were oriented toward first-order theorem proving. The original "cumulative" strategy remembers lemmas forever, and was found to be too inefficient. The previously reported C-literal and unit-lemma strategies, such as "strong regularity", forget them unnecessarily soon in the propositional domain. An intermediate strategy, called "quasi-persistent" lemmas, is introduced. Supplementing this strategy, methods for "eager" lemmas and two forms of controlled "cut" provide further efficiencies.

The techniques have been incorporated into "Modoc", which is an implementation of Model Elimination, extended with a new pruning method that is designed to eliminate certain refutation attempts that cannot succeed. Experimental data show that on random 3CNF formulas at the "hard" ratio of 4.27 clauses per variable, Modoc is not as effective as recently reported model-searching methods. However, on more structured formulas from applications, such as circuit-fault detection, it is superior.

Keywords: Model Elimination, unit lemma, C-literal, quasi-persistent lemma, eager lemma, cut, resolution, refutation, Boolean formula, propositional formula

1. Introduction

In Model Elimination, a *lemma* may be recorded upon the completion of any (sub)refutation [Lov68,Lov69,FLSY74,Lov78], although this is not necessary for completeness. The original method of enlarging the set of input clauses with

* All correspondence should be addressed to this author.

lemma clauses was too inefficient to be practical [FLSY74]. Shostak proposed an efficient *C-literal* mechanism to maintain such lemmas in Model Elimination chains [Sho74,Sho76]; Letz *et al.* generalized it to trees with an ingenious time-stamping method, and also proposed a pruning strategy based on C-literals called *strong regularity* [LMG94]. Lemma strategies have been studied empirically for first-order theorem proving using chains [FLSY74,AS92,Sti94,AL97], as well as trees [LMG94], but we are aware of no empirical studies of lemma strategies on propositional problems.

In the propositional domain, the C-literal strategies tend to forget the lemmas unnecessarily too soon. This paper presents a lemma strategy, called *quasi-persistent lemmas*, that intends to prolong the lifetime of lemmas. It also describes a mechanism, called *eager lemmas*, that derives certain quasi-persistent lemmas ahead of time. Two forms of controlled *cuts* are introduced that allow (sub)searches to be abandoned based on the presence of certain relationships among the lemmas.

The new lemma strategy and other supporting features were implemented in *Modoc*, which is an implementation of propositional Model Elimination, extended with a new pruning method based on the concept of *autarky* [MS85]. Autarky-based pruning (or *autarky pruning* for short) eliminates certain refutation attempts that cannot succeed. Autarky pruning is largely orthogonal with lemma strategies and hence its understanding is not necessary to understand the technical contents of this paper. Details of autarky pruning can be found in [VG97].

The main results are summarized in Section 1.1. Tree structures for Model Elimination is described in Section 2. Modoc is briefly introduced in Section 3. New lemma creation mechanisms are described in Section 4, which also briefly reviews C-literals. Experimental results based on an efficient C implementation are reported in Section 5. Conclusions and future work are in Section 6.

1.1. Summary of Results

This paper reports on new lemma and cut strategies that are efficient to apply in the setting of propositional Model Elimination. Experimental data is based on an efficient C implementation.

New lemma creation mechanisms called quasi-persistent lemmas and eager lemmas are described in Section 4. Quasi-persistent lemmas are a variant of the

C-literal strategy, adapted for efficiency in the propositional case. It is shown that the quasi-persistent strategy retains the lemmas longer than C-literals, but it is incompatible with the strong regularity strategy proposed by Letz *et al.* [LMG94] (Section 4.2).

Eager lemmas supplement C-literals to provide substantial further reductions in proof searching by providing early identification of refutations that will succeed (Section 4.5). Derivation of eager lemmas is closely related to unit-clause propagation.

When the refutation of a literal succeeds in a certain way, not only does the complement of the literal become a quasi-persistent lemma, but also the complements of certain eager lemmas become quasi-persistent lemmas. The eager lemmas whose complements survive as quasi-persistent lemmas are shown to correspond to articulation points of a graph related to the successful refutation (Section 4.5). Articulation points can be identified in linear time by a standard graph algorithm.

The quasi-persistent lemma strategy is supplemented by two forms of controlled cuts. One cut applies when a contradictory pair of C-literals is derived (Section 4.3). The other cut applies when a clause no longer has any literals that need to be refuted (either because it is not necessary, or because it is already known to be refutable) (Section 4.4).

Experiments suggest that each of the four features, quasi-persistent lemmas, lemma-induced cuts, C-reduction-induced cuts, and eager lemmas, provides an additional order-of-magnitude speed-up, bringing times down from hours to seconds.

1.2. Notation and Terminology

Standard terminology for conjunctive normal form (CNF) formulas is used. A finite set of propositional variables is fixed throughout the discussion. The term "propositional variable" is abbreviated to "variable" when no confusion can result.

Definition 1. (literal, clause, formula) A literal is a positive variable x, or a negated variable $\neg x$. Literals x and $\neg x$ are *complementary*. The complement of literal q is denoted $\neg q$, whether q is positive or negative; i.e., double negations are simplified away.

A *clause* is a disjunction of zero or more literals, represented simply as a set of literals. Of special interest are the *empty clause*, denoted by $\emptyset$, representing *false*, and *unit clauses*, consisting of exactly one literal. A clause consisting of literals $p_1, \ldots, p_k$ $(k \geq 1)$ is denoted as $[p_1, \ldots, p_k]$.

A *CNF formula* (*formula* for short, since only CNF formulas are considered) is a conjunction of zero or more clauses, represented simply as a set of clauses (or a multiset, if duplicate clauses occur). The empty formula represents *true*. □

Definition 2. (assignment, satisfaction, model) A partial assignment is a partial function from the set of variables into $\{false, true\}$. This partial function is extended to literals, clauses, and formulas in the standard way. If the partial assignment is a total function, it is called a *total assignment*, or simply an *assignment*.

A clause or formula is *satisfied* by a partial assignment if it is mapped to *true*; a partial assignment that satisfies a formula is called a *model* of that formula. □

A partial assignment is conventionally represented by the (necessarily consistent) set of *unit clauses* that are mapped into *true* by the partial assignment. Note that this representation is a very simple formula. Set-forming braces are omitted sometimes to streamline notation.

Definition 3. (strengthen, unit implication, unit subsumption) Let M be a partial assignment for formula S. The clause $C|M$, read "C strengthened by M", is the (possibly empty) set of literals

$$C|M = C - \{q \mid q \in C \text{ and } \neg q \in M\}$$

This process is called *unit implication* when M consists of one literal.

The formula $S|M$, read "S strengthened by M", is the (possibly empty) set of clauses

$$S|M = \{C|M \mid C \in S \text{ and } C \text{ contains no literal of } M\ \}$$

When M consists of one literal, the process of eliminating clauses containing this literal is called *unit subsumption*. □

Example 4. Let S consist of $[a, b]$, $[\neg a, c]$, and $[b, d]$. Then $S|\{a\} = \{[c], [b, d]\}$, and $S|\{a, c\} = \{[b, d]\}$. □

2. Model Elimination and Derivation Trees

This section informally describes the tree data structures we shall use to describe derivations in Model Elimination (and Modoc). Trees are now recognized as the most appropriate data structures for representation of linear resolution derivations. Interestingly, Loveland's original description of Model Elimination [Lov68] can be viewed as a tree structure, although the algorithm description is not phrased in terms of a tree. The next paper used a chain format [Lov69], and later reports were based on the chain format. Minker and Zanon [MZ82] independently discovered the tree format and recognized the extra flexibility it offers. Recently, Letz *et al.* have given a unified view of the methods of tableau calculus and clause trees [LMG94]. However, these are geared toward first-order application. Our tree data structure is different in that we need not concern ourselves with substitution, and hence it is chosen specifically for propositional resolution. The following technical definitions are illustrated in Example 8 and Figures 1 and 2.

Definition 5. (clause-goal tree, goal ancestor) Let a set S of propositional clauses be given (i.e., a formula). Let $\top$, called *verum*, be a symbol distinct from all propositional variables.

A *clause-goal tree* is a bipartite directed tree with two classes of nodes, called *clause nodes* and *goal nodes*. That is, a clause node may only have goal nodes as children and *vice versa*. Each clause node is labeled with a clause of S, and each goal node is labeled with a literal in S, or with $\top$. Edges are directed from the root to the leaves. Recall that a *branch* of a tree is a path from the root to a leaf. The tree is unordered in the sense that the order of any node's children is immaterial.

A *goal ancestor* of a node v is a goal node on the path from the root to v, including v itself if it is a goal node. Since clause ancestors are not significant, we shall refer to goal ancestors simply as ancestors. The set of all goal ancestors of v is denoted by $ancs(v)$. While $ancs(v)$ is technically a set of nodes, it can also be considered as a set of unit clauses made from the nodes' literal labels, i.e., a formula. □

Definition 6. (propositional derivation tree (PDT), PDT extension) Let S and $\top$ be as in Definition 5. Throughout this definition, all literals and all clauses are assumed to be in S.

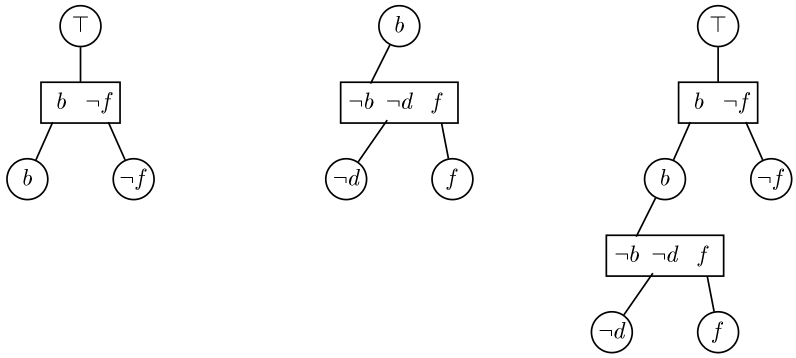

Figure 1. Examples of propositional derivation trees (PDTs). Left: A PDT with *top clause* $[b, \neg f]$. Center: The extension clause $[\neg b, \neg d, f]$ may be thought of as a single-clause PDT. Right: The result of PDT extension.

A *propositional derivation tree* (PDT) is a clause-goal tree in which

1. Each clause contains a literal complementary to its *goal parent* (or its parent is $\top$, in which case the clause is called the *top clause*).
2. No clause contains a literal that is a goal ancestor of the clause.
3. The goal children of each clause consist exactly of the literals that are not complemented in the clause's goal ancestors.

A *PDT extension* adds one clause and the necessary subgoals of that clause to a PDT, maintaining the above properties. The clause is attached as a child of an existing leaf goal node. □

From the definition, we see that every PDT is rooted with a goal node, and every goal node either is a leaf or has exactly one child. Also, it is easy to see that a clause node labeled with C is a leaf in a PDT if and only if every literal in C is complemented in $ancs(C)$.

Definition 7. (refutation) If a PDT labels its root with q and contains only clause nodes as leaves, then this PDT is called a *refutation of* q *with respect to* S. If $q = \top$, it is simply called a *refutation of* S. (These terms are justified in Theorem 9.) □

Example 8. Figure 1 shows examples of PDTs. The PDT at the right is obtained by PDT extension using the other two. Figure 2 shows a clause node in

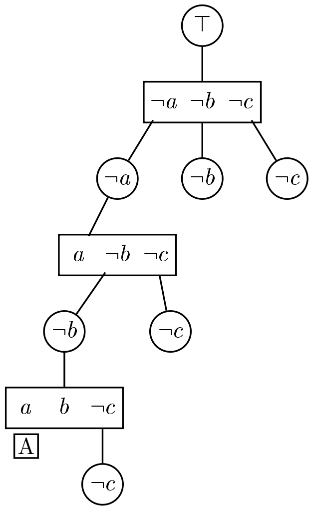

Figure 2. An example of a propositional derivation tree (PDT). The boxed "A" indicates that the corresponding goal node was implicitly reduced.

which an implicit reduction has occurred, as indicated by the boxed "A". (This box is a notation and is not a structural part of the tree.) □

Let us point out that the terminology "PDT extension", as defined above, is a combination of Model Elimination operations called *extension* and *reduction*. In the propositional framework, reduction may be considered mandatory [Sho76]. In a first-order framework, both extension and reduction must normally be considered, due to differing unifiers, so the goal must be created.

For purposes of intuition, a "goal node" means that the goal of the derivation is to *refute* the literal in the node, not to validate it. The term "refutation" in Definition 7 is justified by the following theorem, which essentially states that propositional weak Model Elimination is sound. This is already well known [Lov69,MZ82,LMG94], and is proved in the new terminology in [VG97].

Theorem 9.

(A) If there is a PDT that is a refutation of q with respect to S, then S has no model in which q is true.

(B) If there is a PDT that is a refutation of S, then S is unsatisfiable.

□

3. Modoc

Modoc is a satisfiability testing algorithm based on Model Elimination, extended with a new pruning method based on the concept of *autarky* [MS85]. Autarky pruning eliminates certain branches that cannot lead to a successful (sub)refutation.

Two basic methods have been developed for satisfiability testing: refutation search and model search. The *Modoc* algorithm represents an integrated approach that simultaneously searches for either a refutation or a model of a propositional formula. An important by-product of this duality is that major redundancies are eliminated from refutation search, and factors of 1,000,000 are saved on formulas as small as 20 variables and 90 clauses [VG97].

Modoc tries to construct a refutation tree in a top-down manner. In practice, it is not necessary for it to actually construct a refutation tree, only determine that it *could* be constructed. If the outcome of subtree construction is known, Modoc may work on another part of the tree. There are two cases where this could happen.

1. One is when it is certain that a goal node *has* a successful refutation. This involves the use of lemmas.
   When there is a C-literal that is the complement of a goal node, that goal node need not be processed further as the C-literal indicates that it will succeed. (This can be seen by extending the goal node with the C-literal in full clause form.)

2. Another is when it is known that an extension clause *cannot* lead to a successful refutation. This involves the use of autarkies.
   If a possible extension clause is satisfied by an autarky, then its use in a PDT extension cannot lead to a successful (sub)refutation [VG97]. Modoc exploits this property and disregards any possible extension clauses that are satisfied by the current autarky.
   Autarky pruning is compatible with the use of lemmas and largely orthogonal. An understanding of it is not necessary to understand the technical contents of this paper. Detailed discussion on autarky and autarky pruning can be found in another paper [VG97].

3.1. Pre-Reduction

Because reductions can be considered mandatory in the propositional case Modoc implements what we call *pre-reduction.* Normally, ME waits until a clause is chosen for an extension, then carries out whatever reductions may be possible. Modoc carries out the reduction operation on every clause containing $\neg p$ as soon as p is chosen as a goal node. Similarly, Modoc eliminates all clauses containing p from eligibility for extension as soon as p is chosen as a goal node. By building lists once that detail where p and $\neg p$ occur, these operations are very efficient. For an original formula S, when the set of ancestors is A, Modoc is effectively processing $S|A$ (see Definition 3).

4. Lemmas and C-literals

This section briefly reviews the traditional C-literals, and then introduces *quasi-persistent lemmas* and *eager lemmas*, as well as two forms of *cuts* that permit refutations to be accelerated. In particular, we sketch how lemmas are incorporated into the implementation of the Modoc algorithm; of course, the same techniques could be used with any implementation of Model Elimination. We show that the quasi-persistent strategy retains C-literals longer than previously reported strategies, but it is incompatible with strong regularity. The original formula is represented as S throughout this section.

Suppose the refutation of a literal q is completed in a PDT. Let $B = \{p_1, \ldots, p_m\}$ be the subset of ancestors of q, excluding q itself, that were actually used for reductions in q's refutation. (Note that it is possible for q to be used for reduction also. See Example 11.) Then, a lemma clause $[\neg q, \neg p_1, \ldots, \neg p_m]$ can be derived soundly [Lov68,LMG94]. That is, $S \vdash [\neg q, \neg p_1, \ldots, \neg p_m]$. Equivalently, $S, B \vdash \neg q$.

Definition 10. (C-literal, dependency) In the lemma $[\neg q, \neg p_1, \ldots, \neg p_m]$ described above, literal $\neg q$ is called a *C-literal.* The literals of B are called the *dependencies* of the C-literal. For lemmas we adopt a Prolog-like notation in which the C-literal is to the left of an "$\leftarrow$" symbol, which is read as "if". The literals to the right of the "$\leftarrow$" are the dependencies. Thus the above lemma will be written as $[\neg q \leftarrow p_1, \ldots, p_m]$, or as $[\neg q \leftarrow B]$. (When a set of literals appears in an antecedent position like this, the literals are considered to be joined conjunctively, and the empty set denotes *true*.) □

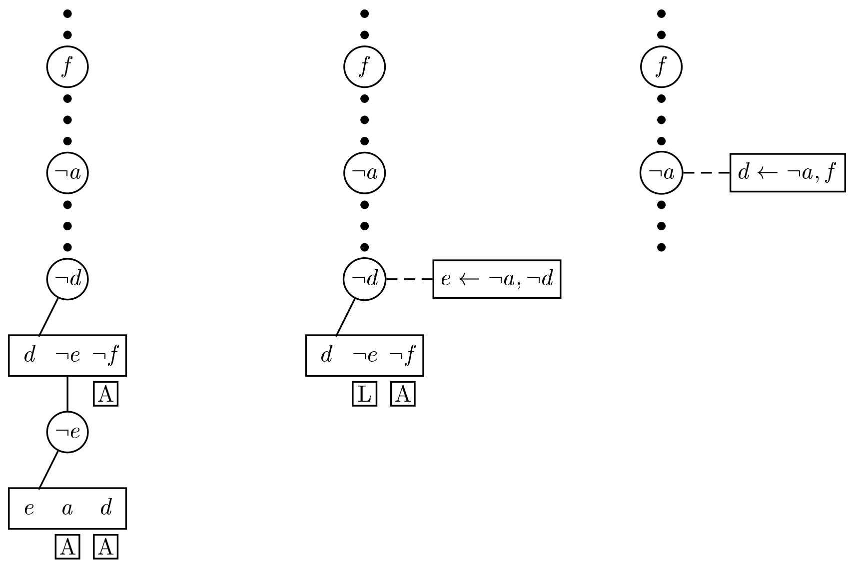

Figure 3. A fragment of a Modoc search tree (or PDT with additional annotations for C-literals) showing the attachments of C-literals e and d, as discussed in Example 11.

The lemma $[\neg q \leftarrow B]$ can be "attached" to the *lowest* ancestor (say p_c) among $B = \{p_1, \ldots, p_m\}$. If $m = 0$, then it is attached to the root of the PDT, which is normally $\top$. With our notation, observe that $S, B \vdash \neg q$ is equivalent to $S \vdash [\neg q \leftarrow B]$. The C-literal can only be used in the subtree of p_c, and in this context, its operational behavior is somewhat like reduction with an ancestor. Hence, the operation is sometimes called *C-reduction.* Once the C-literal is derived, it is effective, or "visible", to all descendent nodes of p_c, as well as p_c itself.

Example 11. Consider the fragments of a Modoc search tree shown in Figure 3. We begin in the left diagram at the point where the Modoc search tree has been extended with clause $[e, a, d]$, at the bottom of the figure. At this point, no C-literals have been attached. Clause $[e, a, d]$ has no subgoals due to reductions by the ancestors $\neg a$ and $\neg d$, and so the goal $\neg e$ is refuted using clause $[e, a, d]$, together with ancestors $\neg a$ and $\neg d$. The latter are the dependencies for the new C-literal e, and it is attached to the lower of these two ancestors, as shown in the middle diagram.

Even though the first lemma corresponds to the clause $[e, a, d]$, which is already in the formula, recording this "lemma" is necessary, as will be seen in the subsequent discussion.

This lemma leads to the refutation of $\neg d$. Again, the dependencies for the new C-literal d are obtained from the clause used for its extension, $[d, \neg e, \neg f]$. Literal f is an ancestor, so becomes a dependency. But e is a C-literal, so its dependencies must be transitively traced back to *its* ancestors, which are $\neg a$ and $\neg d$. The dependency of d on $\neg d$ is ignored. This is because only the dependencies on *proper* ancestors need to be recorded. Finally, the new C-literal d depends on $\neg a$ and f. It is attached to the lower of these two ancestors, in the form $[d \leftarrow \neg a, f]$. Since $\neg a$ may be arbitrarily higher in the tree than $\neg d$, additional clauses below $\neg a$ that have not yet been processed can use d as a unit clause and instantly refute any additional occurrences of $\neg d$.

Observe that, if we did not keep track of the dependencies of the C-literal e we would not have had a way to know that the C-literal d should be attached at $\neg a$. Modoc keeps track of the dependencies, as well as the C-literal. The issues are discussed in Sections 4.1 and 4.2. □

4.1. Traditional C-literals

By traditional methods, if the PDT is abandoned (because the refutation failed, even if it was in a part not relevant to the derivation of the C-literal) then the lemma is forgotten. If the (sub)refutation of p_c is completed, the lemma is also forgotten in the sense that it is not used later in other (sub)refutations. Because of the limited application and lifetime of the lemma, it suffices to attach the C-literal $\neg q$ to the lowest ancestor p_c, and not record the exact dependencies B. Letz *et al.* described an ingenious time-stamping method for doing this [LMG94]. Modoc cannot adopt this simple strategy because it does not completely abandon a PDT when some part of the refutation fails, as discussed in the next subsection.

4.2. Quasi-Persistent Lemmas

Our strategy varies from the C-literal strategy described above in that lemmas derived during failed (sub)refutations are not necessarily forgotten. Normally, a PDT is not completely abandoned, but only the subtree where the refutation failed is abandoned. (In the first-order case, substitutions need to be backed out as well.) The lemma may continue to function as a C-literal until the subtree rooted at p_c is abandoned, or the refutation of p_c is completed (where p_c and other terminology is continued from the previous subsections).

Modoc maintains lemmas attached to p_c until the tree rooted there is abandoned or its refutation is completed. The previously mentioned strategy of Letz *et al.* effectively deletes the lemma as soon as the refutation of any clause ancestor of q fails. There are pros and cons of both strategies. Our strategy makes it unnecessary to re-derive the same lemma at the same attachment point so often, but it makes it necessary to record the full lemma, in particular, the exact dependencies. Also we did not see a way to adapt their time-stamping method to the situation in which only a small part of the tree is abandoned when a goal fails; some kind of "roll-back" of the time-stamps would be needed.

Our strategy is incompatible with the heuristic called strong regularity, introduced by Letz *et al.* However, some of the losses due to the lack of strong regularity are recovered through lemma-induced cuts, as described in Section 4.3.

Definition 12. (strong regularity) The *strong regularity* heuristic specifies that a clause may not be chosen for extension if it contains a literal that is the same as any C-literal. □

Modoc may undertake to refute a goal $\neg q$ in the subtree (rooted at p_c) where $\neg q$ is attached as a lemma. The strong regularity heuristic consists of avoiding such attempts. Strong regularity was shown to be complete under certain conditions, but quasi-persistent lemmas do not meet those conditions. Example 13 shows that incompleteness can result if the two heuristics are combined.

Example 13. This example demonstrates that our quasi-persistent lemma strategy is incompatible with the heuristic called strong regularity, introduced by Letz *et al.* [LMG94]. For brevity, the clause set contains unit clauses and pure literals, but it can be extended to avoid these properties.

Consider the clause set

$$\begin{array}{ll} [p] & \\ [b, \neg p] & [\neg b, h] \\ [\neg b, c, d] & [\neg c, \neg g, \neg h] \\ [\neg c, e] & [g] \\ [\neg c, \neg e, \neg p] & [c, \neg g] \end{array}$$

with $[p]$ as the top clause.

Following Figure 4, suppose goal p is extended with $[b, \neg p]$, then goal b is extended with $[\neg b, c, d]$, and then assume goal c is expanded before d.

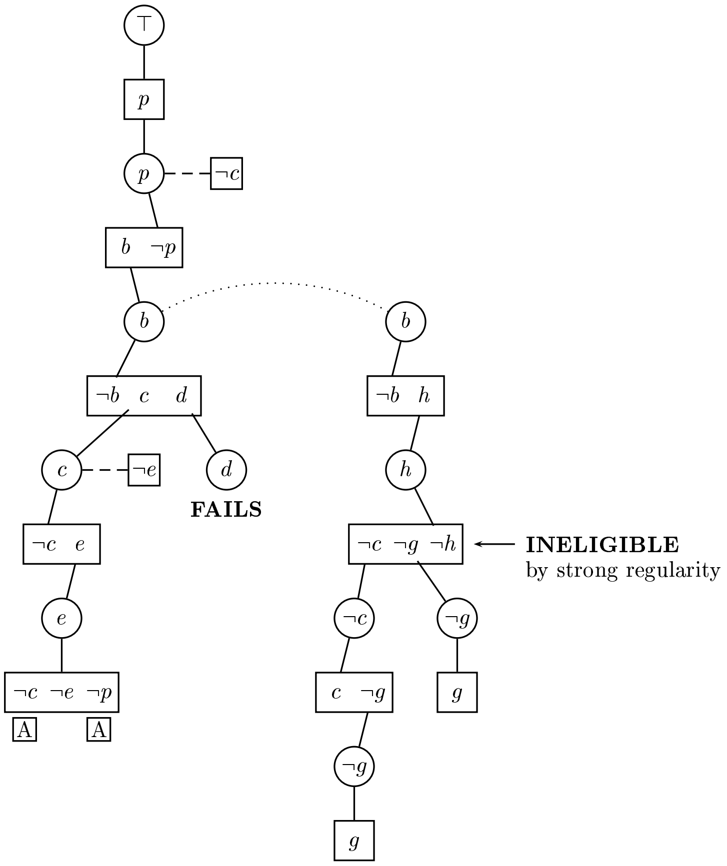

Figure 4. An example showing that the quasi-persistent lemma strategy is incompatible with strong regularity. When the goal c is refuted, all C-literal strategies "forget" the C-literal $\neg e$ that was attached to goal c, and record the new C-literal $\neg c$ at goal p. However, when the refutation attempt fails at goal d, traditional strategies "forget" the C-literal $\neg c$, whereas the quasi-persistent strategy retains it. Example 13 explains the consequences.

Goal c is extended with $[\neg c, e]$, then goal e is extended with $[\neg c, \neg e, \neg p]$, which has no subgoals. (The boxes containing "A" denote that the corresponding subgoals were removed by reductions.) Thus, goal e is refuted, and immediately afterward, goal c is refuted. The latter event causes the C-literal $\neg c$ to be attached to goal p. In the figure, C-literals are shown attached by dashed lines, and their dependencies are not shown.

The next step is to try to refute goal d, but this attempt fails. Thus, the attempt to refute b using clause $[\neg b, c, d]$ has failed.

At this point, traditional C-literal strategies abandon the C-literal $\neg c$ because it was derived in search associated with the clause that was just abandoned. However, the quasi-persistent strategy maintains this C-literal because its derivation did not involve any ancestor goals that were abandoned, and thus, it is still sound.

The next step is to try an alternate clause for goal b, which is $[\neg b, h]$ (Figure 4, right). This leads to goal h, which *might* be extended by $[\neg c, \neg g, \neg h]$.

If we adopted the rule of strong regularity, this clause would be ineligible, due to the presence of $\neg c$, which duplicates a C-literal. (Since traditional methods would have retracted the C-literal $\neg c$, strong regularity does not disqualify this clause for them.) Then, the attempt to refute b would fail. Similarly, goal p would fail, due to disqualification of the clause $[\neg c, \neg e, \neg p]$.

However, extending goal h with $[\neg c, \neg g, \neg h]$ creates goal $\neg c$ (and $\neg g$), which can be extended with $[c, \neg g]$. Then, goal $\neg g$ is refuted by clause $[g]$.

The other goal $\neg g$ is similarly refuted. This completes the refutation. $\square$

As described earlier, under the quasi-persistent lemma strategy, not only the C-literal, but also its dependencies must be recorded. The reason was seen in Example 11. The refutation of $\neg d$ used a C-literal e that was attached to $\neg d$. It was necessary to know the dependencies of e to determine where to attach the new C-literal d.

The quasi-persistent heuristic holds lemmas longer, but spends more time per lemma in bookkeeping, compared to the traditional C-literal method. There is no apparent way to determine which method performs better, except by empirical testing.

4.3. Lemma-Induced Cuts

We now describe the method by which Modoc exploits complementary C-literals. One example of the situation was seen in Figure 4, where Modoc undertook to refute a literal that already existed as a C-literal. This selection would be prohibited by the strong regularity heuristic. Even if the selection had been made before the matching C-literal existed, the current refutation attempt would have to be abandoned, under that heuristic. Modoc employs a different approach, which guarantees that the copy of the C-literal in the current clause is the *only* literal that is processed in this clause (aside from literals that were processed before the C-literal was derived, of course).

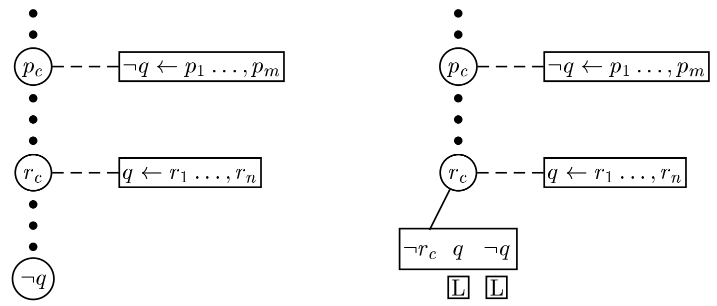

Figure 5. Lemma-induced cuts. Left: Complementary lemmas along a branch can trigger the cut. Right: Extension by a virtual clause immediately completes the refutation. The boxed "L" indicates that the corresponding goal node was closed by a lemma. Details are given in Section 4.3.

Suppose the C-literal $\neg q$ is attached to p_c with dependencies $B_0 = \{p_1, \ldots, p_m\}$ (see Figure 5), and suppose the goal $\neg q$ occurs in a subtree of p_c. Should the refutation of $\neg q$ be successful, there results a new lemma whose C-literal is q, with dependencies $B_1 = \{r_1, \ldots, r_n\}$. Now, complementary C-literals have been derived on one branch. Let r_c be the lowest ancestor among the literals of B_1, or the root $\top$ if $n = 0$.

Now, consider the lower of the two goal nodes p_c and r_c. The situation is symmetric, so let us suppose it is r_c, as shown on the left of Figure 5. Now, add a *virtual clause* $[\neg r_c, q, \neg q]$ to the formula; this is a tautology, so it is harmless. However, extending r_c with this virtual clause creates goals q and $\neg q$, both of which are immediately closed by the lemmas, as shown on the right of Figure 5. (The boxed "L" indicates that the corresponding subgoal was closed by a lemma.) Thus, the goal r_c is immediately refuted, *even though the tree in which the lemmas q and $\neg q$ were derived is never completed to a refutation.* Let $B_2 = B_0 \cup B_1 - \{r_c\}$; i.e., B_2 is the union of the dependencies of C-literals q and $\neg q$, omitting r_c. The lemma just derived is $[\neg r_c \leftarrow B_2]$.

Introduction of the virtual clause described above is essentially a form of the cut rule [LMG94]. If S is the original set of clauses, we have discovered $S| \{r_c, B_2\} \vdash q$ and $S| \{r_c, B_2\} \vdash \neg q$. Now the cut rule infers $S| \{B_2\} \vdash \neg r_c$.

While the introduction of a tautologous clause is always sound, it normally is not practical because the prover has no way to anticipate that each of the complementary literals has a short refutation. However, if a pair of complementary C-literals have been derived, then the prover has that information in hand.

This methodology also can be applied to first-order proofs where the prover

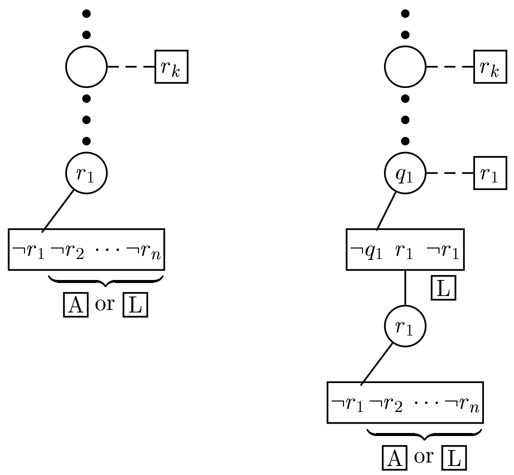

Figure 6. C-reduction-induced cuts. Before the cut operation, r_1 and q_1 were extended with different clauses from those shown. Left: Case when the lowest dependency is an ancestor ($q_1 = r_1$). Right: Case when the lowest dependency is a lemma ($q_1 \neq r_1$). Details are given in Section 4.4.

is not using strong regularity. In this case, a most general unifier of the complementary C-literals would be applied before creating the virtual clause.

4.4. C-Reduction-Induced Cuts

When Modoc selects a goal node for extension, that goal node becomes the new lowest ancestor literal. Modoc strengthens the formula with this ancestor. However, when Modoc installs a C-literal, it only applies unit implication (Definition 3) with this C-literal, which amounts to a form of "eager" C-reduction. If this creates an empty clause, then a *C-reduction-induced cut* may occur.

Suppose the original clause $C = [\neg r_1, \ldots, \neg r_n]$ shrinks to zero length due to unit implication by a new C-literal r_k. Without loss of generality, assume that r_1 is the lowest ancestor or C-literal among $r_1, \ldots, r_n$, chosen to be an ancestor if possible. There are two cases:

1. If r_1 is an ancestor, then extension of r_1 by clause C refutes it immediately, as shown on the left of Figure 6.
2. If r_1 is a C-literal, let q_1 be the ancestor to which it is attached. Now, we extend q_1 with the *virtual clause* $[\neg q_1, \neg r_1, r_1]$, as shown on the right of Figure 6. Subgoal $\neg r_1$ is C-reduced and subgoal r_1 is immediately refuted by extension with clause C.

This kind of cut is similar to the lemma-induced cuts (Section 4.3) in that a (possibly) intermediate goal node, r_1 or q_1, can be refuted without completing the refutation in progress for it.

4.5. Eager Lemma Assertion

Unit-clause propagation is of interest because of its efficiency, and its frequent use as a subroutine in model-searching algorithms, such as DPLL. Dalal and Etherington have shown that unit-clause propagation can be implemented in linear time [DE92]. This implementation is practical, as well as theoretical, and is used by numerous implementers. Modoc uses unit-clause propagation (without unit subsumption) to derive *eager lemmas*.

When Modoc selects a new goal node e, it performs unit implication with it (called *pre-reduction*, see Section 3.1). The *eager lemma strategy* consists of following up with unit-clause propagation. That is, unit implication is repeated with any clauses whose strengthened length reduces to one (see Definition 3). In the rest of the section, the original formula is S and the set of proper ancestors of e is A.

Definition 14. (eager lemma, E-literal, eager refutation) In $S|A$, suppose a new unit clause e (corresponding to the selected goal node) is asserted, and unit-clause propagation is carried out. Each clause, say C, that becomes a unit clause in this process is called an *eager lemma* and its one remaining literal, say p, is called an *E-literal.* Eager lemmas are also written in the Prolog-like notation introduced in Definition 10: $[p \leftarrow B]$. That is, each literal $q \in B$ corresponds to the complement of a literal in C, and p is in C. The literals of B are called *local dependencies* if they are in $C|A$ and are called *nonlocal dependencies* otherwise. That is, the local dependencies were resolved out during the unit-clause propagation following the assertion of e, and the nonlocal dependencies were resolved out earlier. If an empty clause is derived, the process is called an *eager refutation of* e. □

All derived E-literals are attached to the goal node e. Like C-literals, E-literals are logical consequences of $(S, A, [e])$, and can be used in the same manner in the refutation search below e.

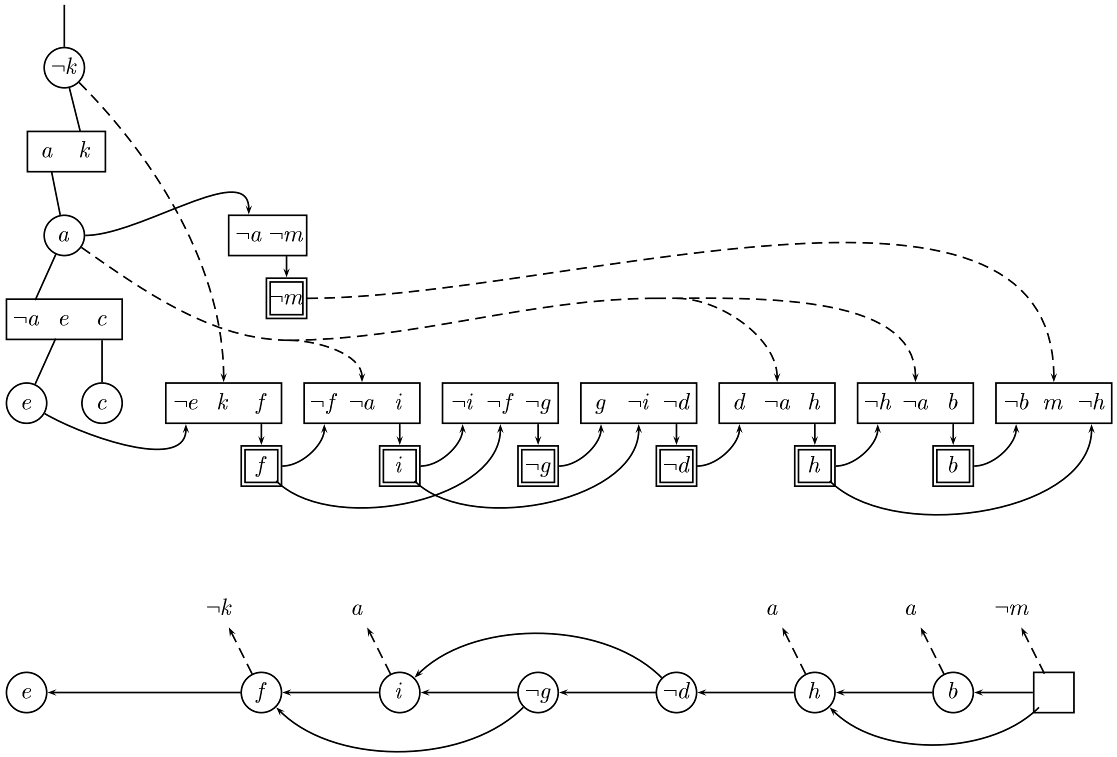

Figure 7. An example of E-literal derivation, discussed in Examples 15, 17, and 20.

Example 15. The upper part of Figure 7 is an example of E-literal derivation. To avoid clutter, parts of the derivation that are not germane to the discussion are omitted.

In Modoc, when subgoal $\neg k$ is selected, clause $[\neg e, k, f]$ is "pre-reduced" through unit implication. Then, goal $\neg k$ is extended with clause $[a, k]$, creating subgoal a. Subgoal a creates an eager literal (E-literal) $\neg m$ by unit-clause propagation. In the process, four clauses are "pre-reduced", as shown by the dashed arrows from a and $\neg m$.

Rather than extending with the clause $[\neg a, \neg m]$, we assume the program extends a with $[\neg a, e, c]$, as shown. This extension creates two subgoals, and we assume e is chosen first. When e is used to initiate unit-clause propagation, a series of six E-literals are derived by unit-clause propagation, as shown in double boxes, and the empty clause is derived. In this process, additional E-literals might have been derived, but are not shown, and the E-literals normally are not derived consecutively.

□

An interesting situation arises when the above unit-clause propagation derives the empty clause, that is, an eager refutation occurs. Clearly, this implies that a refutation of e exists with respect to $S|A$. To extract the quasi-persistent lemma with $\neg e$, the appropriate dependencies must be determined. The derivation of E-literals creates a directed acyclic graph (DAG) of dependencies among E-literals and the selected goal node e, as defined next.

Definition 16. (eager dependency DAG, articulation point) To simplify the definition, let us call e an E-literal, too. The *eager dependency DAG* associated with the eager (unit-clause) derivation of an empty clause contains nodes for various E-literals and one node for *false*, called the 0 node. For each node p *other than* e, define C_p to be the eager lemma associated with p; that is, the clause that was reduced to the single literal p during unit-clause propagation. If p was derived multiple times, only the earliest derivation is used. In the case that $p = 0$, C_0 is the clause that became empty. In this DAG, the edge $p \rightarrow q$ denotes that C_p contains $\neg q$; that is, the eager literal q contributed to the strengthening of C_p. Only nodes reachable from 0 are included in the eager dependency DAG. Node 0 is called the *source node* and node e is called the *sink node.*

For an eager dependency DAG an *articulation point* is a node v, different from 0 and e, such that every path from 0 to e passes through v. □

Example 17. The eager dependency DAG for Example 15 is illustrated in the lower part of Figure 7. The empty box denotes the 0 node. The articulation points are f, $\neg d$, and h. □

It is evident by the construction that the selected goal node e is reachable from every node in the eager dependency DAG, and thus it is the *only* sink node of the DAG. Recall that an *articulation point* in a connected undirected graph is a node v such that for some two other nodes x and y if v and its edges were removed from the graph, then there would be no path from x to y; that is, removal of v disconnects the graph. It is clear that if all edges of the eager dependency DAG were considered to be undirected, then v is an articulation point in the undirected sense if and only if v is an articulation point in the sense of Definition 16. The interest in articulation points is based on the following theorem and corollary.

Theorem 18. Let S be the formula and let A be the set of proper ancestors of a selected goal literal e, for which the eager dependency DAG derives $\emptyset$. With the foregoing notation, if p is an articulation point in the eager dependency DAG, then $S|A \vdash \neg p$. In other words, $\neg p$ can be soundly attached as a C-literal at a depth less than the depth of e. Moreover, the dependencies of $\neg p$ are the nonlocal dependencies of nodes having a path to p in the DAG (i.e., nodes between 0 and p).

Proof: Topologically number the nodes of the eager dependency DAG such that $n(v) < n(w)$ implies there is no path from v to w; thus $n(e) = 1$ and $n(0)$ is maximum.

For the remainder of this proof, p is either an articulation point or is the sink node e. Consider the subgraph G of nodes that can reach p, i.e., all nodes whose topological number is at least $n(p)$. Both p and 0 are in G, and p is the only sink node of G. Define S_G to be the set of clauses $\{C_w \mid w \in G \text{ and } w \neq p\}$. Recall that C_w is the eager lemma associated with w. Let A_G denote the union of all nonlocal dependencies of clauses in S_G; $A_G \subseteq A$. That is, A_G consists of the relevant ancestor literals whose complements occur in the clauses C_w for $w \neq p$ and $w \in G$.

We shall prove by induction on this topological order that, for all nodes $v \in G$, $p, S_G, A_G \vdash v$ (where "$\vdash$" denotes "derives by unit resolution using only clauses C_w, where $w \neq p$ and $w \in G$"). Then by considering the case of $v = 0$, and interpreting 0 as false, we have $S_G, A_G \vdash \neg p$, which is equivalent to the claimed quasi-persistent lemma, $S_G \vdash [\neg p \leftarrow A_G]$.

The base case is $v = p$ and is immediate. For $n(v) > n(p)$, let $v \to u_i$, $i = 1, \ldots, k$ be the DAG edges leaving v. By construction we have that $S_G, A_G, u_1, \ldots, u_k \vdash v$ (using only clause C_v). By the inductive hypothesis we have $p, S_G, A_G \vdash u_i$, for $i = 1, \ldots, k$. Therefore, $p, S_G, A_G \vdash v$. This concludes the induction. □

Corollary 19. With the foregoing notation, dependencies of the C-literal $\neg e$ are the nonlocal dependencies of all nodes other than e in the eager dependency DAG.

Proof: The proof of Theorem 18 showed this. □

If p is not an articulation point, then the refutation implied by the eager dependency DAG, as described in the proof of Theorem 18, uses some node later in the DAG (i.e., with a lower topological number), which is not in A. However, it is still possible that some other derivation exists, so we cannot make the theorem an "if and only if" statement.

Example 20. Continuing with Example 15, $\neg h$, d, and $\neg f$ are the complements of the articulation points of the DAG and they become C-literals. The corresponding quasi-persistent lemmas are $[\neg h \leftarrow a]$, $[d \leftarrow a]$, and $[\neg f \leftarrow a, \neg k]$. The higher-level E-literal $\neg m$ is not itself a dependency, but the ancestors upon which it depends are dependencies of the new C-literals. In this case, $\neg m$ depends on a only. Also, $\neg e$ becomes a C-literal with dependencies a and $\neg k$; i.e., the actual lemma is $[\neg e \leftarrow a, \neg k]$.

These C-literals could have been derived by the normal operations of Model Elimination, but possibly only after lengthy fruitless side trails. To find such a derivation, perform a series of extensions by following DAG edges backwards from e to 0. Now all the articulation points are ancestors. Complete the refutation of e using only clauses in the DAG. □

5. Experimental Results

An efficient implementation of Modoc was programmed in C. Hereinafter, we refer to this C implementation of Modoc as `modoc`. This section reports on performance tests. The CPU time for a Sun Sparcstation 10/41 is reported.

The effects of various lemma and cut strategies are summarized, and compared with `2cl` [VGT96], a model-search program, in Table 1. Background on the formulas and `2cl` is reported elsewhere [VGT96].

The formula classes `rand100` and `rand141` are random 3CNF formulas generated at the clauses-to-variables ratio of 4.27. It is believed that this ratio generates the hardest random 3CNF formulas [MSL92,LT92]. Each formula in `rand100` is on 100 variables and contains 427 clauses. Each formula in `rand141` is on 141 variables and contains 602 clauses.

The formula classes `bf2670` and `ssa2670` originally derived from an automated test pattern generation (ATPG) application [Lar92]. Circuits are taken from the ISCAS85 benchmark. Bridge faults are coded as "bf" and single stuck-at faults are coded as "ssa". Each formula is satisfiable if and only if there is some circuit input for which the correct circuit and a faulty circuit generate a different output. A set of such formulas was contributed to the 1993 *DIMACS Imple-*

Table 1

Improvements due to various lemma and cuts strategies. Times are CPU seconds on a Sun Sparcstation 10/41. Classes `rand100` and `rand141` consist of random 3CNF formulas at the "hard" clause/variable ratio of 4.27, with 100 and 141 variables, respectively. Classes `bf2670` and `ssa2670` derive from the most challenging circuit in the ISCAS-85 benchmark.

		Lemma Features in `modoc`									
formula	num.	none		QP		QP, cuts		QP, cuts, eager		`2cl`	
class	fmlas	avg	max	avg	max	avg	max	avg	max	avg	max
`rand100`	200	≥1200	≥1200	141	485	19	73	1.54	4.74	1.10	3.23
`rand141`	200	≥1200	≥1200	≥968	≥1200	481	1687	25	90	9	22
`bf2670`	53	≥1177	≥1200	≥832	≥1200	396	6124	7	181	439	17725
`ssa2670`	12	≥1200	≥1200	≥1200	≥1200	240	868	30	119	1540	2424

mentation Challenge on Cliques, Coloring and Satisfiability [JT96], and is available from the DIMACS ftp site, `ftp://dimacs.rutgers.edu/pub/challenge/-satisfiability/contributed/UCSC/instances/`. Table 1 shows results on the most challenging circuit in the set, which has 2670 gates.

The table shows significant improvements as various lemma strategies and cuts are incorporated into `modoc`. The column headed "none" corresponds to Model Elimination with autarky pruning incorporated, but no lemma features. The next column adds quasi-persistent lemmas (which are quite similar to traditional lemmas for ME and C-literals). Remaining columns show the incremental effects of the novel lemma-formation strategies described in this paper. We observe that `modoc` runs 50–60 times faster, on average, than `2cl` on the circuit formulas, but runs slower than `2cl` on the random formulas. Since `2cl` was developed, some newer solvers have been reported [SS96,Zha97] that are faster on the circuit formulas, but slower on the random formulas, compared to `2cl`.

Growth rates on random 3CNF formulas are shown in Figure 8, for `modoc` and two model-searching programs, `2cl` and an efficient `DPLL` (Davis-Putnam-Logemann-Loveland), both of which use linear-time techniques for unit-clause propagation [DE92], unit subsumption, and pure literal pruning. This growth plot indicates that `modoc` still has a higher asymptotic growth rate than model-searching methods, on random formulas.

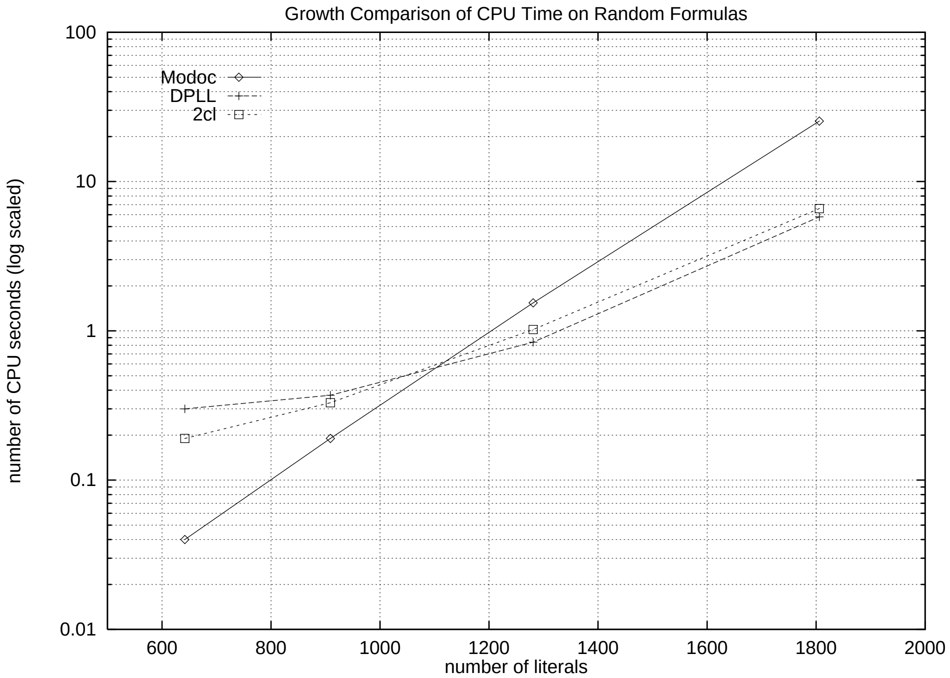

Figure 8. Growth comparison on samples of 200 random 3CNF formulas with 4.27 clauses per variable. (E.g., 100 variables corresponds to 1281 literals.)

6. Conclusions and Future Work

We have introduced lemma and cut strategies that are efficient to apply in propositional Model Elimination. The quasi-persistent strategy retains lemmas longer than the previously reported C-literal strategy. The eager lemma strategy allows early identification of successful (sub)refutations. Two forms of controlled cut further improve the search.

Tests reported in Section 5 contain substantial evidence that the lemma and cut strategies, presented in this paper, together with autarky pruning, allow propositional Model Elimination to overcome its major inefficiency. Modoc is not yet competitive with the leading model-search methods on random formulas, but outperforms them on formulas derived from a circuit-fault testing application. Other lemma-based methods that were recently reported show a similar pattern [SS96,Zha97].

Future work should investigate further improvements to the eager lemma idea, including binary clause reasoning, and global assertion of "powerful" lemmas. Heuristics for guiding the resolution search are needed. An extension to first-order theorem proving might be possible.

Acknowledgements

This work was supported in part by NSF grant CCR-95-03830, by equipment donations from Sun Microsystems, Inc., and software donations from Quintus Computer Systems, Inc. We thank Dr. Reinhold Letz for helpful discussions on many aspects of Model Elimination and tableau methods. We thank the anonymous referees for their careful reading of the paper and for many helpful comments.

References

[AL97] O. L. Astrachan and D. W. Loveland. The use of lemmas in the model elimination procedure. *Journal of Automated Reasoning*, 19:117–141, 1997.

[AS92] O. L. Astrachan and M. E. Stickel. Caching and lemmaizing in model elimination theorem provers. In D. Kapur, editor, *Automated Deduction - CADE-11. Proceedings of 11th International Conference on Automated Deduction (Saratoga Springs, NY, USA, 15-18 June 1992)*, pages 224–38. Springer-Verlag, Berlin, Germany, 1992.

[DE92] M. Dalal and D. Etherington. A hierarchy of tractable satisfiability problems. *Information Processing Letters*, 44:173–180, December 1992.

[FLSY74] S. Fleisig, D. W. Loveland, A. K. Smiley, and D. L. Yarmush. An implementation of the model elimination proof procedure. *JACM*, 21(1):124–139, 1974.

[JT96] D. S. Johnson and M. A. Trick, editors. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science.* American Mathematical Society, 1996.

[Lar92] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):6–22, January 1992.

[LMG94] R. Letz, K. Mayr, and C. Goller. Controlled integration of the cut rule into connection tableau calculi. *Journal of Automated Reasoning*, 13(3):297–337, December 1994.

[Lov68] D. W. Loveland. Mechanical theorem-proving by model elimination. *Journal of the ACM*, 15(2):236–251, April 1968.

[Lov69] D. W. Loveland. A simplified format for the model elimination theorem-proving procedure. *Journal of the Association for Computing Machinery*, 16(3):349–363, July 1969.

[Lov78] D. W. Loveland. *Automated Theorem Proving: A Logical Basis.* North-Holland, Amsterdam, 1978.

[LT92] T. Larrabee and Y. Tsuji. Evidence for a satisfiability threshold for random 3CNF formulas. Technical Report UCSC–CRL–92–42, UC Santa Cruz, Santa Cruz, CA., October 1992.

[MS85] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10:287–295, 1985.

[MSL92] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92), San Jose, CA.*, pages 459–465, July 1992.

[MZ82] J. Minker and G Zanon. An extension to linear resolution with selection function. *Information Processing Letters*, 14(3):191–194, June 1982.

[Sho74] R. E. Shostak. *A Graph-Theoretic View of Resolution Theorem-Proving*. PhD thesis, Center for Research in Computing Technology, Harvard University, 1974. Also available from CSL, SRI International, Menlo Park, CA.

[Sho76] R. E. Shostak. Refutation graphs. *Artificial Intelligence*, 7:51–64, 1976.

[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.

[Sti94] M. E. Stickel. Upside-down meta-interpretation of the model elimination theorem-proving procedure for deduction and abduction. *Journal of Automated Reasoning*, 13(2):189–210, October 1994.

[VG97] A. Van Gelder. Autarky pruning in propositional model elimination reduces failure redundancy. *Journal of Automated Reasoning*, 1997. (to appear, preprint at `ftp://ftp.cse.ucsc.edu/pub/avg/JAR/aut-jar-dist.ps.Z`).

[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.

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