The Partial Rehabilitation of Propositional Resolution

Allen Van Gelder Fumiaki Kamiya
Baskin Center for Computer Engineering and Information Sciences
University of California, Santa Cruz 95064

UCSC-CRL-96-04

E-mail avg@cs.ucsc.edu kamiya@cs.ucsc.edu.

July 26, 1996

Abstract

Resolution has not been an effective tool for deciding satisfiability of propositional CNF formulas, due to explosion of the search space, particularly when the formula is satisfiable. A new pruning method is described, which is designed to eliminate certain refutation attempts that cannot succeed. The method exploits the concept of "autarky", which was introduced by Monien and Speckenmeyer. New forms of lemma creation are also introduced, which eliminate the need to carry out refutation attempts that must succeed. The resulting algorithm, called "Modoc", is a modification of propositional model elimination. Informally, an autarky is a "self-sufficient" model for some clauses, but which does not affect the remaining clauses of the formula. Whereas Monien and Speckenmeyer's work was oriented toward finding a model, our method has as its primary goal to find a refutation in the style of model elimination. However, Modoc finds a model if it fails to find a refutation, essentially by combining autarkies. Unlike the pruning strategies of most refinements of resolution, autarky-related pruning only prunes refutation attempts that ultimately will be unsuccessful; consequently, it will not force the underlying search to find an unnecessarily long refutation. The other major innovation is Modoc's lemma management. Building upon the C-literal strategy proposed by Shostak and studied further by Letz, Mayr and Goller, methods for "eager" lemmas, "quasi-persistent" lemmas, and two forms of controlled cut are described. Experimental data based on an implementation in C is reported. On random formulas, Modoc is not as effective as recently reported model-searching methods. On more structured formulas, such as circuit-fault detection, it is superior.

Key Words:

Satisfiability, Boolean formula, propositional formula, autarky, resolution, refutation, model, theorem proving, model elimination.

1 Introduction

The decision problem of Boolean, or propositional, satisfiability is the "original" NP-hard problem. We assume the reader is generally familiar with it. We shall consider exclusively propositional formulas in *conjunctive normal form* (CNF), also called *clause form*. Each clause is a disjunction of literals, and clauses are joined conjunctively. A closely related problem is to determine *validity* of a formula in *disjunctive* normal form. As *language recognition* problems, satisfiability is in NP, while validity is in co-NP. However, as *decision* problems, they are essentially equivalent, as both "yes" and "no" answers must be produced. As a practical consideration, a program may be required to produce "evidence", or a "certificate", to support its decision, an issue discussed by John Slaney at CADE-12 [Sla94]. In this setting, a *certificate* is a file that can be processed by an independently written program, to verify the solver's conclusion, using simple, highly trusted, computations. If critical decisions will be based on the program's output, such a proof would permit an independent, and straightforward, verification of the program's conclusion. To our knowledge, no previously existing implementations can produce a proof for both "yes" and "no" decisions.

Two basic methods have been developed for satisfiability testing: refutation search and model search.

1. Refutation search seeks to discover a proof that a formula is unsatisfiable, usually employing resolution. If a complete search for a refutation fails, the formula is pronounced satisfiable. Model elimination and SL-resolution typify these methods [Lov69, KK71, Lov72].

2. Model search seeks to discover a satisfying assignment, or model, for the formula. If a complete search for a model fails, the formula is pronounced unsatisfiable. The DPLL algorithm, due to Davis, Putnam, Logemann and Loveland [DP60, DLL62] is the basis for many modern refinements. A different approach is to treat the problem in terms of integer linear programming [BJL86, JW90, HF90, HHT94].

   Several methods use incomplete model searches, so they can only report "don't know" and give up based on resource limits, when they fail to discover a model [SLM92, GW93, SKC95]. However, they have succeeded in finding models on much larger formulas than current complete methods can handle.

Current propositional methods are "one-sided" in the information that they can provide to support their answers. Specifically, refutation methods can produce a proof of unsatisfiability, which is easily checkable, but can only answer "sat" on satisfiable formulas. On the other hand, model-search methods can exhibit a model, again easily checkable, for satisfiable formulas, but can only answer "unsat" on unsatisfiable formulas. This paper describes an integrated propositional approach that simultaneously searches for either a refutation or a model.

In the first-order arena there has been some work on searching for a model as well as a refutation, but it does not seem to carry over effectively to the propositional case. The method of Fermuller and Leitsch [FL93], which is intended to show decidability of certain classes of first-order formulas, first performs hyper-resolution to saturation, but this is too expensive in the propositional setting to be practical. The method of Caferra adds equations soundly to the original formula, again proving decidability of certain first-order classes [Caf93]. Since the equations are based on unifying substitutions, there is no apparent way to use this method in the propositional case. Finally, *failure caching* on first-order Horn formulas has been reported [Elk89, AS92], but *propositional* Horn formulas are not challenging.

Several high performance satisfiability testers have been reported in recent years [Lar92, CA93, ZS94, SFS95, DABC95, JSD95, Pre95, VGT96, *inter alia*]. Interestingly, they have primarily, if not exclusively, been based upon the model-search paradigm. This contrasts with the situation of first-order theorem provers, which are primarily refutation-based.

One problem with existing propositional refutation methods is illustrated dramatically in Figure 1. Model elimination (abbreviated M.E., and regarded as one of the most efficient refutation strategies) is able to solve

		Unsatisfiable			Satisfiable		
Formula Size		No. of	Extensions		No. of	Extensions	
Vars	Clauses	Samples	Avg.	Max	Samples	Avg.	Max
14	60	7	11	17	13	184,888	542,254
15	64	7	13	20	13	573,794	2,019,490
16	69	10	18	39	10	1,016,480	2,462,290
17	73	4	21	36	16	2,623,570	9,160,390

Figure 1: Comparative performances of model elimination on unsatisfiable and satisfiable random 3CNF formulas. On satisfiable formulas a refutation was attempted with each clause as top clause, and the average was computed for that formula. (E.g., some satisfiable 10-variable formula *averaged* 12,068 extensions per top clause.) Implementation and experimental details are given in later sections, along with results for the new Modoc algorithm.

		Unsatisfiable			Satisfiable		
Avg. Formula Size		No. of	Extensions		No. of	Extensions	
Vars	Clauses	Samples	Avg.	Max	Samples	Avg.	Max
29	41	15	9.5	49			
29	53				6	217.3	1,042

Figure 2: Comparison of model elimination on unsatisfiable and satisfiable propositional formulas having at least 12 variables in the TPTP library v1.2.0. Satisfiable formulas were handled as described above.

---

unsatisfiable random 3CNF formulas with up to 100 variables, but it bogs down on *satisfiable* formulas about at 17 variables. It should be noted that, for modern model-search methods, these formulas (both unsatisfiable and satisfiable) are considered *easy* at 100 variables and *trivial* at 20 variables. The same phenomenon occurs on propositional formulas found[1] in the TPTP library v1.2.0 [SS95], as shown in Figure 2. Keep in mind that the numbers shown reflect the time for *one attempt* on a satisfiable formula. To conclude that it is satisfiable based on M.E., numerous attempts must be made.

Three reasons have been mentioned for the lack of high performance propositional refutation systems:

1. The search space is too large, particularly for satisfiable formulas, as shown in Figure 1.

2. The method cannot produce models.

3. The most efficient methods are only guaranteed to find a refutation when some *top clause* is known to be in a minimal unsatisfiable subset of clauses; in some applications such a top clause is not known *a priori*.

The integrated approach of this paper addresses all three of these problems.

The main results are summarized in Section 1.1 and informally illustrated with examples in Section 1.2. Notation and terminology is covered in Section 2. Tree structures for M.E. and background are reviewed in Section 3. Autarky pruning is described in Section 5. New lemma creation mechanisms are described in Section 6, which also briefly reviews previous work on lemmas in M.E. Experimental results based on an efficient C implementation are reported in Section 7. Conclusions and future work are in Section 8.

[1]SAT: num285-1, syn086-1, syn087-1, syn091-1, syn092-1, syn302-1; UNSAT: msc007-1.005, puz013-1, puz014-1, puz015-2, puz016-2, puz033-1, syn003-1, syn004-1, syn010-1, syn089-1, syn090-1, syn093-1, syn094-1, syn097-1, syn098-1. TPTP includes formula generators, but these were not used.

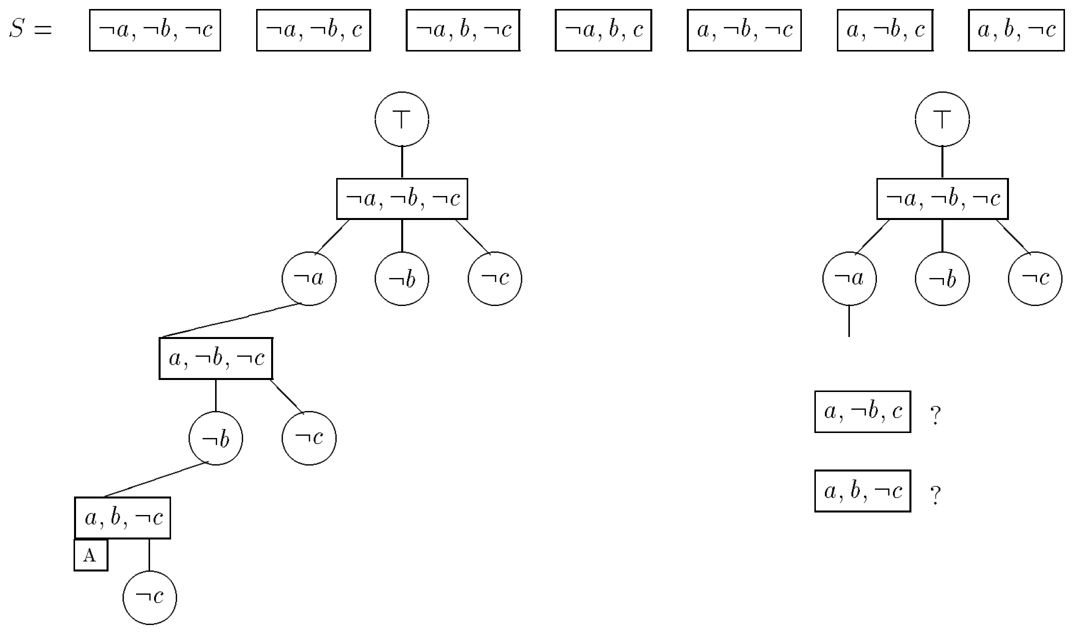

Figure 3: Model elimination search for Example 1.1. (Left) Search fails at lowest $\neg c$ goal. (Right) After backtracking to alternative choices at $\neg a$ goal.

1.1 Summary of Results

This paper reports on the combination of autarky pruning with extended lemma creation methods. Roughly speaking, lemmas and autarkies are duals of each other: a lemma permits curtailment of a resolution attempt that is certain to succeed, while an autarky permits curtailment of a resolution attempt that is certain to fail. Experimental data is based on an efficient C implementation.

New lemma creation mechanisms, called "eager" lemmas and "quasi-persistent" lemmas are described in Section 6. That section also briefly reviews previous work on lemmas in M.E., and discusses the compatibility of autarky pruning and lemma creation. Quasi-persistent lemmas are a variant of the "C-literal" strategy, adapted for efficiency in the propositional case. Eager lemmas supplement them to provide substantial further reductions in proof searching by providing early identification of refutations that will succeed. Derivation of eager lemmas is closely related to unit clause propagation.

When the refutation of a literal succeeds, the complement of that literal becomes a quasi-persistent lemma and the complements of certain eager lemmas become quasi-persistent lemmas also. 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. Articulation points can be identified in linear time by a standard graph algorithm.

Several propositional techniques are used that are not possible, or considerably more complex, in first-order resolution. These issues are mentioned briefly in Section 4.2, Section 6.2, and Section 6.5.

1.2 Informal Overview

For readers familiar with model elimination adapted to tree structures [MZ82, LMG94], we now give simple examples of how autarky pruning and eager lemmas work. Other readers may wish to come back to this example after reading more of the expository material in Section 3.

Example 1.1: The formula S consists of all 3CNF clauses on variables a, b, and c, except for the all positive clause, as shown at the top of Figure 3. Suppose the top clause is all negative. Let us trace out a model elimination search for a refutation. As shown on the left of Figure 3, literal $\neg a$ resolves with clause $[a, \neg b, \neg c]$, then literal $\neg b$ resolves with clause $[a, b, \neg c]$. In the latter clause, literal a can be reduced with the ancestor (or A-literal) $\neg a$, as indicated by the boxed "A". Literal $\neg c$ remains to be refuted, but the search procedure now fails, because each clause containing literal c also contains an ancestor literal.

If we stop and reflect on the meaning of this failure, we see that every clause containing the literal c is satisfied by a partial assignment consisting of the ancestors (A-literals) on this branch, specifically:

$$M = \{\neg a, \neg b, \neg c\}.$$

(In this case the partial assignment happens to be a total assignment.) But obviously, every clause containing the literal $\neg c$ is also satisfied by M, so we conclude that every clause involving the *variable* c is satisfied by M.

Model elimination now backtracks and looks for another clause that resolves with $\neg b$, and does not contain the ancestor $\neg a$. There are none. We can now extend the conclusion of the previous paragraph to say that every clause containing either of the *variables* b or c, either positively or negatively, is satisfied by the partial assignment M.

Model elimination again backtracks and looks for another clause that resolves with $\neg a$ (Figure 3, right). There are two such clauses, as indicated. The standard algorithm would continue trying to construct a refutation using one of these clauses, then the other. But notice that both of these clauses are satisfied by the partial assignment M mentioned above.

After a few moments thought, we can predict that these refutation attempts must fail, without carrying out the search. Intuitively, the reason is that we cannot use a clause that is satisfied by M to "get outside of M". Every extension will have a subgoal that is satisfied by M; no such subgoal can be handled by reduction because all the ancestors are also in M. Eventually, some goal is generated that has no eligible extensions.

Finally, we conclude that the partial assignment M satisfies all clauses in which any of the variables a, b, or c appears. This conclusion holds up even if we add additional clauses to S that do not involve the variables a, b and c. We call such a partial assignment an *autarky* (see Definition 5.1).

This example illustrates, in an over-simplified way, that:

1. Autarky analysis can predict that certain refutation attempts must fail;

2. A model for a satisfiable formula can be constructed as a series of autarkies.

□

Example 1.2: Figure 4 illustrates the main idea of eager lemmas, introduced in Section 6.5. 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 (Definition 2.3). 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.

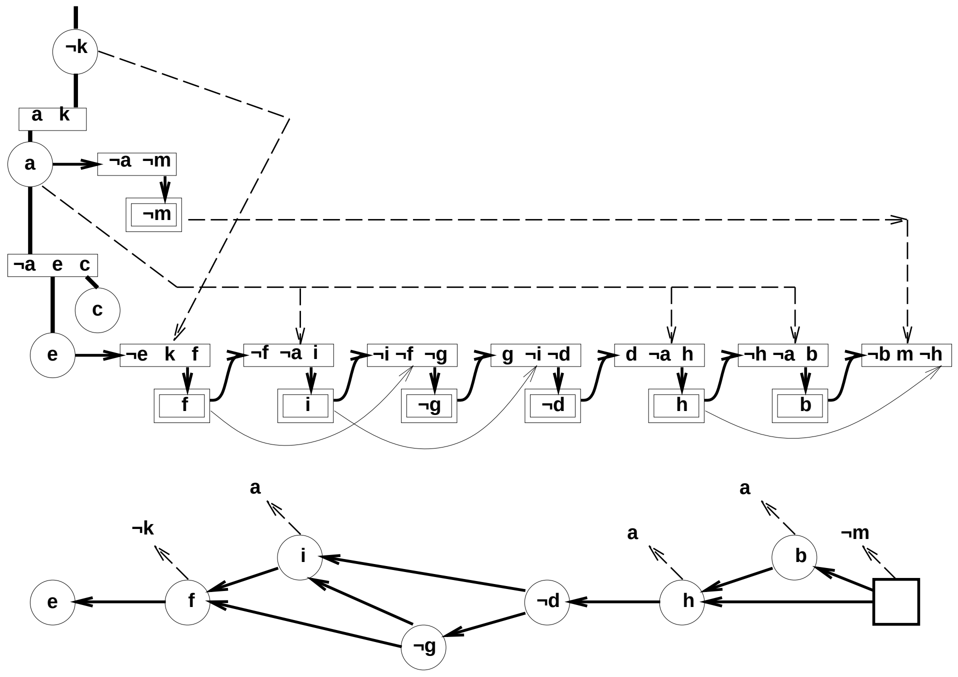

Figure 4: Eager lemmas, discussed in Example 1.2.

Since the empty clause was derived, we know that e can be refuted. To find the clauses relevant to the refutation, we want to work back from the empty clause to e. Remember that there are normally irrelevant E-literals as well. As each E-literal is derived, the clause that led to its derivation, called its *eager parent*, is associated with it. Eager parents are shown immediately above their E-literals. Working back from the empty clause, a directed acyclic graph (DAG) is defined by constructing edges from an E-literal (e.g., $\neg d$) to the E-literals whose complements appear in the clause that derived it (e.g., i and $\neg g$), as shown in the lower part of the figure. This is called the *eager dependency DAG*. Edges to literals that are higher in the tree than e (shown as dashed lines) are not considered part of the DAG.

The key theorem (Theorem 6.1) is that articulation nodes of the DAG are precisely the E-literals whose complements become quasi-persistent lemmas (similar to C-literals). Of course, $\neg e$, the complement of the refuted subgoal, becomes a quasi-persistent lemma, as well. A further interesting property is that the dependencies are found by following the dashed edges from DAG nodes that have a path to the articulation node.

Thus in this example, $\neg h$, d and $\neg f$ become C-literals with dependency a. (The actual lemmas are $[\neg h, \neg a]$, etc.) The E-literal $\neg m$ is not itself a dependency, but the ancestors upon which it depends are; in this case, $\neg m$ depends on a only. Also, $\neg e$ becomes a C-literal with dependencies a and $\neg k$. These C-literals could have been derived by the normal operations of model elimination, but possibly only after lengthy fruitless side trails. □

2 Preliminaries

Standard terminology for conjunctive normal form (CNF) formulas, (disjunctive) clauses, literals, and propositional variables, is used. A finite set of propositional variables is fixed throughout the discussion. If F is a formula, $lits(F)$ denotes the set of literals composed from variables occurring in F.

Definition 2.1: (literal, clause, formula, lits, +, −) The expression $[p_1, \ldots, p_k]$ denotes the disjunctive clause that consists exactly of those literals. The empty clause is denoted $\emptyset$. For brevity, unit clause $[q]$ will simply be denoted as q. Similarly, a formula $\{C\}$ consisting of a single clause may be denoted as C.

We denote *disjoint union* by "+" which is only defined when the operands are disjoint, and *set difference* by "−" which is only defined when the second operand is a subset of the first. □

As an example of overloading, if F denotes a formula (which does not contain the unit clause $[q]$), then $F + q$ denotes $F \cup \{[q]\}$.

Definition 2.2: (assignment, satisfaction, model) A *partial assignment* is a partial function from the set of variables into $\{false, true\}$, and is extended to literals, clauses, and formulas in the standard way. It is conventionally represented by the (necessarily consistent) set of *unit clauses* that it maps into *true*. A clause or formula is *satisfied* by a partial assignment if it is mapped to *true*.

Departing somewhat from standard terminology, A *partial* assignment that satisfies a formula is called a *model* of that formula. □

Definition 2.3: (unit implication, unit subsumption, strengthened formula) 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

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

That is, complements of literals in M are removed from C. This operation is called *unit implication*. Unit implication applied to a formula consists of applying it to each clause.

The operation of *unit subsumption* takes a partial assignment M and a formula S and outputs the formula consisting of all clauses of S that do not contain any literal of M.

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

$$S|M = \left\{C|M \;\middle|\; C \in S \text{ and } C \text{ contains no literal of } M\right\}$$

That is $S|M$ results from applying both unit implication and unit subsumption to S. □

Example 2.1: Let S consist of $[a, b]$, $[\neg a, c]$, and $[b, d]$. Then $S|\{a\} = \{[c], [b, d]\}$, and $S|\{a, c\} = \{[b, d]\}$. □

3 Clause and Derivation Trees

This section describes the tree data structures we shall use, which are chosen especially for propositional resolution. Trees are now recognized as the most appropriate data structures for representation of linear resolution derivations, following Minker and Zanon [MZ82]. Model elimination originally used a chain format [Lov69]. 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. The following technical definitions are illustrated in Example 3.1 and Figure 5.

Definition 3.1: (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 have only goal nodes as children and *vice versa*. 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.

Each clause node is labeled with a clause of S, and each goal node is labeled with a literal of $\mathit{lits}(S)$, or with $\top$. Usually, a node is identified with its label, but when it is necessary to name a specific node, we assume some structural naming scheme. We write $v(q)$ to denote the goal node whose structural name is v and whose label is q, and write $w[C]$ to denote the clause node whose structural name is w and whose label is C.

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 as $\mathit{ancs}(v)$. While $\mathit{ancs}(v)$ 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. $\square$

Definition 3.2: (propositional derivation tree (PDT), PDT extension) Let S and $\top$ be as in Definition 3.1. Throughout this definition all literals are assumed to be in $\mathit{lits}(S)$ and all clauses are assumed to be in S.

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. $\square$

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 $w[C]$ is a leaf in a PDT if and only if every literal in C is complemented in $\mathit{ancs}(w)$.

Definition 3.3: (refutation) If $v(q)$ is the root of a PDT that contains only clause nodes as leaves, then this PDT is called a *refutation of q with respect to S*. If $q = \top$, it is called simply a *refutation of S*. (These terms are justified in Theorem 3.1.) $\square$

Example 3.1: Figure 5 shows PDT examples. The PDT at the right is obtained by PDT extension using the other two. The left side of Figure 3 shows a clause node in which implicit reduction has occurred, as indicated by the boxed "A". (This box is a notation, but not a structural part of the tree.) $\square$

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" normally must 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 3.3 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 [VG95].

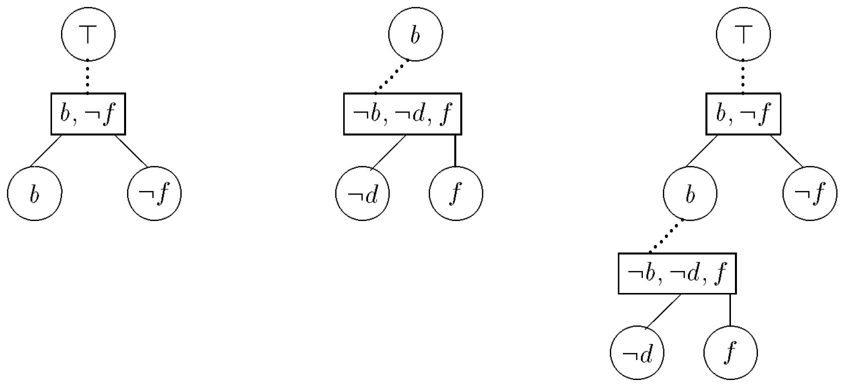

Figure 5: Propositional derivation trees (PDTs) discussed in Example 3.1. Left: A PDT with *top-clause* $[b, \neg f]$. Center: The extension clause $[\neg b, \neg d, f]$ may be thought of as a single-clause PDT. Right: the result of PDT extension.

Theorem 3.1:

(A) If there is a PDT T that is a refutation of q w.r.t 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.

▮

4 Search Trees

This section describes "propositional derivation search trees". These structures have some similarity in purpose to tableau search trees [LMG94]. However, there are some essential structural differences, as discussed in Section 4.2, which are again based on exploiting the simplifications available in the propositional logic. Propositional model elimination can be viewed as exploring PDSTs.

4.1 PDST Basics

Definition 4.1: (propositional derivation search tree (PDST)) Let S and $\top$ be as in Definition 3.1. A propositional derivation search tree (PDST) is a clause-goal tree that differs from a PDT (Definition 3.2) only in that a nonleaf goal node, instead of having one clause child, has a clause child for each clause that *might* appear in that position in a PDT. □

It is convenient to visualize a PDST as a three-dimensional tree, in which goal children of a given clause are arranged left to right, and clause children of a given goal are arranged front to back (see Figure 6, discussed next). Again, it is easy to see that a clause node $w[C]$ is a leaf in a PDST if and only if every literal in C is complemented in $ancs(w)$.

Example 4.1: Consider the clause set S shown in Figure 6. The PDST with top clause $[b, \neg f]$ is also shown. Observe that $[c, \neg d]$ does not occur in the PDST. The boxed "A"s are not a structural part of the tree, but are notations to indicate literals that are subject to implicit reduction. □

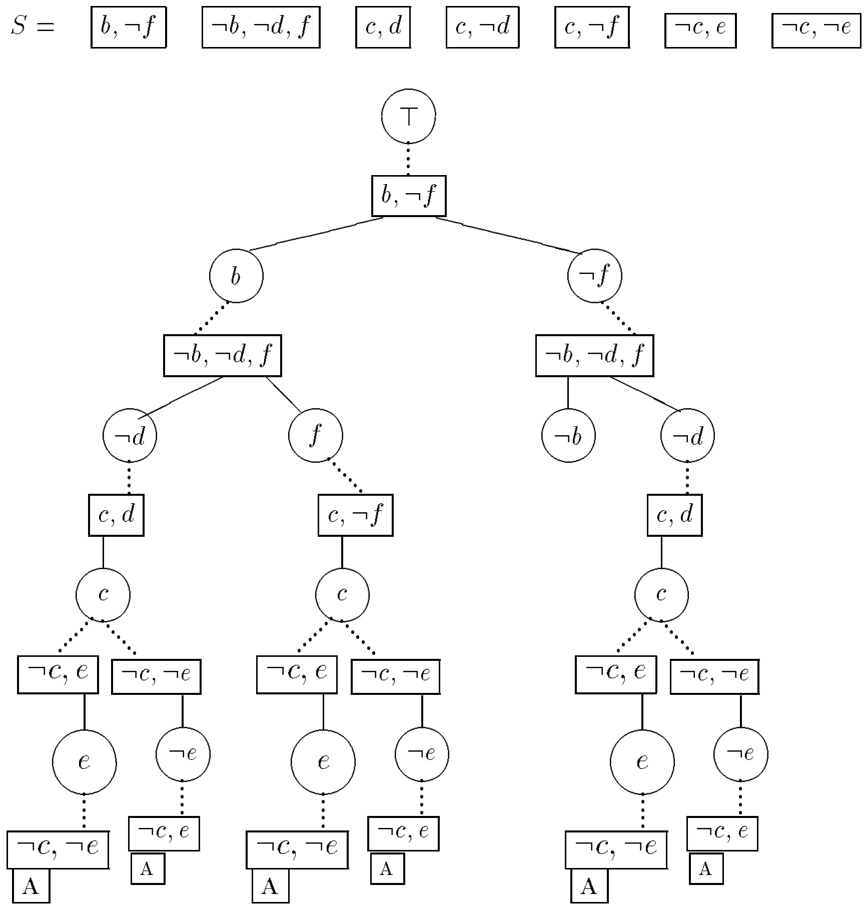

Figure 6: A completed propositional derivation search tree (PDST) with top clause $[b, \neg f]$.

Definition 4.2: (failed goal node, completed PDST, universal PDST) A *failed goal node* in a PDST is a leaf node $v(q)$ such that every clause of S that contains $\neg q$ also contains some literal in $ancs(v(q))$; the branch ending at $v(q)$ is called a *failed branch*.

A PDST is said to be *completed* if every leaf that is a goal node is failed. In this case no further PDST extension is possible.

A *universal PDST for S* is constructed as follows: for each clause $C^{(j)} \in S$ create a completed PDST for S with $C^{(j)}$ as top clause, then merge all their roots. □

We now state some structural properties of PDSTs ([VG95]), which provide the foundation for our implementation of propositional model elimination. The algorithm searches the universal PDST by constructing a representation of the strengthened formula mentioned in the next lemma, using efficient data structures.

Lemma 4.1: Let T be a PDST for S with root $v(q)$, where q is a literal. Let $w[C]$ be a child of v. Then a PDST for $S|\{q\}$ with top clause $C|\{q\}$ (call it T') may be formed as follows:

1. The root of T' is $v'(\top)$;

2. The single subtree of v' is a copy of the tree rooted at w, except the clause labels are strengthened by $\{q\}$.

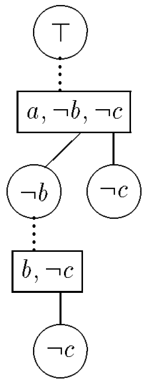

Figure 7: A PDST for a strengthened formula, as discussed in Example 4.2.

▮

Lemma 4.2: Let T be a PDST for S with top clause node $w[C]$. Let $v(q)$ be a child of w. Then the tree rooted at v is a PDST for S with top goal q. ▮

Corollary 4.3: For a given formula S and a given top clause C or given top goal q, the completed PDST is unique, up to reordering of children. Also, this is true for the universal PDST. ▮

Example 4.2: Consider the PDST on the left of Figure 7. The strengthening of S with $\neg a$ yields

$$S|\neg a = \{[\neg b, \neg c], [\neg b, c], [b, \neg c]\}$$

To illustrate Lemma 4.1, the completed PDST for $S|\neg a$ with top clause $[\neg b, \neg c]$ is essentially the left branch of left PDST of Figure 7, with the goal $\neg a$ replaced by $\top$. Clauses containing literal a are replaced by their strengthened forms, producing the PDST shown on the right. □

4.2 Difference from Tableau Search Tree

PDSTs do not generalize straightforwardly to the first-order case. The reason is that a substitution must be applied throughout the derivation tree, not just to the subtree where the goal is unified with a clause. Consequently, in *tableau search trees* [LMG94], a search tree node is labeled with an *entire derivation tree*. An example of a tableau search tree is shown in Figure 8. In a first-order version of this example, the five occurrences of the goal $\neg a$ would in general be different due to differing substitutions in the various search nodes.

5 Autarkies

This section defines "autarky" and indicates how the concept is used in Modoc. The potential value of autarkies is suggested in Lemma 5.1, following the definition. Theorems 5.2 and 5.3 establish connections between autarkies and PDSTs. The application to refutation search efficiency is sketched at the end of the section. All topics of this section are described in more detail in another report [VG95].

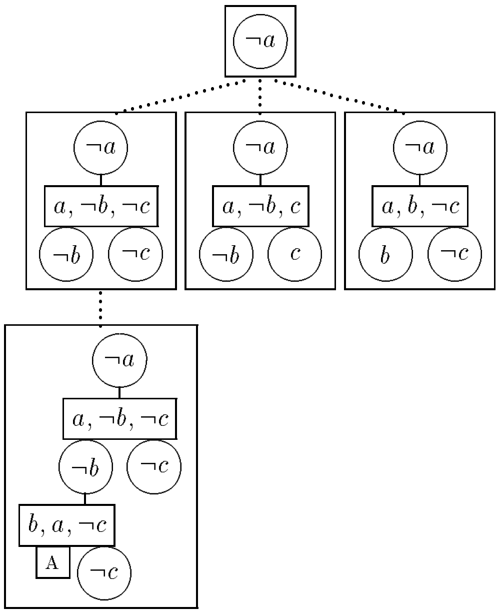

Figure 8: The tableau search tree that corresponds to the PDST on the left of Figure 7.

The concept of "autarky" was (to our knowledge) introduced into logic by Monien and Speckenmeyer, who proposed a new model searching algorithm based on it [MS85]. The word "autarky", used mainly in economics, literally means "self-sufficient country or region".

Definition 5.1: (autarky, *autsat*, *autrem***)** Let S be a set of CNF clauses. A partial assignment M (Definition 2.2), possibly defined on some variables that do not occur in S, is called an *autarky* of S if M partitions S into two disjoint sets,

$$S = autsat(S, M) + autrem(S, M)$$

such that each clause in $autsat(S, M)$ is satisfied by M and each clause in $autrem(S, M)$ has no variables in common with the variables that occur in M. In particular, no literal of a clause in $autrem(S, M)$ is *complemented* in M. □

Lemma 5.1: Let M be an autarky of formula S.

(A) $S|M = autrem(S, M)$.

(B) If S is unsatisfiable, then $autrem(S, M)$ is also unsatisfiable.

(C) If S is satisfiable, then M can be extended to a model of S.

▮

Example 5.1: As in Example 2.1, let

$$S = \{[a, b], [\neg a, c], [b, d]\}$$

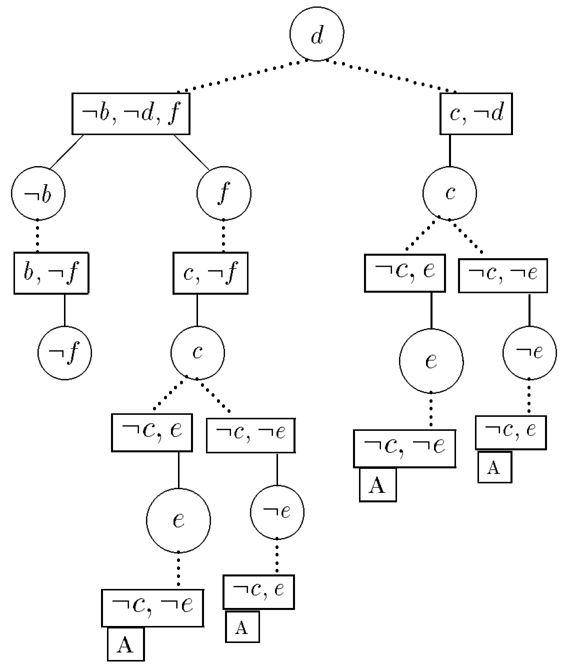

Figure 9: A completed propositional derivation search tree (PDST) with top goal d for the same formula as Figure 6.

Then $\{a, c\}$ is an autarky of S, with

$$autsat(S, \{a, c\}) = \{[a, b], [\neg a, c]\}$$
$$autrem(S, \{a, c\}) = \{[b, d]\}$$

However, $\{a\}$ is not an autarky because of clause $[\neg a, c]$. $\square$

As seen in the previous example, another way to characterize an autarky M is that $S|M \subseteq S$, that is, no clauses are *shortened* by the strengthening, although some clauses may be deleted.

The following theorems indicate how autarkies can interact with a refutation search. The first theorem shows that clauses satisfied by an autarky can be ignored, and the second shows how autarkies can be expanded by failed refutation searches.

Example 5.2: The next theorem is illustrated by Example 4.1 and Figure 6. Let $M = \{\neg b, \neg f\}$, which is an autarky. We see that there is no refutation beginning with the top clause $[b, \neg f]$, so the theorem holds in this case.

Now consider a different PDST for the same S, this time with top goal d, which has two clause children, $[\neg b, \neg d, f]$ and $[c, \neg d]$, as shown in Figure 9. The theorem asserts that a refutation attempt cannot succeed by choosing the first clause, because M satisfies it. Indeed, if the program did choose the first clause, then it would need to refute the resulting subgoal that occurs in M, which is $\neg b$. The program's next choice would necessarily be among clauses that were satisfied by M, in this case $[b, \neg f]$. But now the goal $\neg f$ fails. $\square$

Theorem 5.2: Let T be a completed PDST for formula S, and let M be an autarky for S. Assume that the root of T is labeled either with $\top$ or a literal q such that $\neg q$ is not in M. If clause $C \in autsat(S, M)$, then C does not occur in any PDT with the same root as T that refutes S. ▮

Example 5.3: The next theorem is also illustrated by Example 4.1 and Figure 6. This time, consider the completed PDST rooted at the goal $\neg f$. Let M be the empty set, which is an autarky. There is no PDT refutation of $\neg f$ with top clause $[b, \neg f]$, so the theorem asserts that M can be extended to some autarky M' that contains $\neg f$. Simply adding $\neg f$ is insufficient, due to $[\neg b, \neg d, f]$. However, adding both $\neg f$ and $\neg b$ makes M' an autarky. □

Theorem 5.3: Let T be a completed PDST for formula S with root $v(q)$, where q is a literal, such that there is no PDT refutation of q with the same top clause as T. Let M be an autarky for S such that neither q nor $\neg q$ is in M. Then there is an autarky $M' \supset M$ such that q (interpreted as a unit clause) is in M'. Moreover, $M' - M \subseteq lits(S)$. ▮

5.1 Propositional Application for Autarky Analysis

Based upon Theorem 5.3, we arrive at the following idea for autarky construction during a refutation search (additional details in [VG95]).

1. When the search for a refutation of a specific goal q begins, an "initial autarky" M_0 (possibly $\emptyset$) is passed in. By the next step, we can assume that neither q nor $\neg q$ is in M_0. This set will be augmented to a "current autarky" M_j whenever a clause extension fails to deliver a refutation.

2. Any clause $C^{(j)}$ $(1 \leq j \leq k)$ that is *eligible* for extension, but is satisfied by the current autarky is bypassed (and the current autarky is unchanged). This is the crucial *autarky pruning* operation.

3. For each eligible clause $C^{(j)}$ that is tried as an extension, the current autarky M_{j-1} is passed down into a recursive search, enabling pruning in the subtree. If the extension fails to lead to a refutation, then the recursively called procedure passes back an "autarky increment" ΔM_j, and a new "current autarky" $M_j = M_{j-1} + \Delta M_j$ is computed. The increment is supplied by a goal child of $C^{(j)}$ that could not be refuted.

4. If all clauses fail to lead to a refutation of q, then the procedure passes back $\sum \Delta M_j + q$. This becomes an autarky increment at the higher level.

For efficiency of implementation, Modoc applies unit subsumption (Definition 2.3) at the time q is added to the autarky increment, and undoes it later if the autarky increment is discarded (because some alternative refutation succeeded).

Example 5.4: Again consider Example 4.1 and Figure 6. With $[b, \neg f]$ as the initial top clause, a refutation procedure (selecting literals depth-first, left-right) would refute goal b at level 1, then search for a refutation of $\neg f$ at level 1. This leads to an extension, then to the failed goal $\neg b$ at level 2. Thus $\neg b$ is passed back to level 1 as the final autarky increment. Back at level 1, there are no more clauses to try, so the autarky increment from level 2 is combined with this goal, and passed up to level 0 as $\{\neg b, \neg f\}$. At the top level we had $M_0 = \emptyset$, so $M_1 = \{\neg b, \neg f\}$. This is an autarky for the entire formula S.

Now we know that S is unsatisfiable if and only if $autrem(S, M_1)$ is. In other words, clauses $[\neg b, \neg d, f]$ and $[c, \neg f]$ do not need to be considered as alternate top clauses for new refutation attempts. Since M_1 is now the "current autarky" at level 0, the procedure sketched above bypasses them. In this example, any new top clause selected from $autrem(S, M_1)$ leads to a successful refutation. In general, the next top clause might also fail, and M_1 would be expanded to a larger autarky M_2, etc. □

6 Lemmas and C-Literals

In the model elimination procedure a "lemma" may be recorded upon the completion of any (sub)refutation [Lov69, FLSY74, Lov78], although this is not necessary for completeness. 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], as well as trees [LMG94], but we are aware of no empirical studies of lemma strategies on propositional problems. This section sketches how lemmas are incorporated into the implementation of the Modoc algorithm. We introduce and describe a strategy for "quasi-persistent" lemmas and "eager" lemmas as well as two forms of "cuts" that permit refutations to be accelerated.

In Modoc, autarky pruning is compatible with the use of lemmas, and largely orthogonal. Clauses that are pruned by an autarky cannot participate in a successful refutation (at the point where they are pruned), whether or not lemmas are used to shorten the refutation.

6.1 Background on Lemmas

Suppose the refutation of a literal q is completed in a PDT. Let B be the subset of ancestors of q that were actually used for reductions in q's refutation (say $B = \{p_1, \ldots, p_m\}$, where m may be 0). Then a lemma clause, $[\neg q, \neg p_1, \ldots, \neg p_m]$, can be derived soundly [LMG94].

Example 6.1: Recall the refutation search described in Example 5.4, based on Figure 6. To refute e, the goal is extended with the clause $[\neg c, \neg e]$, which has no subgoals, due to mandatory reduction with ancestor c. Therefore, the lemma $[\neg e, \neg c]$ follows. However, this is already a clause in the formula.

But this also completes the refutation of c. No proper ancestors of c were used for reductions, so the lemma $[\neg c]$ follows.

Similarly, the refutation of goal $\neg d$ is now complete. This refutation used goal c for reduction, but c is beneath $\neg d$ in the tree, so is *not* part of the lemma. The lemma is simply $[d]$. $\square$

In the lemma $[\neg q, \neg p_1, \ldots, \neg p_m]$, literal $\neg q$ is called a "C-literal" and is attached to the *lowest* ancestor (say p_c) among $\{p_1, \ldots, p_m\}$. (If $m = 0$, attach it to the root of the PDT, which is normally $\top$.) 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".

If the PDT is abandoned (because some other part of the refutation fails) 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 dependencies B. 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 section.

6.2 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 fails is abandoned. (In the first-order case, substitutions need to be backed out, as well.) The lemma can 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 subsection).

Modoc maintains lemmas attached at p_c until the tree rooted there is abandoned or its refutation is completed. The previously described 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. 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 is needed. In summary, the structural differences between the PDST and the tableau search tree necessitate differences in lemma handling.

Our strategy is incompatible with the heuristic called "strong regularity", introduced by Letz *et al.* That is, 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, and a counter-example can be constructed if the two heuristics are combined.

Example 6.2: This example continues the refutation search begun in Example 6.1, based on Figure 6. While refuting b the procedure would be able to attach C-literals $\neg c$, d, $\neg f$, and $\neg b$ at the root. When the refutation fails in the right branch, the traditional C-literal technique forgets all of them. Our quasi-persistent method does not, because they are still sound as C-literals. When the refutation search tries a different top clause, the C-literals $\neg c$ and d are available and might shorten the search. (In fact, $[c, \neg d]$ now succeeds immediately.) This can also happen without backtracking to the top level. $\square$

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 empirical testing.

6.3 Lemma-Induced Cuts

We now describe the method by which Modoc exploits complementary C-literals. Suppose, as described above, the C-literal $\neg q$ is attached at p_c and 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. Now complementary C-literals have been derived on one branch. Let the full form of the second lemma be $[q, \neg r_1, \ldots, \neg r_n]$; that is, $\{r_1, \ldots, r_n\}$ is exactly the set of ancestors used in the refutation of $\neg q$. Again, let r_c be the lowest ancestor among the r_i, 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. Let A' be the ancestors of r_c. 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. 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.*

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 $(S + A') \vdash q$ and $(S + A') \vdash \neg q$. Now the cut rule infers $(S + A') \vdash \emptyset$.

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 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".

6.4 C-Reduction-Induced Cuts

As mentioned earlier (Section 4), when Modoc selects an ancestor, it strengthens the formula with this ancestor. Strengthening is the combination of unit subsumption and unit implication (Definition 2.3). Similarly, when Modoc installs a C-literal, it applies unit implication 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 0 length due to unit implication by a new C-literal r_c. 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. Let q_1 be the ancestor at the depth of r_1. (If r_1 is an ancestor, $q_1 = r_1 \neq r_c$.) If $q_1 = r_1$, then extension of q_1 by C refutes it immediately. If $q_1 \neq r_1$, we extend q_1 by the "virtual clause" $[\neg q_1, \neg r_1, r_1]$, again using a form of "cut". Subgoal $\neg r_1$ is C-reduced and subgoal r_1 is immediately refuted by extension with C.

The effect of this cut is similar to the lemma-induced cuts (Section 6.3) in that (possibly) an intermediate goal node can be refuted without completing the refutation in progress for it.

6.5 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 it to derive *eager lemmas*.

When Modoc selects a new goal node e, besides performing unit implication with it (Definition 2.3), it performs unit-clause propagation with any clauses whose (strengthened) length reduces to 1. All such clauses are asserted as *E-literals* attached to the goal node e. Like C-literals, E-literals are logical consequences of $ancs(e)$, and can be used in the same manner in the refutation search below e.

An interesting situation arises when the above unit-clause propagation directly derives the empty clause. Clearly, this implies that a refutation of e exists. 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 illustrated in the lower part of Figure 4, and discussed in Example 1.2.

Let us call this the *eager dependency DAG*. To simplify the remainder of this discussion, let us call e an E-literal, too. In this DAG, the edge $p \rightarrow q$ denotes that q is an E-literal attached to the selected goal e, and the clause C_p that derived p by unit propagation contains $\neg q$. Further, let us call C_p the *eager parent* of p, and let us call the complements of the literals in $C_p - p$, *other than* E-literals, the *nonlocal dependencies* of p. Finally, let C_0 be the clause that became empty. Then C_0 is the source node of the DAG; edge $C_0 \rightarrow q$ denotes that q is an E-literal attached to the selected goal e, and C_0 contains $\neg q$; the complements of the literals of C_0, *other than* E-literals, are the *nonlocal dependencies* of C_0.

Theorem 6.1: With the foregoing notation, if p is an articulation point in the eager dependency DAG, then there is a refutation of e in which $\neg p$ becomes 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. ▮

Corollary 6.2: 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. ▮

If p is not an articulation point, then the refutation implied by the eager dependency DAG uses some reduction with a node later in the DAG, and the C-literal $\neg p$ would be attached there, and would be discarded by the time e was refuted.

Pigeon-Hole Formulas

Pigeons	Vars	Clauses	Modoc CPU Secs.	Model Elim. CPU Secs.
8	56	204	1.26	1.22
9	72	297	11.43	10.85
10	90	415	114.84	107.44
11	110	561	$\geq$1200.00	$\geq$1200.00
12	132	738	$\geq$1200.00	$\geq$1200.00

Pigeon-Hole Formulas less first binary clause

Pigeons	Vars	Clauses	Modoc CPU Secs.	Model Elim. CPU Secs.
8	56	203	0.16	1.09
9	72	296	1.28	9.58
10	90	415	11.54	95.98
11	110	561	180.55	$\geq$1200.00
12	132	738	$\geq$1200.00	$\geq$1200.00

Figure 10: Comparative performances on pigeon-hole formulas (unsat), and pigeon-hole formulas with one binary clause removed (sat). For M.E., one refutation was attempted with the first binary clause of the formula as top clause. Times are for a Sun Sparcstation 10/41.

7 Experimental Results

An efficient implementation of Modoc was programmed in C. Hereinafter, we refer to this C implementation of Modoc as `modoc`. Important data structure issues will be the subject of a future report. This section reports on performance tests. The CPU time for a Sun Sparcstation 10/41 is reported.

The results contain substantial evidence that autarky pruning overcomes the major inefficiency of model elimination. Modoc is not yet competitive with the leading model-search methods on random formulas, but outperforms them on formulas generated from an *automated test pattern generation* (ATPG) program [Lar92]. (Hereinafter, we call these formulas *circuit formulas*. These formulas are satisfiable if and only if the outputs of the fault-free and faulty circuits differ for some common input.)

7.1 Effect of Autarky Pruning

This section compares `modoc` to model elimination without autarky pruning[2]. Both programs used all lemma and cut options described in earlier sections (these options improve performance). For model elimination, only one refutation was attempted to keep running times within reason, even though one failed attempt yields no conclusion about the formula.

Recall that Figure 1 showed that model elimination suffered a rapid performance degradation on small satisfiable random formulas even with all lemma and cut options in effect. Figure 13 shows that autarky pruning overcomes this problem.

Figure 10 shows results on pigeon-hole formulas and modified pigeon-hole formulas. Recall that the k-

[2]Program was obtained by disabling autarky pruning from `modoc`.

formula	num.	none		quasi-persistent		quasi-persistent, cuts		quasi-persistent cuts, eager	
class	fmlas	avg	max	avg	max	avg	max	avg	max
`rand100`	200	$\geq$1200	$\geq$1200	141.32	484.75	19.01	73.07	1.54	4.74
`rand141`	200	$\geq$1200	$\geq$1200	$\geq$968.04	$\geq$1200	481.23	1686.62	25.39	89.95
`bf2670`	53	$\geq$1175	$\geq$1200	$\geq$831.52	$\geq$1200	395.50	6124.36	7.15	181.49
`ssa2670`	12	$\geq$1200	$\geq$1200	$\geq$1200	$\geq$1200	239.75	867.76	30.42	118.78

Figure 11: Improvement in search time (CPU time) using various lemma strategies and cuts. `rand100` is 200 random 3CNF formulas of 100 variables and 427 clauses; `rand141` is the same except with 141 variables and 602 clauses. `bf2670` and `ssa2670` are circuit formulas generated from ATPG.

formula	quasi-persistent		quasi-persistent, cuts		quasi-persistent cuts, eager	
class	t	p	t	p	t	p
`rand100`	-	-	17.6660	0.0000	18.2451	0.0000
`rand141`	-	-	-	-	16.6882	0.0000
`bf2670`	-	-	-	-	2.6673	0.0102
`ssa2670`	-	-	-	-	2.7124	0.0202

Figure 12: Student's t-test to test the significance of improvement due to lemma strategies and cuts. Comparison is to the left column. (For "quasi-persistent", it is relative to no lemma strategies and cuts.) See Figure 11 for brief explanation of formula classes. '-' means that not all data were available to make the comparison.

pigeon problem states the constraints that k pigeons fit into $k-1$ holes, and no hole contains two distinct pigeons. On pigeon-hole formulas `modoc` and model elimination perform essentially the same, because the refutation is found without any backtracking from failed attempts.

Satisfiable versions of pigeon-hole formulas were created by removing the first binary clause. *Modoc* has an easier time with the satisfiable modifications, while model elimination has greater difficulty.

7.2 Effectiveness of Lemmas and Cuts

The previous section showed the importance of autarky pruning. This section examines the relative effectiveness of the lemma and cut options described in Section 6. Experiments were made on random formulas and on two sets of circuit formulas. Figure 11 shows the average and maximum search time (CPU time) for each formula class using various lemma strategies and cuts. Figure 12 shows the significance of improvements for random formulas and circuit formulas using Student's t-test [WEC91]. The use of Student's t-test was advocated in [VGT96, Appendix A]. Recall that a larger t indicates greater difference, while smaller p indicates that the observed improvement is less likely to be by chance.

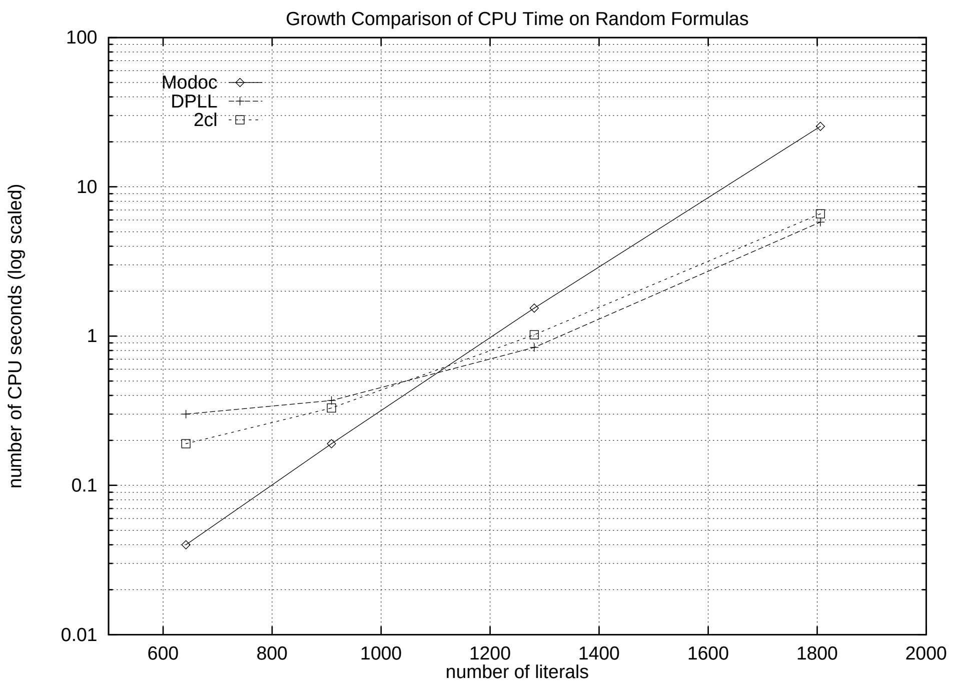

Figure 13: Growth comparison on Random Formulas.

7.3 Comparison with other Satisfiability Testers

In order to assess its competitiveness (or lack of), `modoc` was run against two satisfiability testers—DPLL and `2cl`—both of which are model-searching programs. The DPLL program, due to Davis, Putnam, Logemann and Loveland [DP60, DLL62] is efficiently implemented using the unit propagation method of Dalal and Etherington [DE92]. The `2cl` program is one of the best performing all around satisfiability programs reported in the literature [VGT96].

Results on random 3CNF formulas are summarized in Figure 13. These formulas were generated according to the constant-clause-width model: every clause contains 3 different variables and any combination of literal polarities is equally likely.

The difference in `modoc` performance between satisfiable and unsatisfiable formulas grows as the number of variables (and hence the number of literals) is increased. The ratio (avg. unsatisfiable time/avg. satisfiable time) at the level of 50 variables, was 2.64; it was 3.16 at 141 variables. However, the growth plot shows that `modoc` still underperforms model-searching methods on random formulas.

Results on some circuit formulas are summarized in Figure 14. The same formulas reported in [VGT96] were used. In all but one circuit family, `modoc` ran faster than `2cl` on the average. Indeed, on the harder families it ran 30–40 times faster.

As witnessed in Section 7.1, the performance of Modoc strongly depends on autarky pruning. This implies that if the formula is not "partitionable", Modoc would not perform as intended. Generally speaking, in a random formula, it is expected that the variables are distributed evenly and thus the formula may be harder to partition. Whereas in the case of more structured formulas, such as circuit formulas, many variables occur locally, only in subcircuits, and thus may be amenable to partitioning.

Circuit Family	Ave. # of Variables	Ave. # of Literals	Ave. CPU Time (sec) `modoc`	`2cl`
`ssa0432`	433	2347	0.17	0.77
`ssa2670`	1320	7414	29.16	1539.60
`ssa6288`	10406	87355	0.31	39.45
`ssa7552`	1495	7945	0.10	1.93
`bf0432`	886	7703	25.02	5.85
`bf1355`	2266	18192	3.37	41.81
`bf2670`	1300	7904	7.91	439.01

Figure 14: CPU time comparison of Modoc and `2cl` on circuit formulas.

8 Conclusions and Future Work

We have introduced a method to incorporate autarky analysis into propositional resolution procedures. Experimental results indicate that substantial gains of efficiency can be achieved. We have introduced the eager lemma strategy and two forms of controlled cut, all of which further improve performance measurably.

Future work should proceed along several directions, including heuristics for guiding the resolution search, further improvements to lemma caching, and an extension to first-order theorem proving. Another open question is the worst-case complexity of Modoc or an improved version of Modoc.

Acknowledgments

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.

References

[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.

[BJL86] C. E. Blair, R. G. Jeroslow, and J. K. Lowe. Some results and experiments in programming techniques for propositional logic. *Comput. & Operations Research*, 13(5):633–645, 1986.

[CA93] J. Crawford and L. Auton. Experimental results on the cross-over point in satisfiability problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence; AAAI-93 and IAAI-93 (Washington, DC, USA, 11-15 July 1993)*, pages 21–7. Menlo Park, CA, USA: AAAI Press, 1993.

[Caf93] R. Caferra. A tableaux method for systematic simultaneous search for refutations and models using equational problems. *Journal of Logic and Computation*, 3(1):3–25, February 1993.

[DABC95] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. SAT versus UNSAT. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1995.

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

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

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

[Elk89] C. Elkan. Conspiracy numbers and caching for searching and/or trees and theorem-proving. In *Eleventh Int'l Joint Conf. on Artificial Intelligence*, pages 20–25, Palo Alto, CA, 1989. Morgan Kaufmann.

[FL93] C. Fermuller and A. Leitsch. Model building by resolution. In E. Borger, G. Jager, H. Kleine Buning, S. Martini, et al., editors, *Computer Science Logic. 6th Workshop, CSL '92. (San Miniato, Italy, 28 Sept.-2 Oct. 1992)*, pages 134–48. Berlin, Germany: Springer-Verlag, 1993.

[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.

[GW93] I. P. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for SAT. In *Proceedings of the Eleventh National Conference on Artificial Intelligence; AAAI-93 and IAAI-93 (Washington, DC, USA, 11-15 July 1993)*, pages 28–33. Menlo Park, CA, USA: AAAI Press, 1993.

[HF90] J. N. Hooker and C. Fedjki. Branch-and-cut solution of inference problems in propositional logic. *Annals of Mathematics and Artificial Intelligence*, 1:123–139, 1990.

[HHT94] F. Harche, J.N. Hooker, and G.L. Thompson. A computational study of satisfiability algorithms for propositional logic. *ORSA Journal on Computing*, 6(4):423–35, Fall 1994.

[JSD95] B. Jaumard, M. Stan, and J. Desrosiers. Tabu search and a quadratic relaxation for the satisfiability problem. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1995.

[JW90] R. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[KK71] R. Kowalski and D. Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2(2/3):227–260, Winter 1971.

[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.

[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.

[Lov72] D. W. Loveland. A unifying view of some linear Herbrand procedures. *Journal of the Association for Computing Machinery*, 19(2):366–384, April 1972.

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

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

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

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

[SFS95] J. Slaney, M. Fujita, and M. Stickel. Automated reasoning and exhaustive search: Quasigroup existence problems. *Computers and Mathematics with Applications*, 29(2):115–32, 1995.

[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.

[SKC95] B. Selman, H. A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1995.

[Sla94] J. Slaney. The crisis in finite mathematics: Automated reasoning as cause and cure. In A. Bundy, editor, *Automated Deduction - CADE-12. Proceedings of 12th International Conference on Automated Deduction (Nancy, France, June/July 1994)*, pages 1–13. Springer-Verlag, 1994.

[SLM92] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 440–446, San Jose, CA., July 1992.

[SS95] C. B. Suttner and G. Sutcliffe. TPTP v1.2.0. Technical Report AR–95–03, Institut fur Informatik, TU Munchen, Germany, November 1995.

[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.

[VG95] A. Van Gelder. Simultaneous construction of refutations and models for propositional formulas. Technical Report UCSC–CRL–95–61, UC Santa Cruz, Santa Cruz, CA., 1995. (submitted for publication).

[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.

[WEC91] J. Welkowitz, R. B. Ewen, and J. Cohen. *Introductory Statistics for the Behavioral Sciences.* Harcourt Brace Jovanovich College, fourth edition, 1991.

[ZS94] H. Zhang and M. Stickel. Implementing the Davis-Putnam algorithm by tries. Technical report, Dept. of Computer Science, The University of Iowa, 1994.