Autarky Pruning in Propositional Model Elimination Reduces Failure Redundancy

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

August 19, 1998

Abstract

Goal-sensitive resolution methods, such as Model Elimination, have been observed to have a higher degree of search redundancy than model-search methods, Therefore, resolution methods have not been seen in high performance propositional satisfiability testers. A method to reduce search redundancy in goal-sensitive resolution methods is introduced. The idea at the heart of the method is to attempt to construct a refutation and a model simultaneously and incrementally, based on sub-search outcomes. The method exploits the concept of "autarky", which can be informally described as a "self-sufficient" model for some clauses, but which does not affect the remaining clauses of the formula. Incorporating this method into Model Elimination leads to an algorithm called *Modoc*. Modoc is shown, both analytically and experimentally, to be faster than Model Elimination by an exponential factor. Modoc, unlike Model Elimination, is able to find 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 does not prune any successful refutation; it only prunes attempts that ultimately will be unsuccessful; consequently, it will not force the underlying Modoc search to find an unnecessarily long refutation. To prove correctness and other properties, a game characterization of refutation search is introduced, which demonstrates some symmetries in the search for a refutation and the search for a model. Experimental data is presented on a variety of formula classes, comparing Modoc with Model Elimination and model-search algorithms. On random formulas, model-search methods are faster than Modoc, whereas Modoc is faster on structured formulas, including those derived from a circuit-testing application. Considerations for first-order refutation methods are discussed briefly.

Key Words

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

To appear in: Journal of Automated Reasoning

Contents

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. Unless otherwise qualified, the term "satisfiability" refers to satisfiability of a propositional CNF formula.

Several high performance satisfiability testers have been reported in recent years [Lar92, CA93, SFS95, Zha97, *inter alia*]. See also papers and bibliographies from the DIMACS Second Implementation Challenge, 1993 [DABC95, JSD95, Pre95, VGT96]. Interestingly, they have been based exclusively upon the model-search paradigm (see Section 1.1 for terminology). This contrasts with the situation of first-order theorem provers, which are primarily refutation-based, meaning that they search for *proof* that the (first order) formula is unsatisfiable. The reasons are discussed in Section 1.3. Thus, *Modoc*, described in this paper, seems to be the first refutation-based satisfiability tester that achieves any sort of high performance.

As a practical consideration, a program may be required to produce "evidence", or a "certificate", to back up its decision. 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. For example, a model (truth assignment) can be used easily to verify satisfiability, and a resolution proof can be checked very mechanically to verify unsatisfiability. If critical decisions will be based on the program's output, such a certificate is obviously valuable. To our knowledge, no previously existing implementations can produce *useful certificates* for both "yes" and "no" decisions. The original motivation for Modoc was to be able to fill this gap.

The motivation for studying Model Elimination is its goal-sensitive property. It conducts the search for a refutation proof from a designated *top clause*, and each subsequent operation is related to the immediately previous derivation. In many applications, the client knows a key top clause, in the sense that, *if* the formula is unsatisfiable, then this clause must be part of the reason. In fact, the bulk of clauses often represent a large body of background axioms, known to be consistent, most of which are irrevelant to the reason the key top clause causes inconsistency.

Thus a goal-sensitive search has the potential to be very focussed, and may well find a proof significantly shorter than the original formula. Model-search methods, which have enjoyed recent experimental success on *random* formulas, have no (known) way to achieve this kind of focus on an *application-based* formula.

Despite this apparent advantage, prior experience with propositional resolution has been negative (and consequently, largely unreported). Recent research is establishing a pattern of *tableau-based* methods having highly redundant search spaces (see Section 1.3). Model Elimination is equivalent to a certain tableau procedure [LMG94]. This paper introduces methods that can achieve reductions in the redundancy of Model Elimination by exponential factors. The experimental results in Section 9.5, in particular, show that Modoc takes a major stride toward reclaiming the lead for refutation-based methods on application-based formulas.

The main contributions of this paper are the introduction of a method to construct autarkies incrementally during a Model Elimination search, introduction of a method to prune parts of the refutation search, based on autarky analysis, and proofs that both methods are correct. The main idea is presented informally in Example 1.1. Sections 3–7 contain the technical development.

These methods were developed, as mentioned, for the purpose of extracting a model from a failed refutation attempt (the final autarky is a full model). However, it soon became evident that they imparted a tremendous increase in the efficiency of propositional Model Elimination. This increase in efficiency is demonstrated both experimentally (Section 9) and through a worst-case time analysis (Section 8).

1.1 Existing Propositional Methods

Two fundamental methods are known 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 *goal-sensitive* versions of these methods. Hyper-resolution is a typical non-goal-sensitive method, as is the Davis-Putnam resolution method [DP60, DR94].

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, Loveland and Logemann (see Definition 2.6) is the basis for many modern refinements. A different approach is to treat the problem in terms of integer linear programming [BJL86], with various heuristics [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.

1.2 Two-Sided Search

Currently implemented propositional methods are partially "one-sided" in the sense that, even though they produce both "yes" and "no" answers, they can produce "evidence", or a "certificate", for one side of the decision problem, but not the other. Refutation methods do not produce a model on satisfiable formulas, and model-search methods do not produce a refutation on unsatisfiable formulas.[1] This paper introduces an integrated 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 first performs hyper-resolution to saturation, then if the empty clause has not been derived, there must be a model [FL93]. Their main focus is theoretical, to show that certain classes of first order formulas are decidable, and they do not investigate a specific method to extract a model. This strategy is not practical in the propositional case. The method of Caferra adds equations soundly to the original formula and shows that for some classes the refutation search is guaranteed to terminate [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, and the method has no apparent extension to non-Horn formulas.

1.3 Redundancy in Tableau Procedures

As long ago as 1984, David Plaisted had observed informally that DPLL, the best known model-search procedure (see Definition 2.6), was much more efficient than propositional resolution procedures in practice [Pla84]. Recent research has shown that several *tableau-based* methods exhibit a very high degree of redundancy in their search spaces. Giunchiglia and Sebastiani empirically observed such redundancy in tableau procedures for certain modal logics, and reported improvements obtained by using a model-search approach [GS96a, GS96b]. Methods are known by which any propositional Horn formula can be decided within a linear-sized search space. However, Plaisted showed that numerous goal-sensitive resolution procedures, including Model Elimination with C-Literals (M. E.), will explore an exponential-sized search space on certain families of Horn formulas [Pla94]. These procedures can be cast as various forms of tableau procedure, as shown by Letz *et al.* [LMG94]. Section 8.2 adds to the evidence by exhibiting a super-exponential (Definition 8.1) lower bound for M. E. on a class of non-Horn formulas. Section 9.3 confirms this analysis experimentally, and shows that Modoc achieves a linear-sized search. Section 9.4 presents experimental results on some of Plaisted's formulas, which results show a similar exponential separation between M. E. and Modoc.

[1] It is known in principle how to modify DPLL to construct a resolution proof when no model is found, but we have seen no report on an actual implementation. Probable reasons are that serious implementations usually add additional operations that interfere with the proof construction, and that proof construction may involve considerable space and time overhead.

Formula Size		Unsatisfiable				Satisfiable			
Vars	Clauses	No. of Samples	CPU Avg.	Extensions Avg.	Extensions Max	No. of Samples	CPU Avg.	Extensions Avg.	Extensions Max
15	66	7	0.01	200	390	13	4	300,507	1,039,729
16	71	9	0.01	156	302	11	19	1,262,840	4,311,558
17	75	11	0.01	249	404	9	23	1,816,500	7,891,063
18	81	12	0.02	262	380	8	97	7,302,580	18,380,047
19	85	11	0.02	310	411	9	248	18,899,109	65,519,268
20	90	14	0.03	1,403	10,696	6	1079	53,445,800	143,977,728
50	214	11	1.57	41,610	157,537	9	??	??	??
100	427	9†	611.66	32,748,638	268,067,664	11	??	??	??

(†) One formula timed out after one CPU hour.

Table 1: Comparative performances of Model Elimination with C-literals on unsatisfiable and satisfiable random 3CNF formulas. Times are CPU seconds on a Sun Sparcstation 10/41. "??" indicates that none of these formulas were solved within one CPU hour. On each formula one refutation was attempted with the first clause as top clause. Implementation and experimental details are given in later sections. Comparative results for the new Modoc algorithm appear in Fig. 2.

Formula Size		Unsatisfiable				Satisfiable			
Vars	Clauses	No. of Samples	CPU Avg.	Extensions Avg.	Extensions Max	No. of Samples	CPU Avg.	Extensions Avg.	Extensions Max
15	66	7	0.01	170	290	13	0.01	80	195
16	71	9	0.01	144	260	11	0.01	67	213
17	75	11	0.01	218	356	9	0.01	48	105
18	81	12	0.02	243	347	8	0.00	72	165
19	85	11	0.02	288	408	9	0.01	123	250
20	90	14	0.02	376	537	6	0.01	142	211
50	214	11	1.34	25,255	48,712	9	0.81	15,457	30,378
100	427	9	250.06	3,032,669	4,977,574	11	131.81	1,527,762	3,238,295

Table 2: Performance of Modoc, which consists of Model Elimination with autarky pruning added, and with C-literals. Relative to Model Elimination (Fig. 1), on satisfiable formulas performance has improved by a factor of 1,000,000 on 20-variable formulas. On unsatisfiable formulas, performance has also improved by substantial factors. Moreover, all of the formulas were solved within the one hour cut-off.

The problem with existing propositional refutation methods is illustrated dramatically in Figure 1. M. E. (regarded as one of the most efficient refutation strategies) is able to solve *unsatisfiable* random 3CNF formulas with up to 100 variables, but it bogs down on *satisfiable* formulas about at 20 variables. The problem is exacerbated by the fact that M. E. cannot even conclude the formula is satisfiable after failing with one top clause. It must try other clauses as top clauses before the conclusion is established. For modern model-search methods, these formulas (both unsatisfiable and satisfiable) are considered *easy* at 100 variables and *trivial* at 20 variables.

The companion Figure 2 demonstrates the effect of incorporating autarky pruning, the principal innovation being reported in this paper. Performance on satisfiable formulas has been brought back in line with that of unsatisfiable formulas, and has improved by a factor of 1,000,000 on 20-variable formulas, compared

		Unsatisfiable				Satisfiable			
Formula Size		No. of	CPU	Extensions		No. of	CPU	Extensions	
Vars	Clauses	Samples	Avg.	Avg.	Max	Samples	Avg.	Avg.	Max
15	66	7	0.03	1,362	1,907	13	0.01	394	1,460
16	71	9	0.05	1,721	3,118	11	0.01	324	1,338
17	75	11	0.06	2,753	4,251	9	0.01	294	893
18	81	12	0.07	3,340	5,449	8	0.01	383	1,596
19	85	11	0.08	3,773	5,828	9	0.03	1287	3,032
20	90	14	0.16	7,187	11,139	6	0.03	1203	2,546
50	214	11	1598.55	80,584,426	157,748,144	9	490.01	25,039,874	50,145,756
100	427	9	??	??	??	11	??	??	??

Table 3: For completeness, performance is shown for Model Elimination *without* C-literals, but with autarky pruning added. Relative to Fig. 1, performance on satisfiable formulas has improved by a factor of 30,000 on 20-variable formulas, and the 50-variable formulas were solved within the one hour CPU cut-off. However, performance on unsatisfiable formulas has degraded substantially.

to the original procedure. For completeness, Figure 3 shows how M. E. fares with autarky pruning, but without C-literals. See Section 9 for additional experimental data.

Several reasons can be 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. Observe that the search space is actually greater than 2^n, for satisfiable n-variable formulas. We will return to this point in Section 8.2.
2. The method cannot produce models.
3. The most efficient methods, which are goal-sensitive, 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.

1.4 Summary of Results

A game characterization of refutation search is introduced (Section 4.2). This game, called the *PDST game*, demonstrates some symmetries in the search for a refutation and the search for a model. The game tree can also be viewed as a certain and-or tree whose evaluation indicates whether a refutation exists.

Investigation of PDST game strategies leads to properties of the game that involve autarkies (Section 5), which are the main results of the paper. A method for recursively constructing autarkies is developed. It is shown that clauses satisfied by such autarkies can be pruned from the refutation search. The noteworthy result here is that clauses pruned by such autarkies cannot participate in *any* successful refutation. This contrasts with the weaker property of most resolution refinements that *some* successful refutation remains after pruning, but the remaining refutation may be significantly longer than some that were pruned [LMG94]. Section 6 describes an algorithm, called *Modoc*, that essentially combines Weak Model Elimination with autarky construction and autarky-based pruning.

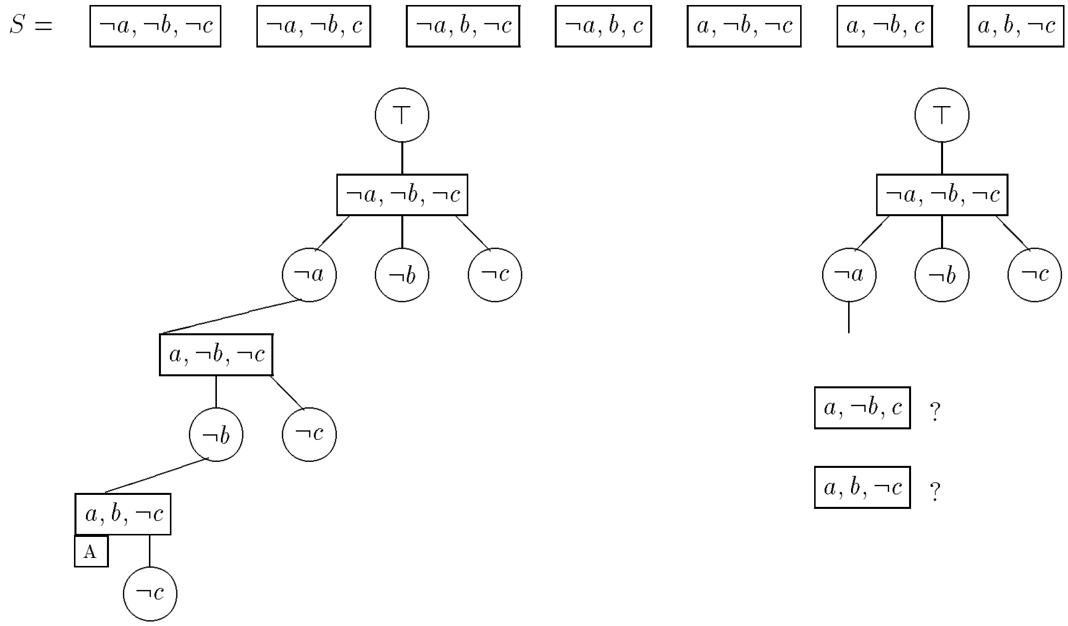

Figure 1: Model Elimination search for Example 1.1. (Left) Search fails at lowest $\neg c$ goal. (Right) After backtracking to alternative choices at $\neg a$ goal.

Model Elimination provides an optional lemma creation mechanism. Section 7 briefly reviews previous work, and discusses the compatibility of autarky pruning and lemma creation. It also introduces and describes a strategy for "quasi-persistent" lemmas.

An algorithm suggested by the above results, called *Modoc*, has been programmed in C. Although it is efficient in terms of data structures, it is virtually devoid of search heuristics. Experimental results are reported in Section 9. They demonstrate the feasibility of building a high-performance resolution-based solver for propositional CNF formulas. The rule seems to be that Modoc's relative performance, compared to model-search methods, is better on structured formulas originating from applications, and not so good on random formulas. While this paper was under review, additional progress on Modoc has been made on several fronts [Oku98, VGO98, VGO97]. With the enhancements reported there, Modoc generally outperforms the well-known incomplete search method, *walksat* [SKC95], on structured formulas that have an identifiable goal, such as those derived from planning applications. However, on random formulas (or structured formulas with no specific goal upon which to focus) Modoc continues to underperform the better model-search methods.

Relationships to first-order theorem proving are mentioned briefly at the end of Section 4.1 and in Section 5.2. Section 10 draws conclusions and indicates future directions.

For readers familiar with Model Elimination adapted to tree structures, we now give a motivating example of how autarky pruning works. Other readers may wish to come back to this example after reading more of the expository material in Section 3, including Example 3.2.

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 1. 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 1, 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.

At this point the search procedure is stuck, because each clause containing literal c also contains an ancestor literal, so it is not eligible for extension at this point in the tree. This rule is variously called "preadmissibility", "tightness", and "regularity".

If we stop and reflect on the meaning of this failure, it can be stated in words, as follows: every clause containing the literal c also contains some *other* literal that is an ancestor (or A-literal) on this path in the tree being constructed. Therefore, 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 1, right). There are two such clauses, as indicated. The standard algorithm continues trying to construct a refutation using one of these clauses, and if that fails, it tries 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". For example, if $\neg a$ is extended with clause $[a, b, \neg c]$, then (possibly among others) there is a goal whose literal is in M, in this case, $\neg c$. This goal must be refuted by extension. The (hypothetical) extension clause contains c, and every clause containing c is known to be satisfied by M, so some *other* literal of that hypothetical clause is in M. In this case, the only other possibility is $\neg b$, because $\neg a$ and $\neg c$ are ancestors at this point. Now $\neg b$ must again be refuted by extension: reduction is impossible because the ancestors on this branch are all in M. But again, the (second hypothetical) extension clause must be satisfied by M, so it must generate another goal that is in M, etc., until 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, the two main themes of the paper:

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.

□

2 Preliminaries

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 2.1: (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*. If F is a formula, then $lits(F)$ denotes the set of all literals composed from propositional variables occurring in F. $\square$

Definition 2.2: (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 formula is *satisfiable* if it is satisfied by some partial assignment; otherwise, it is *unsatisfiable*. A partial assignment that satisfies a formula is called a *model* of that formula. (Thus any model can be extended to a total assignment that is also a model by assigning arbitrary values to the unassigned variables.)

"Consistent" is a synonym for "satisfiable", and "inconsistent" is a synonym for "unsatisfiable". This paper will only apply the terms "consistent" and "inconsistent" to formulas consisting entirely of unit clauses and/or the empty clause. $\square$

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 2.3: (overloading, disjoint union, subset difference) If q denotes a literal, then in a setting where a clause is required $[q]$ may be written simply as q. Similarly, if C denotes a clause, then in a setting where a formula is required $\{C\}$ may be written simply as C. In a setting where a partial assignment is required, $\{q_1, \ldots, q_k\}$ denotes the formula of unit clauses $\{[q_1], \ldots, [q_k]\}$.

Disjoint union is denoted by "+". This partial binary function is defined precisely when its operands are disjoint sets, and produces their union. *Subset difference* is denoted by "−". This partial binary function is defined precisely when its second operand is a subset of its first operand, and produces the set difference. $\square$

As an example of overloading, if F denotes a formula (which does not contain the unit clause $[q]$), then $F + q$ denotes $F \cup \{[q]\}$. The limited definitions of "+" and "−" on sets permits these symbols to be used more algebraically. The following identities are easily established:

Lemma 2.1: When the sets A through D are such that the indicated partial functions are defined,

$$(A - B) + B = A \qquad (C + D) - D = C$$

$\blacksquare$

Definition 2.4: (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

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

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

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

$\square$

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

The next equivalence is a special case of Shannon factorization that has been exploited in many works [DP60, DLL62, AB70].

Lemma 2.2: Formula S is logically equivalent to $(x \wedge S|\{x\}) \vee (\neg x \wedge S|\{\neg x\})$. ▮

Definition 2.5: (formula size) For purposes of induction proofs, the *size of a formula* is usually defined to be the number of occurrences of literals in the formula, and is denoted as $||S||$.

$$||S|| = \sum_{C \in S} |C|$$

where $|C|$ is the usual set cardinality. $\square$

With the notation just introduced we are able to describe the model-search algorithm algorithm called "DPLL".

Definition 2.6: (DPLL) The algorithm of Davis, Putnam, Loveland and Logemann [DP60, DLL62] tests a propositional CNF formula for satisfiability. The published version proceeds as follows:

1. *Triviality test*: If the formula has no clauses, it is satisfiable. If the formula contains an empty clause it is unsatisfiable.
2. *Unit clause rule*: If the formula contains a unit clause $[q]$, then bind literal q to true and simplify the formula with this binding (i.e., strengthen the formula by $\{q\}$).
3. *Pure literal rule*: If the formula contains literal q, but no instance of $\neg q$, then bind literal q to true and strengthen the formula by $\{q\}$.
4. *Splitting rule*: After applying the unit clause rule and the pure literal rule until no further applications are possible, if the formula F_1 is still nontrivial by the triviality test, then *choose a literal from a shortest clause* in F_1. Suppose the chosen literal is p.
   (a) Recursively solve $F_1 + [p]$. If this is satisfiable, return the result.
   (b) If $F_1 + [p]$ was unsatisfiable, then recursively solve $F_1 + [\neg p]$, and return the result.

The splitting rule generates a backtracking search.

The procedure and its variations are often called "Davis-Putnam", or "DP", omitting credit for two of the authors. The "true" Davis-Putnam procedure is resolution-based, and has no backtracking [DP60]. The acronym "DPLL" has been introduced elsewhere to reflect the contributions of all four authors [VGT96].

Many modifications to the DPLL procedure have been studied. However, any theoretical or experimental results stated in this paper for DPLL refer to the published version, summarized above. $\square$

3 Clause and Derivation Trees

Trees are now recognized as the most appropriate data structures for representation of linear resolution derivations. This section describes the tree data structures we shall use, which are chosen especially for propositional resolution.

Two refinements of resolution were proposed independently, called *Model Elimination* [Lov68, Lov69], and *SL-resolution* [KK71], which are closely related [Lov72]. Loveland's original description of Model Elimination [Lov68] can be viewed as a tree structure, although the algorithm description does not explicitly mention a tree. The next paper [Lov69], which is more often cited, introduced a "chain" format, which was also the format for later reports. Within the chain could be found the most recently derived clause, as well as other information about the proof. Minker and Zanon appear to be the first of several researchers to recognize that the tree data structure was more appropriate than a chain [MZ82]. Recently, Letz *et al.* have given a unified view of the methods of tableau calculus and clause trees [LMG94]. These proposals were oriented toward first-order applications. Our trees differ somewhat because propositional resolution does not need to take into account substitutions, which fact permits simplifications in both refutations and searches for refutations. The following technical definitions are illustrated in Example 3.1 and Figure 2.

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 $lits(S)$, 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 $ancs(v)$. 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. $\Box$

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 $lits(S)$ and all clauses are assumed to be in S.

A *propositional derivation tree* (PDT) is a clause-goal tree that can be constructed according to the following inductive definition:

1. The tree consisting only of a goal node, labeled with either a literal or $\top$, is a PDT.
2. A tree with the following structure is a PDT, and is called a *single-clause PDT*.
   (a) The root is a goal node, labeled with q, and has a single child labeled with clause C.
   (b) $C = [\neg q, p_1, \ldots, p_k]$, where k may be zero (in case C is a unit clause), and each literal p_i is different from q.
   (c) Node C has k children, labeled p_i, for $1 \leq i \leq k$. The children of C are leaves.
3. A tree is a PDT, and is called a *top-clause PDT for C*, if it differs from a single-clause PDT (above) in that the root is labeled with $\top$ and there is a leaf corresponding to every literal of C. In this case C is called the *top clause* of the PDT. (Note that this fits the format of a single-clause PDT if we imagine that $\top$ is another variable and C contains an "invisible" literal $\neg\top$.)

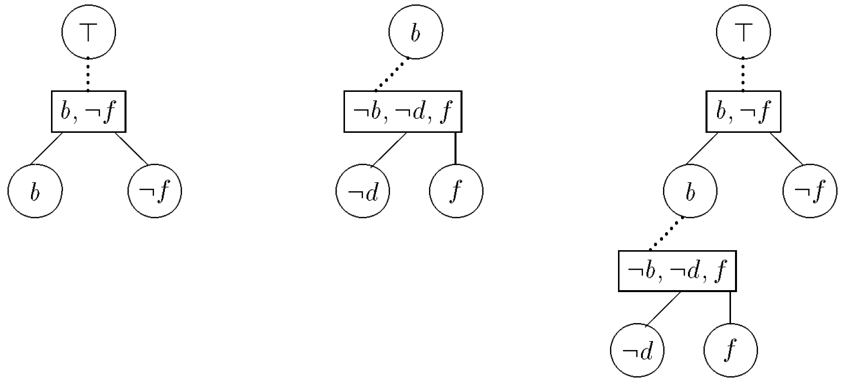

Figure 2: Propositional derivation trees (PDTs) discussed in Example 3.1. Left: *top-clause* PDT for $[b, \neg f]$. Center: *single-clause* PDT with root b and clause child $[\neg b, \neg d, f]$. Right: their combination.

---

4. If T_1 is a PDT with a leaf $v(q)$, and T_2 is a PDT with root q, then a new PDT T_3 may be formed as follows. Let T_2' be obtained from T_2 by pruning all subtrees that are:

   (a) rooted at a goal node that is complementary to a unit clause of $ancs(v)$; or

   (b) rooted at a clause node that contains a literal in $ancs(v)$.

   Then T_3 is obtained by replacing node $v(q)$ in T_1 by T_2'. If T_2 is a single-clause PDT with clause C, then the operation that constructs T_3 is called a *PDT extension of* T_1 *at* $v(q)$ *by* C.

□

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 $ancs(w)$.

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 3.2.) □

Example 3.1: Figure 2 shows PDT examples. The PDT at the right is obtained by PDT extension using the other two. The left side of Figure 1 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.) □

By construction, for any node v of a PDT, $ancs(v)$ (excluding $\top$) is a consistent set of unit clauses, i.e., a partial assignment. Consequently, if V denotes the set of propositional variables, then the PDT has depth at most $2|V| + 1$ (counting both clause and goal nodes, and defining the depth of a tree with one node to be zero).

Also by construction, if $w[C]$ is a clause node of a PDT, then the literals of C are disjoint from those in $ancs(w[C])$, and the variables occurring in any children of C are disjoint from the variables occurring in $ancs(w[C])$. However, C may contain the *complement* of a literal in $ancs(w[C])$.

Let us point out that the terminology "PDT extension" as defined above is a combination of Model Elimination operations called "extension" and "reduction". In some contexts "reduction" has been called "ancestor resolution", "subsumption resolution", or "s-resolution". In the propositional framework, Model

Elimination's "extension" operation need not be considered where a "reduction" is possible, so it is useful to combine "reductions" into the "PDT extension". In a first-order framework, both "extension" and "reduction" normally must be considered, due to differing unifiers, so the goal must be created. A PDT extension can be viewed as a Model Elimination extension followed by as many reductions as possible on the literals of C. To make a PDT tree correspond more closely to a model-elimination tree [MZ82] or a connection tableau [LMG94], one can add a third node type, "reduction", that is automatically attached as the only child of any goal node for which reduction is applicable.

Lemma 3.1: Every PDT can be built from a single-clause tree or a top-clause tree by a series of PDT extensions.

Proof: Note that each PDT extension adds exactly one clause node to the tree. A straightforward induction shows that T_3 in Definition 3.2 can be created by concatenating the series of PDT extensions for T_1 and T_2, eliminating those in T_2 whose clause nodes were pruned from T_2'. Each PDT extension that originated in T_2 (at node v using clause C) and remains in T_2' is "legal" in T_3 because the ancestors of its clause node in T_3 are the union of those in T_2 and $ancs(v)$ in T_1. Therefore, no literal in C occurs in its T_3 ancestors, and the children of this clause node are exactly those literals in C that are not complemented by the T_3 ancestors. ∎

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 3.2 is justified by the following theorem, which essentially states that propositional Weak Model Elimination is sound. Although this is already well known [Lov69, MZ82, LMG94], we outline the proof in the interest of self-containment, and to validate our new formulation, given in Definitions 3.2 and 3.3.

Theorem 3.2:

(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.

Proof: The proof of (A) is by induction on the $||S||$ (Definition 2.5). The base case, $||S|| = 1$, is immediate, as is any case in which the child of the root is the unit clause $\neg q$. Otherwise, let $C = [\neg q, p_1, \ldots, p_k]$, where $k \geq 1$, be the clause child of the root. Form S' by adding the unit clause q to S and removing C. Then $||S'|| < ||S||$, so the inductive hypothesis applies. Denote by T_q the single-clause tree with root $\neg q$ and unit clause $[q]$. By Definition 3.2, the clause node has no children.

For $1 \leq i \leq k$, let T_i be the subtree of T rooted at the goal node p_i that is a child of C. Tree T_i can be transformed into a PDT T_i' for S' by attaching a copy of T_q beneath every clause node whose clause contains $\neg q$. (If D is such a clause node, note that $ancs(D)$ in T_i' is exactly $ancs(D)$ in T with q deleted. Therefore, D in T_i' requires an extra child $\neg q$ to satisfy the PDT constraints.) Since no leaf goal nodes are added in this transformation, T_i' is a refutation of p_i w.r.t. S'. By the inductive hypothesis, S' has no model in which p_i is true.

Clearly, for any i ($1 \leq i \leq k$), if S has a model with q and p_i both true, so does S', and this is impossible due to T_i' above. It is also impossible that S has a model with q true and all p_i false, for $1 \leq i \leq k$, due to clause C. So part (A) is proved. Part (B) follows easily from part (A). ∎

Example 3.2: Figure 3 shows how a PDT might develop under Modoc or M. E. The theorem to be proven is shown at the top of the diagram, where we may interpret:

$$([\neg r, p] \ \& \ [\neg s, p] \ \& \ [\neg p, \neg f, s]\, [\neg p, f, r] \ \& \ [\neg r, \neg s]) \ \stackrel{?}{\rightarrow} \ [\neg f, \neg r]$$

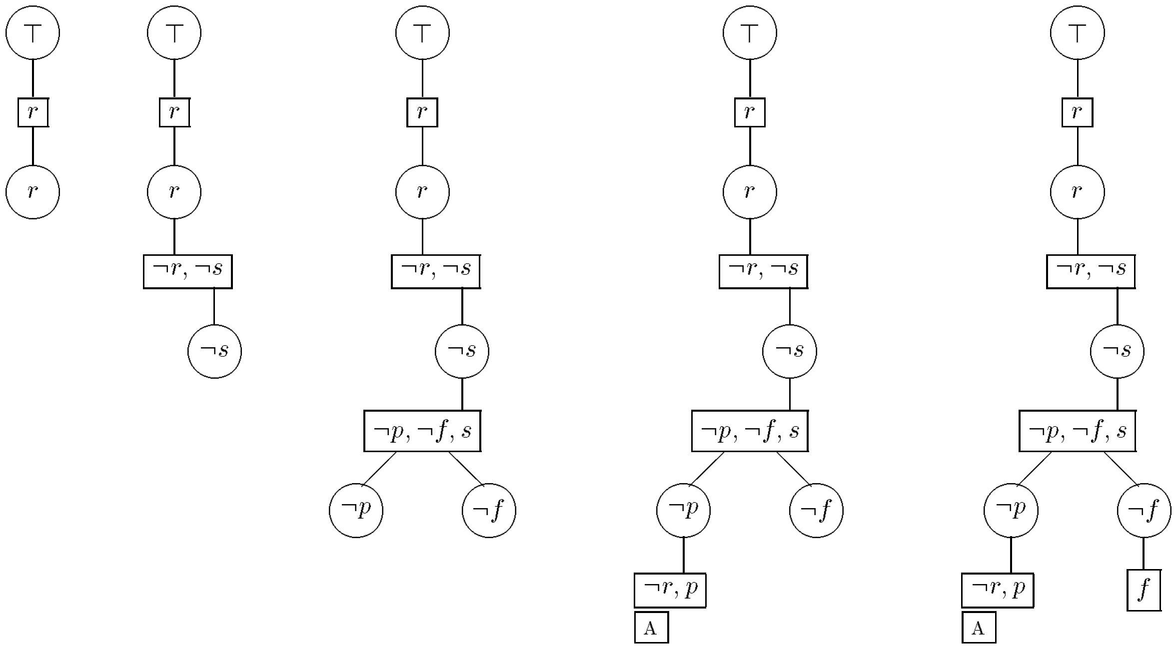

Figure 3: Development of a completed propositional derivation tree (PDT) with top clause $[r]$. See Example 3.2 for discussion.

---

p:	there is precipitation	r:	it is raining
f:	it is freezing	s:	it is snowing

Informally, given the "axioms" in the antecedant, the theorem is "if it is freezing, then it is not raining". To refute the negation, the conclusion becomes a pair of unit clauses, f and r, with the latter being the top clause.

Intuitively, M. E. reasons that *if* there is a model, then it must satisfy $[r]$, so assume r is true. Therefore, *if* there is a model containing r, then it must satisfy $[\neg r, \neg s]$, so assume $\neg s$ is true. Therefore, *if* there is a model containing r and $\neg s$, then it must satisfy $[\neg p, \neg f, s]$, so either $\neg p$ is true or $\neg f$ is true (or both). To prove that there is no model in which r and $\neg s$ are true, it suffices to *eliminate* all possible models in which r, $\neg s$ and $\neg p$ are true, and then *eliminate* all possible models in which r, $\neg s$ and $\neg f$ are true. The left subtree accomplishes the first task and the right subtree accomplishes the second. In the left subtree, clause $[\neg r, p]$ does not generate a subgoal of $\neg r$ because r is an ancestor (i.e., is being assumed). The boxed "A" annotes this fact.

The distinction from model search procedures, such as DPLL, is seen in the above paragraph. Modoc and M. E. first consider the assumption $\neg p$, then consider $\neg f$. DPLL always considers some literal, then the complement of the same literal. □

The above example illustrated the "best case" for building a complete PDT, where each clause that is chosen for extension is a successful choice. In general, many of the clauses that are eligible for extension at a given node will not lead to a refutation. The next section addresses the issues of searching for a way to build a complete PDT.

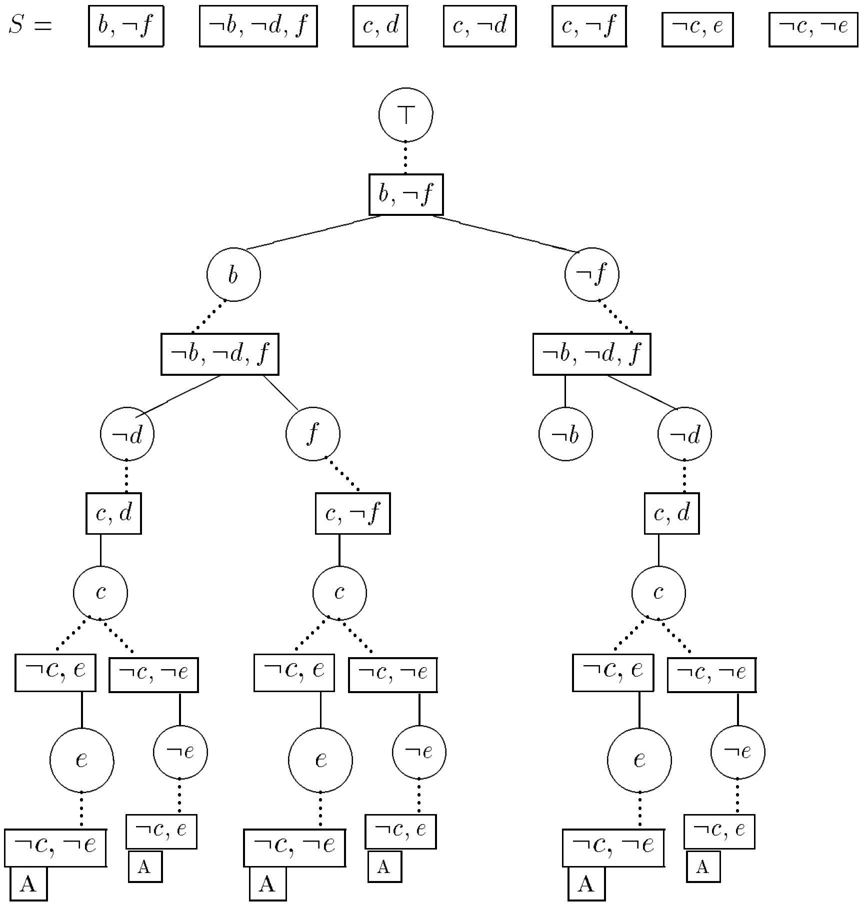

Figure 4: A completed propositional derivation search tree (PDST) with top clause $[b, \neg f]$.

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, even from tableau search trees on propositional formulas, as discussed at the end of Section 4.1, which are again based on exploiting the simplifications available in the propositional logic. We shall show that a propositional derivation search tree can be regarded as a game tree, or an and-or tree, whose evaluation determines the satisfiability of the underlying formula.

Basically, a propositional derivation search tree 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 4, discussed next).

Example 4.1: Consider the clause set S with top clause $[b, \neg f]$, where

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

The PDST appears in Figure 4. 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. □

4.1 Definitions and Basic Properties

This section states the precise definition and basic structural properties of propositional derivation search trees, and illustrates them with examples.

Definition 4.1: (propositional derivation search tree (PDST)) Let S and $\top$ be as in Definition 3.1 and Definition 3.2. Throughout this definition all literals are assumed to be in $lits(S)$ and all clauses are assumed to be in S.

A *propositional derivation search tree* (PDST) is a clause-goal tree that can be constructed according to the following inductive definition:

1. The tree consisting only of a goal node, labeled with either a literal or $\top$, is a PDST.
2. A tree with the following structure is a PDST, and is called a *single-goal PDST for q*.
   (a) The root is a goal node, labeled with q, and has clause children labeled with clauses $C^{(j)}$ for all $C^{(j)}$ that meet the next condition.
   (b) $C^{(j)} = \left[\neg q, p_1^{(j)}, \ldots, p_{k_j}^{(j)}\right]$, where k_j may be zero (in case $C^{(j)}$ is a unit clause), and each literal $p_i^{(j)}$ is different from q.
   (c) Node $C^{(j)}$ has k_j children, labeled $p_i^{(j)}$, for $1 \leq i \leq k_j$. The children of $C^{(j)}$ are leaves.
3. A tree is a PDST, and is called a *single-clause PDST for C*, if it is a top-clause PDT for C (Definition 3.2-3).
4. If T_1 is a PDST with a leaf $v(q)$, and T_2 is a PDST with root q, then a new PDST T_3 may be formed as follows. Let T_2' be obtained from T_2 by pruning all subtrees that are:
   (a) rooted at a goal node that is complementary to a unit clause of $ancs(v)$ (Locations where such pruning occurred are marked with a boxed "A" in diagrams.);
   (b) rooted at a clause node that contains a literal in $ancs(v)$.

   Then T_3 is obtained by replacing node $v(q)$ in T_1 by T_2'. If T_2 is a single-goal PDST with clause C, then the operation that constructs T_3 is called a *PDST extension of T_1 at $v(q)$*.

If the root of a PDST is a literal $q \in lits(S)$, it is called a PDST for S *with top goal q*; if the root is $\top$ and its child is $w[C]$, the tree is called a PDST for S *with top clause node $w[C]$*, or *with top clause C*. □

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 4.2: Figure 5 illustrates Part 4 of Definition 4.1. Two PDSTs on the left are combined to produce the one on the right. In the result, subtrees rooted at *clause nodes* containing $\neg a$ are pruned because $\neg a$ is now an ancestor. In the surviving clause of the lower tree, the subtree rooted at the *goal node* containing a is pruned for the same reason. The boxed "A" is not a structural part of the tree, but is a notation to indicate a literal that is subject to implicit reduction. □

Definition 4.2: (failed goal node, completed 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 any attempt at a PDST extension results in the entire extension tree being pruned, leaving the original tree. □

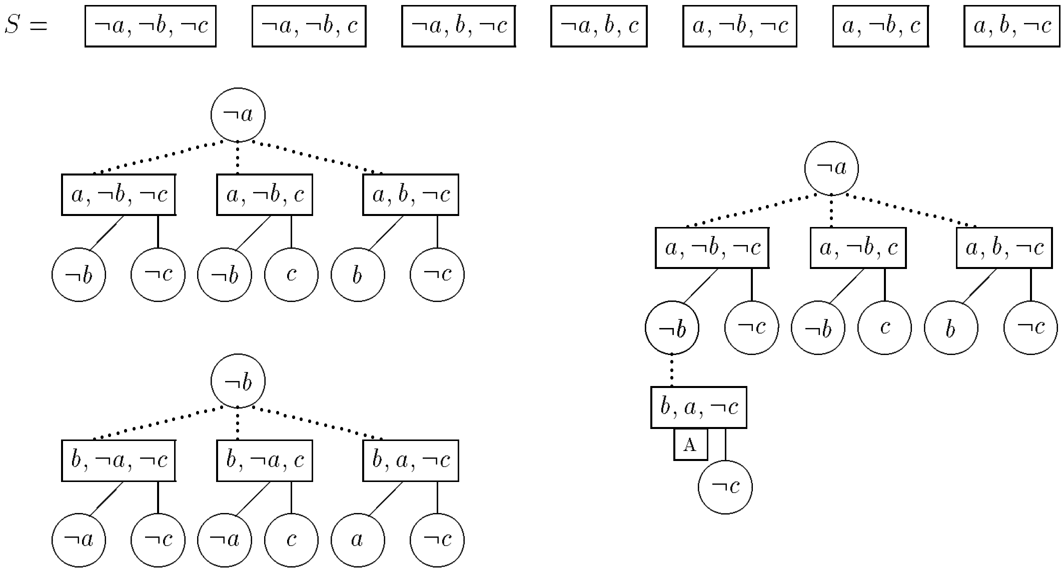

Figure 5: PDST trees, as discussed in Example 4.2. Left: *single-goal* PDSTs for $\neg a$ and $\neg b$. Right: their combination.

Definition 4.3: (universal PDST) For a given formula S 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.

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 $\{q\}$.

Proof: No clause in the tree of w contains q, so all clause labels of T' are well-defined. No goal in the tree of w is labeled by q or $\neg q$, so all the goal nodes of T' have legal labels. Finally, the parent-child structure satisfies the definition of PDST. ▮

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.

Proof: Immediate from the definition. ▮

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.

Proof: By induction on $||S||$ (Definition 2.5), using Lemmas 4.1 and 4.2. ▮

Example 4.3: Consider the PDST of Example 4.2 and Figure 5. The strengthening of S with $\neg a$ yields

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

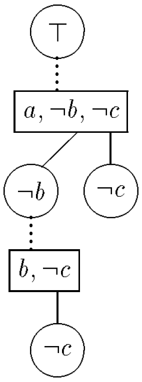

Figure 6: A PDST tree for a strengthened formula, as discussed in Example 4.3.

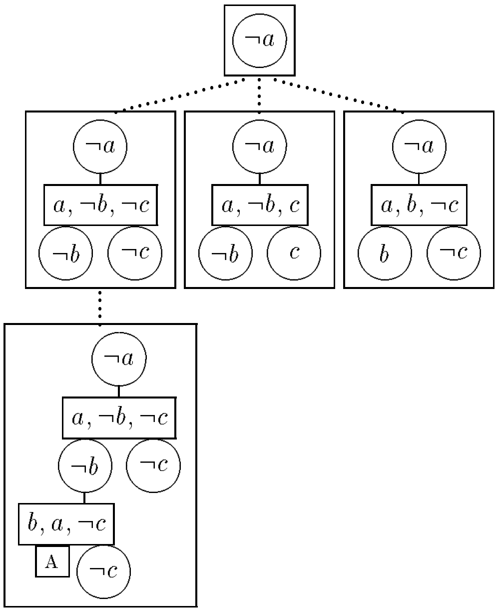

Figure 7: The tableau search tree that corresponds to the PDST tree on the right of Figure 5.

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 Figure 5, with the goal $\neg a$ replaced by $\top$. Clauses containing literal a are replaced by their strengthened forms, producing the tree shown in Figure 6. □

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*, as shown in Figure 7. In this example the five occurrences of the goal $\neg a$ would in general be different due to differing substitutions in the various search nodes.

4.2 The PDST Game

The *PDST game* is a two-player game on a PDST over formula S with top clause C_{top} or top goal q. Although many of the properties of the game hold for arbitrary PDSTs, they are of more interest on completed PDSTs or universal PDSTs. The play begins at the root and follows some branch of the PDST, ending at a leaf node. Alternately, one player, called the *refuter*, chooses a clause node and the other player, called the *spoiler*, chooses a goal node. Each choice must be a child node of the other player's prior choice. To begin play the *refuter* chooses some clause child of the root. The goal of the *refuter* is to end at a clause leaf. The goal of the *spoiler* is to end at a goal leaf. In other words, the player who cannot move loses. In Figures 4–6 the *refuter's* choices correspond to dotted edges and the *spoiler's* choices correspond to solid edges.

Informally, the refuter has a winning strategy for a particular PDST game if it is possible for the refuter to win that PDST game regardless of the choices made by the spoiler. The spoiler has a winning strategy in the dual situation. In Figure 4 the spoiler has a winning strategy by first choosing goal $\neg f$. The refuter must choose $[\neg b, \neg d, f]$, then the spoiler chooses goal $\neg b$. The formal definition follows. Essentially, the minimality criterion states that the refuter strategy is defined only at goal nodes that the spoiler would have an actual opportunity to choose, based on earlier choices in the same refuter strategy.

Definition 4.4: (winning refuter strategy) A *winning refuter strategy* is a partial function $\chi(v)$ from goal nodes to clauses nodes of the PDST upon which the game is being played such that:

1. χ is defined for the root of the PDST; if the root is $\top$, then $\chi(\top) = C_{top}$.
2. If $\chi(v)$ is defined and equals w, then w is a clause child of v, and χ is defined on all children of w (which may be the empty set).
3. $\chi(v)$ is undefined unless it is required to be defined by application of the above rules; that is, $\chi(v)$ is minimally defined.

□

Since it is impossible to define χ on a goal leaf, it is immediate that, if $\chi(v) = w$ and some child of w is a failed goal node, then χ cannot be a winning refuter strategy.

Another view of a PDST is as an and-or tree, in which each clause node is an "or" node and each goal node is an "and" node. In Figures 4–6 the dotted edges descend from and-nodes. The literals labeling goal nodes are immaterial for this evaluation. Goal leaves, being empty conjunctions, evaluate as *true*, while clause leaves evaluate as *false*. Recall that, if the boolean values are ordered as *false* < *true*, then "and" is "minimum" and "or" is "maximum". Therefore, the refuter has a winning strategy if and only if the and-or tree evaluates to *false*.

Some connections between the logical meaning of the PDST tree and the PDST game are given in the following theorems.

Theorem 4.4: Assume the refuter has a winning strategy in the above-described PDST game.

(A) If the root is $\top$, then S is unsatisfiable.

(B) If the root is q, then S has no model in which q is true.

Proof: Let $\chi(v)$ be a winning refuter strategy, according to Definition 4.4, and let W be the set of the nodes upon which $\chi(v)$ is defined. In the PDST, prune away the subtrees rooted at all the unchosen clause nodes, i.e., those not in the image of $\chi(v)$. What remains is a PDT, as every goal node has at most one child. A simple induction shows that every remaining goal node is in W, and therefore has exactly one child. Therefore, according to Definition 3.2, this PDT is a refutation of S in case (A) and is a refutation of q w.r.t. S in case (B). The theorem follows by Theorem 3.2. ∎

Theorem 4.5: Given a PDST for S, if S has a model $\mathcal{M}$ and the root q is in $\mathcal{M}$ or the root is $\top$, then a winning strategy for the spoiler is always to choose a goal node in $\mathcal{M}$.

Proof: At clause node $w[C]$, by construction of a PDST, each literal of C is a child if and only if it is not complemented in $ancs(w[C])$. But by the strategy, every literal in $ancs(w[C])$ is in the model $\mathcal{M}$. Some literal of C is in $\mathcal{M}$. Therefore $w[C]$ has at least one child in $\mathcal{M}$. ∎

Notice that the converse of Theorem 4.4 does not hold, even for completed PDSTs. That is, in case (A) S may be unsatisfiable but the refuter has no winning strategy that begins by choosing C_{top}. Analogously, there is no model-elimination-style refutation with C_{top} as top clause. (Many other refinements of resolution also require one to know a clause or set of clauses in a minimal unsatisfiable set, to achieve completeness.) In case (B) S may have no model in which q is true, yet the refuter has no winning strategy on the completed PDST rooted at q.

Example 4.4: The set of clauses in Figure 4 is actually unsatisfiable, so the spoiler cannot use a model as the basis of a winning strategy. However, for the top clause of the PDST in that figure the two goals, $\neg f$ and $\neg b$, encountered along the path chosen by the spoiler satisfy all of the clauses in which the variables b and f appear, either positively or negatively. By making choices according to this partial assignment, the spoiler never encounters a clause node lacking one of these goals as children. Thus the converse of Theorem 4.4, part (A) fails.

Similarly, the right subtree, rooted at $\neg f$, is a completed PDST for S, and S has no model in which $\neg f$ is true. However, the spoiler wins by choosing goal b. Thus the converse of Theorem 4.4, part (B) fails. □

In analogy with completeness properties of propositional Model Elimination on trees [MZ82, LMG94], it is possible to get approximate converses by strengthening the hypotheses. The theorems are given in the interest of self-containment, since the elegant method of Anderson and Bledsoe [AB70] makes the proofs rather simple.

Theorem 4.6:

(A) If S is a *minimally* unsatisfiable formula and $C_{top} \in S$, then a winning refuter strategy exists on any completed PDST for S with top clause C_{top}.

(B) If S has no model in which q is true and S has some model in which $\neg q$ is true, then a winning refuter strategy exists on any completed PDST for S with top goal q.

(C) If S is unsatisfiable, then a winning refuter strategy exists on the universal PDST for S.

Proof: For part (A) the proof is by induction on $||S||$. The basis, which is the empty clause, is immediate. For $||S|| > 0$, let q be any goal child of C_{top}. We need to show that the refuter has a winning choice at goal q. By Lemma 2.2, $S|q$ is unsatisfiable. Let $\mu(S|q)$ be any *minimally* unsatisfiable subset of $S|q$. The key observation of Anderson and Bledsoe is that $\mu(S|q)$ must contain some clause C' that is a shortened version of $C \in S$. That is, $C' = C - \neg q$. Otherwise, $\mu(S|q)$ would be a proper subset of S, contradicting the hypothesis of minimality. It suffices for the refuter to choose C'. By the inductive hypothesis, a winning refuter strategy exists on the completed PDST for $\mu(S|q)$ with top clause C'. By Lemmas 4.1 and 4.2 this strategy transfers to the PDST for S with top clause C_{top}, establishing part (A).

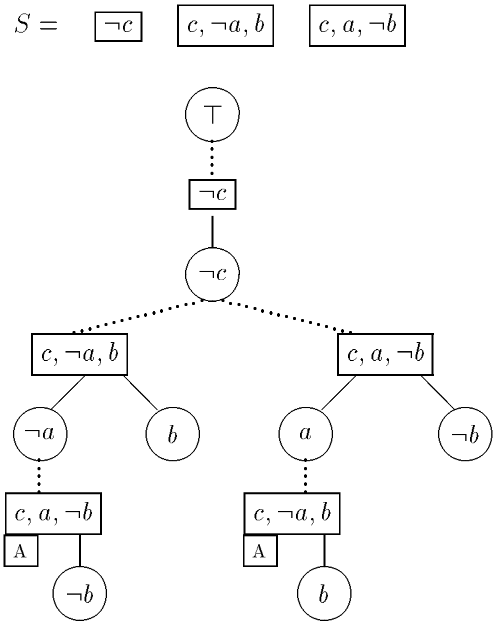

Figure 8: A propositional derivation search tree (PDST) in which the spoiler chooses $\neg a$ in the left clause and chooses a in the right clause. On the left, $\neg b$ is a failed goal node, while on the right b is failed.

Part (B) follows easily from part (A) by adding the unit clause $[q]$ to S, then using it as top clause. Part (C) also follows from part (A), as the refuter can first choose any clause in S in a universal PDST. ∎

The fact that one may have to begin proof searches from many top clauses to find a refutation partly explains why resolution-based methods have not been successful for high performance satisfiability checking. In some of the applications for satisfiability checking, a clause in a minimal unsatisfiable subset is not readily available.

Another problem is seen in the informal converse of Theorem 4.5, which also is not true. That is, when S is satisfiable, there are, in general, winning strategies for the "spoiler" that do not always choose according to some fixed model $\mathcal{M}$. Essentially, Model Elimination search procedures backtrack when they reach a failed node. But the failed nodes reached need not form a consistent set. Figure 8 illustrates this possibility.

The objective of the rest of this paper is to address these problems. We may think of satisfiability testing as evaluating the PDST game to see whether the refuter or spoiler has a winning strategy. The complete PDST conceptually defines the game tree, but it is not materialized. Instead, the procedure tries to explore just enough of the tree to determine the game outcome.

5 Autarkies

This section defines "autarky" and states the main results of the paper. 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 and described in more detail in Section 6.

The concept of "autarky" was (to our knowledge) introduced into logic by Monien and Speckenmeyer, who called it "*autark* truth assignment", employing the German adjective [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 *artarky* 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.

Proof: Part A is immediate from the definition. For parts B and C, observe that $autrem(S, M)$ has no occurrence of any variable that occurs in M. Suppose $autrem(S, M)$ has a model M_{rem}. Then $M + M_{rem}$ is a model of (i.e., satisfies) S. ▮

In the terminology of Monien and Speckenmeyer [MS85], a partial assignment is "*autark* in S" just when it is "an autarky for S" in our terminology. They describe a satisfiability algorithm that is a modification of the basic model searching algorithm of Davis, Logemann and Loveland [DLL62]. (The latter algorithm is often attributed incorrectly to Davis and Putnam [DP60]; we call this algorithm DPLL to acknowledge the contributions of all four authors). The modification consists of testing whether certain subsets of the literals in a shortest clause comprise an autarky before applying the DPLL splitting rule to a variable in that clause. They show that this modification guarantees fewer than 2^n splitting steps on a formula of n variables.

This paper investigates the application of the autarky concept to resolution-based methods. We begin by examining the relationship of autarkies to the PDST game of Section 4.2.

Example 5.1: As in Example 2.1, let

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

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

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

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

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. The actual algorithm appears in Section 6.

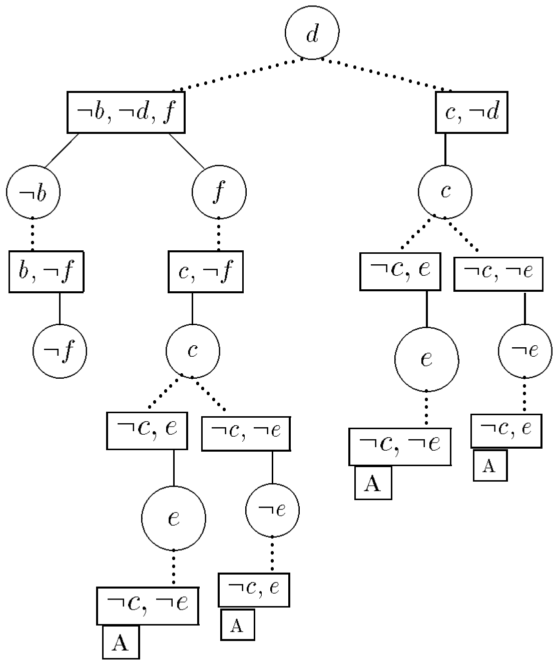

Figure 9: A completed propositional derivation search tree (PDST) with top goal d for the same formula as Fig. 4.

Example 5.2: The next theorem is illustrated by Example 4.1 and Figure 4. Let $M = \{\neg b, \neg f\}$, which is an autarky. We saw in Example 4.4 that the refuter has no winning strategy 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 the refuter cannot succeed by choosing the first clause, because M satisfies it. Indeed, if the refuter did choose the first clause, then the spoiler would choose the goal that occurs in M, which is $\neg b$. The refuter's next choice would necessarily be among clauses that were satisfied by M, in this case $[b, \neg f]$. The spoiler would choose the goal $\neg f$, winning the game. $\square$

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 $C \in \mathit{autsat}(S, M)$, then C is not chosen by any winning refuter strategy χ on T.

Proof: In this proof, the phrase "winning refuter strategy" is abbreviated to "strategy". We give the proof for the case that the root of T is a literal q; the proof for $\top$ is similar. The proof is by induction on h, the goal-height of T.

The base case is goal-height zero (the root is the only goal node). The only possible clause in T is $\neg q$, which is not in $\mathit{autsat}(S, M)$, so the theorem holds vacuously.

For goal-height $h > 0$, assume the theorem holds for trees of goal-height less than h. Let $w[C_0]$ be any child of the root of T, and let $C_0 = [\neg q, p_1, \ldots, p_k]$, where $k \geq 0$. Then w has precisely k goal children, denoted as $v_i(p_i)$.

First, suppose some strategy χ chooses w. For all $1 \leq i \leq k$, partial function χ must be defined for $v_i(p_i)$, so let $w_i[C_i] = \chi(v_i(p_i))$. By the inductive hypothesis on the tree rooted at $v_i(p_i)$, $C_i \notin \mathit{autsat}(S, M)$. But $\neg p_i \in C_i$, so p_i (as well as $\neg p_i$) is not in M. Also, $\neg q$ is not in M. Therefore, C_0 is not in $\mathit{autsat}(S, M)$, which establishes the contrapositive of the theorem for the children of the root.

Now for $1 \le i \le k$, still supposing that strategy χ chooses w, the tree rooted at $v_i(p_i)$ can be transformed into the completed PDST (call it T_i) for $S|\{q\}$ with top goal p_i, as described in Lemmas 4.1 and 4.2. It is immediate that χ defines a strategy on T_i. But M is also an autarky for $S|\{q\}$. By the inductive hypothesis, no clause with a literal in M is chosen by χ. Therefore, χ chooses no clause anywhere in the tree of w.

Finally, let $w[C_0]$ be a child of the root of T such that no strategy chooses w. Then no strategy chooses a clause in the tree of w, by minimality of strategies (Definition 4.4). ∎

Example 5.3: The next theorem is also illustrated by Example 4.1 and Figure 4. This time, consider the completed PDST rooted at the goal $\neg f$. Let M be the empty set, which is an autarky, and does not contain $\neg f$ or f. There is no winning refuter strategy, 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 winning refuter strategy for 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, $M' - M \subseteq \mathit{lits}(S)$.

Proof: In this proof, the phrase "winning refuter strategy" is abbreviated to "strategy". The proof is by induction on the ordered pair (h, m), where h the goal-height of T, and m is the number of children of the root v that are in $\mathit{autrem}(S, M)$. The order is lexicographic on the pairs of integers.

The base case is goal-height $h = 0$ (the root is the only goal node). The hypotheses of the theorem fail for all $m > 0$. When $m = 0$ and no strategy exists, $v(q)$ is a failed goal node, so no clause in S contains $\neg q$. Therefore, $M + q$ is also an autarky for S.

For goal-height $h > 0$, assume the theorem holds for all trees characterized by (h', m'), where $(0, 0) \le (h', m') < (h, m)$ in lexicographic order. Let the children of v that are in $\mathit{autrem}(S, M)$ be $w_j[C^{(j)}]$, for $1 \le j \le m$. If $m = 0$, then every clause of S that contains $\neg q$ also contains a literal in M, so again, $M + q$ is also an autarky for S. If $m > 0$, we will first construct $M_1 \supseteq M$, such that $C^{(1)} \in \mathit{autsat}(S, M_1)$ (and neither q nor $\neg q$ is in M_1, and $M_1 - M \subseteq \mathit{lits}(S)$). Then v has at most $(m - 1)$ children that are in $\mathit{autrem}(S, M_1)$, so the required $M' \supset M_1$ exists by the inductive hypothesis.

To complete the proof, we need to construct M_1. Let the children of w be $v_i(p_i)$, and for each $v_i(p_i)$ consider the completed PDST for $S|\{q\}$ with top goal p_i. Call it T_i. If every T_i had a strategy, these could be combined in T to produce a strategy that chooses w_1, using Lemmas 4.1 and 4.2. Therefore, some T_i has no strategy, and is of goal height $(h - 1)$. Also, M is an autarky for $S|\{q\}$.

1. If $\neg p_i$ is not in M, then the inductive hypothesis implies that an autarky $M_1 \supset M$ exists and contains p_i. M_1 contains neither q nor $\neg q$ because they do not appear in $S|\{q\}$.
2. If $\neg p_i \in M$, then some *other* literal of $C^{(1)}$, say r, must be in M by the definition of an autarky, so let $M_1 = M$ in this case.

In both cases, $C^{(1)} \in \mathit{autsat}(M_1)$, as required. ∎

5.1 Propositional Application for Autarky Analysis

Examining the proof of Theorem 5.3, we arrive at the following idea for autarky construction during a refutation search. Section 6 develops an algorithm in detail.

1. When the search for a refutation of a specific goal q begins, an "initial autarky" M_0 (possibly $\emptyset$) is passed in. Assume that neither q nor $\neg q$ is in M_0.

2. For each clause $C^{(j)}$ $(1 \le j \le k)$ that is *eligible* for extension (i.e., contains $\neg q$ and does not contain any ancestor), if the "current autarky" M_{j-1} satisfies $C^{(j)}$, then $C^{(j)}$ is bypassed, and $M_j = M_{j-1}$.

3. For each clause $C^{(j)}$ $(1 \le j \le k)$ that is tried as an extension, a "current autarky" M_{j-1} is passed down into a recursive search.

   If the extension fails to lead to a refutation, then the recursive procedure passes back up an "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 "final autarky increment" is $\Delta M_{final} = \sum \Delta M_j + q$.

5. If any clause leads to a refutation of q, then the "final autarky increment" is $\emptyset$.

6. Pass back the "final autarky increment".

Of course, the refutation search passes around other information, as well; this outline just mentions that related to autarky computation.

Example 5.4: Again consider Example 4.1 and Figure 4. 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. $\square$

5.2 First-Order Application for Autarky Analysis

Exploiting autarkies in the first-order case is considerably more complicated, and requires considerable further study. This section will present some possibilities by means of an example in first-order logic without equality.

Example 5.5: Let capital letters denote first-order variables, as in Prolog, for this example. Formula $S_{f.o.}$ consists of:

$$\begin{array}{lll}
[\neg p(W,X,Y), \neg s(W,X,Y)] & & \\
[p(W,X,Y), \neg q(W,X,Z), \neg r(X,Y,Z)] & & \\
[p(f(a),b,a), r(a,b,f(a))] & & \\
[q(W,W,Z)] & [q(a,a,Z)] & [q(b,a,f(a))] \\
[s(a,b,f(a))] & &
\end{array}$$

Let the top clause be $[\neg p(W,X,Y), \neg s(W,X,Y)]$. Assume the search tries clauses from top to bottom and selects literals from left to right within the clause. Nodes in the first-order search tree are labeled τ_0, $\tau_1, \ldots$, in prefix order, to facilitate discussion.

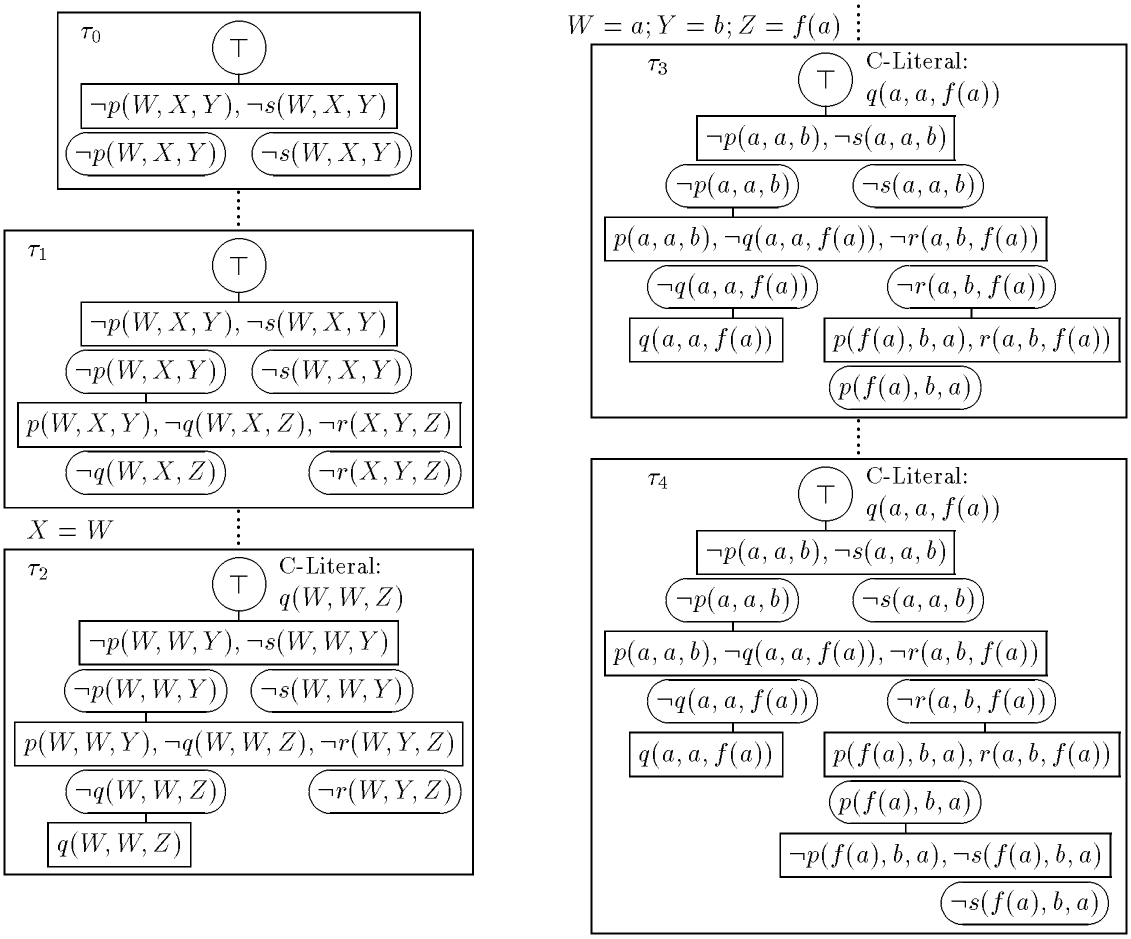

Figure 10: Development of a first-order search tree, leading to failure at node τ_4, as discussed in Example 5.5.

On the left, Figure 10 shows the first-order search tree after two extensions. In node τ_2, $\neg q(W, W, Z)$ has been refuted and the resulting C-literal (see Section 7) has been attached. Literal $\neg r(W, Y, Z)$ is selected next.

On the right of Figure 10 we see the continuation of the search tree after two more extensions from τ_2. Node τ_4 is the first point of failure. Nodes τ_3 and τ_2 have no alternatives, so the search now backtracks to node τ_1. In particular, these goals have failed: $s(f(a), b, a)$, $p(f(a), b, a)$, $\neg r(W, Y, Z)$, $\neg p(W, W, Y)$.

Let us now review briefly the idea of *anti-lemmas* [LMG94]. Subgoal $\neg q(W, W, Z)$ was successfully refuted in search node τ_2, and the corresponding C-literal was introduced. Now τ_2 has been backtracked over, due to a later failure. However, the C-literal is now attached to the goal $\neg q(W, X, Z)$ in search node τ_1 as an *anti-lemma* (see Figure 11). The next clause that resolves with $\neg q(W, X, Z)$ (the selected literal of search node τ_1) is $[q(a, a, Z)]$, leading to node τ_5. But $q(a, a, Z)$ is an instance of the anti-lemma, so this line of derivation must eventually fail also. Therefore, node τ_5 may be abandoned immediately.

The next clause that resolves with $\neg q(W, X, Z)$, the selected literal of search node τ_1, is $[q(b, a, f(a))]$, leading to node τ_6 (see Figure 11). This is *not* an instance of the anti-lemma. However, observe that literal $\neg r(a, Y, f(a))$ in τ_6 is an instance of the literal $\neg r(W, Y, Z)$, which was found not to be refutable in an earlier branch of the search. We claim that a refutation of $\neg r(a, Y, f(a))$ cannot exist. Based on this claim, the

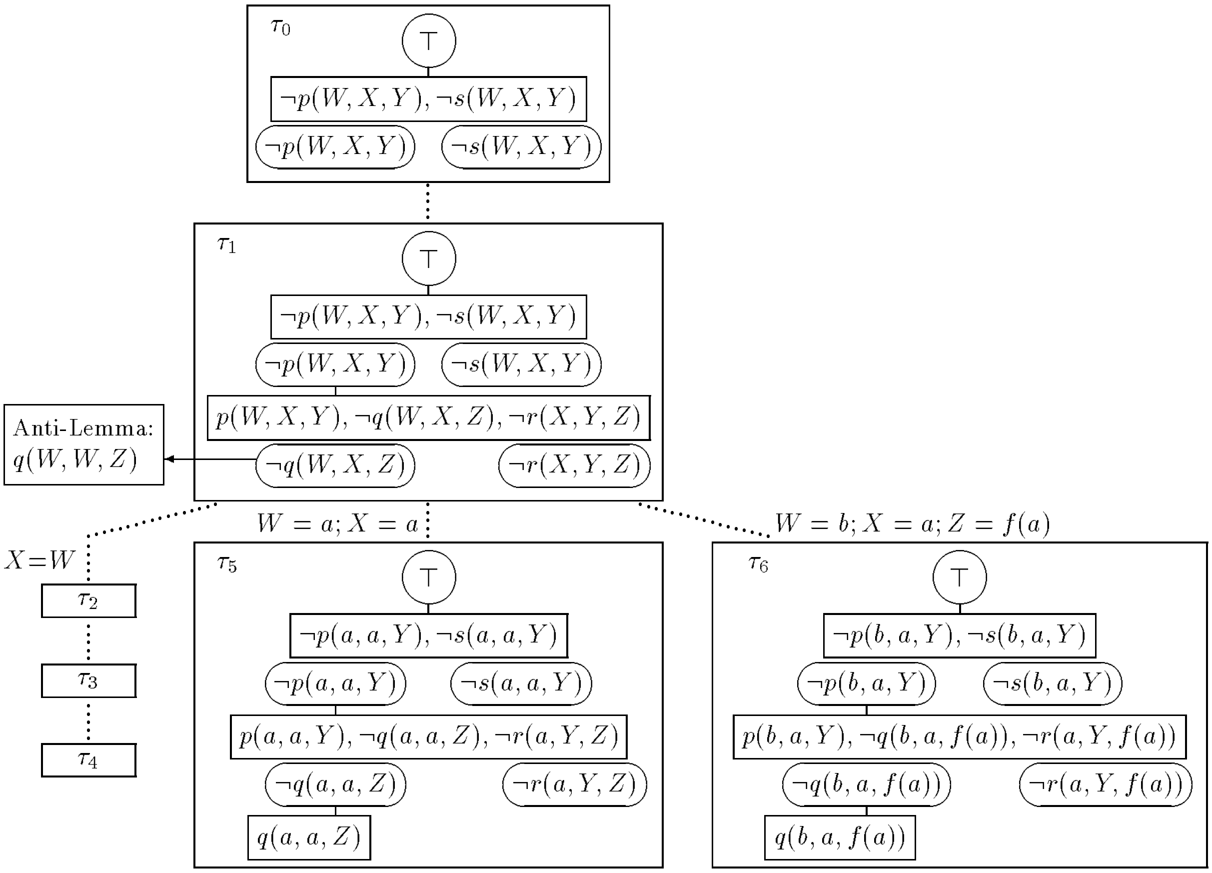

Figure 11: A first-order search tree, developed from Fig. 10, in which the search may be abandoned at nodes τ_5 and τ_6 for reasons discussed in Example 5.5.

---

node τ_6 may be abandoned also.

The reasoning to support the claim runs as follows. Suppose a first-order refutation of $\neg r(a, Y, f(a))$ could be completed in this tree. Then it could be further instantiated, if necessary, into a ground (variable-free) refutation. Consider the same substitutions in the branch of the search that failed (and possibly some extras to force groundedness). Collecting all the ground clauses from the hypothetical successful refutation and the failed branch, we have a finite set of propositional clauses, S, even though the Herbrand universe is infinite. Recall that $lits(S)$ denotes the set of literals that occur in S.

There is an autarky for S that contains all the instances of $\neg r(W, Y, Z)$. Specifically, it is the set M containing all literals in $lits(S)$ that are instances of $s(f(a), b, a)$, $p(f(a), b, a)$, $\neg r(W, Y, Z)$, or $\neg p(W, W, Y)$. It is easy to see that M is consistent (interpreted in the Herbrand universe, so that a, b, and $f(a)$ are distinct). To verify that M must be an autarky, it is sufficient to check for each clause of $S_{f.o.}$ that any instance that has a literal that is complementary to a literal of M also has another literal that is *in* M.

Consequently, no refutation of any instance of $\neg r(a, Y, f(a))$ can exist in this set of propositional clauses, by Theorem 5.2.

We emphasize that the above reasoning applies to this specific example, but we are not prepared to formulate a general statement. □

We believe that this example indicates a possible approach for applying the autarky concept to first-order theorem proving. However, even if the idea can be developed into a general tool, practical experience is needed to discover whether the overhead incurred in autarky analysis pays sufficient dividends in terms of pruning fruitless searches.

6 The Modoc Algorithm

The main procedures of the Modoc algorithm are shown in Figures 12 and 13. The top level is shown in Figure 14. Autarky processing is carried out by the subroutines **inAutRem**, **addAutarky**, and **delAutarky**. Other subroutines implement propositional Model Elimination, or equivalently, a search of the universal PDST (in mode `ALL`) or the PDST with a user-specified top clause (in mode `SINGLE`).

The syntax is an amalgam of C, Pascal and ML, with a bit of overloading thrown in, which should be mostly self-explanatory. Recall Definition 2.3 for "+" and "−" on sets. One notation that might need explanation is

$$(x,\ y,\ z)\ \texttt{:=}\ \texttt{f}(u,\ v);$$

In the spirit of ML, this denotes that function `f` returns a *tuple* of 3 values. These three values are then assigned respectively to x, y and z.

We assume an abstract data type `Context` that maintains implementation-dependent information about the state of the refutation search. The variable `F`, of type `Context`, is a reference, or pointer, to some structure containing this information. *Procedures* with `F` as parameter may change the context state, while *functions* do not. Procedures and functions can be differentiated by the convention that procedures do not return values. The exception is the "constructor function" `setupContext`, which initiates a context with no ancestors, no lemmas, and an empty autarky, and returns a reference (pointer) to it.

Definition 6.1: (context) On a conceptual level a `Context` may be thought of as having four components. If `F` is of type `Context`, then these fields are denoted as follows for this discussion:

clauses	$\texttt{F} \rightarrow S$
ancestors	$\texttt{F} \rightarrow A$
lemmas	$\texttt{F} \rightarrow \mathcal{L}$
autarky	$\texttt{F} \rightarrow M$

Where the meaning is clear, the prefix "$\texttt{F} \rightarrow$" may be omitted. □

The invariant is maintained that the set of "autarky literals" $\texttt{F} \rightarrow M$ is actually an autarky for $S|A$, but not for S itself.

The handling of model-elimination lemmas is not an integral part of the Modoc algorithm. The pseudo-code indicates where lemma-related processing would probably occur, as `isLemma`, `addLemma` and `delLemmas`, but many variations are possible. However, some form of lemma processing is important for the analytical worst case results. Section 7 discusses lemma processing in the basic implementation, which is the basis for the reported experimental results (Section 9), as well as the analytical results (Section 8). Both the analytical and experimental results demonstrate that lemma processing *without* autarky pruning does not achieve acceptable performance.

Ancestors are maintained as a stack. Each ancestor has a set of lemmas associated with it, and a set of "autarky literals" associated with it. Of course, either set might be empty. The association is actually with the *depth* of the ancestor rather than its literal. Procedures that manipulate the state of the context are now described.

pushAncestor, popAncestor: The obvious stack operations on `F` $\rightarrow A$; the `depth` is explicit only for readability.

inAutRem(F, curClause): Returns true if `curClause` has no literal that is in `F` $\rightarrow M$.

delAutarky(F, curM): Removes the literals of `curM` from `F` $\rightarrow M$. The set `curM` should be identical to the subset of `F` $\rightarrow M$ that was added during failed attempts to refute the current goal q. The procedure is called when a refutation of q has been found.

addAutarky(F, q, depth): Add q to the subset of `F` $\rightarrow M$ associated with `depth`. The procedure is called when all attempts to refute q have failed. The `depth` parameter is strictly unnecessary, but is used to aid the checking of internal consistency.

screenResolvables(F, q): Assuming $q \neq \top$, this function finds all clauses containing $\neg q$, then eliminates those having a literal in `F` $\rightarrow A$ and returns the rest in an ordered list. For efficiency, the returned list begins with clauses having 0 subgoals, if any, followed by clauses having 1 subgoal, followed by all others.

If $q = \top$ and mode = ALL, the returned list contains all clauses of S, with the user-specified initial top clause first in the list.

If $q = \top$ and mode = SINGLE, the returned list contains just the user-specified initial top clause.

extractSubgoals(F, lits): Deletes from `lits` those literals that are complementary to $(A \cup \mathcal{L})$. Those remaining are partitioned into failed goals and goals with at least one clause eligible for extension.

6.1 Correctness

This section addresses correctness of Modoc without lemmas. Soundness of lemma processing is an orthogonal issue, and depends on exactly how lemmas are added to the basic algorithm. The main idea is that Modoc simply evaluates the universal PDST for the formula S when mode equals ALL, or the PDST for a user-specified top clause when mode equals SINGLE (see Figure 14). A call to `tryRefuteSubgoal` positions the search at a goal node, and evaluates the refuter's choices of clause. A choice that returns UNSAT is a winning choice for the refuter, so if one such is found, additional choices need not be evaluated. A call to `tryRefuteClause` positions the search at a clause node, and evaluates the spoiler's choices of goal. Now, a choice that returns SAT is a winning choice for the spoiler, so if one such is found, additional choices need not be evaluated. However, knowledge of autarkies is used to avoid searches where the outcome is predictable.

The stack of alternating goals and clause-ids determines a position in the PDST or universal PDST that the search has reached upon an invocation of `tryRefuteSubgoal` or `tryRefuteClause`. The inductive hypothesis is that the algorithm behaves correctly on proper subtrees of the tree rooted at this point.

The notation of Definition 6.1 is used. However, in this section, the set of lemmas `F` $\rightarrow \mathcal{L}$ remains empty. Recall the definition $S|A$ from Definition 2.4.

If `tryRefuteSubgoal` returns UNSAT, then $S|(A + q)$ is unsatisfiable. If `tryRefuteClause` returns UNSAT, then $S|A$ is unsatisfiable. These facts follow from the analogy with PDST trees (Theorem 4.4),

```
tryRefuteSubgoal(F:Context, q:Literal, depth:integer) :  (Status, LitSet, ClauseList)
/* Returns (result, ΔM, ΔR). result = SAT or UNSAT.
** If SAT, ΔM holds autarky Lits. If UNSAT, ΔR holds refutation. */
    begin
    pushAncestor(F, q, depth);
    assert(not inAutarky(F, invertLit(q));
    /* When loop exits, result = SAT or UNSAT. If SAT, curM holds autarky Lits.
    ** If UNSAT, curClause holds the refuted clause id. */
    remClauses := screenResolvables(F, q);
    result := SAT;
    curClause := 0;
    curM := ∅;
    while (remClauses ≠ ∅ and result == SAT)
        begin
        curClause := getMember(remClauses);
        remClauses := remClauses - curClause;
        if (inAutRem(F, curClause))
            begin
            (result, ΔM1, ΔR) := tryRefuteClause(F, curClause, depth + 1);
            curM := curM + ΔM1;
            /* If result = UNSAT, loop will exit. */
            end
        end
    if (result == UNSAT)
        begin
        addLemma(F, invertLit(q), curClause, depth);
        delAutarky(F, curM);
        ΔM := ∅;
        end
    else
        begin
        addAutarky(F, q, depth);
        ΔM := curM + q;
        ΔR := ∅;
        end
    delLemmas(F, depth);
    popAncestor(F, q, depth);
    return (result, ΔM, ΔR);
    end
```

Figure 12: Procedure `tryRefuteSubgoal` of Modoc Algorithm. Two mutually recursive procedures comprise the reasoning engine. See text for discussion.

```
tryRefuteClause(F:Context, curClause:Clause, depth:integer) :  (Status, LitSet, ClauseList)
/* returns (result, ΔM, ΔR). result = SAT or UNSAT.
** If SAT, ΔM holds autarky Lits. If UNSAT, ΔR holds refutation. */
    begin
    lits := getClauseLits(curClause);
    (failedSubgoals, goodSubgoals) := extractSubgoals(F, lits);
    if (failedSubgoals ≠ ∅)
        begin
        addAutarky(F, failedSubgoals);
        ΔM := failedSubgoals;
        ΔR := ∅;
        result := SAT;
        return (result, ΔM, ΔR);
        end

    result := UNSAT;
    remSubgoals := goodSubgoals;
    while (result == UNSAT and remSubgoals ≠ ∅)
        begin
        q := getMember(remSubgoals);
        remSubgoals := remSubgoals - q;
        if (not isLemma(F, invertLit(q)) )
            begin
            (result, ΔM, ΔR1) := tryRefuteSubgoal(F, q, depth);
            ΔR := append(ΔR, ΔR1);
            /* If result = SAT, loop will exit. */
            end
        end
    if (result == UNSAT)
        begin
        ΔM := ∅;
        ΔR := append(list(curClause), ΔR);
        end
    return (result, ΔM, ΔR);
    end
```

Figure 13: Procedure `tryRefuteClause` of Modoc Algorithm. Two mutually recursive procedures comprise the reasoning engine. See text for discussion.

---

and will not be re-proved. As pointed out by Shostack, a list of clauses in the order in which they are used for extensions suffices to specify a propositional model-elimination derivation [Sho76]. It is straightforward that the algorithm returns such a list when it returns UNSAT. Also, by simple inspection of the algorithm, when either function returns UNSAT, then $\mathtt{F} \rightarrow M$ is unchanged upon exit from the value upon function invocation. This follows in `tryRefuteSubgoal` because the call to `delAutarky` deletes any literals added to $\mathtt{F} \rightarrow M$ since the procedure invocation.

Theorem 6.1: Let `F` denote the context. Let A denote $\mathtt{F} \rightarrow A$ upon a function invocation. Let M_0 denote $\mathtt{F} \rightarrow M$ upon a function invocation.

(A) Suppose `tryRefuteSubgoal(F, q, depth)` is called. If M_0 is an autarky for $S|A$ and contains neither q nor $\neg q$, and `tryRefuteSubgoal` returns $(\mathtt{SAT}, \Delta M, \Delta R)$, then upon exit:

```
modoc(S:Formula, initTopClause:Clause, allTop:integer) :  (Status, LitSet)
/* returns (result, M, R).
** result = SAT or UNSAT. If SAT, M holds autarky Lits.
** In this case, if mode was ALL, M is a model of S;
** otherwise, initTopClause is satisfied by M, but S may not be.
** If UNSAT, R holds refutation. */
    begin
    if (allTop == 1) mode := ALL; else mode := SINGLE;
    F := setupContext(S, initTopClause, mode);
    (result, M, R) := tryRefuteSubgoal(F, ⊤, 0);
    return (result, M, R);
    end
```

Figure 14: Top level of Modoc Algorithm. In mode `ALL`, all clauses are eligible as top clauses, but the search tries `initTopClause` first. Either a refutation is found or a model is found. In mode `SINGLE`, either a refutation for `initTopClause` specifically is found or an autarky is found that satisfies that clause. No clause satisfied by the autarky is part of a minimal unsatisfiable set of clauses. See text for discussion.

---

1. $\mathtt{F} \rightarrow A = A$ upon exit.
2. $q \in \Delta M$,
3. $M_0 + \Delta M$ is an autarky for $S|A$,
4. $M_0 + \Delta M$ is the value of $\mathtt{F} \rightarrow M$ upon exit.

(B) Suppose `tryRefuteClause(F, curClause, depth)` is called. If M_0 is an autarky for $S|A$ and does not satisfy `curClause`, and `tryRefuteClause` returns $(\mathtt{SAT}, \Delta M, \Delta R)$, then upon exit:

1. $\mathtt{F} \rightarrow A = A$ upon exit.
2. ΔM satisfies `curClause`,
3. $M_0 + \Delta M$ is an autarky for $S|A$,
4. $M_0 + \Delta M$ is the value of $\mathtt{F} \rightarrow M$ upon exit.

Proof: Consider first `tryRefuteSubgoal`. By hypothesis, $q \notin M_0$, so M_0 is also an autarky for $S|(A + q)$. Let us define

$$M_i = M_0 + \sum_{j=1}^{i} \Delta M_j$$

where ΔM_j is returned by the j-th recursive call of `tryRefuteClause`. (Note that `curM` takes on successive values of $\sum_{j=1}^{i} \Delta M_j$.) It follows that the preconditions of `tryRefuteClause` hold at the time of the recursive call. By Theorem 5.3 (and the assumed correctness of the recursive call), M_i is an autarky for $S|(A + q)$. The calls to `addAutarky` keep $\mathtt{F} \rightarrow M$ equal to M_i as defined here, so any clauses bypassed by the test `inAutRem` must evaluate to SAT with respect to the M_i in effect when the clause was bypassed. But $\Delta M = \mathtt{curM}_{final} + q$. Thus every clause containing $\neg q$ contains a literal in A or in $M_0 + \Delta M$. This is also true for clauses containing q, so Part A is established.

For Part B, we only need to observe that `tryRefuteClause` returns SAT only if a subgoal with no possible extensions is found, or some recursive call to `tryRefuteSubgoal` returns SAT. In the first case correctness is obvious. In the second case, all *prior* recursive calls to `tryRefuteSubgoal` returned UNSAT, and did not change $\mathtt{F} \rightarrow M$. By the inductive hypothesis, $\mathtt{F} \rightarrow M$ is an autarky as of the exit from the recursive `tryRefuteSubgoal` that returned SAT, and clearly it satisfies `curClause`. ∎

Corollary 6.2: If Modoc operates in mode ALL, and returns SAT, then $\mathtt{F} \rightarrow M$ is a model of S.

Proof: Similar to Part A of Theorem 6.1, with $A = \emptyset$, $M_0 = \emptyset$. In the top-level invocation of `tryRefuteSubgoal` a list of all clauses in S is returned by `screenResolvables`. As each clause is considered as `curClause` in the loop, either it is already satisfied by $\mathtt{F} \rightarrow M$ and bypassed, or a refutation is attempted and fails. In the latter case, the clause is satisfied by $\mathtt{F} \rightarrow M$ after `tryRefuteClause` returns. But $\mathtt{F} \rightarrow M$ grows monotonically, so upon exit $\mathtt{F} \rightarrow M$ satisfies every clause in S. ▮

Note: Although the mode ALL guarantees a definitive result for both Modoc and Model Elimination, all of the experiments reported were run in mode SINGLE to avoid possible excessive time by Model Elimination on satisfiable formulas.

7 Lemmas and C-Literals

In the Model Elimination procedure a "lemma" may be recorded upon the completion of any (sub)refutation [Lov69, FLSY74, Lov78]. This is a clause that is logically implied by the original formula. Lemmas are not necessary for completeness of Model Elimination. Shostack proposed a "C-literal" mechanism that is reasonably efficient to implement [Sho74, Sho76]. This approach has been extended by Letz *et al.*, who give a detailed treatment and propose a related pruning strategy, called "strong regularity" [LMG94]. A detailed treatment of lemma strategies is beyond the scope of this paper. This section sketches how lemmas are incorporated into the basic implementation of the Modoc algorithm. We introduce and describe a strategy for "quasi-persistent" lemmas.

In general, the trade-offs are not well understood between the cost of storing and maintaining the lemma (considering it for resolutions, etc.), *vs.* the time saved when it can be used. This topic has been studied empirically for first order theorem proving. Although initial experience was negative [FLSY74], subsequent reports were more positive [AS92, Sti94, LMG94, AL97]. To the best of our knowledge, there are no empirical studies of lemma strategies on propositional problems.

Suppose the refutation of a literal q is completed at a point in a PDT (Definition 3.2) where the set of proper ancestors of q is A. Let B be the subset of A that represents the goals that were actually used in the refutation (say $B = \{p_1, \ldots, p_m\}$, where m may be 0). In model-elimination terms they were used for reductions; in PDT terms the p_i *prevented* the goals $\neg p_i$ from appearing in the subtree below q. Then a lemma clause, $[\neg q, \neg p_1, \ldots, \neg p_m]$, can be derived soundly [LMG94]. This is an implication of the form $(B \supset \neg q)$. (Loveland described the same inference, but using quite different terminology, such as *scope* of A-literals [Lov69].)

Example 7.1: Recall the refutation search described in Example 5.4, based on Figure 4. To refute e, the goal is extended with the clause $[\neg c, \neg e]$, which has no subgoals, due to the ancestor c. (In Model Elimination, the goal $\neg c$ is created, then is immediately reduced using ancestor goal 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 $[d]$. □

Shostack proposed an efficient strategy to maintain such lemmas in Model Elimination chains [Sho76]; Letz *et al.* generalized it to trees [LMG94]. 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 among $\{p_1, \ldots, p_m\}$; call this goal p_c. (If $m = 0$, attach it to the root of the PDT, which is normally $\top$.) This technique implicitly weakens the lemma to $(ancs(p_c) \supset \neg q)$, because the precise set $\{p_1, \ldots, p_m\}$ is not recorded.

The lemma can only be used in the subtree of p_c, and in this context its only use is to "extend" another occurrence of the goal q, because subgoals p_i, being in B, cannot occur beneath p_c. Moreover, when the lemma *is* used, its clause node has no subgoals in the PDT. (In Model Elimination, the extension is followed immediately by reductions on all of the subgoals.) In this sense, a C-literal is somewhat like an ancestor, in that it immediately closes off a branch of the PDT tree, and 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 is actually unnecessary to record it fully. The C-literal $\neg q$ and the lowest ancestor p_c are all that are needed. Thus the difference between B and $ancs(p_c)$ is immaterial with this strategy.

The main idea of Modoc is autarky pruning, which 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. However, lemmas seem to be important for efficiency, so they were incorporated into the basic implementation of Modoc.

7.1 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.

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 7.2: This example continues the refutation search begun in Example 7.1, based on Figure 4. 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.

7.2 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".

8 Worst Case Bounds

This section analyzes worst case bounds on the running times of Model Elimination and Modoc. The analysis is in terms of the size of the search spaces. Because the term "exponential time" is used in various ways, we will adopt the following precise definitions for this section.

Definition 8.1: By a *simple exponential function* of parameter n we mean a function whose growth rate is $\Theta(2^{\alpha n})$, for some positive constant α.

By a *super-exponential function* of parameter n we mean a function that grows faster than $2^{\alpha n}$ for every fixed α. For example, $n!$ is super-exponential in n. □

In terms of unrestricted propositional CNF, no worst case upper bounds on Model Elimination with C-literals have been previously published. This section will show that Model Elimination with C-literals has a super-exponential lower bound in n, the number of propositional variables, and that Modoc has a simple exponential upper bound in n. Thus, an exponential separation exists between Modoc and Model Elimination.

8.1 Known Worst-Case Bounds

Two interesting classes of satisfiability algorithms are: (1) those that are (seriously) implemented, and (2) those that are analyzed. These two classes are disjoint. There have been numerous recent reports of experimental success, but no upper bounds have been reported for the algorithms used. On the other hand, several highly simplified algorithms have been analyzed on certain classes of random formulas, but here the purpose was not to analyze the running time (a trivial exercise), but to analyze the probability of success.

Urquhart has shown that all algorithms that are based on resolution, or can be simulated by resolution with a polynomial blow-up, have a worst-case *lower bound* of $(1+\epsilon)^L$, where L is the formula length, for some $\epsilon > 0$ [Urq87]. However, no value for ϵ has been published, to our knowledge. Of course, this immediately implies a lower bound of $(1+\epsilon)^n$. All algorithms mentioned in this paper fall into the class covered by this lower bound. Most lower bound work in the satisfiability area, including that of Urquhart, and Tseitin and Haken before him, has been based on resolution *proof length*.

Plaisted has argued that a more meaningful measure for actual, specified, algorithms is the size of the search space. This argument does not criticize Urquhart's result, which applies to *all possible* resolution-based algorithms. Plaisted constructed several families of Horn clauses, and evaluated several well-known resolution strategies on Horn families, using the size of the search space as the metric [Pla94]. It is known that all propositional Horn clause formulas can be solved with a linear sized search space, yet he found that several resolution methods constructed exponential[2] sized search spaces for one or more of his Horn families. In particular, Model Elimination *without* C-literals was exponential on certain Horn families, whereas Model Elimination *with* C-literals and *with failure caching* [Elk89, AS92] has a linear sized search space for all propositional Horn formulas. However, failure caching is not used on non-Horn formulas because, as Plaisted and others point out, it would make the Model Elimination procedure incomplete. See Section 9.4 for experimental results.

8.2 Model Elimination Bounds

This section shows that Model Elimination with C-literals has a super-exponential worst case lower bound in n, the number of propositional variables. That is, for any given α, we demonstrate how to construct an infinite family of formulas whose search space size grows faster than $2^{\alpha n}$.

The basis of the construction is called an N-module. Each family of formulas builds formulas by using an N-module, together with a fixed *base module*, consisting of:

$$\begin{array}{ll} [\neg w, y, z] & [\neg w, y, \neg z] \\ [\neg w, \neg y, \neg z] & [\neg w, \neg y, z] \end{array}$$

An N-module consists of these clauses:

$$[\neg x_i, w, x_j] \qquad \text{for } 1 \le i \ne j \le N$$

Definition 8.2: An *N-module formula* is a 3CNF formula with $n = N + 3$ variables whose clauses consist of an N-module and the base module. □

An N-module formula is satisfiable by the assignment $\neg w$, x_1, …, x_N. The base module is not Horn and is not renameable-Horn, by the theorem of Lewis [Lew78], as well as by inspection.

Theorem 8.1: The Model Elimination search space is super-exponential in n on the family of N-module formulas.

Proof: The Model Elimination resolution search will begin at some designated top clause, say $[\neg x_N, w, x_1]$. The argument is similar for any top clause. Whenever w is selected as a goal node, it will be refuted promptly by the base module. We are interested in the branching that occurs when the selected goal node is some positive x_i literal. To get started suppose x_1 is selected at top (first) level. (Again, the argument is similar if $\neg x_N$ is selected, except goal nodes are negative literals.) There are $(N-1)$ clauses of the form $[\neg x_1, w, x_j]$, for $2 \le j \le N$. All of these will be considered as extensions, leading to goals at the second level of the forms

[2] Plaisted did not define "exponential", but apparently intended it to describe any super-polynomial function.

x_j. Similarly, each node at the second level will have $(N-2)$ clauses as eligible extensions. In general, at level d the branching is $(N-d)$. Observe that each attempted refutation descends N levels, then fails for lack of eligible clauses. No C-literal for any x_i or $\neg x_i$ is ever derived. Thus the entire tree just described is explored, and this tree contains at least $(N-1)!$ nodes. By Stirling's formula,

$$(N-1)! \quad = \quad \Theta\left(e^{(N-1/2)(\log N-1)}\right) = \Theta\left(e^{(n-7/2)(\log n-1)}\right) \tag{1}$$

which grows faster than any simple exponential function of n. ∎

The above analysis has been verified experimentally (Section 9).

While we cannot claim that Modoc does less searching than Model Elimination on every formula, we can claim that any gains by Model Elimination are due to accidents in the order in which clauses and goals are chosen. That is, any run of Model Elimination can be "shadowed" by Modoc. Model Elimination may choose a clause that Modoc bypasses due to autarky pruning. Although that clause must eventually fail, it may produce a lemma that survives. However, if Model Elimination ever has an occasion to use that lemma later, say to reduce a potential goal q, then Modoc will be able to repeat M. E.'s lemma derivation at that point simply by choosing q as the next goal. This informal argument can be made precise.

8.3 Upper Bound for Modoc

This section shows that Modoc's search space has an upper bound that is a simple exponential function of n. Together with the previous section, this shows that autarky pruning achieves an exponential speed-up when incorporated into Model Elimination with C-literals.

At any point in the search, let us define *active* literals to be those literals that can possibly become subgoals in the current search subtree. If q is an ancestor literal, then neither q nor $\neg q$ is active. If q is an autarky literal, then neither q nor $\neg q$ is active. If q is a C-literal, then $\neg q$ is not active. Let N denote the number of active literals. At the start of the search, $N = 2n$, for a formula with n variables.

Theorem 8.2: For any clause considered in the procedure `tryRefuteClause`, if there are N active literals, the number of goal nodes created while processing that clause is at most 2^N.

Proof: The proof is by induction on N. The base case, $N = 1$, is immediate, as the search either succeeds or fails after creating one goal node, which reduces the active literal count to 0.

For $N > 1$, assume the claim holds for $1 \le M < N$. There are two cases to consider, depending on whether the current clause C contains a C-literal.

Case I: The current clause C does not contain a C-literal. Some literal q is chosen as the first subgoal. As q becomes an ancestor in the subtree of which it is root, the active literal count is reduced by 2; that is, both q and $\neg q$ are no longer active literals. Now procedure `tryRefuteSubgoal` will try to refute q by considering, in some sequence, the eligible clauses that contain $\neg q$. But each eligible clause that fails to refute q, must return at least two new autarky literals (unless it fails immediately, in which case it created no goal nodes). Each new autarky literal reduces the count of active literals for the next clause by two if the autarky literal is not already a C-literal, and by one if it *is* already a C-literal. Thus the total reduction in active literals from one clause to the next, as attempted by `tryRefuteSubgoal`, is at least two. The number of subgoals created by this invocation of `tryRefuteSubgoal` is the sum of the subgoals created by the attempted clauses, which, by the inductive hypothesis, is at most $2^{N-2} + 2^{N-4} + \ldots$. This sum is bounded by $\frac{1}{3}2^N$.

If the subgoal q eventually fails, then `tryRefuteClause` is also done, so assume that q is eventually refuted. This causes a new C-literal, $\neg q$, to be attached to some ancestor of clause C. Now the remaining subgoals of C, if any, are processed in an environment of $N-1$ active literals. Again, the inductive hypothesis implies that the remaining processing is bounded by 2^{N-1}. Thus the total processing of clause C is at most $\frac{5}{6}2^N < 2^N$, as was to be proved.

Case II: The current clause C does contain a C-literal, say q. Then q is chosen as the first *and only* subgoal of C to be processed. If q cannot be refuted, of course C is done. The point is that, if q *is refuted*, a lemma-induced cut occurs, and no further processing of C occurs (see Section 7.2). As q becomes an ancestor in the subtree of which it is root, the active literal count is reduced by 1; that is, $\neg q$ is no longer active. Now procedure `tryRefuteSubgoal` will try to refute q by considering, in some sequence, the eligible clauses that contain $\neg q$. As in case I, each subsequent clause that is attempted has an environment containing at least two fewer active literals than the preceding clause. However, in this case the first attempted clause has $(N-1)$ active literals, rather than $(N-2)$. Thus the sum over all clauses attempted by this invocation of `tryRefuteSubgoal` is at most $2^{N-1} + 2^{N-3} + \ldots$, which is less than $\frac{2}{3}2^N$. This is also the number of subgoals created while processing clause C, as explained at the beginning of this case. ∎

The bound 2^N obtained in Theorem 8.2 implies a bound of 2^{2n}, showing that Modoc is in simple exponential time in n. Although only goal nodes were counted, it is clear that the clause nodes are within a polynomial factor of the goal nodes. As is clear from the proof, the bound is not tight. In fact, no tight bounds of this kind are known for any complete satisfiability algorithms. Thus the only real value of this bound is to separate the worst case of Modoc from the worst case of Model Elimination (Section 8.2).

9 Experimental Results

A basic implementation of Modoc was programmed in C. While it uses efficient data structures, there are no *heuristics* for ordering the search. (Possible heuristics might consist of choosing among extension clauses according to how many literals they contain, and/or choosing among sibling goal literals according to how many clauses extend them, etc.) The purpose of this section is to demonstrate that the methodology employed by Modoc deserves further investigation, and not to provide an exhaustive empirical study of this (preliminary) implementation. While this paper was under review, additional progress on Modoc has been made on several fronts [Oku98, VGO98, VGO97], and more extensive empirical results may be found in those reports.

This section reports experiments on several classes of formulas, with comparisons among several programs. The formulas in the experimentation range from random formulas, with no structure, to pigeon-hole formulas, "N-module" formulas (Definition 8.2), and certain formulas proposed by Plaisted, which have highly regular structures. They also include circuit-test formulas with irregular structure.

The Programs Four programs were used for experimentation: Modoc; Model Elimination with C-literals and Lemma-Induced Cuts (M. E.), which is essentially Modoc without autarky pruning; the published DPLL (Definition 2.6); and another model-searching algorithm that incorporates 2-closure, called `2cl` [VGT96]. DPLL fills the role of a model-search program without highly tuned search heuristics, to correspond with Modoc's and M. E.'s lack of search heuristics, while `2cl` represents model-search *with* highly tuned search heuristics. (See Section 9.6 for a comparison between `2cl` and `sato3` [Zha97], which was reported while this paper was under review.)

All programs were coded in C, with attention to efficient data structures. Methods described by Dalal and Etherington for unit clause propagation [DE92], and developed independently by several others, were used. Related ideas were applied to pure literal propagation and other frequently used subroutines.

vari-ables	clau-ses	liter-als	Modoc Goals avg.	Modoc CPU secs. avg.	Modoc CPU secs. std.dev.	DPLL Branches avg.	DPLL CPU secs. avg.	DPLL CPU secs. std.dev.	`2cl` Branches avg.	`2cl` CPU secs. avg.	`2cl` CPU secs. std.dev.
50	214	642	8,653	0.29	0.20	287	0.07	0.05	11	0.19	0.03
71	303	909	64,039	2.27	1.83	1,601	0.42	0.40	27	0.33	0.14
100	427	1281	711,578	27.39	19.98	14,196	5.06	5.18	76	1.02	0.56
141	602	1806	15,258,719	676.22	566.35	357,998	180.14	224.19	322	6.60	4.94

Table 4: Comparative performances on random 3CNF formulas of Modoc, DPLL and `2cl`. There were 200 samples at each formula size, so the standard error is .07 times the standard deviation. Model Elimination did not complete enough formulas to compile statistics. See Section 9 for discussion.

Measurements For all programs CPU times on a Sun Sparcstation 10/41 are reported. CPU times are subject to considerable fluctuation, and certainly are not repeatable by another implementation. However, they are comparable across algorithms.

In addition, for each program, a repeatable count of a key operation was reported, to indicate the size of the search space for that program. For the refutation programs, it was goal nodes created during the search. For model-search programs, it was branching nodes, where a selected variable is set first to one truth value, then upon backtracking, to the other. These measures are highly correlated with CPU time, but are not directly comparable across algorithms. Thus the branching count for DPLL cannot be directly compared to the goal node count for Modoc or M. E., or even the branching count of `2cl`. Whenever averages are reported, some measure of dispersion is also reported, usually the standard deviation.

One Top Clause Only for M. E. Due to similarity of implementation, the performance differences between M. E. and Modoc can only be due to autarky pruning. For all M. E. tests, the mode was SINGLE (see Figure 14). That is, a refutation was attempted from one top clause only on each run reported. Unless it is known from the application that the selected top clause must be in a minimal unsatisfiable subset of clauses (if any such subset exists), the failure of this one attempt does not imply that the formula is satisfiable. See Section 9.5 for a case in which such key top clauses *are* known.

Modoc tests start from the same top clause as M. E., but if that attempt fails, Modoc continues on to other clauses, if any eligible clauses remain. Because of autarky pruning, very few eligible clauses remain, as a rule. As a result, Modoc produces a decisive result on both unsatisfiable and satisfiable formulas. Of course, DPLL and `2cl` also produce a decisive result on both unsatisfiable and satisfiable formulas.

9.1 Random Formulas

Programs were tested on random 3CNF formulas generated according to the constant-clause-width model introduced by Franco and Paull [FP83]: every clause containing 3 different variables and any combination of literal polarities is equally likely. They demonstrated that the asymptotic probability of generating a satisfiable formula approaches zero at any clause/variable ratio above 5.27. The number of clauses in these experiments is about 4.27 times the number of variables in all cases. This ratio has been found experimentally to have a probability of about one half of generating a satisfiable formula. Experiments also indicated that formulas generated at this ratio tend to be harder than other ratios [MSL92]. In a natural sense, random formulas can be considered to be the least structured of all possible formulas.

Recall that Figure 1 showed that Model Elimination suffered a rapid performance degradation on small satisfiable random formulas, ranging from 15 to 20 variables, and became hopeless for 50 variables. However, Pigeon-Hole Formulas

pig- eons	vari- ables	clau- ses	Modoc CPU	Modoc goals	M. E. CPU	M. E. goals	DPLL CPU	DPLL branches	2cl CPU	2cl branches
4	12	22	0.06	48	0.05	48	0.02	16	0.05	4
5	20	45	0.07	197	0.08	197	0.02	120	0.07	22
6	30	81	0.11	982	0.10	982	0.10	1,116	0.14	118
7	42	133	0.32	5,877	0.30	5,877	1.15	12,992	0.67	758
8	56	204	1.98	41,108	1.89	41,108	19.48	183,632	4.77	5,210
9	72	297	17.01	328,813	17.73	328,813	383.20	3,062,880	34.37	41,794
10	90	415	193.43	2,959,242	142.76	2,959,242	??	??	310.36	377,084
11	110	561	2225.39	29,592,317	1537.77	29,592,317	??	??	3109.41	3,721,588

Pigeon-Hole Formulas Less One Binary Clause

pig- eons	vari- ables	clau- ses	Modoc CPU	Modoc goals	M. E. CPU	M. E. goals	DPLL CPU	DPLL branches	2cl CPU	2cl branches
4	12	21	0.05	15	0.06	29	0.01	4	0.05	2
5	20	44	0.06	20	0.06	144	0.01	29	0.05	5
6	30	80	0.08	26	0.10	851	0.01	24	0.05	9
7	42	132	0.09	33	0.26	5,762	0.17	1,832	0.07	17
8	56	203	0.10	41	1.49	44,071	0.13	1,049	0.08	27
9	72	296	0.12	50	13.07	377,012	39.91	330,163	0.09	36
10	90	414	0.13	60	147.96	3,575,339	26.49	169,022	0.80	851
11	110	560	0.13	71	1737.91	37,284,310	??	??	0.96	996

Table 5: Comparative performances on pigeon-hole formulas (unsat), and pigeon-hole formulas with one binary clause removed (sat). M. E. attempted one refutation with the first clause as top clause. Thus, on the satisfiable formulas, its conclusion is “don’t know”, rather than “sat”. “??” indicates that these formulas were not solved within one CPU hour. Times are CPU seconds on a Sun Sparcstation 10/41.

Figure 2 showed that Modoc overcomes that problem, and generally solves satisfiable formulas *faster* than comparable unsatisfiable formulas.

Results on medium-scale random 3CNF formulas are shown in Figure 4. Modoc is slower than DPLL in absolute terms, but DPLL is growing at a distinctly faster rate. However, both algorithms are overshadowed by the highly developed `2cl`. For reasons explained, statistics could not be obtained for Model Elimination.

9.2 Pigeon-Hole Formulas

Figure 5 shows results on pigeon-hole formulas and modified pigeon-hole formulas. Recall that the k-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 M. E. perform exactly the same search, because the refutation is found without any backtracking through any *positive* clauses. Shallow backtracking does occur through negative clauses, causing Modoc to derive autarky literals. However, they never get used, and each program creates exactly the same number of goal nodes in every case. Thus, the pigeon-hole table indicates how much bookkeeping overhead in CPU time is associated with autarky analysis. On these highly structured

N-Module Formulas

	variab-	clau-	Modoc		M. E.			DPLL		2cl	
N	les (n)	ses	CPU	goals	CPU	goals (g)	$\log_2(g)$	CPU	br.s	CPU	br.s
4	7	16	0.00	7	0.01	246	7.94	0.01	3	0.04	2
5	8	24	0.01	8	0.03	1,646	10.68	0.01	3	0.04	2
6	9	34	0.00	9	0.23	12,366	13.59	0.01	3	0.05	2
7	10	46	0.00	10	2.01	103,914	16.67	0.01	3	0.05	2
8	11	60	0.00	11	19.72	969,926	19.89	0.01	3	0.05	2
9	12	76	0.00	12	212.84	9,976,470	23.25	0.01	3	0.06	2
10	13	94	0.00	13	2515.64	112,235,406	26.74	0.02	3	0.05	2

Table 6: Performance on N-module formulas, used in the lower bound proof for Model Elimination (Definition 8.2). All clauses have 3 literals. Observe that the number of variables is $n = N + 3$. For Model Elimination, the size of the search space is denoted by g. Simple exponential growth in n would be indicated if $\log_2(g)$ appeared to be converging to a constant. However, at least in the range of this table, g is growing at a super-exponential rate in n.

---

unsatisfiable formulas, we see that the refutation methods outperform the model-search methods.

Satisfiable versions of pigeon-hole formulas were created by removing one binary clause from the approximate middle of the formula. (Results vary widely depending on which clause is removed, but removing a middle clause avoids extremes in either direction.) The main themes are seen again, but not so pronouncedly as for random formulas. Modoc and `2cl` have an easier time with the satisfiable modifications, by several orders of magnitude, and are competitive. DPLL also has an easier time with the satisfiable modifications, confirming the principle that model-search methods do better on satisfiable formulas than on unsatisfiable, but still did not finish the 110-variable formula within an hour. On the other hand, M. E. is losing ground, relative to the original unsatisfiable version.

9.3 N-Module Formulas

Section 8.2 introduced N-module formulas to establish a lower bound on the size of the Model Elimination search space in the worst case. For completeness, and to confirm the analysis, the results of the four programs on this family of formulas are shown in Figure 6. As expected, the formulas are trivial for model-search methods. Confirming the analysis, they become quickly intractable for M. E., and at the astonishingly low level of 14 variables. (Actually, to demonstrate satisfiability, M. E. has to iterate through all top clauses, and would have taken several hours on the 12-variable formula.) However, Modoc exploits autarky pruning to eliminate redundant searching done by M. E., and solves the formulas in trivial time.

9.4 Plaisted Formulas

Plaisted recommended that new theorem-proving proposals be analyzed or tested on certain classes of formulas for which small search spaces are known to be possible [Pla94], as mentioned earlier in Section 8.1. The basic classes were Horn formulas, but he also described transformations to generate related non-Horn formulas that are easy in principle, but which cause problems for many existing methods. We generated and tested formulas from two of those classes with the results shown in Figure 7. Again, we see that Modoc has a knack for finding its way through easy structured formulas, keeping pace with model-search methods, whereas M. E. gets lost in fruitless redundant search.

Plaisted S_N^2 Horn Clause Family

	vari-	clau-	Modoc		M. E.		DPLL		2cl	
N	ables	ses	CPU	goals	CPU	goals	CPU	branches	CPU	branches
5	30	51	0.01	7	0.00	40	0.01	20	0.05	2
10	110	201	0.01	48	0.05	2,048	0.02	91	0.05	3
15	240	451	0.01	97	1.87	81,920	0.04	210	0.06	3
20	420	801	0.02	198	79.39	3,407,872	0.11	381	0.08	3
50	2550	5001	0.12	1,248	??	??	3.51	2,451	0.39	4
100	10100	20001	0.50	4,998	??	??	56.70	9,901	2.92	19

Plaisted $U(S_N^2)$ non-Horn Clause Family

	vari-	clau-	Modoc		M. E.		DPLL		2cl	
N	ables	ses	CPU	goals	CPU	goals	CPU	branches	CPU	branches
5	80	101	0.01	18	0.07	3,200	0.01	25	0.05	2
10	310	401	0.02	74	524.75	25,296,896	0.03	100	0.06	2
15	690	901	0.02	138	??	??	0.06	225	0.10	4
20	1220	1601	0.04	254	??	??	0.12	400	0.15	4
50	7550	10001	0.28	1,394	??	??	2.45	2,500	1.00	8
100	30100	40001	1.12	5,294	??	??	35.22	10,000	5.49	12

Table 7: Performance on Plaisted families S_N^2 and $U(S_N^2)$. "??" indicates that program did not finish within 3600 CPU seconds. Family S_N^2 is Horn, while family $U(S_N^2)$ is non-Horn. See text for details.

9.5 Circuit-Testing Formulas

Hardware and software design verification applications can be supported by high performance satisfiability procedures. Two characteristics of these formulas make a goal-sensitive refutation procedure the most natural choice:

1. The formula often represents the negation of a (conjectured) theorem, and many of the clauses are axioms that may or may not be relevant to the proof. Model-searching methods may spend a lot of time thrashing around among the irrelevant axioms.

2. The application is usually able to identify one, or a small number of, key clauses. One of these, when used as top clause, will lead to a refutation, if *any* refutation exists.

However, in previously known goal-sensitive refutation methods, the search redundancy outweighed the advantage of goal-sensitivity. The intuitive explanation is that an important goal is normally connected to some irrelevant clauses at some point in the search. Once the search gets into this irrelevant subregion, it begins to act like a satisfiable formula, and the high redundancy effects dominate. The results in this section show that Modoc takes a major stride toward reclaiming this ground for refutation-based methods.

The formulas tested in this section derive from the hardware application called *Automated Test Pattern Generation* [Lar92]. Faults simulated include "single stuck-at" (code `ssa`) and "bridge" (code `bf`). Circuits are taken from the ISCAS85 benchmark. We thank Tracy Larrabee and her research group for providing formulas.

At a high level, the idea is that the desired circuit is described by an input/output relation $y = Q(x)$ where both y and x are Boolean vectors. If the circuit has a certain fault f in its fabrication, then its behavior is described by a different relation $z = Q_f(x)$. The fault is detectable if and only if there is some x such that y and z differ on some bit. Let y and z have k bits (i.e., the circuit has k output wires), and let

Circuit Family	No. of Fmlas	Number of Variables Avg.	Range	Avg. No. of Literals	Modoc Goals Avg.	Modoc CPU sec. Avg.	St.dev.	2cl Br.s Avg.	2cl CPU sec. Avg.	St.dev.
`ssa0432`	7	433	(427–435)	2347	23,441	0.82	0.54	268	0.77	0.12
`ssa2670`	12	1320	(986–1359)	7414	5,975,869	250.58	164.51	520,449	1539.60	639.34
`ssa6288`	3	10406	(10404–10410)	87355	1,916,629	83.70	27.73	0	39.45	6.10
`ssa7552`	80	1495	(1391–2013)	7945	56,297	2.25	2.32	23	1.93	0.28
`bf0432`	21	886	(421–1057)	7703	3,479,676	151.39	203.44	1,407	5.85	6.48
`bf1355`	149	2266	(1450–2298)	18192	2,382,166	101.33	122.67	13,254	41.81	20.85
`bf2670`	53	1300	(694–1784)	7904	9,341,926	388.73	1033.05	109,347	439.01	2436.85
aggregate	325	1830	(421–10410)	13213	3,093,128	130.22	429.67	43,227	148.84	991.81

Table 8: Performance comparison of Modoc and `2cl` on circuit-testing formulas. As explained in the text, Model Elimination was unable to solve any of these formulas, while DPLL solved only one family, which was `ssa6288`. On this family DPLL performed 0 branches, with 0.67 seconds average time, and 0.04 standard deviation.

"$\oplus$" denote *exclusive or*. Then the detection condition is

$$y = Q(x) \ \wedge\ z = Q_f(x) \ \wedge\ w = ((y_1 \oplus z_1) \vee \ldots \vee (y_k \oplus z_k)) \ \wedge\ w$$

If the CNF version of this formula is satisfiable, the assignments for the x-bits provide a *test pattern* for the fault f. If the formula is unsatisfiable, fault f is undetectable.

Note that w will become a unit clause in the CNF version of the above formula. Moreover, it is easy to see that the formula must be satisfiable (indeed, by *any* assignment for the x-bits), if the unit clause w is omitted. Thus this is the key top clause, which can be identified automatically by the application. The formulas can be presented so that both refutation and model-search methods will start at this clause. However, model-search methods have no mechanism to maintain this connection, after the first branch in the search tree. (Indeed, the only circuit-testing formulas that DPLL could solve were solved entirely by unit-clause propagation, before any search commenced.)

We did not have computing resources to test M. E. and DPLL for one hour on each of the 325 formulas reported. We tried both programs on several formulas that were solved very easily by Modoc and `2cl`, giving them about 1000 times as much time as was required by Modoc, but they failed on each, with the exception that DPLL succeeded on the `ssa6288` family. M. E. and DPLL were also tried on numerous other arbitrarily chosen formulas, but failed to solve any within one hour each. Therefore, we are unable to report any statistics for M. E., and almost none for DPLL, on the circuit-testing formulas.

Results on circuit-testing formulas are summarized in Figure 8. The same formulas were reported for `2cl` in connection with the 1993 DIMACS Implementation Challenge for Cliques, Coloring and Satisfiability [VGT96]. The most difficult circuit family was `ssa2670`, based on the sum of Modoc and `2cl` times. On this family, Modoc ran six times faster than `2cl`. Modoc was also faster on the second most difficult family, `bf2670`, but was slower than `2cl` on several other families. On all 325 formulas as a group, Modoc was faster than `2cl`, and also more consistent, as shown by the much smaller standard deviation. Because of the great variations among the formulas it is not possible to determine a rigorous measure of statistical significance for the group as a whole.

9.6 Comparison of 2cl and sato3

While this paper was under review a program named `sato3` was compared empirically with eight other satisfiability solvers, and was reported as being "either the best or the second best for every class of problems except `par16`" [Zha97]. One referee of this paper requested that `2cl` be compared to `sato3`, in order to calibrate the empirical results in this section against other complete satisfiability solvers. For the comparison both `2cl` and `sato3` were run with the parameters reported in the literature [VGT96, Zha97], on the same Sun platform. The comparisons for the tables presented in this section (except Table 6, which was too easy for meaningful results) are summarized as follows:

Table:	4	5, unsat	5, sat	7, S_N^2	7, $U(S_N^2)$	8
`sato3`		1.9		4.7		92.6
`2cl`	4.9		92.0		4.0	

Nonblank entries in the `sato3` row are cases where `sato3` ran faster than `2cl` by that factor. Nonblank entries in the `2cl` row are cases where `2cl` ran faster than `sato3` by that factor. For Table 7, $U(S_N^2)$, `sato3` crashed while reading the formula for the last row, so the comparison is based on the first five rows. For all tables the total time taken by each program for all formulas in the table was accumulated, and the ratios computed. Therefore, the harder formulas tend to dominate the ratios above.

10 Conclusions and Future Work

We have introduced a method to incorporate autarky analysis into propositional resolution procedures. Modoc constitutes one implementation of the basic idea with a minimum of frills. The empirical 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 circuit-testing formulas. In the model-search arena, research has improved the empirical performance of the basic DPLL algorithm substantially [Pre95, VGT96, Zha97]. We are optimistic that further research can uncover substantial improvements in Modoc.

Worst case analysis demonstrated that Modoc is exponentially more efficient than Model Elimination, in terms of the size of the search space. Experimental data back up the analysis.

The primary goal of the research was to develop a method that could take advantage of the irregular structure that is present in many application-based formulas. Experimental results on circuit-testing formulas indicate that progress toward this goal has been made.

Future work should proceed along several directions, including research into more effective lemma techniques, search heuristics, and an extension to first-order theorem proving. Applications to other logics, such as modal logics, also deserve investigation. Another open question is the worst-case complexity of Modoc or an improved version of Modoc.

Acknowledgements

This work was supported in part by NSF grants CCR-8958590 and CCR-9503830, by equipment donations from Sun Microsystems, Inc., and software donations from Quintus Computer Systems, Inc. We thank Dr. Reinhold Letz for helpful discussions, and thank Fumiaki Okushi for programming and experimental assistance. We also thank the anonymous referees for helpful comments on the initial draft, and for calling to our attention the work of Giunchiglia and Sebastiani.

References

[AB70] R. Anderson and W. W. Bledsoe. A linear format for resolution with merging and a new technique for establishing completeness. *Journal of the ACM*, 17(3):525–534, 1970.

[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.

[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.

[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.

[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.

[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.

[DR94] R. Dechter and I. Rish. Directional resolution: the davis-putnam procedure, revisited. In *Proc. 4th Int'l Conf. on Principles of Knowledge Representation and Reasoning (KR'94)*, pages 134–145. Morgan Kaufmann, San Francisco, 1994.

[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.

[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.

[FP83] J. Franco and M. Paull. Probabilistic analysis of the Davis-Putnam procedure for solving the satisfiability problem. *Discrete Applied Mathematics*, 5:77–87, 1983.

[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.

[GS96a] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures – the case study of modal K. In *13th International Conference on Automated Deduction*, pages 583–597. Springer-Verlag, 1996.

[GS96b] F. Giunchiglia and R. Sebastiani. A SAT-based decision procedure for ALC. In *International Conference on Principles of Knowledge Representation and Reasoning*, Cambridge, MA, USA, November 1996.

[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.

[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.

[Lew78] H. R. Lewis. Renaming a set of clauses as a Horn set. *Journal of the Association for Computing Machinery*, 25(1):134–135, January 1978.

[Lov68] D. W. Loveland. Mechanical theorem-proving by model elimination. *Journal of the Association for Computing Machinery*, 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.

[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.

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

[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.

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

[Oku98] F. Okushi. Parallel cooperative propositional theorem proving. (Submitted for publication; preliminary version presented at the Fifth International Symposium on Artificial Intelligence and Mathematics: `http://rutcor.rutgers.edu/~amai`), 1998.

[Pla84] D. A. Plaisted. (private communication), 1984.

[Pla94] D. A. Plaisted. The search efficiency of theorem proving strategies. In *12th International Conference on Automated Deduction*, pages 57–71. Springer-Verlag, 1994.

[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.

[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.

[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.

[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.

[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.

[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.

[Urq87] A. Urquhart. Hard examples for resolution. *Journal of the Association for Computing Machinery*, 34(1):209–219, January 1987.

[VGO98] A. Van Gelder and F. Okushi. A propositional theorem prover to solve planning and other problems. (Submitted for publication; preliminary version presented at the Fifth International Symposium on Artificial Intelligence and Mathematics: `http://rutcor.rutgers.edu/~amai`), 1998.

[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.

[VGO97] A. Van Gelder and F. Okushi. Lemma and cut strategies for propositional model elimination. (Submitted for publication; preliminary version presented at the Fifth International Symposium on Artificial Intelligence and Mathematics: `http://rutcor.rutgers.edu/~amai`), 1997.

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