Complexity Analysis of Propositional Resolution with Autarky Pruning

Allen Van Gelder*

Abstract

An algorithm called "Modoc", which has been introduced elsewhere, enhances propositional model elimination with autarky pruning, and other features. The model elimination method is based on linear resolution, and is designed to produce refutations of formulas in conjunctive normal form (CNF). Informally, an autarky is a "self-sufficient" model for some clauses, but which does not affect the remaining clauses of the formula. Modoc finds a model if it fails to find a refutation, essentially by combining autarkies. Although the original motivation for autarky pruning was to extract a model when the refutation attempt failed, practical experience has shown that it also greatly increases the performance, by reducing the amount of search redundancy. This paper presents a worst-case analysis of Modoc as a function of the number of propositional variables in the formula. The analysis sheds light on why autarky pruning improves the performance, compared to "standard" model elimination. A worst-case analysis of the original algorithm of Davis, Putnam, Loveland and Logemann (DPLL) is also presented. The Modoc analysis yields a worst-case upper bound that is not as strong as the best known upper bound for model-searching satisfiability methods, on general propositional CNF. However, it is the first time a nontrivial upper bound on non-Horn formulas has been shown for any resolution-based refutation procedure.

Key Words: Satisfiability, Boolean formula, propositional formula, autarky, resolution, refutation, model, model elimination.

1 Introduction

The *satisfiability problem* has been the subject of continuing research, which has increased in intensity with the advent of high-speed microprocessors. This is the problem of deciding whether a propositional Boolean formula has a satisfying assignment. A closely related problem is to determine whether a formula states a *propositional theorem*. A formula is a theorem if and only if its negation is *unsatisfiable*.

The formula is presented in *conjunctive normal form* (CNF), also called *clause form*. Each clause is a disjunction of literals, and clauses are joined conjunctively. We assume that each clause is nonredundant (no duplicate literals) and nontrivial (no complementary literals).

1.1 Satisfiability Methods

Three basic methods have been developed for satisfiability testing: refutation search, model search, and local search (see [VG95] for additional bibliography). These methods vary in terms of the *certificates* they provide, as discussed in Section 1.3.

1. Refutation search seeks to discover a proof that a formula is unsatisfiable, usually employing resolution. If a complete search for a refutation fails, the formula is "pronounced" satisfiable. Model elimination and SL-resolution typify these methods [Lov69, KK71, Lov72]. However, they cannot provide a model, or other certificate, on satisfiable formulas.

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

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, but without any certificate. The DPLL algorithm, due to Davis, Putnam, Loveland and Logemann [DLL62] is the basis for many modern refinements. This basic algorithm consists of the *unit clause* and *pure literal* rules for simplification, and the *splitting* rule for searching. As published, the splitting rule specifies that the splitting variable be chosen from a shortest clause. A different approach is to treat the problem in terms of integer linear programming.
3. Several methods employ various local search heuristics to perform incomplete model searches. They are more properly characterized as *max-sat* methods. They cannot even "pronounce" a formula to be unsatisfiable. They can only report "don't know" and give up based on resource limits, when they fail to discover a model. However, they have succeeded in finding models on much larger formulas than current complete methods can handle.

1.2 Goal-Sensitivity

The main motivation for developing a high-performance resolution-based tool for satisfiability is the ability to *focus* the refutation attempt. In applications involving unsatisfiable formulas, it is often the case that one key disjunctive clause is known, such that, *if* the formula is unsatisfiable, then this clause is part of the minimal unsatisfiable set of clauses. (The negated conclusion of the theorem is such a clause.) In fact, the bulk of clauses often represent a large body of background axioms, known to be consistent, most of which are irrelevant to the reason the key clause causes inconsistency.

Linear resolution methods can exploit this information by starting the refutation attempts at one or more key clauses. In this sense, they are goal-sensitive, and potentially focused. No straightforward method is known by which model-searching methods, such as DPLL and variants, or max-sat variants, can achieve a similar focus. Despite this apparent advantage, prior experience with propositional resolution has been negative (and consequently, largely unreported). The reason for prior poor performance of propositional resolution is discussed in Section 1.4.

1.3 Certificates

As a practical consideration, a program may be required to produce "evidence", or a "certificate", to support 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, models and resolution proofs can be *checked* without knowing anything about the programs that produced them. 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 "satisfiable" and "unsatisfiable" decisions.

Currently implemented propositional methods are "one-sided" in the information that they can provide to support their answers. As mentioned, DPLL is able to provide a model as a certificate to support a "satisfiable" answer, but cannot support an "unsatisfiable" answer. How to enhance the basic DPLL algorithm to produce a resolution refutation is known, but (a) to our knowledge, it has never been implemented, and (b) most modern implementations include several additional features, for which methods to produce refutations are *not* known.

Refutation methods are able to produce a resolution proof as a certificate for an "unsatisfiable" answer, but cannot support a "satisfiable" answer. The technique of autarky pruning used by Modoc evolved out of investigation of how to extract a model from an unsuccessful attempt to construct a resolution refutation.

Fmla Size		Unsatisfiable				Satisfiable			
		No.	CPU	Extensions		No.	CPU	Extensions	
Vars	Cl.s	Smp.	Avg.	Avg.	Max	Smp.	Avg.	Avg.	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. *Abbreviations*: "Cl.s" for "Clauses", "No. Smp." for "Number of Samples". The clause/variable ratio is chosen so that about half of the formulas are satisfiable and half are unsatisfiable. Times are CPU seconds on a Sun Sparcstation 10/41. "??" indicates that none of these formulas were solved within 1200 seconds. On all formulas a refutation was attempted with the first clause as top clause. On the satisfiable formulas, the failure of this one refutation attempt does *not* imply satisfiability of the formula.

Fmla Size		Unsatisfiable				Satisfiable			
		No.	CPU	Extensions		No.	CPU	Extensions	
Vars	Cl.s	Smp.	Avg.	Avg.	Max	Smp.	Avg.	Avg.	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 autarky pruning added to Model Elimination with C-literals (M. E.). *Abbreviations*: "Cl.s" for "Clauses", "No. Smp." for "Number of Samples". Test conditions are the same as Table 1. Compared to M. E. in that figure, performance has improved by a factor of 1,000,000 on satisfiable 20-variable formulas.

1.4 Search Redundancy of Model Elimination

A serious problem with existing propositional refutation methods is illustrated dramatically in Table 1. Model elimination (abbreviated M. E.) is regarded as one of the most efficient refutation strategies. It is able to solve *unsatisfiable* random 3CNF formulas (4.27 clauses per variable) with up to 100 variables, but it bogs down on *satisfiable* formulas about at 20 variables. It should be noted that, for modern model-search methods, these formulas (both unsatisfiable and satisfiable) are considered *easy* at 100 variables and *trivial* at 20 variables. The same phenomenon occurs on propositional formulas from other applications [VGK96].

Related behavior, which can be described as *search redundancy*, has been observed elsewhere. Plaisted has shown that many goal-sensitive resolution procedures have exponential worst cases on Horn formulas [Pla94]. This indicates a highly redundant search because such formulas can be solved with a linear-size search. Giunchiglia and Sebastiani have observed very high search redundancy in a tableau procedure for

n	denotes the number of variables,
L	denotes the total number of literals,
$P(L)$	is a low-degree polynomial,
k	clauses have at most k literals,
s	clauses have at most $s = k - 1$ subgoals,
γ_k	base of exponential function.

Table 3: Notation for upper bounds analysis.

x, y, z	propositional variables
p, q, r	literals (signed variables)
C, D	disjunctive clauses
$\mathcal{F}$	CNF formula being tested, a set of clauses
$\mathcal{A}$	a set of ancestor literals
$\mathcal{L}$	a set of C-literals, annotated with ancestor dependencies
$\mathcal{M}$	a set of autarky literals
$\mathcal{E}$	a set of eligible clauses
$\mathcal{S}$	a set of subgoal literals
`stat`	status and return code

Table 4: Notation for Modoc procedures.

propositional modal logic [GS96]. M. E. is an instance of a tableau method for classical logic.

Modoc addresses the search-redundancy problems just described by the use of a new technique, called *autarky pruning*, as well as other innovations. Its performance on the same formulas as in Table 1 is shown in Table 2. Its performance has been studied experimentally, and it has solved formulas from a circuit test pattern application with over 10,000 variables and 87,000 literals in 15 seconds on a Sun Sparcstation 10/41 [VGK96]. This paper gives a worst-case upper bound analysis of Modoc.

1.5 Known Worst-Case Bounds

Existing satisfiability algorithms fall into two disjoint classes: those that are implemented, and those that are analyzed. There have been numerous recent reports of experimental success, but no upper bounds have been reported for the algorithms used.

Although there have been many papers that analyze "the Davis-Putnam algorithm" (meaning DPLL, Section 1.1) from one standpoint or another, a closer inspection reveals that in nearly all cases the upper bound analysis deals with either a simplification of DPLL, or an extension of it. Statistical analyses on random populations involve simplifications. Worst-case analyses always include some enhancements to the originally published DPLL algorithm; presently, no substantial experimental work has been reported for analyzed enhancements.

Monien and Speckenmeyer showed an upper bound for general CNF formulas, as a function of the number of propositional variables in the formula (n) and the maximum number of literals in any clause (k) [MS85]. It was based on a version of DPLL, enhanced by autarky analysis, and is of the form:

$$T_k(n) \quad \leq \quad P(L) \;\; {\gamma_k}^n \tag{1}$$

Notation is explained in Table 3. The parameter γ_k is between 1 and 2, and satisfies the equation

$$\gamma^{k-1} = \gamma^{k-2} + \gamma^{k-3} + \ldots + 1 = \frac{\gamma^{k-1} - 1}{\gamma - 1} \tag{2}$$

For $k = 3$, $\gamma_3 = 1.618\ldots$, the "golden ratio". For larger k, γ_k increases toward 2 (see Section 2 and Figure 8).

Recently, algorithms specifically for 3-CNF have been proposed, and values of γ_3 smaller than 1.618 have been derived for them. The algorithms are of the model-search variety, essentially being DPLL with complicated enhancements. They have not been implemented, and some appear to be quite difficult to implement. To put the following numbers in perspective, note that the originally published DPLL has $\gamma_3 < 1.696$, as shown in Section 2. This can be improved to $\gamma_3 = 1.618$ either by the method of Monien and Speckenmeyer, or as described at the end of Section 2. Schiermeyer announced 1.579 in a conference

[Sch92], and Kullmann has a method in the pipeline that is about 1.505 [Kul94]. At the Siena Workshop on Satisfiability, Schiermeyer announced another slight improvement closer to 1.5.

Theoretical results on 3-CNF are interesting, but from a practical standpoint, most applications do not naturally produce 3-CNF formulas, and the transformation from general CNF to 3-CNF introduces new variables, so the upper bound does not transfer usefully. Therefore, we do not expect to see widespread use of methods that are specialized to 3-CNF.

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 $(1 + \epsilon)^n$, for some $\epsilon > 0$, but no value for ϵ has been published [Urq87]. All algorithms mentioned in this paper fall into the class covered by this lower bound.

1.6 Summary of Results and Overview

Section 5 presents a worst-case upper bound analysis of Modoc, as actually implemented, as a function of the number of propositional variables in the formula (n) and the maximum number of literals in any clause (k). A nontrivial upper bound for the original DPLL is also presented for the first time (Section 2). The Modoc upper bound is not as strong as that for DPLL, and is further above that shown by Monien and Speckenmeyer [MS85], for general CNF formulas (see Figure 8). For example, Modoc and DPLL achieve $\gamma_3 = 1.696$, while Monien and Speckenmeyer achieve $\gamma_3 = 1.618$.

Although the provable upper bound for Modoc is not as strong as that of certain model-search algorithms, we believe it is still significant, because

1. Goal-sensitive resolution-based methods may be preferable to model-search in practice because of their ability to focus on key clauses, as mentioned in Section 1.2.
2. Modoc has been implemented and has achieved some experimental success [VGK96].
3. *No* reasonable upper bound, not even 2^n, is known for other resolution-based methods (see Section 5.4). In fact, despite the fact that *linear* time algorithms are known for Horn clause formulas, Plaisted has shown that many resolution methods have an *exponential* worst case on this class [Pla94];
4. The analysis indicates which aspects of the algorithm contribute to the strength of the upper bound (and indicates why other resolution methods have such weak upper bounds).
5. The analysis involves mutual recursion among multivariate functions, with recurrences involving a *max* operator, for which off-the-shelf techniques do not seem to exist. Thus the solution may be interesting in its own right, because similar recurrences can be produced by other algorithms with recursion inside case splits.

The paper is organized as follows. Complexity of the original DPLL on 3-CNF is analyzed in Section 2. The Modoc Algorithm is presented in Section 3. Associated *Modoc search trees* are covered in Section 4. The worst case upper bound analysis is carried out in Section 5. Section 5.4 explains why autarky pruning is crucial to the achievement of the upper bound. Conclusions and future work are mentioned in Section 6.

2 Complexity of Original DPLL and a Simple Modification

This section proves that the original DPLL [DLL62] achieves well under 2^n search steps on 3-CNF, and achieves less than 2^n search steps on general CNF. We call each use of the splitting rule a *search step*. We also show how a simple modification of this rule achieves the same bound on search steps as the algorithm of Monien and Speckenmeyer [MS85]. Recall that DPLL states, for the splitting rule:

"Choose a variable in a shortest clause."

Let $T_k(n)$ denote a worst case upper bound on the number search steps in a k-CNF formula in which the shortest clause has k literals, $k \geq 3$. Let $T_{k,j}(n)$ denote the bound on a k-CNF formula in which the shortest clause has j literals, $2 \leq j < k$. Obviously, $T_k(n) = 2\,T_{k,k-1}(n-1)$.

To simplify the base cases, we do not require that a k-CNF formula actually have a clause of width k, only that no clause *exceeds* width k. It is easily shown that the base cases are $T_{k,j}(j) = 2^{j-2}$.

Consider the situation when the splitting rule selects a shortest clause C, with j literals in it, and then selects literal q within that clause to split upon. Because the splitting rule is only applied on a formula with no pure literals, some other clause(s) contains $\neg q$. Thus, the assignment $q = true$, besides satisfying C, will normally produce some clause of length less than k in the resulting subproblem. Also, the assignment $q = false$ shortens C to $j-1$ literals. This leads to the recurrences

$$T_{k,j}(n) \quad \leq \quad T_{k,k-1}(n-1) + T_{k,j-1}(n-1) \qquad 3 \leq j < k \tag{3}$$

The alternative to the "normal" case is that the claimed short clause does not exist in the subproblem (after application of the pure literal and unit clause rules). But in this case, the subproblem has at most $(n-2)$ variables, we would replace one or both terms on the right-hand side of Eq. 3 with $T_k(n-2) = 2\,T_{k,k-1}(n-3)$, and obtain a more favorable recurrence. For $k \geq 4$, chaining the recurrences of Eq. 3 gives

$$T_{k,k-1}(n) \quad \leq \quad T_{k,k-1}(n-1) + \ldots + T_{k,k-1}(n-k+3) + T_{k,2}(n-k+3) \tag{4}$$

2.1 Analysis for Original Splitting Rule

The recurrence for $T_{k,2}(n)$ varies from Eq. 3 because a unit clause is created in the second subproblem. Let $[b, c]$ be chosen for application of the splitting rule, and assume b is chosen from this clause. For the assignment $b = true$, the first subproblem is governed by $T_{k,k-1}(n-1)$, as above.

For the assignment $b = false$, the unit clause c is created. The second subproblem will have at most $(n-2)$ variables. In most cases, it will have clauses shortened by other occurrences of b, and all occurrences of $\neg c$, after the unit clause rule has been applied with c. In these cases, the second subproblem is governed by $T_{k,k-1}(n-2)$.

Exception: The only significant exception occurs when there is no surviving clause that previously contained either b or $\neg c$. That is, before the assignments to b and c, each occurrence of b was accompanied by c in the same clause, and each occurrence of $\neg c$ was accompanied by $\neg b$ in the same clause. Notice that $\{\neg b, c\}$ is an autarky in this case (see Definition 3.2 and Definition 4.1). When this exception occurs, the second subproblem is governed by $T_k(n-2)$. Of course, other exceptions, due to additional unit clauses and/or pure literals, are possible, but they create still smaller subformulas, and are never worst cases.

Summing the worst cases of both assignments to b, recalling that $T_k(n-2) = 2\,T_{k,k-1}(n-3)$, we arrive at the recurrence

$$T_{k,2}(n) \quad \leq \quad T_{k,k-1}(n-1) + \max\{T_{k,k-1}(n-2),\ 2\,T_{k,k-1}(n-3)\} \tag{5}$$

Substituting into Eq. 4 (if $k \geq 4$), the recurrence becomes

$$T_{k,k-1}(n) \quad \leq \quad T_{k,k-1}(n-1) + \ldots + T_{k,k-1}(n-k+2) + \max\{T_{k,k-1}(n-k+1),\ 2\,T_{k,k-1}(n-k)\} \tag{6}$$

We can prove that $T_{k,k-1}(n) \leq \gamma^n$ provided that γ satisfies:

$$\gamma^{k-2} + \ldots + \gamma + 1 \quad \leq \quad \gamma^{k-1} \tag{7}$$

$$\gamma^{k-1} + \ldots + \gamma^2 + 2 \quad \leq \quad \gamma^k \tag{8}$$

The smallest γ that satisfies these constraints is called γ_k. Equation 8 is due to the "exception" described above. For all $\gamma > 1$, satisfaction of Eq. 8 implies satisfaction of Eq. 7. Therefore, for the original DPLL, γ_k

is the largest root of Eq. 8, with the inequality replaced by equality. For example, γ_3 is between 1.695 and 1.696. See Figure 8 for other k, up to 6.

2.2 A Simple Improvement

Any additional criterion for choosing the splitting variable that prevents Eq. 8 from applying can improve the performance of the original DPLL. Without Eq. 8, which was due to the "exception", the value of γ_k is governed by Eq. 7, giving the bounds obtained by Monien and Speckenmeyer (see Eq. 2 and Figure 8). Essentially, Monien and Speckenmeyer avoided this "exception" by performing an autarky analysis on the clause selected for splitting. (They analyze several splits in advance when the chosen clause has more than two literals; the above analysis shows that the same bound can be obtained by analyzing only when the splitting clause is binary.)

Another such criterion that is practical to implement is the following:

> "Choose a shortest clause (the splitting clause), then choose the *literal* that occurs the most frequently in the formula, from among the literals in that clause."

The word "literal" is emphasized because the counts should *not* include occurrences of the complement of that literal.

For the case of a binary splitting clause, $[b, c]$, assume b is chosen by this criterion. Then b occurs at least as often as c. The only way the above "exception" might apply in the second subproblem is if each occurrence of b is accompanied by a c. But then the first assignment, $b = \mathit{true}$, causes $\neg c$ to become pure. The first subproblem might have no short clauses in this case, but it would have at most $(n-2)$ variables, and be governed by $T_k(n-2)$. The second subproblem would also be governed by $T_k(n-2)$. But $T_k(n-2) = 2\,T_{k,k-1}(n-3)$. Combined, and substituting into Eq. 4 (if $k \geq 4$), they produce the recurrence

$$\begin{aligned} T_{k,k-1}(n) \quad \leq \quad & T_{k,k-1}(n-1) + \ldots + T_{k,k-1}(n-k+3) + \\ & \max\{T_{k,k-1}(n-k+2) + T_{k,k-1}(n-k+1),\ 4\,T_{k,k-1}(n-k)\} \end{aligned} \tag{9}$$

We can prove that $T_{k,k-1}(n) \leq \gamma^n$ provided that γ satisfies Eq. 7, as well as:

$$4 \quad \leq \quad \gamma^3 \qquad (\text{if } k = 3) \tag{10}$$

$$\gamma^{k-1} + \ldots + \gamma^3 + 4 \quad \leq \quad \gamma^k \qquad (\text{if } k \geq 4) \tag{11}$$

The smallest γ that satisfies these constraints is called γ_k for the refined DPLL. However, if γ satisfies Eq. 7, then we have $\gamma^2 + \gamma > 4$, from which it follows that Eq. 10 or 11, whichever is applicable, is also satisfied.

To summarize, with the refined choice of splitting variable, if the "exception" applies in the second subproblem, then first subproblem is smaller than usual, and the sum is not a worst case. Therefore, the bounds obtained by Monien and Speckenmeyer also apply to this modification of DPLL. Again, the bound only requires the refinement on binary splitting clauses.

3 The Modoc Algorithm

The essence of the Modoc algorithm for testing satisfiability of a propositional CNF formula is contained in two mutually recursive procedures, `tryRefuteGoal` and `tryRefuteClause`. These procedures are described abstractly, using the notation of Table 4, in Figures 1 and 2. Correctness has been proven in detail elsewhere [VG95], but is sketched in Sects. 4.1 and 4.2 to provide a more self-contained presentation. This paper is concerned with the worst case performance.

We assume that each input clause is nonredundant (no duplicate literals) and nontrivial (no complementary literals). This restriction is easily met by preprocessing, if necessary.

	tryRefuteGoal$(p, \mathcal{A}_{in}, \mathcal{M}_{in}, \mathcal{L}_{in})$
trg-01	$\mathcal{A} = \mathcal{A}_{in} + \{p\}$
trg-02	$\mathcal{M}_{new} = \emptyset$; $\mathcal{M} = \mathcal{M}_{in}$
trg-03	$\mathcal{L}_{new} =$ `unitClausePropagation`$(\mathcal{A} + \mathcal{L}_{in}) - \mathcal{L}_{in}$; $\mathcal{L} = \mathcal{L}_{in} + \mathcal{L}_{new}$
trg-04	`stat` = `fail`
trg-05	Extract into $\mathcal{E}$ clauses of $\mathcal{F}$ that contain $\neg p$ and do *not* contain any literal in $(\mathcal{A} + \mathcal{M})$. (Note 1)
trg-06	`while (stat == fail` **and** $\mathcal{E} \neq \emptyset$`)`
trg-07	Remove from $\mathcal{E}$ a clause, C, preferring one with no subgoals (see line trc-03)
trg-08	$($`stat`$, \mathcal{M}_C, \mathcal{L}_C) =$ `tryRefuteClause`$(C, \mathcal{A}, \mathcal{M}, \mathcal{L})$
trg-09	$\mathcal{M}_{new} = \mathcal{M}_{new} + \mathcal{M}_C$; $\mathcal{M} = \mathcal{M}_{in} + \mathcal{M}_{new}$
trg-10	$\mathcal{L}_{new} = \mathcal{L}_{new} + \mathcal{L}_C$; $\mathcal{L} = \mathcal{L}_{in} + \mathcal{L}_{new}$
trg-11	`if (stat == cut` **and** `isCutGoal(`$p, \mathcal{L}$`))` (Note 2)
trg-12	$q =$ `getCutLit`$(p, \mathcal{L})$
trg-13	`stat` = `succeed`
trg-14	Remove from $\mathcal{E}$ any clauses that contain a literal in $\mathcal{M}_C$. (Note 1)
trg-15	`if (stat == succeed)`
trg-16	$\mathcal{M}_{new} = \emptyset$
trg-17	Annotate $\neg p$ with ancestor dependencies, and add it to $\mathcal{L}_{new}$. (Note 3)
trg-18	`else if (stat == fail)`
trg-19	$\mathcal{M}_{new} = \mathcal{M}_{new} + \{p\}$
trg-20	`else if (stat == cut)`
trg-21	$\mathcal{M}_{new} = \emptyset$
trg-22	Remove from $\mathcal{L}_{new}$ any C-literals that depend on p.
trg-23	`return (stat,` $\mathcal{M}_{new}$`,` $\mathcal{L}_{new}$`)`

Figure 1: Abstract version of the `tryRefuteGoal` procedure. See notes in Figure 3.

	tryRefuteClause$(C, \mathcal{A}_{in}, \mathcal{M}_{in}, \mathcal{L}_{in})$
trc-01	$\mathcal{L}_{new} = \emptyset$; $\mathcal{L} = \mathcal{L}_{in}$
trc-02	`stat` = `succeed`
trc-03	Extract into $\mathcal{S}$ those literals of C whose complements are *not* in $\mathcal{A}_{in} + \mathcal{L}$.
trc-04	`while (stat == succeed` **and** $\mathcal{S} \neq \emptyset$`)`
trc-05	Remove from $\mathcal{S}$ a literal, p. (Note 4)
trc-06	$($`stat`$, \mathcal{M}_g, \mathcal{L}_g) =$ `tryRefuteGoal`$(p, \mathcal{A}_{in}, \mathcal{M}_{in}, \mathcal{L})$
trc-07	$\mathcal{L}_{new} = \mathcal{L}_{new} + \mathcal{L}_g$; $\mathcal{L} = \mathcal{L}_{in} + \mathcal{L}_{new}$
trc-08	`if (`$\mathcal{L}$ is contradictory`)`
trc-09	`stat = cut`
trc-10	Remove from $\mathcal{S}$ any literals whose complements are in $\mathcal{L}_g$
trc-11	`if (stat == fail)`
trc-12	$\mathcal{M}_{new} = \mathcal{M}_g$
trc-13	`else`
trc-14	$\mathcal{M}_{new} = \emptyset$
trc-15	`return (stat,` $\mathcal{M}_{new}$`,` $\mathcal{L}_{new}$`)`

Figure 2: Abstract version of the `tryRefuteClause` procedure. See notes in Figure 3.

Procedures take and return tuples of values, and the notation $(a, b, c) = p(d, e)$, in the style of ML, denotes that the 3-tuple of values returned by procedure p are stored in variables a, b, and c. Disjoint union of sets is denoted by "+". The equality test is ''=='', and assignment is "=", as in C. Scopes are indicated

Notes:

1. If a clause, D, contains a literal, r, whose complement, $\neg r$, is in $\mathcal{M}$, then that clause will be removed. Due to the *conditional autarky property* of $\mathcal{M}$ (see text), some *other* literal of D occurs in $(\mathcal{A} + \mathcal{M})$.
2. If $\mathcal{L}$ contains a complementary pair of C-literals, q and $\neg q$, such that q has an ancestor dependency on p, then `isCutGoal`$(p, \mathcal{L})$ returns *true* and `getCutLit`$(p, \mathcal{L})$ returns q.
3. When the refutation of C succeeds on line trg-08, then `tryRefuteClause` returns `succeed` as the value of `stat`. In this case, the *ancestor dependencies* of the new C-literal, $\neg p$, are derived from the *complements* of all the literals in C, except p itself; call them r_i. Each $\neg r_i$ is either an ancestor or a C-literal. If $\neg r_i$ is an ancestor, then $\neg p$ depends on $\neg r_i$. If $\neg r_i$ is a C-literal, then $\neg p$ depends on the *ancestor dependencies* of $\neg r_i$. If $\neg r_i$ is both an ancestor and a C-literal, then the C-literal interpretation is used.

   The other possibility to exit the loop of lines trg-06–14 is that `tryRefuteClause` returns `cut` as the value of `stat`. In this case, if `isCutGoal`$(p, \mathcal{L})$ is true, let $q =$ `getCutLit`$(p, \mathcal{L})$. Then the *ancestor dependencies* of the new C-literal, $\neg p$, consist of the ancestor dependencies of the C-literals q and $\neg q$. The situation is as though p were refuted by a "virtual tautological clause", $[\neg p, q, \neg q]$.
4. If any literal in $\mathcal{S}$ will cause `tryRefuteGoal` to fail immediately, by having an empty $\mathcal{E}$, such a literal must be selected. Otherwise, if any literal in $\mathcal{S}$ occurs in $\mathcal{L}$, some such literal must be selected.

Figure 3: Notes for Figs. 1 and 2.

by indentation. In this discussion, line identifiers "trg" and "trc" refer to Figures 1 and 2.

The procedures are almost duals in terms of their overall logic. Set $\mathcal{E}$ (lines trg-05, trg-07, trg-14) is called the set of *eligible* clauses. Essentially, `tryRefuteGoal` iterates through $\mathcal{E}$ calling `tryRefuteClause` until either it succeeds or $\mathcal{E}$ is exhausted. Set $\mathcal{S}$ (lines trc-03, trc-05, trc-10) is called the set of *subgoals*. Essentially, `tryRefuteClause` iterates through $\mathcal{S}$ calling `tryRefuteGoal` until either it fails or $\mathcal{S}$ is exhausted.

Definition 3.1: (C-Literal) A *C-literal* is a notational device to record an earlier derivation for possible later use [LMG94, VG95]. When a goal p is *refuted*, then a conditional conclusion of $\neg p$ follows, and $\neg p$ is called the C-literal. The conditions are certain goal ancestors in the current search tree (see Note 3 in Figure 3). The attachment point is the lowest (furthest from root) of these ancestors. Clearly, anywhere below the attachment point in the search tree, all the conditions hold, so $\neg p$ may function as a unit clause. (See Example 4.3 and Figure 5.) □

When a procedure succeeds, it returns one or more C-literals in $\mathcal{L}_{new}$. When a procedure fails, it returns one or more autarky literals in $\mathcal{M}_{new}$, and possibly some C-literals, as well. The upper bound analysis is based on counting these literals.

Returning some useful information upon failure is the essential improvement of Modoc over former linear resolution methods. Section 5.4 shows that it has a dramatic effect on the worst-case upper bound.

The conditional autarky property mentioned in Note 1 is stated formally in Definition 3.2 below. This property is also exploited in the upper bound analysis (see Definition 5.1). See Section 4.1 for an example.

Definition 3.2: (Conditional Autarky Property) Let $\mathcal{F}$ be a CNF formula and let $\mathcal{A}$ and $\mathcal{M}$ be disjoint sets of literals such that $(\mathcal{A}+\mathcal{M})$ is consistent. Let $\mathcal{F}_1$ be the set of clauses in $\mathcal{F}$ that contain the complement of some literal in $\mathcal{M}$, and let $\mathcal{F}_2$ be the set of clauses in $\mathcal{F}$ that contain some literal in $(\mathcal{A} + \mathcal{M})$. Then $\mathcal{M}$ is said to have the *conditional autarky property* with respect to $\mathcal{A}$ if and only if $\mathcal{F}_1 \subseteq \mathcal{F}_2$. □

Another novel and important feature, which is exploited in Section 5.1 is the *lemma-induced cut* (lines trg-11 to trg-13, trc-08 to trc-09, and Notes 2 and 3). When two complementary C-literals are derived, say

q and $\neg q$, the current refutation search can be abandoned, and "rolled back" to the lowest ancestor upon which one of q and $\neg q$ depends. This ancestor literal has been refuted. In the terminology introduced in Section 5.1, clauses and goals containing C-literals are called "C-limited". It is shown there that C-limited clauses and goals do not contribute to the exponential size of the search.

The top level of Modoc is similar to `tryRefuteGoal` except that the role of p, the new ancestor, is filled by a special symbol $\top$ (which can be thought of a *true*). The initial set of eligible clauses, $\mathcal{E}$, is either all of $\mathcal{F}$ or is a set of key clauses specified by the user. Of course the sets $\mathcal{A}_{in}$, $\mathcal{M}_{in}$, $\mathcal{L}_{in}$ are empty. If some clause, C, in $\mathcal{E}$ succeeds, the returned value of $\mathcal{L}_{new}$ will contain at least the complement of every literal in C. If no $C \in \mathcal{E}$ succeeds, the returned value of $\mathcal{M}_{new}$ will be an *autarky* for $\mathcal{F}$ (that is, a conditional autarky with respect to $\emptyset$) that satisfies at the clauses initially in $\mathcal{E}$.

Subsections 4.1 and 4.2 give additional details and examples on Modoc that help to explain the collapsed-tree construction (Section 5.1), which is needed in the upper bound derivation.

4 Modoc Search Trees

A particular run of Modoc can be characterized by a bipartite tree with a *goal* node corresponding to each invocation of `tryRefuteGoal`, and a *clause* node corresponding to each invocation of `tryRefuteClause`. Edges go from caller to subroutine in the obvious manner. Children are ordered left to right in call-order. Goal and clause nodes are labeled with the corresponding goal p and clause C, respectively. This structure is called a *Modoc search tree* (see Figure 6).

It is convenient to think of a C-literal as being "attached" to the lowest goal node among the C-literal's ancestor dependencies (see Example 4.3 and Figure 5). Observe that all of a C-literal's ancestor dependencies are actual ancestors in this tree. Once a C-literal is derived (line trg-17 in Figure 1), it is effective, or "visible" in nodes that are on or to the right of the path from the attachment node to the derivation node.

4.1 Background on Autarkies

This section defines "autarky" and indicates how the concept is used in Modoc. The operation of *autarky-based pruning* is the major contributer to Modoc's improved efficiency, relative to traditional model elimination. Space limitations prevent a complete exposition, but additional details, examples, and background may be found elsewhere [VG95]. This subsection is not essential for following the upper bound analysis, but may help in understanding the correctness of Modoc.

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

Definition 4.1: (autarky, *autsat*, *autrem*) Let S be a set of CNF clauses. A partial assignment M (normally represented as the set of literals assigned to "true"), possibly defined on some variables that do not occur in S, is called an *autarky* of S if M partitions S into two disjoint sets,

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

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

Example 4.1: Let $S = \{[a, b], [\neg a, c], [b, d]\}$. Then $\{a, c\}$ is an autarky of S, with

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

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

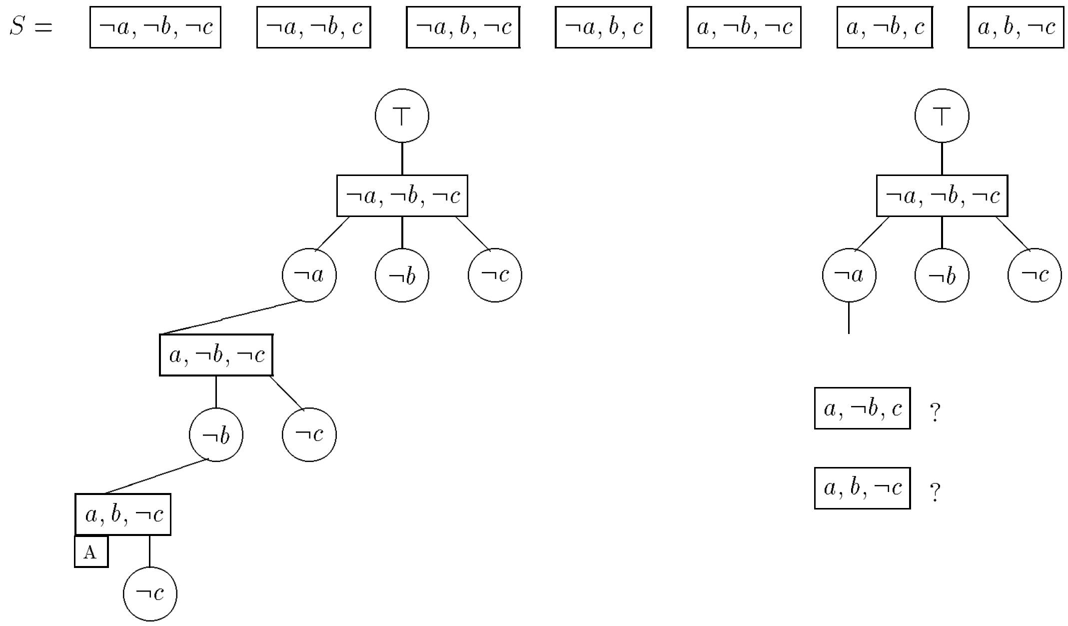

Figure 4: Model elimination search for Example 4.2. (Left) Search fails at lowest $\neg c$ goal. (Right) After backtracking to alternative choices at $\neg a$ goal.

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.

Example 4.2: We now give a simple example of how autarky pruning works in Modoc. The tree data structure is called a propositional derivation tree (PDT), which is essentially the successful part of a Modoc search tree. Citations to lines "trg" and "trc" refer to Figs. 1 and 2.

The formula S consists of all 3-CNF clauses on variables a, b, and c, except for the all positive clause, as shown at the top of Figure 4. Suppose the top clause is all negative, $[\neg a, \neg b, \neg c]$. Let us trace out a model elimination search for a refutation [MZ82, LMG94]. As shown on the left of Figure 4, 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, ancestor literal $\neg a$ prevents a from entering $\mathcal{S}$ at line trc-03, as indicated by the boxed "A". Similarly, b does not enter $\mathcal{S}$, so literal $\neg c$ is the only literal that remains to be refuted. So far, Modoc and model elimination are proceeding in lock-step. But the search procedure now fails, because each clause containing literal c also contains an ancestor literal, either $\neg a$ or $\neg b$.

If we stop and reflect on the meaning of this failure, we see that every clause containing the literal c is satisfied by a partial assignment consisting of the ancestors 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.

In Modoc, `tryRefuteGoal` reports this state of affairs by returning $\mathcal{M}_{new} = \{\neg c\}$ (line trg-19), which is passed up by `tryRefuteClause` (line trc12).

Modoc (and model elimination) now backtrack and look for another clause that resolves with $\neg b$, and does not contain the ancestor $\neg a$. There is none. We can now extend the conclusion of the earlier 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.

In Modoc, `tryRefuteGoal` at node $\neg b$ collects the literals returned by failing calls to `tryRefuteClause`, and adds $\neg b$, returning $\mathcal{M}_{new} = \{\neg c, \neg b\}$.

Modoc (and model elimination) again backtrack and look for another clause that resolves with $\neg a$ (Figure 4, right). There are two such clauses, as indicated. The standard model elimination algorithm would continue trying to construct a refutation using one of these clauses, then the other. But notice that both of these clauses are satisfied by the partial assignment M mentioned above.

After a few moments thought, we can predict that these refutation attempts must fail, without carrying out the search. Intuitively, the reason is that we cannot use a clause that is satisfied by M to "get outside of M". Every clause that might be passed into `tryRefuteClause` will have a subgoal that is satisfied by M; any such subgoal goes into $\mathcal{S}$ at line trc-03 because no ancestor is complementary to any literal in M. This ensures that $\mathcal{S}$ is not empty. Eventually, some goal is generated that has no eligible clauses at line trg-05.

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*.

This is where Modoc and model elimination part company. Model elimination carries out the fruitless searches with the clauses $[a, \neg b, c]$ and $[a, b, \neg c]$. However, in Modoc, `tryRefuteGoal` at node $\neg a$ has collected $\mathcal{M}_C = \{\neg c, \neg b\}$ from the failing call to `tryRefuteClause` (line trg-08) and has removed the additional clauses from its set of remaining eligible clauses ($\mathcal{E}$ in line trg-14). Since there are no remaining eligible clauses now, it fails without further searching, and adds $\neg a$ to $\mathcal{M}_{new}$ (line trg-19), which already contains $\neg b$ and $\neg c$ (from line trg-09). This $\mathcal{M}_{new}$ is passed back.

As argued in Section 5.4, the redundant searches can make an exponential difference between model elimination and Modoc. Tables 1 and 2 demonstrate that the difference is orders of magnitude in practice, even on relatively small formulas.

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

1. Autarky analysis can predict that certain refutation attempts must fail;
2. A model for a satisfiable formula can be constructed as a series of autarkies.

□

The conditional autarky property mentioned in Note 1 is stated formally in Definition 3.2, where $\mathcal{F}_1$ and $\mathcal{F}_2$ are defined. Modoc has this property at lines trg-05 and trg-14 [VG95], where $\mathcal{E}$ is caused to be disjoint from $\mathcal{F}_2$. Therefore, at line trg-07, $\mathcal{E}$ is disjoint from $\mathcal{F}_1$.

4.2 More Details on Modoc

This section gives some additional explanation of the Modoc algorithm, as presented abstractly in Figures 1 and 2, in Section 3.

To simplify the abstract presentation, each procedure invocation is assumed to have a private copy of the data structures, except that the original CNF formula, $\mathcal{F}$, is global. Making such copies does not affect the upper bound analysis substantially, as it only affects the polynomial $P(L)$ in Eq. 1. The analysis is concerned with the value of γ_k in that equation.

When a procedure succeeds, it returns one or more C-literals in $\mathcal{L}_{new}$. When a procedure fails, it returns one or more autarky literals in $\mathcal{M}_{new}$, and possibly some C-literals, as well. As mentioned earlier, returning some useful information upon failure is the essential improvement of Modoc over former linear resolution methods (Example 4.2). The use of C-literals was already known [LMG94].

Example 4.3: Consider the fragment of a Modoc search tree shown in Figure 5. We begin at the point where `tryRefuteClause` has been called with clause $[e, a, d]$, at the bottom of the figure. At this point no C-literals have been attached. Clause $[e, a, d]$ has no subgoals, so `tryRefuteClause` succeeds immediately. Therefore, $\neg e$ is refuted using clause $[e, a, d]$ together with ancestors $\neg a$ and $\neg d$. The latter are the dependencies for

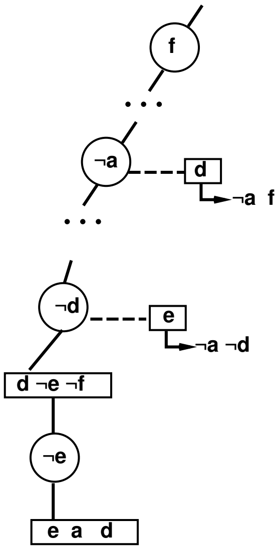

Figure 5: A fragment of a Modoc search tree showing the attachment of C-literals and their dependencies, as discussed in Example 4.3.

---

the new C-literal e, and it is attached at the lower of these two. This leads to the refutation of $\neg d$, which requires the ancestor f. Thus the new C-literal d depends on $\neg a$, through e, and directly on f. It is attached at the lower of these two. Since $\neg a$ may be arbitrarily higher in the tree than $\neg d$, additional clauses to the right of the path from $\neg a$ to $\neg d$ may use d as a unit clause, to instantly refute any additional occurrences of $\neg d$. $\square$

Another novel and important feature is the *lemma-induced cut* (lines trg-11 to trg-13, trc-08 to trc-09, and Notes 2 and 3). When two complementary C-literals are derived, say q and $\neg q$, the current refutation search can be abandoned, and "rolled back" to the lowest ancestor upon which one of q and $\neg q$ depends. This ancestor literal has been refuted.

Whenever a clause to be processed by `tryRefuteClause` contains a literal, p, which is also a C-literal, then p is the *only* literal of the clause for which `tryRefuteGoal` will be called, in view of Note 4. If p fails, of course the loop in `tryRefuteClause` is exited. But if p succeeds, its C-literal, $\neg p$, produces a contradiction in $\mathcal{L}$, and generates a lemma-induced cut. In the terminology introduced in Section 5.1, p and the clause in which it occurs are called "C-limited". It is shown there that C-limited clauses and goals do not contribute to the exponential size of the search.

5 Upper Bound Analysis

The upper bound analysis will be based on finding an upper bound for the number of nodes in a Modoc search tree. To make the analysis tractable, we first define *collapsed* trees, C-limited variables, and unlimited variables (Section 5.1), and then find an upper bound on the number of *goal* nodes in the *collapsed* tree (Section 5.2 and 5.3).

Lemma 5.2 shows that the number of *goal* nodes in a Modoc search tree is at most a factor on n larger than the number of goal nodes in the corresponding collapsed tree. It is straightforward to see that the *total* number of nodes in a Modoc search tree is at most a factor of $(s+1)$ larger than the number of *goal* nodes in that search tree, where $(s+1) \leq n$ is the maximum number of literals in any clause. Finally, only polynomial time is spent in any node of a Modoc search tree. Therefore, the exponential growth rate found in Section 5.3 applies to the worst-case running time of Modoc.

5.1 Collapsed Trees

We now introduce some needed terminology to enable us to precisely describe the refutation search procedure of Modoc. In this discussion, line identifiers "trg" and "trc" refer to Figures 1 and 2.

Definition 5.1: (unlimited, C-limited, active, eligible, currently pure) These definitions are relative to a particular point in the refutation search, in that they depend on the current sets of ancestors, autarky literals, and C-literals.

An *active C-literal* is a C-literal that does not appear, either positively or negatively, in the current set of autarky literals.

An *unlimited variable* is a propositional variable of the original formula that does not appear, either positively or negatively, in any of the current sets of ancestors, autarky literals, and active C-literals. An *unlimited goal* is a goal whose literal contains an unlimited variable. A *C-limited goal* is a goal whose literal duplicates an active C-literal in the current set. A *C-limited clause* is one that contains a C-limited subgoal. An *unlimited clause* is one that contains only unlimited subgoals (possibly no subgoals). (Recall that *subgoals* of a clause are just the literals placed in the initial $\mathcal{S}$ at line trc-03.)

The term *unlimited autarky literal*, apparently a self-contradictory phrase, refers to a literal that is unlimited relative to the current set of autarky literals, but has just been returned by a child procedure as part of the set of additional autarky literals that it found. See $\mathcal{M}_C$ in line trg-08 of Figure 1, and $\mathcal{M}_g$ in line trc-06 of Figure 2.

An *active clause* is a clause none of whose literals occurs in the current ancestors, or in the current autarky. An *eligible clause* for literal q is an active clause that contains a literal $\neg q$. A *currently pure* literal is one for which there are no eligible clauses. □

Lemma 5.1: At a particular point in the search, every eligible clause is either C-limited or unlimited.

Proof: The only point that is not immediate from the definitions is that any eligible clause containing a literal that is complementary to an autarky literal also contains some *other* literal that is in the set of autarky literals. This is a property of conditional autarkies (see Definition 3.2). ∎

Key concepts for proof of worst-case efficiency are those of C-efficient trees, selected C-limited subgoals, and redundant subgoals, as defined next. Modoc never processes redundant subgoals, justifying the terminology, because it always processes the selected C-limited subgoal first (line trc-05 and Note 4). This subgoal either succeeds, producing a lemma-induced cut, or fails. (Recall that a goal "succeeds" by being refuted.) In either event, the redundant subgoals are not processed.

Definition 5.2: (C-efficient, selected, redundant) A *C-efficient* Modoc search tree is one in which every C-limited clause either has no subgoals, or has some subgoals, of which one is designated as the *selected C-limited subgoal*, and any others are designated as *redundant subgoals*, and do not appear as goal nodes. The selected C-limited subgoal may or may not be a leaf. □

Definition 5.3: (collapsed tree) A *collapsed tree* is a clause-goal tree that is derived from a Modoc search tree (see beginning of Section 4) as follows:

> Any path from an unlimited goal q to a C-limited clause C to a C-limited goal r to a clause D is collapsed into a single edge from goal q to clause D, eliminating the intervening nodes from the tree. Since every C-limited clause has at most one edge leaving it, and that edge goes to a C-limited subgoal, the collapsing process is well-defined.

Thus, the only C-limited goals in a collapsed tree are leaves and the only C-limited clauses remaining are immediately above such leaves. □

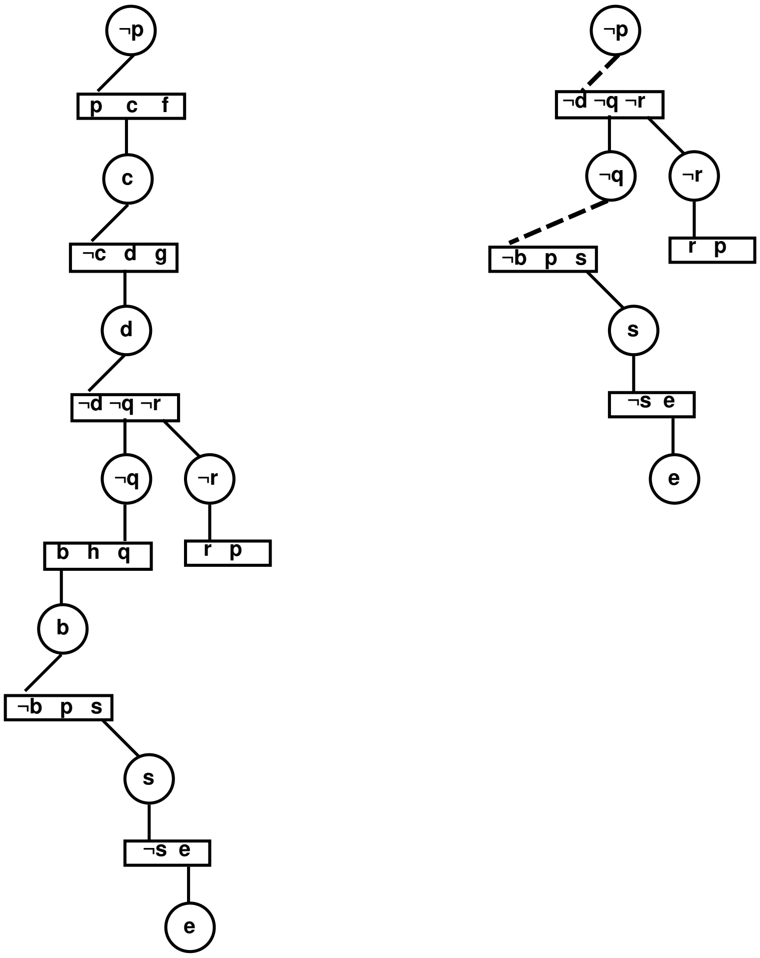

Figure 6: A fragment of a Modoc search tree (left), and the corresponding collapsed tree (right). See Example 5.1 for discussion.

Example 5.1: Consider the fragment of a Modoc search tree on the left of Figure 6. Assume that b, c, d, and e are active C-literals when this part of the tree was formed; that is, they are attached at upper levels of the tree, not shown. Observe that subgoals f, g, and h are redundant subgoals, per Definition 5.2.

The collapsed tree is shown on the right of that figure. Observe that the path from goal $\neg p$ to clause $[\neg d, \neg q, \neg r]$ collapsed to a single edge, and the path from goal $\neg q$ to clause $[\neg b, p, s]$ collapsed to a single edge. The C-limited clause $[\neg s, e]$ remains because e is a leaf. □

Roughly speaking, a propositional derivation tree (PDT) can be created from a Modoc search tree by removing all subtrees rooted at a failed clause node. In a nontrivial PDT each goal node has exactly one clause child, which either succeeded in refuting that goal node, or provides a path to a lower refutation that activated a lemma-induced cut. Collapsed PDTs can be defined analogously.

Lemma 5.2: The size of a C-efficient Modoc search tree, measured by the number of goal nodes, is at most $(\lambda + 1)$ times the size of its collapsed tree, where λ is the number of active C-literals.

Proof: All goal nodes (except $\top$) correspond to nonredundant subgoals of the clause parent. No goal literal can be repeated on any path in a Modoc search tree. Associate every deleted goal node with the closest goal ancestor that was not deleted. At most λ deleted nodes can be associated with any node remaining in the collapsed tree. ∎

5.2 Upper Bound Recurrences

This section analyzes the worst-case size of a collapsed tree (Definition 5.3), measured as the number of goal nodes, as a function of the number of *unlimited variables* (Definition 5.1) and the maximum clause width. The number of goal nodes may be called the *cost* of the collapsed tree or subtree in this section. Analysis of the collapsed tree is justified by Lemma 5.2.

In this section n denotes the number of unlimited variables, and s denotes the maximum number of subgoals of any clause below the top level, so $(s+1)$ is the maximum clause width. We shall seek upper bounds for the following quantities.

$E(n)$	cost of successful clause node
$F(n,j,c)$	cost of failed clause node in which c unlimited subgoals succeeded before the $(c+1)$-th failed, and in which the failed subgoal returned j unlimited autarky literals
$G(n)$	cost of successful goal node
$H(n,j)$	cost of failed goal node that returned j unlimited autarky literals

Subscripts on E, F, G, and H denote separate cases that obey different recurrence constraints, over which a maximum must be taken to obtain a recurrence constraint on the unsubscripted symbol.

First, consider recurrences for clause nodes, with clause C. A key observation, which will be used several times, is that each successful *unlimited* subgoal (call it $\neg r$) creates at least one new C-literal, namely r (line trc-07), which is returned (possibly through several layers of calls) to the procedure attempting to refute C. The point is that line trg-22 of any intervening calls will *not* remove the C-literal r because C-literals cannot depend on C-limited goal nodes (see Note 3 in Figure 1). (Line identifiers "trg" and "trc" refer to Figures 1 and 2.)

Success of a *C-limited* subgoal does *not* reduce the number of unlimited variables, which explains why we want to collapse them out of the collapsed tree, for analysis purposes.

Suppose C has b *unlimited* subgoals. As just argued, the C-literals returned by each successful *unlimited* subgoal reduce the number of unlimited variables for the remainder of the clause by at least one. Clearly, the worst case is $b = s$, so:

$$E(n) \quad \leq \quad \sum_{i=0}^{s-1} G(n-i) \tag{12}$$

For failing clauses, $F(n,j,c)$ applies to the case where C has $(c+1)$ or more unlimited subgoals, and c of them succeed before a subgoal fails. As before, each successful unlimited subgoal creates at least one new C-literal, which is returned to the procedure attempting to refute C, reducing the number of unlimited variables for the remainder of the clause. The failing literal must return at least one unlimited autarky literal, itself.

$$F(n,j,c) \quad \leq \quad \sum_{i=0}^{c-1} G(n-i) + H(n-c,j) \qquad (1 \leq j < n,\ 0 \leq c < s) \tag{13}$$

Note that the sum is empty when $c = 0$.

Now, consider goal nodes, with goal literal q. Let $G_{j,c}$ denote the case in which j unlimited autarky literals and c C-literals were returned by the *first* unlimited clause child of q (call it C), as it failed. Each successful unlimited subgoal of C creates at least one C-literal, which reduces the number of unlimited variables. At most c subgoals of C succeeded. Thus the remaining cost of that node is the same as if it began with $(n-(j+c))$ unlimited variables. Let G_0 denote the case in which the first clause succeeded.

$$G_0(n) \quad = \quad 1 + E(n-1) \tag{14}$$

$$G_{j,c}(n) \quad = \quad F(n-1,j,c) + G(n-(j+c)) \qquad (1 \leq j < n,\ 0 \leq c < s) \tag{15}$$

$$G(n) \quad \leq \quad \max\{G_0(n),\ G_{j,c}(n)\} \tag{16}$$

The maximum is over the terms that are within the subscripts' ranges.

If an unlimited goal node q fails returning just one autarky literal, that must be q itself. Otherwise, let $H_{m,c}$ denote the case in which m unlimited autarky literals and c C-literals were returned by the *first* unlimited clause child of q (call it C), as it failed. As argued for $G_{j,c}$, the remaining cost after the failure of one clause is the same as though node q began with $(n-(m+c))$ unlimited variables.

$$H(n,1) = 1 \tag{17}$$

$$H_{m,c}(n,j) = F(n-1,m,c) + H(n-(m+c), j-m) \qquad (1 \le m < j < n,\ 0 \le c < s) \tag{18}$$

$$H(n,j) \le \max\{H_{m,c}(n,j)\} \tag{19}$$

The maximum is over the terms that are within the subscripts' ranges.

Eliminating E and F from Eqs. 14–19, and using Eq. 17, gives:

$$G_0(n) \le 1 + \sum_{i=1}^{s} G(n-i) \tag{20}$$

$$G_{j,c}(n) \le \sum_{i=1}^{c} G(n-i) + H(n-1-c,j) + G(n-j-c) \qquad (1 \le j < n,\ 0 \le c < s) \tag{21}$$

$$G(n) \le \max\{G_0(n),\ G_{j,c}(n)\} \tag{22}$$

and:

$$H(n,1) = 1 \tag{23}$$

$$H_{m,c}(n,j) \le \sum_{i=1}^{c} G(n-i) + H(n-1-c,m) + H(n-m-c,j-m) \qquad (1 \le m < j < n,\ 0 \le c < s) \tag{24}$$

$$H(n,j) \le \max\{H_{m,c}(n,j)\} \tag{25}$$

5.3 Solution of Recurrences

There are no well established techniques for solving Eqs. 20–25. The approach we shall take is first to assume $G(n) = g\,\gamma^n$ and $H(n,j) = h\,\gamma^n$. However, the latter is an overestimate when j is small, and special cases are needed. Then we shall investigate constraints among the constants g, h, and γ that are sufficient to support a proof by induction of the assumed solution. Each combination of possible maxima in Eqs. 22 and 25 need to be investigated.

It is important to notice that the direction of the inequality *reverses* in going from a *given* constraint (involving E, F, G, and H) to a constraint that is *needed* for the proof (involving γ and possibly g and h). For example, Eq. 20 leads to the *needed* constraint:

$$\gamma^{s-1} + \gamma^{s-2} + \ldots + 1 \le \gamma^s \tag{26}$$

That is, suppose γ is chosen to satisfy Eq. 26. Then Eq. 20, Eq. 26, and the inductive hypothesis (that $G(m) \le g\,\gamma^m$ for all $1 \le m < n$) imply that $G_0(n) \le g\,\gamma^n$. Other parts of the induction proof follow a similar pattern. See also Section 2.

First, we can bound the range in which the solution value of γ lies for the system (20–25). It is easily verified that $\gamma = 2$, $g = h = 1$, satisfies all constraints. Also, Eq. 26 forces γ to be at least the golden ratio $1.618\ldots$ (larger for $k > 3$, see rightmost column of Figure 8).

Notice that Eq. 26 is the same as Eq. 2, as obtained by Monien and Speckenmeyer [MS85]. Therefore, the best we can hope for this analysis is to equal their bound. In fact, G_0 is not necessarily the maximum term in Eq. 22, and we do somewhat worse.

For $j \le 3$, at least one recursive H must be $O(1)$ in Eq. 24. Then, assuming G is an increasing function, $c = s-1$ maximizes the resulting expression, giving:

$$H(n,2) = \sum_{i=1}^{s-1} G(n-i) + 2 \tag{27}$$

Source	Constraint
G_0	$\frac{(\gamma^s-1)}{(\gamma-1)\gamma^s} \le 1$
$G_{2,c}$	$\frac{(\gamma^c-1)}{(\gamma-1)\gamma^c} + \frac{(\gamma^{s-1}-1)}{\gamma^{c+1}(\gamma-1)\gamma^{s-1}} + \frac{1}{\gamma^{c+2}} \le 1$
$G_{3,c}$	$\frac{(\gamma^c-1)}{(\gamma-1)\gamma^c} + \frac{(\gamma^{s-1}-1)}{\gamma^{c+1}(\gamma-1)\gamma^{s-1}} + \frac{(\gamma^{s-1}-1)}{\gamma^{s+c+1}(\gamma-1)\gamma^{s-1}} + \frac{1}{\gamma^{c+3}} \le 1$
$G_{4,c}$	$\frac{(\gamma^c-1)}{(\gamma-1)\gamma^c} + \frac{(\gamma^{s-1}-1)}{\gamma^{c+1}(\gamma-1)\gamma^{s-1}} + \frac{(\gamma^{s-1}-1)}{\gamma^{s+c+1}(\gamma-1)\gamma^{s-1}} + \frac{(\gamma^{s-1}-1)}{\gamma^{s+c+2}(\gamma-1)\gamma^{s-1}} + \frac{1}{\gamma^{c+4}} \le 1$
$G_{5,c}$	$\frac{g\,(\gamma^c-1)}{(\gamma-1)\gamma^c} + \frac{h}{\gamma^{c+1}} + \frac{g}{\gamma^{c+5}} \le g$
$H_{2,c}$	$\frac{g\,(\gamma^c-1)}{(\gamma-1)\gamma^c} + \frac{g\,(\gamma^{s-1}-1)}{\gamma^{c+1}(\gamma-1)\gamma^{s-1}} + \frac{h}{\gamma^{c+2}} \le h$
$H_{3,c}$	$\frac{g\,(\gamma^c-1)}{(\gamma-1)\gamma^c} + \frac{g\,(\gamma^{s-1}-1)}{\gamma^{c+1}(\gamma-1)\gamma^{s-1}} + \frac{g\,(\gamma^{s-1}-1)}{\gamma^{s+c+1}(\gamma-1)\gamma^{s-1}} + \frac{h}{\gamma^{c+3}} \le h$
$H_{4,c}$	$\frac{g\,(\gamma^c-1)}{(\gamma-1)\gamma^c} + \frac{g\,(\gamma^{s-1}-1)}{\gamma^{c+1}(\gamma-1)\gamma^{s-1}} + \frac{g\,(\gamma^{s-1}-1)}{\gamma^{s+c+1}(\gamma-1)\gamma^{s-1}} + \frac{g\,(\gamma^{s-1}-1)}{\gamma^{s+c+2}(\gamma-1)\gamma^{s-1}} + \frac{h}{\gamma^{c+4}} \le h$
$H_{5,c}$	$\frac{g\,(\gamma^c-1)}{(\gamma-1)\gamma^c} + \frac{h}{\gamma^{c+1}} + \frac{h}{\gamma^{c+5}} \le h$

Figure 7: Needed constraints on γ, g and h, as discussed in Section 5.3. The range for c is $0 \le c \le s-1$.

$$H(n,3) = \sum_{i=1}^{s-1} G(n-i) + \sum_{i=s+1}^{2s-1} G(n-i) + 3 \tag{28}$$

For larger j it is not immediate that the maximum occurs at $c = s-1$.

The case $j = 4$ is the first one for which both recursive H's may be large in Eq. 24. By inspection, $H_{2,c}(n,4) > H_{3,c}(n,4) = H_{1,c}(n,4)$. Also, for $\gamma \ge 1.618\ldots$, under the assumption that $G(n) = g\,\gamma^n$, inspection shows that $H_{2,c+1}(n,4) > H_{2,c}(n,4)$. Therefore, $c = s-1$ again provides the maximum, and

$$H(n,4) = \sum_{i=1}^{s-1} G(n-i) + \sum_{i=s+1}^{2s-1} G(n-i) + \sum_{i=s+2}^{2s} G(n-i) + 4 \tag{29}$$

Finally, for $j \ge 5$, we simply assume $H(n,j) = h\,\gamma^n$. This gives us a finite number of cases to consider, for each s. In Eq. 21 values of j that are greater than 5 will just produce "needed" constraints on γ that are weaker than those needed for $j = 5$. Similarly, in Eq. 24, we can assume j is large and consider values of m up to 5. Other combinations produce "needed" constraints weaker than those considered.

Needed constraints are summarized in Figure 7, using the identity:

$$\sum_{i=1}^{c} \gamma^{-i} = \frac{(\gamma^c-1)}{(\gamma-1)\gamma^c} \tag{30}$$

To determine whether all the constraints can be satisfied, we can collect the terms in the constraints involving g and h and isolate the ratio h/g. Consider the case $H_{5,1}(n,j)$ in Figure 7. Rearranging terms, we need:

$$\frac{\gamma^5}{\gamma^6 - \gamma^4 - 1} \le \frac{h}{g} \tag{31}$$

Similarly, for $G_{5,1}(n)$, we need:

$$\frac{h}{g} \le \frac{\gamma^4}{\gamma^6 - \gamma^5 - 1} \tag{32}$$

Other cases involving g and h are rearranged similarly. The terms h/g can be eliminated in the style of the Fourier elimination technique. The smallest γ that satisfies all the constraints is the solution. If no γ satisfies all the constraints, after h/g is eliminated, then the system is inconsistent.

Alternatively, one can guess a suitable value for h/g, then check whether the resulting constraints (now involving γ only) can be satisfied. As it turns out, choosing $h/g = 1$ permits all constraints to be satisfied, and reduces the number of distinct cases to be considered: $H_{m,c}$ become the same cases as $G_{m,c}$. Now, it turns out that none of the constraints involving h/g is "tight", so the solution is optimal. The tight constraints correspond to $G_{2,s-1}$.

However, if we choose a coarser approximation, assuming that $H(n,j) = h\gamma^n$ for $j \geq 4$ (instead of having a special case for $H(n,4)$), then some constraints involving h/g *are* "tight", and larger values of γ are obtained.

In Figure 7, it can be shown that if a particular inequality is satisfied by $\gamma > 1.6$ for c then it is also satisfied by $c-1$ in place of c. We are assuming $g = h$ here. Let $L(c) \leq 1$ denote a constraint on γ after $g = h$ is divided out. To show that this constraint implies $L(c-1) \leq 1$, the technique is to assume that $L(c-1) > 1$. But that would imply that $L(c) < L(c-1)$. However, in each case in Figure 7 the latter constraint is unsatisfiable for $\gamma > 1.6$. It follows that the bounds are dictated by the inequalities with $c = s - 1$.

Now, still working with $g = h$, consider the constraints for G_0, $G_{2,s-1}$, $G_{3,s-1}$, and $G_{4,s-1}$ in Figure 7. The same technique as above demonstrates that:

1. Constraint $G_{3,s-1}$ implies constraint $G_{4,s-1}$.
2. Constraint $G_{2,s-1}$ implies constraint $G_{3,s-1}$.
3. Constraint $G_{2,s-1}$ implies constraint G_0.

The situation with $G_{5,s-1}$ is more complicated, because $G_{2,s-1}$ does not imply it for all γ. However, we can show that the *smallest* $\gamma > 1$ that satisfies $G_{2,s-1}$ also satisfies $G_{5,s-1}$. For this smallest γ, constraint $G_{2,s-1}$ gives:

$$1 = \frac{(\gamma^{s-1}-1)}{(\gamma-1)\gamma^{s-1}} + \frac{(\gamma^{s-1}-1)}{\gamma^s(\gamma-1)\gamma^{s-1}} + \frac{1}{\gamma^{s+1}} \tag{33}$$

$$\frac{1}{\gamma^s} = \frac{(\gamma^{s-1}-1)}{\gamma^s(\gamma-1)\gamma^{s-1}} + \frac{(\gamma^{s-1}-1)}{\gamma^{2s}(\gamma-1)\gamma^{s-1}} + \frac{1}{\gamma^{2s+1}} \tag{34}$$

while constraint G_0 gives:

$$\frac{1}{\gamma^{s-1}} \geq \frac{(\gamma^s-1)}{\gamma^{s-1}(\gamma-1)\gamma^s} \tag{35}$$

Now assume that constraint $G_{5,s-1}$ is false:

$$\frac{(\gamma^{s-1}-1)}{(\gamma-1)\gamma^{s-1}} + \frac{1}{\gamma^s} + \frac{1}{\gamma^{s+4}} > 1 \tag{36}$$

Substituting Eq. 33 and 34 into Eq. 36, then combining with Eq. 35:

$$\frac{1}{\gamma^3} + \frac{1}{\gamma^s} + \frac{1}{\gamma^{s-1}} > 1 + \frac{1}{\gamma^{2s-1}} \tag{37}$$

But, for $\gamma \geq 1.618\ldots$, the golden ratio, and $s \geq 2$, this inequality is unsatisfiable. Therefore, γ_{s+1} is governed by constraint $G_{2,s-1}$.

Calculated values for γ for clause widths 3–6 are given in Figure 8, with the corresponding values obtained by Monien and Speckenmeyer, as well as the original DPLL. The crucial constraint for each case is also given. Comparison with Eq. 8 shows that γ_3 for Modoc is the same as that for the original DPLL. However, for $k \geq 4$ the DPLL bounds are stronger.

	Critical Constraint			Monien and	Original
	Case	Equation	Modoc	Speckenmeyer	DPLL
γ_3	$G_{2,1}$	$\gamma^3 - \gamma^2 - 2 = 0$	1.695621	1.618034	1.695621
γ_4	$G_{2,2}$	$\gamma^5 - \gamma^4 - \gamma^3 - 2\gamma - 1 = 0$	1.897180	1.839287	1.853561
γ_5	$G_{2,3}$	$\gamma^7 - \gamma^6 - \gamma^5 - \gamma^4 - 2\gamma^2 - \gamma - 1 = 0$	1.959098	1.927562	1.930404
γ_6	$G_{2,4}$	$\gamma^9 - \gamma^8 - \gamma^7 - \gamma^6 - \gamma^5 - 2\gamma^3 - \gamma^2 - \gamma - 1 = 0$	1.981974	1.965948	1.966560

Figure 8: Values of γ_3 through γ_6 achievable by Modoc, compared to those obtained by Monien and Speckenmeyer [MS85], as well as the original DPLL (Section 2).

5.4 Model Elimination Upper Bound

Now we show informally why model elimination, essentially Modoc without autarky pruning, has no reasonable upper bound, not even 2^n. Consider Eqs. 24 and 25 with m and j equal to 0, corresponding to the assumption that there are no autarky literals. The case $c = 0$ is possible, leading (apparently) to nonconvergence: $H(n, 0) = F(n-1, 0, 0) + H(n, 0)$. Actually the program iterates through all the eligible clauses, $\mathcal{E}$, and $F(n-1, 0, 0) \geq H(n-1, 0)$, so we get

$$H(n,0) \quad \geq \quad |\mathcal{E}| H(n-1,0) \qquad \text{(autarky pruning disabled)} \tag{38}$$

A value of 6 for $|\mathcal{E}|$ is not at all unusual, giving $H(n, 0) \geq \Omega(6^n)$, and this is not even the worst case.

6 Conclusions and Future Work

We have shown a worst-case upper bound on the time complexity of Modoc, as a function of the number of variables and the width of the widest clause of the formula. This is the first nontrivial upper bound to be shown for any propositional resolution method. However, stronger bounds have been shown for model-search methods. We have also presented the first nontrivial upper bound for the original DPLL algorithm, and shown that a simple refinement of its splitting rule improves the bound substantially.

Section 5.4 discussed informally why Model Elimination does not satisfy a similar upper bound. The problem was that failed refutation sub-searches produce no information, and therefore permit high search redundancy when all searches fail. Modoc addresses this shortcoming by returning autarky literals from failed sub-searches, which literals prune subsequent searching.

Future algorithmic work should proceed along several directions, including heuristics for guiding the resolution search, further improvements to lemma caching, and an extension to first-order theorem proving.

Future analysis work may address several questions. Can the upper bound can be improved by showing that many clauses do not actually have s subgoals, as the analysis assumed? Can an interesting upper bound be shown that is a function of the total number of literals in the formula, which is closer to the traditional measure of problem size for complexity theory?

Acknowledgments

We thank the anonymous referees for their careful reading of the paper and helpful suggestions on improving the presentation. This work was supported in part by NSF grant CCR-95-03830, by equipment donations from Sun Microsystems, Inc., and software donations from Quintus Computer Systems, Inc.

References

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

[GS96] 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.

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

[Kul94] O. Kullmann. Methods for 3-SAT-decision in less than 1.5045^n steps. Technical report, University of Frankfurt, 1994.

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

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

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

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

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

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

[Sch92] I. Schiermeyer. Solving 3-satisfiability in less than 1.579^n steps. In *6th Workshop on Computer Science Logic*, pages 379–94, Berlin, Germany, 1992. Springer-Verlag.

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

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

[VGK96] A. Van Gelder and F. Kamiya. The partial rehabilitation of propositional resolution. Technical Report UCSC–CRL–96–04, UC Santa Cruz, Santa Cruz, CA., 1996.