UNIVERSITY OF CALIFORNIA
SANTA CRUZ

PROPOSITIONAL THEOREM PROVING: ADVANCED LEMMA STRATEGIES AND MULTI-AGENT SEARCH

A dissertation submitted in partial satisfaction
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Fumiaki Okushi

September 1998

The dissertation of Fumiaki Okushi
is approved:

Professor Allen Van Gelder, Chair

Professor Phokion Kolaitis

Professor Richard Hughey

Dean of Graduate Studies

Contents

List of Figures

List of Tables

Propositional Theorem Proving: Advanced Lemma Strategies and Multi-Agent Search

Fumiaki Okushi

Abstract

The Satisfiability Problem is to decide whether or not a boolean CNF formula has a satisfying truth assignment. Its acceptance problem is NP-complete, suggesting that it is unlikely for a polynomial-time algorithm to be found. However, because of its importance and applications in areas such as circuit design, finite mathematics, and planning, many practical algorithms have been introduced. Among such algorithms is *Modoc*. Modoc extends propositional Model Elimination with a mechanism to prune away certain subrefutation attempts that cannot succeed. The pruning information is encoded in a partial truth assignment called an *autarky*. As a descendent of Model Elimination, Modoc also includes a mechanism to record successful subrefutations as *lemmas* and recall them as necessary. The exact mechanism follows that of C-literals.

The results contained in this dissertation are presented in three parts. The first part describes a formula-simplification scheme suitable for backward-chaining propositional theorem provers, such as Modoc. The scheme preserves satisfiability, models, and theorem clauses.

The second part describes various enhancements made to (basic) Modoc. The *quasi-persistent lemma* strategy improves upon the C-literal strategy and may retain lemmas longer. The *eager lemma* strategy derives certain lemmas early. In certain cases, articulation points in a graph implicitly constructed during an eager-lemma derivation may derive additional lemmas. *Lemma-induced cuts* and *C-reduction-induced cuts* allow a subrefutation attempt to be completed by a short alternate proof.

The third part describes *Parallel Modoc*. Parallel Modoc is a multi-agent search procedure that uses (enhanced) Modoc as search agents. Agents communicate new autarkies and lemmas as they are found. Combining autarkies may not be straightforward because two autarkies found by two separate agents may have conflicting assignments. This part presents an algorithm that combines two arbitrary autarkies to form another autarky that is no smaller than the first two autarkies.

Experimental results show that enhanced Modoc outperforms many model-search procedures on formulas derived from applications. Parallel Modoc often achieves speedup greater than the number of agents. Formulas that could not be solved in an hour by Modoc were often solved by Parallel Modoc in the order of minutes, and in some cases, in seconds.

To my friends.

Acknowledgements

The author wishes to thank his advisor Allen Van Gelder for his advisement and support; he has taken time to discuss in detail various aspects of satisfiability testing and has also provided necessary support in the form of research assistantship and for travel to conferences. The author also wishes to thank everyone who served on his committees—Allen Van Gelder, Phokion Kolaitis, Richard Hughey (Computer Engineering Department), Debra Lewis (Mathematics Department), and Darrell Long. All five served on the Qualifying Examination Committee, and the first three have also agreed to serve on the Reading Committee.

Formulas used during the author's work were obtained from several sources. Most of the planning formulas were generated using two "SAT compilers"—Satplan, developed by Henry Kautz and Bart Selman (AT&T Bell Laboratories), and Medic, developed by Michael Ernst, Todd Millstein, and Daniel Weld (University of Washington). Random formulas were generated using `mkcnf` developed by Yumi Tsuji (University of California, Santa Cruz). The circuit-diagnosis formulas were obtained from Tracy Larrabee's research group at the University of California, Santa Cruz.

Satisfiability testers other than `modoc` and `pmodoc` were provided by their respective authors—`walksat`, by Henry Kautz and Bart Selman (AT&T Bell Laboratories), `C-A`, by James Crawford (University of Oregon) and Larry Auton (AT&T Bell Laboratories), and `dpll` and `2cl`, by Allen Van Gelder and Yumi Tsuji (University of California, Santa Cruz).

Work reported in this dissertation was supported in part by NSF grant CCR-95-03830.

Chapter 1

Introduction

The Satisfiability Problem is the problem of deciding whether or not a boolean formula in conjunctive normal form has a truth assignment that makes the formula true. Its acceptance problem is NP-complete [11, 19], suggesting that it is unlikely for a polynomial-time algorithm to be found. However, because of its importance and applications in areas such as circuit design [29], finite mathematics [18, 49, 51, 50], and planning [27, 28, 15], many practical algorithms have been introduced.

Most of the algorithms found for the satisfiability problem can be classified into one of two classes of procedures—the class of *model-search* procedure and the class of *refutation-search* procedures. A model-search procedure tries to show that the formula is satisfiable by finding a satisfying truth assignment. If one is found, the procedure announces that the formula is satisfiable. If the procedure is “complete”, meaning that the procedure is guaranteed to find a satisfying truth assignment for every satisfiable formula, the procedure may conclude that the formula is unsatisfiable after it fails to find a satisfying truth assignment.

Many of the model-search procedures are descendents of the Davis-Putnam-Loveland-Logemann (DPLL) algorithm [14, 13]. The DPLL algorithm consists of three rules—the *unit-clause rule,* the *pure-literal rule,* and the *splitting rule*. The unit-clause rule makes literals that occur in clauses of length one true. The pure-literal rule makes literals whose complements do not appear in the formula true. The splitting rule takes a variable and creates two formulas, one resulting from making the variable true, and another resulting from making the variable false, and considers the two formulas for satisfiability.

A different approach to model search is to formulate the problem as an integer linear-programming problem [5, 26, 23, 21]. This is accomplished by translating boolean constraints into inequalities. Various techniques developed for integer linear programming can then be applied.

Several model-search procedures employ a non-systematic approach, often involving stochastic choices [35, 39, 28]. These procedures are necessarily “incomplete”, meaning that there is no guarantee that the procedure will find a satisfying truth assignment for every satisfiable formula. On unsatisfiable formulas, these procedures can run for ever. Because of these potential problems, these procedures are usually run with resource limits, such as CPU time and/or the number of guesses. If the procedure fails to find a satisfying truth assignment within the given resource limits, it abandons the search. In this case, no sure information is obtained. Despite their shortcomings, stochastic model-search procedures have been successful in finding satisfying truth assignments for satisfiable formulas that are much larger than what the current complete methods can generally handle.

A refutation-search procedure, on the other hand, tries to show that the formula is unsatisfiable by finding a refutation proof. A refutation proof shows that the formula is inconsistent, i.e., it has no satisfying truth assignment. If a refutation proof is found, the procedure announces that the formula is unsatisfiable. If the procedure is “complete”, meaning, in this case, that the procedure is guaranteed to find a refutation proof for every unsatisfiable formula, the procedure may conclude that the formula is satisfiable after it fails to find a refutation proof. An example of a refutation-search procedure is Model Elimination [32].

One class of algorithms that do not fall into either of the two classes of algorithms previously mentioned is the *counting method* [24, 25, 42]. A counting method tries to obtain a bound on the number of satisfying truth assignments. If a non-zero lower bound is obtained, the algorithm concludes that the formula is satisfiable. If a zero upper bound is obtained, the algorithm concludes that the formula is unsatisfiable.

Modoc [43] is a refutation-search procedure based on propositional Model Elimination. It extends Model Elimination with a mechanism to prune away certain subrefutation attempts that cannot succeed. The pruning information is encoded in a partial truth assignment called an *autarky* [36]. As a descendent of Model Elimination, Modoc also includes a mechanism to record successful subrefutations as *lemmas* and recall them as necessary.

1.1 Summary of Results

The results contained in this dissertation are presented in three parts. The first part describes *goal-sensitive simplification*. Compared to traditional simplification, goal-sensitive simplification is designed specifically for use with a backward-chaining propositional theorem prover, such as Modoc. It preserves satisfiability, models, and the clauses that describe the negated conclusion of the theorem (which we will call the *theorem clauses*) across simplification. The second part describes various enhancements made to the basic design of Modoc. The enhancements include improving upon the original lemma strategy employed in Modoc, other opportunities to derive lemmas, and strategies to accelerate certain subrefutation attempts. The third part describes *Parallel Modoc*. Parallel Modoc is a multi-agent search procedure that uses (enhanced) Modoc as search agents. Modoc agents *cooperate* by communicating lemmas and autarkies as they are found. When multiple Modoc agents search for refutations, it is possible for them to derive autarkies that have conflicting assignments. Because of this, combining autarkies found by different Modoc agents may not be straightforward. This part presents properties found concerning multiple autarkies and an algorithm to combine two arbitrary autarkies to form another autarky that is no smaller than the first two autarkies.

1.2 Organization

The remainder of the dissertation is organized as follows. Chapters 2 and 3 cover background material. More specifically, Chapter 2 reviews previous work related to Modoc and Chapter 3 reviews the idea of planning as satisfiability testing. Chapter 4 summarizes the results contained in this dissertation. Chapters 5, 6, and 7 provide detailed discussion of the results. More specifically, Chapter 5 describes goal-sensitive simplification, Chapter 6 describes various enhancements made to the basic design of Modoc, and Chapter 7 describes Parallel Modoc. Conclusions and future research directions are summarized in Chapter 8.

Parts of the results were presented at the Workshop on Model-Based Automated Reasoning (in connection with IJCAI-97, August 23, 1997, Nagoya, Japan) [45], and at the Fifth International Symposium on Artificial Intelligence and Mathematics (January 4–6, 1998, Fort Lauderdale, Florida) [46, 47, 37]. More specifically, parts of Chapter 5 were presented in [45] and in [47], parts of Chapter 6 were presented in [46], and parts of Chapter 7 were presented in [37].

In the next section, we standardize the terminologies and the notations used throughout the dissertation.

1.3 Terminologies and Notations

This section defines terminologies and notations used throughout the dissertation. The terminologies and notations follow standard use in the computing literature, with the possible exception of the followings:

- Use of set notation to express clauses, formulas, and truth assignments. (See Notations 1.2 and 1.3.)
- Not requiring a satisfying truth assignment to be a total function. (See Definition 1.8.)
- Notation for strengthening. (See Definition 1.9.)

This dissertation is concerned with the algorithmic means to solve the Satisfiability Problem, which is defined below.

Definition 1.1 (satisfiability problem) The *Satisfiability Problem* is, given a boolean formula in *conjunctive normal form* (CNF), decide whether or not the formula has a *satisfying truth assignment*. □

Formally, the problem is stated as a decision problem, that is, to obtain either a "yes" or a "no" answer. However, in practice, we are often interested in a satisfying truth assignment should the formula turn out to be satisfiable. For example, for the planning formulas used in Sections 6.10

and 7.5, each satisfying truth assignment encodes a successful plan. Thus, to obtain a successful plan, one needs to obtain a satisfying truth assignment first.

We now elaborate on the elements composing the satisfiability problem. We begin with the formula. It is assumed that some arbitrary set of (boolean) variables is defined.

Definition 1.2 (literal, clause, CNF formula) A *literal* is either a variable or its negation. A *clause* is a disjunction of literals. A *CNF formula* is a conjunction of clauses. □

In this dissertation, we are only concerned with CNF formulas. Thus, the mention of the word "formula" in this dissertation will always imply a *CNF* formula. Further, we assume that a formula satisfies the following two conditions:

1. All the clauses are *non-tautologous*. That is, no clause contains complementary literals (Definition 1.4).

2. All the clauses are *non-redundant*. That is, no clause contains more than one copy of the same literal.

A formula that does not meet these conditions can be transformed into a logically-equivalent (Definition 1.10) formula that satisfies these conditions using a linear-time and -space algorithm.

One class of formulas that will be used in testing the performance of Modoc and other satisfiability testers is the class of k-CNF formulas, which is defined below.

Definition 1.3 (k-CNF formula) A *k-CNF formula* is a CNF formula whose clauses contain exactly k literals. □

Notation 1.1 (negation, complement) The symbol "$\neg$" will be used to denote negation of a variable. It is also used to denote the complement of a literal. Double negation "$\neg\neg$" is removed, as usual. □

Notation 1.2 (clause, formula) Throughout this dissertation, formulas and clauses will be expressed using set notation; for clarity, clauses will use square brackets ("[" and "]") instead of the usual curly brackets ("{" and "}"). □

Example 1.1 This example illustrates the use of notations introduced in Notations 1.1 and 1.2. Consider the following boolean formula in conjunctive normal form:

$$(a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee c) \wedge (\neg a \vee \neg c) \wedge (\neg a \vee \neg b \vee d) \wedge (\neg a \vee b \vee \neg d).$$

In this dissertation, the above formula will be expressed using set notation, as

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

□

As extreme cases, *empty clauses* and *empty formulas* are considered. An empty clause is a clause that contains no literals. An empty formula is a formula that contains no clauses.

Certain clauses and literals are often of interest to a satisfiability testing algorithm. Appropriate terminologies are introduced below to refer to such clauses and literals.

Definition 1.4 (unit clause, pure literal, complementary literals) A clause is called a *unit clause* if it contains exactly one literal. A literal x is a *pure literal* in formula F if $\neg x$ does not appear in F. Literals x and $\neg x$ are called *complementary literals*. □

We now elaborate on the other element composing the satisfiability problem, that is, the satisfying truth assignment.

Definition 1.5 (partial truth assignment) A *partial truth assignment* is a partial function from the set of variables to the boolean set. To elaborate, the word "partial" implies that a variable may or may not have a boolean value associated to it. □

Notation 1.3 (partial truth assignment) Throughout this dissertation, a partial truth assignment will be expressed using set notation, as the exact set of literals that are true under the partial truth assignment. □

Example 1.2 Consider a partial truth assignment ν that is defined by

$$\nu(x) = \begin{cases} \mathsf{true} & \text{if } x = a \text{ or } x = c \\ \mathsf{false} & \text{if } x = b \\ \text{undefined} & \text{otherwise} \end{cases}.$$

In this dissertation, ν will be expressed using set notation as

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

□

Definition 1.6 (satisfied clause, satisfied formula) A clause is said to be *satisfied* by a partial truth assignment if *at least one* of its literals is true under the partial truth assignment. A formula is said to be *satisfied* by a partial truth assignment if *all* its clauses are satisfied by the partial truth assignment. □

Definition 1.7 (satisfiable, unsatisfiable) A clause (formula) is *satisfiable* if there is a partial truth assignment that satisfies the clause (formula). A clause (formula) is *unsatisfiable* if *no* partial truth assignment can satisfy the clause (formula). □

By Definitions 1.6 and 1.7, the satisfiability of an empty clause and that of an empty formula become obvious.

Corollary 1.1 An empty clause is unsatisfiable. An empty formula is satisfiable. □

Definition 1.8 (satisfying truth assignment) A *satisfying truth assignment* is a partial truth assignment that satisfies the formula. That is, for each clause in the formula, there is at least one literal that is true under the truth assignment. □

Note that by Definition 1.8, a satisfying truth assignment is not required to be a total function in this dissertation. This may appear to deviate from the traditional definition of a satisfying truth assignment being a total function. However, this definition will not change the satisfiability of any formula. In particular, if a formula has a satisfying truth assignment in the sense of Definition 1.8 above, that satisfying truth assignment can be augmented to be a total function (and still satisfy the formula) by simply assigning arbitrary boolean values to the variables that are unassigned.

Because of Definition 1.8, there is no need to distinguish total truth assignments from partial truth assignments in this dissertation. Thus, the mention of the words “truth assignment” in this dissertation will always imply a *partial* truth assignment.

One benefit of using set notation to express clauses, formulas, and truth assignments is that testing whether a truth assignment satisfies a clause or a formula becomes a simple matter of finding a common element. A clause is satisfied by a partial truth assignment if and only if there is a common literal between the clause and the truth assignment. A formula is satisfied by a partial truth assignment if and only if each clause has a literal in common with the truth assignment.

An operation used often in a satisfiability testing algorithm is that of *strengthening*. Intuitively, strengthening takes a formula and a partial truth assignment, and removes satisfied clauses and false literals from the formula.

Definition 1.9 (strengthened clause, strengthened formula) Let A be a partial truth assignment, C be a clause, and F be a formula. Then, the *strengthened clause* of C by A, denoted by $C|_A$, is defined by

$$C|_A = \{q \in C \,|\, q \notin A\},$$

and the *strengthened formula* of F by A, denoted by $F|_A$, is defined by

$$F|_A = \{C|_A \,|\, C \in F \text{ and } C \cap A = \emptyset\}.$$

□

Note that the two formulas, before and after strengthening, do not necessarily share satisfiability. That is, it is possible for $F|_A$ to be unsatisfiable, yet for F to be satisfiable. (However, if $F|_A$ is satisfiable, then so is F.)

Some authors use the term “simplification” to refer to the operation of strengthening. However, in this dissertation, that term is reserved to refer to operations that make a formula “smaller” while preserving satisfiability (and possibly other attributes). (See Section 5 for a discussion on simplification.)

In below, we introduce other terminologies that originate from logic. Because our concern is with propositional satisfiability, the definitions below are specialized for the propositional domain.

Definition 1.10 (logically follows, logically equivalent) Let F_1 and F_2 be two formulas. Then, F_2 is said to *logically follow* F_1, denoted by $F_1 \models F_2$, if any satisfying truth assignment of F_1 is also a satisfying truth assignment of F_2. Also, F_1 and F_2 are said to be *logically equivalent*, denoted by $F_1 \equiv F_2$, if $F_1 \models F_2$ and $F_2 \models F_1$. $\square$

Note that the above definition may be extended to include clauses, by viewing clauses as formulas with a single clause.

Definition 1.11 (subsumed) Let C_1 and C_2 be two clauses. Then, C_2 is said to be *subsumed* by C_1 (or that C_1 subsumes C_2) if $C_1 \models C_2$. $\square$

In the propositional domain, and under our convention of using set notation to express clauses, subsumption becomes equivalent to set containment. That is, C_2 is subsumed by C_1 if and only if C_2 contains C_1.

The implication of subsumed clauses in the input formula is that these clauses may safely be removed from the input formula as they provide no additional constraints.

Deduction is the act of inferring new information from known facts. One of the well-known inference rules is the *resolution* [38], which we define below.

Definition 1.12 (resolution, resolvent) Let two clauses C_1 and C_2 be as follows:

$$\begin{aligned} C_1 &= [x, y_1, \ldots, y_m]; \\ C_2 &= [\neg x, z_1, \ldots, z_n]. \end{aligned}$$

That is, they have a common variable which occurs in the opposite polarity, more specifically, that x occurs in C_1 and $\neg x$ occurs in C_2. Define C_3 as follows:

$$C_3 = [y_1, \ldots, y_m, z_1, \ldots, z_n].$$

Then, C_3 is called a *resolvent* of C_1 and C_2, or that it was obtained by *resolution* using C_1 and C_2. $\square$

The use of resolution is supported by the following lemma [38], which we quote without proof.

Lemma 1.1 Resolution is a sound inference rule. That is, a resolvent logically follows the two input clauses. $\square$

Chapter 2

Background: Related Works

This chapter and the next review the background material. The purpose of this chapter is to describe the basic design of Modoc. Section 2.1 briefly describes Model Elimination, which provides the foundation for Modoc, Section 2.2 describes autarky, which is used in Modoc to encode certain pruning information, and finally, Section 2.3 describes Modoc.

2.1 Model Elimination

This section informally describes *Model Elimination*, a proof procedure introduced by Loveland [32]. Model Elimination provides the foundation for Modoc, which will be described in Section 2.3. Because the dissertation is concerned with propositional satisfiability, the description of Model Elimination contained in this section is specialized for the propositional domain. A detailed description of the procedure, including its correctness proofs, can be found elsewhere [32, 33].

The main idea of Model Elimination is to show that the set of clauses (i.e., a CNF formula) is inconsistent by iteratively eliminating all possible models. (For our purpose, "model" is another term for "satisfiable truth assignment".) If and when all possible models are eliminated, the procedure concludes that the formula is inconsistent, i.e., unsatisfiable. Since Model Elimination is a "complete" procedure (page 2), if such an attempt fails, the procedure concludes that the formula has a model, i.e., is satisfiable.

Currently, there are two ways to represent the current state of search in Model Elimination.

One uses a linear sequence of literals called a *chain*, and another uses a tree of literals called a *clause tree*. Section 2.1.1 describes Model Elimination using chains, and Section 2.1.2 describes Model Elimination using clause trees.

To avoid repeating subrefutation attempts that succeed, Model Elimination has a mechanism to record successful subrefutations as *lemmas* and recall them as necessary. The original implementation of lemmas was to record them as clauses [32]; this is described in Section 2.1.1. A different implementation called *C-literals* embeds special literals in chains [40]; this is described in Section 2.1.3.

2.1.1 Model Elimination on Chains

The original description of Model Elimination used a linear sequence of literals called a *chain* to represent the current state of search [32]. For the sake of illustration, we will assume a chain to grow from left to right. The literals in a chain are classified into being either *A-literals* or *B-literals*, and the chain is modified at the right end using three basic operations—*extension*, *reduction*, and *contraction*. The given formula is inconsistent if and only if an empty chain can be derived using the three operations, starting from a chain that is the literals of a clause in the formula in some order.

Not all chains are of interest to Model Elimination. The notion of *preadmissibility* and *admissibility* defines the chains that are of interest to the proof procedure. Any derived chain that is not preadmissible is simply discarded.

Definition 2.1 (preadmissible, admissible) A chain is *preadmissible* if the following conditions are met [32]:

- A pair of complementary B-literals are separated by an A-literal.
- If a literal occurs twice, once as an A-literal and again as a B-literal, then the A-literal must occur to the right of the B-literal.
- No two A-literals use the same variable.

A chain is *admissible* if it is preadmissible and the rightmost literal is a B-literal. □

b	a			(1)	Initial chain from clause $[a,b]$.
b	$\underline{a}$	c		(2)	Extension with clause $[\neg a,c]$.
b	$\underline{a}$	$\underline{c}$	$\neg a$	(3)	Extension with clause $[\neg a,\neg c]$.
b	$\underline{a}$	$\underline{c}$		(4)	Reduction of $\neg a$ with $\underline{a}$.
b	$\underline{a}$			(5)	Contraction of $\underline{c}$.
b				(6)	Contraction of $\underline{a}$.
$\underline{b}$	a			(7)	Extension with clause $[a,\neg b]$.
$\underline{b}$	$\underline{a}$	c		(8)	Extension with clause $[\neg a,c]$.
$\underline{b}$	$\underline{a}$	$\underline{c}$	$\neg a$	(9)	Extension with clause $[\neg a,\neg c]$.
$\underline{b}$	$\underline{a}$	$\underline{c}$		(10)	Reduction of $\neg a$ with $\underline{a}$.
$\underline{b}$	$\underline{a}$			(11)	Contraction of $\underline{c}$.
$\underline{b}$				(12)	Contraction of $\underline{a}$.
$\square$				(13)	Contraction of $\underline{b}$.

Figure 2.1: An execution of Model Elimination on chains. The formula is $\{[a,b], [a,\neg b], [\neg a,c], [\neg a,\neg c], [\neg a,\neg b,d], [\neg a,b,\neg d]\}$. An underline indicates that the literal is an A-literal; all other literals are B-literals. An empty chain is denoted by $\square$; this indicates that the formula is unsatisfiable.

The *extension* operation is the operation that lengthens a chain. It takes an admissible chain, whose rightmost literal is x, and a clause, that contains $\neg x$. The extension operation then reclassifies the x in the chain into an A-literal (by definition of admissibility (Definition 2.1), this literal was previously a B-literal), and attaches all the literals in the clause, except $\neg x$, to the right end of the chain in some order.

There are two operations that shortens a chain—reduction and contraction. The *reduction* operation takes an admissible chain that contains a pair of complementary literals x, which is a B-literal, and $\neg x$, which is an A-literal that occurs to the left of the x. The reduction operation then modifies the chain by removing the x from the chain. The *contraction* operation takes a preadmissible chain and removes the rightmost A-literal. Note that by repeatedly executing the contraction operation, a preadmissible chain will eventually become an admissible chain.

Example 2.1 This example illustrates the operations of Model Elimination on chains. In particular, it shows how Model Elimination may conclude that a given formula is inconsistent. Figure 2.1 shows the execution of Model Elimination on the formula $\{[a,b], [a,\neg b], [\neg a,c], [\neg a,\neg c], [\neg a,\neg b,d], [\neg a,b,\neg d]\}$. Each step is explained alongside the chains in the figure. $\square$

Note that in Figure 2.1, steps (8) through (12) are exactly the same as steps (2) through (6).

b	a			(1)	Initial chain from clause $[a, b]$.
b	$\underline{a}$	c		(2)	Extension with clause $[\neg a, c]$.
b	$\underline{a}$	$\underline{c}$	$\neg a$	(3)	Extension with clause $[\neg a, \neg c]$.
b	$\underline{a}$	$\underline{c}$		(4)	Reduction of $\neg a$ with $\underline{a}$.
b	$\underline{a}$			(*5*)	Contraction of $\underline{c}$. Derives a lemma clause $[\neg a, \neg c]$, which is already in the formula.
b				(*6*)	Contraction of $\underline{a}$. Derives a lemma clause $[\neg a]$.
$\underline{b}$	a			(7)	Extension with clause $[a, \neg b]$.
$\underline{b}$	$\underline{a}$			(*8*)	Extension with lemma clause $[\neg a]$.
$\underline{b}$				(*12*)	Contraction of $\underline{a}$. Derives a lemma clause $[\neg a]$, which is already derived.
$\square$				(*13*)	Contraction of $\underline{b}$. Derives a lemma clause $[\neg b]$.

Figure 2.2: An execution of Model Elimination with lemmas on chains. This shows how the addition of lemmas may change the execution shown in Figure 2.1. Numbers indicate the corresponding steps in Figure 2.1. Steps numbered in *italics* show the changes. In particular, the contraction operation derives a lemma clause $[\neg a]$ in step (*6*), and the lemma clause $[\neg a]$ is used for extension in step (*8*). This allows steps (9) through (11) to be skipped. An underline indicates that the literal is an A-literal; all other literals are B-literals. An empty chain is denoted by $\square$; this indicates that the formula is unsatisfiable.

What had happened was that two exact same subrefutations were made for two occurrences of the same literal a. To avoid such duplication of efforts, Model Elimination has a mechanism to record successful subrefutations as *lemmas* and recall them as necessary. In the preceding example, when the A-literal $\underline{a}$ was removed at step (6), meaning that literal a was successfully refuted, the procedure could have recorded this fact by adding a new clause $[\neg a]$ to the set of clauses. This would have allowed the proof to be completed in a fewer number of steps, as shown in Figure 2.2.

Example 2.2 This example illustrates the derivation and use of lemmas in Model Elimination. In particular, it continues from Example 2.1 and shows how the execution shown in Figure 2.1 may change with the addition of lemmas. The execution is shown in Figure 2.2. Each step is explained alongside the chains, and further explanation is provided in the caption. $\square$

A lemma is an implication that records what set of literals allows a literal to logically follow (Definition 1.10). In Model Elimination, a lemma is derived when an A-literal is removed by a contraction operation. The original lemma strategy of Model Elimination recorded lemmas as clauses and added them to the input formula. A different mechanism to record lemmas, called *C-literals*, will be described in Section 2.1.3.

An early implementation by Loveland et al. showed that the use of lemmas was generally "detrimental" [17]. However, later studies by Loveland [3] and others [40, 46] showed that a judicious use of lemmas is actually beneficial.

2.1.2 Model Elimination on Clause Trees

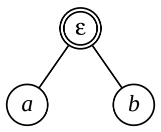
One disadvantage of the chain representation is that a chain can be extended only at the rightmost literal. This is an overly strict restriction since the addition of literals from the extension clause could be in any order. To eliminate this restriction, Minker and Zanon proposed the use of a tree structure, called *clause trees*, to represent the current state of search [34]. This allowed Model Elimination to extend any B-literal in the derivation.

Using this representation, the given formula is inconsistent if and only if an empty clause tree can be derived. A clause tree is a tree of literals whose root is labeled with a special symbol ε. Each node is classified into being either an A-literal or a B-literal. Initially, the tree consists only of ε, which is classified as a B-literal.

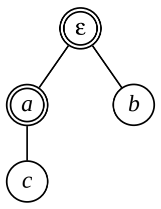
The three Model Elimination operations function similarly, except now on clause trees. The extension operation takes a clause tree, that contains a B-literal x, and a clause, that contains $\neg x$, reclassifies the x into an A-literal, and attaches each literal in the clause, except $\neg x$, as child nodes of x. The reduction operation takes a clause tree that contains a pair of complementary literals x and $\neg x$, where x is a B-literal and $\neg x$ is an A-literal that is also an ancestor of x, and removes the x from the clause tree. The contraction operation takes a clause tree and removes a leaf A-literal.

Example 2.3 This example illustrates the operations of Model Elimination on clause trees. In particular, it shows how Model Elimination may conclude that a given formula is inconsistent. Figure 2.3 shows the execution of Model Elimination on the formula $\{[a,b], [a,\neg b], [\neg a,c], [\neg a,\neg c], [\neg a,\neg b,d], [\neg a,b,\neg d]\}$. Each step is explained beneath the clause trees in the figure. $\square$

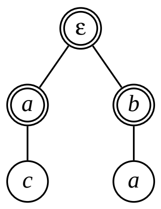
Letz et al. independently presented a general framework for Model Elimination as a connection tableau, which also resulted in the use of a tree structure [31].



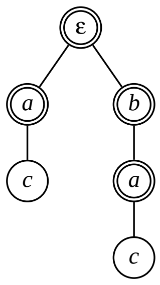
(a) Initial clause tree from clause $[a,b]$.



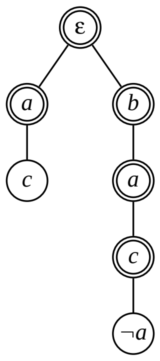
(b) Extension of a with clause $[\neg a,c]$.



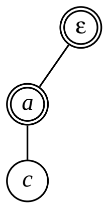
(c) Extension of b with clause $[a,\neg b]$.



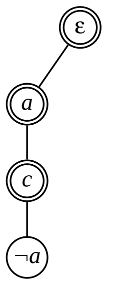
(d) Extension of a (along the right branch) with clause $[\neg a,c]$.



(e) Extension of c (along the right branch) with clause $[\neg a,\neg c]$.



(f) Reduction of $\neg a$, followed by contractions of c, a, and b, all along the right branch.



(g) Extension of c with clause $[\neg a,\neg c]$. The refutation proof can be completed by a reduction of $\neg a$, followed by contractions of c, a, and ε.

Figure 2.3: An execution of Model Elimination on clause trees. The formula is $\{[a,b],\ [a,\neg b],\ [\neg a,c],\ [\neg a,\neg c],\ [\neg a,\neg b,d],\ [\neg a,b,\neg d]\}$. Double-circle nodes indicate that the literal is an A-literal; all other nodes are B-literals. An empty clause tree indicates that the formula is unsatisfiable.

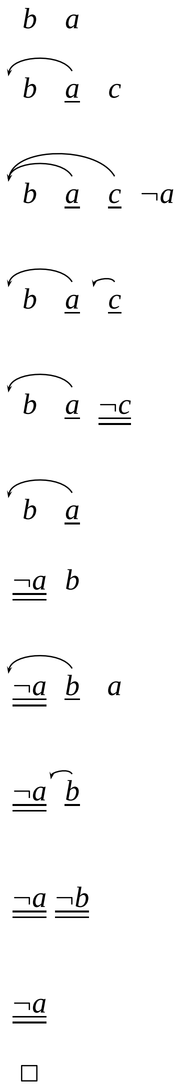
2.1.3 C-Literals as Lemmas

A number of problems had been reported about the original lemma strategy of recording lemmas as clauses. Loveland et al. write about the lack of selection rules and the increase in the number of eligible extension clauses that need to be tried [17]. Shostak writes that lemma clauses tend to be highly redundant because they are often subsumed (Definition 1.11) by other lemma clauses and/or clauses from the formula [40].

The *C-literal* strategy is a lemma strategy introduced by Shostak to solve these problems [40]. Instead of recording lemmas as clauses and adding them to the set of clauses from the formula, the C-literal strategy embeds the consequent of the lemma in the appropriate location in the chain. Because of this, the C-literal strategy does not increase the number of clauses, and in particular, it does not increase the number of possible extension clauses that need to be tried. Once in a chain, C-literals can be used like A-literals in reduction operations. C-literals can also be removed by contraction operations when they are the rightmost literal in the chain.

A C-literal is derived as the complement of the A-literal that was removed by a contraction operation. The location at which the C-literal is inserted is called the *C-point*. The C-point is maintained for each A-literal during a proof derivation. Initially, it is at the left end of the chain. Every time a reduction operation is applied to the chain, the C-points of all A-literals between the A-literal and the B-literal that were involved in the reduction operation may have their C-points adjusted. To be exact, if the C-point was to the left of the A-literal involved in the reduction operation, it is moved to the position immediately to the right of the A-literal.

Example 2.4 This example illustrates the derivation and use of C-literals in Model Elimination. In particular, it continues from Example 2.2 and shows how the execution shown in Figure 2.2 may change with the use of C-literals as the lemma strategy. The execution is shown in Figure 2.4. Each step is explained alongside the chains, and further explanation is provided in the caption. □



(1) Initial chain from clause $[a,b]$.

(2) Extension with clause $[\neg a, c]$. C-point of $\underline{a}$ is at the left end of the chain.

(3) Extension with clause $[\neg a, \neg c]$. C-point of $\underline{c}$ is at the left end of the chain.

(4) Reduction of $\neg a$ with $\underline{a}$. C-point of $\underline{c}$ moves to immediately after $\underline{a}$.

(5) Contraction of $\underline{c}$. Derives a C-literal $\underline{\underline{\neg c}}$ and inserts it at $\underline{c}$'s C-point.

Contraction of $\underline{\underline{\neg c}}$.

(6) Contraction of $\underline{a}$. Derives a C-literal $\underline{\underline{\neg a}}$ and inserts it at $\underline{a}$'s C-point.

(7) Extension with clause $[a, \neg b]$. C-point of $\underline{b}$ is at the left end of the chain.

(12) Reduction of a with $\underline{\underline{\neg a}}$. C-point of $\underline{b}$ moves to immediately after $\underline{\underline{\neg a}}$.

(13) Contraction of $\underline{b}$. Derives a C-literal $\underline{\underline{\neg b}}$ and inserts it at $\underline{b}$'s C-point.

Contraction of $\underline{\underline{\neg b}}$.

Contraction of $\underline{\underline{\neg a}}$.

Figure 2.4: An execution of Model Elimination on chains using C-literals as the lemma strategy. This shows how the use of C-literals may change the execution shown in Figure 2.2. Arrows point to the C-point for each A-literal; this is where the complement of the A-literal will be inserted as a C-literal if it is contracted. Numbers indicate the corresponding steps in Figure 2.2. Steps (5) onward illustrate the difference. In particular, the C-literal $\neg a$ derived in step (6) is used in a reduction operation in step (12), making a subrefutation attempt for literal a to be unnecessary. An underline indicates that the literal is an A-literal; a double underline indicates that the literal is a C-literal; all other literals are B-literals. An empty chain is denoted by $\square$; this indicates that the formula is unsatisfiable.

2.2 Autarky

This section describes a class of partial truth assignments called an *autarky*. Autarky was first introduced by Monien and Speckenmeyer for use in their model-search procedure [36]. However, it is also used in Modoc to identify certain subrefutation attempts that cannot succeed. The purpose of this section is to introduce the concept of autarky, with some examples, and some terminologies.

Definition 2.2 (autarky) An autarky A of a CNF formula $\mathcal{F}$ is a partial truth assignment that partitions $\mathcal{F}$ into two subsets, $autsat(\mathcal{F},A)$ and $autrem(\mathcal{F},A)$, such that any clause in $autsat(\mathcal{F},A)$ has a *literal* in common with A (and hence is satisfied by A), and any clause in $autrem(\mathcal{F},A)$ has no *variable* in common with A (and hence is not affected by the assignments made to the variables in A). $\square$

Intuitively, an autarky A of a formula $\mathcal{F}$ is a partial truth assignment that can "reduce" the satisfiability problem of the set of clauses $\mathcal{F}$ to that of a subset of $\mathcal{F}$, namely, $autrem(\mathcal{F},A)$. Note that the word "reduce" in the preceding sentence has the same meaning as its use in, say, complexity theory. That is, $\mathcal{F}$ is satisfiable *if and only if* $autrem(\mathcal{F},A)$ is satisfiable. Thus, if a partial truth assignment is an autarky, we can "commit" to this partial truth assignment and consider only the formula resulting from strengthening (Definition 1.9) the original formula with this autarky.

Example 2.5 This example illustrates what is and what is not an autarky. It also illustrates its consequences. Consider the following formula $\mathcal{F}$

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

and two partial truth assignments

$$\{a,c\},$$

$$\{a,b,c\}.$$

The first partial truth assignment $\{a,c\}$ is not an autarky of $\mathcal{F}$. This is because

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

is not a partition of F. (The clause $[\neg a, b, d]$ does not belong to either of the two subsets.)

The second partial truth assignment $\{a,b,c\}$ is an autarky of F. This is because

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

is a partition of F.

Using the autarky $\{a,b,c\}$, the satisfiability problem of F is reduced to that of $autrem(F, \{a,b,c\})$, namely, $\{[\neg d,e]\}$. Since $\{[\neg d,e]\}$ is satisfiable, so is F. A satisfying truth assignment of F can be constructed as the disjoint union of the autarky $\{a,b,c\}$ and a satisfying truth assignment of $\{[\neg d,e]\}$, say $\{\neg d\}$, as $\{a,b,c,\neg d\}$. □

A quick way to test whether a partial truth assignment is an autarky or not is to examine the clauses that contain literals that are false under the partial truth assignment. If each of these clauses also contains a literal that is true under the partial truth assignment, then the partial truth assignment is an autarky. Otherwise, it is not an autarky.

As extreme cases, an empty truth assignment and a satisfying truth assignment are both autarkies.

In general, a formula may have multiple autarkies. While some pairs of autarkies are *compatible*, meaning that the two truth assignments do not disagree on the assignments they have both made, some pairs of autarkies are *conflicting*, meaning that the two truth assignments disagree on some of the assignments. These concepts are formally defined below.

Definition 2.3 (compatible autarkies, conflicting autarkies) Let A_1 and A_2 be two autarkies of formula F. Then, A_1 and A_2 are called *compatible autarkies* if no complementary literals can be found in $A_1 \cup A_2$. Otherwise, A_1 and A_2 are called *conflicting autarkies*. □

Since an autarky cannot contain complementary literals, if A_1 and A_2 are conflicting autarkies, then there is some literal x for which x is in A_1 and $\neg x$ is in A_2.

Example 2.6 This example illustrates compatible and conflicting autarkies introduced in Definition 2.3. Continuing with Example 2.5, $\{a,d,e\}$ and $\{\neg a,\neg b,\neg c\}$ are additional autarkies of F.

While $\{a,b,c\}$ and $\{a,d,e\}$ are compatible autarkies, $\{a,b,c\}$ and $\{\neg a,\neg b,\neg c\}$ are conflicting autarkies. □

2.3 Modoc

This section describes a satisfiability testing algorithm called *Modoc*. The purpose of this section is to introduce sufficient material about Modoc for later reference without going into too much detail. A detailed description of the procedure, including correctness proofs, can be found elsewhere [44].

Modoc is a satisfiability testing algorithm introduced by Van Gelder [44]. Unlike many of the algorithms introduced for satisfiability testing, Modoc is a refutation-search procedure; that is, instead of trying to show that the formula is satisfiable by finding a satisfying truth assignment, it tries to show that the formula is unsatisfiable by finding a refutation proof. Modoc is based on propositional Model Elimination, which it extends with a new pruning technique based on the concept of autarky (Section 2.2). Although the concept of autarky was first introduced for use in a model-search procedure [36], Van Gelder adapted it to be used in a refutation-search procedure to prune away certain subrefutation attempts that cannot succeed. Van Gelder also showed that autarkies can be constructed during failed subrefutation attempts.

One advantage of Modoc (and other backward-chaining search procedures) over model-search procedures is that it is able to be *goal sensitive* [44]. Many real-world problems can be viewed as theorem-proving problems. A formula derived from such a problem comprises of two parts—the *axioms*, which account for the majority of the clauses, and the negated conclusion of the conjectured theorem, which we will call the *theorem clauses*. The axioms are obviously consistent; thus, to test whether the formula is inconsistent or not using a backward-chaining theorem prover, it is sufficient to start refutation attempts only from the theorem clauses. Goal-sensitive search has allowed Modoc to achieve search performance comparable to incomplete model-search procedures (which are considered to be among the fastest methods to find satisfying truth assignments) on various planning formulas [47].

As a refutation-search procedure, the aim of Modoc is to find a refutation proof demonstrating that the formula is unsatisfiable. In Modoc, a refutation proof is embodied in a *refutation* *tree*, and its progress is represented by a *propositional derivation tree* (PDT). Modoc tries to construct a refutation tree using the single basic operation called *PDT extension*. These concepts are formalized below.

We first define PDT. PDT is essentially the same as the clause trees (Section 2.1.2).

Definition 2.4 (propositional derivation tree) A *propositional derivation tree* (PDT) is a tree in which two types of nodes—*clause nodes* and *goal nodes*—alternate by level. A clause node is labeled with a clause in the formula, and a goal node is labeled with a literal in the formula.

A clause node labeled with C has exactly one goal node labeled with g as a parent if and only if

- $\neg g$ is in C (or g is $\top$, described later), and
- no literal in C labels an ancestor goal node.

A clause node labeled with C has a goal node labeled with g as a child if and only if

- g is a literal in C, and
- $\neg g$ does not label any ancestor goal node of C.

$\square$

Example 2.7 Figure 2.5 shows an example of a propositional derivation tree for the formula $\{[a,b]$, $[a,\neg b,c]$, $[\neg a,c]$, $[\neg a,\neg c]$, $[\neg b,\neg c]\}$. $\square$

To avoid wordiness, we may simply write "clause node C" in place of "clause node labeled with C" unless this may cause confusion. Similarly for goal nodes.

Definition 2.5 (refutation tree) A *refutation tree* is a PDT whose root is a goal node labeled with a special symbol $\top$, called the *verum*, and whose leaf nodes are all clause nodes. A subtree of a refutation tree is called a *refutation subtree*. $\square$

Example 2.8 The propositional derivation tree shown in Figure 2.5 is also a refutation tree for the formula. $\square$

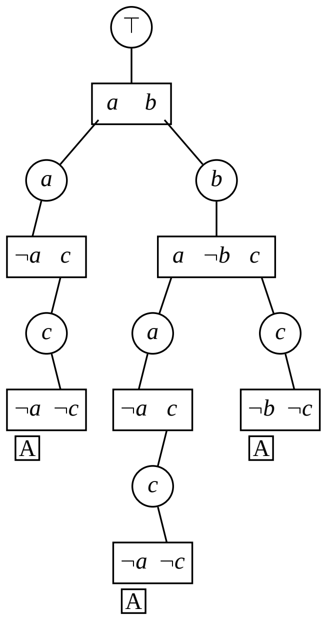


Figure 2.5: An example of a propositional derivation tree for the formula $\{[a,b],\ [a,\neg b,c],\ [\neg a,c],\ [\neg a,\neg c],\ [\neg b,\neg c]\}$. $\boxed{\text{A}}$ indicates that the complement of the corresponding literal is an ancestor goal node. This tree is also a refutation tree for the formula.

Note that if all the literals in a clause node are complements of some ancestor goal nodes, then this clause node has no child goal nodes.

The connection between refutation trees and the satisfiability of formulas are given by the following theorem [34, 31, 44, 43].

Theorem 2.1 If a refutation tree can be constructed for a formula, then the formula is unsatisfiable. If no refutation tree can be constructed for a formula, then the formula is satisfiable. □

Modoc tries to construct a refutation tree in a depth-first fashion starting from a tree with only the verum $\top$ using its only operation, *PDT extension*.

Definition 2.6 (PDT extension) The *PDT extension* operation extends a goal node with a clause that satisfies the following two conditions:

1. The clause contains the complement of the goal node.

2. The clause does not contain any ancestor goal nodes.

(The only exception to this rule is that the operation may extend the verum $\top$ with any clause.) Further, it adds goal nodes beneath the new clause node. There is one goal node for each literal that satisfies the following two conditions:

1. The literal is in the new clause node.

2. The literal is not the complement of some ancestor goal node.

We say that goal node creation was *suppressed* for a literal in a clause node if the complement of the literal labels a non-parent ancestor goal node (i.e., an ancestor goal node that is not the parent of the clause node). □

Definition 2.7 (top clause) The clause used to extend the verum $\top$ is called the *top clause*. □

For the sake of efficiency, it is not necessary for Modoc to actually construct every part of a refutation tree, or to attempt construction using every possible choice. That is, if the outcome of subtree construction can be foreseen, Modoc may move on to other parts of the tree whose outcome is unknown. There are two types of situations when this could happen. One is when it can be foreseen that a refutation subtree can be constructed beneath a goal node. Another is when it can be foreseen that *no* refutation subtree can be constructed beneath a clause node. The former involves the use of *lemmas*, and the latter involves the use of *autarkies*.

When a goal node is successfully refuted, Modoc records this fact as a lemma. The mechanism used in Modoc is essentially that of C-literals (Section 2.1.3) adapted for PDT. The presence of a C-literal indicates that the complement of the C-literal can be refuted in the subtree below its attachment point. Therefore, there is no need to attempt refutation of a goal node labeled with the complement of the C-literal in this subtree.

A feature that is new in Modoc is that of *autarky pruning*, which is based on the following theorems by Van Gelder [44, 43].

Theorem 2.2 A failed refutation of a goal node derives an autarky. □

Theorem 2.3 If a clause is satisfied by an autarky, then the use of that clause in a PDT extension operation cannot lead Modoc to a successful subrefutation. □

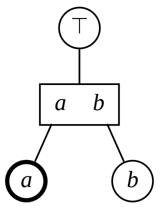
Modoc uses these theorems to derive and use autarkies to eliminate from the set of possible extension clauses certain clauses that cannot lead Modoc to a successful subrefutation.

The incorporation of lemmas and autarky pruning causes the actual PDT extension operation used in an implementation of Modoc to be modified as follows.

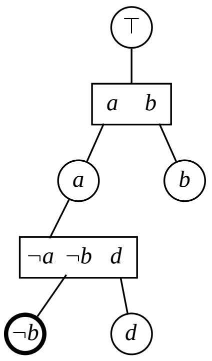
1. The clause used to extend the goal node is further required *not* to be satisfied by the current autarky.

2. A literal used to create a new goal node is further required *not* to be the complement of some C-literal attached to some ancestor goal node.

Example 2.9 This example illustrates the PDT extension operation of Modoc. In particular, it shows how Modoc may conclude that a given formula is inconsistent. Figure 2.6 shows an execution of Modoc on the formula $\{[a,b],\ [a,\neg b],\ [\neg a,c],\ [\neg a,\neg c],\ [\neg a,\neg b,d],\ [\neg a,b,\neg d]\}$. Of particular interests are suppression of goal node creation in Figures 2.6(c) and 2.6(g), autarky derivation in Figures 2.6(c) and 2.6(d), autarky pruning in Figure 2.6(f), C-literal derivation in Figures 2.6(h) and 2.6(i), and use of a C-literal in Figure 2.6(i). □

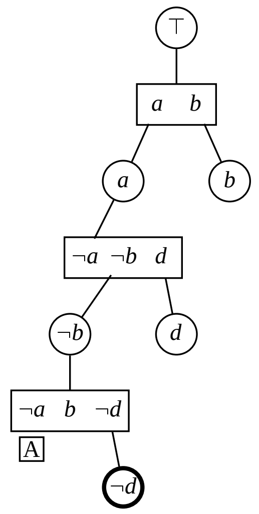


(a) Clause $[a,b]$ is chosen as the top clause. Two goal nodes a and b are immediately created.

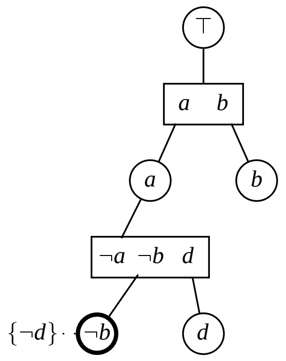


(b) All four clauses containing $\neg a$ are eligible to extend goal node a. Here, Modoc extends goal node a with clause $[\neg a, \neg b, d]$. This creates two new goal nodes $\neg b$ and d.

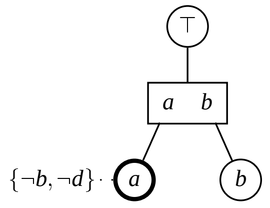
Figure 2.6: An execution of Modoc. The formula is $\{[a,b], [a,\neg b], [\neg a, c], [\neg a, \neg c], [\neg a, \neg b, d], [\neg a, b, \neg d]\}$. Clause nodes are shown in rectangles and goal nodes are shown in circles. Thick circles indicate where the search is. The example continues to page 28.



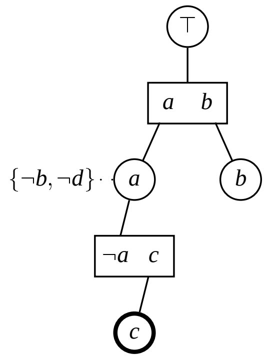
(c) Only clause $[\neg a, b, \neg d]$ is eligible to extend goal node $\neg b$. (Clause $[a, b]$ is ineligible because it contains an ancestor goal node a.) Thus, Modoc extends goal node $\neg b$ with clause $[\neg a, b, \neg d]$. This creates a new goal node $\neg d$. Note that the creation of goal node $\neg a$ was suppressed. This is because its complement a is a non-parent ancestor goal node. Modoc now tries to extend goal node $\neg d$. However, no clause is eligible to extend it. (Clause $[\neg a, \neg b, d]$ is ineligible because it contains an ancestor goal node $\neg b$.) This implies that the refutation attempt for goal node $\neg d$ has failed. This causes Modoc to derive an autarky $\{\neg d\}$.



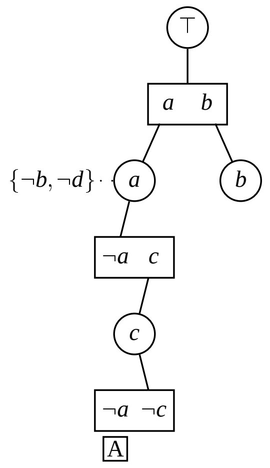
(d) Modoc now backtracks to goal node $\neg b$ with autarky $\{\neg d\}$. The autarky is *conditional* in the sense that it is an autarky for the formula resulting from strengthening the original formula with the partial truth assignment implicit by the set of ancestor goal nodes—in this case, $\{a, \neg b\}$. Modoc now tries to extend goal node $\neg b$ with some other eligible clause. However, no other clause is eligible to extend it. This implies that the refutation attempt for goal node $\neg b$ has failed. This causes Modoc to derive an autarky $\{\neg b, \neg d\}$, constructed as the union of $\neg b$ (the current goal node) and the current autarky $\{\neg d\}$.



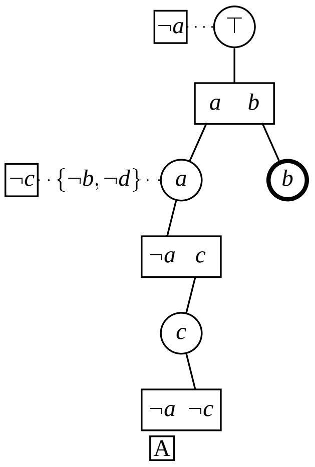
(e) Modoc now backtracks to goal node a with autarky $\{\neg b, \neg d\}$. Again, the autarky is a conditional autarky that is conditioned on the set of ancestor goal nodes $\{a\}$. Modoc now tries to extend goal node a with some other eligible clause.



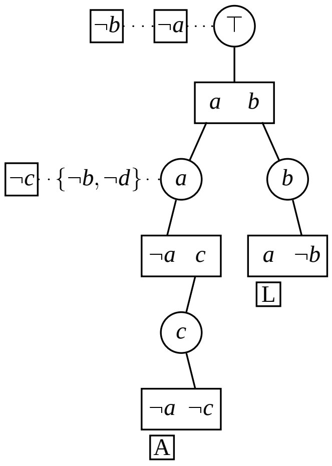
(f) There are two clauses eligible to extend goal node a. (Clause $[\neg a, b, \neg d]$ is not eligible as it is satisfied by the autarky $\{\neg b, \neg d\}$.) Here, Modoc extends goal node a with clause $[\neg a, c]$. This creates a new goal node c.



(g) Only clause $[\neg a, \neg c]$ is eligible to extend goal node c. Thus, Modoc extends goal node c with clause $[\neg a, \neg c]$. The creation of goal node $\neg a$ was suppressed because its complement a is a non-parent ancestor goal node. This completes the refutation along this branch.



(h) The successful subrefutation of goal node *c* causes a C-literal ¬*c* to be derived and attached to goal node *a*. It also causes a C-literal ¬*a* to be derived and attached to the verum ⊤. Search must now continue to refute goal node *b*.



(i) There are two clauses eligible to extend goal node *b*. Here, Modoc extends goal node *b* with clause $[a, \neg b]$. The creation of goal node *a* is suppressed because its complement ¬*a* is a C-literal attached to an ancestor. This completes the refutation along this branch (because no new goal nodes are created), and also the refutation for this formula, as all leaf nodes are now clause nodes. This causes a C-literal ¬*b* to be derived and attached to the verum ⊤.

Chapter 3

Background: Planning as Satisfiability Testing

This chapter reviews how planning problems can be formulated using logic, and in particular, as propositional satisfiability problems. It also demonstrates that a planning formula comprises of two parts—the axioms and the negated conclusion of the conjectured theorem, as briefly mentioned in Section 2.3. Backward-chaining theorem provers, such as Modoc, may exploit this structure and perform goal-sensitive search (Section 2.3). Section 3.1 describes how a planning problem may be formulated as a propositional satisfiability problem, in particular, as a CNF formula. Section 3.2 describes how the additional view of theorem proving may give additional clues on where to start a search when a backward-chaining theorem prover is used to determine the satisfiability of the formula.

Note that although the discussion in this chapter deals with planning problems, the same techniques could be applied on many other types of problems.

3.1 Planning as Propositional Satisfiability

This section describes how planning problems can be formulated as propositional satisfiability problems. The formulation is such that a formula is satisfiable if and only if the original planning problem has a successful plan. A successful plan can be obtained by interpreting the assignments made by a satisfying truth assignment to propositions that represent actions.

To the author's best knowledge, Kautz and Selman were the first to actually test the idea of formulating planning problems as propositional satisfiability problems [27]. The idea has further been refined by Kautz and Selman [28] and by Ernst et al. [15] The intuitive idea is to express all the requirements for a successful plan as boolean constraints, in particular, as boolean clauses. A truth assignment that satisfies these clauses is hence a successful plan.

More specifically, the requirements will comprise of several components. The largest component is the domain constraints, which are requirements that are independent of individual problem instances but are common to all planning problems of the same type. For instance, in a block-world planning problem, rules on how and when a block can be moved from one place to another would be one of the domain constraints. Domain constraints are also called *axioms*. Given the domain constraints, individual problem instances are characterized by their initial and final conditions. The initial condition expresses the requirements for the initial state, and the final condition expresses the requirements for the final state.

Kautz and Selman identified a sufficient set of axioms that is necessary to formulate a planning problem [27, 28]. These are listed below:

- Actions imply both their preconditions and effects.
- Exactly one action occurs at each time step.
- The initial state is completely specified.
- If an action does not change a relation, then the relation holds in the next time step.

A number of software tools, such as Satplan [28] and Medic [15], are available to generate planning formulas from planning problems. Satplan takes hand-coded axioms, while Medic takes axioms described as STRIPS-style operators [16].

3.2 Additional Theorem-Proving View

This section describes the additional theorem-proving view that could be brought into the planning formulas described in Section 3.1. This view may provide backward-chaining theorem provers, such as Modoc, additional clues on where to focus their search effort.

A common technique used to prove that a given formula

$$\Sigma \Rightarrow \phi$$

is a theorem, where Σ is a conjoined set of expressions that is commonly consistent and ϕ is another expression, is to show that

$$\Sigma \wedge \neg\phi$$

is inconsistent. When a backward-chaining theorem prover is used, $\neg\phi$ is used as the top clause.

The planning formulas described in Section 3.1 have the form

$$\text{axioms} \wedge \text{init} \wedge \text{final}$$

where “axioms”, “init”, and “final” are each a set of clauses that express the axioms, the initial condition, and the final condition, respectively. Obviously, the “axioms” are consistent, and so is the union of the “axioms” and “init”. Therefore, the formula could be viewed as a formula prepared for the sake of proving, by refutation, that the following formula is a theorem:

$$\text{axioms} \wedge \text{init} \Rightarrow \neg\text{final}.$$

That is, given the axioms and the initial condition, it logically follows that the final condition can never be attained.

When a backward-chaining theorem prover is used to determine the satisfiability of a planning formula, the theorem-proving view provides the clue that the clauses that express the final condition are a suitable and a sufficient set of clauses that should be tried as the top clause.

Chapter 4

Summary of Results

This chapter summarizes the results contained in the remainder of this dissertation. These include a formula-simplification scheme suitable for propositional backward-chaining theorem provers, various enhancements made to the basic design of Modoc, and a multi-agent search procedure that uses Modoc as search agents.

4.1 Goal-Sensitive Simplification

This section summarizes the results contained in Chapter 5 on *goal-sensitive simplification*. Goal-sensitive simplification is a formula-simplification scheme suitable for propositional backward-chaining theorem provers, such as Modoc.

Simplifying a formula prior to running a satisfiability tester has become standard practice as it generally allows the search to complete in less time. However, traditional simplification procedures are geared toward use with model-search procedures. Because of this, certain simplification procedures cannot guarantee anything beyond the preservation of satisfiability.

Backward-chaining theorem provers generally require the simplified formula to retain not only satisfiability, but also other attributes such as models and theorem clauses. The chapter reviews some of the common simplification procedures (Section 5.1) and then describes a simplification scheme (Section 5.2) that achieves the preservation of all three attributes—satisfiability, models, and theorem clauses.

4.2 Enhancements to Modoc

This section summarizes the results contained in Chapter 6 on various enhancements made to the basic design of Modoc, which was described in Section 2.3.

The *quasi-persistent lemma* strategy (Section 6.2) potentially improves upon the C-literal strategy (Section 2.1.3). A problem with the C-literal strategy is that if a subrefutation attempt fails, all lemmas derived during that attempt are discarded. The quasi-persistent lemma strategy derives the exact same lemmas, but the lemmas need not be discarded unless the lemmas are attached to goal nodes that are being abandoned. The benefit is that the lemmas may be retained longer. However, it requires the attachment points to be computed *after* the lemmas are derived (as opposed to them being computed *during* proof derivation by moving the C-points. See Section 2.1.3 for details.)

The *eager lemma* strategy (Section 6.3) allows certain lemmas to be derived without successful subrefutation attempts (which is the normal way to derive lemmas). Under certain circumstances, the graphs implicitly constructed during eager-lemma derivations can be used to derive additional lemmas (Section 6.4). The articulation points in the graphs correspond to nodes that can be turned into lemmas.

Cuts allow a current subrefutation attempt to be abandoned and to be replaced by a short alternate proof. Two cuts are introduced—*lemma-induced cuts* (Section 6.5) and *C-reduction-induced cuts* (Section 6.6). Cuts can be invoked when certain conditions are met.

Several experimental features are also described (Sections 6.7 to 6.9).

Experimental results (Section 6.10) show that many of the features improve the search speed and allow Modoc to outperform many model-search procedures on planning formulas. However, the growth rate of Modoc search time on random formulas is currently worse than the growth rate of the satisfiability testers it outperforms on application-derived formulas.

4.3 Parallel Modoc

This section summarizes the results contained in Chapter 7 on *Parallel Modoc*. Parallel Modoc is a multi-agent search procedure that uses enhanced Modoc (Chapter 6) as search agents. Each agent uses a different theorem clause as its top clause and attempts to find a refutation. During the search for refutation, autarkies and lemmas found by one agent are communicated to other agents. The agent receiving the autarkies and lemmas may use them as if they had been derived by the agent itself. As a result, the search is *cooperative*, as opposed to being independent. It is hoped that by doing so, search for a refutation would be expedited.

When multiple searches are made at the same time, it is possible for multiple Modoc agents to derive conflicting autarkies (Definition 2.3). Section 7.2 proves some properties concerning multiple autarkies and presents two algorithms—the first algorithm allows two arbitrary autarkies to be combined to form an autarky that is at least as large as the two given autarkies, and the second algorithm is an optimization of the first algorithm and allows only the new autarky literals to be communicated to accomplish the same task.

One problem with the current design of Parallel Modoc is that the maximum number of agents that could be utilized is tied to the number of theorem clauses. Section 7.4 describes two methods to increase the number of clauses suitable as the top clause, in hopes of increasing the degree of parallel search and subsequently the amount of cooperation.

Experimental results (Section 7.5) show that the speedup of Parallel Modoc over Modoc is very often greater than the number of agents. Formulas that could not be solved in an hour by Modoc were often solved by Parallel Modoc in the order of minutes, and in some cases, in seconds, with only 2 to 6 agents. Experiments also show that increasing the number of possible top clauses, and hence agents, allows faster search in many instances.

Chapter 5

Goal-Sensitive Simplification

This chapter describes *goal-sensitive simplification*. Goal-sensitive simplification is a formula-simplification scheme specifically designed for use with a backward-chaining theorem prover, such as Modoc. It preserves certain attributes of the formula that traditional simplifiers may not preserve across simplification. Section 5.1 reviews many of the common simplification procedures and describes the problems associated with some of them for use prior to a backward-chaining theorem prover. Section 5.2 describes a solution.

5.1 Simplifier and Satisfiability Testing

This section reviews many of the common simplification procedures used in satisfiability testing and describes the problems associated with some of them for use with a backward-chaining theorem prover. A solution to the problem is described in Section 5.2.

It has become common practice to simplify the formula before running a satisfiability tester on it. A simplified formula is generally smaller, most often allowing the satisfiability tester to determine its satisfiability in less time. A simplifier takes a formula and applies various quick and easy simplification procedures that preserve satisfiability. Examples of simplification procedures are summarized in Table 5.1.

Any formula resulting from any combination of the simplification procedures listed in Table 5.1 will still be an acceptable input formula to a backward-chaining theorem prover. However,

unit implication	Remove all clauses that contain a literal in a unit clause, and remove all occurrences of the complement of that literal.
pure-literal elimination	Remove all clauses that contain a pure literal (Definition 1.4).
equivalent-literal substitution	Replace all occurrences of literals that are logically equivalent (Definition 1.10) with a single literal. This is commonly achieved by reasoning on the binary clauses, in particular, by considering all binary clauses as implications, constructing a graph of literals with arcs representing implications, and running a strongly connected components algorithm; then, each strongly connected component is a set of logically-equivalent literals.
subsumption	Remove all clauses that are subsumed (Definition 1.11) by some other clause in the formula.

Table 5.1: Common simplification procedures used in satisfiability testing.

when a backward-chaining theorem prover is used, it is strongly recommended that certain attributes be retained in the simplified formula. First, all theorem clauses should be preserved across simplification. This allows goal-sensitive search to be performed. Should a simplification procedure eliminate the theorem clauses, the prover is left with a formula with no place to focus its search effort on. Second, the simplified formula should be logically equivalent (Definition 1.10) to the original formula. (However, this is *not* an absolute requirement.) Among the simplification procedures listed in Table 5.1, pure-literal elimination is not guaranteed to produce a logically-equivalent formula, and unit implication may eliminate theorem clauses, as they tend to be unit clauses in practice.

5.2 Simplifier for Backward-Chaining Theorem Provers

Section 5.1 described some of the problems associated with the use of common simplification procedures prior to running a backward-chaining theorem prover. This section describes a solution.

To overcome the problems described in Section 5.1, the author, in collaboration with Allen Van Gelder, devised a simplification scheme that will guarantee all three requirements—preservation of satisfiability, models, and theorem clauses. The scheme is called *goal-sensitive simplification* [47]. (To emphasize the difference, traditional simplification will be called goal-

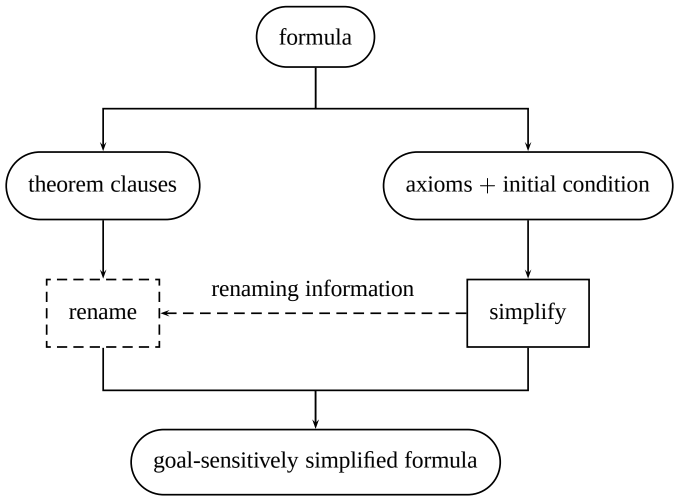

formula
theorem clauses
axioms + initial condition
rename
renaming information
simplify
goal-sensitively simplified formula

Figure 5.1: Steps involved in goal-sensitive simplification. Details are given in Section 5.2.

*in*sensitive simplification.)

Figure 5.1 outlines the steps involved in goal-sensitive simplification. For the purpose of illustration, we assume that the input formula is a planning formula (Section 3.1). (Note, however, that the scheme could be applied to any formula whose original problem could be viewed as a theorem-proving problem.) The clauses in the input formula are partitioned into two sets—one that consists of the theorem clauses, and another that consists of the clauses that describe the axioms and the initial condition. Then, we run a regular simplifier to the second set (the axioms and the initial condition), with the following requirements:

1. Pure-literal elimination is not to be used.

2. Any renaming of the variables that occurred during simplification is to be recorded.

Simplification procedures such as equivalent-literal substitution may rename literals. If such procedures are used, we need to make sure that the literals in the first set (the theorem clauses), which was not subjected to the simplifier, are renamed in the same way. The goal-sensitively simplified formula is obtained as the union of the first set (renaming done, if applicable) and the simplified second set.

Note that a goal-sensitively simplified formula can be considered as being in an intermediate form to becoming a goal-insensitively simplified formula. In particular, applying goal-insensitive simplification to a goal-sensitively simplified formula produces the same formula as the formula produced by applying goal-insensitive simplification directly to the original formula. This means that the formula that is goal-sensitively simplified is generally slightly larger than the formula that is goal-insensitively simplified.

Chapter 6

Enhancements to Modoc

This chapter describes various enhancements made to the basic design of Modoc (Section 2.3). Before describing the enhancements, Section 6.1 discusses some of the design decisions made in implementing Modoc, as they relate to the enhancements. Section 6.2 describes the quasi-persistent lemma strategy, which potentially improves upon the C-literal strategy (Section 2.1.3). Section 6.3 describes the eager lemma strategy, which allows certain lemmas to be derived without successful subrefutation attempts. Section 6.4 describes how articulation points in certain graphs implicitly constructed during eager-lemma derivations can derive additional lemmas. Section 6.5 describes lemma-induced cuts and Section 6.6 describes C-reduction-induced cuts. These cuts allow a subrefutation attempt in progress to be completed by a short alternate proof. Section 6.7 describes how refutation can be propagated during cuts. Section 6.8 describes how pure literals found in the formula can be exploited in Modoc. Section 6.9 describes a lookahead strategy. Section 6.10 reports experimental results obtained from the enhanced version of Modoc.

Most of the work contained in this chapter was performed in collaboration with Allen Van Gelder.

6.1 Implementation of Modoc

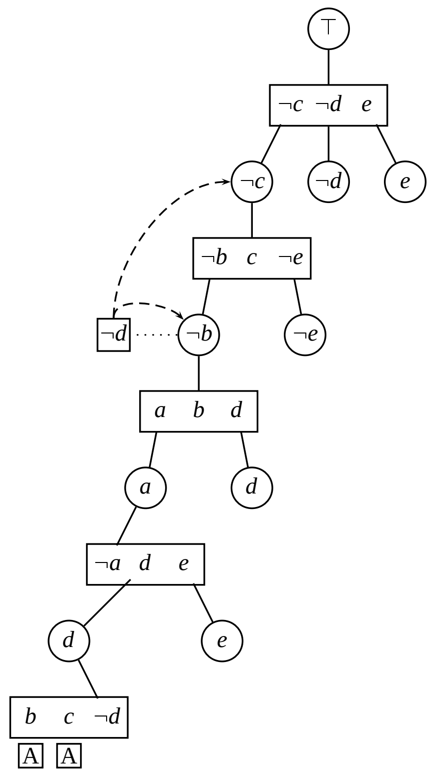
This section discusses some of the design decisions made in implementing Modoc. We refer to the implementation as `modoc` (note the change in typeface). The discussion will be limited to those that have some relevance to the various enhancements described later in this chapter.

A key decision made at the beginning of the implementation was to do *pre-reduction*. That is, whenever a goal node is added to the current set of ancestor goal nodes, or a C-literal is attached to a goal node, the length of the clauses in the formula are modified to reflect the actual number of literals that need to be refuted. To elaborate, whenever a goal node x is added to the current set of ancestor goal nodes, all the clauses that contain $\neg x$ will have their length decreased by one. (Recall from Section 1.3 that we assumed each clause to be non-duplicating.) Also, whenever a C-literal y is attached to the PDT, all the clauses that contain $\neg y$ will have their length decreased by one. Pre-reduction allows certain enhancements to be implemented efficiently, namely, *eager-lemma* derivations (Section 6.3) and *C-reduction-induced cuts* (Section 6.6).

6.2 Quasi-Persistent Lemmas

This section describes the *quasi-persistent lemma* strategy. The quasi-persistent lemma strategy potentially improves upon the C-literal strategy. The C-literal strategy avoids the problems associated with the original lemma strategy of Model Elimination of recording the lemmas as clauses. (See Section 2.1.3 for details.) However, it has a problem that if a subrefutation attempt fails, all the lemmas derived during that subrefutation attempt are discarded. Obviously, this is a needlessly conservative policy to maintain lemmas because the lemmas are nonetheless sound logical implications.

Van Gelder proposed a more “persistent” variant of the C-literal strategy called the *quasi-persistent lemma* strategy [44]. The quasi-persistent lemma strategy derives the exact same lemmas as does the C-literal strategy. However, unlike the C-literal strategy, a lemma is discarded *only if* it is attached to a goal node that is being “abandoned” (because the search has moved on to another part of the refutation tree). Potentially, this means that quasi-persistent lemmas may be retained longer than C-literals. A downside of it is that, unlike C-literals, determining the attachment point of a lemma must be postponed until *after* the lemma is derived (as opposed to it being computed *during* the derivation by moving the C-point; see Section 2.1.3 for details). The attachment point is determined as the lowest *proper* ancestor goal node that the subrefutation that derived the lemma either directly or indirectly used in a reduction operation. To be able to do this, the exact antecedent



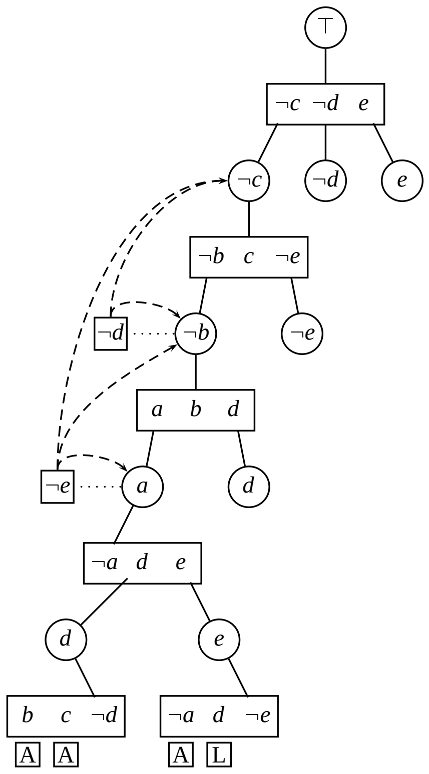
(a) Goal node *d* is refuted and this derives a lemma literal ¬*d*. The refutation is directly dependent on ancestor goal nodes ¬*b* and ¬*c*, as marked by $\boxed{\text{A}}$. Since neither is the parent goal node (in this case, *d*), the lemma is attached to the lower of the two, namely, ¬*b*. Dependencies are shown by dashed arrows.

Figure 6.1: Determining the attachment points of quasi-persistent lemmas. The example continues to page 43.

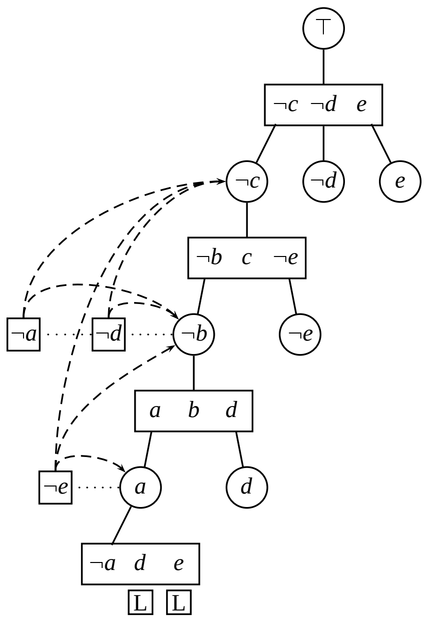
ancestor goal nodes (which we will call the *dependencies*) must be recorded for each lemma. This means that more bookkeeping must be done. Another downside of it is that it is incompatible with a heuristic called *strong regularity* found by Letz et al. [31, 46]

Example 6.1 This example illustrates how dependencies and attachment points are determined for quasi-persistent lemmas. Figures 6.1(a) and 6.1(b) show two simple cases. Details are given in the captions. □

When a goal node is refuted because all its (non-empty set of) sub-goal nodes are refuted, in order to determine the attachment point of the new lemma literal, we would normally have to



(b) Goal node e is refuted and this derives a lemma literal $\neg e$. The refutation is directly dependent on ancestor goal node a (indicated by $\boxed{\text{A}}$) and on lemma literal $\neg d$ (indicated by $\boxed{\text{L}}$). Since lemma literal $\neg d$ is dependent on ancestor goal nodes $\neg b$ and $\neg c$ (indicated by the dashed arrows), the new lemma is dependent on a, $\neg b$, and $\neg c$. Since none of them is the parent goal node (in this case, e), the new lemma literal $\neg e$ is attached to the lowest of the three, namely, goal node a. Dashed arrows show the dependencies of lemma literal $\neg e$.



(c) An equivalent PDT that could be considered to determine the dependencies and the attachment point of lemma literal $\neg a$. This shows that the new lemma literal $\neg a$ is dependent on lemma literals $\neg d$ and $\neg e$, which are dependent on ancestor goal nodes $\neg b$ and $\neg c$, and a, $\neg b$, and $\neg c$, respectively. The new lemma is attached to the lowest *proper* ancestor goal node, namely $\neg b$.

find all the ancestor goal nodes that were used either directly or indirectly in reduction operations during the refutation of the goal node. However, in practice, this need not be done. The fact that the goal node is refuted means that all its sub-goal nodes were successfully refuted and that their complements are now attached as lemma literals. Thus, to determine the attachment point of the new lemma literal, we need not consider the actual PDT that derived the new lemma literal but could instead consider an equivalent PDT in which the creation of the sub-goal nodes are suppressed because of the lemma literals.

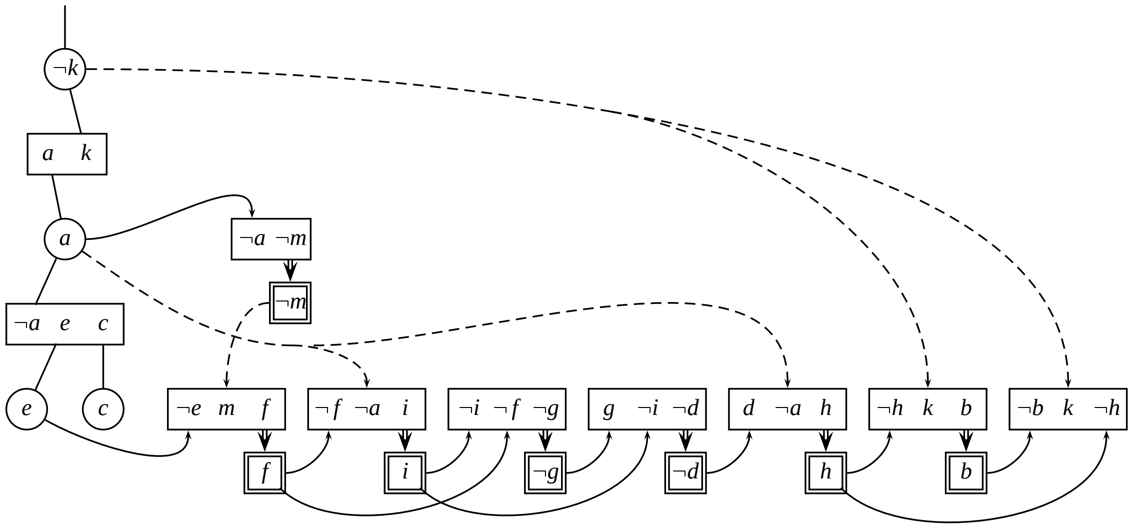
Example 6.2 This example continues from Example 6.1. Note that Figure 6.1(b) also shows that goal node a has been refuted and that a lemma literal $\neg a$ can be derived. To determine the attachment point of the lemma literal $\neg a$, we would normally have to examine the PDT in Figure 6.1(b). However, in practice, we could instead consider an equivalent PDT in Figure 6.1(c). Details are given in the caption. □

6.3 Eager Lemmas

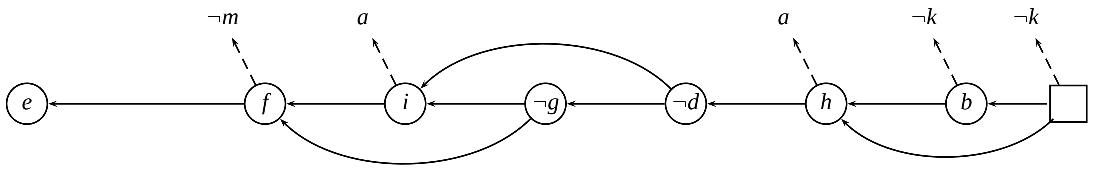
This section describes the *eager lemma* strategy. The eager lemma strategy allows certain lemmas to be derived without successful subrefutation attempts (which is the normal way to derive lemmas). The mechanism is closely related to unit propagation, which is often used in model-search procedures.

As described in Section 6.1, when a goal node is added to the current set of ancestor goal nodes, pre-reduction is performed using this goal node. The eager lemma strategy attaches the literals in unit clauses (to be exact, clauses that have just been shortened by pre-reduction to length one) as lemmas to the current goal node. The new lemmas could then be used to perform further pre-reductions, which may in turn derive more unit clauses and hence lemmas. The derivation of lemmas may continue until no new unit clauses are derived.

Example 6.3 This example illustrates the derivation of eager lemmas. Figure 6.2(a) shows how eager lemmas are derived after adding goal node e to the current set of ancestor goal nodes. □

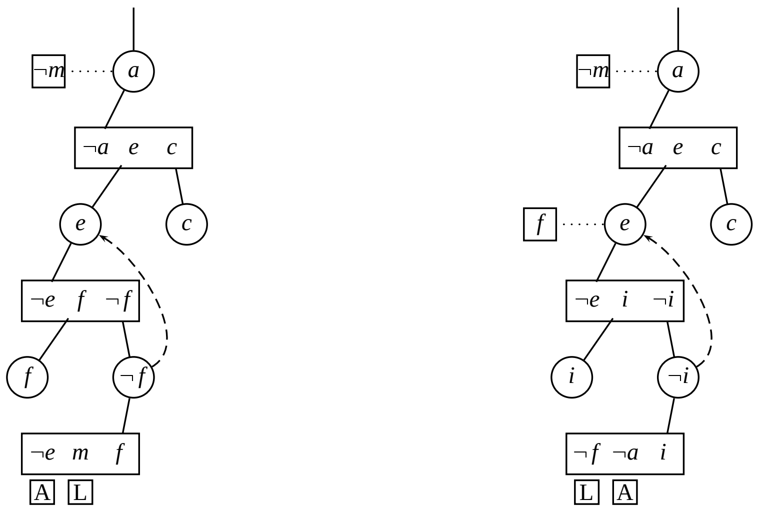


(a) A derivation of eager lemmas. Eager lemmas are shown in double squares. Solid arrows indicate depth-local pre-reductions, and dashed arrows indicate depth-non-local pre-reductions. The addition of goal node e to the current set of ancestor goal nodes starts a chain of pre-reductions, eventually causing clause $[\neg b, k, \neg h]$ to be pre-reduced to an empty clause.



(b) Eager dependency graph of the eager lemma derivation for the depth of goal node e. Solid arrows indicate depth-local dependencies, and dashed arrows indicate depth-non-local dependencies.

Figure 6.2: An eager-lemma derivation. Figure (a) shows an eager lemma derivation, and Figure (b) shows the corresponding eager dependency graph for the depth of goal node e.



(a) Justification of eager lemma f (b) Justification of eager lemma i

Figure 6.3: Justification of eager lemmas. The refutation subtrees above show that eager lemmas f and i in Figure 6.2(a) can be derived as quasi-persistent lemmas as results of successful subrefutation attempts. In both cases, the lemmas are attached to goal node e, which is where they were attached as eager lemmas. The refutations use tautologous clauses that are not part of the given formula. However, their introduction obviously preserves satisfiability. Dashed arrows point to where the lemma literals will be attached.

To demonstrate that eager lemmas are indeed lemmas, Example 6.4 shows that eager lemmas could be derived as quasi-persistent lemmas that are attached to the same goal nodes had we not used the eager lemma strategy.

Example 6.4 This example illustrates that it is possible to derive eager lemmas as quasi-persistent lemmas that will be attached to the same goal nodes. Figure 6.3 shows successful subrefutation attempts that will derive the first two eager lemmas f and i in Figure 6.2(a) as quasi-persistent lemmas. The refutations use a tautologous clause. Details are given in the caption. □

6.4 Eager Dependency and Articulation Points

This section describes how certain nodes in graphs implicitly constructed during eager-lemma derivations may derive additional lemmas. The graph is called an *eager dependency graph*, and if an empty clause was derived during an eager-lemma derivation, the articulation points in the graph can be complemented and attached as lemmas.

Example 6.5 This example continues from Example 6.3. Figure 6.2(b) shows the corresponding eager dependency graph for the depth of goal node *e*. □

To demonstrate that articulation points can be made into lemmas, Example 6.6 shows that the complements of articulation points in the eager dependency graph could be derived as quasi-persistent lemmas attached to the same goal nodes had we not used the eager lemma strategy.

Example 6.6 This example illustrates that the complements of articulation points in eager dependency graphs are indeed lemmas. Figure 6.4 shows a subrefutation attempt continuing from goal node *e* using the exact same clauses that derived the eager lemmas in Figure 6.2(a). Dashed arrows indicate where the lemmas would be attached. Note that the only lemmas that would survive after refuting goal node *e* are lemma literals that are attached to goal nodes higher than goal node *e*. The complements of such lemma literals exactly correspond to articulation points in the eager dependency graph of Figure 6.2(b). □

The standard biconnectivity algorithm found in algorithms textbooks (such as [2, 1, 4]) is sufficient for our purpose. However, because eager dependency graphs have certain structures, a simpler algorithm was designed and used in `modoc`.

By looking at the goal node that started unit propagation and the clause that was pre-reduced to an empty clause as the two end points, an eager dependency graph can be viewed as a lattice. Because of this, any “walk” along the arcs of the eager dependency graph from the empty clause will eventually arrive at the goal node *without any backtracking*. Obviously, any articulation point will be on this path. Let us call this path the *main path*. To determine the articulation points, all there is left to do is to see if there are any *bypasses* that will allow nodes on the main path to be skipped (such nodes cannot be articulation points).

The algorithm used in `modoc` is based on depth-first search and has two modes. The purpose of the first mode is to find the main path. This can be found by simply following any successor among the set of successor nodes at each visited node. Once the main path is found, the

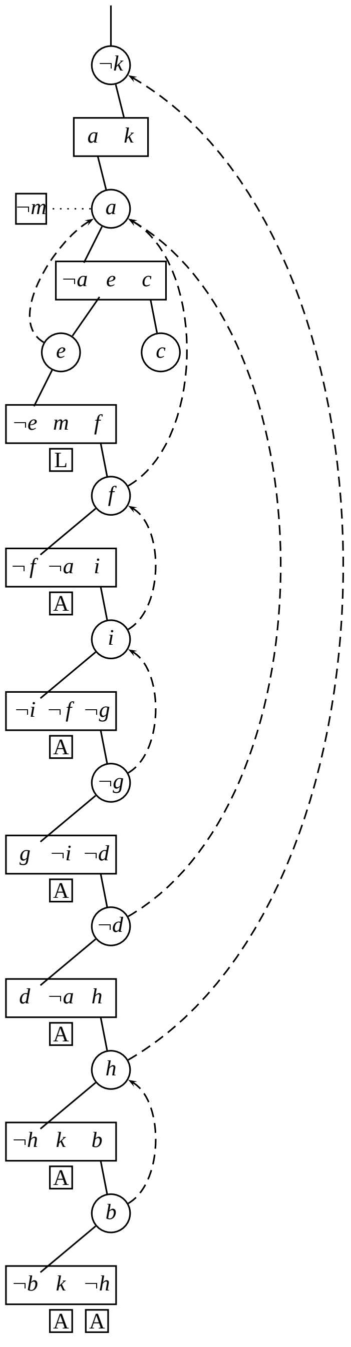


Figure 6.4: Corresponding subrefutation tree for the eager-lemma derivation shown in Figure 6.2(a). All goal nodes from e to b can be turned into quasi-persistent lemmas. Dashed arrows indicate where the lemma literals will be attached. Since the search is actually at goal node e and it will next try to refute goal node c, the only lemmas that will make sense to record are those that attach to nodes above goal node e. These are $\neg h$, d, $\neg f$, and $\neg e$, which are exactly the complements of the articulation points in the eager dependency graph shown in Figure 6.2(b).

algorithm starts to execute in the second mode while continuing the depth-first search on the lattice. The purpose of the second mode is to find all bypasses that could be used to skip nodes along the main path. For each bypass found, any node along the main path that is properly between the beginning and the end of the bypass are excluded from the set of candidates for articulation points. The algorithm continues until all possibilities are examined.

Example 6.7 This example illustrates the articulation-points algorithm on a sample graph. Figure 6.5 shows the execution of the algorithm. We assume that the successors are ordered from top to bottom for each node. Details are given in the captions. □

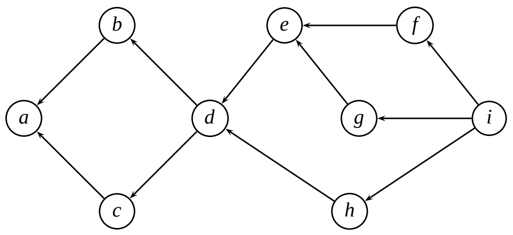
As presented, the algorithm is not guaranteed to run in time linear in the number of edges. This is because when a bypass is found, the algorithm must "walk" along the main path between the beginning and the end of the bypass, whose length is only bound by the number of nodes in the graph. In particular, there is no means to avoid eliminating a node that is already eliminated. Thus, as presented, the algorithm may take quadratic time to run.

Example 6.8 This example continues from Example 6.7. When the bypass $i \rightarrow h \rightarrow d$ is found in Figure 6.5(e), the algorithm, as currently presented, has no means to not re-eliminate node f. □

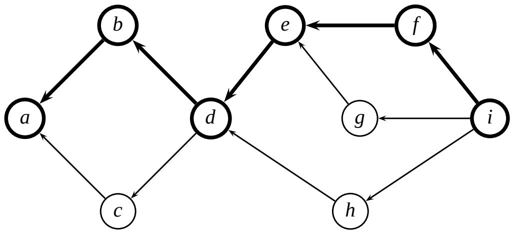
However, the algorithm can be fixed and turned into a linear-time algorithm as follows. Instead of eliminating nodes as bypasses are found, the revised algorithm will collect sufficient information during depth-first search so that a post-search stage can eliminate non-articulation points in one sweep across the graph (actually, one sweep across the main path).

During the search for the main path, we number (only) the nodes along the main path as they are visited. Every node maintains an additional attribute called *maxreachable*, which is the largest node number among the nodes along the main path that could be reached from this node by a bypass alone. After the search is done, beginning with the node from which the depth-first search started, *maxreachable* indicates the next articulation point along the main path.

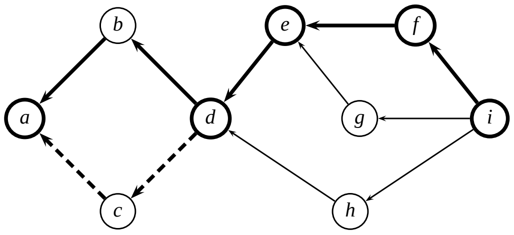
Example 6.9 This example illustrates the revised articulation-points algorithm. Figure 6.6 shows the execution of the algorithm on the same graph used in Figure 6.5. Details are given in the captions. □



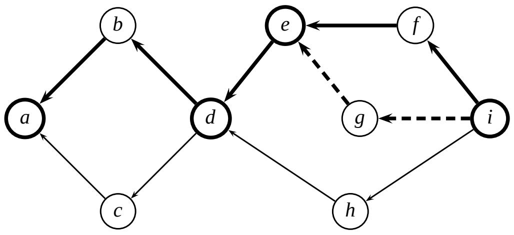
(a) The sample graph.



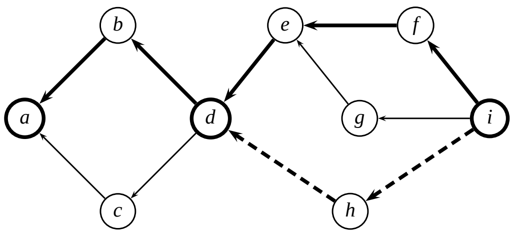
(b) The main path (indicated by the thick lines) is found. Candidates for articulation points are indicated by thick circles.



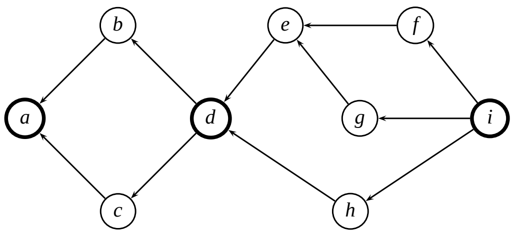
(c) A bypass (indicated by the thick dashed lines) is found. This eliminates node *b*.



(d) Another bypass (indicated by thick dashed lines) is found. This eliminates node *f*.

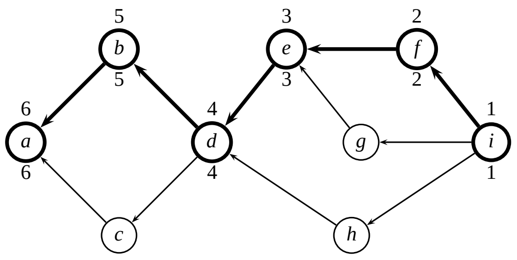


(e) Yet another bypass (indicated by the thick dashed lines) is found. This eliminates node *e*.

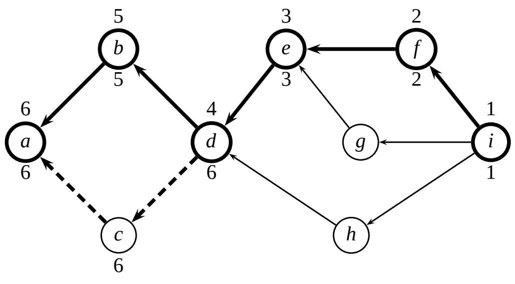


(f) The articulation points are obtained as the remaining thick circles.

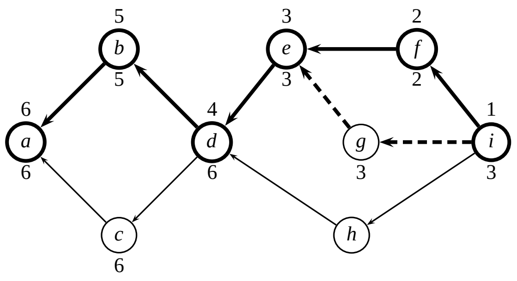
Figure 6.5: Execution of the articulation-points algorithm on a sample graph. In this sample graph, node *a* corresponds to the current goal node, and node *i* corresponds to the empty clause. Details of the algorithm are given in Section 6.4.



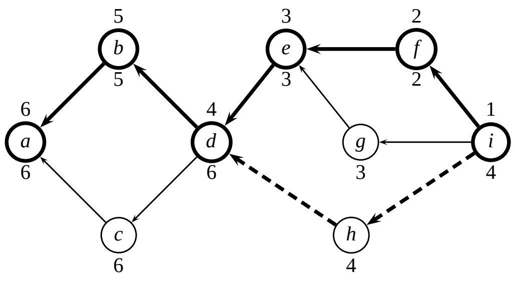
(a) Nodes along the main path are numbered (shown above each node) as they are found. The *maxreachable* attribute of each node (shown below each node) is initialized to its node number.



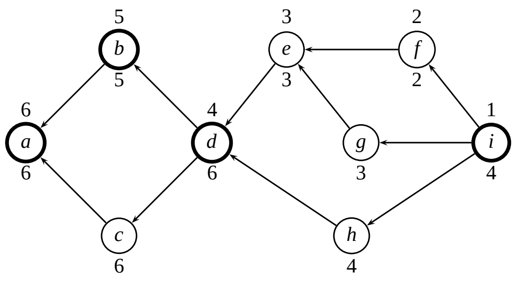
(b) A bypass is found. The *maxrechable* for nodes *c* and *d* are set to 6.



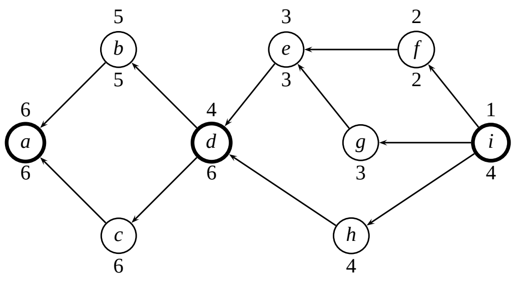
(c) Another bypass is found. The *maxrechable* for nodes *g* and *i* are set to 3.



(d) Yet another bypass is found. The *maxrechable* for nodes *h* and *i* are set to 4.



(e) Since *maxreachable* of node *i* is 4, which refers to node *d*, nodes *e* and *f* are eliminated.



(f) Since *maxreachable* of node *d* is 6, which refers to node *a*, node *b* is eliminated. The remaining thick circles indicate the articulation points.

Figure 6.6: Execution of the revised articulation-points algorithm on the sample graph shown in Figure 6.5(a). Thick lines indicate the main path, thick dashed lines indicate bypasses, and thick circles indicate the candidate nodes for articulation points. Details of the algorithm are given in Section 6.4.

6.5 Lemma-Induced Cuts

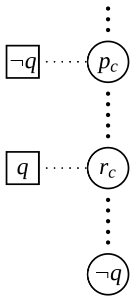
This section and the next describe *cuts*. Cuts are search optimization operations that allow a sub-refutation attempt to be abandoned and to be replaced by a short alternate proof. A cut may be invoked when certain conditions are met. This section describes *lemma-induced cuts*, which may be invoked when complementary lemma literals are attached to goal nodes along the same branch.

When complementary lemma literals are attached to goal nodes along the same branch, the lower goal node can be refuted with a short alternate proof. The proof consists of extending the lower goal node with a tautologous clause that exactly contains the complement of the goal node and the two complementary lemma literals. Because of the complementary lemma literals, goal node creation is suppressed for the new clause. Hence, the refutation for the lower goal node is completed.

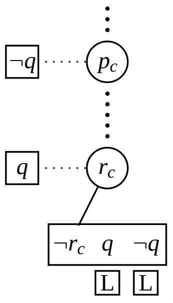
The tautologous clause used to extend the goal node is not in the formula. (Recall from Section 1.3 that we assumed that the input formula contains no tautologous clauses.) However, its introduction is harmless. This is because its introduction into the set of input clauses cannot change the set of satisfying truth assignments of the formula; in particular, it cannot change the satisfiability of the formula. The introduction of tautologous clauses is essentially a form of the cut rule due to Letz et al. [31]

While the introduction of tautologous clauses is not necessary to conclude that the lower goal node can be refuted, its introduction is useful to compute the dependencies of the new lemma literal that is derived as a result of the lemma-induced cut. However, it is impractical to add them to the set of input clauses. The current implementation of Modoc, `modoc`, temporarily introduces the clause for the purpose of computing the dependencies and then discards it immediately. Because of its ephemeral nature, the tautologous clause is called a *virtual clause*.

Example 6.10 This example illustrates the use of lemma-induced cut. Consider the situation shown in Figure 6.7(a). Goal node $\neg q$ has just been refuted and its complement q is attached as a quasi-persistent lemma to goal node r_c, which we assume, without loss of generality, is a descendent of goal node p_c, to which a lemma literal $\neg q$ is attached. The lemma-induced cut could then be



(a) Goal node $\neg q$ has successfully been refuted and its complement is attached as a lemma literal to an ancestor goal node r_c. This causes complementary lemma literals q and $\neg q$ to be attached along the same branch.



(b) By extending goal node r_c with a virtual clause $[\neg r_c, q, \neg q]$, goal node r_c is immediately refuted.

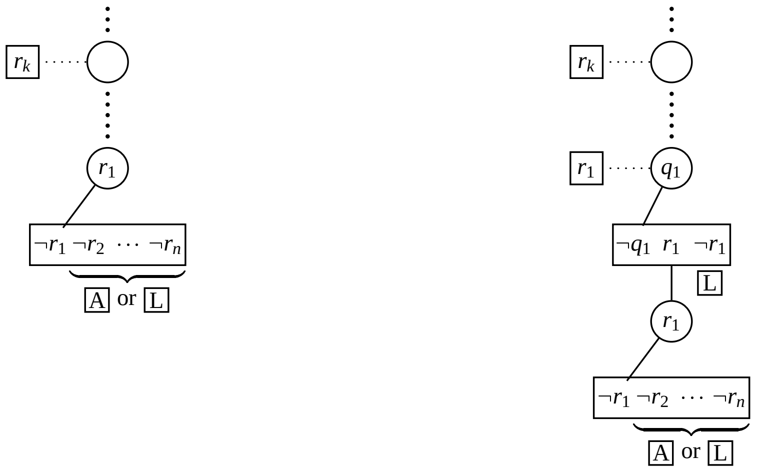
Figure 6.7: An example of lemma-induced cut. Figure (a) shows a situation in which lemma-induced cut may be invoked. The lemma-induced cut allows goal node r_c to be refuted. Figure (b) shows the refutation. Details are given in Example 6.10.

invoked to refute goal node r_c by an alternate proof as follows. Extend subgoal r_c with a virtual clause $[\neg r_c, q, \neg q]$ (Figure 6.7(b)). No goal nodes are created beneath the extension clause because the presence of lemma literal q suppresses the creation of goal node $\neg q$ and the presence of lemma literal $\neg q$ suppresses the creation of goal node q. (These are indicated by $\boxed{\text{L}}$ beneath the extension clause.) Thus, goal node r_c is refuted. $\square$

6.6 C-Reduction-Induced Cuts

This section describes *C-reduction-induced cuts*. A C-reduction-induced cut may be invoked when pre-reduction (Section 6.1) causes a clause to become empty. Like lemma-induced cuts (Section 6.5), C-reduction-induced cuts allow a subrefutation attempt in progress to be abandoned and replaced by a short alternate proof.

As described in Section 6.1, `modoc` performs pre-reduction when either a goal node is added to the current set of ancestor goal nodes, or when a lemma literal is attached to a goal node. When the pre-reduction causes a clause to become empty, C-reduction-induced cut may be invoked. C-reduction-induced cut allows the lowest goal node that caused the clause to become empty, either


r_k
r_1
$\neg r_1 \neg r_2 \cdots \neg r_n$
A or L
r_k
r_1
q_1
$\neg q_1 \; r_1 \; \neg r_1$
L
r_1
$\neg r_1 \neg r_2 \cdots \neg r_n$
A or L

(a) Case when the lowest dependency is an ancestor ($q_1 = r_1$).

(b) Case when the lowest dependency is a lemma ($q_1 \neq r_1$).

Figure 6.8: Example of C-reduction-induced cut. Two cases are possible, each depending on what the lowest dependency is. Details are given in Example 6.11.

directly or indirectly, to be refuted with a short alternate proof.

Example 6.11 This example illustrates the use of C-reduction-induced cuts. Consider the situation where clause $C = [\neg r_1, \ldots, \neg r_n]$ pre-reduces to an empty clause because of a new lemma literal r_k (where $1 \leq k \leq n$). This means that r_1, …, r_n are either ancestor goal nodes or lemma literals attached to ancestor goal nodes. Without loss of generality, assume that r_1 is the lowest ancestor goal node or lemma literal among r_1, …, r_n, chosen to be an ancestor if possible. Let q_1 be the ancestor goal node at the depth of r_1. The C-reduction-induced cut could be invoked to refute goal node q_1 by an alternate proof as follows. If $q_1 = r_1$, meaning that r_1 is an ancestor goal node, the extension of q_1 with clause C refutes it immediately, as shown in Figure 6.8(a). If $q_1 \neq r_1$, we first extend goal node q_1 with a virtual clause $[\neg q_1, \neg r_1, r_1]$; the creation of goal node $\neg r_1$ is suppressed because of the lemma literal r_1, and goal node r_1 is immediately refuted by extension with clause C. This is shown in Figure 6.8(b). □

6.7 Propagation of Refutation During Cuts

This section describes how additional lemmas may be derived when a cut is invoked. This is achieved by "propagating" refutations.

As described in Sections 6.5 and 6.6, when a cut is invoked, all subrefutation attempts for goal nodes below the goal node that the cut operation has just refuted are abandoned. However, a careful examination of the situation shows that some of the subrefutation attempts may have just succeeded and hence it may be possible to derive some additional lemmas.

Whenever a cut is invoked, a lemma was derived immediately before it. This means that a goal node was successfully refuted. If this was the last goal node that needed to be refuted for its parent goal node to be refuted, then it means that the parent goal node was refuted too. Since it is possible for the attachment point of the complement of the parent goal node to be higher than the goal node that is refuted by the cut, it may be possible to obtain additional lemmas by deriving all possible lemmas along the current goal node to the goal node refuted by the cut.

Example 6.12 This example illustrates the conditions under which goal nodes along the path from the current goal node to the goal node refuted by the cut could derive lemmas. Figure 6.9 shows a portion of a PDT. Assume, for the sake of illustration, that the refutation attempts for goal nodes are performed in left-to-right order. That is, all goal nodes to the left of the current branch have successfully been refuted. Goal nodes along the current branch are labeled by p, q, r, s, and t.

Consider the following situation. Goal node t has just been refuted and Modoc has just derived a lemma literal $\neg t$. The pre-reduction using lemma literal $\neg t$ invokes a cut and refutes goal node p.

Since goal node t is refuted, so is goal node s. However, goal node r is not refuted because it still has one goal node u left to be refuted. Further, no goal nodes above goal node r and below goal node p are refuted. □

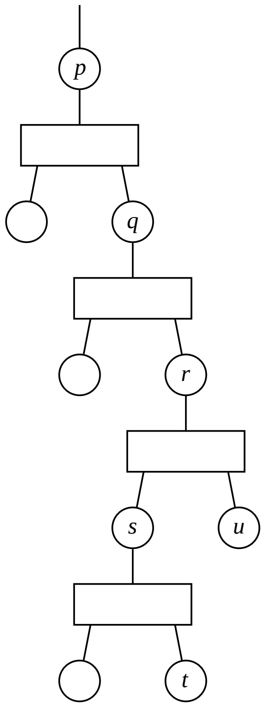


Figure 6.9: Propagation of refutation during a cut. When the successful refutation of goal node *t* causes a cut to refute goal node *p*, goal node *s* can be turned into a lemma as well. Details are given in Example 6.12.

6.8 Pure Literals as Autarky

This section describes one way Modoc may exploit pure literals in the formula. The method is related to the pure-literal rule of DPLL (Chapter 1).

Recall that the pure-literal rule of DPLL makes the pure literals true. The corresponding operation in Modoc would be to add the pure literals to the current autarky as if refutation attempts for them had just failed. Justification for this operation is given by the following lemma.

Lemma 6.1 Let x be a pure literal in formula F. Then, $\{x\}$ is an autarky of F. $\square$

Note that pure literals are already exploited to a certain degree in `modoc`. When a goal node has no eligible extension clauses, the literal in the goal node is added to the current set of autarky literals. This is because it implies that the goal node was a pure-literal. The operation described in this section is not a brand new operation but is rather a supplementary operation to fully exploit pure literals.

6.9 One-Layer Lookahead

This section describes a lookahead operation that is currently being experimented with `modoc`.

When the search reaches a clause node, there are several child goal nodes that need to be refuted. Normally, Modoc will order the goal nodes in some order and attempt refutation of each goal node, until a refutation attempt fails for a goal node. However, with one-layer lookahead, the refutation of a clause node takes place in two phases. During the first phase, *all* the goal nodes are attempted to see if they can be refuted using *only* pre-reduction and eager-lemma derivation (but no PDT extension). Any successful refutation will result in a lemma being attached to a goal node no lower than the parent of the clause node. The second phase attempts regular refutation from any goal node that remains to be refuted.

Note that during the first phase, failure to refute only by pre-reduction and eager-lemma derivation does not cause the goal node to be added to the current autarky. This is because the failure does not mean that there is no refutation for the goal node but merely that it could not be refuted using only pre-reduction and eager-lemma derivation.

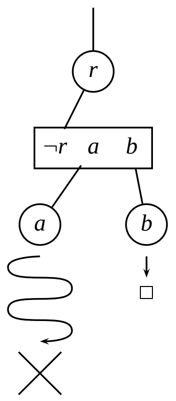


Figure 6.10: An example of how one-layer lookahead may help. Suppose that goal node *a* cannot be refuted (indicated by a wiggly arrow leading to a large “X”) and that goal node *b* can be refuted by pre-reduction and eager-lemma derivation (indicated by a short arrow leading to “□”). Further, suppose, for the sake of illustration, that Modoc attempts refutation in a left-to-right order. Normal Modoc will first attempt refutation of goal node *a*, and since it fails, it will then backtrack to goal node *r* to try to extend the goal node with some other clause. In comparison, Modoc with one-layer lookahead will first test all the goal nodes of whether they can be refuted using only pre-reduction and eager-lemma derivation. In this example, it will find out that goal node *b* can be refuted by only pre-reduction and eager-lemma derivation and will therefore attach lemma literal $\neg b$ to some goal node no lower than goal node *r*. Eventually, it will attempt refutation of goal node *a*, which will fail, and the search will backtrack to goal node *r* as before. However, the lemma literal $\neg b$ is retained and can be used in future refutation attempts.

Example 6.13 This example illustrates a potential benefit arising from incorporating one-layer lookahead in Modoc. Figure 6.10 shows a situation where Modoc has just extended goal node r with clause $[\neg r, a, b]$. Details are given in the caption. □

6.10 Experimental Results

This section assesses the performance of `modoc` by comparing it with other satisfiability testers (Section 6.10.1), and also by turning on, or off, each feature of enhanced Modoc (Section 6.10.2). Results were obtained by experiments on two types of formulas—planning formulas and random formulas.

The majority of the planning formulas were generated using Satplan [28] and Medic [15]. (The checker-interchange formulas used in Table 6.2 were generated using a purpose-built script.) For each problem (and in the case of Medic, each encoding), two formulas with different deadlines were generated. One formula had the deadline set to the optimal plan length, making it satisfiable, and another formula had the deadline set to one less than the optimal plan length, making it unsatisfiable. After the formulas were generated, they were subjected to goal-sensitive simplification (Section 5.2).

Random formulas have no clause to focus on, and thus, they are not a good class of formulas to demonstrate goal-sensitive search. However, they are easy to generate as many formulas as necessary with the given characteristics. Because of this, it is suitable, nonetheless, for use in an experimental study of Modoc's (and other satisfiability testers') growth rate of search times.

The random formulas used in this section are random 3-CNF formulas (Definition 1.3). They are generated using a probability model in which each non-redundant non-tautologous clause of length three is equally probable. The ratio of the number of clauses to the number of variables is set to 4.27. Experimental results suggest that this ratio generates the hardest random 3-CNF formulas [30, 35].

Other satisfiability testers used in the experiments are summarized below.

`walksat` A stochastic model-search procedure. More specifically, it is a greedy hill-climbing search algorithm with probabilistic "back-off" and periodic restart [39]. The procedure is

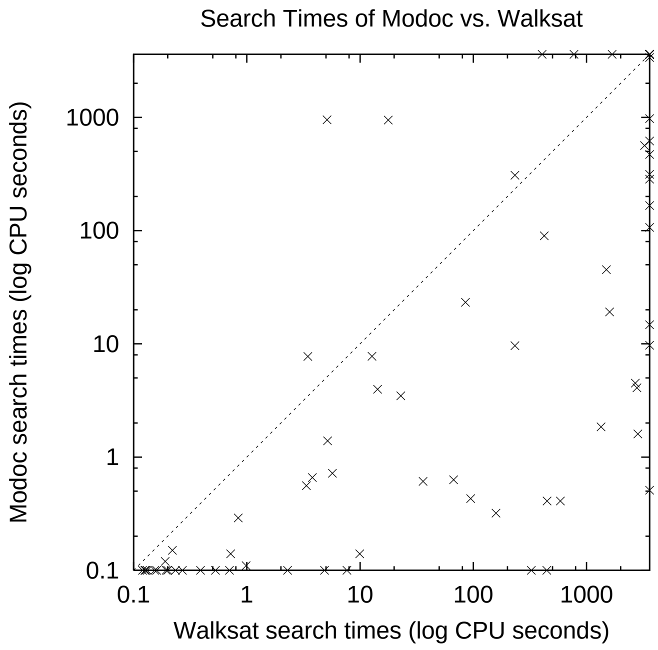


Figure 6.11: Comparison of search times between `modoc` and `walksat`. Formulas are planning formulas generated by Medic and Satplan. All formulas are satisfiable. Times are CPU seconds on an SGI with 150MHz R4400. `walksat` times are average of 5 runs.

incomplete and thus cannot confidently determine unsatisfiability. (See page 2 for a brief discussion on incomplete model-search procedures.)

`dpll` An implementation of DPLL [14, 13].

`C-A`[1] A complete model-search procedure based on DPLL. It incorporates heuristics to select splitting variables and a highly-optimized unit propagation [12].

`2cl` A complete model-search procedure based on DPLL. It incorporates heuristics to select splitting variables and a reasoning capability on short clauses [48].

Since the search times of `walksat` may change from run to run, five runs were made for `walksat` and the average search time is reported.

[1]The algorithm is described as the Tableau algorithm in [12]. However, to avoid confusion with the proof procedure with the same name, it will be referred to as `C-A` in this dissertation.

problem/ deadline	num of vars	num of literals	search time (seconds)			
			`walksat`	`C-A`	`2cl`	`modoc`
logistics.a/11	638	13,089	1	3026	1	1
logistics.c/13	897	21,412	2	??	90	3
bw_large.c/14	2,222	78,146	146	14	12855	15439
bw_large.d/18	4,714	205,559	1494	??	??	58

(a) Search times on the satisfiable Satplan formulas.

problem/ deadline	num of vars	num of literals	search time (seconds)			
			`walksat`	`C-A`	`2cl`	`modoc`
logistics.a/10	541	10,598	—	16019	2	3
logistics.c/12	787	18,244	—	??	11154	1132
bw_large.c/13	1,935	66,547	—	47	2769	5389
bw_large.d/17	4,275	184,180	—	??	??	??

(b) Search times on the unsatisfiable Satplan formulas.

Table 6.1: Search times of various satisfiability testers on the hard planning formulas generated by Satplan. Number of variables and literals are after simplification. Times are CPU seconds on an SGI with 150MHz R4400. `walksat` times are average of 5 runs. '—' indicates that the run was not attempted; this is because `walksat` cannot confidently determine unsatisfiability. '??' indicates that the run was terminated after 5 hours.

6.10.1 Comparison with Other Testers

Figure 6.11 compares `modoc` search times against the average of five `walksat` search times on a large collection of planning formulas generated by Medic [15] and Satplan [28]. Formulas are generated from problems such as Block-World Planning, Tower of Hanoi, Monkey and the Banana, Flat Tire, and Fridge Fixing, and were also used by Ernst et al. [15] and by Kautz and Selman [28]. The deadlines were set to the optimal plan lengths, and thus, all the formulas are satisfiable. (No unsatisfiable formulas were used because `walksat` cannot confidently determine unsatisfiability.) To avoid clutter, formulas that were solved in 0.1 seconds by both programs are not plotted. Plots in the lower-right triangle represent formulas for which `modoc` was faster than `walksat` on average. Plots in the upper-left triangle represent formulas for which `modoc` was slower than `walksat` on average. We observe that `modoc` was faster than `walksat` on the majority of the formulas.

num of checkers	dead-line	num of vars	num of literals	search time (seconds)			
				`walksat`	`C-A`	`2cl`	`modoc`
2	8	105	3,900	1.08	0.09	1.90	0.02
3	15	282	18,294	12564	22446	??	39
4	24	597	55,934	??	??	—	12883

(a) Search times on the satisfiable checker-interchange formulas.

num of checkers	dead-line	num of vars	num of literals	search time (seconds)			
				`walksat`	`C-A`	`2cl`	`modoc`
2	7	90	3,238	—	0.16	2.00	0.12
3	14	261	16,794	—	9427	??	121
4	23	570	53,252	—	??	—	??

(b) Search times on the unsatisfiable checker-interchange formulas.

Table 6.2: Search times of various satisfiability testers on the checker-interchange formulas. Number of variables and literals are after simplification. Times are CPU seconds on an SGI with 150MHz R4400. `walksat` times are average of 5 runs. '—' indicates that the run was not attempted; for `walksat`, this is because it cannot confidently determine unsatisfiability. '??' indicates that the run was terminated after 5 hours; for `walksat`, it means that none of the 5 runs found a solution in 5 hours.

Table 6.1 compares `modoc` search times against search times of `walksat` (average of five runs), `C-A`, and `2cl` on formulas generated by Satplan. The "logistics" formulas are derived from transportation problems, and the "bw_large" formulas are derived from block-world planning problems. The same formulas were also used by Kautz and Selman [28]. With the exception of bw_large.c, we observe that `modoc` is superior to other satisfiability testers. The reason for the poor performance of bw_large.c will be discussed in Section 7.1.

Table 6.2 compares `modoc` search times against the search times of `walksat` (average of five runs), `C-A`, and `2cl` on the checker-interchange formulas. The checker-interchange formulas are planning formulas generated from a game based on the one-dimensional version of Chinese Checkers. Figure 6.12 shows the aim of the checker-interchange problem for 4 checkers; it also shows the possible first few moves. The problem is interesting in that it is believed to have only one plan (by always starting with the black coin to break symmetry) regardless of the number of

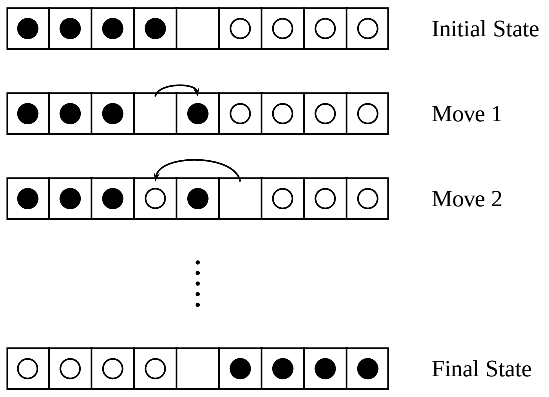


Figure 6.12: Checker-interchange problem for 4 checkers. The aim is to exchange all the black coins on the left with all the white coins on the right. A black coin can only move right and a white coin can only move left. At each step, a coin may either move or jump a coin to occupy the empty space.

checkers. Again, `modoc` was able to outperform other satisfiability testers on both the satisfiable formulas and the unsatisfiable formulas.

Figure 6.13 shows a plot of the search times of `dpll`, `2cl`, and `modoc` on four classes of random formulas: rand050, rand071, rand100, and rand141. Each class consists of 200 formulas, and the number refers to the number of variables. The number of clauses is set to be 4.27 times the number of variables. That is, 214 clauses for rand050, 303 clauses for rand071, 427 clauses for rand100, and 602 clauses for rand141. Experimental results show that this ratio generates the hardest random 3-CNF formulas [30, 35]. Note that the vertical axis uses a log scale. Therefore, the slope indicates the growth rate in the exponent. The plot shows that the growth rate of `modoc` is approximately half-way between `dpll` and `2cl`.

The original DPLL algorithm has been improved over the years by incorporating clever schemes to choose the next splitting variable. One such scheme uses a scoring mechanism to denote the desirability of a variable based on the clauses it appears in. A similar scheme was experimented in `modoc` to choose the next goal node. The respective improvements are shown in Figure 6.14. The amount of improvement in growth rate by `modoc` is much smaller than that by `dpll`. We believe that this is because, in Modoc, the choice of goal nodes is already limited to a small number, i.e., to the literals in the clause nodes, whereas in `dpll`, all unassigned variables are possible choices.

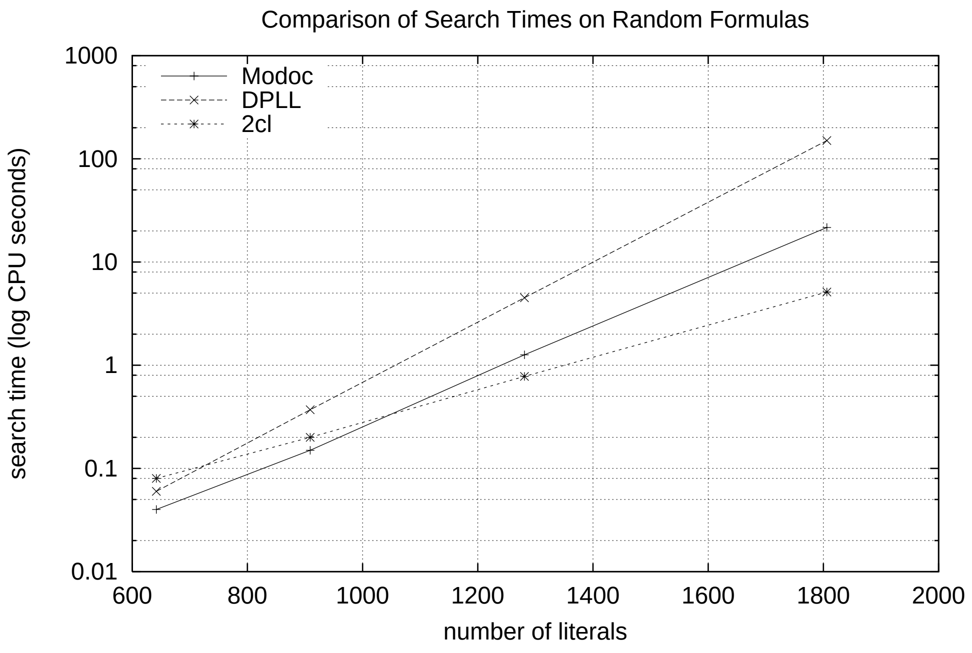


Figure 6.13: Comparison of average search times of various satisfiability testers on random 3-CNF formulas. The formulas are generated at the clauses-to-variables ratio of 4.27, which is believed to generate the hardest random 3-CNF formulas [30, 35]. Search times were obtained on a Sun SparcStation 4/110.

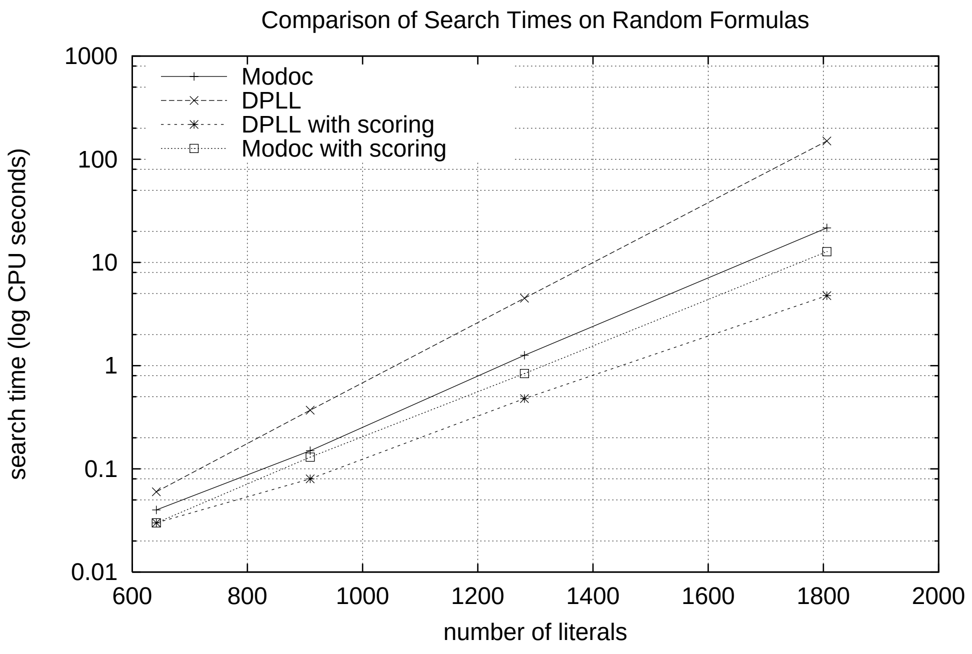


Figure 6.14: Comparison of improvements in average search times by using a scoring scheme. The same formulas used in Figure 6.13 were used.

formula class		sample size	search time (seconds)				speedup
			without APA		standard `modoc`		
			ave.	std.dev.	ave.	std.dev.	
rand100	all	200	1.74	1.12	1.26	0.88	1.38
	sat	109	0.97	0.94	0.70	0.67	1.39
	unsat	91	2.65	0.85	1.92	0.60	1.38
rand141	all	200	34.43	27.83	21.63	17.46	1.59
	sat	117	18.10	21.30	11.30	13.28	1.60
	unsat	83	57.45	17.93	36.19	11.11	1.59
rand200	all	20	3398	2356	1722	1177	1.97
checker.3	sat	1	78.30	-	48.99	-	1.60
	unsat	1	223.65	-	184.84	-	1.21
logistics.a	sat	1	2.09	-	1.26	-	1.66
	unsat	1	8.14	-	4.18	-	1.95
logistics.c	sat	1	5.07	-	6.43	-	0.79
	unsat	1	3404.30	-	1494.94	-	2.28

Table 6.3: Change in search time by articulation-points analysis (APA).

6.10.2 Improvements by Individual Features

This section reports on experimental results obtained to assess the performance improvement made to Modoc by means of enhancements described in this chapter. The results were obtained by running `modoc` with and without each feature on two sets of random formulas—rand100 and rand141. Rand100 consists of 200 randomly generated 3-CNF formulas of 100 variables and 427 clauses. Rand141 consists of 200 randomly generated 3-CNF formulas of 141 variables and 602 clauses. When the speedup for rand141 was larger than that for rand100, a small collection of larger random formulas, called rand200, was used to confirm whether or not this was a trend. Rand200 consists of 20 randomly generated 3-CNF formulas of 200 variables and 854 clauses. Further, to test whether the general trend observed on the random formulas would follow on other classes of formulas, the same experiment was repeated on three of the planning formulas used in Tables 6.1 and 6.2. Unless specified otherwise, the following features were used by default—quasi-persistent lemmas, eager lemmas, articulation-points analysis, lemma-induced cuts, C-reduction-induced cuts, and propagation of refutation during cuts. We call this configuration of `modoc` the *standard `modoc`*. Search times were obtained on a Sun SparcStation 4/110.

formula class		sample size	search time (seconds)				speedup
			without EL, APA		standard `modoc`		
			ave.	std.dev.	ave.	std.dev.	
rand100	all	200	15.04	10.64	1.26	0.88	11.94
	sat	109	8.15	7.92	0.70	0.67	11.64
	unsat	91	23.28	6.99	1.92	0.60	12.13
rand141	all	200	349.16	282.15	21.63	17.46	16.14
	sat	117	178.99	212.99	11.30	13.28	15.84
	unsat	83	589.03	171.64	36.19	11.11	16.28
rand200	all	20	38604	26469	1722	1177	22.42
checker.3	sat	1	1860.36	-	48.99	-	37.97
	unsat	1	3151.34	-	184.84	-	17.05
logistics.a	sat	1	44.17	-	1.26	-	35.06
	unsat	1	53.53	-	4.18	-	12.81
logistics.c	sat	1	98.60	-	6.43	-	15.33
	unsat	1	25535.21	-	1494.94	-	17.08

Table 6.4: Change in search time by eager lemmas (EL) and articulation-points analysis (APA).

Table 6.3 shows the change in search time of `modoc` with and without articulation-points analysis. It shows a moderate speedup, which appears to increase with the size of the formula.

Table 6.4 shows the change in search time of `modoc` with and without the eager lemma strategy and articulation-points analysis. (Note that articulation-points analysis makes no sense without eager lemmas. See Section 6.4 for details.) It shows a significant speedup, which appears to increase with the size of the formula. The speedup due to the addition of eager lemma strategy and articulation-points analysis was the largest among the features tested.

Table 6.5 shows the change in search time of `modoc` with and without lemma-induced cuts. With the exception of the unsatisfiable version of logistics.c, virtually no speedup was observed. Table 6.6 shows the change in the number of extensions and of goal nodes from the same runs. Again, virtually no change in numbers were observed.

Table 6.7 shows the change in search time of `modoc` with and without C-reduction-induced cuts. It shows a small speedup, which appears to increase with the size of the formula.

Table 6.8 shows the change in search time of `modoc` with and without propagating refutation during cuts. Very small speedup was observed on rand100 but virtually none on rand141. Table 6.9 shows the change in the number of extensions and of goal nodes from the same runs.

formula class		sample size	search time (seconds)				speedup
			without LIC		standard `modoc`		
			ave.	std.dev.	ave.	std.dev.	
rand100	all	200	1.26	0.88	1.26	0.88	1.00
	sat	109	0.71	0.68	0.70	0.67	1.01
	unsat	91	1.93	0.60	1.92	0.60	1.01
rand141	all	200	21.71	17.53	21.63	17.46	1.00
	sat	117	11.35	13.33	11.30	13.28	1.00
	unsat	83	36.32	11.19	36.19	11.11	1.00
checker.3	sat	1	49.18	-	48.99	-	1.00
	unsat	1	186.74	-	184.84	-	1.01
logistics.a	sat	1	1.27	-	1.26	-	1.01
	unsat	1	3.80	-	4.18	-	0.91
logistics.c	sat	1	6.47	-	6.43	-	1.01
	unsat	1	2943.52	-	1494.94	-	1.97

Table 6.5: Change in search time by lemma-induced cuts (LIC).

formula class		sample size	ave. num. of extensions			ave. num. of goal nodes		
			without LIC	standard `modoc`	ratio of change	without LIC	standard `modoc`	ratio of change
rand100	all	200	3,615	3,600	1.00	6,214	6,180	1.01
	sat	109	2,108	2,099	1.00	3,522	3,501	1.01
	unsat	91	5,419	5,398	1.00	9,437	9,389	1.01
rand141	all	200	48,504	48,233	1.01	83,926	83,330	1.01
	sat	117	25,661	25,522	1.01	43,961	43,657	1.01
	unsat	83	80,705	80,248	1.01	140,263	139,255	1.01

Table 6.6: Change in the number of extensions and goal nodes by lemma-induced cuts (LIC). This table confirms that the change in search times reported in Table 6.5 for random formulas is consistent with other statistics.

formula class		sample size	search time (seconds)				speedup
			without RIC		standard modoc		
			ave.	std.dev.	ave.	std.dev.	
rand100	all	200	1.53	1.08	1.26	0.88	1.21
	sat	109	0.86	0.83	0.70	0.67	1.23
	unsat	91	2.32	0.74	1.92	0.60	1.21
rand141	all	200	27.37	22.11	21.63	17.46	1.27
	sat	117	14.44	16.94	11.30	13.28	1.28
	unsat	83	45.60	14.34	36.19	11.11	1.26
rand200	all	20	2284	1573	1722	1177	1.33
checker.3	sat	1	74.53	-	48.99	-	1.52
	unsat	1	269.49	-	184.84	-	1.46
logistics.a	sat	1	1.26	-	1.26	-	1.00
	unsat	1	6.46	-	4.18	-	1.55
logistics.c	sat	1	7.20	-	6.43	-	1.12
	unsat	1	3658.82	-	1494.94	-	2.45

Table 6.7: Change in search time by C-reduction-induced cuts (RIC).

Virtually no change in numbers were observed.

Table 6.10 shows the change in search time of `modoc` with and without pure-literal processing. With pure-literal processing, search took more time. Table 6.11 shows the change in the number of extensions and of goal nodes from the same runs. A negligible amount of decrease in numbers were observed. We conclude that the overhead to keep track of the number of occurrences of literals (which is necessary to implement pure-literal processing) simply overwhelmed the small benefit.

Table 6.12 shows the change in search time of `modoc` with and without one-layer lookahead. It shows a very small speedup, which appears to increase with the size of the formula. However, with the exception of the unsatisfiable version of logistics.a, the same trend was not observed on the planning formulas.

formula class		sample size	search time (seconds)				speedup
			without PoR		standard `modoc`		
			ave.	std.dev.	ave.	std.dev.	
rand100	all	200	1.28	0.90	1.26	0.88	1.02
	sat	109	0.72	0.69	0.70	0.67	1.03
	unsat	91	1.95	0.61	1.92	0.60	1.02
rand141	all	200	21.61	17.44	21.63	17.46	1.00
	sat	117	11.29	13.26	11.30	13.28	1.00
	unsat	83	36.17	11.09	36.19	11.11	1.00
checker.3	sat	1	47.79	-	48.99	-	0.98
	unsat	1	180.85	-	184.84	-	0.98
logistics.a	sat	1	1.35	-	1.26	-	1.07
	unsat	1	4.45	-	4.18	-	1.06
logistics.c	sat	1	6.74	-	6.43	-	1.05
	unsat	1	1604.00	-	1494.94	-	1.07

Table 6.8: Change in search time by propagation of refutation (PoR).

formula class		sample size	ave. num. of extensions			ave. num. of goal nodes		
			without PoR	standard `modoc`	ratio of change	without PoR	standard `modoc`	ratio of change
rand100	all	200	3,601	3,600	1.00	6,181	6,180	1.00
	sat	109	2,099	2,099	1.00	3,502	3,501	1.00
	unsat	91	5,399	5,398	1.00	9,390	9,389	1.00
rand141	all	200	48,240	48,233	1.00	83,346	83,330	1.00
	sat	117	25,528	25,522	1.00	43,668	43,657	1.00
	unsat	83	80,256	80,248	1.00	139,277	139,255	1.00

Table 6.9: Change in the number of extensions and goal nodes by propagation of refutation (PoR). This table confirms that the change in search times reported in Table 6.8 for random formulas is consistent with other statistics.

formula class		sample size	search time (seconds)				speedup
			standard `modoc`		with PLP		
			ave.	std.dev.	ave.	std.dev.	
rand100	all	200	1.26	0.88	1.39	0.97	0.91
	sat	109	0.70	0.67	0.78	0.74	0.90
	unsat	91	1.92	0.60	2.12	0.66	0.91
rand141	all	200	21.63	17.46	23.45	18.93	0.92
	sat	117	11.30	13.28	12.25	14.40	0.92
	unsat	83	36.19	11.11	39.23	12.04	0.92
checker.3	sat	1	48.99	-	64.22	-	0.76
	unsat	1	184.84	-	244.87	-	0.75
logistics.a	sat	1	1.26	-	1.40	-	0.90
	unsat	1	4.18	-	5.37	-	0.78
logistics.c	sat	1	6.43	-	21.56	-	0.30
	unsat	1	1494.94	-	2006.90	-	0.74

Table 6.10: Change in search time by pure-literals processing (PLP).

formula class		sample size	ave. num. of extensions			ave. num. of goal nodes		
			standard `modoc`	with PLP	ratio of change	standard `modoc`	with PLP	ratio of change
rand100	all	200	3,600	3,559	1.01	6,180	6,171	1.00
	sat	109	2,099	2,052	1.02	3,501	3,487	1.00
	unsat	91	5,398	5,365	1.01	9,389	9,386	1.00
rand141	all	200	48,233	47,965	1.01	83,330	83,308	1.00
	sat	117	25,522	25,322	1.01	43,657	43,626	1.00
	unsat	83	80,248	79,883	1.00	139,255	139,246	1.00

Table 6.11: Change in the number of extensions and goal nodes by pure-literal processing (PLP). This table shows that the cost of extra bookkeeping to implement PLP overwhelmed the slight reduction in the number of extensions and goal nodes.

formula class		sample size	search time (seconds)				speedup
			standard `modoc`		with LA		
			ave.	std.dev.	ave.	std.dev.	
rand100	all	200	1.26	0.88	1.22	0.85	1.03
	sat	109	0.70	0.67	0.68	0.65	1.03
	unsat	91	1.92	0.60	1.87	0.58	1.03
rand141	all	200	21.63	17.46	19.99	16.09	1.08
	sat	117	11.30	13.28	10.47	12.27	1.08
	unsat	83	36.19	11.11	33.41	10.19	1.08
rand200	all	20	1722	1177	1520	1033	1.13
checker.3	sat	1	48.99	-	85.64	-	0.57
	unsat	1	184.84	-	232.38	-	0.80
logistics.a	sat	1	1.26	-	1.87	-	0.67
	unsat	1	4.18	-	2.63	-	1.59
logistics.c	sat	1	6.43	-	7.87	-	0.82
	unsat	1	1494.94	-	2222.25	-	0.67

Table 6.12: Change in search time by one-layer lookahead (LA).

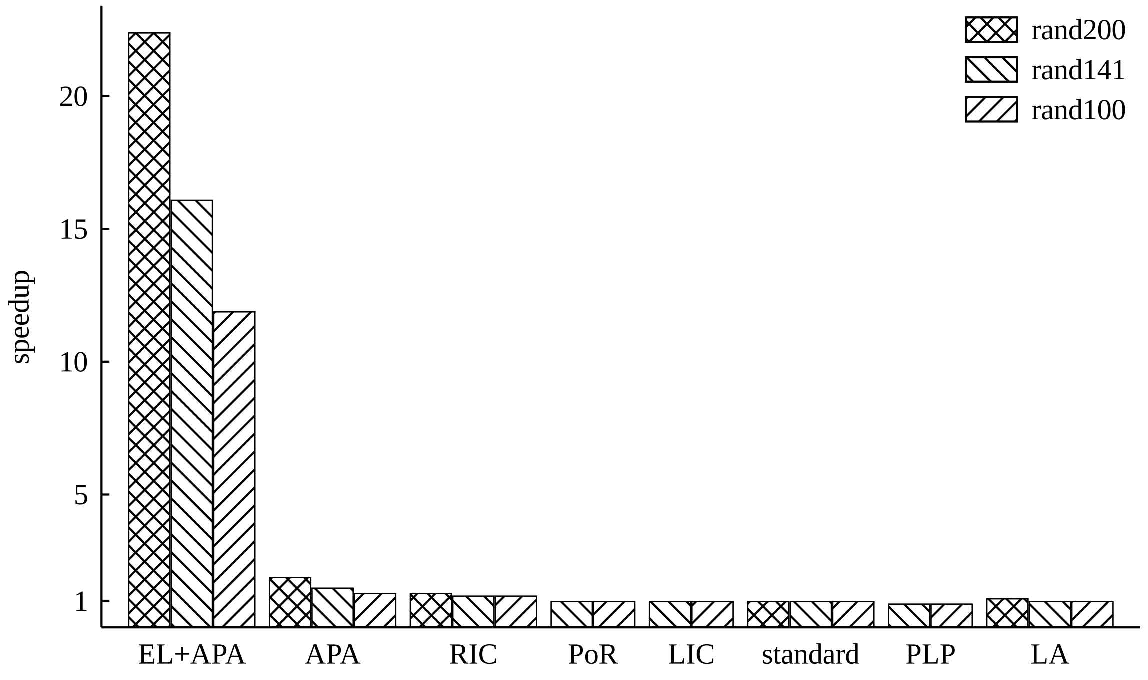


Figure 6.15: Comparison of `modoc` speedup on the random formulas by various features. Abbreviations are explained in Table 6.13.

6.11 Summary

Figure 6.15 summarizes the speedups observed by `modoc` on the random formulas reported in Section 6.10.2. Clearly, the most improvement was made by eager-lemma strategy and articulation-points analysis. (Recall that articulation-points analysis makes no sense without eager lemmas.) This was followed by articulation-points analysis. For these two, the speedup appears to increase with the size of the formula. Figure 6.16 compares the speedup achievable by eager lemmas and articulation-points analysis with other features, namely, lemma-induced cuts, C-reduction-induced cuts, and propagation of refutation during cuts combined. It shows that the speedup achievable by use of other features was negligible compared to the speedup achieved by eager-lemma strategy and articulation-points analysis.

abbrev.	description
EL	eager lemmas (Section 6.3)
APA	articulation-points analysis (Section 6.4)
RIC	C-reduction-induced cuts (Section 6.6)
PoR	propagation of refutation (Section 6.7)
LIC	lemma-induced cuts (Section 6.5)
PLP	pure-literal processing (Section 6.8)
LA	one-layer lookahead (Section 6.9)

Table 6.13: Abbreviations used in Figures 6.15 and 6.16.

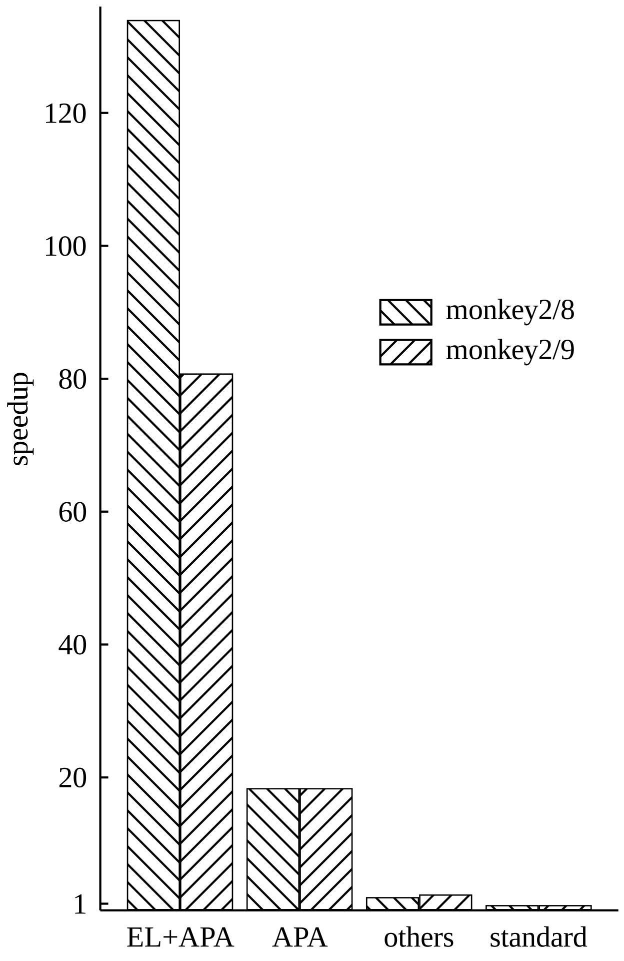


Figure 6.16: Comparison of `modoc` speedup on two planning formulas. Abbreviations are explained in Table 6.13. “others” means lemma-induced cuts, C-reduction-induced cuts, and propagation of refutation during cuts combined.

Chapter 7

Parallel Modoc

This chapter describes *Parallel Modoc*. Parallel Modoc is a multi-agent search procedure that runs (enhanced) Modoc as search agents. Each agent executes Modoc using a different theorem clause as the top clause. If an agent finds a new lemma or a new autarky, it communicates the new lemma or the new autarky to other agents. It is expected that by doing so, other agents may be able to benefit from the lemma and the autarky that it did not derive on its own.

As a parallel satisfiability tester, Parallel Modoc differs from many of the other parallel satisfiability testers reported in the literature. In previous parallel satisfiability testers [41, 49, 6], each process would work independently; communication is used merely to balance the work load. The main use of multiple processing units in these algorithms is to merely increase throughput, i.e., to examine more search nodes in a unit time. However, in Parallel Modoc, there is an equal emphasis on *cooperation* among the multiple processing units (i.e., the agents).

The idea of cooperative search has been tested by others [10, 9, 22]. There, hints that *may* lead to a solution are communicated. That is, there is no guarantee that a hint will be globally applicable. However, Parallel Modoc is different in this regard. It only communicates information that is globally applicable.

Parallel Modoc is *not* a parallelization of the Modoc algorithm but a parallel execution scheme to run multiple cooperating agents, each running the Modoc algorithm.

This chapter consists of several sections. Section 7.1 is independent from the rest of the

theorem clause num	search time
3	911
7	4
8	24
11	745

Table 7.1: Search times of `modoc` on a block-world planning formula (bw_large.c for deadline 14) using different theorem clauses as the top clause. Times are CPU seconds on an SGI 150MHz R4400. The formula has 15 theorem clauses. Theorem clauses not listed exceeded the one-hour time limit.

chapter and discusses the author's experience with Modoc, which led to the development of Parallel Modoc. An initial difficulty in putting together Parallel Modoc was the possibility of conflicting autarkies (Definition 2.3). Section 7.2 describes a number of properties regarding multiple autarkies, and describes two algorithms, one to combine two arbitrary autarkies, and another, an optimization of the first algorithm. Section 7.3 describes some aspects of the implementation of Parallel Modoc (which we denote by `pmodoc`), particularly about the mechanism used to communicate autarkies and lemmas. Because the number of theorem clauses is dictated by the formula, this effectively limits the degree of parallel search. Section 7.4 describes two ways to increase the number of clauses that are suitable for use as the top clause. Section 7.5 reports experimental results.

7.1 Motivation

This section discusses the author's experience with Modoc that led to the design and development of Parallel Modoc. In particular, it discusses the extreme skewness of the distribution of Modoc search times observed using different theorem clauses as the top clause.

Goal-sensitive search (see Section 2.3) has generally been a success for Modoc. However, there remains a problem of which theorem clause to try as the first top clause when there are multiple theorem clauses. It is quite likely that many of the theorem clauses would lead Modoc to determine the satisfiability of the formula. However, the author's experience using Modoc suggests that the search times are extremely skewed depending on which theorem clause was chosen as the top clause. An implication of this is that starting from a "bad" top clause could make the formula appear to be

fmla	num goal nodes				
num	ave	std.dev.	min	max	max/min
1	2,694	2,549	175	9,729	55.6
2	10,804	2,553	4,813	16,042	3.4
3	4,787	3,103	737	12,118	16.5
4	8,547	1,719	5,599	13,099	2.4
5	3,227	2,376	104	8,410	80.9
6	3,166	2,740	140	10,966	78.4
7	4,680	4,069	71	17,880	251.9
8	4,766	2,860	888	10,640	12.0
9	9,919	2,079	4,669	14,531	3.2
10	1,707	1,588	186	6,144	33.1

Table 7.2: Number of goal nodes examined by `modoc` on random formulas using different clauses as the top clause. Formulas contain 100 variables and 427 clauses. For each formula, 1/10th of the clauses were sampled and used as the top clause. (A separate study shows an extremely high linear correlation (> 0.99) between search time and the number of goal nodes examined for this class of formulas.)

too difficult to solve in practice.

Table 7.1 shows how `modoc` search times vary on one planning formula. It shows that while `modoc` was not able to determine the satisfiability of the formula in one hour using many of the theorem clauses as the top clause, it was able to do so in an amazing four seconds using the seventh theorem clause. This means that there is at least a factor of 900 between the longest and the shortest search times for this formula. (A separate study shows that running `modoc` using the first theorem clause as the top clause took 15439 seconds. This means that the factor is at least 3800.) By default, `modoc` uses the first theorem clause as the top clause.

The skewness of search times does not appear to be a behavior specific to structured formulas. Random formulas, considered to have no structure, exhibit such behaviors, albeit in a smaller magnitude, as well. Tables 7.2 and 7.3 show the distribution of the number of goal nodes examined by `modoc` using different clauses as the top clause on random formulas. Because all clauses are equally eligible as the top clause in random formulas, only one tenth of the clauses were sampled and used as the top clause. Further, because some of the runs were reported to have run in 0.00 seconds, we report the number of goal nodes examined instead. A separate study shows that the search time and the number of goal nodes examined have an extremely high linear correlation

fmla	num goal nodes				
num	ave	std.dev.	min	max	max/min
1	26,319	46,394	120	240,244	2002.1
2	115,692	21,737	76,361	161,692	2.2
3	46,131	35,628	214	125,072	584.5
4	132,626	24,926	79,797	177,924	2.3
5	11,762	16,161	139	63,928	460.0
6	47,057	42,747	1,059	173,864	164.2
7	85,690	19,480	48,488	138,229	2.9
8	46,321	52,336	306	184,504	603.0
9	64,340	50,052	4,993	250,536	50.2
10	43,404	37,167	1,095	149,284	136.4

Table 7.3: Number of goal nodes examined by `modoc` on random formulas using different clauses as the top clause. Formulas contain 141 variables and 602 clauses. For each formula, 1/10th of the clauses were sampled and used as the top clause. (A separate study shows an extremely high linear correlation (> 0.99) between search time and the number of goal nodes examined for this class of formulas.)

(> 0.99). The tables show that the factor between the most and the least number of goal nodes examined ranged from 2.4 to 251.9 on the ten formulas with 100 variables and 427 clauses, and from 2.2 to 2002.1 on the ten formulas with 141 variables and 602 clauses. It appears that the range of factors widens with the increase in the size of the formula.

Because of the extreme skewness of Modoc search times depending on the choice of the top clause, which could make a formula appear to be too difficult to solve in practice, for Modoc to be a practical solver, it is important that Modoc be able to choose a “good” top clause and not a “bad” top clause. At this time, we have no means to correctly distinguish good top clauses from bad top clauses, and it is doubtful that we would some day be able to correctly do so without actually solving the formula. Not knowing which ones are “good” and which ones are “bad”, one obvious solution to this would be to simply run as many Modoc processes as there are theorem clauses, giving each process a different theorem clause. This was the beginning of Parallel Modoc. However, as a research project, the author was interested in knowing whether it is possible to improve the performance beyond a simple parallel execution.

7.2 Multiple Autarkies

This section first describes properties found regarding multiple autarkies and then describes two algorithms. The first algorithm combines two arbitrary autarkies to form another autarky that is no smaller than the two original autarkies, and the second algorithm is an optimization of the first algorithm which allows only the difference from the last autarky found to be transmitted.

In general, dealing with multiple autarkies is not straightforward. This is because a formula may have conflicting autarkies (Definition 2.3). Although a single Modoc cannot derive conflicting autarkies by itself, when multiple agents are executing Modoc and making multiple searches at the same time, it is possible for the multi-agent search procedure to derive conflicting autarkies. Without a means to combine them, it would require each agent to store multiple autarkies separately, which may quickly become a bookkeeping nightmare. Theorem 7.1 shows that there is a simple way to combine any two autarkies to form a new autarky that satisfies exactly the same set of clauses satisfied by either of the two given autarkies.

Before we present the theorem, and hence the algorithm, we introduce a new operator over the set of partial truth assignments.

Definition 7.1 Let A_1 and A_2 be partial truth assignments. Then, we define $A_1 \curvearrowleft A_2$ as

$$A_1 \curvearrowleft A_2 = A_1 \cup (A_2 - \overline{A_1})$$

where $\overline{A_1} = \{\neg x \,|\, x \in A_1\}$. We will say that A_1 is given *preference* over A_2 in resolving conflicting assignments. $\square$

Example 7.1 This example illustrates the use of the operator introduced in Definition 7.1.

Let two partial truth assignments A_1 and A_2 be as follows:

$$\begin{aligned} A_1 &= \{u_1, \ldots, u_m, w_1, \ldots, w_k\}; \\ A_2 &= \{v_1, \ldots, v_n, \neg w_1, \ldots, \neg w_k\}. \end{aligned}$$

That is, only the variables in $\{w_1, \ldots, w_k\}$ have different polarities in A_1 and A_2. (Note that some of the u_is may be v_js and vice versa.) Then,

$$A_1 \curvearrowleft A_2 = \{u_1, \ldots, u_m, v_1, \ldots, v_n, w_1, \ldots, w_k\},$$

$$A_2 \curvearrowleft A_1 \quad = \quad \{u_1, \ldots, u_m, v_1, \ldots, v_n, \neg w_1, \ldots, \neg w_k\}.$$

$\square$

Note that, in general, $(A_1 \curvearrowleft A_2) \neq (A_2 \curvearrowleft A_1)$. Also, while $A_1 \subseteq (A_1 \curvearrowleft A_2)$, it is generally the case that $A_2 \not\subseteq (A_1 \curvearrowleft A_2)$.

Theorem 7.1 Let two autarkies of a CNF formula F be as follows:

$$\begin{aligned} A_1 &= \{u_1, \ldots, u_m, w_1, \ldots, w_k\}; \\ A_2 &= \{v_1, \ldots, v_n, \neg w_1, \ldots, \neg w_k\}. \end{aligned}$$

That is, only the variables in $\{w_1, \ldots, w_k\}$ have different polarities in A_1 and A_2. (Note that some of the u_is may be v_js and vice versa.) Then, the following statements are true:

1. Both $A_1 \curvearrowleft A_2$ and $A_2 \curvearrowleft A_1$ are autarkies of F.

2. Both $A_1 \curvearrowleft A_2$ and $A_2 \curvearrowleft A_1$ satisfy exactly the same set of clauses as the set of clauses satisfied by either A_1 or A_2.

Proof: We only prove for $A_1 \curvearrowleft A_2$. The other case can be shown to be true by symmetry.

1. It is sufficient to show that any clause that contains $\neg u_i$ or $\neg v_i$ or $\neg w_i$ for some i also contains a literal that is in $A_1 \curvearrowleft A_2$.

   Let C be a clause that contains $\neg w_i$ for some i. Since A_1 is an autarky that contains w_i, A_1 satisfies C. This means that there is some literal x in A_1 that is also in C. Since $A_1 \curvearrowleft A_2$ is a superset of A_1, x must also be in $A_1 \curvearrowleft A_2$. A similar argument can be made for a clause that contains $\neg u_i$ for some i.

   Let C be a clause that contains $\neg v_i$ for some i. Since A_2 is an autarky that contains v_i, A_2 satisfies C. This means that there is some literal x in A_2 that is also in C. There are two cases to consider: (1) x is v_j for some j, and (2) x is $\neg w_j$ for some j. In the first case, we are done as v_j is also in $A_1 \curvearrowleft A_2$. The second case reduces to a case in the previous paragraph.

2. It is sufficient to show that set containments hold both ways between the sets of clauses satisfied by the autarkies.

   Let C be a clause that is satisfied by $A_1 \curvearrowleft A_2$. This means that there is some literal x in C that is also in $A_1 \curvearrowleft A_2$. There are three cases to consider: (1) x is u_i for some i, (2) x is v_i for some i, and (3) x is w_i for some i. In the first and third cases, x is also in A_1, and thus, C is satisfied by A_1. In the second case, x is also in A_2, and thus, C is satisfied by A_2.

   Let C be a clause that is satisfied by A_1. This means that there is some literal x in C that is also in A_1. Since $A_1 \curvearrowleft A_2$ is a superset of A_1, x must also be in $A_1 \curvearrowleft A_2$. Thus, C is satisfied by $A_1 \curvearrowleft A_2$.

   Let C be a clause that is satisfied by A_2. This means that there is some literal x in C that is also in A_2. There are two cases to consider: (1) x is v_i for some i, and (2) x is $\neg w_i$ for some i. In the first case, x is also in $A_1 \curvearrowleft A_2$, and thus, C is satisfied by $A_1 \curvearrowleft A_2$. In the second case, since $A_1 \curvearrowleft A_2$ is an autarky that contains w_i, C is satisfied by $A_1 \curvearrowleft A_2$.

□

The use of autarkies in Modoc is to eliminate certain clauses whose use in the PDT extension operation cannot lead the search to a successful subrefutation. In this respect, Theorem 7.1 has the following implication.

Corollary 7.1 For the purpose of autarky pruning, the algorithm described in Theorem 7.1 does not lose any pruning information. □

As a practical concern, particularly in a distributed computing environment where communication is made over a (relatively slow) computer network, transmitting new autarkies as they are found may be costly, as they tend to be large in practice. Theorem 7.2 shows that it is sufficient to transmit only the new autarky literals found since the last transmission.

Theorem 7.2 Let A_1, A_2, and A_3 be autarkies for a formula such that $A_2 \subseteq A_3$. Then,

1. $(A_1 \curvearrowleft A_2) \curvearrowleft A_3 = (A_1 \curvearrowleft A_2) \curvearrowleft (A_3 - A_2)$,

2. $A_3 \curvearrowleft (A_2 \curvearrowleft A_1) = (A_3 - A_2) \curvearrowleft (A_2 \curvearrowleft A_1)$.

Proof: We only prove 1. The same approach could be used to prove 2.

Let A_1, A_2, and A_3 be as follows:

$$\begin{aligned} A_1 &= \{u_1, \ldots, u_m, w_1, \ldots, w_k\}; \\ A_2 &= \{v_1, \ldots, v_{n'}, \neg w_1, \ldots, \neg w_{k'}\}; \\ A_3 &= \{v_1, \ldots, v_{n'}, \ldots, v_n, \neg w_1, \ldots, \neg w_{k'}, \ldots, \neg w_k\}. \end{aligned}$$

That is, only the variables in $\{w_1, \ldots, w_k\}$ have different polarities in A_1 and A_3. (Note that some of the u_is may be v_js and vice versa.) Then,

$$\begin{aligned} (A_1 \curvearrowleft A_2) \curvearrowleft A_3 &= (\{u_1, \ldots, u_m, w_1, \ldots, w_k\} \curvearrowleft \{v_1, \ldots, v_{n'}, \neg w_1, \ldots, \neg w_{k'}\}) \\ &\quad \curvearrowleft \{v_1, \ldots, v_n, \neg w_1, \ldots, \neg w_k\} \\ &= \{u_1, \ldots, u_m, v_1, \ldots, v_{n'}, w_1, \ldots, w_k\} \curvearrowleft \{v_1, \ldots, v_n, \neg w_1, \ldots, \neg w_k\} \\ &= \{u_1, \ldots, u_m, v_1, \ldots, v_n, w_1, \ldots, w_k\}, \end{aligned}$$

$$\begin{aligned} (A_1 \curvearrowleft A_2) \curvearrowleft (A_3 - A_2) &= (\{u_1, \ldots, u_m, w_1, \ldots, w_k\} \curvearrowleft \{v_1, \ldots, v_{n'}, \neg w_1, \ldots, \neg w_{k'}\}) \\ &\quad \curvearrowleft (\{v_1, \ldots, v_n, \neg w_1, \ldots, \neg w_k\} - \{v_1, \ldots, v_{n'}, \neg w_1, \ldots, \neg w_{k'}\}) \\ &= \{u_1, \ldots, u_m, v_1, \ldots, v_{n'}, w_1, \ldots, w_k\} \\ &\quad \curvearrowleft \{v_{n'+1}, \ldots, v_n, \neg w_{k'+1}, \ldots, \neg w_k\} \\ &= \{u_1, \ldots, u_m, v_1, \ldots, v_n, w_1, \ldots, w_k\}. \end{aligned}$$

□

Remark 7.1 It should be noted that when only the difference is transmitted between two agents, as described in Theorem 7.2, the same agent must always be given preference in resolving conflicting assignments. A counter-example can be constructed if this is not followed. See Example 7.2. □

Example 7.2 This example illustrates that, between two agents, if the same agent is not given preference in resolving conflicting assignments, it is possible to end up with a partial truth assignment that is not an autarky.

Let the formula F be $\{[x], [y,v], [z], [u], [w], [t]\}$. Suppose there are two agents α_1 and α_2, and that while agent α_1 finds autarky $A_1 = \{x, \neg y, u, v, w\}$, agent α_2 finds autarky $A_2 = \{x, y, z\}$ and then later finds autarky $A_3 = \{x, y, z, u, \neg v, t\}$.

Now, consider the following sequence of autarky combining:

1. Combine A_1 and A_2, giving preference to agent α_1 to resolve conflicting assignments. Let A_{12} denote the resulting partial truth assignment.

2. Combine $A_3 - A_2$ and A_{12}, giving preference to agent α_2 to resolve conflicting assignments. Let A_{312} denote the resulting partial truth assignment.

Then,

$$\begin{aligned}
A_{12} &= A_1 \curvearrowleft A_2 \\
&= \{x, \neg y, u, v, w\} \curvearrowleft \{x, y, z\} \\
&= \{x, \neg y, z, u, v, w\}, \\
A_{312} &= (A_3 - A_2) \curvearrowleft A_{12} \\
&= (\{x, y, z, u, \neg v, t\} - \{x, y, z\}) \\
&\quad \curvearrowleft \{x, \neg y, z, u, v, w\} \\
&= \{u, \neg v, t\} \curvearrowleft \{x, \neg y, z, u, v, w\} \\
&= \{x, \neg y, z, u, \neg v, w, t\}.
\end{aligned}$$

The partial truth assignment A_{312} is not an autarky because clause $[y, v]$ contains a false literal but it does not contain a true literal. $\square$

7.3 Implementation

This section describes some aspects of the current implementation of Parallel Modoc. The implementation will hereinafter be referred to as `pmodoc`. Section 7.3.1 describes the "blackboard" that is used by the Modoc agents to communicate lemmas and autarkies that are found during their search.

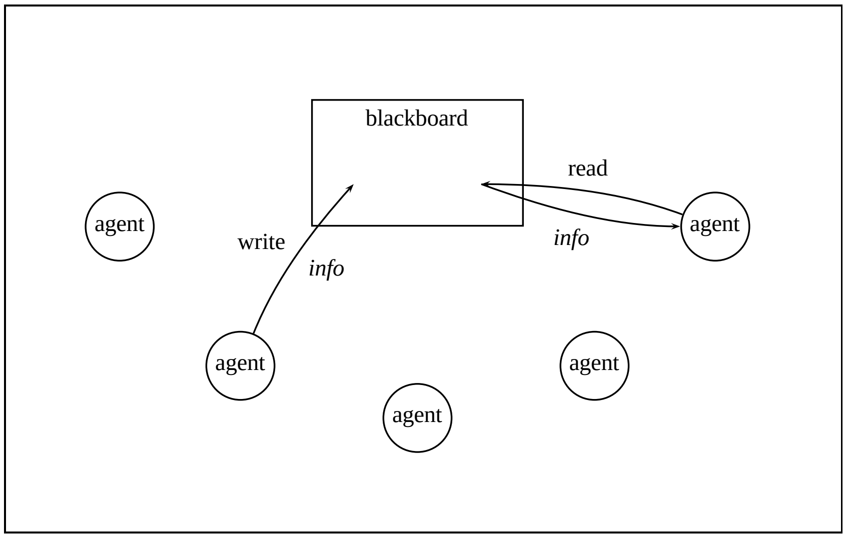


Figure 7.1: A blackboard is used to communicate new autarkies and lemmas between agents in `pmodoc`.

Section 7.3.2 discusses how the implementation deals with *conditional* autarkies (Section 2.3). Sections 7.3.3 and 7.3.4 describe how lemmas and autarkies are communicated among the agents using the blackboard, respectively

7.3.1 Blackboard

The current implementation uses a “blackboard” to communicate between different agents. As shown in Figure 7.1, a blackboard is a shared resource to which agents may write new information and from which agents may obtain new information. In `pmodoc`, it is implemented as System V shared memory segments. When a Modoc agent finds a new autarky or a new lemma, it writes the new autarky or the new lemma to the blackboard. Other agents may, at their convenience, obtain the new autarky or the new lemma from the blackboard and incorporate it into their collection of autarkies and lemmas.

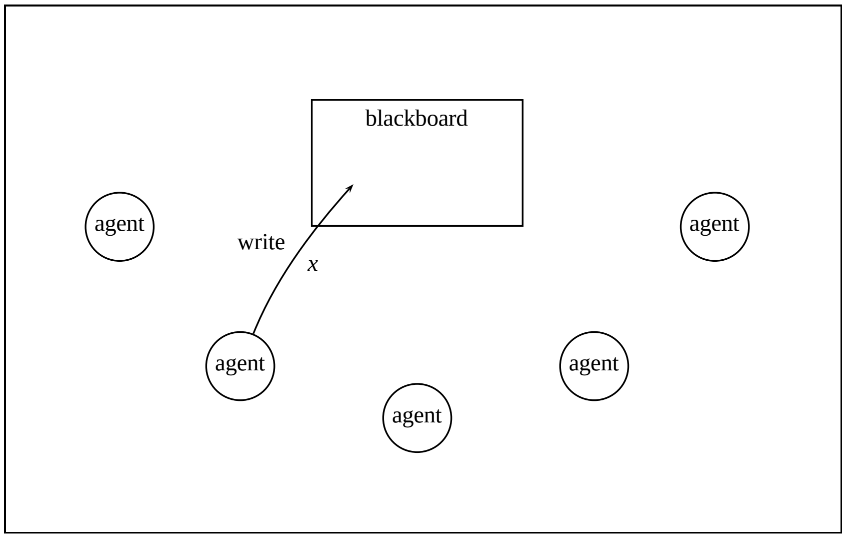
At this time, only the autarkies and lemmas that are attached to the verum $\top$ are communicated. Such autarkies and lemmas are called *top-level* autarkies and *top-level* lemmas, respectively. Top-level autarkies and lemmas have no “premise” under which they hold true (i.e., they are *always* true), thus allowing immediate use by other agents.

The blackboard is set up so that information written to the blackboard is never retracted or revised. While this would not pose any problem for lemmas, it may for autarkies because of conflicting autarkies. In `pmodoc`, this is solved by giving the blackboard preference over all the agents. This also satisfies the condition stipulated in Remark 7.1, which is necessary to use the algorithm described in Theorem 7.2.

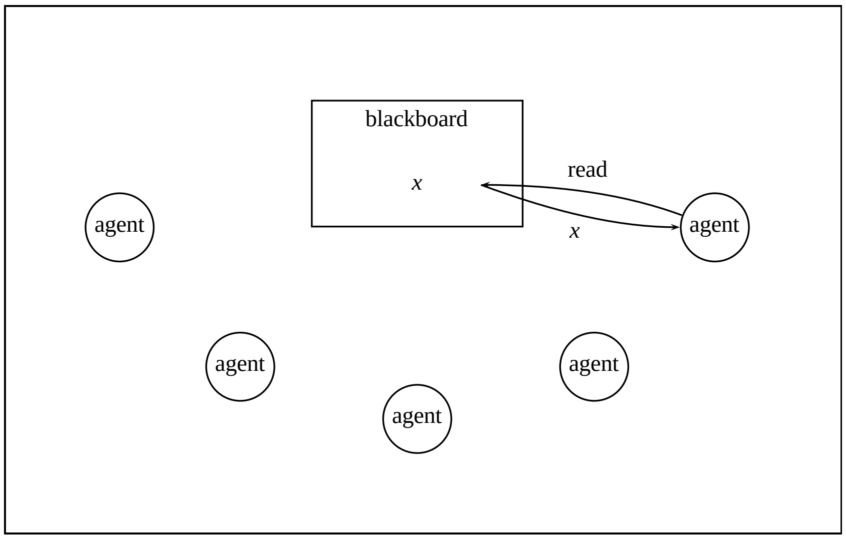
7.3.2 Dealing with Conditional Autarky

Section 7.2 discussed the possibility of conflicting autarkies in Parallel Modoc and presented a solution as Theorem 7.1. The theorem described an algorithm to combine two *unconditional* autarkies. However, in practice, when a new unconditional autarky is communicated to an agent, it is very likely for the agent to be dealing with a *conditional* autarky, that is, an "autarky" that is an autarky for the formula resulting from strengthening the input formula with the current set of ancestor goal nodes. Thus, it is further necessary to devise a mechanism to combine an unconditional autarky with a possibly conditional autarky.

Two solutions are possible, each corresponding to which of the two autarkies gets preference. The current implementation of Parallel Modoc rejects the idea of giving preference to the conditional autarky, which is the local autarky, for the following reason. Let A_u denote the unconditional autarky that is being communicated to the agent, and A_c denote the possibly conditional autarky that is the current autarky being dealt with by the agent. Suppose there is a literal x where x is in A_u and $\neg x$ is in A_c. If we were to give preference to A_c over A_u, then x would be discarded in favor of $\neg x$. Let q denote the goal node to which the autarky literal $\neg x$ is attached within the agent. If the refutation of goal node q succeeds, then all autarky literals attached to this goal node and any descendent goal nodes will be discarded. This would mean that the agent would lose $\neg x$ from its current autarky. By then, the x that the agent discarded in favor of $\neg x$ is long gone. However, if we were to give preference to A_u over A_c, then none of this could happen. Because of this, `pmodoc` gives preference to the unconditional autarky over the possibly conditional autarky. This, however, requires the Modoc agent to possibly revise its current autarky and hence partially back out from its current search. Further details are given in Section 7.3.4.



(a) A Modoc agent finds a new lemma x and writes it to the blackboard.



(b) A different Modoc agent incorporates the new lemma x into its collection of lemmas.

Figure 7.2: Communicating lemmas in `pmodoc`.

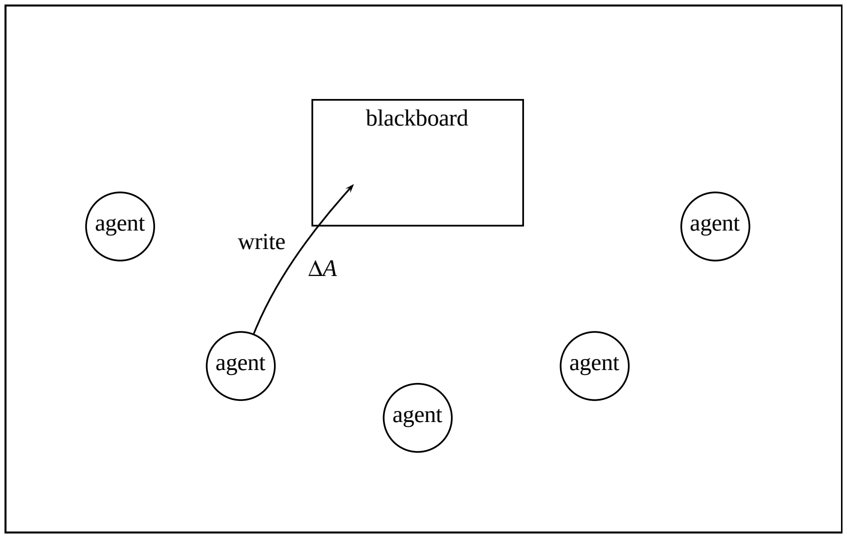
7.3.3 Communicating Lemmas

Figure 7.2 shows how an agent may communicate a new top-level lemma to other agents in `pmodoc`. Note that a top-level lemma consists only of a single literal (as it has no "premise"). When an agent derives a new top-level lemma x, it writes x to the blackboard (Figure 7.2(a)). Other agents, at their convenience, will incorporate x into their collection of top-level lemmas (Figure 7.2(b)). The new lemma could then be used as if it had been derived locally by the agent. Further, if the agent is currently attempting a refutation of a goal node labeled with $\neg x$, the search could be backed up to the parent clause node of the goal node, as the goal node is now obviously refutable.

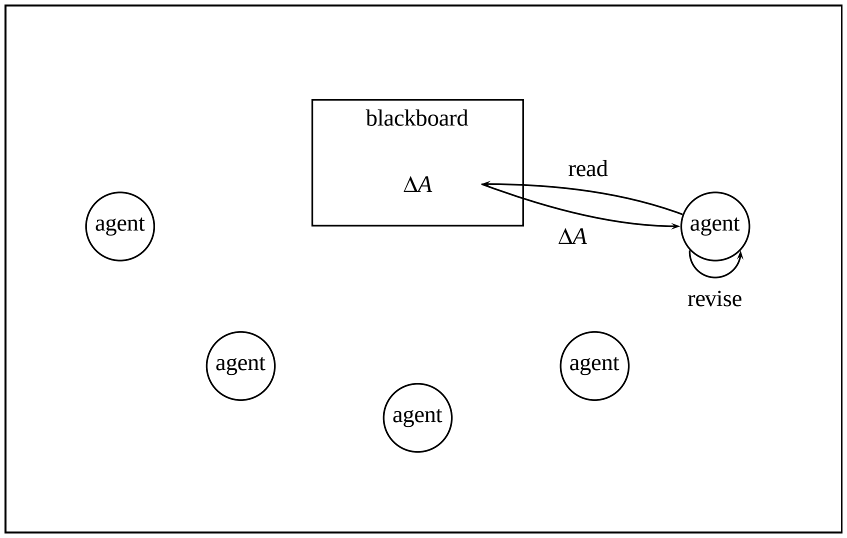
7.3.4 Communicating Autarkies

Figure 7.3 shows how an agent may communicate a new top-level autarky to other agents in `pmodoc`. When an agent finds a new top-level autarky, it combines it with the autarky already in the blackboard (Figure 7.3(a)). Because it is possible for the two autarkies to be conflicting autarkies (Definition 2.3), the algorithm described in Theorem 7.1 is used to combine the new autarky with the autarky already in the blackboard. The preference is given to the autarky already in the blackboard. To avoid transmitting the whole autarky every time a new autarky is found, the algorithm described in Theorem 7.2 is used. Since the autarky already in the blackboard is given preference in resolving conflicting assignments, the condition stipulated in Remark 7.1 is also satisfied.

Other Modoc agents, at their convenience, will incorporate the new autarky literals by possibly revising their current autarky (Figure 7.3(b)). The enlarged autarky could then be used as if it had been derived locally by the agent. Further, if the agent is currently attempting a refutation that has performed an extension with a clause now satisfied by the enlarged autarky, the refutation attempt can be backed up to the parent of the highest such clause, as this subrefutation cannot succeed.



(a) A Modoc agent finds a new autarky. It writes to the blackboard the new autarky literals that do not conflict with the autarky in the blackboard. The agent must revise its autarky if there is a conflict.



(b) A different Modoc agent incorporates the new autarky literals by possibly revising its autarky.

Figure 7.3: Communicating autarkies in `pmodoc`.

7.4 Increasing the Number of Goal Clauses

A problem with the current design of Parallel Modoc is that the number of theorem clauses in the formula limits the maximum number of Modoc agents Parallel Modoc can utilize. This is unfortunate because it would mean that the degree of parallel search will be dictated by the formula.

This section introduces the notion of *goal clauses* as the clauses that are suitable for use as a top clause. So far, the set of goal clauses and the set of theorem clauses have been identical. The idea that will be presented in this section is to construct a set of goal clauses whose number will likely be greater than the number of theorem clauses, to allow a higher degree of parallel search.

To investigate the potential improvement by increasing the number of goal clauses, two simple methods were considered. Both methods use resolution (Definition 1.12). The first method, called *clause exhaustion* (CE), generates all possible resolvents involving the theorem clauses, deletes the theorem clauses, and adds the resolvents as the goal clauses. The second method, called *variable exhaustion* (VE), generates all possible resolvents over the variables used in the theorem clauses, deletes all the clauses that were used in resolution, adds the resolvents, and makes the resolvents that were derived from a theorem clause the goal clauses. Example 7.3 illustrates the two methods.

Example 7.3 This example illustrates how clause exhaustion and variable exhaustion may allow construction of a set of goal clauses whose number is likely to be greater than the number of theorem clauses. Consider the formula

$$\{\underline{[a,b]},[a,\neg c,\neg f],[\neg a,b,c],[\neg a,\neg c,\neg e],[b,d],[\neg b,c],[\neg b,d],[e,f]\},$$

where the only theorem clause is $[a,b]$ (shown above with an underline).

The clause-exhaustion method will first derive all the resolvents involving the theorem clause $[a,b]$. They are

$$[a,c],[a,d],[b,c],[b,\neg c,\neg e].$$

The new formula would then be obtained by removing the theorem clause $[a,b]$ from the formula and adding the resolvents as the goal clauses, as

$$\{\underline{[a,c]},\underline{[a,d]},[a,\neg c,\neg f],[\neg a,b,c],[\neg a,\neg c,\neg e],\underline{[b,c]},\underline{[b,\neg c,\neg e]},[b,d],[\neg b,c],[\neg b,d],[e,f]\}.$$

The variable-exhaustion method will first derive all the resolvents over the variables that appear in the theorem clause, in this case, *a* and *b*. They are

$$\underline{[a,c]}, \underline{[a,d]}, [\neg a, c], [\neg a, c, d], \underline{[b,c]}, [b, c, \neg c, \neg f], \underline{[b, \neg c, \neg e]}, [c,d], [\neg c, \neg e, \neg f], [d].$$

Of the above clauses, the four underlined clauses were derived from the theorem clause. The new formula is then obtained by removing all the clauses that contain either *a* or *b* from the formula, and adding the resolvents.

$$\{\underline{[a,c]}, \underline{[a,d]}, [\neg a, c], [\neg a, c, d], \underline{[b,c]}, [b, c, \neg c, \neg f], \underline{[b, \neg c, \neg e]}, [c,d], [\neg c, \neg e, \neg f], [d], [e,f]\}$$

The resolvents that were derived from the theorem clause are used as the goal clauses (shown underlined above). □

Initial experiments suggest that many of the resolvents are either tautologous (page 5) or subsumed (Definition 1.11) by other clauses in the formula. Thus, before running Parallel Modoc, formulas generated by both methods were subjected to the removal of tautologous clauses and limited subsumption.

7.5 Experimental Results

This section compares the performance of `pmodoc` against other satisfiability testers. Results were obtained by experiments on various planning formulas, circuit-diagnosis formulas, pigeon-hole formulas, and random formulas. The origin of the planning formulas was described in Section 6.10. Other satisfiability testers used in the experiments are `walksat`, `2cl`, and `modoc`. Both `walksat` and `2cl` were briefly described in Section 6.10.

In summary, the results show that `pmodoc` is able to improve `modoc` via simultaneous execution of multiple searches, sometimes aided by cooperation among the Modoc agents. However, for simple formulas, the overhead of parallel execution tends to dominate the search cost. Improvement due to communicating autarkies and lemmas varies from formula to formula.

All experiments in this section were performed on an SGI with four 150MHz R4400 processors. Each Modoc agent was executed as a separate Unix process.

In reporting search times for `pmodoc`, depending on the purpose, either the total elapsed time and/or the elapsed time per agent were computed and reported. The former allows us to compare `pmodoc` to other satisfiability testers with regards to its use of computing resource. The latter allows us to estimate the potential of `pmodoc` on a parallel computer with as many CPUs as there are goal clauses.

7.5.1 Improvements by Parallel Search and Communicating Information

Planning Formulas

Table 7.4 shows the search times on hard planning formulas generated by Medic. (Medic may generate formulas in a number of different encodings. The encoding has been included in the tables to help identify the formula. This dissertation does not discuss encoding issues; explanation of these encodings may be found elsewhere [15].) The problems were selected based on an earlier study in which the corresponding satisfiable formulas caused `modoc` to time out after 600 seconds. (Because of this selection criteria, the search times of `modoc` in this table are not a fair representation of its capabilities.)

In many cases, the speedup by `pmodoc` was far greater than the number of agents. Formulas that could not be solved in one hour using other satisfiability testers were often solved in the order of minutes, and in some cases, in seconds. This shows that `pmodoc` benefits greatly from the parallel searches and not simply from the increase in the number of processors.

Communicating autarkies did not help, in the sense that when an autarky was found, it was actually a satisfying truth assignment, and thus, no further search was necessary. Communication of lemmas occurred on all the formulas, but no correlation appears to exist between the number of lemmas communicated (not shown) and the amount of improvement in search time. Actually, this was expected. A derivation of a lemma only implies a *potential* to save time, not a guarantee. Unless a goal node labeled with the complement of the lemma literal is created, the lemma is of no value at all. In fact, a small overhead to derive and record the lemma must be incurred at the time of derivation, making lemma strategies costly if the lemmas are never (or rarely) used. For each problem, the unsatisfiable formula generally had more lemmas communicated than the satisfiable

problem/ deadline	#TC	en- cod- ing	num of vars	num of literals	search time (seconds)						
					`walksat`	`2cl`	`modoc`	`pmodoc`			
								no comm		comm	
big-bw1/11	6	ccse	980	77,629	?	?	?	?		?	
		ecse	798	14,090	5	28	951	16	10	16	10
fridge2/13	2	cbse	180	10,485	?	?	?	25	25	24	24
		ccse	346	10,370	?	?	?	78	78	78	78
		crse	310	9,880	774	?	?	582	582	581	581
hanoi3/7	3	cbse	158	25,468	?	244	618	98	98	70	70
monkey2/9	2	cbse	250	37,696	?	18	?	?		?	
		cfst	331	14,465	?	686	974	909	909	387	387
		crse	601	48,757	?	?	?	?		?	
tire2/14	6	ccse	677	41,343	?	?	3375	164	110	83	55
		cfst	623	34,944	?	?	?	?		?	
		crse	628	40,002	?	?	?	?		1281	854
		efst	512	19,309	1917	478	?	3	2	3	2

(a) Search times on the satisfiable hard planning formulas generated by Medic.

problem/ deadline	#TC	en- cod- ing	num of vars	num of literals	search time (seconds)						
					`walksat`	`2cl`	`modoc`	`pmodoc`			
								no comm		comm	
big-bw1/10	6	ccse	888	69,851	—	?	?	?		?	
		ecse	707	12,375	—	231	222	223	149	115	76
fridge2/12	2	cbse	166	9,616	—	?	?	216	216	108	108
		ccse	318	9,492	—	?	?	2113	2113	1869	1869
		crse	285	9,043	—	?	?	1528	1528	444	444
hanoi3/6	3	cbse	135	21,388	—	104	79	45	45	26	26
monkey2/8	2	cbse	222	33,194	—	2365	?	?		1701	1701
		cfst	291	12,532	—	364	301	302	302	61	61
		crse	529	42,641	—	?	1626	462	462	178	178
tire2/13	6	ccse	625	38,014	—	?	?	?		?	
		cfst	577	32,177	—	?	?	?		?	
		crse	580	36,783	—	?	?	?		?	
		efst	467	17,385	—	192	1900	1902	1268	1968	1312

(b) Search times on the unsatisfiable hard planning formulas generated by Medic.

Table 7.4: Search times of various satisfiability testers on the hard planning formulas generated by Medic. Number of variables and literals are after simplification. Times are elapsed seconds; for `pmodoc`, both the measured elapsed seconds (first column) and the computed elapsed seconds per agent (second column) are shown. They were obtained on an SGI with four 150MHz R4400s. `walksat` times are average of 5 runs. '—' indicates that the run was not attempted; this is because `walksat` cannot confidently determine unsatisfiability. '?' indicates that the run was terminated after 1 hour; for `walksat`, it means that none of the 5 runs found a solution in 1 hour; for `pmodoc`, it means that none of the agents found a solution in 1 hour. (#TC = number of theorem clauses)

problem/ deadline	#TC	num of vars	num of literals	search time (seconds)						
				`walksat`	`2cl`	`modoc`	`pmodoc`			
							no comm		comm	
logistics.a/11	8	638	13,089	1	1	1	9	4	3	2
logistics.c/13	7	897	21,412	2	90	3	39	22	5	3
bw_large.c/14	15	2,222	78,146	146	12855	15439	25	7	17	5
bw_large.d/18	19	4,714	205,559	1494	??	58	239	50	101	21

(a) Search times on the satisfiable Satplan formulas

problem/ deadline	#TC	num of vars	num of literals	search time (seconds)						
				`walksat`	`2cl`	`modoc`	`pmodoc`			
							no comm		comm	
logistics.a/10	8	541	10,598	—	2	3	10	5	5	2
logistics.c/12	7	787	18,244	—	11154	1132	2401	1372	1195	685
bw_large.c/13	15	1,935	66,547	—	2769	5389	2695	719	2010	536
bw_large.d/17	19	4,275	184,180	—	??	??	??		??	

(b) Search times on the unsatisfiable Satplan formulas.

Table 7.5: Search times of various satisfiability testers on the hard planning formulas generated by Satplan. Number of variables and literals are after simplification. Times are elapsed seconds; for `pmodoc`, both the measured elapsed seconds (first column) and the computed elapsed seconds per agent (second column) are shown. They were obtained on an SGI with four 150MHz R4400s. `walksat` times are average of 5 runs. '—' indicates that the run was not attempted; this is because `walksat` cannot confidently determine unsatisfiability. '??' indicates that the run was terminated after 5 hours; for `pmodoc`, it means that none of the agents found a solution in 5 hours. (#TC = number of theorem clauses)

formula.

On four satisfiable formulas (fridge2 in cbse, ccse, and crse encodings, and tire2 in efst encoding), the two Modoc derivations that led to determining that the formula was satisfiable, one communicating autarkies and lemmas and another not communicating at all, were the same. This means that the lemmas that were communicated did not help at all to shorten the search for these formulas.

Table 7.5 shows the search times on hard planning formulas generated by Satplan. The cause of exceptional improvement by `pmodoc` over `modoc` on the satisfiable bw_large.c formula was that `modoc` started with a "bad" top clause. A separate study (Figure 7.1) shows that had `modoc` started with the seventh goal clause, it could have solved the formula in 4 seconds. Although this

num of checkers	dead-line	num of theorem clauses	num of vars	num of literals	search time (seconds)						
					`walksat`	`2cl`	`modoc`	`pmodoc`			
								no comm		comm	
2	8	4	105	3,900	1.08	1.90	0.02	0.08	0.08	0.13	0.13
3	15	6	282	18,294	12564	??	39	2	2	3	2
4	24	8	597	55,934	??	—	12883	1626	813	1606	803

(a) Search times on the satisfiable checker-interchange formulas.

num of checkers	dead-line	num of theorem clauses	num of vars	num of literals	search time (seconds)						
					`walksat`	`2cl`	`modoc`	`pmodoc`			
								no comm		comm	
2	7	4	90	3,238	—	2.00	0.12	0.30	0.30	0.48	0.48
3	14	6	261	16,794	—	??	121	199	132	186	124
4	23	8	570	53,252	—	—	??	??		??	

(b) Search times on the unsatisfiable checker-interchange formulas.

Table 7.6: Search times of various satisfiability testers on the checker-interchange formulas. Number of variables and literals are after simplification. Times are elapsed seconds; for `pmodoc`, both the measured elapsed seconds (first column) and the computed elapsed seconds per agent (second column) are shown. They were obtained on an SGI with four 150MHz R4400s. `walksat` times are average of 5 runs. '—' indicates that the run was not attempted; for `walksat`, this is because it cannot confidently determine unsatisfiability. '??' indicates that the run was terminated after 5 hours; for `walksat`, it means that none of the 5 runs found a solution in 5 hours; for `pmodoc`, it means that none of the agents found a solution in 5 hours.

formula may be an exceptional case, this is exactly the kind of situation Parallel Modoc attempts to "rescue" by means of parallel searches. For a formula like this, even running `pmodoc` on a single-processor system can easily outperform `modoc`.

Table 7.6 shows the search times on the checker-interchange formulas (see Section 6.10.1). Improvements by `pmodoc` were observed on the satisfiable formulas. On the unsatisfiable checker-interchange formula for 3 checkers, the search time of `pmodoc` was more than the search time of `modoc`. This was reflected in the number of PDT extension operations performed by the Modoc agent that proved that the formula was unsatisfiable; while `modoc` found a refutation in 698,469 PDT extensions, the Modoc agent that found a refutation in `pmodoc` took 720,413 PDT extensions without communication and 722,043 PDT extensions with communication.

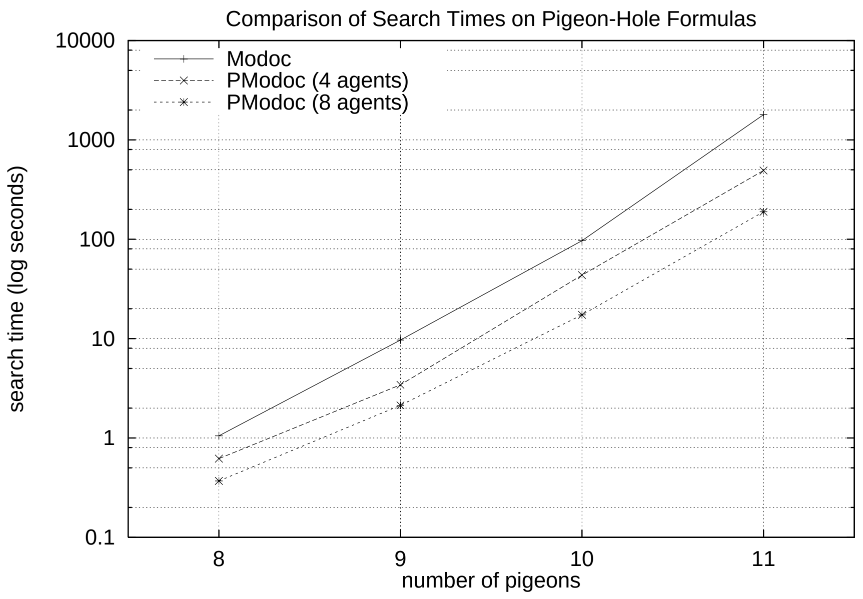


Figure 7.4: Search times of `modoc` and `pmodoc` on the pigeon-hole formulas. Times are elapsed seconds per agent and were obtained on an SGI with four 150MHz R4400s. All formulas are unsatisfiable by construction of the problem.

Pigeon-Hole Formulas

The pigeon-hole problem for *n* pigeons asks the question of whether or not it is possible to place *n* pigeons into $n-1$ boxes that contain only a single pigeon in each one of them. The problem is obviously impossible and hence the corresponding formula is unsatisfiable. The pigeon-hole formulas were used to show that certain class of formulas cannot have refutation proofs that are polynomially bound [20].

Unlike planning problems, pigeon-hole problems have no clear partition of the formula into axioms and the theorem. Rather, they are a collection of constraints that need to be satisfied. We call such problems *global optimization problems*. Because of this, it is not suited for goal-sensitive search.

Figure 7.4 shows the search times of `modoc` and `pmodoc` on the pigeon-hole formulas. It shows that `pmodoc` on an *n* processor computer would be able to find a solution faster than `modoc`, and that the growth rate of `pmodoc` is slower with 8 agents than with 4 agents. Figure 7.5 shows the efficiency of search by `pmodoc` as the ratio of total time consumed by `pmodoc` over the

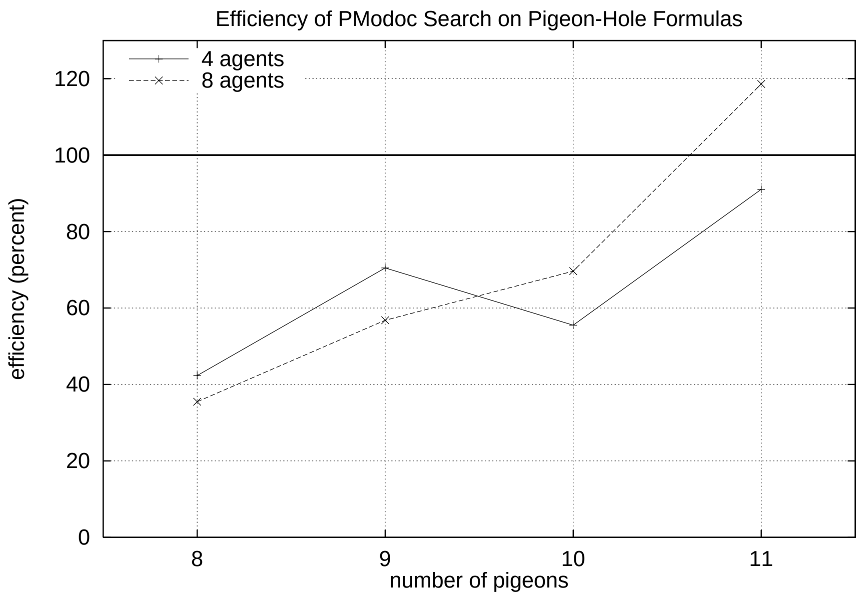


Figure 7.5: Efficiency of `pmodoc` search on the pigeon-hole formulas, as measured by $\frac{\text{total } \texttt{pmodoc} \text{ time}}{\texttt{modoc} \text{ time}}$. Times were obtained on an SGI with four 150MHz R4400s.

time consumed by `modoc` on the same formula. While the search efficiency of `pmodoc` was low (meaning that it consumed more time than `modoc`) on smaller formulas, it improved with the size of the formula. For 11 pigeons, the search efficiency of `pmodoc` with 8 agents was above 100%, meaning that it consumed less time than `modoc` to find a solution.

Random Formulas

Figure 7.6 shows the average search times of `modoc` and `pmodoc` on rand141, a collection of 200 3-CNF random formulas with 141 variables and 602 clauses. Figure 7.7 shows the speedup on the same collection of random formulas. A greater speedup is observed for the satisfiable formulas.

7.5.2 Improvements by Increasing the Number of Goal Clauses

The effect of a larger number of goal clauses than the number of theorem clauses constructed using the two methods described in Section 7.4 was examined on two classes of formulas—the circuit-diagnosis formulas and the planning formulas. In summary, the methods allowed the formulas to be solved in a shorter time in many cases, but there were a few cases where the formula became too

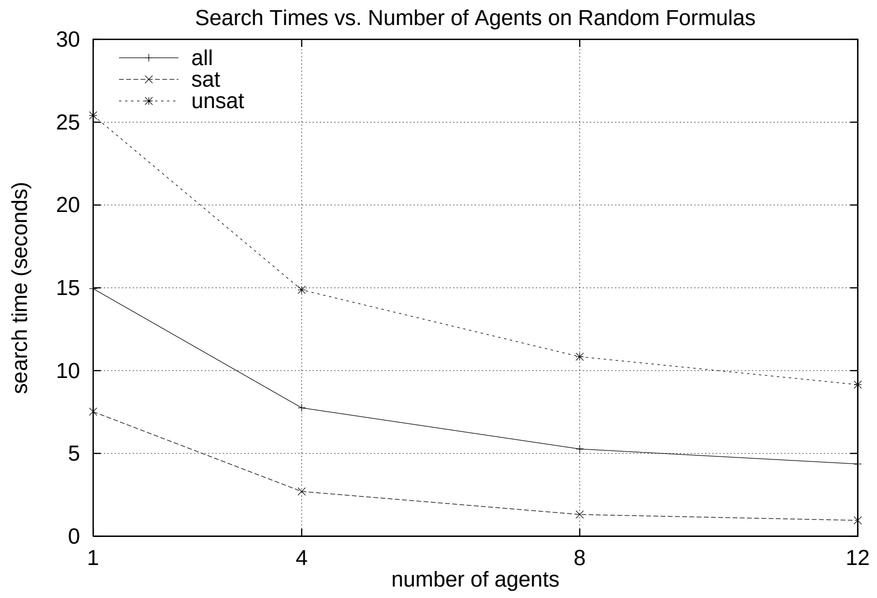


Figure 7.6: Average search times of `modoc` (1 agent) and `pmodoc` (4, 8, and 12 agents) on rand141 (200 3-CNF random formulas with 141 variables and 602 clauses). Times are elapsed seconds per agent and were obtained on an SGI with four 150MHz R4400s.

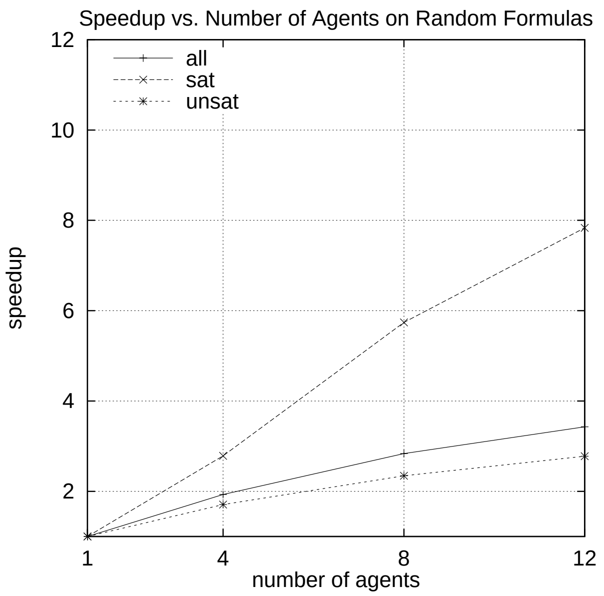


Figure 7.7: Speedup by `pmodoc` over `modoc` on rand141 (200 3-CNF random formulas with 141 variables and 602 clauses). Times are elapsed seconds per agent and were obtained on an SGI with four 150MHz R4400s.

difficult to be solved within the alloted time.

Circuit-Diagnosis Formulas

Circuit-diagnosis formulas are formulas generated by a *diagnosis test pattern generation* (DTPG) program [7, 8]. Such a formula has the property that it is satisfiable if and only if the outputs of two differently faulty circuits differ for some common input. For our tests, we have used formulas generated from the C6288 circuit in the ISCAS85 benchmark. C6288 is a 16-bit multiplier with 2406 gates. Faults simulated are "single stuck-at" faults.

Unfortunately, we were not given sufficient information to determine the theorem clauses. Therefore, an educated guess was made to pick a single clause as the theorem clause for each formula. While this was sufficient for Modoc, for Parallel Modoc, we wish to have more than one theorem clause so that multiple search attempts could be made at the same time.

Table 7.7 shows the search times of `walksat`, `2cl`, `modoc`, and `pmodoc` on the circuit-diagnosis formulas. Results were mixed. While there were formulas for which `pmodoc` did extremely well (e.g., formulas 1, 12, and 14), there were formulas for which `pmodoc` did much worse (e.g., formula 3).

Planning Formulas

Table 7.8 compares search times of `pmodoc` on some of the formulas listed in Table 7.4. Improvements of varying degree were observed on most of the formulas.

Degradation of performance was observed on fridge2 with cbse encoding when using clause exhaustion for both the satisfiable and the unsatisfiable formulas. (This was also observed on a few formulas using variable exhaustion in Table 7.7.) What was solvable in a few minutes now could not be solved in one hour.

It is worth mentioning that there is no guarantee that the searches performed using the original formula will be among the searches performed using the formula with a larger number of goal clauses. This is despite that running `pmodoc` on the formula with a larger number of goal clauses created by either clause exhaustion or variable exhaustion is essentially running Modoc

fmla	num	num	#GC		search time (seconds)						
num	of	of			`walksat`	`2cl`	`modoc`	`pmodoc`			
	vars	literals	CE	VE				CE		VE	
1	3,230	24,151	6	8	?	?	?	6	4	7	4
2	3,238	24,677	6	8	?	?	39	8	6	99	50
3	3,432	26,063	6	8	?	?	953	1356	904	?	
4	3,119	23,239	6	8	?	408	98	31	21	62	31
5	4,357	34,639	6	9	?	29	?	?		?	
6	4,355	34,616	6	9	?	551	?	?		?	
7	3,011	22,178	6	9	?	?	183	52	35	2623	1166
8	3,011	22,180	6	9	?	?	146	17	11	30	13
9	3,011	22,180	6	9	?	?	76	8	5	10	4
10	3,003	22,103	9	15	?	?	268	38	17	53	14
11	3,433	26,035	6	8	?	?	29	53	35	?	
12	2,897	21,110	8	14	?	?	2348	9	5	19	5
13	2,897	21,110	8	14	?	1256	131	12	6	354	101
14	2,897	21,110	8	14	?	1269	2425	7	4	14	4
15	3,971	31,117	6	8	?	62	?	?		?	
16	3,058	22,640	6	8	?	?	?	?		?	
17	3,008	22,228	6	8	?	97	36	58	39	76	38
18	2,779	22,180	8	14	?	76	92	16	8	908	259

Table 7.7: Search times of various satisfiability testers on the circuit-diagnosis formula set. All formulas are known to be satisfiable. For `pmodoc`, multiple goal clauses were created using clause exhaustion (CE) and variable exhaustion (VE), described in Section 7.4. Number of variables and literals are after simplification. Times are elapsed seconds; for `pmodoc`, both the measured elapsed seconds (first column) and the computed elapsed seconds per agent (second column) are shown. They were obtained on an SGI with four 150MHz R4400s. '?' indicates that the run was terminated after 1 hour; for `pmodoc`, it means that none of the agents found a solution in 1 hour. (#GC = number of goal clauses)

problem/ deadline	en- cod- ing	orig			CE			VE		
		#GC	search seconds		#GC	search seconds		#GC	search seconds	
big-bw1/11	ecse	6	16	10	18	36	8	18	32	7
fridge2/13	cbse	2	24	24	18	?		18	30	7
	ccse	2	78	78	24	259	43	24	267	45
	crse	2	581	581	24	1356	226	24	1334	222
hanoi3/7	cbse	3	70	70	45	582	52	45	445	40
monkey2/9	cbse	2	?		15	?		15	?	
	cfst	2	387	387	14	1057	302	14	1065	304
tire2/14	efst	6	3	2	23	4	1	23	4	1

(a) Search times on the satisfiable formulas.

problem/ deadline	en- cod- ing	orig			CE			VE		
		#GC	search seconds		#GC	search seconds		#GC	search seconds	
big-bw1/10	ecse	6	115	76	18	215	48	18	222	49
fridge2/12	cbse	2	108	108	18	?		18	434	96
	ccse	2	1869	1869	24	10603	1767	24	10588	1765
	crse	2	444	444	24	1633	272	24	4190	698
hanoi3/6	cbse	3	26	26	45	221	20	45	192	17
monkey2/8	cbse	2	1701	1701	15	5942	1854	15	5928	1581
	cfst	2	61	61	14	156	45	14	135	39
tire2/13	efst	6	1968	1312	23	921	160	23	881	153

(b) Search times on the unsatisfiable formulas.

Table 7.8: Effect of clause exhaustion (CE) and variable exhaustion (VE), described in Section 7.4, on `pmodoc` search times. The formulas were derived from the formulas reported in Table 7.4. Times are measured elapsed seconds (first column) and computed elapsed seconds per agent (second column); they were obtained on an SGI with four 150MHz R4400s. (#GC = number of goal clauses)

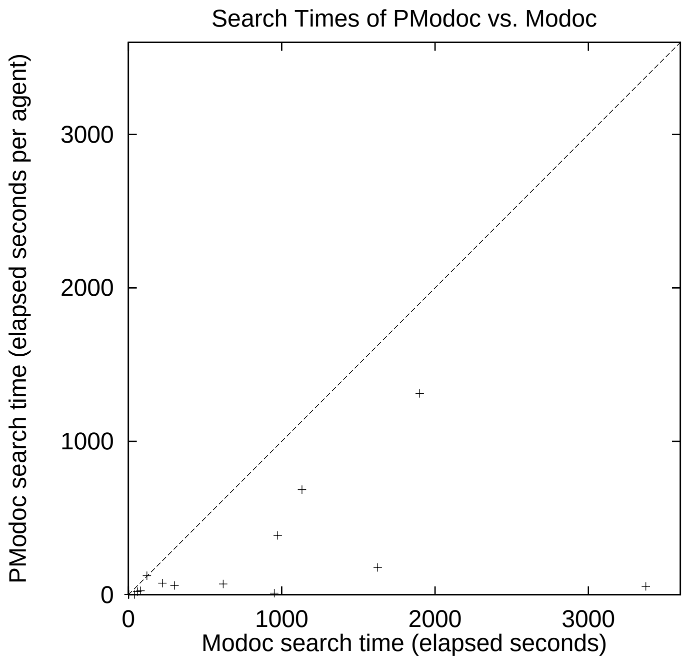


Figure 7.8: Comparison of search times between `pmodoc` and `modoc` on planning formulas. Times are elapsed seconds per agent and were obtained on an SGI with four 150MHz R4400s.

for each clause node that is one level-down from the top clauses with the original formula. This is because of the following reason. In deciding the order in which to extend the goal nodes, the Modoc search procedure in `pmodoc` tests whether they are lemmas or not; that is, goal nodes that are lemmas are deferred in favor of goal nodes that are not. Thus, with a different set of lemmas, it is possible for a search to be steered away from the search in the previous run, even if the search started from the same top clause.

7.6 Summary

Figure 7.8 compares the search times of `pmodoc` against `modoc` on the planning formulas reported in Section 7.5. (Only formulas for which both `modoc` and `pmodoc` found a solution in 1 hour are shown.) Plots in the lower-right triangle represent formulas for which `pmodoc` was able to find a solution faster than `modoc`. In almost all cases, `pmodoc` was faster than `modoc`. Further, there were many cases (not shown in Figure 7.8; see Table 7.4) where `modoc` was not able to find a solution in 1 hour yet `pmodoc` was able to find a solution in the order of minutes, and in some cases, seconds.

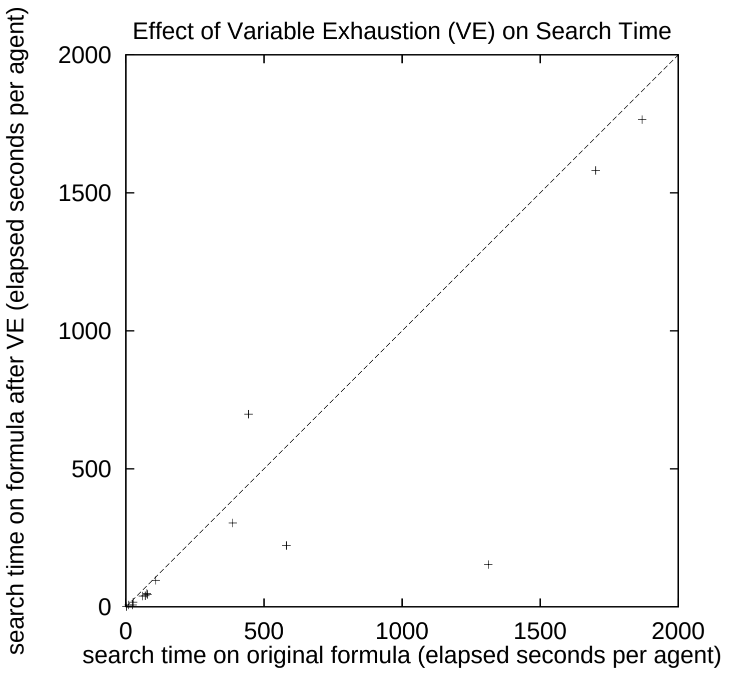


Figure 7.9: Effect of variable exhaustion on `pmodoc` search time. Times are elapsed seconds per agent and were obtained on an SGI with four 150MHz R4400s.

As designed, Parallel Modoc cannot utilize any more agents than the number of theorem clauses. Variable exhaustion (VE) is one of the two methods that was explored to break this constraint. Figure 7.9 shows the effect of the variable-exhaustion method on the search times of `pmodoc`. Plots in the lower-right triangle represent formulas for which the method allowed `pmodoc` to find a solution faster. While the method was able to increase the degree of parallel search and allowed `pmodoc` to find a solution in less time in most cases, there were few cases where it causes `pmodoc` to take more time.

Chapter 8

Conclusions

This dissertation examined the problem of determining satisfiability in a parallel cooperative computing environment. Parallel Modoc (Chapter 7) executes multiple Modoc agents and provides a mechanism to allow the agents to communicate information discovered about the formula. The Modoc algorithm (Section 2.3) is a refutation-search procedure based on propositional Model Elimination, extended to prune away certain branches that will not lead to a successful subrefutation. Modoc allows goal-sensitive search, an ability to focus its search on clauses that would determine the satisfiability of the formula. Experience shows that the search time varies widely depending on the clause chosen to focus the search on. Modoc, as a sequential search procedure, must make a choice among the several possible choices. At this time, it is possible for Modoc to choose the "wrong" clause, causing it to not being able to find a solution within a reasonable amount of time. However, as a multi-search procedure, Parallel Modoc is able to exploit this variation in search time, and because of this, it is often able to find a solution much faster than Modoc. It is not uncommon for Parallel Modoc to achieve speedup greater than the number of agents. Parallel Modoc has allowed many formulas that were too hard to be solved in practice to be solved within a reasonable amount of time.

Several future research directions are possible for both Modoc and Parallel Modoc. Little work has been done to study the effect of various ordering heuristics in the current implementation of Modoc. We have just started experimenting with scoring techniques to order goal nodes, similar to the techniques used in some DPLL-type algorithms to order splitting variables. The study should definitely be expanded to order eligible extension clauses, where there are relatively more choices.

For Parallel Modoc, the degree of parallel search and the amount of cooperation should be increased. With regards to increasing the degree of parallel search, some preliminary results have been obtained from methods based on resolution to construct (hopefully) a lager number of goal clauses than the number of theorem clauses. Certainly, this direction is promising, but at the same time, other methods to introduce parallelization at various places should be examined. With regards to increasing the amount of cooperation, expanding the types of information that is communicated among the agents could be considered. As an example, communicating lemmas that are not top-level lemmas but only have few dependencies might be a good candidate.

Another direction would be to extend Parallel Modoc to a non-homogeneous multi-agent search procedure. While Modoc agents (or some satisfiability testing agent) may have to do most of the work, other types of agents may be able to provide vital information using different algorithms. An example would be to have an agent that reasons exclusively on binary clauses and communicates any new finding to the Modoc agents.

Today, with the proliferation of personal computers, many of them connected to computer networks, it is common to have several fast computers with idle cycles available at a single site. As the need for faster satisfiability testers grows, with the abundance of computing resource, a satisfiability tester in the form of a parallel cooperative search may allow a breakthrough in the size of formulas that may be solved in practice.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

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

[4] S. Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, second edition, 1988.

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

[6] M. Böhm and E. Speckenmeyer. A fast parallel SAT-solver—Efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17(3-4):381–400, 1996.

[7] B. Chess. Diagnostic test pattern generation and the creation of small fault dictionaries. Master's thesis, University of California, Santa Cruz, June 1995.

[8] B. Chess, D. B. Lavo, F. J. Ferguson, and T. Larrabee. Diagnosis of realistic bridging faults with single stuck-at information. In *1995 IEEE/ACM International Conference on Computer-Aided Design*, pages 185–192, 1995.

[9] S. H. Clearwater, T. Hogg, and B. A. Huberman. Cooperative problem solving. In B. A. Huberman, editor, *Computation: The Micro and the Macro View*, pages 33–70. World Scientific, 1992.

[10] S. H. Clearwater, B. A. Huberman, and T. Hogg. Cooperative solution of constraint satisfaction problems. *Science*, 254(5035):1181–1183, 1991.

[11] S. Cook. The complexity of theorem-proving procedures. In *Proceedings 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[12] 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*, pages 21–27. AAAI Press, 1993.

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

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

[15] M. D. Ernst, T. D. Millstein, and D. S. Weld. Automatic SAT-compilation of planning problems. In *15th International Joint Conference on Artificial Intelligence*, pages 1169–1176, 1997.

[16] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.

[17] S. Fleisig, D. W. Loveland, A. K. Smiley, and D. L. Yarmush. An implementation of the model elimination proof procedure. *Journal of the Association for Computing Machinery*, 21(1):124–39, January 1974.

[18] M. Fujita, J. Slaney, and F. Bennett. Automatic generation of some results in finite algebra. In *IJCAI-93. Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 52–57. Morgan Kaufmann Publishers, 1993.

[19] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, 1979.

[20] A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[21] 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.

[22] T. Hogg and C. P. Williams. Solving the really hard problems with cooperative search. In *Proceedings Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 231–236, 1993.

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

[24] K. Iwama. Complementary approaches to CNF Boolean equations. In *Discrete Algorithms and Complexity*, pages 223–236. Academic Press, NY, 1987.

[25] K. Iwama. CNF satisfiability test by counting and polynomial average time. *SIAM Journal on Computing*, 18(2):385–391, April 1989.

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

[27] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings Tenth European Conference on Artificial Intelligence*, pages 359–363. Wiley, 1992.

[28] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *13th National Conference on Artificial Intelligence*, pages 1194–1201, 1996.

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

[30] T. Larrabee and Y. Tsuji. Evidence for a satisfiability threshold for random 3CNF formulas. Technical Report UCSC-CRL-92-42, Baskin Center for Computer Engineering and Information Sciences, University of California, Santa Cruz, CA, USA, October 1992.

[31] 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.

[32] 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.

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

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

[35] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *AAAI-92. Proceedings Tenth National Conference on Artificial Intelligence*, pages 459–465, 1992.

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

[37] 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.

[38] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, January 1965.

[39] 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*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 521–531. American Mathematical Society, 1996.

[40] R. E. Shostak. Refutation graphs. *Artificial Intelligence*, 7(1):51–64, 1976.

[41] E. Speckenmeyer, B. Monien, and O. Vornberger. Superlinear speedup for parallel backtracking. In E.N. Houstis, T.S. Papatheodorou, and C.D. Polychronopoulos, editors, *Supercomputing. 1st International Conference Proceedings*, pages 985–993. Springer-Verlag, 1988.

[42] Y. Tanaka. A dual algorithm for the satisfiability problem. *Information Processing Letters*, 37:85–89, 1991.

[43] A. Van Gelder. Autarky pruning in propositional model elimination reduces failure redundancy. *Journal of Automated Reasoning*. (to appear).

[44] A. Van Gelder. Simultaneous construction of refutations and models for propositional formulas. Technical Report UCSC-CRL-95-61, Baskin Center for Computer Engineering and Information Sciences, University of California, Santa Cruz, CA, USA, 1995.

[45] A. Van Gelder and F. Kamiya. Model-based pruning for propositional model elimination. In U. Furbach, editor, *Model-Based Automated Reasoning*, August 1997. (Workshop held in connection with IJCAI-97.).

[46] 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.

[47] 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.

[48] 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*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 559–586. American Mathematical Society, 1996.

[49] H. Zhang and M. Bonacina. Cumulating search in a distributed computing environment: A case study in parallel satisfiability. In H. Hong, editor, *First International Symposium on Parallel Symbolic Computation PASCO '94*, pages 422–431. World Scientific, 1994.

[50] H. Zhang, M. P. Bonacina, and J. Hsiang. PSATO: A distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4–6):543–560, 1996.

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