Annals of Mathematics and Artifical Inteligence 0 (1998) ?–?

A Propositional Theorem Prover to Solve Planning and Other Problems

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

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

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

Classical STRIPS-style planning problems are formulated as theorems to be proven from a new point of view: that the problem is *not* solvable. The result for a refutation-based theorem prover may be a propositional formula that is to be proven unsatisfiable. This formula is identical to the formula that may be derived directly by various "SAT compilers", but the theorem-proving view provides valuable additional information not in the formula, namely, the theorem to be proven. Traditional satisfiability methods, most of which are based on model search, are unable to exploit this additional information. However, a new algorithm called "Modoc" is able to exploit this information and has achieved performance comparable to the fastest known satisfiability methods, including stochastic search methods, on planning problems that have been reported by other researchers, as well as formulas derived from other applications. Unlike most theorem provers, Modoc performs well on both satisfiable and unsatisfiable formulas.

Modoc works by a combination of back-chaining from the theorem clauses and forward-chaining on tractable subformulas. In some cases, Modoc is able to solve a planning problem without finding a complete assignment because the back-chaining methodology is able to ignore irrelevant clauses. Although back-chaining is well known in the literature, a high level of search redundancy existed in previous methods; Modoc incorporates a new technique called "autarky pruning", which reduces search redundancy to manageable levels, permitting the benefits of back-chaining to emerge, for certain problem classes.

Experimental results are presented for planning problems and formulas derived from other applications.

Keywords: Planning, Modoc, autarky, Model Elimination, satisfiability, resolution, refutation.

* All correspondence should be addressed to this author.

1. Introduction

One of the classical techniques for problem solving in artificial intelligence is to construct a logical formulation of the problem, formulate a conjecture that a solution exists, then try to prove the conjecture in first-order logic [FN71, Nil80]. If the attempt is successful, the solution can be read off from the variable substitutions that are present in the proof. This is the motivating concept behind the programming language Prolog [VEK76,Kow79].

Recently, Kautz and Selman proposed a different technique, which is often called *SAT compilation*, in which the problem is formulated as a propositional formula that is satisfiable if and only if the problem is solvable [KS96]. The idea of SAT compilation has been further developed by Ernst *et al.* [EMW97]

This paper shows that the same formula can be arrived at by formulating a conjecture different from the classical method. The advantage of this alternative point of view is that we now have a conjecture whose truth or falsity provides a focal point for the ensuing search. This permits *goal-sensitive* techniques (usually back-chaining) to be brought to bear.

A recently developed algorithm called *Modoc* is applied to the solution of planning problems and related problems from other domains. Modoc has its roots in the Model Elimination (M.E.) method (see Section 2.1). Unfortunately, M.E. has major performance problems in the propositional domain due to search redundancy [Pla94]. Similar inefficiencies have been observed in the related tableau procedure for modal logic [GS96]. A new pruning technique called *autarky pruning* is introduced in Modoc to overcome this search redundancy [VG97]. See also Section 5.2.

Modoc also takes advantage of the propositional domain to gain certain efficiencies not available in the first-order domain. It incorporates some forms of forward chaining in the form of *eager lemmas*, which are derived very efficiently along the lines of the linear-time technique for unit-clause propagation [DE92]. This technique is extended in Modoc to maintain the additional information needed in the back-chaining search. It also maintains *C-literals* [Sho76,LMG94] to record successful sub-derivations, so they will not have to be rediscovered. Several extensions of the C-literal technique are introduced to produce further efficiencies, but these are described elsewhere [VGO97].

1.1. Goal-Sensitive Back-Chaining

Experiments on formulas derived from applications have indicated that Modoc's back-chaining ability is a major advantage, compared to most other known complete satisfiability methods. Moreover, it is competitive with the best known incomplete methods on such formulas when they are satisfiable. (Of course, incomplete methods do not work at all on unsatisfiable formulas.) Also, Modoc does not use any randomness, and has no parameters to tune.

Modoc is a back-chaining theorem prover that works from application-specified *top clauses*. It can be turned into a general-purpose satisfiability engine by providing all the clauses in the formula as potential top clauses, but this loss of focus normally entails a substantial degradation of performance. For example, on random formulas, Modoc underperforms the same algorithms that it outperforms on application-based formulas. On application-based formulas, without the focus of the appropriate top clauses, Modoc sometimes is able to perform well just by following the structure in the formula, but results are highly variable.

Therefore, it is important for SAT compilers and other prospective users of Modoc to retain the domain knowledge needed to identify the appropriate top clauses, and to avoid preprocessing steps that are inconsistent with their use. Previously reported SAT compilers have discarded this information, simply because it was of no use to earlier satisfiability testers, but they may be easily modified to take advantage of Modoc, as discussed in Section 4.

1.2. Outline

The paper is organized as follows. Satisfiability methods and other background are reviewed in Section 2. The main idea of formulating planning problems as propositional theorems is described in Section 3. Issues of goal-sensitive simplification are discussed in Section 4. Autarkies are described in Section 5. Section 5.1 explains how an autarky can furnish a problem solution without finding a complete satisfying assignment to the formula, while Section 5.2 illustrates how autarky pruning reduces the search redundancy that is inherent in Model Elimination. The experimental results are reported in Section 6. Finally, Section 7 draws conclusions and discusses future directions.

2. Review of Satisfiability Methods

The *satisfiability problem* has been the subject of continuing research, which has increased in intensity with the advent of high-speed microprocessors. This is the problem of deciding whether a propositional Boolean formula has a satisfying assignment. A closely related problem is to show that the formula is inconsistent, a common technique used in theorem proving. The theorem is proved true if and only if the formula is found *unsatisfiable*.

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

Three basic methods have been developed for satisfiability testing: refutation search, model search, and local search. (See [VG97] for additional bibliography.)

1. *Refutation search* seeks to discover a proof that a formula is unsatisfiable, usually employing resolution. Model Elimination and SL-Resolution typify these methods [Lov69,KK71,Lov72]. However, they generally cannot provide a model on satisfiable formulas. The technique of autarky pruning used by Modoc evolved out of investigation on how to extract a model from an unsuccessful attempt to construct a resolution refutation.

2. *Model search* seeks to discover a satisfying assignment, or a model, for the formula. The DPLL algorithm, due to Davis, Putnam, Loveland, and Logemann [DP60,DLL62], is the basis for many modern refinements. This basic algorithm consists of the *unit-clause* and *pure-literal* rules for simplification, and the *splitting* rule for searching. As published, the splitting rule specifies that the splitting variable be chosen from a shortest clause.

3. Several methods employ various *local search*, or *stochastic search*, heuristics to perform incomplete model searches [SLM92,GW93,SKC95]. They might also be characterized as *Max-SAT* methods. Unlike complete model-search methods, these methods cannot terminate naturally when they fail to discover a model; it is necessary to specify resource limits after which they will give up and report "don't know". However, they have succeeded in finding models on much larger random formulas than current complete methods can handle.

2.1. Overview of Model Elimination

Model Elimination (M.E.) is a back-chaining resolution system that, except for bookkeeping issues, is nearly equivalent to a form of linear resolution called *SL-Resolution* [Lov69,KK71,Lov72]. The theme of linear resolution systems is that one of the two operands for every resolution operation is the most recently derived clause (or a user-designated *top clause* to start the procedure). M.E. is a refutation-based system, in that it seeks to derive the empty clause.

In "simple" M.E., the second operand is restricted to be either a clause of the original formula (an *input* clause), or a certain type of previously derived clause called an *ancestor* clause. If the most recently derived clause is $[p, \alpha]$, then the ancestor clause must fit the pattern $[\neg p, \beta]$, where $\beta \subseteq \alpha$. Thus, the resolvent is simply $[\alpha]$. In this case, the operation is called variously *ancestor resolution*, *s-resolution*, *subsumption resolution*, and *reduction*. When an input clause is used, the operation is called *extension*.

M.E. can be extended with a *lemma* facility, which in effect is a bookkeeping system to record earlier successful sub-refutations, for later re-use. The earliest experiment with lemmas was unfavorable [FLSY74], but Shostak developed a more efficient system and coined the term *C-literal* to describe it [Sho76]. The C-literal idea has been used more successfully by several researchers [Sti94,LMG94, AL97].

Later research discovered that trees could be used as an effective and natural data structure for M.E. [MZ82], and this led to the recognition that M.E. can also be viewed as a special case of the *tableau* method [LMG94].

Example 1. Figure 1 illustrates propositional M.E. operations in the context of a tree data structure. The formula F consists of all 3-CNF clauses on variables a, b, and c, except for the all-positive clause, as shown at the top of the figure. The tree alternates by levels between goal nodes (circles) and clause nodes (rectangles). The root is a dummy goal called *verum*. The *top clause* for the attempted refutation is attached to the verum to get started, and it immediately generates three goal nodes corresponding to its literals.

Clause $[a, \neg b, \neg c]$ clashes with the goal node $\neg a$, so is eligible for an *extension* operation, which is carried out by attaching this clause as a child of the goal node. This clause generates new goal nodes for those of its literals that do not clash with the parent goal node, in this case, $\neg b$ and $\neg c$.

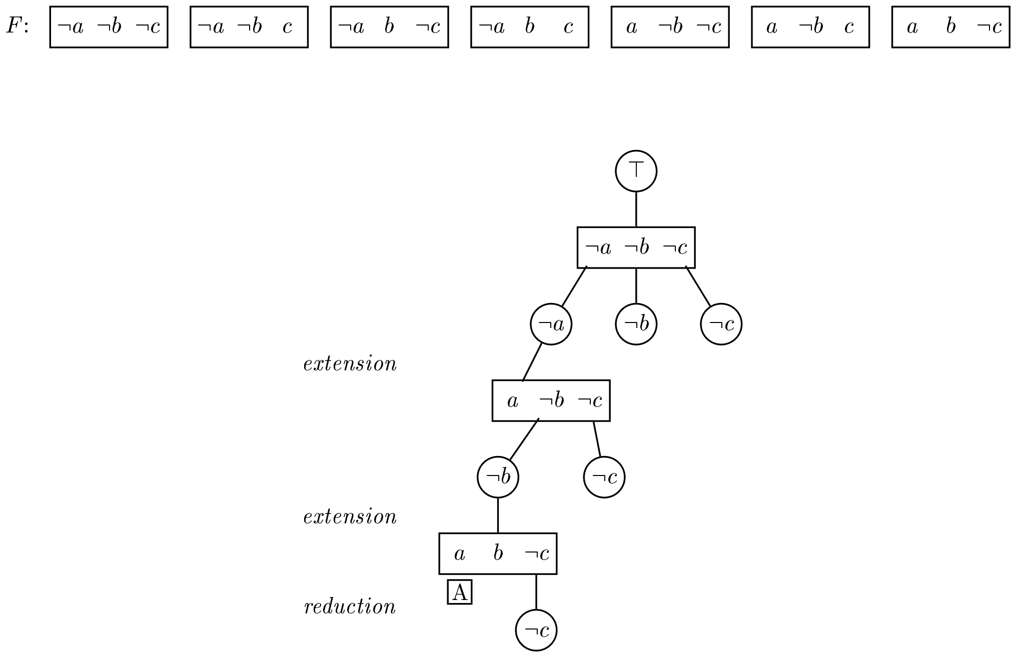

Figure 1. Model Elimination operations, as discussed in Example 1.

The attachment of $[a, b, \neg c]$ to the goal $\neg b$ is the second extension. This clause generates goals a and $\neg c$, because b is the clashing literal. But goal a clashes with a different ancestor, $\neg a$, so it may be removed through the *reduction* operation. The boxed "A" marks where this operation occurred.

The current refutation attempt now fails because every clause that clashes with $\neg c$ also contains one of the ancestors, $\neg a$ or $\neg b$; such clauses are said to violate the *regularity*, or *tightness*, condition of M.E. This example is continued in Example 7. □

A clause becomes a leaf in the tree when it has no subgoals, possibly after reductions have removed them. A tree becomes a refutation when all leaves are clause nodes. This variant of Model Elimination is called *weak* because identical goal nodes are not combined–the tree structure is maintained.

At a high level, Modoc further extends this methodology by adding a book-keeping system to record certain earlier *unsuccessful* sub-refutation attempts (see Section 5). This turns out to be crucial for obtaining satisfactory performance on propositional formulas, as discussed in Section 2.2.

2.2. Previous Goal-Sensitive Methods

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

Linear resolution methods can exploit this information by starting the refutation attempts at one or more of the key clauses. In this sense, they are *goal sensitive* and potentially focused. No straightforward method is known by which model-searching methods (DPLL, its variants, stochastic search methods, and other local search methods) can achieve a similar focus.

Despite this apparent advantage for back-chaining resolution, prior experience with propositional resolution has been negative (and consequently, largely unreported). The reason for prior poor performance of propositional resolution is discussed in detail elsewhere [VG97]. There, it is reported that Model Elimination (M.E.) experiences a redundancy factor of 1,000,000 on formulas as small as 20 variables and 90 clauses. That is, M.E. does 1,000,000 times as many operations as Modoc on average.

Related behavior, which can be described as *search redundancy*, has been observed elsewhere. Plaisted has shown that many goal-sensitive resolution procedures have exponential worst cases on Horn formulas [Pla94]. This indicates a highly redundant search because such formulas can be solved with a linear-time search. Giunchiglia and Sebastiani have observed very high search redundancy in a tableau procedure for propositional modal logic [GS96]. M.E. is an instance of a tableau method for classical logic.

Modoc addresses the search-redundancy problems just described by the use of a new technique called *autarky pruning*, which is described in Section 5. With this problem conquered, experimental results presented in Section 6 show that goal-sensitive back-chaining becomes a very powerful technique.

3. Problem Solving as Theorem Proving, Revisited

At a very high level, the classical approach to problem solving through first-order theorem proving proceeded as follows.

1. Formulate the problem as a set of axioms, $\texttt{axioms}(P)$, that depend on a plan P (also called a control strategy), a specification of the initial state, `init`, and the goal, `goal`.
2. Formulate the conjecture $\exists P(\texttt{axioms}(P) \wedge \texttt{init} \rightarrow \texttt{goal})$.
   In this high level description, other logical variables are suppressed, but these are normally universally quantified within the scope of the `axioms` formula.
3. Try to prove the conjecture, usually by *refuting* its negation, which is
   $$\forall P(\texttt{axioms}(P) \wedge \texttt{init} \wedge \neg\texttt{goal}).$$
4. For back-chaining resolution theorem provers, the formula is transformed into prenex conjunctive normal form, and the clauses that originated from $\neg$`goal` are used as top clauses in the refutation search.
5. If a refutation is found, the substitution for P constitutes a successful plan.

Now suppose we change our point of view slightly and decide to formulate the conjecture that the problem is *unsolvable*:

$$\forall P(\texttt{axioms}(P) \wedge \texttt{init} \rightarrow \neg\texttt{goal}),$$

This amounts to trying to *refute* the formula

$$\exists P(\texttt{axioms}(P) \wedge \texttt{init} \wedge \texttt{goal}).$$

But now, P is existentially quantified and must be skolemized if resolution methods are to be employed. With an appropriate representation for P, the resulting formula may be "grounded" when the state space is finite. The resulting formula is much the same as one that might be produced by a SAT compiler before post-processing simplifications. However, reported SAT compilers just view the process as constraint generation, not representing any particular theorem.

To recapitulate, we use a technique that could almost be called "double psychology". To solve the problem, we "pretend" we are trying to prove that it is *unsolvable*. If this conjecture is indeed false, Modoc outputs an "explanation" of why it "failed". The form of this explanation is a set of truth assignments that satisfies all clauses that are "relevant" (in a manner made precise later) to the specified top clauses, and *does not intersect* the remaining clauses. But this explanation is just the solution we were looking for to begin with.

Of course, the conjecture (of unsolvability) may be true, in which case Modoc (given enough time and space resources) verifies it, confirming that the problem is unsolvable. A mixture of solvable and unsolvable problems arises normally in optimization problems. For example, to verify that one has found a plan with a minimum number of steps, it is necessary to prove that the problem, constrained to have one fewer steps, is unsolvable [KS96,EMW97].

For some problems, the focus imparted by specific top clauses enables Modoc to solve the problem without constructing a complete satisfying assignment. That is, not only are some *variables* unassigned, but some *clauses* may contain only unassigned literals. This occurs when the clause is "irrelevant" to the goal. Being able to ignore the problem of satisfying such clauses is a significant gain for Modoc.

To make the notion of "irrelevant" more concrete, let us consider as an example the group of problems called "logistics" (see Section 6.2). Packages are moved around by trucks and airplanes. The goal is to get certain packages to certain locations by the *deadline*. Now suppose the solution found involves a truck delivering a package two steps before the deadline, and after that the truck has no further tasks to perform toward the overall goal. Then, Modoc will report a partial truth assignment that specifies the truck's movements through the time of delivery, but may well not specify the movements subsequently. Even if the earlier movements are "nondeterministic" in the sense that the truck could follow several routes and still deliver at the same time, Modoc will instantiate one of the routes in its solution.

The technique by which Modoc constructs partial assignments that ignore "irrelevant" clauses involves the concept of *autarky*, which is discussed in Section 5. As mentioned above, autarky construction is the method by which Modoc eliminates most of the redundancy inherent in Model Elimination.

4. Goal-Sensitive Simplification

Existing SAT compilers perform goal-*insensitive* simplifications, after deriving the basic propositional formula [KS96,EMW97]. Both papers reported that simplification was a major factor in performance. To work optimally in conjunction with Modoc, they need to perform slightly different goal-*sensitive* simplifications. This involves just a change in the post-processing of the compilation, and does not involve any change in the basic design of the compiler itself, other than to output what the goal clauses are. Because several *ad hoc* modifications of the simplification process are known to lose completeness for back-chaining, we wish to lay the foundation for a more formal approach.

We now outline how to conduct the simplification post-processing in a manner that preserves the ability to use back-chaining. In principle, a refutation theorem prover could either try to refute the goal clauses, or try to refute the initial-state clauses. Normally, there are fewer goal clauses, making them the better choice. Let us assume that we try to refute the goal clauses.

1. Construct the formula consisting of axioms and initial state, but without the goal clauses.
2. Apply any simplifications that produce a *logically equivalent* formula, in the sense that it has the identical set of models as the original.
3. Add the goal clauses back into the simplified formula.
4. Optionally, apply pure-literal elimination.

Several comments are in order about the simplification step 2.

1. Pure-literal elimination does *not* produce a logically equivalent formula; the set of models may be reduced. (All that is guaranteed is that the set of models is not changed from a non-empty set to an empty set.) When the goal clauses are not in the formula, some literals may appear to be pure, but in fact are not.

2. To maintain logical equivalence, variables that are eliminated from the main formula have to be kept around in a "side formula". Generally, existing simplifiers do not pass the "side formula" on to the solver, and many simplifiers do not maintain the "side formula" at all. Such simplifiers cannot be used safely with a goal-sensitive solver.
   Technically, to maintain logical equivalence, it is necessary to keep track of variables that disappeared from the formula without any constraint on their truth assignment. This could be done by means of trivial clauses of the form $[v, \neg v]$. However, this is not necessary for completeness. Remaining points below address cases when keeping track of variables *is* necessary.

3. Subsumption and resolution produce a logically equivalent formula.

4. The well-known unit-clause propagation procedure can be formulated as a combination of resolution and subsumption. The side formula consists of unit clauses whose variables no longer appear in the main formula.

5. Another efficient form of resolution is *2-closure*, the processing of binary and unit clauses [VGT96]. Although this is seen less often, it has been found to be a powerful adjunct to unit-clause propagation because of the prevalence of binary clauses in planning formulas [EMW97] and related applications. Larrabee exploited binary clauses for circuit test pattern generation [Lar92]. In the case of 2-closure, the side formula may contain subformulas of the form $v = q$ (or the equivalent pair of binary clauses: $[v, \neg q]$, $[q, \neg v]$). Here, v is a variable that no longer appears in the main formula, but q is a literal whose variable *may* still appear in the main formula. In any case, any constraints on q apply also to v.

6. Some applications benefit from the so-called *subsumption resolution* (Section 2.1). We have reduced application-generated formulas by factors of 3 to 10 in some cases, turning intractable problems into easy problems. Bit-encodings [EMW97] tend to be amenable to this simplification.

7. As an implementation device, it may be possible to "protect" the goal clauses from participating in any simplifications, by adding three new literals to each. This and other *ad hoc* tricks depend on knowing the internals of a particular simplifier.

The formula is now ready for Modoc, but if a goal-insensitive satisfiability tester will also be used, the final formula can be re-simplified to provide goal-insensitive input. This will have the same total effect as simplifying the entire formula, including goal clauses, to begin with.

5. Autarkies in Modoc

This section explains how Modoc, given a list of all relevant top clauses, is able to return an *autarky*, which satisfies only part of the formula, yet is a complete solution to the underlying problem. It also describes how Modoc uses autarkies internally to avoid fruitless searches. The operation of *autarky-based pruning* is the major contributer to Modoc's improved efficiency, relative to traditional Model Elimination [VG97].

The term *autarky* originates in Economics, where it means "self-sufficient country or region". It was introduced into computer science by Monien and Speckenmeyer [MS85] to designate a partial truth assignment that satisfies some clauses of a formula without restricting the possible models for the set of remaining clauses. They proposed a new model-searching algorithm based on checking for certain autarkies in the formula, which, if present, would permit the search to be reduced.

Definition 2. (autarky, *autsat*, *autrem*) Let F be a set of CNF clauses. A partial assignment A (normally represented as the set of literals assigned to "true"), possibly defined on some variables that do not occur in F, is called an *autarky* of F if A partitions F into two disjoint sets

$$F = autsat(F, A) + autrem(F, A)$$

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

More specifically, if an autarky makes p true, then any clause containing $\neg p$ must contain some *other* literal that has been assigned to true by this autarky; otherwise the possible models for that clause will have been reduced. Notice that, as limiting cases, the empty partial assignment and a completely satisfying partial assignment are autarkies.

Example 3. Let $F = \{[a, b], [\neg a, c], [b, d]\}$. Then $A_1 = \{a, c\}$ is an autarky of F, with

$$\begin{aligned} autsat(F, A_1) &= \{[a, b], [\neg a, c]\}, \\ autrem(F, A_1) &= \{[b, d]\}. \end{aligned}$$

However, $A_2 = \{a\}$ is not an autarky because of clause $[\neg a, c]$. Unit-clause simplification by A_2 shortens it without satisfying it. □

Definition 4. (strengthened formula, conditional autarky) Let S be a set of clauses and let A be a set of literals. Then $S|A$, read "S strengthened by A", consists of the formula that results by discarding any clause of S that contains a literal in A, and removing all occurrences of any literal whose complement is in A from remaining clauses. (This process is also called *unit simplification*.) A set of literals M is said to be a *conditional autarky* of S with respect to the set of literals A if M is an autarky of $S|A$. □

Suppose that $(M \cup A)$ is consistent. If $(M \cup A)$ is an autarky of S, then M is a conditional autarky of S with respect to A, but not necessarily *vice versa*. Conditional autarkies are developed by Modoc, as described in the next section, and are a major factor in its efficiency, as shown elsewhere [VG97].

5.1. Autarkies as Problem Solutions

When Modoc begins the attempt to refute a goal literal, say $\neg p$, it has a context of three sets of literals: the ancestor literals, unit lemmas (C-literals), and conditional autarky literals (i.e., a set of literals that comprise a conditional autarky with respect to the ancestor literals, see Definition 4). Ancestor literals and C-literals are standard elements of M.E. Conditional autarky literals are new to Modoc. Within the refutation search tree under the goal literal $\neg p$ any clause containing a literal that occurs in the current conditional autarky may be pruned from the search. The justification is that no M.E. sub-refutation can succeed using such a clause [VG97]. See also Section 5.2.

Example 5. Consider Figure 2, where it is assumed that the goal $\neg p$ has no autarky literals in its context. We assume $\neg p$ is resolvable with three clauses, as shown. If any of these three leads to a sub-refutation, the others are immaterial. Now suppose, as indicated in the figure, that clause 1 fails to yield a sub-refutation because subgoal $\neg q$ cannot be refuted. The sub-refutation attempt returns "Autarky 1", a set of conditional autarky literals including $\neg q$, and possibly others. In this example, we suppose that r is also in "Autarky 1". As stated in Definition 4, this set is called a conditional autarky because it behaves like an autarky only if the formula is strengthened with respect to the ancestor literals, $\neg q$ and above.

Now suppose clause 2 turns out to have literal r, which occurs in "Autarky 1". Then, this clause is pruned from the search immediately. We further suppose that clause 3 does *not* intersect with "Autarky 1". That is, clauses 1 and 2 are in "*autsat* of "Autarky 1", while clause 3 is in "*autrem*.

Next, we suppose that clause 3 also fails to produce a sub-refutation. Then, because it was not pruned by the current conditional autarky, it must return a

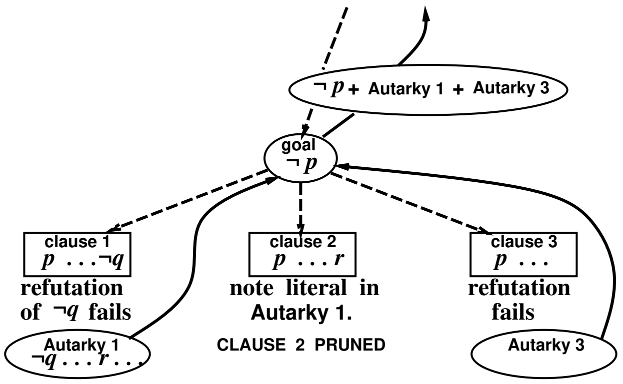

Figure 2. Modoc accumulates autarky literals returned by failing sub-refutations of clauses 1 and 3, as discussed in Example 5. If the current goal finally fails to be refuted, it is added to the accumulation of autarky literals, forming a conditional autarky to be returned from goal $\neg p$. If goal $\neg p$ *is* refuted, no autarky literals are returned.

nonempty set of literals that are not in the current conditional autarky, but may be added to it, producing an enlarged conditional autarky.

To conclude the example, there are no more clauses to try, so goal $\neg p$ fails to be refuted, and Modoc returns a conditional autarky consisting of the cumulative set of autarky literals produced during the search, plus $\neg p$. □

When Modoc is started from a designated top clause, say C_1, and it fails to find a refutation, it returns an autarky, say A_1, that satisfies a set of clauses, $F_{sat,1}$, and $C_1 \in F_{sat,1}$. The autarky A_1 partitions the original formula F into $F_{sat,1}$ and $F_{rem,1}$.

Now, if there were multiple top clauses, and any of them are in $F_{rem,1}$, Modoc begins a new refutation attempt with $F_{rem,1}$ as the formula and one of the originally designated top clauses, say C_2, as the top clause of *this* refutation attempt. Of course, a refutation of $F_{rem,1}$ is also a refutation of F. But if this attempt also fails, an autarky for $F_{rem,1}$ is returned, say A_2. But A_2 partitions $F_{rem,1}$ into $F_{sat,2}$ and $F_{rem,2}$, and $C_2 \in F_{sat,2}$.

Remembering that none of the variables appearing in A_1 also appear in $F_{rem,1}$, it is obvious that $(A_1 + A_2)$ constitutes a satisfying partial assignment for $(F_{sat,1} + F_{sat,2})$. Also, none of the variables appearing in A_2 also appear in $F_{rem,2}$. Therefore, $(A_1 + A_2)$ is also an autarky for the original F.

The process of "growing" the autarky and reducing the remaining set of clauses continues until all designated top clauses are satisfied by the autarky, or until a refutation is found. This is shown schematically in Figure 2, and was discussed in general in Example 5. However, now we are working through top clauses, rather than clauses that are resolvable with a particular goal literal ($\neg p$ in the figure). But the general procedure is the same.

Using the same approach that was used to show the ground completeness of many resolution strategies [AB70], it can be shown that if any designated top clause is part of a minimally unsatisfiable set of clauses in F, then (given enough time and space resources) Modoc finds a refutation [VG97].

Now we return to the issue of using an autarky as a solution to the underlying problem. As in Section 3, assume the formula is the conjunction

$$F = \mathtt{axioms} \wedge \mathtt{init} \wedge \mathtt{goal}$$

and the set of top clauses is `goal`. (Here `goal` refers to the overall goal of the problem, whereas in the previous example, "goal" referred locally to a literal to be refuted.) In practice, ($\mathtt{axioms} + \mathtt{init}$) is usually easily satisfiable before the goals are introduced. As mentioned earlier, Modoc returns a top-level autarky when it cannot find a refutation from any top clause.

Proposition 6. With F as above, if ($\mathtt{axioms} + \mathtt{init}$) is satisfiable and Modoc returns an autarky $(A_1 + \ldots + A_k)$, as described above, then F is satisfiable.

Proof. The final set of clauses $F_{rem,k} \subseteq (\mathtt{axioms} + \mathtt{init})$. Let Q be *any* model of $(\mathtt{axioms} + \mathtt{init})$; then Q is a model of $F_{rem,k}$. Now restrict Q to the variables still in $F_{rem,k}$, giving $Q_{rem,k}$. The disjoint union $(A_1 + \ldots + A_k + Q_{rem,k})$ is a model for F. □

In some cases Modoc is able to avoid a lot of work by ignoring $F_{rem,k}$. However, model-search methods must continue until even the irrelevant clauses have been satisfied, because they have no knowledge of what is relevant.

5.2. Reducing Search Redundancy

This section shows how autarky pruning in Modoc reduces search redundancy, compared to Model Elimination (M.E.). A particular run of Modoc or M.E. can be characterized by a tree called a *propositional derivation tree* (PDT). A PDT is a bipartite tree with *goal nodes* and *clause nodes* appearing at every other level, as we saw in Figure 1 and Example 1. M.E. tries to build a refuta-

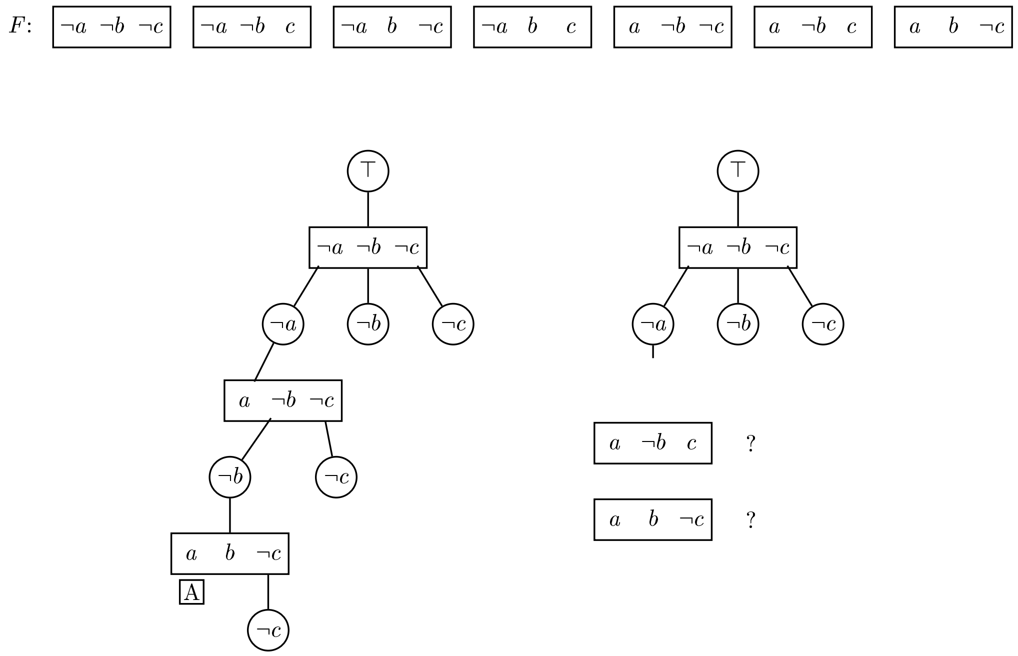

Figure 3. Model Elimination and search redundancy, as discussed in Section 5.2. Left: Search fails at lowest $\neg c$ goal. Right: After backtracking to alternative choices at $\neg a$ goal.

tion by constructing a *complete* PDT, in which all leaves are clause nodes. The searching aspect of M.E. comes into play when choosing a clashing clause for extension at a particular goal node. (In propositional M.E. reductions may be treated as mandatory, and have priority over extension.) A clause is said to be *eligible* for extension at a particular goal node p if one of its literals is $\neg p$ and none of the other literals agree with any ancestor goal node. The latter condition is called *regularity*, or *tightness*. Whenever multiple clauses are eligible in this sense for extension, it may be necessary to try all of them, through backtracking from those tries that do not produce a refutation.

Example 7. This example continues from Example 1 (see Figure 3). Recall that the formula F consists of all 3-CNF clauses on variables a, b, and c, except for the all-positive clause, and the top clause is all negative, $[\neg a, \neg b, \neg c]$. We pick up from Example 1 after two extensions and a reduction, giving the tree shown on the left of Figure 3. So far, Modoc and M.E. are proceeding in lock-step. But the search procedure now fails, because each clause containing literal c also contains an ancestor literal, either $\neg a$ or $\neg b$, and so is not *eligible* as defined above.

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

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

(In this case the partial assignment happens to be a total assignment, but this generally is not the case.) But obviously, every clause containing the literal $\neg c$ is also satisfied by M, so we conclude that every clause involving the *variable* c is satisfied by M.

Modoc (and M.E.) now backtrack from goal $\neg c$ and the clause that contained it. As described in Example 5, Modoc returns $\{\neg c\}$ up the search tree upon backtracking. Both procedures now look for another clause that is *eligible* at goal node $\neg b$, but there is none.

We can now extend the conclusion of the earlier paragraph to say that every clause containing either of the *variables* b or c, either positively or negatively, is satisfied by the partial assignment M.

Modoc (and M.E.) again backtrack. Modoc returns $\{\neg b, \neg c\}$ up the search tree. Both procedures now look for another clause that is *eligible* at goal node $\neg a$ (Figure 3, right). There are two such clauses, as indicated. The standard M.E. algorithm would continue trying to construct a refutation using one of these clauses, then the other. But notice that both of these clauses are satisfied by the partial assignment M that was implicitly constructed on the first branch.

After a few moments thought, we can predict that these refutation attempts must fail, without carrying out the search. Intuitively, the reason is that each eligible extension will produce *some* new goal node whose literal is in M. Thus *some* branch will not terminate with a clause as leaf. Eventually, some goal is generated that has no clauses eligible for extension.

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

Returning to the search procedures, this is where Modoc and Model Elimination part company. Model Elimination carries out the fruitless searches with the clauses $[a, \neg b, c]$ and $[a, b, \neg c]$. These attempts will re-explore clauses in differing permutations.

However, Modoc is now back at node $\neg a$ and has collected $\{\neg b, \neg c\}$ from failing to refute $\neg a$ after extending it with $[a, \neg b, \neg c]$. Modoc nows applies its stronger eligibility criterion that, besides regularity or tightness, includes the requirement that the candidate clause must not have a literal that occurs in $\{\neg b, \neg c\}$. This causes all the remaining clauses to be removed from the set of eligible clauses, and Modoc fails at goal node $\neg a$ without further searching. The set $\{\neg a, \neg b, \neg c\}$ is returned up the tree. □

The key point of the previous example is that the set of literals returned up the tree by Modoc after failing to complete a refutation is a conditional autarky at that goal node. Any M.E. refutation attempt using a clause that has a literal in this conditional autarky will have some branch that fails. These facts are proved elsewhere [VG97].

6. Experimental Results

To assess the performance of Modoc and compare it to other fast satisfiability testers, we implemented Modoc in C (we refer to it as `modoc`) and ran it against a number of fast satisfiability testers reported in the literature on a large collection of planning formulas. Detailed comparisons are presented for `walksat` [SKC95], `sato3` [Zha97], and `2cl` [VGT96]. A few other programs that we tried are mentioned briefly, but these programs were orders of magnitude slower than those just mentioned. All of these programs are coded in C. According to published reports, all have used efficient data structures and are highly optimized. After reviewing these programs in Section 6.1, results on various classes of formulas are presented in subsequent sections, and discussed in Section 6.7. All times are CPU times on an SGI Onyx with a 150-MHz R4400 CPU.

6.1. Other Model-Search Programs

The well-publicized `walksat` is an incomplete stochastic search program [SKC95]. It has enjoyed success on many large problems that could not be handled by complete procedures. Several earlier studies of planning via satisfiability used this program [KS96,EMW97]. Therefore, it is a natural candidate for comparison with `modoc`.

Among complete model-search programs, for the classes of formulas reported here, the theme that emerges is that the more reasoning a program does, the better it performs. For the programs we considered, "reasoning" consists of deriving new clauses by resolution. We did test several programs with a high emphasis on efficient search, but they performed very poorly on these problems, compared to programs with more of a reasoning component, and do not appear in the tables. Among such programs are `sato3-g0`, which is `sato3` with lemma derivation switched off, `tableau`, included in the Satplan distribution, but attributed to Crawford and Auton, and our own unadorned implementation of DPLL. After presenting the experimental results, we return to the topic of reasoning in Section 6.7.

Some stable implementations of complete model-search programs that have a reasoning component that goes beyond unit clause propagation are `2cl` [VGT96], which performs binary-clause resolutions, Grasp [SS96] and `sato3` [Zha97], which derive certain lemmas, up to a length set by the user, which is 20 by default for `sato3`. Sato3 is essentially `sato2` with a more sophisticated rule for choosing the splitting variable and its own implementation of Grasp lemmas added [Zha97]. The latter addition appears to be the most important (see Section 6.7). The value of 20 was determined empirically by optimizing for the set of 325 benchmark formulas reported in Section 6.5 [SS96]. Their lemma methodology is discussed further in Section 6.7. We found that Grasp is generally slower than `sato3`, which agrees with other reports [Zha97]. Therefore, we report detailed comparisons of `modoc` with `2cl` and `sato3`.

6.2. Planning Problems

Planning formulas were generated using Satplan [KS96] and Medic [EMW97]. The formulas were goal-*sensitively* simplified, as described in Section 4, for Modoc. The resulting formulas were further goal-*insensitively* simplified to provide the most favorable input format for `walksat`, `sato3`, and `2cl`.

Formulas are from various domains including Logistics, Blocks-World, Towers of Hanoi, Monkey and Banana, Flat Tire, and Fridge Fixing. The latter three are single problems, although variations may be created. Although Towers of Hanoi is a family, it gets hard too rapidly to produce many practical examples.

The Blocks-World family is classical. Beginning at an initial configuration, a mechanical arm moves single blocks, one by one, to arrive at a goal configuration within the deadline. The optimization problem is to find the minimum feasible deadline, but each propositional problem is based on a given deadline. This

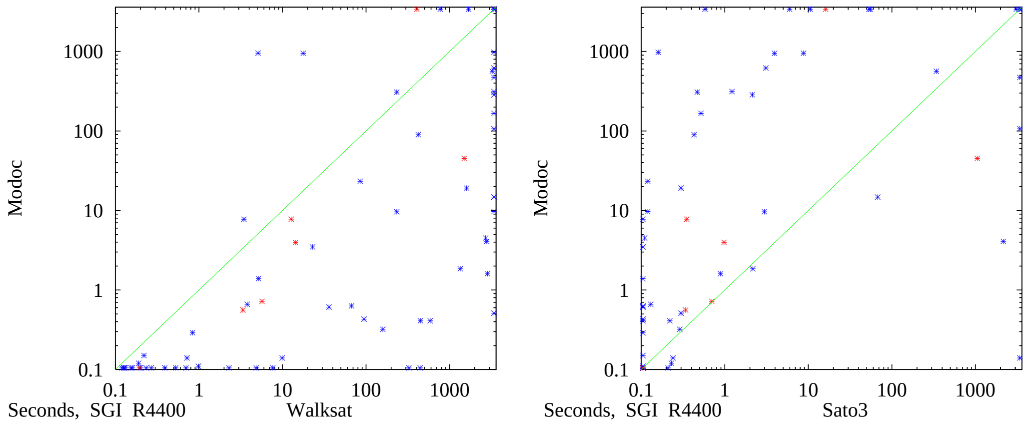

Figure 4. Search time comparison of `modoc` against `walksat` (average of 5 runs) and `sato3`. Times are in CPU seconds on SGI 150MHz R4400. Seventy-two formulas were generated using Satplan and Medic, and were all satisfiable.

family is easily parameterized by the number of blocks, and the distributions of initial and goal configurations, which may be randomly generated [KS96]. For example, bw_large.c has 15 blocks, and bw_large.d has 19 blocks.

The family of logistics problems requires a set of packages to be transferred from their initial locations to specified destinations within the deadline. A graph specifies which locations are adjacent; adjacency means the distance between the locations can be traversed in one time step by an appropriate vehicle. Packages may be transported between certain locations by truck, and between others by airplane, so the task becomes one of scheduling the trucks and airplanes. This family might be viewed as a more complex version of Blocks-World in that there are multiple agents with different capabilities. It is also easily parameterized [KS96]. For example, logistics.a has 8 packages, 6 locations, and logistics.c has 7 packages, 8 locations.

Figure 4 compares `modoc` search times against the average of five `walksat` search times and against `sato3` search times. All formulas were satisfiable, since Walksat is an incomplete procedure. Deadlines were set for optimal plan lengths. Formulas are from the domains mentioned above. Runs were timed out after 1 CPU hour, but included in the average, for this plot. Formulas solved in 0.1 seconds by all three programs are not shown.

For `walksat`, we primarily used the same parameters as used by Ernst *et al.* [EMW97]:

Table 1
Search time comparison of various SAT testers on hard planning formulas generated by Satplan. For each problem, the formula in the upper line is unsatisfiable and the one in the lower line is satisfiable. This establishes that the deadline on the lower line is optimal. `walksat` was not run on unsatisfiable formulas (—), and was averaged over 5 runs otherwise. Time-outs are denoted by "??" (5 hours).

problem	dead-line	number of vars	number of literals	CPU seconds (SGI R4400, 150MHz) `walksat`	`sato3`	`2cl`	`modoc`
logistics.a	10	541	10,598	—	1	2	3
	11	638	13,089	1	1	1	1
logistics.c	12	787	18,244	—	4	11154	1132
	13	897	21,412	2	1	90	3
bw_large.c	13	1935	66,547	—	5	2769	5389
	14	2222	78,146	146	4	12855	15439
bw_large.d	17	4275	184,180	—	264	??	??
	18	4714	205,559	1494	161	??	58

`walksat -tries 1000000 -noise 30 100 -cutoff` $2n^2$,

where n is the number of variables. We also tried a cutoff of $10n^2$ on problems that timed out, but results were substantially the same. However, some other parameter setting may work better. For `sato3` we used the default parameters. By default, derived clauses of up to 20 literals are retained in the clause set.

In Figure 4, dots in the lower-right triangle represent formulas for which `modoc` was faster than `walksat` on average, or faster than `sato3`. Dots in the upper-left are cases where `walksat` or `sato3` was faster. We see that `modoc` was usually faster than `walksat`, which is a significant accomplishment for a complete search procedure. However, `sato3` was usually faster than `modoc`.

Table 1 compares `modoc` times against search times of `walksat` (average of five runs), `sato3`, and `2cl`. The formulas are from those reported in [KS96]. We see that `sato3` does much better than the other programs in the unsatisfiable cases, and in most of the satisfiable cases, as well. Although `modoc` was somewhat faster on the satisfiable version of bw_large.d, it did very badly on the objectively easier formula, bw_large.c. The poor performance of `modoc` on bw_large.c is analyzed in Section 6.3. We believe that `modoc` is the first program that was able to show (directly) that logistics.c for deadline 12 is unsatisfiable. (Kautz and Selman determined this fact indirectly by constructing logistics.b,

Table 2
Search time of `modoc` on bw_large.c for deadline 14 for different top-clause orders. Times are in CPU seconds on SGI 150MHz R4400. The formula has 15 goal clauses, which were cyclically permuted to create 15 runs. Permutations not listed exceeded the one-hour time limit.

top-clause order	time
3,...,15,1,2	911
7,...,15,1,...,6	4
8,...,15,1,...,7	24
11,...,15,1,...,10	745

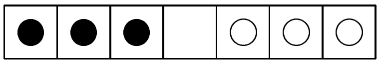

Figure 5. The goal of the checker-interchange problem is to interchange the positions of the white and black checkers through a series of moves and jumps, somewhat like Chinese checkers. White checkers may move left, or jump left over a black checker, into an empty space. Black checkers have the opposite capabilities.

a simplified version of logistics.c, and showing that logistics.b is unsatisfiable.) However, `sato3` now gets this result easily, and solves bw_large.d with deadline 17, which is previously unsolved, as far as we know.

6.3. Top Clause Order

Table 2 illustrates the extremely wide variation in search times, based on the order of top clauses for bw_large.c with deadline 14. While `modoc` cannot find a solution within an hour using many cyclic permutations of the top clauses, it was able to do so when starting at clause 7 in an amazing 4 seconds! Similar behavior has been observed on other structured formulas.

One way to exploit this wide variability is to run multiple copies of Modoc, each using a different top-clause order. Because of extreme variation in the search time, one may be able to find a solution using less computing resource than with a single execution of Modoc, although performance improvement is not guaranteed. This direction of work is currently being investigated as *Parallel Modoc* [Oku98].

6.4. Checker Interchange

The checker-interchange problem is illustrated in Figure 5 for three checkers of each color. The problem is interesting in that it is believed to have only

Table 3

Search time comparison of various SAT testers on checker-interchange formulas. For each problem, the formula in the upper line is unsatisfiable and the one in the lower line is satisfiable. This establishes that the deadline on the lower line is optimal. `walksat` was not run on unsatisfiable formulas (—), and was averaged over 5 runs otherwise. Time-outs are given in hours (h). Time-outs for `walksat` mean that all 5 runs timed out.

num of checkers	dead-line	num of vars	num of literals	CPU seconds (SGI R4400, 150MHz)			
				`walksat`	`sato3`	`2cl`	`modoc`
2	7	90	3,238	—	0.07	2.00	0.12
	8	105	3,900	1.08	0.02	1.90	0.02
3	14	261	16,794	—	14	(>5h)	121
	15	282	18,294	12564	18	(>5h)	39
4	23	570	53,252	—	(>15h)	—	(>15h)
	24	597	55,934	(>5h)	(>15h)	—	12883

one solution if white moves first and one symmetrical solution if black moves first, regardless of the number of checkers. The minimum feasible deadline (if the problem is solvable at all) can be calculated as $n(n+2)$, where there are n checkers of each color.

In each configuration, there are at most two "legal" actions. Therefore, a special-purpose program can be expected to solve $n = 4$ and perhaps $n = 5$ by brute force. It might do even better with the domain knowledge built in that two whites to the left of two blacks forms a permanent blockade. However, the encoding is general (each square is in one of three states, individual checkers are not identified), and each possible action has consequences that have to be propagated through many axioms. Therefore, the satisfiability programs bogged down at $n = 4$.

Table 3 compares `modoc` times against search times of `walksat` (average of five runs), `sato3`, and `2cl` on the checker-interchange family. Results are shown for the maximum infeasible deadline and the minimum feasible deadline in each case. As with the Towers of Hanoi, this family gets hard too fast to provide a good range of useful examples. However, these few examples again demonstrate that `modoc` is able to outperform both incomplete methods and complete methods that lack goal-sensitivity.

Table 4
Performance comparison of `modoc`, `sato3`, and `2cl` on the circuit-testing formulas. Times are average CPU seconds (SGI R4400, 150MHz) for each group; numbers preceded by "±" are standard deviations.

	Circuit Family						
	ssa0432	ssa2670	ssa6288	ssa7552	bf0432	bf1355	bf2670
hline No. Fmlas	7	12	3	80	21	149	53
Vars–Avg	433	1320	10406	1495	886	2266	1300
–Min	427	986	10404	1391	421	1450	694
–Max	435	1359	10410	2013	1057	2298	1784
Lits–Avg	2347	7414	87355	7945	7703	18192	7904
Program							
modoc	0.13	9.65	0.15	0.35	1.02	1.32	4.89
	±0.06	±21.23	±0.01	±0.07	±0.88	±1.98	±15.83
sato3	0.07	6.70	0.37	0.07	1.07	0.16	3.14
	±0.02	±6.46	±0.01	±0.01	±1.19	±0.08	±12.63
2cl	0.30	1246.25	24.23	0.83	4.06	56.02	359.31
	±0.09	±513.28	±3.98	±0.16	±5.25	±58.54	±1991.14

6.5. Circuit Fault Detection

Hardware and software design verification applications can be supported by high performance satisfiability procedures. As with planning formulas, a goal-sensitive theorem-proving view is productive. The formulas tested in this section derive from the hardware application called *Automated Test Pattern Generation* [Lar92]. Faults simulated include "single stuck-at" (code `ssa`) and "bridge" (code `bf`). Circuits are taken from the ISCAS-85 benchmark. We thank Tracy Larrabee and her research group for providing the formulas.

For this application, the conjectured theorem is that there is no setting of the input bits such that the correct circuit and the faulted circuit exhibit a different output. Such a fault is called "redundant" in the literature. If the conjecture is false, the "counter-example" provides a setting of the input bits that permits the specified fault to be detected.

Results on circuit-testing formulas are summarized in Table 4. These formulas were contributed to the 1993 DIMACS Implementation Challenge for Cliques, Coloring, and Satisfiability, and results for `2cl` have been reported [VGT96], as well as for newer solvers [SS96,Zha97]. Because of the mix of satisfiable and unsatisfiable formulas, an incomplete method such as `walksat` is unsuitable.

The most difficult circuit families were `ssa2670` and `bf2670`, although these were not the largest formulas. On all 325 formulas as a group, `2cl` took 42572 CPU seconds (12 hours), `modoc` was 68 times faster at 622 seconds (10 minutes), and `sato3` was twice as fast again with a total of just 300 seconds (5 minutes).

6.6. Global Constraint Problems

Global constraint problems constitute a category of problems that have no specific goals upon which to focus. Graph coloring is a typical example. For such problems, the ideas in this paper are not applicable in any straightforward way because there is no natural division into axioms and conjectured theorem.

When constraint problems involve arithmetic, the satisfiability encodings are extremely verbose. As an example, a variation of the logistics problem might be to deliver all packages at a minimum total cost. We doubt that satisfiability encoding will ever be very effective for these problems, no matter what solvers are developed.

6.7. Discussion

For the classes of formulas reported here the theme that emerges is that the more reasoning a program does, the better it performs, at least among the complete programs we tested. The programs that go beyond unit clause propagation are `2cl`, Grasp, `sato3`, and `modoc`. For these programs, "reasoning" consists of deriving new clauses by resolution. In contrast, `sato3-g0`, which is `sato3` with lemma derivation switched off, and `tableau`, the Crawford-Auton solver found in the Satplan distribution, use only unit clause propagation.

The reasoning theme is exemplified in microcosm with the problem logistics.a from the Satplan distribution [KS96] (see Section 6.2). This problem is represented by one unsatisfiable formula with deadline 10 and one satisfiable formula with deadline 11 Table 1. We now examine the behavior of several programs on this pair of formulas.

The program `sato2` [ZS94] is an implementation of DPLL that was optimized to do extremely fast searching with a minimum of reasoning, or other "overhead". It uses unit clause propagation, but does not even implement the pure literal rule during search. Let us denote `sato3` with lemma derivation switched off as `sato3-g0`. As reported, `sato3-g0` enhances `sato2` with a more sophisticated rule for choosing the splitting variable, but otherwise uses the same search engine. This new rule considers binary clause counts (*cf.* `tableau`). It maintained a search rate of 3600 br/s (branches per CPU second) on the two formulas related to logistics.a.

By comparison, `sato3` [Zha97], which includes the Grasp lemma facility [SS96], ran at only 600 br/s. We can infer that more than 80% of the time in `sato3` is devoted to lemma derivation. Grasp itself ran at only 38 br/s. However, `sato3-g0` did not solve either formula within 5 CPU hours, whereas `sato3` solved them in about one second each, and Grasp required 10–15 seconds each.

For another comparison, `tableau` uses heuristics based on binary clauses, but no reasoning beyond unit clause propagation. On the logistics.a pair it runs at about 1800 br/s, solved the satisfiable formula in 50 CPU minutes, and required 4.4 CPU hours for the unsatisfiable formula. On the other hand, `2cl` actually performs binary-clause reasoning, runs at only 60 br/s, but solves each formula in about one second. There is no equivalent searching rate for `modoc` because it is not a model-search program; `walksat` has a searching rate of 50,000 flips/second on the satisfiable logistics.a formula.

For a satisfiable formula any program's guessing heuristic might be lucky, but for an unsatisfiable formula the entire space must be eliminated through some combination of reasoning and searching. The table below summarizes how many search steps the programs just discussed required for logistics.a with deadline 10.

`sato3`	`2cl`	`Grasp`	`tableau`	`sato3-g0`
544	74	372	28,988,778	(>67,797,019)

The trend seen here carries through to all formula classes reported in this paper. For example, `sato3-g0` did not solve any formula in Table 1 within one CPU hour, and was running 60 times slower than `2cl` on Table 4 when we killed the run. Similar observations apply to `tableau`. Thus `2cl` and `sato3`, which combine reasoning and searching, emerged as the most effective solvers based on model searching.

The motivation for the Grasp lemma system is to achieve dependency-directed backtracking [SS96]. (This is also called intelligent backtracking, non-chronological backtracking, back-jumping, etc.) The idea of dependency-directed backtracking in DPLL has been described briefly by Lee and Plaisted, but with few details [LP92]. Silva and Sakallah rediscovered the idea and sketched an efficient implementation, called Grasp, which turned out to derive lemma clauses. Modoc's lemma methodology [VGO97] is very closely related to theirs. However, Silva and Sakallah decided to assert the lemmas globally, provided they had few enough literals, whereas Modoc keeps the lemmas only as long as they are effectively unit clauses. This may be an important distinction.

Zhang incorporated the Grasp strategy into `sato3-g0`, creating `sato3`, with very impressive results [Zha97]. However, in `sato3`, the lemmas are presented more as a bookkeeping device to control backtracking than as a goal in themselves; e.g., the program does not report anything about lemmas.

It has been known for a long time in the folklore how to extract resolution refutation from a completed DPLL search tree [VGT93]. A careful look at Silva and Sakallah's lemma derivation method indicates that these lemmas are precisely the clauses that this extraction method would derive, at least near the leaves of the tree. This connection appears not to be recognized by the authors. Search pruning at the p-node is possible whenever the first guessed assignment, say $\neg p$, results in a derived clause (i.e., a lemma) that does not contain p, but consists entirely of complements of certain ancestors of the p-node. This clause is also reduced to the empty clause after guessing the opposite assignment at the p-node, so that second guess need not be carried out. Although, this pruning method has been known "in theory" for a long time, Silva and Sakallah described an efficient implementation that achieved excellent practical results. This suggests that the resolvents extracted from completed DPLL subtrees merit further investigation from both theoretical and practical standpoints.

Of the programs extensively tested, it may be reasonably argued that `modoc` and `sato3` are the most based on reasoning, in the sense of deriving new clauses, and these two have performed by far the best on the planning-oriented problem classes. Yet the hardest problems for one program have little correlation with the hardest problems for the other. For example, among the 325 problems in Table 4 only three required over a minute for either program, one for `sato3` and two for `modoc`, and in each case the other program took under 10 seconds. Modoc solved one 4-checker problem in 3.5 hours while `sato3` failed after 15 hours. However, `sato3` solved blocks-world-c problems in seconds where `modoc` took hours. This suggests that, even though `sato3` is faster than `modoc` on average, each program has major strengths lacking in the other, and occasionally these differing strengths are critical. One of the strengths of `modoc` that is absent in `sato3` (and Grasp) is goal sensitivity. A strength of `sato3` (and Grasp) that is absent in `modoc` is the ability to update the global formula with derived clauses. Another strength of `sato3` is its very efficient *trie* data structure, which allows overhead to be controlled through efficient subsumption checking. Can these separate strengths be integrated in some kind of combined approach?

7. Conclusions and Future Work

Approaching planning problems as propositional theorem-proving problems allows focused search and demonstrated better performance using Modoc, an instance of propositional Model Elimination with autarky pruning. Results show that the approach is competitive with and often faster than incomplete stochastic search procedures, which were believed to be the fastest method to find a satisfying truth assignment for encoded planning problems. Among other benefits are that the same program can be used to show unsatisfiability, which is necessary to show optimality of plan lengths. Modoc is definitely faster than model-search programs reported in the literature that rely primarily on search, as opposed to reasoning. However, a recent method of deriving lemmas during the model search is also very effective, and the `sato3` implementation of this idea usually outperforms the current implementation of Modoc. All remarks here apply to the classes of problems studied, which can be cast as planning problems or something closely related.

The planning problems actually solved are simple enough that a human can solve them (when presented in natural language, of course) in a few minutes, even in cases that take hours for programs, or are still unsolved by programs. This indicates that further research on problem representation is needed. We cannot hope for improvements in satisfiability solving to scale up enough to handle genuinely difficult planning problems with the same encoding techniques.

When a problem generates many top clauses, there is a risk that Modoc will get bogged down in an unfavorable top clause, while some other clause leads to a rapid solution. Okushi reports on a parallel method that explores from all designated top clauses simultaneously, and allows multiple "agents" to cooperate by sharing lemmas and autarkies [Oku98].

Future work should proceed along several directions, such as heuristics for guiding the resolution search and further improvements to lemma caching. The success of global lemmas in Grasp and Sato3 requires us to rethink the relatively local scope of lemmas in Modoc. An extension to first-order theorem proving deserves investigation. A closer integration between the SAT compiler and Modoc should produce a more powerful automated planning system.

Acknowledgements

The authors wish to thank Henry Kautz and Bart Selman for providing Satplan and `walksat`, and Michael Ernst, Todd Millstein, and Daniel Weld for providing Medic. Yumi Tsuji wrote most of the code for `2cl`. We thank Hantao Zhang for providing `sato`. This work was supported in part by NSF grant CCR-95-03830, by equipment donations from Sun Microsystems, Inc., and software donations from Quintus Computer Systems, Inc.

References

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

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

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

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

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

[EMW97] 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.

[FN71] 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.

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

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

[GS96] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures – the case study of modal K. In *13th International Conference on Automated Deduction*, pages 583–597. Springer-Verlag, 1996.

[KS96] 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.

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

[Kow79] R. A. Kowalski. *Logic for Problem Solving.* North-Holland, Amsterdam, 1979.

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

[LP92] S.-J. Lee and D. A. Plaisted. Eliminating duplication with the hyper-linking strategy. *Journal of Automated Reasoning*, 9(1):25–42, August 1992.

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

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

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

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

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

[Nil80] N. Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto, CA, 1980.

[Oku98] F. Okushi. Parallel cooperative propositional theorem proving. *Annals of Mathematics and Artificial Intelligence*, 1998. (to appear, preprint at `ftp://ftp.cse.ucsc.edu/pub/avg/AMAI/pmodoc-dist.ps.Z`).

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

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

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

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

[SS96] J. P. Silva and K. A. Sakallah. GRASP–a new search algorithm for satisfiability. In *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design*, pages 220–227. IEEE Comput. Soc. Press, 1996.

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

[VEK76] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *JACM*, 23(4):733–742, 1976.

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

[VGO97] A. Van Gelder and F. Okushi. Lemma and cut strategies for propositional model elimination. *Annals of Mathematics and Artificial Intelligence*, 1997. (to appear, preprint at `ftp://ftp.cse.ucsc.edu/pub/avg/AMAI/lemmas-dist.ps.Z`).

[VGT93] A. Van Gelder and Y. K. Tsuji. Incomplete thoughts about incomplete satisfiability procedures. In *Second DIMACS Challenge Workshop: Cliques, Coloring and Satisfiability*, October 1993.

[VGT96] A. Van Gelder and Y. K. Tsuji. Satisfiability testing with more reasoning and less guessing. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 559–586. American Mathematical Society, 1996.

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

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