Parallel Cooperative Propositional Theorem Proving*

Fumiaki Okushi

Department of Computer Science, California State University, Bakersfield, CA 93311, U.S.A.
E-mail: okushi@cs.csubak.edu

A parallel satisfiability testing algorithm called *Parallel Modoc* is presented. Parallel Modoc is based on *Modoc*, which is based on propositional Model Elimination with an added capability to prune away certain branches that cannot lead to a successful subrefutation. The pruning information is encoded in a partial truth assignment called an *autarky*.

Parallel Modoc executes multiple instances of Modoc as separate processes and allows processes to *cooperate* by sharing lemmas and autarkies as they are found. When a Modoc process finds a new autarky or a new lemma, it makes the information available to other Modoc processes via a "blackboard". Combining autarkies generally is not straightforward because two autarkies found by two separate processes may have conflicting assignments. The paper presents an algorithm to combine two arbitrary autarkies to form a larger autarky.

Experimental results show that for many of the formulas, Parallel Modoc achieves speedup greater than the number of processors. 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.

Keywords: Satisfiability, Model Elimination, autarky, Modoc, cooperation, parallel search

1. Introduction

The Satisfiability Problem (SAT) is a fundamental problem in computer science. Its acceptance problem is NP-complete, suggesting that it is unlikely for it to have a polynomial-time algorithm. However, because of its importance, many practical algorithms have been proposed.

Modoc [25] is a SAT decision procedure based on propositional Model Elimination [17], extended to prune away certain branches that cannot lead to a

* Majority of the work was done while the author was at University of California, Santa Cruz.

successful subrefutation. The pruning information is encoded in a partial truth assignment called an *autarky* [20]. As a descendant of Model Elimination, Modoc also includes a mechanism to record successful subrefutations as *lemmas* and to recall them as necessary.

An advantage of Modoc over the more traditional model-search procedures is that it is able to be *goal sensitive* [25]. Many real-world problems can be viewed as theorem-proving problems. Given this view, Modoc can start a search from one of the *goal clauses*, i.e., the clauses expressing the negated conjectured theorem; this may allow it an efficient backward-chaining search to be performed.

When there is more than one goal clause, there is currently no sure way to predict which goal clause would find a proof in the shortest amount of time. It has been observed that the search time varies widely depending on the order of goal clauses. (See Table 1 for an example.) Because of this, if we could identify "good" goal clauses that would allow Modoc to reach a conclusion quickly, we may be able to cut down on the search time dramatically. Not knowing which ones are good, one way to simply avoid this problem is to simultaneously run multiple Modocs, one for each goal clause, all at the same time. Although this would obviously work, the question is, can we do better?

The paper describes a parallel SAT algorithm called *Parallel Modoc*, which executes multiple instances of Modoc as separate processes. Modoc processes communicate with each other to share autarkies and lemmas. The processes benefit from each other as, for instance, a lemma found by one process can be used by another process to complete its subproof in progress. Combining autarkies generally is not straightforward because two autarkies found by two separate processes may have conflicting assignments. The paper presents an algorithm that allows two arbitrary autarkies to be combined to form a larger autarky. (See Section 4.1.)

Parallel Modoc differs in spirit from other parallel SAT algorithms [24,27,2]. In these algorithms, each process works independently on its share of the search space, with communication used to balance the work load. However, in Parallel Modoc, the processes *cooperate* [4,3,12]. Communication is used to deliver vital information about the formula, which could then be used to shorten the cost of finding a refutation proof, or showing that no refutation is possible.

After standardizing terminologies and notations in Section 2, we begin with an informal review of Modoc in Section 3. Section 4 describes Parallel Modoc, including theorems on multiple autarkies that are used in Parallel Modoc. Sec-

tion 5 reports experimental performance results obtained by running several SAT testers, including Parallel Modoc, on planning formulas. The main part of the paper ends with a summary and on future works in Section 6. For self-containment, two appendices are provided on topics that are not directly related to Parallel Modoc but are referred to in the paper. Appendix A reviews the ideas behind formulating planning problems as SAT problems [13], and Appendix B summarizes *goal-sensitive* simplification [26], which is a simplification scheme suited for backward-chaining theorem provers.

2. Terminologies and Notations

We consider formulas in *conjunctive normal form* (CNF). A CNF formula is a conjunction of *clauses*, each of which is a disjunction of *literals*. A literal is either a variable, or the negation of a variable. Negation will be denoted by "¬". Double negation is ignored, as usual. Formulas and clauses will be expressed using set notation; for clarity, clauses will use "[...]" instead of the usual "{...}". A formula will thus be expressed as in $\{[a, b],\ [a, \neg b],\ [\neg a, c],\ [\neg a, \neg c],\ [\neg a, \neg b, d],\ [\neg a, b, \neg d]\}$.

A *partial truth assignment* is a partial function from the set of variables to the boolean set. Partial truth assignments will be expressed using set notation, as in $\{a, \neg b, c\}$; this means that variable a is assigned true, variable b is assigned false, and variable c is assigned true, and that no other variables are assigned values. A *satisfying truth assignment* is a partial truth assignment that satisfies the formula; that is, each clause has a literal that is in the truth assignment.

The *Satisfiability Problem* (SAT) is: Given a CNF formula, determine whether it has a satisfying truth assignment or not. Although the problem is stated as a decision problem, in practice, we are often interested in a satisfying truth assignment, should the formula turn out to be satisfiable. (For example, in the planning formulas used in Section 5, satisfying truth assignments encode successful plans.)

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). Autarky A allows the satisfiability problem of $\mathcal{F}$ to be reduced to that of $autrem(\mathcal{F}, A)$. The concept

of autarky was first introduced by Monien and Speckenmeyer [20].

Example 1. Let the formula $\mathcal{F}$ be $\{[a, \neg c, \neg e],\ [\neg b, c],\ [\neg a, b, d],\ [\neg d, e]\}$. Then, $\{a, b, c\}$ is an autarky of $\mathcal{F}$, but $\{a, c\}$ is not. Using the autarky $\{a, b, c\}$, the satisfiability problem of $\mathcal{F}$ is reduced to that of $\{[\neg d, e]\}$. □

A property of autarkies that will be used later in a proof is the following.

Lemma 2. A partial truth assignment A is an autarky of $\mathcal{F}$ if and only if every clause in $\mathcal{F}$ that contains a false literal, also contains a truth literal.

Proof. Suppose A is an autarky. Then, A partitions $\mathcal{F}$ into $autsat(\mathcal{F}, A)$ and $autrem(\mathcal{F}, A)$. Since clauses in $autrem(\mathcal{F}, A)$ have none of their literals assigned values, any clause that contains a false literal must belong to $autrem(\mathcal{F}, A)$, which implies that they are satisfied.

Suppose that any clause that contains a false literal also contains a true literal. This allows $\mathcal{F}$ to be partitioned into two sets—the set of satisfied clauses, and the set of clauses none of whose literals are assigned values. This qualifies A to be an autarky. □

3. Modoc

Modoc is a SAT decision algorithm introduced by Van Gelder [25]. It is based on propositional Model Elimination [17] and incorporates a new pruning technique based on the concept of *autarky*. Although the concept of autarky was first introduced by Monien and Speckenmeyer for use in their model-search algorithm [20], Van Gelder adapted it to be used in a refutation-search procedure, to prune away certain branches that cannot lead to a successful subrefutation. Van Gelder also showed that autarkies can be derived during failed subrefutation attempts, and that a satisfying truth assignment can be obtained by combining the autarkies, should the formula turn out to be satisfiable. This section informally describes the algorithm. Details of Modoc can be found elsewhere [25].

As a refutation-search procedure, the aim of Modoc is to find a *refutation proof*, demonstrating that the formula is inconsistent. 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 basic operation—*PDT extension*.

We define PDT below. PDT essentially has the same structure as the clause trees used by others [19,15].

1. 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 (or with $\top$, described later).
2. 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.
3. A clause node labeled with C has a goal node labeled with g as a child if and only if
   (a) g is a literal in C, and
   (b) $\neg g$ does not label any ancestor goal node of C.

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

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. If no refutation tree can be constructed, the formula is satisfiable. For convenience, we will call a subtree of a refutation tree a *refutation subtree*.

Modoc tries to construct a refutation tree in a depth-first fashion starting from a tree with only the verum node using its only operation, *PDT extension*. The *PDT extension* operation extends a goal node with a clause node that contains the complement of the goal node but does not contain any of the ancestor goal nodes. (If the goal node is the verum node, then any clause can be used to extend the goal node. The clause used to extend the verum node is called the *top clause*.) This operation also adds goal nodes beneath the just-added clause node; there is one goal node for each literal in the clause node that 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 an ancestor goal node other than the parent goal node.

For sake of efficiency, it is not necessary for Modoc to actually construct every part of a refutation tree. That is, if the outcome of subtree construction

is known, Modoc may move on to other parts of the tree whose outcome is not known. There are two situations where this could happen. One is when it is known that it is possible to construct a refutation subtree beneath a goal node. Another is when it is known that it is *not* possible to construct a refutation subtree using a particular clause in a PDT extension operation. The former involves the use of a *lemma*, and the latter involves the use of an *autarky*.

When a refutation subtree is successfully constructed for a goal node, that fact could be recorded as a *lemma*. A lemma records the complement of the literal labeling the goal node, as well as the "premise" in which it was successful; the premise consists of the ancestor goal nodes that suppressed the creation of goal nodes below the refuted goal node. At a later time, if a goal node with the same literal is created, and the set of its ancestor goal nodes is a superset of the premise in which the literal had a refutation subtree constructed earlier, subtree construction need not be carried out as it is obvious that a refutation subtree could be constructed.

The original lemma strategy of Model Elimination was to record lemmas as clauses. However, this had the problem of increasing the number of eligible extension clauses that need to be tried [9]. Also, the lemma clauses tend to be highly redundant because they were often subsumed by other clauses [23]. Modoc's lemma strategy is an extension of Shostak's *C-literal* strategy [23] that solved these problems.

A behavior discovered by Van Gelder and exploited in Modoc is that an autarky can be derived during a failed attempt to construct a refutation subtree. In essence, all the goal nodes that were part of the failed attempt can be turned into an autarky. (See Example 3.) Further, he also discovered that any clause that is satisfied by an autarky cannot label any clause node in a successful refutation subtree. Modoc uses this fact to safely exclude clauses from the set of possible PDT extension clauses.

Example 3. Consider the formula

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

Figures 1a through 1g show how Modoc may try to construct a refutation tree for the formula. Details of the operations of Modoc are given in the captions. In particular, suppression of goal node creation can be seen in Figures 1c and 1g, autarky derivation can be seen in Figures 1c and 1d, autarky pruning can be seen

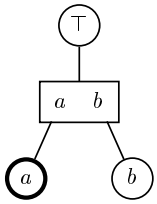

Figure 1a. (Example of Modoc execution.) We consider the formula $\{[a, b], [a, \neg b], [\neg a, c], [\neg a, \neg c], [\neg a, \neg b, d], [\neg a, b, \neg d]\}$ in this example. Clause nodes are shown in rectangles and goal nodes are shown in circles. Thick circles indicate where the search is. Clause $[a, b]$ is chosen as the top clause. Two goal nodes a and b are immediately created.

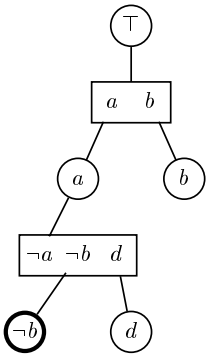

Figure 1b. (Example of Modoc execution.) All four clauses containing $\neg a$ are eligible to extend goal node a. Here, Modoc extends goal node a with clause node $[\neg a, \neg b, d]$. This creates two new goal nodes $\neg b$ and d.

in Figure 1f, lemma derivation can be seen in Figure 1h, and the use of a lemma can be seen in Figure 1i. □

One advantage of Modoc (and other backward-chaining search procedures) over model-search procedures is that it is able to be "goal sensitive" [25]. Many real-world problems can be viewed as theorem-proving problems. A formula derived from such a problem comprises of two parts—the axioms, and the negated conjectured theorem, i.e., the *goal clauses*. The axioms are obviously consistent; thus, to test whether the formula is inconsistent or not, it is sufficient to start a refutation attempt only from the goal 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 [26].

While the idea of goal-sensitive search has been a success for Modoc, a

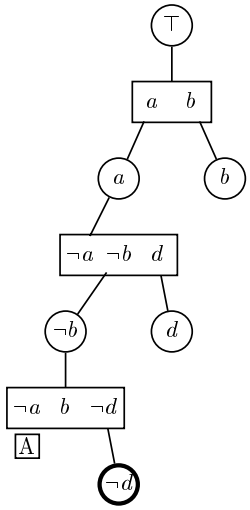

Figure 1c. (Example of Modoc execution.) Only clause $[\neg a, b, \neg d]$ is eligible to extend goal node $\neg b$. (Clause $[a, b]$ is ineligible because it contains an ancestor goal node a.) Thus, Modoc extends goal node $\neg b$ with clause node $[\neg a, b, \neg d]$. This creates a new goal node $\neg d$. Note that the creation of goal node $\neg a$ was suppressed (indicated by A). 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 $\{\neg d\}$.

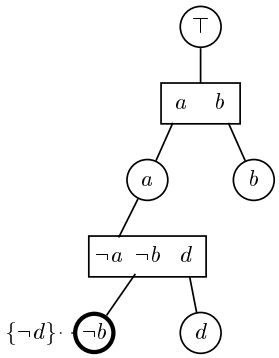

Figure 1d. (Example of Modoc execution.) Modoc now backtracks to goal node $\neg b$ with autarky $\{\neg d\}$. The autarky is *conditional* in the sense that it is an autarky for the formula resulting from strengthening the original formula with the partial truth assignment implicit by the set of ancestor 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 $\{\neg d\}$.

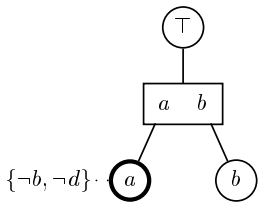

Figure 1e. (Example of Modoc execution.) Modoc now backtracks to goal node a with autarky $\{\neg b, \neg d\}$. Again, the autarky is a conditional autarky that is conditioned on the set of ancestor goal nodes $\{a\}$. Modoc now tries to extend goal node a with some other eligible clause.

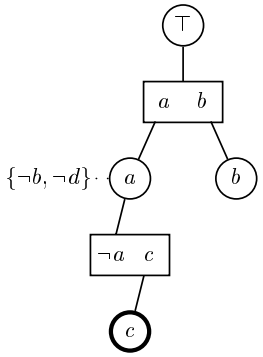

Figure 1f. (Example of Modoc execution.) There are two clauses eligible to extend goal node a. (Clause $[\neg a, b, \neg d]$ is not eligible as it is satisfied by the autarky $\{\neg b, \neg d\}$.) Here, Modoc extends goal node a with clause node $[\neg a, c]$. This creates a new goal node c.

Table 1
Search times of `modoc` on a block-world planning formula (bw_large.c for deadline 14) for different goal-clause orders. Times are CPU seconds on an SGI Challenge (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.

goal-clause order	CPU seconds
3,...,15,1,2	911
7,...,15,1,...,6	4
8,...,15,1,...,7	24
11,...,15,1,...,10	745

problem remains in deciding in *what order* should we try the goal clauses? Our experience with Modoc shows that the search time varies widely depending on the order of goal clauses. Table 1 shows how search time varies on one planning formula reported in Section 5. Because of the wide variability, from a practical

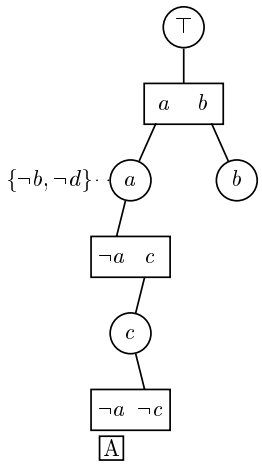

Figure 1g. (Example of Modoc execution.) Only clause $[\neg a, \neg c]$ is eligible to extend goal node c. Thus, Modoc extends goal node c with clause node $[\neg a, \neg c]$. The creation of goal node $\neg a$ was suppressed (indicated by A) because its complement a is a non-parent ancestor goal node. This completes the refutation along this branch.

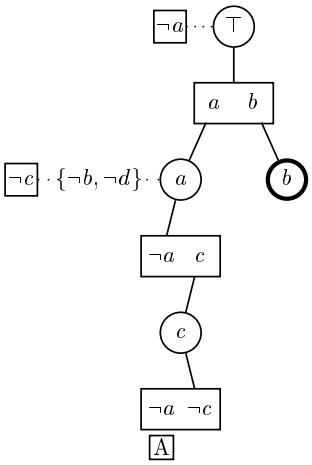

Figure 1h. (Example of Modoc execution.) The successful refutation of goal node c causes a lemma literal $\neg c$ to be derived and attached to goal node a. It also causes a lemma literal $\neg a$ to be derived and attached to the verum node. Search must now continue to refute goal node b.

point of view, it is crucial for us to be able to defer "bad" goal clauses in favor of "good" goal clauses. However, it is unlikely that we will one day be able to tell them apart without actually solving the problem. Not knowing which ones are "good" and which ones are "bad", one obvious solution to this would be to simply run multiple copies of Modoc, one for each goal clause, all at the same

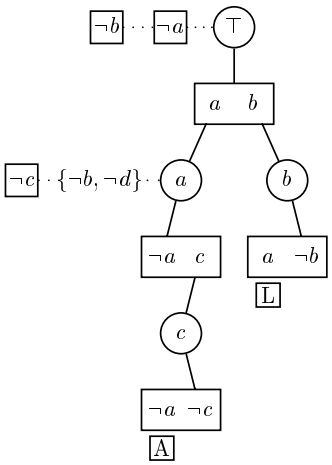

Figure 1i. (Example of Modoc execution.) There are two clauses eligible to extend goal node b. Here, Modoc extends goal node b with clause $[a, \neg b]$. The creation of goal node a is suppressed because its complement $\neg a$ is a lemma literal attached to an ancestor (indicated by L). This completes the refutation along this branch, and also the refutation for this formula, as all leaf nodes are now clause nodes.

time. While this would avoid the problem of a search getting "caught" with a bad top clause, as a research question, we would like to know if it is possible to do better.

4. Parallel Modoc

Parallel Modoc executes multiple instances of Modoc as separate processes, one for each goal clause. It can be viewed as a simple multi-agent system, in which Modoc processes act as agents. The processes *cooperate* [4,3,12] in finding a solution by sharing lemmas and autarkies as they are found. As an example, a lemma found by process 3 could help process 2 complete its subproof in progress, which then derives a lemma that could be used by process 1 to complete its subproof. This contrasts with many other parallel SAT testers [24,27,2] whose processes work independently, and communication used merely to balance the work load. It could be said that the main use of parallel processing in these algorithms is to simply increase throughput (i.e., to examine more "nodes" in a unit time). In Parallel Modoc, parallel processing is also used to allow cooperation. It executes multiple searches starting from different goal clauses and allows information discovered about the formula to be shared.

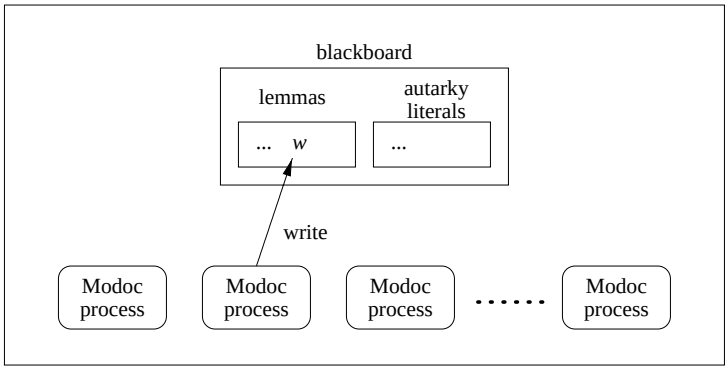

Figure 2a. (Lemma sharing in Parallel Modoc.) A Modoc process finds a new lemma w and writes it to the blackboard.

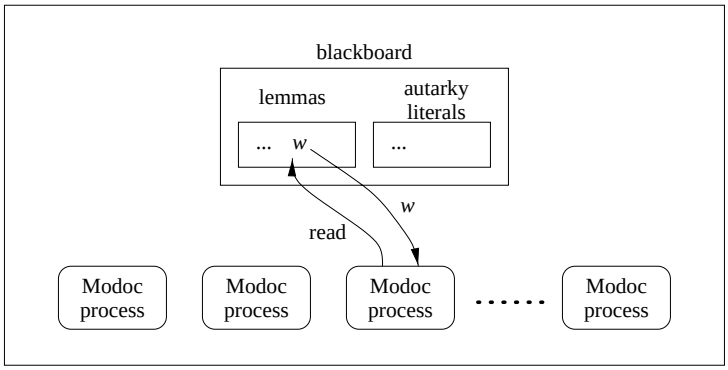

Figure 2b. (Lemma sharing in Parallel Modoc.) A different Modoc process incorporates the new lemma w into its collection of lemmas.

In Parallel Modoc, when a Modoc process finds a new autarky or a new lemma, apart from saving it for its own future use, it makes the information available to other Modoc processes via a "blackboard". The blackboard is a shared data structure to which new information about the formula is written. Other Modoc processes will later incorporate the new information found in the blackboard into their local pool of information.

In the current implementation of Parallel Modoc, only the autarkies and lemmas that are not conditioned on any ancestor goal nodes are shared. 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 processes.

Figures 2a and 2b describe how Modoc processes may share a top-level

lemma with other Modoc processes in Parallel Modoc. Note that a top-level lemma consists only of a single literal (as it has no "premise"). When a Modoc process derives a new top-level lemma w, it writes it to the blackboard (Figure 2a). At a later time, other Modoc processes will incorporate w into their collection of lemmas (Figure 2b). If the process is currently attempting to refute a goal node labeled with $\neg w$, the lemma may be used to complete that subrefutation. Further, the process may use the new lemma to avoid constructing a refutation subtree for goal nodes labeled with $\neg w$ in future refutation attempts,

4.1. Combining Autarkies

Sharing autarkies is not straightforward. This is because a formula may have *conflicting* autarkies, that is, two autarkies A_1 and A_2 such that for some variable x, x is in A_1 while $\neg x$ is in A_2. Although this would not happen within a lone Modoc process, with multiple Modoc processes running allowing multiple searches to be made at the same time, it becomes a possibility. Without a means to combine them, it would require Modoc to store multiple autarkies separately, which may possibly become a bookkeeping nightmare. However, Theorem 6 below 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 autarkies.

To do this, we first define a new operator over the set of partial truth assignments.

Definition 4. Let A_1 and A_2 be partial truth assignments. Then, we define $A_1 \curvearrowleft A_2$ as

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

where $\bar{A_1} = \{\neg x \mid x \in A_1\}$. We will say that A_1 is given *preference* over A_2 in resolving conflicting assignments. □

Example 5. Let two partial truth assignments A_1 and A_2 be as in Theorem 6 below. Then,

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

□

Theorem 6. Let two autarkies of a CNF formula $\mathcal{F}$ be as follows:

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

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,

1. both $A_1 \curvearrowleft A_2$ and $A_2 \curvearrowleft A_1$ are autarkies of $\mathcal{F}$, and
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. Let $A = A_1 \curvearrowleft A_2$ for brevity.

1. By Lemma 2, it is sufficient to show that any clause that contains $\neg u_i$ or $\neg v_i$ or $\neg w_i$ for some i contains a literal that is in A.
   Let C be a clause that contains $\neg w_i$ for some i. Since A_1 is an autarky that includes w_i, A_1 satisfies C, and thus, there must be some literal x in A_1 that is also in C. Since A_1 is a subset of A, x must also be in A. 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 includes v_i, A_2 satisfies C, and thus, there must be 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 in A. The second case reduces to an earlier case.
2. It is sufficient to show that set containment holds both ways.
   Let C be a clause that is satisfied by A. This means that there is some literal x in C that is also in A. 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 is a subset of A, x must also be in A, and thus, C is satisfied by A.
   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,

and thus, C is satisfied by A. In the second case, since A is an autarky that contains w_i, C is satisfied by A as well.

□

As a result of combining, it is possible for a literal that was in one of the two autarkies to not be in the new autarky. This may raise a concern that the new autarky may not encode the information that any clause containing this literal cannot lead to a successful subrefutation. However, one need not worry. By Theorem 6, any clause that contains this literal is satisfied by the new autarky. This means that for each clause that contains this literal, there is some other literal in that clause that is in the new autarky. Thus, we arrive at the following corollary.

Corollary 7. From the point of view of pruning away certain branches that cannot lead to a successful subrefutation, no information is lost during combining autarkies. □

As a practical concern, particularly in a distributed computing environment where communication is made over a computer network, transmitting new autarkies as they are found may be costly, as they tend to be large. Theorem 8 below shows that it is sufficient to transmit only the new autarky literals found since the last transmission.

Theorem 8. Let A_1, A_2, and A_3 be autarkies for a formula such that $A_2 \subseteq A_3$. Then,

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

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:

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

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 9. It should be noted that when only the difference is transmitted between two processes, the same process must always be given preference in resolving conflicting assignments. A counter-example can be constructed if this is not followed. (See Example 10 below.)

Example 10. Let the formula $\mathcal{F}$ be $\{[x], [y, v], [z], [u], [w], [t]\}$. Suppose there are two processes P_1 and P_2, and that while P_1 finds autarky $A_1 = \{x, \neg y, u, v, w\}$, P_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 combining autarkies:

1. Combine A_1 and A_2, giving preference to P_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 P_2 to resolve conflicting assignments. Let A_{312} denote the resulting partial truth assignment.

Then,

$$\begin{aligned}A_{12} &= A_1 \curvearrowleft A_2 \\ &= \{x, \neg y, u, v, w\} \curvearrowleft \{x, y, z, \} \\ &= \{x, \neg y, z, u, v, w\};\end{aligned}$$

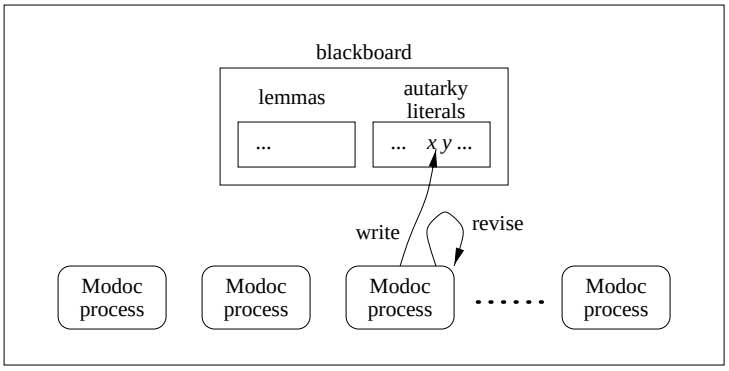

Figure 3a. (Autarky sharing in Parallel Modoc.) A Modoc process finds a new autarky. It writes to the blackboard the new autarky literals that do not conflict with the autarky in the blackboard. The process must revise its autarky if there is a conflict.

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

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

One way to satisfy the condition in Remark 9 is to give only one entity preference over all other entities in resolving conflicts among autarky assignments. In the current implementation of Parallel Modoc, the blackboard is given preference over all Modoc processes. This requires a Modoc process to *revise* its autarky if there is a conflict between its autarky and the autarky in the blackboard.

Figures 3a and 3b show how autarkies are communicated in Parallel Modoc. When a Modoc process finds a new autarky, it writes to the blackboard new autarky literals that do not conflict with any of the autarky literals in the blackboard (Figure 3a). If there are conflicts, the process must revise its autarky. At a later time, other Modoc processes will incorporate the new autarky literals by possibly revising their own autarky (Figure 3b). If the process is currently attempting a subrefutation that includes a clause node that is now satisfied by the enlarged autarky, the search may fail and backtrack to the parent goal node of the highest such clause node. Further, the process may use the new autarky to

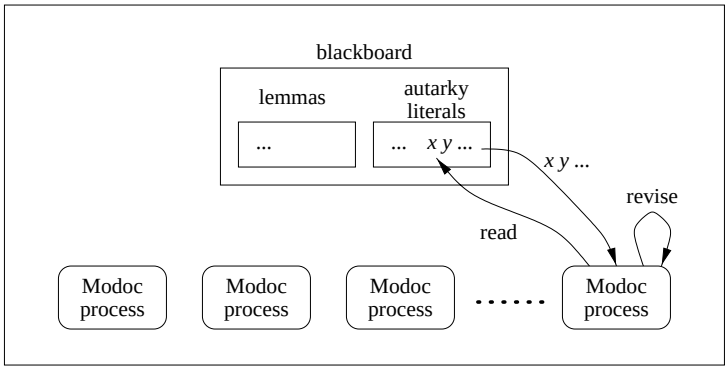

Figure 3b. (Autarky sharing in Parallel Modoc.) A different Modoc process incorporates the new autarky literals by possibly revising its autarky.

exclude clauses that are now satisfied by it from future PDT extension operations.

5. Experimental Results

To assess the relative performance of Parallel Modoc over Modoc and other SAT testers, Parallel Modoc was implemented in C. In this section, results obtained on running various SAT testers on planning formulas are reported. The results show that Parallel Modoc often exhibits super-linear speedup. Such improvement is due to the extreme skewness of the distribution of search times depending on the top clause, as observed in Tables 1, often helped by cooperation among the Modoc processes. The amount of improvement varies widely depending on the formula.

The skewed distribution of combinatorial search times has been studied by others [22,10]. While the author's experience with Modoc agrees with the notion of "heavy-tailed" distribution, the author does not have sufficient data to fully explain the distribution of Modoc search times at this time. Others have studied the behavior of parallel searches [18], among them are studies showing that not only is super-linear speedup possible, it can be expected under certain circumstances [21,22].

Planning formulas are formulas generated from planning problems. In the experiments, the formulas were generated using two "SAT compilers"—Satplan [14] and Medic [7]. Satplan takes hand-coded axioms, while Medic takes axioms described as STRIPS-style operators [8]. Each formula has a deadline associated to it and has the property that it is satisfiable if and only if there is a

successful plan that meets the deadline. For the experiments, two formulas were generated for each problem. 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. Basic ideas behind formulating planning problems as SAT can be found in Appendix A. Details can be found elsewhere [13,14,7].

After the formulas were generated, they were simplified. For Modoc and Parallel Modoc, simplification must preserve not only satisfiability, but also models and goal clauses. We call this type of simplification *goal-sensitive* simplification [26]. More information on goal-sensitive simplification can be found in Appendix B. In contrast, we will refer to the usual simplification as *goal*-in*sensitive* simplification.

The particular C implementations of Modoc and Parallel Modoc that were used in the experiments are denoted by `modoc` and `pmodoc`, respectively. In `pmodoc`, the blackboard was implemented in System V shared memory segments. Other SAT testers included in the experiments are `satz` [16] and `relsat` [1], which are both complete model-search procedures based on DPLL [6,5]. `satz` incorporates a highly optimized branch-variable selection heuristic that uses unit propagation to score variables. `relsat` incorporates a two-step branch-variable selection heuristic that may make stochastic choices, as well as a mechanism to record (and discard) *nogoods*, which are information derived from failed searches. Learn orders of 3 and 4 were used, as recommended by the author. However, because learn order of 4 did better in almost all cases, only the search times from that runs are reported.

Two four-processor computer systems with different memory sizes were used to run the experiments. The SGI Challenge system had 64MB of main memory and was used to run experiments with smaller formulas. The SGI Onyx system had 256MB of main memory and was used to run experiments with larger formulas. Both systems had four 150MHz R4400 processors. The time-sharing capability provided by the operating system (IRIX version 5.3) was used to execute as many Modoc processes as necessary.

5.1. Improvements by Parallel Search and Sharing

Tables 2a and 2b show the search times on hard planning formulas generated by Medic. (Medic may generate formulas in a number of different encodings.

Table 2a

Search times of various SAT testers on the satisfiable hard planning formulas generated by Medic. Because of the criterion used to choose the formulas (see Section 5.1 for details), the numbers are not a fair representation of `modoc`'s capabilities. Number of variables and literals are after simplification. Times are CPU seconds for `satz`, `relsat`, and `modoc`, and elapsed seconds for `pmodoc`; they were obtained on a 4-CPU SGI Challenge (150MHz R4400). `relsat` times are average of 5 runs. '?' indicates that the run was terminated after 1 hour; for `relsat`, it means that none of the 5 runs found a solution in 1 hour.

problem/ deadline	num of goal clauses	encoding	num of vars	num of literals	search time (seconds)				
					`satz`	`relsat`	`modoc`	`pmodoc` (share?)	
								(no)	(yes)
big-bw1/11	6	ccse	980	77,629	?	863	?	?	?
		ecse	798	14,090	3	13	?	16	16
fridge2/13	2	cbse	180	10,485	?	9	?	25	24
		ccse	346	10,370	?	17	?	78	78
		crse	310	9,880	?	15	?	582	581
hanoi3/7	3	cbse	158	25,468	?	2088	618	98	70
monkey2/9	2	cbse	250	37,696	?	31925	?	?	?
		cfst	331	14,465	?	22	974	909	387
		crse	601	48,757	1075	77235	?	?	?
tire2/14	6	ccse	677	41,343	5	598	3375	164	83
		cfst	623	34,944	?	3213	?	?	?
		crse	628	40,002	66	3097	?	?	1281
		efst	512	19,309	229	14	?	3	3

The encoding has been included in the tables to merely help identify the formula. Explanation of these encodings may be found elsewhere [7].) The problems were selected based on an earlier study in which the corresponding satisfiable formulas caused `modoc` to time out after 10 minutes. Because of this selection criterion, the numbers are not a fair representation of `modoc`'s capabilities. The number of variables and clauses are for the formula generated after goal-insensitive simplification.

With few exceptions, the factor of improvement by `pmodoc` over `modoc` was far greater than the number of processors. Formulas that could not be solved in one hour by `modoc` 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.

Table 2b
Search times of various SAT testers on the unsatisfiable hard planning formulas generated by Medic. More information can be found in the caption for Table 2a.

problem/ deadline	num of goal clauses	encod- ing	num of vars	num of literals	search time (seconds) satz	relsat	modoc	pmodoc (share?) (no)	(yes)
big-bw1/10	6	ccse	888	69,851	?	1408	?	?	?
		ecse	707	12,375	5	3	222	223	115
fridge2/12	2	cbse	166	9,616	?	11	?	216	108
		ccse	318	9,492	?	13	?	2113	1869
		crse	285	9,043	?	16	?	1528	444
hanoi3/6	3	cbse	135	21,388	?	628	79	45	26
monkey2/8	2	cbse	222	33,194	?	?	?	?	1701
		cfst	291	12,532	2241	16	301	302	61
		crse	529	42,641	903	?	1626	462	178
tire2/13	6	ccse	625	38,014	802	1016	?	?	?
		cfst	577	32,177	?	2358	?	?	?
		crse	580	36,783	623	4325	?	?	?
		efst	467	17,385	94	7	1900	1902	1968

Autarky sharing 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. Lemma sharing occurred on all the formulas, but no correlation appears to exist between the number of shared lemmas (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 attempted refutation, 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 shared lemmas than the satisfiable 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 using sharing and another without sharing, were the same. This means that the shared lemmas did not help at all to shorten the search for these formulas.

Table 3a

Search times of various SAT testers on the satisfiable hard planning formulas generated by Satplan. Number of variables and literals are after simplification. Times are CPU seconds for `satz`, `relsat`, and `modoc`, and elapsed seconds for `pmodoc`; they were obtained on a 4-CPU SGI Onyx (150MHz R4400). `relsat` times are average of 5 runs. '??' indicates that the run was terminated after 5 hours.

problem/ deadline	num of goal clauses	num of vars	num of literals	search time (seconds)				
				`satz`	`relsat`	`modoc`	`pmodoc` (share?)	
							(no)	(yes)
logistics.a/11	8	638	13,089	1.07	0.31	0.54	8.71	3.38
logistics.c/13	7	897	21,412	413	10	3	39	5
bw_large.c/14	15	2,222	78,146	7	31	15439	25	17
bw_large.d/18	19	4,714	205,559	4518	431	58	239	101

Table 3b

Search times of various SAT testers on the unsatisfiable hard planning formulas generated by Satplan. More information can be found in the caption for Table 3a.

problem/ deadline	num of goal clauses	num of vars	num of literals	search time (seconds)				
				`satz`	`relsat`	`modoc`	`pmodoc` (share?)	
							(no)	(yes)
logistics.a/10	8	541	10,598	0.72	0.24	3.15	9.83	4.55
logistics.c/12	7	787	18,244	2412	149	1132	2401	1195
bw_large.c/13	15	1,935	66,547	17	16	5389	2695	2010
bw_large.d/17	19	4,275	184,180	2796	663	??	??	??

In comparison with the two other SAT testers, `satz` and `relsat`, `pmodoc` did better on some of the formulas and worse on the others. (`relsat` runs that timed out were rerun with the same seed if one of the runs for the same formula had finished within 1 hour. This was to allow the average to be computed. However, this has allowed more numbers to be represented for `relsat` than for the other SAT testers.) As a SAT tester, the performance of `pmodoc` is limited to some degree by the underlying search engine, `modoc`. At this time, `modoc` incorporates very limited heuristics to guide its search.

Tables 3a and 3b show 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 mini-study (Figure 1) shows that had it started with the 7th goal clause, it could have solved the formula in 4 seconds. Although this formula

Table 4a

Search times of various SAT testers on the satisfiable checker formulas. Number of variables and literals are after simplification. Times are CPU seconds for `satz`, `relsat`, and `modoc`, and elapsed seconds for `pmodoc`; they were obtained on a 4-CPU SGI Onyx (150MHz R4400). `relsat` times are average of 5 runs. '??' indicates that the run was terminated after 5 hours; for `relsat`, it means that none of the 5 runs found a solution in 5 hours.

num of checkers	dead-line	num of goal clauses	num of vars	num of literals	search time (seconds)				
					`satz`	`relsat`	`modoc`	`pmodoc` (share?)	
								(no)	(yes)
2	8	4	105	3,900	0.34	0.07	0.02	0.08	0.13
3	15	6	282	18,294	4	64	39	2	3
4	24	8	597	55,934	13431	11377	12883	1626	1606

Table 4b

Search times of various SAT testers on the unsatisfiable checker formulas. More information can be found in the caption for Table 4a.

num of checkers	dead-line	num of goal clauses	num of vars	num of literals	search time (seconds)				
					`satz`	`relsat`	`modoc`	`pmodoc` (share?)	
								(no)	(yes)
2	7	4	90	3,238	0.33	0.15	0.12	0.30	0.48
3	14	6	261	16,794	93	112	121	199	186
4	23	8	570	53,252	??	??	??	??	??

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`. In fact, `pmodoc` solved it in 76 (elapsed) seconds on an equivalent single-processor system.

Tables 4a and 4b show results on the "checker" formulas. The checker formulas are planning formulas generated from a game based on the one-dimensional version of Chinese Checkers. Figure 4 shows the aim of the 4-checker problem; it also shows the possible first few moves. The problem is interesting in that it is believed to have only two plans, counting symmetries, regardless of the number of checkers.

Improvements by `pmodoc` were observed on the satisfiable formulas. However, sharing did not help at all in most cases. On the unsatisfiable 3-checker formula, the (elapsed) search time of `pmodoc` was more than the search time of `modoc`. This was reflected in the number of PDT extensions performed by

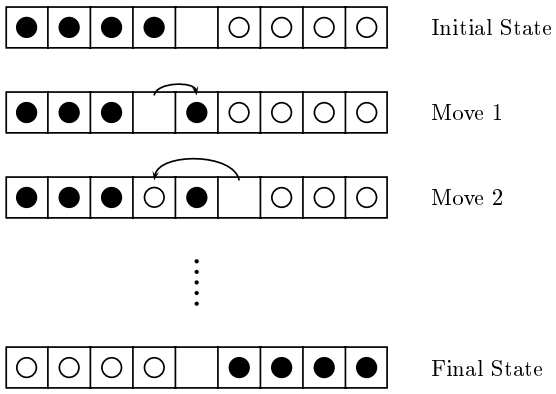

Figure 4. Aim of the 4-checker problem. The possible first few moves are also shown.

the Modoc process that proved that the formula was unsatisfiable—while `modoc` found a refutation in 698,469 PDT extensions, the Modoc process that found a refutation in `pmodoc` took 720,413 PDT extensions without sharing and 722,043 PDT extensions with sharing.

5.2. Improvements by Increasing the Number of Goal Clauses

One problem in the current implementation of Parallel Modoc is that the amount of parallelism is tied to the number of goal clauses in the formula. To study the effect and the potential improvement due to increasing the number of goal clauses, a preprocessor was run on the formulas in Tables 2a and 2b to increase the number of goal clauses. The preprocessor first derived all the resolvents involving the goal clauses, and then it replaced the goal clauses with the resolvents and made the resolvents the new goal clauses.

Tables 5a and 5b compare search times of `pmodoc` on some of the formulas listed in Tables 2a and 2b. To allow comparison between different numbers of goal clauses (and hence between different numbers of processes on a 4-CPU system), times were estimated for an equivalent system with as many processors as goal clauses using the following formula:

$$T(\text{estimated}) = \begin{cases} T(\text{measured}) & \text{if } \mathit{Num}(\text{goal clauses}) \leq 4 \\ T(\text{measured}) \times \dfrac{4}{\mathit{Num}(\text{goal clauses})} & \text{if } \mathit{Num}(\text{goal clauses}) > 4 \end{cases}.$$

The estimates computed using the above formula were confirmed to give rough estimates by running some of the runs on a similar computer system with more

Table 5a

Effect of the increased number of goal clauses on `pmodoc` search times. The formulas are satisfiable and were derived from the formulas reported in Table 2a. Times were obtained on a 4-CPU SGI Challenge (150MHz R4400) and then estimated for an equivalent computer system with as many CPUs as goal clauses.

problem/ deadline	encoding	num of goal clauses		`pmodoc` search time (elapsed seconds)	
		orig	inc'd	original	increased
big-bw1/11	ecse	6	18	11	6
fridge2/13	cbse	2	18	24	11
	ccse	2	28	78	63
	crse	2	28	581	218
hanoi3/7	cbse	3	45	70	40
monkey2/9	cbse	2	15	?	?
	cfst	2	14	387	308
tire2/14	efst	6	23	2	2

Table 5b

Effect of the increased number of goal clauses on `pmodoc` search times. The formulas are unsatisfiable and were derived from the formulas reported in Table 2b. More information can be found in the caption for Table 5a.

problem/ deadline	encoding	num of goal clauses		`pmodoc` search time (elapsed seconds)	
		orig	inc'd	original	increased
big-bw1/10	ecse	6	18	77	55
fridge2/12	cbse	2	18	108	107
	ccse	2	28	1869	1759
	crse	2	28	444	773
hanoi3/6	cbse	3	45	26	18
monkey2/8	cbse	2	15	1701	1208
	cfst	2	14	61	45
tire2/13	efst	6	23	1312	190

processors. Improvements of varying degree were observed on most of the formulas.

Improvement in search time on the unsatisfiable version of fridge2 in cbse en-

coding was negligible. However, the execution log shows that the Modoc process that found the refutation proof found it in 463,830 PDT extensions, which is down 26% from 553,709 PDT extensions with the original formula, and that the number of shared top-level lemmas increased from 28 to 41. However, it also shows that the number of PDT extensions performed per processor-second (Pe/ps) went down from 4824 Pe/ps to 3873 Pe/ps, down 20%. This may have countered the decrease in the number of PDT extensions performed by the Modoc process that found a refutation, and explain the near lack of improvement in search time. However, we are currently not able to fully explain the decrease in the number of PDT extensions per processor-second.

Similar decrease in the number of PDT extensions per processor-second was observed on the unsatisfiable version of fridge2 in crse encoding; it went down from 5140 Pe/ps to 3994 Pe/ps, down 22%. However, what made this run interesting was that the number of PDT extensions performed by the Modoc process that found the refutation went up from 2,177,401 to 3,239,738, by 49%. 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 the number of goal clauses increased. This is despite that running `pmodoc` on the formula with the number of goal clauses increased is essentially running Modoc 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; 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.

6. Summary and Future Works

A parallel satisfiability tester called Parallel Modoc was presented. Parallel Modoc runs multiple Modoc processes to allow simultaneous searches from different goal clauses and to allow cooperation among the processes by sharing valuable information found about the formula. The approach contrasts with other parallel SAT testers in that communication is a vital part of the algorithm.

Experimental results show that the factor of improvement over Modoc on many of the planning formulas was greater than the number of processors. This

shows that Parallel Modoc benefits greatly from the parallel searches and not simply from the increase in the number of processors. Potential improvement due to increasing the number of goal clauses was also observed. In all cases, the amount of improvement varied widely depending on the formula.

It is worth mentioning that Parallel Modoc does not require a parallel computer to run it. As long as multiprocessing is supported, Parallel Modoc will run, albeit slower. As an extreme case, running Parallel Modoc on a single-processor computer system may still be faster than running Modoc. This is because of the extremely skewed distribution of search times depending on the order of goal clauses, found in many of the formulas. Parallel Modoc takes advantage of this distribution and executes multiple searches from different goal clauses, all at once.

Future work could proceed along several directions. One would be to increase the degree of parallel search and the extent of cooperation. Another would be to decouple the maximum number of Modoc processes from the number of goal clauses.

Acknowledgements

The author wishes to thank Allen Van Gelder for discussion on Modoc. "SAT compilers" were provided by their respective authors. The author wishes to thank Henry Kautz and Bart Selman for providing Satplan, and Michael Ernst, Todd Millstein, and Daniel Weld for providing Medic. SAT testers other than `modoc` and `pmodoc` were provided by their respective authors. The author wishes to thank Chu Min Li for providing `satz`, and Roberto Bayardo for making `relsat` available via his home page. The author's work at the University of California, Santa Cruz was supported in part by NSF grant CCR-95-03830.

References

[1] R. J. Bayardo, Jr. and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 203–208, 1997.

[2] 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.

[3] 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.

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

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

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

[7] 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.

[8] 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.

[9] 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.

[10] C. P. Gomes, B. Selman, and N. Crato. Heavy-tailed distributions in combinatorial search. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP97)*, pages 121–135, 1997.

[11] C. Green. Application of theorem proving to problem solving. In *Proceedings First International Joint Conference on Artificial Intelligence*, 1969.

[12] 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.

[13] H. Kautz and B. Selman. Planning as satisfiability. In *ECAI 92. 10th European Conference on Artificial Intelligence Proceedings*, pages 359–363, Chichester, UK, 1992. Wiley.

[14] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI '96)*, 1996.

[15] 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.

[16] C. M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problem. In *Proceedings International Joint Conference on Artificial Intelligence*, pages 366–371, 1997.

[17] D. W. Loveland. A simplified format for the model elimination theorem-proving procedure. *JACM*, 16(3):349–363, 1969.

[18] W. G. Macready, A. G. Siapas, and S. A. Kauffman. Criticality and parallelism in combinatorial optimization. *Science*, 271, 5 January 1996.

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

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

[21] V. N. Rao and V. Kumar. On the efficiency of parallel backtracking. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):427–437, April 1993.

[22] R. Shonkwiler and E. Van Vleck. Parallel speed-up of monte carlo methods for global optimization. *Journal of Complexity*, 10:64–95, 1994.

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

[24] 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. First International Conference Proceedings*, pages 985–993. Springer-Verlag, 1988.

[25] 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`).

[26] A. Van Gelder and F. Okushi. A propositional theorem prover to solve planning and other problems. *Annals of Mathematics and Artificial Intelligence*, 1997. (to appear, preprint at `ftp://ftp.cse.ucsc.edu/pub/avg/AMAI/planning-dist.ps.Z`).

[27] 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.

Appendix

A. Planning as SAT: A Review

One way to approach the planning problem is to formulate it in some suitable logic and solved it as a theorem-proving problem. In this view, legal operations are described as axioms, and initial and final conditions are added as further constraints. To deal with changing facts over time, *situation variables* [11] are traditionally introduced. For example, axioms describe changes made to situations by possible actions. Finding a plan thus becomes finding a situation s in which the final conditions hold:

$$\text{axioms} \wedge \text{init} \rightarrow \exists s[\text{final}(s)].$$

This is often achieved by resolution, by proving that the following formula is inconsistent:

$$\text{axioms} \wedge \text{init} \wedge \forall s[\neg\text{final}(s)].$$

A successful plan can then be obtained from the substitutions made in the situation variables.

Alternatively, it is possible to formulate the problem as an implausibility

$$\text{axioms} \wedge \text{init} \rightarrow \forall s[\neg\text{final}(s)].$$

and then try to show that this is not the case by showing that the following formula, after Skolemizing s, is satisfiable:

$$\text{axioms} \wedge \text{init} \wedge \exists s[\text{final}(s)].$$

In this formulation, a model describes a successful plan.

Recently, formulation in propositional clausal logic has been reported with much success [14,7]. This allows a planning problem to be solved as a SAT problem. The formulation is based on the second formulation. Since no quantification symbol nor function symbol are allowed in propositional logic, this requires few changes to the formulation. First, the quantification must be constrained over a finite domain. This is to allow the subformulas that are in the scope of universal quantifiers to be "grounded" for each domain value. Second, we rewrite axioms to describe what relations hold on or between *discrete time instances*. This change is necessary because we are no longer able to construct arbitrary situations (because of the lack of function symbols). This also requires actions to be represented by propositions (as opposed to functions). Since the domain must be finite, we must also set a *deadline*. Refinements to the original propositional formulation [13] can be found elsewhere [14,7].

If the desire is to seek an optimal-length plan, it is necessary to determine the satisfiability of at least two formulas. That is, to show that L is the optimal plan length, it is necessary to show that the formula with deadline L is satisfiable and that the formula with deadline $(L-1)$ is unsatisfiable. In practice, several formulas with different deadlines must be generated before an optimal solution is found.

Once a formula is found satisfiable, and a satisfying truth assignment obtained, a successful plan could be obtained by interpreting the assignments made to the variables.

In this paper, we are interested in the performance of SAT testers on formulas generated from planning problems. Thus, we only generate formulas for the optimal plan length, and one less than the optimal plan length.

B. Goal-Sensitive Simplification

After a formula is generated, the formula is usually subjected to a *simplifier*. A simplifier tries to make the formula as small as possible using various quick and easy simplification techniques while preserving satisfiability. Examples of

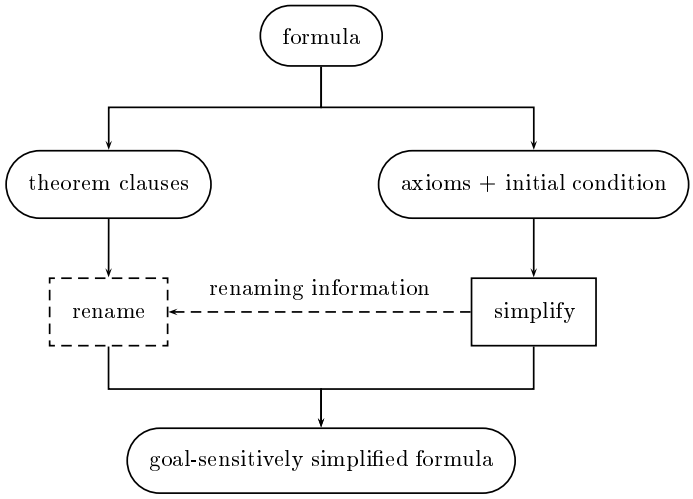

Figure 5. Steps involved in goal-sensitive simplification. For details, see Appendix B.

simplification techniques include unit-implication/unit-propagation, pure-literal elimination, equivalent-literal substitution, and subsumption.

When solving SAT using a backward-chaining theorem prover, it is strongly recommended that additional attributes be preserved across simplification. First, the simplified formula should be logically equivalent to the original formula. Second, all goal clauses should be retained across simplification. Among the simplification techniques listed earlier, pure-literal elimination is not guaranteed to produce a logically equivalent formula, and unit implication may eliminate goal clauses, which tend to be unit clauses in practice. A quick solution would be to not use these techniques at all. While giving up pure-literal elimination is not considered a huge loss, giving up unit implication generally is.

To overcome these problems, Van Gelder and Okushi [26] came up with a simplification scheme that will guarantee all three requirements—preservation of satisfiability, models, and goal clauses. The process is called *goal-sensitive* simplification. (To make the distinction clear, we call the traditional simplification *goal*-in*sensitive* simplification.)

Figure 5 outlines the steps involved in goal-sensitive simplification. The clauses in the input formula are partitioned into two sets—one that consists of the goal clauses and another that consists of the clauses that express 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 literals that occurred during simplification is to be recorded.

Simplification techniques such as equivalent-literal substitution may rename literals. If such techniques are used, we need to make sure that the literals in the first set (the goal 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 formulas that are goal-sensitively simplified are generally slightly larger than formulas that are goal-insensitively simplified.