Propositional Theorem Proving by Semantic Tree Trimming for Hardware Verification

by

William John Yakowenko

A Dissertation submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

1999

Approved by:

David Plaisted, Advisor

Siddhartha Chatterjee, Reader

David Stotts, Reader

WILLIAM JOHN YAKOWENKO. Propositional Theorem Proving by Semantic Tree Trimming for Hardware Verification. (Under the direction of David Plaisted.)

ABSTRACT

The present work describes a new algorithm for testing the satisfiability of statements in propositional logic. It was designed to efficiently handle the most obvious kinds of pathological cases for the Davis-Putnam algorithm. Its performance is compared with a very efficient implementation of Davis-Putnam on a large number of problems, and it is shown to be superior. A recently-developed version of Davis-Putnam with a related algorithmic enhancement is better still, but it is conjectured that the same enhancement can apply to the present work, with a similar boost in performance.

Acknowledgments

My advisor, David Plaisted, has been a constant source of encouragement, even when progress was scant. His words have more meaning per unit than anyone else I have known.

The remainder of my committee, James Anderson, Siddhartha Chatterjee, Gyula Magó, and David Stotts, were there for me when I needed them, and I wish I had more fully taken advantage of their collective knowledge. Sid Chatterjee is due a special thanks for going through this paper in detail, and for the great improvements that resulted.

Jonathan Marshall, Jan Prins, Janet Jones, and Brian White each rescued me at one time or another, perhaps without even being aware of it. I am indebted to each. Jan Prins also provided me with assistantships for more semesters than some need to complete an entire degree.

Donald Stanat guided me through some rough spots and kept my work from stalling in my advisor's absence. For his thorough critique of my practice defense talk, he is due thanks not only from me, but from the audience at the defense as well.

Paramasivam (P2) and Yunshan Zhu each provided encouragement as needed, and research results to envy. I am sure their present colleagues enjoy the same benefits.

My friends Jon Cohen, Suzy Maska, Carl Mueller, Leonard Ray, and Danforth Olsen helped to make this long journey a worthwhile one. May our paths continue to cross.

Jyoti Chaudhry, my fiancée, has stood uncomplaining at my side through all of this; she must really like me. Thanks and dog biscuits are due to her dog Yogi, for arranging our introduction.

My family's support has been continuous and deeply reassuring. This work is dedicated to them, and especially to my father, Nicholas Yakowenko.

I am sure that there must be names that belong here that were overlooked. You have my apologies for this. You know who you are: thank you.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Before proceeding, I review propositional and first-order logic, and work related to hardware verification.

1.1.1 Logic

Logic is the formalization of correct reasoning. In a formal setting, symbols are used to represent objects, statements about them, and relations between them. For instance, if we are concerned with the weather, we might let the symbol S represent the statement "Today it will snow", and Q represent the statement "Today it will be cold".

If we needed to compare the weather on different days, we could make the values of S and Q depend on another piece of information – the date. We might let the symbol X represent some date, and write $S(X)$ to mean "On date X, it did (or will) snow". In this statement, S is called a *predicate*, because it is an assertion which can be true or false. X is called a *variable*, which means it is a placeholder for some information to be filled in later. In this case, X is called the *argument* of S. The number of arguments required by a predicate is called its *arity*; S's arity is one. Then, the statement "it snows every day" would be written as $\forall X : S(X)$, which is read "for all X, S of X". Defining the value of a variable this way is called *universal quantification*; we are quantifying, or limiting, the values that X represents to be universal, or all possibilities. When a variable is universally quantified, we can make *instances* of the statement, in which that variable is replaced by any constant we like, because the original was stated to be true for all values. If we do this for all variables in the statement, we call the statement *ground*.

We might also want to talk about the weather on days that are somehow related to the days we have already mentioned. For instance, if we have been making statements about the weather on day X, we might well want to mention the weather exactly one week after that. To represent this, we define a *function*, which looks syntactically very much like a predicate, but instead of representing a statement which can be true or false, it represents another value, which is somehow determined from its given argument values. So, we might define the function W of arity one, whose value is its argument's plus one week. Then, the statement "Whenever it is cold, it snows one week later" could be written like this: $\forall X : Q(X) \rightarrow S(W(X))$. The $W(X)$ part is called a *term*. Every constant and every variable is a term, as is every function symbol when given the right number of terms as arguments. Similarly, a predicate given the right number of terms as arguments is called an *atom*.

Larger statements can be made from smaller ones by joining them with logical connectives such as *AND* and *OR*, or negating them with *NOT*. (We can now see why a predicate with the right arguments is called an atom; it cannot be subdivided and still represent a true-or-false value. This contrasts with a statement, which might be made of several atoms glued together by those logical connectives.) In symbolic logic, AND and OR are expressed using the symbols "$\wedge$" and "$\vee$", respectively, while negation is expressed by the symbol "$\neg$". In my work, for the sake of simplicity, I often use the simple minus-sign to represent logical negation. Now, we might make this statement: $\forall X : \neg S(X) \vee Q(X)$, which given our definitions, means "for every date, either it is not snowing or it is cold" – an elliptical way of saying that if it is snowing it must be cold. This is perfectly true, but it depends on the definitions for S and Q. It is easy to give definitions that make the statement become false; for example, swap S's definition with Q's. Such a set of definitions for constants, functions, and predicates is called an *interpretation*. An interpretation that makes a statement true is called a *model* for that statement. We are usually interested how some statement behaves when the interpretation is allowed to vary. If a statement is always true no matter what interpretation we give it, then it is said to be *valid*. Another name for a valid statement is a *tautology*. This is most often used in reference to a particular kind of subformula called a *clause*, to be discussed later. Less ambitiously, if there are one or more interpretations that make the statement true, it is *satisfiable*. Note that satisfiability is with respect to changing the definitions of predicates and functions;

variables are still constrained by their quantifiers. That is, the statement "$\forall X : S(X)$" will be unsatisfiable if S must be false for some values of X – satisfiability is *not* about the predicate S being true for some values of X, but about the entire statement "$\forall \cdots$" being true for some definitions of S and not for others.

Whenever we quantify a symbol, there must be some domain that the values of the symbol could be chosen from. Similarly, functions are defined by the interpretation to have some value based on the values of their arguments. In the examples above in which I made statements about the weather, the domain for the variable X and the function $W()$ was that of dates. But a domain could be much more abstract. One domain which is often useful to consider is that of well-formed ground terms. (Actually, since an interpretation only assigns meanings to constants, function symbols, and predicate symbols, it does not directly assign any meaning to any term of more than one symbol. But it is straightforward to define each of those pieces so that any well-formed ground term will be mapped to itself. That is the intent here.) This is like taking every term at face-value; its meaning is nothing more than the string of symbols that make it up. This set of all well-formed ground terms is called the Herbrand universe [13, p. 52], and any interpretation that maps each of them to itself is called a Herbrand interpretation. Note that there is no restriction on the definitions of predicate symbols to be a Herbrand interpretation; only the definitions of function and constant symbols are predetermined. Of course, every interpretation must include a definition for every predicate symbol, the point is that being a Herbrand interpretation does not preclude any such set of definitions. Finally, the set of all well-formed atoms (i.e., predicates given the right number of well-formed ground terms as arguments) is called the Herbrand base. As a technicality, if there are no constants in the statement, then a single constant will be made up just for the purpose of having a non-empty Herbrand base.

In some sense, a Herbrand interpretation is the most general, in that it makes the fewest assumptions about its symbols. Other interpretations will typically map more than one formula to each element of the domain. For instance, if our domain is the integers, and the function $S(X)$ represents $X + 1$, then there are many ways to represent any integer. In that interpretation, 5, $S(4)$, $S(S(3))$, and $S(S(S(2)))$ all represent the same thing. But the Herbrand interpretation for those four formulas is that they are four distinct entities. This is useful because in testing a statement for validity or satisfiability, we do not want the answer to depend on some particular interpretation; indeed, what we need to know is how the formula behaves under different interpretations. In any other interpretation, in which the symbols are given some deeper meaning, we would be constrained as to what truth values we are allowed to assign to related atoms. If we had defined the function $S()$ on integers as before, then there could be no interpretation which had $P(5)$ being true and $P(S(4))$ being false — those two atoms would represent the same thing, and could have only one truth-value. But a Herbrand interpretation could have this, or any other, mapping of atoms to truth values, including those allowed by any other interpretation.

We might consider making a decision tree showing all possible assignments of truth values to the elements of the Herbrand base. This is called a *semantic tree*. Of course, for any non-trivial statement, the Herbrand base is infinite. The presence of any function symbol implies the existence of an infinite number of terms. (Assuming that there is no such thing as a function symbol that takes zero arguments — that would be called a constant.) For example, if our statement includes a function symbol $F()$ that takes a single argument, then the Herbrand base would include c, $F(c)$, $F(F(c))$, and so on. However, even though it is potentially infinite, a semantic tree is still useful as an abstract tool. When I write of a *path* through a semantic tree, I will always mean a path starting at the root node, not repeating any nodes, and going from one node to another only if there is an edge connecting them. I also assume that all edges are directed away from the root node, so I can write of the edges *from* a node, meaning the edges connecting the node to its subtrees. Each edge in a semantic tree specifies truth assignments for some atoms. No path in a semantic tree can contain conflicting truth assignments, and the disjunction of truth assignments on the edges from any node must form a tautology. The most common form of a semantic tree is a binary tree, in which a single literal is assigned different truth values (true and false) on the two edges from any node, and every path has the same ordering of literals.

The truth assignments along some path can serve as the definition of predicate symbols, and therefore, implicitly, a Herbrand interpretation. In general, given a statement, some interpretations may make it true while others make it false, or, respectively, some interpretations satisfy the statement while others contradict it. For those interpretations that contradict the statement, we might not need to look at the entire interpretation to see that it is so. For instance, the statement $A \wedge F(B)$ will be contradicted by any interpretation that assigns A to be false. There are an infinite number of other atoms specified by the interpretation, so the semantic tree will be infinitely deep. But we need not look past the depth at which A and $F(B)$ are assigned truth values to see if the interpretation contradicts our statement. In short, we can

not	$\neg$	
and	$\wedge$	
or	$\vee$	$A \vee B \equiv \neg(\neg A \wedge \neg B)$
implication	$\rightarrow$	$A \rightarrow B \equiv \neg A \vee B$
equal	$\leftrightarrow$	$A \leftrightarrow B \equiv (A \wedge B) \vee (\neg A \wedge \neg B)$

Figure 1.1: Logical connectives.

often work with a *partial* interpretation. For a ground statement, we can always decide this as soon as we know the truth assignments to the finite set of atoms used in the statement. And, as demonstrated above, we might sometimes be able to recognize contradicting interpretations having seen just truth values for just some of the atoms from the statement. On each path representing a contradicting interpretation, we can find one particular node at which we first know that the interpretation contradicts the statement. Such a node is called a *failure* node, and all of the paths that contain it are called *closed* at that node. We may also sometimes refer to a subtree as closed if all of the paths through it are closed. Figure 3.1b on page 30 shows a semantic tree, with failure nodes marked by "X"s. (The statement that caused these nodes to be closed is shown in Figure 3.1a, but it is in a form that has not been explained yet.)

Propositional Logic

In the simplest form of logic, there are no variables, no quantification, and no functions. Only predicates of arity zero, called *propositions*, and logical connectives are left. (Technically, functions and predicates with greater than zero arity may be allowed, but in the absence of quantification these extensions do not bring any expressive power, and so may be considered absent as well.) There are five commonly used logical connectives, though the last three could be replaced by combinations of the first two, as shown in Figure 1.1.

The language of satisfiable propositional formulas is recursive, or decidable, though NP-complete. Figure 1.2 shows two propositional formulas, of which the first is satisfiable (and a model for it is shown), and the second is unsatisfiable – there is no assignment of truth values to its variables that makes the entire statement true.

Satisfiable: $(A \vee B) \wedge (-B \vee -C) \wedge (-A \vee -C)$
in clause form: $[\ A,\ B\ ]\ [\ -B,\ -C\ ]\ [\ -A,\ -C\ ]$
(one model for this is $A\ -C$)

Unsatisfiable: $(A \vee B) \wedge (-B \vee -C) \wedge (-A \vee -C) \wedge C$
in clause form: $[\ A,\ B\ ]\ [\ -B,\ -C\ ]\ [\ -A,\ -C\ ]\ [\ C\ ]$
(this has no model)

Figure 1.2: Sample formulas.

First-order Logic

As mentioned earlier, quantification gives meaning to the variables in a formula, by either saying that the formula is true for all possible values (universal quantification), or that there is at least one such value (existential quantification). If no quantification is allowed, only individual objects can be talked about; there is no other mechanism for talking about "all objects having such and such characteristic". If we allow quantification over these individual objects, we have a way to define and talk about such groups, by making predicates to recognize such objects. For instance, we can assert some statement Q about all even numbers by saying "$\forall X : even(X) \rightarrow Q(X)$". This level of quantification is called *first-order*, because the things quantified over are the simplest possible: individual objects. We could take more steps, allowing quantification over sets of first-order objects to get second-order objects, and so on. First-order logic has the same connectives as propositional logic, but also allows quantification; so variables, functions, and predicates with arity greater than zero become meaningful. The language of logical consequences in first-order logic is recursively enumerable, or semi-decidable; there exist procedures that will tell that a given formula is a logical consequence of the given axioms, but any such procedure will fail to halt for some formula that is not a logical consequence.

Normal Forms

To make the algorithms simpler, formulas are often translated into a form that is in some sense equivalent to the original, but simpler. For instance, rather than using all five of the logical connectives that we defined earlier, we might decide only to use the first two, and re-write our formulas using the rules in Figure 1.1 to accomplish this.

Conjunctive Normal Form, or CNF, is one such form, in which only the connectives AND, OR, and NOT are allowed. Furthermore, subexpressions involving ANDs or ORs cannot be negated, only individual predicates can; and conjunctions cannot be OR'ed together. More loosely stated, the NOTs have to be pushed all the way inside, and the ANDs have to be pulled all the way outside. This is an example of a propositional expression in CNF:

$$(A \vee B \vee \neg C) \wedge E \wedge (C \vee \neg D) \wedge \neg F$$

The entire formula is the *conjunction* of four *conjuncts*, each of which is a *disjunction*. (Both E and $\neg F$ can be considered to be degenerate cases of disjunction, having only one *disjunct* each). It can be seen that using fewer distinct connectives, a program to deal with this form can be simpler than one that had to worry about more; at the very least, the code for dealing with those superfluous connectives can be omitted.

As a further simplification step, we would like to get rid of quantifiers. One well-known procedure for eliminating existential quantifiers is called Skolemization [24, p. 188], after T. Skolem, who invented it. Essentially, this process replaces each existentially quantified variable with a constant name, which is just a name for whatever element in the domain satisfies the stated conditions. We need not know exactly which element it is, but since we are asserting that there is one, it is perfectly reasonable to assign it a name.

The only tricky business here is that the value of that constant may depend on values of other variables that were bound by previous quantifiers. For instance, consider the statement "Every person has a birthday". Symbolically, we might write this as $\forall P : \exists D : B(P, D)$, where B is a predicate that is true when its first argument is a person, its second is a date, and that date is the person's birthday. But we do not get quite the same meaning if we simply replace D with some constant, say K, in the formula: $\forall P : B(P, K)$ means something much stronger (and false); that everyone has the *same* birthday K. We need to somehow state that the value of D depends on that of P, and for each value of P the value of D is constant. For this we introduce a *Skolem function*: $\forall P : B(P, K(P))$ says that every person P has a birthday $K(P)$, but that value of $K(P)$ is really a function of which person we choose; it might well be that no single value for K will do. Note that not only do we not have to know exactly what value this Skolem function represents; we do not even have to know how to compute it. All we are doing is assigning a name to whatever element of the domain was being talked about in that earlier existential quantifier. The fact that we are making K into a function guarantees that it will represent exactly one value for any given P and any given interpretation. Skolemization changes the meaning slightly by introducing those new symbols, but preserves satisfiability. So satisfiability proofs of Skolemized formulas are valid for the un-Skolemized versions as well.

Finally, the form used by many automatic theorem-provers is called *clause form*. It is essentially the same as Skolemized CNF, in which each conjunct is called a *clause*, each clause is a disjunction of *literals*, and in which universal quantifiers are assumed for all variables but never explicitly stated. Since it is understood that the formula is the *conjunction* of the clauses, and that each clause is the *disjunction* of its literals, these connectives are customarily not written. Instead, each clause is given as a set of literals. So, our earlier statement about it always snowing one week after being cold would be written like this:

$$[\neg Q(X), S(W(X))]$$

It is important to remember the definition of satisfiability when using this notation. Even without the universal quantifiers being spelled out, they are implied, and satisfiability is still about whether or not an interpretation exists that makes the entire statement true, universal quantifier and all. It is not about its truth value for different values of its arguments in any single interpretation. So the clause set $[\neg P(a)], [P(b)], [P(X)]$ is not satisfiable, even though $P(X)$ is true for some value of X (namely, b). It is not satisfiable because the statement means "for all values of X $\ldots$", and we know at least one such value of X for which it is not true (namely, a), no matter what meaning gets assigned to P by any interpretation.

Since each clause is a disjunction, and the entire formula is the conjunction of the clauses, in order for the formula to be true, there must be at least one literal in each clause that is true. Another way of reading this is that each clause is an implication; if all the literals save one are false, then that one must be the true one. So, to write $A \to B$, the clause would be $[\neg A, B]$; to equate the truth value of A and B, we need two clauses:

$$[A, \neg B]$$

$$[\neg A, B]$$

Similarly, to assert that A and B have opposite truth values, we write:

$$[A, B]$$

$$[\neg A, \neg B]$$

This is most easily read as "one of them must be true, and one of them must be false".

In clause form, "false" is represented by the empty clause. While this may seem odd at first, it is compatible with the operation of all the inference rules that make sense on non-empty clauses. Intuitively, given a clause in which some literals are false, we know that we can delete those false literals without changing the clause's truth value. If *all* of its literals were false, then the clause is false, and the clause with those false literals removed must have the same value. So the empty clause is false. Using similar logic, one can see why it makes sense that empty sets of clauses are treated as true.

A consequence of this is that unsatisfiability is a *local* property; if it is found in any subset of a set of clauses, then the entire clause set is unsatisfiable as well. Thus a single empty clause makes the entire clause set unsatisfiable, and adding new clauses onto an unsatisfiable set cannot produce a satisfiable one. In contrast, satisfiability is a *global* property; one cannot be certain that a clause set is satisfiable without examining it in its entirety. Satisfiability of a subset does not imply satisfiability of the whole, and one can add clauses onto a satisfiable set to make an unsatisfiable one.

1.1.2 Formal Verification

Verification is the process of proving that some design "lives up to its advertising", that its *implementation* satisfies its *specification*. Planning to have a computer do this implies that we have both the implementation and specification in machine-usable form.[1] This is a non-trivial assumption, but one that seems unavoidable if we want to have these proofs done.

Verification contrasts with *validation*, which is just testing. Since the number of possible inputs to any interesting circuit is very large, testing all of them is out of the question. The literature contains many examples to show this, typically having more

[1] Machine-usable is stronger than machine-readable; I have seen digitized drawings at a site that was claiming to store verification benchmark examples!

processors than atoms in the universe, each covering a huge number of test cases per second since the dawn of time, which would now be finished with some tiny fraction of the total problem. But all this does not make the problem go away; we still want to know that our circuit really works as specified. By verifying the design, we are sure that it is right, without having to spend a fair fraction of eternity (or every atom in the universe!) running test cases. Verification can be faster than validation primarily because it makes use of symbolic techniques to cover many cases with a few logical inferences. Although there are true statements that are not provable in first-order logic (cf. Göedel's incompleteness theorem [33, p. 179]), it is wildly unlikely that a hardware designer will rely on these when defining a circuit, and it is not unreasonable to declare such designs out-of-bounds. Therefore, although verification can be slow, it cannot be effectively worse than exhaustive testing for non-trivial circuits, and in principle can handle anything not designed by insane mathematicians.

Propositional logic is nearly ideal for representing the design of a digital circuit. Each proposition, like each wire in the circuit, must have one of two values; the value of each wire, like the value of each proposition, is determined by simple logical combinations of the values of others; and both are customarily organized as lists of these relationships, whether gates, transistors, or clauses. On the negative side, propositional logic cannot handle time-varying signals. Therefore, propositional logic is good for combinational parts of circuits, and something more is needed to handle sequential ones. One possibility for that "something more" is called *universal generalization*. This is the process of substituting a new constant name, about which nothing has been asserted, as a substitute for some universally quantified variable. For instance, if we have statements about points in the circuit at different times, we might substitute "t" for the time variable, producing ground statements suitable for use in a propositional prover.

Specifications, on the other hand, are almost never propositional, except perhaps for the simplest of circuits. This is because the specification is thought of as being an *abstraction* of the design, at a higher level, closer to the original idea though more formally stated. First order logic may or may not be the language of choice for expressing such abstract ideas; many may be more comfortable with state-machines, higher-order logic, temporal logic, or any of a great number of alternatives. However, first order logic has the distinct advantage of being semi-decidable for logical consequence.

Whatever form they take, the specification and implementation will be handed over to a prover, which might do any amount of preprocessing and re-organization, but will eventually indicate whether or not the entire statement is satisfiable, and if so, produces a model that demonstrates it. Given a refutation style theorem prover, there is very little additional input needed; a few axioms stating the equivalence of inputs to the specification and implementation, and the negation of the conclusion, which is that the outputs differ. This by itself is a naive solution to the problem at hand. Unfortunately, these proofs typically take much too long, so there is work to be done in breaking the problem into smaller ones that can be handled more easily.

Breaking apart the implementation is called *partitioning*, meaning to separate it into distinct pieces, with no overlap and nothing left out. This is easy, because each clause represents a component that will not be subdivided, so any circuit partitioning corresponds neatly to a partitioning of the clauses representing it.

The corresponding operation on the specification is more complex, because the specification may not correspond neatly to the structure of the circuit. We may want to break the circuit apart in places that the specification does not even mention. This process has to untangle the functions computed by parts of the specification, separate them, and connect the separated parts in a way that preserves the original meaning. Since this seems akin to the reverse of putting the specification together by composing parts, it is called *decomposition*.

1.2 Related Work

In this section I give a very brief description of some related work. The brevity of the presentation belies the quantity of work, but more would be superfluous to the purpose at hand.

1.2.1 Hardware Verification

The most oft-cited prover for hardware verification is HOL (Higher Order Logic) [26], based on LCF-LSM (the Logic of Computable Functions) [27]. More recently, Isabelle [44] has subsumed HOL, along with several other logics. Though perhaps not designed with hardware verification in mind, both Boyer and Moore's Nqthm [6] and Constable's Nuprl [18] fit the same mold, and in fact both have made forays into the field. Though certainly useful, these are not interesting from the perspective of automatic theorem-proving because they are human-guided, taking instructions from the user on how to complete the proof. This is made significantly less painful by the use of tactics and tacticals, which are essentially macro-instructions for the prover. But this still falls short of being automatic, and these "provers" could still be viewed as bookkeeping systems for the users who do the proofs. Perhaps these systems stop short of complete automation because their internal representations are boolean expressions, and it is very difficult to build a fully automatic prover that operates on arbitrary expressions. It should be noted though, that these systems are extensible, and so allow for the possibility of defining automated inference rules, and thereby subsuming the techniques I plan to use. This, however, merely reflects positively on the foresight of the authors of those systems, rather than detracting in any way from further research in automatic theorem proving.

The other main approach to verification uses temporal logic to check the behavior of a circuit over time, rather than checking the relation between its inputs and outputs. The favored approach is called Model Checking, and proves statements in a carefully limited form of temporal logic. Though interesting and promising, there are major hurdles to be overcome. Perhaps the highest is the state explosion problem; even a four-kilobit memory has on the order of 10^{1300} states to be modeled, which is approximately the current state of the art [15]. Most provers that follow this approach use Binary Decision Diagrams as a way to symbolically represent sets of states.

1.2.2 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are a canonical representation for boolean functions [1]. Since the BDD for any function is unique, functional comparisons can be done by constructing and comparing their BDDs. In 1986, Bryant produced a set of algorithms for efficiently manipulating BDDs [9]. This led to their widespread use, first in hardware simulators, and later in VLSI CAD systems and Model Checking symbolic verifiers [10, 41]. Though this was a tremendous breakthrough, BDDs are not a panacea; for some not-terribly-rare circuits, such as multipliers, the size of a BDD will be exponential in the size of the circuit. (But see [11].) Moreover, though more efficient than the ad-hoc techniques that they replaced, BDDs are still not as fast as the Davis-Putnam algorithm on some classes of problems, including some of the smaller problems in the IFIP Boolean Equivalence Benchmark suite. For larger problems, BDDs gain an advantage in being more symbolic, and therefore handling more cases per operation, than the Davis-Putnam algorithm to be discussed later. But it seems few formal comparisons have been done between the two [51].

1.2.3 Partitioning and Decomposition

Many existing partitioning algorithms were intended to assist in VLSI placement and routing, and so do not pay attention to the same issues that we want for theorem-proving.

There has been a great deal of research into "spectral" techniques for circuit partitioning [31]. These use eigenvectors of a circuit's Laplacian (which is closely related to its adjacency matrix) as coordinates for each node in some Euclidean space. Clustering in Euclidean space is more-or-less straightforward and efficient, because computing the distance between any two nodes is. But the purpose of this is to find the partitioning that is most natural in terms of balanced module sizes and minimum connections between them, and not necessarily one that corresponds to any other partitioning in any particular way.

By minimizing the amount of communication necessary between the modules of a partitioned circuit, Communication-Based Partitioning [4] seeks to increase chip speeds. This is because on-chip communications are much faster than communication between chips. So, from the perspective of speed, an imbalance in chip sizes is irrelevant, while a minimum of off-chip wiring is critical. This might be closer in spirit to the kind of partitioning criteria we need, because module definitions should be simpler if their interface is smaller.

1.2.4 Summary

In this chapter, I have reviewed propositional and first-order logic, and work related to hardware verification. With this background, I will next introduce automated theorem proving.

Chapter 2

Automated Theorem Proving

An automatic theorem prover takes as input some axioms and an alleged theorem, and returns a boolean telling if the theorem follows from the axioms or not. In this respect, a theorem prover can be viewed as recognizing a language, in the same sense as a Turing machine in automata theory.

The process of deriving new true statements from old ones is called *inference*. Most provers have a relatively small number of inference rules, and work by repeatedly choosing which rule to use, and which statements to use them on.

Depending on what we allow as input, the prover may be capable of achieving this goal for all inputs. For instance, since the propositional logic is decidable, a propositional prover could conceivably halt with the correct answer for any input. On the other hand, first-order provers cannot. Typically they will return true if the alleged theorem really was one; but if it was not, they might either return false or just fail to halt. The property of always finding an answer if an answer exists is called *completeness*.

Complementary to completeness, we are also concerned with the *soundness* of a prover; that it never proves anything that is false. Since provers repeatedly apply a small number of inference rules, we can insure soundness just by making sure that each of those inference rules is sound. If no single step ever produces a false result, then chaining together any number of such steps will still produce only true results. Of course, any interesting theorem is sufficiently distant from its axioms that a great number of steps are needed, and there are many choices to make at each step. The completeness of a method does not guarantee it to be efficient. If those choices were made randomly, we might expect to prove a great many interesting theorems, which would all be completely irrelevant to the proof at hand, before finally stumbling onto the result we want. This is characteristic of algorithms that work forward from axioms, looking at the theorem-to-be only to compare it with results achieved so far. This approach is called *forward chaining*.

There is an alternative, of course. More sophisticated algorithms look at the goal, and try to work backwards from it to the axioms. Since everything generated this way is related to the goal, there tends to be less wasted effort, and the proof is generally found faster. This is called *backward chaining*.

Many theorem provers, including the ones developed by David Plaisted here at UNC, work by *refutation*. Rather than deriving the theorem from axioms, these derive a contradiction from the conjunction of the axioms and the negation of the theorem. The refutation can be in the form of a derivation of a statement that is known to be false. Since the inference rules were all sound, the derivation of "false" demonstrates that the input statement must have been false. But that input statement was just the conjunction of the axioms and the negated theorem. Since we are assuming the axioms to be true, that can only mean that the negated theorem was false, and therefore that the original theorem was true.

One motivation for using the refutational style of proof in formal verification is that it is more helpful when the circuit fails to meet its specification. While a non-refutational proof of this would simply indicate that the statement was unsatisfiable, the refutational prover will find that it is satisfiable, and the model will show exactly how.

2.1 Subsumption

Since provers repeatedly generate new statements from old ones, it is often beneficial for them to do some simplifications, to prevent the sheer number of statements from becoming a hindrance. Perhaps the simplest of these is based on *subsumption*. We say one statement A subsumes another B, if B is a logical consequence of A. In practice, we are often concerned merely with catching the obvious subsumptions. When statements are just clauses, this amounts to A being a subset of B, except that some variables in A might have to be given more specific values to make them match those in B. For instance, $[W, X, Y]$ subsumes $[W, X, Y, Z]$, because any time the first is true, the second must also be true. To see the necessity for filling in of variables, consider these two clauses: [1] $[f(V), G]$, and $[f(s(Z)), G, H]$. The first is a subset of the second only if we let the variable V have the value $s(Z)$. Still, this only means that the first clause is that much more general than the second, that a special case of the first subsumes the second.

2.2 Unification

Since propositional logic is not always enough, we must look at the machinery that allows first-order provers to work. Most often, the interesting part of finding a proof in first-order clause form involves logically connecting clauses that do not exactly say anything in common. For instance, given the clauses:

$$[p(X, i(X), e)]$$

$$[q(Z, R), \neg p(f(Y), i(Z), R)]$$

we would like the prover to find the connection, and produce this:

$$[q(f(Y), e)]$$

That second clause can be read as an implication:

$$p(f(Y), i(Z), R)) \rightarrow q(Z, R)$$

The process that enables this is called *unification*, and is closely related to the filling-in of variables we did to check for subsumption. Unification matches two terms by finding an assignment of values to variables (a *substitution*) that makes them match identically. It does this by starting at the beginning of two terms, and advancing through them as long as they either match identically, or at the places that they do not match one of them is a variable which has not yet been assigned a value. So, in that example, $p(X, i(X), e)$ unified with $p(f(Y), i(Z), R)$ by first seeing that "p" is identical to "p", then giving the value "$f(Y)$" to X, then seeing that the "i"s are

[1] From this point on, when expressing anything more complicated than propositional logic, I will follow the convention of the Prolog programming language: variable names begin with upper-case letters, and all other names begin with lower-case. However, I will continue to write propositional statements as single upper-case letters. Since a variable can never appear in the position of a predicate, it should be clear from context which is which. Until this point, I used upper-case letters for everything to highlight the similarities between propositional and first-order logic.

identical, then giving the value of X (which is now "$f(Y)$") to Z, and finally giving the value "e" to R. This substitution is called the *most general unifier*, or MGU. We could have been even more specific, by assigning a value to Y as well. But that would have been excessively specific; we do not need to do that to unify these terms. Unification thus allows us to connect clauses that are not identical, but have some overlap in meaning because of the flexibility of having universally quantified variables.

(I have intentionally glossed over the issue of *variable capture* here, as it is essentially a mechanical nuisance of no profound value. Just note that variables in different clauses represent different things, even if they sometimes happen to have the same name. So to avoid problems, whenever bringing together information from different clauses, first rename variables as needed to insure that distinct variables remain so.)

2.3 Resolution

In 1965, Robinson proposed his *resolution* rule for automated theorem proving [46]. Somewhat simplified, it take two clauses C_1 and C_2, where C_1 contains some literal L_1, and C_2 contains its complement $L_2 = \neg L_1$. From these, the resolution rule produces their resolvent, which is a clause containing all the literals from C_1 but L_1, and all the literals from C_2 but L_2. This is sound, because either L_1 or L_2 must be false, so either the remaining literals in C_1 must contain one that is true, or the remaining literals in C_2 must; all of those are in the resolvent. As stated, resolution would be pretty weak, only operating when a literal in one clause and its exact negation in another clause were found. But the use of unification (as resolution is actually defined) grants resolution some real power — it is complete for first-order logic. Resolution was important because it was simple, sound, and complete — just the thing everyone had been looking for. Unfortunately, it is not always as efficient as had been initially hoped, and numerous variants have been proposed to overcome difficulties. Still, some very successful theorem provers have been written using resolution as the primary inference rule, and even theorem provers that largely rely on other techniques also use resolution to some degree, to speed some proofs.

2.4 Hyperlinking

In contrast to resolution, which puts clauses together (sometimes too often) into longer and longer results, most of the provers developed here at UNC are *instance*

clause number	clause	origin
1	$[\ A,\ B\ ]$	input
2	$[\ -B,\ -C\ ]$	input
3	$[\ -A,\ -C\ ]$	input
4	$[\ C\ ]$	input
5	$[\ -B\ ]$	resolved clauses 2 and 4 on C
6	$[\ A\ ]$	resolved clauses 1 and 5 on B
7	$[\ -A\ ]$	resolved clauses 3 and 4 on C
8	$[\,]$	resolved clauses 6 and 7 on A

Figure 2.1: Resolution on an unsatisfiable problem

based, which means they generate ground instances of clauses, and then check those for satisfiability using a propositional prover. By Herbrand's theorem, a set of clauses is unsatisfiable if and only if it has a finite unsatisfiable set of ground instances. The reason for doing this is that propositional proofs tend to be relatively fast, in which case the interesting work is in seeing how to connect statements together to make the proof. It would be easy to generate useless instances of clauses; just make up constant names, and replace variables with them. But this is not likely to generate an unsatisfiable ground set, because the instances are not likely to connect to each other. That is, the ground clauses will be statements about unrelated symbols, so it will be easy to find an interpretation that makes them all true. For example, if we have the clauses $[p(X)]$, and $[\neg p(b)]$, we could let X have the value a. The resulting two clauses are now $[p(a)]$, and $[\neg p(b)]$, and it is easy to see that these are satisfiable; it is no challenge to make up a model in which these are both true. In the same way that subsumption and resolution only work when there is something matching between clauses, ground proofs need that as well. Therefore, the values substituted-in are taken from terms in other clauses that could unify with the one in question. That way, we generate a set of ground clauses that talk about the same symbols, and might more easily say something impossible. Each pair of complementary literals is a *link*, and the set of links covering all literals in a clause is a *hyperlink*. For instance, in the preceding example, the terms $p(X)$ and $\neg p(b)$ can unify, so there is a good chance (especially in this contrived example!) that letting X take the value b will produce a ground clause that is relevant to the proof.

Hyperlinking is this method of generating useful ground instances. In more advanced provers, the mathematical meaning (or *semantics*) of the symbols is used to help decide which instances to generate and which not to bother with, in order to zero in on the contradiction and complete the proof.

2.5 Human Guidance

Finding a proof is not always a straightforward task, and different representations lend themselves more or less well to different approaches. Humans would rarely be tempted to use a procedure like resolution or hyperlinking, perhaps because clause form itself seems unwieldy to us. Many attempts have been made to build provers that work with a data representation more comfortable to humans; the so-called *natural deduction* style. Unfortunately, this lends itself well to the way humans think, and not so well to the way computers operate. Having more logical connectives, quantifiers, and inference rules is of great benefit to humans doing proofs, because we somehow internalize all of it, and can generalize and extrapolate from the experiences of our entire lives, producing insights that cannot be easily attributed even by the person experiencing them. Mechanizing this has proved to be difficult. Perhaps this is why many provers have been developed that use this style of notation, but require a human to guide them through lemmas, until at last a proof is found. This is not to disparage that line of work at all — it both is useful and interesting — but the task at hand here is to automate the proof process as much as possible. Therefore it seems prudent to stick with hyperlinking, resolution, Davis-Putnam, and similar techniques as much as possible.

It may be pointed out that even these "automatic" provers require the setting of some parameters, weighting of strategies, time limits, and so on, and that some circuit partitioning might be best done by humans who have a feel for logical places to separate pieces. While these are indeed human inputs, they are fundamentally different from guiding the prover through the proof. Furthermore, the optimal settings for a problem are often the same as those for other problems in the same class, where classes might be problems that involve equality, set theory, real arithmetic, or so on. We might therefore expect to find a reasonable setting for hardware verification problems, hard-wire them into the system, and forget about them. And finding "good" partitionings remains an open problem, which might well be solved within the coming years. (Some work on this is a part of this dissertation; I have developed heuristics that seem to consistently deliver results within an order of magnitude of the best that I could do by hand.)

Any first-order theorem prover can work on propositional statements as well, by treating them as special cases of first-order logic in which all predicates have arity zero. But first-order provers must necessarily include facilities for dealing with quantified variables, and so are generally more complex than need be to handle propositional logic. Also, first order proofs have their own unique difficulties, so first order provers tend to focus on those. So their performance on propositional problems is not generally so good.

2.6 The Davis-Putnam Algorithm

We are now ready to consider a well-known propositional logic decision procedure, the Davis-Putnam algorithm (which should be more properly known as the Davis-Putnam-Logemann-Loveland algorithm [20]). Starting from a set of propositional clauses, this algorithm will either produce a model if one exists, or else indicate that the set is unsatisfiable. This algorithm can be seen as repeatedly applying two kinds of rules, after removing tautologies. (Note that tautologies are easy to find in propositional clauses: they are exactly those clauses in which a literal and its negation both appear). The first kind of rules are simplifications. These rules effectively look at the clause set, and deduce the values of some of the literals. With those values known, the set can be simplified, making the rest of the proof faster. The second kind of rule involves trying different truth values for literals until either one works or all fail.

There are two simplification rules: the *unit-clause* rule and the *pure literal* rule. Both of these deduce some values for literals, and then simplify the clause set based on that. The simplifications are based on subsumption: conjunctions containing "true" are subsumed by copies without it, and disjunctions containing "false" are subsumed by copies without it. For example, $A \wedge B \wedge \mathit{true} \wedge C$ is subsumed by $A \wedge B \wedge C$, and $A \vee B \vee \mathit{false} \vee C$ is subsumed by $A \vee B \vee C$. In clause form, this means we can delete clauses that contain true literals (the whole statement is a conjunction of clauses, and that one is known to be true), and we can delete false literals from clauses (the clause is a disjunction, and that literal is known to be false).

Deducing these truth values is simple in two cases. Unit clauses are those which contain only one literal. Since each clause must be true for the whole to be true, and a clause is true exactly when one or more of its literals are true, we can conclude that any literal in a unit clause is true.

A pure literal occurs in the proof with only one sign; never negated. For instance, if $\neg A$ occurs, but A (the negation of $\neg A$) never does, then $\neg A$ is a pure literal. Pure literals can be assumed true, since any model in which it was false would still be a model when it is true, and if no model exists when it is true, then none exists when it is false either.

These two simplifications are applied in any order, until there are no pure literals or unit clauses left.

When none of the simplification rules apply, we make progress by guessing a truth value for some literal, doing whatever simplifications we can, and then applying the algorithm recursively to the new simplified set. If we find a model this way, then it does not matter that it was just a guess; it is still a model and the algorithm is finished. If the set turns out to be unsatisfiable after making that guess, we guess the opposite truth value for that literal, and try again. This is called backtracking. As before, if we find a model this way, we are done. If the set is still unsatisfiable after trying both possibilities, then it is unsatisfiable no matter what value we give to this literal, and we can conclude that the input set was unsatisfiable. Since this rule essentially splits the proof into two cases, it is called the *splitting rule*, or sometimes *branching*.

Note that the proof always terminates due to a simplification. If we produce the empty clause by deleting literals, the input set was unsatisfiable. If we delete the last clause because it contained a true literal, then the set was satisfiable, and the simplification and splitting rules chose values that make a model for the input set. If neither of these two occur, we do not know yet, and have to continue the process. Examples of this are given in Figures 2.2, 2.3, and 2.4.

We can view the Davis-Putnam procedure as the traversal of a semantic tree. The splitting rule handles the exploration of the subtrees of a node, and the simplification rules close off some subtrees so that they need not be explored. This idea will be used later.

Since the satisfiability problem is NP-complete, and Davis-Putnam solves it, we know it needs more than polynomial running time in the worst case. More precisely, we can see that the splitting rule dominates the running time, because each split requires two subproofs, while each simplification requires only one. In the worst case, only one literal will be simplified away after each split, and the first of the two cases will not result in a model. In that situation, running time will be $O(2^N)$, where N

Input clause set: [A, B] [$-B$, $-C$] [$-A$, $-C$]

There are no unit clauses.
$-C$ is a pure literal (C does not occur).
So $-C$ is true (and C is false).

Clause set after simplification: [A, B]

There are no unit clauses.
A and B are both pure literals; arbitrarily choose A.
So A is true (and $-A$ is false).

Clause set after simplification: (empty)

So the original clause set is satisfied by $-C$ A.

Figure 2.2: Davis-Putnam on a satisfiable problem

is the number of distinct literals. This amounts to almost a brute-force enumeration of all possible combinations of truth values for the atoms in the problem. In practice, typical running times for many domains tend to be nearly linear. Hardware verification is unfortunately not one of these domains.

2.7 Lemma Generation

One technique for speeding up proofs, which has recently been implemented to great advantage in a prover called Sato, is lemma generation. This is the saving and re-use of logical information that is discovered as the proof proceeds. In Davis-Putnam, this means generating new clauses under certain circumstances, when nodes are closed. Later in the traversal of the semantic tree, one of those newly-generated clauses might close a node that would otherwise have required a traversal of its subtree, possibly identical to the traversal that resulted in that newly-generated clause in the first place. Of course, as we deal with more and more clauses, the tree traversal itself can be slower, because we may have to search through more clauses to decide whether any particular node is closed. So ideally, we would only generate lemmas that will

Input clause set: [A, B] [$-B$, $-C$] [$-A$, $-C$] [C]

C occurs in a unit clause.
So C is true.

Clause set after simplification: [A, B] [$-B$] [$-A$]

$-A$ and $-B$ occur in unit clauses; arbitrarily choose $-A$.
So $-A$ is true.

Clause set after simplification: [B] [$-B$]

B and $-B$ occur in unit clauses; arbitrarily choose B.
So B is true.

Clause set after simplification: [ ]

Derived the empty clause; this interpretation is contradicted.
There were no assumptions; no backtracking to do.
No models were found.
So the original clause set is unsatisfiable.

Figure 2.3: Davis-Putnam on an unsatisfiable problem

save more work than they will cause. As will be seen later, lemma generation is in some ways similar to the present work.

2.8 Related Work

In a recent paper Biere et al. have taken the first steps toward applying the Davis-Putnam algorithm in place of BDDs in the Model Checking algorithm [5]. If this line of research bears fruit, then Davis-Putnam and related techniques may suddenly become central players in this behavioral style of hardware verification.

The 1960 JACM paper by Martin Davis and Hilary Putnam [21] includes a method which generates increasingly large propositional statements from a first-order statement, and tests these for satisfiability. While the end goal was to test the satisfiability

Input clause set: [A, $-B$, $-C$] [$-A$, B, C] [$-A$, $-B$, $-C$]

There are no unit clauses, and no pure literals.
Arbitrarily choose A for splitting.
Assume A is true.

Clause set after simplification: [B, C] [$-B$, $-C$]

There are no unit clauses, and no pure literals.
Arbitrarily choose B.
Assume B is true.

Clause set after simplification: [C] [$-C$]

C and $-C$ occur in unit clauses; arbitrarily choose C.
C is true.

Clause set after simplification: [ ]

Derived the empty clause.
Un-do the most recent assumption (that B is true).
B is false.

Clause set after simplification: [C]

C occurs in a unit clause.
So C is true.

Clause set after simplification: (empty)

The original clause set is satisfied by A $-B$ C.
No further searching is needed.

Figure 2.4: Davis-Putnam doing backtracking

of the original first-order formula, this necessarily included an algorithm for testing propositional formulas. Their propositional satisfiability algorithm repeatedly chose and eliminated a literal, by re-writing all of the clauses containing it, in a way that preserved satisfiability but no longer included that literal. The main problem with this was that a large number of clauses would be generated in the process of re-writing. As will be seen, there are some similarities between this first method of theirs, and the present work.

Two years later, Martin Davis, George Logemann, and Donald Loveland followed up Davis and Putnam's 1960 paper with a modification to its propositional satisfiability test [20]. This corrected the flaw of the original algorithm that caused it to tend to bury itself in generated clauses, and has since come to be recognized as the best available technique for general propositional theorem proving. Rather than re-writing sets of clauses in order to get rid of a variable, this modified method simply assumes a value for it, simplifies it away, and continues. If the clause set turns out to be unsatisfiable, then backtracking is used to return to this assumption, and the alternate value is then chosen, and again the method continues. In this way, all possibilities are explored. The flaw with this, of course, is that there are some problems that will cause too much backtracking.

A recent propositional prover, Sato, is a very efficient implementation of the Davis-Putnam algorithm [54]. A refinement of Sato, Sato3, which generates lemmas as it goes, has dramatically better performance than the original [55]. This is interesting relative to the current work because it is in some ways similar, and because features of Sttrim and Sato could likely be combined to gain the advantages of each.

Propositional provers are also built into many first-order provers, either explicitly, as in the case of instance-based provers, or implicitly, because propositional logic is a subset of first-order. (First order predicates are usually allowed to have arity zero, in which case they are simply propositions.)

2.9 Summary

An automated theorem prover tests a statement for satisfiability. Refutational provers prove that a conclusion follows from its axioms by negating that conclusion and showing that the result is unsatisfiable. I gave a detailed description of the Davis-Putnam algorithm, which is a propositional automated theorem prover, and explained that it, and related approaches, may soon play an important role in the formal verification of hardware.

Chapter 3

Semantic Tree Trimming Algorithm

In the present work, I present a new propositional calculus decision procedure, called "Semantic Tree Trimming" (Sttrim), which on certain classes of formulas of interest for hardware verification, significantly outperforms the fastest implementation of the Davis-Putnam method.

It was designed to overcome one particular kind of pathological case for Davis-Putnam, by propagating information forward rather than backtracking to try alternatives, and to avoid some of the unnecessary repetitive work that backtracking entails. In this respect, it is quite similar to lemma generation, which also seeks to avoid the repetition of unnecessary work. The mechanics are different, but each generates new clauses to sum up the discovery of some closed portion of the semantic tree. It seems likely that lemma generation could be added to Sttrim, with much the same effect that it has on Davis-Putnam.

Although it was originally intended for use in hardware verification, my algorithm also shows promise as a general-purpose propositional theorem prover.

3.1 High Level Operation

The central idea of Sttrim is to handle a large proof as a series of subproofs, each of which is quite small. This is good because for common algorithms, many classes of problem require time exponential in their size, but the time required to solve many smaller problems can add up to much less than the time that would have been required to solve the original in one step. Finding a subproblem in this case means choosing a subset R of clauses of the original clause set S, which is "simplified" and then merged with $S - R$ to form a new simpler set S', in such a way that S' is satisfiable if and only if S was. By repeating this process, we eventually produce either a clause set containing an empty clause, which is unsatisfiable; or the empty clause set, which is satisfiable. The choice of R and the method of simplifying it are such that the process can be proved to terminate within a finite number of passes. Furthermore, for many problems, each of those steps is small enough to avoid the bulk of the exponential explosion that would have occurred had the entire proof been tried in a single pass. This is shown in algorithm 1, "Test for satisfiability" on page 44.

To choose the clause subset to simplify, called R above, we first choose some small set of variables. All of the clauses that contain chosen variables are included in R. In the clauses of R, there will generally be some non-chosen variables in addition to the chosen ones. These non-chosen variables in chosen clauses are called *extra* variables. The simplification of R involves rewriting R in such a way that none of the chosen variables occur in the result. Since some variables are entirely removed from the clause set with each such pass, and the clause set started out with a finite set of variables, it is clear that the process will terminate with a finite number of passes. This is shown in algorithm 2, "Choose variables and clauses to work on" on page 45.

3.2 Simplification

To understand the simplification of R, it is necessary to consider the semantic tree T on the variables of R. In this semantic tree, all of the chosen variables occur higher than any of the extra variables. That is, chosen variables occur toward the leaves of the semantic tree; extra ones toward the root. Call the upper section of the tree, corresponding to the chosen variables, $L1$, and the lower section of the tree, corresponding to the extra variables, $L2$. Note that it is the edges that are labeled with variables, not the nodes. Call the nodes that are pointed to by the top-most edges in the $L2$ part of the tree "L2-leaves", and call the tree with each path truncated at an L2-leaf (but including that leaf) an "L2-truncation". Sttrim works by trimming off the top levels of the semantic tree, those in $L1$, and closing paths in the new tree as necessary to preserve its satisfiability. The new semantic tree, T', is an L2-truncation of the original, with all the same closed nodes that occurred in the $L2$ part of T, plus possibly some new ones to close paths in $L2$ for which there was no extension open all the way through $L1$ in T. If any path was open in T, all the way through $L2$ and

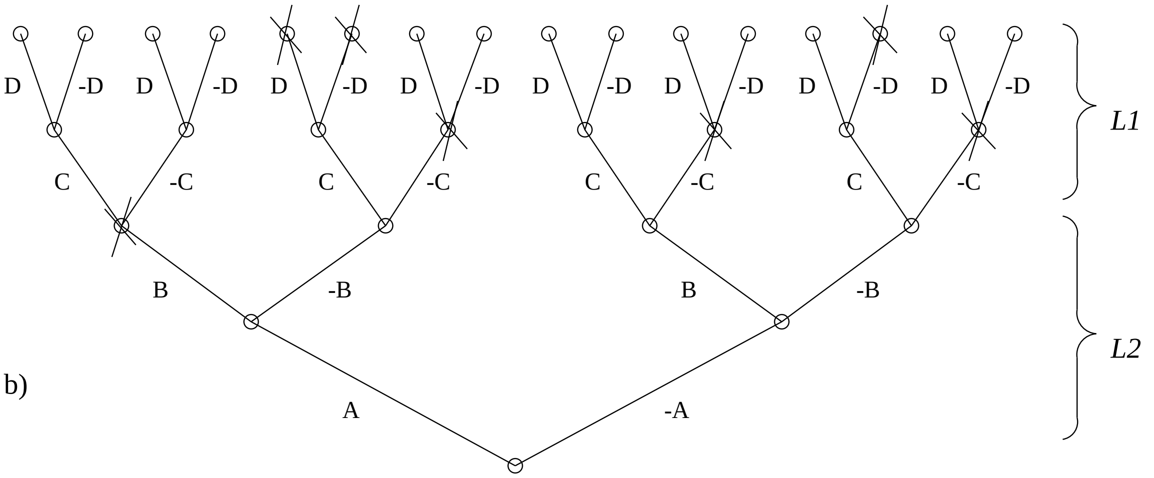
$L1$, then that path through $L2$ will be open in T'. Similarly, all paths that are closed at nodes in $L2$ in T will be closed at the same node in T'. But T' also has paths closed at its leaves for any L2-leaf nodes in T through which all paths were closed in $L1$. That is, if none of the nodes leading up to some L2-leaf node were closed, but all of the paths that include that as a prefix were closed in $L1$, then this node will be closed in T'. The result is that T' has open paths through $L2$ in exactly the places where T has open paths all the way through both $L2$ and $L1$. See Figure 3.1 for a small example.

We can extract from T' a set of clauses R' which exactly contradicts its closed nodes. The clause set $S - R + R'$ is satisfiable exactly when S is. We can see this by noting that every open path through S has a corresponding (but shorter) open path through $S - R + R'$, and that every open path through $S - R + R'$ has at least one, and possibly more than one, open path through S. Therefore, replacing the subset R with R' in S preserves its satisfiability, while reducing the number of distinct variables that occur in S.

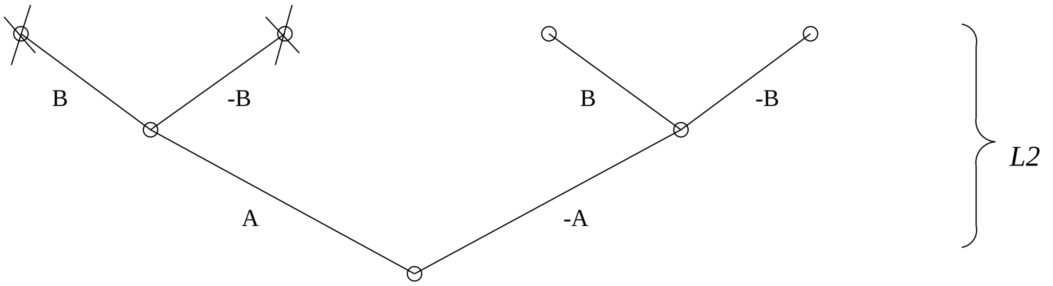
The simplification of R could be done in a number of ways. Any procedure that produces the results described above would do. In particular, if resolution was used, then the entire method becomes a special case of Davis-Putnam's original 1960 algorithm. However, resolution has been found to be less suitable for propositional logic than the later, 1962, Davis-Putnam algorithm, so I adopt a variation of that, which works as follows.

We build and traverse the semantic tree depth-first, reducing the clause set at each step, as in Davis-Putnam. (False literals are deleted, as are true clauses.) As mentioned above, $L2$ variables are all kept closer to the root than any $L1$ variables, so they are assigned truth values first. When a failure node is encountered in $L2$, the clause that contradicted it is output, and we act as if this was a leaf node and continue the traversal by backtracking from it. When a path has no failure nodes in $L2$, we continue the traversal into $L1$, but in $L1$ the results are handled differently. We will refer to the L2-leaf node of that path as P_{L2}. If we find a failure node in $L1$, we do not output the clause that caused it. After all, the goal is to produce clauses that have no $L1$ variables in them, but the clause that caused a failure node in $L1$ must contain an $L1$ literal. Instead, we continue traversing the tree in $L1$, until either finding a path through P_{L2} that continues through $L1$ with no failure nodes, or finding that all paths through P_{L2} are closed in $L1$. If all paths through P_{L2} were closed, then we output a clause that closes the path up to P_{L2}. If there was an open path, then we

a) [-A, -B], [-D, -A], [D, B], [C]



b)



c)

d) [-A, -B], [-A, B]

Figure 3.1: The simplification of a clause set. a) The initial clauses. Literals C and D are chosen for elimination. b) The original semantic tree, showing nodes closed by clauses. c) The new semantic tree. d) The new clause set.

give up searching any more paths through P_{L2}. In either case, we pretend that P_{L2} was a leaf node, and just continue the traversal from there.

Actually, there is a little more to do besides the output of a clause, in those places above where it says to output a clause. One thing is that we do not necessarily output the clause quite yet. Although the algorithm would still be correct if we did this, we can do better. Rather than generating a clause for each closed subtree, we just keep track of which literals were involved in their closure, until such time as we find one subtree is closed and its sibling is not. That closed subtree is *maximal*, and we output a clause that includes all of the literals that were used in closing it. This all means a more complicated algorithm, but helps avoid the tendency of new-clause-producing algorithms to drown themselves in clauses. Figure 3.2 shows this; several nodes are closed, but only one is eventually recognized (at point 7) to be maximal, and a clause is output for it. This is shown in algorithm 3, "Simplify a set of clauses" on page 46.

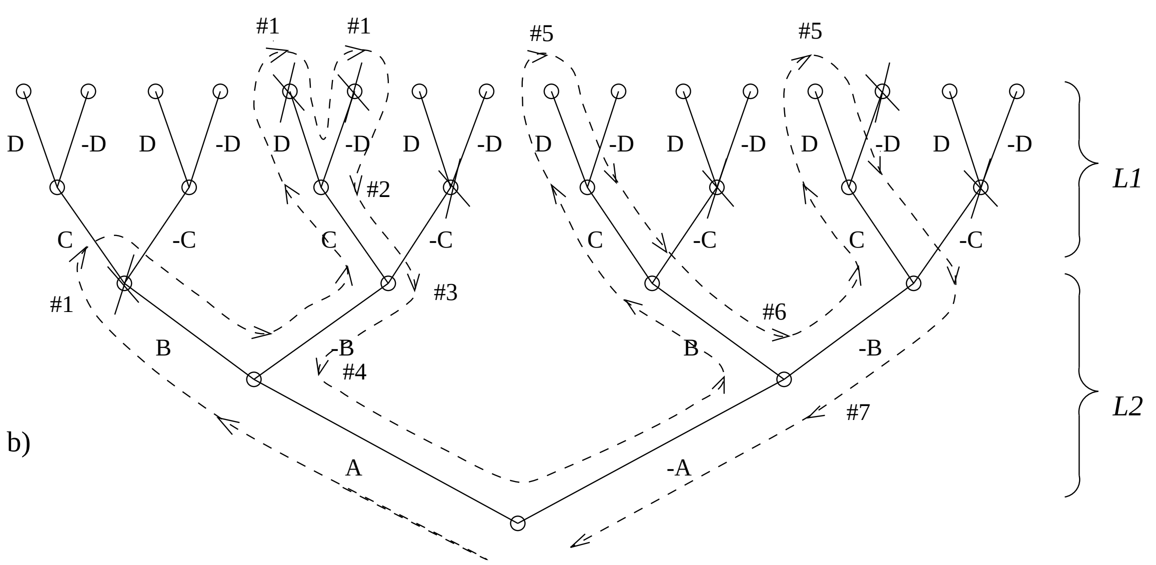
Another benefit of keeping track of the literals used in closing a subtree is that we may sometimes skip over subtrees when we know that they will not produce any new results. For instance, suppose we have the interpretation [A, B, C], and have just found that the corresponding subtree is closed. If none of the clauses that closed nodes in that subtree mentioned C at all, then we need not consider the interpretation [A, B, $-C$], because those same clauses that closed the first subtree would close that subtree as well. In Figure 3.2, this happens at point 3. Its first subtree was closed by clauses $[-D, -A]$ and $[D, B]$, neither of which involve the literal C. So we need not bother assigning C to be false and trying again, because exactly the same clauses will close the subtree again.

It is worth noting that there could have been some clauses involving C that would have helped close the second subtree, which did not help in closing the first subtree. But given that we already know that the second subtree will be closed, any such additional clauses do not matter; closed is closed. However, later in the traversal, we encounter a different situation. After finding an open subtree at point 6, we cannot skip over its sibling subtree. In this case, the additional clauses that we might find in the second subtree could close it (although in this example, they do not). So we can sometimes skip the sibling of a closed subtree, but never the sibling of an open one.

The asymmetry arises because additional clauses only close more paths; they can close a subtree that would have been open without them, but cannot open one that was already closed without them. So knowing that a first subtree is closed, we may be able to infer that its sibling is closed as well, in which case additional clauses cannot

a) [-A, -B], [-D, -A], [D, B], [C]

b)



Scenic points in the traversal

#1 - This interpretation is contradicted by some clause

#2 - Both children are closed, so this node is closed too
(But this is not a maximal closed subtree)

#3 - Skip the second subtree because C was unused in closing the first one
(This subtree is closed, but is not maximal)

#4 - Node is closed, and will later be recognized as maximal

#5 - Open path through L1 - backtrack into L2

#6 - Found one open subtree, must still traverse the second

#7 - Open sibling of a closed subtree
(That closed subtree is now known to be maximal)

c) [-A]

Figure 3.2: The simplification of a clause set, showing optimizations. a) The initial clauses. Literals C and D are chosen for elimination. b) The traversal of the semantic tree, with scenic points labeled c) The new clause set.

prevent it. But knowing that a first subtree is open, we cannot infer that its sibling is open, because additional clauses could make it closed.

The traversal of the semantic tree is therefore actually handled by repeatedly building and testing interpretations. Each interpretation corresponds to one path through the semantic tree. In a departure from the most common form of a semantic tree, this allows the ordering of literals to be different along different paths, as long as no literal is repeated. (But note that this does meet the definition of a semantic tree in [13].) As each literal is selected to be added to the path, the clause set is simplified by the interpretation that the path represents. This can affect the choice of later literals on the path. For instance, it is possible that the simplification of the clause set could remove all clauses containing some literal, even if that literal was not explicitly assigned a value by the interpretation. In that case, the literal need not appear on the path at all. Furthermore, the clause set might come to contain some unit clauses. Literals from unit clauses are selected first, because just as in [20], the total amount of work is reduced by handling these obvious simplifications earlier. (When all clauses containing some literal are deleted in the process of assigning truth values to other literals, it just means that the literal that disappeared was irrelevant to the satisfiability of the clause set given the circumstances of the current interpretation.)

Each time we assign a truth value to a literal, we reduce the clause set by it. For instance, suppose that we assigned A to be *true*. Then any clause containing A should be deleted, and any clause that contains $-A$ should have that term removed. This is because the statement is a conjunction of clauses, and removing *true* from a conjunction preserves its truth value. Similarly, each clause is a disjunction, and removing *false* from a disjunction does not change its truth value. So we have shrunk the clause set without changing its meaning.

The set of clauses in Figure 3.3 represents two simple circuits made of exclusive-or gates, both having the same inputs, and a statement that their outputs are different. Since the two circuits represent the same function on the same inputs, but I have included a statement that their outputs differ, this clause set is unsatisfiable. I also include a spurious clause that will help illustrate the algorithm. Since the clause set already given is unsatisfiable, adding this demonstration clause does not change the end result.

The first of the two circuits is a chain of exclusive-ors, in which all but the first gate take the output of the previous as one input. That is, it computes the boolean

$$[a, b, -e][-a, -b, -e][a, -b, e][-a, b, e]$$

$$[c, e, -f][-c, -e, -f][c, -e, f][-c, e, f]$$

$$[d, f, -x][-d, -f, -x][d, -f, x][-d, f, x]$$

$$[a, b, -g][-a, -b, -g][a, -b, g][-a, b, g]$$

$$[c, d, -h][-c, -d, -h][c, -d, h][-c, d, h]$$

$$[g, h, -y][-g, -h, -y][g, -h, y][-g, h, y]$$

$$[x, -y][-x, y]$$

$$[e, -g]$$

Figure 3.3: Clauses stating that circuits of exclusive-or's compute different results. This is unsatisfiable.

function:

$$X = (((A \oplus B) \oplus C) \oplus D)$$

The second circuit is a tree of exclusive-ors, which is the boolean function:

$$Y = ((A \oplus B) \oplus (C \oplus D))$$

First, we choose variables to eliminate. Suppose the limit on the number of involved variables is five. We arbitrarily select a variable from the those having the smallest degree. We can see from the adjacency graph in Figure 3.4 that four variables are tied for this honor; a, b, x, and y. We choose the first of these, a. Because they are adjacent to a, the variables b, e, and g are also involved in the proof.

Only four variables are involved, and we decided that the limit was to be five, so we can involve only one more. We consider the neighbors of variables that have already been chosen for elimination. Since only a has been chosen thus far, we consider its neighbors; e, g, and b.

The variable e would involve a, b, c, and f. Of these four, a and b are already involved, so this would involve two new variables, making the total six. Since this is over the limit of five, we cannot yet choose e for elimination.

	Adjacency matrix										Degrees
	a	b	c	d	e	f	g	h	x	y	
a		+			+		+				3
b	+				+		+				3
c				+	+	+		+			4
d			+			+		+	+		4
e	+	+	+			+					4
f			+	+	+				+		4
g	+	+						+		+	4
h			+	+			+			+	4
x				+		+				+	3
y							+	+	+		3

Figure 3.4: Adjacency matrix for the atoms in clauses of Figure 3.3.

Similarly, the variable g would involve two new variables (h and y), so it also cannot be chosen.

The variable b is adjacent to a, e, and g, all of which are already involved. So we choose b for elimination with a, and the total number of involved literals remains four.

Re-examining the set of neighbors of chosen variables, we find that choosing either of them would still bring the total number involved literals to six, so we can choose neither. This makes sense, of course, because choosing b did not alter the set of involved literals at all.

Having exhausted the set of neighbors of chosen variables, we may now consider uninvolved literals. Although it rarely pays to do so, we do this here for purposes of demonstration.

The process for each of these is quite similar to that employed above when considering neighbors. To speed the discussion I show in Figure 3.5 the adjacency table again with the rows and columns for involved variables removed, and the degrees column updated to reflect that. This shows how many additional literals would become involved with each choice, and which literals they would be. Since none of these remaining variables is yet involved, choosing to eliminate one of them in this batch means increasing the involved count by one (for the variable itself) plus its degree as shown in this reduced table (for its as-yet uninvolved neighbors). Since we have room for only one more involved variable, this means that the degree would have to be zero. Since no variables meet this criterion, we are finished choosing variables to

Adjacency matrix							Degrees
	c	d	f	h	x	y	
c		+	+	+			3
d	+		+	+	+		4
f	+	+			+		3
h	+	+				+	3
x		+	+			+	3
y				+	+		2

Figure 3.5: Adjacency matrix for uninvolved variables after the initial choice.

eliminate.

Now the chosen variables are a and b and the extra variables are e and g, for a total of four involved variables. There are no more variables that we can choose for elimination without going over the limit of five involved variables, so we must stop.

Having chosen variables, we now choose clauses. Any clauses that contain our chosen variables (a and b) must be included. These are the following:

$$[a, b, -e][-a, -b, -e][a, -b, e][-a, b, e]$$

$$[a, b, -g][-a, -b, -g][a, -b, g][-a, b, g]$$

As expected, these clauses contain literals other than just the ones chosen for elimination; they include the extra variables e and g.

We also include any clauses that consist solely of those extra variables. There is just one such clause, and it was included in the original set solely to demonstrate this aspect of the algorithm:

$$[e, -g]$$

So, in all, we have chosen nine clauses:

$$[a, b, -e][-a, -b, -e][a, -b, e][-a, b, e]$$

$$[a, b, -g][-a, -b, -g][a, -b, g][-a, b, g]$$

$$[e, -g]$$

We now simplify the set of chosen clauses.

We will call the set of chosen variables $L1$, and the set of extras $L2$. Using rules similar to those for Davis-Putnam, we repeatedly choose and simplify away variables, making assumptions and backtracking as necessary. We must eliminate all variables from $L2$ before choosing any from $L1$. There are no pure literals or unit clauses on $L2$ variables, so we arbitrarily choose one, and apply the splitting rule to it.

(I tried always choosing the literal that occurred in the greatest number of clauses, but the work of finding that outweighed any performance improvement. Similarly for the literal that occurred in the fewest clauses.)

We choose e, and first assume it to be true. The chosen clause set is thus reduced to this:

$$[a, b][-a, -b]$$

$$[a, b, -g][-a, -b, -g][a, -b, g][-a, b, g]$$

The only remaining $L2$ variable is g, so we must choose it. It is neither pure nor does it occur in a unit clause, so we again apply the splitting rule, assuming for now that g is true:

$$[a, b][-a, -b]$$

We have now exhausted the supply of $L2$ variables, and can choose one from $L1$. There are no pure literals, and no unit clauses, so we arbitrarily choose a and apply the splitting rule to it, assuming it to be true:

$$[-b]$$

There is now a unit clause, so its literal must be true. (We could also apply the pure literal rule here, to the same effect.) Since $-b$ is true, b is false, and we again simplify the clause set:

$$(emptyset)$$

Since we have eliminated all clauses, we have found a model: e, g, a, $-b$. We now backtrack in search of more. But we are really only concerned with models of the $L2$ part of the semantic tree, and we just found one (e, g). So there is no point in backtracking within $L1$ and looking for other values of a and b that form models while keeping the same values for variables in $L2$; we must backtrack all the way into $L2$ and look for models with some different values for e and g.

We therefore revert to the clause set as it existed just before simplifying away g, and then assume the still-untried value for g (false). Just before simplifying away g, we had this:

$$[a, b][-a, -b]$$

$$[a, b, -g][-a, -b, -g][a, -b, g][-a, b, g]$$

Assuming g to be false this time, we end up with this:

$$[a, b][-a, -b]$$

$$[a, -b][-a, b]$$

Lacking any pure literals or unit clauses, we must again use the splitting rule, arbitrarily choose the variable a, and assume it to be true:

$$[-b]$$

$$[b]$$

We now have unit clauses, and arbitrarily choose the first one. (We could have chosen the second one, to the same effect.) Since $-b$ occurs in that unit clause, it must be true, and so b must be false. Simplifying away b, we produce the empty clause:

$$[]$$

This signifies that the current interpretation (e, $-g$, a, $-b$) contradicts the clause set, and is not a model. This is means that the current path through the semantic tree is closed at the current node.

Not yet having found a model, we backtrack within $L1$, and try the opposite of the assumption that made a true; a is now false. (We need not try making b true, because we made no assumption about it; we did not use the splitting rule to split the search into two subcases.) To make a long story short, backtracking to a different assignment on a also results in the empty clause. All paths through e and $-g$ are closed in $L1$, so the subtree rooted at that node is entirely closed.

We have now explored subcases for both possible values of g; one found a model while the other was entirely closed. Therefore, that closed subtree is maximal, and the clause that represents it ($[-e, g]$) will be in the output of this pass.

We are now halfway done with this batch of clauses; we have covered the case in which e is true, and must backtrack and consider the case in which e is false. Starting again with the chosen clause set, and simplifying away $-e$, we get this set:

$$[a, -b][-a, b]$$

$$[a, b, -g][-a, -b, -g][a, -b, g][-a, b, g]$$

$$[-g]$$

We now have a unit clause containing the literal $-g$. We still must try both g and $-g$ in order to generate all the right output clauses, but the subcase having $g = \mathit{true}$ becomes trivial. We do the trivial case first; assume g is true. After simplifying, we get this:

$$[a, -b][-a, b]$$

$$[a, b][-a, -b]$$

$$[]$$

Since this contains the empty clause, we have found a closed path for the interpretation $-e$ and g.

We next cover the other subcase, and assume $-g$ to be true (and so g must be false). Simplifying for that results in the following:

$$[a, -b][-a, b]$$

Using the splitting rule on a, and assuming it to be true, we get:

$$[b]$$

This results in a unit clause, whose literal must be true. (Once again, we could have employed the pure-literal rule instead, with the same results.) Simplifying on b, we get the empty set of clauses, signifying that we have again found a model in the current interpretation: $-e$, $-g$, a, b. As before, there is no point in backtracking within $L1$ after finding a model, so we continue backtracking into $L2$.

The current value of g was already the result of backtracking. For one of its values the subtree was completely closed, while the other produced a model. Therefore that first closed subtree was maximal, and the clause that represents it is output: $[e, -g]$.

The value of e (false) was already the result of backtracking; it is the second of two subcases. Neither of those two subtrees was completely closed, so neither can be maximally closed, and we need not output any clauses at this point.

So we have now completely explored the semantic tree, and generated only two clauses to replace the chosen set: $[-e, g]$ and $[e, -g]$.

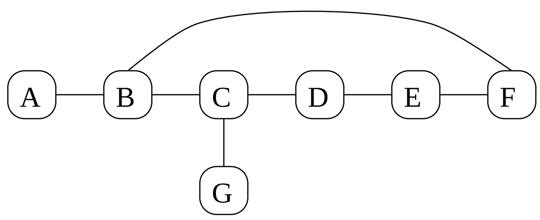
Notice that the clause $[e, -g]$ in the original clause set did not alter the results of the simplification, but greatly reduced the amount of work involved in half of the tree traversal. (The shortcut to knowing this is that the chosen clauses represented a pair of exclusive-or gates, both of which had the same inputs. So their outputs must always have the same values, which is exactly what the output pair of clauses states. If you were already used to the notation, you would have seen this coming.)

3.3 Selecting Literals to Eliminate

In selecting clauses to work on, recall that we first had to choose literals to eliminate. The goal is to find some literals that we can eliminate while keeping the proof small. Recall that after choosing these literals, we will take every clause that contains one or more of these, and those clauses will contain extra literals as well. Since we will be traversing a semantic tree on the literals in those clauses, the extra literals contribute just as much toward the difficulty of the problem as the chosen ones. We say that the chosen literals and extra literals are all *involved*. Therefore, to assist in choosing literals, we view the clause set as a *co-occurrence* graph, in which the nodes are atoms, and two atoms are adjacent in the graph if and only if they occur together in some clause. Note that for purposes of this graph, we are dealing with atoms, ignoring the signs of the literals; A is the same as $-A$. Now we can get a good estimate of the difficulty that will result from choosing a literal, by looking at that literal's degree. If the literal has a high degree, then a large number of other literals will be dragged into the proof with it.

a) [A B] [-B C] [B -C] [-C D]
[-D E] [-E F] [-F B] [-C G]

b)



c)

Degree	Literals
0	
1	A G
2	D E F
3	B C

Figure 3.6: The simplification of a clause set, as per Algorithm 2. a) The set of clauses. b) The co-occurrence graph. c) The literals having each degree.

See Figure 3.6 for an example of a co-occurrence graph. Note that in that figure, literals B and C have degree 3, despite each appearing with another literal in four clauses. This is because the second and third clauses both contain B and C; B counts as C's neighbor only once, and vice-versa.

After choosing our literal, it is possible that the total number of literals involved could be well below the limit that we can easily handle in one step. If that happens, we might try to include more literals in the chosen set. It is helpful to consider only neighbors of literals already chosen, because it reduces the number of literals to consider, while still usually finding the best candidates. This is based on the property of most real-world problems of locality; literals are not adjacent to each other in a random pattern, but tend to share neighbors. One apparent cause of this is that we build systems hierarchically; there is a self-imposed limit on the range of things that any component might connect to. This implies that including a neighbor will tend to increase the number of involved literals by less than including a non-neighbor, because its neighbors are likely already involved.

3.4 Selecting Clauses

At a minimum, each pass must include all clauses that contain any of the chosen literals. However, there can be some benefit to including other clauses as well. The only restriction is that all of the literals in any included clauses become involved in the proof, potentially deepening the semantic tree. However, we might find clauses that are made up exclusively of literals that are already involved. If we include those, not only do we not risk deepening the semantic tree, but can actually reduce the amount of work needed to simplify it.

Referring once again to Figure 3.6, selecting variables A and G means we must select the first and last clauses. But choosing literals A and G means involving B and C as well, we can also select the second and third clauses, which contain only these extra variables.

Note that we have already included all of the clauses that contained any of the chosen literals; there are no more of those to be had. So, after doing that, if we find any more clauses that contain only involved literals, it means those new clauses must contain only extra literals — the ones that were contained in the set of chosen clauses that were not chosen for elimination. Since the tree that we build will put the extra literals closer to the root, this means that these extra clauses will close nodes in $L2$. Depending on how deep the tree is, and how low in the tree the closed node is, this could save quite a bit of work in the tree traversal.

The worst case seems to be that the extra clause closes a node which we will never visit anyway, for being on a path that is somehow closed at an even lower node. In that case, we would have wasted a little bit of work by including that clause, and not gotten any benefit from it. But that cannot actually happen, because all of the initially chosen clauses contain at least one chosen literal, so they will close nodes in $L1$ in the semantic tree. Only clauses made up solely of extra literals can directly close nodes in $L2$, so only one of these extra clauses (or a set of them) could make another such clause useless. So, at most some small fraction of all extra clauses is at all likely to be useless.

3.5 Algorithm

We are now ready to see the full algorithm.

In Algorithm 1 we see a few simple data definitions, and the highest level loop of the process in which sets of literals are repeatedly chosen and eliminated. The notation used for this pseudo-code is a mish-mash of set notation and many real programming languages including C, Prolog, and others. The *linkedarray* data structure is discussed in detail in chapter 4, but can be thought of as a mapping from integers to data items. The top-level program function is *satisfiable*, which takes as its argument a set of clauses, and returns a boolean indicating whether that set was satisfiable. The crux of the main loop is the call to the *simplify* function, which returns a triple. In this case the first two elements of the returned triple are ignored, and the third is assigned to R'. But that function is recursive; when it calls itself the three elements of the returned triple are all needed.

In Algorithm 2 are the functions that choose literals to be eliminated, and choose the clauses containing those literals. I employ set notation here, deferring implementation details. The *chooseLiterals* function first chooses a literal of minimum degree, exploiting the knowledge that the linkedarray *degrees* is in ascending order. It then continues choosing literals until the number involved meets or exceeds the threshold of ten. To choose clauses containing those literals, *chooseClauses* simply makes use of the data structure that lists all clauses for each literal. Since the result is a set, no clause is included more than once.

The recursive function called from the main loop is shown in Algorithm 3. It takes as inputs a "state" structure representing the clause set after simplification by the interpretation, the original clause set, and the interpretation. At first, the interpretation is empty, and the state matches the original clause set. With each additional recursion, one literal is added to the interpretation and the state represents a reduced version of the clause set. At the beginning of the function, the state is tested for being "final" – empty or containing the empty clause. If either of these two conditions hold, then the function returns with an indication that all or none of the interpretations extending the current one contradict the clause set (i.e., that all paths through this node of the semantic tree are closed.) If not in such a final state, a literal which has not yet been assigned a truth value is selected. It is temporarily assumed to be true, and the *stateTransition* function is called to compute the effect of simplifying the clause set for this addition to the interpretation. The results of exploring that first of two subtrees are cached, and then the state is restored to its value prior to the truth assignment. Under some circumstances, we will not need to use the negated value of that literal and explore the second subtree. One such case occurs when an open path is found in the $L1$ portion of the tree, the other is when the literal just selected was not used in closing the first subtree. When it is necessary, the second subtree is handled just like the first, except that the negation of the selected literal is used instead of its positive form. After checking the second subtree, the results of the two subtrees are combined to produce a result for this invocation.

Algorithm 1 (Test for satisfiability)

> ***type*** *LiteralInfo* $\equiv$ ***record***
>   ***integer***: *degree*
>   ***map***$\langle$***integer*** $\rightarrow$ ***integer***$\rangle$: *neighborCounts*
>   ***set***$\langle$***ref to*** *clause*$\rangle$: *containingClauses*
> ***end***
> ***array***$\langle$*LiteralInfo*$\rangle$: *litinfo*
> ***map***$\langle$***integer*** $\rightarrow$ ***set***$\langle$***integer***$\rangle$ $\rangle$: *degrees*
>
> Input: *A set of propositional clauses*
> Output: *True if the clause set is satisfiable, or False if it is not.*
> ***function*** *satisfiable(S)* : ***returns boolean***
>   *initializeStructures(S)*
>   ***while*** ($S \neq \{\}$ ***and*** $[] \notin S$ ***and not*** *trivialModel(S)*) ***do***
>   *L1* $\leftarrow$ *chooseLiterals(S)*
>   *R* $\leftarrow$ *chooseClauses(S,L1)*
>   *Q0* $\leftarrow$ *initialState(R,L1)*
> $(\_,\_,R')$ $\leftarrow$ *simplify(Q0,R,[])*
> $S \leftarrow S - R + R'$
>   *unitPureSimplify(S)*
>   ***enddo***
>   ***return*** $(S = \{\})$ -- satisfiable iff reduced to empty set
> ***end***

Algorithm 2 (Choose variables and clauses to work on)

```
 Input: A set of propositional clauses
Output: A set of literals to eliminate
function chooseLiterals(S) returns set⟨literal⟩
    chosen ← first( first( degrees ) )   -- minimum-degree literal
    while ( |involved(chosen,S)| < 10 ) do
        neighbors ← { L—L ∈ involved(chosen,S), L ∉ chosen }
        for each neighbor in neighbors do
            if ( |involved(chosen+neighbor,S)| < 10 ) then
                chosen ← chosen + neighbor
            endif
        enddo
    enddo
    return chosen
end

 Input: A set of literals and a set of propositional clauses
Output: The subset clauses containing those literals
function chooseClauses(L1,S) : returns set⟨clause⟩
    return { C | L ∈ L1, C ∈ litinfo[L].containingClauses }
end
```

Algorithm 3 (Simplify a set of clauses)

```
 Input: A state, a set of propositional clauses, and an interpretation
Output: A flag indicating closure, a list of literals used,
and a list of clauses closing maximal subtrees
function simplify(Q,S,I)
: returns (ClosureFlag,set⟨literal⟩,set⟨clause⟩)
    if finalStateUnsat(S,Q) return (ALL,abs(contradicted(S,Q)),{})
    if finalStateSat(S,Q) return (NONE,{},{})
    A ← chooseUnassignedLiteral(S,I,L1)
    NA ← −A

    stateTransition(Q,A)
    (rc1,used1,max1) ← simplify( Q, S, I ∪ {A} )
    undoStateTransition(Q,A)
    if (A ∈ L1 and rc1 ≠ ALL) return (NONE,{},{})

    if (rc1 = ALL and A ∉ used1) return (ALL,used1,{})

    stateTransition(Q,NA)
    (rc2,used2,max2) ← simplify( Q, S, I ∪ {NA} )
    undoStateTransition(Q,NA)
    if (A ∈ L1 and rc2 ≠ ALL) return (NONE,{},{})

    if (rc1 = NONE and rc2 = NONE) return (NONE,{},{})
    if (rc1 = ALL and rc2 = ALL) then
        return (ALL,(used1−A)+(used2−A),{})
    endif
    if (rc1 = ALL) max1 ← negate(unsignedIntersect(I+A,used1))
    if (rc2 = ALL) max2 ← negate(unsignedIntersect(I+NA,used2))
    return (SOME,used1+used2−A,max1+max2)
end
```

Algorithm 4 (Helper functions)

Input: *A clause set, an interpretation, and a set of preferred literals*
Output: *a literal that has not yet been assigned a value*
function *chooseUnassignedLiteral(S,I,L1) :* ***returns*** *literal*
    assert *S contains at least one literal not assigned a value by I*
    $Literals \leftarrow \{\ abs(L) \mid C \in S,\ L \in C\ \}$
    $Unassigned \leftarrow Literals - unsignedIntersect(Literals,I)$
    $U1 \leftarrow Unassigned \cap L1$
    $U2 \leftarrow Unassigned - U1$
    assert $U1 \neq \{\}$ ***or*** $U2 \neq \{\}$
    -- prefer L2 to L1
    $U \leftarrow$ ***if*** $U2 \neq \{\}$ ***then*** *U2* ***else*** *U1* ***endif***
    $Unit \leftarrow \{\ L \mid C \in S,\ abs(C) - I = \{L\},\ L \in U\ \}$
    -- prefer literals from unit clauses
    return *anyof(* ***if*** $Unit \neq \{\}$ ***then*** *Unit* ***else*** *U* ***endif****)*
end

Input: *A set of literals and a set of clauses*
Output: *The set of all literals from clauses containing input literals*
function *involved(lits,clauses) :* ***returns set***$\langle literal \rangle$
    return $\{\ L \mid C \in clauses,\ M \in lits,\ \pm M \in C,\ L \in C\ \}$
end

Input: *Two sets of literals*
Output: *A subset of the first set, containing those literals that occur in the second set, ignoring signs*
function *unsignedIntersect(A,B) :* ***returns set***$\langle literal \rangle$
    return $\{\ L \mid L \in A,\ (L \in B$ ***or*** $-L \in B)\ \}$
end

3.6 Early Termination

At the end of each pass, we do one quick test for a simple characteristic that indicates satisfiability. Failure to detect satisfiability at this point does not necessarily indicate that the clause set is unsatisfiable, it just means that we have more work to do before we can decide. This test is not strictly necessary for the prover to operate correctly, but is very inexpensive, and has been found to greatly increase the speed of some proofs.

The test is this: if every clause contains at least one positive literal, then the clause set is satisfiable, and the model is that every variable is assigned *true*. The complement of this test is also run; if all clauses contain at least one negative literal, then there is a model in which every variable is assigned *false*.

The test fails to produce any useful result if there are some clauses that contain no positive literal (i.e., there are all-negative clauses), and some clauses that contain no negative literal (all-positive clauses). In this case, the prover must continue with its ordinary routine to conclude one way or the other.

Also at the end of each pass, we do some quick simplifications of the clause set. These are the same two simplifications that Davis-Putnam uses during its semantic tree traversal; unit literal and pure literal deletion. As in Davis-Putnam, use of the pure literal simplification means that we will no longer find all models if the clause set is satisfiable. This can be easily omitted if desired.

Also like Davis-Putnam, these simplifications can terminate the proof by producing an empty clause or by deleting the last clause. But even if they do not directly complete the proof, they can reduce the clause set, making future passes of the main procedure that much more efficient.

3.7 Correctness

For a theorem prover to be correct, it must be both *sound* and *complete*. The proof that Sttrim is complete shows that the input clause set contains a finite number of distinct literals, and that this number is reduced with each pass. The proof of soundness is that each pass preserves satisfiability. Given that, we know that the end result is satisfiable if and only if the original was. So the prover ends indicating that the clause set was satisfiably only if it really was; it is sound.

3.7.1 Completeness

Theorem 1 *Sttrim is complete.*

That the input clause set has a finite number of distinct literals is a given. Showing that each pass reduces the number of literals therefore demonstrates that the process will terminate after a finite number of passes for any input.

Lemma 2 *Each pass reduces the number of literals.*

Proof: The input to a pass is a clause set, from which are selected clauses containing at least one *chosen* literal and some number of *extra* literals. Furthermore, all clauses that contain any chosen literal are chosen; the remainder of the clause set contains no chosen literals. The operation of the pass is to replace the selected clauses with a newly generated set. That newly generated clause set contains only those extra literals, it does not contain any chosen literals. The output of the pass is composed of those newly generated clauses, plus the remainder of the input clause set. Neither of those clause sets contains any chosen literals, nor have any new literals been added. Therefore, the number of literals has been reduced. ■

3.7.2 Soundness

Theorem 3 *Sttrim is sound.*

Since each pass preserves satisfiability, the entire sequence of passes does as well. Therefore, proof that each pass preserves satisfiability is sufficient to show that the entire sequence of passes is sound.

Lemma 4 *Each pass preserves satisfiability.*

Restated, the output of a pass is satisfiable if and only if its input was.

Proof:

($\leftarrow$) I show that if the input clause set is satisfiable, then the output clause set was satisfiable as well. If the input set was satisfiable, it had some model. That means there was some path through the semantic tree of the input clause set that was not contradicted by any of those clauses. The output clause set consists of clauses taken from the input set plus some newly generated ones. Those taken from the input clause set obviously cannot contradict the model. The newly generated clauses close only the paths that were closed by the selected clauses from the input set. Therefore, any path that was open for the input clause set remains open for the output clause set;

if the input was satisfiable, then the output is as well. The only subtlety is that the model of the input clause set will contain assignments to literals that do not exist in the output set. Stripping those out results in a model of the output set.

($\rightarrow$) I show that if the output clause set is satisfiable, then the input clause set was satisfiable as well. To see this, consider the semantic tree again. The paths that are closed in the output set are closed either in the subset that was inherited from the input set, or were generated because they were closed by the selected clauses in the input set. So any path that remains open in the output set must have had some open path in the input set.

$\blacksquare$

3.8 Summary

This chapter presented the algorithm for Semantic Tree Trimming, stepped through some tiny examples, and showed its correctness. At a high level, it operates by trimming layers off of a semantic tree while preserving open paths. A number of refinements and hueristics were also presented, that allow it to operate more efficiently.

Chapter 4

Implementation

This chapter describes some of the implementation details, including heuristics, data structures, and other details not presented as part of the algorithm proper.

4.1 Multiple-Assignment Semantics

Early versions of the prover were written in Prolog. Although the asymptotic behavior showed promise, the overhead in the manipulation of data structures was too high. The main problem seems to be that the algorithm requires multiple-assignment semantics; modifications to the clause set without multiple-assignment semantics entails either making a copy of the entire set with the changes incorporated, or some kind of edit list. But an edit list means either inefficient access or some kind of index that itself would need to be modified, bringing us back to the original problem of modifying a large structure.

Eventually, this cost drove me to re-implement the entire prover in C. With the C implementation immediately came an order of magnitude improvement in performance. Another order of magnitude came with the careful re-work of some data structures that were ported too literally from Prolog. In particular, many linked lists and sequential searches of them were replaced with arrays or linked arrays, which are a data structure that will be described later. But the biggest gain was from simply being able to over-write data in-place. In places where the Prolog implementation might have had to copy thousands of clauses to make a change to a single one, the C version simply changes that one in-place.

Of course, Prolog exists for a reason, and some of its features had to be imitated in the C implementation. The most notable of these is backtracking, which is needed during the tree traversal in the elimination of a batch of literals. The C code to do this handles it by a recursive function which calls itself once for each subtree, so the backtracking is implicit in the return from that function call.

4.2 Heuristics in Choosing Literals to Eliminate

Recall that we chose some literals to eliminate with each pass. The method could work with any choice of variables for each pass, but some choices will improve performance, and we employ a heuristic to find these.

All of the techniques tried involved making some initial choice of literals to eliminate, and then adding more until a limit was reached.

The best of these involved operations on the adjacency graph. Early implementations partitioned this into subgraphs, and then considered the groups of literals from each of those subgraphs as a whole. This had the advantages that there were fewer candidates to choose from, and the choice of literals was done at a higher level. Other implementations considered literals individually instead. This lower-level choice runs the risk of being too greedy at early stages, resulting in poorer performance later on. Still, the expense of graph partitioning outweighed its benefits. Perhaps the benefit of graph partitioning would be greater if the simplification algorithm allowed for larger pieces, thus giving greater scope for this stage to find a good partitioning.

When considering literals for inclusion, the primary limitation is that we need to keep the number of involved literals to a minimum. When choosing an initial single literal, this means choosing the literal with the smallest degree. As we will see, this can be done fairly efficiently if we maintain data structures as the clause set changes.

The choice can be made with no maintenance work required if we simply choose an initial literal arbitrarily. As might be expected, although the choice itself can be made quickly, it very often chooses a literal that involves too many others, making the later simplification step perform very poorly. Comparison against this baseline showed that this initial choice of literals is important.

Having made an initial choice, we will generally want to add onto it. If our initial choice had been done by partitioning the co-occurrence graph, we might consider including other whole partitions. Whether or not we do that, we might also consider individual literals for inclusion. Whether we are considering whole partitions or individual literals, the best candidates are nearly always among the neighbors of those already chosen. Although earlier approaches tried considering other literals if the search of neighbors failed, this turned out to produce very poor results relative to the amount of work involved. So in the current implementation, only literals that are neighbors of some already-chosen literal are even considered.

The data structures necessary for quickly finding the literal of smallest degree are mostly arrays, linked lists, and a combination of those two called a *linked array*. A diagram showing the structure of a linked array appears in Figure 4.1. The array is pre-allocated to the necessary size, and the linked list is initially empty. Each element of the array is either a reference to a node on the linked list or null, and each node on the linked list contains both the data item and an integer index back into the array, indicating the array element that contains its reference. Thus there is a one-to-one relationship between non-null elements of the array and nodes on the linked list. The result is a map from integers to data elements, in which certain operations are very efficient. Given an integer to look up, we can find it in the array in $O(1)$ time, and we can enumerate the N elements in the array in $O(N)$ time. If the linked list must be kept sorted (that is, if we will need to quickly enumerate the elements in sorted order), then there is some necessary overhead in insertion of new elements. If the data has the right properties, this overhead can be shown to be small. On the other hand, if the data need not be enumerated in sorted order, then we can use unsorted insertions, which are $O(1)$. Deletions are also $O(1)$. The cost of all this, of course, is space — $O(N)$ space, where N is the maximum number of items that the structure can hold. And this structure is really only beneficial if the data is sparse, so the space complexity is a serious concern. Some few of the benchmark problems could actually not run to completion because of this space limitation. The obvious replacement for this is a threaded tree, allowing $O(\log N)$ operations in $O(N)$ space. However, this is an item for future work.

The current form of Sttrim uses these data structures, and the heuristics of first choosing the literal of minimum degree, and then including neighbors as long as the total number of involved literals does not exceed ten. Early implementations that used these heuristics, but simple linked lists (requiring frequent linear-time searches) achieved run times within an order of magnitude of the best hand-made partitioning I was able to produce. These data structures improved results still more. Note that quite a lot of work went into partitioning some of the early problems by hand, at an earlier stage of this work at which it was not apparent that automated partitioning would be practical. Therefore it is quite pleasant to have an automatically-generated partitioning work nearly as well, even if it has not yet surpassed the manually gener-

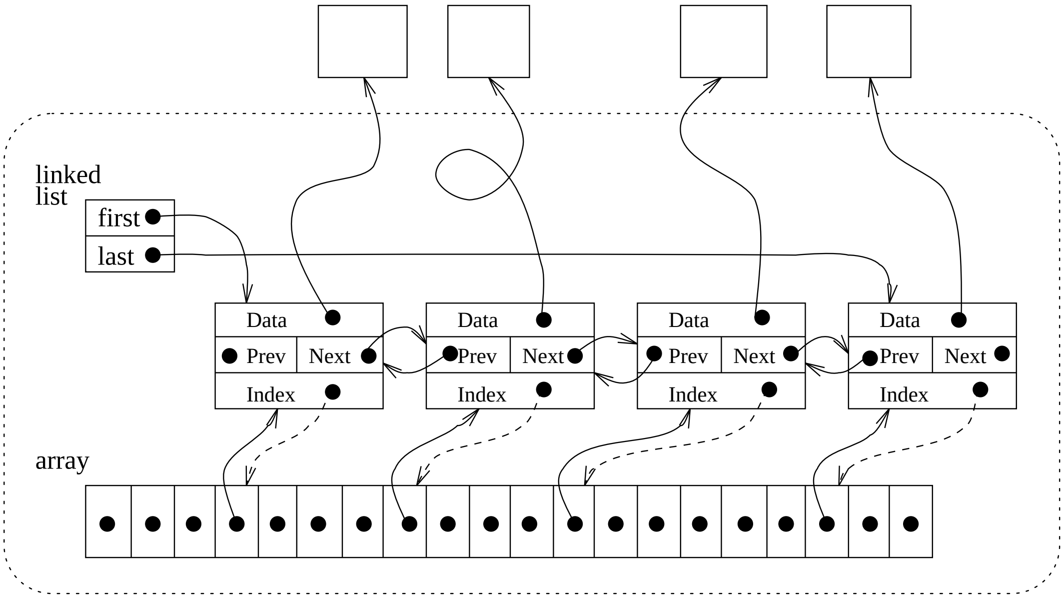


Figure 4.1: A linked array consists of an array and a linked list. The solid-line arrows are references, and the dashed-line arrows are indexes into the array. Dots with no outgoing arrows are null references.

ated ones in all cases.

There is an obvious reason to place an upper limit on the number of literals involved in a pass: to bound the depth of the semantic tree. We might also set some lower limit on how many literals will be chosen for elimination in each batch. This obviously conflicts with having an upper limit on how many literals can be involved, so to be useful this lower limit must have higher priority. Unfortunately, when such a lower limit is specified, the current implementation will choose that many literals even if it turns out to be very expensive. With a reasonably small lower limit, the effect is usually beneficial, with most passes finding enough literals to eliminate without involving too many extra literals. But a few passes in every proof seem to drag in far too many extra literals, costing far more time than was saved by all of the other passes put together. So overall, such a fixed lower limit is not beneficial.

As the proof proceeds, the smallest degree tends to increase, because we continue removing literals with small degrees, and must sometimes add literals with larger ones. In some proofs, this means it eventually gets to the point that any single literals that we choose necessarily involves a large number of extras. At that point, our strategy of biting off "small" pieces of the proof no longer helps very much. To handle this, we allow another upper limit, designating the degree at which we should simply choose all remaining literals for elimination in a single pass. Note that with such an upper limit, our method in the worst case degenerates to Davis-Putnam.

4.3 Heuristics in Simplifying a Clause Set

All of the clauses that contain chosen variables are included in R. To improve performance, we might choose to include some clauses that do not contain any chosen variables, but we need not do so for the method to work.

In the previous chapter, I indicated that clauses are generated only for maximally-closed subtrees. To facilitate this, the simplification is organized as a recursive function, which takes an interpretation as one of its arguments, indicating a node in the semantic tree. It returns an indicator of whether that entire subtree is closed; one or more of its subtrees are closed; or it is entirely open. This function is called once for each node that is explored. Indeed, it is the calling of this function that explores the node. So, when a closed node is found, instead of a clause being generated, the function just returns with the indicator telling that this entire subtree was closed. After the calling instance of that function has completed its two function calls to evaluate its two subtrees, it uses the completion information returned by them to decide whether or not to generate a clause. If both subtrees are closed, then their common parent node is effectively closed, and no clause need be generated yet. If one subtree was entirely closed and the other was not, a clause is generated for the closed one. Finally, if those subtrees were partially closed and partially open (that is, some of their subtrees were closed), then the clauses representing those maximally closed subtrees were generated by the function when it discovered them, and need not be dealt with by this this higher-level invocation of the function.

For each invocation of the simplification function, the interpretation must first be extended by one or more variables. If it were not extended, then the invoked function would explore the same semantic tree node as its caller, to no benefit. The choice of variables to add onto the interpretation is the subject of another heuristic. The only real restriction on this choice is that the algorithm only works correctly if maximally-closed subtrees are found for interpretations that do not include any $L1$ variables. In practice, this means eliminating all $L2$ variables before choosing any $L1$ variables.

Whether there are any $L2$ variables to choose or not, it would be best to minimize the amount of work necessary to traverse the tree. To the end, I tried choosing the variables that occurred in the greatest number of clauses. But the work of finding these exceeded the benefit of it. Surprisingly, choosing the variables that occurred in the smallest number of clauses resulted in about the same performance as choosing those that occurred in the greatest number. The reason for this seems to be that literals are not generally distributed so unevenly that the choice makes much difference, and the work done in finding them is the same whether searching for the smallest or the greatest. The best overall heuristic has therefore turned out to be choosing the first available variable from the appropriate set, either $L1$ or $L2$.

After choosing a variable to add to the interpretation, we choose a truth value for it, add it to the interpretation, and simplify the clause set by it. This means deleting clauses that have become true, and deleting literals that have become false from the clauses that contain them. And this has to be done in a way that can be un-done efficiently, because the algorithm will backtrack over each assignment to try other ones.

To accomplish this, we keep a two-dimensional structure much like overlapping linked lists, in which any node can be independently unlinked from its row or column. A diagram showing one of these appears in Figure 4.2. When unlinked, a node keeps a copy of the links that were broken, so they can easily be restored later. Columns represent clauses, and each node that is linked into a column represents a literal that remains in the clause. Each row represents a literal, and the nodes linked into it indicate which clauses contain that literal. Rows are ordered in such a way as to group all $L1$ literals together at the beginning of the list, followed by all $L2$ literals. The pair of rows for a literal and its negation can be viewed as transition rules for a finite state machine, in which each assignment of a variable and the subsequent simplification of the clause set is a transition. The rows indicate which clauses need to be adjusted for any assignment of a truth value to a variable. The search for a model can then be viewed as a nondeterministic finite state machine, which we simulate on a deterministic machine by means of backtracking. This is reflected in Algorithm 5, which defines the two-dimensional structure as part of the machine state. For each simplification, two rows form the transition rule; the row for the positive form of the literal and the row for its negative form. Depending on which truth value was assigned to the variable, one of these rows indicates which clauses have become true and should be deleted, and the other indicates clauses from which the newly false literal should be removed. The two dimensional structure is important because not only do we adjust clauses as we go, but we also adjust the transition rules themselves: deleted clauses are removed from transition rules, making future transitions more efficient.

Because each node keeps copies of links that are broken when it is unlinked from its row or column, the simplifications done by the algorithm can easily be un-done during backtracking. The initialization of this structure is shown in Algorithm 6.

Algorithm 5 (State and rulenode definitions)

```
type state ≡ record
    -- A sparse two-dimensional matrix of rulenodes for fast simplification.
    -- A row holds info about all instances of a literal
    -- A column holds info about all literals in a clause
   array⟨linkedlist of ref to rulenode⟩ : rows;

    -- Lists categorize clauses for fast selection
   linkedlist of ref to clause :
       empty, removed, l2Unit, l2Any, l1Unit, l1NoL2;
end

type rulenode ≡ record
    -- A node in a two-dimensional matrix, used as a
    -- placeholder for info about a literal in a clause.

    -- links to my row and column.
   linkedlistnode : rnode, cnode;
    -- Remember which clause this literal belongs to.
   ref to clause : cla;
    -- Remember which literal too.
   literal : lit;
end
```

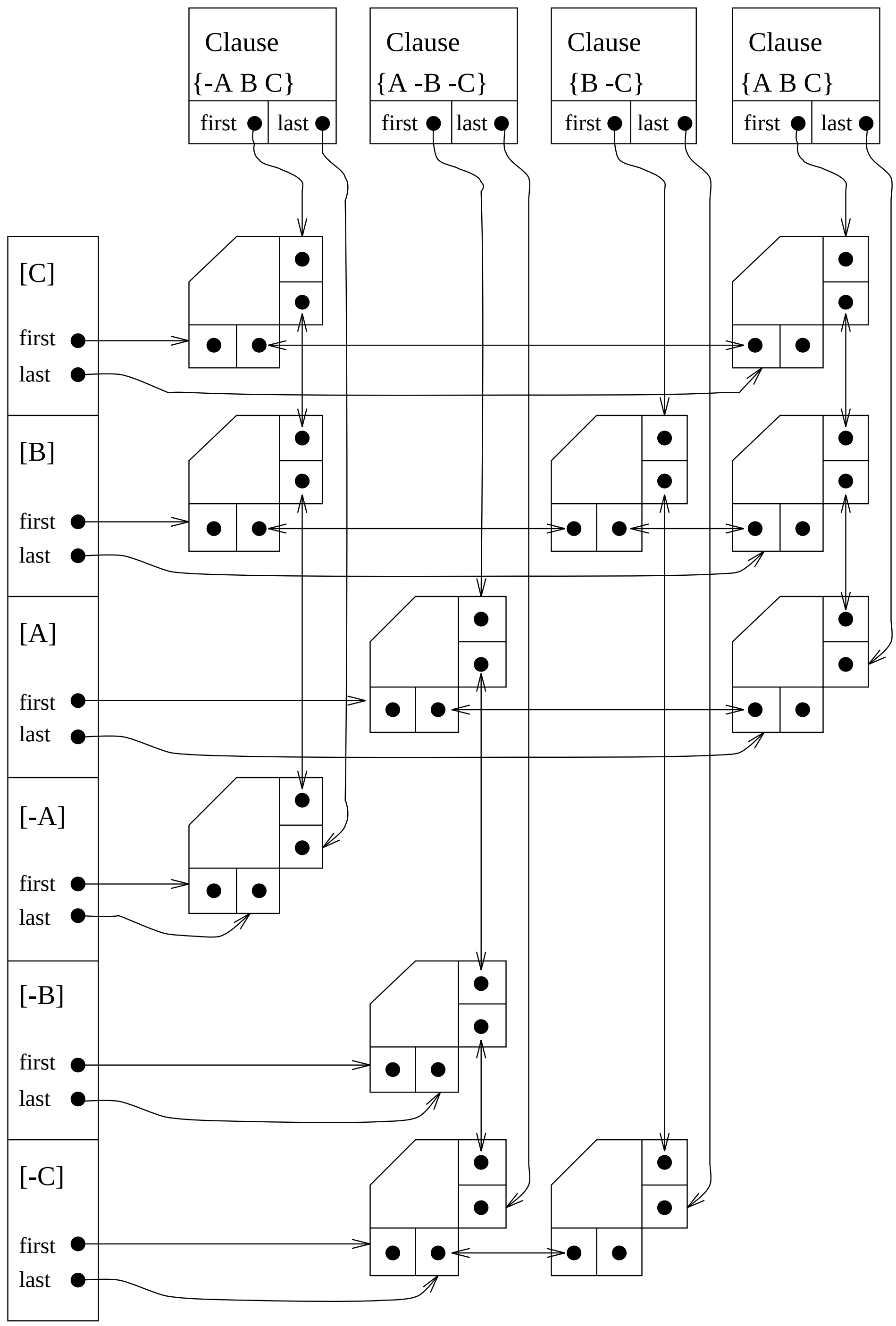


Figure 4.2: A 2-D linked list, showing just the high-level structure. The row list headings are kept in an array that is indexed by literals, and the column list headings are kept in the structures that describe clauses.

Algorithm 6 (State initialization function)

```
Input: A list of clauses and a list of chosen literals
Output: An initial state. Also builds transition rules deltaP, deltaN
function initialState(S,L1) : returns state
    Q0 ← new state
    Q0.rows ← new array⟨linkedlist⟩ [ -maxLit .. maxLit ]
    for ref to clause : C  in S do
        integer: l1count ← 0, l2count ← 0
        for literal : L  in C do
            rulenode : r ← new rulenode
            r.cla ← C
            r.lit ← L
            ListAppend( Q0.row , r.rnode )
            ListAppend( C.col , r.cnode )
            if (L  in L1) then l1count ++ else l2count ++ endif
        enddo
        ref to linkedlist : category ← COND (
            l1count == 0 and l2count == 0, Q0.empty
            l1count == 1 and l2count == 0, Q0.l1Unit
            l1count == 0 and l2count == 1, Q0.l2Unit
            l1count ≥ 1 and l2count == 0, Q0.l1NoL2
            l2count ≥ 1, Q0.l2Any
        )
        ListAppend( category, C )
    enddo
    return Q0
end
```

There are two basic operations needed on this data structure: one to eliminate literals that have become false, and one to eliminate clauses that have become true. These are shown in Algorithm 7. Also needed are the inverse operations. Deletion of false literals is the simpler of the two; we simply traverse the list of clauses containing that false literal, and unlink each node from its clause. Since those nodes remain on the list for this literal, we can restore them to their clauses by traversing that list again and restoring the broken links from the saved copies. Each of these operations is $O(N)$, where N is the number of nodes, equal to the number of occurrences of that false literal that remain in the clause set at the time of the operation. Deletion of clauses that have become true is more involved, because all literals of each clause that becomes true are removed from their transition rules. We do this by first unlinking from its column each node of the row for the literal that has become true, just as we did above for false literals. Then we traverse the list of nodes on the row we just unlinked, and for each node n, get the pointer to its containing clause, and traverse its list of nodes. Each of that clause's remaining nodes is unlinked from its transition rule. (The node for the true literal has already been removed from this clause's list.) After doing all this, those clauses are effectively removed from all transition rules, except for the row of the literal that made it happen. Therefore we can still un-do this by starting with that one transition rule and following the same steps as before. Each of these two operations leaves a path to the deleted nodes through the literal that was assigned a truth value and necessitated the work. Although they are still present in the structure, this is effectively deletion, because that literal will not be chosen for assignment again.

Algorithm 7 (State transition functions)

```
 Input: A state and a literal
Output: modifies the state to take account of L being assigned True
     (L may be –M, in which case M is False)
function stateTransition(Q,L)
     -- remove this literal from all clauses
     -- unlink its two rows
   for each literal : lit  in { L, -L } do
      for each rulenode : n  in Q.rows [ lit ] do
         ref to clause : c ← n.cla
         ListUnlink( c.col, n.cnode )
      enddo
   enddo

     -- remove clauses containing L
     -- unlink the column for each clause
   for each rulenode : n  in Q.rows [ L ] do
      ref to clause : C ← n.cla
       -- Clause C contains L; remove it from all rows
      Foreach rulenode : m  in C.col do
         literal : CL ← m.lit
         ListUnlink( Q.rows[ CL ], m.rnode )
      enddo
   enddo
end

function undoStateTransition(Q,L)
     -- Undoes the changes made by stateTransition(),
     -- in the obvious way.
end
```

Changes to this structure obey stack semantics; it is restored in exactly the reverse of the order in which is was altered. So when the time comes to restore those broken links, we know that the nodes that the saved pointers refer to are in-place, even if they had themselves been unlinked at some deeper level of the function's recursion. The details would have been much more complicated if we were not guaranteed this behavior.

One heuristic employed here was the decision to remove nodes of true clauses from their transition rules. The alternative, which would have been faster, would have been to just mark those clauses as true, but leave their nodes linked. The advantage of removing those nodes is that, although it needs more work, it reduces the amount of work to be done by later passes. The exact amount of that reduction depends on the structure of the clause set. If the literals of those deleted clauses are later used in closing paths in the semantic tree, then the continued existence of those clauses would mean more work each time one of those other transition rules is traversed, which is each time one of those literals is assigned a truth value.

In addition to that two-dimensional linked list, we also keep six lists of clauses, to increase the efficiency of deciding which literal should be assigned a truth value next. Like the two-dimensional linked list, these lists are shown in Algorithm 5, and their initialization is shown in Algorithm 6. During the tree traversal, each clause will appear on exactly one of these six lists. Choosing a literal to assign a truth value is done by first choosing a clause from one of these lists, and then choosing a literal from that. For each list, it is an $O(1)$ operation to get the first one of these, or to recognize that the list is empty. The first list checked is of unit clauses containing an $L2$ literal. Failing that, we choose from a list of clauses that contain some number of $L2$ literals, or the list of $L1$ unit clauses, or the list of clauses containing more than one $L1$ literal but no $L2$ literals, in that order. The remaining two lists are kept for other purposes; the list of empty clauses and the list of clauses that have been removed for containing a literal that was assigned to be true. Having chosen a clause, we can find a literal from it using the two-dimensional linked list described earlier. Since the literals in any clause are arranged with all $L1$ literals preceding all $L2$ literals, we can conveniently get a literal belonging to whichever set we prefer by simply choosing the first or last node on that clause's list.

With these lists, we can implement our heuristics to choose a literal in $O(1)$ time, and each clause that we simplify (or restore during backtracking) can be moved to the appropriate list in $O(1)$ time.

Each time a literal is assigned a truth value, it is possible that the newly-extended interpretation satisfies the clause set, or is contradicted by some clause in the set. Checking for these things can be done in $O(1)$ time, as shown in Algorithm 8. If every clause has at least one literal that is known to be true, then all of those six lists will be empty except for the list of removed clauses. This constitutes five $O(1)$ operations, and indicates that the interpretation is a model. (It is only five operations, and not six, because if five of lists are empty then the sixth must contain all clauses and need not be tested.) On the other hand, if there is any clause which has been emptied, then that clause contradicts the interpretation. This means checking if the empty-clause list is non-empty, which is a single $O(1)$ operation. If neither of these conditions holds, then the interpretation should be extended again by assigning a truth value to some as-yet unassigned literal.

Algorithm 8 (State test functions)

Input: *A list of clauses and a state*
Output: *A boolean: the clause set has been satisfied*
function *finalStateSat(S,Q)* : ***returns boolean***
    −− satisfied if all clauses have been removed
    −− i.e., if all other categories are empty
  return ($Q.empty = \{\}$
    and $Q.l1Unit = \{\}$
    and $Q.l2Unit = \{\}$
    and $Q.l1NoL2 = \{\}$
    and $Q.l2Any = \{\}$)
end

Input: *A list of clauses and a state*
Output: *A boolean: the clause set has been contradicted*
function *finalStateUnsat(S,Q)* : ***returns boolean***
    −− contradicted if any clause has been emptied
  return ($Q.empty$ *neq* $\{\}$)
end

Input: *A list of clauses and a state*
Output: *One of the smallest clauses contradicted in the given state*
function *contradicted(S,Q)* : ***returns set***$\langle$*literal*$\rangle$
  assert *at least one clause has been reduced to []*
    −− Clauses themselves are not altered by simplification,
    −− only the state structure changes.
    −− So we can still see the non-simplified clauses here.
    return *smallest*($Q.empty$)
end

4.4 Memory Allocation

Because the prover uses several different smallish data structures in great number, the memory allocation routines provided in the C standard library caused some troubles. Those routines never group together small blocks of memory that have been de-allocated, so they become permanently unavailable for allocation to larger blocks. In some cases, the number of small objects reaches a peak that is so high that there is inadequate memory left for the number of larger objects that will be needed later, even if those smaller objects had been de-allocated in the meantime.

To address this, I wrote higher-level memory allocation routines that request a large block of memory from which to allocate smaller structures of a given type. Not all of the data types used in the prover get this treatment, just the ones that have a fixed size and are needed in great number. All such blocks are of equal size, so a block that was allocated for one structure and subsequently deallocated can be re-allocated for use by another structure.

Each data type has a free list, so allocations are $O(1)$ if there is space left in an allocated block. Similarly, de-allocations are $O(1)$.

On the other hand, allocating a new block entails some overhead. Each of the objects in the block must be added onto the free list for that type. Also, each block has a bitmap that must be initialized. That bitmap indicates which objects are currently in use at any time. It can be used to test relatively quickly if the block is entirely free. This is important because we might have more than one block allocated for a single data type, so the free list for that type might contain objects from different blocks. Deciding whether or not a block is entirely free based on the free list alone would have been very inefficient, because the entire list might have to be searched.

But updating this bitmap means being able to find which block an object belongs to. Early implementations did a sequential search of the blocks for the appropriate data type, comparing the address of the object with the bounds of each block. This was typically not bad, because the number of blocks for any type of object is not usually huge. Still, allocation and deallocation are frequent enough that it was worthwhile to optimize this. To accomplish this, a little extra memory is reserved for each object; enough to hold a pointer. That pointer refers to the block that contains the object. With that, an object can be deallocated in $O(1)$ time.

4.5 Summary

Because the algorithm constantly modifies small parts of a very large data structure, multiple-assignment semantics are called for. Given that, the challenge was to create data structures that allow very fast searches and updates. The two-dimensional linked list was important for this, but keeping ordinary one-dimensional linked lists of categorized clauses was very helpful as well, as was adjusting memory management to take advantage of Sttrim's memory usage characteristics.

Chapter 5

Theoretical Behavior and Experimental Results

The main reason for the development of Sttrim was the existence of ordinary-seeming problems that are pathological cases for Davis-Putnam. These cases are difficult precisely because they require Davis-Putnam to do a great deal of backtracking. Examination of some of these reveals that propagating a small amount of information forward could make such backtracking unnecessary. For instance, given a circuit and its specification, the statement that we will try to prove will normally consist of those, plus a statement to the effect that the inputs to both are the same, but the outputs are different. Of course, if the circuit does meet its specification, then that statement will be unsatisfiable; this is proof by refutation. The problem for Davis-Putnam occurs when it has no unit or pure-literal simplifications to do, and decides to split on a variable that represents one of the inputs to the circuit or its specification. Choosing either truth value for a circuit input immediately results in the corresponding specification input being assigned the same value, because one of the clauses specifying their equality gets reduced to a unit clause, which is then simplified away. The proof proceeds, with more variables being assigned values and clauses being reduced, until eventually the empty clause is derived signifying that the assignment was not a model. The prover must then backtrack and try other assignments for each place where a value was chosen by the splitting rule. Depending on the structure of the circuit and its spec, after the alternative assignment, the subsequent subproof can be identical to the first one, suggesting that re-doing all of that work is unnecessary.

One way to view the function of Sttrim, then, is that it repeatedly builds a kind of summary of a part of the statement, and uses that instead of the original part. This summary can carry the kind of information whose absence causes Davis-Putnam to backtrack. This might be as simple as was suggested above, that two literals have the same truth value. Or, since the information is just another set of clauses, it can be as complex as the situation warrants.

The clause set produced at each stage contains only literals that were "extra" in the selection process; the "chosen" literals are eliminated. So the size of that clause set is bounded by the number of extra literals. This is an exponential bound, so we expect the method to become expensive when the number of extra literals is large. The selection of literals makes some attempt to keep this bound small, but since we are solving the satisfiability problem, there has to be some pathological case.

Our clause sets are typically the result of some human design. Because of this, they typically exhibit locality and hierarchy. That is, the patterns of adjacency between literals are not random; there is a tendency for literals to share neighbors, and for any literal's neighbors to comprise only a small fraction of the total set. These same characteristics that enable a person to understand the relationships between parts of a system also tend to make it amenable to my method. In practical terms, this means that with each pass, we eliminate some literals from some adjacency lists, and also introduce some new ones. Because of locality, the number of newly adjacent literals will typically be small. If it is equal to the number of eliminated literals, then we have not made things any worse for future passes. Indeed, it is possible that future passes will become easier, as adjacency lists can shrink, resulting in fewer involved literals when those literals are selected for elimination.

As an optimization, we recognize the situation in which all variables will be involved no matter which we choose to eliminate. The simplest way that this can happen is for a single clause to contain every variable. In that case, we choose all variables for elimination, leaving no extras. When this happens, the method degenerates to Davis-Putnam; the entire semantic tree is $L1$, we do a depth-first search for a model, we use the same simplification rules as Davis-Putnam, and the first model found terminates the process.

We may be interested in knowing which structures Semantic Tree Trimming outperforms Davis-Putnam. This is more easily addressed by characterizing the performance of the two algorithms individually. Semantic Tree Trimming then outperforms Davis-Putnam simply on those problems which have two sets of properties - those which favor the former and those which hinder the latter.

Semantic Tree Trimming performs poorly on problems which force a large number of extra variables at each stage. As mentioned above, hierarchical and modular designs work to reduce this. But in addition to the simple absence of those properties, there is at least one simple characteristic that can cause Semantic Tree Trimming to exhibit pathological behavior: every variable occurs in some large clause. Then, no matter which variables are selected, a large number of variables become involved in the proof. With the current selection strategy, there is a limit on the number of variables that will be eliminated in each stage, so there is the potential for a large number of extra variables; even if a large number of variables are involved, only some small number of them will be eliminated.

Note that at each pass, we eliminate some variables by replacing a subset of the clause set with one that is equivalent, as far as satisfiability of the entire set is concerned. The newly generated clauses can have adjacencies that were not present in the original clause set. For instance, the original clause set may have had clauses $[A, C]$ and $[B, -C]$. After eliminating the variable C, we get the clause $[A, B]$. Literals A and B did not occur together in any clause in the original set, but they do occur together in the generated clause. So, even if the original clause set did not contain any variables having many neighbors, these can be created by the proof process. Since the prover is continually choosing variables and clauses to minimize the number of involved variables, the variables in these long clauses will not be chosen until there is no alternative, and they can therefore accumulate. However, this does not seem to happen often in practice, for reasons to be stated below.

One way of seeing how this may happen is to consider the batches of variables chosen for elimination as modules, and consider the adjacency patterns of those modules. Call the variables that are in one module and adjacent to those in another, *border* variables. With respect to a particular module, the *inside* border consists of variables in the module and adjacent to other modules; the *outside* border consists of variables in other modules that are adjacent to variables in this one. With respect to a pair of modules, the *interface* is the set of all variables in those modules that are adjacent to variables in the other.

Before eliminating some module, it might be that its adjacent modules are connected only indirectly by way of the module being eliminated. That is, it may be that there is no clause contains border variables from more than one adjacent module. In eliminating the module, clauses are generated that represent the relationship that it induced between its adjacent modules. Afterward, there may be clauses that contain all of its outside border variables, even if there were no such clauses before.

So, one way to prevent the accumulation of clauses of greater and greater length would be to make sure that the total number of outside border variables does not consistently exceed the number of interface variables for any of the adjacent modules. This is because the outside border variables may all be made adjacent with each other. But before elimination, clauses did not involve more than one interface. From the point of view of each adjacent module, the eliminated module's inside border variables have been replaced with the outside border variables of all other adjacent modules.

This special characteristic generally holds of linear structures. Each module has only two adjacent modules, so its elimination makes them adjacent. If the interface sizes are roughly the same, as one would expect in a good balanced partitioning, then the new interface between those two modules will be roughly the same size as each of the interfaces those had with the eliminated module. So the longest generated clause will not tend to increase as the proof proceeds and modules are eliminated.

In contrast, if eliminating nodes from a tree structure starting at the root, each node eliminated could increase the size of the interface with adjacent nodes. From a sibling node's point of view, its interface with its sibling has been replaced with an interface to both of that sibling's children. So the size of the interface, and the upper bound on clause lengths, can continuously increase as the proof proceeds.

Of course, my heuristics would normally prevent this from happening, because it chooses the literals with smallest degrees. So even if it eliminated the root node first, resulting in a few nodes with literals of higher degree, it would not immediately choose those higher-degree nodes to eliminate next. So the build-up of clause lengths would occur only if the input was contrived to make it unavoidable. The problem SYN091, which as mentioned earlier is the worst case among all the sample problems for sttrim, has exactly this kind of structure. It is a pair of superimposed lower-triangular diamond graphs with some extra edges. A diamond graph is basically a rectangular grid of nodes, in which each node has an edge to the node directly above it and another edge to the node directly to its left. In the case of SYN091, only the nodes in the lower triangle exist, and there are other complications as well; some of the edges in each triangle go to the corresponding node in the superimposed graph, rather than the node that they would go to in an ordinary diamond graph, and there are some edges from nodes within the lower triangles to nodes on the diagonals too.

The result is that the elimination of any node increases the degree not just of its parent node, but of its siblings as well, so that there are no isolated pieces that can be eliminated; there is no modularity.

The identification of structures on which Semantic Tree Trimming outperforms Davis-Putnam has another aspect: which does the latter have particular trouble with? Since the problem arises due to backtracking, clause sets in which the pure and unit simplification rules continuously apply are handled easily. These simplification rules are forms of inference. When there is nothing further that can be inferred about the truth value of any literal, Davis-Putnam resorts to backtracking. Call such a literal *independent*. Given a set of independent literals, a proof by Davis-Putnam amounts to enumerating all possibilities on these, with the inference of other dependent literals following efficiently as ever. In the domain of hardware verification, this has an important meaning: inputs to a circuit are independent. Given values for all inputs, we should be able to infer the values of all intermediate nodes and outputs, because the circuit is deterministic. This gives us a lower bound on the amount of backtracking Davis-Putnam will need.

The actual amount of backtracking could be greater though, for two reasons. One is that the circuit specification could be loose, resulting in independent literals that are not inputs to the circuit. The other is that Davis-Putnam might not pick circuit inputs first, in either the implementation or the specification part of the clause set. Since Davis-Putnam does not know anything about the circuit structure (indeed, it does not even know that its input represents a circuit!), it may choose some intermediate node to split on. Having assigned a value to some intermediate node, there still may not be enough information to infer values for the inputs, so those input values will still have to be enumerated by backtracking. Consider a simple AND gate. If the output is low, we do not have enough information to infer the values of either input, although we know at least one of them must have been low.

Furthermore, because clause form essentially removes causality from the relationship between inputs and outputs, automatically recognizing which literals represent inputs is not straightforward. Even more hindrance comes in the form of "environmental promises", which are statements that rule out illegal inputs, so that the proof does not find that the circuit broke its specification for some input that we know will never happen. With these additional statements, even the inputs to the circuit can appear to be driven by other clauses, though those clauses generally will express a complicated relationship between inputs rather than simply setting a fixed value for any input.

5.1 Input Problems

Sttrim was run on two major problem sets, plus a handful of problems that were generated during its construction. One of the problem sets was the propositional problems from the "SYN" category of TPTP [50]. The other major set of problems came from a test pattern generator [39], for particular kinds of hardware faults. Finally, a few circuit implementations and specifications were coded here at the University of North Carolina during the early construction of the prover, for proof of concept.

5.1.1 TPTP

The TPTP problem library contains both propositional and first-order problems [50]. Sttrim, of course, is a propositional theorem prover. However, first order problems which are ground can be treated as propositional.

Some of the problems in the TPTP library have "generators", which allow for the same basic problem to be instantiated with multiple sizes. Note that the TPTP documentation uses the term "size" as a parameter to the problem generator; it is not the length of the problem as one would use for complexity analysis. To avoid confusion, I will use the term "TPTP size" to mean the former, and reserve for the term "size" its more usual meaning of the number of symbols in the problem statement (or anything within a polynomial of this). This is necessary because some of the generated problems increase in size exponentially for each increment in TPTP-size. For instance, the number of clauses in instances of SYN001 is exactly 2^N, and each clause in that problem has length N, where N is the TPTP-size. Most problems, however, seem to be of size either linear or quadratic in their TPTP-sizes.

The problems tested were taken from the SYN category, simply because there are more propositional and ground problems there than in the remainder of the TPTP suite combined. Even more, although some of these problems are available only in a single size, most of them have generators. While the fixed-size problems provide some information about the performance of these provers, it is the variable-size problems that are of the greatest interest, because these give better insight about how that performance scales up on problems of similar structure. The SYN category is also interesting because many of the problems contained in that were created solely to break specific theorem-proving strategies. It could therefore be enlightening to consider which problems are handled easily and which are not, bearing in mind the purpose for which each of those was constructed.

5.1.2 UCSC

These problems came from Tracey Larrabee's test pattern generator [39] from problems in ISCAS '85 [8]. They were generated to check for particular kinds of wiring errors: "single-stuck-at" faults and "bridge" faults. These problems have names that begin with "ssa" and "bf", respectively.

Larrabee also has a satisfiability checker that solves all of these problems quickly, but it is rather specialized. Some of the ideas involved in that might be useful in a general-purpose theorem prover, but others would not.

Although these problems are interesting for being related to the intended domain, each one constitutes an essentially independent data point, telling nothing about the scalability of the algorithm.

5.1.3 UNC

These hardware-verification problems were generated during the construction of early versions of the prover, and are each an implementation and specification of a circuit along with a statement that the former implements the latter. Since they were constructed as part of the present work, they do not by themselves constitute a trustworthy measure of the prover's ability. However, aside from having the structure for which Davis-Putnam was known to have problems, they were not in adjusted in any way for any idiosyncrasy of Sttrim. So these results are still of interest.

Each of these problems was generated in varying sizes, to allow the exploration of how Sttrim's performance scales up. The closer in structure an input problem is to others, the more meaningful is a comparison of the results, because we know that the differences are more related to problem size than any oddity of the organization of the problem.

The first is a ripple-carry adder. The size is the number of bits in the adder. The implementation consists of some number of full adders chained together so that the carry output from each stage is added into the next. Each full adder is made up of eleven gates; five AND gates, five OR gates, and one inverter. Each full adder has three inputs and two outputs. The inputs consist of bits of the numbers being added, and a carry bit from the preceding stage. One output is the bit in the sum; the other is the carry output to the next stage. The formulas expressing this follow.

$$carry\_out = (x1 \wedge (x2 \vee x3)) \vee (x2 \wedge x3)$$

$$sum\_out = (x1 \wedge x2 \wedge x3) \vee ((x1 \vee x2 \vee x3) \wedge \neg carry\_out)$$

In English, the formula for *carry_out* means "any two inputs", while the formula for *sum_out* means "all three inputs, or any one input when there is no carry".

The specification is also structured as a chain, but each node consists of just the clauses that specify the behavior of the stage. This is roughly the same as having a truth table for each stage, where each row in a truth table is represented by a single clause.

Ideally, the specification and implementation would have had a much different structure, to avoid the danger of making the same mistake in both. One might even like the specification and implementation to be written by different people, to minimize the obvious danger of a single person repeating the same mistake in both. But in this case, the point was not to test the adder, but to test the theorem prover, and the situation did not permit the assignment of this task to different individuals. Therefore, both were written by the author, and they share the same high-level structure.

Next is a simple parity checker, which functions very much like the chain of exclusive-OR gates used as an example in section 3.2. The difference is that it takes one bit at a time over a series of timesteps, rather than taking all bits at once and propagating a signal through a series of exclusive-ORs. So each stage of the implementation is expressed as a multiplexor that selects either the output of the previous stage or its inverse, depending on the input value for that stage. Since it was a sequential design, it defines a delay element whose output at time N is the same as its input at time $N-1$. This provides the connection between the circuit at successive timesteps. The first stage is special, because it has no preceding stage from which to choose an output or inverse of that. Aside from that minor difficulty, the implementation is expressed by the following formula:

$$out = mux(in, not(prev(out)), prev(out))$$

The multiplexor "mux" has three inputs. If the first input is true, then the output is equal to the second input; otherwise the output is equal to the third input. The second and third inputs involve the output from the previous timestep, which means a single-timestep delay register.

As with the adder, the specification is a chain of specifications of the individual stages. Also as with the adder, the specification is much simpler than the implementation; in this case the specification has one fourth as many clauses as the implementation.

There is also an arbiter, adapted from the IFIP Hardware Verification Benchmark Suite [49]. This is a circuit which is intended to be a node in a binary tree. Each of its children may request some token, which it may grant to them. But it should never grant the token unless it has first been granted it by its parent, and it should never grant the token to both of its children at the same time.

Unlike the previous circuits, this one is parameterized over *time*. That is, we are not checking that the circuit of some size matches a specification, but that the circuit of fixed size matches its specification over some number of time steps. Given that we know how many bits of state the circuit has, we can choose a number of timesteps large enough that any state could be reached.

There are two kinds of properties we might like to verify in such a circuit: safety and liveness. These can be memorably phrased as making sure that nothing bad ever happens, and making sure that something good eventually happens, respectively. The former is easier, because it is sufficient to look at a single state to see that it is legal. However, liveness requires the examination of a history of states; failing to enter state B might be a problem only if we were in state A at some time in the past. Naturally, propositional logic is not powerful enough to express such relationships extended over arbitrary durations. In view of this theoretical limitation, I settled for verifying liveness within a given time bound. That is, we will choose some maximum number of time steps N within which the circuit must respond in order to be deemed correct. This depends on some knowledge of the circuit, and on having a definite meaning for a timestep.

The implementation consists of three fairly simple combinational circuits; one for each output. The logic for passing a grant on to one of the two children is slightly asymmetric to handle the case in which both children make their request at exactly the same time. Child 1 gets the grant if child 2 does not already have it, but child 2 gets the grant only if child 1 does not want it. Of course, once it has a grant, that continues until the child cancels its request. This is one of the correctness properties.

This logic is expressed by the following formulas:

$$\begin{aligned} req &= (req1 \vee req2) \wedge (\neg prev(gr) \vee prev(req)) \\ gr1 &= req1 \wedge req \wedge gr \wedge \neg(prev(req2) \wedge prev(gr2) \wedge req2) \\ gr2 &= req2 \wedge req \wedge gr \wedge \neg((\neg prev(req2) \vee \neg prev(gr2)) \wedge req1) \end{aligned}$$

The correctness properties were originally expressed in English, and some effort was required to state them unambiguously in terms of propositional logic. Its operation was defined by a four-phase protocol:

1. A node requests the token by setting req to true.
2. When gr becomes true, the node has the token and may pass it down the tree.
3. The token is handed back by setting req to false.
4. When gr becomes false, a new request can be made.

The objective was to verify that the arbiter grants access to at most one client, and that it does indeed follow the given protocol. This original version of the problem stated in the benchmark suite is just a safety specification. To verify liveness, four additional properties are tested:

1. After req1 or req2 becomes true, req becomes true.
2. After gr becomes true, gr1 or gr2 becomes true (corresponding to either req1 or req2, as appropriate).
3. After req1 and req2 become false, gr1 and gr2 become false.
4. After gr1 and gr2 become false, req becomes false.

All versions of the arbiter verification also include clauses that promise something about the behavior of the environment. This allows the proof to rule out cases in which the circuit breaks a correctness property in the presense of invalid inputs. These environmental promises correspond closely to the safety properties of the arbiter itself, because it was intended to be replicated as nodes in a tree, interacting with other instances of itself. So, for instance, phase three of the protocol indicates that the token is released by setting "req" to false. This means that until "req" is set to

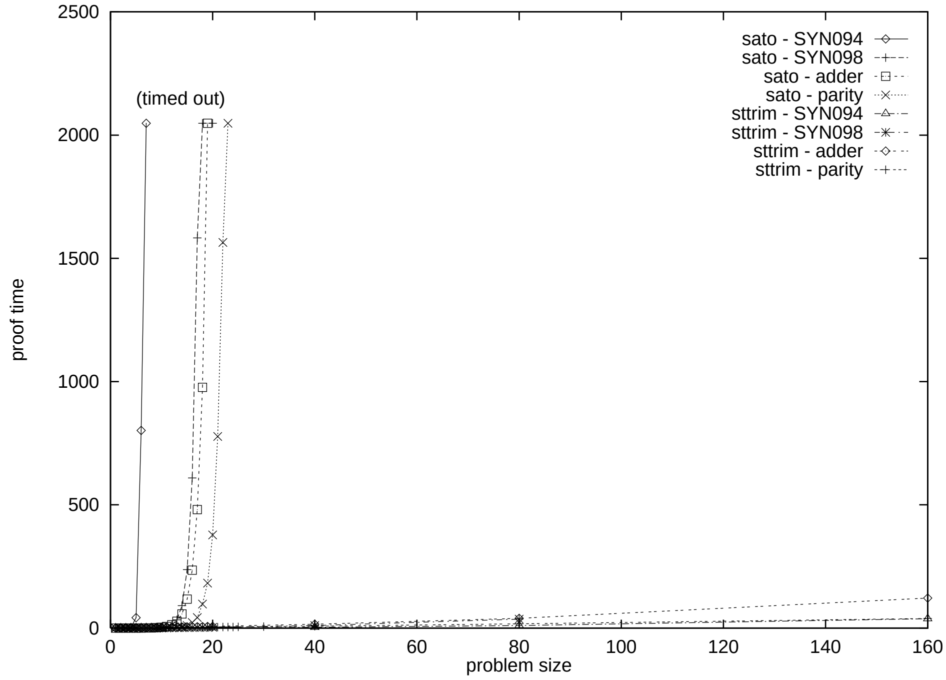


Figure 5.1: Some of the problems that show the best performance of Sttrim over Sato.

false, the token has not been handed back. So the environment must promise that once it has granted the token by raising “gr”, it must not to drop “gr” until we drop “req”. Otherwise the arbiter would be in the position, at least for some short time, of granting a token which it had lost.

5.2 Timing Results

The best cases for Sttrim relative to Sato appear to be the problems that are related to hardware verification: the “bf” and “ssa” problems from the UCSC set, and the hardware problems generated locally. Although Davis-Putnam (in the form of Sato) is much faster on some problems, it is much slower on far more of them. This is strong evidence that Sttrim is more suitable than Davis-Putnam for hardware verification problems. Figure 5.1 shows some of the most dramatic cases of improvement. Note that we are comparing against the version of Sato that is merely a very efficient implementation of Davis-Putnam; not against Sato3, which has a significant algorithmic enhancement.

The worst case for Sttrim is SYN091, which was one of a set of related problems developed to expose weaknesses in different theorem-proving strategies [45]. This particular instance is called $Sym(S_n^2)$, and is known to cause provers based on ordered-resolution to generate exponentially many clauses. From a high level perspective, Sttrim is quite similar to that kind of prover, in that it repeatedly removes literals, generating clauses that summarize the new relationships between the remaining ones. Of course, Sttrim should not suffer quite as badly as resolution-based provers because it is generating clauses only for maximally closed subtrees, rather than all possible resolvents. Still, it is apparent that this is a pathological case for Sttrim. Thankfully, its structure is contrived enough that while it exposes a theoretical weakness (which we knew had to be there), it seems unlikely to occur in practice, at least within the domain of hardware verification.

For a few problems, Sttrim ran out of memory. For a few others, Sato exceeded an internal table's size. The latter problem could be easily remedied by recompiling Sato with a higher preset limit, provided that the machine on which it is to run has enough memory. The former is a little more serious than that, but not a fatal flaw. As mentioned in the discussion, Sttrim actually allocates memory for an N by N adjacency table, and so requires N^2 space in the number of literals. While this allows for $O(1)$ queries, I could instead have used some kind of balanced B-tree, keeping the space requirements nearly linear, at the price of $O(\log N)$ time for queries. This extra cost is less significant for larger problem sizes. In addition, because Sttrim is a research tool, it is not yet as refined as one might hope, and at least some of the memory usage is known to be due to a failure to re-allocate memory that is no longer in use; the so-called "memory leaks".

We can see that most problems which pose the greatest difficulty for Sato are handled with relative ease by Sttrim. There are also some problems which are more difficult for Sttrim than for Sato, but the difference in most of these cases is far less dramatic.

Proofs were terminated after 2048 seconds if not completed by then. Figure 5.3 shows the distribution of times for the problems that were actually solved. It is apparent that extending the time limit by any small amount would not allow a significant additional number of proofs to complete. In fact, in retrospect, it seems that a much smaller time limit could have been adopted, resulting in a much lower cost in experimenting time and no significant change in results.

Appendix A lists the results numerically, showing some details of where Sttrim spent its time, and that it agrees with Sato on whether or not each input problem is satisfiable. Figure 5.5 shows the information graphically. There are three large

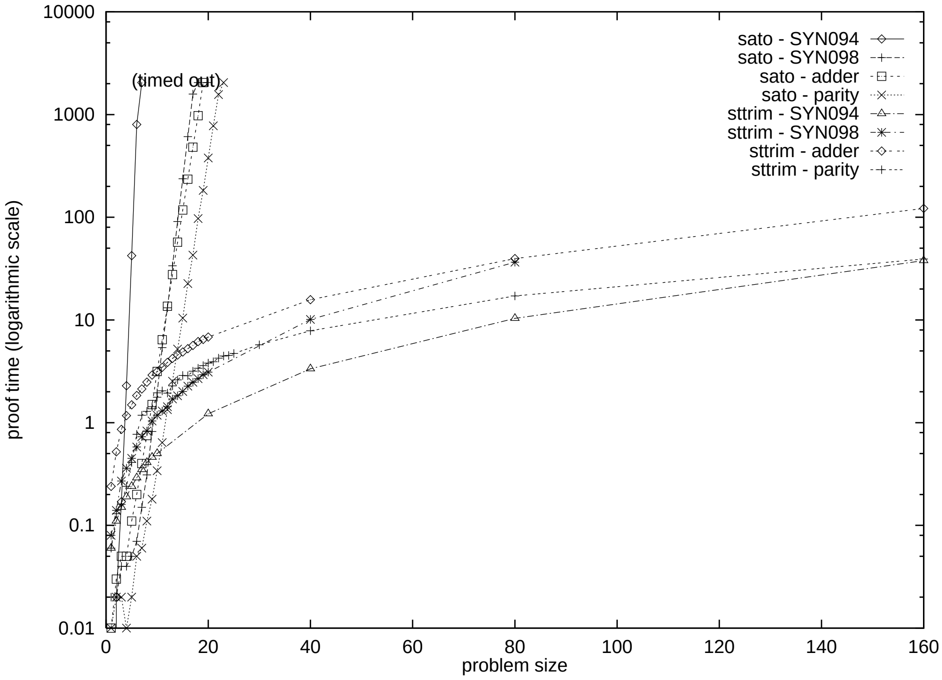


Figure 5.2: Some of the problems that show the best performance of Sttrim over Sato, with a logarithmic vertical scale. The straightness of the lines for Sato on the log scale indicates that those times are increasing exponentially.

clusters of problems in this graph. One cluster consists of problems that are solved easily by both provers, near the bottom-left of the graph. In this group, Sato generally found the proof first, with Sttrim not far behind. Another cluster appears at the top-left of the graph, showing a large number of problems on which Sato timed out, which Sttrim solved. The final cluster does not look like a cluster on the graph, because the data points all fall on exactly the same spot. Both Sato and Sttrim timed out on these problems. Since the time limit is the same in all cases, these all fall on the same location at the top-right of the graph, and appear as a single point. There are also a handful of problems which do not fall into any of these three groups, but their number is relatively small.

Figure 5.4 shows a summary of the information from Appendix A. The great advantage of Sttrim over Sato lies not in the cases in which one prover solved a problem more quickly than the other, but in those cases in which one prover completes the proof and the other does not. For each of the three major groups of problems, and for some finer subgroups, we see how many problems fall into each of the six categories.

The first three categories count problems on which at least one of the provers

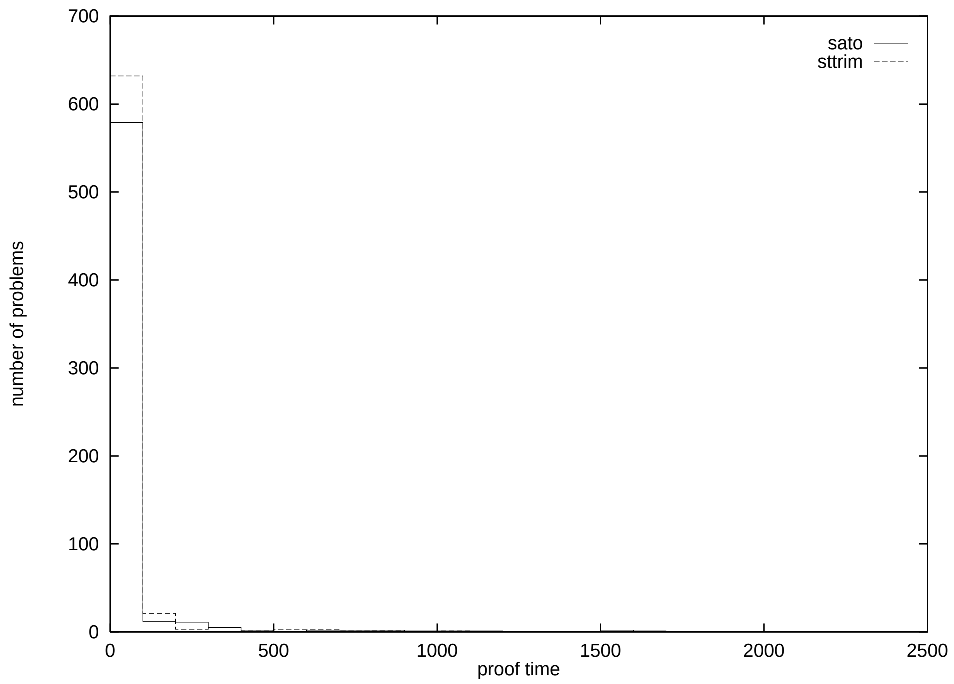


Figure 5.3: The distribution of completed proof times. The vast majority of proofs that completed did so in 100 seconds or less, and only a handful completed after 500 seconds. The large amount of blank space is an indication of how conservatively the upper time limit was chosen; the results would have been almost identical if the upper limit had been 100 seconds.

exceeded the time limit; either both provers did, or just Sttrim did, or just Sato did. Without knowing how long the proof would have taken, it is impossible to do a mathematical comparison, which is the reason these categories exist. Although they are not strictly time-outs, the few cases in which a prover terminated because of some kind of error are counted as such.

The next three categories count problems on which both provers completed, and attempt to distinguish between times that were roughly the same, and those which showed one prover clearly outperforming the other.

The criteria chosen for "clearly outperforming" is that one prover takes more than ten times longer than the other, and the difference is thirty seconds or more. This effectively forces most problems to be a tie, and indeed, most problems are solved by both provers in less than one minute. This is obviously a subjective criteria; for a view that is less amenable to bias, see Figure 5.5 or the numbers on which these totals are based.

The above subjective criteria is not as important as it might first seem, because

group	both time out	Sttrim times out	Sato times out	roughly equal	Sttrim wins	Sato wins
UNC	4	0	47	55	13	0
adder	0	0	5	13	5	0
parity	0	0	7	16	6	0
arb	4	0	35	26	2	0
UCSC	1	0	224	42	13	45
bf	1	0	181	28	6	7
ssa	0	0	43	14	7	38
TPTP	8	4	32	197	10	7
ALL	13	4	303	294	36	52

Figure 5.4: Summary of timing results.

the greater number of problems fall into the first three categories. That is, one prover or the other fails to find an answer even after quite a long period of time (2048 seconds). Again, Figure 5.5 shows the pattern more clearly.

Appendix B show the results with several different settings of control parameters. For Sato3, larger lemma sizes extend its reach, allowing it to solve more problems, at the cost of being generally slower; smaller sizes generally make it faster at the cost of solving fewer problems. Overall, the default setting seems best. For Sttrim, there are several parameters to adjust. Each of these seems to help with some parts of the proof, but also hinders other parts more than enough to compensate. So, as with Sato3, the default settings seem best overall. In some future version of Sttrim, those settings might be adjusted dynamically as the proof proceeds.

There are some cases in which Sttrim clearly beats Davis-Putnam even with lemma generation, as in Sato3. This indicates that the two techniques are in some ways complementary. We therefore expect that a prover incorporating both ideas would be superior to either of its predecessors. The question of why Sttrim wins on these cases actually turns out to be more a question of why Sato3 loses; it is Sato3's behavior in these instances that is atypical, judging by performance on related problems. Some of these problems were pathological for Davis-Putnam to begin with, and lemma generation did not change that (cf. the arbiter). Another was actually caused by lemma generation itself (SYN098). In either case, we expect Sttrim's characteristic

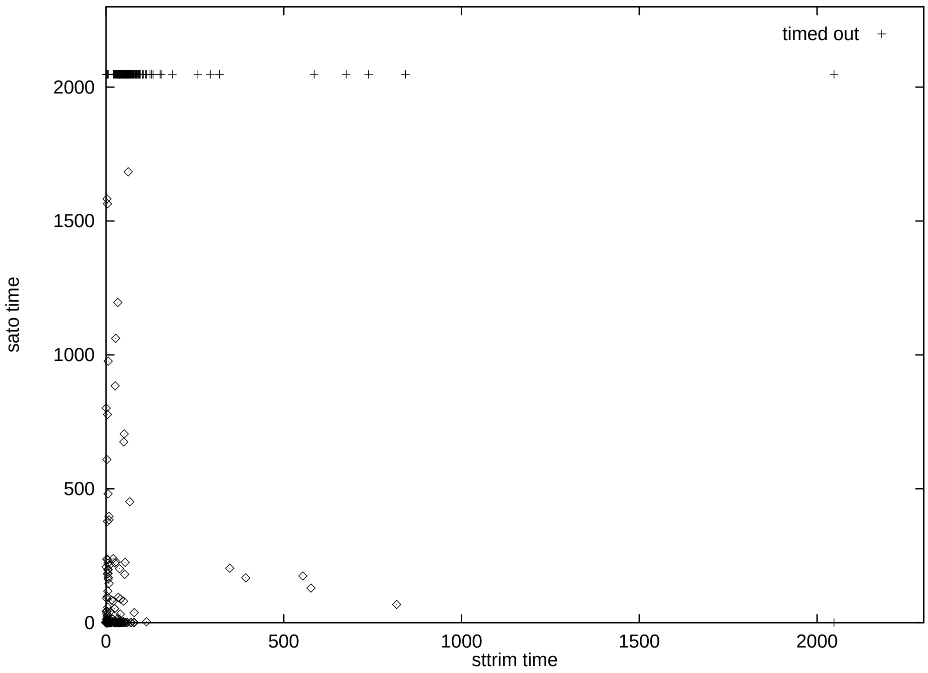


Figure 5.5: A comparison of times taken by Sttrim and Sato. The time limit was 2048 seconds, and proofs that were cut off at that limit are marked by a different symbol than those that completed.

of operating on small semantic trees to ameliorate the problem, first by reducing the maximum possible number and size of lemmas, and second by being capable of handling the pathological cases that lemma generation does not. Since lemma generation could possibly benefit Sttrim as much as it did Davis-Putnam, I have chosen to evaluate the performance of Sttrim (currently without lemma generation) relative to Davis-Putnam without that.

It should be noted that, except in the last step as previously discussed, and except for one of the simplifications that happens between passes, Sttrim is generating enough information to find all models, rather than just one. This may or may not be necessary in different situations, but in those cases where all models are necessary, Davis-Putnam becomes inefficient almost to the point of being impractical. This is because one of its two simplification rules, the pure literal rule, is no longer of any use, and the search for models becomes akin to an enumeration of all possible truth assignments. It is not quite as bad as such an enumeration, because the other simplification rule, the unit clause rule, still operates to reduce the number of truth assignments which must be explored. Even Sato shows this difficulty on satisfiable input problems if it is not told to stop after finding the first model. However, since

Sttrim's model finder is not yet implemented, it would not be entirely fair to make Sato generate all models. So the numbers reported are for Sato seeking at most one model.

It is clear that Davis-Putnam with lemma generation is the overall best of the three techniques, winning by a fair margin in almost all cases. Semantic Tree Trimming comes in second place overall, beating Sato3 in a few cases, and losing dramatically in a few, but doing respectably overall. Davis-Putnam without lemma generation is good in many cases, even beating semantic tree trimming in quite a few, but fails spectacularly on so many problems as to make it infeasible for use in real work. It remains to be seen how Semantic Tree Trimming with lemma generation will fare, when that is added.

While Figure 5.5 might lead one to believe that Davis-Putnam's performance is bimodal, doing very well on many problems and very poorly on many others with very few in between, this is quite likely misleading. Because they were stopped at the time limit, we do not know how long those timed-out problems would have taken to complete. It is likely that they would have been spread out along the time axis, with some taking much longer than others.

5.3 Summary

On input problems with a linear structure, there are theoretical reasons that Semantic Tree Trimming should exhibit drastically better behavior than Davis-Putnam. The theory is born out by experimental results, in which Davis-Putnam shows exponential performance on these problems, while Semantic Tree Trimming shows nearly linear performance on them. Furthermore, suites of test problems that were not chosen for having any particular structure show that Semantic Tree Trimming's advantage is also apparent on problems which do not have exactly that structure, and in most cases it is competitive with, and often much better than, Davis-Putnam.

Davis-Putnam, when modified to use lemma generation as in Sato3, greatly outperforms Semantic Tree Trimming in most cases. This feature could also be added to Semantic Tree Trimming, but it is unclear whether it would bring such a drastic improvement.

Chapter 6

Future Work

There are many obvious paths to follow from here. I regard it as a strength of the algorithm that there are so many modular parts that could be independently adjusted. In the current implementation, each was chosen after some amount of careful thought, and in some cases some experimentation. But the amount of time and effort devoted to each part was necessarily limited. As such, many of the heuristics chosen could likely be improved upon. Also, because of the domain, there must necessarily be some pathological cases which could motivate further refinement. Here I enumerate some of the possibilities.

6.1 Selecting Literals to Eliminate

We might be able to do a good deal better than the simple minimum-involved-literals rule when choosing literals to eliminate. For instance, we might adjust the limit on how many literals we can involve, if it keeps turning out that we get only one literal per batch (as happens in the later stages of many proofs). There is the danger of having twice as large a space to explore with each additional literal that we allow, but there could be times when we could eliminate more than twice as many literals by doing so. Although we have a lower limit on how many literals must be eliminated, it poses a problem because meeting this lower limit often means involving a very large number of extra literals, making the proof expensive. Ideally, we would have some control allowing the elimination of more literals only if it does not become too expensive.

We might also choose a different arbitrary bound on the number of involved literals. The current implementation allows for setting this at the command line, but it might be beneficial to have the program choose for itself based on some characteristic of the input problem, such as average (or mode) clause length. The current setting of ten seems to work well in practice, but more experimentation could be done to clarify if this is really the best fixed setting. It does seem clear that the bound should not normally exceed, say, thirty, because a single pass involving so many literals can take more time than most test runs have taken for the entire proof. Of course, in unusual circumstances, such a high bound might be both feasible and desirable.

Another interesting possibility is that of estimating the future impact of choosing any particular literal or set of literals. Up to this point, we have considered only the impact on the pass about to be executed, with the intention of making it small. But we might be able to invest a small amount of work in ranking the amount of "damage" done to future passes, based on the changes to adjacency lists that are likely to result from a literal's elimination. If we can find a set of literals that is larger than the new adjacencies that are introduced by its selection, then future passes involving those newly-adjacent literals are likely to be easier than they would have been had we not eliminated these first. For instance, if literals A and C have the same neighbors, among which is some literal B, and if B's neighbors are a subset of A's and C's, then eliminating B would reduce the degree of both A and C. (If B had some neighbors that were not included in A's and C's, then eliminating B could result in the degree of A or C increasing.) Even when comparing choices that are all beneficial in this respect (or all harmful), we might get a ranking that allows us to make the best choice in the long-run. It might even turn out to be appropriate to give this enough weight in the decision process that the present pass is sacrificed for future gain; it is far from clear that the greedy solution that we have been using all along is necessarily the best.

6.2 Data Structures

Although it seemed like a good idea at the time, in retrospect it is clear that some kind of self-balancing tree should have been used in place of linked arrays, especially for adjacency lists. The adjacency lists were simple to program, and are obviously extremely fast for the expected access patterns. However, they do require $O(N)$ space, and therefore $O(N)$ time to initialize; since there are $O(N^2)$ adjacencies to record, this amounts to $O(N^2)$ time and space for the algorithm, even in the typical case. Using trees would certainly slow down each access from $O(1)$ to $O(\log N)$, but would use only $O(N)$ time and space to initialize; and so only $O(\log^2 N)$ for the $O(N^2)$ adjacencies. Since memory usage seems to be the hardest limitation on the prover as it is now, this would clearly be a win for large problems.

Sttrim's evolution has been continually toward data structures which require incremental maintenance; and away from those which were simpler but needed to be periodically rebuilt from scratch. There is obviously a trade-off here, because there are input problems for which the data structure maintenance does not pay off. But on the whole, the performance on hard problems has improved with each such change. So it seems likely that more of the same is called for.

6.3 Clause Splitting

Some pathological cases might consist *solely* of long clauses. In those cases, it might be possible to split them into smaller ones by the introduction of new variables. Since those new variables would each be adjacent to many other literals, we would not expect them to be chosen for elimination in any early round. But it might be difficult, or perhaps even impossible, to do a good split; one that introduces a minimum of new variables while actually reducing all adjacency lists to be within acceptable limits. This is especially true because we are likely to need one new variable for each clause that gets split.

6.4 Lemma Generation

Sato has demonstrated the value of lemma generation in Davis-Putnam. This might apply in an obvious way to Sttrim when it is traversing the $L1$ portion of the semantic tree, because this is quite similar to Davis-Putnam. But less obviously, we could also apply the same idea to in the $L2$ portion of the tree, by generating new clauses and adjusting the existing transition rules. This might have much the same effect in Sttrim that it did in Sato, which was tremendous.

6.5 Analysis

Sttrim cannot be good on all input problems. Although it seems to behave well for a very large range of inputs, there are some which cause it troubles. There is a clear need for a study of the cases that are pathological. In the same way that an analysis of Davis-Putnam lead to Sttrim, there is the expectation that further work will lead to even greater refinements, perhaps until the only remaining pathological cases are obviously contrived.

6.6 Selecting Used Literals

In the process of simplifying a selected batch of clauses, a semantic tree is traversed and maximally closed subtrees are recorded. But there are, in general, closed subtrees that are not maximal; their siblings are also closed. At the lowest level, a subtree is closed not because its subtrees are both closed, but because there is a clause that contradicts the interpretation represented by the root node of that subtree. In some cases, there may be more than one such clause. The question then arises: which literals were used in closing this subtree?

This is potentially important, because this subtree's sibling may also be closed, and if it is, there is some set of literals responsible for closing it. When our subtree and its sibling are both closed, neither is maximal, their parent node is also closed, and we merge the two sets of literals used in closing the subtrees to get the set that we say was used to close their parent node.

The existing implementation is greedy; it takes the smallest clause that closed a node, and its literals are the ones that we say were used. But this might sometimes be a bad choice; the sibling tree might be closed because of a set that is identical with another, longer, clause that closed this node. In such a case, the set of literals used to close their parent node would be smaller had the longer clause been selected as the used set.

Unfortunately, no efficient and provably best algorithm exists for dealing with this problem; it amounts to selecting a minimum covering set of the literals for all closed descendent nodes of the maximally closed node. Still, there may well be a better heuristic choice than the greedy one.

6.7 Finding Models

As mentioned earlier, the model-finding portion of the algorithm remains unimplemented. This should be straightforward to do, and is of great value.

If the process terminates with no clauses left in the set, then the original clause set is satisfiable. However, this by itself has not produced a model. A model is greatly desirable, of course, but the part of the algorithm to find one has not yet been implemented. Nevertheless, it is possible to do this, as follows.

At each pass, we simplified away some small number of literals. In doing so, we reduced the depth of the semantic tree, leaving open paths in the smaller tree wherever there were open paths in the larger one. The logic is that we can then ignore those eliminated literals, because for every one of the paths that we left open through the remainder of the semantic tree, there was some extension of it that was open through the entire tree; we have not altered the existence (or non-existence) of a model. So, we may have quite a number of partial models that look good after each pass. Furthermore, the partial models of any two passes may have some variables in common. The trick is to find a set of such partial models that is consistent, choosing one from each pass. That is, if the partial model from one pass includes literal A, and the partial model from another includes $\neg A$, then these two partial models cannot be combined; they are inconsistent. But there is at least one set of partial models, selecting one from each pass, that are mutually consistent. To facilitate finding this, we collect all of the partial models from each pass. They could be written into a file, allowing the model finding program to be entirely independent of the core of the prover, and thus avoid the use of memory space for all those partial models during the proof.

To find a model of the entire clause set, we look first at the final pass, and the partial model that we found for it. Since each pass leaves open only those paths that can be extended, any path through the final batch of clauses can be part of some path through the entire clause set. If it were not consistent with some other partial model, it would have been closed. We now repeatedly extend the model back by looking at the results of each pass in reverse order of how they were generated. For each pass, we have a list of partial models that include some of the literals in the model we are extending, and some new literals. We must choose one of these that agrees with the model we are extending for all literals that they have in common. That is, if the model that we are extending contains A, then we cannot choose a partial model from its preceding pass that includes $-A$. For exactly the same reasons as before, we know that this newly extended model can be extended to a complete model of the initial set of clauses.

Note that the final pass degenerates to Davis-Putnam, because all literals are chosen for elimination. So there is really no great amount of searching to be done; we can find a single model in time linear in the size of the partial models. Alternatively, we can find a number of models by growing a tree of them, starting with the partial model from the final pass as the root, and finding *all* compatible partial models generated by each preceding pass.

Note that this has still not found all models, because that final pass terminated as soon as it found any single model. We could find all models, by making that final pass continue searching until it had found all partial models for the final batch of clauses, and form a forest of models using those as roots. Note that only the final pass must be altered to allow for this. All preceding passes would still backtrack completely out of the $L1$ part of the tree whenever a model was found for that. Only in the final pass, in which the entire tree is $L1$, and only if we wanted to find all models, would we continue traversing the tree in $L1$ after finding a model there. This should not make much of a difference in most proofs, because it is only the final pass that must behave differently, to continue after finding its first model.

There are (at least) two obvious ways to go about producing a list of partial models for the sets of clauses that were produced at each step of the satisfiability proof.

The naive approach would be to build and traverse another semantic tree. This time, instead of the relatively complicated two-level rule which we used when deciding that the clause set was satisfiable, we use an algorithm much more similar to Davis-Putnam, but we do not let it terminate as soon as it finds a model; it has to continue to find all models. One complication is that the Davis-Putnam simplification rules are geared towards finding any model as quickly as possible, so they make some assumptions during the search that may make it miss some models. Of course, this is acceptable because the goal of Davis-Putnam is not to find all models, but just to decide whether or not one exists. But for our present purposes, such assumptions no longer hold true.

In particular, the pure-literal rule cannot be used in our situation. This rule is the one that notices that a literal always occurs with one sign and never with the other. When Davis-Putnam is searching for just any model, it is safe to let pure literals become true, because any model of those clauses in which the pure literal was set to false would still be a model if that literal was made true. So if there is a model to be had, it can be found with that literal true, and we need not bother with the alternative.

The unit-clause rule still applies when finding all models, because no model can possibly be overlooked by the application of this rule. Similarly, the splitting rule still applies, with the caveat noted above, that we cannot terminate the search as soon as some single model is found; we must traverse the entire tree.

Davis-Putnam exhibits remarkably good behavior in large part due to its ability to zoom in on a model, and terminate as soon as one is found. In our quest to find *all* models rather than just a single one, we have reduced the efficiency of that. But, just as during the satisfiability proof itself, the structure of the problem works to mitigate the damage.

First, the number of literals involved in each pass was kept to a minimum during the satisfiability proof, for precisely the same reason that we would like it to be small now: working on a number of shallow trees rather than one deep one means bypassing most of the exponential growth of the number of nodes to visit. Two trees of depth five have many fewer nodes than a single tree of depth ten.

Second, the depths of the trees traversed during model finding will be even less than those traversed during the proof. The literals that were eliminated during each pass of the proof contributed to the depth of the tree then, but are gone by the time we begin searching for models.

Still, the naive approach involves traversing a tree that we have already been through. Rather than repeat that work, the fancier approach adjusts the proof algorithm to to save more information than just the generated clauses at each step. Whenever we find a maximally closed subtree during the proof, we know that its sibling has at least one open path — a model. In fact, this is not all that we know. Not only does the proof algorithm recognize maximally closed subtrees, but it also recognizes their dual: completely open subtrees. With a minor modification, it could easily generate not only its list of clauses that closed those maximal subtrees, but also a list of maximal open subtrees, which represent the models we are after. This completely bypasses the need for any sort of post-proof tree traversal to find models, at the cost of making the proof algorithm slightly more complicated.

6.8 Summary

Because the implementation, and even the algorithm itself, are modular, there are many obvious avenues for exploration, adjusting the various parts. The recent success of lemma generation in Davis-Putnam is also a motivation for some work that should be straightforward. But perhaps the most interesting path lies in identifying and overcoming the most inconvenient of Sttrim's pathologies.

Appendix A

Detailed Timing Results

In the following tables are presented the results of running two theorem provers on a large number of input problems. Also presented are some characteristics of each input problem: the number of clauses, the number of distinct literals, and the total length of the problem. The length is the sum of the number of literals in all clauses. If any result is not available for whatever reason, its entry in the table is left with "N/A". The circumstances which can cause this will be described below.

For each prover, if a result was found (satisfiable or not), it is shown in the column labeled "sat?". Results might not be found if the prover fails to complete, which can happen for a variety of reasons. In any of these circumstances, the timing results for that prover are not available ("N/A", as mentioned above). The most common reason is that the prover was still working when the preset time limit of 2048 seconds was reached. Other possibilities are that the prover used all available memory, or encountered some kind of internal error which prevented its running to completion. All of these situations are indicated by something other than a simple number in the column that would normally show the total proof time. For instance, ">2048" in that column indicates that the prover exceeded the time limit and was terminated, while ">mem" (Sttrim) or ">atoms" (Sato) means that the prover used more memory than was available to it. (These differ because Sato uses an internal table size that is fixed at compilation time, while Sttrim continues to request memory until the memory allocation routine fails.)

Timing results are also shown. For Sato, only the total time to find the proof is available. For Sttrim, five measures are available, showing the time used in different aspects of the proof, as follows. The column labeled "simp" shows the amount of time used by the initial unit-clause simplification. If this simplification removes all clauses, then the problem is satisfiable, and all other time fields will be left set to "N/A". "Init" shows the time used to initialize Sttrim's internal data structures, in preparation for repeatedly choosing and eliminating variables. "Check" shows the amount of time spent in checking for trivial models. One such check is run after simplification and before initialization. Note that if it detects a trivial model at that point, then the proof is complete, initialization never occurs, and no "init" time will be available. "Triv" shows the total proof time for proofs which were terminated by the discovery of a trivial model. If the proof is terminated either by finding that the problem is unsatisfiable, or that it is satisfiable because all clauses were eventually removed, then no trivial model is ever found, and the "triv" column is left with "N/A". Finally, the "full" column shows the total proof time. Due to an oddity of the current implementation, for proofs which are completed by the discovery of a trivial model after the initialization of the data structures, such as most sizes of SYN091, the "full" column is left set to "N/A". As with all proofs which are found by trivial model, the total time is shown in the "triv" column, so this omission is minor.

					Sttrim						Sato	
	circuit				time (sec.)						time	
problem	size	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
adder	001	48	17	126	0.00	0.05	0.00	N/A	0.24	F	0.01	F
	002	92	32	244	0.00	0.07	0.00	N/A	0.52	F	0.03	F
	003	136	47	362	0.00	0.10	0.00	N/A	0.86	F	0.05	F
	004	180	62	480	0.00	0.11	0.01	N/A	1.17	F	0.05	F
	005	224	77	598	0.00	0.13	0.02	N/A	1.49	F	0.11	F
	006	268	92	716	0.00	0.15	0.02	N/A	1.84	F	0.20	F
	007	312	107	834	0.00	0.17	0.01	N/A	2.13	F	0.40	F
	008	356	122	952	0.00	0.21	0.02	N/A	2.48	F	0.75	F
	009	400	137	1070	0.00	0.25	0.04	N/A	2.91	F	1.50	F
	010	444	152	1188	0.00	0.27	0.06	N/A	3.14	F	3.15	F
	011	488	167	1306	0.01	0.29	0.04	N/A	3.46	F	6.42	F
	012	532	182	1424	0.00	0.36	0.06	N/A	3.86	F	13.59	F
	013	576	197	1542	0.01	0.37	0.05	N/A	4.22	F	27.63	F
	014	620	212	1660	0.01	0.42	0.07	N/A	4.55	F	57.21	F
	015	664	227	1778	0.01	0.46	0.12	N/A	4.87	F	117.53	F
	016	708	242	1896	0.01	0.50	0.10	N/A	5.25	F	234.83	F
	017	752	257	2014	0.01	0.56	0.11	N/A	5.64	F	480.94	F
	018	796	272	2132	0.01	0.62	0.13	N/A	6.15	F	976.49	F
	019	840	287	2250	0.01	0.65	0.15	N/A	6.46	F	>2048	N/A
	020	884	302	2368	0.02	0.70	0.19	N/A	6.82	F	N/A	N/A
	040	1764	602	4728	0.02	2.22	0.75	N/A	15.75	F	N/A	N/A
	080	3524	1202	9448	0.05	7.51	3.16	N/A	39.60	F	N/A	N/A
	160	7044	2402	18888	0.11	28.90	15.00	N/A	121.91	F	N/A	N/A

					Sttrim						Sato	
	circuit				time (sec.)						time	
problem	size	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
parity	001	39	19	91	0.00	0.04	0.00	N/A	0.06	F	0.01	F
	002	61	29	145	0.00	0.06	0.00	N/A	0.13	F	0.02	F
	003	83	39	199	0.00	0.06	0.00	N/A	0.16	F	0.02	F
	004	105	49	253	0.00	0.07	0.00	N/A	0.24	F	0.01	F
	005	127	59	307	0.00	0.07	0.00	N/A	0.41	F	0.02	F
	006	149	69	361	0.00	0.09	0.00	N/A	0.77	F	0.05	F
	007	171	79	415	0.00	0.09	0.00	N/A	1.18	F	0.06	F
	008	193	89	469	0.00	0.09	0.00	N/A	1.29	F	0.11	F
	009	215	99	523	0.01	0.10	0.00	N/A	1.43	F	0.18	F
	010	237	109	577	0.01	0.09	0.00	N/A	1.77	F	0.34	F
	011	259	119	631	0.01	0.12	0.00	N/A	2.04	F	0.64	F
	012	281	129	685	0.01	0.12	0.00	N/A	1.94	F	1.34	F
	013	303	139	739	0.01	0.14	0.00	N/A	2.28	F	2.53	F
	014	325	149	793	0.01	0.14	0.01	N/A	2.61	F	5.21	F
	015	347	159	847	0.02	0.15	0.00	N/A	2.88	F	10.44	F
	016	369	169	901	0.02	0.16	0.00	N/A	2.86	F	22.60	F
	017	391	179	955	0.02	0.14	0.01	N/A	3.17	F	42.86	F
	018	413	189	1009	0.02	0.19	0.02	N/A	3.39	F	97.24	F
	019	435	199	1063	0.02	0.21	0.02	N/A	3.59	F	182.51	F
	020	457	209	1117	0.02	0.22	0.00	N/A	3.81	F	378.06	F
	021	479	219	1171	0.02	0.23	0.02	N/A	3.92	F	777.22	F
	022	501	229	1225	0.02	0.23	0.02	N/A	4.25	F	1564.28	F
	023	523	239	1279	0.02	0.27	0.01	N/A	4.46	F	>2048	N/A
	024	545	249	1333	0.02	0.27	0.02	N/A	4.51	F	N/A	N/A
	025	567	259	1387	0.02	0.28	0.04	N/A	4.72	F	N/A	N/A
	030	677	309	1657	0.02	0.36	0.03	N/A	5.73	F	N/A	N/A
	040	897	409	2197	0.04	0.55	0.04	N/A	7.84	F	N/A	N/A
	080	1777	809	4357	0.09	1.74	0.25	N/A	17.14	F	N/A	N/A
	160	3537	1609	8677	0.19	5.83	0.96	N/A	39.22	F	N/A	N/A

					Sttrim						Sato	
	circuit				time (sec.)						time	
problem	size	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
arb2	001	108	41	229	0.00	0.04	0.00	N/A	0.36	F	0.03	F
	002	192	70	422	0.01	0.09	0.01	N/A	1.88	F	0.04	F
	003	276	99	615	0.01	0.15	0.00	N/A	3.96	F	0.07	F
	004	360	128	808	0.01	0.21	0.03	N/A	6.62	F	0.23	F
	005	444	157	1001	0.01	0.28	0.04	N/A	12.03	F	0.78	F
	006	528	186	1194	0.01	0.33	0.04	N/A	17.49	F	3.08	F
	007	612	215	1387	0.02	0.43	0.09	N/A	18.59	F	12.87	F
	008	696	244	1580	0.02	0.53	0.12	N/A	24.26	F	52.43	F
	009	780	273	1773	0.02	0.62	0.11	N/A	25.22	F	222.50	F
	010	864	302	1966	0.02	0.76	0.17	N/A	27.14	F	1061.84	F
	011	948	331	2159	0.03	0.87	0.23	N/A	35.84	F	>2048	N/A
	012	1032	360	2352	0.03	1.00	0.27	N/A	37.15	F	N/A	N/A
	013	1116	389	2545	0.03	1.10	0.32	N/A	41.47	F	N/A	N/A
	014	1200	418	2738	0.04	1.26	0.31	N/A	55.17	F	N/A	N/A
	015	1284	447	2931	0.04	1.52	0.34	N/A	52.72	F	N/A	N/A
	016	1368	476	3124	0.04	1.59	0.37	N/A	54.77	F	N/A	N/A
	017	1452	505	3317	0.05	1.74	0.42	N/A	56.64	F	N/A	N/A
	018	1536	534	3510	0.05	1.95	0.50	N/A	62.85	F	N/A	N/A
	019	1620	563	3703	0.05	2.16	0.49	N/A	68.56	F	N/A	N/A
	020	1704	592	3896	0.05	2.29	0.58	N/A	80.49	F	N/A	N/A
	040	3384	1172	7756	0.12	8.22	2.41	N/A	187.72	F	N/A	N/A
	080	6744	2332	15476	N/A	N/A	N/A	N/A	>mem	N/A	N/A	N/A
	160	13464	4652	30916	N/A	N/A	N/A	N/A	>mem	N/A	N/A	N/A

					Sttrim						Sato	
	circuit				time (sec.)						time	
problem	size	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
arb2fp	001	111	44	234	0.00	0.07	0.00	N/A	0.37	F	0.02	F
	002	196	75	430	0.00	0.10	0.02	N/A	1.94	F	0.03	F
	003	281	106	626	0.01	0.16	0.02	N/A	3.73	F	0.07	F
	004	366	137	822	0.01	0.22	0.03	N/A	7.20	F	0.23	F
	005	451	168	1018	0.01	0.27	0.05	N/A	10.89	F	0.75	F
	006	536	199	1214	0.02	0.35	0.07	N/A	13.75	F	3.16	F
	007	621	230	1410	0.02	0.44	0.11	N/A	18.23	F	13.34	F
	008	706	261	1606	0.02	0.52	0.16	N/A	23.90	F	52.26	F
	009	791	292	1802	0.02	0.66	0.17	N/A	28.10	F	227.01	F
	010	876	323	1998	0.02	0.77	0.23	N/A	33.19	F	1195.61	F
	011	961	354	2194	0.03	0.89	0.29	N/A	39.49	F	>2048	N/A
	012	1046	385	2390	0.03	1.01	0.35	N/A	44.10	F	N/A	N/A
	013	1131	416	2586	0.04	1.17	0.43	N/A	48.34	F	N/A	N/A
	014	1216	447	2782	0.03	1.32	0.49	N/A	50.79	F	N/A	N/A
	015	1301	478	2978	0.04	1.49	0.58	N/A	55.07	F	N/A	N/A
	016	1386	509	3174	0.05	1.68	0.59	N/A	60.70	F	N/A	N/A
	017	1471	540	3370	0.05	1.85	0.75	N/A	65.93	F	N/A	N/A
	018	1556	571	3566	0.05	2.01	0.87	N/A	76.33	F	N/A	N/A
	019	1641	602	3762	0.05	2.29	1.05	N/A	74.74	F	N/A	N/A
	020	1726	633	3958	0.05	2.47	1.15	N/A	86.94	F	N/A	N/A
	040	3426	1253	7878	0.12	8.44	4.69	N/A	195.26	F	N/A	N/A
	080	6826	2493	15718	N/A	N/A	N/A	N/A	>mem	N/A	N/A	N/A
	160	13626	4973	31398	N/A	N/A	N/A	N/A	>mem	N/A	N/A	N/A

					Sttrim						Sato	
	circuit				time (sec.)						time	
problem	size	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
arb2lp	005:001	650	284	1412	0.04	0.18	0.01	N/A	0.87	F	0.07	F
	005:002	738	338	1580	0.03	0.63	0.07	N/A	18.92	F	2.74	F
	005:003	826	392	1748	0.04	0.70	0.07	N/A	20.32	F	2.85	F
	005:004	914	446	1916	0.05	0.78	0.10	N/A	21.65	F	3.06	F

					Sttrim						Sato	
	circuit				time (sec.)						time	
problem	size	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
arb2lp	005:004	914	446	1916	0.05	0.78	0.10	N/A	21.65	F	3.06	F
	006:004	1084	524	2284	0.07	1.08	0.16	N/A	33.46	F	17.27	F
	007:004	1254	602	2652	0.07	1.48	0.20	N/A	40.66	F	88.03	F
	008:004	1424	680	3020	0.09	1.91	0.22	N/A	66.91	F	451.93	F
	009:004	1594	758	3388	0.11	2.43	0.28	N/A	89.68	F	>2048	N/A
	010:004	1764	836	3756	0.12	3.01	0.48	N/A	162.56	F	N/A	N/A
	011:004	1934	914	4124	0.13	3.61	0.60	N/A	100.44	F	N/A	N/A
	012:004	2104	992	4492	0.15	4.24	0.68	N/A	161.87	F	N/A	N/A
	013:004	2274	1070	4860	0.17	5.01	0.98	N/A	1084.74	F	N/A	N/A
	014:004	2444	1148	5228	0.18	5.86	1.13	N/A	695.07	F	N/A	N/A
	015:004	2614	1226	5596	0.20	6.69	1.24	N/A	439.66	F	N/A	N/A

					Sttrim						Sato	
	circuit				time (sec.)						time	
problem	size	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
arb2lp	010:001	1260	539	2772	0.08	0.43	0.02	N/A	1.98	F	0.20	F
	010:002	1428	638	3100	0.07	2.24	0.34	N/A	51.22	F	>2048	N/A
	010:003	1596	737	3428	0.10	2.61	0.37	N/A	65.90	F	N/A	N/A
	010:004	1764	836	3756	0.12	3.01	0.48	N/A	162.56	F	N/A	N/A
	010:005	1932	935	4084	0.15	3.43	0.47	N/A	659.43	F	N/A	N/A
	010:006	2100	1034	4412	0.18	3.81	0.50	N/A	228.16	F	N/A	N/A
	010:007	2268	1133	4740	0.21	4.18	0.49	N/A	397.51	F	N/A	N/A
	010:008	2436	1232	5068	0.25	4.44	0.50	N/A	126.48	F	N/A	N/A

problem	TPTP size	clauses	literals	length	Sttrim time (sec.) simp	init	check	triv	full	sat?	Sato time (sec.)	sat?
SYN001-1	001	2	1	2	0.00	N/A	N/A	N/A	0.00	F	0.02	F
	002	4	2	8	0.00	0.04	0.00	N/A	0.04	F	0.01	F
	003	8	3	24	0.00	0.04	0.00	N/A	0.06	F	0.01	F
	004	16	4	64	0.00	0.05	0.00	N/A	0.07	F	0.02	F
	005	32	5	160	0.00	0.07	0.00	N/A	0.12	F	0.03	F
	006	64	6	384	0.00	0.08	0.00	N/A	0.18	F	0.02	F
	007	128	7	896	0.00	0.13	0.00	N/A	0.34	F	0.03	F
	008	256	8	2048	0.00	0.28	0.01	N/A	0.77	F	0.04	F
	009	512	9	4608	0.01	0.59	0.00	N/A	1.78	F	0.07	F
	010	1024	10	10240	0.03	1.36	0.01	N/A	4.21	F	0.15	F
	011	2048	11	22528	0.07	2.87	0.02	N/A	9.48	F	0.29	F
	012	4096	12	49152	0.14	6.61	0.05	N/A	21.81	F	0.62	F
	013	8192	13	106496	0.32	15.49	0.10	N/A	49.71	F	1.30	F
	014	16384	14	229376	0.67	35.12	0.20	N/A	113.59	F	2.80	F

problem	TPTP size	clauses	literals	length	Sttrim time (sec.) simp	init	check	triv	full	sat?	Sato time (sec.)	sat?
SYN003-1	002	8	6	14	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	003	12	9	24	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	004	16	12	34	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	005	20	15	44	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	006	24	18	54	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	007	28	21	64	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	008	32	24	74	0.00	N/A	N/A	N/A	0.00	F	0.02	F
	009	36	27	84	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	010	40	30	94	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	020	80	60	194	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	040	160	120	394	0.01	N/A	N/A	N/A	0.02	F	0.04	F
	080	320	240	794	0.04	N/A	N/A	N/A	0.04	F	0.04	F
	160	640	480	1594	0.12	N/A	N/A	N/A	0.12	F	0.07	F
	320	1280	960	3194	0.50	N/A	N/A	N/A	0.51	F	0.16	F
	640	2560	1920	6394	2.13	N/A	N/A	N/A	2.16	F	0.28	F

problem	TPTP size	clauses	literals	length	Sttrim time (sec.) simp	init	check	triv	full	sat?	Sato time (sec.)	sat?
SYN004-1	002	5	4	10	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	003	7	6	16	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	004	9	8	22	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	005	11	10	28	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	006	13	12	34	0.00	N/A	N/A	N/A	0.00	F	0.02	F
	007	15	14	40	0.00	N/A	N/A	N/A	0.00	F	0.02	F
	008	17	16	46	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	009	19	18	52	0.00	N/A	N/A	N/A	0.00	F	0.02	F
	010	21	20	58	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	020	41	40	118	0.00	N/A	N/A	N/A	0.00	F	0.02	F
	040	81	80	238	0.02	N/A	N/A	N/A	0.02	F	0.02	F
	080	161	160	478	0.07	N/A	N/A	N/A	0.07	F	0.03	F
	160	321	320	958	0.21	N/A	N/A	N/A	0.22	F	0.05	F
	320	641	640	1918	0.91	N/A	N/A	N/A	0.91	F	0.10	F
	640	1281	1280	3838	4.40	N/A	N/A	N/A	4.42	F	0.19	F

problem	TPTP size	clauses	literals	length	Sttrim time (sec.) simp	init	check	triv	full	Sttrim sat?	Sato time (sec.)	Sato sat?
SYN008-1	N/A	6	7	13	0.00	N/A	N/A	N/A	0.00	F	0.01	F
SYN010-1	005:005	27	26	132	0.00	N/A	N/A	N/A	0.00	F	0.03	F
SYN011-1	N/A	8	7	17	0.00	0.04	0.00	N/A	0.06	F	0.01	F
SYN028-1	N/A	6	5	12	0.00	N/A	N/A	N/A	0.00	F	0.02	F
SYN029-1	N/A	5	3	10	0.00	0.03	0.00	N/A	0.04	F	0.02	F
SYN030-1	N/A	9	5	22	0.00	0.05	0.00	N/A	0.07	F	0.01	F
SYN032-1	N/A	7	6	17	0.00	0.05	0.00	N/A	0.07	F	0.02	F
SYN040-1	N/A	4	2	6	0.00	N/A	N/A	N/A	0.00	F	0.01	F
SYN041-1	N/A	4	2	4	0.00	N/A	N/A	N/A	0.00	F	0.00	F
SYN044-1	N/A	6	3	13	0.00	0.05	0.00	N/A	0.07	F	0.01	F
SYN045-1	N/A	4	3	7	0.00	N/A	N/A	N/A	0.00	F	0.01	F
SYN046-1	N/A	3	2	4	0.00	N/A	N/A	N/A	0.00	F	0.01	F
SYN047-1	N/A	5	4	10	0.00	N/A	N/A	N/A	0.00	F	0.01	F
SYN063-1	N/A	7	7	22	0.00	0.04	0.00	N/A	0.05	T	0.01	T

problem	TPTP size	clauses	literals	length	Sttrim time (sec.) simp	init	check	triv	full	Sttrim sat?	Sato time (sec.)	Sato sat?
SYN085-1	001	3	2	4	0.00	N/A	N/A	N/A	0.00	F	0.02	F
	002	4	3	6	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	003	5	4	8	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	004	6	5	10	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	005	7	6	12	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	006	8	7	14	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	007	9	8	16	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	008	10	9	18	0.00	N/A	N/A	N/A	0.00	F	0.03	F
	009	11	10	20	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	010	12	11	22	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	020	22	21	42	0.00	N/A	N/A	N/A	0.00	F	0.02	F
	040	42	41	82	0.00	N/A	N/A	N/A	0.00	F	0.03	F
	080	82	81	162	0.00	N/A	N/A	N/A	0.00	F	0.04	F
	160	162	161	322	0.00	N/A	N/A	N/A	0.00	F	0.09	F
	320	322	321	642	0.00	N/A	N/A	N/A	0.00	F	0.29	F
	640	642	641	1282	0.01	N/A	N/A	N/A	0.01	F	1.13	F

problem	TPTP size	clauses	literals	length	Sttrim time (sec.) simp	init	check	triv	full	Sttrim sat?	Sato time (sec.)	Sato sat?
SYN086-1	001	3	2	5	0.00	N/A	0.00	0.00	0.00	T	0.01	T
	002	9	6	21	0.00	N/A	0.00	0.00	0.00	T	0.02	T
	003	19	12	49	0.00	N/A	0.00	0.00	0.00	T	0.01	T
	004	33	20	89	0.00	N/A	0.00	0.00	0.00	T	0.01	T
	005	51	30	141	0.00	N/A	0.00	0.00	0.00	T	0.03	T
	006	73	42	205	0.00	N/A	0.00	0.00	0.00	T	0.02	T
	007	99	56	281	0.00	N/A	0.00	0.00	0.00	T	0.02	T
	008	129	72	369	0.00	N/A	0.00	0.00	0.00	T	0.02	T
	009	163	90	469	0.00	N/A	0.00	0.00	0.00	T	0.02	T
	010	201	110	581	0.00	N/A	0.00	0.00	0.00	T	0.03	T
	020	801	420	2361	0.01	N/A	0.00	0.02	0.03	T	0.08	T
	040	3201	1640	9521	0.09	N/A	0.02	0.14	0.14	T	0.34	T
	080	12801	6480	38241	0.37	N/A	0.09	0.58	0.58	T	1.32	T
	160	51201	25760	153281	N/A	N/A	N/A	N/A	>2048	N/A	>atoms	N/A

problem	TPTP size	clauses	literals	length	Sttrim time (sec.) simp	init	check	triv	full	sat?	Sato time (sec.)	sat?
SYN087-1	001	5	6	10	0.00	N/A	0.00	0.00	0.00	T	0.01	T
	002	13	10	34	0.00	N/A	0.00	0.00	0.00	T	0.02	T
	003	21	14	58	0.00	N/A	0.00	0.00	0.00	T	0.01	T
	004	29	18	82	0.00	N/A	0.00	0.00	0.00	T	0.01	T
	005	37	22	106	0.00	N/A	0.00	0.00	0.00	T	0.01	T
	006	45	26	130	0.00	N/A	0.00	0.00	0.00	T	0.01	T
	007	53	30	154	0.00	N/A	0.00	0.00	0.00	T	0.03	T
	008	61	34	178	0.00	N/A	0.00	0.00	0.00	T	0.02	T
	009	69	38	202	0.00	N/A	0.00	0.00	0.00	T	0.02	T
	010	77	42	226	0.00	N/A	0.00	0.00	0.00	T	0.02	T
	020	157	82	466	0.00	N/A	0.00	0.00	0.01	T	0.03	T
	040	317	162	946	0.00	N/A	0.00	0.01	0.01	T	0.04	T
	080	637	322	1906	0.01	N/A	0.00	0.02	0.02	T	0.08	T
	160	1277	642	3826	0.02	N/A	0.00	0.04	0.04	T	0.13	T
	320	2557	1282	7666	0.04	N/A	0.02	0.08	0.08	T	0.26	T
	640	5117	2562	15346	0.09	N/A	0.04	0.18	0.18	T	0.55	T

problem	TPTP size	clauses	literals	length	Sttrim time (sec.) simp	init	check	triv	full	sat?	Sato time (sec.)	sat?
SYN089-1	001	5	2	7	0.00	N/A	N/A	N/A	0.00	F	0.02	F
	002	13	6	25	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	003	25	12	55	0.00	N/A	N/A	N/A	0.00	F	0.02	F
	004	41	20	97	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	005	61	30	151	0.00	N/A	N/A	N/A	0.00	F	0.02	F
	006	85	42	217	0.00	N/A	N/A	N/A	0.00	F	0.04	F
	007	113	56	295	0.00	N/A	N/A	N/A	0.00	F	0.03	F
	008	145	72	385	0.00	N/A	N/A	N/A	0.01	F	0.02	F
	009	181	90	487	0.01	N/A	N/A	N/A	0.01	F	0.03	F
	010	221	110	601	0.00	N/A	N/A	N/A	0.01	F	0.04	F
	020	841	420	2401	0.07	N/A	N/A	N/A	0.07	F	0.10	F
	040	3281	1640	9601	0.60	N/A	N/A	N/A	0.63	F	0.37	F
	080	12961	6480	38401	4.70	N/A	N/A	N/A	4.82	F	1.46	F
	160	51521	25760	153601	N/A	N/A	N/A	N/A	>2048	N/A	>atoms	N/A

problem	TPTP size	clauses	literals	length	Sttrim time (sec.) simp	init	check	triv	full	sat?	Sato time (sec.)	sat?
SYN090-1	001	9	6	14	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	002	17	10	38	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	003	25	14	62	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	004	33	18	86	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	005	41	22	110	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	006	49	26	134	0.00	N/A	N/A	N/A	0.00	F	0.02	F
	007	57	30	158	0.00	N/A	N/A	N/A	0.00	F	0.02	F
	008	65	34	182	0.00	N/A	N/A	N/A	0.00	F	0.03	F
	009	73	38	206	0.01	N/A	N/A	N/A	0.01	F	0.03	F
	010	81	42	230	0.01	N/A	N/A	N/A	0.01	F	0.02	F
	020	161	82	470	0.03	N/A	N/A	N/A	0.03	F	0.03	F
	040	321	162	950	0.11	N/A	N/A	N/A	0.11	F	0.04	F
	080	641	322	1910	0.47	N/A	N/A	N/A	0.48	F	0.09	F
	160	1281	642	3830	1.97	N/A	N/A	N/A	1.98	F	0.14	F
	320	2561	1282	7670	9.08	N/A	N/A	N/A	9.10	F	0.30	F
	640	5121	2562	15350	39.09	N/A	N/A	N/A	39.14	F	0.61	F

					Sttrim						Sato	
	TPTP				time (sec.)						time	
problem	size	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
SYN091-1	001	6	4	10	0.00	N/A	0.00	0.00	0.00	T	0.01	T
	002	18	12	42	0.00	0.04	0.00	N/A	0.07	N/A	0.00	T
	003	38	24	98	0.00	0.06	0.00	0.09	N/A	T	0.01	T
	004	66	40	178	0.00	0.08	0.00	0.44	N/A	T	0.02	T
	005	102	60	282	0.00	0.11	0.02	0.98	N/A	T	0.02	T
	006	146	84	410	0.00	0.14	0.02	5.23	N/A	T	0.03	T
	007	198	112	562	0.00	0.19	0.05	7.27	N/A	T	0.04	T
	008	258	144	738	0.00	0.28	0.12	34.92	N/A	T	0.05	T
	009	326	180	938	0.00	0.35	0.23	38.47	N/A	T	0.06	T
	010	402	220	1162	0.01	0.45	0.35	186.25	N/A	T	0.07	T
	011	486	264	1410	0.01	0.59	0.68	303.29	N/A	T	0.07	T
	012	578	312	1682	N/A	N/A	N/A	N/A	>2048	N/A	0.06	T
	013	678	364	1978	N/A	N/A	N/A	N/A	N/A	N/A	0.10	T
	014	786	420	2298	N/A	N/A	N/A	N/A	N/A	N/A	0.13	T
	015	902	480	2642	N/A	N/A	N/A	N/A	N/A	N/A	0.16	T

					Sttrim						Sato	
	TPTP				time (sec.)						time	
problem	size	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
SYN092-1	001	10	12	20	0.00	0.04	0.00	N/A	0.06	N/A	0.02	T
	002	26	20	68	0.00	0.05	0.00	N/A	0.08	N/A	0.02	T
	003	42	28	116	0.00	0.05	0.00	N/A	0.16	N/A	0.04	T
	004	58	36	164	0.00	0.08	0.00	N/A	0.20	N/A	0.02	T
	005	74	44	212	0.00	0.10	0.00	N/A	0.21	N/A	0.02	T
	006	90	52	260	0.00	0.10	0.00	N/A	0.23	N/A	0.01	T
	007	106	60	308	0.00	0.12	0.00	N/A	0.26	N/A	0.02	T
	008	122	68	356	0.00	0.12	0.00	N/A	0.27	N/A	0.04	T
	009	138	76	404	0.00	0.15	0.00	N/A	0.31	N/A	0.03	T
	010	154	84	452	0.00	0.16	0.00	N/A	0.34	N/A	0.03	T
	011	170	92	500	0.00	0.18	0.00	N/A	0.35	N/A	0.04	T
	012	186	100	548	0.00	0.18	0.00	N/A	0.36	N/A	0.04	T
	013	202	108	596	0.00	0.19	0.00	N/A	0.39	N/A	0.04	T
	014	218	116	644	0.00	0.19	0.00	N/A	0.38	N/A	0.04	T
	015	234	124	692	0.00	0.22	0.00	N/A	0.44	N/A	0.05	T
	020	314	164	932	0.00	0.32	0.00	N/A	0.59	N/A	N/A	N/A
	040	634	324	1892	0.00	0.85	0.00	N/A	1.36	N/A	N/A	N/A
	080	1274	644	3812	0.02	2.75	0.01	N/A	3.97	N/A	N/A	N/A
	160	2554	1284	7652	0.04	9.56	0.02	N/A	12.98	N/A	N/A	N/A

					Sttrim						Sato	
	TPTP				time (sec.)						time	
problem	size	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
SYN093-1	001	10	7	21	0.00	N/A	N/A	N/A	0.00	F	0.01	F
	002	26	19	63	0.00	0.06	0.00	N/A	0.09	F	0.01	F
	003	50	37	129	0.00	0.07	0.00	N/A	0.14	F	0.03	F
	004	82	61	219	0.00	0.09	0.00	N/A	0.25	F	1.37	F
	005	122	91	333	0.00	0.13	0.01	N/A	0.41	F	208.36	F
	006	170	127	471	0.00	0.16	0.00	N/A	0.49	F	>2048	N/A
	007	226	169	633	0.01	0.25	0.00	N/A	0.77	F	N/A	N/A
	008	290	217	819	0.01	0.36	0.00	N/A	1.03	F	N/A	N/A
	009	362	271	1029	0.02	0.48	0.02	N/A	1.39	F	N/A	N/A
	010	442	331	1263	0.02	0.67	0.03	N/A	1.86	F	N/A	N/A
	011	530	397	1521	0.02	0.89	0.04	N/A	2.30	F	>2048	N/A
	012	626	469	1803	0.02	1.18	0.04	N/A	2.97	F	N/A	N/A
	013	730	547	2109	0.03	1.53	0.05	N/A	3.75	F	N/A	N/A
	014	842	631	2439	0.04	1.97	0.05	N/A	4.70	F	N/A	N/A
	015	962	721	2793	0.04	2.48	0.07	N/A	5.73	F	N/A	N/A
	020	1682	1261	4923	0.08	7.07	0.11	N/A	14.72	F	N/A	N/A
	040	6562	4921	19443	N/A	N/A	N/A	N/A	>mem	N/A	N/A	N/A
	080	25922	19441	77283	N/A	N/A	N/A	N/A	>2048	N/A	N/A	N/A

					Sttrim						Sato	
	TPTP				time (sec.)						time	
problem	size	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
SYN094-1	001	18	15	40	0.00	0.04	0.00	N/A	0.06	F	0.00	F
	002	34	27	88	0.00	0.05	0.00	N/A	0.11	F	0.02	F
	003	50	39	136	0.00	0.07	0.00	N/A	0.15	F	0.17	F
	004	66	51	184	0.00	0.09	0.00	N/A	0.19	F	2.29	F
	005	82	63	232	0.00	0.09	0.00	N/A	0.24	F	42.27	F
	006	98	75	280	0.00	0.12	0.00	N/A	0.29	F	801.49	F
	007	114	87	328	0.00	0.14	0.01	N/A	0.35	F	>2048	N/A
	008	130	99	376	0.00	0.16	0.01	N/A	0.41	F	N/A	N/A
	009	146	111	424	0.00	0.18	0.01	N/A	0.46	F	N/A	N/A
	010	162	123	472	0.00	0.19	0.02	N/A	0.50	F	N/A	N/A
	020	322	243	952	0.01	0.45	0.04	N/A	1.22	F	N/A	N/A
	040	642	483	1912	0.02	1.32	0.15	N/A	3.35	F	N/A	N/A
	080	1282	963	3832	0.03	4.49	0.72	N/A	10.38	F	N/A	N/A
	160	2562	1923	7672	0.07	16.47	2.96	N/A	37.90	F	N/A	N/A

					Sttrim						Sato	
	TPTP				time (sec.)						time	
problem	size	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
SYN097-1	001	20	14	42	0.00	N/A	N/A	N/A	0.00	F	0.02	F
	002	52	38	126	0.00	0.07	0.00	N/A	0.11	F	0.01	F
	003	100	74	258	0.00	0.11	0.00	N/A	0.18	F	0.02	F
	004	164	122	438	0.01	0.16	0.00	N/A	0.45	F	0.02	F
	005	244	182	666	0.01	0.28	0.01	N/A	0.81	F	0.03	F
	006	340	254	942	0.01	0.45	0.01	N/A	1.20	F	0.06	F
	007	452	338	1266	0.02	0.68	0.02	N/A	1.75	F	0.11	F
	008	580	434	1638	0.02	1.05	0.03	N/A	2.78	F	0.20	F
	009	724	542	2058	0.03	1.53	0.04	N/A	3.81	F	0.62	F
	010	884	662	2526	0.04	2.18	0.04	N/A	5.31	F	1.38	F
	011	1060	794	3042	0.04	3.05	0.05	N/A	7.12	F	3.61	F
	012	1252	938	3606	0.06	4.18	0.07	N/A	9.41	F	9.98	F
	013	1460	1094	4218	0.07	5.45	0.07	N/A	12.14	F	42.04	F
	014	1684	1262	4878	0.08	7.09	0.13	N/A	15.28	F	84.35	F
	015	1924	1442	5586	0.09	9.14	0.14	N/A	19.82	F	239.12	F
	016	2180	1634	6342	0.11	11.70	0.11	N/A	25.48	F	884.45	F
	017	2452	1838	7146	0.12	14.53	0.17	N/A	30.70	F	>2048	N/A
	018	2740	2054	7998	0.14	18.03	0.22	N/A	37.78	F	N/A	N/A
	019	3044	2282	8898	0.16	22.96	0.19	N/A	46.02	F	N/A	N/A
	020	3364	2522	9846	0.17	27.74	0.27	N/A	55.07	F	N/A	N/A
	040	13124	9842	38886	N/A	N/A	N/A	N/A	>mem	N/A	N/A	N/A

					Sttrim						Sato	
	TPTP				time (sec.)						time	
problem	size	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
SYN098-1	001	36	30	80	0.00	0.06	0.00	N/A	0.08	F	0.02	F
	002	68	54	176	0.00	0.08	0.00	N/A	0.14	F	0.02	F
	003	100	78	272	0.00	0.12	0.00	N/A	0.27	F	0.04	F
	004	132	102	368	0.00	0.16	0.00	N/A	0.36	F	0.04	F
	005	164	126	464	0.00	0.18	0.00	N/A	0.45	F	0.05	F
	006	196	150	560	0.00	0.25	0.01	N/A	0.58	F	0.07	F
	007	228	174	656	0.00	0.29	0.01	N/A	0.73	F	0.15	F
	008	260	198	752	0.00	0.33	0.01	N/A	0.83	F	0.31	F
	009	292	222	848	0.00	0.39	0.02	N/A	1.04	F	0.82	F
	010	324	246	944	0.01	0.46	0.03	N/A	1.18	F	1.95	F
	011	356	270	1040	0.01	0.51	0.01	N/A	1.30	F	5.39	F
	012	388	294	1136	0.01	0.59	0.02	N/A	1.43	F	13.12	F
	013	420	318	1232	0.01	0.68	0.02	N/A	1.70	F	33.68	F
	014	452	342	1328	0.01	0.74	0.03	N/A	1.83	F	90.83	F
	015	484	366	1424	0.01	0.82	0.05	N/A	2.01	F	237.03	F
	016	516	390	1520	0.01	0.93	0.06	N/A	2.27	F	608.93	F
	017	548	414	1616	0.01	0.98	0.07	N/A	2.47	F	1582.79	F
	018	580	438	1712	0.01	1.10	0.07	N/A	2.68	F	>2048	N/A
	019	612	462	1808	0.01	1.20	0.07	N/A	2.93	F	N/A	N/A
	020	644	486	1904	0.02	1.34	0.07	N/A	3.10	F	>2048	N/A
	040	1284	966	3824	0.03	4.76	0.32	N/A	10.13	F	N/A	N/A
	080	2564	1926	7664	0.07	16.74	1.50	N/A	36.54	F	N/A	N/A
	160	5124	3846	15344	N/A	N/A	N/A	N/A	>mem	N/A	N/A	N/A
	320	10244	7686	30704	N/A	N/A	N/A	N/A	>mem	N/A	N/A	N/A
	640	20484	15366	61424	N/A	N/A	N/A	N/A	>mem	N/A	N/A	N/A

	TPTP				Sttrim						Sato	
					time (sec.)						time	
problem	size	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
SYN302-1	001	8	2	24	0.00	N/A	0.00	0.00	0.00	T	0.01	T
	002	32	8	96	0.00	N/A	0.00	0.00	0.00	T	0.00	T
	003	72	18	216	0.00	N/A	0.00	0.00	0.00	T	0.02	T
	004	128	32	384	0.00	N/A	0.00	0.00	0.00	T	0.02	T
	005	200	50	600	0.00	N/A	0.00	0.00	0.00	T	0.03	T
	006	288	72	864	0.00	N/A	0.00	0.00	0.00	T	0.04	T
	007	392	98	1176	0.00	N/A	0.00	0.01	0.01	T	0.05	T
	008	512	128	1536	0.01	N/A	0.00	0.01	0.01	T	0.05	T
	009	648	162	1944	0.01	N/A	0.00	0.02	0.02	T	0.06	T
	010	800	200	2400	0.01	N/A	0.00	0.02	0.02	T	0.08	T
	020	3200	800	9600	0.05	N/A	0.02	0.10	0.10	T	0.29	T
	040	12800	3200	38400	0.21	N/A	0.08	0.41	0.41	T	1.22	T

					Sttrim						Sato	
					time (sec.)						time	
problem	number	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
bf0432	005	3668	1040	9574	0.52	3.35	0.09	N/A	53.57	F	225.04	F
	007	3668	1040	9574	0.42	4.95	0.27	N/A	585.50	F	>2048	N/A
	011	3743	1053	9768	0.40	5.24	0.33	N/A	393.02	F	167.54	F
	015	3685	1044	9614	0.50	4.83	0.16	N/A	319.34	F	>2048	N/A
	017	3685	1044	9614	0.50	4.72	0.27	N/A	675.65	F	>2048	N/A
	033	2784	865	7197	0.29	2.84	0.09	N/A	23.75	F	>2048	N/A
	035	2784	865	7197	0.27	3.34	0.11	N/A	46.99	F	>2048	N/A
	039	2790	864	7211	0.29	2.98	0.04	N/A	26.64	F	>2048	N/A
	041	2783	865	7194	0.27	2.87	0.10	N/A	34.26	F	>2048	N/A
	043	2783	865	7194	0.27	3.29	0.11	N/A	63.77	F	>2048	N/A
	048	1037	428	2398	0.05	1.03	0.06	N/A	5.61	F	3.33	F
	053	3766	1057	9828	0.41	5.80	0.40	N/A	347.94	F	203.11	F
	070	3761	1057	9816	0.37	5.16	0.32	N/A	553.52	F	174.22	F
	077	3685	1044	9614	0.53	4.78	0.21	N/A	842.22	F	>2048	N/A
	079	3685	1044	9614	0.50	4.81	0.16	N/A	318.94	F	>2048	N/A
	091	3761	1057	9812	0.41	5.27	0.30	N/A	817.30	F	67.58	F
	103	3746	1055	9777	0.44	5.51	0.27	N/A	576.26	F	128.80	F
	122	3763	1057	9826	0.33	5.79	0.33	N/A	738.55	F	>2048	N/A
	135	1031	424	2371	0.04	1.00	0.07	N/A	8.62	F	212.85	F
	138	1000	421	2278	0.05	0.98	0.06	N/A	8.30	F	383.98	F
	139	1010	423	2302	0.05	0.99	0.06	8.55	N/A	T	4.18	T

					Sttrim						Sato	
					time (sec.)						time	
problem	number	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
bf1355	075	6778	2180	17110	1.08	15.92	0.09	N/A	53.45	F	>2048	N/A
	076	6778	2180	17110	1.10	17.47	0.40	N/A	93.01	F	>2048	N/A
	119	7306	2297	18551	1.25	19.01	0.04	N/A	39.20	F	>2048	N/A
	120	7306	2297	18551	1.24	18.20	0.07	N/A	34.53	F	>2048	N/A
	123	7306	2297	18551	1.31	17.42	0.15	N/A	63.01	F	>2048	N/A
	124	7306	2297	18551	1.32	19.34	0.18	N/A	66.41	F	>2048	N/A
	127	7306	2297	18551	1.09	17.78	0.12	N/A	62.34	F	>2048	N/A
	128	7306	2297	18551	1.13	18.88	0.45	N/A	131.95	F	>2048	N/A
	131	7271	2287	18465	1.16	17.69	0.05	N/A	35.24	F	>2048	N/A
	132	7271	2287	18465	1.20	18.96	0.00	N/A	30.12	F	>2048	N/A
	143	7289	2291	18511	1.24	17.23	0.25	N/A	62.65	F	>2048	N/A
	144	7289	2291	18511	1.18	19.48	0.27	N/A	66.60	F	>2048	N/A
	147	7289	2291	18511	1.23	19.05	0.08	N/A	38.50	F	>2048	N/A
	148	7289	2291	18511	1.24	17.91	0.10	N/A	32.32	F	>2048	N/A
	151	7295	2293	18525	1.31	17.35	0.13	N/A	59.04	F	>2048	N/A
	152	7295	2293	18525	1.18	19.46	0.14	N/A	65.94	F	>2048	N/A
	159	7305	2297	18549	1.37	19.24	0.12	N/A	38.15	F	>2048	N/A
	160	7305	2297	18549	1.37	18.03	0.14	N/A	33.44	F	>2048	N/A
	167	7262	2284	18443	1.19	17.83	0.11	N/A	57.56	F	>2048	N/A
	168	7262	2284	18443	1.26	18.67	0.15	N/A	69.28	F	>2048	N/A
	175	7248	2281	18402	1.15	17.50	0.06	N/A	34.85	F	>2048	N/A
	176	7248	2281	18402	1.19	19.21	0.05	N/A	39.33	F	>2048	N/A
	227	7307	2298	18553	1.25	19.35	0.05	N/A	37.75	F	>2048	N/A
	228	7307	2298	18553	1.24	17.45	0.07	N/A	34.21	F	>2048	N/A
	239	7307	2298	18553	1.04	25.22	0.16	N/A	91.25	F	>2048	N/A
	240	7307	2298	18553	1.03	25.65	0.41	N/A	123.43	F	>2048	N/A
	243	4031	1450	9779	0.19	9.97	0.14	N/A	45.29	F	>2048	N/A
	291	7266	2284	18451	1.24	19.50	0.12	N/A	38.40	F	>2048	N/A
	292	7266	2284	18451	1.23	17.94	0.05	N/A	34.64	F	>2048	N/A
	295	6291	2075	15760	1.02	15.63	0.23	N/A	48.04	F	>2048	N/A
	296	6291	2075	15760	1.09	14.47	0.09	N/A	33.58	F	>2048	N/A
	303	7273	2288	18469	1.19	19.39	0.14	N/A	38.44	F	>2048	N/A
	304	7273	2288	18469	1.17	17.63	0.09	N/A	49.01	F	>2048	N/A
	315	7273	2287	18461	1.14	17.78	0.10	N/A	60.42	F	>2048	N/A
	316	7273	2287	18461	1.17	19.03	0.23	N/A	103.45	F	>2048	N/A
	319	7248	2281	18402	1.15	17.49	0.15	N/A	57.58	F	>2048	N/A
	320	7248	2281	18402	1.18	18.73	0.30	N/A	89.17	F	>2048	N/A
	323	7271	2286	18465	1.29	17.89	0.10	N/A	32.62	F	>2048	N/A
	324	7271	2286	18465	1.31	19.37	0.06	N/A	52.81	F	>2048	N/A
	327	7248	2281	18402	1.21	17.58	0.16	N/A	64.51	F	>2048	N/A
	328	7248	2281	18402	1.18	19.02	0.20	N/A	71.67	F	>2048	N/A
	347	7271	2286	18465	1.09	17.56	0.18	N/A	48.59	F	>2048	N/A
	348	7271	2286	18465	1.13	18.76	0.23	N/A	95.03	F	>2048	N/A
	351	7248	2281	18402	1.28	17.70	0.09	N/A	45.95	F	>2048	N/A
	352	7248	2281	18402	1.32	19.04	0.04	N/A	38.75	F	>2048	N/A
	355	7273	2288	18469	1.29	17.49	0.30	N/A	62.08	F	>2048	N/A
	356	7273	2288	18469	1.18	18.78	0.52	N/A	93.76	F	>2048	N/A
	359	7307	2298	18553	1.37	17.73	0.14	N/A	67.82	F	>2048	N/A

					Sttrim						Sato	
					time (sec.)						time	
problem	number	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
bf1355	360	7307	2298	18553	1.19	18.98	0.41	N/A	127.64	F	>2048	N/A
	363	7288	2292	18497	1.18	17.66	0.06	N/A	48.28	F	3.91	F
	364	7288	2292	18497	1.18	18.92	0.09	N/A	39.25	F	0.98	F
	445	7305	2297	18549	1.30	18.37	0.24	N/A	57.78	F	>2048	N/A
	446	7305	2297	18549	1.20	18.74	0.48	N/A	76.35	F	>2048	N/A
	449	7296	2297	18517	1.18	17.49	0.04	N/A	35.11	F	11.72	F
	450	7296	2297	18517	1.20	19.10	0.09	N/A	38.51	F	1.39	F
	453	7295	2293	18523	1.29	18.47	0.20	N/A	59.83	F	>2048	N/A
	454	7295	2293	18523	1.18	18.61	0.12	N/A	53.82	F	>2048	N/A
	457	7271	2287	18465	1.24	17.73	0.06	N/A	56.94	F	>2048	N/A
	458	7271	2287	18465	1.18	18.53	0.21	N/A	65.37	F	>2048	N/A
	461	7305	2297	18549	1.16	17.64	0.14	N/A	58.54	F	>2048	N/A
	462	7305	2297	18549	1.19	19.20	0.36	N/A	83.06	F	>2048	N/A
	465	7295	2293	18525	1.16	17.83	0.11	N/A	54.87	F	>2048	N/A
	466	7295	2293	18525	1.21	19.28	0.14	N/A	61.97	F	>2048	N/A
	469	7289	2291	18511	1.14	17.62	0.20	N/A	53.97	F	>2048	N/A
	470	7289	2291	18511	1.18	19.24	0.34	N/A	72.97	F	>2048	N/A
	473	7295	2293	18523	1.15	18.01	0.21	N/A	59.25	F	>2048	N/A
	474	7295	2293	18523	1.18	19.42	0.18	N/A	77.57	F	>2048	N/A
	477	7244	2279	18390	1.15	17.69	0.05	N/A	33.92	F	>2048	N/A
	478	7244	2279	18390	1.18	18.97	0.07	N/A	54.08	F	>2048	N/A
	481	7271	2287	18465	1.15	17.68	0.16	N/A	50.61	F	>2048	N/A
	482	7271	2287	18465	1.18	19.06	0.09	N/A	63.34	F	>2048	N/A
	485	7296	2294	18527	1.20	17.94	0.18	N/A	59.95	F	>2048	N/A
	486	7296	2294	18527	1.25	19.23	0.20	N/A	69.90	F	>2048	N/A
	489	7271	2286	18465	1.30	17.44	0.17	N/A	64.60	F	>2048	N/A
	490	7271	2286	18465	1.30	18.97	0.45	N/A	293.22	F	>2048	N/A
	493	7273	2287	18461	1.18	17.42	0.05	N/A	36.83	F	4.96	F
	494	7273	2287	18461	1.19	18.87	0.12	N/A	52.93	F	1.01	F
	497	6768	2178	17081	1.20	15.99	0.09	N/A	53.11	F	>2048	N/A
	498	6768	2178	17081	1.10	16.90	0.27	N/A	64.45	F	>2048	N/A
	505	6767	2177	17079	1.07	15.87	0.20	N/A	48.95	F	>2048	N/A
	506	6767	2177	17079	1.10	17.39	0.07	N/A	59.27	F	>2048	N/A
	509	6768	2178	17081	1.07	15.85	0.17	N/A	53.15	F	>2048	N/A
	510	6768	2178	17081	1.09	17.22	0.12	N/A	59.25	F	>2048	N/A
	513	7270	2286	18463	1.15	17.77	0.14	N/A	53.67	F	>2048	N/A
	514	7270	2286	18463	1.19	19.01	0.11	N/A	63.04	F	>2048	N/A
	517	7288	2293	18499	1.17	19.16	0.32	N/A	84.97	F	>2048	N/A
	518	7288	2293	18499	1.37	17.94	0.29	N/A	53.48	F	>2048	N/A
	521	7264	2287	18436	1.18	19.01	0.32	N/A	69.86	F	>2048	N/A
	522	7264	2287	18436	1.36	17.65	0.31	N/A	62.28	F	1683.88	F
	525	6788	2184	17134	1.10	17.37	0.14	N/A	65.45	F	>2048	N/A
	526	6788	2184	17134	1.11	16.12	0.17	N/A	50.93	F	704.87	F
	529	7305	2296	18549	1.36	19.08	0.05	N/A	38.27	F	>2048	N/A
	530	7305	2296	18549	1.37	17.98	0.05	N/A	29.12	F	>2048	N/A
	533	7296	2294	18527	1.31	19.18	0.05	N/A	30.29	F	>2048	N/A
	534	7296	2294	18527	1.30	17.79	0.13	N/A	31.74	F	>2048	N/A
	537	6711	2167	16919	1.20	16.94	0.13	N/A	51.66	F	>2048	N/A

					Sttrim						Sato	
					time (sec.)						time	
problem	number	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
bf1355	538	6711	2167	16919	1.26	15.84	0.21	N/A	49.91	F	675.06	F
	557	7296	2294	18527	1.23	18.25	0.27	N/A	58.54	F	>2048	N/A
	558	7296	2294	18527	1.19	18.74	0.68	N/A	105.90	F	>2048	N/A
	561	7262	2284	18443	1.23	17.57	0.41	N/A	77.76	F	>2048	N/A
	562	7262	2284	18443	1.18	18.46	0.84	N/A	186.49	F	>2048	N/A
	565	7298	2296	18523	1.29	19.17	0.30	N/A	94.42	F	>2048	N/A
	566	7298	2296	18523	1.32	17.84	0.23	N/A	58.26	F	>2048	N/A
	573	6778	2180	17110	1.07	15.89	0.12	N/A	54.12	F	>2048	N/A
	574	6778	2180	17110	1.11	17.46	0.25	N/A	74.94	F	>2048	N/A
	577	6793	2185	17146	1.08	15.80	0.18	N/A	54.86	F	>2048	N/A
	578	6793	2185	17146	1.11	17.44	0.45	N/A	77.67	F	>2048	N/A
	581	7288	2292	18497	1.16	17.74	0.12	N/A	60.49	F	>2048	N/A
	582	7288	2292	18497	1.18	19.28	0.22	N/A	84.76	F	>2048	N/A
	585	7270	2286	18463	1.28	17.73	0.02	N/A	35.07	F	>2048	N/A
	586	7270	2286	18463	1.32	19.01	0.04	N/A	36.34	F	>2048	N/A
	589	7262	2284	18443	1.22	17.44	0.05	N/A	27.97	F	>2048	N/A
	590	7262	2284	18443	1.25	19.35	0.13	N/A	30.04	F	>2048	N/A
	593	6778	2180	17110	1.08	15.93	0.15	N/A	53.44	F	>2048	N/A
	594	6778	2180	17110	1.10	17.41	0.46	N/A	92.94	F	>2048	N/A
	597	7298	2296	18523	1.16	17.60	0.21	N/A	59.72	F	>2048	N/A
	598	7298	2296	18523	1.18	19.21	0.40	N/A	83.67	F	>2048	N/A
	601	7283	2291	18487	1.15	17.72	0.12	N/A	58.29	F	>2048	N/A
	602	7283	2291	18487	1.19	19.24	0.21	N/A	102.95	F	>2048	N/A
	617	7290	2294	18503	1.25	17.65	0.10	N/A	49.27	F	80.22	F
	618	7290	2294	18503	1.24	18.89	0.11	N/A	52.70	F	180.22	F
	621	7298	2296	18523	1.31	17.39	0.05	N/A	34.34	F	94.67	F
	622	7298	2296	18523	1.32	19.61	0.04	N/A	38.08	F	201.01	F
	625	7266	2284	18451	1.29	17.56	0.29	N/A	103.65	F	>2048	N/A
	626	7266	2284	18451	1.30	18.70	0.43	N/A	258.21	F	>2048	N/A
	629	6767	2177	17079	1.21	15.88	0.14	N/A	45.88	F	>2048	N/A
	630	6767	2177	17079	1.09	17.26	0.20	N/A	79.98	F	>2048	N/A
	633	7270	2286	18463	1.30	17.62	0.16	N/A	53.00	F	>2048	N/A
	634	7270	2286	18463	1.18	18.97	0.21	N/A	86.57	F	>2048	N/A
	637	6768	2177	17081	1.20	15.59	0.15	N/A	64.37	F	>2048	N/A
	638	6768	2177	17081	N/A	N/A	N/A	N/A	>2048	N/A	>2048	N/A
	653	7307	2298	18553	0.88	25.62	0.14	N/A	90.09	F	>2048	N/A
	654	7307	2298	18553	0.88	25.39	0.73	N/A	155.17	F	>2048	N/A
	661	7296	2294	18527	0.88	25.60	0.27	N/A	110.80	F	>2048	N/A
	662	7296	2294	18527	0.88	25.38	0.77	N/A	151.90	F	>2048	N/A
	665	7305	2297	18549	0.88	25.29	0.17	N/A	74.04	F	>2048	N/A
	666	7305	2297	18549	0.89	25.40	0.22	N/A	79.55	F	>2048	N/A
	669	7305	2297	18549	0.58	25.22	0.22	N/A	82.95	F	>2048	N/A
	670	7305	2297	18549	0.59	25.39	0.08	N/A	66.76	F	>2048	N/A
	681	7248	2281	18402	1.21	17.56	0.02	N/A	25.82	F	>2048	N/A
	682	7248	2281	18402	1.25	18.90	0.02	N/A	29.71	F	>2048	N/A
	701	7288	2292	18497	1.30	18.75	0.13	N/A	60.49	F	>2048	N/A
	702	7288	2292	18497	1.30	18.06	0.14	N/A	59.43	F	>2048	N/A
	741	7307	2298	18553	0.73	25.34	0.05	N/A	71.92	F	>2048	N/A
bf1355	742	7307	2298	18553	0.73	25.26	0.11	N/A	66.46	F	>2048	N/A
	749	7295	2293	18523	1.20	19.05	0.12	N/A	52.78	F	>2048	N/A
	750	7295	2293	18523	1.19	18.10	0.08	N/A	35.65	F	>2048	N/A
	757	7306	2297	18551	0.88	25.86	0.59	N/A	113.66	F	>2048	N/A
	758	7306	2297	18551	0.89	25.66	0.24	N/A	88.57	F	>2048	N/A

					Sttrim						Sato	
					time (sec.)						time	
problem	number	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
bf2670	001	3434	1393	8011	0.60	7.66	0.07	N/A	30.03	F	>2048	N/A
	007	1633	698	3842	0.14	2.45	0.02	N/A	8.03	T	0.18	T
	008	1643	700	3897	0.15	2.51	0.03	N/A	7.11	T	0.18	T
	033	1640	695	3944	0.09	2.57	0.05	N/A	8.42	T	0.18	T
	035	1699	705	4183	0.13	2.66	0.08	N/A	8.30	F	146.22	F
	038	2286	843	5783	0.18	3.31	0.05	N/A	14.73	T	0.24	T
	042	2169	837	5401	0.17	3.36	0.02	N/A	12.36	T	0.21	T
	051	3440	1389	8027	0.78	7.95	0.04	N/A	27.26	F	>2048	N/A
	052	3440	1389	8027	0.59	7.97	0.04	N/A	30.35	F	>2048	N/A
	121	1619	694	3811	0.15	2.52	0.04	N/A	7.55	T	0.17	T
	126	1613	694	3793	0.14	2.43	0.03	N/A	7.46	T	0.17	T
	130	4766	1784	11719	0.37	13.96	0.29	N/A	46.53	F	>2048	N/A
	143	1662	710	3939	0.17	2.59	0.02	N/A	7.41	T	0.20	T
	144	1625	698	3836	0.15	2.52	0.05	N/A	6.96	T	0.19	T
	145	1643	700	3897	0.16	2.50	0.02	N/A	7.03	T	0.18	T
	187	3391	1373	7909	0.23	8.92	0.34	N/A	41.48	F	>2048	N/A
	189	3417	1379	7988	0.16	8.85	0.08	N/A	41.15	F	>2048	N/A
	190	3417	1379	7988	0.26	9.13	0.24	N/A	39.76	F	>2048	N/A
	200	3369	1365	7855	0.20	8.83	0.09	N/A	41.19	F	>2048	N/A
	202	3439	1387	8042	0.16	9.03	0.37	N/A	40.84	F	>2048	N/A
	203	3439	1387	8042	0.26	9.08	0.17	N/A	40.43	F	>2048	N/A
	208	3423	1379	8006	0.33	8.84	0.27	N/A	35.85	F	>2048	N/A
	240	4941	1734	12013	0.75	10.65	0.07	N/A	54.94	F	>2048	N/A
	241	4941	1734	12013	0.75	10.43	0.07	N/A	47.97	F	>2048	N/A
	244	4269	1670	10133	0.83	12.25	0.14	N/A	73.82	F	>2048	N/A
	248	4324	1674	10320	0.96	12.49	0.18	N/A	97.08	F	>2048	N/A
	269	3496	1407	8190	0.62	6.02	0.04	N/A	22.64	F	>2048	N/A
	275	4333	1676	10368	0.83	10.88	0.87	N/A	40.13	T	32.03	T
	278	4334	1674	10354	0.80	7.51	0.50	28.79	N/A	T	0.50	T
	400	3249	1339	7508	0.50	8.38	0.42	N/A	56.52	F	>2048	N/A
	404	2324	985	5250	0.30	4.20	0.21	N/A	26.89	F	>2048	N/A
	414	2940	1207	6836	0.34	7.16	0.39	N/A	60.22	T	0.32	T
	449	1677	714	3981	0.18	2.64	0.03	N/A	7.89	T	0.18	T
	461	2169	837	5401	0.17	3.32	0.01	N/A	11.80	T	0.23	T
	468	3859	1488	9117	1.17	6.42	0.32	N/A	20.24	F	80.42	F
	483	4391	1686	10462	0.79	8.23	0.14	N/A	37.00	T	0.50	T
	487	3609	1423	8514	0.21	9.60	0.05	N/A	39.79	F	>2048	N/A
	491	3361	1363	7829	0.53	8.51	0.34	N/A	46.51	F	>2048	N/A
	492	3361	1363	7829	0.54	8.23	0.54	N/A	35.40	F	>2048	N/A
	493	3383	1371	7883	0.52	8.60	0.32	N/A	45.41	F	>2048	N/A
	494	3383	1371	7883	0.54	8.30	0.49	N/A	35.98	F	>2048	N/A
	495	3401	1371	7952	0.33	8.87	0.21	N/A	37.08	F	>2048	N/A
	501	4941	1734	12013	0.80	10.74	0.21	N/A	57.44	F	>2048	N/A
	502	4941	1734	12013	0.79	10.48	0.22	N/A	50.66	F	>2048	N/A
	510	4941	1734	12013	0.65	11.64	0.11	N/A	82.70	F	>2048	N/A
	511	4941	1734	12013	0.76	10.50	0.27	N/A	57.35	F	>2048	N/A
	522	3414	1378	7957	0.16	8.77	0.11	N/A	40.52	F	>2048	N/A
	524	3436	1386	8011	0.16	9.00	0.29	N/A	39.55	F	>2048	N/A

					Sttrim						Sato	
					time (sec.)						time	
problem	number	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
bf2670	532	3841	1476	9075	1.03	9.02	0.04	N/A	37.98	F	>2048	N/A
	533	3841	1476	9075	1.02	8.73	0.05	N/A	35.97	F	>2048	N/A
	538	4712	1668	11378	0.62	6.63	0.04	N/A	23.82	F	>2048	N/A
	546	3737	1464	8794	0.92	8.60	0.06	N/A	35.62	F	>2048	N/A
	547	3737	1464	8794	0.88	8.43	0.06	N/A	34.70	F	>2048	N/A

					Sttrim						Sato	
					time (sec.)						time	
problem	number	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
ssa0432	001	1027	435	2364	0.09	0.73	0.02	N/A	5.78	F	199.33	F
	002	1020	433	2339	0.07	0.82	0.03	N/A	6.16	F	169.34	F
	003	1027	435	2364	0.08	0.70	0.04	N/A	5.34	F	221.56	F
	004	1027	435	2364	0.09	0.73	0.03	N/A	5.79	F	183.72	F
	011	1027	435	2364	0.07	0.74	0.03	N/A	5.39	F	196.13	F
	014	1004	427	2293	0.07	0.86	0.02	N/A	8.19	F	396.72	F
	015	1020	433	2339	0.07	0.84	0.01	N/A	6.17	F	162.63	F

problem	number	clauses	literals	length	Sttrim time (sec.) simp	init	check	triv	full	sat?	Sato time (sec.)	sat?
ssa2670	127	3296	1355	7654	0.45	8.32	0.32	N/A	44.74	F	>2048	N/A
	128	3321	1359	7725	0.52	8.34	0.50	N/A	43.70	F	>2048	N/A
	129	3296	1355	7654	0.45	8.35	0.44	N/A	44.05	F	>2048	N/A
	130	3321	1359	7725	0.51	8.29	0.32	N/A	43.80	F	>2048	N/A
	131	3314	1357	7708	0.33	8.64	0.20	N/A	49.23	F	>2048	N/A
	132	3296	1355	7654	0.45	8.30	0.48	N/A	44.41	F	>2048	N/A
	133	3296	1355	7654	0.45	8.45	0.38	N/A	44.79	F	>2048	N/A
	134	3273	1347	7583	0.38	8.44	0.36	N/A	59.09	F	>2048	N/A
	136	3245	1343	7510	0.37	8.38	0.38	N/A	58.80	F	>2048	N/A
	139	3229	1337	7464	0.31	8.24	0.41	N/A	58.47	F	>2048	N/A
	140	3201	1327	7397	0.34	7.89	0.39	N/A	55.56	F	>2048	N/A
	141	2315	986	5238	0.20	4.90	0.24	N/A	38.99	F	>2048	N/A

problem	number	clauses	literals	length	Sttrim time (sec.) simp	init	check	triv	full	sat?	Sato time (sec.)	sat?
ssa6288	001	34202	10404	87321	4.98	N/A	N/A	N/A	5.27	F	>atoms	N/A
	047	34238	10410	87423	3.87	N/A	N/A	N/A	4.17	F	>atoms	N/A
	048	34202	10404	87321	4.97	N/A	N/A	N/A	5.27	F	>atoms	N/A

					Sttrim						Sato	
					time (sec.)						time	
problem	number	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
ssa7552	001	3614	1534	8311	0.24	7.47	0.04	N/A	37.67	T	>2048	N/A
	002	3614	1534	8311	0.24	7.46	0.06	N/A	39.89	T	>2048	N/A
	021	3614	1534	8309	0.56	7.39	0.04	26.31	N/A	T	>2048	N/A
	032	3605	1530	8282	0.57	7.64	0.02	N/A	31.22	T	>2048	N/A
	037	3575	1501	8248	0.20	10.31	0.09	N/A	59.86	T	0.42	T
	038	3575	1501	8248	0.53	10.28	0.04	N/A	42.69	T	0.46	T
	039	3470	1483	7952	0.45	8.78	0.07	N/A	35.31	T	0.43	T
	040	3449	1496	7883	0.37	6.59	0.03	N/A	23.99	T	>2048	N/A
	042	3598	1528	8265	0.36	7.48	0.04	N/A	36.88	T	>2048	N/A
	043	3629	1507	8389	0.17	10.51	0.08	N/A	78.75	T	0.46	T
	044	3501	1514	8039	0.64	9.85	0.08	N/A	35.56	T	>2048	N/A
	045	3586	1503	8276	0.20	10.51	0.08	N/A	69.49	T	0.46	T
	046	3586	1503	8276	0.28	10.46	0.08	N/A	78.05	T	0.44	T
	047	3449	1496	7883	0.34	9.68	0.10	N/A	44.41	T	>2048	N/A
	048	3460	1498	7910	0.46	6.49	0.04	N/A	22.21	T	>2048	N/A
	049	3458	1498	7905	0.51	9.50	0.06	37.75	N/A	T	>2048	N/A
	050	3466	1502	7928	0.50	9.56	0.07	37.81	N/A	T	>2048	N/A
	051	3466	1502	7928	0.57	9.69	0.05	N/A	35.62	T	>2048	N/A
	052	3456	1496	7895	0.37	9.82	0.05	N/A	41.79	T	>2048	N/A
	053	3598	1501	8305	0.20	10.42	0.07	N/A	79.29	T	0.46	T
	054	3300	1448	7505	0.41	8.18	0.03	N/A	25.80	T	0.44	T
	055	3442	1473	7877	0.33	8.77	0.03	N/A	35.58	T	0.41	T
	058	3622	1530	8320	0.46	10.58	0.07	N/A	55.54	T	0.46	T
	059	3523	1512	8064	0.40	10.14	0.05	N/A	50.77	T	0.43	T
	060	3446	1494	7871	0.47	9.50	0.08	36.50	N/A	T	327.44	T
	061	3586	1503	8276	0.21	10.44	0.07	N/A	71.10	T	0.46	T
	062	3586	1503	8276	0.29	10.50	0.06	N/A	78.18	T	0.45	T
	063	3289	1446	7478	0.37	8.46	0.05	N/A	33.48	T	0.42	T
	064	3579	1501	8256	0.42	10.19	0.04	N/A	44.17	T	0.44	T
	065	3579	1501	8256	0.42	10.09	0.06	N/A	44.24	T	0.43	T
	066	3463	1481	7933	0.40	8.90	0.05	N/A	35.76	T	0.42	T
	067	3300	1448	7505	0.41	8.16	0.04	N/A	25.75	T	0.42	T
	074	3458	1480	7922	0.30	9.67	0.05	N/A	43.51	T	0.43	T
	075	3282	1446	7460	0.32	9.12	0.07	N/A	32.81	T	0.41	T
	076	3282	1446	7460	0.31	9.23	0.06	N/A	33.58	T	0.42	T
	077	3282	1446	7458	0.22	9.45	0.07	N/A	34.90	T	0.41	T
	080	3635	1536	8356	0.17	10.74	0.07	N/A	68.16	T	>2048	N/A
	081	3622	1505	8369	0.35	10.02	0.04	N/A	45.90	T	0.45	T
	082	3297	1449	7503	0.34	8.38	0.04	N/A	25.99	T	0.42	T
	083	3298	1448	7500	0.41	8.24	0.04	N/A	29.26	T	0.41	T
	084	3300	1448	7505	0.41	8.07	0.04	N/A	25.81	T	0.41	T
	085	3442	1473	7877	0.33	8.71	0.04	N/A	34.92	T	0.43	T
	090	3665	1517	8489	0.36	10.23	0.03	N/A	47.02	T	>2048	N/A
	091	3297	1449	7503	0.34	8.14	0.05	N/A	25.65	T	0.38	T
	092	3298	1448	7500	0.43	8.20	0.04	N/A	25.74	T	0.45	T
	093	3300	1448	7505	0.42	8.14	0.05	N/A	25.70	T	0.44	T
	094	3442	1473	7877	0.33	8.75	0.04	N/A	36.77	T	>2048	N/A
	095	3331	1461	7598	0.41	8.39	0.04	N/A	26.14	T	0.43	T

					Sttrim						Sato	
					time (sec.)						time	
problem	number	clauses	literals	length	simp	init	check	triv	full	sat?	(sec.)	sat?
ssa7552	096	3460	1498	7910	0.44	6.37	0.02	N/A	21.81	T	>2048	N/A
	098	4123	1831	9133	0.76	7.30	0.10	N/A	32.67	T	0.98	T
	102	3623	1532	8327	0.39	10.62	0.04	N/A	53.30	T	>2048	N/A
	103	3460	1498	7910	0.44	6.41	0.02	N/A	22.21	T	>2048	N/A
	104	3458	1498	7905	0.41	9.48	0.05	38.01	N/A	T	>2048	N/A
	106	3463	1481	7935	0.40	8.93	0.04	N/A	35.83	T	0.44	T
	107	3575	1501	8248	0.53	10.31	0.04	N/A	43.02	T	0.43	T
	108	3470	1483	7952	0.43	8.86	0.05	N/A	35.11	T	0.43	T
	109	3463	1481	7935	0.40	8.82	0.04	N/A	34.54	T	0.43	T
	110	3463	1481	7935	0.40	8.89	0.03	N/A	36.23	T	0.41	T
	111	3442	1473	7877	0.33	8.87	0.04	N/A	36.06	T	0.42	T
	112	3275	1442	7444	0.55	8.24	0.05	N/A	25.84	T	0.41	T
	113	3665	1542	8441	0.40	10.82	0.08	N/A	54.87	T	>2048	N/A
	114	3460	1498	7910	0.44	6.44	0.02	N/A	21.83	T	>2048	N/A
	115	3458	1498	7905	0.40	9.40	0.05	37.78	N/A	T	>2048	N/A
	116	3466	1502	7928	0.40	9.49	0.04	37.79	N/A	T	>2048	N/A
	117	3440	1494	7861	0.60	9.60	0.05	N/A	35.47	T	>2048	N/A
	118	3440	1494	7861	0.45	6.53	0.05	N/A	21.07	T	>2048	N/A
	119	3239	1430	7326	0.34	8.84	0.08	N/A	34.32	T	0.40	T
	121	3523	1512	8064	0.44	10.14	0.05	N/A	52.03	T	0.44	T
	124	4795	2013	10986	0.57	17.05	0.05	N/A	79.28	T	37.45	T
	125	3523	1512	8064	0.41	10.18	0.05	47.62	N/A	T	0.43	T
	126	3434	1490	7842	0.46	9.21	0.05	36.74	N/A	T	324.91	T
	127	3239	1430	7324	0.24	9.17	0.05	N/A	37.43	T	0.41	T
	128	3347	1454	7599	0.30	9.62	0.05	N/A	43.71	T	0.43	T
	129	3239	1430	7326	0.30	8.87	0.06	N/A	35.17	T	0.44	T
	130	3234	1426	7307	0.40	8.67	0.04	N/A	35.38	T	0.41	T
	131	3302	1440	7483	0.34	8.97	0.05	N/A	38.74	T	0.41	T
	132	3282	1446	7460	0.35	9.14	0.07	N/A	33.94	T	0.41	T
	156	3282	1444	7461	0.41	8.37	0.03	N/A	22.49	T	0.39	T
	157	3431	1471	7850	0.33	9.09	0.03	N/A	38.02	T	>2048	N/A
	160	3126	1391	7025	0.21	8.48	0.04	N/A	31.89	T	>2048	N/A

Appendix B

Results of Various Parameter Settings

In the following tables are presented the results of running three theorem provers on the same problems as in Appendix A. Only the "bottom line" is shown for each problem: the total amount of time taken by the prover. In addition, Sttrim and Sato3 were each run with several different sets of control options, to explore their effect. For more details of the size of each problem, and the time used by the parts of Sttrim, see Appendix A.

For Sttrim, there are two control options shown, "i" and "e". The "i" option gives the upper limit on how many literals should be involved in each pass. (The letter "i" was chosen for the word "involved"). The "e" option specified the minimum number of literals that must be eliminated in each pass. (The letter "e" was chosen for the word "eliminate").

For Sato3, I varied only one control option, "g", which controls the maximum size of lemmas that can be generated.

Runs which did not find a result are indicated by the amount of time spent, preceded by a greater-than symbol. This happens in two different circumstances. The script that runs these tests one after the other sets a time limit for each problem, normally 2048 seconds. If this time limit is exceeded, the proof is terminated. If the problem that exceeded this time limit was an instance of a parameterized problem, then the remainder of the group is skipped, because such groups are run in increasing order of size, and it is expected that all later problems in the group will require even more time.

The other way that a prover can terminate without finding a result is if it decides for itself that there is no point in continuing the proof. Sato3 does this quite early in some cases, especially with the lemma size set to zero. Since the control script does not recognize these as being time-outs, it continues with the next problem in the group, even if the group is made of increasingly difficult instances of a parameterized problem.

One other artifact of the script that ran the tests is that some parameterized problems will show a time-out for some size, followed by some number of problems that were not attempted, followed by another time-out. This is a result of the problem sizes being attempted in more than one batch, and a time-out occurring in the first batch. When the second batch begins, the first size in that will also time out, resulting in the second such indication in the table.

		time (sec.)								
		Sttrim					Sato	Sato3		
	circuit	i10	i10	i10	i10	i20				
problem	size	e1	e2	e3	e4	e1	N/A	g10	g20	g50
adder	001	0.24	0.26	0.25	0.25	0.14	0.01	0.02	0.04	0.03
	002	0.52	0.54	0.55	0.54	0.27	0.03	0.05	0.05	0.04
	003	0.86	0.83	0.88	1.09	0.43	0.05	0.06	0.05	0.05
	004	1.17	1.16	1.32	1.18	0.58	0.05	0.05	0.07	0.07
	005	1.49	1.52	1.54	1.85	0.73	0.11	0.07	0.08	0.07
	006	1.84	1.83	1.92	1.96	0.90	0.20	0.09	0.08	0.09
	007	2.13	2.11	2.16	2.57	1.01	0.40	0.12	0.10	0.12
	008	2.48	2.46	2.57	2.73	1.18	0.75	0.10	0.12	0.13
	009	2.91	2.88	2.84	3.45	1.39	1.50	0.13	0.15	0.15
	010	3.14	3.16	3.25	3.55	1.52	3.15	0.15	0.16	0.16
	011	3.46	3.48	3.45	4.17	1.62	6.42	0.17	0.16	0.17
	012	3.86	3.85	3.95	4.39	1.86	13.59	0.18	0.17	0.20
	013	4.22	4.23	4.20	5.07	1.98	27.63	0.21	0.22	0.24
	014	4.55	4.56	4.55	5.18	2.11	57.21	0.22	0.22	0.21
	015	4.87	4.89	4.83	5.86	2.28	117.5	0.23	0.23	0.24
	016	5.25	5.31	5.30	6.07	2.48	234.8	0.27	0.30	0.29
	017	5.64	5.61	5.54	6.67	2.67	480.9	0.26	0.26	0.28
	018	6.15	6.11	6.00	6.97	2.85	976.4	0.30	0.32	0.34
	019	6.46	6.44	6.52	7.66	3.02	>2048	0.35	0.37	0.37
	020	6.82	6.93	6.79	7.94	3.20	N/A	0.36	0.37	0.38
	040	15.75	15.72	15.21	18.36	7.50	N/A	1.72	1.64	1.70
	080	39.60	39.71	37.91	43.94	19.02	N/A	10.21	10.63	10.66
	160	121.9	133.3	116.0	129.7	60.26	N/A	96.54	97.23	95.27

		time (sec.)								
		Sttrim					Sato	Sato3		
	circuit	i10	i10	i10	i10	i20				
problem	size	e1	e2	e3	e4	e1	N/A	g10	g20	g50
parity	001	0.06	0.05	0.08	0.08	0.08	0.01	0.02	0.03	0.03
	002	0.13	0.11	0.14	0.11	0.10	0.02	0.04	0.02	0.04
	003	0.16	0.16	0.16	0.14	0.13	0.02	0.02	0.04	0.04
	004	0.24	0.21	0.23	0.23	0.25	0.01	0.05	0.04	0.05
	005	0.41	0.37	0.41	0.39	0.31	0.02	0.04	0.04	0.03
	006	0.77	0.72	0.73	0.71	0.43	0.05	0.05	0.05	0.05
	007	1.18	1.13	1.22	1.21	0.58	0.06	0.06	0.06	0.05
	008	1.29	1.38	1.31	1.27	0.73	0.11	0.07	0.06	0.07
	009	1.43	1.38	1.45	1.43	0.88	0.18	0.07	0.07	0.07
	010	1.77	1.73	1.82	1.68	1.21	0.34	0.09	0.07	0.09
	011	2.04	1.92	2.06	1.87	1.67	0.64	0.08	0.08	0.09
	012	1.94	1.87	2.00	1.92	1.73	1.34	0.10	0.10	0.09
	013	2.28	2.20	2.29	2.23	1.92	2.53	0.10	0.10	0.10
	014	2.61	2.50	2.62	2.51	2.23	5.21	0.11	0.13	0.12
	015	2.88	2.71	2.81	2.71	2.39	10.44	0.14	0.12	0.13
	016	2.86	2.77	2.91	2.68	2.43	22.60	0.16	0.15	0.14
	017	3.17	3.11	3.22	3.00	2.56	42.86	0.14	0.15	0.17
	018	3.39	3.27	3.42	3.13	2.77	97.24	0.17	0.18	0.18
	019	3.59	3.45	3.61	3.50	2.86	182.5	0.19	0.18	0.18
	020	3.81	3.67	3.87	3.52	3.02	378.0	0.21	0.20	0.23
	021	3.92	3.78	3.96	3.83	3.31	777.2	0.20	0.21	0.23
	022	4.25	4.08	4.29	4.05	3.53	1564	0.24	0.23	0.25
	023	4.46	4.28	4.50	4.18	3.55	>2048	0.24	0.25	0.26
	024	4.51	4.34	4.62	4.40	3.73	N/A	0.27	0.27	0.29
	025	4.72	4.61	4.83	4.58	3.92	N/A	0.28	0.28	0.30
	030	5.73	5.55	5.86	5.58	5.00	N/A	0.37	0.39	0.42
	040	7.84	7.52	7.95	7.59	6.91	N/A	0.75	0.76	0.80
	080	17.14	16.58	17.32	16.51	14.74	N/A	>478.4	>612.1	698.7
	160	39.22	38.16	39.52	38.17	33.63	N/A	>538.1	>683.5	>1416

		time (sec.)								
		Sttrim					Sato	Sato3		
	circuit	i10	i10	i10	i10	i20				
problem	size	e1	e2	e3	e4	e1	N/A	g10	g20	g50
arb2	001	0.36	0.42	0.40	0.34	0.29	0.03	0.03	0.02	0.04
	002	1.88	2.28	3.30	2.55	6.50	0.04	0.04	0.06	0.04
	003	3.96	7.15	11.62	20.48	22.78	0.07	0.09	0.09	0.10
	004	6.62	16.50	16.46	31.78	28.62	0.23	0.19	0.17	0.23
	005	12.03	21.07	41.14	58.28	35.06	0.78	0.47	0.44	0.70
	006	17.49	31.66	54.81	159.4	67.26	3.08	0.99	0.74	1.43
	007	18.59	87.54	117.2	183.1	79.54	12.87	4.01	1.93	3.54
	008	24.26	106.5	125.0	242.0	105.7	52.43	7.41	4.68	7.33
	009	25.22	59.69	166.3	195.8	140.1	222.5	27.75	9.60	18.85
	010	27.14	64.82	650.6	317.2	137.3	1061	41.16	12.63	15.64
	011	35.84	75.52	1210	1439	92.38	>2048	>91.04	23.17	84.33
	012	37.15	67.93	682.4	364.9	100.9	N/A	>95.81	64.86	68.02
	013	41.47	77.02	379.8	543.9	125.9	N/A	>103.8	>132.9	80.09
	014	55.17	89.56	645.7	709.7	120.6	N/A	>119.2	>130.7	102.5
	015	52.72	105.8	432.0	568.5	244.7	N/A	>127.4	>147.2	185.5
	016	54.77	108.6	333.6	867.0	236.1	N/A	>115.5	>114.2	265.8
	017	56.64	130.6	518.6	677.3	272.0	N/A	>126.9	>144.8	333.2
	018	62.85	111.7	602.5	1273	266.3	N/A	>137.5	>126.5	>828.6
	019	68.56	139.4	661.5	724.6	272.6	N/A	>141.0	>139.4	>659.9
	020	80.49	158.8	424.2	>2048	238.2	N/A	>107.2	>132.6	>478.9
	040	187.7	379.6	1587	N/A	956.8	N/A	>231.2	>262.2	>288.9
	080	>mem	928.6	>2048	N/A	>2048	N/A	>543.4	>548.5	>550.8
	160	>mem	>mem	N/A	N/A	N/A	N/A	>373.2	>1120	>1137

problem	circuit size	time (sec.)								
		Sttrim					Sato	Sato3		
		i10 e1	i10 e2	i10 e3	i10 e4	i20 e1	N/A	g10	g20	g50
arb2fp	001	0.37	0.36	0.39	0.34	0.28	0.02	0.03	0.04	0.04
	002	1.94	2.06	3.34	2.34	5.94	0.03	0.06	0.05	0.06
	003	3.73	7.61	16.73	20.35	24.04	0.07	0.10	0.09	0.09
	004	7.20	10.46	27.43	22.09	23.85	0.23	0.20	0.18	0.20
	005	10.89	17.80	60.97	47.66	65.58	0.75	0.42	0.36	0.77
	006	13.75	32.00	120.4	180.5	64.64	3.16	1.18	0.99	1.55
	007	18.23	31.48	187.7	409.3	76.19	13.34	2.52	1.85	3.75
	008	23.90	41.04	275.6	1156	113.9	52.26	5.30	3.27	6.89
	009	28.10	40.36	222.8	813.4	156.3	227.0	21.01	6.13	11.12
	010	33.19	50.63	368.5	404.5	157.6	1195	25.05	9.71	16.51
	011	39.49	54.41	387.3	1135	178.5	>2048	>106.6	13.92	27.38
	012	44.10	70.86	1189	409.8	226.8	N/A	>109.1	44.83	34.11
	013	48.34	75.20	952.1	599.3	278.1	N/A	104.0	122.0	80.42
	014	50.79	74.55	541.5	>2048	327.3	N/A	>128.4	210.0	94.75
	015	55.07	111.5	802.8	N/A	370.6	N/A	>131.2	>276.1	207.1
	016	60.70	87.80	1440	N/A	384.8	N/A	>127.4	>275.3	365.5
	017	65.93	92.22	>2048	N/A	427.7	N/A	>132.5	>266.0	701.2
	018	76.33	147.6	N/A	N/A	429.9	N/A	>102.6	>287.0	>1472
	019	74.74	108.2	N/A	N/A	483.5	N/A	>145.5	>348.7	1167
	020	86.94	158.6	N/A	N/A	479.5	N/A	>117.4	>168.6	1434
	040	195.2	282.6	N/A	N/A	1136	N/A	>246.1	>117.0	>368.1
	080	>mem	744.3	N/A	N/A	>2048	N/A	>542.5	>547.9	>696.2
	160	>mem	>mem	N/A	N/A	N/A	N/A	>348.9	>312.8	>1247

problem	circuit size	time (sec.)								
		Sttrim					Sato	Sato3		
		i10 e1	i10 e2	i10 e3	i10 e4	i20 e1	N/A	g10	g20	g50
arb2lp	005:001	0.87	0.83	1.04	0.87	1.74	0.07	0.11	0.10	0.10
	005:002	18.92	33.72	59.01	142.6	93.57	2.74	0.92	0.97	1.75
	005:003	20.32	55.87	543.5	302.0	98.09	2.85	0.96	1.26	1.50
	005:004	21.65	68.11	170.0	149.5	66.12	3.06	0.96	0.94	1.75

problem	circuit size	time (sec.)								
		Sttrim					Sato	Sato3		
		i10 e1	i10 e2	i10 e3	i10 e4	i20 e1	N/A	g10	g20	g50
arb2lp	005:004	21.65	68.11	170.0	149.5	66.12	3.06	0.96	0.94	1.75
	006:004	33.46	202.9	828.3	874.4	203.4	17.27	3.09	2.16	6.82
	007:004	40.66	401.0	2013	2020	232.1	88.03	8.02	5.16	16.31
	008:004	66.91	521.9	>2048	>2048	371.4	451.9	16.00	13.34	19.89
	009:004	89.68	1846	N/A	N/A	676.8	>2048	69.16	46.15	68.94
	010:004	162.5	>2048	N/A	N/A	2007	N/A	>138.1	90.17	203.3
	011:004	100.4	>2048	N/A	N/A	>2048	N/A	>163.6	130.4	188.1
	012:004	161.8	N/A	N/A	N/A	N/A	N/A	>123.9	566.6	512.8
	013:004	1084	N/A	N/A	N/A	N/A	N/A	>87.22	>695.7	915.5
	014:004	695.0	N/A	N/A	N/A	N/A	N/A	>99.73	>484.0	>2048
	015:004	439.6	N/A	N/A	N/A	N/A	N/A	>115.5	>288.6	N/A

problem	circuit size	time (sec.)								
		Sttrim					Sato	Sato3		
		i10 e1	i10 e2	i10 e3	i10 e4	i20 e1	N/A	g10	g20	g50
arb2lp	010:001	1.98	1.96	2.09	2.11	4.68	0.20	0.20	0.21	0.22
	010:002	51.22	216.2	580.5	>2048	356.9	>2048	>119.3	60.43	140.1
	010:003	65.90	751.9	>2048	N/A	1075	N/A	>181.3	105.5	184.3
	010:004	162.5	>2048	N/A	N/A	2007	N/A	>138.1	90.17	203.3
	010:005	659.4	N/A	N/A	N/A	916.3	N/A	169.7	114.3	92.06
	010:006	228.1	N/A	N/A	N/A	610.1	N/A	138.9	58.35	136.6
	010:007	397.5	N/A	N/A	N/A	>2048	N/A	179.7	45.51	147.2
	010:008	126.4	N/A	N/A	N/A	N/A	N/A	103.8	36.86	140.2

problem	TPTP size	time (sec.) Sttrim i10 e1	i10 e2	i10 e3	i10 e4	i20 e1	Sato N/A	Sato3 g10	g20	g50
SYN001-1	001	0.00	0.00	0.00	0.00	0.00	0.02	0.02	0.02	0.02
	002	0.04	0.06	0.06	0.05	0.04	0.01	0.03	0.02	0.02
	003	0.06	0.05	0.08	0.06	0.04	0.01	0.03	0.04	0.03
	004	0.07	0.07	0.07	0.06	0.05	0.02	0.03	0.03	0.03
	005	0.12	0.08	0.12	0.11	0.09	0.03	0.02	0.03	0.02
	006	0.18	0.18	0.16	0.19	0.16	0.02	0.04	0.02	0.02
	007	0.34	0.33	0.35	0.34	0.32	0.03	0.04	0.04	0.03
	008	0.77	0.75	0.75	0.74	0.72	0.04	0.07	0.06	0.05
	009	1.78	1.73	1.74	1.71	1.73	0.07	0.09	0.09	0.09
	010	4.21	4.11	4.11	4.11	4.11	0.15	0.17	0.17	0.17
	011	9.48	9.57	9.79	9.47	9.67	0.29	0.37	0.37	0.37
	012	21.81	21.93	22.58	21.88	22.30	0.62	0.77	0.75	0.79
	013	49.71	49.76	51.36	49.79	50.59	1.30	1.71	1.62	1.66
	014	113.5	114.1	116.9	114.1	115.3	2.80	3.57	3.48	3.52

problem	TPTP size	time (sec.) Sttrim i10 e1	i10 e2	i10 e3	i10 e4	i20 e1	Sato N/A	Sato3 g10	g20	g50
SYN003-1	002	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.03	0.01
	003	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.02	0.04
	004	0.00	0.00	0.00	0.00	0.00	0.01	0.05	0.04	0.03
	005	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.03	0.03
	006	0.00	0.00	0.00	0.00	0.00	0.01	0.04	0.02	0.03
	007	0.00	0.00	0.00	0.00	0.00	0.01	0.04	0.03	0.04
	008	0.00	0.00	0.00	0.00	0.00	0.02	0.02	0.02	0.04
	009	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.03	0.04
	010	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.03	0.04
	020	0.00	0.00	0.00	0.01	0.00	0.01	0.03	0.03	0.04
	040	0.02	0.01	0.01	0.01	0.02	0.04	0.04	0.04	0.04
	080	0.04	0.04	0.04	0.04	0.04	0.04	0.07	0.08	0.07
	160	0.12	0.12	0.12	0.12	0.13	0.07	0.10	0.10	0.11
	320	0.51	0.50	0.50	0.50	0.50	0.16	0.19	0.20	0.22
	640	2.16	2.12	2.13	2.19	2.13	0.28	0.38	0.37	0.39

problem	TPTP size	time (sec.) Sttrim i10 e1	i10 e2	i10 e3	i10 e4	i20 e1	Sato N/A	Sato3 g10	g20	g50
SYN004-1	002	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.02	0.04
	003	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.03	0.02
	004	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.02	0.03
	005	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.03	0.04
	006	0.00	0.00	0.00	0.00	0.00	0.02	0.02	0.03	0.04
	007	0.00	0.00	0.00	0.00	0.00	0.02	0.03	0.04	0.03
	008	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.03	0.03
	009	0.00	0.00	0.00	0.00	0.00	0.02	0.03	0.03	0.03
	010	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.02	0.03
	020	0.00	0.01	0.00	0.00	0.01	0.02	0.03	0.02	0.04
	040	0.02	0.02	0.02	0.02	0.02	0.02	0.04	0.03	0.04
	080	0.07	0.06	0.06	0.06	0.06	0.03	0.05	0.05	0.04
	160	0.22	0.22	0.23	0.23	0.23	0.05	0.05	0.06	0.09
	320	0.91	0.89	0.89	0.93	0.91	0.10	0.14	0.13	0.11
	640	4.42	4.33	4.34	4.41	4.39	0.19	0.23	0.22	0.24

		time (sec.)								
		Sttrim					Sato	Sato3		
	TPTP	i10	i10	i10	i10	i20				
problem	size	e1	e2	e3	e4	e1	N/A	g10	g20	g50
SYN008-1	N/A	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.01	0.03
SYN010-1	005:005	0.00	0.00	0.00	0.00	0.00	0.03	0.03	0.04	0.02
SYN011-1	N/A	0.06	0.08	0.07	0.07	0.06	0.01	0.04	0.03	0.03
SYN028-1	N/A	0.00	0.00	0.00	0.00	0.00	0.02	0.04	0.04	0.02
SYN029-1	N/A	0.04	0.06	0.05	0.06	0.06	0.02	0.02	0.02	0.01
SYN030-1	N/A	0.07	0.08	0.07	0.06	0.07	0.01	0.02	0.02	0.02
SYN032-1	N/A	0.07	0.08	0.05	0.07	0.07	0.02	0.04	0.03	0.03
SYN040-1	N/A	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.04	0.02
SYN041-1	N/A	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.01	0.01
SYN044-1	N/A	0.07	0.05	0.05	0.07	0.07	0.01	0.02	0.04	0.01
SYN045-1	N/A	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.03	0.04
SYN046-1	N/A	0.00	0.00	0.00	0.00	0.00	0.01	0.05	0.02	0.02
SYN047-1	N/A	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.02	0.02
SYN063-1	N/A	0.05	0.06	0.06	0.06	0.05	0.01	0.02	0.04	0.02

		time (sec.)								
		Sttrim					Sato	Sato3		
	TPTP	i10	i10	i10	i10	i20				
problem	size	e1	e2	e3	e4	e1	N/A	g10	g20	g50
SYN085-1	001	0.00	0.00	0.00	0.00	0.00	0.02	0.03	0.02	0.02
	002	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.03	0.03
	003	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.03	0.04
	004	0.00	0.00	0.00	0.00	0.00	0.01	0.04	0.04	0.03
	005	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.02	0.02
	006	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.02	0.03
	007	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.03	0.03
	008	0.00	0.00	0.00	0.00	0.00	0.03	0.01	0.03	0.04
	009	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.02	0.02
	010	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.02	0.03
	020	0.00	0.00	0.00	0.00	0.00	0.02	0.01	0.03	0.02
	040	0.00	0.00	0.00	0.00	0.00	0.03	0.03	0.03	0.02
	080	0.00	0.00	0.00	0.00	0.00	0.04	0.03	0.04	0.03
	160	0.00	0.00	0.00	0.00	0.00	0.09	0.04	0.05	0.04
	320	0.00	0.00	0.01	0.01	0.00	0.29	0.06	0.05	0.05
	640	0.01	0.01	0.02	0.02	0.01	1.13	0.14	0.12	0.14

		time (sec.)								
		Sttrim					Sato	Sato3		
	TPTP	i10	i10	i10	i10	i20				
problem	size	e1	e2	e3	e4	e1	N/A	g10	g20	g50
SYN086-1	001	0.00	0.00	0.00	0.00	0.00	0.01	N/A	N/A	N/A
	002	0.00	0.00	0.00	0.00	0.00	0.02	0.02	0.03	0.03
	003	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.02	0.01
	004	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.04	0.01
	005	0.00	0.00	0.00	0.00	0.00	0.03	0.02	0.04	0.03
	006	0.00	0.00	0.00	0.00	0.00	0.02	0.03	0.05	0.04
	007	0.00	0.00	0.00	0.00	0.00	0.02	0.04	0.02	0.03
	008	0.00	0.01	0.00	0.00	0.00	0.02	0.04	0.04	0.04
	009	0.00	0.01	0.00	0.00	0.00	0.02	0.05	0.05	0.04
	010	0.00	0.00	0.00	0.01	0.01	0.03	0.06	0.04	0.04
	020	0.03	0.03	0.02	0.03	0.03	0.08	0.13	0.12	0.12
	040	0.14	0.14	0.15	0.14	0.14	0.34	0.42	0.45	0.44
	080	0.58	0.57	0.59	0.57	0.58	1.32	1.71	1.74	1.74
	160	>2048	>2048	>2048	>2048	>2048	>atoms	N/A	N/A	N/A

		time (sec.)								
		Sttrim					Sato	Sato3		
	TPTP	i10	i10	i10	i10	i20				
problem	size	e1	e2	e3	e4	e1	N/A	g10	g20	g50
SYN087-1	001	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.02	0.04
	002	0.00	0.00	0.00	0.00	0.00	0.02	0.04	0.02	0.02
	003	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.02	0.03
	004	0.00	0.00	0.00	0.00	0.00	0.01	0.04	0.03	0.03
	005	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.03	0.02
	006	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.02	0.04
	007	0.00	0.00	0.00	0.00	0.00	0.03	0.04	0.02	0.04
	008	0.00	0.00	0.00	0.00	0.00	0.02	0.03	0.03	0.02
	009	0.00	0.00	0.00	0.00	0.00	0.02	0.05	0.03	0.02
	010	0.00	0.00	0.00	0.00	0.00	0.02	0.03	0.04	0.03
	020	0.01	0.00	0.01	0.00	0.00	0.03	0.03	0.04	0.05
	040	0.01	0.01	0.01	0.01	0.01	0.04	0.06	0.08	0.07
	080	0.02	0.02	0.02	0.02	0.02	0.08	0.12	0.10	0.10
	160	0.04	0.03	0.03	0.04	0.04	0.13	0.18	0.19	0.21
	320	0.08	0.08	0.08	0.08	0.08	0.26	0.34	0.34	0.35
	640	0.18	0.17	0.17	0.17	0.17	0.55	0.68	0.72	0.73

		time (sec.)								
		Sttrim					Sato	Sato3		
	TPTP	i10	i10	i10	i10	i20				
problem	size	e1	e2	e3	e4	e1	N/A	g10	g20	g50
SYN089-1	001	0.00	0.00	0.00	0.00	0.00	0.02	0.02	0.01	0.01
	002	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.02	0.03
	003	0.00	0.00	0.00	0.00	0.00	0.02	0.04	0.03	0.03
	004	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.04	0.04
	005	0.00	0.00	0.00	0.00	0.00	0.02	0.02	0.04	0.02
	006	0.00	0.00	0.00	0.00	0.00	0.04	0.04	0.04	0.04
	007	0.00	0.00	0.00	0.01	0.00	0.03	0.04	0.03	0.04
	008	0.01	0.01	0.00	0.00	0.00	0.02	0.04	0.04	0.04
	009	0.01	0.01	0.01	0.01	0.01	0.03	0.04	0.04	0.04
	010	0.01	0.01	0.01	0.01	0.01	0.04	0.06	0.05	0.05
	020	0.07	0.07	0.08	0.08	0.08	0.10	0.14	0.14	0.14
	040	0.63	0.64	0.63	0.63	0.62	0.37	0.49	0.49	0.52
	080	4.82	4.82	4.83	4.83	4.86	1.46	1.91	1.91	1.98
	160	>2048	>2048	>2048	>2048	>2048	>atoms	N/A	N/A	N/A

		time (sec.)								
		Sttrim					Sato	Sato3		
	TPTP	i10	i10	i10	i10	i20				
problem	size	e1	e2	e3	e4	e1	N/A	g10	g20	g50
SYN090-1	001	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.03	0.03
	002	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.02	0.04
	003	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.03	0.02
	004	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.02	0.02
	005	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.04	0.03
	006	0.00	0.00	0.00	0.00	0.00	0.02	0.04	0.04	0.02
	007	0.00	0.00	0.00	0.00	0.00	0.02	0.03	0.04	0.04
	008	0.00	0.01	0.01	0.01	0.00	0.03	0.04	0.04	0.03
	009	0.01	0.00	0.00	0.01	0.01	0.03	0.03	0.04	0.04
	010	0.01	0.01	0.01	0.00	0.00	0.02	0.04	0.04	0.04
	020	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.05	0.04
	040	0.11	0.11	0.12	0.12	0.11	0.04	0.05	0.07	0.07
	080	0.48	0.47	0.47	0.47	0.47	0.09	0.12	0.13	0.10
	160	1.98	1.97	1.95	1.98	1.98	0.14	0.22	0.21	0.20
	320	9.10	8.96	9.02	9.09	9.10	0.30	0.33	0.38	0.37
	640	39.14	38.46	38.64	39.16	39.08	0.61	0.71	0.73	0.77

problem	TPTP size	time (sec.) Sttrim i10 e1	i10 e2	i10 e3	i10 e4	i20 e1	Sato N/A	Sato3 g10	g20	g50
SYN091-1	001	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.02	0.01
	002	0.07	0.07	0.07	0.07	0.06	0.00	0.03	0.01	0.01
	003	0.09	0.11	0.12	0.11	0.10	0.01	0.03	0.03	0.03
	004	0.44	0.46	0.54	0.56	0.12	0.02	0.03	0.03	0.05
	005	0.98	1.26	0.68	0.73	3.71	0.02	0.04	0.03	0.05
	006	5.23	7.57	8.49	29.35	16.44	0.03	0.05	0.04	0.06
	007	7.27	13.39	4.36	34.60	11.30	0.04	0.07	0.05	0.05
	008	34.92	66.75	339.8	96.15	144.3	0.05	0.07	0.07	0.08
	009	38.47	78.21	438.6	676.1	59.95	0.06	0.08	0.09	0.09
	010	186.2	450.2	>2048	1669	377.8	0.07	0.09	0.09	0.11
	011	303.2	289.6	>2048	1332	546.8	0.07	0.10	0.12	0.12
	012	>2048	>2048	N/A	>2048	>2048	0.06	0.13	0.13	0.14
	013	N/A	N/A	N/A	N/A	N/A	0.10	0.15	0.16	0.16
	014	N/A	N/A	N/A	N/A	N/A	0.13	0.19	0.18	0.20
	015	N/A	N/A	N/A	N/A	N/A	0.16	0.20	0.22	0.23

problem	TPTP size	time (sec.) Sttrim i10 e1	i10 e2	i10 e3	i10 e4	i20 e1	Sato N/A	Sato3 g10	g20	g50
SYN092-1	001	0.06	0.07	0.08	0.06	0.06	0.02	0.02	0.04	0.03
	002	0.08	0.07	0.07	0.07	0.10	0.02	0.02	0.04	0.03
	003	0.16	0.18	0.21	0.20	0.11	0.04	0.04	0.03	0.02
	004	0.20	0.20	0.24	0.29	0.12	0.02	0.03	0.04	0.03
	005	0.21	0.21	0.39	0.30	0.21	0.02	0.04	0.04	0.05
	006	0.23	0.21	0.41	0.32	2.62	0.01	0.04	0.03	0.05
	007	0.26	0.26	0.44	0.34	4.24	0.02	0.04	0.04	0.06
	008	0.27	0.28	0.44	0.37	4.23	0.04	0.04	0.05	0.04
	009	0.31	0.31	0.45	0.38	4.32	0.03	0.04	0.04	0.05
	010	0.34	0.33	0.48	0.41	4.30	0.03	0.04	0.04	0.06
	011	0.35	0.35	0.54	0.45	5.18	0.04	0.05	0.05	0.06
	012	0.36	0.39	0.58	0.48	5.28	0.04	0.05	0.05	0.06
	013	0.39	0.41	0.60	0.51	5.44	0.04	0.07	0.05	0.07
	014	0.38	0.45	0.61	0.50	5.59	0.04	0.06	0.06	0.07
	015	0.44	0.49	0.68	0.54	5.39	0.05	0.05	0.05	0.07
	020	0.59	0.62	0.79	0.68	5.39	N/A	0.07	0.09	0.07
	040	1.36	1.46	1.61	1.46	6.28	N/A	0.16	0.16	0.14
	080	3.97	4.14	4.18	4.06	8.80	N/A	0.40	0.45	0.39
	160	12.98	13.33	13.17	13.03	17.83	N/A	1.54	1.50	1.52

problem	TPTP size	time (sec.) Sttrim i10 e1	i10 e2	i10 e3	i10 e4	i20 e1	Sato N/A	Sato3 g10	g20	g50
SYN093-1	001	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.02	0.03
	002	0.09	0.08	0.10	0.09	0.09	0.01	0.02	0.02	0.04
	003	0.14	0.13	0.13	0.14	0.25	0.03	0.03	0.02	0.03
	004	0.25	0.23	0.26	0.23	0.66	1.37	0.04	0.04	0.05
	005	0.41	0.39	0.36	0.39	2.84	208.3	0.04	0.05	0.03
	006	0.49	0.55	0.54	0.52	3.19	>2048	0.05	0.06	0.06
	007	0.77	0.79	0.78	0.73	4.21	N/A	0.07	0.08	0.08
	008	1.03	1.09	1.04	0.96	5.65	N/A	0.09	0.07	0.10
	009	1.39	1.42	1.41	1.39	6.94	N/A	0.14	0.13	0.14
	010	1.86	1.82	1.83	1.71	8.55	N/A	0.19	0.18	0.20
	011	2.30	2.51	2.40	2.24	12.65	>2048	0.23	0.24	0.26
	012	2.97	3.13	3.06	2.83	14.48	N/A	0.29	0.32	0.34
	013	3.75	3.93	3.86	3.48	17.44	N/A	0.42	0.45	0.46
	014	4.70	4.90	4.75	4.43	19.79	N/A	0.52	0.54	0.54
	015	5.73	5.96	5.90	5.32	22.69	N/A	0.73	0.71	0.76
	020	14.72	14.27	13.85	12.91	35.20	N/A	2.17	2.17	2.23
	040	>mem	>mem	>mem	>mem	>mem	N/A	25.17	24.97	25.05
	080	>2048	>2048	>2048	>2048	>2048	N/A	223.5	226.9	N/A

		time (sec.)								
		Sttrim					Sato	Sato3		
	TPTP	i10	i10	i10	i10	i20				
problem	size	e1	e2	e3	e4	e1	N/A	g10	g20	g50
SYN094-1	001	0.06	0.07	0.07	0.06	0.07	0.00	0.03	0.03	0.04
	002	0.11	0.11	0.12	0.11	0.19	0.02	0.02	0.03	0.03
	003	0.15	0.14	0.16	0.16	0.19	0.17	0.05	0.04	0.05
	004	0.19	0.18	0.19	0.18	0.42	2.29	0.03	0.05	0.04
	005	0.24	0.22	0.23	0.21	0.50	42.27	0.07	0.06	0.07
	006	0.29	0.30	0.29	0.27	0.58	801.4	0.10	0.11	0.11
	007	0.35	0.35	0.32	0.33	0.65	>2048	0.39	0.39	0.41
	008	0.41	0.39	0.38	0.36	0.68	N/A	1.66	1.73	1.83
	009	0.46	0.47	0.40	0.42	0.75	N/A	3.11	3.07	3.42
	010	0.50	0.51	0.45	0.45	0.78	N/A	>18.23	>18.04	>18.70
	020	1.22	1.22	1.04	0.98	1.42	N/A	>18.74	>18.68	>19.66
	040	3.35	3.50	2.67	2.61	3.02	N/A	>27.29	>27.20	>29.70
	080	10.38	10.71	8.12	7.89	8.19	N/A	>34.81	>36.37	>37.38
	160	37.90	38.94	27.38	27.32	25.96	N/A	>76.79	>76.61	>78.14

		time (sec.)								
		Sttrim					Sato	Sato3		
	TPTP	i10	i10	i10	i10	i20				
problem	size	e1	e2	e3	e4	e1	N/A	g10	g20	g50
SYN097-1	001	0.00	0.00	0.00	0.00	0.00	0.02	0.03	0.03	0.03
	002	0.11	0.09	0.09	0.09	0.08	0.01	0.04	0.04	0.02
	003	0.18	0.18	0.17	0.17	0.29	0.02	0.04	0.04	0.04
	004	0.45	0.34	0.32	0.31	0.71	0.02	0.04	0.04	0.04
	005	0.81	0.82	0.75	0.75	2.96	0.03	0.04	0.05	0.07
	006	1.20	1.37	1.31	1.22	3.50	0.06	0.10	0.10	0.09
	007	1.75	1.91	1.86	1.75	6.78	0.11	0.13	0.14	0.14
	008	2.78	2.90	2.57	2.57	9.43	0.20	0.20	0.19	0.20
	009	3.81	3.85	3.64	3.47	11.40	0.62	0.28	0.29	0.29
	010	5.31	5.40	5.09	4.91	14.18	1.38	0.39	0.41	0.39
	011	7.12	7.05	6.68	6.34	17.80	3.61	0.57	0.59	0.62
	012	9.41	9.38	8.62	8.53	21.30	9.98	0.87	0.78	0.81
	013	12.14	12.02	11.29	10.80	24.93	42.04	1.08	1.06	1.08
	014	15.28	15.20	14.19	13.51	30.83	84.35	1.40	1.40	1.44
	015	19.82	19.26	18.24	17.11	35.01	239.1	1.85	1.87	1.90
	016	25.48	24.50	22.90	21.47	40.74	884.4	2.42	2.42	2.47
	017	30.70	30.27	28.07	26.26	49.14	>2048	3.03	3.00	3.07
	018	37.78	37.09	35.15	32.51	57.12	N/A	3.70	3.75	3.83
	019	46.02	45.99	43.87	39.69	76.33	N/A	4.46	4.43	4.48
	020	55.07	55.76	51.16	50.90	99.45	N/A	5.33	5.37	5.50
	040	>mem	>mem	>mem	>mem	>mem	N/A	53.39	53.04	53.99

		time (sec.)								
		Sttrim					Sato	Sato3		
	TPTP	i10	i10	i10	i10	i20				
problem	size	e1	e2	e3	e4	e1	N/A	g10	g20	g50
SYN098-1	001	0.08	0.08	0.08	0.08	0.06	0.02	0.02	0.03	0.02
	002	0.14	0.13	0.14	0.10	0.21	0.02	0.03	0.05	0.04
	003	0.27	0.22	0.27	0.29	0.68	0.04	0.04	0.05	0.06
	004	0.36	0.36	0.38	0.36	1.04	0.04	0.12	0.08	0.10
	005	0.45	0.44	0.46	0.39	1.15	0.05	0.18	0.20	0.20
	006	0.58	0.57	0.56	0.58	1.49	0.07	0.43	0.47	0.51
	007	0.73	0.73	0.72	0.62	1.65	0.15	2.01	2.30	2.58
	008	0.83	0.84	0.82	0.71	1.84	0.31	12.49	13.93	15.43
	009	1.04	1.04	0.92	0.88	1.92	0.82	28.27	32.26	35.90
	010	1.18	1.23	1.07	0.97	2.19	1.95	>197.3	>218.8	>232.0
	011	1.30	1.43	1.22	1.10	2.70	5.39	>196.5	>225.8	>232.3
	012	1.43	1.62	1.43	1.27	2.93	13.12	>218.3	>252.7	>257.3
	013	1.70	1.83	1.60	1.42	3.05	33.68	>273.3	>311.4	>320.0
	014	1.83	1.95	1.71	1.51	3.32	90.83	>301.8	>337.2	>347.7
	015	2.01	2.20	2.04	1.73	3.56	237.0	>325.4	>367.9	>372.8
	016	2.27	2.52	2.13	1.93	3.81	608.9	>378.9	>419.5	>426.2
	017	2.47	2.63	2.23	2.07	3.96	1582	>417.3	>446.2	>459.6
	018	2.68	2.97	2.46	2.25	4.19	>2048	>454.3	>482.4	>509.3
	019	2.93	3.15	2.70	2.44	4.45	N/A	>517.3	>551.6	>565.1
	020	3.10	3.44	2.85	2.65	4.76	>2048	>564.5	>578.2	>593.9
	040	10.13	10.66	8.61	8.10	11.01	N/A	>2048	>2048	>2048
	080	36.54	38.42	28.66	27.86	30.78	N/A	N/A	N/A	N/A
	160	>mem	>mem	105.4	105.9	101.7	N/A	N/A	N/A	N/A
	320	>mem	>mem	>mem	>mem	>mem	N/A	N/A	N/A	N/A
	640	>mem	>mem	>mem	>mem	>mem	N/A	N/A	N/A	N/A

		time (sec.)								
		Sttrim					Sato	Sato3		
	TPTP	i10	i10	i10	i10	i20				
problem	size	e1	e2	e3	e4	e1	N/A	g10	g20	g50
SYN302-1	001	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.01	0.03
	002	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.02	0.02
	003	0.00	0.00	0.00	0.00	0.00	0.02	0.04	0.04	0.04
	004	0.00	0.00	0.00	0.00	0.00	0.02	0.04	0.03	0.03
	005	0.00	0.00	0.01	0.00	0.00	0.03	0.05	0.04	0.05
	006	0.00	0.01	0.01	0.00	0.00	0.04	0.05	0.07	0.05
	007	0.01	0.01	0.01	0.01	0.01	0.05	0.07	0.07	0.06
	008	0.01	0.01	0.01	0.01	0.01	0.05	0.09	0.09	0.09
	009	0.02	0.02	0.02	0.02	0.02	0.06	0.09	0.10	0.10
	010	0.02	0.02	0.02	0.02	0.02	0.08	0.12	0.10	0.12
	020	0.10	0.10	0.10	0.10	0.10	0.29	0.39	0.42	0.42
	040	0.41	0.43	0.41	0.41	0.40	1.22	1.55	1.52	1.59

		time (sec.)								
		Sttrim					Sato	Sato3		
		i10	i10	i10	i10	i20				
problem	number	e1	e2	e3	e4	e1	N/A	g10	g20	g50
bf0432	005	53.57	102.9	156.0	571.7	227.8	225.0	3.09	2.43	2.41
	007	585.5	>2048	>2048	>2048	>2048	>2048	21.19	19.61	14.85
	011	393.0	>2048	>2048	>2048	>2048	167.5	0.60	0.64	0.74
	015	319.3	>2048	>2048	>2048	>2048	>2048	18.69	18.37	21.93
	017	675.6	>2048	>2048	>2048	>2048	>2048	18.91	13.11	13.20
	033	23.75	47.88	660.0	214.7	203.7	>2048	19.28	6.81	3.51
	035	46.99	128.9	1342	>2048	626.4	>2048	7.52	7.04	16.25
	039	26.64	65.38	207.3	>2048	224.7	>2048	7.42	17.18	9.55
	041	34.26	85.24	698.5	>2048	580.9	>2048	4.67	4.21	3.70
	043	63.77	147.6	725.4	>2048	346.1	>2048	8.11	8.47	8.34
	048	5.61	5.96	5.78	6.44	27.56	3.33	0.24	0.28	0.26
	053	347.9	>2048	>2048	>2048	>2048	203.1	0.89	1.01	1.02
	070	553.5	>2048	>2048	>2048	>2048	174.2	0.77	0.87	0.91
	077	842.2	>2048	>2048	>2048	857.7	>2048	14.06	11.48	12.27
	079	318.9	>2048	>2048	>2048	1053	>2048	20.46	11.46	12.19
	091	817.3	>2048	>2048	>2048	>2048	67.58	0.71	0.66	0.69
	103	576.2	>2048	>2048	>2048	804.6	128.8	0.80	0.86	0.89
	122	738.5	>2048	>2048	>2048	>2048	>2048	1.24	1.13	1.14
	135	8.62	9.10	9.99	14.37	78.83	212.8	0.48	0.51	0.40
	138	8.30	8.50	9.58	14.54	91.94	383.9	0.34	0.36	0.39
	139	8.55	9.48	11.50	16.71	104.8	4.18	0.35	0.42	0.45

		time (sec.)								
		Sttrim					Sato	Sato3		
		i10	i10	i10	i10	i20				
problem	number	e1	e2	e3	e4	e1	N/A	g10	g20	g50
bf1355	075	53.45	62.06	84.90	194.1	445.1	>2048	1.00	1.05	1.05
	076	93.01	75.15	95.86	233.8	397.8	>2048	1.14	1.17	1.16
	119	39.20	39.38	40.30	86.44	233.8	>2048	1.27	1.28	1.31
	120	34.53	34.71	34.22	37.19	86.42	>2048	1.53	1.53	1.58
	123	63.01	108.5	>2048	330.3	839.7	>2048	1.63	1.60	1.59
	124	66.41	136.5	75.73	503.3	405.9	>2048	1.78	2.00	2.01
	127	62.34	62.77	82.42	152.4	562.5	>2048	1.69	1.65	1.73
	128	131.9	>2048	>2048	204.4	752.5	>2048	1.49	1.48	1.52
	131	35.24	34.77	35.51	85.74	615.7	>2048	0.99	1.01	1.04
	132	30.12	30.14	30.05	50.33	>2048	>2048	1.04	1.08	1.14
	143	62.65	77.85	116.6	>2048	513.2	>2048	1.59	1.53	1.63
	144	66.60	78.87	122.9	>2048	314.7	>2048	1.39	1.39	1.43
	147	38.50	38.04	38.92	31.38	613.1	>2048	1.00	1.02	1.02
	148	32.32	32.01	31.79	65.03	155.8	>2048	1.01	1.04	1.08
	151	59.04	66.54	86.46	>2048	476.1	>2048	1.78	1.75	1.77
	152	65.94	72.84	103.0	657.3	386.1	>2048	1.46	1.50	1.51
	159	38.15	37.64	38.94	31.06	352.7	>2048	0.94	1.02	1.05
	160	33.44	33.56	33.43	32.06	145.5	>2048	1.02	1.01	1.01
	167	57.56	61.51	137.8	273.6	698.8	>2048	1.59	1.51	1.55
	168	69.28	78.84	141.6	358.8	1213	>2048	1.57	1.62	1.65
	175	34.85	35.00	37.15	102.5	146.5	>2048	1.05	1.07	1.09
	176	39.33	39.67	40.57	82.19	101.9	>2048	1.15	1.13	1.21
	227	37.75	37.49	39.71	195.6	81.10	>2048	1.15	1.10	1.15
	228	34.21	34.36	34.88	33.89	132.5	>2048	1.75	1.79	1.79
	239	91.25	136.5	133.1	1467	233.7	>2048	1.62	1.62	1.64
	240	123.4	147.2	>2048	669.7	>2048	>2048	1.69	1.49	1.50
	243	45.29	59.97	105.9	185.4	1070	>2048	6.21	5.27	4.04
	291	38.40	37.83	39.62	49.62	112.2	>2048	1.05	1.09	1.12
	292	34.64	34.45	34.85	37.22	126.6	>2048	1.05	1.08	1.15
	295	48.04	48.46	53.77	26.06	117.2	>2048	0.95	0.90	0.98
	296	33.58	32.98	46.86	50.37	129.5	>2048	0.88	0.93	0.91
	303	38.44	38.48	40.25	82.46	124.4	>2048	1.06	1.06	1.13
	304	49.01	49.13	54.96	67.67	143.8	>2048	1.11	1.12	1.14
	315	60.42	59.89	70.05	766.6	422.4	>2048	1.20	1.22	1.23
	316	103.4	125.3	123.5	193.8	635.6	>2048	1.57	1.65	1.63
	319	57.58	56.91	119.2	186.2	454.8	>2048	1.48	1.47	1.50
	320	89.17	113.5	183.2	109.9	428.5	>2048	1.30	1.32	1.30
	323	32.62	32.33	27.81	31.29	151.5	>2048	1.14	1.01	1.10
	324	52.81	52.82	59.09	110.0	102.8	>2048	1.00	1.05	1.07
	327	64.51	65.20	106.8	460.9	302.3	>2048	1.16	1.25	1.22
	328	71.67	72.56	118.8	271.9	993.4	>2048	1.28	1.31	1.37
	347	48.59	48.70	55.87	101.9	404.2	>2048	1.59	1.62	1.62
	348	95.03	113.4	187.7	196.5	>2048	>2048	1.62	1.60	1.60
	351	45.95	45.67	65.00	157.3	156.8	>2048	0.95	0.96	1.00
	352	38.75	38.25	38.92	33.90	98.77	>2048	0.96	0.94	1.02
	355	62.08	79.10	1108	181.0	413.5	>2048	1.54	1.58	1.61
	356	93.76	160.3	317.8	693.1	412.0	>2048	1.57	1.83	1.86
	359	67.82	76.91	80.83	317.8	214.6	>2048	1.30	1.27	1.35

problem	number	time (sec.)								
		Sttrim					Sato	Sato3		
		i10 e1	i10 e2	i10 e3	i10 e4	i20 e1	N/A	g10	g20	g50
bf1355	360	127.6	199.3	504.8	893.3	599.6	>2048	1.75	1.80	1.82
	363	48.28	48.72	52.56	35.68	119.9	3.91	0.88	0.93	0.89
	364	39.25	38.79	40.79	49.73	295.4	0.98	0.87	0.89	0.93
	445	57.78	62.49	73.53	89.50	505.2	>2048	1.50	1.50	1.53
	446	76.35	101.4	113.3	1237	1604	>2048	1.79	1.79	1.89
	449	35.11	35.26	36.79	36.45	123.4	11.72	0.98	1.01	1.04
	450	38.51	38.17	38.60	81.26	582.3	1.39	0.89	0.89	0.94
	453	59.83	62.87	116.4	78.80	917.1	>2048	1.19	1.21	1.21
	454	53.82	55.01	133.5	297.4	682.6	>2048	2.05	2.04	2.05
	457	56.94	61.02	104.8	>2048	434.0	>2048	1.73	1.78	1.81
	458	65.37	69.11	133.5	227.3	>2048	>2048	1.72	1.72	1.76
	461	58.54	77.19	63.20	462.5	250.7	>2048	1.66	1.67	1.72
	462	83.06	102.2	132.4	370.2	1113	>2048	1.33	1.38	1.41
	465	54.87	57.41	81.00	255.3	238.1	>2048	1.10	1.11	1.12
	466	61.97	66.70	149.1	249.1	447.9	>2048	1.18	1.22	1.21
	469	53.97	75.89	74.95	220.4	622.7	>2048	1.28	1.30	1.36
	470	72.97	103.9	150.0	526.6	756.5	>2048	1.30	1.33	1.41
	473	59.25	64.38	55.62	224.9	378.3	>2048	1.05	1.10	1.06
	474	77.57	103.2	252.5	372.8	519.6	>2048	1.26	1.22	1.28
	477	33.92	34.21	34.63	88.64	>2048	>2048	1.30	1.26	1.29
	478	54.08	54.25	56.28	49.99	104.1	>2048	1.35	1.37	1.42
	481	50.61	50.77	64.98	>2048	325.1	>2048	1.41	1.37	1.45
	482	63.34	80.84	123.0	591.6	425.4	>2048	1.58	1.58	1.63
	485	59.95	75.75	66.23	318.4	440.1	>2048	1.45	1.48	1.52
	486	69.90	72.82	>2048	166.4	1072	>2048	1.49	1.57	1.64
	489	64.60	138.5	65.09	302.8	1358	>2048	1.75	1.79	1.84
	490	293.2	147.2	108.0	940.0	633.8	>2048	1.52	1.51	1.55
	493	36.83	37.12	39.48	34.43	118.0	4.96	0.85	0.91	0.91
	494	52.93	54.23	58.08	48.51	565.8	1.01	0.91	0.86	0.88
	497	53.11	52.63	99.13	>2048	1251	>2048	1.43	1.42	1.49
	498	64.45	68.05	278.1	616.3	752.7	>2048	1.57	1.61	1.61
	505	48.95	52.07	61.35	134.4	338.8	>2048	1.04	1.06	1.10
	506	59.27	65.98	151.2	440.1	559.7	>2048	1.05	1.06	1.14
	509	53.15	50.69	62.28	>2048	319.9	>2048	1.16	1.21	1.24
	510	59.25	69.35	132.3	1612	375.6	>2048	1.07	1.09	1.13
	513	53.67	53.73	63.08	138.9	221.8	>2048	1.50	1.49	1.56
	514	63.04	65.39	132.7	171.0	550.6	>2048	1.48	1.53	1.55
	517	84.97	83.99	169.6	186.4	450.0	>2048	1.13	1.15	1.16
	518	53.48	54.92	89.77	>2048	388.2	>2048	0.97	0.95	1.02
	521	69.86	83.94	97.95	168.5	352.5	>2048	1.15	1.13	1.22
	522	62.28	70.77	100.9	155.6	404.2	1683	1.06	1.01	1.05
	525	65.45	75.32	105.5	412.9	338.8	>2048	0.97	0.95	0.98
	526	50.93	70.69	83.71	116.8	440.0	704.8	0.93	0.96	0.99
	529	38.27	37.82	39.05	91.60	439.3	>2048	1.00	0.99	1.07
	530	29.12	28.76	28.76	33.96	574.9	>2048	1.06	1.00	1.10
	533	30.29	30.01	29.86	30.73	78.72	>2048	1.02	0.94	1.07
	534	31.74	31.98	31.53	36.95	434.9	>2048	1.00	1.03	1.06
	537	51.66	52.11	64.87	222.3	544.8	>2048	0.83	0.83	0.85

		time (sec.)								
		Sttrim					Sato	Sato3		
		i10	i10	i10	i10	i20				
problem	number	e1	e2	e3	e4	e1	N/A	g10	g20	g50
bf1355	538	49.91	50.74	65.32	118.4	108.9	675.0	0.83	0.79	0.84
	557	58.54	74.08	90.02	150.9	535.8	>2048	1.39	1.34	1.40
	558	105.9	176.8	769.9	661.5	423.8	>2048	1.76	1.84	1.87
	561	77.76	66.06	635.3	498.4	419.2	>2048	1.43	1.44	1.46
	562	186.4	117.0	275.9	419.3	419.8	>2048	1.46	1.45	1.51
	565	94.42	153.0	143.0	101.2	606.9	>2048	1.08	1.05	1.06
	566	58.26	60.11	72.21	197.0	344.3	>2048	1.05	1.05	1.08
	573	54.12	62.05	119.7	111.6	627.8	>2048	1.07	1.07	1.11
	574	74.94	73.39	95.95	162.3	199.1	>2048	1.16	1.12	1.20
	577	54.86	60.29	64.08	108.7	244.4	>2048	1.16	1.18	1.18
	578	77.67	71.55	106.9	151.6	966.8	>2048	1.32	1.34	1.35
	581	60.49	63.52	101.1	266.6	323.4	>2048	1.29	1.29	1.37
	582	84.76	78.61	87.14	205.0	515.3	>2048	1.51	1.46	1.62
	585	35.07	34.85	35.74	68.52	92.04	>2048	1.06	1.05	1.10
	586	36.34	36.04	38.03	48.30	157.2	>2048	1.03	1.03	1.02
	589	27.97	27.93	27.49	94.44	602.2	>2048	0.97	1.02	1.01
	590	30.04	29.69	29.72	86.50	104.0	>2048	1.00	1.03	1.04
	593	53.44	57.81	91.86	83.86	400.0	>2048	1.14	1.12	1.13
	594	92.94	73.42	102.0	168.6	295.4	>2048	1.18	1.12	1.19
	597	59.72	65.16	73.42	152.9	563.3	>2048	1.31	1.33	1.37
	598	83.67	71.48	184.1	243.0	>2048	>2048	1.44	1.46	1.45
	601	58.29	63.24	81.07	954.5	>2048	>2048	1.29	1.27	1.30
	602	102.9	79.26	100.9	232.9	270.0	>2048	1.26	1.26	1.29
	617	49.27	49.48	55.21	62.26	112.4	80.22	0.93	0.89	0.99
	618	52.70	53.00	59.50	108.9	91.25	180.2	0.96	0.98	1.01
	621	34.34	34.31	34.94	66.48	98.88	94.67	1.15	1.18	1.22
	622	38.08	37.68	39.27	50.33	120.3	201.0	1.20	1.17	1.16
	625	103.6	85.94	122.1	353.0	333.8	>2048	1.33	1.33	1.38
	626	258.2	126.0	122.0	373.5	415.4	>2048	1.56	1.58	1.61
	629	45.88	45.25	61.20	75.89	306.2	>2048	1.41	1.37	1.44
	630	79.98	99.97	105.2	135.6	1581	>2048	1.38	1.42	1.41
	633	53.00	50.41	55.30	71.76	329.9	>2048	2.02	1.95	2.04
	634	86.57	78.01	1049	>2048	647.6	>2048	1.61	1.65	1.64
	637	64.37	175.3	62.86	194.9	655.4	>2048	1.16	1.16	1.18
	638	>2048	177.1	89.46	744.9	530.2	>2048	1.46	1.48	1.52
	653	90.09	119.9	176.2	868.4	223.2	>2048	1.30	1.33	1.34
	654	155.1	200.7	146.4	845.1	236.1	>2048	1.19	1.20	1.16
	661	110.8	141.4	473.1	370.1	674.9	>2048	1.27	1.28	1.31
	662	151.9	110.9	142.1	876.1	571.1	>2048	1.20	1.16	1.21
	665	74.04	75.30	106.9	>2048	691.6	>2048	1.41	1.40	1.46
	666	79.55	68.48	102.9	>2048	583.5	>2048	1.35	1.39	1.47
	669	82.95	91.15	429.8	>2048	613.2	>2048	1.34	1.32	1.36
	670	66.76	69.17	415.3	1425	375.3	>2048	1.37	1.41	1.45
	681	25.82	26.35	25.71	29.37	314.2	>2048	0.97	0.96	0.99
	682	29.71	29.85	29.31	31.03	144.1	>2048	1.12	1.16	1.11
	701	60.49	66.33	128.3	262.1	>2048	>2048	1.06	1.03	1.10
	702	59.43	63.95	98.73	158.1	370.5	>2048	1.03	1.04	1.02
	741	71.92	81.02	110.4	665.6	>2048	>2048	1.09	1.07	1.12

		time (sec.)								
		Sttrim					Sato	Sato3		
		i10	i10	i10	i10	i20				
problem	number	e1	e2	e3	e4	e1	N/A	g10	g20	g50
bf1355	742	66.46	81.75	111.8	346.8	213.4	>2048	1.07	1.09	1.12
	749	52.78	51.80	58.83	88.98	310.1	>2048	1.46	1.53	1.46
	750	35.65	35.51	34.97	33.87	491.7	>2048	1.57	1.54	1.54
	757	113.6	145.5	641.1	868.5	579.8	>2048	1.43	1.41	1.48
	758	88.57	104.4	205.9	343.4	562.5	>2048	1.48	1.48	1.50

		time (sec.)								
		Sttrim					Sato	Sato3		
		i10	i10	i10	i10	i20				
problem	number	e1	e2	e3	e4	e1	N/A	g10	g20	g50
bf2670	001	30.03	30.24	36.53	41.84	248.3	>2048	0.55	0.57	0.62
	007	8.03	7.62	7.91	8.20	28.38	0.18	0.36	0.37	0.37
	008	7.11	7.10	7.29	7.37	19.65	0.18	0.33	0.33	0.37
	033	8.42	8.41	8.66	8.72	32.38	0.18	0.36	0.31	0.35
	035	8.30	8.16	8.13	8.52	18.96	146.2	0.43	0.43	0.44
	038	14.73	16.54	16.63	68.65	52.57	0.24	0.46	0.49	0.49
	042	12.36	13.07	14.50	18.86	51.04	0.21	0.45	0.45	0.47
	051	27.26	27.92	34.40	37.20	186.4	>2048	0.59	0.59	0.63
	052	30.35	32.92	41.22	51.33	259.2	>2048	0.65	0.68	0.71
	121	7.55	7.62	7.50	8.02	24.39	0.17	0.34	0.35	0.35
	126	7.46	7.52	7.48	7.96	25.90	0.17	0.33	0.35	0.36
	130	46.53	39.47	62.16	69.11	356.9	>2048	40.96	14.94	10.40
	143	7.41	7.41	7.55	7.62	29.37	0.20	0.33	0.35	0.36
	144	6.96	7.03	7.12	7.02	26.44	0.19	0.34	0.34	0.32
	145	7.03	7.07	7.10	6.98	20.80	0.18	0.34	0.34	0.37
	187	41.48	43.71	58.70	134.4	367.4	>2048	0.50	0.49	0.52
	189	41.15	45.24	50.41	86.64	301.5	>2048	3.06	5.82	2.55
	190	39.76	46.29	53.00	175.7	343.7	>2048	7.08	4.38	3.52
	200	41.19	46.85	55.99	123.9	343.6	>2048	0.52	0.54	0.55
	202	40.84	43.69	47.27	59.60	331.1	>2048	3.37	4.62	2.86
	203	40.43	45.85	54.64	61.95	377.5	>2048	40.50	4.44	2.71
	208	35.85	35.70	54.23	66.85	340.2	>2048	0.51	0.52	0.52
	240	54.94	68.43	247.9	896.6	406.3	>2048	4.36	4.45	3.01
	241	47.97	55.07	124.1	808.4	351.0	>2048	25.12	4.32	2.92
	244	73.82	108.5	129.6	161.7	365.9	>2048	51.26	50.93	31.45
	248	97.08	492.0	408.9	1425	607.0	>2048	41.78	122.2	71.10
	269	22.64	23.33	26.18	30.93	148.9	>2048	0.57	0.55	0.55
	275	40.13	47.28	99.93	906.2	350.1	32.03	1.33	2.02	2.49
	278	28.79	28.31	32.33	52.34	237.8	0.50	0.80	0.75	0.73
	400	56.52	71.17	117.8	1480	352.4	>2048	170.0	111.3	83.89
	404	26.89	29.87	42.11	59.14	154.8	>2048	27.36	10.68	21.82
	414	60.22	75.10	370.0	729.0	309.4	0.32	0.74	0.71	0.77
	449	7.89	7.84	7.74	8.08	26.03	0.18	0.34	0.37	0.38
	461	11.80	11.85	15.18	14.68	44.61	0.23	0.47	0.44	0.46
	468	20.24	19.93	22.47	21.98	148.3	80.42	0.66	0.67	0.72
	483	37.00	51.04	49.38	110.6	287.3	0.50	1.05	1.06	1.10
	487	39.79	41.00	76.20	64.39	265.5	>2048	1.13	1.05	1.08
	491	46.51	55.14	80.21	299.5	389.9	>2048	105.3	5.35	4.73
	492	35.40	37.42	37.34	56.98	293.5	>2048	2.35	1.88	1.23
	493	45.41	56.57	58.20	108.2	337.1	>2048	39.55	5.32	6.30
	494	35.98	37.36	40.37	54.89	285.6	>2048	2.36	1.04	1.12
	495	37.08	42.88	58.32	74.28	361.3	>2048	0.53	0.51	0.52
	501	57.44	72.46	120.6	266.8	383.7	>2048	7.33	3.30	3.71
	502	50.66	63.59	83.14	406.9	404.7	>2048	6.79	>266.1	3.82
	510	82.70	155.1	230.2	>2048	433.8	>2048	21.50	4.68	7.71
	511	57.35	75.76	144.0	373.2	378.8	>2048	23.98	3.32	2.24
	522	40.52	44.57	54.92	66.54	328.3	>2048	3.79	3.03	2.84
	524	39.55	41.11	44.72	89.65	308.5	>2048	3.08	2.89	2.14

		time (sec.)								
		Sttrim					Sato	Sato3		
		i10	i10	i10	i10	i20				
problem	number	e1	e2	e3	e4	e1	N/A	g10	g20	g50
bf2670	532	37.98	43.88	51.81	75.43	272.5	>2048	3.16	2.27	2.13
	533	35.97	41.93	51.11	80.14	263.2	>2048	4.78	2.35	2.50
	538	23.82	25.35	27.63	35.19	206.1	>2048	0.62	0.66	0.66
	546	35.62	38.34	55.34	68.23	279.5	>2048	2.57	1.45	1.30
	547	34.70	36.38	58.16	64.94	238.5	>2048	2.58	1.51	1.55

		time (sec.)								
		Sttrim					Sato	Sato3		
		i10	i10	i10	i10	i20				
problem	number	e1	e2	e3	e4	e1	N/A	g10	g20	g50
ssa0432	001	5.78	6.08	6.00	7.08	59.68	199.3	0.23	0.30	0.30
	002	6.16	6.31	6.52	7.73	39.20	169.3	0.31	0.30	0.30
	003	5.34	5.52	5.44	7.30	50.97	221.5	0.33	0.38	0.37
	004	5.79	6.15	6.04	6.94	60.08	183.7	0.37	0.44	0.45
	011	5.39	5.56	5.39	7.32	50.95	196.1	0.34	0.38	0.38
	014	8.19	8.57	9.00	17.19	47.38	396.7	0.39	0.57	0.59
	015	6.17	6.24	6.48	7.72	39.27	162.6	0.28	0.28	0.32

		time (sec.)								
		Sttrim					Sato	Sato3		
		i10	i10	i10	i10	i20				
problem	number	e1	e2	e3	e4	e1	N/A	g10	g20	g50
ssa2670	127	44.74	51.38	186.4	102.5	349.6	>2048	13.49	5.40	4.18
	128	43.70	50.26	93.22	618.4	317.0	>2048	12.37	14.40	4.08
	129	44.05	51.13	93.14	614.8	313.7	>2048	12.55	13.99	4.07
	130	43.80	50.66	186.3	102.1	354.5	>2048	12.93	5.35	4.14
	131	49.23	68.76	168.3	514.4	437.0	>2048	110.5	37.25	33.29
	132	44.41	50.51	93.25	617.5	316.8	>2048	12.28	14.11	4.15
	133	44.79	51.20	186.8	102.1	351.2	>2048	12.90	5.35	4.16
	134	59.09	69.16	87.44	417.1	325.1	>2048	83.54	84.29	82.89
	136	58.80	70.06	87.63	424.0	333.0	>2048	>233.4	51.49	40.62
	139	58.47	69.70	86.80	421.8	330.6	>2048	>236.9	50.88	40.27
	140	55.56	59.94	121.8	393.7	303.9	>2048	>232.6	40.79	26.96
	141	38.99	52.56	67.69	712.6	198.1	>2048	114.1	16.19	14.83

		time (sec.)								
		Sttrim					Sato	Sato3		
		i10	i10	i10	i10	i20				
problem	number	e1	e2	e3	e4	e1	N/A	g10	g20	g50
ssa6288	001	5.27	5.26	5.27	5.29	5.25	>atoms	3.70	3.77	3.98
	047	4.17	4.16	4.17	4.18	4.16	>atoms	3.79	3.82	4.13
	048	5.27	5.25	5.23	5.28	5.27	>atoms	3.74	3.81	4.07

		time (sec.)								
		Sttrim					Sato	Sato3		
		i10	i10	i10	i10	i20				
problem	number	e1	e2	e3	e4	e1	N/A	g10	g20	g50
ssa7552	001	37.67	47.52	73.00	228.7	253.6	>2048	0.57	0.59	0.56
	002	39.89	48.87	98.26	299.8	255.0	>2048	0.56	0.57	0.61
	021	26.31	33.78	31.49	40.67	136.4	>2048	0.53	0.48	0.52
	032	31.22	41.25	56.63	44.17	159.7	>2048	0.51	0.49	0.55
	037	59.86	71.76	88.07	137.3	302.7	0.42	0.55	0.57	0.55
	038	42.69	45.28	63.04	62.19	256.9	0.46	0.50	0.50	0.54
	039	35.31	35.58	39.13	46.79	223.9	0.43	0.54	0.57	0.53
	040	23.99	27.43	85.76	38.97	201.2	>2048	0.54	0.55	0.56
	042	36.88	45.16	81.26	132.9	254.2	>2048	0.57	0.60	0.62
	043	78.75	81.89	139.8	182.4	400.9	0.46	0.59	0.60	0.63
	044	35.56	38.77	39.98	44.54	231.5	>2048	0.60	0.57	0.59
	045	69.49	88.75	143.8	177.8	613.0	0.46	0.58	0.59	0.62
	046	78.05	80.54	114.7	161.8	367.9	0.44	0.48	0.53	0.53
	047	44.41	45.63	47.43	72.48	272.4	>2048	0.56	0.57	0.60
	048	22.21	24.72	25.69	31.37	164.1	>2048	0.54	0.52	0.61
	049	37.75	40.75	39.56	48.12	240.3	>2048	0.53	0.58	0.58
	050	37.81	40.65	40.43	47.45	240.7	>2048	0.52	0.55	0.56
	051	35.62	38.22	39.99	42.77	232.0	>2048	0.57	0.58	0.60
	052	41.79	44.38	51.86	50.34	261.7	>2048	0.52	0.56	0.58
	053	79.29	76.00	102.1	143.9	430.8	0.46	0.52	0.53	0.54
	054	25.80	25.96	25.05	25.60	106.2	0.44	0.48	0.47	0.49
	055	35.58	34.73	39.02	44.59	224.4	0.41	0.52	0.53	0.58
	058	55.54	83.04	98.47	105.3	310.2	0.46	0.60	0.59	0.64
	059	50.77	50.68	57.94	71.23	287.3	0.43	0.54	0.55	0.57
	060	36.50	39.50	40.35	41.01	230.6	327.4	0.55	0.56	0.59
	061	71.10	73.58	116.8	165.9	393.5	0.46	0.64	0.59	0.66
	062	78.18	83.10	102.6	140.1	352.7	0.45	0.51	0.54	0.54
	063	33.48	33.74	35.44	33.93	190.8	0.42	0.45	0.49	0.48
	064	44.17	43.61	60.53	62.59	259.8	0.44	0.52	0.51	0.53
	065	44.24	46.32	50.30	58.00	275.8	0.43	0.61	0.61	0.59
	066	35.76	35.81	41.05	49.49	209.9	0.42	0.62	0.68	0.67
	067	25.75	25.84	24.29	25.65	106.2	0.42	0.46	0.48	0.48
	074	43.51	44.46	42.15	58.23	261.8	0.43	0.53	0.57	0.58
	075	32.81	32.23	31.21	31.47	142.9	0.41	0.44	0.45	0.51
	076	33.58	33.82	35.62	43.81	218.6	0.42	0.47	0.53	0.55
	077	34.90	35.03	36.15	40.24	211.3	0.41	0.45	0.51	0.50
	080	68.16	71.66	141.0	113.5	435.1	>2048	0.70	0.72	0.72
	081	45.90	49.52	60.16	57.58	260.6	0.45	0.54	0.52	0.57
	082	25.99	25.75	25.02	25.79	107.0	0.42	0.45	0.47	0.50
	083	29.26	25.92	25.21	25.81	106.3	0.41	0.47	0.48	0.48
	084	25.81	25.88	24.81	25.87	106.9	0.41	0.46	0.50	0.49
	085	34.92	36.98	40.03	50.56	218.7	0.43	0.48	0.50	0.52
	090	47.02	53.63	64.56	86.97	266.9	>2048	0.51	0.51	0.55
	091	25.65	25.85	27.80	25.62	111.7	0.38	0.46	0.48	0.49
	092	25.74	26.17	24.27	26.06	106.3	0.45	0.44	0.48	0.47
	093	25.70	25.79	24.10	25.59	106.0	0.44	0.46	0.46	0.50
	094	36.77	36.74	42.12	100.3	232.6	>2048	0.49	0.49	0.51
	095	26.14	25.95	25.71	26.01	106.8	0.43	0.46	0.47	0.48

problem	number	time (sec.)								
		Sttrim					Sato	Sato3		
		i10 e1	i10 e2	i10 e3	i10 e4	i20 e1	N/A	g10	g20	g50
ssa7552	096	21.81	24.43	25.66	31.47	164.8	>2048	0.55	0.55	0.56
	098	32.67	47.87	41.40	45.82	108.4	0.98	0.52	0.51	0.57
	102	53.30	55.47	111.3	200.9	337.1	>2048	0.74	0.75	0.78
	103	22.21	24.72	26.11	31.82	163.7	>2048	0.54	0.57	0.58
	104	38.01	41.39	39.95	51.58	241.2	>2048	0.52	0.57	0.60
	106	35.83	38.01	39.32	41.33	211.9	0.44	0.54	0.53	0.57
	107	43.02	47.22	47.62	58.49	254.2	0.43	0.51	0.51	0.57
	108	35.11	35.34	39.18	46.68	222.6	0.43	0.51	0.54	0.55
	109	34.54	35.02	39.11	41.66	229.4	0.43	0.52	0.54	0.58
	110	36.23	37.83	38.70	41.07	211.4	0.41	0.54	0.55	0.58
	111	36.06	35.10	40.52	44.84	215.9	0.42	0.54	0.55	0.61
	112	25.84	25.49	25.56	26.17	107.8	0.41	0.44	0.43	0.47
	113	54.87	66.55	64.32	136.0	360.1	>2048	0.77	0.72	0.79
	114	21.83	24.67	25.71	31.64	164.2	>2048	0.55	0.54	0.55
	115	37.78	40.96	40.39	48.26	241.5	>2048	0.57	0.58	0.59
	116	37.79	41.00	40.01	46.77	240.5	>2048	0.57	0.58	0.60
	117	35.47	38.11	38.13	42.45	229.9	>2048	0.56	0.54	0.55
	118	21.07	24.04	25.97	29.73	162.4	>2048	0.54	0.52	0.58
	119	34.32	35.35	39.11	38.76	206.5	0.40	0.49	0.48	0.51
	121	52.03	55.24	61.64	148.5	272.4	0.44	0.54	0.56	0.57
	124	79.28	140.4	139.3	1409	352.9	37.45	0.63	0.65	0.69
	125	47.62	54.49	71.62	61.79	288.6	0.43	0.55	0.54	0.59
	126	36.74	39.69	38.26	41.19	231.1	324.9	0.53	0.50	0.55
	127	37.43	34.65	34.10	37.61	237.6	0.41	0.47	0.46	0.50
	128	43.71	45.77	51.24	56.71	276.6	0.43	0.53	0.50	0.53
	129	35.17	34.75	35.00	40.07	218.2	0.44	0.45	0.50	0.49
	130	35.38	37.31	33.31	40.15	205.5	0.41	0.45	0.49	0.48
	131	38.74	38.47	42.49	46.83	234.0	0.41	0.48	0.53	0.54
	132	33.94	33.70	35.38	43.77	221.2	0.41	0.52	0.51	0.57
	156	22.49	22.24	21.74	22.45	78.02	0.39	0.47	0.47	0.48
	157	38.02	37.27	44.25	47.52	220.6	>2048	0.51	0.54	0.55
	160	31.89	31.02	30.64	37.22	185.1	>2048	0.49	0.49	0.47

Bibliography

[1] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, C-27(6):509–16, June 1978.

[2] S. B. Akers. Functional testing with binary decision diagrams. In *International Symposium on Fault-Tolerant Computing Digest 1978*, pages 75–82, 1978.

[3] C. Alpert and A. Kahng. Geometric embeddings for faster and better multi-way netlist partitioning. In *30th ACM/IEEE DAC Proceedings, 1993*, pages 743–8, 1993.

[4] M. Beardslee, B. Lin, and A. Sangiovanni-Vincentelli. Communication based logic partitioning. In *Euro-DAC 92 / Euro-VHDL 92*, pages 32–7, 1992.

[5] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*. Springer-Verlag, 1999.

[6] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, Inc., New York, 1979.

[7] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*, volume 23 of *Perspectives in Computing*. Academic Press, Inc., Harcourt Brace Jovanovich, 1988.

[8] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in FORTRAN. In *Proc. 1985 IEEE Int. Symp. Circuits and Sys.*, June 1985. Special Session.

[9] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, C-35(8):677–91, Aug. 1986.

[10] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–70, June 1992.

[11] J. R. Burch. Using BDDs to verify multipliers. In *28th ACM/IEEE DAC Proceedings, 1991*, 1991. Paper 24.

[12] P. K. Chan, M. D. Schlag, and J. Y. Zien. Spectral k-way ratio-cut partitioning and clustering. In *30th ACM/IEEE DAC Proceedings, 1993*, pages 749–54, 1993.

[13] C.-L. Chang and R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Harcourt Brace Jovanovich, 1973.

[14] H. Chu. *Semantically Guided First-Order Theorem Proving with Hyper-Linking.* PhD thesis, University of North Carolina at Chapel Hill, 1994.

[15] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Prog. Lang. Syst.*, 16(5):1512–42, Sept. 1994.

[16] A. Cohn. A proof of correctness of the Viper microprocessor: The first level. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, The Kluwer International Series in Engineering and Computer Science, chapter 2, pages 27–71. Kluwer Academic Publishers, 1988.

[17] J. Cong and M. Smith. A parallel bottom-up clustering algorithm with applications to circuit partitioning in VLSI design. In *30th ACM/IEEE DAC Proceedings, 1993*, pages 755–60, 1993.

[18] R. Constable, S. Allen, H. Bromley, W. Cleaveland, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1986.

[19] W. Cullyer. Implementing safety-critical systems: The Viper microprocessor. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, The Kluwer International Series in Engineering and Computer Science, chapter 1, pages 1–25. Kluwer Academic Publishers, 1988.

[20] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–7, July 1962.

[21] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–15, July 1960.

[22] D. Duffy. *Principles of Automated Theorem Proving.* Wiley, 1991.

[23] C. Fiduccia and R. Mattheyses. A linear-time heuristic for improving network partitions. In *19th ACM DAC Proceedings, 1982*, pages 175–81, 1982.

[24] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, 1990.

[25] A. V. Gelder and Y. K. Tsuki. *Satisfiability Testing with More Reasoning and Less Guessing*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 559–586. American Mathematical Society, 1993.

[26] M. J. Gordon. Hol: A proof generating system for higher-order logic. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, The Kluwer International Series in Engineering and Computer Science, chapter 3, pages 73–128. Kluwer Academic Publishers, 1988.

[27] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. Edinburgh LCF. *Lecture Notes in Computer Science*, 78, 1979.

[28] S. W. Hadley, B. L. Mark, and A. Vannelli. An efficient eigenvector approach for finding netlist partitions. *IEEE Transactions on Computer-Aided Design*, pages 885–92, July 1992.

[29] L. Hagen and A. B. Kahng. A new approach to effective circuit clustering. In *IEEE/ACM International Conference on Computer-Aided Design ICCAD-92 Digest of Technical Papers 1992*, pages 422–7, 1992.

[30] L. Hagen and A. B. Kahng. New spectral methods for ratio-cut partitioning and clustering. *IEEE Transactions on Computer-Aided Design*, pages 1074–85, Sept. 1992.

[31] K. Hall. An r-dimensional quadratic placement algorithm. *Management Science*, 17(3):219–29, Nov. 1970.

[32] S. Hazelhurst and C.-J. H. Seger. A simple theorem prover based on symbolic trajectory evaluation and OBDDs. Technical Report TR-93-41, Computer Science Department, University of British Columbia, 1993.

[33] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[34] A. J. Hu and D. L. Dill. Reducing BDD size by exploiting functional dependencies. In *30th ACM/IEEE DAC Proceedings, 1993*, pages 266–71, 1993.

[35] S. C. Johnson. Hierarchical clustering schemes. *Psychometrika*, 32(3):241–254, Sept. 1967.

[36] J. J. Joyce. Formal verification and implementation of a microprocessor. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, The Kluwer International Series in Engineering and Computer Science, chapter 4, pages 129–57. Kluwer Academic Publishers, 1988.

[37] J. J. Joyce and C.-J. H. Seger. Linking BDD-based symbolic evaluation to interactive theorem-proving. In *30th ACM/IEEE DAC Proceedings, 1993*, pages 469–74, 1993.

[38] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal*, pages 291–307, Feb. 1970.

[39] T. Larrabee. Test pattern generation using boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, II(1):4–15, Jan. 1992.

[40] S.-J. Lee. *CLIN: An Automated Reasoning System Using Clause Linking*. PhD thesis, University of North Carolina at Chapel Hill, 1990.

[41] S. Minato. *Binary Decision Diagrams and Applications for VLSI CAD*. Kluwer Academic Publishers, 1996.

[42] J. S. Moore. Introduction to the OBDD algorithm for the ATP community. *Journal of Automatic Reasoning*, 12(1):33–45, Feb. 1994.

[43] T. J. Park and A. V. Gelder. Partitioning methods for satisfiability testing on large formulas. In M. A. McRobbie and J. K. Slaney, editors, *Proceedings of the Thirteenth International Conference on Automated Deduction*, pages 748–762. Springer, July 1996.

[44] L. C. Paulson. *Isabelle - A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1994.

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

[46] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[47] C.-J. Seger. An introduction to formal hardware verification. Technical Report TR-92-13, Computer Science Department, University of British Columbia, 1992.

[48] M. Shih and E. S. Kuh. Quadratic boolean programming for performance-driven system partitioning. In *30th ACM/IEEE DAC Proceedings, 1993*, pages 761–65, 1993.

[49] J. Staunstrup. IFIP WG 10.2 collection of circuit verification examples. FTP, Nov. 1993.

[50] G. Suttcliffe and C. B. Suttner. The TPTP problem library. Technical Report 96/09, James Cook University, Australia, 1996.

[51] T. E. Uribe and M. E. Stickel. Ordered binary decision diagrams and the Davis-Putnam procedure. In J.-P. Jouannaud, editor, *Constraints in Computational Logics, First International Conference, CCL '94*, pages 34–49. Springer, Sept. 1994.

[52] Y.-C. Wei and C.-K. Cheng. Towards efficient hierarchical designs by ratio-cut partitioning. In *IEEE International Conference on Computer-Aided Design ICCAD-89 Digest of Technical Papers 1989*, pages 298–301, 1989.

[53] C.-W. Yeh and C.-K. Cheng. A general purpose multiple way partitioning algorithm. In *28th ACM/IEEE DAC Proceedings, 1991*, pages 421–6, 1991.

[54] H. Zhang. SATO: a decision procedure for propositional logic. *Association for Automated Reasoning Newsletter*, (22):1–3, Mar. 1993.

[55] H. Zhang. SATO: An efficient propositional prover. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, pages 272–275, July 1997.

[56] H. Zhang and M. E. Stickel. Implementing the Davis-Putnam algorithm by tries. Technical report, The University of Iowa, 1994.