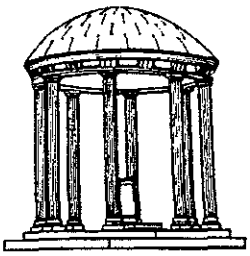
Inference by Clause Linking

TR90-022

May, 1990

David Plaisted
Shie-Jue Lee



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

Inference by Clause Linking*

David A. Plaisted and Shie-Jue Lee
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599–3175

May 16, 1990

Abstract

We present a novel theorem proving method based on the idea of never combining parts of different assertions (clauses) into a single assertion. This avoids some combinatorial problems with other existing strategies. We mention some refinements of this idea which improve its performance. Results of an implementation are discussed, including tests on logic puzzles, set theory problems, and temporal logic theorems. We discuss methods used to make the prover self-controlling to an unusually high degree so that novice users need not specify which strategy to use. The system is particularly suited to the incorporation of specialized decision procedures. This and other extensions are discussed. Also, this work is placed in the context of general issues and developments in theorem proving and artificial intelligence. This paper is an extended version of a chapter to appear in the book **INTELLIGENT SYSTEMS**: *State of the art and future directions* by Zbigniew Ras and Maria Zemankova.

1 Introduction

Virtually all theorem proving methods combine parts of separate formula during the process of inference. This combination can cause combinatorial problems, since a small number of formulas may combine in many different ways. We present a new method based on the idea of never combining parts of different formulas. This method has been implemented and shown to be of wide applicability in several different problem areas. We will present this method and results of an implementation, as well as comment on general issues in theorem proving and artificial intelligence.

Our method, which we call the hyper-linking strategy, has some analogies to human problem solving methods. It consists of a two stage process. The first stage generates selected instances of the formulas given; the second stage tests the formulas for satisfiability using a case analysis method. The first stage may be considered as creative and the second as analytic. The method is also related to the associative network model of artificial

*This research was supported by NSF under grant CCR–8802282.

intelligence, in which nodes are connected by links and activations can spread from node to node along the links. For us, the nodes are assertions, the links are unifications between parts of formulas, and the activations are substitutions that propagate from node to node. There is an obvious if possibly superficial resemblance to neural networks. The similarities with semantic networks in artificial intelligence, and with connection graph methods in theorem proving, are closer.

In [Pla88][NP90] we presented about the ultimate extension of the subgoaling idea to theorem proving in full first order logic. Subgoaling is a common problem solving paradigm. However, the extension to full first order logic is difficult, and we now feel that subgoaling may not be the right model for general problem solving in full first order logic, except for problems that fit naturally into that form; these are the Horn or near-Horn problems [HW74].

We present our method in the context of Skolemized first order formulas (clause form) and satisfiability. However, the ideas seem to be more general than this, and should apply to non-clausal proof systems which use explicit quantifiers. The idea is to systematically search for parts of formulas that can be made identical by suitable instantiations of variables.

One objective of this presentation is to show the amount of detailed knowledge needed to do general problem solving well. There is a surprising amount of complexity and sophistication in our theorem prover. It seems that mechanical theorem proving is a difficult area, and that only a complex program can do it well. If our prover were extended to deal with equality or higher order logic or extended in other ways, the complexity would be even greater; now it works on pure first order logic. This may be one reason that progress in theorem proving has been slow, namely, the amount of knowledge needed.

This suggests that the lack of success of general problem solvers in artificial intelligence may have been due as much to the lack of general problem solving knowledge as to the lack of specialized knowledge. Such a general problem solver requires a large body of general problem solving knowledge to be successful. Further, it would be unlikely for a writer of a specific AI application to put this much general search knowledge into his or her program. In fact, we continue to find more such knowledge that needs to be added to our prover. In addition to general search knowledge, specific domain knowledge is often essential. Many of the theorems proven by the system are harder than problems solved by typical expert systems and AI reasoning systems, we feel. However, no specific verification of this belief has been attempted. One way to embed domain specific knowledge in our prover is to weight the links between clauses depending on experience; other methods will also be presented.

One goal of our work is the rational reconstruction of human theorem proving activity. How could a human mathematician have obtained a proof, using plausible heuristics? We would like to take the mystery out of the process of proof discovery. We propose that simple low-level heuristics account for most of human theorem proving ability. By low-level heuristics, we mean such methods as preferring terms that are small, or contain a small number of variables, and so on. Some of the methods used in theorem proving, such as variable elimination [Hin88] and the complexity measure of Wang [WB87], successful though they are, seem largely to replace one mystery with another, since it is unlikely that a human would use such methods. We present some ideas that seem more natural, and

illustrate some of these ideas on the intermediate value theorem.

An advantage of first order logic is that it is a universal language. This makes it easy to compare theorem provers since the same problems can be run on all of them. Also, this makes it easy to say how much knowledge is being given to the program to help guide the search, since the problem representation is often fixed. Another advantage of this universality is that if theorem proving methods are designed as mappings from clauses to clauses, then any combination of such methods can be used together.

2 Past Work in Automatic Theorem Proving

Some of the early work in theorem proving was similar in spirit to ours, such as the linked conjunct method of [Dav63]. Even the methods of [Gil60][PPV60] have similarities to ours. However, they also have significant differences. Gilmore's method is not based on unification. This means that the number of instances generated may be very large. Wang's method and the linked conjunct method are based on something like unification. However, they both consider a large number of possibilities for which literals to unify, resulting in many cases to consider. The paper [Dav63] (linked conjunct method) also doesn't explicitly give a completeness proof or a method of generating instances. Our method is based on unification but does not generate such a large number of cases. In this our method is similar to resolution. This may explain the effectiveness of resolution (and our method) as compared to methods used prior to 1965. Methods developed since 1965, such as model elimination and connection graph methods, also tend to be based on unification without requiring a large number of cases to be considered. Since 1965, much work has been done on the resolution method of Robinson [Rob65]. Unlike our method, resolution combines parts of different clauses. This has advantages and disadvantages. The connection graph methods of [Bib82] are close to ours, but even these combine literals from different clauses. We feel that instance-based approaches deserve more attention, and our experience with our implementation confirms this belief. We propose that instance-based provers can be competitive or superior to resolution provers using equivalent technology.

The motivation of our work is to avoid the duplication of instances of clauses. There are two kinds of duplication. The first we call *duplication by combination.* This appears in resolution, model elimination, the connection graph method, and other strategies. The same instances of a clause can contribute to many resolvents, for example. The second kind of duplication is *duplication by case analysis.* This appears in the linked conjunct method of [Dav63] as well as in Davis and Putnam's propositional calculus decision procedure [DP60]. The same instances of clauses may appear in more than one case of a case analysis, although in each case there may be no duplication. Duplication by case analysis is more storage efficient than duplication by combination, because only one case needs to be considered at a time. Some early provers used duplication by case analysis, but few if any clause form provers currently use it. Andrews' prover [MCA82] seems to be one of the few current provers that use duplication by case analysis. We avoid both kinds of duplication for first-order logic. We do use a Davis and Putnam-like method in our prover, but only for a propositional problem obtained from the first order theorem. For propositional calculus, case analysis is a natural and efficient strategy, in our experience.

We first present the theorem proving problem and notation in a simplified form sufficient for our purposes. Then we clarify the above points about duplication of instances. Our presentation is based on the theorem of Skolem-Herbrand-Gödel which is popularly known as Herbrand's theorem.

We use lower case letters, possibly subscripted, for individual constants, function constants, and predicate constants, and upper case letters, possibly subscripted, for (individual) variables. Each predicate constant and function constant has an *arity*, which is a non-negative integer telling how many arguments it takes. A *term* is a well formed expression composed of individual constants, variables, and function constants. An *atom* is a predicate constant followed by a list of terms. A *literal* is an atom or an atom preceded by a negation sign. We represent a negation sign by either "not" or "¬". A literal is called *positive* if it is an atom, *negative* if it is the negation of an atom. A literal and its negation are called *complementary literals.* A *clause* is a disjunction of literals, expressed as a set. The empty clause denotes the logical constant FALSE. A *set of clauses* represents the conjunction of the clauses in the set. A *positive clause* is a clause containing only positive literals; a *negative clause* is a clause containing only negative literals. A clause containing only one literal is called *unit clause.* A term is a *ground* term if it contains no variables, and similarly for literals and clauses. A *substitution* is a mapping Θ from variables to terms, such that for only finitely many variables X do we have $\Theta(X) \neq X$. We represent a substitution by a finite set $\{X_1 \leftarrow t_1, \ldots, X_n \leftarrow t_n\}$, indicating that $\Theta(X_i)$ is t_i and $\Theta(X)$ is X if X is a variable distinct from the X_i. If t is a term, then by $t\Theta$ we indicate the result of simultaneously replacing all variables X_i in t by t_i. We say a term u is an *instance* of a term t if there is a substitution Θ such that u is $t\Theta$, and similarly for literals and clauses. A *unifier* of two terms (literals, clauses) L and M is a substitution Θ such that $L\Theta$ and $M\Theta$ are identical. A *most general unifier* Θ of two terms (literals, clauses) L and M is a unifier of L and M such that for any other unifier Γ of L and M there is a substitution Φ such that $L\Theta\Phi$ is identical to $L\Gamma$ (or $M\Theta\Phi$ is identical to $M\Gamma$). A set of ground clauses is *unsatisfiable* if there is no assignment of truth values, i.e. TRUE or FALSE, to ground atoms in S such that all clauses contained in S are true simultaneously.

Theorem proving in first order logic can be reduced to the following problem, by Herbrand's theorem: Given a set S of clauses, to determine if there is a set T of ground clauses such that for each clause D in T there is a clause C in S such that D is an instance of C, and such that T is propositionally unsatisfiable. We call this the *ground instantiation problem.* For example, given S = {{p(a)}, {¬p(X), q(f(X))}, {¬q(f(Y))}}, the set T = {{p(a)}, {¬p(a), q(f(a))}, {¬q(f(a))}} is as specified. This set T is propositionally unsatisfiable because p(a) and (¬p(a) ∨ q(f(a))) imply q(f(a)). This contradicts ¬q(f(a)). Note that if the ground instantiation problem can be solved, then S, with free variables regarded as universally quantified, is unsatisfiable. The search for the proof of any theorem in first order logic can be reduced to the ground instantiation problem by negating the theorem and then applying a well known process called Skolemization [CL73][Lov78]. For example, consider the theorem that (∀X p(X)) implies (∀X (p(X) ∨ q(X))). This theorem is first negated, giving (∀X p(X)) and ¬(∀X (p(X) ∨ q(X))), which is equivalent to (∀X p(X)) and (∃X (¬p(X) ∧ ¬q(X))). Replacing existential quantifiers by a new constant a, we obtain (∀X p(X)) and (¬p(a) ∧ ¬q(a)). Removing universal quantifiers, we have p(X) and (¬p(a) ∧ ¬q(a)). This is converted to the following set S of clauses: {{p(X)}, {¬p(a)},

{¬q(a)}}. For details of this process, see [CL73][Lov78]. There is the set T of ground instances {{p(a)}, {¬p(a)}} which is propositionally unsatisfiable. Therefore the original theorem (∀X p(X)) implies (∀X (p(X) ∨ q(X))) is valid. Note that if S contains the empty clause, then we can choose T to be the set containing the empty clause, solving the ground instantiation problem.

Our theorem proving method constructs the set T of instances directly, then uses a propositional calculus decision procedure to test if T is unsatisfiable. We now make more precise how resolution and other methods involve the duplication of instances. Given clauses C and D, resolution computes a most general unifier Θ and then combines literals of CΘ and DΘ to obtain a resolvent. Note that if C and CΘ are identical, then literals may appear in both C and CΘ. This is one kind of duplication of instances by combination. Another kind appears when a clause C resolves against many clauses D_1, D_2, ..., D_n using the substitutions Θ_1, Θ_2, ..., Θ_n. In this case, it may be that many of the instances $C\Theta_i$ are identical. Then the literals in $C\Theta_i$ may appear in many different resolvents. This is another kind of duplication by combination. If C or D is a unit clause, then this resolution is called a *unit resolution* [WOLB84] and it turns out that duplication by combination does not occur. Duplication by combination occurs for similar reasons in model elimination and the connection graph method. The linked conjunct method of [Dav63] involves duplication by case analysis. Given a set S of clauses, a linked conjunct is obtained by matching the literals in clauses in S in different ways. Each matching has to be looked at separately; each such matching is another case to be considered. The same instances will appear in many such matchings; thus we have duplication by case analysis. Even if S is propositional, there may be many cases to consider and the method may be slow. Since a given literal may match many other literals, there may be many cases to consider for each literal. Similarly, Davis and Putnam's method involves case analysis on the truth values of atoms appearing in S. If an atom A appears in S, the cases A = TRUE and A = FALSE are considered separately. Many clauses and literals will appear in both cases, so we have duplication by case analysis. However, since there are only two cases per literal, the case analysis appears to be more efficient than in the linked conjunct method. Davis and Putnam's method also has a number of optimizations for deleting literals from clauses in S whenever possible and eliminating some clauses from S when possible. In fact, our theorem prover appears to be much better than resolution and similar methods for problems that are largely propositional; this seems to be evidence that the duplication referred to is hindering these other methods. A related problem with connection graph methods is the need to store the connections. A connection is a pair (L, M) of literals such that L and the complement of M are unifiable. If there are n literals, there may be about n^2 connections between them. This can be a large number of connections, and may require excessive storage and time to process. In fact, we tried storing some of these connections in earlier versions of our prover, but the cost was too great. Our current prover only stores instances of clauses and not any connections between them. The mating strategy of [And81] is similar to the connection graph method and has similar properties. In fact, the mating strategy is much like the linked conjunct procedure of [Dav63]. Since it explicitly constructs links, or connections, between literals, the number of links with a particular literal may be much larger than two. Mating appears to have duplication by case analysis rather than duplication by combination, since matings are constructed and considered one

by one. However, since the number of links per literal may be much larger than two, the number of cases may be much larger than in Davis and Putnam's method for propositional calculus, it appears. We should mention that Andrews' prover also works on non-clause form formulas. Even our modified problem reduction format prover [Pla88] has duplication by combination on non Horn problems.

3 Issues and Developments in Theorem Proving

Many lines of work are progressing in theorem proving. Some work deals with complete strategies for general first order logic; a strategy is *complete* if it can prove all valid formulas. Specialized rules for various domains have been shown to increase the power of theorem provers; Bledsoe's group has used this idea to obtain a good prover for theorems in analysis (the study of continuous functions). The theorem prover of Wu [Cho84][Wu78] performs spectacularly on many geometry theorems. Special decision procedures for certain theories have been developed by Nelson and Oppen and others [NO80]. Another approach is to develop languages in which a user can specify strategies that can be applied; Bundy [Bun88] and others have used this approach effectively. Special unification algorithms for various equational theories continue to be developed [SS82]. Term rewriting techniques [DJ90] are particularly effective for problems that may be stated as systems of equations and inequations. Recently, techniques from logic programming have been applied to theorem proving, to obtain very fast rates of inference, on the order of thousands or tens of thousands of inferences per second. This approach was initiated by Stickel [Sti86] and others have extended it to parallel processing theorem provers [ABCM88]. Another way to obtain speed is by the use of good data structures and a low level programming language, as in the OTTER prover of McCune [McC89] which is written in C and uses discrimination nets for fast look-up of potential unifications. Provers are being used more and more for program verification and circuit verification and VLSI design. Theorem provers are now capable of proving respectable mathematical theorems, while also missing many theorems humans would find obvious.

4 The Hyper-Linking Strategy Prover

The philosophy of the hyper-linking strategy is never to combine literals from two clauses. Such combinations are done temporarily a couple of places in the prover, but if they do not lead to a proof, they are immediately forgotten. We tried several prior implementations of this idea. The idea first appeared in [Pla80] and the first implementation was in [JP84]. We attempted to extend this to a unification-based prover, without much success [PM88]. The key to our current success seems to be that we do not store any links between clauses, which sets our method apart from the connection graph methods of [Bib82] which rely heavily on such links. Also, a key to our success is the idea of hyper-linking, that is, considering all the literals in a clause at the same time, rather than one by one as in our previous attempts. The use of support strategies, explained below, was also important. Finally, in previous strategies, we computed a "fully linked set," which was essentially the

same as the linked conjunct of [Dav63]. However, we found that it takes too much time to compute this set, and instead we consider the set of all clauses, which may be much larger.

Another advantage of this method is that clauses are kept as lists of literals, as in resolution. All literals are on an equal footing. This contrasts with the modified problem reduction format of [Pla88], in which contrapositives of clauses are considered separately. The conceptual simplicity of having clauses as lists has enabled us to add some refinements to the prover without having to deal with the complexities of contrapositives.

4.1 Overview of the Theorem Prover

There are a number of steps to our theorem proving method. Given a set S of clauses, we want to find a set T of instances that are propositionally unsatisfiable, if it exists. Our method successively generates more instances of S using a hyper-linking operation, and periodically tests these instances for unsatisfiability. The phases in our method are hyper-linking, unit simplification, propositional unsatisfiability checking, and small proof checking. Unit simplification implements special rules for unit (one literal) clauses, as described below. Small proof checking searches for small proofs of a certain form, and often permits the method to terminate early. These four phases are repeated again and again until a proof is found or until a time limit is exceeded.

This prover performs well on a number of problems, including the salt and mustard problem, the latin squares problem [Rob63], apabhp [Sti86], and the zebra problem [LS86]. Solutions are obtained much faster than by the modified problem reduction format prover of [Pla88]. For example, the salt and mustard problem is solved in 30.217 seconds, and apabhp is solved in 472.417 seconds. The prover seems particularly good on problems that are nearly propositional in nature. It also seems good on other problems. We cannot think of a reason why the speed advantage on propositional problems should not extend to general first order logic, but we do not have as much evidence for this. For a listing of the times taken by our prover on some standard problems, see Table 4 at the end of the paper. Problems 1-93 are from [Sti86]. The problem "example" is a theorem presented by Pellitier and Rudnicki in AAR Newsletter No. 6, 1986. The problem "exx5" is a verification condition from Hoare's FIND program. The problem "exx7" is a situation calculus theorem developed by David Plaisted. "latinsq" is the latin square problem from [Rob63]. "liar" is the teller and liar problem. "salt" is the salt and mustard problem. The problem "schubert" is Schubert's statement [Wal84]. The stack problem is a proof of a trivial property of push down stores. Problems 104-110 are pigeonhole problems. Problems 111-114 are implicational calculus theorems.

The current implementation is storage inefficient due to the implementation in ALS prolog, which compiles all assertions. Despite this, we usually do not run out of storage.

4.2 The Hyper-Linking Strategy

Given a set S of clauses, define a *link* in S to be a pair (L, M) of literals such that both L and M appear in clauses in S and such that L and the complement of M are unifiable. Note that this is essentially the same as a connection in the connection graph method,

except that we don't explicitly store the links, so that they cannot be deleted as they are in connection graph resolution.

Definition. If C = $\{L_1, \ldots, L_m\}$ is a clause in a set S of clauses, then a *hyper-link* of C in S is a set $\{(L_1, M_1), \ldots, (L_m, M_m)\}$ of links in S such that there exists a substitution Θ such that $L_i\Theta$ and $M_i\Theta$ are complementary for all i, $1 \leq i \leq m$. A most general such Θ is called the *substitution* of the hyper-link and CΘ for this Θ is called the *instance* of the hyper-link. We call this instance generation a *hyper-link operation.* We call C the *nucleus* of the hyper-link and we call the M_i (or clauses D_i containing M_i) the *electrons* of the hyper-link.

The hyper-linking phase of the prover works this way: For each clause C in S, all hyper-links of C in S are computed, together with their substitutions Θ. Then, the set of all such instances CΘ are added to S, for all clauses C in S and all substitutions Θ of their hyper-links. For example, if {¬p(X), q(X)} is in S and literals p(a) and ¬q(a) appear in clauses in S, then the instance {¬p(a), q(a)} is generated. If {¬p(X), q(f(X))} is in S and the literals p(a) and ¬q(X) appear in clauses in S then the instance {¬p(a), q(f(a))} is generated.

In practice, hyper-linking is implemented in Prolog using backtracking, to examine all possibilities of links for all literals in C. Each instance is asserted into the database. Note that if some literal in C is a ground literal, or becomes a ground literal due to previous links, then no backtracking is needed since linking will not instantiate this literal. Also, note that our prover does not delete instances of clauses. Thus it is possible to have both a clause C and an instance D of C at the same time. This seems to be a disadvantage, but does not appear to cause problems. Also, instance deletion can be done in some cases, as explained below.

After the hyper-linking phase, our prover performs a unit simplification phase which is analogous to operations in the Davis and Putnam procedure. If a unit clause {L} is derived and a clause C is obtained by hyper-linking such that C contains M, and M is an instance of the complement of L, then C is replaced by C - {M}. We can write this as an inference rule as follows:

$$\frac{S \cup \{C\} \cup \{\{L\}\},\ M \text{ in } C,\ M \text{ is an instance of } \neg L}{S \cup \{C - \{M\}\} \cup \{\{L\}\}}$$

This is called *unit literal deletion.* Also, if a unit clause {L} is derived, and clause C is derived, and C contains a literal M that is an instance of L, then C can be deleted. This is called *unit subsumption.* This may be written as an inference rule as follows:

$$\frac{S \cup \{C\} \cup \{\{L\}\},\ M \text{ in } C,\ M \text{ is an instance of } L}{S \cup \{\{L\}\}}$$

Note that unit literal deletion may produce more unit clauses, which may enable more unit literal deletions, and so on. Thus, a considerable amount of inference may occur in the unit simplification phase. As an example, suppose S is {{p}, {¬p, q}, {¬q, r}}. Then unit literal deletion with p produces {{p}, {q}, {¬q, r}}. There is now a new unit clause

q, and unit literal deletion with q finally produces the set of clauses {{p}, {q}, {r}}. As an example of unit subsumption, if S is {{p}, {p, q}}, then unit subsumption with p causes {p, q} to be deleted, producing the set {{p}} of clauses.

The next step of our prover is propositional unsatisfiability testing. Suppose set S of clauses remains after unit simplification. Then we *ground* S, that is, replace all variables in S by a new constant symbol $, to obtain the set Gr(S) of ground clauses. Thus, if S is {{¬p(X)}, {q(X, Y)}} then Gr(S) is {{¬p($)}, {q($, $)}}. We then test Gr(S) for propositional unsatisfiability using a propositional calculus decision procedure similar to Davis and Putnam's method. It turns out that this is complete. That is, if a set S is unsatisfiable, then after some number of rounds of hyper-linking, Gr(S) will be propositionally unsatisfiable.

After the propositional test, we then apply a small proof check to S (not Gr(S)). This is a fast check to see if S has a "small proof." The details will be explained below. For example, if S contains two unit clauses {L} and {M}, and L and the complement of M are unifiable, then we know that S is unsatisfiable. However, we can test for this condition much faster than by performing another round of hyper-linking, unit simplification, and propositional satisfiability testing. Our small proof check is more general than this test for complementary unifiable literals, but still can be performed efficiently.

We now give an example to show how our prover works. Suppose S is {{p(a), q(a)}, {¬p(X), q(Y)}, {p(X), ¬q(Y)}, {¬p(X), ¬q(Y)}}. We call these clauses C_1, C_2, C_3, and C_4, respectively. First, we hyper-link with nucleus C_2, obtaining the instance {¬p(a), q(Y)}. Then, we hyper-link with nucleus C_4, obtaining the instance {¬p(a), ¬q(Y)}. Next, we hyper-link with nucleus C_3, obtaining the instance {p(a), ¬q(a)}. At this point the set of instances is {{p(a), q(a)}, {¬p(X), q(Y)}, {p(X), ¬q(Y)}, {¬p(X), ¬q(Y)}, {¬p(a), q(Y)}, {¬p(a), ¬q(Y)}, {p(a), ¬q(a)}}. Finally, we ground all clauses, replacing all variables with $, obtaining the set of ground instances {{p(a), q(a)}, {¬p($), q($)}, {p($), ¬q($)}, {¬p($), ¬q($)}, {¬p(a), q($)}, {¬p(a), ¬q($)}, {p(a), ¬q(a)}}. This set is found to be propositionally unsatisfiable. To see this, consider the subset {{p(a), q(a)}, {p(a), ¬q(a)}, {¬p(a), q($)}, {¬p(a), ¬q($)}}. The first two clauses imply p(a). This, together with the third and fourth clauses, implies q($) and ¬q($), respectively, which is contradictory. Note that if we had grounded the set S of input clauses, this set would be satisfiable. The extra instances generated by hyper-linking were needed for the proof.

We now give an example which shows the use of unit simplification during hyper-linking. Suppose S is {{p(a)}, {¬p(X), q(f(X))}, {¬q(X), r(g(X))}, {¬r(X)}}. Let's call these clauses C_1, C_2, C_3, and C_4, respectively. Hyper-linking with nucleus C_2, we obtain the instance {¬p(a), q(f(a))}, which is simplified to {q(f(a))} using unit literal deletion with the unit clause {p(a)}. Hyper-linking with nucleus C_3, we obtain the instance {¬q(f(a)), r(g(f(a)))}, which simplifies to {r(g(f(a)))} using the new unit clause {q(f(a))}. Also, using the unit clause {¬r(X)}, this further simplifies to the empty clause {}, indicating that a contradiction has been found, and that S is unsatisfiable. For this proof, the propositional calculus prover is not needed, because the proof is obtained by hyper-linking and unit simplification alone. This is often the case. Also, those who are familiar with resolution may notice the similarity of this sequence of hyper-links to a unit resolution proof.

4.3 Completeness

We now prove the completeness of the hyper-linking strategy.

Definition. A mapping ϕ from clauses to clauses is an *instance mapping* if for all clauses C, ϕ(C) is an instance of C.

Lemma 1. Let T be an unsatisfiable set of ground instances of set S of clauses. Suppose Gr(S) is satisfiable. Let

$$\phi : T \rightarrow S$$

be an instance mapping. Extend ϕ to map the literals of C to the corresponding literals of ϕ(C); for this, it may be necessary to distinguish the literals in different clauses. Then there are literals L, M in T that are complementary but Gr(ϕ(L)) and Gr(ϕ(M)) are not complementary.

Proof. Let I be a model of Gr(S). Suppose the lemma is false; then for all literals L, M of clauses in T, if L and M are complementary then Gr(ϕ(L)) and Gr(ϕ(M)) are complementary. Then I can be used to give a model I′ of T defined by I′ $\models$ L iff I $\models$ Gr(ϕ(L)). This contradicts the assumption that T is unsatisfiable.

Definition. For a clause C, ||C|| is the sum of the number of occurrences of non-variable symbols in C, that is, individual, function, and predicate constants. ||L|| is defined similarly for a literal L. If

$$\phi : T \rightarrow S$$

is an instance mapping and T is $\{D_1, \ldots, D_n\}$ then $\|\phi\|$ is the sum over i of $\|D_i\|$.

Definition. Suppose (L,M) is a link in S and L and M are contained in clauses C and D respectively. A *link operation* on S with (L,M) produces two instances Cθ and Dθ, where θ is the most general unifier of L and M.

Lemma 2. Let T be an unsatisfiable set of ground instances of set S of clauses. Let ϕ be as in the above lemma. Suppose Gr(S) is satisfiable. Then there are clauses C_1 and C_2 obtained by a link operation on S and there is an instance mapping

$$\phi' : T \rightarrow S \cup \{C_1, C_2\}$$

such that $\|\phi'\| > \|\phi\|$.

Proof. By Lemma 1, there must exist literals L, M in clauses D_1, D_2 of T such that L and M are complementary but Gr(ϕ(L)) and Gr(ϕ(M)) are not complementary. However, it must be that their most general unifier θ binds at least one variable to a non-variable term. Then,

$$\| L\theta \| + \| M\theta \| > \| L \| + \| M \|$$

Define ϕ' by ϕ'(D) = ϕ(Dθ) for D = D_1 or D = D_2, and ϕ'(D) = ϕ(D) otherwise. Then $\|\phi'\| > \|\phi\|$.

Theorem. If S is an unsatisfiable set of clauses, then there is a set S″ of clauses obtained by a sequence of link operations on S such that Gr(S″) is unsatisfiable.

Proof. If Gr(S) is unsatisfiable, we are done. Suppose Gr(S) is satisfiable. We know by Herbrand's theorem that there is an unsatisfiable set T = $\{D_1, \ldots, D_n\}$ of ground

instances of clauses of S. Let ϕ be an instance mapping from T to S. Now, $\|\phi\|$ is bounded by the sum over i of $\|D_i\|$. By Lemma 2, if Gr(S) is satisfiable then we can perform a link operation on two clauses of S obtaining S′ and

$$\phi' : T \rightarrow S'$$

with $\|\phi'\| > \|\phi\|$. If Gr(S′) is satisfiable, this can be repeated. Since the $\|\phi\|$ are bounded, this process must stop. Letting S″ be the set of instances at that time, Gr(S″) will be unsatisfiable by Lemma 2.

Corollary. If S is an unsatisfiable set of clauses, then there is a set S″ of clauses obtained by a sequence of hyper-link operations on S such that Gr(S″) is unsatisfiable.

Proof. For each link operation as in the theorem, we can find two hyper-links that will generate clauses that are instances of those generated by the link operation. However, these clauses will still permit an instance mapping from T, and eventually a set S″ will be obtained with Gr(S″) unsatisfiable.

Note the simplicity of the proof, which seems simpler than the proof of the completeness of resolution, and is directly related to Herbrand's theorem. This suggests that the completeness proofs for extensions and refinements of this method (including equality and paramodulation) will be simpler than for resolution. Note also that our method does not have a factoring operation on clauses; a factor of a clause is an instance with two literals unified.

5 Support Sets

In resolution, set-of-support strategies [WRC65] are widely used. Such strategies seem essential for applications in which there are a large number of clauses. We also use a number of support strategies to restrict which hyper-links are performed while maintaining completeness. One of these support strategies simulates forward chaining, another simulates backward chaining, and another simulates set-of-support in resolution. We permit these support methods to be interleaved in arbitrary orders on successive hyper-linking rounds, which allows us to combine forward and backward chaining, for example. We have found such strategies to be effective in reducing the size of the search space. We also permit combinations of support strategies to be used on a given round.

We now define the various support criteria for the hyper-linking strategy. Suppose C is a nucleus of a hyper-link and $L_1 \ldots L_m$ are the positive literals that are electrons for C and $M_1 \ldots M_n$ are the negative literals that are electrons for C. Thus C has m negative literals, linked by L_i, and n positive literals, linked by M_j. Let $A_1 \ldots A_m$ be the clauses containing $L_1 \ldots L_m$, respectively, and let $B_1 \ldots B_n$ be the clauses containing $M_1 \ldots M_n$. Let D be the instance of C obtained by this hyper-link. Then D is *forward supported* if m = 0 (C is all positive) or if all A_i are forward supported. D is *backward supported* if n = 0 (C is all negative) or if all B_j are backward supported. D is *user supported* if C was user supported or if some A_i or B_j was user supported. Initially, the user can specify a user support set T with the input clauses; all clauses in T are considered to be user supported initially. A round of hyper-linking can be restricted to be forward supported, backward supported, user supported, or any combination of the three. If forward support is specified,

for example, then only forward supported hyper-links are performed. If a round is forward and user supported, then only instances which are both forward and user supported, are retained. Also, various support criteria can be interleaved. For example, [f,b] indicates that alternate hyper-linking rounds will be forward supported and backward supported.

We also define distances for each kind of support. Thus a clause may have a *forward distance*, a *backward distance*, and a *user distance*. These are defined as follows: Let D be an instance as above. Then the forward distance of D is zero, if D is all positive, otherwise it is 1 + max{forward distances of A_i}. The backward distance of D is zero, if D is all negative, otherwise it is 1 + max{backward distances of B_j}. The user distances of all clauses are initially infinity, except clauses in the user support set, which have a user distance of zero. The user distance of D is 1 + min{user distances of A_i and B_j}. These distances measure how closely related D is to various support sets. The larger the distance, the less related D is. We have a priority measure that can be made to depend on these distances, and prefer more "relevant," or closely related, clauses.

User support seems necessary when there are large numbers of clauses. The user will typically specify the user support set to be those clauses that come from the theorem to be proven, and the other clauses will typically be axioms of some theory. User support restricts attention to clauses that are in some way related to the theorem. This appears to be necessary when there are large numbers of axioms, for otherwise many clauses will be derived purely from the axioms. The set of support restriction in resolution has a similar effect.

We also have implemented a "throw away" strategy, which permits a nucleus to be deleted in certain cases. Suppose we are doing a round of forward support. Suppose a nucleus C is forward supported. Then C can be deleted after hyper-linking. The reason is that any instances needed for the proof, will be generated during that round, so that C is not needed.

A problem with user support is that many instances become user-supported rapidly, since only one electron needs to be user-supported for an instance to be user supported. Forward support is more restrictive because all the electrons for the negative literals have to be forward supported in order for an instance to be forward supported. Often clauses have many negative literals and only one or two positive literals. Forward support has the problem that it does not restrict attention to the theorem. We have a method of combining the advantages of both kinds of support, although this has not yet been implemented. Given two types of support x and y, where x and y may be user, forward, or backward, we define x:y support as follows: Suppose C is a nucleus of a hyper-link and D is the instance of the hyper-link. Then D is x:y supported if either C is not x supported and D is x supported, or if C is x supported and D is both x and y supported. The restriction to x:y support is complete for a number of combinations of x and y, including user:forward, user:backward, forward:backward, and backward:forward. In fact, x:y support is complete for x and y chosen arbitrarily from user, forward, and backward, but the given four combinations seem most interesting. For example, user:forward first generates user supported instances, and then restricts attention to their instances that are both user and forward supported. This is complete because eventually all clauses needed for the proof will be user supported, and after that, forward support can be used. Thus we restrict attention to instances related to the theorem, and also have the advantages of forward support.

Our distance measure is an attempt to restrict the proof size. However, we are really measuring the depth of the proof, not its size. This different measure may explain some differences in performance between our prover and the iterative deepening prover of Stickel [Sti86] as well as the modified problem reduction format prover [Pla88]. Both of those provers measure the size rather than the depth of a proof. Sometimes it may be preferable to use one measure, and sometimes it may be preferable to use another.

6 Unit Clauses

We now present some special rules for unit clauses. These substantially help our prover. The first rule is that no instances of unit clauses are kept. That is, if L and M are two unit clauses, and M is an instance of L, then M can be deleted. This is actually a special case of unit subsumption, described earlier. The unit literal deletion rule permits literals to be deleted from clauses, and was also described earlier. Both rules simplify the set of clauses, in the sense that the size of the set of clauses is reduced, but proofs do not become harder to obtain. We need to treat one case specially, that is, the case in which there are two unit clauses L and M such that L and the complement of M are unifiable, but L is not an instance of the complement of M, nor is M an instance of the complement of L. For example, suppose that S is {{p(a, X)}, {¬p(Y, b)}}. Then we can hyper-link with nucleus {p(a, X)} to obtain the instance p(a, b). However, this is deleted by unit subsumption since it is an instance of {p(a, X)}. In fact, we never do hyper-links with unit clauses as nuclei for this reason. When we ground S, we obtain the set {{p(a, $)}, {¬p($, b)}} of ground instances, which are satisfiable, so a proof cannot be found. Therefore we need a special test for the case of unit clauses L and M which are unifiable with each other's complement.

We note that the UR resolution strategy [WOLB84], though incomplete, is compatible with the philosophy of our prover, namely, that literals from different clauses should never be combined. UR means "unit resulting," and this strategy only generates resolvents that are unit clauses. This strategy is often used by the Argonne group. We have also found that this strategy is effective in our prover, not surprisingly. UR resolution can be viewed as a kind of hyper-linking operation, in which all but one of the electrons is a unit clause, and the other electron is absent. For example, if the nucleus is {¬p(X), ¬q(X), ¬r(X)} and the electrons for the first and third literals are p(a) and r(a), then the instance {¬p(a), ¬q(a), ¬r(a)} is generated. The electron for the second literal ¬q(X) is missing here. The literals ¬p(a) and ¬r(a) are immediately deleted by unit literal deletion, resulting in the unit clause ¬q(a). Thus we have obtained a unit clause ¬q(a) by resolving the given clause with two unit clauses; this is essentially the same as UR resolution.

7 Instance Deletion

Since our prover does not delete instances of clauses, it may keep more clauses than a resolution prover. First, we note that not deleting instances can be an advantage, because it means that duplicate checking can be done much faster. Hashing or a discrimination net can be used for this purpose, in time essentially independent of the number of instances.

However, partially because of our use of Prolog as the implementation language, we have not fully exploited this possibility.

We have experimented with methods of instance deletion, and have obtained some improvement on a few problems. For example, it is complete to only delete instances of non-forward supported clauses. The reason is that eventually all the instances needed will be forward supported, and therefore retained. Also, if the set of input clauses is a Horn set [HW74], then unit resolution is complete, and all instances can be deleted. In fact, instance deletion is complete whenever unit resolution suffices to obtain a proof, although this may not be known in advance. We have obtained significant improvements on some problems using these ideas, but significant slowdowns on others. It seems that retaining instances is not usually a major problem for this prover.

Instance deletion can be made more efficient in the following way: Hyper-linking is performed by creating a list of all the literals that appear in clauses in S and using these literals as electrons in the hyper-links. If all instances are deleted, then we can delete literals in this list that are instances of other literals in this list, because such literals will produce clauses that are instances of clause produced by more general literals. A more restrictive but similar idea can be used if instances of non-forward supported clauses are deleted.

8 The Propositional Calculus Prover

After hyper-linking and unit simplification, the set S of instances generated so far is given to the propositional calculus prover. First, all variables in S are replaced by a new individual constant \$ to obtain a set Gr(S) of ground clauses. Thus, the clause {¬p(X, Y), q(Y)} would be replaced by {¬p(\$, \$), q(\$)}, for example. Then Gr(S) is tested for satisfiability using the propositional prover. If Gr(S) is unsatisfiable, then the ground instantiation problem has been solved and we know S is unsatisfiable. Otherwise, small proof checking or more rounds of hyper-linking and so on, are done.

We have found that the time taken by the propositional prover is usually much smaller than the time taken by the other steps of the prover. Therefore, even if P = NP, this would not help most of the time. It is only for a few problems such as the "pigeonhole problems" [Pel86] that the propositional prover takes most of the time. The method we use is a modification of Davis and Putnam's method [DP60] and uses a technique similar to the dependency-directed backtracking methods discussed in [dK86][dK89] and to McAllester's RUP procedure [McA80]. However, even Davis and Putnam's method performs about as well most of the time. It is surprising that such a simple method should be good enough to make the time for propositional proving negligible most of the time. Also, since the only duplication of instances in our method occurs in the propositional prover, the speed of the prover shows that this duplication of instances by case analysis is not hurting the prover most of the time.

8.1 Davis and Putnam's Method

We first describe Davis and Putnam's method, and then our modification of it. The following simplifications are used by both methods:

1. A clause containing TRUE is replaced by TRUE.
2. FALSE may be deleted from a clause containing FALSE.
3. A clause containing both L and ¬L may be replaced by TRUE.
4. TRUE may be deleted from a set of clauses containing TRUE.
5. A set of clauses containing FALSE may be replaced by FALSE.

Thus, {{TRUE, p}, {FALSE,q}} simplifies to {TRUE, {q}} which simplifies to {{q}}.

We represent Davis and Putnam's procedure by a procedure DP(S) which takes as input a set of propositional clauses and outputs TRUE, if S is satisfiable, and FALSE otherwise. This procedure is fairly simple, and may be described as follows:

```
procedure DP(S)
   Simplify S to S';
   If S' is TRUE or FALSE, return S';
   If S' contains a unit clause C then   % simplify using unit clause
      if C is {L} for positive literal L,
         replace L by TRUE in S', and simplify to obtain S";
         return DP(S");
      if C is {¬L} for positive literal L,
         replace L by FALSE in S', and simplify to obtain S";
         return DP(S");
   If S' contains a clause C containing a literal L and
      no other clause contains ¬L then   % L is a pure literal
      if L is positive, replace L by FALSE in S',
         and simplify to obtain S";
         return DP(S");
      if L is ¬M for some M,
         replace M by TRUE in S', and simplify to obtain S";
         return DP(S");
   If none of the above holds, then   % do case analysis
      pick a positive literal L in some clause C in S';
      replace L by TRUE, and simplify S' to S_1;
      if DP(S_1) is TRUE then return TRUE
      else replace L by FALSE, and simplify S' to S_2;
           if DP(S_2) is TRUE, then return TRUE
           else return FALSE;
end DP.
```

8.2 Our Procedure

Our procedure is as follows:

```
procedure PC(S)
   Simplify S to S';
   If S' is TRUE or FALSE, return S';
   If S' contains a unit clause C then   % simplify using unit clause
     if C is {L} for positive literal L,
       replace L by TRUE in S', and simplify to obtain S";
       return PC(S");
     if C is {¬L} for positive literal L,
       replace L by FALSE in S', and simplify to obtain S";
       return PC(S");
   If none of the above holds, then   % do case analysis
     if S' does not contain a positive or a negative clause
     then return TRUE
     else Let C be a smallest negative clause in S';
          pick a literal ¬L in C;
          replace L by FALSE, and simplify S' to S_1;
          if PC(S_1) is TRUE then return TRUE
          else if the assignment of FALSE to L was not
                  needed to show that S_1 is unsatisfiable
               then return FALSE   % right cutoff
               else replace L by TRUE, and simplify S' to S_2;
                    if PC(S_2) is TRUE, then return TRUE
                    else if the assignment of TRUE to L was not
                            needed to show that S_2 is unsatisfiable
                         then return FALSE   % left cutoff
                         else return FALSE;
end PC.
```

Our procedure is similar to the Davis-Putnam method, except that there is no test for a pure literal, and the clause used for case analysis is chosen as an all-negative clause, if one exists. However, the major difference is in the case analysis itself. If PC(S_1) is FALSE, we keep track of whether the assignment to L is used in this part of the proof. If not, we call this a *right cutoff* and return FALSE, since the same proof could be done if the other truth value were assigned to L. If the truth assignment to L was used in the first case, but in the second case, PC(S_2) is FALSE and the truth assignment to L is not used, we call this a *left cutoff*, and PC(S) returns FALSE. Right cutoffs correspond to cases in which our procedure does less case analysis than Davis and Putnam's method. We have observed a number of right cutoffs, especially on the pigeonhole problems. However, we have never observed a left cutoff.

This procedure seems to be efficient. The times are about the same as Davis and Putnam's method, sometimes a little worse, and sometimes a lot better. For example, the times for the pigeonhole problems are listed in Table 1, where ph8 means 8 balls in 7 slots,

problem	cpu(sec)
ph3	0.017
ph4	0.133
ph5	0.833
ph6	5.083
ph7	34.400
ph8	258.750

Table 1: Times for sample pigeonhole problems

and so on. We can construct a resolution proof of the empty clause from the run of our propositional prover, as described in [Pla89]. In general, we can show that if a left cutoff does not occur, then our procedure runs in time polynomial in the size of its input and in the size of the proof it finds. The size of the proof is measured by the size of the proof tree. Since a proof tree may have many occurrences of the same sub-proof, the tree size may be larger than the length of a straight-line proof of the empty clause. Since we have never observed a left cutoff, in practice our procedure seems to run in polynomial time, in this weak sense.

8.3 Logic Puzzles

We have run our prover on some logic puzzles. These are solved mostly by the propositional prover, and our prover seems to do well on them. Although the time required to solve them is usually larger than that for a customized Prolog program, our input is more declarative, being in pure first order logic. The method of solution makes use of a different translation of a first-order formula into clause form than is standard for theorem provers. For logic puzzles, there is usually a finite domain D specified. For example, there may be a finite number of people, a finite number of jobs, and so on, and the problem is to match the people with the jobs so as to satisfy a number of constraints. Given such a domain D, we translate a first-order formula, containing quantifiers, into clause form in which individual constants and variables may appear but no function symbols of arity (number of arguments) greater than zero may appear. This is essentially the same as Schonfinkel-Bernays form in logic [Lew80], which is known to be decidable. The usual clause form translation introduces Skolem functions, which may make the search space infinite.

Given such a D = $\{d_1, d_2, \ldots, d_n\}$, we can translate a formula ($\exists$X A[X]) into the formula A[d_1] $\vee$ A[d_2] $\vee \ldots \vee$ A[d_n]. This can be further optimized if the formula is of the form ($\exists$X (p(X) $\wedge$ A[X])), where some subset E = $\{e_1, e_2, \ldots, e_m\}$ of D satisfies p. For example, p(X) may mean that X is a person, and some of the d_i may be persons, and some may be jobs, and so on. We can translate ($\exists$X) (p(X) $\wedge$ A[X]) into the formula A[e_1] $\vee$ A[e_2] $\vee \ldots \vee$ A[e_m]. Using such techniques, a formula without function symbols may be obtained. This translation can result in an exponential size increase if there are many nested existential quantifiers. To avoid this, we can introduce new predicates for sub-formulas, as in the structure-preserving clause form translation of [PG86]. In this way, a polynomial size formula may be obtained. Some problems involve equality. We note that this can be expressed by the axiom X = X together with the axioms $d_i \neq d_j$ for i $\neq$ j. For

example, to express the fact that every person has a unique job, equality is needed.

Once the input is in the proper form, our prover can be used. However, in order to show that a set H of hypotheses implies a certain conclusion R, we need to convert the formula $(H \wedge \neg(R))$ to clause form and test it for satisfiability using the prover. This requires that the conclusion R be known. Usually in logic puzzles, finding R is the hard part. Therefore, we had to modify our prover to permit us to find the solutions to logic puzzles. To do this, we give the prover the hypotheses H. After some number of rounds of hyper-linking, the prover will detect that no more new instances can be generated, and that the set T of instances generated so far is satisfiable. At this point, the propositional prover can print out a model of Gr(T), that is, a truth assignment that makes all clauses in Gr(T) true. This model will give the solution to the logic puzzle, for example, an assignment of jobs to persons that satisfies the constraints. Once a model is found, a clause can be added which negates this truth assignment, and the prover can be run again. In this way, more than one solution to the logic puzzle can be found. When all such solutions have been found, the prover will detect that the clauses are unsatisfiable, indicating that there are no more solutions. For example, if a model $\{p, \neg q, r\}$ is found, this indicates that assigning TRUE to p and r and FALSE to q satisfies all the constraints. We can negate this model by adding the clause $\{\neg p, q, \neg r\}$, to the set of clauses. (Recall that a clause indicates the disjunction of its literals, that is, $\neg p \vee q \vee \neg r$.)

This procedure works fairly well. For example, we can solve the jobs puzzle of [WOLB84] in 349.866 seconds and verify that there are no other solutions in 347.883 seconds. We can solve the Zebra puzzle of [LS86] in 1807.666 seconds and verify that there are no more solutions in 941.850 seconds. Only a small part of this time is spent in the propositional prover; the rest is spent in hyper-linking. This suggests that a systematic procedure for instantiating all variables to all possible elements of D, followed by the propositional prover, would be faster. For example, we can use replace rules to make the generation of the needed instances faster; for a discussion of replace rules see section 10. In fact, Davis and Putnam's procedure would probably do about as well on these puzzles. Therefore, the significance of our result is mainly that it shows the applicability of our prover to problems that could be solved by other known methods. However, we have found that other provers often have a hard time with these logic puzzles. Also, we feel that our ability to systematically transform a first-order formula into a form that the prover can use, makes this a more natural input format than the customized Prolog programs often used to solve such puzzles in less time.

To illustrate the idea, let's solve the following puzzle:

1. There are three friends: Michael, Richard, and Simon; three nationalities: American, Australian, and Israeli; three sports: basketball, cricket, and tennis.

2. Each friend has a unique nationality and plays a unique sport.

3. These friends came first, second, and third in a programming competition.

4. Michael plays basketball, and did better than the American.

5. Simon, the Israeli, did better than the tennis player.

6. The cricket player came first.

Suppose c(X, Y) means X came in the Yth order, n(X, Y) means the nationality of X is Y, pl(X, Y) means X plays Y, pr(X) means X is a person, s(X) means X is a sport, o(X) means order X, and d(X, Y) means X did better than Y in the programming competition. The above rules are transformed to the following axioms:

- % three persons
  1. {pr(michael)}
  2. {pr(richard)}
  3. {pr(simon)}

- % three nationalities
  4. {n(american)}
  5. {n(australian)}
  6. {n(israeli)}

- % three sports
  7. {s(basketball)}
  8. {s(cricket)}
  9. {s(tennis)}

- % three orders
  10. {o(1)}
  11. {o(2)}
  12. {o(3)}

- % each person has a nationality and no two different persons have same nationality
  13. {¬pr(P), n(P, american), n(P, australian), n(P, israeli)}
  14. {¬n(N), ¬n(michael, N), ¬n(richard, N)}
  15. {¬n(N), ¬n(michael, N), ¬n(simon, N)}
  16. {¬n(N), ¬n(richard, N), ¬n(simon, N)}

- % each person plays a sport and no two different persons plays same sport
  17. {¬pr(P), pl(P, basketball), pl(P, cricket), pl(P, tennis)}
  18. {¬s(S), ¬pl(michael, S), ¬pl(richard, S)}
  19. {¬s(S), ¬pl(michael, S), ¬pl(simon, S)}
  20. {¬s(S), ¬pl(richard, S), ¬pl(simon, S)}

- % no two different persons came to the competition together
  21. {¬pr(P), c(P, 1), c(P, 2), c(P, 3)}
  22. {¬o(O), ¬c(michael, O), ¬c(richard, O)}

23. {¬o(O), ¬c(michael, O), ¬c(simon, O)}
24. {¬o(O), ¬c(richard, O), ¬c(simon, O)}

- % michael plays basketball, and did better than the american
  25. {pl(michael, basketball)}
  26. {¬pr(P), ¬n(P, american), d(michael, P)}

- % simon is Israeli, and did better than the tennis player
  27. {n(simon, israeli)}
  28. {¬pr(P), ¬pl(P, tennis), d(simon, P)}

- % no two different persons did the same in the competition
  29. {¬pr(P1), ¬pr(P2), ¬d(P1, P2), ¬d(P2, P1)}

- % transitivity of did_better
  30. {¬pr(P1), ¬pr(P2), ¬pr(P3), ¬d(P1, P2), ¬d(P2, P3), d(P1, P3)}

- % the cricket player came first
  31. {¬pr(P), ¬pl(P, cricket), c(P, 1)}

The prover detects that these axioms are satisfiable, and outputs a model including the following literals:

- c(simon, 1)
- n(michael, australian), n(richard, american), n(simon, israeli),
- pl(michael, basketball), pl(richard, tennis), pl(simon, cricket),
- d(michael, richard), d(simon, richard)

These literals are indeed the solution of the puzzle. This can be verified by putting the negation of these literals and the axioms together and running the prover to get a contradiction.

9 Small Proof Checking

Some provers gain speed by looking for pairs L and M of unit clauses such that L and the complement of M are unifiable. If such are found, then a proof of unsatisfiability is immediate. Searching directly for such pairs may be much faster than performing many inferences until a proof of unsatisfiability is found. Our small proof checking is a generalization of this "early termination" idea. Instead of looking for pairs of unit clauses,

we look for a hyper-link in which all the electrons are unit clauses. If this is found, then we immediately know that the clauses are unsatisfiable. For example, if the nucleus of the hyper-link is {¬p(X), ¬q(X), r(X)} and the electrons are unit clauses p(a), q(a), and ¬r(a), then the instance {¬p(a), ¬q(a), r(a)} is generated. This will be simplified to the empty clause by unit literal deletion using the electrons. Searching for such hyper-links is much faster than doing a round of hyper-linking, since only unit electrons are used, and since the instances are not saved unless an empty clause is derived, which stops the search. We call this *small proof checking* because such a hyper-link corresponds to a small proof that the set of clauses is unsatisfiable. It turns out that it is only necessary to do this search for nuclei which are input clauses, that is, in the original set S of clauses. If such a small proof can be found for any nucleus, it can be found for an input clause. Also, such a small proof cannot be found unless the original theorem can be proved entirely by unit resolution [WOLB84], it turns out. This check for small proofs has been effective in reducing search times for many theorems in our prover.

We have also implemented a more sophisticated small proof check which searches for small proofs of a more complicated structure. Instead of looking for a hyper-link in which all of the electrons are unit clauses already existing in the set of instances, we look for a hyper-link in which all of the electrons are either existing unit clauses, or derivable by one UR-resolution step. This allows us to find larger proofs, and sometimes terminate the search even earlier than with the simple small proof check described above. However, the time taken for this complicated small proof check can be much larger than the time to hyper-link, and so the time has to be controlled. We do this by defining the size of the small proof to be $(d_1 + d_2 + \ldots + d_n)$, where $M_1, M_2, \ldots, M_n$ are the unit electrons, and if M_i is an existing instance, then d_i is 0, but if M_i is a UR resolvent from a nucleus D_i, then d_i is one less than the number of literals in D_i. Note that a small proof of size 0 corresponds to a proof that can be found by the simple small proof check given above. We first search for small proofs of size 0, then of size 1, then of size 2, et cetera, until some specified time bound is exceeded. We take care to avoid repeated work, so that the same work will not be performed for more than one size bound. Also, we avoid backtracking for ground literals, as in hyper-linking. Suppose the nucleus C is $\{L_1, L_2, \ldots, L_m\}$. Then we pick some clause D_1, unify a literal of D_1 with the complement L_1, and try to unify the other literals of D_1 with complements of unit clauses. Then we pick another clause D_2, unify one literal of D_2 with L_2, and try to unify the other literals of D_2 with complements of unit clauses, and so on. At each step we keep track of the total size of the proof constructed so far. If all clauses D_i can be found for all literals in C, and all their literals can be unified with complements of unit clauses as specified, then we have a small proof and we know S is unsatisfiable.

There are some refinements to this idea. First, time control needs to be done carefully because the time to search for one size bound may be very large. Therefore, we check the total time used so far before and after unifying the literals of each chosen clause D_i. This ensures that the gaps between time checks are not too large. Also, we can delete instances of clauses before doing small proof checking. That is, if clause B_1 is an instance of B_2, then we need not consider B_1 at all during small proof checking. We also order the clauses by number of literals. A simplest small proof is always found first if it exists. A refinement we haven't implemented yet is to not count ground literals of D_i in the size bound. Ground

literals don't cost much time, since no backtracking is done, so a proof with many ground literals can be found quickly. We may consider not only ground literals in D_i but literals that are ground literals by the time they are linked, because of the substitution that has been applied to D_i so far. A further refinement would be to use something like Stickel's PTTP [Sti86] when searching for small proofs, since the MESON strategy used by PTTP is complete and would help for some problems that cannot be helped by our current small proof check. However, this might also cost extra time.

10 Replace Rules

We have a facility in the prover which simulates the idea of replacing predicates by their definitions. This has yielded spectacular improvements in efficiency in set theory problems, and also helps temporal logic theorems. It seems likely to help obtain harder theorems such as the intermediate value theorem, also. This idea was used in Potter and Plaisted [PP88], but fits in more naturally with the current prover because a special tautology tester does not need to be written. A number of provers can solve set theory problems quickly using the idea of replacing predicates by their definitions, using explicit quantifiers. The problem for us with replacing predicates by their definitions is that the definitions may contain quantifiers. Since we are working in a quantifier-free setting, these definitions cannot be directly used. A quantifier-free prover has advantages because a simple unification algorithm may be used. However, we have found a way to get the same effect of replacing predicates by their definitions in this Skolemized, quantifier-free setting.

In more general terms, the replace rule facility permits us to add instances without general search. These rules permit information to propagate more quickly than by hyper-linking. We may consider the replace rules like learned responses, while hyper-linking is like general search or unguided thought used in the absence of specific knowledge of what to do. The replace rules may also be viewed as clause production rules, that is, production rules in a clausal setting. Another way to look at it is to view the replace rules as focusing attention on objects that have already been constructed. The replace rules usually permit a large number of inferences that do not construct new objects. Only when these are exhausted, are new terms constructed by hyper-linking. This seems similar to the general philosophy of our prover, namely, to separate the process of combining literals from the process of reasoning about the instances already obtained, and also seems to correspond to human problem solving methods. Sometimes the replace rules construct new terms, but this can be done in a controlled way.

Suppose we have defined a predicate p(X) as $(q(X) \wedge r(X))$. We would like to replace p(X) by $(q(X) \wedge r(X))$ everywhere. We would also like to do this if p appears negated in a clause. Thus $\neg p(a)$ should be replaced by $\neg(q(a) \wedge r(a))$. Note that the definition of p(X) may be expressed by the formula $p(X) \equiv (q(X) \wedge r(X))$. This corresponds to the following clauses:

1. $\{\neg p(X), q(X)\}$
2. $\{\neg p(X), r(X)\}$
3. $\{p(X), \neg q(X), \neg r(X)\}$

Suppose the clause {¬s(b), p(f(b))} is generated. We get the effect of replacing p(f(b)) by its definition, by adding the instances {¬p(f(b)), q(f(b))} and {¬p(f(b)), r(f(b))} to the set of clauses. These clauses logically imply the desired result (¬s(b) ∨ (q(f(b)) ∧ r(f(b))), which is obtained by replacing p(f(b)) in the clause {¬s(b), p(f(b))} by its definition q(f(b)) ∧ r(f(b)). Similarly, if a clause containing ¬p(c) is generated, then we get the effect of replacing p(c) by its definition, by adding the instance {p(c), ¬q(c), ¬r(c)}. Then it may be necessary to further replace q(c) and r(c) by their definitions, if they exist. We call the literals of clauses 1, 2, and 3 above containing p or ¬p the *distinguished literals* since they are the literals being defined. They are treated differently than the other literals in the clause.

Formally, we define a replacement like a hyper-link, with some modifications.

Definition. If C = $\{L_1, \ldots, L_m\}$ is a clause in a set S of clauses, and $\{L_1, \ldots, L_k\}$ are the distinguished literals of C, then a *replacement* of C in S is a set $\{(L_1, M_1), \ldots, (L_k, M_k)\}$ of links in S such that there exists a substitution Θ such that $L_i\Theta$ and $M_i\Theta$ are complementary for all i, $1 \leq i \leq k$. Θ allows the variables in M_i substituted only by individual constants. A most general such Θ is called the *substitution* of the replacement and CΘ for this Θ is called the *instance* of the replacement. We call C the *nucleus* of the replacement and we call the M_i (or clauses D_i containing M_i) the *electrons* of the replacement. The notation

$$M_1, M_2, \ldots, M_k \rightarrow N_1, \ldots, N_n$$

will be used to indicate the replace rule

$$\{L_1, L_2, \ldots, L_k, N_1, N_2, \ldots, N_n\}$$

where $L_1, L_2, \ldots, L_k$ are the distinguished literals and M_i is $\neg L_i$, for $1 \leq i \leq k$. Thus → represents implication and the distinguished literals are to the left of the arrow.

A distinguished literal may be a *context literal* that is deleted from the instance when generated. Context literals sometimes are useful for reducing the number of instances that can be generated from a replace rule.

The clauses {¬p(f(b)), q(f(b))}, {¬p(f(b)), r(f(b))}, and {p(c), ¬q(c), ¬r(c)} are instances of appropriate replacements indicated above. Our definition of a replacement allows more than one literal to be distinguished. This permits a more general form of definition that combines literals in different clauses.

Sometimes a definition involves a quantifier. If p(X) is defined as (∃Y q(X, Y)), then we can express this definition by the formula (p(X) ≡ (∃Y q(X, Y))), which after conversion to clause form yields the clauses

1. {¬p(X), q(X, f(X))}
2. {p(X), ¬q(X, Y)}

These would be represented in our notation as the rules

1. p(X) → q(X, f(X))
2. ¬p(X) → ¬q(X, Y)

The literals containing p or $\neg$p are the distinguished literals. If a literal p(g(a)) appears in a clause, then the first clause results in the instance {$\neg$p(g(a)), q(g(a), f(g(a)))} being added by replacement. If a literal not p(c) appears in a clause, then a replacement with the second clause as a nucleus results in the instance {p(c), $\neg$q(c, Y)} being added to the set of clauses.

Our prover performs replacements after each round of hyper-linking, and once before the first round of hyper-linking. Replacements are done until no new instances may be generated. Note that replacement may terminate in cases where the usual replacement of predicates by their definitions would not. For example, if a predicate p(X, Y) is defined to be p(Y, X), then replacement of p(a, b) by its definition would successively yield p(b, a), and p(a, b), and p(b, a), ad infinitum. However, the replace rules would only generate the instances {$\neg$p(a, b), p(b, a)} and {$\neg$p(b, a), p(a, b)}, and would then stop.

There is also a technicality that restricts the use of replace rules. Namely, when we apply a replace rule {L, M, N} with distinguished literal L, generating an instance {L′, M′, N′}, then the literal L′ may not be used as an electron in further replacements. This is because this literal represents something that is being replaced, not a part of a definition, and we only want the definitions themselves to be available for future replacement.

We now show how replace rules would help in the proof of a simple set theory problem. Consider the following theorem

$$\forall X, Y(X \cup Y = Y \cup X)$$

which states that the union operation is commutative. The replace rules we needed are listed below:

1. $X = Y \rightarrow X \subseteq Y$
2. $X = Y \rightarrow Y \subseteq X$
3. $\neg(X = Y) \rightarrow \neg(X \subseteq Y), \neg(Y \subseteq X)$
4. $X \subseteq Y \rightarrow \neg(Z \in X), Z \in Y$
5. $\neg(X \subseteq Y) \rightarrow f(X, Y) \in X$
6. $\neg(X \subseteq Y) \rightarrow \neg(f(X, Y) \in Y)$
7. $Z \in (X \cup Y) \rightarrow Z \in X, Z \in Y$
8. $\neg(Z \in (X \cup Y)) \rightarrow \neg(Z \in X)$
9. $\neg(Z \in (X \cup Y)) \rightarrow \neg(Z \in Y)$

These rules are obtained easily from the definitions of $=$, $\subseteq$, and $\cup$. We transform the negation of the theorem into the clause

$$\{\neg(a \cup b = b \cup a)\}$$

The following clauses are generated from the replace rules before the first round of hyper-linking:

1. $\{\neg((a \cup b) \subseteq (b \cup a)), \neg((b \cup a) \subseteq (a \cup b))\}$
2. $\{(a \cup b) \subseteq (b \cup a), f((a \cup b), (b \cup a)) \in (a \cup b)\}$
3. $\{(a \cup b) \subseteq (b \cup a), \neg(f((a \cup b), (b \cup a)) \in (b \cup a))\}$
4. $\{(b \cup a) \subseteq (a \cup b), f((b \cup a), (a \cup b)) \in (b \cup a)\}$
5. $\{(b \cup a) \subseteq (a \cup b), \neg(f((b \cup a), (a \cup b)) \in (a \cup b))\}$
6. $\{\neg(f((a \cup b), (b \cup a)) \in (a \cup b)), f((a \cup b), (b \cup a)) \in a, f((a \cup b), (b \cup a)) \in b\}$
7. $\{f((a \cup b), (b \cup a)) \in (b \cup a), \neg(f((b \cup a), (a \cup b)) \in b\}$
8. $\{f((a \cup b), (b \cup a)) \in (b \cup a), \neg(f((b \cup a), (a \cup b)) \in a\}$
9. $\{\neg(f((b \cup a), (a \cup b)) \in (b \cup a)), f((b \cup a), (a \cup b)) \in b, f((b \cup a), (a \cup b)) \in a\}$
10. $\{f((b \cup a), (a \cup b)) \in (a \cup b), \neg(f((b \cup a), (a \cup b)) \in a\}$
11. $\{f((b \cup a), (a \cup b)) \in (a \cup b), \neg(f((b \cup a), (a \cup b)) \in b\}$

The propositional calculus prover decides that the above clauses are unsatisfiable. Therefore, we have proved the theorem. That is, the union operator is commutative. Notice that we didn't use hyper-linking.

10.1 Application to the Intermediate Value Theorem

We now show how replace rules may help to guide the search for a proof in a problem that is beyond the reach of most theorem provers. Following are first-order axioms for the intermediate value theorem, which says that if a continuous function f is negative at a and positive at b, and $a \leq b$, then there is a point X between a and b such that $f(X) = 0$. Here $p(X, Y)$ means $X \leq Y$.

- % two end points
  $\{p(a, b)\}$
  $\{p(f(a), 0)\}$
  $\{p(0, f(b))\}$
- % least upper bound axioms
  $\{p(X, l), \neg p(X, b), \neg p(f(X), 0)\}$
  $\{p(l, X), p(f(g(X)), 0)\}$
  $\{p(l, X), p(g(X), b)\}$
  $\{p(l, X), \neg p(g(X), X)\}$
- % inequality axioms
  $\{p(X, X)\}$
  $\{p(X, Z), \neg p(X, Y), \neg p(Y, Z)\}$
  $\{p(X, Y), p(Y, X)\}$

- % interpolation axioms
  {p(X, Y), ¬p(X, q(Y, X))}
  {p(X, Y), ¬p(q(Y, X), Y)}

- % continuity axioms
  {p(f(X), 0), ¬p(a, X), ¬p(X, b), ¬p(X, h(X))}
  {¬p(f(Z), 0), p(Z, h(X)), ¬p(a, X), ¬p(X, b), ¬p(Z, X), p(f(X), 0)}
  {p(0, f(X)), ¬p(a, X), ¬p(X, b), ¬p(k(X), X)}
  {¬p(0, f(Z)), p(k(X), Z), ¬p(a, X), ¬p(X, b), ¬p(X, Z), p(0, f(X))}

- % negation of the theorem
  {¬p(f(X), 0), ¬p(0, f(X))}

The least upper bound axioms assert that l is the least upper bound of the set of X such that $f(X) \leq 0$ and $X \leq b$. Call this set A. The interpolation axioms state that there is a real number between any two real numbers. The continuity axioms state that if $f(X) < 0$ then there is an interval following X such that $f(Z) < 0$ for all Z in this interval, and if $f(X) > 0$ then there is an interval preceding X such that $f(Z) > 0$ for all Z in this interval. The proof goes like this: Suppose $f(l) < 0$. Then there is some interval following l such that $f(Z) < 0$ for Z in this interval. This interval cannot include b since $f(b) \geq 0$. Then there is some Z in this interval, $Z \leq b$, such that $f(Z) < 0$, by the interpolation axioms. This contradicts the fact that l is the least upper bound of A. Suppose $f(l) > 0$. Then there is some interval preceding l such that $f(l) > 0$. Then any Z in this interval is an upper bound of A, hence l is not the least upper bound of A. This completes the proof.

One would think that such a simple proof would be easy for a computer. However, most provers cannot obtain this proof. One of the few that can is the prover of Bledsoe and Hines [BH80] at Austin, which uses special inference rules for dense linearly ordered sets. Note that most of the inference steps involve reasoning about terms that have already been constructed, and not many involve constructing new terms, except in certain reasonable ways. We can formalize this by converting the clauses to replace rules, and then the above proof can be entirely obtained by the application of replace rules. We obtain the following replace rules from the axioms:

1. p(X, b), p(f(X), 0) → p(X, l)
2. ¬p(l, X) → p(f(g(X)), 0)
3. ¬p(l, X) → p(g(X), b)
4. ¬p(l, X) → ¬p(g(X), X)
5. ¬p(f(g(X)), 0) → p(l, X)
6. ¬p(g(X), b) → p(l, X)
7. p(g(X), X) → p(l, X)
8. ¬p(Y, X) → p(X, Y)

9. $p(a, X) \rightarrow \neg p(X, b), p(f(X), 0), \neg p(X, h(X))$

10. $p(a, X), p(X, b), \neg p(f(X), 0), p(Z, X), \neg p(Z, h(X)) \rightarrow \neg p(f(Z), 0)$

11. $p(a, X) \rightarrow \neg p(X, b), p(0, f(X)), \neg p(k(X), X)$

12. $p(a, X), p(X, b), \neg p(0, f(X)), p(X, Z), \neg p(k(X), Z) \rightarrow \neg p(0, f(Z))$

13. $p(a, X), p(X, b), p(X, Z), \neg p(0, f(X)) \rightarrow p(k(X), Z), \neg p(0, f(Z))$

14. $p(f(X), 0) \rightarrow \neg p(0, f(X))$

15. $p(0, f(X)) \rightarrow \neg p(f(X), 0)$

16. $p(X, Y), p(Y, Z) \rightarrow p(X, Z)$

17. $\neg p(X, Y) \rightarrow \neg p(X, q(Y, X))$

18. $\neg p(X, Y) \rightarrow \neg p(q(Y, X), Y)$

These replace rules are fairly natural. For example, the least upper bound l is defined in terms of whether p(X,l) and p(l,X) are true for arbitrary X. Hence it is natural to make the literal containing p(X,l) or p(l,X) the distinguished literal.

Using these axioms and replace rules, the prover obtains the following instances among others:

1. $\{p(a, l)\}$

2. $\{\neg p(l, b), p(f(l), 0), \neg p(l, h(l))\}$

3. $\{p(l, h(l)), p(f(g(h(l))), 0)\}$

4. $\{p(l, h(l)), p(g(h(l)), b)\}$

5. $\{p(l, h(l)), \neg p(g(h(l)), h(l))\}$

6. $\{p(l, b), p(g(b), b)\}$

7. $\{p(g(b), b), p(l, b)\}$

8. $\{\neg p(g(h(l)), b), \neg p(f(g(h(l))), 0), p(g(h(l)), l)\}$

9. $\{\neg p(l, b), p(0, f(l)), \neg p(k(l), l)\}$

10. $\{\neg p(0, f(l)), \neg p(f(l), 0)\}$

11. $\{\neg p(l, b), p(f(l), 0), \neg p(g(h(l)), l), p(g(h(l)), h(l)), \neg p(f(g(h(l))), 0)\}$

12. $\{\neg p(f(l), 0), \neg p(0, f(l))\}$

13. $\{\neg p(l,b), p(0, f(l)), p(k(l), b)\}$

14. $\{p(k(l), l), \neg p(k(l), q(l, k(l)))\}$

problem	cpu(sec)
gcd	74.516
lcm	13.700
am8	148.066

Table 2: Times for some other hard problems

15. {p(k(l), l), ¬p(q(l, k(l)), l)}

16. {p(k(l), q(l, k(l))), p(q(l, k(l)), k(l))}

17. {p(q(l, k(l)), l), p(l, q(l, k(l)))}

18. {¬p(l, b), p(0, f(l)), ¬p(l, q(l, k(l))), p(k(l), q(l, k(l))), ¬p(0, f(q(l, k(l))))}

19. {p(0, f(q(l, k(l)))), p(f(q(l, k(l))), 0)}

20. {¬p(q(l, k(l)), k(l)), ¬p(k(l), b), p(q(l, k(l)), b)}

21. {¬p(q(l, k(l)), b), ¬p(f(q(l, k(l))), 0), p(q(l, k(l)), l)}

These instances are detected unsatisfiable by the propositional calculus prover. The prover spends 1467.216 seconds for finding this proof.

It seems that a proper use of replace rules helps significantly in getting this proof. Note that the replace rules are fairly natural. Even the instantiation of Z to the term of the form q(X,Y) is fairly natural; the idea is that if you know there is a non-empty interval (X, Y) in which some property holds, then this property holds for the element q(X, Y) of this interval. This idea is similar to the variable elimination idea of Bledsoe, but we think, more natural. We believe that similar techniques will also help with am8 and other problems in analysis discussed in [WB87].

If we use context literals in the two rules for interpolation something like

- $Y < X, \neg(Y < c), \neg(c < X) \rightarrow q(Y, X) < X$
- $Y < X, \neg(Y < c), \neg(c < X) \rightarrow Y < q(Y, X)$

the proof can be obtained in 137.783 seconds. The literals $\neg(Y < c)$ and $\neg(c < X)$ are context literals here. These rules only generate the term q(Y, X) if there is a need for a Z in the interval (Y, X), so the search space is smaller. We also tried some other hard problems, such as am8 [WB87]. The result is listed in Table 2.

We have not addressed the issue of termination. For this example, it is possible that the rules we have chosen, cause the generation of instances to fail to terminate. This problem can be addressed with a priority system described in section 11. However, we do give some general conditions guaranteeing termination.

10.2 Termination of Replacement

Here we apply some ideas similar to those used to show termination of term rewriting systems [Der82] to prove termination of replacements.

Definition. An ordering $\succeq$ on literals is called a *replacement ordering* if this ordering is reflexive and transitive and has the property that if $L \succeq M$ then for all substitutions Θ, $L\Theta \succeq M\Theta$. Also, we require that for any literal L, the set of M such that $L \succeq M$, must be finite.

For example, we can say that $L \succeq M$ if the size of L is at least as large as that of M, and if each variable in M occurs at least as many times in L as in M. Thus $p(X, f(X)) \succeq q(X, X)$ but we don't have $p(X, f(Y)) \succeq q(X, X)$. The reason is that the substitution of a large term for X could make q larger than p.

Definition. A set R of replace rules satisfies the *termination condition* for a replacement ordering $\succeq$ if for all rules r in R, for all literals L in r, if L is not a distinguished literal of r then there is a distinguished literal M of r such that $M \succeq L$.

Theorem. Let R be a set of replace rules and $\succeq$ be a replacement ordering such that R satisfies the termination condition for $\succeq$. Then replacement using R always terminates.

Proof. Let S be the (finite) set of clauses to which replacement is applied. Whenever an instance r′ of a replace rule r is generated, for every literal L′ in r′, there exists a literal L in S such that $L \succeq L'$. This can be shown by induction on the number of replacements. Since for each L in S, the set of L′ such that $L \succeq L'$, is finite, there are only finitely many literals that can appear in instances r′ obtained by replacements. Thus the number of instances r′ is finite, and replacement will eventually stop.

In general, a set of replace rules will terminate if the distinguished literals are larger than the other literals of each rule, in some reasonable ordering. Rules like $p(X, Y) \rightarrow p(Y, X)$ do not cause a problem; it is only rules like $p(X) \rightarrow p(f(X))$ that cause a problem, because the non-distinguished literal is larger than the distinguished literal in any replacement ordering.

11 The Sliding Priority System

The prover has a priority system that relieves the user of the necessity of setting a maximum size bound for instances that may be generated by hyper-linking. In effect, the prover automatically sets the bound in a reasonable way. This feature has made the prover easier to use, as well as making many proofs easier to obtain. This idea has been explained in another context in the paper [NP90]. The same effect could be obtained by best-first search, but our method permits a simpler control structure. This simpler control structure (breadth-first search) fits in better with the hyper-linking strategy.

We begin with an example. Suppose we decide in advance that some number (say 3) of clauses is all that we can store. Each clause has a priority, which is an integer giving its size or some other measure of complexity. We assume that for each n, there are only finitely many clauses having priority bounded by n. Suppose we want to set a priority bound so that no more than 3 clauses will be retained. Clauses with a priority that is too large will be deleted. However, we do not know in advance what the priority bound will be. Suppose we generate clauses of priority value 4, 10, 5, 7. After these four clauses are generated,

we know that all clauses of size greater than 7 cannot be retained, since there are three clauses with size 7 or less. Therefore, the priority bound is 7. Next, we generate clauses of size 12 and 8, which are deleted because their priorities are larger than 7. Finally, we generate a clauses of priority 6. The priority bound is now 6, and the clause of priority 7 is deleted. Suppose there are no more clauses generated. We call this a round with a work bound of 3. This is called sliding priority because the priority bound keeps decreasing, or "sliding," in each round. If no proof can be found with a work bound of 3, we may double the work bound and try again. Eventually, the work bound will be large enough to obtain a proof, if the strategy is complete. Notice that the prover is automatically setting the priority to insure that the work bound is not exceeded. Notice also that the total number of clauses generated with work bound 3 is 7. The number of clauses that are generated and not deleted right away is 5. We call this the actual work for work bound 3. If a clause is deleted when it is generated, we don't count it in the actual work.

This idea is attractive, and has enabled us to get a number of proofs more easily than without sliding priority, since the behavior of a prover is often very sensitive to the priority bound chosen. This idea permits the prover to be more self-guiding, requiring less input and insight from the user. However, there is a problem with sliding priority, namely, that there is no necessary relationship between the work bound and the actual work for a round of search. For example, if the first clause generated has priority 100, the second has priority 99, the third has priority 98, ..., and so on, the actual work can be 100 for a work bound of 3 (or even 1). If the first clause has priority 10000, the second clause has priority 9999, and so on, the actual work can be 10000 for a work bound of 1. There is no upper bound to the actual work for a given work bound.

We solve this problem by letting the actual work for a round be the sum of the priorities of the clauses generated, rather than the number of clauses generated. Presumably it takes more work to generate and store a clause of size 100 than a clause of size 10. If the work bound is 3, then a clause of size 100 (or 10) would be deleted immediately. With this approach, we can give a good upper bound for the actual work for a given work bound. Suppose the work bound is b. Then the worst case for actual work is when a clause of size b is generated, then a clause of size b-1, then a clause of size b-2, ..., then 2 clauses of size b/2, then 2 clauses of size b/2 - 1, ...then 3 clauses of size b/3, et cetera. The number of clauses generated is then about b/2 + 2(b/2 - b/3) + 3(b/3 - b/4) + ...+ (b - 1)(b/(b-1) - b/b). This simplifies to b/2 + b/3 + b/4 + ...+ b/b, or, b(1/2 + 1/3 + 1/4 + ...+ 1/b), which is proportional to $b \log b$. Thus there is a reasonable upper bound for the number of clauses generated in a round. Also, the sum of the priorities of these clauses is easily seen to be proportional to b^2. This method of counting the work per clause is implemented in our prover.

It is possible to have a more general sliding priority than this, although we have not implemented this idea. Suppose there are two parameters for a clause, such as its size and its depth of nesting of function symbols. Suppose we want to set bounds on the size and depth to insure that a work bound is not exceeded. However, we do not restrict the relationship between size and depth. This gives a kind of two-dimensional sliding priority. Something can still be done in this case. For example, suppose the work bound is 3. To make the illustration simpler, suppose that we only count each clause as one unit of work, regardless of its size and depth. Suppose 3 clauses have been generated, and all have size

less than 10 and depth less than 3. Then a clause of size 11 and depth 5 cannot be retained. The reason is that this clause can only be retained if the size and depth bounds are at least 11 and 5, respectively, and in this case, four clauses will be retained, exceeding the work bound. In the same way, any clause with size at least 10 and depth at least 3 should be deleted. This kind of a sliding priority poses interesting problems in data structures, to be able to efficiently decide whether a clause should be retained.

In general, we may have a partial ordering $\prec$ on clauses such that if C $\preceq$ D, then if D is retained, C should be also. For example, if C has smaller size and depth than D, then C $\preceq$ D in our above example. Or, there may be 3 measures of complexity of a clause, and we say C $\preceq$ D if all measures for D are at least as large as for C. Thus this formalism generalizes one, two, and higher dimensional sliding priorities, and even priorities computed as some linear sum of measures of C. In this case, we can delete a clause D if the number of clauses C with C $\preceq$ D, is at least b. Or, if we associate a non-unit work w(C) with each clause C, we can delete D if the sum of w(C), for C $\preceq$ D, is at least b. However, for this system we need to consider the issue of whether $\preceq$ is efficiently computable, and whether the condition for deleting a clause is efficiently computable.

We have found that our prover is very sensitive to the choice of priority measure. For example, we have tried both the maximum literal size in a clause, and the sum of literal sizes, as priority measures. Some problems that are easily solved with one measure cannot be solved with the other, and vice versa. We have also used the support distances as priority measures. In some tests, we have compared whether the maximum of a collection of priority measures is a better measure than a linear sum. For example, we can let the priority of a clause be the maximum of $c_i \times p_i$ where c_i are positive coefficients and p_i are various complexity measures of the clause. The maximum often seems better, though we now use maximum literal size as our default, which is just one measure of the complexity of a clause. Sometimes the prover works much better when only clauses with a small number of variables are used. Therefore it may make sense to allow the user to weight variables more highly than constant symbols when computing the size of a term. In fact, it may often be useful to only retain ground instances during hyper-linking.

The sliding priority idea can be used to control replacements. Some replace rules may not terminate. To handle this, we can use sliding priority and let each instance of a replace rule contribute to the work for the prover. When the work bound is exceeded, then replace rule instances with a high priority will be deleted. Since there are only a finite number of instances with a bounded priority, the replacement round will eventually stop.

For some problems we found an amusing dependence of the time taken to find a proof on the time limit allocated to the prover. The instances are generated in order, with the nuclei having smallest number of literals used first. This is a simple way of ordering instances by priority. When the time limit is exceeded, those instances generated so far are used in unit simplification, small proof checking, and propositional calculus satisfiability testing. Some problems were proven more quickly with a small time bound than with a large one because of this.

12 The Top-Level Supervisor

It is important that a theorem prover be usable by those with no knowledge of theorem proving strategies, since few users have such knowledge. Also, some applications of a prover should be fully automatic, and a human may not be available to set the options properly in a theorem prover. We have developed a top-level supervisor to entirely free the user from the necessity of understanding the prover or the merits of different strategies. This supervisor embodies much of our expert knowledge about how to prove theorems. Of course, much of our knowledge is also embodied in the prover itself. With the top-level supervisor, all the user has to do is input his problem in clause form, and the prover does the rest. Note that the use of sliding priority already frees the user from the task of setting size bounds. The top-level prover also frees the user from the task of choosing a strategy.

To use the top-level supervisor, the user sets a minimum and a maximum time bound; these are set to 100 seconds and 1600 seconds, respectively, if the user does not set them. The prover also has a list of strategies to try on each problem. When a problem is tried using the top-level supervisor, each strategy on the list is tried in order, with a time limit of 100 seconds. Also, sliding priority is used for each strategy. If a proof is not found, the time limit is multiplied by two, and the process is repeated. This continues until the maximum time bound is exceeded. Often, one strategy is so much better than another for a problem, that it makes sense to try them all, with an increasing time bound, rather than just to try the first strategy with a large time bound, then the second strategy with a large time bound, and so on.

Some strategies, like unit-resulting resolution, are not complete but are very fast when they are applicable. Thus, unit-resulting resolution appears early on the list. Some strategies are complete for restricted kinds of clauses. For example, deleting all instances of a clause is complete for Horn sets, but not in general. The prover will apply such strategies when appropriate. Some strategies require a user support set to be given. If this is not given, these strategies are omitted. Some strategies are not usually helpful, but may be good at times. To deal with this, we permit a time multiplier to be specified with each strategy. If this multiplier is .5, then the time bound is multiplied by .5 when using the strategy. On the initial run, the time bound is 100 seconds, so only 50 seconds are used on the initial run for strategies with a multiplier of .5. Sometimes it is helpful to set a bound on the number of literals in a hyper-link instance. We therefore include in the list of strategies, settings of this bound to 2 and 3. The combination of user support and unit-resulting resolution has been spectacularly effective on some problems, so we include it in the list. Also, the combinations of forward and backward support on alternate rounds have been effective. All these combinations, and others, are tried in a reasonable order, with increasing time bounds as specified above.

We have found that this top-level supervisor often reduces the total time required to find a proof. When we ran it on some problems, we found that unit-resulting resolution often obtained a proof faster than our best previous time. The inconvenience of trying a strategy that may not be complete, inhibited us from using it on these examples, but the top-level supervisor removed that inconvenience. Therefore we feel that this idea is a significant enhancement of the prover, eliminating the need for some tedious bookkeeping by a human in trying a sequence of strategies.

We also have a batch mode for the top-level supervisor, in which a list of problems can all be run together. In this mode, the first strategy with the first time bound is tried on all problems, then the second strategy with the first time bound is tried on all problems not already solved, and so on. In this way, the theorems that can be proven quickly are solved early, and the theorems that are difficult do not delay finding the easy proofs.

13 Temporal Logic

We have applied our prover to temporal logic theorems, with some success. Special methods for temporal logic are known [SC85], but it seemed that a general theorem prover might perform better in some cases, and also permit an easier transition to first-order temporal logic, for which no decision procedure exists. The use of replace rules has been especially helpful for these theorems, and in fact our experience with temporal logic has refined our replace rule mechanism substantially.

We consider discrete linear time temporal logic, with $\Box$ (henceforth), $\Diamond$ (eventually), **U** (until), and $\bigcirc$ (next state) operators. Each formula is true at various times; we use at(F, T) to indicate that a formula is true at time T. To show a formula F is always true, we prove at(F, now) where "now" is a new constant symbol. The operators are defined in the following way:

$$\begin{gathered}
\text{at}(\Box F, T) \equiv (\forall U)\ U \geq T \wedge \text{at}(F, U) \\
\text{at}(\Diamond F, T) \equiv (\exists U)\ U \geq T \wedge \text{at}(F, U) \\
\text{at}(\bigcirc F, T) \equiv \text{at}(F, s(T)), \text{ where } s(T) \text{ is } T{+}1 \\
\text{at}(F\mathbf{U}G, T) \equiv (\exists X)\ ((\forall Y)\ T \leq Y < X \rightarrow (p(Y) \wedge \neg q(Y))) \wedge q(X)) \\
\text{at}(F \wedge G, T) \equiv (\text{at}(F, T) \wedge \text{at}(G, T)) \\
\text{at}(F \vee G, T) \equiv (\text{at}(F, T) \vee \text{at}(G, T)) \\
\text{at}(\neg F, T) \equiv \neg \text{at}(F, T)
\end{gathered}$$

With these definitions, the reader may verify that formulas such as $\Box p$ implies $\Box\bigcirc p$ are true. Such a language is useful for reasoning about concurrent computer programs, in which time is important.

This formalism is ideally suited to replace rules. We can express the definition of henceforth by a clause, as follows:

$$\neg\text{at}(\Box F, T), \neg(U \geq T), \text{at}(F, U)$$

This can be expressed as a replace rule as follows:

$$\text{at}(\Box A, T) \rightarrow \neg(U \geq T), \text{at}(A, U)$$

The following rule is also sometimes useful:

$$\text{at}(\Box A, T), U \geq T \rightarrow \text{at}(A, U)$$

We also have rules for pushing in negation, as follows:

$$\text{at}(\neg\Box A, T) \rightarrow \text{at}(\Diamond(\neg A), T)$$

To deal with eventualities, we define first(F, T) to be the first time U such that $U \geq T$ and such that at(F, U). We then axiomatize the fact that at(F, W) does not hold for W such that $U > W \geq T$. This gives us the following replace rules:

$$\begin{gathered}
\mathrm{at}(\Diamond A, T) \rightarrow \mathrm{at}(A, \mathrm{first}(A, T)) \\
\mathrm{at}(\Diamond A, T) \rightarrow \mathrm{first}(A, T) \geq T \\
\mathrm{at}(\Diamond A, T) \rightarrow \neg(U \geq T), U \geq \mathrm{first}(A, T), \mathrm{at}(\neg A, U) \\
\mathrm{at}(\neg\Diamond A, T) \rightarrow \mathrm{at}(\Box(\neg A), T)
\end{gathered}$$

For next state we have the rules

$$\begin{gathered}
\mathrm{at}(\bigcirc A, T) \rightarrow \mathrm{at}(A, s(T)) \\
\mathrm{at}(\bigcirc A, p(T)) \rightarrow \mathrm{at}(A, T) \\
\mathrm{at}(\neg\bigcirc A, T) \rightarrow \mathrm{at}(\bigcirc(\neg A), T)
\end{gathered}$$

Also, we have some rules for until, as well as some axioms and rules for inequalities, equalities, successor s(T), and predecessor p(T) of time T.

We illustrate how a simple proof may be found entirely by replacement using these rules. Consider the theorem

$$\Box\bigcirc w \rightarrow \bigcirc\Box w$$

To prove it, we negate it and show unsatisfiability. Thus we get the unit clause

$$\{\mathrm{at}(\neg(\Box\bigcirc w \rightarrow \bigcirc\Box w), \mathrm{now})\}$$

Suppose we have the replace rule $X \geq s(Y) \rightarrow p(X) \geq Y$ and also replace rules as above for temporal operators. Replace rules for logical connectives and negation convert the negation of the theorem to the two unit clauses

- $\{\mathrm{at}(\Box\bigcirc w, \mathrm{now})\}$
- $\{\mathrm{at}(\neg\bigcirc\Box w, \mathrm{now})\}$

We then obtain successively

- $\{\mathrm{at}(\bigcirc\neg\Box w, \mathrm{now})\}$
- $\{\mathrm{at}(\neg\Box w, s(\mathrm{now}))\}$
- $\{\mathrm{at}(\Diamond\neg w, s(\mathrm{now}))\}$
- $\{\mathrm{first}(\neg w, s(\mathrm{now})) \geq s(\mathrm{now})\}$
- $\{\mathrm{at}(\neg w, \mathrm{first}(\neg w, s(\mathrm{now})))\}$
- $\{\neg\mathrm{at}(w, \mathrm{first}(\neg w, s(\mathrm{now})))\}$
- $\{p(\mathrm{first}(\neg w, s(\mathrm{now}))) \geq \mathrm{now}\}$
- $\{\mathrm{at}(\bigcirc w, p(\mathrm{first}(\neg w, s(\mathrm{now}))))\}$

- {at(w, first(¬w, s(now)))}

Note that clauses {¬at(w, first(¬w, s(now)))} and {at(w, first(¬w, s(now)))} are complementary.

Using these axioms and rules plus some axioms and rules for equalities and inequalities, we have found a number of proofs of propositional temporal logic theorems very fast. Many of these proofs have been found essentially without search, entirely by applying replace rules to the input clauses and using the propositional calculus decision procedure. For this work, as elsewhere, we have found the *ground substitution* of replacement to be useful; this means that only replacements are done in which all the electrons are ground literals. For example, we solved the temporal theorem

$$\Box\Box w \rightarrow \Box\Diamond w$$

in 3.983 seconds. For the times for other temporal logic theorems, see Table 3.

It is possible to have nontermination of replace rules if care is not taken. For example, suppose the theorem contains the unit clause at(□◇p, now). Suppose we also have the following replace rules:

$$\begin{gathered} at(\Box F, T), U \geq T \rightarrow at(F, U) \\ at(\Diamond F, T) \rightarrow at(F, first(F, T)) \\ at(\Diamond F, T) \rightarrow first(F, T) \geq T \\ X \geq Y, Y \geq Z \rightarrow X \geq Z \end{gathered}$$

Then, using replacement and unit simplification, we can derive the literals

- at(◇p, now)
- at(p, first(p, now))
- first(p, now) ≥ now
- at(◇p, first(p, now))
- at(p,first(p, first(p, now))
- first(p, first(p, now)) ≥ first(p, now)
- first(p, first(p, now)) ≥ now
- at(◇p, first(p, first(p, now)))
- ...

and so on.

We have often found that in order to prove A ≡ B, it is much faster to prove A implies B and B implies A separately. This could be built in to the prover automatically if desired.

One advantage of this formalism is that by modifying the axioms and strategies, we obtain many different proof procedures for temporal logic. Correctness is not a problem because the underlying search is done by a theorem prover. This contrasts with special

no	problem	cpu(sec)
1	$\Diamond\Box w \rightarrow \Box\Diamond\Box w$	6.383
2	$\neg\Diamond w \equiv \Box\neg w$	6.617
3	$\Diamond w \vee \Diamond\neg w$	0.683
4	$w \rightarrow \Diamond w$	0.500
5	$\bigcirc w \rightarrow \Diamond w$	1.783
6	$\Box w \equiv \Box\Box w$	3.050
7	$\Diamond w \equiv \Diamond\Diamond w$	3.367
8	$\Diamond\neg w \equiv \neg\Box w$	1.583
9	$\Box(w_1 \rightarrow w_2) \rightarrow (\Diamond w_1 \rightarrow \Diamond w_2)$	3.067
10	$\Box(w_1 \wedge w_2) \equiv (\Box w_1) \wedge (\Box w_2)$	10.250
11	$\Diamond(w_1 \vee w_2) \equiv (\Diamond w_1) \vee (\Diamond w_2)$	9.717
12	$(\Box w_1 \vee \Box w_2) \rightarrow \Box(w_1 \vee w_2)$	3.250
13	$\Diamond(w_1 \wedge w_2) \rightarrow (\Diamond w_1 \wedge \Diamond w_2)$	3.317
14	$(\Box w_1 \wedge \Diamond w_2) \rightarrow \Diamond(w_1 \wedge w_2)$	2.967
15	$\bigcirc(w_1 \wedge w_2) \equiv (\bigcirc w_1 \wedge \bigcirc w_2)$	8.283
16	$\bigcirc(w_1 \vee w_2) \equiv (\bigcirc w_1 \vee \bigcirc w_2)$	8.317
17	$\bigcirc(w_1 \rightarrow w_2) \equiv (\bigcirc w_1 \rightarrow \bigcirc w_2)$	8.367
18	$\bigcirc(w_1 \equiv w_2) \equiv (\bigcirc w_1 \equiv \bigcirc w_2)$	19.033
19	$\bigcirc\Box w \equiv \Box \bigcirc w$	8.800
20	$\bigcirc\Diamond w \equiv \Diamond \bigcirc w$	9.000
21	$\Box\Diamond\Box w \equiv \Diamond\Box w$	46.983
22	$\Diamond\Box\Diamond w \equiv \Box\Diamond w$	44.467
23	$\Box w \equiv (w \wedge \bigcirc\Box w)$	30.867
24	$\Diamond w \equiv (w \vee \bigcirc\Diamond w)$	27.167
25	$(w \wedge \Diamond\neg w) \rightarrow \Diamond(w \wedge \bigcirc\neg w)$	117.167
26	$(\neg w)\mathbf{U}w \equiv \Diamond w$	6.700
27	$\Box w_1 \wedge \Diamond w_2 \rightarrow w_1\mathbf{U}w_2$	6.350
28	$(w_1\mathbf{U}w_2)\mathbf{U}w_2 \rightarrow (w_1\mathbf{U}w_2)$	14.833
29	$w_1\mathbf{U}(w_1\mathbf{U}w_2) \rightarrow (w_1\mathbf{U}w_2)$	25.133
30	$(\Box w_1 \wedge (w_2\mathbf{U}w_3)) \rightarrow (w_1 \wedge w_2)\mathbf{U}(w_1 \wedge w_3)$	34.983
31	$(w_1 \wedge w_2)\mathbf{U}w_3 \equiv (w_1\mathbf{U}w_3) \wedge (w_2\mathbf{U}w_3)$	27.033
32	$w_1\mathbf{U}(w_2 \vee w_3) \equiv (w_1\mathbf{U}w_2) \vee (w_1\mathbf{U}w_3)$	236.183
33	$(\Diamond w_1 \vee \Diamond w_2) \rightarrow (((\neg w_1)\mathbf{U}w_2) \vee ((\neg w_2)\mathbf{U}w_1))$	119.367
34	$w_1\mathbf{U}(w_2 \wedge w_3) \rightarrow (w_1\mathbf{U}w_2) \wedge (w_1\mathbf{U}w_3)$	32.417
35	$(w_1\mathbf{U}w_3) \vee (w_2\mathbf{U}w_3) \rightarrow (w_1 \vee w_2)\mathbf{U}w_3$	11.817
36	$(w_1 \rightarrow w_2)\mathbf{U}w_3 \rightarrow (w_1\mathbf{U}w_3 \rightarrow w_2\mathbf{U}w_3)$	10.000
37	$(w_1\mathbf{U}w_2) \wedge ((\neg w_2)\mathbf{U}w_3) \rightarrow w_1\mathbf{U}w_3$	33.583
38	$(w_1\mathbf{U}w_2)\mathbf{U}w_3 \rightarrow (w_1 \vee w_2)\mathbf{U}w_3$	20.267
39	$w_1\mathbf{U}(w_2\mathbf{U}w_3) \rightarrow (w_1 \vee w_2)\mathbf{U}w_3$	33.367

Table 3: Times for sample propositional temporal theorems

purpose procedures where correctness may be more of an issue. A disadvantage of our formalism is that we do not have a decision procedure. This means that even if a formula is satisfiable, we may spend a long time looking for a proof. This could be overcome, we feel, by setting a bound on the maximum term size needed to find a proof. However, if a proof cannot be found in a reasonable time, it is not clear that such a bound would help.

14 Abstraction

The idea of abstraction was developed in [Pla81] and seems promising for hard problems, although progress to date has been limited. We give some possibilities for using abstraction in our hyper-linking strategy prover. Since the prover is so sensitive to which measure is used for the priority of a clause, it seems reasonable to let abstraction modify the priority of a clause. The idea of abstraction is to map a set S of clauses onto an "abstract" set T of clauses, find proofs from T, and use these proofs as a guide to the search for a proof from S. Usually it is necessary to find all proofs from T in order to make this method complete; this is a disadvantage since there may be many such proofs and we cannot stop at the first one. To date, mostly simple syntactic abstractions have been tried. We have generally found that abstraction can help bad resolution strategies but not usually good ones. However, abstraction is good at detecting when there is an error in a set of clauses that prevents a proof from being found.

In [Pla89] we presented a new abstraction called the *joint subsumption abstraction* which seems to have many advantages. Possibly this or other abstractions can be used with our hyper-linking strategy prover, in the following way. If a clause C corresponds to a clause in an abstract proof, then we can reduce its priority measure by some amount, thereby in effect preferring this clause. This will guide the search so as to prefer clauses that correspond to clauses in the abstract proofs. Another idea is to prefer clauses that are closely related to such clauses C. We can measure how closely related two clauses are by their distance in the connection graph, analogous to the support distances already computed by the prover. This permits clauses to be used in the proof that do not correspond to clauses in the abstract proof, but which are closely related to them.

15 Extensions

We give some possible extensions to the hyper-linking strategy prover. Some equality method would be useful on many problems. We plan to implement a term-rewriting facility, but more sophisticated techniques like associative-commutative unification [Sti81] are sometimes needed. In general, this prover is well suited to the use of specialized decision procedures, since it generates a set of ground clauses for testing by the propositional prover. Many specialized decision procedures are especially suited to ground clauses. One way to build in a specialized decision procedure, then, is just to add it to the propositional prover. For example, equality could be built in by adding a congruence closure test [NO80] to the propositional prover. However, this is not complete unless the right instances of the input clauses are generated. Such instances will not necessarily be generated unless the equality axioms are included in the input. Including such axioms largely negates the

benefits of a specialized decision procedure. Another possibility is to use a kind of "theory hyper-linking," analogous to the theory resolution of Stickel [Sti85], to generate the needed instances and guarantee completeness. The idea of theory hyper-linking is similar to the idea of theory links mentioned by Murray and Rosenthal [MR86]. One area where a specialized decision procedure might help is temporal logic, where a decision procedure for discrete linearly ordered sets could be applied.

In general, we would like to use semantics in the prover. Since this prover is not subgoal-oriented, as the modified problem reduction format is [Pla88], it is more difficult to use semantics. There is one way to use semantics, however. If the negation of the theorem is of the form A[X_1, ..., X_n] for free variables X_1, ..., X_n, and the axioms are H, then we are trying to show that (H $\wedge$ A[X_1, ..., X_n]) is unsatisfiable. It could be that from semantics we may suspect that (H $\wedge$ A[t_1, ..., t_n]) is unsatisfiable, for certain terms t. For example, it could be that A[t_1, ..., t_n] is false in a model of H. If so, we can instantiate the X_i to t_i and search for a proof. For example, in the intermediate value theorem, we are searching for an X such that f(X) is zero. We may know from examples that X may be the least upper bound l. Then we can attempt to show that f(l) is zero, which is easier. The same idea can be used to instantiate clauses in H by using other models. Another way to use semantics is to modify forward and backward support to support relative to some arbitrary interpretation; forward support implicitly uses the all-positive interpretation and backward support uses the all-negative interpretation. However, the problem with this idea is that it is difficult to tell if an arbitrary non-ground clause is true in a non-trivial interpretation.

It is possible to extend the basic hyper-linking strategy to be more careful about which hyper-links to perform. To illustrate, suppose we are using forward support. Initially, the positive clauses have forward distance 0 and the other clauses have forward distances in the interval [1, ∞]. The first round of hyper-linking will generate some instances of the non-all-positive clauses; these instances will have forward distance 1. The forward distances of the non-all-positive input clauses will then be in the interval [2, ∞], and so on, since the instances of distance 1 will have already been generated. In general, each clause has an interval giving its minimum and maximum forward distance. If the same instance appears twice with intervals [a_1, b_1] and [a_2, b_2], then the interval should be changed to [min(a_1, a_2), max(b_1, b_2)]. Finally, in a hyper-link, we can require that the distances of the nucleus and electrons be close. If C is a nucleus and L is a positive electron from clause D, then there must exist integers i and i+1 such that i is in the interval for D and i+1 is in the interval for C. A similar idea can be applied to all support sets, forward, backward, and user support.

Of course, it might be useful to extend the prover to sorted logics, mathematical induction, and higher order logic. Mathematical induction might be difficult because the prover does not explicitly generate lemmas. However, we feel that there are enough general issues arising in first-order logic to keep us occupied for a while.

Table 4. Running times of sample problems[1]

no	problem	number of input clauses	cpu (sec)	no	problem	number of input clauses	cpu (sec)
1	burstall	19	5.200	2	shortburst	11	1.267
3	prim	9	5.800	4	haspartst1	8	5.033
5	haspartst2	8	46.700	6	ances2	7	0.233
7	num1	7	0.883	8	group1	6	0.950
9	group2	7	4.100	10	ew1	6	0.300
11	ew2	5	0.250	12	ew3	9	0.283
13	rob1	3	0.250	14	rob2	7	3.917
15	dm	4	0.650	16	qw	3	1.633
17	mqw	5	4.017	18	dbabhp	14	8.333
19	apabhp	19	748.450	20	ex4t1	7	131.183
21	ex4t2	7	131.333	22	ex5	14	97.883
23	ex6t1	8	17.517	24	ex6t2	8	17.267
25	wos1	17	30.933	26	wos2	16	21.467
27	wos3	20	2.450	28	wos4	25	20.550
29	wos5	16	4.883	30	wos6	20	37.833
31	wos7	19	6.183	32	wos8	18	3.283
33	wos9	20	6.117	34	wos10	20	35.983
35	wos11	22	41.167	36	wos12	21	2.683
37	wos13	22	3.950	38	wos14	21	4.433
39	wos15	24	3548.966	40	wos16	27	9.567
41	wos17	30	73.183	42	wos18	25	2.933
43	wos19	33	6352.617	44	wos20	33	45003.100
45	wos21	30	2005.217	46	wos22	34	73.467
47	wos23	31	4.700	48	wos24	29	5.750
49	wos25	36	6.267	50	wos26	24	31.900
51	wos27	30	44.850	52	wos28	74	1880.667
53	wos29	40	13.367	54	wos30	42	5.883
55	wos31	20	54992.967	56	wos32	26	3.117
57	wos33	26	9.567	58	ls5	4	0.660
59	ls17	12	8.117	60	ls23	6	1.000
61	ls26	9	1.317	62	ls28	22	4.367

[1]All the times are obtained by running ALS-Prolog on a SUN3/60M workstation with 12 MB memory.

Table 4. Running times of sample problems (continued)

no	problem	number of input clauses	cpu (sec)	no	problem	number of input clauses	cpu (sec)
63	ls29	13	5.400	64	ls35	6	4.350
65	ls36	20	56.466	66	ls37	18	13311.517
67	ls41	11	1.117	68	ls55	13	1.533
69	ls65	20	43.350	70	ls68	15	1.633
71	ls75	16	10.517	72	ls76t1	17	1.817
73	ls87	23	68.667	74	ls100	9	1.000
75	ls103	14	20.367	76	ls105	14	2.450
77	ls106	14	2.500	78	ls108	16	491.350
79	ls111	14	2.483	80	ls112	23	1248.250
81	ls115	21	7.417	82	ls116	20	25.900
83	ls118	29	17.883	84	ls121	21	4.800
85	chang_lee_1	5	0.867	86	chang_lee_2	7	4.150
87	chang_lee_3	5	1.450	88	chang_lee_4	5	0.933
89	chang_lee_5	9	1.167	90	chang_lee_6	9	1.317
91	chang_lee_7	7	0.850	92	chang_lee_8	9	5.950
93	chang_lee_9	8	4.667	94	example	6	21.467
95	expq	4	0.167	96	exx	4	0.267
97	exx5	9	5.533	98	exx7	17	4.467
99	latinsq	17	88.500	100	liar	8	7.650
101	salt	44	32.167	102	schubert	30	135.950
103	stack	3	0.167	104	ph2	3	0.117
105	ph3	9	0.167	106	ph4	22	0.567
107	ph5	45	1.800	108	ph6	81	7.283
109	ph7	133	39.733	110	ph8	204	269.417
101	il	3	27.983	112	sl	3	23.133
113	ipl	4	132.283	114	pl	3	1948.500

References

[ABCM88] P. E. Allen, S. Bose, E. M. Clarke, and S. Michaylov. PARTHENON: A parallel theorem prover for non-horn clauses, system abstract. In *Proceedings of the 9th International Conference on Automated Deduction*, pages 764–765, 1988.

[And81] P. B. Andrews. Theorem proving via general matching. *Journal of the Association for Computing Machinery*, 28:193–214, 1981.

[BH80] W. W. Bledsoe and L. Hines. Variable elimination and chaining in a resolution-based prover for inequalities. In *Proceedings of the 5th International Conference on Automated Deduction*, pages 70–87, 1980.

[Bib82] W. Bibel. *Automated Theorem Proving.* Vieweg, 1982.

[Bun88] A. Bundy. The use of explicit proof plans to guide inductive proofs. In *Proceedings of the 9th International Conference on Automated Deduction*, pages 111–120, 1988.

[Cho84] S.-C. Chou. Proving elementary geometry theorems using Wu's algorithm. In W. W. Bledsoe and D. Loveland, editors, *Automated Theorem Proving: After 25 Years, Contemporary Mathematics*, volume 29, pages 243–286. 1984.

[CL73] C. Chang and R. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York, 1973.

[Dav63] M. Davis. Eliminating the irrelevant from machanical proofs. In *Proceedings Symp. of Applied Math*, volume 15, pages 15–30, 1963.

[Der82] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science.* North-Holland, Amsterdam, 1990.

[dK86] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[dK89] J. de Kleer. A comparison of ATMS and CSP techniques. In *Proceedings of 11th International Joint Conference on Artificial Intelligence*, pages 290–296, 1989.

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

[Gil60] P. C. Gilmore. A proof method for quantification theory. *IBM Journal of Research and Development*, 4:28–35, 1960.

[Hin88] L. Hines. Hyper-chaining and knowledge-based theorem proving. In *Proceedings of the 9th International Conference on Automated Deduction*, pages 469–486, 1988.

[HW74] L. Henschen and L. Wos. Unit refutations and Horn sets. *Journal of the Association for Computing Machinery*, 21:590–605, 1974.

[JP84] S. Jefferson and D. Plaisted. Implementation of an improved relevance criterion. In *First Conference on Artificial Intelligence Applications*, pages 476–482, 1984.

[Lew80] H. Lewis. Complexity results for classes of quantificational formulas. *Journal of Computer and System Sciences*, 21:317–353, 1980.

[Lov78] D. Loveland. *Automated Theorem Proving: A Logical Basis.* North-Holland, New York, 1978.

[LS86] Sterling L. and E. Shapiro. *The Art of Prolog.* The MIT Press, Massachusetts, 1986.

[McA80] D. McAllester. An outlook on truth maintenance. Technical Report AIM-551, Artificial Intelligence Laboratory, MIT, Cambridge, MA, 1980.

[MCA82] D. Miller, E. Cohen, and P. B. Andrews. A look at TPS. In *Proceedings of the 6th International Conference on Automated Deduction*, pages 50–69, 1982.

[McC89] W. W. McCune. *Otter 1.0 Users' Guide.* Mathematics and Computer Science Division, Aggonne National Laboratory, Argonne, Illinois, January 1989.

[MR86] N. Murray and E. Rosenthal. Theory links in semantic graphs. In *Proceedings of the 8th International Conference on Automated Deduction*, pages 353–364, 1986.

[NO80] G. Nelson and D. Oppen. Fast decision procedures based on congruence closure. *Journal of the Association for Computing Machinery*, 27:356–364, 1980.

[NP90] X. Nie and D. Plaisted. A complete semantic back chaining proof system. In *Proceedings of the 10th International Conference on Automated Deduction*, 1990.

[NP90] X. Nie and D. Plaisted. Refinements to depth-first iterative deepening search in theorem proving. *Artificial Intelligence*, 41:223–235, 1989/90.

[Pel86] F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986.

[PG86] D. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986.

[Pla80] D. Plaisted. An efficient relevance criterion for mechanical theorem proving. In *Proceedings of the First Annual National Conference on Artificial Intelligence*, pages 79–83, 1980.

[Pla81] D. Plaisted. Theorem proving with abstraction. *Artificial Intelligence*, 18:227–261, 1981.

[Pla88] D. Plaisted. Non-Horn clause logic programming without contrapositives. *Journal of Automated Reasoning*, 4:287–325, 1988.

[Pla89] D. Plaisted. Mechanical theorem proving. In R. Banerji, editor, *A Sourcebook on Formal Techniques in Artificial Intelligence.* Elsevier, Amsterdam, 1989.

[PM88] D. Plaisted and U. Meyer. A matching strategy for first-order theorem proving. 1988.

[PP88] R. Potter and D. Plaisted. Term rewriting: Some experimental results. In *Proceedings of the 9th International Conference on Automated Deduction*, pages 435–453, 1988.

[PPV60] D. Prawitz, H. Prawitz, and N. Voghera. A mechanical proof procedure and its realization in an electronic computer. *Journal of the Association for Computing Machinery*, 7:102–128, 1960.

[Rob63] J. Robinson. Theorem proving on the computer. *Journal of the Association for Computing Machinery*, pages 163–174, 1963.

[Rob65] J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12:23–41, 1965.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the Association for Computing Machinery*, 32:733–749, 1985.

[SS82] J. Siekmann and P. Szabo. Universal unification and a classification of equational theories. In *Proceedings of the 6th International Conference on Automated Deduction*, pages 369–389, 1982.

[Sti81] M. E. Stickel. A unification algorithm for associative-commutative functions. *Journal of the Association for Computing Machinery*, 28:423–434, 1981.

[Sti85] M. E. Stickel. Automated deduction by theory resolution. *Journal of Automated Reasoning*, 1:333–355, 1985.

[Sti86] M. E. Stickel. A prolog technology theorem prover: Implementation by an extended prolog compiler. In *Proceedings of the Eight International Conference in Automated Deduction*, pages 573–587, 1986.

[Wal84] C. Walther. A mechanical solution of schubert's steamroller by many-sorted resolution. In *Proceedings of the 4th National Conference on Artificial Intelligence*, pages 330–334, 1984.

[WB87] T. C. Wang and W. W. Bledsoe. Hierarchical deduction. *Journal of Automated Reasoning*, 3:35–77, 1987.

[WOLB84] L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning: Introduction and Applications.* Prentice-Hall, 1984.

[WRC65] L. Wos, G. Robinson, and D. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *Journal of the Association for Computing Machinery*, 12:536–541, 1965.

[Wu78] Went-tsün Wu. On the decision problem and the mechanization of theorem proving in elementary geometry. *Scientia Sinica*, 21:159–172, 1978.