Journal of Automated Reasoning **9**: 25–42, 1992.

© 1992 *Kluwer Academic Publishers. Printed in the Netherlands.*

Eliminating Duplication with the Hyper-Linking Strategy*

SHIE-JUE LEE and DAVID A. PLAISTED
Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175, U.S.A. Tel. (919)962-{1934/1751}, {lee/plaisted}@cs.unc.edu.

(Received 24 August 1990; accepted 19 October 1990)

Abstract. The efficiency of almost all theorem proving methods suffers from a phenomenon called duplication of instances of clauses. In this paper, we present a novel technique, called the hyper-linking strategy, to eliminate such duplication. This strategy is complete for the full first-order predicate calculus. We show the effectiveness of this strategy by comparing it with other proving methods. We give empirical evidence that both the Davis–Putnam procedure and the hyper-linking strategy are comparable to each other and better than other common theorem proving strategies on propositional calculus problems. The fact that the Davis–Putnam procedure is faster than resolution and other common methods on propositional problems seems not to be appreciated by a large segment of the theorem proving community. Also, we give empirical evidence that the hyper-linking strategy is better than other common theorem proving methods on near-propositional problems like logic puzzles. We attempt to explain the superior behavior of the hyper-linking strategy and the Davis–Putnam procedure by examining the kinds of duplication that can occur during the search with the different methods. In addition, we show the completeness of the hyper-linking strategy combined with several support strategies.

Key words. Theorem proving, links, hyper-links, instances, the propositional calculus decision procedure.

1. Introduction

Duplication of instances of clauses is a common phenomenon during the process of inference in various theorem proving methods. This duplication is a significant hindrance to these theorem proving methods. There are two kinds of duplication. The first we call *duplication by combination*. This appears in resolution [28], model elimination [14], the connection graph method [3], the modified problem reduction format [24], and other strategies. The same instances of a clause can contribute to many resolvents, for example. Such duplication may cause combinatorial problems since a small number of formulas may combine in many different ways. The second kind of duplication is *duplication by case analysis*. This appears in the linked conjunct procedure of [6], mating strategy [1], as well as in the Davis–Putnam propositional calculus decision procedure [6, 7]. 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. Such duplication may waste work since identical clauses are to be processed again and again.

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

Given clauses C and D, resolution computes a most general unifier Θ and then combines literals of $C\Theta$ and $D\Theta$ to obtain a resolvent E. Note that if C and $C\Theta$ are identical, then the same literals may appear in both C and the resolvent E. This is one kind of duplication of instances by combination. Another kind appears when a clause C resolves against many clauses $D_1, D_2, \ldots, D_n$ using the substitutions $\Theta_1, \Theta_2, \ldots, \Theta_n$. 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. Duplication by combination occurs for similar reasons in model elimination and the connection graph method. The linked conjunct procedure of [6] involves duplication by case analysis. Given a set of S 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, the Davis–Putnam procedure of [6] (which differs slightly from that of [7]) 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.

A theorem prover may gain efficiency by avoiding duplication of instances of clauses. Such avoidance can be achieved by reducing first order logic to propositional calculus and then applying a propositional calculus decision procedure. In this paper, we present a novel strategy, the hyper-linking strategy, of this kind. The strategy is a refutation theorem proving method based on Skolemized first-order formulae (clause form). It basically generates selected instances of the input clauses by a process called hyper-linking, then applies a propositional calculus decision procedure to test the unsatisfiability of the ground set of the retained clauses. The strategy is complete for the full first-order predicate calculus.

The hyper-linking strategy has some analogies to human problem solving methods. The hyper-linking phase may be considered as creative and the unsatisfiability test phase as analytic. It is also related to the associative network model of artificial 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. The hyper-linking strategy also has some similarity to the connection graph method of Bibel [3].

A prover which is an implementation of the new strategy has been written in Prolog. Experiments show that the prover works well. We compare the hyper-linking strategy prover with other theorem provers and show that both the Davis–Putnam procedure and the hyper-linking strategy are comparable to each other and better than other

common theorem proving strategies on propositional calculus problems. Also, we show that the hyper-linking strategy is better than other common theorem proving methods on near-propositional problems like logic puzzles. A near-propositional problem is one in which the function symbols play a minor role; for example, a problem with only variables and constants in the input clauses but no k-ary function symbols for $k \geqslant 1$, is near-propositional. The hyper-linking strategy prover is also good on other problems.

2. The Hyper-Linking Strategy

The philosophy of the hyper-linking strategy is never to combine literals from two clauses. The method consists of a two stage process. The first stage, hyper-linking, generates the selected instances of the given clauses. The second stage, a propositional unsatisfiability test, tests for unsatisfiability on the ground set of clauses retained in the database using a propositional calculus decision procedure. We completely avoid duplication by combination in the hyper-linking stage. A little duplication by case analysis occurs in the propositional calculus decision procedure. However, there are at most two cases, TRUE and FALSE, to be considered per literal. It turns out that this propositional calculus decision procedure is so efficient that such duplication is negligible most of the time. The two stages are performed iteratively until the input problem is solved or until the time limit is exceeded.

We keep clauses as lists of literals, as in resolution. All literals are on an equal footing. This contrasts with the modified problem reduction format of [24], 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 method without having to deal with the complexities of contrapositives.

2.1. HYPER-LINKING

Given a set S clauses, define a *link* in S to be a pair (L, M) of literals such that both L and M appear in (distinct) clauses in S and such that L and the complement of M are unifiable. We may have to rename the variables in L or M so that L and M have no common variables. We treat (L, M) and (M, L) as identical. Note that this is essentially the same as a connection in Kowalski's proving procedure for Horn clauses [11] or in connection graph resolution [3], except that we don't explicitly store the links, so that they cannot be deleted as they are in Kowalski's proving procedure or connection graph resolution.

DEFINITION. Suppose S is a set of clauses. The *connection graph* of S, denoted (S, E), is the graph whose *nodes* are the clauses of S, and which has an *edge* labeled (L_1, L_2) in E between C_1 and C_2 if (L_1, L_2) is a link and C_1 and C_2 contain L_1 and L_2 respectively.

DEFINITION. Suppose S is a set of clauses. Gr(S) is defined to be $\{C\Phi\colon C \in S\}$, where Φ is a substitution replacing all variables of all clauses in S by a new constant symbol \$. We call Gr(S) a *ground set* of S.

For example, if S is $\{\{\neg p(X)\}, \{q(X, Y)\}\}$ then Gr(S) is $\{\{\neg p(\$)\}, \{q(\$, \$)\}\}$.

DEFINITION. Suppose (S, E) is a connection graph. If $C = \{L_1, \ldots, L_m\}$ is a clause then a *hyper-link* of C in E is a set $\{(L_1, M_1), \ldots, (L_m, M_m)\}$ of links in E such that there exists a substitution Φ such that $L_i\Theta$ and $M_i\Theta$ are complementary for all i, $1 \leqslant i \leqslant m$. We may have to rename the variables in L_i or M_i so that L_i and M_i have no common variables. A most general such Θ is called the *substitution* of the hyper-link and $C\Theta$ 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.

Suppose we have the following clauses: $\{p(a)\}$, $\{\neg q(X), r(a, X)\}$, $\{\neg p(Y), \neg s(Z), \neg r(Y, Z)\}$, $\{q(b)\}$, and $\{s(b)\}$. Let's call them C_1 through C_5 respectively. The connection graph for these clauses is shown in Figure 1. Each node represents one clause. There are four links (edges). m_1 is $(p(a), \neg p(Y))$, m_2 is $(q(b), \neg q(X))$, m_3 is $(r(a, X), \neg r(Y, Z))$, and m_4 is $(\neg s(Z), s(b))$. There are five hyper-links. The hyper-link with nucleus C_1 contains link m_1. The hyper-link with nucleus C_2 contains links m_2 and

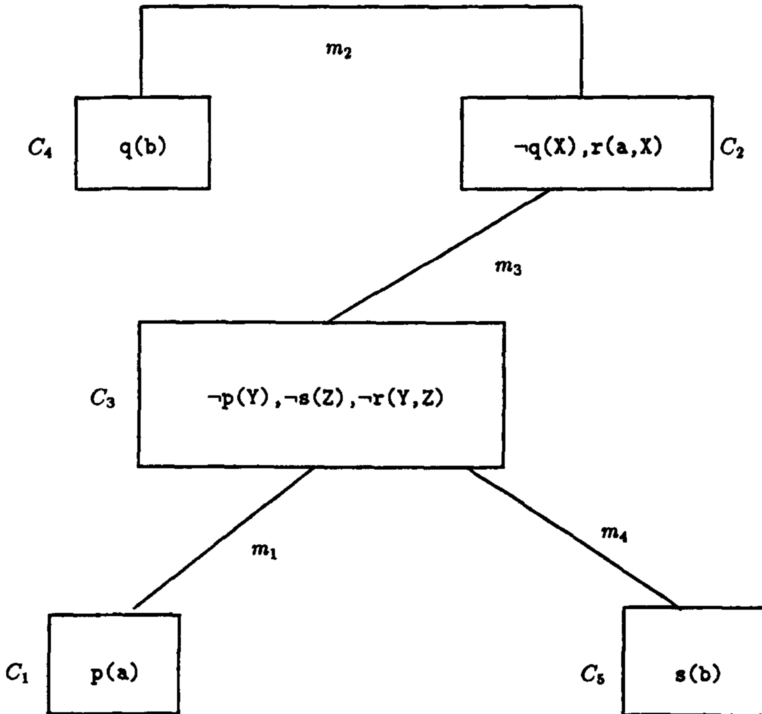

Fig. 1. A connection graph.

m_3. The hyper-link with nucleus C_3 contains links m_1, m_3, and m_4. The hyper-link with nucleus C_4 contains link m_2. The hyper-link with nucleus C_5 contains link m_4.

Given a nucleus $C = \{L_1, \ldots, L_m\}$ of a hyper-link $\{(L_1, M_1), \ldots, (L_m, M_m)\}$ the computation of an instance is as follows: Compute the most general unifier θ_i such that

$$L_i\theta_1 \cdots \theta_{i-1}\theta_i = (\neg M_i)\theta_i$$

for $1 \leqslant i \leqslant m$. The substitution Θ of the hyper-link then is $\theta_1 \cdots \theta_m$. The instance corresponding to this substitution is $C\Theta$.

The hyper-linking stage works this way: suppose (S, E) is a connection graph. For each clause C in S, all hyper-links of C in E are computed, together with their substitutions Θ. Then, the set of all such instances $C\Theta$ and consequent links are added to (S, E), for all clauses C in S and all substitutions Θ of their hyper-links. This results in a new graph (S', E'). We call this expansion of (S, E) to (S', E') a *round of hyper-linking* or a *round*. For example, if $\{\neg p(X), q(X)\}$ is in S and literals $p(a)$ and $\neg q(a)$ appear in clauses in S, then the instance $\{\neg p(a), q(a)\}$ is generated. If $\{\neg p(X), q(f(X))\}$ is in S and the literals $p(a)$ and $\neg q(X)$ appear in clauses in S then the instance $\{\neg p(a), q(f(a))\}$ is generated. However, the set E of links is not stored explicitly.

As we mentioned earlier, the hyper-linking strategy performs hyper-linking and unsatisfiability tests iteratively. Suppose S' is the set of clauses retained after a round of hyper-linking. We then test $\mathrm{Gr}(S')$ for propositional unsatisfiability using the propositional calculus decision procedure to be explained later. We now give an example to show how the hyper-linking strategy works. Suppose S is $\{\{p(a)\}, \{\neg p(X), p(f(X))\}, \{\neg p(f(f(a)))\}\}$. We call these clauses C_1, C_2, and C_3, respectively. There are three hyper-links. First, we consider the hyper-link of $(\neg p(X), p(a))$, $(p(f(X)), \neg p(X))$ with nucleus C_2. The instance of this hyper-link is $\{\neg p(a), p(f(a))\}$. Another hyper-link contains $(\neg p(X), p(f(X)))$ and $(p(f(X)), \neg p(X))$ and the instance of this hyper-link with nucleus C_2 is $\{\neg p(f(X)), p(f(f(X)))\}$. The third hyper-link contains $(\neg p(X), p(f(X)))$ and $(p(f(X)), \neg p(f(f(a))))$ and the instance of this hyper-link with nucleus C_2 is $\{\neg p(f(a)), p(f(f(a)))\}$. This completes the first round of hyper-linking. At this point the set S' of clauses is $\{\{p(a)\}, \{\neg p(X), p(f(X))\}, \{\neg p(f(f(a)))\}, \{\neg p(a), p(f(a))\}, \{\neg p(f(X)), p(f(f(X)))\}, \{\neg p(f(a)), p(f(f(a)))\}\}$. We ground all clauses, replacing all variables with \$, obtaining the set $\mathrm{Gr}(S')$ of ground instances $\{\{p(a)\}, \{\neg p(\$), p(f(\$))\}, \{\neg p(f(f(a)))\}, \{\neg p(a), p(f(a))\}, \{\neg p(f(\$)), p(f(f(\$)))\}, \{\neg p(f(a)), p(f(f(a)))\}\}$. This set is found to be propositionally unsatisfiable by the propositional calculus decision procedure to be presented later. Note that if we have grounded the set S of input clauses, we would get $\{\{p(a)\}, \{\neg p(\$), p(f(\$))\}, \{\neg p(f(f(a)))\}\}$. This set is satisfiable. The extra instances generated by hyper-linking were needed for the proof.

2.2. THE PROPOSITIONAL CALCULUS DECISION PROCEDURE

After a round of hyper-linking, the set S of clauses available so far is given to the propositional calculus decision procedure for an unsatisfiability test. First, all variables in S are replaced by a new individual constant \$ to obtain a set $\mathrm{Gr}(S)$ of ground clauses.

Thus, the clause $\{\neg p(X, Y), q(Y)\}$ would be replaced by $\{\neg p(\$, \$), q(\$)\}$, for example. Then Gr($S$) is tested for unsatisfiability using the propositional calculus decision procedure. If Gr(S) is unsatisfiable, then the ground instantiation problem has been solved and we know S is unsatisfiable. Otherwise, more rounds of hyper-linking and so on, are done.

The propositional calculus decision procedure is a modification to the Davis–Putnam procedure [6]. It uses a technique similar to dependency-directed back-tracking [8, 9] and McAllester's RUP [16, 17] in the area of Artificial Intelligence. One disadvantage of the Davis–Putnam procedure is that it carries out two cases in a case analysis all the time, which results in unnecesary duplication by case analysis. The motivation of our propositional calculus decision procedure is to remove such unnecessary duplication. Our propositional calculus decision procedure always performs about as well as the Davis–Putnam procedure on the problem tested, and for some problems it performs much better.

Although the satisfiability problem is NP-complete [5], we have found that the time taken by the propositional calculus decision procedure is usually small. It is only for a few problems such as the 'pigeonhole problems' [23] that the propositional calculus decision procedure takes most of the time. It is surprising that such a simple method works so well. Also, since the only duplication by case analysis in the hyper-linking strategy occurs in the propositional calculus decision procedure, the speed of the propositional calculus decision procedure shows that this duplication is negligible most of the time.

The following simplification rules are used by the propositional calculus decision procedure:

1. $\neg$FALSE can be replaced by TRUE.
2. A clause containing TRUE can be deleted.
3. $\neg$TRUE can be replaced by FALSE.
4. FALSE may be deleted from a clause containing FALSE.
5. A clause containing both L and $\neg L$ can be deleted.
6. TRUE may be deleted from a set of clauses containing TRUE.
7. A set of clauses containing FALSE may be replaced by FALSE.

Thus, $\{\{\text{TRUE}, p\}, \{\text{FALSE}, q\}\}$ simplifies to $\{\{q\}\}$.

The propositional calculus decision procedure basically performs unit simplification and case analysis iteratively. It is described as follows:

```
procedure PC(S)
    simplify S to S' using simplification rules;
    If S' is TRUE or FALSE, return S';
    If S' contains a unit clause C then % do unit simplification
      if C is {L} for positive literal L,
        replace L by TRUE in S', and simplify S' to S'' using simplification rules;
        return PC(S'');
      if C is {¬L} for positive literal L,
```

```
      replace L by FALSE in S′, and simplify S′ to S″ using simplification rules;
      return PC(S″);
  If none of the above holds, then
    if S′ does not contain a positive or a negative clause
    then return TRUE
    else let C be a smallest negative clause in S′; % do case analysis
        pick a literal ¬L in C;
        replace L by FALSE, and simplify S′ to S1 using simplification rules;
        if PC(S1) is TRUE then return TRUE
        else if the assignment of FALSE to L was not
            needed to show that S1 is unsatisfiable
          then return FALSE % right cutoff
          else replace L by TRUE, and simplify S′ to S2 using simplification rules;
              if PC(S2) is TRUE, then return TRUE
              else if the assignment of TRUE to L was not
                  needed to show that S2 is unsatisfiable
                then return FALSE % left cutoff
                else return FALSE;
  end PC.
```

This procedure returns TRUE if S is satisfiable, FALSE otherwise. It is similar to the Davis–Putnam procedure except for the following features:

1. We can detect the input set of clauses satisfiable earlier than the Davis–Putnam procedure. Whenever the set being considered does not contain a positive or a negative clause, we conclude that the input set of clauses is satisfiable. Notice that the Davis–Putnam procedure can't conclude that the input set of clauses is satisfiable until an empty set is derived.
2. The clause used for case analysis is chosen as a smallest negative clause, if one exists. This simulates backward-chaining which is goal-oriented and is usually a good strategy.
3. The major improvement to the Davis–Putnam procedure 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 the Davis–Putnam procedure. Left cutoffs do not save work on the current case analysis.
4. We don't need a special procedure to carry out pure literal deletion. Pure literal deletion is essential to the Davis–Putnam procedure. Suppose L is a pure literal and is not deleted. The Davis–Putnam procedure may choose L to do case analysis.

Then both L assigned TRUE and L assigned FALSE are to be considered, which wastes time. Case analysis in our propositional calculus decision procedure carries out pure literal deletion automatically. Suppose a pure literal p (p is negative) is picked in a case analysis. All the clauses containing occurrences of p in the set being considered are deleted. The remaining clauses do not contain occurrences of $\neg p$ since p is a pure literal. So the proof of the subcase with p does not use p and we have a right cutoff. Therefore we don't need to try the other subcase. This essentially achieves the same effect done by pure literal deletion on p. However, we don't take any action on p if p is positive since p can't be picked by case analysis.

2.3. SOUNDNESS AND COMPLETENESS

The hyper-linking strategy is obviously sound, that is, the proof that is found logically follows the axioms. This is justified by the following universal instantiation rule:

$$\frac{\forall X F(X)}{F(c)}$$

where F is a formula and c is some arbitrary constant in the domain.

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, C is an instance of $\phi(C)$.

LEMMA 1. *Let T be an unsatisfiable set of ground instances of set S of clauses. Suppose* $\mathrm{Gr}(S)$ *is satisfiable. Let*

$$\phi : T \to S$$

be an instance mapping. Extend ϕ to map the literals of C to the corresponding literals of $\phi(C)$; 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 $\mathrm{Gr}(\phi(L))$ *and* $\mathrm{Gr}(\phi(M))$ *are not complementary.*

Proof. Let I be a model of $\mathrm{Gr}(S)$. Suppose the lemma is false; then for all literals L, M of clauses in T, if L and M are complementary then $\mathrm{Gr}(\phi(L))$ and $\mathrm{Gr}(\phi(M))$ are complementary. Then I can be used to give a model I' of T defined by $I' \models L$ if and only if $I \models \mathrm{Gr}(\phi(L))$. 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

$$\phi : T \to S$$

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

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 $C\theta$ and $D\theta$, 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 and T such that L and M are complementary but $\mathrm{Gr}(\phi(L))$ and $\mathrm{Gr}(\phi(M))$ are not complementary. However, it must be that their most general unifier θ binds at least one variable to a non-variable term. Then,

$$\|\phi(L)\theta\| + \|\phi(M)\theta\| > \|\phi(L)\| + \|\phi(M)\|$$

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

THEOREM 3. *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 4. *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. From the theorem we know that there is a sequence of link operations producing a set of instances as desired. We convert this into a sequence of hyper-link operations. For this part of the proof we assume that T is a minimal unsatisfiable set of ground instances and R is the corresponding minimal sequence of link operations. Suppose a link operation between clauses C and D is in R. Suppose this link operation produces the instances $C\theta$ and $D\theta$. We show that there is a hyper-link operation on C producing an instance of $C\theta$ suitable for our purposes, and similarly for D. Suppose C is $\{L_1, \ldots, L_m\}$. Now C must be $\phi(C')$ for some ground clause $C' = \{L'_1, \ldots, L'_m\}$ in T. For each L'_i, pick clause D_i in T and literal M_i in D_i such that $\mathrm{Gr}(L'_i)$ and $\mathrm{Gr}(M_i)$ are complementary. Such clauses must exist since T is minimal unsatisfiable (since clauses containing pure literals can be deleted without affecting satisfiability). Then if we work on the hyper-link $\{(L_1, \phi(M_1)), \ldots, (L_m, \phi(M_m))\}$ we will obtain an

instance of $C\theta$. As in the proof of the theorem, we can construct a new instance mapping ϕ' and show that if a suitable set of ground instances has not been found, $\|\phi'\| > \|\phi\|$, hence we will eventually find a set of instances as desired.

3. Unit Simplification

The retained clauses after each hyper-linking round can be simplifed by a unit simplification phase which is analogous to operations in the Davis–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 $\{\{p\}, \{\neg p, q\}, \{\neg q, r\}\}$. Then unit literal deletion with p produces $\{\{p\}, \{q\}, \{\neg q, r\}\}$. 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.

Unit simplification simplifies 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. Moreover, unit simplifiction maintains the completeness of the hyper-linking strategy.

4. Support Sets

In resolution, set-of-support strategies [32] are widely used. Given a set S of clauses, a support set is a subset T of S such that $S - T$ is satisfiable. Such strategies restrict the generation of irrelevant resolvents and seem essential for applications in which there are a large number of clauses. The idea can also be used in the hyper-linking strategy to improve its efficiency. By specifying support sets, we can restrict which hyper-links are performed. Thus reduces the number of instances generated in each round of hyper-linking.

We provide three basic support strategies. One simulates forward chaining, another simulates backward chaining, and the other 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 in a given round.

Let S be the set of input clauses. We have the following definition.

DEFINITION. A *forward support set* is the set of all positive clauses in S. A *backward support set* is the set of all negative clauses in S. A *user support set* is a set of clauses provided by the user. If T is the user support set then we assume that $S - T$ is satisfiable.

We say the clauses in the support set are forward supported, backward supported, or user supported, respectively. As the hyper-link operation proceeds, the number of supported clauses increases. More and more clauses are supported according to the following criterion. Suppose C is a nucleus of a hyper-link and $L_1 \cdots L_m$ are the positive literals that are electrons for C and $M_1 \cdots M_n$ 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 $A_1 \cdots A_m$ be the clauses containing $L_1 \cdots L_m$, respectively, and let $B_1 \cdots B_n$ be the clauses containing $M_1 \cdots M_n$. 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. By specifying that all instances be user supported, forward supported or backward supported, the number of instances generated can be reduced.

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.

The use of support sets maintains completeness of the hyper-linking strategy as shown below.

Let I be an interpretation. We define $nt(I)$ to be the number of propositions assigned TRUE by I.

LEMMA 5. *Suppose S is a minimal unsatisfiable set of ground clauses obtained by hyper-linking. For all interpretations I there is a clause C in S such that C is false in I and C is forward supported after some number of rounds of hyper-linking with forward support strategy.*

Proof. The proof is done by induction on $nt(I)$. Suppose $nt(I)$ is 0. Then I assigns all propositions FALSE. Since S is unsatisfiable, some clause C in S is assigned FALSE by I. C must be positive, and therefore C is forward supported.

Suppose the lemma is true for $nt(I) < n$. We show it for $nt(I) = n$. Let L_1, $L_2, \ldots, L_n$ be propositions assigned TRUE by I. Let $I_1, I_2, \ldots, I_n$ be defined this way:

- I_1 assigns $L_2, L_3, \ldots, L_n$ TRUE, others FALSE
- I_2 assigns $L_1, L_3, \ldots, L_n$ TRUE, others FALSE

  ⋮
- I_n assigns $L_1, L_2, \ldots, L_{n-1}$ TRUE, others FALSE

Thus $nt(I_j) = n - 1$. Since S is unsatisfiable, none of I_j makes S true. By assumption, there is a clause C_j false in I_j. If C_j does not contain L_j then C_j is false in I also. Also, since $nt(I_j) = n - 1$, by induction, we know that C_j is forward supported eventually. We pick $C = C_j$. So, if some C_j is false in I_j for some j, and C_j does not contain L_j, we are done. Suppose now that for all j, the clause C_j contains L_j. Let C be some clause of S that is false in I. The literals $\neg L_1, \neg L_2, \ldots, \neg L_n$ may appear in C or some subset of them may appear; no other negative literals may appear. After all the clauses C_j have been found to be forward supported, the electrons $L_1, L_2, \ldots, L_n$ will be available, and so another round of hyper-linking with forward support strategy will cause C to be forward supported.

THEOREM 6. *Suppose S is a minimal unsatisfiable set of ground clauses obtained by hyperlinking. Then all clauses in S will be forward supported after some number of rounds of hyper-linking with forward support strategy.*

Proof. From Lemma 5, we know that for each interpretation I we get a forward supported clause C that is false in I. Let T be the set of all such C. Then $T \subseteq S$ and T is unsatisfiable. Since S is minimal unsatisfiable, we have $T = S$. Thus all clauses in S will be forward supported eventually.

THEOREM 7. *Suppose S is an unsatisfiable set of clauses and T is a minimal unsatisfiable set of ground instances of S. Then eventually every D in T will have some more general C in S that is forward supported.*

Proof. Let ϕ be an instance mapping from T to S. We show by induction on $nt(I)$ that if D is false in I then eventually some C will be produced by hyper-linking such that C is forward supported and $C = \phi(D)$.

Suppose $nt(I)$ is 0. The D is positive as before. So C is positive too, and therefore forward supported.

Suppose the theorem is true for $nt(I) < n$. We show it for $nt(I) = n$. Suppose I assigns $L_1, L_2, \ldots, L_n$ TRUE, other propositions FALSE. Let $I_1, I_2, \ldots, I_n$ and D_1, $D_2, \ldots, D_n$ be as before; D_j is false in I_j. If any D_j does not contain L_j then D_j is false in I too and we can let D be D_j. Suppose all D_j contain L_j. Let $C_1, C_2, \ldots, C_n$ be $\phi(D_1), \phi(D_2), \ldots, \phi(D_n)$. Let D be some clause false in I. By induction, some such C_j will eventually be forward supported. Then electrons $L'_1, L'_2, \ldots, L'_n$ will be available, where $L'_j \in C_i$ and $L'_1 = \phi(L_i)$. These electrons can be used to produce a forward supported instances of D by a hyper-link operation. This completes the proof.

COROLLARY 8. *The hyper-linking strategy is complete with forward support strategy.*

Proof. Suppose the set S of clauses is unsatisfiable. From Theorem 7, we know that eventually $\phi(D)$ is forward supported for all D. Thus any hyper-link between the clauses $\phi(D)$ is allowed. Since unrestricted hyper-linking strategy is complete, we will get a proof for S eventually.

COROLLARY 9. *The hyper-linking strategy is complete with backward support strategy.*

Proof. The proof is similar to that for forward support strategy. However, we replace $nt(I)$ by $nf(I)$ which means the number of propositions assigned FALSE by I.

COROLLARY 10. *The hyper-linking strategy is complete with user support strategy if the clauses not contained in the user support set are ebsatisfiable.*

Proof. The proof is similar to that for forward support strategy if we change the signs of atoms in the right way. Forward support is like user support if positive clauses are chosen as the support set. By changing signs of some atoms we can get the effect of user support.

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, we may specify one round to be forward supported and the next round to be backward supported.

5. Implementation

We have implemented the hyper-linking strategy in Prolog. We use backtracking to examine all possibilities of links for all literals in a nucleus C. Each instance is asserted into the database. If some literal in C is a ground literal, or becomes a ground literal due to previous unifications, then no backtracking is needed since linking will not instantiate this literal. Also, for efficiency, ground literals N_i are considered first.

We don't store links as in connection graph resolution. This has advantages and disadvantages. It saves space since usually the number of links is very big. However, identical instances are generated in different rounds and it takes time to generate and delete them.

There is another kind of duplication in the current implementation. If the number of iterations needed to obtain a proof is greater than one, then identical ground clauses may tested by the propositional calculus decision procedure in each iteration. We call this *duplication by iteration*. However, this duplication causes no problem in general since the number of iterations needed is usually small (often 3 or 4).

6. Comparison with Other Theorem Provers

In this section, we compare our hyper-linking strategy prover with some other theorem provers. The other provers considered are DP [6] (the Davis–Putnam Procedure), OTTER [18], 'sprfn' [24, 20, 21], and PTTP [30]. DP is a propositional calculus decision procedure. We coded this procedure in Prolog for comparison purposes. OTTER is a fast resolution-style theorem prover. It is written in *C* and uses discrimination nets for fast look-up of potential unifications. 'sprfn' is based on the modified problem reduction format (MPRF) [24]. It is written in Prolog and uses depth-first iterative deepening search and caching of answers to avoid repeated solution of the same subgoal. PTTP uses the MESON procedure [14] which is a refinement of Model Elimination. It uses depth-first iterative-deepening search. It also uses the same data structures and optimization techniques as for a Prolog implementation, to obtain very fast rates of inference, on the order of thousands or tens of thousands of inferences per second. The version we used for our comparison is a Prolog version that is not as fast as that reported in [30], and does not have all of the optimizations of the faster version.

We use some propositional and near-propositional problems. Propositional problems contain no variables. All clauses contained in them are ground. A near-propositional problem is one in which the function symbols play a minor role. Usually it does not contain many variables and the search space is finite. Since clauses of these problems are mainly propositional, the instance generation strategy has little effect on the performance of the provers. Therefore, the performance of the provers on these problems reveals the effect of duplication of instances during the proof.

The times spent on these problems for different proving systems are listed in Table I. All the times are obtained on a SUN3/60 workstation with 12 MB memory. The Prolog system used for running DP, PTTP, 'sprfn', and our prover is ALS-Prolog (Version 0.60) of Applied Logic System, Inc. [2]. An entry with '–' indicates we stopped the run on the problem in the entry after the time spent is greater than

Table I. Performance comparison on some propositional and near-propositional problems.

problem	no. of input clauses	DP (sec)	OTTER (sec)	PTTP (sec)	sprfn (sec)	our prover (sec)
ph4	22	0.167	63.620	–	88.833	0.600
ph5	45	1.017	38606.760	–	7145.450	1.800
ph6	81	6.183	–	–	–	7.300
ph7	133	40.133	–	–	–	39.600
ph8	204	289.000	–	–	–	276.467
ph9	297	2368.250	–	–	–	2266.617
example	6	NA	142.360	–	9.766	17.183
latinsq	16	NA	–	–	–	56.350
liar	8	NA	0.320	76.483	8.530	5.433
salt	44	NA	1523.820	–	5728.383	28.016
jobs	45	NA	–	–	–	331.000
zebra	128	NA	–	–	–	866.116
lion	51	NA	100.480	–	2391.133	231.400

24 hours. An entry with 'NA' means the prover is not applicable to the problem in the entry. The first six problems,'ph4' through 'ph9' are pigeon hole problems [23]. 'ph4' means 4 objects in 3 holes, etc. They are propositional problems. The rest are near-propositional problems. 'example' is a theorem presented by Pellitier and Rudnicki in AAR Newsletter No. 6, 1986. 'latinsq' is the latin square problem from [27]. 'liar' is the truthtellers and liars problem [15]. 'salt' is the salt and mustard problem from Lewis Carroll [15]. 'jobs' and 'zebra' are logic puzzles from [31] and [29] respectively. 'lion' is the lion and unicorn problem from Raymond Smullyan [22]. Note that DP is only applicable to the propositional problems. The strategies used for OTTER are hyperresolution and UR resolution together, plus forward and backward subsumptions. For those problems that can't be solved by OTTER, we also tried the strategies above and binary resolution, respectively, with reasonable support sets specified, i.e., the clauses corresponding to the negation of the answers in puzzles, without any success. As another comparison, Murray and Rosenthal's dissolver takes 168 seconds for 'ph6' [19].

The hyper-linking strategy prover runs as well as the Davis–Putnam procedure on the propositional problems. This is reasonable since both methods work in the same manner for propositional problems. However, the hyper-linking strategy prover performs much better than the other provers for most of the cases. Since our prover attempts to eliminate duplication of instances of clauses, this indicates that such duplication might be a real problem that is hindering the other proving methods.

The hyper-linking strategy prover also runs well for other problems. For a comparison with other provers on these problems, see [12, 25].

7. Related Work

Some of the early work in theorem proving was similar in spirit to ours, such as the linked conjunct procedure of [6]. Even the methods of [10, 26] have similarities to ours. However, they also have significant differences. In Gilmore's procedure, instances are generated by *blind substitution.* It replaces variables in a clause by elements of the Herbrand universe with level saturation. This results in exponential explosion of the search space. Consider the set S of the following two clauses [4]

- $\{p(X, g(X), Y, h(X, Y), Z, k(X, Y, Z))\}$
- $\{\neg p(U, V, e(V), W, f(V, W), X\}$

The Herbrand universe of level 0 contains 1 element, level 1 contains 6 elements, . . . , and level 5 contains about 10^{65} elements. Gilmore's procedure can't find a proof of S until level 5 in which about 10^{256} instances are generated at level 5. Furthermore, the unsatisfiability test is done using the multiplication method. Clauses are transformed to a disjunctive normal form before the test. This transformation is very inefficient. It results in exponential explosion of the number of conjuncts. It also combines parts of different clauses in a conjunct.

The Prawitz procedure needs duplicate clauses. Each duplicate may have independent instantiations. For example, consider the following three clauses: $\{p(a)\}$, $\{p(b)\}$, and $\{\neg p(X)\}$. Two copies of the third clause are required. One copy is instantiated to $\{\neg p(a)\}$; the other copy is instantiated to $\{\neg p(b)\}$. One problem with the Prawitz procedure is related to the determination of the number of duplicates for an input clause. It also uses the multiplication method; the use of disjunctive normal form inevitably produces the same difficulties as Gilmore's procedure.

The linked conjunct procedure also needs duplicates as the Prawitz procedure. It unifies literals in a clause one at a time, in contrast to our method which unifies all literals in a clause simultaneously. Due to this global unification, the instances generated by our method are fewer and more useful than those generated by the linked conjunct procedure. The linked conjunct procedure may have to consider a large number of cases. Suppose each literal has m possible mates (literals it can complementarily unify with). If there are n literals, then there may be m^n cases to be considered. Even if each case can be done fast, m^n may still be a large number. This might involve a lot of duplication by case analysis since identical clauses may appear in many cases. The linked conjunct procedure also performs the unsatisfiability test more often than our method. Moreover, Ref. [6] (the linked conjunct procedure) 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 as the linked conjunct procedure. 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 [28]. Unlike our method, resolution combines parts of different clauses. This has advantages and disadvantages. The connection graph methods of [3] are close to ours, but even these combine literals from different clauses. Also, we do not store any links between clauses as in the connection graph methods, which rely heavily on such links.

8. Concluding Remarks

Duplication of instances of clauses is very common in theorem proving methods. This duplication can cause combinatorial problems and redundant work. The hyper-linking strategy eliminates duplication by combination and makes duplication by case analysis negligible. The effectiveness of the strategy has been demonstrated by comparing it with other theorem provers. It works better than other provers on propositional and near-propositional problems. It is also good on other problems.

The current implementation of the hyper-linking strategy uses breadth-first search. The search may cause memory-paging problems for some problems. However, the problem doesn't occur on propositional and near-propositional problems we have

tested. Moreover, we have developed a mechanism, called sliding priority [12], to reduce the memory-paging problem.

The efficiency of the implementation can be improved. First we may require that the electrons come from different clauses than the nucleus itself. This may reduce the number of hyper-links and the number of instances generated. Secondly, we may require that one of the electrons for a hyper-link be generated in the last hyper-linking round. This avoids the generation and deletion of duplicate instances occurring in different rounds since we don't store links. However, it may be complicated if alternative support sets are used.

The hyper-linking strategy prover can work as a problem solver, i.e. finding answers for a set of given constraints. This is done by the capability of model finding of our propositional calculus decision procedure. The propositional calculus decision procedure prints out a model if the input set of clauses is satisfiable. For example, the hyper-linking strategy prover finds a solution for the salt and mustard problem in 53.650 seconds. If the problem has multiple solutions, the hyper-linking strategy prover can find all of them one by one. This method is explained in [13].

References

1. Andrews, P. B., 'Theorem proving via general matings', *Journal of the Association for Computing Machinery* **28**, 193–214 (1981).
2. Applied Logic Systems, Inc., *ALS Prolog User's Guide and Reference Manual*, Syracuse, New York, March 1988.
3. Bibel, W., *Automated Theorem Proving*, Vieweg (1982).
4. Chang, C. and Lee, R., *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York (1973).
5. Cook, S.A., 'The complexity of theorem-proving procedures', *In Proceedings of the 3rd Annual ACM Symposium on Theory of Computing* (1971) pp. 151–158.
6. Davis, M., 'Eliminating the irrelevant from mechanical proofs', In *Proceedings Symp. of Applied Math*, (1963) Vol. 15, pp. 15–30.
7. Davis, M. and Putnam, H., 'A computing procedure for quantification theory', *Journal of the Association for Computing Machinery* 7, 201–215 (1960).
8. de Kleer, J., 'An assumption-based TMS', *Artificial Intelligence* **28**, 127–162 (1986).
9. de Kleer, J., 'A comparison of ATMS and CSP techniques', In *Proceedings of 11th International Joint Conference on Artificial Intelligence*, (1989) pp. 290–296.
10. Gilmore, P.C., 'A proof method for quantification theory', *IBM Journal of Research and Development* **4**, 28–35 (1960)
11. Kowalski, R., 'A proof procedure using connection graphs', *Journal of the Association for Computing Machinery* **22**, 572–595 (1975).
12. Lee, S.-J., *CLIN. An Automated Reasoning System Using Clause Linking*. PhD thesis, University of North Carolina at Chapel Hill, 1990.
13. Lee, S.-J. and Plaisted, D., 'New applications of a fast propositional calculus decision procedure', In *Proceedings of the 8th Biennial Conference of Canadian Society for Computational Studies of Intelligence*, pp. 204–211 (1990).
14. D. Loveland, *Automated Theorem Proving: A Logical Basis*, North-Holland, New York (1978).
15. Lusk, E. and Overbeek, R., 'Non-Horn problems', *Journal of Automated Reasoning* **1**, 103–114 (1985).
16. McAllester, D., 'An outlook on truth maintenance', *Technical Report AIM-551*, Artificial Intelligence Laboratory, MIT, Cambridge, MA (1980).
17. McAllester, D., 'Reasoning utility package user's manual'. *Technical Report AIM-667*, Artificial Intelligence Laboratory, MIT, Cambridge, MA (1982).

18. McCune, W.W., *Otter 1.0 Users' Guide.* Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois (1989).
19. Murray, N. and Rosenthal, E., 'DISSOLVER: A dissolution-based theorem prover, system abstract, In *Proceedings of the 10th International Conference on Automated Deduction*, pp. 665–666 (1990).
20. Nie, X. *Implementation Techniques in Automatic Theorem Proving.* PhD thesis, University of North Carolina, at Chapel Hill, 1989.
21 Nie, X and Plaisted, D., 'A complete semantic back chaining proof system', In *Proceedings of the 10th International Conference on Automated Deduction* (1990).
22. Ohlhach, H.J. and Schmidt-Schauss, M., 'The lion and the unicorn', *Journal of Automated Reasoning* **1**, 327–332 (1985).
23. Pelletier, F.J., 'Seventy-five problems for testing automatic theorem provers', *Journal of Automated Reasoning* **2**, 191–216 (1986).
24. Plaisted, D., 'Non-Horn clause logic programming without contrapositives', *Journal of Automated Reasoning* **4**, 287–325 (1988).
25. Plaisted, D. and Lee, S.-J., 'Inference by clause linking', *Technical Report TR90-022*, University of North Carolina - Chapel Hill, Chapel Hill, NC (1990).
26. Prawitz, D., Prawitz, H. and Voghera, N., 'A mechanical proof procedure and its realization in an electronic computer', *Journal of the Association for Computing Machinery* **7**, 102–128 (1960).
27. Robinson, J., 'Theorem proving on the computer', *Journal of the Association for Computing Machinery* **10**, 163–174 (1963).
28. Robinson, J., 'A machine-oriented logic based on the resolution principle', *Journal of the Association for Computing Machinery*, **12** 23–41 (1965).
29 Sterling, L. and Shapiro, E., *The Art of Prolog*, MIT Press, Cambridge, Massachusetts (1986).
30. Stickel, M.E., 'A Prolog technology theorem prover: Implementation by an extended prolog compiler', *Journal of Automated Reasoning*, **4**, 353–380 (1988).
31. Wos, L., Overbeck, R., Lusk, E. and Boyle, J., *Automated Reasoning: Introduction and Applications*, Prentice Hall, Englewood Cliffs, N.J. (1984).
32 Wos, L., Robinson, G. and Carson, D., 'Efficiency and completeness of the set of support strategy in theorem proving', *Journal of the Association for Computing Machinery* **12**, 536–541 (1965).