Improving the Efficiency of a Hyperlinking-Based Theorem Prover by Incremental Evaluation with Network Structures*

SHIE-JUE LEE AND CHIH-HUNG WU
Department of Electrical Engineering, National Sun Yat-Sen University, Kaohsiung, Taiwan 80424, ROC
e-mail: {leesfj,johnwu}@ee.nsysu.edu.tw

(Received: 6 December 1993)

Abstract. Recently Lee and Plaisted proposed a theorem-proving method, the hyperlinking proof procedure, to eliminate duplication of instances of clauses during the process of inference. A theorem prover, CLIN, which implements the procedure was also constructed. In this implementation, redundant work on literal unification checking, partial unification checking, and duplicate instance checking is performed repetitively, resulting in a large overhead when many rounds of hyperlinking are needed for an input problem. We propose a technique that maintains information across rounds in shared network structures, so that the redundant work in each hyperlinking round can be avoided. Empirical performance comparison has been done between CLIN and CLIN-net, which is the theorem prover with shared network structures, and some results are shown. Problems related to memory overhead and literal ordering are discussed.

Key words: first-order logic, theorem proving, hyperlinks, unification, clause network.

1. Introduction

The hyperlinking proof procedure proposed in [13] is a refutational clause-form theorem-proving method for first-order logic. It eliminates duplication of instances of clauses during the process of inference. A theorem prover called CLIN, which implements the procedure, was also constructed. CLIN has been shown to be competitive with some major contemporary theorem-proving systems on a spectrum of typical test problems. However, redundant literal unification checking, partial unification checking, and duplicate instance checking are performed repetitively in CLIN, resulting in a large overhead when many rounds of hyperlinking are needed for an input problem. We propose a technique that maintains information across rounds by constructing shared network structures, called the clause network, to keep information across rounds, so that redundant unification checking and duplicate instance checking in each hyperlinking round can be avoided. The redundant work is reduced significantly when the required number of hyperlinking rounds is large.

The hyperlinking proof procedure can be modified to find solutions for a given problem description. Before a solution is derived, a number of hyperlinking rounds are needed to generate exhaustively all hyperinstances. Redundant unification checking and duplicate instance checking occur badly in the CLIN-based problem solver. Since no new hyperinstances are generated in the last round of hyperlinking, most literal unification

checks, partial unification checks, and duplicate instance checks performed in this round are redundant.

We start with a brief introduction to the hyperlinking proof procedure. Then we show where the inefficiency of CLIN comes from. Next, the network structures used for dealing with this inefficiency are described. Then empirical results are given to show that our approach is effective in both proving theorems and finding solutions. Some problems related to memory overhead and literal ordering are then discussed.

2. The Hyperlinking Proof Procedure

The hyperlinking proof procedure consists of two phases: hyperlinking and propositional unsatisfiability test. Suppose \( S \) is a set of clauses. Hyperlinking performs unifications between literals of clauses in \( S \) to instantiate these clauses little by little. If \( C = L_1 \lor \cdots \lor L_m \) is a clause in \( S \), and \( M_1, \ldots, M_m \) are literals from clauses in \( S \), with variables renamed to avoid conflicts, and \( \Theta \) is a most general unifier such that \( L_i \Theta \) and \( M_i \Theta \) are complementary for all \( i, 1 \leq i \leq m \), then we call \( C \Theta \) a hyperinstance of \( C \). We call each \( (L_i, M_i) \), \( 1 \leq i \leq m \), a link. The set \( \{(L_1, M_1), \ldots, (L_m, M_m)\} \) is called a hyperlink, and \( \Theta \) is called the substitution of the hyperlink. We also call \( C \) the nucleus and the literals \( M_i \) electrons.

If \( R \) is a set of clauses, let \( H(R) \) be the set of hyperinstances of clauses in \( R \). Let \( S_0 \) be the set \( S \) of input clauses. Let \( S_1 = S_0 \cup H(S_0) \). In general, let \( S_i = S_{i-1} \cup H(S_{i-1}) \). Hyperlinking computes this sequence \( S_0, S_1, S_2, \ldots \) of sets of clauses. If \( R \) is a set of clauses, let \( Gr(R) \) be \( R \) with all variables replaced by the same constant symbol \( \$ \). A propositional unsatisfiability test similar to the David-Putnam procedure [6] is applied to the sets \( Gr(S_i) \) of ground clauses. If the propositional test returns an answer of 'unsatisfiable' for some such \( Gr(S_i) \), then we know that the original \( S \) is unsatisfiable. The procedure can be described as follows.

\[
\text{procedure Hyperlinking}
\]

\[
R \leftarrow S;
\]

\[
\text{while } Gr(R) \text{ is propositionally satisfiable do}
\]

\[
\text{compute } H(R);
\]

\[
\text{if } H(R) = \emptyset/\ast \text{ if no new hyperinstance was generated }/\ast
\]

\[
\text{then return and print 'S is satisfiable'}
\]

\[
\text{else } R \leftarrow R \cup H(R)
\]

\[
\text{endwhile}
\]

\[
\text{return and print 'S is unsatisfiable'}.
\]

Each iteration in the above description is called a round of hyperlinking. The hyperlinking procedure is complete. That is, if \( S \) is unsatisfiable, then the procedure guarantees to stop and return 'unsatisfiable' [13].

Hyperlinking is compatible with the set of support strategy. Suppose we are given an input set \( S \) of clauses. A support set \( T \) for \( S \) is \( T \subseteq S \) such that \( S - T \) is satisfiable. Three support sets are defined. 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 \( S \), and a user support set is