Semantic Selection of Premisses for Automated Theorem Proving

Petr Pudlák¹ ¹Charles University, Prague petr.pudlak@mff.cuni.cz

Abstract

We develop and implement a novel algorithm for discovering the optimal sets of premisses for proving and disproving conjectures in first-order logic. The algorithm uses interpretations to semantically analyze the conjectures and the set of premisses of the given theory to find the optimal subsets of the premisses. For each given conjecture the algorithm repeatedly constructs interpretations using an automated model finder, uses the interpretations to compute the optimal subset of premisses (based on the knowledge it has at the point) and tries to prove the conjecture using an automated theorem prover.

1 Importance of selecting appropriate premisses in automated theorem proving

A proper set of premisses¹ can be essential for proving a conjecture by an automated theorem prover. Clearly, the larger the number of the initial premisses the larger the number of the inferred formulae. And as for the most proving techniques the number of inferred formulae is in general super-exponential in the number of input formulae, the impact on the performance of an automated theorem prover is quite significant. Removing even a single superfluous premiss can make the difference between being or not being able to prove the conjecture.

This major problem is even more serious when reasoning within large mathematical theories, which can contain hundreds of premisses and thousands of conjectures. In such cases, selecting proper premisses for proving the conjectures becomes a necessity.

In our previous work [Pud06a, Pud06b] we described a method for compacting proofs of conjectures. By constructing lemmas from the proofs and by syntactically analyzing both the lemmas and the proofs, we constructed sets of premisses that produced shorter proofs of given conjectures, or allowed to construct proofs much faster. However, the assumption was that for each conjecture we already know some set of premisses from which we were able to prove the conjecture (although possibly redundant and inefficient for constructing a proof). If we were not able to prove the conjecture at all, the situation was more complicated.

¹Using the notion *axioms* for the formulae the prover uses as assumptions sometimes causes confusion, therefore we shall prefer the notion *premisses* instead. See section 2.1.

One possibility is to syntactically analyze the formulae and/or use an AI algorithm for guessing the proper premisses. Successful examples of such procedures are Josef Urban's tools for the Mizar Project [Urb06a, Urb06b], reducing axiom sets in software verification [RS98] or filtering of axioms for machine-generated problems [MP06]. Although even simple syntactic heuristics can be very effective, the syntactic approach is generally restricted to the cases when the (syntactic) structure of formulae well reflects their semantics. Clearly, this is not always the case. Moreover, the syntactic analysis is usually only a heuristic procedure that tries to learn what premisses could be needed but is likely to fail on a new kind of problem. Syntactic filters are also often incomplete in the sense that they can eventually remove too many premisses.

The procedure we shall describe in this article uses *semantic* analysis. By observing which formulae are true in which interpretations, we can get a deeper insight into the nature of a conjecture and compute such a set of premisses that is proper for proving the conjecture. As far as we are aware, this is a novel approach, which has not been researched before.

We shall focus on two interconnected goals:

- Determine which sets of premisses are sufficient for proving the conjecture;
- among those sets, choose such a set that contains no redundant premisses and that is minimal with respect to some criterion.

The criterion can be just a simple one – to minimize the number of premisses, or a more complex one, for example, to avoid premisses of a certain kind that complicate the proving process.

2 Semantic analysis of problem using interpretations

2.1 Notation

Throughout the whole article we shall always work within the domain of first-order logic. We shall always assume that a language is given and all formulae we work with are formulated in the language and are closed.

We shall denote formulae by regular letters (e.g. A, F), interpretations by calligraphic letters (e.g. \mathcal{M}), sets of formulae by boldface letters (e.g. \mathbf{B} , \mathbf{A}_C) and sets of sets or sets of interpretations by script letters (e.g. \mathscr{F} , \mathscr{N}). We shall also use the following symbols:

$\mathcal{M} \models B$	\mathcal{M} is a model of a formula B	(1)
$\mathcal{M}\models\mathbf{B}$	\mathcal{M} is a model of all formulae in \mathbf{B}	
	(i.e. a model of the conjunction of all formulae in ${\bf B})$	(2)
$\mathbf{B}\models G$	for all possible interpretations \mathcal{M} , if $\mathcal{M} \models \mathbf{B}$ then $\mathcal{M} \models G$	
	(i.e. G is a consequence of \mathbf{B})	(3)
$\mathbf{B}\vdash G$	G can be proved from \mathbf{B}	(4)

For brevity we shall often say that "a formula B avoids an interpretation \mathcal{M} " (resp. B avoids a set of interpretations \mathscr{M}) if and only if $\mathcal{M} \not\models B$ (resp. $\mathcal{M} \not\models B$ for all $\mathcal{M} \in \mathscr{M}$).

Often formulae given to the prover as a basis for (dis)proving conjectures are simply called *axioms*. However, this sometimes causes confusion, because these given formulae actually do not have to be the axioms of a particular theory. They can be also propositions that have already been proved before, lemmas, etc. Therefore, we shall instead call such assumptions *premisses*.

2.2 Basic principle

Let a conjecture C and a set of premisses \mathbf{A} (finite, but presumably large) be given. We would like to prove (or to refute) C from \mathbf{A} using an automated theorem prover. But since \mathbf{A} is large, the prover is overloaded by the high number of premisses and we are not able to prove C directly from \mathbf{A} . We expect that only a small subset of \mathbf{A} is necessary to prove or to refute C and we would like to find such a subset.

The idea of the semantic analysis is based on two principal properties of first-order logic:

- 1. If $\mathbf{B} \vdash C$ then for any interpretation \mathcal{M} such that $\mathcal{M} \not\models C$ also $\mathcal{M} \not\models \mathbf{B}$. That is, there is $B \in \mathbf{B}$ such that $\mathcal{M} \not\models B$.
- 2. Having a set of premisses **B**, then
 - either C is provable from **B**, hence $\mathbf{B} \vdash C$, which is equivalent to $\mathbf{B} \models C$ by the completeness theorem,
 - or there is an interpretation \mathcal{M} such that $\mathcal{M} \models \mathbf{B} \cup \{\neg C\}$.

Using these two properties we shall construct a sequence of interpretations \mathcal{M}_i and a sequence of premisses F_i (for convenience let us set $\mathbf{B}_i = \{F_0, \ldots, F_{i-1}\}$). At each step we select F_i to be from the set $\mathbf{A}_i = \{F \in \mathbf{A} \mid \mathcal{M}_i \not\models F\}$ and eventually some set \mathbf{B}_k will be sufficient for proving C.

Let us start with an empty set of premisses $\mathbf{B}_0 = \emptyset$. If *C* is a tautology, then $\mathbf{B}_0 \vdash C$ and we are finished. Otherwise, there is some interpretation \mathcal{M}_0 such that $\mathcal{M}_0 \models \neg C$ (see Fig. 1a). Let

$$\mathbf{A}_0 = \{ F \in \mathbf{A} \, | \, \mathcal{M}_0 \not\models F \}$$

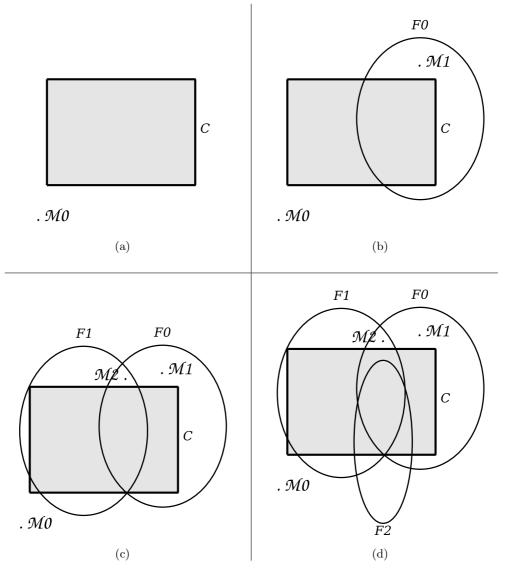
Let us select any $F_0 \in \mathbf{A}_0$ and set $\mathbf{B}_1 = \{F_0\}$. Because $\mathcal{M}_0 \not\models F_0$, also $\mathcal{M}_0 \not\models \mathbf{B}_1$. Now, either $\mathbf{B}_1 \models C$, thus $\mathbf{B}_1 \vdash C$, or there is an interpretation \mathcal{M}_1 such that $\mathcal{M}_1 \models \mathbf{B}_1 \cup \{\neg C\}$ (see Fig. 1b). We set

$$\mathbf{A}_1 = \{ F \in \mathbf{A} \, | \, \mathcal{M}_1 \not\models F \}$$

and continue in similar fashion. In step i, either already $\mathbf{B}_i \models C$ (as in Fig. 1d, where i = 3) or there is an interpretation \mathcal{M}_i such that $\mathcal{M}_i \models \mathbf{B}_i \cup \{\neg C\}$ (as in Fig. 1c, where i = 2). In such a case we set

$$\mathbf{A}_i = \{ F \in \mathbf{A} \, | \, \mathcal{M}_i \not\models F \}$$

If \mathbf{A}_i is empty, then $\mathcal{M}_i \models \mathbf{A} \cup \{\neg C\}$, hence C is not a theorem of \mathbf{A} . Otherwise, we set $\mathbf{B}_i = \mathbf{B}_{i-1} \cup \{F_{i-1}\}$, where F_i is an arbitrary formula chosen from \mathbf{A}_i .



Each point of the plane of the diagram represents a single possible interpretation of the language. A formula is represented as a shape that contains precisely the interpretations in which the formula is true. The conjecture Cis illustrated by the gray rectangle. First, we construct a model \mathcal{M}_0 of $\neg C$ (a). We pick some premiss F_0 that avoids the model. Next, we construct a model \mathcal{M}_1 of $\{F_0, \neg C\}$ (b) and pick some other premiss F_1 which avoids this model. Then, because we still cannot prove C from $\{F_0, F_1\}$, we construct a model \mathcal{M}_2 of $\{F_0, F_1, \neg C\}$ (c) and pick yet another premiss F_2 that avoids it. Now, there is no model of $\{F_0, F_1, F_2, \neg C\}$ (d) and hence we can prove C from $\{F_0, F_1, F_2\}$.

Figure 1: Illustration of the semantic selection of premisses.

Observe that at each step \mathbf{B}_i avoids all constructed interpretations $\mathcal{M}_1, \ldots, \mathcal{M}_{i-1}$. Also note that $\mathbf{B}_i \cap \mathbf{A}_i = \emptyset$, since $\mathcal{M}_i \models \mathbf{B}_i$, but for each $F \in \mathbf{A}_i$ we know that $\mathcal{M}_i \not\models F$ by the definition of \mathbf{A}_i . Therefore, at each step, a different formula is moved from \mathbf{A}_i into \mathbf{B}_i (and possibly other formulae are removed from \mathbf{A}_i) and $|\mathbf{B}_i| = i$. And since \mathbf{A} is finite, either $\mathbf{B}_k \models C$ for some k, or $\mathbf{A}_l = \emptyset$ for some l and thus $\mathbf{A} \not\models C$.

2.3 Adaptation for automated theorem provers

Automated theorem provers, model finders and other similar tools are always limited in available resources such as time or computer memory. Hence, if we are given a set of premisses **B** and a conjecture C, the result of a computation could be that we either

- find a proof of C from **B**, or
- find an interpretation \mathcal{M} such that $\mathcal{M} \models \mathbf{B} \cup \{\neg C\}$, or
- run out of resources.

We have to take into account also the last possible outcome and modify the procedure described in the previous section to be able to deal with such a result.

The simplest possible solution is that if at step n we are neither able to prove C from \mathbf{B}_n nor to find an interpretation \mathcal{M}_n such that $\mathcal{M}_n \models \mathbf{B}_n \cup \{C\}$, we backtrack and at some previous step k < n we try to pick a different formula F_i from \mathbf{A}_k .

Nevertheless, we shall now describe a more general solution that allows us to proceed in those cases when we are neither able to prove nor disprove C. The procedure remembers all failed attempts (sets of premisses) in a system of sets \mathscr{F} to avoid trying the same attempt again and thereby getting into an infinite cycle. All interpretations that are constructed during the computation are stored in a set \mathscr{M} and are then reused for selecting the premisses.

Let us give a schema of the procedure:

- 1. Initialize $\mathscr{F} = \emptyset$ and $\mathscr{M} = \emptyset$.
- 2. Repeat:
 - (a) Construct a subset of premisses $\mathbf{B} \subseteq \mathbf{A}$ such that $\mathbf{B} \notin \mathscr{F}$ and such that for each $\mathcal{M} \in \mathscr{M} : \mathcal{M} \not\models \mathbf{B}$. If no such \mathbf{B} can be found, report failure and exit.
 - (b) Try to prove C from **B**. If successful, print the proof and exit.
 - (c) Otherwise, try to construct an interpretation M such that M ⊨ B ∪ {¬C}. In our case, this step is performed by an automated model finder like Paradox or Darwin.
 - (d) If such a model \mathcal{M} is found, then
 - i. if $\mathcal{M} \models \mathbf{A}$ then report that $\mathbf{A} \not\models C$ and exit;
 - ii. otherwise add \mathcal{M} to \mathscr{M} , and
 - iii. loop to 2.
 - (e) If no such model is found, add **B** to \mathscr{F} and loop to 2.

We only need to remember the failed attempts, because if we successfully construct a model $\mathcal{M} \models \mathbf{B}$, the condition $\mathcal{M} \not\models \mathbf{B}$ in 2a prevents choosing **B** again.

The procedure described in section 2.2 was a special case of this generalized procedure. As there were no failed attempts, the system \mathscr{F} remained empty and in *i*-th cycle the set **B** constructed in 2a was the set **B**_{*i*}.

As mentioned in step 2c, we use an external tool to construct the interpretations. There is no restriction on the constructed interpretations except that we are able to decide whether a formula is true in it or not. However, as far as we are aware, all current model finders are limited to constructing only finite interpretations.

The important part, which we have omitted until now, is the construction of the set of premisses \mathbf{B} in 2a. We shall examine this problem in the next section and give a specification of a complete algorithm.

3 Specification of the algorithm

3.1 Converting the task to the weighted set cover problem

In the previous section we have outlined a schema of an algorithm for selecting premisses for proving a conjecture. In this section we shall construct a complete, generalized algorithm that tries to find a minimal set of premisses with respect to a given criterion. The criterion will be represented by a weight function on premisses and our goal will be to find a set of premisses for which the sum of the weights is minimized.

Definition 1 (Weight function). Let \mathbf{A} be a set of premisses. A weight function β is a function that maps formulae from \mathbf{A} into positive real numbers:

$$\beta: \mathbf{A} \to \mathbb{R}^+$$

We shall now focus on the construction of the optimal set of premisses in step 2a of the procedure given in section 2.3. Our aim is to find such a subset of premisses $\mathbf{B} \subseteq \mathbf{A}$ that avoids all known interpretations of $\neg C$ and for which the sum $\sum_{B \in \mathbf{B}} \beta(B)$ is minimized.

For each premiss $B \in \mathbf{A}$ let \mathscr{C}_B be the set of interpretations that B avoids:

$$\mathscr{C}_B = \{ \mathcal{M} \in \mathscr{M} \mid \mathcal{M} \not\models B \}$$

$$\tag{5}$$

Thus, **B** avoids all the interpretations in \mathscr{M} if and only if $\bigcup_{B \in \mathbf{B}} \mathscr{C}_B = \mathscr{M}$. By this assignment we have converted our problem to the well-known weighted set cover problem:

Definition 2 (The weighted set cover problem). For the input

- ground elements $\mathcal{M} = \mathcal{M}_1, \ldots, \mathcal{M}_n$
- subsets $\mathscr{C}_B \subseteq \mathscr{M}$ where $B \in \mathbf{A}$
- weights $\beta(B)$ defined for $B \in \mathbf{A}$

find a set $\mathbf{B} \subseteq \mathbf{A}$ that minimizes $\sum_{B \in \mathbf{B}} \beta(B)$ such that $\bigcup_{B \in \mathbf{B}} \mathscr{C}_B = \mathscr{M}$. For the unweighted set cover problem we take $\beta(B) = 1$ for all Bs.

In our case, the elements being covered are the interpretations we have constructed so far and the covering sets are defined by (5).

There are many theoretical results regarding this problem in literature. For us the most important results concern the greedy algorithm, which approximates the problem:

Definition 3 (Greedy algorithm for the weighted set cover problem). The greedy algorithm for set cover selects sets according to the rule:

At each stage, select the set that minimizes the cost (with respect to the weight function β) per additional element covered.

More formally, if \mathcal{N} is the set of uncovered ground elements, select B for which $\frac{\beta(B)}{|\mathscr{C}_B \cap \mathcal{N}|}$ is minimal.

For the unweighted set cover problem this means that we select the set for which $|\mathscr{C}_B \cap \mathscr{N}|$ is maximal, that is the set which contains the largest number of uncovered elements.

By the following results, the algorithm gives the best possible polynomial approximation of the weighted set cover problem:

Remark 1 (Properties of the greedy algorithm). The greedy algorithm for the weighted set cover problem has the following properties:

- The weighted set cover problem is NP-complete (see [Kar72]).
- The approximation ratio² of the greedy algorithm is $H(\max_{B \in \mathbf{B}} |\mathscr{C}_B|)$, where $H(n) = \sum_{k=1}^{n} \frac{1}{k} \leq \ln(n) + 1$ (see [Chv79]).
- This is essentially the best possible approximation (see [Fei98]).

3.2 Searching for sets of premisses using the greedy set cover algorithm

The greedy algorithm described in the previous section will find an approximation of the optimal subset of premisses. If we are able to prove C from **B**, we are finished, and if we are able to find a model of $\mathbf{B} \cup \{\neg C\}$ we use the greedy algorithm to find a new optimal subset. The problem arises if we are neither able to prove C or find a new interpretation. Therefore, we slightly modify the greedy algorithm so that we are able to backtrack and find another next-to-optimal set of premisses that covers all the interpretations. Hereby we are able to construct a sequence of sets of premisses until we prove C or find a new interpretation.

The work of the modified algorithm can be viewed as traversing a search tree, where each node of the tree corresponds to a subset of premisses selected from \mathbf{A} . The original

 $^{^2\}mathrm{An}$ algorithm Γ is an $\alpha\text{-approximation}$ algorithm for a minimization problem Π if

 $^{-\}Gamma$ runs in polynomial time.

[–] Γ always produces a solution which is within a factor of α of the value of the solution of the optimal algorithm Π .

The least α that satisfies these conditions is called *the approximation ratio of* Γ .

greedy algorithm traverses only a single branch of the search tree until it finds a covering set. The modified algorithm remembers its position in the search tree and if it is neither able to prove the conjecture from the premisses nor to find a new interpretation, the algorithm continues traversing the tree according to the greedy rule. If it exhausts the whole tree, it exits with a failure.

Remark 2. The theoretical results described in the previous section cover only the original greedy algorithm. The question whether the subsequent covering sets returned by this extended algorithm also well approximate the optimum is still to be researched. However, we believe that it a reasonable assumption.

The algorithm in pseudo-code. Let us now look at the schema of the algorithm. The main function searchForProofOrModel recursively searches the sub-tree that corresponds to a given set of premisses **B**. It sorts the remaining admissible premisses B_1, \ldots, B_k by their cost per additional covered element (with respect to the weight function β) and recursively processes the sets $\mathbf{B} \cup \{B_i\}$ until either a model is found or the whole sub-tree is exhausted (the original greedy algorithm always picks $\mathbf{B} \cup \{B_1\}$). The function returns either a new model, or a **NIL** value if it exhausts all nodes of the sub-tree.

The traversed nodes are remembered using a global set variable \mathscr{F} . It is constructed so that the set $\{Y \subseteq \mathbf{A} \mid \text{there is } X \in \mathscr{F} \text{ such that } X \subseteq Y\}$ is the set of all visited nodes of the tree. Every time we visit a whole sub-tree, we replace all the visited sets of premisses occurring in the sub-tree by the set of premisses corresponding to the root of the sub-tree (line 40). This allows us to keep \mathscr{F} small and thus to efficiently recognize the sets of premisses we have already inspected.

- ² try to prove C from \emptyset ; if successful, **print** \models C and **exit**;
- 3 try to find a model $\mathcal{M} \models \neg C$;
- 4 if successful, set $\mathcal{M} := \{\mathcal{M}\};$
- 5 otherwise **exit** "Failed to prove nor disprove C from **A**";
- 6 /* now we have at least one element to cover */
- 7 label ADD_MODEL:
- s for $B \in \mathbf{A}$ do
- 9 $\mathscr{C}_B := \{ \mathcal{M} \in \mathscr{M} : \mathcal{M} \not\models B \}; /* initialize the covering sets \mathscr{C}_B * / \}$
- 10 $\mathscr{F} := \emptyset$; /* initialize the set of failed attempts */
- 11 $\mathcal{M} := \operatorname{searchForProofOrModel}(\emptyset); /* search for a new model */$
- 12 if $\mathcal{M} \neq \mathbf{NIL then} /*$ if a new model is found, search again */
- 13 if $\mathcal{M} \models \mathbf{A}$ then
- 14 **exit** "C not provable from **A**"
- 15 /* invariant: $\mathcal{M} \notin \mathcal{M}$, see Corollary 1 that follows */
- 16 $\mathcal{M} := \mathcal{M} \cup \{\mathcal{M}\};$
- 17 goto ADD_MODEL;
- 18 /* otherwise exit with failure */
- 19 exit "Failed to prove nor disprove C from A";
- ²⁰ function searchForProofOrModel(set of premisses **B**)
- returns new model or NIL
- 22 begin
- 23 $\mathcal{N} := \mathcal{M} \setminus \bigcup_{B \in \mathbf{B}} \mathscr{C}_B; /* \text{ the set of uncovered models }*/$

^{1 /*} first, try to find any model of $\neg C$ */

```
let B_1, \ldots, B_k be the premisses from A for which (\mathscr{C}_{B_i} \cap \mathscr{N}) \neq \emptyset
24
                 sorted by \frac{\beta(B)}{|\mathscr{C}_{B_i} \cap \mathscr{N}|} in ascending order;
25
         for i := 1 to k do
26
         begin
27
            \mathbf{B}' := \mathbf{B} \cup \{B_i\};
28
            if \neg((\exists X \in \mathscr{F}) : X \subseteq B') then
29
                \mathcal{N}' := \mathcal{N} \setminus \mathscr{C}_{B_i};
30
                if \mathcal{N}' = \emptyset then
31
                   try to prove C from B'; if successful, print B' \models C and exit;
32
                   try to find a model \mathcal{M} \models \mathbf{B}' \cup \{\neg C\}; if successful, return \mathcal{M};
33
                else /* \mathcal{N}' \neq \emptyset, i.e. there are still uncovered models */
34
                   \mathcal{M} := \text{searchForProofOrModel}(\mathbf{B}'); /* recursively try to find a model */
35
                   if \mathcal{M} \neq \mathbf{NIL} then
36
                      return \mathcal{M};
37
                /* whether \mathcal{N}' = \emptyset or not, remember the visited sub-tree: */
38
                /* replace all descendants of \mathbf{B}' by \mathbf{B}' * /
39
                \mathscr{F} := \{\mathbf{B}'\} \cup (\mathscr{F} \setminus \{\mathbf{X} \in \mathscr{F} \mid \mathbf{B}' \subseteq \mathbf{X}\});
40
         end;
41
          /* exhausted all possibilities, neither proof nor new model was found */
42
         return NIL;
43
      end;
44
```

3.2.1 Constructed interpretations and termination of the algorithm

Lemma 1. Let \mathcal{M}_1 and \mathcal{M}_2 be two interpretations constructed by the algorithm, \mathcal{M}_2 after \mathcal{M}_1 . Let \mathbf{B}_1 and \mathbf{B}_2 be their corresponding sets of premisses \mathbf{B}' from which the interpretations were constructed at line 33 ($\mathcal{M}_i \models \mathbf{B}_i \cup \{\neg C\}$). Then $\mathbf{B}_1 \neq \mathbf{B}_2$ and there is a formula F such that $\mathcal{M}_2 \models F$ but $\mathcal{M}_1 \not\models F$.

Proof. \mathbf{B}_2 is selected so that for every $\mathcal{M} \in \mathscr{M}$ at that point $\mathcal{M} \not\models \mathbf{B}_2$. In particular $\mathcal{M}_1 \not\models \mathbf{B}_2$, hence there is $F \in \mathbf{B}_2$ such that $\mathcal{M}_1 \not\models F$. But $\mathcal{M}_2 \models F$ (because $\mathcal{M}_2 \models \mathbf{B}_2$), hence F has the desired properties. And because $\mathcal{M}_1 \models \mathbf{B}_1, F \notin \mathbf{B}_1$, thus $\mathbf{B}_1 \neq \mathbf{B}_2$.

Corollary 1. No two constructed interpretations are isomorphic.

Theorem 1. The algorithm terminates.

Proof. By lemma 1, every time the program constructs an interpretation it uses a different subset of premisses. There only $2^{|\mathbf{A}|}$ possible subsets of \mathbf{A} , hence the algorithm can construct at most $2^{|\mathbf{A}|}$ interpretations and must eventually terminate.

4 Implementation

We have designed an implementation of the algorithm that is aimed to prove or disprove several conjectures at once. The program is written in the JAVA programming language and is connected to E prover ver. 0.99 [Sch02, Sch07] and two model finders: Darwin ver. 1.3FM [FBT07, BFdNT06] and Paradox ver. 2.0 [CS07, CS03]. When constructing models the program runs both these model finders simultaneously and takes the result

of the first one that completes. As the model finders take quite different approaches, this arrangement leads to a better success rate, and on multi-processor machines also to shorter times.

4.1 Input of the program

The input of the program is a list of $TPTP^3$ files. The formulae in each file can be divided into these categories:

- **Conjecture (denoted by** *C***).** The formula which we want to prove or disprove from the premisses given in the same file. (TPTP name conjecture.)
- **Premisses (denoted by A_C).** The set of formulae from which the algorithm should select the optimal set of premisses for (dis)proving the conjecture C. (TPTP name axiom.)
- **Definitions (denoted by \mathbf{D}_{C}).** The formulae which are always included as premisses for proving C. In some cases the user may surmise that some formulae are indispensable for the proof, for example definitions of essential functional or predicate symbols. (TPTP name definition.)

4.2 Execution of the program

Recall that we denote by \mathbf{C} the set of all conjectures the program is asked to prove, by \mathbf{A}_C the input set of premisses for proving each conjecture $C \in \mathbf{C}$, by \mathbf{D}_C the definitions given for proving C and finally by \mathscr{M} the set of constructed interpretations.

At the beginning $\mathcal{M} = \emptyset$. The program runs in a loop until either all conjectures are (dis)proved or until it runs out of time. During each pass of the loop the program performs the following step:

- 1. Decide which conjecture C to try to (dis)prove at this step (we shall describe this in more detail later).
- 2. Use the greedy algorithm to select the optimal set of premisses $\mathbf{B} \subseteq \mathbf{A}_C$ that covers all interpretations from $\{\mathcal{M} \in \mathcal{M} : \mathcal{M} \models \neg C\}$.
- 3. Run the model finder and the prover on $\mathbf{B} \cup \{\neg C\}$.
 - If a model \mathcal{M} is found, set $\mathcal{M} := \mathcal{M} \cup \{\mathcal{M}\};$
 - or if $\mathbf{B} \models C$, report the achievement and remove C from further processing $(\mathbf{C} := \mathbf{C} \setminus \{C\});$
 - or if neither a model is found nor C proved (we shall call this outcome *a* failed attempt), save the state of the greedy algorithm to be able to restore it next time C is selected and to construct another possible covering set.
- 4. In all cases, loop again to 1.

³Thousands of Problems for Theorem Provers, see [SS98].

Selecting the conjecture. When designing the way how to select the next conjecture to be processed, we had two primary requirements:

- 1. The program should not spend too much time trying to prove a single, possibly too hard conjecture. Instead, it should alternate between the given conjectures.
- 2. The program should favour the easier conjectures because it is likely it will be able to prove them earlier.

Here too we can take an advantage of the information stored in the constructed interpretations. Both above requirements are satisfied by assigning the following weight w_C to each conjecture and selecting the one with the highest weight:

$$w_C = |\{\mathcal{M} \in \mathscr{M} \mid \mathcal{M} \models C\}| - |\mathbf{C}| \cdot u_C$$
(6)

$$u_C$$
 = the number of previous failed attempts when C was selected. (7)

The more interpretations C is valid in, the more general C is likely to be and thus supposedly easier to prove. And each time C is selected we construct an interpretation \mathcal{M} such that $\mathcal{M} \models \neg C$, hence w_C does not increase. On the other hand, if another conjecture D is true in \mathcal{M} , w_D increases. This happens for example if $D \in \mathbf{A}_C$ and it is selected as a premiss for proving C. Subtracting the number of failed attempts (multiplied by the number of conjectures for better efficiency) prevents the program from sticking to a single conjecture in the case when it is unable to construct a new interpretation for a long time and therefore the weights do not change.

4.3 Optimizations

4.3.1 Large number of constructed interpretations

One of the main drawbacks of the algorithm is that when it constructs a new interpretation $\mathcal{M} \models \mathbf{B} \cup \{\neg C\}$, only a small subset **B** of all possible premisses \mathbf{A}_C is true in \mathcal{M} . Most of the other premisses from $\mathbf{A} \setminus \mathbf{B}$ are usually false in \mathcal{M} and therefore they cover \mathcal{M} . High number of possible coverings then leads to construction of a large number of interpretations. One possible solution is to add some of the remaining premisses and/or conjectures $(\mathbf{A} \setminus \mathbf{B}) \cup (\mathbf{C} \setminus \{C\})$ to **B** in such a way that they will not affect the possibility and the difficulty of constructing \mathcal{M} . In particular, if $A \in (\mathbf{A} \setminus \mathbf{B}) \cup (\mathbf{C} \setminus \{C\})$ has no common predicate symbol with any of the formulae in $\mathbf{B} \cup \{\neg C\}$, we can safely add Ato **B**. Depending on the structure of the problem, we can enlarge **B** by several formulae and thus reduce the number of possible coverings of \mathcal{M} , especially at the beginning of the process when the sets of selected premisses are small. On the other hand, in some cases this optimization can considerably slow down the model finder.

4.3.2 Formulae problematic for the model finder

Another question that highly affects the efficiency of the algorithm is the ability of the model finder to construct an interpretation. It is not surprising that some formulae can make the process of searching for models much more difficult. By assigning higher weights β to such problematic formulae we can discourage the algorithm from selecting such formulae.

For example, the model finder **Darwin** is not optimized for equality reasoning and therefore we primarily wanted to avoid formulae with equality. Secondarily, we wanted to assign higher weights to formulae with many variables, because such formulae also make the process of model construction more difficult (this was suggested by Koen Claessen, the author of Paradox). Therefore, we assigned the weight β of a formula A as

$$\beta(A) = \theta(A) \quad \text{if } A \text{ does not contain '='}
\beta(A) = \theta(A) + 10000 \quad \text{if } A \text{ contains '='}$$
(8)

where $\theta(A)$ is defined recursively as

$$\begin{array}{ll}
\theta(P) &= & 0 & \text{if } P \text{ is an atom} \\
\theta((\forall x)F) &= & \theta(F) + 1 & \\
\theta((\exists x)F) &= & \theta(F) + 1 & \\
\theta(\neg F) &= & \theta(F) & \\
\theta(F\Omega G) &= & \max(\theta(F), \theta(G)) & \text{ where } \Omega \text{ is any binary connective} \\
\end{array}$$
(9)

 $\theta(A)$ is simply the maximum number of quantifiers we can encounter when traversing from an atom of A to A's topmost connective. Such proper assignment of weights can indeed reduce the unwanted outcomes when the model finder is not able to find an interpretation.

We also tried to set the weight to $\beta(A) = 2^{\theta(A)}$ (resp. $\beta(A) = 2^{\theta(A)} + 1000$) as this number more closely corresponds to the number of possible variable assignments that must to be examined when deciding the validity of A. This weight also produced very good results.

There are of course many other possibilities how to assign the weights, depending on the nature of the particular problem. For example, when deciding relationships between modal logic systems in [Pud06b] we based the weights on the number of modal symbols occurring in the formula.

5 Empirical evaluation

For the first time we have used the algorithm when deciding relationships between modal systems in [Pud06b]. The first implementation of the algorithm allowed us to decide several cases that we were not able to solve with an ordinary theorem prover. The recent improved version of the program has not yet been evaluated on a large sets of problems. However, the present experiments already show that the method is quite promising.

5.1 Tests on the bushy division of The MPTP Challenge

We have tested the program on the bushy division The MPTP Challenge [US06]. The challenge is focused on automated theorem proving in environments with many axioms, predicates and functors. For the test we have selected 121 problems on which E prover spent more than 10s (or failed at all) in the MPTP referential tests.

All experiments were run on a Linux machine with 4 Dual Core AMD OpteronTM 1.8 GHz Processors and 5 GB of memory. The program was able to take advantage of the multiple processors by running the prover and the model finders simultaneously.

orticlo	thm.	E proof CPU	proved after	proof time	# of interpre-	# of initial	# of selected	# of failed proof
article	name	time [s]	[s]	[s]	tations	prems.	prems.	attempts
finset_1	t17		87	0.20	4860	87	32	392
funct_1	t21		485	0.20	4000	47	$\frac{52}{21}$	322
funct_1	t57		485 90	0.93	135	46	21 20	40
funct_1	t62		30 40	0.03	118	40	20 16	13
funct_1	t02 t145		391	0.04	171	40	10	253
lattice3	t140 t1	14	0.91	0.15	111	143	10	200
lattice3	t3	9				140		
orders_2	t25	147				84		
ordinal1	t23	111	8	0.19	56	44	12	3
pre_topc	t48	1114	320	0.15	2074	71	17	2
relat_1	t12	1117	520 76	0.24	160	36	14	93
relat_1	t25	24	210	0.02	266	34	13	149
relat_1	t44	L-I	276	0.04	249	34	15	351
relat_1	t45		651	0.01	230 284	34	15	1247
relat_1	t74		1728	0.26	466	38	13	2140
relat_1	t86		134	0.60	186	38	18	55
relat_1	t88	14	47	0.12	153	35	16	19
relat_1	t115		66	0.62	228	38	18	22
relat_1	t117	15	39	0.10	145	35	15	15
relat_1	t143	10	58	0.06	253	38	18	24
relat_1	t166		69	0.08	207	38	16	33
relat_1	t167		759	0.04	508	39	18	890
$relat_1$	t174	30	101	0.66	170	39	16	36
relat_1	t178		41	0.11	199	35	12	11
relset_1	t12	729				36		
$subset_1$	t43	89	3	0.18	29	26	14	C
waybell_7	t8	2				195		
waybell_9	t29	31				249		
wellord1	129		12	0.03	93	36	15	3
wellord1	t19		86	0.58	161	54	25	62
wellord1	t21		9	0.48	89	41	12	C
wellord1	t24	31				35		
wellord2	t3		15	0.10	131	55	19	(
wellord2	t5		45	0.06	153	55	21	C
yellow_0	t42		472	0.07	3237	89	27	C
yellow_0	t60		790	0.05	4119	91	27	C
yellow_0	t61		771	0.09	2385	96	34	180
yellow_1	t2	16				183		
zfmisc_1	t99		3	0.60	11	13	7	C
$z fmisc_1$	t136	96				8		

Table 1: Results for 121 "difficult" problems from the bushy division of MPTP.

The results of the test are summarized in Table 1. The table shows only the problems solved by E or by the program (or by both). The third column shows the CPU time of standalone E. The fourth column shows the wall-clock time⁴ of the program. The fifth column shows the CPU time of E on the premisses selected by the program. The sixth column shows the number of interpretations constructed by the program during the search. The last column shows the number of failed attempts, that is the number of cases where the program was neither able to construct a new model nor to prove the conjecture. If the program (or E) were not able to solve a problem, the corresponding columns are left blank.

The presented experiments were conducted with the CPU limit set to 1.2s for both the prover and the model finders. When have noticed that the program performs better when the time limits for the prover and for the model finder are very low.⁵ This is explained by the observation that in vast majority of cases if a model cannot be found within a few seconds it is not found at all. And if an optimal (or near-optimal) set of premisses is eventually found, the prover spends very little time on proving the conjecture. Thus, if the time limits are higher, the program wastes much of its time on unsuccessful attempts and is not able to construct as many interpretations.

The program was able to prove 31 problems of 121 compared to 15 of standalone E. 6 problems were solved by both E and the program. The total time is mostly affected by the number of failed attempts and by the total number of constructed interpretations. As expected, the proofs from the selected subsets of premisses (fifth column) indeed took very little time, just fractions of seconds. E exhibits the usual behavior of proving tools that with increasing time it becomes overwhelmed by the number of derived clauses. In our case, E successfully proved only two conjectures after time longer than 150s. On the other hand, when the program was given enough time, it was able to prove many more difficult conjectures.

5.2 SRASS

The idea was recently implemented by Geoff Sutcliffe and Yury Puzis in SRASS system [SP07]. The system uses more advanced techniques to further improve performance, including a syntactic relevance measure as an ordering heuristic to guide the selection process. SRASS is able to solve many problems that can not be solved by the underlying theorem prover alone and is among the most successful systems entering The MPTP challenge [US06].

6 Conclusion and further work

The main difference from the commonly used proving techniques is that the algorithm is capable of selecting only those premisses that are necessary for the proof and that its decisions are based not on a syntactic but on a semantic basis.

⁴The program runs in a platform-independent language and moreover it spawns multiple child processes (the prover and the model finders), therefore we were not able to gather the total CPU time of the program.

⁵Of course this observation applies only to this particular set of problems.

The program was eventually able to find a proof for some conjectures that were too difficult for the classical black-box prover. When an optimal set of premisses was found, the proof constructed from this set usually took just a fraction of second. Thus, even if it takes the program a long time to find a proof, the optimal set of premisses provides the information how to prove the conjectures efficiently.

6.1 Advantages

6.1.1 Reusing interpretations

The interpretations constructed during the process of proving one conjecture can be reused by the algorithm for proving other conjectures, even from different theories. It is only required that the theories share the same language.

By accumulating such interpretations, the program gains knowledge about relationships between theorems and premisses within the theory. As the process is fully automated, the algorithm could be used to gather a large amount of such interpretations without any user intervention and these interpretations could be then used to answer queries in the future.⁶ Therefore, we surmise that the algorithm could prove its worth for large databases of mathematical knowledge, both for automated and interactive proving of theorems.

6.1.2 Saving the state of the algorithm

The state of the algorithm can be easily saved and restored. The state is fully described by the set of unsuccessful attempts \mathscr{F} and by the set of constructed interpretations \mathscr{M} . Hence, the algorithm can be easily suspended, if there is a more important task to be solved, and resumed later in the future. Or, like in our implementation, the program can alternate the conjecture it tries to prove according to some criterion.

6.1.3 More efficient on long runs than conventional provers

As we have noticed in our example, classical black-box provers usually become ineffective (and tend to consume a lot of memory) after a few minutes, when the number of inferred clauses becomes too large. The program does not suffer from this issue, as it only needs to remember the constructed interpretations⁷ and the set of failed attempts.

6.2 Disadvantages

6.2.1 A high number of constructed interpretations

A major disadvantage of the algorithm is that the number of constructed interpretations can become quite high, especially if there are a lot of similar axioms and lemmas in the set of premisses. Although the program was able to handle even several tens of thousands

 $^{^6\}mathrm{This}$ idea was first proposed by Jiří Vyskočil in a personal discussion.

⁷In fact, it is even not necessary to remember the interpretations. It is sufficient to store the information of what premisses from \mathbf{A}_C are true in each interpretation, that is just $|\mathcal{M}| \cdot |\mathbf{A}_C|$ Boolean values.

of interpretations, we believe that reducing the number would make the method much more efficient. We suggest the following improvements:

- 1. Focus on constructing interpretations in which a high number of the premisses is true. The higher the number, the less is the number of the remaining premisses that can be used to avoid the interpretation. Thus, the algorithm would be led more directly towards finding the optimal set of premisses.
- 2. Try to construct interpretations in which several of the conjectures are false. A single interpretation can be used to guide the search for all conjectures which are false in the interpretation.

We have already partially implemented these two suggestions as described in section 4.3.1.

3. If possible, analyze what interpretations became unnecessary and remove them from the pool. As we have seen, the algorithm never constructs two isomorphic interpretations (Corollary 1). Therefore, every constructed interpretation carries a unique piece of knowledge. It will be necessary to devise a more sophisticated technique for detecting which interpretations participate only negligibly on the selection of premisses and thus can be discarded.

6.2.2 Limitation to finite interpretations

In general, the problem of constructing a model of a set of formulae is algorithmically undecidable. Any algorithm we construct will be only able to construct some specific class of interpretations. In most cases, it will be the class of finite interpretations. And as many interesting theories (like arithmetic or set theory) have infinite interpretations, this may impose a strict restriction on the method. However, the situation needs not to be as hopeless as it seems to be. Recall that we only construct interpretations of subsets of the given axioms. If we force the algorithm to elude those axioms that cause the interpretation to be infinite, we may still be able to achieve good results. This can be easily arranged by assigning high weights to those problematic axioms. However, we are not aware how to automatically identify such axioms.

Another solution would be to construct and represent some class of infinite interpretations. For example, one could construct Herbrand interpretations, saturations or more generally represent the interpretations as complex functions or programs that compute the validity of formulae.

6.3 Comparison with syntactic techniques

As it turns out, the algorithm falls short of the original expectation that it could very effective when the number of premisses is very large (hundreds or more). The reason is the high number of interpretations that need to be constructed to determine the optimal set of premisses in such cases. Syntactic approaches as [Urb06a, Urb06b, RS98, MP06] seem to be much more efficient. However, the value of the algorithm emerges in the cases where even the number of premisses is moderate, the prover is overwhelmed by unnecessary premisses.

It seem quite possible that a combination of the semantic approach with syntactic techniques might be very efficient. This assumption is supported by the results of the SRASS system [SP07] in the MPTP Challenge [US06] (see also Section 5.2).

6.4 Further work

We have already suggested many possible areas of improvement in the previous section. We believe that considerable improvements could be attained by close cooperation with the designers of model finders and theorem provers to implement some of the following proposals:

- 1. Guide the model finder when constructing interpretations. We suggest that the model finder would be given two sets of formulae, **A** and **L**. It would be required to find a model $\mathcal{M} \models \mathbf{A}$ such that it is also a model of as much as possible formulae from **L** (for example within some given time limit, domain size, etc.).
- 2. Reveal which premisses are well suited for a particular theorem prover or a model finder. This information could be used to minimize the number of cases where neither a proof nor a new interpretation are found, and secondly to develop a better founded, automated process of assigning weights to premisses.

The program would also benefit from a graphical user interface that would facilitate visual and interactive analysis of the process.

Although the method is not mature yet and many aspects still need to be researched, we believe that it can bring significant benefit to the task of automated proving of theorems. Not only it can allow to prove conjectures that are hard to prove by conventional prover, but it also opens the possibility to further analyze the relationships between the conjectures, the premisses and the interpretations that it constructs.

References

- [BFdNT06] Peter Baumgartner, Alexander Fuchs, Hans de Nivelle, and Cesare Tinelli. Computing finite models by reduction to function-free clause logic. to appear, preliminary version, June 2006.
 - [Chv79] Vašek Chvátal. A greedy heuristic for the set-covering problem. Math. Oper. Res., 4:233–235, 1979.
 - [CS03] Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style model finding. In Proc. of Workshop on Model Computation (MODEL), 2003.
 - [CS07] Koen Claessen and Niklas Sörensson. Paradox a first-order logic model finder. WWW pages, 2007. http://www.cs.chalmers.se/~koen/paradox/.
 - [FBT07] Alexander Fuchs, Peter Baumgartner, and Cesare Tinelli. Darwin a theorem prover for the model evolution calculus. WWW pages, 2007. http://combination.cs.uiowa.edu/Darwin/.

- [Fei98] Uriel Feige. A threshold of $\ln(n)$ for approximating set cover. J. ACM, 45(4):634-652, 1998.
- [Kar72] Richard Manning Karp. Reducibility among combinatorial problems. In Complexity of Computer Computations, pages 85–103. Plenum Press, 1972.
- [MP06] Jia Meng and Lawrence Paulson. Lightweight relevance filtering for machine-generated resolution. volume 192 of *CEUR Workshop Proceedings*, 2006.
- [Pud06a] Petr Pudlák. Search for faster and shorter proofs using machine generated lemmas. In G. Sutcliffe, R. Schmidt, and S. Schulz, editors, Proceedings of the FLoC'06 Workshop on Empirically Successful Computerized Reasoning, 3rd International Joint Conference on Automated Reasoning, volume 192 of CEUR Workshop Proceedings, pages 34–52, 2006.
- [Pud06b] Petr Pudlák. Verification of Mathematical Proofs. PhD thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2006. Available online at http://lipa.ms.mff.cuni.cz/~pudlak/pp-thesis.pdf.
 - [RS98] W. Reif and G. Schellhorn. Theorem proving in large theories. In W. Bibel and P. Schmitt, editors, Automated Deduction—A Basis for Applications, volume III, 2. Kluwer Academic Publishers, Dordrecht, 1998.
 - [Sch02] S. Schulz. E A brainiac theorem prover. Journal of AI Communications, 15(2-3):111–126, 2002.
 - [Sch07] Stephan Schulz. The E equational theorem prover. WWW pages, 2007. http://www.eprover.org/.
 - [SP07] Geoff Sutcliffe and Yury Puzis. SRASS a semantic relevance axiom selection system, 2007. http://www.cs.miami.edu/~tptp/ATPSystems/SRASS/.
 - [SS98] G. Sutcliffe and C. B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. Journal of Automated Reasoning, 21(2):177–203, 1998.
- [Urb06a] Josef Urban. MizarMode an integrated proof assistance tool for the Mizar way of formalizing mathematics. *Journal of Applied Logic*, 4(1):414–427, 2006.
- [Urb06b] Josef Urban. MoMM fast interreduction and retrieval in large libraries of formalized mathematics. *International Journal on Artificial Intelligence Tools*, 15(1):109–130, 2006.
 - [US06] Josef Urban and Geoff Sutcliffe. The MPTP \$100 Challenges, 2006. http://www.cs.miami.edu/~tptp/MPTPChallenge/.