

A Procedural Implementation of “Minimize” in Smodels

James D. Jones
Eric Nelson
Umit Topaloglu
Hemant Joshi

Department of Computer Science
University of Arkansas at Little Rock
james.d.jones@acm.org

Abstract

This paper proposes a two pass procedural method to find all of the answer sets of a logic program that are minimal with respect to user specified criteria. This is of concern in situations where “fewer are preferred to more”. For instance, in a court setting, either John lied while under oath, or Tom, Mary, and Sally all lied while under oath. In this circumstance, it is easier to believe that there is one liar than that there are three liars. The Smodels minimize function incorrectly returns the first minimal model that it computes. This has relevance to homeland security and national defense, and in fact, surfaced while doing such work.

Introduction

This problem surfaced while doing preliminary work for DOD on the automatic detection of deception. In a contrived scenario involving espionage, the sequence of events was supposed to be as follows. We were trying to identify double agents. Initially, there is equilibrium, and there are no suspected double agents. At some point, an agent communicates a message. Lacking any reason to doubt the credulity of the agent, we accept the communication as true.

At a later point, another agent communicates another message. The two messages are in conflict with each other, and for the moment, we assume that one of the agents must be lying. (Of course, this is a simplification. There could be other explanations such as one of the agents making a mistake, or an agent communicating false information which he/she believes to be true, etc.) At this point, we do not know which of the two agents is lying. In this scenario, we believe that fewer liars are preferred to more liars, hence only one of the agents is lying, rather than both lying. Therefore, we want to maintain two views of the world: one in which the first agent is lying, and one in which the second agent is lying. (This ability to maintain multiple views of the world is a powerful feature of our paradigm, and is not provided by any other reasoning mechanism.) It is at this very point that the *minimize* statement of SMOBELS “fails”.

It returns just one model, stating that the first agent is lying. (In this paper, “answer set” and “stable model” are used interchangeably. Almost all the recent literature prefers the term “answer set”, but the “compiler” SMOBELS uses the term “stable model”.) If we had not used the *minimize* statement, SMOBELS would have returned all answer sets; in this case, there would have been just two answer sets, the two we were anticipating. However, even though NOT using the *minimize* statement would have yielded the correct results at this step, it would also have yielded incorrect results in the next paragraph. We need a more consistent approach that does not require foreknowledge of when certain features are to be used, and when they are to be avoided. Investigation revealed that the problem exists not with our program, but with the platform (the “compiler”, Smodels.)

Continuing our scenario, at yet a later time, a third agent communicates another message. This message corroborates the message uttered by the first agent, and is in conflict with the message uttered by the second agent. Therefore, either the first agent and the third agent are lying, or the second agent is lying. Preferring fewer liars to more liars, the software correctly tells us that the second agent is lying. The reason SMOBELS using the *minimize* statement gives us the correct result is that there was only one answer set that met the *minimize* criteria. If there had been more than one answer set meeting the *minimize* criteria, SMOBELS would have returned just the first one computed.

Let us digress, and mention some technical details of the paradigm we are using. The main technical tool we are using for reasoning about deception is *A-Prolog* -- a language of logic programs under the answer set (stable model) semantics (Gelfond and Lifschitz, 88), (Gelfond and Lifschitz, 91). *A-Prolog* can be viewed as a purely declarative language with roots in logic programming (Kowalski 1974; Kowalski 1979), syntax and semantics of standard Prolog (Colmerauer, et. al., 1973), (Clark 1978), and in the work on nonmonotonic logic (Reiter 1980), (Moore 1985). The inference engine used is SMOBELS. This inference engine is aimed at computing answer sets (stable models) of programs of *A-Prolog* (Niemela and Simons, 1997; Niemela and Simons, 2000; Cholewinski, et. al., 1996).

That is, Smodels is an implementation of A-Prolog. (It is very much akin to a “compiler”.) It computes all the answer sets of a program. An answer set is a collection of facts about a problem. These facts are based on rules and conclusions inferred from these rules. It is possible for a problem to have one or many answer sets.

Problem

Figure 1 presents a simple program to demonstrate the issues we have discussed. In this program, we have 4 agents (the top part of the figure), and we have 4 rules that help us decide if an agent is also a double agent (the middle part of the figure.) Each rule for drawing the conclusion that an agent is a double agent depends upon whether certain other agents are not double agents. This program has more than one answer set (the bottom part of the figure.) The possible answer sets are that a1 is a double agent (identified as “answer 1”), that a2 is a double agent (identified as “answer 2”), or that both a1 and a2 are double agents (identified as “answer 3”). These results are normal, and what is expected.

```

agent (a1) .
agent (a2) .
agent (a3) .
agent (a4) .

holds(double_agent(a1),0,1) :- not holds(double_agent(a2),0,1),
                               not holds(double_agent(a3),0,1) .
holds(double_agent(a2),0,1) :- not holds(double_agent(a1),0,1),
                               not holds(double_agent(a3),0,1) .

holds(double_agent(a3),0,1) :- not holds(double_agent(a1),0,1),
                               not holds(double_agent(a2),0,1) .
holds(double_agent(a4),0,1) :- not holds(double_agent(a1),0,1),
                               not holds(double_agent(a2),0,1) .

smodels version 2.26. Reading...done
Answer: 1
Stable Model: holds(double_agent(a1),0,1) agent(a4) agent(a3) agent(a2)
agent(a1)
Answer: 2
Stable Model: holds(double_agent(a2),0,1) agent(a4) agent(a3) agent(a2)
agent(a1)
Answer: 3
Stable Model: holds(double_agent(a3),0,1) holds(double_agent(a4),0,1)
agent(a4) agent(a3) agent(a2) agent(a1)

```

Figure 1 - our original program

Now suppose we take the posture that “fewer is preferred to more”, with respect to double agents. That is, among the possible models of a program, we want to consider only those models that have the fewest number of double agents. With regard to the program presented in figure 1, we want our result to consider only “answer 1” and “answer 2”. Those two models have one agent identified as a double agent, whereas the answer set we want to exclude, “answer 3” has two agents identified as double agents.

SMODELS has a feature which on the surface, seems to provide what we have just described. That feature is the “minimize” statement. In our case, wanting to minimize on the number of double agents, we would write the statement as follows:

$$\text{minimize}\{\text{holds}(\text{double_agent}(\text{a1}),0,1), \text{holds}(\text{double_agent}(\text{a2}),0,1), \text{holds}(\text{double_agent}(\text{a3}),0,1)\}$$

Unfortunately, the syntax requires us to use only ground literals within the statement. We wish we could use something simpler, such as

$$\text{minimize}\{\text{holds}(\text{double_agent}(\text{X}),\text{Y},\text{Z})\}$$

Our program from figure 1 including this statement is found in figure 2. The difference between figure 1 and figure 2 is the last line of figure 2. Based upon our

intuition of how *minimize* should work, the program in Figure 2 should produce the output in Figure 3.

```
agent(a1).
agent(a2).
agent(a3).
agent(a4).

holds(double_agent(a1),0,1) :- not holds(double_agent(a2),0,1),
                               not holds(double_agent(a3),0,1).
holds(double_agent(a2),0,1) :- not holds(double_agent(a1),0,1),
                               not holds(double_agent(a3),0,1).

holds(double_agent(a3),0,1) :- not holds(double_agent(a1),0,1),
                               not holds(double_agent(a2),0,1).
holds(double_agent(a4),0,1) :- not holds(double_agent(a1),0,1),
                               not holds(double_agent(a2),0,1).

minimize{holds(double_agent(a1),0,1), holds(double_agent(a2),0,1),
holds(double_agent(a3),0,1)}.
```

Figure 2 - modified program include *minimize* statement

```
smodels version 2.26. Reading...done
Answer: 1
Stable Model: holds(double_agent(a1),0,1) agent(a4) agent(a3) agent(a2)
agent(a1)
Answer: 2
Stable Model: holds(double_agent(a2),0,1) agent(a4) agent(a3) agent(a2)
agent(a1)
```

Figure 3 - EXPECTED results of the *minimize* statement

Unfortunately, the *minimize* function call actually returns only the first model computed that meets the criteria of the *minimize* statement. In figure 1, we see that there were 3 answer sets for our original program. Minimizing among those answer sets with respect to *double_agent* should yield the two answer sets of figure 3. (The order of the literals specified in the *minimize* statement dictate which model will be computed.) The actual results are presented in figure 4.

```
smodels version 2.26. Reading...done
Answer: 1
Stable Model: holds(double_agent(a2),0,1) agent(a4) agent(a3) agent(a2)
agent(a1)
```

Figure 4 - ACTUAL results of the *minimize* statement

Procedure to Achieve *Minimize*

We have found a way to implement our intuitive view of minimize. This is a two pass procedural execution. First run the program using the minimize statement. Then rerun the program substituting a choice rule in place of the minimize function.

The purpose for this first run is to tell us how many literals from the *minimize* criteria are required to be in any model. Since we have *minimized* upon those literals, we know that no model with fewer literals exists. We also know that since we are *minimizing* upon those literals, we do not want any models with greater than that number of literals. Hence, we discover the cardinality that precisely defines how many of the *minimized* literals are to be in each model. Figure 5 shows us that if we are *minimizing* with respect to *double_agents*, then each acceptable model of this program must have exactly one *double_agent*. (Note the bottom of the results from figure 5. The minimize statement has been repeated, and a new piece of information has been provided to us: “min=1”. This is important information for us, but was eliminated from earlier figures.)

The second step of this process is to run the same program, replacing the *minimize* statement with a *choice* rule. The *choice* rule for our program is:

$$1 \{ \text{holds}(\text{double_agent}(A), 0, 1) : \text{agent}(A) \} 1$$

You will notice that the differences between this rule, and a regular rule in our program are: 1) the statement is bracketed by curly braces, and 2) there are integers outside the curly braces. In our case here, it is merely circumstantial that the head of the rule and the body of the rule contain variables, rather than ground terms. (The rest of the rules in our program are *ground rules*, and contain only ground terms. Again, this is only circumstance.)

The integer to the left of the rule is called the lower bound. It indicates the minimum number of such literals must be in the accepted models. In our case, the lower bound for the choice rule can be 0 or 1; we have chosen 1. The integer to the right of the rule is the upper bound, and specifies the maximum number of such literals that can be in the accepted models. The upper bound of the choice rule should be the *min* value returned from the minimize execution (see Figure 5). Using the *min* value as the upper bound allows us to compute all the stable models that have that number of literals or less in the models. (Since we *minimized*, we know that no models exist with fewer than *min* literals.) Or, for simplicity, we can use the *min* value for both the upper and lower bounds. This will compute all the stable models that have only that number of literals in the model, which is what we want, and is presented in figure 6. It is cumbersome to run a program twice, replacing the minimize statement with a choice rule. Nonetheless, give the limitations of the current version of SMOBELS, this is necessary in order to achieve dynamic results. Programs can be arbitrarily large, and in truly volatile situations, it will not be known in advance the number of literals that should be in minimized models. Figure 6 is the result we want. Unfortunately, we have to go through the process of producing something like figure 5 in order to get the results of figure 6.

```

agent (a1).
agent (a2).
agent (a3).
agent (a4).

holds(double_agent(a1),0,1) :- not holds(double_agent(a2),0,1),
                               not holds(double_agent(a3),0,1).
holds(double_agent(a2),0,1) :- not holds(double_agent(a1),0,1),
                               not holds(double_agent(a3),0,1).

holds(double_agent(a3),0,1) :- not holds(double_agent(a1),0,1),
                               not holds(double_agent(a2),0,1).
holds(double_agent(a4),0,1) :- not holds(double_agent(a1),0,1),
                               not holds(double_agent(a2),0,1).

minimize{holds(double_agent(a1),0,1), holds(double_agent(a2),0,1),
         holds(double_agent(a3),0,1)}.

smodels version 2.26. Reading...done
Answer: 1
Stable Model: holds(double_agent(a2),0,1) agent(a4) agent(a3) agent(a2)
agent(a1)
{ holds(double_agent(a1),0,1), holds(double_agent(a2),0,1),
  holds(double_agent(a3),0,1) } min = 1

```

Figure 5 - minimal models must have only one *double_agent*

```

agent (a1).
agent (a2).
agent (a3).
agent (a4).

holds(double_agent(a1),0,1) :- not holds(double_agent(a2),0,1),
                               not holds(double_agent(a3),0,1).
holds(double_agent(a2),0,1) :- not holds(double_agent(a1),0,1),
                               not holds(double_agent(a3),0,1).

holds(double_agent(a3),0,1) :- not holds(double_agent(a1),0,1),
                               not holds(double_agent(a2),0,1).
holds(double_agent(a4),0,1) :- not holds(double_agent(a1),0,1),
                               not holds(double_agent(a2),0,1).

1{holds(double_agent(A),0,1) : agent(A)}1.

smodels version 2.26. Reading...done
Answer: 1
Stable Model: holds(double_agent(a2),0,1) agent(a4) agent(a3) agent(a2)
agent(a1)
Answer: 2
Stable Model: holds(double_agent(a1),0,1) agent(a4) agent(a3) agent(a2)
agent(a1)

```

Figure 6 - ALL models meeting the *minimized* criteria

Summary

Logic programming, and in particular, the answer set semantics provide us an incredibly powerful reasoning paradigm. Commercially available implementations of Prolog exist, and are very efficient. However, these implementations are based upon simpler semantics. To enjoy the benefits of the state of the art in the semantics of logic programs, the most sophisticated implementations (“compilers”) available are SMOBELS and DLV. DLV is limited in that it does not allow functions as terms. (This is similar to the situation that exists with DATALOG vs. PROLOG.) Hence, from a semantics perspective, to provide the most capable reasoning possible, SMOBELS is the only implementation available.

We have identified a need to minimize among models according to certain user specified criteria. This would be needed in applications where “less is preferred to more” with respect to the user specified criteria. SMOBELS attempts to provide this ability, but fails. We have identified a 2-step procedural solution to this problem. The first step is to run the program with a minimize statement. This execution gives us results that do not match our intuition. Nonetheless, this execution gives us a very important piece of information: the minimum number of literals with respect to the user specified criteria that must be in each model. The second step is to run the program again, replacing the *minimize* rule with a *choice* rule, where the *min* value from step 1 is used as the lower bound and the upper bound in the choice rule.