Implementing Pstable

Alejandra López Fernández

Universidad de las Américas-Puebla, Sta. Catarina Mártir, C.P. 72820, Cholula, Puebla, México
alejandra.lopezfz@udlap.mx

Abstract. We present an implementation of pstable model semantics. Our implementation uses the well known tools: MiniSat and Lparse.

1 Introduction

The stable model semantics for logic programming lies within the context of nonmonotonic reasoning, [1]. It has been widely used and it is perhaps the most well known semantics for knowledge representation. The pstable model semantics, introduced in [2] shares several properties with stable model semantics, but it is closer to classical logic than stable.

Pstable model semantics has at least the same expressiveness of stable, this means we can model anything that is modeled in stable, and probably more. This is why we are interested in its implementation. The purpose of this paper is to show the implementation of pstable model semantics.

We present in Section 3 an implementation of pstable for normal programs in terms of MiniSat y Lparse. Finally, Section 4 concludes the paper and offers future work. We assume that the reader has some familiarity with logic.

2 Background

We introduce some concepts which are needed for further understanding of the work and the results presented on this document. We present the definitions of the pstable model semantics that is the main topic of this paper, more basic definitions can be found in [2].

2.1 Normal Program

As we have already said in the introduction we have a particular interest in the class of normal programs. These are programs formed by *normal rules* for which the body is a conjuntion of literals.

Definition 1. *[1] The programs under consideration are sets of rules of the form:*

$$a \leftarrow l_1 \wedge l_2 \wedge .... \wedge l_n \tag{1}$$

where a is an atom, $l_1, l_2, ..., l_n$ is a set of literals and $n \geq 0$. We say that $l_1 \wedge l_2 \wedge .... \wedge l_n$ is the body of the rule and a is the head of the rule. When every literal in the body is an atom, we say that the rule is positive.

2.2 CNF Program

We say that a program is in conjunctive normal form (CNF) if it is a conjunction of formulas, and each formula is a disjunction of literals.

Example 1. The following logic program P is in CNF.

$(a \vee b)$
$(b \vee c)$

2.3 Reduction

Before we present the stable and pstable model semantics definitions, we show the reduction defined in [2].

Definition 2. *Let P be a normal program and M a set of atoms:*

$$\mathcal{R}ed_M(P) = \{a \leftarrow \beta^+ \wedge \neg(\beta^- \cap M) \mid a :- \beta^+ \wedge \neg\beta^- \in P\} \tag{2}$$

and where β^+ and β^- are sets containing, respectively, the positive and negative atoms that occur in the body of the clauses of P.

In other words, with this reduction every atom that is not in the model M and is negated at the body of a rule is deleted.

Example 2. Take the following logic program P:

$b \leftarrow \neg a.$
$a \leftarrow \neg b.$

Given $M = \{a\}$, it follows that $Red_M(P)$ is the program:

$b \leftarrow \neg a.$
$a.$

2.4 Pstable Model Semantics

The definition of the pstable model semantics was shown in [2] and is the following:

Definition 3. *Let P be a normal program, and M a set of atoms. We say that M is a pstable model of P if M is a model of P and $Red_M(P) \models M$.*

3 Pstable Model Finder

In order to make some tests and evaluate some examples we made a preliminar implementation of a *Pstable model finder*. Pstable model finder is a model finder that applies the Pstable models semantics. In this section first we show the design of the finder, later we describe the tools that were used in each part of the implementation and finally we explain and exemplify how each part works.

3.1 Design

There are four main blocks *Interface*, *Exec_Lparse*,*MinModel*, and *Pstable_ Model*.

- *Interface.* The user selects a program and asks to get the pstable models or the stable models, the models found are showed at the interface.
- *Exec_Lparse.* Once the normal program is given by the user, the Lparse program is executed, this gives us a program free of variables that it would be useful in order to get a CNF program.
- *MinModel.* A CNF program is build using the output of Lparse. This CNF program is used to get with a simple process all minimal models, the SAT solver used for this purpose is Mini-SAT.
- *Pstable_Model.* Each minimal model is checked in order to determine if it is also a pstable model, if it turns out that it is a pstable model, this is showed with all the pstable models we get.

All these blocks are explained in detail later.

3.2 Tools

Now we will describe the tools we used to build the system. First we describe *Lparse* which is the most feature-rich of the different parsers and front ends. Later we describe MiniSat, what is it and how is it used.

Lparse. Lparse is a front-end that generates a variable-free simple logic program. As input this receives a normal program in a file with extension .lp. We allow only normal programs. We use the 1.0.17 version, it is available in http://www.tcs.hut.fi/Software/smodels/.

Example 3. Take the following logic program P.

$b \leftarrow \neg a.$
$a \leftarrow \neg b.$

The input for Lparse must have the following format:

$b : -not\ a.$
$a : -not\ b.$

Note: The comma is used for indicate the conjuntion of literals.
Using this program P as input of Lparse, we get the following output:

1 1 1 1 2
1 2 1 1 1
0
1 b
2 a
0
$B+$
0
$B-$
0
1

The first part of the listing consists of the the rules of the program:

1 1 1 1 2
1 2 1 1 1
0

The first number denotes the rule type, this is one for basic rules, in this example all the rules are basic. The next number identifies the head of the rule. In this case, the atom a is represented by 2 and the atom b by 1. Next comes the body definition; the first number is the total number of literals in the body and the second one is the number of negative literals. The rest of the line contains the numbers of the literals, with negative ones being in front. The line with just 0 shows the end of the rules.

The second part is the symbol table containing the atom definitions:

1 b
2 a
0

The third part contains the compute statement:

$B+$
0
$B-$
0
1

After B+ comes the list of atoms that should be true in the model. After B- comes a list of atoms that shouldn't be in the model. The last 1 signifies the number of models that should be calculated.

MiniSat. We decided to use MiniSat because of its features mainly its highly efficient. How we saw at MiniSat web page, this SAT solver is the winner of all the industrial categories of the SAT 2005 competition. We use the 1.14 version, it is available at http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/MiniSat.html.

This solver operates on problems which are specified in conjunctive normal form (CNF).

Example 4. Take the following program:

$a \vee b$
$b \vee c$

The input for MiniSat must have the following format:

$p\ cnf$ 3 2
2 1 0
2 3 0

The first line consists of the description of the program. After p, the word *cnf* indicates that the program is in conjunctive normal form, later the first number indicates the total number of literals in the program, the second number is the total number of rules.

$p\ cnf\ 3\ 2$

Later all the rules are listed, each row represent a disjunction rule, and each line contains the literals of each disjunction, the 0 shows the end of the row. The literals must be integer numbers, the positive ones represent positive atoms and the negative ones represent negative atoms.

The output MiniSat gives us is the interpretation for each atom:

SAT
$-1\ 2\ -3\ 0$

The first line says if the program is satisfiable or not, the satisfiability is expressed with the word SAT and the unsatisfiability with UNSAT.

If the program in satisfiable, the second line contains the interpretation for each atom; if interpretation is 0, the atom appears negative, otherwise if interpretation is 1, the atom appears positive. The last 0 indicates the end of the line.

3.3 Implementation

The Pstable model finder was build using Java language. All the tools used for the implementation have already been described.

The input of the system is a file with extension *.lp*, this file *name.lp* must contain the basic standard normal program format, the format of this program was defined in section 3.2.

The input program can have facts, rules with just an atom as head and in the body any literals separated by commas, the comma is a conjunction symbol.

Example 5. The following program P is a correct input program:

$b.$
$a :- not\ b,\ c,\ not\ d.$
$b :- not\ c.$
$c :- a,\ d.$

Exec_Lparse Once that we have an input file *name.lp*, the system executes the *Lparse* program through the class *Runtime* that is part of the Java API. This class lets us to run any external program from within our system. The output of the *Lparse* is written in a file (*name.in*). In this we have the description of the logic program:

Example 6. Take the following program P:

$a :- \ not \ b.$
$b :- \ not \ a.$

The output file that Lparse gives us is the following:

1 1 1 1 2
1 2 1 1 1
0
1 b
2 a
0
$B+$
0
$B-$
0
1

Using the output file we get from Lparse, a *CNF* program is build, this means we convert a normal program to a *CNF* one.

Definition 4. *Given any implication formula $X \rightarrow Y$, we can express it as a disjuction formula as follows:*

$\neg X \vee Y$

Using the definition above we convert a normal program to a CNF one converting each normal rule in a disjunction rule.

Example 7. Take the following program P:

$a \leftarrow \neg b \wedge c.$
$b \leftarrow \neg a.$

Using the definition 4 the CNF program of P is the following:

$b \vee \neg c \vee a.$
$a \vee b.$

MinModel. This new CNF program is the input for the class *MinModel*. Because Mini-SAT is a SAT solver it does not get minimal models, but with a simple process we obtain all minimal models, if there is any.

Definition 5. *Given a set of atoms M we define $Neg(M)$ as the following program:*

- *a conjunction of all negative atoms $\in M$.*
- *a disjuction of all positive atoms $\in M$ but negated.*

Given a program P we can find out if a model M is also a minimal model adding to the program P the program $Neg(M)$.

Example 8. Take the following program P:

$a \vee b \vee c.$

Given $M = \{a, b, c\}$, if we add $Neg(M)$ to the program P we get the following program:

$a \vee b \vee c.$
$\neg a \vee \neg b \vee \neg c.$

This program has a model $M = \{a, b, \neg c\}$, so we add $Neg(M)$ to the program.

$a \vee b \vee c.$
$\neg a \vee \neg b \vee \neg c.$
$\neg c.$
$\neg a \vee \neg b.$

Now we get a model $M = \{a, \neg b, \neg c\}$ so we add $Neg(M)$ also.

$a \vee b \vee c.$
$\neg a \vee \neg b \vee \neg c.$
$\neg c.$
$\neg a \vee \neg b.$
$\neg b.$
$\neg a.$

This program does not have any model so the last model $M = \{a\}$ is a minimal model of P.

The algorithm for getting the minimal models is the following:

```
MinModel(P)
begin
   M = ExecuteMiniSat(P); //MiniSat gets a model for P
   while(SAT) //While there is models
   begin
       Pc = P U M; //Add to P last model we get but negated
       Mc = ExecuteMiniSat(Pc); // MiniSat gets a model for Pc
       while(SAT) //While there is models
       begin
          Pc = Pc U Mc //Add to Pc last model we get but negated
          Mc = ExecuteMiniSat(Pc); // MiniSat gets a model for Pc
       end
       //There is not any model, the last one obtained is minimal
       push(MinmodelsStack, Mc);
       P = P U Mc; //Add to P last minmodel we get but negated
       M = ExecuteMiniSat(P); //MiniSat gets a model for P
   end
   return MinmodelsStack;
end
```

Pstable Model. In this part we implemented the concepts described in section 2 (background), $Red_M(P)$. Applying this reductions we verify if a minimal model we obtain is also a pstable model. It is easier to understand this process with an example.

Example 9. Considering the following logic program P:

$b \leftarrow \neg a.$
$a \leftarrow \neg b.$

Given $M = \{a\}$, one of the minimal models of P, it follows that $Red_M(P)$ gives as result the program:

$b \leftarrow \neg a.$
$a.$

Later, with the resultant program of $Red_M(P)$, we convert a normal program into a CNF one, this gives us as result:

$b \vee a$
$a.$

In this case, we can directly see that $Red_M(P) \models M$, but in order to prove M, after the reduction, we simply add to the program a disjunction of the atoms of M but negated, $\neg M$.

$b \vee a.$
$a.$
$\neg a.$

This new program is parsed to CNF and is used as an input of MiniSat, if we get UNSAT then we have proved that M is a *pstable model*, otherwise it is just a model. In this case the result of MiniSat is UNSAT so the model $M = \{a\}$ is a pstable model of P.

4 Conclusions

Our main conclusion is that MiniSat and Lparse are powerful tools that facilitate the implementation of Pstable. For future work we would like to improve the software, making explicit the use of constraints rules.

References

1. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press, URL citeseer.ist.psu.edu/gelfond88stable.html.
2. Mauricio Osorio Galindo and Juan Antonio Navarro Pérez and José R. Arrazola Ramírez and Verónica Borja Macías". Logics with common weak completions. *Journal of Logic and Computation*, 2006. URL doi: 10.1093/logcom/exl013