Possibilistic Stable Model Computing

Pascal Nicolas and Claire Lefèvre

LERIA - Faculty of Sciences - University of Angers
2 Boulevard Lavoisier, 49045 Angers Cedex 01, France
{pascal.nicolas, claire.lefevre}@univ-angers.fr

Abstract. Possibilistic Stable model Semantics is an extension of Stable Model Semantics that allows to merge uncertain and non monotonic reasoning into a unique framework. To achieve this aim, knowledge is represented by a normal logic program where each rule is given with its own degree of certainty. By this way, it formally defines a distribution of possibility over atom sets that, on its turn, induces for each atom a possibility and a necessity measures. The latter underpins the definition of a possibilistic stable model in which every consequence of the program is given with a level of certainty.
In this work we explain how we can compute the possibilistic stable models of a possibilistic normal logic program by using available softwares for Answer Set Programming and we describe the main lines of the system that we have developed.

1 Introduction

Answer Set Programming (ASP) [1] is an appropriate formalism to represent various problems issued from Artificial Intelligence and arising when available knowledge is incomplete as in non monotonic reasoning, planning, diagnosis... In ASP, knowledge is encoded by logical rules and solutions are obtained as models. Each model is a minimal set of atoms containing some facts and deductions obtained by applying by default some rules. So, conclusions rely on present and absent data, they form a coherent set of hypotheses and represent a rational view on the world described by the rules. In whole generality there is not a unique set of conclusions but maybe many ones or none.

Possibilistic logic [2] is issued from Zadeh's possibility theory [3]. It offers a framework for representation of states of partial ignorance owing to the use of a dual pair of possibility and necessity measures. Possibilistic logic provides a sound and complete machinery for handling qualitative uncertainty with respect to a semantics expressed by means of possibility distributions which rank-order the possible interpretations [4]. In other words, it deals with uncertainty by means of classical 2-valued interpretations that can be more or less certain.

In [5], we have defined *Possibilistic Stable Model Semantics* that is a new framework dealing with uncertainty in ASP. It has been developed to handle normal logic programs in which each rule is given with a certainty degree and it is based on the introduction into ASP of possibility theory concepts. Figure 1 positions this work within other linked formalisms.

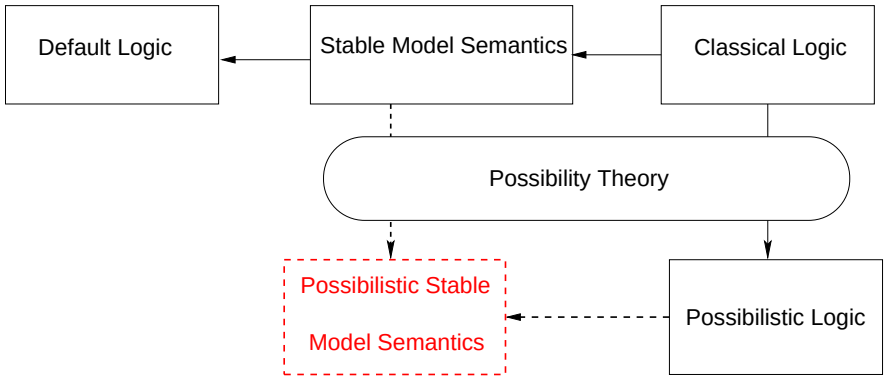

Fig. 1. Possibilistic Stable Model Semantics

Next section 2 recalls some theoretical backgrounds about possibilistic stable model semantics and, in section 3, we expose the main lines of the system that we have developed in order to compute possibilistic stable model semantics.

2 Possibilistic Stable Model Semantics

In [5] we have extended stable model semantics in order to take into account some certainty degrees on rules. For this, we consider given a finite set of atoms $\mathcal{X}$ and a finite, totally ordered set of necessity values $\mathcal{N} \subseteq]0,1]$. Then, a *possibilistic atom* is a pair $p = (x, \alpha) \in \mathcal{X} \times \mathcal{N}$. We denote by $p^* = x$ the classical projection of p and by $n(p) = \alpha$ its necessity degree. These notations can also be extended to a *possibilistic atom set* (p.a.s.) that is a set of possibilistic atoms in which every atom x occurs at most one time.

Definition 1. *Consider $\mathcal{A} = 2^{\mathcal{X}\times\mathcal{N}}$ the finite set of all p.a.s. induced by $\mathcal{X}$ and $\mathcal{N}$. $\forall A, B \in \mathcal{A}$, we define:*

$$\begin{array}{rcl} A \sqcap B & = & \{(x, \min\{\alpha, \beta\}), (x,\alpha) \in A, (x,\beta) \in B\} \\ A \sqcup B & = & \{(x,\alpha) \mid (x,\alpha) \in A, x \notin B^*\} \\ & \cup & \{(x,\beta) \mid x \notin A^*, (x,\beta) \in B\} \\ & \cup & \{(x, \max\{\alpha,\beta\}) \mid (x,\alpha) \in A, (x,\beta) \in B\} \\ A \sqsubseteq B & \iff & \{A^* \subseteq B^*, \text{ and } \forall a,\alpha,\beta, (a,\alpha) \in A \wedge (a,\beta) \in B \Rightarrow \alpha \leq \beta \end{array}$$

Proposition 1. *$\langle \mathcal{A}, \sqsubseteq \rangle$ is a complete lattice.*

A *possibilistic definite logic program* (p.d.l.p.) is a set of *possibilistic rules* of the form:

$$(c \leftarrow a_1, \ \ldots, \ a_m. \ , \ \alpha) \text{ where } m \geq 0, \{c, \ a_1, \ \ldots, \ a_m\} \subseteq \mathcal{X}, \alpha \in \mathcal{N}$$

and a *possibilistic normal logic program* (p.n.l.p.) is a set of possibilistic rules of the form:

$$(c \leftarrow a_1,\ \ldots,\ a_m,\ not\ b_1,\ \ldots,\ not\ b_n.\ ,\ \alpha)\ m \geq 0, n \geq 0$$

for which we just have to precise that $\forall i, b_i \in \mathcal{X}$, all the rest being the same as for a p.d.l.p.

$n(r) = \alpha$ is a necessity degree representing the certainty level of the information described by the rule r. The higher is $n(r)$ the more certain is r. The *classical projection* of a possibilistic rule r is $r^* = c \leftarrow a_1,\ \ldots,\ a_m.$ (or $r^* = c \leftarrow a_1,\ \ldots,\ a_m,\ not\ b_1,\ \ldots,\ not\ b_n.$). If R is a set of possibilistic rules, then $R^* = \{r^* \mid r \in R\}$ is the program obtained from R by forgetting all the necessity values. For a classical rule r we use the following notations (extended to a rule set as usual): the positive prerequisites of r, $body^+(r) = \{a_1,\ \ldots,\ a_n\}$; the negative prerequisites of r, $body^-(r) = \{b_1,\ \ldots,\ b_m\}$; the conclusion of r, $head(r) = c$ and the positive projection of r, $r^+ = head(r) \leftarrow body^+(r)$.

Example 1. The p.n.l.p.

$$P = \left\{ \begin{array}{l} (mary., 1), \\ (stormy\_meeting \leftarrow mary, bob., 1), \\ (stormy\_meeting \leftarrow mary, john., 0.8) \\ (bob \leftarrow not\ peter., 0.8) \\ (peter \leftarrow not\ bob., 0.5) \\ (john., 0.2) \end{array} \right\}$$

represents the following informations.
It is certain that Mary comes.
It is certain that if Mary and Bob come, then the meeting will be stormy.
It is almost sure that if Mary and John come, then the meeting will be stormy.
It is almost sure that Bob comes if Peter does not.
Maybe, Peter comes if Bob does not.
It is very uncertain that John comes.

In view of this example, let us mention that certainty degrees are qualitative. The exact numerical value does not matter. The only important point is their relative values that allow to rank the rules from the more certain to the less certain.

As in the classical case without necessity value, we need to define what is the reduction of a program and what is the *applicability degree of a rule.*

Definition 2. *Let P be a p.n.l.p. and A be an atom set. The* possibilistic reduct *of P wrt. A is the p.d.l.p. $P^A = \{((r^*)^+, n(r)) \mid r \in P, body^-(r^*) \cap A = \emptyset\}$.*

Definition 3. *Let $r = (c \leftarrow a_1,\ \ldots,\ a_m., \alpha)$ be a definite possibilistic rule ($body^-(r^*) = \emptyset$) and A be a p.a.s.,*

- *r is α-applicable in A if $body^+(r^*) = \emptyset$*

- r *is* β*-applicable in* A *if* $\begin{cases} \{(a_1, \alpha_1), \ldots, (a_n, \alpha_n)\} \subseteq A \\ \beta = \min\{\alpha, \alpha_1, \ldots, \alpha_n\} \end{cases}$
- r *is 0-applicable otherwise.*

And, for a given p.d.l.p. P*, an atom set* A *and an atom* x*,*

$$App(P, A, x) = \{r \in P \mid head(r^*) = x, r \text{ is } \nu\text{-applicable in } A, \nu > 0\}$$

The applicability degree of a rule r captures the certainty of the inference process realized when r is applied wrt. a p.a.s. A. If the body of r is empty, then r is applicable with its own certainty. If the body of r is not verified (not satisfied by A), then r is not at all applicable. Otherwise, the applicability level of r depends on the certainty level of the propositions inducing its groundedness and on its own necessity degree. Firstly, the necessity degree of a conjunction (the rule body) is the minimal value of the necessity values of subformulae (atoms) involved in it. Secondly, the certainty of a rule application is the minimal value between the rule certainty and the certainty of the rule body. This approach is similar to the Graded Modus Ponens in possibilistic logic [2].

Definition 4. *Let* P *be a p.d.l.p. and* A *be a possibilistic atom set. The* immediate possibilistic consequence operator ΠT_P *maps a p.a.s. to another one by this way:*

$$\Pi T_P(A) = \left\{ (x, \beta) \;\middle|\; \begin{array}{l} x \in head(P^*), App(P, A, x) \neq \emptyset, \\ \beta = \max\limits_{r \in App(P,A,x)} \{\nu \mid r \text{ is } \nu\text{-applicable in } A\} \end{array} \right\}$$

then the iterated operator ΠT_P^k *is defined by*

$$\Pi T_P^0 = \emptyset \quad \text{and} \quad \Pi T_P^{n+1} = \Pi T_P(\Pi T_P^n), \forall n \geq 0$$

Here, we can remark that if one conclusion is obtained by different rules, its certainty is equal to the greatest certainty with which it is obtained in each case (operator max). Again, it is in accordance with the inference process in possibilistic logic.

Proposition 2. *Let* P *be a p.d.l.p., then* ΠT_P *has a least fix-point* $\sqcup_{n \geq 0} \Pi T_P^n$ *that we called the set of* possibilistic consequences *of* P *and we denote it by* $\Pi Cn(P)$*.*

At this point, we have recalled everything necessary to introduce *possibilistic stable model semantics.*

Definition 5. *Let* P *be a p.n.l.p. and* S *a p.a.s.*
S *is a possibilistic stable model of* P *if and only if* $S = \Pi Cn(P^{(S^*)})$*.*

Example 2. Let P be the p.d.l.p. of example 1, then

$$S_1 = \{(mary, 1), (peter, 0.5), (john, 0.2), (stormy\_meeting, 0.2)\}$$

is a p.s.m. of P since

$$P^{(S_1^*)} = \left\{ \begin{array}{l} (mary., 1), \\ (stormy\_meeting \leftarrow mary, bob., 1) \\ (stormy\_meeting \leftarrow mary, john., 0.8), \\ (peter., 0.5), \\ (john., 0.2), \end{array} \right\}$$

and

$$\begin{array}{l} \Pi T^0_{P^{(S_1^*)}} = \emptyset \\ \Pi T^1_{P^{(S_1^*)}} = \{(mary, 1), (peter, 0.5), (john, 0.2)\} \\ \Pi T^2_{P^{(S_1^*)}} = \{(mary, 1), (peter, 0.5), (john, 0.2), (stormy\_meeting, 0.2)\} \\ \Pi T^k_{P^{(S_1^*)}} = \Pi T^2_{P^{(S_1^*)}}, \forall k > 2 \end{array}$$

Thus, $\Pi Cn(P^{(S_1^*)}) = S_1$ proving that S_1 is a p.s.m. of P.

Moreover,

$$S_2 = \{(mary, 1), (bob, 0.8), (john, 0.2), (stormy\_meeting, 0.8)\}$$

is the second (and last one) p.s.m. of P.

In this example, we can see that our framework allows us to draw conclusions labeled with a certainty degree. We are able to distinguish between the two possibilistic stable models one (S_2) in which the conclusion saying that the meeting will be stormy is more certain than it is in the other one (S_1).

3 Possibilistic Stable Model Computation

3.1 Algorithm

Let us begin by mentioning that, given a p.n.l.p. P there is a one to one mapping between the p.s.m. of P and the stable models of P^* (see [5]). Firstly, this property implies that the complexity class of the existence problem of a p.s.m. is the same that those of the existence problem of a stable model. Secondly, it ensures that, given S a stable model of P^* then $\Pi Cn(P^S)$ is a p.s.m. of P. Moreover, it is easy to establish that the computation of $\Pi Cn(P^S)$ can be done polynomially (see definition 4 and proposition 2). So, the computation of a p.s.m. has not to be significantly harder than the computation of a classical stable model since this last one can be exponential. Despite this high level of complexity, some efficient ASP solvers are available today :

- DLV [6] `http://www.dbai.tuwien.ac.at/proj/dlv`
- Smodels [7] `http://www.tcs.hut.fi/Software/smodels`
- Cmodels [8] `http://www.cs.utexas.edu/users/tag/cmodels.htms`
- Nomore++ `http://www.cs.uni-potsdam.de/wv/nomore++`
- ...

computeAllPSM(in Solv : an ASP solver, P : a p.n.l.p)
begin
  while (Solv(P^*) returns a s.m. S)
    write $\Pi Cn(P^S)$
  endwhile
end

Fig. 2. General algorithm

So, the extension of one of them to compute p.s.m. has to be realizable without losing to much efficiency and the general algorithm for this purpose is sketched in figure 2.

So, starting from this general algorithm we have decided to develop a system in C++ by choosing Smodels, and its associated grounder Lparse, as underlying ASP solver. Our choice has been guided by a compromise taking into account the system performances, the source code availability and our familiarity with the system. Moreover, any ASP system could have been used. But, the possibility to clearly separate the grounding of the rules and the computation of the stable models has been also one reason of our choice.

Let us mention that from now and without loss of generality, we shall use necessity degrees belonging to $\mathbb{N}^*$. As recall in section 2, these values have not a numerical meaning, but only a qualitative and relative significance. So, for convenience we have chosen integer values to encode them. Thus, an input file containing a p.n.l.p. will have to be presented as a sequence of expressions

$$\alpha \quad c \text{ :- } a_1,\ \ldots,\ a_m,\ not\ b_1,\ \ldots,\ not\ b_n.$$

where $\alpha \in \{1, \ldots, 100\}$ if 100 is enough and encodes the full certainty for instance. For the rest of our presentation we shall use this syntax when giving some examples.

3.2 Rule grounding

If we want to use ASP paradigm to solve some large and realistic problems, then using rules with variables to encode problems is absolutely necessary. It means that the input of our system has to be a set of possibilistic rules where variables are allowed as it is shown in the following p.n.l.p..

$$P_1 = \left\{ \begin{array}{l} 50\ b(X) \text{ :- } a(X), not\ c(X). \\ 100\ c(X) \text{ :- } a(X), not\ b(X). \\ 100\ a(1). \\ 20\ a(2). \\ 30\ a(3). \\ 100\ b(2). \\ 80\ d(4). \end{array} \right\}$$

Grounding a program consists in producing all rules that can be obtained by replacing every variable by every constant of the language. At a first sight, it can be achieved very easily, but if we do not take care, the resulting propositional program may contain a huge number of useless rules. In fact, the grounding task is itself a difficult process that has to be done carefully and that is why we preferred to base our work on an already existing system like Lparse.

For the aforementioned program P_1, its classical part is

$$P_1^* = \left\{ \begin{array}{l} b(X) \text{ :- } a(X), not\ c(X). \\ c(X) \text{ :- } a(X), not\ b(X). \\ a(1). \quad a(2). \quad a(3). \\ b(2). \\ d(4). \end{array} \right\}$$

whose grounded version by Lparse is

$$\overline{P^*} = \left\{ \begin{array}{lll} b(1) \text{ :- } not\ c(1). & b(2) \text{ :- } not\ c(2). & b(3) \text{ :- } not\ c(3). \\ c(1) \text{ :- } not\ b(1). & & c(3) \text{ :- } not\ b(3). \\ a(1). & a(2). & a(3). \\ & b(2). & \\ d(4). & & \end{array} \right\}$$

The reader can see, that no useless instantiation, for instance with $X = 4$, is made. Actually, a rule like $b(4)$:- $a(4), not\ c(4)$. is useless since $a(4)$ is impossible to derive with these rules. Moreover, many rule simplifications are made as the suppression of $a(X)$ in the positive bodies of rules and the deletion of the rule $c(2)$:- $a(2), not\ b(2)$. since $b(2)$ is given.

But, for our system, we have an additional task to achieve. It is to keep the necessity degree affected to a rule with variables to every fully instantiated rule generated from this rule. Since Lparse does a lot of simplifications (partial evaluation) it would be very difficult, and sometimes impossible, to reassign the right certainty degrees to the grounded rules. For instance, for next program

$$P_2 = \left\{ \begin{array}{l} 100\ c(X) \text{ :- } a(X). \\ \ 50\ c(X) \text{ :- } b(X). \\ \ 80\ a(1). \\ 100\ b(1). \end{array} \right\}$$

Lparse produces the grounded program

$$\overline{P_2^*} = \{c(1). \quad a(1). \quad b(1)\}$$

for which it is impossible to decide which certainty degree to assign to $c(1)$ by only using this output and the original p.n.l.p. P_2 and not redoing the grounding process. Even if we modified Lparse to not perform these partial evaluations its output would be something like : $\{c(1)$:- $a(1).,\ c(1)$:- $b(1)., a(1)., b(1).\}$. In this case again, to reassign the right degrees would be a time consuming process since it would be something like a unification problem. So, that is why we propose the

following preprocessing that maps every possibilistic rule into a normal one in which a special new atom is inserted to record the certainty degree.

$$preproc(r) = r' s.t. \begin{cases} head(r') = head(r) \\ body^+(r') = body^+(r) \cup \{nu\_\_(\alpha(r))\} \\ body^-(r') = body^-(r) \end{cases}$$

The generalization of this process to a p.n.l.p. P is defined as follows

$$\begin{aligned} Preproc(P) = & \{preproc(r) \mid r \in P\} \\ & \cup \{\sharp external\ nu\_\_(X).\} \\ & \cup \{nu\_\_(\alpha). \mid \alpha \in \mathcal{A}\} \end{aligned}$$

The directive $\sharp external\ nu\_\_(X).$ is a special feature used by Lparse to indicate that expressions $nu\_\_(X)$ are special atoms that could be given in a second step (see Lparse manual for details). For us, the useful point is that Lparse keeps every such atoms in the resulting grounded program. By this way, our initial goal: grounding every rule by keeping the trace of the necessity degree, is achieved as it can be seen in the two programs below.

– output of the preprocessing of P_1

$$Preproc(P_1) = \left\{ \begin{array}{l} b(X) \text{ :- } a(X), not\ c(X), nu\_\_(50). \\ c(X) \text{ :- } a(X), not\ b(X), nu\_\_(100). \\ a(1) \text{ :- } nu\_\_(100). \quad a(2) \text{ :- } nu\_\_(20). \quad a(3) \text{ :- } nu\_\_(30). \\ b(2) \text{ :- } nu\_\_(100). \\ d(4) \text{ :- } nu\_\_(80). \\ \sharp external\ nu\_\_(X). \\ nu\_\_(100). \quad nu\_\_(80). \quad nu\_\_(50). \quad nu\_\_(30). \quad nu\_\_(20). \end{array} \right\}$$

– output of the grounding process done by Lparse

$$\overline{Preproc(P_1)} = \left\{ \begin{array}{l} b(1) \text{ :- } a(1), nu\_\_(50), not\ c(1). \\ b(2) \text{ :- } a(2), nu\_\_(50), not\ c(2). \\ b(3) \text{ :- } a(3), nu\_\_(50), not\ c(3). \\ c(1) \text{ :- } a(1), nu\_\_(100), not\ b(1). \\ c(2) \text{ :- } a(2), nu\_\_(100), not\ b(2). \\ c(3) \text{ :- } a(3), nu\_\_(100), not\ b(3). \\ a(1) \text{ :- } nu\_\_(100). \quad a(2) \text{ :- } nu\_\_(20). \quad a(3) \text{ :- } nu\_\_(30). \\ b(2) \text{ :- } nu\_\_(100). \\ d(4) \text{ :- } nu\_\_(80). \end{array} \right\}$$

To end this description, let us say that we have implemented these techniques in a program named `preprocLparse` that is able to accept a possibilistic logic program that contains strong negations or constant declarations. In fact, these particular points have no influence on our preprocessing and they are managed as usual by Lparse. Thus, the following chain has to be used to realize whole preprocessing

```
preprocLparse inputfile | lparse --true-negation
```

where `--true-negation` is to allow the treatment of strong negation. The output of this process is the input (in the internal Smodels'format) of our program `posSmodels` described in the next section.

3.3 Possibilistic Stable model Computation

Here, we describe how we compute the p.s.m. of a p.n.l.p. by using the grounded normal logic program produced by the preprocessing step described in previous subsection. The whole algorithm, presented in figure 3, is formed by three parts. The first thing to do is to read the output of the first step, that is a normal logic program encoded in the internal Smodels'format, and to rebuild its corresponding possibilistic logic program (see part 1 of the algorithm). By this way, and because of the preprocessing treatment, we are provided with a grounded p.n.l.p. P' where every rule is given with the right certainty degree. The second step is to build for Smodels the non possibilistic corresponding n.l.p. SP' in order to compute its stable models. This is done via the Smodels programming API. Note that this causes a duplication of atoms and rules of the program : one representation for Smodels, and one for posSmodels. But this is the price to pay to stay independent of Smodels, and it is made so that we can directly access from one kind of atoms to the other one. Thus, the computation of p.s.m. is now possible by following the general algorithm of the figure 2 detailed in the third part of figure 3. The whole process implements, the most efficiently as possible, the immediate possibilistic consequence operator ΠT_P introduced in definition 4.

Let us comment some particular points.

- Since we know that every set S is a s.m. of P'^* then we can restrict our attention to the program $PP \subseteq P'^S$. In fact, it is useless to take into account rules with head or positive body not included in S since they can never be involved in the p.s.m. corresponding to S that we want to compute.
- $L(R)$ is a counter initiated with the body length of rule R and decreased each time a new atom belonging to the body of R is added in Res. By this way, at each step the applicable rules are those with $L(R) = 0$.
- An applicable rule with a certainty α and a head x can be dropped from PP when an atom (x, α) is added in Res since this rule (because of the definition of ν-applicability) will never produce a new possibilistic atom (x, β) with $\beta > \alpha$. But, a rule can not be discarded before since it can be used many times to produce the same atom but with an increasing certainty degree at each time as in the next example

$$P_3 = \left\{ \begin{array}{l} 20\ a. \\ 100\ x. \\ 100\ b \text{ :- } a. \\ 100\ a \text{ :- } x. \end{array} \right\} \qquad \begin{array}{l} Res = \emptyset \\ Res = \{(a, 20), (x, 100)\} \\ Res = \{(x, 100), (b, 20), (a, 100)\} \\ Res = \{(x, 100), (a, 100), (b, 100)\} \end{array}$$

posSmodels(in P : a n.l.p, Solv : an ASP solver)
begin
/* read n.l.p. P produced by lparse and construct corresponding p.n.l.p. P' */
$P' \leftarrow \emptyset$
while (read a rule R in P)
$body^+(R) \leftarrow body^+(R) \setminus \{nu\_\_(\alpha)\}$
$P' \leftarrow P' \cup \{(R, \alpha)\}$
endwhile
/* build program SP' in smodels (corresponding to P'^*) */
create program SP' in smodels
for each rule $R \in P'$
create rule R^* in SP'
endfor
/* compute all p.s.m. */
while (Solv(SP') returns a s.m. S) do
$PP \leftarrow \{(r^{*+}, n(r)) \mid r \in P', head(r^*) \in S \text{ and } body^+(r^*) \subseteq S \text{ and } body^-(r^*) \cap S = \emptyset\}$
for each rule $R \in PP$ compute $L(R)$, the length of its (positive) body endfor
$Res \leftarrow \emptyset$ /* the p.s.m. to compute */
repeat
$FixPoint \leftarrow true$
for all rule $R \in PP$
if($L(R) == 0$) then
$deg \leftarrow$ applicability degree of R in Res /* definition 3*/
if ($head(R) \notin Res^*$)then /* a new atom is added */
for each rule $R' s.t.\ head(R) \in body^+(R')$
$L(R') \leftarrow L(R') - 1$
endfor
endif
$Res \leftarrow Res \sqcup \{(head(R), deg)\}$ /* $\sqcup$ of definition 1 */
if (Res has been modified) then $FixPoint \leftarrow false$ endif
if ($n(R) == deg$) then remove R from PP endif /* this rule is now useless */
endif
endfor
until $FixPoint$
write Res
endwhile
end

Fig. 3. Possibilistic stable model computation

in which we can see that rule 100 b :- a. is used two times: firstly with $(a, 20)$ to produce $(b, 20)$, and secondly with $(a, 100)$ (when this atom has been produced by rule 100 a :- x.) to produce $(b, 100)$.

To end, let us shortly present our representation of atoms and rules. An atom A have : a name, an array of rules whose head is this atom A, an array of rules in which A appears in the positive body, and, if it is a possibilistic atom, a

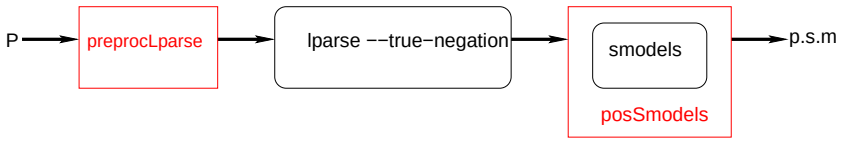

Fig. 4. Process chain.

degree. A rule is naturally represented by its head (an atom), two sets of atoms for the positive and negative bodies, a counter ($L(R)$) for the number of atoms in its positive body that are not already in the p.s.m. actually computed, and its degree. This allows us to access directly to all informations we need. Finally, given an atom of Smodels, we need to access to our own atoms. This task is efficiently achieved by deriving from Smodels *Atom* class a new *AtomExt* class with an additional field for referencing our atom. When creating program for Smodels, we create also atoms of class *AtomExt*. So that, after a stable model S has been computed by Smodels, we can easily recover our own representation of atoms in S.

3.4 Examples and evaluations

Our whole system, sketched in the figure4, has been implemented in C++ and is available at:

`http://www.info.univ-angers.fr/pub/pn/Softwares/PosSmodels`

Its usage is :

```
preprocLparse inputfile | lparse --true-negation | posSmodels k
```

In the figure 5 we summarize some experimental results. In all cases our goal is to estimate the overhead of "possibilistic computation" so that is why we compare the performance of our whole system (as in the chain of the figure 4) with the performance of Lparse and Smodels on the same programs with or without certainty degrees.

- In the left side graph, we have reported the whole CPU times needed to compute all the possibilistic and classical stable models of a program encoding a Hamiltonian cycle problem in a so-called simplex graph. The program contains 30183 rules and 672 atoms and has 948 different models. We have successively computed 1, 100, ..., 900 (classical and possibilistic) stable models and we can see that in the two cases the times linearly grow w.r.t. the number of models to compute.
- In the right side graph, we have reported the ratio of the time consumed by our whole system over the time consumed by Lparse+Smodels to compute one, possibilistic or not, stable model of a program. This program encodes a problem of Hamiltonian path in a complete graph. The number of nodes in the graph growths from 5 to 35. For this last instance, the corresponding program has 3747 atoms and 90684 rules and Smodels needs roughly 100 seconds of CPU time to compute the first stable model. In this graph, we

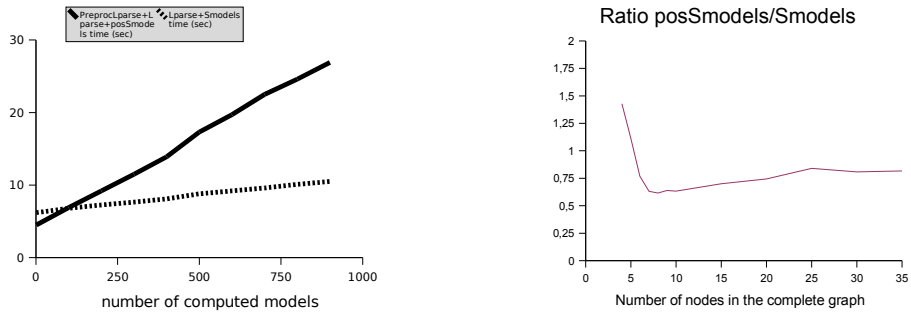

Fig. 5. Performance evaluation

can see that the time consumed to treat the possibilistic part is negligible in front of the time consumed to compute the classical stable models. This is in accordance with our theoretical remark mentioned in subsection 3.1.

Let us end our presentation by illustrating the possible use of possibilistic stable semantics to compare several solutions of a combinatorial problem. For

$$P_{color} = \left\{ \begin{array}{l} 100\ v(1..4). \\ 100\ e(1,X) \text{ :- } v(X), X! = 1. \\ 100\ colored(X) \text{ :- } v(X), red(X). \\ 100\ colored(X) \text{ :- } v(X), green(X). \\ 100\ red(X) \text{ :- } v(X), not\ green(X). \\ \ 50\ green(X) \text{ :- } v(X), not\ red(X). \\ 100\ \text{ :- } e(X,Y), red(X), red(Y). \\ 100\ \text{ :- } e(X,Y), green(X), green(Y). \end{array} \right\}$$

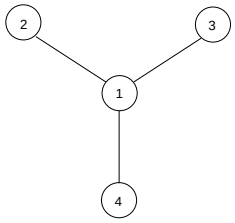

Fig. 6. 2-coloring graph problem

instance, in figure 6 we can see the p.n.l.p. P_{color} that encodes a 2-coloring problem for the graph given on the right side of the same figure. In this particular case, and by interpreting certainty degrees as preference degrees, we have expressed that color *red* is preferred over color *green* to color every vertex. From P_{color}, our system returns the two p.s.m.:

$$S_1 = \left\{ \begin{array}{llll} (v(1),100) & (v(2),100) & (v(3),100) & (v(4),100) \\ (e(1,2),100) & (e(1,3),100) & (e(1,4),100) & \\ (red(2),100) & (red(3),100) & (red(4),100) & (green(1),50) \\ (colored(1),50) & (colored(2),100) & (colored(3),100) & (colored(4),100) \end{array} \right\}$$

and

$$S_2 = \left\{ \begin{array}{llll} (v(1),100) & (v(2),100) & (v(3),100) & (v(4),100) \\ (e(1,2),100) & (e(1,3),100) & (e(1,4),100) & \\ (red(2),100) & (red(3),100) & (red(4),100) & (green(1),50) \\ (colored(1),100) & (colored(2),50) & (colored(3),50) & (colored(4),50) \end{array} \right\}$$

By adding an optimization criterion, for instance maximize $\Sigma\{\alpha \mid (colored(X),\alpha) \in S\}$, we could see that answer S_1 is a better solution than S_2 to color the graph since it uses more the red color than the green one. But, this optimization problem is not solved by our actual system.

4 Conclusion

In this work, we have described the implementation of the theoretical work introduced in [5], that is the stable model semantics. We have based our development on the existing ASP softwares Lparse and Smodels, and our resulting system is able to compute all the possibilistic stable models of a possibilistic normal logic program. As it was expected by some theoretical complexity results, the computation of the certainty degree of each consequence of the program does not increase significantly the whole time computation as soon as the computation of one classical stable model is not direct.

References

1. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9(3-4)** (1991) 363–385
2. Dubois, D., Lang, J., Prade, H.: Possibilistic logic. In Gabbay, D., Hogger, C., Robinson, J., eds.: Handbook of Logic in Artificial Intelligence and Logic Programming. Volume 3. Oxford University Press (1995) 439–513
3. Zadeh, L.: Fuzzy sets as a basis for a theory of possibility. In: Fuzzy Sets and Systems. Volume 1. Elsevier Science (1978) 3–28
4. Dubois, D., Prade, H.: Possibility theory: qualitative and quantitative aspects. In Smets, P., ed.: Handbook of Defeasible Reasoning and Uncertainty Management Systems. Volume 1. Kluwer Academic Press (1998) 169–226
5. Nicolas, P., Garcia, L., Stéphan, I.: Possibilistic stable models. In: International Joint Conference on Artificial Intelligence, Edinburgh, Scotland (2005)
6. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The dlv system for knowledge representation and reasoning. ACM Transactions on Computational Logic (to appear in 2005)
7. Syrjänen, T., Niemelä, I.: The Smodels systems. In: International Conference on Logic Programming and NonMonotonic Reasoning, Vienna, Austria, Springer-Verlag (2001) 434–438
8. Lierler, Y., Maratea, M.: Cmodels-2: Sat-based answer set solver enhanced to non-tight programs. In: International Conference on Logic Programming and NonMonotonic Reasoning. Volume 2923 of LNCS., Springer-Verlag (2004) 346–350