Implementing the p-stable semantics

Angel Marín George,
Claudia Zepeda Cortés

Benemérita Universidad Autónoma de Puebla,
Facultad de Ciencias de la Computación
misterilei@hotmail.com, czepedac@gmail.com

Abstract. In this paper we review some theoretical results about the p-stable semantics, and based on that, we design some algorithms that search for the p-stable models of a normal program. An important point is that some of these algorithms can also be used to compute the stratified minimal models of a normal program.

Key words: non-monotonic reasoning, p-stable, stratified minimal model

1 Introduction

Currently, is a promising approach to model features of commonsense reasoning. In order to formalize NMR the research community has applied monotonic logics. In [7], Gelfond and Lifschitz defined the stable model semantics by means of an easy transformation. The stable semantics has been successfully used in the modeling of non-monotonic reasoning (NMR). Additionally, Pearce presented a characterization of the stable model semantics in terms of a collection of logics in [20]. He proved that a formula is "entailed by a disjunctive program in the stable model semantics if and only if it belongs to every intuitionistically complete and consistent extension of the program formed by adding only negated atoms". He also showed that in place of intuitionistic logic, any proper intermediate logic can be used. The construction used by Pearce is called a weak completion.

In [14], a new semantics for normal programs based on weak completions is defined with a three valued logic called G'_3 logic. The authors call it the P-stable semantics. In [12], the authors define the *p-stable semantics* for disjunctive programs by means of a transformation similar to the one used by Gelfond and Lifschitz in their definition of the stable semantics. The authors also prove that the p-stable semantics for disjunctive programs can be characterized by means of a concept called weak completions and the G'_3 logic, with the same two conditions used by Pearce to characterize the stable semantics of disjunctive programs, that is to say, for normal programs it coincides with the semantics defined in [14]. In fact, a family of paraconsistent logics studied in [12] can be used in this characterization of the p-stable semantics.

In [13], the authors offer an axiomatization of the G'_3 logic along with a soundness and completeness theorem, i.e., every theorem is a tautology and vice-versa. We also remark that the authors of [12] present some results that

give conditions under which the concepts of stable and p-stable models agree. They present a translation of a disjunctive program D into a normal program N, such that the p-stable model semantics of N corresponds to the stable semantics of D when restricted to the common language of the theories. Besides, they show that if the size of the program D is n then the size of the program N is bounded by An^2 for a constant A. The relevance of this last result is that it shows that the p-stable model semantics for normal programs is powerful enough to express any problem that can be expressed with the stable model semantics for disjunctive programs.

One major recent result of the p-stable semantics is in the context of argumentation theory [6], which explores ways to carry out into the theory of computation the mechanisms humans use in argumentation. The three major semantics of argumentation theory (grounded, stable, and preferred) can be characterized in terms of three logic programming semantics: the well founded semantics [22], the stable semantics [7] and the p-stable semantics, respectively in terms of a unique normal logic program P, that is constructed only in terms of the argumentation framework AF [3]. Argumentation theory does not depend on a particular semantics. Then, if we want to obtain the stable semantics of AF, we compute the stable semantics of logic programming over P_{AF}. If, on the other hand, we want to obtain the preferred semantics of AF, we compute the p-stable semantics over P_{AF}. Moreover, if we want to obtain the grounded semantics of AF, we compute the well founded semantics over P_{AF}.

There are also initial work about other two approaches for knowledge representation based on the p-stable semantics: updates, and preferences. In case intelligent agents get new knowledge and this knowledge must be added or updated to their knowledge base, it is important to avoid inconsistencies. An update semantics for update sequences of programs based on p-stable semantics is proposed in [15]. The concept of preferences is considered a vital component of reasoning with real-world knowledge. In [16], the authors introduce preference rules which allow us to specify preferences as an ordering among the possible solutions of a problem. Their approach allow us to express preferences for arbitrary programs. They also define a semantics for those programs. The formalism used to develop their work is p-stable semantics.

It is important to mention that the p-stable semantics, which can be defined in terms of paraconsistent logics, shares several properties with the stable semantics, but is closer to classical logic. For example, the following program $P = \{a \leftarrow \neg b, a \leftarrow b, b \leftarrow a\}$ does not have stable models. However, the set $\{a, b\}$ could be considered the intended model for P in classical logic. In fact, it is the only p-stable model of P.

In [18], a schema for the implementation of the p-stable semantic using two well known open source tools: Lparse and Minisat is described. In [18], a prototype[1] written in Java of a tool based on that schema is also presented. In this paper we present an implementation of the p-stable semantics implementation with aim to improve the implementation presented in [18], considerable

[1] http://cxjepa.googlepages.com/home

effort has been made in the optimization the code and in the design of the algorithms used, which are theoretically more efficient than those used in [18]. As this implementation started recently, we have not yet made definitive tests to warrantee that the solver is error-free, nor we can give conclusive comparative tests between our implementation and that on [18].

In the section 2 the basic concepts about the p-stable semantics are introduced, specially the transformations. In the section 3 are presented the algorithms used to apply the transformations. In the section 4 and 5 is explained how to construct the graph of dependencies of the program and how this graph is used to reduce the search space when looking for the p-stable models.

2 Background

In this section, we define some basic concepts in terms of logic programming semantics, including the definitions of the transformations, which are used to simplify a program.

2.1 Syntax

A signature $\mathcal{L}$ is a finite set of elements that we call atoms. A *literal* is either an atom a, called *positive literal*, or the negation of an atom $\neg a$, called *negative literal*. Given a set of atoms $\{a_1, ..., a_n\}$, we write $\neg\{a_1, ..., a_n\}$ to denote the set of atoms $\{\neg a_1, ..., \neg a_n\}$. A *normal* clause or *normal* rule, r, is a clause of the form

$$a \leftarrow b_1, \ldots, b_n, \neg b_{n+1}, \ldots, \neg b_{n+m}.$$

where a and each of the b_i are atoms for $1 \leq i \leq n+m$, and the commas mean logical conjunction. In a slight abuse of notation we will denote such a clause by the formula $a \leftarrow B^+(r) \cup \neg B^-(r)$ where the set $\{b_1, \ldots, b_n\}$ will be denoted by $B^+(r)$, the set $\{b_{n+1}, \ldots, b_{n+m}\}$ will be denoted by $B^-(r)$, and $B^+(r) \cup B^-(r)$ denoted by $B(r)$. We use $H(r)$ to denote a, called the head of r. We define a *normal program* P, as a finite set of normal clauses. If for a normal clause r, $B(r) = \emptyset$, $H(r)$ is known as a *fact* . We write $\mathcal{L}_P$, to denote the set of atoms that appear in the clauses of P.

2.2 Semantics

From now on, we assume that the reader is familiar with the single notion of *minimal model*[8]. In order to illustrate this basic notion, let P be the normal program $\{a \leftarrow \neg b., \quad b \leftarrow \neg a., \quad a \leftarrow \neg c., \quad c \leftarrow \neg a.\}$. As we can see, P has five models: $\{a\}$, $\{b, c\}$, $\{a, c\}$, $\{a, b\}$, $\{a, b, c\}$; however, P has just two minimal models: $\{b, c\}$, $\{a\}$. We will denote by $MM(P)$ the set of all the minimal models of a given logic program P. Usually MM is called *minimal model semantics.*

Now we give the definition of p-stable model semantics for normal programs.

Definition 1. *[19] Let P be a normal program and M be a set of atoms. We define the reduction of P with respect to M as* $\boldsymbol{RED}(P, M) = \{a \leftarrow B^+ \cup \neg(B^- \cap M) | a \leftarrow B^+ \cup \neg B^- \in P\}$.

Definition 2. *[19] A set of atoms M is a p-stable model of a normal program P iff* $\boldsymbol{RED}(P, M) \models M$*, where the symbol* $\models$ *means logical consequence under classical logic semantics. The set of p-stable models of P is denoted by* $PS(P)$.

We say that two normal programs P and P' are equivalent if and only if they have the same set of p-stable models, this relation is denoted by $P \equiv P'$. An important theorem relating the p-stable and minimal models is the following.

Theorem 1. *[14] Let P be a normal program. If* $M \in PS(P)$ *then* $M \in \boldsymbol{MM}(P)$.

2.3 Transformations preserving equivalence

The main purpose of the transformations presented in this section is to simplify the input normal program, reducing its size. What allows the use of those transformations under the p-stable semantics are the propositions proved in [17, 4] about equivalence of normal programs under these transformations. Let P be a normal program, the definition of the transformations **SUB**, **TAUT**, **RED**$^-$, **RED**$^+$, **SUCC**, **LOOP**, **NAF** and **EQUIV** when applied to P is as follows

1. $\mathbf{SUB}(P) = P \setminus \{r'\} \iff r' \in P, \exists r'' \in P : r' \neq r'', B^-(r'') \subseteq B^-(r'), B^+(r'') \subseteq B^+(r'), H(r) = H(r')$.
2. $\mathbf{TAUT}(P) = P \setminus \{r'\} \iff r' \in P, H(r') \in B^+(r')$.
3. $\mathbf{EQUIV}(P) = (P \setminus \{r'\}) \cup \{r\} \iff r' \in P, H(r') \in B^-(r'), B^-(r) = B^-(r') \setminus \{H(r')\}, B^+(r) = B^+(r'), H(r) = H(r')$.
4. $\mathbf{SUCC}(P) = (P \setminus \{r'\}) \cup \{r\} \iff r' \in P, B^-(r) = B^-(r'), H(r) = H(r'), \exists r'' \in P : B(r'') = \emptyset, H(r'') \in B^+(r'), B^+(r) = B^+(r') \setminus \{H(r'')\}$.
5. $\mathbf{RED}^+(P) = (P \setminus \{r'\}) \cup \{r\} \iff r' \in P, B^+(r) = B^+(r'), H(r) = H(r'), \exists a \in \mathcal{L}_P : a \in B^-(r'), B^-(r) = B^-(r') \setminus \{a\}, \nexists r'' \in P : H(r'') = a$.
6. $\mathbf{RED}^-(P) = P \setminus \{r'\} \iff r' \in P, \exists r'' \in P : B(r'') = \emptyset, H(r'') \in B^-(r')$.
7. $\mathbf{NAF}(P) = P \setminus \{r'\} \iff r' \in P, \exists a \in \mathcal{L}_P : a \in B^+(r'), \nexists r'' \in P : H(r'') = a$.
8. $\mathbf{LOOP}(P) = \{r' : r' \in P, B^+(r') \subseteq M\}$, where M is the unique minimal model of $\mathbf{MM}(\mathbf{POS}(P))$. The definition of $\mathbf{POS}(P)$ is given by
   $\mathbf{POS}(P) = \{r' : B^-(r') = \emptyset, \exists r \in P : B^+(r) = B^+(r'), H(r) = H(r')\}$.

Proposition 1. *[17, 4] Let P be a normal program, and let P′ be the resulting program from the application to P of any of the transformations* ***SUB****,* ***TAUT****,* ***EQUIV****,* ***SUCC****,* ***RED***$^+$*,* ***RED***$^-$*,* ***NAF*** *or* ***LOOP****. Then* $P \equiv P'$.

3 Computing the p-stable models

Now we present the implementation of a p-stable model solver. To find the p-stable models of a program P we can first apply the transformations to P, however the application of the transformations is not absolutely necessary nor sufficient to find the p-stable models of P. In this section we start presenting the implementation of the application of the transformations, and then we give three approaches to find the p-stable models of P, one that follows from the theorem 1, and the others from the theorem 2 which is also presented in this section.

3.1 Implementing the transformations

Given a program P as input to the p-stable solver, we associate to each atom $a \in \mathcal{L}_P$ three sets that are used for the application of the transformations, those sets are initialized as follows,

1. $H(a) = \{r : r \in P, H(r) = a\}$.
2. $P(a) = \{r : a \in P, B^+(r)\}$.
3. $N(a) = \{r : a \in P, B^-(r)\}$.

With the information in those sets it is efficient the application of some transformations, because it is avoided the use of a search algorithm, for example, it will not be necessary to search through all the program P when we require all the rules whose head is a particular atom a. Each atom $a \in \mathcal{L}_P$ also has associated a variable $state(a)$, which can hold one of the following values (we refer to this variable as the state of a): if $state(a) = state_fact$ then a is a fact, if $state(a) = state_no_fact$ then a can not be a fact, if $state_undefined$ then we can not tell yet if a is or can not be a fact.

The application of the transformations **TAUT** and **EQUIV** is easily implemented, and in this paper we do not write the algorithms that apply those transformations. The transformations **SUCC**, **RED**$^+$, **RED**$^-$, **NAF** and a particular case of **SUB** are applied iteratively by the algorithm $transformations_iterated(...)$. This particular version of **SUB** consists in deleting the rules whose head is a fact. For the general case of **SUB** it is necessary to check for set inclusion between the bodies of all pairs of rules with the same head, some of our experimental implementations showed that it was very inefficient and in most cases only a few rules were deleted. The **LOOP** transform was implemented using the Dowling-Gallier algorithm [5] (which finds the unique minimal model of a positive program in linear time) it is presented in the $LOOP(...)$ algorithm. Also a watched variables scheme[9] might be used, this technique is effective in some cases when the deletion of atoms from rules needs to be continuously undone, but at this stage we do not need undo any changes.

For the algorithm $transformations_iterated(...)$ we need two lists (see the algorithm at the end of the paper), they are Lp and Lf, which have to be initialized before calling to $transformations_iterated(...)$. Lp is initialized with the atoms that are facts, then for all $a \in Lp$ it is assigned $state(a) = state_fact$. Lf initialized with the atoms a such that there is no rule with head a, then for all $a \in Lf$ it is assigned $state(a) = state_no_fact$. In $transformations_iterated(...)$ the auxiliary algorithms $remove_rule(r)$ and $remove_atom_from_rule(a, r, B)$ are used. $remove_rule(r)$ removes the rule r and if all the rules with head $H(r)$ have been removed, set $state(H(r)) = state_no_fact$ and add a to Lf. $remove_atom_from_rule(a, r, B)$ removes the atom a from B, where $B = B^+(r)$ or $B = B^-(r)$, if after removing a from B, $|B| = 0$, then set $state(H(r)) = state_fact$ and add a to Lp

In the next example we illustrate how the algorithm $transformations_iterated(...)$ behaves with the program P as input

Example 1. Example of the execution of $transformations_iterated$

$\mathbf{P}$ ={
$r_1 : u \leftarrow not\ v.$
$r_2 : v \leftarrow not\ u.$
$r_3 : u \leftarrow not\ r, not\ x, b.$
$r_4 : x \leftarrow y, r, not\ c.$
$r_5 : y \leftarrow not\ x, z, not\ v.$
$r_6 : x \leftarrow not\ z, t, not\ d.$
$r_7 : z \leftarrow t, v.$
$r_8 : z \leftarrow r, not\ u, not\ x.$
$r_9 : r \leftarrow not\ t, not\ a.$
$r_{10} : t \leftarrow not\ r, b.$
$r_{11} : a \leftarrow not\ a, c.$
$r_{12} : b \leftarrow not\ d.$
$r_{13} : d \leftarrow c, d.$
$r_{14} : c \leftarrow not\ a, b, d, z.$
$r_{15} : c \leftarrow not\ b,\ a, x.$
$r_{16} : b \leftarrow not\ a, not\ u.\}$

$\boldsymbol{\Pi_1}$ ={
$r_1 : u \leftarrow not\ v.$
$r_2 : v \leftarrow not\ u.$
$r_3 : u \leftarrow not\ r, not\ x, b.$
$r_4 : x \leftarrow y, r, not\ c.$
$r_5 : y \leftarrow not\ x, z, not\ v.$
$r_6 : x \leftarrow not\ z, t, not\ d.$
$r_7 : z \leftarrow t, v.$
$r_8 : z \leftarrow r, not\ u, not\ x.$
$r_9 : r \leftarrow not\ t, not\ a.$
$r_{10} : t \leftarrow not\ r, b.$
$r_{11} : a \leftarrow c.$
$r_{12} : b \leftarrow not\ d.$
$r_{14} : c \leftarrow not\ a, b, d, z.$
$r_{15} : c \leftarrow not\ b,\ a, x.$
$r_{16} : b \leftarrow not\ a, not\ u.\}$
$Lp = \{\},\ Lf = \{d\}.$

$\boldsymbol{\Pi_2}$ ={
$r_1 : u \leftarrow not\ v.$
$r_2 : v \leftarrow not\ u.$
$r_3 : u \leftarrow not\ r, not\ x, b.$
$r_4 : x \leftarrow y, r, not\ c.$
$r_5 : y \leftarrow not\ x, z, not\ v.$
$r_6 : x \leftarrow not\ z, t.$
$r_7 : z \leftarrow t, v.$
$r_8 : z \leftarrow r, not\ u, not\ x.$
$r_9 : r \leftarrow not\ t, not\ a.$
$r_{10} : t \leftarrow not\ r, b.$
$r_{11} : a \leftarrow c.$
$r_{12} : b \leftarrow .$
$r_{15} : c \leftarrow not\ b,\ a, x.$
$r_{16} : b \leftarrow not\ a, not\ u.\}$
$Lp = \{b\},\ Lf = \{\}.$

$\boldsymbol{\Pi_3}$ ={
$r_1 : u \leftarrow not\ v.$
$r_2 : v \leftarrow not\ u.$
$r_3 : u \leftarrow not\ r, not\ x.$
$r_4 : x \leftarrow y, r, not\ c.$
$r_5 : y \leftarrow not\ x, z, not\ v.$
$r_6 : x \leftarrow not\ z, t.$
$r_7 : z \leftarrow t, v.$
$r_8 : z \leftarrow r, not\ u, not\ x.$
$r_9 : r \leftarrow not\ t, not\ a.$
$r_{10} : t \leftarrow not\ r.$
$r_{11} : a \leftarrow c.\}$
$Lp = \{\},\ Lf = \{c\}.$

$\boldsymbol{\Pi_4}$ ={
$r_1 : u \leftarrow not\ v.$
$r_2 : v \leftarrow not\ u.$
$r_3 : u \leftarrow not\ r, not\ x.$
$r_4 : x \leftarrow y, r.$
$r_5 : y \leftarrow not\ x, z, not\ v.$
$r_6 : x \leftarrow not\ z, t.$
$r_7 : z \leftarrow t, v.$
$r_8 : z \leftarrow r, not\ u, not\ x.$
$r_9 : r \leftarrow not\ t, not\ a.$
$r_{10} : t \leftarrow not\ r.\}$
$Lp = \{\},\ Lf = \{a\}.$

$\boldsymbol{\Pi_5}$ ={
$r_1 : u \leftarrow not\ v.$
$r_2 : v \leftarrow not\ u.$
$r_3 : u \leftarrow not\ r, not\ x.$
$r_4 : x \leftarrow y, r.$
$r_5 : y \leftarrow not\ x, z, not\ v.$
$r_6 : x \leftarrow not\ z, t.$
$r_7 : z \leftarrow t, v.$
$r_8 : z \leftarrow r, not\ u, not\ x.$
$r_9 : r \leftarrow not\ t.$
$r_{10} : t \leftarrow not\ r.\}$
$Lp = \{\},\ Lf = \{\}.$

Before starting the $transformations_iterated(...)$ algorithm, the **EQUIV** and **TAUT** transformations are applied obtaining Π_1 (this is an optional step), also Lp and Lf have to be initialized. Then we call $transformations_iterated(\Pi_1)$. In the first iteration d is removed from Lf, we apply **NAF** and **RED$^+$** until d do not appear in the program, obtaining Π_2. In the second iteration remove b from Lp then apply **SUB** by removing all the rules whose head is b, then **SUCC** and **RED$^-$** are applied until b do not appear in the program, obtaining Π_3. In the third iteration remove c from Lf, and apply **NAF** and **RED$^+$** until c do not appear in the program obtaining Π_4. In the fourth iteration remove a from Lf, and apply **NAF** and **RED$^+$** until a do not appear in the program obtaining Π_5. At this point Lp and Lf are empty making the algorithm stop.

During the execution of $transformations_iterated(P)$ have been removed all the rules whose head is a fact, including the rules r for which $B(r) = \emptyset$, which are precisely the rules that indicate that $H(r)$ is a fact, it does not represent any problem because we can recover the atoms that are facts just by gathering the atoms whose state is $state_fact$. As the transformations applied preserve equivalence, $P \equiv \Pi_5 \cup \{b \leftarrow\}$, we added the rule $b \leftarrow$ to Π_5 because $state(b) =$

$state_fact$. The state of all the atoms that were added to Lp was set to $state_fact$ (in this case $\{b\}$), and the state of the atoms that were added to Lf was set to $state_no_fact$ ($\{d, a, c\}$), it means that b must be in all the p-stable models of P and $\{d, a, c\}$ can not be in any.

It is not hard to see that after the application of this algorithm we can no longer apply **SUCC**, **RED**$^-$, **RED**$^+$ or **NAF**.

3.2 The graph of dependencies

In most cases the application of the transformations is not enough to find a p-stable model of a normal program, and other techniques are required. One of those techniques is to partition the program into sets of rules called modules. Those modules are created based on its graph of dependencies. Before explaining this technique in detail we give the next definition

Definition 3. *A strongly connected component C of a graph G is a subset of nodes of G. C is a maximal set of nodes(maximal respect to inclusion) such that there is a directed cycle in G that contains all the nodes in C.*

When we have the rule $a_h \leftarrow a_1, a_2, ..., a_m, not\ a_{m+1}, not\ a_{m+2}, ..., not\ a_n.$, a_h is dependent of all the $a_i's$, $i = 1...n$, in a graph this dependencies can be represented with the directed edges $(a_i, a_h), i = 1, ..., n$. The graph of dependencies of a program P represents all the dependencies between the atoms in $\mathcal{L}_P$, given by all the rules of P.

The following definition states the definition of stratification and module of a program P.

Definition 4. *Let P be a normal program which can be partitioned into the disjoint sets of rules $\{P_1, ..., P_n\}$. Let $P_i, P_j \in \{P_1, ..., P_n\}, P_i \neq P_j$, we say that $P_i < P_j$ if $\exists r \in P_j : \exists r' \in P_i : H(r') \in B(r)$, if from this condition we do not conclude that $P_i < P_j$ or $P_j < P_i$ then we can choose to hold whether $P_i < P_j$ or $P_j < P_i$ as long as the following properties hold. For every $X, Y, Z \in \{P_1, ..., P_n\}$, the strict partial order relation properties and the totality property hold:*

1. *$X < X$ is false (this property holds trivially).*
2. *If $X < Y$ then ($Y < X$ is false).*
3. *If ($X < Y$ and $Y < Z$) then $X < Z$.*
4. *$P_1 < ... < P_n$*

then we refer to this partition as the stratification of P, sometimes we will write it as $P = P_0 \cup ... \cup P_n$. And we will refer to $P_i, 1 \leq i \leq n$ as a module of P.

Fortunately we do not have to worry about how to construct this order because there exists a well known algorithm [21](that we implemented in the algorithm $getModules(P)$) which obtains a sequence $C_1, ..., C_n$ of strongly connected components of the graph of dependencies of P, from which we can construct a stratification of P, $P = P_1 \cup ... \cup P_n$, where $P_i = \{r : r \in P, H(r) \in C_i\}, 1 \leq i \leq n$.

Now we define $h(P_i, P)$

Definition 5. *Let P be a normal program, C_i a strongly connected component of the graph of dependencies of P, and P_i the module constructed from C_i. We define $h(P_i, P)$ as the atoms which are represented as nodes in C_i.*

3.3 Computing the p-stable models

The purpose of this section is to show how to compute the p-stable models of a program, three approaches to find the p-stable models of a program P are presented. One way is a direct application of the theorem 1, it consist in computing the minimal models of P and choosing those which satisfy the definition of p-stable model. By theorem 2 the p-stable models of P can be computed by computing the p-stable models of the modules of P after the application of certain transformations.

Theorem 2. *[19] Let P be a normal logic program, and M a model of P with stratification $P = P_1 \cup P_2$, then $\boldsymbol{RED}(P, M) \models M$ iff $\boldsymbol{RED}(P_1, M_1) \models M_1$ and $\boldsymbol{RED}(P'_2, M_2) \models M_2$ with P'_2, M_1, and M_2 defined as follows: $M = M_1 \cup M_2$, $M_1 = h(P_1, P) \cap M$, $M_2 = h(P_2, P) \cap M$, and P'_2 is obtained by transforming P_2 as follows:*

1. *Removing from P_2 the rules r' such that $B^-(r') \cap M_1 \neq \emptyset$ or $B^+(r') \cap (h(P_1, P) \setminus M_1) \neq \emptyset$, obtaining a new program P''_2.*
2. *For every $r \in P''_2$, removing from $B(r)$ the occurrences of the atoms in $h(P_1, P)$, obtaining P'_2.*

In other words M is a p-stable model of P iff M_1 is a p-stable model of P_1 and M_2 is a p-stable model of P'_2, where P'_2 is obtained by removing from P_2 the occurrences of the atoms in $h(P_1, P)$ according to the theorem 2. If P can be stratified as $P = P_1 \cup ... \cup P_n$, $n > 2$, then $P = P_1 \cup Q$ with $Q = P_2 \cup ... \cup P_n$ is also an stratification of P that has only two modules, and then we can apply the theorem 2.

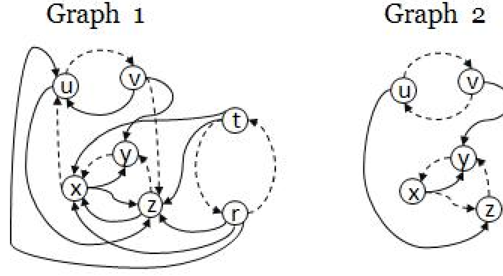
From theorems 1 and 2, three different approaches to compute the p-stable models of a program P are known. The first approach has been implemented in [18]. We propose the other two approaches that reduce the search space respecto to the first approach in many practical examples.

1. From theorem 1, it follows that in order to find the p-stable models of P, we can generate minimal models $M \in \mathbf{MM}(P)$ and accept as p-stable models those for which the consequence test $\mathbf{RED}(P, M) \models M$ is true. To implement the consequence test we take advantage of the fact that $\mathbf{RED}(P, M) \models M \Longleftrightarrow \mathbf{RED}(P, M) \cup \{\neg M\}$ is unsatisfiable, to test if $\mathbf{RED}(P, M) \cup \{\neg M\}$ is unsatisfiable it is used a SAT solver, in this case we used MINISAT[1], the consequence test is implemented in the algorithm *consequence_test*(...)(not presented here for lack of space). Using this approach the search space to find the p-stable models of P is $MM(P)$. The computation of the p-stable models following this approach is implemented in *new_PS*(...).

2. As we have said, if P can be stratified into n modules $P = P_1 \cup ... \cup P_n$, then P can also be stratified into two modules $P = P_1 \cup Q$, where $Q = P_2 \cup ... \cup P_n, n > 2$, and now we can apply the theorem 2 to find the p-stable models of P, this approach is based on the fact that Q' (obtained from Q according to theorem 2) can be stratified as $Q' = Q_2{}' \cup ... \cup Q_n{}'$ where $Q_i{}', 2 \leq i \leq n$ is obtained by removing from P_i the occurrences of $\mathcal{L}_{P_1}$ according to theorem 2. If $n > 3$, to compute the p-stable models of Q', we argument the same way but now instead of P, we have Q' with stratification $Q' = Q_2{}' \cup ... \cup Q_n{}'$. We can proceed this way whenever we want to compute the p-stable models of a program stratified into more than two modules, when it only has two modules, we just apply the theorem 2. The search space to find all the p-stable models of a program P using this approach is in the worst case $MM(P)$, but sometime this search space is bigger than in the third approach. The advantage of this approach over the third approach is that it is not necessary to compute the stratification of a module before computing its p-stable models, we compute only once the stratification of P.

3. Again following the theorem 2, to compute a p-stable model of P with stratification $P = P_1 \cup P_2$(as we have said a program with n modules can also be stratified into two modules), we can compute a p-stable model of P_1 and one p-stable model of P_2', and if P_2' can be stratified into $P_2' = Q_1 \cup Q_2$, we can apply again the theorem and compute the p-stable models of Q_1 and Q_2'. The difference with the second approach is that in the second approach the stratification is computed only once, and in this approach we compute the stratification with $getModules(...)$ each time we want to compute a p-stable model of a module. Using this approach the search space is in many cases considerably smaller than $MM(P)$, this is because the problem of finding the p-stable models of a program is translated into the problem of finding the p-stable models of many small subprograms, the smaller these subprogram, more chances are to reduce the search space. We distinguish between finding the first and the subsequent p-stable models, to find the first p-stable model use the $first_PS_recursive(...)$ algorithm and to find the another p-stable model use $next_PS_recursive(...)$.

We illustrate the second approach by finding the p-stable models of the program Π_5 that resulted from the application of the transformations in the example 1. In the graph 1 we see the graph of dependencies of Π_5, the dotted edges trace maximal cycles, and determine the strongly connected components of Π_5.



Based on the graph of dependencies of Π_5 (graph 1), we see that Π_5 can be stratified into two modules $\Pi_5 = P_1 \cup P_2$. To compute a p-stable model of Π_5 we start by finding a p-stable model of P_1, in this case we found $\{r\}$ to be a p-stable model of P_1. Then compute P_2' by removing the occurrences of the atoms r and t from P_2 according to the theorem 2, in the next figure you can see P_1, P_2 and P_2'

$\boldsymbol{P_2}$ ={
$r_1 : u \leftarrow not\ v.$
$r_2 : v \leftarrow not\ u.$
$r_3 : u \leftarrow not\ r, not\ x.$
$r_4 : x \leftarrow y, r.$
$r_5 : y \leftarrow not\ x, z, not\ v.$
$r_6 : x \leftarrow not\ z, t.$
$r_7 : z \leftarrow t, v.$
$r_8 : z \leftarrow r, not\ u, not\ x.\}$

$\boldsymbol{P_1}$ ={
$r_9 : r \leftarrow not\ t.$
$r_{10} : t \leftarrow not\ r.\}$

$P_2' = \{$
$r_1 : u \leftarrow not\ v.$
$r_2 : v \leftarrow not\ u.$
$r_4 : x \leftarrow y.$
$r_5 : y \leftarrow not\ x, z, not\ v.$
$r_8 : z \leftarrow not\ u, not\ x.\}$

Then find a minimal model of P_2' and do the consequence test, a minimal model of P_2' is $\{v, x\}$, but when doing the consequence test we find that $\mathbf{RED}(P_2', \{v, x\}) \cup \{\neg\{v, x\}\} = \mathbf{RED}(P_2', \{v, x\}) \cup \{\neg v \vee \neg x\}$ is satisfiable, thus $\{v, x\}$ is not a p-stable model of P_2'. We generate another minimal model of P_2', $\{v, z\}$, this time $\mathbf{RED}(P_2', \{v, z\}) \cup \{\neg v \vee \neg z\}$ is unsatisfiable and $\{v, z\}$ is accepted as a p-stable model of P_2'. Merging the p-stable model of P_1 and the p-stable model of P_2' we obtain a p-stable of Π_5, $\{r, v, z\}$. To look for another p-stable model of Π_5, we start by looking for another p-stable model of P_2', we find that $\{u\}$ is a p-stable model of P_2', for instance another p-stable model of Π_5 is $\{r, u\}$. Again we try to generate another p-stable model of P_2', but we note that all the p-stable models of P_2' have been generated (all $M \in MM(P_2')$ have been computed), then we go to the anterior module(P_1) and try to generate another p-stable model of P_1, we find that $\{t\}$ is another p-stable model of P_1, then compute P_2' again :

$\boldsymbol{P_2'}$ ={
$r_1 : u \leftarrow not\ v.$
$r_2 : v \leftarrow not\ u.$
$r_3 : u \leftarrow not\ x.$
$r_5 : y \leftarrow not\ x, z, not\ v.$
$r_6 : x \leftarrow not\ z.$
$r_7 : z \leftarrow v.\}$

We encounter the two p-stable models of P'_2, $\{\{x,u\},\{x,v\}\}$, from which we find another two p-stable models of Π_5, $\{\{t,x,u\},\{t,x,v\}\}$. When trying to generate another p-stable model of P'_2 we can not find any so we go to P_1, and can not find another p-stable model of P_1 so we stop. We end up with the four p-stable models of Π_5, $\mathbf{PS}(\Pi_5) = \{\{t,x,u\},\{t,x,v\},\{r,v,z\},\{r,u\}\}$. Recall form the example 1 that b was found to be a fact after the application of the transformations to P, for instance $\mathbf{PS}(P) = \{\{b,t,x,u\},\{b,t,x,v\},\{b,r,v,z\},\{b,r,u\}\}$.

When using the third approach, we stratify Π_5 into $\Pi_5 = P_1 \cup P_2$ as in the second approach, but after obtaining P'_2, try to stratify P'_2, we can see in the graph of dependencies of P'_2 (graph 2), that P'_2 can be stratified into two modules, so P'_2 is stratified into $P'_2 = Q_1 \cup Q_2$ which can be seen in the next figure

$\boldsymbol{Q_1} = \{$
$r_1 : u \leftarrow not\ v.$
$r_2 : v \leftarrow not\ u.\}$
$Q_2 = \{$
$r_4 : x \leftarrow y.$
$r_5 : y \leftarrow not\ x, z, not\ v.$
$r_8 : z \leftarrow not\ u, not\ x.\}$

$\boldsymbol{Q'_2} = \{$
$r_4 : x \leftarrow y.$
$r_8 : z \leftarrow not\ x.\}$

$\boldsymbol{R_1} = \{\}$
$\boldsymbol{R_2} = \{x \leftarrow y.\}$
$\boldsymbol{R_3} = \{z \leftarrow not\ x.\}$
$\boldsymbol{R'_2} = \{\}$
$\boldsymbol{R'_3} = \{z \leftarrow .\}$

Applying again the theorem 2, first choose a minimal model of Q_1, $\{v\}$ and when doing the consequence test we find that $\{v\}$ is a p-stable model of Q_1. Now obtain Q'_2 (see the anterior figure) which is further partitioned into $Q'_2 = R_1 \cup R_2 \cup R_3$ which are respectively the sets of rules with head y, x and z. Then compute a p-stable model of R_1, the only p-stable model of R_1 is the empty set, R'_2 is also empty and also has an empty p-stable model. R'_3 (obtained by removing from R_3 the occurrences of $\mathcal{L}_{R_1}$ and $\mathcal{L}_{R_2}$) only has an empty rule $z \leftarrow$ and its unique p-stable model is $\{z\}$. Merging the p-stable model of the modules P_1, Q_1, R_1, R'_2 and R'_3 we get a p-stable model of Π_5, it is $\{r\} \cup \{v\} \cup \{\} \cup \{\} \cup \{z\} = \{r,v,z\}$. When trying to generate another p-stable model, first we try to generate another p-stable model of R'_3, it does not have more p-stable models, thus we go back and try on R'_2 but it has neither, then try on R_1 that has no more p-stable models, finally we find that Q_1 has another p-stable model, $\{u\}$, from this point, to find the other p-stable models of Π_5, we proceed as we did when we had $\{v\}$ as a p-stable model of Q_1.

To show the performance of each approach we use some examples in [2]. In the next table we can see the time in seconds it took to find a p-stable model of some of those examples using each of the three approaches. A "-" character means that we stopped the execution after 10 seconds. In the fifth column is the time to find a stable model by smodels. The sixth and seventh columns show the number of atoms and rules of each test program. In the last column is the number of modules in which the program was initially partitioned.

problem	app. 1	app. 2	app. 3	smodels	atoms	rules	modules
blocks world	-	4	.14	.228	2848	28238	600
blocks world variant	-	.15	.18	.188	3227	28817	487
n queens	.01	1.24	.83	.016	149	1539	3
spanning tree	.14	.05	.1	.008	930	1572	276
planning from initial	-	.21	.2	.26	4910	41439	510
weight constraints	.26	.22	.23	.228	887	29662	115
planning in observations	-	-	-	6.172	24639	1098619	22100

It is worth to mention that if we modify the $consequence_test(...)$ such that it always return true, instead of using $tranformations_iterated(...)$ we apply some similar reductions (see [10]), and if all the tautological rules like $a \leftarrow b, not\ b, c$ are removed, then we end up with a stratified minimal model solver, a detailed description of our stratified minimal model semantics solver can be found in [10].

4 Conclusions and future work

Some preliminary tests to the p-stable solver presented in this paper, shows a bad performance respect to a stable semantics solver(smodels[11]), for big programs, due to the backtracking driven by the failure of the consequence test, one of the inconveniences of this implementation is that the search space is very big for some programs, research is needed to develop algorithms that reduce the search space or heuristics that show a good performance for some classes of programs.

Algorithm 1 function $getModules()$

```
ModuleList modules {list of modules}
create the graph of dependencies G
numerate the nodes of G according to the DFS exiting time
transpose the graph(compute G^T)
Do a DFS(depth first search), and in the main loop choose the nodes with bigger
exiting time first.
{Each tree found by the DFS represents a strongly connected component}
for each strongly connected component c do
   create a new module mod
   for each atom a in c do
      add to mod all the rules r that H(r) = a
   end for
   modules.add(mod)
end for
```

Algorithm 2 function $transformations_iterated(Module\ P))$

```
{when P is a module obtained from the stratification of a bigger set of rules, use
{r ∈ P : a ∈ B + (r)} instead of P(a), and use {r ∈ P : a ∈ B − (r)} instead of
N(a)}
global List Lp,Lf
while |Lp ∪ Lf| > 0 do
   if Lp is not empty then
      a = Lp.removeElement()
      {apply a partial version of SUB}
      remove all rules in H(a)
      for all r in P(a) do
         {apply SUCC}
         remove_atom_from_rule(a, r, B^+(r))
      end for
      for all r in N(a) do
         {apply RED^- }
         remove_rule(r)
      end for
   else
      a = Lf.removeElement()
      for all r in P(a) do
         {apply NAF }
         remove_rule(r)
      end for
      for all r in N(a) do
         {apply RED^+ }
         remove_atom_from_rule(a, r, B^-(r))
      end for
   end if
end while
```

References

1.
2. Chitta Baral. Knowledge representation, reasoning and declarative problem solving. http://www.baral.us/bookone/code/smodels.html.
3. J. Carballido, J.C. Nieves, and M. Osorio. Inferring preferred extensions by pstable semantics. *Accepted in Revista Iberomericana de Inteligencia Artificial*, 2008.
4. J. L. Carballido. PhD thesis, BUAP, 2008.
5. W.F. Dowling and J.H. Gallier. Linear time algorithm for testing the satisfiability of propositional horn formulae. *Journal of logic programming*, 1984.
6. Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77(2):321–358, 1995.
7. Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In R. Kowalski and K. Bowen, editors, *5th Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.
8. John W. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, second edition, 1987.
9. Yuting Zhao Lintao Zhang Matthew H. Moskewicz, Conor F. Madigan and Sharad Malik. Chaff. Chaff: Engineering an efficient sat solver. *In DAC 01: Proceedings of the 38th Conference on Design Automation, Las Vegas, NV, USA, June 2001.*, 2001.
10. Angel Marin George Mauricio Osorio, Juan Carlos Nieves. Computing the stratified minimal models semantic(unpublished). 2009.
11. Ilkka Niemela and Patrik Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. volume 1265 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 420–429, 1997.
12. Mauricio Osorio, José Arrazola, and José Luis Carballido. Logical weak completions of paraconsistent logics. *Journal of Logic and Computation*, Published on line on May 9, 2008.
13. Mauricio Osorio and Jose Luis Carballido. Brief study of G'$_3$ logic. *To appear in Journal of Applied Non-Classical Logic*, 18(4):79–103, 2009.
14. Mauricio Osorio, Juan Antonio Navarro, José Arrazola, and Verónica Borja. Logics with common weak completions. *Journal of Logic and Computation*, 16(6):867–890, 2006.
15. Mauricio Osorio and Claudia Zepeda. Update sequences based on minimal generalized pstable models. In *MICAI*, pages 283–293, 2007.
16. Mauricio Osorio and Claudia Zepeda. Pstable theories and preferences. In *Electronic Proceedings of the 18th International Conference on Electronics, Communications, and Computers (CONIELECOMP 2008)*, March, 2008.
17. S. Pascucci and A. Lopez. Syntactic transformation rules under p-stable semantics: theory and implementation. 2009.
18. S. Pascucci and A. Lopez. Implementing p-stable with simplification capabilities. *Submmited to Inteligencia Artificial, Revista Iberoamericana de I.A.*, Spain, 2008.
19. Simone Pascucci. Syntactic properties of normal logic program under pstable semantics: theory and implementation. Master's thesis, March 2009.
20. David Pearce. Stable Inference as Intuitionistic Validity. *Logic Programming*, 38:79–91, 1999.
21. Clifford Stain Thomas H. Cormen, Charles E. Leiserson Ronald L. Rivest.
22. Allen van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38:620–650, 1991.

Algorithm 3 function **LOOP**($Module\ P$)

{for this algorithm it is necessary to associate to each rule r an integer $pCount(r)$ and to each atom a a boolean $ofMM(a)$. }
List prop
initialize $pCount(r) = |B^+(r)|$ for each rule r
initialize $ofMM(a) = false$ for each atom a
initialize $prop = \{a \in \mathcal{L}_P : \exists r \in P : H(r) = a, B^+(r) = \emptyset\}$
{ sets $ofMM(a) = true$ iff $a \in MM(\mathbf{POS}(P))$ }
while $|prop| > 0$ **do**
  $a = prop.remove_element()$
  set $ofMM(a) = true$
  for each r such that $a \in B^+(r)$ **do**
    $pCount(r) = pCount(r) - 1$
    if $pCount(r) = 0$ **then**
      $prop.add(H(r))$
    end if
  end for
end while
{removes rules r for which $ofMM(a) = false$ and $a \in B^+(r)$ }
for each atom a for which $ofMM(a) = false$ **do**
  for each r in $P(a)$ **do**
    $remove_rule(r)$
  end for
end for

Algorithm 4 function bool $new_PS(Module\ P)$

{*solver* is a MINISAT solver associated to P, $IDS(a)$ is the integer that represents the atoms a in *solver*, when $solver = NULL$ no models of P have been computed}
if $solver = NULL$ **then**
  $transformations_iterated(P)$
end if
if the number of atoms in $\mathcal{L}_P$ whose state is $state_undefined$ is >= 2 **then**
  {initialize solver}
  if $solver = NULL$ **then**
    $solver$=$new\ solver()$
    add to *solver* all the rules in P
    set $i = -1$
    for each $a \in \mathcal{L}_P$ for which $state(a) = state_undefined$ **do**
      $solver.newVar()$
      set $IDS(a) = ++i$
    end for
  end if
  {generates minimal models of P, returns true if the consequence test returns true}
  while $solver.solve()$ **do**
    if $consequence_test(solver.model)$ **then**
      {create a clause *lits* with the model generated but negated}
      for each atom a in the solver **do**
        if $solver.model[IDS(a)] = l_True$ **then**
          set $state(a) = state_fact$
          $lits.push(\sim IDS(a))$
        else
          set $state(a) = state_no_fact$
        end if
      end for
      {add the clause to the solver to generate a different minimal model the next iteration}
      if $i > 0$ **then**
        $solver.addClause(lits)$
      else
        add to solver the clauses $\{a\}$ and $\{\neg a\}$ for some atom a
      end if
      return true
    end if
  end while
  {returns false because no p-stable model of P was found}
  solver=NULL
  return false
else
  {there is nothing to solve, the p-stable model was found by $transformations_iterated(P)$}
  if $solver \neq NULL$ **then**
    set $solver = NULL$
    return false
  else
    set $solver = (Solver*)1${just assign a non-NULL value}
    return true
  end if
end if

Algorithm 5 function *bool* $first_PS_recursive(Module\ P)$

{stratify(P) returns true iff P could be stratified into more than two modules }
{If P is stratified into $P = P_1 \cup ... \cup P_n$, as the modules P_i are in the list *submodules*, $head(submodules(P)) = P_1$, $rear(submodules(P)) = P_n$, $next(P_i) = P_{i+1}$ $1 <= i < n$, $back(P_i) = P_{i-1}$ $1 < i <= n$, $next(P_n) = back(P_1) = NULL$}

```
if stratify(P) then
   reductions_iterated(P)
   set Q = head(submodules(P)) {picks the first element of submodules(P)}
   if not new_PS(Q) then
      return false
   end if
   set Q = next(Q)
   while Q ≠ NULL {visits all the modules in submodules(P)} do
      while not first_PS_recursive(Q) do
         {backtracking}
         if not next_PS_recursive(back(Q)) then
            return false
         end if
      end while
      set Q = next(Q)
   end while
else
   if not new_PS(P) then
      return false
   end if
end if
return true
```

Algorithm 6 function *bool* $next_PS_recursive(P)$

```
repeat
   {if P was not divided into more than one modules}
   if |submodules(P)| = 0 then
      if new_PS(P) then
         return true
      end if
   else
      Q = rear(submodules(P))
      if next_PS_recursive(Q) then
         return true
      end if
   end if
   if back(P) = NULL or not next_PS_recursive(back(P)) then
      return false
   end if
   {leaves the module as it was when created}
   reset_component(P)
until first_PS_recursive(P)
return true
```