Classical Methods in Nonmonotonic Reasoning

Yannis Dimopoulos

MPI–I–94–229 August 1994

Classical Methods in Nonmonotonic Reasoning

Yannis Dimopoulos

MPI–I–94–229 August 1994

Author's Address

Yannis Dimopoulos
Max-Planck-Institut für Informatik
Im Stadtwald,
66123 Saarbrücken, Germany,
e-mail: yannis@mpi-sb.mpg.de

Abstract

In this paper we present and compare some classical problem solving methods for computing the stable models of logic programs with negation. In particular linear programming, propositional satisfiability, constraint satisfaction, and graph algorithms are considered. Central to our approach is the representation of the logic program by means of a graph.

Keywords

Logic Programming, Semantics, Graph Theory, Algorithms.

1 Introduction

Over the last years, a large body of research has been devoted to the semantics of logic programs with negation. One of the most prominent proposals is the *stable model* semantics, introduced in [GL88]. This semantics is closely related to Default Logic introduced in [Rei80], in the sense that the stable models of a logic program can be faithfully captured by the extensions of an associated Default Theory.
Unfortunately both Default Logic and stable models semantics, turned out to be intractable, even in very simple cases. Despite their computational hardness, a relatively small effort has been expended on deriving effective ways for reasoning with these formalisms. Recently, some classical problem solving methods for computing the semantics of logic programs and default theories have been proposed ([BED91], [BED92a], [BED92b], [BNNS92], [BNNS93], [DM94], [DMP93], [DT93]). These include linear programming, propositional satisfiability, constraint satisfaction, and graph algorithms. It is now evident that techniques emerging from these problem solving paradigms can be used in Nonmonotonic Reasoning.

In this paper we are concerned with methods for computing the stable model semantics of propositional logic programs. We first investigate the case of normal logic programs, and later we show how these methods can be extended to the case of disjunctive logic programs. Central to our approach is the representation of the logic program by means of a graph. The idea was introduced in [DM94] for the case of disjunction free default theories, and it is only briefly presented here. The set of stable models of a logic program corresponds to a subset of the *kernels* of the associated graph. The graph model gives rise to four different methods for computing the stable models of a normal logic program.

The first alternative is algorithms that explicitly enumerate the kernels of the graph. Apart from the algorithm for the general case, we show that for logic programs with no odd cycles a subset of their stable models can be efficiently computed. The second possibility is to express the graph structure in terms of propositional logic, and use a classical satisfiability algorithm. The models of the resulting theory, which are in direct correspondence with the kernels of the associated graph, are a superset of the stable models of the program at hand. It turns out that this set of models coincides with the set of models of the Clark's completion of the program. As an alternative, a variant of the Davis-Putman method that computes exactly the stable models, is presented.

The third method considered is a formulation the problem as a Linear Programming one, which can be solved with any of the known methods for this class of problems. This method is compared to the linear programming formulation of the problem presented in [BNNS93]. The last possibility is to use some of the constraint satisfaction algorithms. We show that one of these algorithms is closely related to the graph algorithm considered in the paper.

To extend these methods to the case of the disjunctive logic programs we first transform such a program to a normal one, and show that the set of models of the completion of the the latter is a superset of the stable models of the former.

Finally, we present our computational experience with some implementations of satisfiability, linear programming and graph methods.

2 Preliminaries

A disjunctive logic program (DLP) P is a finite set of rules of the form

$$A_1|A_2|\dots|A_m \leftarrow B_1, B_2, \dots, B_k, \neg C_1, \dots, \neg C_n$$

where every A_i, B_j, C_l is an atom. If for every rule of P, $n = 0$ ($k = 0$) holds, we call P *positive* (*negative*) while if $m = 1$ we call P *normal*.

Definition 2.1 *Let P be a logic program and M an interpretation. We define $G_M(P)$ to be the logic program obtained by*

- *Deleting every rule with a negative literal that occurrs in its body and does not belong to M*
- *Deleting all negative literals from the remaining rules*

□

See that $G_M(P)$ is a positive disjunctive logic program (PDLP) and may have many minimal models.

Definition 2.2 *([GL91]) Let P be a logic program and M an interpretation. Then M is a* stable model *for P if one of the minimal models of $G_M(P)$ coincides with M.* □

If P is a DLP, the *reduced form* of P, denote by P^-, is the normal logic program that is obtained form P by replacing every rule $A_1|A_2|\dots|A_m \leftarrow B_1, B_2, \dots, B_k, \neg C_1, \dots, \neg C_n$ of P by the set of rules

$$A_1 \leftarrow B_1, B_2, \dots, B_k, \neg C_1, \dots, \neg C_n, \neg A_2, \dots \neg A_m$$

$$\dots$$

$$A_m \leftarrow B_1, B_2, \dots, B_k, \neg C_1, \dots, \neg C_n, \neg A_1, \dots \neg A_{m-1}$$

One of the early attempts to define the semantics of normal logic programs is Clark's predicate completion ([Cla78]). Given a propositional normal logic program P, its completion, denoted by *comp*(P) is obtained in two steps

- Replace every rule of the form

$$A \leftarrow B_1 \wedge \dots \wedge B_n \wedge \neg C_1 \wedge \dots \neg C_m$$

by the implication

$$A \leftarrow B_1 \wedge \ldots \wedge B_n \wedge \sim C_1 \wedge \ldots \sim C_m$$

where $\sim$ denotes classical negation.

- Let

$$Q \leftarrow Body_1$$
$$\ldots$$
$$Q \leftarrow Body_k$$

be all the clauses with Q in the head. If the clause $'Q \leftarrow'$ belongs to this set then replace this set by Q. Otherwise replace the set of clauses by

$$Q \leftrightarrow Body_1 \vee \ldots \vee Body_k$$

If Q occurs nowhere in the head of the implications add $\sim Q$ in *comp*(P). □

Let $G = (N, E)$ be a directed graph, where N is the set of nodes and E the set of edges. Then for $n_i \in N$, we define $\Gamma^+(n_i) = \{n_j | (n_i, n_j) \in E\}$ and $\Gamma^-(n_i) = \{n_j | (n_j, n_i) \in E\}$. The basic graph theoretic concept used in this paper is that of the kernel of a directed graph.

Definition 2.3 *Let $G = (N, E)$ be a directed graph. A set of nodes $K \subseteq N$ is a* kernel *for G if for every two nodes $n_i, n_j \in K$, the edges (n_i, n_j) and (n_j, n_i) do not belong to E (such a set is called independent), and for every node $n_j \in N-K$ there is a node $n_i \in K$ such that $n_i \in \Gamma^-(n_j)$.* □

3 Graphs for Normal Logic Programs

3.1 The basic Construction

Throughout this and the next section we refer exclusively to normal logic programs. If not otherwise stated, we assume that for every literal p or $\neg p$ which occurs in the body of a rule of a program P, the corresponding positive literal p occurs in the head of some rule in P. We call this class of programs *complete* logic programs. It is easy to see that every logic program can be transformed to a complete one.

We now briefly introduce the way the *rule graph* $G_P = (N, E)$, of a logic program P is constructed (for a detailed discussion see [DM94]). The set of nodes is $N = R \cup A$, $R = \{r_i | r_i \text{ is a rule of } P\}$ and $A = \{a_i | \text{ for each atom } a_i$ that occurs in $P\}$. The set $E = \{(r_i, r_j) | \neg p \in \text{body}(r_j) \text{ and } p \in \text{head}(r_i)\} \cup \{(a_i, r_j) | a_i \in \text{body}(r_j)\} \cup \{(r_i, a_j) | a_j = \text{head}(r_i)\}$.

We can prove that every stable model of P corresponds to a kernel of G_P. Namely, if M is a stable model for a program P, then there is a kernel K for the rule graph G_P such that for every $p \in M^+$ (M^+ denotes the set of positive literals in a set of literals M) there is a node r_i in K such that head$(r_i) = p$. However the converse is not true. This is because of possible circular support between the rules. For example consider the program $P = \{a \leftarrow b, b \leftarrow a\}$. Its rule graph is depicted in figure 1. The graph has two kernels $K_1 = \{r_1, r_2\}$ and $K_2 = \{a, b\}$. See that only the second kernel corresponds to the stable model of P, namely $M = \{\neg a, \neg b\}$.

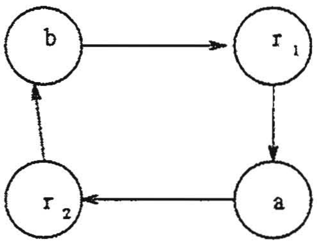

Figure 1

For a kernel K to correspond to a stable model, it must be the case that for the set $K \cap R$, there exists a nonnegative integer function ϕ called *sequential valuation* such that $\phi_{r_i} = max_j(min_k(\phi_{r_{jk}}))$, where $r_{jk}, r_i \in K \cap R$, for every $a_j \in A$ such that $(r_{jk}, a_j), (a_j, r_i) \in E$. Obviously if $\Gamma^-(r_i) \cap A = \emptyset$ then $= \phi_{r_i} = 0$. We call the kernels complying with this property *sequential kernels*. Intuitively speaking, the value ϕ_{r_i} denotes the earliest step at which the rule r_i can be applied.

3.2 Relation to Program Completion

The graph model is equivalent to Clark's completion, in the sense that the kernels of the rule graph are in direct correspondence with the models of the completed program.

Theorem 3.1 *Let P be a normal logic program, K a kernel of G_P, and $S = \{p_i | \exists r_i, p_i \in head(r_i), r_i \in K \cap R\}$. Then the valuation $V(p_i) = true$ iff $p_i \in S$, is a model for comp(P).*

Proof (sketch): Let $V(p_i) = true$ because $p_i \in$ head(r_i), $r_i \in K$. Since $r_i \in K \cap R$ then for every $n_i \in \Gamma^-(r_i)$, $n_i \notin K$ holds. If $n_i \in A$ then $n_i \in \Gamma^+(r_j)$, $r_j \in K \cap R$, which means that all the positive atoms in the body of r_i are assigned the value true in V. On the other hand all the nodes $n_i \in \Gamma^-(r_i) \cap R$ are assigned the value false which means that every negative literal in r_i is assigned the value false in V. Hence the implication $p_i \rightarrow E_1 \vee E_2 \vee \ldots \vee E_n$ is satisfied because of some E_i corresponding to r_i. On the other hand all implications $E_j \rightarrow p_i$ are

trivially satisfied.
Now assume that $V(p_i) = false$. Then all the implications $p_i \rightarrow E_1 \vee E_2 \vee \ldots \vee E_n$ are trivially satisfied. Since p_i is assigned the value false then none of the nodes that have p_i in their heads is included in K. Since K is a dominating set this means that the negation of some literal in the body of each rule which has p_i in its head is included in V, which in turn means that all the implication $E_{ij} \rightarrow p_i$ are satisfied. □

The following theorem states that the convesre also holds.

Theorem 3.2 *Let P be a normal logic program and V a model of comp(P). Then the set $K = \{r_i | r_i \in R, \forall l_i \in body(r_i), V(l_i) = true\} \cup \{a_i | a_i \in A, V(a_i) = false\}$ is a kernel for G_P.*

Proof (sketch): We first show that K is an independent set. Assume that $(r_i, r_j) \in E$, $r_i, r_j \in K \cap R$. Then every literal in the body of r_j is true, including the literal the negation of which occurs in the head of r_i. But since the literals in the body of r_i are all true while the head is false, V is not a model of *comp*(P), a contradiction. Assume now that $(a_j, r_i) \in E$, $r_i \in K \cap R$ and $a_j \in K \cap A$. Then $a_j \in \text{body}(r_i)$ and therefore $V(a_i) = false$ and $V(a_i) = true$ at the same time, a contradiction. Finally assume that $(r_i, a_j) \in E$, $r_i \in K \cap R$ and $a_j \in K \cap A$. Since every literal in the body of r_i is true the head of r_i has to be assigned true as well, while since $a_j \in K \cap A$, $V(a_i) = false$. Hence V is not a model for *comp*(P), again contradiction.
We now show that K is dominating as well. Assume that $r_i \in R$, $r_i \notin K$. Then there must be a literal in the body of r_i having assigned the value false. If this literal is an atom, say a_i, then the node a_i belongs to K and dominates r_i. If the literal is the negation of an atom, then there is a rule the head of that is is true in V, and hence the literals in its body are all true. Then this rule is included in K and dominates r_i.
Let now $a_i \in A$ and $a_i \notin K$. Then $V(a_i) = true$ which means that there is at least one rule r_i which contains a_i in its head and all the literals its body are true. Then $r_i \in K$, and r_i dominates a_i. □

4 Different Methods for Normal Programs

We describe four different methods for computing the stable models of a normal logic program. Whereever the term logic program is used, it refers to normal logic programs. In all the methods we present the input is a representation of the rule graph of the logic program at hand, and the problem to be solved is to compute the kernels of this graph.

4.1 Satisfiability Algorithms

In this section we show how the problem of computing the stable models of a logic program can be expressed in terms of propositional logic. Let $G_P = (N, E)$ be the graph associated with a logic program P. The task is to compute the kernels of G_P. Our purpose is to construct a propositional theory T_P such that its atoms correspond to the nodes of G_P and its satisfying truth assignments to the kernels of G_P. The convention is that every set of nodes assigned the value true in a satisfying truth assignment of T_P must form a kernel for the associated graph and vice versa. If K is a kernel, then every node adjacent to some node $n_i \in K$ must not belong to the kernel, which in terms of propositional logic can be expressed by the implication $n_i \rightarrow \neg n_1 \wedge \ldots \wedge \neg n_k$ (or the set of clauses $\neg n_i \vee \neg n_1, \ldots, \neg n_i \vee \neg n_k$), for all $n_j \in \Gamma^+(n_i)$, $1 \leq j \leq k$. On the other hand if a node n_i does not belong to K then some node $n_j \in \Gamma^-(n_i)$ must be in K. This can again be expressed by the proposition $\neg n_i \rightarrow n_1 \vee \ldots \vee n_k$ for all $n_j \in \Gamma^-(n_i)$, $1 \leq j \leq k$. A clause of this form is said to be *indexed by n_i*. Then the theory T_P is defined to be the smallest set of clauses that subsumes the above set of implications. It is easy to prove every model of T_P will induce a kernel for the associated graph G, and vice-versa. Consequently, we can use a classical propositional satisfiability algorithm to compute the kernels of the graph. The size of T_P is given by the next proposition.

Proposition 4.1 *Let T_P be the propositional theory of a logic program P. The size of T_P, denoted as $\| T_P \|$, is less than $(r_P + 2) \times \| P \| + \| B_L \|$, where r_P is the maximum number of literals in the body of any rule in P, and B_L is the set of propositional atoms occurring in P.*

Proof (sketch): Let a_i be a literal node of the rule graph G_P. Then $\Gamma^-(a_i) = \{r_1, r_2, \ldots, r_k\}$ is the set of rule nodes with edges incoming to a_i. See that the size of $\Gamma^-(a_i)$ is equal to the number of rules that contain a_i in their head, denoted as $numcl(a_i)$. Then T_P contain a set of $numcl(a_i)$ clauses with two negative literals and one clause of size $numcl(a_i) + 1$, which makes a total of $numcl(a_i) + 1$ clauses, so far.

Now consider the set of nodes $M_j = \{k_{jm} | k_{jm} \in \Gamma^-(r_j)\}$, for every $r_j \in \Gamma^-(a_i)$. See that $\| M_j \|$ is equal to the number of literals in the body of the rule r_j, which is smaller or equal to r_P. Hence each node $r_j \in \Gamma^-(a_i)$ leads to at most $r_P + 1$ clauses and since there are $numcl(a_i)$ such nodes then $numcl(a_i) \times (r_P + 1)$ clauses.

If we add to this number the $numcl(a_i) + 1$ encountered in the beginning, we end up with a total of $numcl(a_i) \times (r_P + 2) + 1$ clauses. Then the size of T_P will be the sum $\sum_{a_i \in L} (numcl(a_i) \times (r_P + 2) + 1) = (r_P + 2) \times \sum_{a_i \in L} numcl(a_i) + \| B_L \|$. Since $\sum_{a_i \in L} numcl(a_i) = \| R \|$, we get $\| T_P \| = (r_P + 2) \times \| R \| + \| B_L \|$. □

However, the interesting satisfying truth assignments are only those which correspond to sequential kernels. In order to obtain this subset of models we can employ two different methods.

The first, and most straightforward method, is to compute all models of T_P by using a satisfiability algorithm, and then test them for stability. It is easy to see that this test takes time polynomial in the size of the program.

The second possibility is to enhance a given propositional satisfiability algorithm with metarules that restrict the search to the interesting assignments. In the sequel of this section we present such a set of metarules for the Davis-Putman procedure and show that the method is sound and complete wrt the set of sequential kernels and consequently the set of stable models.

The Davis-Putman algorithm (see [CL73]) is a sound procedure for propositional satisfiability consisting of the following 4 rules, which can be applied iteratively to simplify a set of clauses C:

1. *Tautology Rule*: Delete all clauses in C which are tautologies.

2. *One-literal Rule*: If there is a unit clause L in C, then assign the value true to L and delete all clauses in C containing L. If the resulting set of clauses, C', is empty then the C is satisfiable. Otherwise delete form the clauses in C' all occurrences of $\neg L$.

3. *Pure-literal Rule*: If a literal L occurs in a clause of C and the literal $\neg L$ does not occur in any clause of C, then assign L the value true and delete all clauses containing L.

4. *Splitting Rule*: If the set C can be expressed in the form

   $$(A_1 \vee L) \wedge (A_2 \vee L) \wedge \ldots \wedge (A_m \vee L) \wedge (B_1 \vee \neg L) \wedge (B_2 \vee \neg L) \wedge \ldots \wedge (B_n \vee \neg L) \wedge R$$

   where R does not contain any of the L or $\neg L$, then split the search space into two, the first being $A_1 \wedge A_2 \wedge \ldots A_m \wedge R$ and the second $B_1 \wedge B_2 \wedge \ldots B_m \wedge R$. The first branch corresponds to the value assignment true to L while the second to the assignment false.

If the above rules are applied iteratively, starting from a set of clauses C, and at each point where the splitting rule is used one of the two possible assignments to L is chosen then, if the empty set is derived the corresponding value assignment obtained is a model of C. If at some point a contradiction is derived the algorithm backtracks to an earlier splitting point and considers a different assignment. If all of the search paths fail then the set of clauses C is unsatisfiable. Since only some of the models of a given set of clauses T_P correspond to the stable models of P we need to restrict the search space by augmenting Davis-Putman method with metarules.

Given a logic program P we can easily obtain its positive counterpart P^+, by deleting all negative literals from the body of the rules. Then by constructing the rule graph of P^+, denoted as G_{P^+}, we can identify the strongly connected components of G_{P^+} and the associated directed acyclic graph $G^S_{P^+}$. The graph $G^S_{P^+}$

induces an ordering on the components, such that $O(C_i) = max\{O(C_j)|C_j \in \Gamma^-(C_i)\} + 1$ (all nodes with in-degree 0 are assigned the value 1). We call this value *depth* of the component. Hence, every atom can be considered as parameterized by the depth of the component to which it belongs.

On the other hand Davis-Putman method is supplied with a priority among the four rules it consists of. The first three rules have equal priority which is higher than the priority of the splitting rule. This means that the splitting rule will be applied if none of the other three rules can be applied. Furthermore, the splitting rule is applied to a literal in the current top component C_i induced by the depth ordering. A particular literal from C_i is chosen as follows. If there exists a rule node $r_i \in R \cup C_i$, such that the clause indexed by r_i does not contain any literal associated to a node $a_i \in A$, then the splitting is performed upon r_i. If no such atom exists, then an atom $r_j \in R \cup C_i$ is chosen and is assigned the value false, while the branch which assigns true to r_j is omitted.

Given a set of clauses C, at each point the above method imposes a truth assignment to a subset of atoms occuring in C. This partial value assignment is called a *MDP-valuation* of C. A complete MDP-valuation for C that leads to the empty set, and consequently is a model for C, is called a *MDP-model* for C.

Example 4.2 *Consider the following logic program P*

$c \leftarrow \neg d$ (r_1) $\qquad$ $d \leftarrow \neg c$ (r_2)

$a \leftarrow d$ (r_3) $\qquad$ $b \leftarrow a$ (r_4)

$a \leftarrow b$ (r_5)

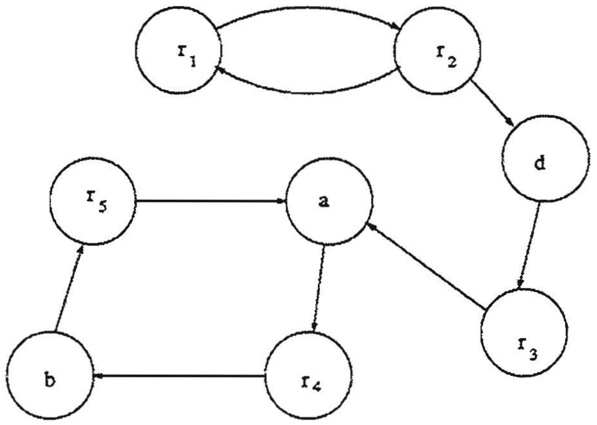

Figure 2

The rule graph of this program is depicted in Figure 2. The associated propositional theory (the value of the atom c is not considered) is

$\neg r_1 \vee \neg r_2$, $\neg r_2 \vee \neg d$,
$\neg d \vee \neg r_3$, $\neg r_3 \vee \neg a$, $\neg a \vee \neg r_4$,
$\neg r_4 \vee \neg b$, $\neg b \vee \neg r_5$, $\neg r_5 \vee \neg a$,
$r_1 \vee r_2$, $d \vee r_2$,
$r_3 \vee d$, $a \vee r_3 \vee r_5$, $r_4 \vee a$,
$b \vee r_4$, $r_5 \vee b$.
A possible ordering of the components is $(C_2, C_1, C_3, C_4, C_5)$, where $C_1 = \{r_1\}$, $C_2 = \{r_2\}$, $C_3 = \{d\}$, $C_4 = \{r_3\}$, $C_5 = \{a, r_4, b, r_5\}$. First the value true is assigned to r_2 which leads to the MDP-model $\{r_2, \neg r_1, \neg d, r_3, \neg a, r_4, \neg b, r_5\}$ with no further use of the splitting rule. When the algorithm backtracks to the splitting point it will assign the value false to the literal r_2. Then, by using the first three rules, Davis-Putman method assigns the values $\{\neg r_2, r_1, d, \neg r_3\}$ which leads to the simplified theory
$\neg a \vee \neg r_4$, $\neg r_4 \vee \neg b$,
$\neg b \vee \neg r_5$, $\neg r_5 \vee \neg a$,
$a \vee r_5$, $r_4 \vee a$,
$b \vee r_4$, $r_5 \vee b$.
At this point the atom r_4 is chosen by the splitting rule, which after been assigned the value false leads to the MDP-model $\{\neg r_2, r_1, d, \neg r_3, \neg r_4, b, \neg r_5, a\}$ and the procedure terminates. The two stable models of P are $M_1 = \{a, b, \neg c, d\}$, and $M_2 = \{\neg a, \neg b, c, \neg d\}$. $\square$

The following theorem demonstrates the correctness and completeness of the method wrt the sequential kernels, and consequently wrt the stable models.

Theorem 4.3 *Let P a logic program, $G_P = (N, E)$ its rule graph, and T_P the associated propositional theory. Then a valuation V to the literals is a MDP-model iff V induces a sequential kernel for G_P.*

Proof (sketch): ($\Rightarrow$) It is easy to see that every MDP-model V induces a kernel $K = \{n_i | n_i \in N, V(n_i) = \mathit{true}\}$, for G_P. We will show, inductively on the depth of G^S_{P+} that this kernel is sequential.
Let r_i be a literal which is assigned the value true in V and belongs to a strongly connected component C_i of depth 1 in G^S_{P+}. See that it must be the case that $|C_i| = 1$, otherwise the splitting rule would be applicable, assigning to every such node the value false. In turn, $|C_i| = 1$ means that there are no positive literals in the body of r_i, hence $\phi_{r_i} = 0$.
Assume that there exists a nonnegative function ϕ which assigns a value to all nodes $r_i \in R$, for which $V(r_i) = \mathit{true}$ and r_i belong to a strongly connected component of depth less or equal to k.
We will show that the literals $r_i \in R$ for which $V(r_i) = \mathit{true}$ and r_i belong to a strongly connected component of depth $k+1$, can be assigned an appropriate ϕ-function value.
Let $r_i \in C_i$ be such a literal, where C_i is a component in depth $k+1$. If $|C_i| = 1$ then all the nodes $a_i \in \Gamma^-(r_i) \cap A$ are assigned the value false and all the nodes

$r_k \in \Gamma^-(a_i)$ belong to components of depth less than $k+1$. Hence a value ϕ_{r_i} can be assigned according to the definition of ϕ.
We consider now the case $|C_i| > 1$. Let R_i be the set defined $R_i = \{r | r \in C_i \cap R, V(r) = true\}$. Assume that for every $r_j \in R_i$, there is $a_j \in A$, such that $(a_j, r_j) \in E$ and for each $r_m \in \Gamma^-(a_i) \cap R$, $r_m \in C_m$, $V(r_m) = false$, where r_m belongs to a component C_m of depth less than $k+1$. Then the splitting rule would be applied assigning to all nodes r_j the value false which contradicts the assumption $V(r_i) = true$. Hence there must be a set of rule literals $R_{i1} \subseteq R_i$ the members of which can be assigned a ϕ function value. If $r_i \in R_{i1}$ we are done. Otherwise we can apply again the same argument for the nodes $R_i - R_{i1}$. This can be iterated until r_i is encountered.

($\Leftarrow$) Let K be a sequential kernel of G_P. We show, inductively on the depth of G^S_{P+}, that there is a MDP-model V, such that $V(m) = true$ iff $m \in K$.
Let $C_{11}, C_{12}, \ldots, C_{1m}$ be the components in depth 1. Define $R_1 = R \cap (C_{11} \cup C_{12} \cup \ldots \cup C_{1m})$. First consider the case $r_i \in R_1$ and $r_i \in C_{1i}$, where $|C_{1i}| = 1$. If $\Gamma^-(r_i) = \emptyset$ then $r_i \in K$, while in T_P there is the unit clause r_i, which means that $V(r_i) = true$. If $\Gamma^-(r_i) \neq \emptyset$, then in the clause indexed by r_i in T_P, no atom $a_i \in A$ occurs. Hence any of the values $V(r_i) = true$, $V(r_i) = false$ can be assigned by Davis-Putman to each of these literals. Hence for these literals $V(r_i) = true$ iff $r_i \in K$ is a MDP-valuation. On the other hand in every component C_{1j}, with $|C_{1j}| > 1$ all nodes $a_j \in A \cup C_{1j}$, belong to K, while none of the nodes $r_j \in R \cup C_{1j}$ belongs to K. For these literals Davis-Putman method assign $V(a_j) = true$ and $V(r_j) = false$. Hence for every literal k_i in depth 1, $V(k_i) = true$ iff $k_i \in K$ is a MDP-valuation.
Assume that for all literals m in components of depth less than k, $V(m) = true$ iff $m \in K$ is a MDP-valuation.
Let $C_{k1}, C_{k2}, \ldots, C_{km}$ be the components in depth k. For the literals m_i such that $C_{ki} = \{m_i\}$ see that a value $V(m_i) = true$ if $m_i \in K$ and $V(m_i) = false$ if $m_i \notin K$.
Let C_{ki} be a component and $|C_{ki}| > 1$. Then there are two possible cases.
1. If for every $a_i \in C_{ki} \cap A$ there is an edge (r_i, a_i), $r_i \in C_{ki}$, then all of the nodes $a_i \in C_{ki} \cap A$ will belong to K, while none of the nodes $r_i \in C_{ki} \cap R$ will belong to K. See that in this case, $V(a_i) = true$ and $V(r_i) = false$ is a MDP-valuation.
2. If C_{ki} is a component, $|C_{ki}| > 1$, and the conditions of 1 above do not hold, then define $L^1_{ki} = \{a_i | a_i \in C_{ki} \cap A, \forall (r_i, a_i) \in E, r_i \notin C_{ki}\}$. Then for every $a_i \in L^1_{ki}$, the assignment $V(a_i) = true$ if $a_i \in K$ and $V(a_i) = false$ if $a_i \notin K$, is a MDP-valuation. Now define $R^1_{ki} = \{r_i | r_i \in C_{ki} \cap R, \forall (a_i, r_i) \in E, a_i \in L^1_{ki}\}$. Then if $r_i \in R^1_{ki}$ and there exists $a_i \in \Gamma^-(r_i) \cap K$ then $r_i \notin K$ and $V(r_i) = false$ as well. On the other hand if for some $r_i \in R^1_{ki}$ the previous condition does not hold, still the clause indexed by r_i, at this point, does not contain any literal from A hence any value can be assigned any value depending on whether $r_i \in K$ or not.

Iterating the arguments outlined above for the nodes in $C_{ki} - (L^1_{ki} \cup R^1_{ki})$ it can be proved that for every literal k_i in depth k, $V(k_i) = true$ iff $k_i \in K$ is a MDP-valuation. This concludes the proof that $V(k_i) = true$ iff $k_i \in K$ is a MDP-valuation for T_P.
Finally, since K is a kernel this MDP-valuation satisfies all the clauses in T_P, it is a MDP-model. □

4.2 Linear Programming Algorithms

4.2.1 A Straightforward Translation of the Problem

Given that the kernel problem can be formulated in terms of propositional logic, the corresponding satisfiability problem can be solved by an integer linear programming algorithm. Any satisfiability problem can be express as a integer linear programming one where, to every literal A in the set of clauses a binary variable X_A is associated, the objective function is empty, while the set of constraints is the following:

1. For every clause $p_1 \vee p_2 \ldots \vee p_n \vee \neg k_1 \vee \neg k_2 \ldots \vee \neg k_m$, where p_i, k_j atoms, we add the constraint $\sum_{i=1}^{n}(1 - X_{p_i}) - \sum_{i=1}^{m} X_{k_i} \geq 1$.
2. For every variable X_A we add the constraints that it is a binary variable, that is, X_A can be only assigned the values 0 and 1.

Every solution for this set of constraints is a model for the associated set of clauses.

Formulating T_P in linear programming takes $(r_P + 2)\times \parallel P \parallel + \parallel B_L \parallel$ constraints to represent each clause in T_P, plus $2\times \parallel P \parallel + 2\times \parallel B_L \parallel$ more to express the fact that these are 0-1 variables. This gives us a total of $(r_P+4)\times \parallel P \parallel + 3\times \parallel B_L \parallel$, and leads to a simplex tableau of size $((r_P+4)\times \parallel P \parallel + 3\times \parallel B_L \parallel) \times (\parallel P \parallel + \parallel B_L \parallel)$. We denote the set of constraints formalizing T_P, as $ct(P)$. Again, not all of these assignments correspond to stable models. The following proposition determines a subset of models of T_P that include all stable models of P.

Proposition 4.4 *Let K be a sequential kernel for the rule graph G_P of a program P. Then K is a minimal kernel wrt the set of the rule nodes it contains.*

Proof (sketch): It can be proved inductively on the depth of the G^S_{P+} graph. □

Since the sequential kernels are in direct correspondence with the stable models, the models which are minimal wrt to the literals associated to rules, will include the stable models of the program P. Hence the objective function in this case becomes the minimization of the sum $\sum_{A \in R} X_A$. This is not exactly what we need, since only the assignments with the smallest number of variables from R will be computed, i.e. the assignments with the *minimal cardinality*.

Our aim is to compute all the minimal assignments wrt set inclusion. Hence, similar to what is proposed in [BNNS93], we have to iterate this process adding at each step the constraint $\sum_{B\in M} X_B \leq (k-1)$, where M is the set of variables from R which have been assigned the value true during the last iteration, and k is the cardinality of M. This procedure will compute all the minimal models in order of non-decreasing cardinality.

However, the converse of proposition 4.4 does not always hold, and the minimal models computed by this procedure must be tested for stability.

4.2.2 An Alternative Approach

In this section we discuss the approach presented in [BNNS93], which is based on the method for computing the minimal models of logic programs presented in [BNNS92]. The authors present three algorithms for computing the stable models. We discuss here only two of them, which are briefly introduced in the sequel.

Given a rule R, $A \leftarrow B_1 \wedge \ldots B_n \wedge \neg C_1 \ldots \neg C_m$, $if(R)$ denotes the constraint $X_A \geq 1 - \sum_{i=1}^{n}(1-X_{B_i}) - \sum_{j=1}^{m} X_{C_j}$. If P is a logic program then $if(P)$ denotes the set $\{if(R) | R \text{ is a rule in } P\} \cup \{0 \leq X_K \leq 1 | \text{ for every atom } K\}$.

Let now *comp*(P) be the completion of a program P and C a formula in P. Then if C is of the form $\neg A$, the constraint version of C, denoted by $lc(C)$, is $X_A = 0$. On the other hand if C is of the form $A \leftrightarrow E_1 \vee \ldots \vee E_k$ where $E_i \equiv L_{i,1} \wedge \ldots \wedge L_{i,m_i}$, then the constraint version $lc(C)$ of C is given by the set of constraints $\{if(A \leftarrow E_i)\}$ together with the constraint $X_A \leq \sum_{i=1}^{k} \prod_{j=1}^{m_i} X_{L_{i,j}}$. Since the second set of constraints is not linear, this set is linearized, resulting in the constraint version of the *comp*(P) which is denoted by *lccomp*(P).

Given this constraint representation of the problem the task then is to iteratively optimize the function $\sum X_A$, where A is an atom, subject to the constraints $if(P) \cup AC$ or *lccomp*$(P) \cup AC$, where AC, initialized to the empty set, is a set of additional constraints added at each iteration. These constraints are identical to those described in the previous section, and are of the form $\sum_{B\in M} X_B \leq (k-1)$ where M is an optimal solution computed during the last iteration.

These two algorithms compute the minimal models of $if(P)$ and *lccomp*(P) respectively, in order of non-decreasing cardinality. The models are then tested for stability.

Theorems 3.1 and 3.2 ensure that the satisfying assignment of *lccomp*(P) coincide with those satisfying *ct*(P). We now prove that the optimal solutions of the two corresponding minimization problems also coincide.

Proposition 4.5 *Let P be a logic program, T_P the associated propositional theory, and comp(P) its completion. Then if S_1 is a model of T_P minimal in the set of literals associated with rules then $S_2 = \{p_i | \exists r_i \in S_1 \cap R, p_i \in head(r_i)\}$ is a minimal model for comp(P).*

Proof (sketch): Let S_1 be a model of T_P which is minimal in the set of literals associated to rules. Then by theorem 3.1, the set S_2 defined above is a model for *comp*(P). Let now S'_2 be another model of *comp*(P), such that $S'_2 \subset S_2$. Then by theorem 3.2 there exists a kernel for G_P (and consequently a model for T_P), say S'_1, that includes more literal nodes than S_1. This means that for the associated models $S'_1 \cap R \subset S_1 \cap R$. Hence S_1 is not minimal in the set of literals associated with rules, which is a contradiction. □

Proposition 4.6 *Let P be a logic program, T_P the associated propositional theory, and comp(P) its completion. Then every minimal model of comp(P) induces a model for T_P which is minimal in the literals associated with rules.*

Proof (sketch): Let S_1 be a minimal model for *comp*(P). Then by theorem 3.2 there is a corresponding model for T_P, say S_2. Assume that S_2 is not minimal in the set of literals associated with rules, because S'_2 is a model of T_P and $S'_2 \cap R \subset S_2 \cap R$. By theorem 3.1 there is a model for *comp*(P) associated with S'_2, say S'_1, such that $S'_1 \subset S_1$. Then S_1 is not minimal for *comp*(P), a contradiction. □

Hence, minimizing either *ct*(P) wrt the literals corresponding to rules, or *lccomp*(P) wrt to the atoms, leads to the same set of minimal models, namely the minimal models of *comp*(P). However *ct*(P) offers an advantage. Namely, the size of the problem to be optimized is smaller. As it is show in [BNNS93], the size of the simplex tableau for *lccomp*(P) is, in the worst case, $((r_P + 3) \times \parallel P \parallel + 2 \times \parallel B_L \parallel) \times ((r_P + 6) \times \parallel P \parallel + 3 \times \parallel B_L \parallel)$, which is obviously larger than the size of the *ct*(P) problem.

Furthermore, in the same paper the proposed algorithms are compared, by using some benchmark databases, and it is argued that computing the minimal models of $if(P)$ overall performs better than computing the minimal models of *comp*(P). In the next section we show that computing the minimal models of $if(P)$, in the case of negative logic programs, amounts to computing the minimal dominating sets of the rule graph, the number of which can be exponential in the size of the graph (the size of the program).

4.2.3 A graph-theoretic characterization of minimal models

In this section we investigate the case of *negative logic programs*, that is, programs with rules that contain only negative literals in their body. If P is a negative program, then $if(P)$ is a set of implications of the form $\neg C_1 \wedge \neg C_2 \wedge \ldots \wedge \neg C_m \rightarrow A$, while G_P contains only rule nodes. In this section we drop the restriction that the programs are complete, or in other words, that for every negative literal in the body of a rule, its negation occurs in the head of some rule.

A *dominating set* for a directed graph $G = (N, E)$, is a set of nodes $D \subseteq N$, such that for every node $n_j \notin D$, there is a node $n_i \in D$, such that $(n_i, n_j) \in E$.

Proposition 4.7 *If P is a negative program then, every dominating set for G_P induces a model for $if(P)$.*

Proof (sketch): Let D be a dominating set for G_P and $M = \{p_i | p_i$ is the consequent of an implication $r_i \in if(P), r_i \in D\}$. See that all implications for which their consequents belong to M are satisfied. On the other hand every implication r_i with a consequent that does not belong to M is dominated by an implication of D, which means that at least one of its antecedents is false, hence r_i is also satisfied. □

It turns out that the minimal dominating sets correspond to minimal models.

Proposition 4.8 *If P is a negative program then, every minimal dominating set for G_P induces a minimal model for $if(P)$.*

Proof (sketch): Let D be a minimal dominating set for G_P and $M = \{p_i | p_i$ is the consequent of an implication $r_i \in if(P), r_i \in D\}$ its associated model for $if(P)$. For any node $r_i \in D$ there are two possibilities. The first is that r_i is not dominated by any other node in D. In this case the set $M - \{p_i\}$, where p_i is the consequent of the implication r_i, is not a model for $if(P)$ since the implication r_i is not satisfied. The second possibility is that r_i dominates a node r_j which is not dominated by any other node in D. Then the set $M - \{p_i\}$, where p_i is the consequent of the implication r_i, does not satisfy the implication r_j. Hence M is a minimal model. □

The converse of proposition 4.7 does not hold in general. However, in the case of complete negative program every model of P corresponds to a dominating set.

Proposition 4.9 *Let P be a complete negative program. Then every model of P induces a dominating set for G_P.*

Proof (sketch): Let M be a model for P. We will prove that $D = \{r_i | r_i$ is an implication and the consequent of r_i belongs to $M\}$ is a dominating set. If the rule associated with an implication r_j does not belong to D, then one of the negative literals in the antecedent of r_j is assigned the value false, and hence the corresponding atom is true. Given that P is complete there must be an implication $r_i \in D$ such that the edge (r_i, r_j) belongs to G_P. Therefore D is a dominating set. □

Any method based on the computation of the minimal models of $if(P)$ is confronted with the fact that the number of the minimal dominating sets can be exponential in the number of the nodes of the graph as the following example demonstrates.

Example 4.10 *Consider n disconnected copies of the graph with $4n$ nodes, shown in Figure 3. The associated negative logic program is $P = \{q_{i1} \leftarrow \neg p_{i4}, q_{i2} \leftarrow \neg p_{i1}, q_{i3} \leftarrow \neg p_{i2}, q_{i4} \leftarrow \neg p_{i2} \wedge \neg p_{i3}\}$, for $1 \leq i \leq n$.*

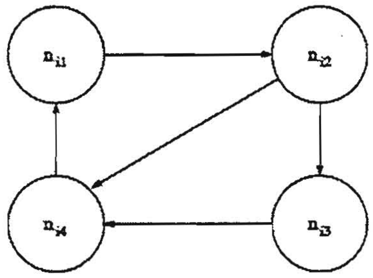

Figure 3

Each of these disconnected parts has 3 minimal dominating sets, namely $D_1 = \{n_{i1}, n_{i2}\}$, $D_2 = \{n_{i1}, n_{i3}\}$ *and* $D_3 = \{n_{i2}, n_{i4}\}$, *which correspond to the minimal models* $M_1 = \{p_{i1}, p_{i2}, \neg p_{i3}, \neg p_{i4}\}$, $M_2 = \{p_{i1}, p_{i3}, \neg p_{i2}, \neg p_{i4}\}$ *and* $M_3 = \{p_{i2}, p_{i4}, \neg p_{i3}, \neg p_{i1}\}$, *respectively. Choosing one of them for every* i, *leads to a total of* 3^n *different minimal dominating sets for* G_P *and hence to* 3^n *minimal models for a program consisting of* $4n$ *rules. Furthermore* P *has just one stable model, namely* $M = \{p_{i1}, p_{i3}, \neg p_{i2}, \neg p_{i4}\}$, $1 \leq i \leq n$. □

The above example shows that the minimal models of $if(P)$ may be exponentially many, while only a very small number of them are the actual stable models.

Notice that in the case of negative logic programs the stable models are in direct correspondence with the kernels of the rule graph. A kernel is a dominating and independent set. Furthermore, it is a minimal dominating set which is independent. This property implies a method that generates all, possibly exponential, minimal dominating sets first, and then test them for independence. A successful independence test ensures stability.

On the other hand, a kernel is a maximal independent set (MIS) which is dominating. Therefore, an alternative method to compute the kernels of a graph, would be to first generate all maximal independent sets (which, like in the case of minimal dominating sets, can be exponentially many), and then test them for dominance. It is important to note that the maximal independent sets can be generated with *polynomial delay* (see [JPY88]). That is, the complexity of computing all MIS is $O(p(N)C)$, where $p(N)$ is a polynomial in the size of the input, while C is the number of MIS's. Moreover, the time required between computing two consecutive solutions is polynomial in the size of the input.

See that the computational model based on the MIS gives us a better characterization of complexity of computing the stable models than the already known general intractability results. Nevertheless, our computational experience has shown that for random negative logic programs, this method of generating the MIS's, leads to poor performance compared to this of the satisfiability or graph algorithms (see next section).

4.3 Graph algorithms

Another alternative is to solve the kernel problem by a graph algorithm. A first attempt towards this direction is the algorithm presented in [DMP93]. It is a backtracking algorithm with a worst time complexity $O(p(N)2^{|F|})$, where F is a *feedback vertex set* for graph and $p(N)$ a polynomial in the size of the input. A feedback vertex set is a set of nodes which when removed result in a acyclic graph. Since computing the feedback vertex set with the minimal cardinality is a NP-hard problem, approximation algorithms can be used, leading to a, hopefully small, feedback vertex set. According to this method, given a program P, the rule graph G_P is constructed and the graph algorithm computes the kernels of G_P, in order to determine whether they are sequential.

Despite the hardness of the general case, a class of logic programs is easier to reason with. Namely, in [DMP93] an algorithm for computing a subset of the kernels of a odd cycle free graph is presented. The basic graph-theoretic property used, is the fact that if G is a strongly connected graph without odd cycles, then G is bipartite. A graph $G = (N, E)$ is called *bipartite* if N can be split into two parts say K, L such that $E \subseteq (K \times L) \cup (L \times K)$. See that any of the two sets K, L is a kernel for G.

An algorithm that computes this set of "standard" kernels is the following. Initially $K = \emptyset$. Repeat the following until G is empty: First, find the strongly connected components of the graph. Since the graph is odd cycle-free, each component is a bipartite graph, say $C_i = (N_{i1}, N_{i2}, E_i)$. For each such component $C_i = (N_{i1}, N_{i2}, E_i)$ with no incoming edges, select $j \in \{1, 2\}$, set $K := K \cup N_{ij}$, and delete from G N_{i1}, N_{i2}, and all nodes v which there is an edge (u, v) with $u \in N_{ij}$.

A modification of the above algorithm leads to a procedure that computes a sequential kernel of an odd cycle free rule graph G_P of a logic program P. Following [PY92], we call a set of literal nodes U an *unfounded set* if the subgraph of G_{P^+} induced by U and their preceding rule nodes has no source.

```
**Compute-Sequential-Kernels**($G_P, K$);
begin
if $G_P = \emptyset$ then return($K$) else
  begin
  Compute the top components of the graph $G_P$, say $C_1, \ldots, C_k$;
  For i:=1 to k do;
      begin
        if for the greatest unfounded set $U_i$ of $C_i$, $U_i \neq \emptyset$ holds then
        Add every node $n_i \in U_i \cap A$ to $B$
        else
        begin
        if $|C_i| = 1$ then add $C_i$ to $B$ else
```