HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **A:** Research Reports
No. **35**; September 1995
ISSN 0783–5396
ISBN 951–22–2752–5

EFFICIENT IMPLEMENTATION OF THE STABLE MODEL SEMANTICS FOR NORMAL LOGIC PROGRAMS

PATRIK SIMONS

Digital Systems Laboratory
Department of Computer Science
Helsinki University of Technology
Otaniemi, FINLAND

Helsinki University of Technology
Department of Computer Science
Digital Systems Laboratory
Otaniemi, Otakaari 1
FIN–02150 ESPOO, FINLAND

HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **A:** Research Reports
No. **35**; September 1995

ISSN 0783–5396
ISBN 951–22–2752–5

Efficient Implementation of the Stable Model Semantics for Normal Logic Programs

PATRIK SIMONS

Abstract: The aim of this work is to develop an efficient implementation method of the stable model semantics of logic programs with negation using as a basis a lately proposed decision procedure for Reiter's default logic. The problem of finding a stable model of a logic program is NP-complete. An algorithm for computing the stable model semantics of propositional logic programs is presented. The time complexity of the algorithm is examined and found to be exponential in the size of the input in the worst case. The algorithm is shown to compute the unique stable model of a logic program with a total well-founded model in time roughly quadratic in the size of the program. The algorithm is compared with three other advanced algorithms for computing the stable models semantics. It is found to be outperforming the other algorithms mainly due to a better technique for pruning the search space. A method for implementing the algorithm efficiently is presented and a prototype implementation is developed. As examples of applications of stable model semantics, model-based diagnosis of logic circuits and the problem of finding Hamiltonian circuits and N-colorings in undirected graphs are encoded as propositional logic programs. These encodings are then used as test cases in a comparison between the prototype implementation of the algorithm presented in this work and the SLG system, a state-of-the-art implementation of the stable model semantics. The performance of the prototype implementation compares favorably to that of the SLG system.

Keywords: stable model semantics, well-founded semantics, logic programming, nonmonotonic reasoning, automatic theorem proving.

Printing: TKK Monistamo; Otaniemi 1995

Helsinki University of Technology
Department of Computer Science
Digital Systems Laboratory
Otaniemi, Otakaari 1
FIN–02150 ESPOO, FINLAND

Phone: $\frac{90}{+358\text{–}0}$ 4511
Telex: 125 161 htkk fi
Telefax: +358–0–465 077
E-mail: lab@saturn.hut.fi

Contents

List of Figures

1 Introduction

In recent years declarative semantics for logic programs with negation have been intensively investigated. However, there has been quite limited amount of research on developing implementation techniques for these semantics. In this work we consider the stable model semantics, one of the leading declarative semantics for logic programs with negation. Our aim is to develop an efficient implementation technique for this semantics using as a basis a recently proposed decision procedure for Reiter's default logic [Nie95].

An early proposition for the interpretation of negation in logic programs was the negation-as-failure rule of prolog. This procedural approach of the goal directed prolog interpreter assumes that if a goal fails, then that goal is false. Thus, when a negative goal is encountered by the interpreter, it tries to prove the goal and if that fails the negative goal succeeds. The negative side of this proof procedure is that it never terminates for some logic programs.

A declarative interpretation of negation in logic programs soon followed; the program completion of Clark [Cla78], later extended by Lloyd [Llo84]. The program completion completes the logic program in the sense that the rules in the program are assumed to be logical equivalences instead of implications or inference rules.

Another proposition for the interpretation of negation in logic programs was the closed world assumption of Reiter [Rei78]. The closed world assumption is the assumption that all facts that are not logical consequences of the program are false. Hence, the logic program defines the world, and everything that is not mentioned in the program is false.

A later approach to the problem of defining a semantics for logic programs with negation consists of declaring a canonical model of the program. A canonical model is one model of possibly several that is preferred over the others. Typically, these semantics define a unique canonical model only for a small class of logic programs. The iterated fixpoint semantics defines a unique canonical model for the class of stratified logic programs. The stratified logic programs have been examined by [CH85, ABW88, Lif88, VG88] and others. The well-founded semantics [VGRS88, VGRS91] and the stable model semantics of Gelfond and Lifschitz [GL88] are two generalizations of the iterated fixpoint semantics that define unique canonical models for classes of programs that include the class of stratified programs. A final generalization of the well-founded semantics is the tie-breaking semantics of Papadimitriou and Yannakakis [PY92].

The intuition behind the stable model semantics is derived from autoepistemic logic [Moo85]. A stable model is a set of beliefs of a rational agent whose inference rules are those of the logic program and who is able to perform negative introspection; if the agent does not believe a fact, it concludes the negation of the fact. The stable model semantics is thus nonmonotonic, as adding rules to the program may cause the agent to retract some of its beliefs. In addition, this intuition makes it possible for a logic program to lack a stable model, have one or several.

We give some examples. Consider the program

$$\begin{aligned} &flies(X) \leftarrow bird(X), \mathrm{not}(abnormal(X)) \\ &bird(Tweety) \leftarrow \end{aligned}$$

which is intended to express that birds that are not abnormal can fly and that Tweety is a bird. Now, a rational agent believes that Tweety is not abnormal as nothing in the program implies otherwise. Hence, the agent concludes that Tweety can fly. However, if we add the rule

$$abnormal(Tweety) \leftarrow$$

then the agent can no longer infer that Tweety can fly. The only stable model of the original program is thus $\{flies(Tweety), bird(Tweety)\}$ and the only stable model of the expanded program is $\{bird(Tweety)\}$.

Consider the program

$$\begin{aligned} &green(apple) \leftarrow \mathrm{not}(red(apple)) \\ &red(apple) \leftarrow \mathrm{not}(green(apple)) \end{aligned}$$

which says that the apple is green if it is not red and red if it is not green. A rational agent may in this case hold the belief that the apple is green, as it then cannot infer that the apple is red. Conversely, the agent may believe that the apple is red, as it then cannot infer that the apple is green. Hence, the program has two stable models; $\{green(apple)\}$ and $\{red(apple)\}$.

Finally, consider the program

$$cold \leftarrow \mathrm{not}(cold)$$

A rational agent cannot hold any beliefs whether it is cold or not using this program. If the agent assumes that it is not cold then it must believe that it is cold, which is a contradiction. Hence, as the agent is rational, it cannot believe that it is not cold. But then it cannot infer that it is cold either. This program has therefore no stable models.

The stable model semantics is a form of nonmonotonic reasoning that is closely related to the default logic of Reiter [Rei80, MS92], circumscription of McCarthy [McC80, McC86, Lif88], and the autoepistemic logic of Moore [Moo85, Gel87]. As the stable model semantics has become a standard method for supplying semantics to nonmonotonic logic programs, there is considerable interest in automating the computation of stable models. The earliest methods for finding stable models were based on the truth maintenance system of Doyle [Doy79] and the assumption-based TMS of deKleer [dK86], see [Elk90, Esh90, PR91]. Recently, more specialized algorithms have been developed [CW93, CW94, BNNS94, SNV95]. The problem of computing stable models is not trivial. In fact, the problem of finding a stable model of a logic program is NP-complete [MT91].

In this work we develop a novel algorithm for computing the stable models of finite propositional logic programs. As a basis for the algorithm we use a recently proposed decision method for default logic [Nie95]. We devise a number of optimizations resulting in a simpler and more efficient implementation method for the stable model semantics. A key task in the algorithm is the computation of deductive closures for monotonically increasing and decreasing sets of premises. We develop an efficient incremental method for computing the closures. We find an upper bound on the time complexity of the algorithm and discuss how it computes well-founded models. We examine a possibility of improving the backtracking of the algorithm and present a method of handling variables by restricting their use. We also present some applications of logic programs with stable model semantics; diagnosis of logic circuits and two combinatorial graph problems. We develop a prototype implementation of the algorithm and compare the implementation with the SLG system, a state-of-the-art implementation of the stable model semantics, using the applications as test cases.

1.1 Outline of the Work

In Section 2 we recall definitions and theorems concerning the stable model semantics and the well-founded semantics. In Section 3 we present two algorithms for computing stable models of finite propositional logic programs together with proofs of their correctness. In Section 4 we continue by examining how one can compute the deductive closures needed by the algorithms efficiently. We note how the algorithm computes well-founded models in Section 5. An upper bound on the time complexity of the novel algorithm is given in Section 6. In Section 7 we examine a strategy for backtracking. Logic programs with variables are examined in Section 8. In Section 9 we describe how to implement the algorithm in an efficient way. We compare the algorithms presented in this work with some other algorithms computing the stable model semantics in Section 10. In Section 11 we present some applications of logic programs with stable model semantics. We compare the execution times of the algorithm and the SLG system [CW93, CW94], a state of the art implementation of the stable model semantics, in Section 12. Finally, in Section 13 we present our conclusions.

2 Fundamental Concepts

In this section we present some concepts and definitions that are needed in the following sections. In Section 2.1 we present the stable model semantics of [GL88] and a characterization of the semantics defined in [Nie95, NS95]. In Section 2.2 we present the well-founded semantics of [VGRS91].

2.1 Stable Model Semantics

We define stable models following the presentation in [Nie95, NS95].

Consider a logic program that consists of rules of the form

$$h \leftarrow a_1, \ldots, a_n, \mathrm{not}(b_1), \ldots, \mathrm{not}(b_m) \tag{1}$$

where $a_1, \ldots, a_n$, $b_1, \ldots, b_m$, and h are propositional atoms. We call the expression $\mathrm{not}(b)$ a not-atom. For a set of atoms A,

$$\mathrm{not}(A) = \{\mathrm{not}(a) \mid \ a \in A\} \tag{2}$$

For a set of atoms and not-atoms Λ we denote the the set of atoms in Λ by Λ^+ and the set of not-atoms by Λ^-. Atoms and not-atoms will also be called literals.

The stable model semantics for a general logic program P is defined as follows [GL88]. The reduct P^S of P with respect to the set of atoms S is obtained by deleting each rule in P that has a not-atom $\mathrm{not}(b)$ in its body such that $b \in S$ and by deleting all not-atoms in the remaining rules. A set of atoms S is a stable model of P if and only if S is the deductive closure of P^S when the rules in P^S are seen as inference rules.

Example 2.1 Let P be the set of rules $\{a \leftarrow \mathrm{not}(b);\ b \leftarrow b\}$. The only stable model of P is $\{a\}$ which is seen as follows. If the atom b is not in the set of atoms S, then the reduct of P with respect to S is $P^S = \{a \leftarrow;\ b \leftarrow b\}$. The closure of this reduct is $\{a\}$. If, on the other hand, b belongs to the set S then the reduct $P^S = \{b \leftarrow b\}$. But, as the closure of $\{b \leftarrow b\}$ is empty, the set S is not a stable model of P. □

The deductive closure of a set of rules P and a set of atoms and not-atoms L is denoted by $\mathrm{Dcl}(P, L)$. $\mathrm{Dcl}(P, L)$ is the smallest set of atoms that contains L^+ and is closed under $\mathrm{R}(P, L)$ where

$$\begin{aligned} \mathrm{R}(P, L) \;=\; & \{h \leftarrow a_1, \ldots, a_n \mid \ h \leftarrow a_1, \ldots, a_n, \mathrm{not}(b_1), \ldots, \mathrm{not}(b_m) \in P \\ & \quad \text{and } \mathrm{not}(b_i) \in L^-, \text{ for } i = 1, \ldots, m\} \end{aligned} \tag{3}$$

is seen as a set of inference rules. Notice that if $L \subseteq L'$ then $\mathrm{Dcl}(P, L) \subseteq \mathrm{Dcl}(P, L')$, i.e., $\mathrm{Dcl}(P, \cdot)$ is monotonic.

Example 2.2 Let P be the set of rules $\{a \leftarrow \mathrm{not}(b);\ b \leftarrow \mathrm{not}(c)\}$ and let L be the set of literals $\{\mathrm{not}(b), c\}$. Then, $\mathrm{R}(P, L) = \{a \leftarrow\}$ and the deductive closure $\mathrm{Dcl}(P, L) = \{a, c\}$. □

The following proposition is stated in [Nie95, NS95].

Proposition 2.1 *The set Δ is a stable model of a set of rules P if and only if $\Delta = \mathrm{Dcl}(P, \mathrm{not}(\overline{\Delta}))$.*

Proof. Note that $P^{\Delta} = \mathrm{R}(P, \mathrm{not}(\overline{\Delta}))$. □

For a set of rules P we denote the set of negative antecedents in P by $\mathrm{NAnt}(P)$, i.e., the set of atoms a such that $\mathrm{not}(a)$ appears in P. For example, $\mathrm{NAnt}(\{a \leftarrow \mathrm{not}(b);\ d \leftarrow c\}) = \{b\}$. Evidently,

$$\mathrm{Dcl}(P, \mathrm{not}(\overline{\Delta})) = \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - \Delta)) \tag{4}$$

holds.

We call a set $\Lambda \subseteq \mathrm{not}(\mathrm{NAnt}(P))$ of not-atoms P-full if and only if it is the case that for all $a \in \mathrm{NAnt}(P)$, $\mathrm{not}(a) \in \Lambda$ if and only if $a \notin \mathrm{Dcl}(P, \Lambda)$. Interestingly, there is a 1-1 correspondence between the P-full sets and the stable models of P.

Theorem 2.2 *If Λ is P-full, then $\Delta = \mathrm{Dcl}(P, \Lambda)$ is a stable model of P and $\Lambda = \mathrm{not}(\mathrm{NAnt}(P) - \Delta)$. If Δ is a stable model of P, then $\Lambda = \mathrm{not}(\mathrm{NAnt}(P) - \Delta)$ is a P-full set and $\Delta = \mathrm{Dcl}(P, \Lambda)$.*

Proof. Let Λ be a P-full set. Let $\Delta = \mathrm{Dcl}(P, \Lambda)$. Now, for all $a \in \mathrm{NAnt}(P)$, $a \notin \Delta$ if and only if $\mathrm{not}(a) \in \Lambda$, which implies that $\mathrm{not}(\mathrm{NAnt}(P) - \Delta) = \Lambda$. Thus, $\Delta = \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - \Delta))$ and Δ is a stable model of P.

Let Δ be a stable model of P. Let $\Lambda = \mathrm{not}(\mathrm{NAnt}(P) - \Delta)$. Then, $\Delta = \mathrm{Dcl}(P, \Lambda)$ as Δ is a stable model of P. Hence, for all $a \in \mathrm{NAnt}(P)$, $\mathrm{not}(a) \in \Lambda$ if and only if $a \notin \Delta$ if and only if $a \notin \mathrm{Dcl}(P, \Lambda)$, which is the definition of a P-full set. □

Example 2.3 Consider the set of rules $P = \{a \leftarrow \mathrm{not}(b);\ b \leftarrow \mathrm{not}(a)\}$. The set of rules P has two stable models $\{a\}$ and $\{b\}$. For instance, the set $\{\mathrm{not}(b)\}$ is P-full as $b \notin \mathrm{Dcl}(P, \{\mathrm{not}(b)\}) = \{a\}$. Therefore, we see that the P-full set $\{\mathrm{not}(b)\}$ corresponds to the stable model $\{a\}$. The empty set serves as a counter example. As the closure $\mathrm{Dcl}(P, \emptyset) = \emptyset$ and as the atom $a \in \mathrm{NAnt}(P)$ but $a \notin \emptyset$, the empty set is not a P-full set. □

Let Λ be a set of atoms and not-atoms, let P be a set of rules, and let A be a set of atoms. The set Λ is *grounded* in P if $\Lambda^+ \subseteq \mathrm{Dcl}(P, \Lambda^-,)$. The set of atoms A *agrees* with Λ if for all atoms $a \in \Lambda^+$, $a \in A$ and for all $\mathrm{not}(a) \in \Lambda^-$, $a \notin A$. Finally, Λ is said to *cover* A if for every $a \in A$ either $a \in \Lambda$ or $\mathrm{not}(a) \in \Lambda$.

Example 2.4 We examine the set of rules $P = \{a \leftarrow \mathrm{not}(b);\ b \leftarrow b\}$. The set $\Lambda = \{a, \mathrm{not}(b)\}$ is grounded in P as $a \in \mathrm{Dcl}(P, \Lambda^-) = \mathrm{Dcl}(P, \{\mathrm{not}(b)\})$. However, the set $\{b\}$ is not grounded in P as $b \notin \mathrm{Dcl}(P, \emptyset)$. The set of atoms $A = \{a\}$ agrees with Λ as $b \notin A$ and as $a \in A$. As a is not in the empty set, we know that the empty set does not agree with Λ. Finally, Λ covers the set $\{a, b\}$ but not the set $\{c\}$. □

2.2 Well-Founded Semantics

We review the well-founded semantics of [VGRS91]. A set of literals I is an interpretation if for every atom a, $a \in I$ implies $\mathrm{not}(a) \notin I$. Thus, the set $\{a, \mathrm{not}(b)\}$ is an interpretation but the set $\{b, \mathrm{not}(b)\}$ is not.

Define $\mathrm{Atoms}(P)$ to be the set of atoms such that $a \in \mathrm{Atoms}(P)$ if and only if there exists a rule $r \in P$ such that a is the head of r, or a or $\mathrm{not}(a)$ is in the body of r. In other words, the set $\mathrm{Atoms}(P)$ consists of the atoms that appear in some rule of P. For example, $\mathrm{Atoms}(\{a \leftarrow \mathrm{not}(b);\ c \leftarrow d\}) = \{a, b, c, d\}$.

Definition 2.3 *Let P be a logic program, let I be an interpretation, and let U be a set of atoms. The set U is an unfounded set of P with respect to I if every atom $a \in U$ satisfies the following condition: for every rule $r \in P$ whose head is a, at least one of the following conditions holds.*

1. *there is a literal b or $\mathrm{not}(c)$ in the body of r such that $\mathrm{not}(b) \in I$ or $c \in I$.*
2. *there is an atom b in the body of r such that $b \in U$.*

Example 2.5 Consider the set of rules $P = \{p \leftarrow \mathrm{not}(p);\ q \leftarrow \mathrm{not}(p);\ a \leftarrow b, \mathrm{not}(c);\ b \leftarrow a\}$. Then, $\{a, b, c\}$ is an unfounded set of P with respect to the empty set. The set $\{p, q\}$ is not an unfounded set with respect to the empty set. However, the set $\{p, q\}$ is an unfounded set with respect to the interpretation $\{p\}$. The larger set $\{a, b, c, p, q\}$ is also an unfounded set with respect to $\{p\}$. □

The union of all unfounded sets of P with respect to I is the greatest unfounded set of P with respect to I. We denote this set by $\mathrm{GUS}_P(I)$ and define

$$\mathbf{U}_P(I) = \mathrm{not}(\mathrm{GUS}_P(I)) \tag{5}$$

Definition 2.4 *Let P be a logic program and let I be an interpretation. The transformation $\mathbf{T}_P(I)$ is defined as follows. An atom a is in $\mathbf{T}_P(I)$ if and only if there is a rule in P with head a such that all literals in the body of the rule are in I.*

For example, if $P = \{a \leftarrow b;\ d \leftarrow \text{not}(c)\}$, then the transformation $\mathbf{T}_P(\{b\}) = \{a\}$ and $\mathbf{T}_P(\{\text{not}(c)\}) = \{d\}$.

Define a sequence of interpretations as follows. Let

$$I_0 = \emptyset \tag{6}$$

and for all $k \geq 0$,

$$I_{k+1} = \mathbf{T}_P(I_k) \cup \mathbf{U}_P(I_k) \tag{7}$$

The well-founded model of the program P is then

$$I^* = \bigcup_{k=0}^{\infty} I_k \tag{8}$$

An interpretation of a logic program P that covers Atoms(P) is a total interpretation. Similarly, a well-founded model of P is a total well-founded model if it covers all atoms in Atoms(P).

Example 2.6 Let P be the set of rules

$$\{a \leftarrow \text{not}(b);\ b \leftarrow \text{not}(a), \text{not}(c);\ d \leftarrow a, b;\ c \leftarrow\}.$$

We will compute the well-founded model of P. As the interpretation $I_0 = \emptyset$, $\mathbf{T}_P(\emptyset) = \{c\}$, and $\mathbf{U}_P(\emptyset) = \emptyset$, we set $I_1 = \{c\}$. Now, $\mathbf{T}_P(I_1) = \{c\}$ and $\mathbf{U}_P(I_1) = \{\text{not}(b), \text{not}(d)\}$ which gives $I_2 = \{c, \text{not}(b), \text{not}(d)\}$. Thus, the transformation $\mathbf{T}_P(I_2) = \{a, c\}$, $\mathbf{U}_P(I_2) = \{\text{not}(b), \text{not}(d)\}$, and $I_3 = \{a, c, \text{not}(b), \text{not}(d)\}$. Finally, $I_4 = I_3$ as $\mathbf{T}_P(I_3) = \{a, c\}$ and $\mathbf{U}_P(I_3) = \{\text{not}(b), \text{not}(d)\}$. Hence, the interpretations $I_4 = I_5 = I_6 = \ldots$ and the well-founded model of P is $\bigcup_{k=0}^{\infty} I_k = \{a, c, \text{not}(b), \text{not}(d)\}$. By noting that Atoms($P$) $= \{a, b, c, d\}$ we see that the well-founded model of P is, in fact, a total well-founded model. □

3 An Algorithm for Computing Stable Models

In this section we examine an algorithm for computing stable models. This algorithm was presented in [Nie95] as a decision procedure for default logic [Rei80]. In [NS95] it was used to implement the stable model semantics. The algorithm is based on ideas introduced in [Nie94] in the context of autoepistemic reasoning.

We describe the algorithm in Section 3.1. In Sections 3.2–3.3 we describe an implementation of the two functions **expand** and **conflict** needed by the algorithm. In Section 3.4 we note a redundancy in the algorithm. Finally, in Section 3.5 we present a novel algorithm based on the algorithm described in Sections 3.1–3.3 that is simpler and prunes the search space more effectively.

3.1 The Algorithm

The algorithm **smodels** described in Figure 1 is a procedure for computing stable models of a set of rules P. The algorithm takes as input a set of rules P, a set of literals B, and a set of atoms F. The aim of the algorithm is to compute a stable model Δ of P agreeing with $B \cup F$ such that the function **test**(Δ) returns true. This goal is achieved by constructing P-full sets until a stable model that is accepted by **test** is found.

The P-full sets are constructed one at a time by adding not-atoms to the set B. All P-full sets are found using backtracking. The not-atoms that are removed from B^- during backtracking are stored as atoms in the set F. The set F therefore contains the atoms that have to be included in the stable model. The algorithm **smodels** has to examine a very large search space as every subset of $\mathrm{NAnt}(P)$ is a possible P-full set. In order to reduce the search space, **smodels** uses two functions; **expand** and **conflict**. The function **expand** tries to expand the set B as much as possible. Literals that can not be in the P-full set under construction are added to B^+ and literals that have to be in the P-full set are added to B^-. The remaining search space is halved each time a literal is added to the set B. By remaining search space we mean all subsets of the set of atoms in $\mathrm{NAnt}(P)$ not covered by $B \cup F$. The function **conflict** detects conflicts, i.e., situations where there are no stable models agreeing with $B \cup F$. It is therefore possible to begin backtracking as soon as a conflict is detected. The remaining search space is also halved when a conflict is found.

The algorithm **smodels** is sound and complete if the functions **expand** and **conflict** fulfill some basic conditions. Soundness guarantees that if **smodels** returns true, there is a stable model Δ of P agreeing with $B \cup F$ such that **test**(Δ) returns true. Completeness, on the other hand, guarantees that if

function smodels(P, B, F)
$B_n :=$ **expand**(P, B, F)
if conflict(P, B_n, F) returns true **then**
    return false
else if $\mathrm{NAnt}(P)$ is covered by $B_n \cup F$ **then**
    if $F \subseteq B_n$ **then**
        return test$(\mathrm{Dcl}(P, B_n^-))$
    else
        return false
    endif
else
    take some $\chi \in \mathrm{NAnt}(P)$ not covered by $B_n \cup F$.
    if smodels$(P, B_n \cup \{\mathrm{not}(\chi)\}, F)$ returns true **then**
        return true
    else
        return smodels$(P, B_n, F \cup \{\chi\})$
    endif
endif

Figure 1: The Decision Procedure for Logic Programs with the Stable Model Semantics

there is a stable model Δ agreeing with $B \cup F$ such that $\mathbf{test}(\Delta)$ returns true, then **smodels** also returns true.

Let $B_n = \mathbf{expand}(P, B, F)$. We assume that the function **expand** fulfills the following conditions.

E1 $B \subseteq B_n$.

E2 If B is grounded in P, then B_n is grounded in P.

E3 If there is an extension Δ of P such that Δ agrees with $B \cup F$, then Δ agrees with $B_n \cup F$.

E4 For all $a \in \mathrm{NAnt}(P)$, $a \in \mathrm{Dcl}(P, B_n)$ implies that a is covered by B_n.

The function **conflict** returns true or false and satisfies the following conditions.

C1 The function **conflict** returns true if there exists a literal $\mathrm{not}(a) \in B$ such that $a \in \mathrm{Dcl}(P, B^-)$.

C2 If **conflict** returns true, then there exists no extension Δ of P such that Δ agrees with $B \cup F$.

We now prove that **smodels** is sound and complete.

Theorem 3.1 (Soundness) *Let P be a set of rules, let B be a set of literals grounded in P, and let F be a set of atoms. If* $\mathbf{smodels}(P, B, F)$ *returns true, then there exists a stable model Δ of P agreeing with $B \cup F$ such that* $\mathbf{test}(\Delta)$ *returns true.*

Proof. The proof is by induction on the number of atoms in $\mathrm{NAnt}(P)$ not covered by $B \cup F$.

We prove that if $\mathrm{NAnt}(P)$ is covered by $B \cup F$, if B is grounded in P and if the function $\mathbf{smodels}(P, B, F)$ returns true, then there exists a stable model $\Delta = \mathrm{Dcl}(P, B_n^-)$ of P agreeing with $B \cup F$ such that $\mathbf{test}(\Delta)$ returns true.

Assume that $\mathrm{NAnt}(P)$ is covered by $B \cup F$, B is grounded in P and that $\mathbf{smodels}(P, B, F)$ returns true. We have that $F \subseteq B_n$, $\mathrm{NAnt}(P)$ is covered by B_n and that $\mathbf{conflict}(P, B_n, F)$ returns false. Now, $a \in \mathrm{Dcl}(P, B_n^-)$ implies $\mathrm{not}(a) \notin B_n$ by C1. If $a \in \mathrm{NAnt}(P)$ and $\mathrm{not}(a) \notin B_n$, then $a \in B_n$. Since B_n is grounded in P by E2, $a \in B_n$ implies $a \in \mathrm{Dcl}(P, B_n^-)$. Thus, B_n is P-full and as $B \subseteq B_n$ by E1, there exists a stable model $\Delta = \mathrm{Dcl}(P, B_n^-)$ of P agreeing with $B \cup F$ such that $\mathbf{test}(\Delta)$ returns true.

Induction hypothesis: assume that if the number of atoms in $\mathrm{NAnt}(P)$ not covered by $B \cup F$ is less than n, if B is grounded in P and if the function **smodels**(P, B, F) returns true, then there exists a stable model Δ of P agreeing with $B \cup F$ such that **test**(Δ) returns true.

Suppose the number of atoms in $\mathrm{NAnt}(P)$ not covered by $B \cup F$ is n, B is grounded in P and **smodels**(P, B, F) returns true. As **smodels**(P, B, F) returns true, **conflict**(P, B_n, F) returns false. If $\mathrm{NAnt}(P)$ is covered by $B_n \cup F$, then, by the proof above, $\Delta = \mathrm{Dcl}(P, B_n^-)$ is a stable model of P agreeing with $B \cup F$ such that **test**(Δ) returns true. Otherwise, an atom $\chi \in \mathrm{NAnt}(P)$ not covered by $B_n \cup F$ is taken. Now, either **smodels**$(P, B_n \cup \{\mathrm{not}(\chi)\}, F)$ returns true or **smodels**$(P, B_n, F \cup \{\chi\})$ returns true. In addition, $B \cup F \subseteq B_n \cup F \cup \{\mathrm{not}(\chi)\}$ and $B \cup F \subseteq B_n \cup F \cup \{\chi\}$ by the condition E1. Hence, the induction hypothesis proves that there exists a stable model Δ of P agreeing with $B \cup F$ such that **test**(Δ) returns true. □

Theorem 3.2 (Completeness) *Let P be a set of rules, let B be a set of literals, and let F be a set of atoms. If there exists a stable model Δ of P agreeing with $B \cup F$ such that* **test**(Δ) *returns true, then* **smodels**(P, B, F) *returns true.*

Proof. The proof is by induction on the number of atoms in $\mathrm{NAnt}(P)$ not covered by $B \cup F$.

We prove that if there exists a stable model Δ of P agreeing with $B \cup F$ such that **test**(Δ) returns true, then **smodels**(P, B, F) returns true.

Assume there exists a stable model Δ of P agreeing with $B \cup F$ such that **test**(Δ) returns true. Assume that $\mathrm{NAnt}(P)$ is covered by $B \cup F$. The function **conflict**(P, B_n, F) returns false by the conditions E3 and C2. As Δ agrees with $B_n \cup F$ and as E1 implies that $\mathrm{NAnt}(P)$ is covered by $B_n \cup F$, $\Delta = \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - \Delta)) = \mathrm{Dcl}(P, B_n \cup F) = \mathrm{Dcl}(P, B_n^-)$. Thus, $F \subseteq B_n$ holds by condition E4. Hence, **smodels**(P, B, F) returns true as $\Delta = \mathrm{Dcl}(P, B_n^-)$ and **test**(Δ) returns true.

Induction hypothesis: if there exists a stable model Δ of P agreeing with $B \cup F$ such that **test**(Δ) returns true and if the number of atoms in $\mathrm{NAnt}(P)$ not covered by $B \cup F$ is less than n, then **smodels**(P, B, F) returns true.

Assume that the number of atoms in $\mathrm{NAnt}(P)$ not covered by $B \cup F$ is n. Assume there exists a stable model Δ agreeing with $B \cup F$ such that **test**(Δ) returns true.

The conditions E3 and C2 implies that **conflict**(P, B_n, F) returns false. If $\mathrm{NAnt}(P)$ is covered by $B_n \cup F$, then **smodels** returns true by the proof above. If not, an atom $\chi \in \mathrm{NAnt}(P)$ not covered by $B_n \cup F$ is taken. If there exists a stable model Δ' agreeing with $B_n \cup F \cup \{\text{not}(\chi)\}$ such that **test**(Δ') returns true, then **smodels**$(P, B_n \cup \{\text{not}(\chi)\}, F)$ returns true by E1 and the induction hypothesis. Otherwise, Δ is a stable model agreeing with $B_n \cup F \cup \{\chi\}$ and **smodels**$(P, B_n, F \cup \{\text{not}(\chi)\})$ returns true. Thus, **smodels**(P, B, F) returns true. □

These theorems imply that it is possible to solve different reasoning tasks by choosing suitable **test** functions.

Corollary 3.3 *Let P be a set of rules and let ϕ be an atom.*

1. *If* **test**(Δ) *returns true for all Δ and ϕ, then* **smodels**$(P, \emptyset, \emptyset)$ *returns true if and only if P has a stable model.*
2. *If* **test**(Δ) *returns true if and only if $\phi \in \Delta$, then* **smodels**$(P, \emptyset, \emptyset)$ *returns true if and only if there exists a stable model of P containing ϕ.*
3. *If* **test**(Δ) *returns true if and only if $\phi \notin \Delta$, then* **smodels**$(P, \emptyset, \emptyset)$ *returns true if and only if there exists a stable model of P not containing ϕ and, in other words,* **smodels**$(P, \emptyset, \emptyset)$ *returns false if and only if ϕ is contained in every stable model of P.*

3.2 Expanding B

An implementation of the function **expand** is given in Figure 2. The function **expand** enlarges the set B by assuming that there exists a stable model Δ agreeing with $B \cup F$ and by using the relation

$$\mathrm{Dcl}(P, B^-) \subseteq \Delta \subseteq \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - (B \cup F))) \tag{9}$$

As $\Delta = \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - \Delta))$ and as $B^- \subseteq \mathrm{not}(\mathrm{NAnt}(P) - \Delta)$ if Δ agrees with B^-, we see that $\mathrm{Dcl}(P, B^-) \subseteq \Delta$ holds. The other set inclusion is proved in Lemma 3.4. We prove that the function satisfies the conditions E1-E4.

Lemma 3.4 *Let Δ be a stable model of the set of rules P and let Δ agree with a set of literals Λ. Then $\phi \in \Delta$ implies $\phi \in \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - \Lambda))$.*

Proof. If $\phi \notin \Delta$, then $\phi \notin \Lambda$ as Δ agrees with Λ. Thus, $\mathrm{NAnt}(P) - \Delta \subseteq \mathrm{NAnt}(P) - \Lambda$ and as $\Delta = \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - \Delta))$, we can conclude that the stable model $\Delta \subseteq \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - \Lambda))$. □

Proposition 3.5 *The implementation in Figure 2 satisfies conditions E1–E4.*

function **expand**(P, B, F)
repeat
  $B' := B$
  $B := B \cup \{a \in \mathrm{NAnt}(P) \mid \ a$ not covered by B, $a \in \mathrm{Dcl}(P, B^-)\}$
  $B := B \cup \{\mathrm{not}(a) \mid a \in \mathrm{NAnt}(P)$, a not covered by B,
      $a \notin \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - (B \cup F)))\}$
until $B = B'$
return B

Figure 2: Implementation of the Function **expand**

function **conflict**(P, B, F)
if there exists $\mathrm{not}(a) \in B$ such that $a \in \mathrm{Dcl}(P, B \cup F)$ **then**
  return true
else if there exists $a \in F$ such that $a \notin \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - (B \cup F)))$ **then**
  return true
else
  return false
endif

Figure 3: Implementation of the Function **conflict**

Proof. The condition E1 obviously holds as no literals are removed from B. The conditions E2 and E4 hold as $B^+ = \{a \in \mathrm{NAnt}(P) \mid \ a$ not covered by B^-, $a \in \mathrm{Dcl}(P, B^-)\}$ holds when **expand** returns.

If Δ agrees with $B \cup F$, then Δ agrees with B and $\mathrm{Dcl}(P, B^-) \subseteq \Delta$ as $B^- \subseteq \mathrm{not}(\overline{\Delta})$. Since Lemma 3.4 guarantees that if $\phi \notin \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - (B \cup F)))$ then $\phi \notin \Delta$, we have shown that **expand** also fulfills condition E3. □

3.3 Detecting Conflicts

The function **conflict** is described in Figure 3. The function takes advantage of the fact that the set F is supposed to be in the final stable model to test for conflicts. We prove that the function satisfies the conditions C1–C2.

Proposition 3.6 *The implementation in Figure 3 satisfies conditions C1–C2.*

Proof. Since $\mathrm{Dcl}(P, B^-) \subseteq \mathrm{Dcl}(P, B \cup F)$ the condition C1 holds. To prove C2, assume there exists a stable model Δ of P agreeing with $B \cup F$. Then, $B^+ \cup F \subseteq \Delta$ and $B^- \subseteq \mathrm{not}(\overline{\Delta})$. As $\Delta = \mathrm{Dcl}(P, \mathrm{not}(\overline{\Delta}))$, we immediately get

$\mathrm{Dcl}(P, B \cup F) \subseteq \Delta$. Hence, $\mathrm{not}(\phi) \in B$ implies $\phi \notin \mathrm{Dcl}(P, B \cup F)$. If $\phi \in F$, then $\phi \in \Delta$ and by Lemma 3.4, $\phi \in \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - (B \cup F)))$. Thus, condition C2 is satisfied. □

3.4 The F Subset of B_n Test

The test $F \subseteq B_n$ in the function **smodels** is not needed when one is using the **conflict** and **expand** functions defined in the previous sections. Assume that the atom $a \in F$ but $a \notin B_n$, **conflict**(P, B_n, F) returns false and that $\mathrm{NAnt}(P)$ is covered by $B_n \cup F$. Then, $a \in F$ implies $a \in \mathrm{Dcl}(P, B_n \cup F)$ which implies that $\mathrm{not}(a) \notin B_n^-$ as **conflict** returns false. Furthermore, if a literal $\mathrm{not}(b) \notin B_n$ but $b \in \mathrm{NAnt}(P)$, then $b \in B_n$ or $b \in F$ as $\mathrm{NAnt}(P)$ is covered by $B_n \cup F$. Thus, $\mathrm{not}(\mathrm{NAnt}(P) - (B_n \cup F)) \subseteq B_n^-$. By the monotonicity of the deductive closure operator $\mathrm{Dcl}(P, \cdot)$ we have that $\mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - (B_n \cup F))) \subseteq \mathrm{Dcl}(P, B_n^-)$. Now, for $a \in \mathrm{NAnt}(P)$, $a \notin B_n$ and $\mathrm{not}(a) \notin B_n$ implies $a \notin \mathrm{Dcl}(P, B_n^-)$ by the definition of **expand**. But $a \in F$ implies $a \in \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - (B_n \cup F)))$ as **conflict** returns false. This contradiction proves that $a \in F$ implies $a \in B_n$ or in other words that $F \subseteq B_n$. Hence, this test is superfluous and can be removed.

3.5 Improving the Algorithm

Eliminating the $F \subseteq B_n$ test simplifies the algorithm somewhat. It is, however, possible to simplify it even further. By strengthening the **expand** function one improves the algorithm, as the search space is then pruned more effectively. We strengthen the **expand** function in a suitable way, thereby making it possible to eliminate the set F. The new algorithm is described in Figures 4–6.

The algorithm is improved in three ways. Instead of three different deductive closures only two different deductive closures need to be computed. The function **expand**$_2$ is strengthened by replacing $B := B \cup \{a \in \mathrm{NAnt}(P) \mid a \text{ not covered by } B,\ a \in \mathrm{Dcl}(P, B^-)\}$ with $B := B \cup \{a \in \mathrm{NAnt}(P) \mid a \text{ not covered by } B,\ a \in \mathrm{Dcl}(P, B)\}$. As the set B in **expand**$_2$ contains the set F of **expand**, the search space in pruned more effectively. Finally, removing the set F makes the algorithm simpler which in turn makes it easier to implement.

We prove that the algorithm is sound and complete.

Theorem 3.7 (Soundness) *Let P be a set of rules and let B be a set of literals. If* **smodels**$_2(P, B)$ *returns true, then there exists a stable model Δ of P agreeing with B such that* **test**(Δ) *returns true.*

function $\mathbf{smodels}_2(P, B)$
B_n := $\mathbf{expand}_2(P, B)$
if $\mathbf{conflict}_2(P, B_n)$ returns true **then**
    return false
else if $\mathrm{NAnt}(P)$ is covered by B_n **then**
    return $\mathbf{test}(\mathrm{Dcl}(P, B_n))$
else
    take some $\chi \in \mathrm{NAnt}(P)$ not covered by B_n.
    if $\mathbf{smodels}_2(P, B_n \cup \{\mathrm{not}(\chi)\})$ returns true **then**
        return true
    else
        return $\mathbf{smodels}_2(P, B_n \cup \{\chi\})$
    endif
endif

Figure 4: Improved Decision Procedure for Logic Programs with the Stable Model Semantics

function $\mathbf{expand}_2(P, B)$
repeat
    $B' := B$
    $B := B \cup \{a \in \mathrm{NAnt}(P) \mid \ a \text{ not covered by } B,\ a \in \mathrm{Dcl}(P, B)\}$
    $B := B \cup \{\mathrm{not}(a) \mid a \in \mathrm{NAnt}(P),\ a \text{ not covered by } B,$
                    $a \notin \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))\}$
until $B = B'$
return B

Figure 5: Improved Version of the Function **expand**

function $\mathbf{conflict}_2(P, B)$
if there exists $\mathrm{not}(a) \in B$ such that $a \in \mathrm{Dcl}(P, B)$ **then**
    return true
else if there exists $a \in B$ such that $a \notin \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$ **then**
    return true
else
    return false
endif

Figure 6: Improved Version of the Function **conflict**

Proof. The proof is by induction on the number of atoms in $\mathrm{NAnt}(P)$ not covered by B.

We prove that if $\mathrm{NAnt}(P)$ is covered by B and if $\mathbf{smodels}_2(P, B)$ returns true, then there exists a stable model $\Delta = \mathrm{Dcl}(P, B_n)$ agreeing with B such that $\mathbf{test}(\Delta)$ returns true.

Assume that $\mathrm{NAnt}(P)$ is covered by B and that $\mathbf{smodels}_2(P, B)$ returns true. Then, $B = B_n$ as $\mathbf{expand}_2$ does not remove any atoms from B and $\mathrm{not}(B_n^+) \cap B_n^- = \emptyset$ as $\mathbf{conflict}_2(P, B_n)$ returns false. Furthermore, $a \in B_n$ implies $a \in \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B_n))$ as $\mathbf{conflict}_2$ returns false. Hence, the deductive closure $\mathrm{Dcl}(P, B_n) \subseteq \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B_n))$ as not-atoms must be in $\mathrm{not}(\mathrm{NAnt}(P))$ to have any effect on the size of the closure. As $\mathrm{not}(\mathrm{NAnt}(P) - B_n) \subseteq B_n^-$, we deduce that $\mathrm{Dcl}(P, B_n) = \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B_n)) = \mathrm{Dcl}(P, B_n^-)$. It is then clear that $a \notin \mathrm{Dcl}(P, B_n^-)$ implies $\mathrm{not}(a) \in B_n^-$. Finally, if $\mathrm{not}(a) \in B_n$, then $a \notin \mathrm{Dcl}(P, B_n)$ as $\mathbf{conflict}_2$ returns false. Hence, $\mathrm{NAnt}(P) \cap B_n^-$ is P-full and there exists a stable model $\Delta = \mathrm{Dcl}(P, B_n)$ agreeing with B such that $\mathbf{test}(\Delta)$ returns true.

Induction hypothesis: assume that if the number of atoms in $\mathrm{NAnt}(P)$ not covered by B is less than n and if $\mathbf{smodels}_2(P, B)$ returns true, then there exists a stable model Δ of P agreeing with B such that $\mathbf{test}(\Delta)$ returns true.

Suppose the number of atoms in $\mathrm{NAnt}(P)$ not covered by B is n and suppose that $\mathbf{smodels}_2(P, B)$ returns true. As $\mathbf{smodels}_2(P, B)$ returns true, $\mathbf{conflict}_2(P, B_n)$ returns false. As $\mathbf{expand}_2$ does not remove any atoms from B, we see that $B \subseteq B_n$. If the set $\mathrm{NAnt}(P)$ is covered by B_n, then, as is proved above, $\Delta = \mathrm{Dcl}(P, B_n)$ is a stable model of P agreeing with B such that $\mathbf{test}(\Delta)$ returns true. Otherwise, an atom $\chi \in \mathrm{NAnt}(P)$ not covered by B_n is taken. Now, either $\mathbf{smodels}_2(P, B_n \cup \{\mathrm{not}(\chi)\})$ returns true or $\mathbf{smodels}_2(P, B_n \cup \{\chi\})$ returns true. Hence, the induction hypothesis proves that there exists a stable model Δ of P agreeing with B such that $\mathbf{test}(\Delta)$ returns true. $\square$

If the stable model Δ agrees with B, then $\mathrm{Dcl}(P, B) \subseteq \Delta$ as $B^- \subseteq \mathrm{not}(\overline{\Delta})$ and as $B^+ \subseteq \Delta$. Lemma 3.4 proves that if $\phi \notin \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$, then $\phi \notin \Delta$. Thus, we have proved that $\mathbf{expand}_2$ fulfills condition E3 of Section 3.1.

Theorem 3.8 (Completeness) *Let P be a set of rules and let B be a set of literals. If there exists a stable model Δ of P agreeing with B such that* $\mathbf{test}(\Delta)$ *returns true, then* $\mathbf{smodels}_2(P, B)$ *returns true.*

Proof. The proof is by induction on the number of atoms in $\mathrm{NAnt}(P)$ not covered by B.

We prove that if there exists a stable model Δ of P agreeing with B such that $\mathbf{test}(\Delta)$ returns true, then $\mathbf{smodels}_2(P, B)$ returns true.

Assume there exists a stable model Δ of P agreeing with B such that the function $\mathbf{test}(\Delta)$ returns true. Assume that $\mathrm{NAnt}(P)$ is covered by B. Then, $B = B_n$ as $\mathbf{expand}_2$ does not remove any atoms from B. Now, $\phi \in \mathrm{NAnt}(P) - \Delta$ if and only if $\phi \in \mathrm{NAnt}(P)$ and $\phi \notin \Delta$, which is equivalent to $\phi \in \mathrm{NAnt}(P)$ and $\phi \notin B_n^+$, as $\mathrm{NAnt}(P)$ is covered by B_n. Thus, $\mathrm{not}(\mathrm{NAnt}(P) - \Delta) = \mathrm{not}(\mathrm{NAnt}(P)) \cap B_n^-$ and therefore $\Delta = \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - \Delta)) = \mathrm{Dcl}(P, B_n^-) = \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B_n))$. As $B_n^+ \subseteq \Delta$, we conclude that $\Delta = \mathrm{Dcl}(P, B_n) = \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B_n))$. Hence, $\mathbf{conflict}_2(P, B_n)$ returns false, $\mathbf{test}(\mathrm{Dcl}(P, B_n))$ returns true, and $\mathbf{smodels}_2(P, B)$ returns true.

Induction hypothesis: if there exists a stable model Δ of P agreeing with B such that $\mathbf{test}(\Delta)$ returns true and if the number of atoms in $\mathrm{NAnt}(P)$ not covered by B is less than n, then $\mathbf{smodels}_2(P, B)$ returns true.

Assume that the number of atoms in $\mathrm{NAnt}(P)$ not covered by B is n and there exists a stable model Δ agreeing with B such that $\mathbf{test}(\Delta)$ returns true.

The condition E3 implies that Δ agrees with B_n. Now, $\phi \in B_n^+$ implies $\phi \in \Delta$ which by Lemma 3.4 implies that $\phi \in \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B_n))$. In addition, $\mathrm{not}(\phi) \in B_n$ implies $\phi \notin \Delta$, which in turn implies $\phi \notin \mathrm{Dcl}(P, B_n)$ as $\mathrm{Dcl}(P, B_n) \subseteq \Delta$. Thus, $\mathbf{conflict}_2(P, B_n)$ returns false. If $\mathrm{NAnt}(P)$ is covered by B_n, then $\mathbf{smodels}_2$ returns true by the proof above. If not, an atom $\chi \in \mathrm{NAnt}(P)$ not covered by B_n is taken. If there exists a stable model Δ' agreeing with $B_n \cup \{\mathrm{not}(\chi)\}$ such that $\mathbf{test}(\Delta')$ returns true, then $\mathbf{smodels}_2(P, B_n \cup \{\mathrm{not}(\chi)\})$ returns true by the induction hypothesis. Otherwise, Δ is a stable model agreeing with $B_n \cup \{\chi\}$ and $\mathbf{smodels}_2(P, B_n \cup \{\chi\})$ returns true. Thus, $\mathbf{smodels}_2(P, B)$ returns true. □

Example 3.1 By examining the execution of **smodels** and $\mathbf{smodels}_2$ on a simple example, we show that $\mathbf{smodels}_2$ indeed prunes the search space more effectively than **smodels**. Let P be the set of rules

$$\begin{aligned}&\{a \leftarrow \mathrm{not}(a);\ b \leftarrow a;\ h \leftarrow \mathrm{not}(b);\\ &\ h \leftarrow \mathrm{not}(h);\ a \leftarrow \mathrm{not}(h);\ b \leftarrow \mathrm{not}(h)\}.\end{aligned}$$

We examine the execution of the function call $\mathbf{smodels}(P, \emptyset, \emptyset)$. The call to $\mathbf{expand}(P, \emptyset, \emptyset)$ returns the empty set and $\mathbf{conflict}(P, \emptyset, \emptyset)$ returns false. Assume that a is chosen from $\mathrm{NAnt}(P)$. Then, we call $\mathbf{smodels}(P, \{\mathrm{not}(a)\}, \emptyset)$. As the deductive closure $\mathrm{Dcl}(P, \{\mathrm{not}(a)\}) = \{a, b\}$, **conflict** returns true and $\mathbf{smodels}(P, \{\mathrm{not}(a)\}, \emptyset)$ returns false. Next, $\mathbf{smodels}(P, \emptyset, \{a\})$ is called, $\mathbf{expand}(P, \emptyset, \{a\})$ returns the empty set and **conflict** returns false. Now, assume that b is chosen and that $\mathbf{smodels}(P, \{\text{not}(b)\}, \{a\})$ is called. Then, $\mathbf{expand}(P, \{\text{not}(b)\}, \{a\})$ returns the set $\{h, \text{not}(b), \text{not}(a)\}$ and the function **conflict** returns true. Hence, $\mathbf{smodels}(P, \{\text{not}(b)\}, \{a\})$ returns false and we call $\mathbf{smodels}(P, \emptyset, \{a, b\})$. The call to $\mathbf{expand}(P, \emptyset, \{a, b\})$ returns the empty set and **conflict** returns false. Next, $\mathbf{smodels}(P, \{\text{not}(h)\}, \{a, b\})$ is called, whereupon $\mathbf{expand}(P, \{\text{not}(h)\}, \{a, b\})$ returns $\{a, b, \text{not}(h)\}$ and the function **conflict** returns true. Finally, $\mathbf{smodels}(P, \emptyset, \{a, b, h\})$ is called, the function $\mathbf{expand}(P, \emptyset, \{a, b, h\})$ returns $\{\text{not}(a), \text{not}(b), \text{not}(h)\}$, and **conflict** returns true. Thus, we conclude that $\mathbf{smodels}(P, \emptyset, \emptyset)$ returns false after four conflicts.

We now compare this with the execution of $\mathbf{smodels}_2(P, \emptyset)$. The call to the function $\mathbf{expand}_2(P, \emptyset)$ returns the empty set and $\mathbf{conflict}_2(P, \emptyset)$ returns false. Assume that a is chosen from the atoms in $\text{NAnt}(P)$. Then, we call $\mathbf{smodels}_2(P, \{\text{not}(a)\})$. The function $\mathbf{expand}_2(P, \{\text{not}(a)\})$ returns the set $\{b, \text{not}(a)\}$, $\mathbf{conflict}_2$ returns true and $\mathbf{smodels}_2(P, \{\text{not}(a)\})$ returns false. In the next step, $\mathbf{smodels}_2(P, \{a\})$ is called, the function $\mathbf{expand}_2(P, \{a\})$ returns the set $\{a, b\}$ and $\mathbf{conflict}_2$ returns false. Now, h is chosen and $\mathbf{smodels}_2(P, \{a, b, \text{not}(h)\})$ is called. Then, the function $\mathbf{conflict}_2$ returns true as $\text{Dcl}(P, \{a, b, \text{not}(h)\}) = \{a, b, h\}$. Finally, $\mathbf{smodels}_2(P, \{a, b, h\})$ is called, whereupon $\mathbf{conflict}_2$ returns true as $\text{Dcl}(P, \text{not}(\text{NAnt}(P) - \{a, b, h\})) = \emptyset$. Thus, we conclude that $\mathbf{smodels}_2(P, \emptyset)$ returns false after three conflicts, which is one less than what **smodels** needs. $\square$

4 Computing Deductive Closures

We describe two linear time algorithms for computing deductive closures of a set of rules. The deductive closures are calculated using a modified version of the first linear time algorithm of Dowling and Gallier [DG84] for computing the deductive closure of a set of Horn clauses. The modification is necessary as the rules contain, apart from ordinary atoms, also not-atoms. In addition, we exploit the fact that the set $\mathrm{R}(P, B)$ shrink or grow when the set of literals B shrink or grow to compute the deductive closures of the set of rules P incrementally.

In Section 4.1 we describe the basic algorithm for computing deductive closures and describe how it can be used to compute deductive closures incrementally. In Section 4.2 we modify the basic algorithm to compute diminishing deductive closures.

4.1 Deductive Closures

We assume that the rules and atoms are stored in arrays. In order to guarantee constant access times we let an atom or rule be the index into the array. If r is the rule $h \leftarrow a_1, \ldots, a_n, \mathrm{not}(b_1), \ldots, \mathrm{not}(b_m)$, we let *r.head* denote the head of the rule, i.e., the atom h. In addition, we join two counters *poslit* and *neglit* to each rule. These counters are initially initialized with the number of respectively atoms and not-atoms in the body of the rule. For each atom a we initialize a list *a.poslitlist* with all rules whose bodies contain the atom a as a positive literal. Similarly, for each atom a we initialize a list *a.headlist* with all rules whose heads are a. An array *closure* is used to tell which atoms belong to the closure, an atom a belonging to the closure if $closure[a] = \mathrm{true}$.

The algorithm that computes the deductive closure $\mathrm{Dcl}(P, B)$ of a set of rules P and a set of literals B is described in Figure 7. The algorithm works by putting rules whose heads belong in the closure, but are not yet in the closure, on the queue. When a rule is removed from the queue and when the head of the rule is not in the closure, the algorithm adds the head to the closure and then puts every newly activated rule on the queue. By an activated rule we mean, of course, a rule which has the atoms in its body in the closure array and which has the not-atoms in its body in B^-. The counter *r.poslit* of a rule r holds the number of atoms in the body of r not in the closure array and the counter *r.neglit* holds the number of not-atoms in the body of r not in B. As the deductive closure $\mathrm{Dcl}(P, B)$ is the closure of the set $\mathrm{R}(P, B)$, we only want to consider the rules whose *neglit* counters are zero. Therefore, the algorithm checks this before adding a rule to the queue *queue*. This is the modification to the Dowling-Gallier algorithm mentioned above. In order to compute the

function dcl($queue, P$)
while queue is not empty **do**
    r := pop($queue$)
    $a := r.head$
    if $closure[a]$ = false **then**
        $closure[a]$:= true
        for each rule $r' \in a.poslitlist$ **do**
            $r'.poslit := r'.poslit - 1$
            if $r'.poslit = 0$ and $r'.neglit = 0$ **then**
                push(r', $queue$)
            endif
        end for
    endif
end while

Figure 7: The Procedure for Computing Deductive Closures

deductive closure $\mathrm{Dcl}(P, B)$, we have to initialize the queue $queue$ with the rules in P whose bodies are contained in the set B, i.e., with the rules whose counters are zero.

The algorithm computes the deductive closure linearly in the size of the set of rules P as each rule is added at most once to the queue and as each atom is added at most once to the deductive closure.

The algorithm can be used to compute the deductive closure incrementally. Suppose the algorithm has computed the closure $\mathrm{Dcl}(P, B)$. Suppose we want to compute $\mathrm{Dcl}(P, B')$ and suppose $B \subseteq B'$. In order to compute the larger closure starting from the smaller, we only have to initialize the queue with the rules whose bodies are in B' and whose heads are not in the closure. The algorithm will then compute the new closure when invoked.

4.2 Diminishing Deductive Closures

What can we do if the set B' is contained in B? A slightly different algorithm is needed if we want to compute the closure efficiently in this case, see Figure 8. The idea is that instead of propagating the fact that an atom should belong to the closure, we propagate the fact that an atom should be removed from the closure. This is done by pushing rules whose heads should be removed from the deductive closure onto the queue $queue_2$. When a rule is removed from the queue, we check if the head h of the rule is in the closure. If this is the case, then the head is removed from the closure and for each rule who has h in

function $\mathbf{dcl}_2(queue_2, P, B)$
$N := \emptyset$
while $queue_2$ is not empty **do**
  $r :=$ pop($queue_2$)
  $a := r.head$
  if $closure[a] =$ true and $a \notin B$ **then**
    $closure[a] :=$ false
    $N := N \cup \{a\}$
    for each rule $r' \in a.poslitlist$ **do**
      $r'.poslit := r'.poslit + 1$
      if $r'.poslit = 1$ and $r'.neglit = 0$ **then**
        push(r', $queue_2$)
      endif
    end for
  endif
end while
$queue :=$ empty
for each atom $a \in N$ **do**
  if there exists a rule $r \in a.headlist$ such that $r.poslit = 0$,
    and $r.neglit = 0$ **then**
    push(r, $queue$)
  endif
end for
dcl($queue, P$)

Figure 8: The Procedure for Computing Diminishing Deductive Closures

its body, the counter *poslit* is incremented. Finally, those rules whose *poslit* counters are thus set to one and whose *neglit* counters are zero are added to the queue as they no longer imply that their heads should belong to the closure.

The problem with this approach is that too many atoms are possibly removed from the closure. When the algorithm removes an atom from the closure, it removes the atom even if the atom is the head of a rule which is active in the final closure. To remedy this problem we store the atoms removed from the closure in a temporary set N and then use the first algorithm to add the atoms wrongly removed from the closure to the closure. We can not abstain from removing atoms which are heads of active rules, as we do not know if the rules will still be active in the final closure.

We give a simple example demonstrating this. Let P be the set of rules $\{a \leftarrow b;\ b \leftarrow a;\ b \leftarrow \mathrm{not}(c)\}$. If we have computed the closure $\mathrm{Dcl}(P, \{\mathrm{not}(c)\})$ and want to compute $\mathrm{Dcl}(P, \emptyset)$, we remove $\mathrm{not}(c)$. This makes the rule $b \leftarrow \mathrm{not}(c)$ inactive. Now, if we at this point check if b is the head of an active rule we find that it is. The rule $b \leftarrow a$ is still active. Thus, if we for this reason do not remove b we see that both a and b remain in the closure. However, as $\mathrm{Dcl}(P, \emptyset) = \emptyset$, this is not correct.

The part of the algorithm that propagates the fact that atoms should be removed from the closure works in linear time in the size of the set of rules P, as each rule is added to the queue and each atom is removed from the closure at most once. The part that adds rules, whose heads have been wrongly removed from the closure, to the queue *queue* is also linearly bounded by the size of P, as the size of N is bounded by the number of atoms and as the number of rules in the *headlist* lists are bounded by the number of rules in P. Thus, the time complexity of the algorithm $\mathbf{dcl}_2$ is linear in the size of P as the time complexity of **dcl** is linear in the size of P.

If $\mathrm{Dcl}(P, B)$ has been computed and we want to compute $\mathrm{Dcl}(P, B')$, we initialize the queue with the rules whose bodies are entirely contained in B but not contained in B'. The second algorithm will then compute the new closure correctly. This method of computing the deductive closure is, however, not always efficient. There is always the possibility that the second algorithm will remove all atoms from the closure before calling the first algorithm. In this case it would be better to only call the first algorithm, provided that the counters *poslit* and *neglit* can be initialized with precomputed values.

5 Logic Programs with Total Well-Founded Models

We will now examine the computation of stable models of sets of rules with a total well-founded model. We are going to prove that $\mathbf{expand}_2(P, \emptyset)$ in a sense computes the well-founded model of P. We begin with a definition and a theorem that together characterize the set $\mathbf{U}_P(I)$.

Definition 5.1 *Let P be a logic program and let I be an interpretation. Define the set of potentially founded atoms in P with respect to the interpretation I as* $\mathrm{Potfound}(P, I) = \bigcup_{i=0}^{\infty} \mathrm{Potf}(P, I)^i$, *where*

$$\begin{aligned}
\mathrm{Potf}(P,I)^0 &= \{a \mid a \textit{ is the head of a rule } r \in P \textit{ whose body contains} \\
&\quad \textit{only } \mathrm{not}\textit{-atoms, and if } \mathrm{not}(b) \textit{ is in the body of } r \\
&\quad \textit{then } b \notin I\} \\
\mathrm{Potf}(P,I)^{i+1} &= \mathrm{Potf}(P,I)^i \cup \{a \mid a \textit{ is the head of a rule } r \in P, \textit{ if } b \textit{ is in} \\
&\quad \textit{the body of } r \textit{ then } \mathrm{not}(b) \notin I \textit{ and } b \in \mathrm{Potf}(P,I)^i, \textit{ and} \\
&\quad \textit{if } \mathrm{not}(b) \textit{ is in the body of } r \textit{ then } b \notin I\}
\end{aligned}$$

Example 5.1 Let P be the set of rules

$$\{a \leftarrow \mathrm{not}(b);\ b \leftarrow \mathrm{not}(a), \mathrm{not}(c);\ d \leftarrow a;\ c \leftarrow\}.$$

Then, $\mathrm{Potf}(P, \{c\})^0 = \{a, c\}$ and $\mathrm{Potf}(P, \{c\})^i = \{a, c, d\}$ for $i > 0$. Hence, $\mathrm{Potfound}(P, \{c\}) = \{a, c, d\}$. □

Theorem 5.2 ([BF91]) *Let P be a logic program and let I be an interpretation. Then,* $\mathbf{U}_P(I) = \mathrm{not}(\mathrm{Atoms}(P) - \mathrm{Potfound}(P, I))$.

Consider the sequence of interpretations defining the well-founded model of the program P. Recall that $I_0 = \emptyset$ and for all $k \geq 0$, $I_{k+1} = \mathbf{T}_P(I_k) \cup \mathbf{U}_P(I_k)$.

Lemma 5.3 *Let P be a logic program, let I^* be the well-founded model of P, and let $B = \mathbf{expand}_2(P, \emptyset)$. Then, the well-founded model*

$$I^* \subseteq \mathrm{Dcl}(P, B) \cup \mathrm{not}(\mathrm{Atoms}(P) - \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))).$$

Proof. We prove by induction on k that for every $k = 0, 1, \ldots$, $I^k \subseteq \mathrm{Dcl}(P, B) \cup \mathrm{not}(\mathrm{Atoms}(P) - \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B)))$.

Note that $I_0 \subseteq \mathrm{Dcl}(P, B) \cup \mathrm{not}(\mathrm{Atoms}(P) - \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B)))$ as $I_0 = \emptyset$. Induction hypothesis: assume that $I_k \subseteq \mathrm{Dcl}(P, B) \cup \mathrm{not}(\mathrm{Atoms}(P) - \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B)))$.

If $a \in \mathbf{T}_P(I_k)$, then there exists a rule r such that the body of r is in I_k. If b is in the body of r, then $b \in I_k$ and by the induction hypothesis $b \in \mathrm{Dcl}(P, B)$. If $\mathrm{not}(b)$ is in the body of r, then $\mathrm{not}(b) \in I_k$ and by the induction hypothesis $b \notin \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$, which by the definition of $\mathbf{expand}_2$ implies that $\mathrm{not}(b) \in B$ as $\mathrm{Dcl}(P, B) \subseteq \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$. Thus, the body of r is in $\mathrm{Dcl}(P, B) \cup B$ which implies $a \in \mathrm{Dcl}(P, B)$. Hence, $\mathbf{T}_P(I_k) \subseteq \mathrm{Dcl}(P, B)$.

If $\mathrm{not}(a) \in \mathbf{U}_P(I_k)$, then $a \in \mathrm{Atoms}(P) - \mathrm{Potfound}(P, I_k)$ by Theorem 5.2, which in turn implies $a \notin \mathrm{Potfound}(P, I_k)$. If $b \in \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$, then $b \notin \mathrm{Atoms}(P) - \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$ which implies $\mathrm{not}(b) \notin I_k$. On the other hand, if $\mathrm{not}(b) \in \mathrm{not}(\mathrm{NAnt}(P) - B)$, then $b \notin B$ which implies $b \notin \mathrm{Dcl}(P, B)$. Thus, $b \notin I_k$ by the induction hypothesis. Now, define $A^0 = \emptyset$ and $A^{i+1} = A^i \cup \{a \mid a$ is the head of a rule $r \in P$, if b is in the body of r then $b \in A$ and if $\mathrm{not}(b)$ is in the body of r then $\mathrm{not}(b) \in \mathrm{not}(\mathrm{NAnt}(P) - B)\}$. Clearly, $\mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B)) = \bigcup_{i=0}^{\infty} A^i$. Furthermore, $A^0 \subseteq \mathrm{Potf}(P, I_k)^0$ and if $A^i \subseteq \mathrm{Potf}(P, I_k)^i$ then $A^{i+1} \subseteq \mathrm{Potf}(P, I_k)^{i+1}$ by the definition of $\mathrm{Potf}(P, I_k)^{i+1}$. Therefore, induction shows that $a \in \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$ implies $a \in \mathrm{Potfound}(P, I_k)$ or in other words $\mathbf{U}_P(I_k) \subseteq \mathrm{not}(\mathrm{Atoms}(P) - \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B)))$.

Hence, $I_{k+1} \subseteq \mathrm{Dcl}(P, B) \cup \mathrm{not}(\mathrm{Atoms}(P) - \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B)))$ and induction proves that the well-founded model $I^* = \bigcup_{k=0}^{\infty} I_k$ is a subset of $\mathrm{Dcl}(P, B) \cup \mathrm{not}(\mathrm{Atoms}(P) - \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B)))$. □

Lemma 5.4 *Let P be a logic program, let I^* be the well-founded model of P, and let $B = \mathbf{expand}_2(P, \emptyset)$. Then, the set of literals*

$$\mathrm{Dcl}(P, B) \cup \mathrm{not}(\mathrm{Atoms}(P) - \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))) \subseteq I^*.$$

Proof. We prove by induction on i that for every $i = 0, 1, \ldots$, $\mathrm{Potf}(P, I^*)^i \subseteq \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$.

If $a \in \mathrm{Potf}(P, I^*)^0$, then there is a rule r with head a such that there is no atoms in its body and if the not-atom $\mathrm{not}(b)$ is in its body then $b \notin I^*$, which implies $b \notin B$ by Lemma 5.3, which in turn implies $\mathrm{not}(b) \in \mathrm{not}(\mathrm{NAnt}(P) - B)$. Thus, $a \in \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$ and $\mathrm{Potf}(P, I^*)^0 \subseteq \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$.

Induction hypothesis: assume that $\mathrm{Potf}(P, I^*)^i \subseteq \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$.

If $a \in \mathrm{Potf}(P, I^*)^{i+1}$, then there is a rule r with head a such that if b is in the body of r, then $b \in \mathrm{Potf}(P, I^*)^i$ and if $\mathrm{not}(b)$ is in the body of r, then $b \notin I^*$. Now, $b \in \mathrm{Potf}(P, I^*)^i$ implies $b \in \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$ by the induction hypothesis and $b \notin I^*$ implies $\mathrm{not}(b) \in \mathrm{not}(\mathrm{NAnt}(P) - B)$ as in the case $i = 0$. Thus, $a \in \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$ and therefore $\mathrm{Potf}(P, I^*)^{i+1} \subseteq \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$. Hence, induction proves that $\mathrm{Potfound}(P, I^*) \subseteq \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$.

Therefore, $\mathrm{not}(\mathrm{Atoms}(P) - \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))) \subseteq \mathbf{U}_P(I^*)$ by the definition of $\mathbf{U}_P(I^*)$.

We prove that $\mathrm{Dcl}(P, B)$ is a subset of I^* is a loop invariant of the function $\mathbf{expand}_2(P, \emptyset)$. If $B' \subseteq B$ is a subset of I^*, then $\mathrm{Dcl}(P, B')$ is also a subset of I^*, as I^* is closed under the transformation $\mathbf{T}_P(I^*)$. Furthermore, $\mathrm{not}(\mathrm{NAnt}(P) - \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B')))$ is a subset of $\mathrm{not}(\mathrm{Atoms}(P) - \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B)))$ which is a subset of $\mathbf{U}_P(I^*)$. Thus, $B \subseteq I^*$ as $I^* = \mathbf{T}_P(I^*) \cup \mathbf{U}_P(I^*)$. Consequently, $\mathrm{Dcl}(P, B) \subseteq I^*$.

Hence, if $B = \mathbf{expand}_2(P, \emptyset)$, then the inclusion $\mathrm{Dcl}(P, B) \cup \mathrm{not}(\mathrm{Atoms}(P) - \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))) \subseteq I^*$ holds. □

Theorem 5.5 *Let P be a logic program, let I^* be the well-founded model of P, and let $B = \mathbf{expand}_2(P, \emptyset)$. Then, the well-founded model*

$$I^* = \mathrm{Dcl}(P, B) \cup \mathrm{not}(\mathrm{Atoms}(P) - \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))).$$

Proof. Immediate from lemmas 5.3 and 5.4. □

Theorem 5.6 *Let I^* be the total well-founded model of the logic program P. Let the set $B = \mathbf{expand}_2(P, \emptyset)$. Then, the program P has a unique stable model $\Delta = \mathrm{Dcl}(P, B)$.*

Proof. Assume there is an atom $a \in \mathrm{NAnt}(P)$ such that a is not covered by B. Thus, $a \notin \mathrm{Dcl}(P, B)$ and $a \in \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$ by the definition of B. But $a \in \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$ implies $\mathrm{not}(a) \notin \mathrm{not}(\mathrm{Atoms}(P) - \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B)))$, which by Lemma 5.3 implies that a is not covered by I^*. This is a contradiction as I^* is a total well-founded model of P. Hence, $\mathrm{NAnt}(P)$ is covered by B.

Now, for $a \in \mathrm{NAnt}(P)$, $\mathrm{not}(a) \notin B^-$ implies $a \in B$ which implies $a \in \mathrm{Dcl}(P, B^-)$. Furthermore, $a \in \mathrm{Dcl}(P, B^-)$ implies $a \in B$ which in turn implies $\mathrm{not}(a) \notin B^-$. Thus, the set B^- is P-full and $\Delta = \mathrm{Dcl}(P, B)$ is a stable model of P.

Finally, Δ is the only stable model of P as the fact that $\mathbf{expand}_2(P, \emptyset)$ covers $\mathrm{NAnt}(P)$ implies that $\mathbf{smodels}_2(P, \emptyset)$ cannot invoke itself recursively and thus passes at most one stable model to **test** and as Theorem 3.8 guarantees that the function $\mathbf{smodels}_2$ is complete. □

Corollary 5.7 *The function $\mathbf{expand}_2$ computes the full set of all effectively stratified [BF91], weakly stratified [PP88], locally stratified [Prz88], and stratified [CH85] logic programs.*

Proof. The classes of stratified, locally startified, and weakly stratified logic programs are included in the class of effectively stratified logic programs and a logic program is effectively stratified if and only if it has a total well-founded model [BF91]. □

We now consider the complexity of the computation in $\mathbf{expand}_2(P, \emptyset)$. The initialization of the arrays used when computing the deductive closures $\mathrm{Dcl}(P, B)$ and $\mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$ can be done in linear time. As the deductive closure $\mathrm{Dcl}(P, B)$ grows during the computation and as each atom in $\mathrm{NAnt}(P)$ only can be added once to B, the total work done computing $\mathrm{Dcl}(P, B)$ is linear in the size of P. However, the complexity of calculating $\mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$ is linear in the size of P and as the number of times it is calculated is bounded by the size of $\mathrm{NAnt}(P)$, we see that the total complexity of computing the stable model of a set of rules with a total well-founded model is at most quadratic in the size of the set of rules. More precisely, if the size of the set of rules is p and the number of atoms in $\mathrm{NAnt}(P)$ is n then the time complexity of $\mathbf{expand}_2$ is bounded by

$$\mathcal{O}(np) \tag{10}$$

It is possible to compute the unique stable model of a stratified set of rules in linear time with respect to the size of the set [NR94]. The function $\mathbf{expand}_2$ fails to do this as the computations of $\mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$ might add and remove an atom from the deductive closure several times.

6 Complexity Analysis

We give a naive upper bound on the time complexity of the improved algorithm **smodels**$_2$. Let P be a set of rules, let the size of P be p, and let the number of atoms in $\mathrm{NAnt}(P)$ be n. Disregarding the **expand**$_2$ function we see that **smodels**$_2$ have to add literals to the set B at most $\Sigma_{i=1}^{n} 2^i = 2^{n+1} - 2$ times. The function must remove them equally many times. As the time complexity of **expand**$_2$ is $\mathcal{O}(np)$ we see that the time complexity of **smodels**$_2$ is

$$\mathcal{O}(np2^n) \tag{11}$$

We continue with an example showing that the algorithm has worst-case exponential time complexity.

Example 6.1 Let P be the set of rules

$$\begin{aligned} \{h \leftarrow h, \mathrm{not}(a_1), \ldots, \mathrm{not}(a_n);\ h \leftarrow a_1, \ldots, a_n; \\ h \leftarrow \mathrm{not}(h);\ a_1 \leftarrow h;\ \cdots;\ a_n \leftarrow h\} \end{aligned}$$

If the set B does not contain h, $\mathrm{not}(h)$, nor all of $a_1, \ldots, a_n$, then $\mathrm{Dcl}(P, B) = \emptyset$ and $\mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B)) = \mathrm{Atoms}(P)$. Consequently **expand**$_2(P, B) = B$ and **conflict**$_2(P, B)$ returns false. If **smodels**$_2$ chooses the atoms $a_1, \ldots, a_n$ before choosing the atom h, then the algorithm adds literals to the set B, by choosing or backtracking, a total of

$$\Sigma_{i=1}^{n+1} 2^i - 1 = 2^{n+2} - 3$$

times.

When an atom is chosen in **smodels**$_2$ it is first added to B^- and then upon backtracking added to B^+. Thus, the first atom chosen is added two times to B. The second is added two times to B for each time the first atom is added to B. Similarly, every atom is added two times to B for each time the previously chosen atom is added to B. This is the reason for the sum $\Sigma_{i=1}^{n+1} 2^i$. However, since the atom h is added to B by the rule $h \leftarrow a_1, \ldots, a_n$ when all of the atoms $a_1, \ldots, a_n$ are in B^+, the literal $\mathrm{not}(h)$ is not added to B^- prior to the addition of h to B^+. Hence, we have to decrease the sum by one. $\square$

7 Backtracking Based on the Dependency Graph

The algorithm **smodels**$_2$ backtracks in a simple manner, the atoms added last to B are removed first. This straightforward approach has, however, a weakness. The algorithm may repeatedly backtrack to points that do not influence the subsequent computations. Identical computations will then be performed several times. We examine the use of the dependency graph as a means to improve the way the algorithm backtracks.

In Section 7.1 we define the dependency graph and in Section 7.2 we examine a backtracking strategy based on the dependency graph.

7.1 The Dependency Graph

Definition 7.1 *A dependency graph $\mathcal{D}$ of a set of rules P is a pair (A, D), where $A = \mathrm{Atoms}(P)$ and D is a subset of $A \times A$ for which the following holds. The pair $(a, b) \in D$ if and only if there exists a rule $r \in P$ such that a is in the head of r and b is in the body of r.*

We call an edge (a, b) in the dependency graph a not-edge if there is a rule r such that a is in the head of r and $\mathrm{not}(b)$ is in the body of r.

Definition 7.2 *An atom a depends on another atom b in a dependency graph $\mathcal{D}$ if there is a directed path from a to b in the graph.*

Definition 7.3 *A strongly connected component A is dependent on another strongly connected component B in a dependency graph $\mathcal{D}$ if there is a directed path from some atom in A to any atom in B in the graph.*

We also say that a literal depends on another literal if the atom appearing in the first literal is dependent on the atom in the second literal.

7.2 Backtracking and the Dependency Graph

We demonstrate some problems one encounters if one wants to use the dependency graph as an aid in backtracking. Think of a set of rules P. Let a and b be two atoms in the set $\mathrm{NAnt}(P)$ such that a is dependent on b but b is not dependent on a. Let B be a set of literals and consider the deductive closure $\mathrm{Dcl}(P, B)$. Now, b might be in the deductive closure or it might not. This fact is, however, not dependent on whether a is covered by B or not. One is thus led to believe that the dependency graph can be used when improving the way the algorithm backtracks. The idea would be to compute the strongly connected components of the dependency graph and choose the atoms from one component at a time. The algorithm would, for instance, choose a strongly connected component that is not dependent on any component that is not fully covered by the partially constructed P-full set. If the algorithm then had to backtrack, it would backtrack normally inside the strongly connected components, but use the dependency graph when choosing the next component to backtrack in. This would hopefully help the algorithm to avoid recomputing parts of the P-full set.

Definition 7.4 *Let P be a set of rules, let B be a set of atoms. Define* $\mathrm{Rules}_P(B) = \{r \in P \mid \textit{the head of } r \textit{ is in } B\}$.

Consider a set of rules P such that the dependency graph consists of two separate strongly connected components; A and B. Assume that $\mathrm{Rules}_P(A)$ has two stable models and that $\mathrm{Rules}_P(B)$ has no stable model. If the algorithm starts with computing one stable model of $\mathrm{Rules}_P(A)$ and then fails to compute a stable model of $\mathrm{Rules}_P(B)$, it is clear that P has no stable models. The subsequent backtracking and computation of the second stable model of $\mathrm{Rules}_P(A)$ are therefore unnecessary.

On the other hand, assume that $\mathrm{Rules}_P(B)$ also has two stable models. Assume the algorithm computes one stable model of $\mathrm{Rules}_P(A)$ and then computes the first and second stable models of $\mathrm{Rules}_P(B)$, and finally fails to compute a third stable model of $\mathrm{Rules}_P(B)$. Then, the algorithm has to backtrack and compute the second stable model of $\mathrm{Rules}_P(A)$ in order to find all stable models of P.

The above example makes it evident that it is not possible to implement a simple backtracking method that only exploits the way the strongly connected components in the dependency graph are connected. The problem lies in guaranteeing completeness of the algorithm. If the algorithm is not complete, it might not compute all stable models, thereby failing to compute the single stable model of some sets of rules. Examples of sets of rules showing this are, however, dependent on the details of the backtracking strategy. The previous example demonstrates the possibility of using the dependency graph for early detection of sets of rules with no stable models.

A backtracking scheme that is only based on the strongly connected components in the dependency graph, presumes that the heuristic choosing atoms not covered by $\mathrm{NAnt}(P)$, chooses all atoms in one component before choosing any in another component. Consider a set of rules P with two strongly connected components A and B such that A is dependent on B, but B is not dependent on A. As the backtracking scheme uses the dependency graph, one might think that it is beneficial to choose the atoms in B before the atoms in A. But this is not always the case. To mention a simple example showing this, assume that B has a lot of stable models and that A has none. Now, if the algorithm begins with computing a stable model of B, it will have to compute all stable models of B before arriving at the conclusion that P has no stable models. If the algorithm instead tries to compute a stable model of A, it might very well be the case that it fails immediately, thereby avoiding a huge amount of computation.

8 Handling Rules with Variables

In this section we examine a class of logic programs with variables that are easily grounded, i.e., converted to propositional logic programs without variables. The reason for allowing the use of variables is to simplify the writing of logic programs. The reason for limiting this use is to avoid the floundering problems of ungrounded negative literals [She88]. If the use of variables is suitably limited it is also possible to avoid the generation of grounded rules that do not influence the set of stable models of the program. In Section 8.1 we describe logic programs with variables and how they are grounded. In Section 8.2 we describe the class of range restricted programs and how they can be grounded.

8.1 Ground Logic Programs

We begin by defining logic programs with variables. A logic program with variables is a set of rules with variables. A rule with variables is a rule that contain apart from atoms also predicates. A predicate is of the form $p(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are variables or atoms. A not-predicate is, of course, of the form $\mathrm{not}(p(X_1, \ldots, X_n))$.

A substitution is a mapping from the variables in the logic program to the atoms in the logic program. A substitution is applied to a rule by replacing every variable in the rule by the atom it is mapped to. A program is grounded by replacing each rule r by the rules obtained by applying all possible substitutions of the program to the rule r.

Example 8.1 Let P be the logic program

$$\{p(X) \leftarrow q(X,Y);\ r(a) \leftarrow r(b)\}$$

where variables are denoted by uppercase letters. The corresponding ground program is

$$\{p(a) \leftarrow q(a,a);\ p(a) \leftarrow q(a,b);\ p(b) \leftarrow q(b,a);\ p(b) \leftarrow q(b,b);\ r(a) \leftarrow r(b)\}$$

□

The predicates of a ground logic program can be considered as atoms. This allows us to interpret a ground logic program as an ordinary logic program without predicates.

8.2 Range Restricted Logic Programs

When a logic program is grounded, a lot of rules are typically generated that may be removed from the ground program without changing the set of stable models of the program. By limiting the use of variables in the rules, it is possible to generate a smaller program that has the same stable models as the ground program. We examine the class of range restricted logic programs.

A rule containing variables is range restricted if every variable that occurs in a predicate in the head or in a not-predicate in the body of the rule also occurs in a predicate in the body of the rule. A logic program is range restricted if every rule in the program is range restricted.

Example 8.2 The rules $a(X) \leftarrow b(X), \mathrm{not}(c)$ and $p \leftarrow q(X), \mathrm{not}(r(X))$ are range restricted, but the rules $a(X) \leftarrow b(Y)$ and $p \leftarrow \mathrm{not}(q(X))$ are not. □

All rules in a ground logic program do not affect the set of stable models of the program.

Proposition 8.1 *Let P be a ground logic program. Then, any rule in P that has its head or an atom in its body outside of* $\mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P)))$ *can be removed from P without affecting the set of stable models of P.*

Proof. For any stable model Δ of P, $\Delta = \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - \Delta)) \subseteq \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P)))$. □

In Figure 9 we present an algorithm that directly computes the smaller ground program from the ungrounded one.

The algorithm **ground**(P) works by computing the deductive closure of P while disregarding the not-atoms and not-predicates in the rules. When a ground predicate is found to belong to the closure it is put in the set A. The predicates in the set A are then used to ground the rules in P. When a predicate has been used it is moved from the set A to the set B to avoid using the same predicate twice. The ground rules in P are moved to the set P_2 as they are created. When the set A is empty, it is not possible to ground any more rules and the algorithm returns the grounded program P_2.

Example 8.3 The program

$$P = \{p(X) \leftarrow q(X,Y);\ r(a) \leftarrow r(b)\}$$

in the example in the previous section is range restricted. The ground program contains five rules, but the smaller program **ground**(P) does not contain any rules. Both the empty program and the ground program have one stable model which is empty. □

```
function ground(P)
P_2 := {r ∈ P | the body of r does not contain any positive predicates}
P := P − P_2
A := {a | there exists a rule r ∈ P_2 whose head is a}
B := ∅
while A is not empty do
    Let a ∈ A.
    A := A − {a}
    B := B ∪ {a}
    for each rule r in P do
        for every rule r′ resulting from applying a substitution
            to r such that a is in the head or body of the rule r′ do
            if  r′ is grounded and all positive literals
                in the body of r′ is in A ∪ B then
                P_2 := P_2 ∪ {r′}
                if  the head of r′ is not in B then
                    A := A ∪ {the head of r′}
                endif
            else
                P := P ∪ {r′}
            endif
        end for
    end for
end while
return P_2
```

Figure 9: The Procedure for Grounding Range Restricted Programs

Even if the algorithm **ground** can reduce the number of rules in the ground program, the program might still grow substantially when grounded. For example, the program

$$P = \{p(X_1, \ldots, X_m) \leftarrow q(X_1), \ldots, q(X_m);\ q(a_1) \leftarrow; \cdots;\ q(a_n) \leftarrow\}$$

contains $n+1$ rules and the program $\mathbf{ground}(P)$ contains $n + n^m$ rules.

9 Implementation

In this section we describe the implementation specific choices we made to ensure an efficient implementation of the $\mathbf{smodels}_2$ algorithm. In particular, this concerns the choice of data structures, the implementation of the deductive closures, the detection of conflicts, and the way backtracking is handled. As the algorithm contains a non-deterministic choice, we describe a heuristic for making a hopefully good one. In Section 9.1 we describe the data structures. In Section 9.2 we describe the implementation of the two deductive closures needed and how conflicts are detected. Finally, in Section 9.3 we describe how we handle backtracking and in Section 9.4 we describe the heuristic.

9.1 Data Structures

The algorithm $\mathbf{smodels}_2(P, B)$ relies heavily on the use of sets. It is therefore necessary to implement these efficiently. A closer inspection of the algorithm reveals that the set operations needed are addition or removal of an element from a set and membership test. In addition, we must be able to determine if the set $\mathrm{NAnt}(P)$ is covered by the sets B^+ and B^-.

As we know all possible members of the sets, we implement the sets as arrays. The elements are represented by indices into the arrays and the value of the array at index n tells if element n belongs to the set or not. If the value is zero, the element does not belong to the set and if the value is one it does.

The particular problem of determining if $\mathrm{NAnt}(P)$ is covered by the sets B^+ and B^- is solved by introducing a variable that keeps count of the number of elements in $\mathrm{NAnt}(P)$ not covered by B^+ and B^-. The variable is initialized with the number of elements in $\mathrm{NAnt}(P)$ not covered by B. Then, the variable is decremented or incremented each time an element is added to or removed from the sets B^+ and B^-. When the variable has value zero, $\mathrm{NAnt}(P)$ is covered.

9.2 Deductive Closures and Conflicts

The computation of the deductive closures is to a great extent described in Section 4. We fill in some details regarding the implementation of the deductive closures. We also describe how to detect conflicts as early as possible.

The rules are represented as indices into an array. To each rule we must associate the head of the rule, the rules in whose bodies the head appears, and the two counters *poslit* and *neglit*. The value of an element of the array a rule points into, therefore contains the head of the rule and the counters *poslit* and *neglit*. But as the number of rules a particular head appears in varies for different heads, we store the rules which has a particular head in their bodies consecutively in a different array. The beginning and the end of this segment of the array is then stored in the same element as the head.

We want to reduce the amount of computation as much as possible. Therefore, we make the conflict test each time an atom or not-atom is added to B in the **expand**$_2$ function. This incurs very little overhead as the deductive closures needed already are calculated when the literal is added to the set B and as we only have to test the literal being added. Thus, we gain the ability to interrupt the computations of the deductive closures and the **expand**$_2$ function as soon as a conflict is detected.

9.3 Backtracking

The algorithm **smodels**$_2$ handles backtracking by recursively passing along the changing set B as one of its parameters. This practice would in an implementation require a lot of memory. To avoid this inconvenience we use iteration instead of recursion and use a stack to record the information necessary for backtracking.

When a literal is added to B^- or B^+ we push it onto the stack. If the literal is added to B^- as the result of a choice in $\mathrm{NAnt}(P)$, we mark it. The state of the arrays used when computing the deductive closures must be reset appropriately when backtracking. We avoid storing any backtracking information pertaining to these arrays. Instead, we take advantage of our ability to compute both growing and diminishing deductive closures. When a literal is removed from the stack during backtracking we modify the queues used by the deductive closure functions accordingly. Then, we call the opposite deductive closure function to what is normally called to reset the state of the arrays. For example, when we compute $\mathrm{Dcl}(P, B)$ we call the function computing the growing deductive closure. During backtracking we therefore call the function computing the diminishing deductive closure.

When backtracking we remove literals from the stack until a marked literal has been removed. This not-atom is then added as an atom to the set B after the states of the deductive closures have been reset.

9.4 A Heuristic

The order the atoms are picked from $\mathrm{NAnt}(P)$ drastically affect the performance of the stable models algorithm. If only atoms that appear in a specific stable model are chosen, the algorithm finds the first stable model without backtracking. It is therefore important to use a good heuristic when selecting atoms.

The heuristic selects a not-atom from the body of an applicable rule with the least number of not-atoms not in B^- in its body. An applicable rule is a rule that might change the set B, i.e., the head of an applicable rule is not in $\mathrm{Dcl}(P, B)$, all atoms in the body of an applicable rule are in $\mathrm{Dcl}(P, B)$, and no not-atom in the body of an applicable rule covers any atom in B^+. Thus, the heuristic tries to select the atom that has the greatest possibility of changing B, thereby helping the algorithm to find conflicts or full sets with a minimal number of choices. If there are no applicable rules left, the atoms in $\mathrm{NAnt}(P)$ not covered by B can not affect the deductive closures. Therefore, they are immediately added as not-atoms to B^-. We give an explanation of this.

Adding not-atoms to B does not change the set $\mathrm{not}(\mathrm{NAnt}(P) - B)$ nor the deductive closure $\mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$. Adding not-atoms to B does not change $\mathrm{Dcl}(P, B)$ either, as there are no applicable rules left. Let B' denote the set B to which all atoms in $\mathrm{NAnt}(P)$ not covered by B have been added as not-atoms. Then, $\mathrm{Dcl}(P, B) = \mathrm{Dcl}(P, B')$ and $\mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B)) = \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B'))$ which guarantees that $\mathbf{expand}_2$ does not change B'. Now, $\mathbf{conflict}_2(P, B)$ returns false as otherwise the not-atoms would not be added to B by the heuristic. Thus, $\mathbf{conflict}_2(P, B')$ also returns false and B' is a P-full set. But this means that $\mathrm{Dcl}(P, B') = \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B'))$ and consequently that $\mathrm{Dcl}(P, B) = \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - B))$. Hence, Lemma 3.4 implies that there is only one stable model agreeing with B, which in turn implies that no atom in $\mathrm{NAnt}(P)$ can be added to B without introducing a conflict. The addition of the not-atoms to B is therefore justified.

Example 9.1 Consider the rule $a \leftarrow b, \mathrm{not}(c)$ in the set of rules P. If the atom b is not in $\mathrm{Dcl}(P, B)$, then adding $\mathrm{not}(c)$ to B will not affect whether a belongs to $\mathrm{Dcl}(P, B)$ or not. On the other hand, if b is in $\mathrm{Dcl}(P, B)$ and a is not, then adding $\mathrm{not}(c)$ to B will make a a member of $\mathrm{Dcl}(P, B)$. □

The heuristic can be implemented by examining the rules that have not-atoms in their bodies that are not covered by B. The counter *poslit* provides an easy way of detecting whether all atoms in the body of a rule are in $\mathrm{Dcl}(P, B)$. Hence, computing the heuristic is linear in the number of rules in the logic program in the worst case.

We examine the example in Section 6 that was used to show that the algorithm can exhibit worst-case exponential time complexity. The set of rules P is

$$\{h \leftarrow h, \mathrm{not}(a_1), \ldots, \mathrm{not}(a_n);\ h \leftarrow a_1, \ldots, a_n;\\ h \leftarrow \mathrm{not}(h);\ a_1 \leftarrow h;\ \cdots;\ a_n \leftarrow h\}$$

Regardless of the value of n the heuristic will first choose to add $\mathrm{not}(h)$ to the set B. The rule $h \leftarrow \mathrm{not}(h)$ immediately gives rise to a conflict. Then, the literal $\mathrm{not}(h)$ is removed from the set B and the atom h is added. Now, all the atoms $a_1, \ldots, a_n$ are added to B, a conflict is found and the algorithm returns false. Hence, the algorithm no longer needs exponential time to deduce that P has no stable models.

10 Comparison with other Algorithms

We compare the algorithm described in this work with some other algorithms for computing stable models of logic programs. In Section 10.1 we examine the branch and bound algorithm of [SNV95], in Section 10.2 the SLG system of [CW93, CW94], and in Section 10.3 the mixed integer programming methods of [BNNS94]. Finally, we present the comparison in Section 10.4.

10.1 Branch and Bound

The branch and bound algorithm of [SNV95] computes the stable models of ground logic programs. The algorithm works in two stages. In the first stage the logic program is simplified while the algorithm computes the well-founded semantics of the program. In the second stage, the branch and bound stage, all stable models of the logic program are constructed.

The algorithm computes the stable models by generating smaller and smaller programs from programs already generated, starting with the simplified program. Two new programs are generated from a program P by choosing an atom whose truth value is unknown according to the well-founded semantics and then simplifying the program P assuming that the atom belongs or do not belong to the stable model under construction. The algorithm prunes the search space by disregarding every newly created program that is inconsistent or whose partial stable model is a superset of a stable model already found.

Although the algorithm prunes the search space effectively, it must keep all constructed stable models, as well as, all partially constructed stable models in memory. This indicates that the algorithm will necessarily perform badly if the number of stable models is large. Even if the number of stable models is small, there is no guarantee that the number of partially constructed models also stays small. The algorithm is therefore not suitable if one wants to find one or maybe a few stable models of a ground logic program.

10.2 The SLG System

The SLG system [CW93, CW94] supports goal-oriented query evaluation of logic programs under the well-founded semantics. The system simplifies the logic program during query evaluation producing a residual program. The SLG System can then compute the stable models of the residual program using an *assume-and-reduce* algorithm, provided that all negative literals in the residual program are ground.

The *assume-and-reduce* algorithm constructs the stable models of a logic program in a similar fashion to the branch and bound algorithm. The *assume-and-reduce* algorithm differs in that it constructs one model at a time, finding all models by backtracking, and in that it does not enlarge the partially constructed stable model using the well-founded semantics. The algorithm alternately assumes the truth values of atoms in the program and reduces the logic program. The reduction of the program that at the same time enlarges the model under construction does not simplify the program as much as is possible if one uses the well-founded semantics. This means that the algorithm does not prune the search space as effectively as the branch and bound algorithm. However, the *assume-and-reduce* algorithm can derive whether an atom belongs to the stable model or not using backward propagation in some specialized circumstances. This improves the pruning of the search space slightly.

10.3 The Mixed Integer Programming Approach

The mixed integer programming methods of [BNNS94] computes the stable models of a logic program by translating the program into an integer linear program. The integer linear program is then used to compute all minimal models of the logic program. The minimal models are then tested by another integer linear program to exclude those models that are not stable. As the number of minimal models of a logic program can be very large compared to the number of stable models of the same program, we conclude that this approach is very inefficient for that class of logic programs.

10.4 Comparison

We compare the branch and bound algorithm, the SLG system, and the mixed integer programming methods with the improved algorithm presented in this work. The performance of the branch and bound algorithm deteriorates, due to the amount of memory needed, when the number of stable models is large. This is not the case with the algorithm **smodels**$_2$ nor with the SLG system. Both the branch and bound algorithm and the SLG system prunes the search space less effectively than **smodels**$_2$. The mixed integer programming approach computes all minimal models of a logic program and then tests if these models are stable. This corresponds to a very weak pruning of the search space. For instance, the set of rules $\{a_i \leftarrow \mathrm{not}(b_i)\}_{i=1}^{n}$ has one stable model but 2^n minimal models.

We conclude that the improved algorithm, as well as, the original algorithm are significantly better than the branch and bound algorithm and the mixed integer programming methods and slightly better than the SLG system. However, experimental results in [NS95] and Section 12 indicate that both **smodels** and **smodels**$_2$ also perform significantly better than the SLG system.

11 Applications

In this section we exhibit some applications of logic programs with the stable model semantics. In Section 11.1 we present an encoding of logic circuits whose stable models represent diagnoses of the circuit. In Section 11.2 we present the encodings of two NP-complete graph problems [GJ79]; the N-coloring and the Hamiltonian circuit problem in undirected graphs. These encodings can be used when evaluating the performance of algorithms computing the stable model semantics. We end with some remarks on the algorithm in Section 11.3.

11.1 Diagnosis

Given a description and some observations of a system, a diagnosis is a conjecture that some minimal set of components are faulty, the rest being normal, that explains the behavior of the system, i.e., that explains the discrepancy between the observations and the correct behavior of the system [Rei87]. We will use this definition to encode logic gates, their connections, and their inputs and outputs as logic programs.

Reiter [Rei87] assumes that the logic encoding the description of the system contains at least the logical connectives conjunction and negation with their normal interpretation. He then encodes the system description as implications; the assumption that a component is not abnormal implies the correct behavior of the component. As we encode logic circuits, it suffices to use propositional logic to describe the circuits.

Definition 11.1 *A system is a pair (SD, C), where SD, the system description, is a finite set of propositional logic sentences and C, the system components, is a finite set of constants. An observation OBS of a system is a finite set of propositional sentences.*

As a rule in a logic program with stable model semantics is not an implication, we cannot use the encoding in [Rei87] directly. We have to encode A implies B and not B implies not A separately.

We proceed with the encoding of the system description of the not-, xor-, and- and or-gates. The not- and xor-gates are easily encoded as three sets of rules. The first set encodes the normal behavior of the gate from the input to the output side. The second set encodes the normal behavior from the output to the input side. Finally, the third set encodes input and output combinations that lead to abnormal behavior of the gate. The rules corresponding to the not-gate and the xor-gate are presented in Figures 10 and 11. We use a notation containing the variable X. The variable X should be replaced by a constant representing a component of the logic circuit in an actual encoding. The resulting construct then denotes a propositional atom. For example, if xor_1 is a constant then $ab(xor_1)$ is a propositional atom that determines whether the component xor_1 is abnormal. The rules are the encoding of a system description where negation is replaced by *not* and the $\leftarrow$ symbol replaces the implication sign.

We encode the and- and or-gates in a similar way to the not- and xor-gates. However, we have to add some rules as some outputs of these two gates does not uniquely determine their inputs. The problem lies in that we would need to use disjunction to encode the and- and or-gates correctly. For instance, if the output of an and-gate is zero, the first or the second input should be zero. We simulate the disjunction by using two rules to determine two different stable models in which one or the other of the inputs is zero. This technique has its drawbacks, however, as it in some cases leads to multiple stable models per diagnosis. The and- and or-gates are described in Figures 12 and 13.

Example 11.1 Let a be an and-gate. Then, the rules that encode the disjunction are

$$\begin{array}{l}
p(a) \leftarrow and(a), out(a,0), \mathrm{not}(q(a)), \mathrm{not}(r(a)), \mathrm{not}(ab(a)) \\
q(a) \leftarrow and(a), out(a,0), \mathrm{not}(p(a)), \mathrm{not}(r(a)), \mathrm{not}(ab(a)) \\
r(a) \leftarrow and(a), out(a,0), in1(a,1), \mathrm{not}(ab(a)) \\
r(a) \leftarrow and(a), out(a,0), in2(a,1), \mathrm{not}(ab(a)) \\
in1(a,0) \leftarrow and(a), p(a) \\
in2(a,0) \leftarrow and(a), q(a)
\end{array}$$

The middle two rules puts $r(a)$ in the stable model if the output is zero and one of the inputs is one. The first two rules determine that either $p(a)$ or $q(a)$ should be in the stable model if the output is zero and if $r(a)$ is not in the stable model, i.e., if neither of the inputs is one. Finally, the last two rules set the first input to one if $p(a)$ is in the stable model and set the second input to one if $q(a)$ is in the stable model. □

The interconnections between different gates are encoded using four rules as follows. Suppose the first input of the and-gate and_1 and the output of the or-gate or_1 are connected. This is encoded using the rules

$$\begin{array}{c}
\{out(or_1,0) \leftarrow in1(and_1,0);\ out(or_1,1) \leftarrow in1(and_1,1); \\
in1(and_1,0) \leftarrow out(or_1,0);\ in1(and_1,1) \leftarrow out(or_1,1)\}.
\end{array}$$

If an input or an output is set to zero or one, we encode this using one rule. Assume that the second input of the and-gate and_1 is zero and that the output

Forward

$$out(X,1) \leftarrow nt(X), in(X,0), \mathrm{not}(ab(X))$$
$$out(X,0) \leftarrow nt(X), in(X,1), \mathrm{not}(ab(X))$$

Backward

$$in(X,1) \leftarrow nt(X), out(X,0), \mathrm{not}(ab(X))$$
$$in(X,0) \leftarrow nt(X), out(X,1), \mathrm{not}(ab(X))$$

Abnormal

$$ab(X) \leftarrow nt(X), in(X,0), out(X,0)$$
$$ab(X) \leftarrow nt(X), in(X,1), out(X,1)$$

Figure 10: Not-gate

Forward

$$out(X,0) \leftarrow xor(X), in1(X,0), in2(X,0), \mathrm{not}(ab(X))$$
$$out(X,1) \leftarrow xor(X), in1(X,0), in2(X,1), \mathrm{not}(ab(X))$$
$$out(X,1) \leftarrow xor(X), in1(X,1), in2(X,0), \mathrm{not}(ab(X))$$
$$out(X,0) \leftarrow xor(X), in1(X,1), in2(X,1), \mathrm{not}(ab(X))$$

Backward

$$in1(X,0) \leftarrow xor(X), in2(X,0), out(X,0), \mathrm{not}(ab(X))$$
$$in1(X,1) \leftarrow xor(X), in2(X,0), out(X,1), \mathrm{not}(ab(X))$$
$$in1(X,1) \leftarrow xor(X), in2(X,1), out(X,0), \mathrm{not}(ab(X))$$
$$in1(X,0) \leftarrow xor(X), in2(X,1), out(X,1), \mathrm{not}(ab(X))$$
$$in2(X,0) \leftarrow xor(X), in1(X,0), out(X,0), \mathrm{not}(ab(X))$$
$$in2(X,1) \leftarrow xor(X), in1(X,0), out(X,1), \mathrm{not}(ab(X))$$
$$in2(X,1) \leftarrow xor(X), in1(X,1), out(X,0), \mathrm{not}(ab(X))$$
$$in2(X,0) \leftarrow xor(X), in1(X,1), out(X,1), \mathrm{not}(ab(X))$$

Abnormal

$$ab(X) \leftarrow xor(X), in1(X,0), in2(X,0), out(X,1)$$
$$ab(X) \leftarrow xor(X), in1(X,0), in2(X,1), out(X,0)$$
$$ab(X) \leftarrow xor(X), in1(X,1), in2(X,0), out(X,0)$$
$$ab(X) \leftarrow xor(X), in1(X,1), in2(X,1), out(X,1)$$

Figure 11: Xor-gate

Forward

$$out(X,0) \leftarrow and(X), in1(X,0), in2(X,0), \mathrm{not}(ab(X))$$
$$out(X,0) \leftarrow and(X), in1(X,0), in2(X,1), \mathrm{not}(ab(X))$$
$$out(X,0) \leftarrow and(X), in1(X,1), in2(X,0), \mathrm{not}(ab(X))$$
$$out(X,1) \leftarrow and(X), in1(X,1), in2(X,1), \mathrm{not}(ab(X))$$

Backward

$$in1(X,1) \leftarrow and(X), out(X,1), \mathrm{not}(ab(X))$$
$$in2(X,1) \leftarrow and(X), out(X,1), \mathrm{not}(ab(X))$$
$$in1(X,0) \leftarrow and(X), in2(X,1), out(X,0), \mathrm{not}(ab(X))$$
$$in2(X,0) \leftarrow and(X), in1(X,1), out(X,0), \mathrm{not}(ab(X))$$
$$p(X) \leftarrow and(X), out(X,0), \mathrm{not}(q(X)), \mathrm{not}(r(X)), \mathrm{not}(ab(X))$$
$$q(X) \leftarrow and(X), out(X,0), \mathrm{not}(p(X)), \mathrm{not}(r(X)), \mathrm{not}(ab(X))$$
$$r(X) \leftarrow and(X), out(X,0), in1(X,1), \mathrm{not}(ab(X))$$
$$r(X) \leftarrow and(X), out(X,0), in2(X,1), \mathrm{not}(ab(X))$$
$$in1(X,0) \leftarrow and(X), p(X)$$
$$in2(X,0) \leftarrow and(X), q(X)$$

Abnormal

$$ab(X) \leftarrow and(X), in1(X,0), out(X,1)$$
$$ab(X) \leftarrow and(X), in2(X,0), out(X,1)$$
$$ab(X) \leftarrow and(X), in1(X,1), in2(X,1), out(X,0)$$

Figure 12: And-gate

Forward

$$out(X,0) \leftarrow or(X), in1(X,0), in2(X,0), \mathrm{not}(ab(X))$$
$$out(X,1) \leftarrow or(X), in1(X,0), in2(X,1), \mathrm{not}(ab(X))$$
$$out(X,1) \leftarrow or(X), in1(X,1), in2(X,0), \mathrm{not}(ab(X))$$
$$out(X,1) \leftarrow or(X), in1(X,1), in2(X,1), \mathrm{not}(ab(X))$$

Backward

$$in1(X,0) \leftarrow or(X), out(X,0), \mathrm{not}(ab(X))$$
$$in2(X,0) \leftarrow or(X), out(X,0), \mathrm{not}(ab(X))$$
$$in1(X,1) \leftarrow or(X), in2(X,0), out(X,1), \mathrm{not}(ab(X))$$
$$in2(X,1) \leftarrow or(X), in1(X,0), out(X,1), \mathrm{not}(ab(X))$$
$$p(X) \leftarrow or(X), out(X,1), \mathrm{not}(q(X)), \mathrm{not}(r(X)), \mathrm{not}(ab(X))$$
$$q(X) \leftarrow or(X), out(X,1), \mathrm{not}(p(X)), \mathrm{not}(r(X)), \mathrm{not}(ab(X))$$
$$r(X) \leftarrow or(X), out(X,1), in1(X,0), \mathrm{not}(ab(X))$$
$$r(X) \leftarrow or(X), out(X,1), in2(X,0), \mathrm{not}(ab(X))$$
$$in1(X,1) \leftarrow or(X), p(X)$$
$$in2(X,1) \leftarrow or(X), q(X)$$

Abnormal

$$ab(X) \leftarrow or(X), in1(X,0), in2(X,0), out(X,1)$$
$$ab(X) \leftarrow or(X), in1(X,1), out(X,0)$$
$$ab(X) \leftarrow or(X), in2(X,1), out(X,0)$$

Figure 13: Or-gate

is one. This is encoded as $\{in2(and_1, 0) \leftarrow;\ out(and_1, 1) \leftarrow\}$. The encoding of the outputs can be considered an observation of the system.

Example 11.2 Consider the logic circuit consisting of two and-gates and_1 and and_2, where the output of and_1 is connected to both inputs of and_2. Set the inputs of and_1 to one and set the output of and_2 to zero. The resulting circuit has a diagnosis where the and-gate and_1 is abnormal and the and-gate and_2 is normal. To this diagnosis corresponds two stable models. One of them contains the atom $p(and_2)$ and the other contains the atom $q(and_2)$. □

Definition 11.2 ([Rei87]) *A diagnosis for (SD, C) and OBS is a minimal set $\Delta \subseteq C$ such that*

$$SD \cup OBS \cup \{ab(c) \mid c \in \Delta\} \cup \{\neg ab(c) \mid c \in \Delta - C\}$$

is consistent.

Let P be the encoding of a logic circuit including its inputs and outputs as described in the previous paragraphs. This program P determines a system description SD and the observations OBS of the logic circuit if the rules are replaced by propositional logic sentences in the following way. A rule

$$a \leftarrow b_1, \ldots, b_n, \mathrm{not}(c_1), \ldots, \mathrm{not}(c_m)$$

is replaced by the propositional sentence

$$b_1 \wedge \ldots \wedge b_n \wedge \neg c_1 \wedge \ldots \wedge \neg c_m \rightarrow a$$

The set components C of the system is determined by the atoms of P of form $ab(\cdot)$. We note that the p, q, r rules should, in the propositional case, be replaced by suitable sentences containing disjunctions. We now show that each diagnosis of (SD, C) and OBS determines at least one stable model of P and that each stable model determines one diagnosis.

Lemma 11.3 *Let P be a set of rules and let B be a P-full set. Then, B is maximal, i.e., B is not a proper subset of any P-full set.*

Proof. Let B' be a subset of $\mathrm{NAnt}(P)$ such that B is a proper subset of B'. Take an arbitrary not-atom $\mathrm{not}(a) \in B'$ not in B. Now, $\mathrm{not}(a) \notin B$ implies $a \in \mathrm{Dcl}(P, B)$ as B is P-full. But $a \in \mathrm{Dcl}(P, B)$ implies $a \in \mathrm{Dcl}(P, B')$ and as $\mathrm{not}(a) \in B'$ we see that B' is not P-full. □

A diagnosis of a logic circuit is a minimal set of gates considered abnormal. Equivalently, a diagnosis is a maximal set of gates not considered abnormal. If we interpret not-atoms of the form $\mathrm{not}(ab(X))$ as meaning that gate X is not abnormal, we can interpret sets of not-atoms as diagnoses of the corresponding system description SD and observations OBS. Now, if P is a set of rules encoding a logic circuit, we have to show that a set is P-full if and only if the set is a diagnosis.

Let the set B be a P-full set and consider the rules defining an and-gate and_1. If not$(ab(and_1))$ is not in B, then not$(p(and_1))$, not$(q(and_1))$, and not$(r(and_1))$ are in B. On the other hand, assume that not$(ab(and_1))$ is in B. If $out(and_1, 0) \notin \mathrm{Dcl}(P, B)$ holds then not$(p(and_1))$, not$(q(and_1))$, and not$(r(and_1))$ are in B. If $out(and_1, 0) \in \mathrm{Dcl}(P, B)$, then either one of $in1(and_1, 1)$ or $in2(and_1, 1)$ belongs to $\mathrm{Dcl}(P, B)$ and the literals not$(p(and_1))$ and not$(q(and_1))$ are in B, or not$(r(and_1))$ and one of not$(q(and_1))$, and not$(r(and_1))$ is in B. A similar relation holds for or-gates. Thus, a maximal P-full set contains a maximal number of gates not considered abnormal.

Let Δ be a stable model of the set P and let Δ' be the components that correspond to the atoms $ab(\cdot)$ in Δ. As the stable model is the deductive closure of the rules and the P-full set and as no atom in the stable model is covered by the P-full set, the set

$$SD \cup OBS \cup \{ab(c) \mid c \in \Delta'\} \cup \{\neg ab(c) \mid c \in \Delta' - C\}$$

is consistent. Hence, a P-full set determines a diagnosis. Conversely, as the set of sentences is consistent and as the diagnosis is minimal, we see that a diagnosis determines at least one P-full set.

Logic-based abduction [EG95] provides a slightly different approach to diagnosis than Reiter's consistency-based approach. A diagnosis is in this case a subset of a given set of hypotheses that is consistent with the system description and logically entails the observations. An additional condition that the subset of hypotheses is inclusion or cardinality minimal is often added.

As the stable models are the deductive closures of the program P with respect to the P-full sets, the diagnoses are also abductive if we take the set of hypotheses to be the set of not-atoms. The additional condition is in this case inclusion maximality.

11.2 Graph Problems

We encode two NP-complete graph problems as logic programs such that the stable models of the programs correspond to the solutions to the problems. But first we clarify a part of the terminology connected with graphs.

Definition 11.4 *A graph $\mathcal{G}$ is a pair (V, E), where V is a set of vertices and $E \subseteq V \times V$ is a set of edges.*

Let $\mathcal{G} = (V, E)$ be a graph. The graph $\mathcal{G}$ is undirected if for any two vertices a and b such that there is an edge from a to b, i.e., $(a, b) \in E$, there is also an edge from b to a. Two vertices a and b are neighbors if there is an edge between them. A path in a graph is a finite sequence of vertices $a_1, \ldots, a_n$ such that there is an edge from a_i to a_{i+1} for $0 < i < n$. Finally, the path is closed if $a_1 = a_n$.

The N-coloring problem consists of finding an N-coloring of an undirected graph. An N-coloring assigns one of N colors to each vertex of the graph, in such a way that no two neighbors have a color in common. A Hamiltonian circuit in a graph is a closed path that visits each vertex exactly once. The Hamiltonian circuit problem is then the problem of finding a Hamiltonian circuit.

The N-coloring problem is translated from a graph into a logic program as follows. For each vertex a with neighbors $p_1, \ldots, p_j$ and each color $i \in \{0, \ldots, N-1\}$, include the rule

$$\begin{array}{rcl} \mathit{color}(a,i) & \leftarrow & \mathrm{not}(\mathit{color}(p_1,i)), \ldots, \mathrm{not}(\mathit{color}(p_j,i)), \\ & & \mathrm{not}(\mathit{color}(a, i+1 \bmod N)), \\ & & \ldots \\ & & \mathrm{not}(\mathit{color}(a, i+N-1 \bmod N)) \end{array}$$

and for each vertex a include the rule

$$h \leftarrow \mathrm{not}(\mathit{color}(a,0)), \ldots, \mathrm{not}(\mathit{color}(a, N-1)).$$

Finally add the rules $h_2 \leftarrow \mathrm{not}(h)$ and $h_2 \leftarrow \mathrm{not}(h_2)$. The number of rules is thus $(N+1)V + 2$ if V is the number of vertices.

Example 11.3 We give the translation of the 3-coloring problem into a logic program for the complete graph of two vertices a and b.

$$\begin{array}{l} \{\mathit{color}(b,0) \leftarrow \mathrm{not}(\mathit{color}(a,0)), \mathrm{not}(\mathit{color}(b,1)), \mathrm{not}(\mathit{color}(b,2)); \\ \ \mathit{color}(b,1) \leftarrow \mathrm{not}(\mathit{color}(a,1)), \mathrm{not}(\mathit{color}(b,2)), \mathrm{not}(\mathit{color}(b,0)); \\ \ \mathit{color}(b,2) \leftarrow \mathrm{not}(\mathit{color}(a,2)), \mathrm{not}(\mathit{color}(b,0)), \mathrm{not}(\mathit{color}(b,1)); \\ \ \mathit{color}(a,0) \leftarrow \mathrm{not}(\mathit{color}(b,0)), \mathrm{not}(\mathit{color}(a,1)), \mathrm{not}(\mathit{color}(a,2)); \\ \ \mathit{color}(a,1) \leftarrow \mathrm{not}(\mathit{color}(b,1)), \mathrm{not}(\mathit{color}(a,2)), \mathrm{not}(\mathit{color}(a,0)); \\ \ \mathit{color}(a,2) \leftarrow \mathrm{not}(\mathit{color}(b,2)), \mathrm{not}(\mathit{color}(a,0)), \mathrm{not}(\mathit{color}(a,1)); \\ \ h \leftarrow \mathrm{not}(\mathit{color}(a,0)), \mathrm{not}(\mathit{color}(a,1)), \mathrm{not}(\mathit{color}(a,2)); \\ \ h \leftarrow \mathrm{not}(\mathit{color}(b,0)), \mathrm{not}(\mathit{color}(b,1)), \mathrm{not}(\mathit{color}(b,2)); \\ \ h_2 \leftarrow \mathrm{not}(h);\ h_2 \leftarrow \mathrm{not}(h_2)\} \end{array}$$

$\square$

Proposition 11.5 *Let $\mathcal{G} = (V, E)$ be an undirected graph. Let P be the logic program encoding the N-coloring problem for the graph $\mathcal{G}$. Then, Δ is a stable model of P if and only if $C = \{color(a, i) \mid color(a, i) \in \Delta, 0 \leq i < N\}$ is an N-coloring of $\mathcal{G}$ in the sense that $a \in V$ has color i if $color(a, i) \in C$.*

Proof. ($\Rightarrow$) Let Δ be a stable model of P. Let a be a vertex and let $i \in \{0, \dots, N-1\}$. Note that there is only one rule in P with head $color(a, i)$ and that $\mathrm{not}(color(a, i + 1 \bmod N)), \dots, \mathrm{not}(color(a, i + N - 1 \bmod N))$ is in the body of this rule. Thus, $color(a, i), color(a, j) \in \Delta$ for $i \neq j$ is not possible. Furthermore, as the rules $h_2 \leftarrow \mathrm{not}(h)$ and $h_2 \leftarrow \mathrm{not}(h_2)$ are in P we must have $h \notin \Delta$. The rule $h \leftarrow \mathrm{not}(color(a, 0)), \dots, \mathrm{not}(color(a, N-1))$ then implies that there is an i such that $color(a, i) \in \Delta$. Hence, each vertex has exactly one color. The one rule in P with head $color(a, i)$ has $\mathrm{not}(color(b, i))$ in its body for each neighbor b of a. Thus, $color(a, i)$ and $color(b, i)$ cannot both belong to the stable model Δ. Consequently, no two neighbors have a color in common and C is an N-coloring.

($\Leftarrow$) Let $C \subseteq \{color(a, i) \mid a \in V, 0 \leq i < N\}$ be an N-coloring of $\mathcal{G}$. Let

$$\Lambda = \{\mathrm{not}(color(a, i)) \mid a \in V, 0 \leq i < N, color(a, i) \notin C\} \cup \{\mathrm{not}(h)\}.$$

If $\mathrm{not}(color(a, i)) \in \Lambda$ then $color(a, j) \in C$ for a $j \neq i$. Thus, $color(a, i) \notin \mathrm{Dcl}(P, \Lambda)$. On the other hand, if $\mathrm{not}(color(a, i)) \notin \Lambda$, then $color(a, i) \in C$ and $color(a, j) \notin C$ for $j \neq i$, $0 \leq j < N$. Therefore, $\mathrm{not}(color(a, j)) \in \Lambda$ and $\mathrm{not}(color(b, i)) \in \Lambda$ for all neighbors b of a. Thus, $color(a, i) \in \mathrm{Dcl}(P, \Lambda)$. In addition, $h \notin \mathrm{Dcl}(P, \Lambda)$ and $h_2 \in \mathrm{Dcl}(P, \Lambda)$. Hence, Λ is P-full and $\mathrm{Dcl}(P, \Lambda)$ is a stable model of P. □

The translation of the Hamiltonian circuit problem is somewhat more complicated. For each pair (a, b) of vertices in the graph such that there is an edge between a and b, and where $b_1, \dots, b_i$ are the neighbors of a excluding b, and $a_1, \dots, a_j$ are the neighbors of b excluding a, include the rule

$$\begin{aligned} edge(a, b) \quad \leftarrow \quad & \mathrm{not}(edge(a, b_1)), \dots, \mathrm{not}(edge(a, b_i)), \\ & \mathrm{not}(edge(a_1, b)), \dots, \mathrm{not}(edge(a_j, b)), \\ & \mathrm{not}(edge(b, a)). \end{aligned}$$

Fix a vertex d. For each pair (a, b) of vertices in the graph such that there is an edge between a and b, if $b \neq d$ include the rule

$$b \leftarrow a, edge(a, b)$$

and if $b = d$ include the rule

$$d' \leftarrow a, edge(a, d).$$

Finally, for each vertex $a \neq d$ add $a \leftarrow \mathrm{not}(a)$, and then add the rules $d \leftarrow$ and $d' \leftarrow \mathrm{not}(d')$. If the number of vertices is denoted by V and the number of edges by E, the number of rules in this problem is $2E + V + 1$.

Example 11.4 We give the set of rules for the translation of the Hamiltonian circuit problem for the complete graph of the three vertices a, b and c.

$$
\begin{aligned}
\{&edge(a,b) \leftarrow \mathrm{not}(edge(a,c)), \mathrm{not}(edge(c,b)), \mathrm{not}(edge(b,a));\\
&edge(b,a) \leftarrow \mathrm{not}(edge(b,c)), \mathrm{not}(edge(c,a)), \mathrm{not}(edge(a,b));\\
&edge(a,c) \leftarrow \mathrm{not}(edge(a,b)), \mathrm{not}(edge(b,c)), \mathrm{not}(edge(c,a));\\
&edge(c,a) \leftarrow \mathrm{not}(edge(c,b)), \mathrm{not}(edge(b,a)), \mathrm{not}(edge(a,c));\\
&edge(b,c) \leftarrow \mathrm{not}(edge(b,a)), \mathrm{not}(edge(c,a)), \mathrm{not}(edge(c,b));\\
&edge(c,b) \leftarrow \mathrm{not}(edge(c,a)), \mathrm{not}(edge(a,b)), \mathrm{not}(edge(b,c));\\
&b \leftarrow a, edge(a,b);\ c \leftarrow a, edge(a,c);\ c \leftarrow b, edge(b,c);\\
&a' \leftarrow b, edge(b,a);\ b \leftarrow c, edge(c,b);\ a' \leftarrow c, edge(c,a);\\
&a' \leftarrow \mathrm{not}(a');\ b \leftarrow \mathrm{not}(b);\ c \leftarrow \mathrm{not}(c);\ a \leftarrow\}
\end{aligned}
$$

□

Proposition 11.6 *Let $\mathcal{G} = (V, E)$ be an undirected graph. Let P be the logic program encoding the Hamiltonian circuit problem for the graph $\mathcal{G}$. Then, Δ is a stable model of P if and only if $F = \{(a,b) \in E \mid edge(a,b) \in \Delta\}$ is a Hamiltonian circuit in $\mathcal{G}$.*

Proof. ($\Rightarrow$) Let Δ be a stable model of P. Assume there are vertices a, b, and c such that $b \neq c$ and $edge(a,b) \in \Delta$. Then, $edge(a,c) \notin \Delta$ as there is only one rule in P with head $edge(a,b)$ and $\mathrm{not}(edge(a,c))$ is in the body of this rule. Thus, at most one edge begins at any vertex. Similarly, at most one edge ends at any vertex. As $a \leftarrow \mathrm{not}(a) \in P$ and $d \leftarrow \in P$, $a \in \Delta$ for $a \in V \cup \{d'\}$. But a vertex a is in Δ only if there is a directed path from d to a in F as is seen by the following argument. If there is an edge in F going from a to b and a is in Δ, then $edge(a,b) \in \Delta$, and as $b \leftarrow a, edge(a,b) \in P$ we have that $b \in \Delta$. Now, as $d \in \Delta$ we have proved by induction that there is a directed path from d to a in F if $a \in \Delta$. Finally, $d' \in \Delta$ implies that the path is a Hamiltonian circuit.

($\Leftarrow$) Let $F \subseteq E$ be a Hamiltonian circuit in $\mathcal{G}$. Let $\Lambda = \{\mathrm{not}(edge(a,b)) \mid (a,b) \in E - F\}$. If $\mathrm{not}(edge(a,b)) \in \Lambda$ then $(a,b) \notin F$ and there exists a vertex c such that $(a,c) \in F$. Thus, $\mathrm{not}(edge(a,c)) \notin \Lambda$ and $edge(a,b) \notin \mathrm{Dcl}(P,\Lambda)$. If, on the other hand, $\mathrm{not}(edge(a,b)) \notin \Lambda$, then $(a,b) \in F$. As F is a Hamiltonian circuit, $(a,c) \notin F$ for all neighbors of a except b, $(d,b) \notin F$ for all neighbors of b except a, and $(b,a) \notin F$. Hence, $edge(a,b) \in \mathrm{Dcl}(P,\Lambda)$. In addition, as there is a directed path from d to any vertex a, $a \in \mathrm{Dcl}(P,\Lambda)$. We therefore conclude that Λ is P-full. □

11.3 Remarks

The algorithm **smodels**$_2$ is not especially suited for finding diagnoses. The algorithm finds the diagnoses in a rather arbitrary order. Furthermore, one would certainly wish to relate a probability of failure to the components in the system one is diagnosing, thus finding the more probable diagnoses first. Although, the algorithm does not directly support such specialized behavior, it serves as a good framework for implementing application specific algorithms.

Another example of possible specialization can be taken from the Hamiltonian circuit problem. If we add weights to the edges in the graph and search for the Hamiltonian circuit with least total weight, we obtain the traveling salesman problem. A test for comparing the weight of a partial circuit with the weight of the last circuit found is easily incorporated in the algorithm.

We remark that the algorithm **smodels**$_2$ is a general algorithm for computing stable models of logic programs that can be enhanced to cope with application specific demands.

12 Experimental Results

In this section we compare the execution times of the algorithm **smodels$_2$** and the SLG system [CW94, CW93] by using the two algorithms to search for 3-colorings and Hamiltonian circuits in different graphs. The graphs were created using the Stanford GraphBase [Knu93], which is a highly portable collection of programs and data that serves as a platform for combinatorial algorithms. In addition, **smodels$_2$** has been tested on the two benchmark programs used in [CW94] and on two diagnosis problems; a ripple carry adder and a parity circuit.

Two types of graphs have been used in the tests. A special type of planar graph, created by the *plane* function found in the Stanford GraphBase, and a 'ladder'-graph, see Figure 14. The planar graphs are constructed by Delaunay triangulation of randomly inserted points in a plane. The test cases are generated from the graphs by translating the graphs into propositional logic programs using the techniques introduced in Section 11.

Since the SLG system is goal-directed we must add some rules to the translations of the logic circuits and the graph problems to make the system compute the whole stable model. We add the rule $s \leftarrow d'$ to the Hamiltonian circuit problem. For each vertex a and each color i we add the rule $s \leftarrow color(a, i)$ to the 3-coloring problem. Finally, for each atom ab that determines whether a gate is abnormal we add a rule $s \leftarrow ab$ to the diagnosis problem. The resulting set of rules is then used when testing the SLG system, using the goal s, and the algorithm **smodels$_2$**.

Figure 14: Ladder Graph

The two benchmark programs found in [CW94] have also been used to generate test cases. The first benchmark program consists of the following rules:

$$
\begin{array}{ll}
z1(X) \leftarrow v1(X), w1(X) & z2(X) \leftarrow v1(X), w2(X) \\
z3(X) \leftarrow v2(X), w1(X) & z4(X) \leftarrow v2(X), w2(X) \\
v1(X) \leftarrow s(X) & v2(X) \leftarrow t(X) \\
w1(X) \leftarrow p(X) & w2(X) \leftarrow q(X) \\
t(X) \leftarrow \mathrm{not}(s(X)) & s(X) \leftarrow \mathrm{not}(t(X)) \\
p(X) \leftarrow \mathrm{not}(q(X)) & q(X) \leftarrow \mathrm{not}(p(X)) \\
m(X) \leftarrow z1(X) & m(X) \leftarrow z2(X) \\
m(X) \leftarrow z3(X) & m(X) \leftarrow z4(X)
\end{array}
$$

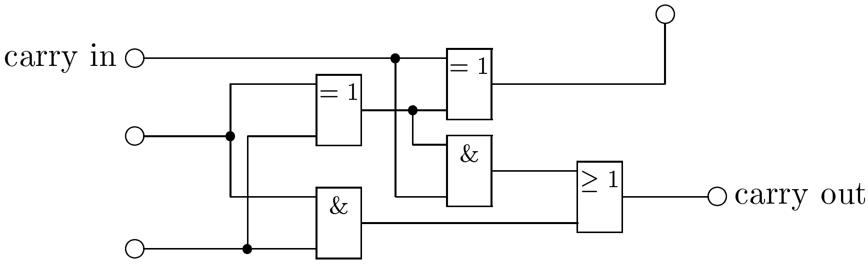

Figure 15: Ripple Carry Adder

Before the stable models are computed a number of constants are introduced and the program is grounded, i.e., the variables in the rules are instantiated with the constants. If the number of constants is n, the number of stable models is 2^{2n}.

The second benchmark program consists of the following rules:

$$\begin{array}{ll} s(X) \leftarrow p(X), q(X) & s(X) \leftarrow p(X), r(X) \\ s(X) \leftarrow q(X), r(X) & p(X) \leftarrow \mathrm{not}(q(X)) \\ q(X) \leftarrow \mathrm{not}(r(X)) & r(X) \leftarrow \mathrm{not}(p(X)) \end{array}$$

We introduce a number of constants and ground the program. This program does not have any stable models.

Both benchmark programs are very simple as there are few interdependencies between the propositional atoms. This fact is clearly reflected in the timing results.

The circuit diagram of the ripple carry adder is shown in Figure 15 and the circuit diagram of the parity circuit is shown in Figure 16. We set all inputs of the ripple carry adder and the parity circuit to zero. In order to get diagnoses we set the last output of the adder and the output of the parity circuit to one.

The results are tabulated in Figures 17–19. They have been calculated as the minimum, maximum and average times for ten different runs on a pseudo-randomly shuffled set of rules. The rationale for shuffling the set of rules is that a particular ordering of the rules might help the algorithm to avoid backtracking and thus give a skewed picture of its behavior. All times are in seconds and they represent the time to find one stable model if one exists, or the time to decide that there are no stable models. As an exception all stable models of the first benchmark program were computed. A blank entry indicates that the corresponding test was not performed.

All tests were run on a SPARCstation IPX workstation, with 16 MB of memory and the SunOS 4.1.3 operating system. The SLG system was compiled in

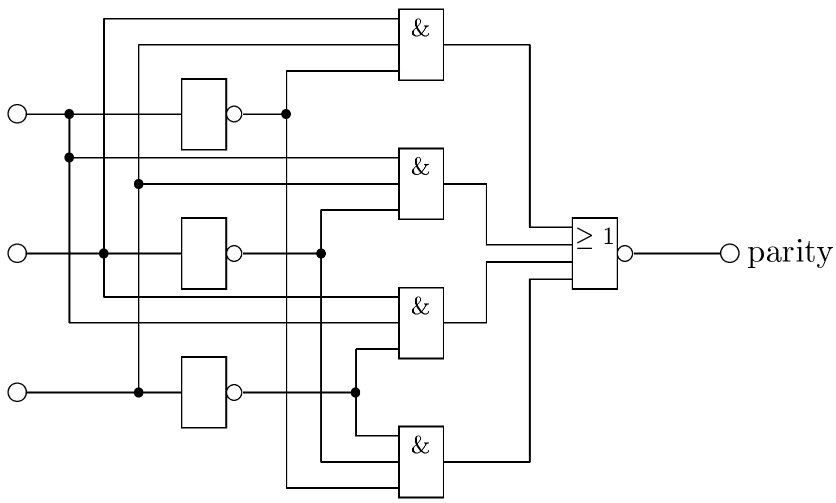

Figure 16: Parity Circuit

SICStus 2.1 #5 with the fastcode compilation-flag set. The C++ program **smodels**$_2$ was compiled using gcc version 2.5.8. The timing was handled by the SICStus command 'statistics' in SLG, and by the C function 'ftime' in **smodels**$_2$.

The test results show quite clearly that **smodels**$_2$ computes the stable model semantics significantly faster than the SLG system. We can explain some of the performance dissimilarities by the differing implementation techniques; **smodels**$_2$ is implemented in C++ while SLG is implemented as a meta-interpreter on top of Prolog. However, the execution times display, in spite of this, drastic differences.

The more efficient pruning technique of **smodels**$_2$ is reflected in the timing results of the 3-coloring and the Hamiltonian circuit tests. The SLG system performs poorly on the 3-coloring tests, while the performance of **smodels**$_2$ is much better. The SLG system does not perform any better on the Hamiltonian circuit tests. The performance of **smodels**$_2$ is not as good as on the 3-coloring tests, but **smodels**$_2$ can take better advantage of the simple structure of the ladder graph than can SLG.

The pruning capabilities of the SLG system and **smodels**$_2$ do not play an important role in the simple benchmark tests. The differences in these tests are better explained by the different implementation techniques. The SLG system prunes the search space by reducing the program. This probably adds more overhead than what the direct computation of deductive closures adds for the **smodels**$_2$ algorithm.

The logic circuit tests demonstrate an intermediate level of complexity. The ripple carry adders are easily handled by both algorithms. The parity circuits

Ladder graphs, 3-coloring							
Vertices	Rules	SLG			**smodels**$_2$		
		min	max	ave	min	max	ave
10	73	0.49	1.23	0.72	0.02	0.03	0.02
12	87	0.62	5.31	1.18	0.02	0.07	0.03
14	101	0.77	18.24	3.74	0.03	0.04	0.03
16	115	0.87	79.30	14.08	0.03	0.08	0.04
18	129	1.05	508.58	89.33	0.03	0.07	0.04
20	143	1.15	1102.02	133.29	0.03	0.07	0.04
22	157	1.34	660.56	279.65	0.04	0.08	0.05
100	703				0.24	0.30	0.26
200	1403				0.69	0.77	0.72
300	2103				1.47	1.60	1.53
400	2803				2.54	2.70	2.63
500	3503				4.06	4.43	4.22

Planar graphs, 3-coloring							
Vertices	Rules	SLG			**smodels**$_2$		
		min	max	ave	min	max	ave
10	73	7.56	34.63	23.91	0.03	0.07	0.05
12	87	2.68	140.48	82.91	0.03	0.06	0.04
14	101	35.33	661.36	392.09	0.04	0.05	0.04
16	115	3.73	2451.46	1425.46	0.04	0.10	0.06
18	129				0.04	0.09	0.06
20	143				0.05	0.07	0.05
100	703				0.19	0.25	0.22
200	1403				0.42	0.50	0.45
300	2103				0.59	0.83	0.69
400	2803				0.85	1.07	0.97
500	3503				1.04	1.57	1.17

Figure 17: Test Results

on the other hand are easy tests for the **smodels**$_2$ algorithm but hard tests for the SLG system.

To give an idea of the size of the search spaces we note that the search space of the program P is all subsets of the set $\mathrm{NAnt}(P)$, i.e., all sets of atoms that might be a stable model. Thus, if there are n atoms in $\mathrm{NAnt}(P)$ then the size of the search space is 2^n. For example, the encoding of the 3-coloring problem for the ladder graph of 500 vertices contains 1502 not-atoms. These atoms produce a search space of size 2^{1502} which is roughly of the order 10^{452}. Of course, the number of 3-colorings of the ladder graph of 500 vertices is $2 \cdot 3^{250}$ which is about 10^{119}. This fact alone should make it possible for an algorithm to reduce the search space quite a lot.

Ladder graphs, Hamiltonian circuit							
Vertices	Rules	SLG			**smodels**$_2$		
		min	max	ave	min	max	ave
10	64	1.01	1.37	1.22	0.02	0.06	0.03
14	92	2.89	6.00	4.72	0.04	0.08	0.05
18	120	5.57	25.72	19.07	0.05	0.18	0.09
22	148	24.04	89.63	68.06	0.07	0.27	0.13
26	176	18.36	421.71	241.99	0.09	0.46	0.24
30	204	67.12	1657.42	904.48	0.11	0.78	0.30
40	274				0.18	1.56	0.66
50	344				0.26	3.37	0.92
60	414				0.37	12.10	4.47
70	484				0.50	12.48	2.86
80	554				0.66	23.15	12.54
90	624				0.78	46.09	15.03
100	694				0.98	68.07	19.98

Planar graphs, Hamiltonian circuit							
Vertices	Rules	SLG			**smodels**$_2$		
		min	max	ave	min	max	ave
10	104	34.01	60.70	47.73	0.04	0.07	0.05
12	130	139.87	525.45	344.87	0.07	0.30	0.12
14	156	1145.51	3751.47	2139.53	0.07	0.52	0.23
16	174				0.10	1.90	0.39
18	200				0.25	7.50	1.33
20	226				0.16	314.53	44.14
22	252				0.23	1010.78	157.63

SLG benchmark 1, all stable models							
Constants	Rules	SLG			**smodels**$_2$		
		min	max	ave	min	max	ave
1	16	0.08	0.13	0.10	0.01	0.05	0.02
2	32	0.27	0.30	0.29	0.04	0.08	0.05
3	48	0.91	0.98	0.95	0.10	0.18	0.14
4	64	3.57	3.66	3.61	0.49	0.57	0.53
5	80	14.92	15.33	15.12	2.16	2.25	2.20
6	96	66.09	67.25	66.66	9.05	9.25	9.18
7	112	285.06	289.65	287.24	37.01	37.73	37.32

Figure 18: Test Results

SLG benchmark 2							
Constants	Rules	SLG			**smodels**$_2$		
		min	max	ave	min	max	ave
50	300	1.36	1.43	1.40	0.05	0.06	0.05
100	600	2.86	2.94	2.91	0.09	0.10	0.09
150	900	4.43	4.53	4.47	0.14	0.20	0.15
200	1200	5.99	6.11	6.05	0.19	0.23	0.20
250	1500	7.53	7.62	7.60	0.23	0.28	0.25
300	1800	9.15	9.26	9.22	0.28	0.33	0.29

Ripple carry adder							
Bits	Rules	SLG			**smodels**$_2$		
		min	max	ave	min	max	ave
1	138	1.01	1.06	1.04	0.03	0.05	0.03
2	279	2.26	2.41	2.35	0.05	0.09	0.07
3	420	3.81	3.89	3.84	0.08	0.13	0.09
4	561	5.34	5.54	5.47	0.11	0.17	0.13
5	702	7.10	7.34	7.20	0.15	0.15	0.15
6	843	8.95	9.19	9.05	0.18	0.23	0.20
7	984	10.89	11.18	11.04	0.21	0.29	0.24
8	1125	13.07	13.30	13.18	0.26	0.37	0.28
9	1266	15.26	15.43	15.35	0.29	0.36	0.31
10	1407	17.62	17.77	17.70	0.35	0.41	0.37

Parity circuit							
Bits	Rules	SLG			**smodels**$_2$		
		min	max	ave	min	max	ave
3	233	2.21	5.54	3.04	0.05	0.09	0.06
5	468	5.11	488.45	163.29	0.11	0.17	0.12
7	703	12.48	16608.21	2237.42	0.17	0.26	0.21
9	938				0.28	0.31	0.30
11	1173				0.35	0.45	0.42

Figure 19: Test Results

13 Conclusions

In this work we have made improvements on an algorithm of Niemelä [Nie95, NS95] for computing stable models of propositional logic programs. The improvements consists of removing redundancies from the algorithm and strengthening the ability of the algorithm to prune the search space. Removing redundancies simplifies the algorithm and has a slight impact on the performance of the algorithm. A simpler algorithm is also easier to implement. Improving the pruning capabilities has the potential of drastically increasing the performance of the algorithm since the search space is exponentially dependent on the size of the input in the worst case.

We have examined how the algorithm computes stable models of logic programs with a total well-founded model. The algorithm computes the well-founded model roughly in time proportional to the square of the size of the input. The set of logic programs with a total well-founded model contains the important set of stratified logic programs. As it is possible to compute the total well-founded model of a stratified logic program in time proportional to the size of the program, the question whether the same time complexity bound is attainable for programs with a total well-founded model naturally arises. Finding a tight lower bound for the time complexity needed to compute a well-founded model is an interesting area of further research.

We have given a simple upper bound on the time complexity of the algorithm. We have also shown that the algorithm has worst-case exponential time complexity. This was not surprising since the problem the algorithm solves is NP-complete. The problem of finding a better upper bound is primarily connected to the problem of computing the well-founded model semantics. The final question regarding the time complexity is, of course, not answered until the tantalizing open problem concerning the structure of the polynomial time hierarchy is solved.

In connection with the problem of computing well-founded models we have presented two linear time algorithms for computing deductive closures of logic programs. The algorithms are based on a linear time algorithm of Dowling and Gallier [DG84]. We have shown that the first algorithm, which is almost identical to the algorithm of Dowling and Gallier, can be used to incrementally compute the deductive closures of a program with a growing set of literals. The second algorithm have been shown to compute the deductive closures of programs with a shrinking set of literals more efficiently than a plain recomputation of the whole closure. The efficiency with which the deductive closures are computed immediately affects the time complexity of the algorithm computing stable models.

The dependency graph and its strongly connected components can be used to efficiently compute the stable model semantics for stratified logic programs. We have demonstrated that this is not possible in the general case. Allowing variables in the logic programs can reduce the number of rules needed to encode a problem, thus simplifying the work of the programmer. By limiting the use of variables to range restricted logic programs, we have demonstrated that the number of rules in the ground program can be reduced. We have also exhibited logic program encodings of the problem of finding diagnoses in logic circuits and of finding Hamiltonian circuits and N-colorings in undirected graphs. These encodings can be used when comparing implementations of algorithms for computing stable models. Their use in real world applications is limited as such applications often contain restrictions that are hard to express in propositional logic programs. The algorithm presented in this work serves, however, as a good base for implementations handling these restrictions.

The algorithm for computing the stable model semantics have been compared to three other algorithms. One of these, the SLG system, is considered the state of the art among the algorithms computing stable models. We have arrived at the conclusion that the algorithm presented in this work surpasses all the other three algorithms in performance. We have described how to implement the algorithm in order to achieve this high performance and we have developed a prototype implementation. Finally, we have experimentally compared the implementation to the SLG system and concluded that the test results clearly reflects the better performance of the algorithm.

Acknowledgments

This report contains results of my work in the Digital Systems Laboratory at Helsinki University of Technology from February to August 1995. I thank Professor Leo Ojala and Dr.Tech. Ilkka Niemelä for giving me the opportunity to work in the laboratory. I am very grateful to Dr.Tech. Ilkka Niemelä for his comments and advice. I also thank my colleagues in the laboratory.

Otaniemi, September, 1995

Patrik Simons

References

[ABW88] K.R. Apt, H.A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, Los Altos, 1988.

[BF91] N. Bidoit and C. Froidevaux. Negation by default and unstratified logic programs. *Theoretical Computer Science*, 78:85–112, 1991.

[BNNS94] C. Bell, A. Nerode, R.T. Ng, and V.S. Subrahmanian. Mixed integer programming methods for computing nonmonotonic deductive databases. *Journal of the ACM*, 41(6):1178–1215, November 1994.

[CH85] A. Chandra and D. Harel. Horn clause queries and generalizations. *Journal of Logic Programming*, 2:1–15, 1985.

[Cla78] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[CW93] W. Chen and D.S. Warren. The SLG system, 1993. Available by ftp from seas.smu.edu:pub or sbcs.sunysv.edu:pub/XSB.

[CW94] W. Chen and D.S. Warren. Computation of stable models and its integration with logical query processing. Available by ftp from ftp.ms.uky.edu: pub/lpnmr/chen2.ps, 1994.

[DG84] W.F. Dowling and J.H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 3:267–284, 1984.

[dK86] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[Doy79] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[EG95] T. Eiter and G. Gottlob. The complexity of logic-based abduction. *Journal of the ACM*, 42(1):3–42, January 1995.

[Elk90] C. Elkan. A rational reconstruction of nonmonotonic truth maintenance systems. *Artificial Intelligence*, 43:219–234, 1990.

[Esh90] K. Eshghi. Computing stable models by using the ATMS. In *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 272–277, Boston, MA, USA, July 1990. The MIT Press.

[Gel87] M. Gelfond. On stratified autoepistemic theories. In *Proceedings of the 6th National Conference on Artificial Intelligence*, pages 207–211, Seattle, USA, July 1987. American Association for Artificial Intelligence.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability*. W.H. Freeman and Company, San Francisco, 1979.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080, Seattle, USA, August 1988. The MIT Press.

[Knu93] D.E. Knuth. The Stanford GraphBase, 1993. A collection of programs and data. Available by ftp from labrea.stanford.edu:pub/sgb.

[Lif88] V. Lifschitz. On the declarative semantics of logic programs with negation. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 177–192. Morgan Kaufmann Publishers, Los Altos, 1988.

[Llo84] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1984.

[McC80] J. McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[McC86] J. McCarthy. Applications of circumscription to formalizing commonsense knowledge. *Artificial Intelligence*, 28:89–116, 1986.

[Moo85] R.C. Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25:75–94, 1985.

[MS92] W. Marek and V.S. Subrahmanian. The relationship between stable, supported, default and autoepistemic semantics for general logic programs. *Theoretical Computer Science*, 103(2):365–386, September 1992.

[MT91] W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the ACM*, 38:588–619, 1991.

[Nie94] I. Niemelä. A decision method for nonmonotonic reasoning based on autoepistemic reasoning. In *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning*, pages 473–484, Bonn, Germany, May 1994. Morgan Kaufmann Publishers.

[Nie95] I. Niemelä. Towards efficient default reasoning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 312–318, Montreal, Canada, August 1995. Morgan Kaufmann Publishers.

[NR94] I. Niemelä and J. Rintanen. On the impact of stratification on the complexity of nonmonotonic reasoning. *Journal of Applied Non-Classical Logics*, 4(2):141–179, 1994.

[NS95] I. Niemelä and P. Simons. Evaluating an algorithm for default reasoning. Accepted to be presented at the *IJCAI'95 Workshop on Applications and Implementations of Nonmonotonic Reasoning Systems*, August 1995.

[PP88] H. Przymusinska and T.C. Przymusinski. Weakly perfect model semantics for logic programs. In K.A. Bowen and R.A. Kowalski, editors, *Proceedings of the 5th International Conference and Symposium on Logic Programming*, pages 1106–1120, Cambridge, MA, USA, 1988. The MIT Press.

[PR91] S.G. Pimentel and W.L. Rodi. A nonmonotonic assumption-based TMS using stable bases. In *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 485–495, Cambridge, MA, USA, April 1991. Morgan Kaufmann Publishers.

[Prz88] T.C. Przymusinski. On the declarative semantics of stratified deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann Publishers, 1988.

[PY92] C.H. Papadimitriou and M. Yannakakis. Tie-breaking semantics and structural totality. In *Proceedings of the 11th Symposium on Principles of Database Systems*, New York, June 1992. ACM.

[Rei78] R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, New York, 1978.

[Rei80] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[Rei87] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[She88] J.C. Shepherdson. Negation in logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Morgan Kaufmann Publishers, Los Altos, 1988.

[SNV95] V.S. Subrahmanian, D. Nau, and C. Vago. Wfs + branch and bound = stable models. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):362–377, June 1995.

[VG88] A. Van Gelder. Negation as failure using tight derivations for general logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 149–176. Morgan Kaufmann Publishers, Los Altos, 1988.

[VGRS88] A. Van Gelder, K.A. Ross, and J.S. Schlipf. Unfounded sets and the well-founded semantics for general logic programs, extended abstract. In *Proceedings of the 7th Symposium on Principles of Database Systems*, pages 221–230, Austin, Texas, March 1988. ACM Press.

[VGRS91] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, July 1991.