HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **A:** Research Reports
No. **47**; August 1997

ISSN 0783–5396
ISBN 951–22–3712–1

TOWARDS CONSTRAINT SATISFACTION THROUGH LOGIC PROGRAMS AND THE STABLE MODEL SEMANTICS

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
P.O.Box 1100
FIN–02015 HUT, FINLAND

HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **A:** Research Reports
No. **47**; August 1997

ISSN 0783–5396
ISBN 951–22–3712–1

Towards Constraint Satisfaction through Logic Programs and the Stable Model Semantics

PATRIK SIMONS

Abstract: Logic programs with the stable model semantics can be thought of as a new paradigm for constraint satisfaction, where the rules of a program are seen as constraints on the stable models. In this work the paradigm is realized by developing an efficient procedure for computing the stable models of ground logic programs. A strong pruning technique based on two deductive closures is introduced. The technique is further strengthened by the introduction of backjumping, which is an improvement over chronological backtracking, and lookahead, a new pruning method. Moreover, a strong heuristic is derived.

The two deductive closures are given linear-time implementations that provide a linear-space implementation method for the decision procedure. A high lower bound on the least upper bound on the complexity of the procedure is found. In order to generalize the procedure such that it can handle programs with variables, an algorithm for grounding a function-free range restricted logic program that avoids generating the whole ground instantiation is presented. Finally, an implementation of the decision procedure is compared with a state-of-the-art satisfiability checker using hard satisfiability problems as a test domain.

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

Printing: Libella Painopalvelut Oy, 1997

Helsinki University of Technology
Department of Computer Science
Digital Systems Laboratory
P.O.Box 1100
FIN–02015 HUT, FINLAND

Phone: $\frac{09}{+358\text{–}9}$ 4511
Telex: 125 161 htkk fi
Telefax: +358–9–451 3369
E-mail: lab@saturn.hut.fi

Preface

This thesis is a result of my work in the Digital Systems Laboratory at Helsinki University of Technology during the years 1996 and 1997. I'd like to thank Professor Leo Ojala and Docent Ilkka Niemelä for making this possible, and my colleagues in the laboratory for creating a pleasant working atmosphere.

Part of this work was funded by a grant from the Jenny ja Antti Wihurin rahasto.

Otaniemi, August 1997

Patrik Simons

Contents

List of Symbols and Notational Conventions

$\wedge$	conjunction
$\vee$	disjunction
$\neg$	negation
$\emptyset$	the empty set
$-$	set difference
$\cap$	set intersection
$\cup$	set union
$\subseteq$	set inclusion
$\in$	set membership
$\overline{\Delta}$	the complement of Δ
$\|\Delta\|$	the number of elements in Δ
$\Omega(f)$	a lower bound on complexity
$O(f)$	an upper bound on complexity
$a \leftarrow b, \ldots, \mathrm{not}\,(c)$	a rule in a logic program
$\mathrm{Atoms}\,(P)$	the set of atoms in the set of rules P
B^+	the set of atoms in the set of literals B
B^-	the set of not-atoms in the set of literals B
$\mathrm{Dcl}\,(P, B)$	the deductive closure of P with respect to B
$\mathrm{NAnt}\,(P)$	the negative antecedents of P
$\mathrm{not}\,(a)$	a not-atom
$\mathrm{Proof}\,(P, B)$	the proof from B in P
P^S	the reduct of P with respect to S
P_B	the reduct of P with respect to B
$\mathrm{R}\,(P, B)$	the deductive closure reduct of P with respect to B

List of Figures

1 Introduction

In this work we develop an algorithm for computing the stable models of ground logic programs. The computation of stable models is not logic programming as query evaluation, the way it is usually seen. It is in reality closer to constraint satisfaction. Namely, we interpret the rules of a program as constraints on its models.

Since the stable model semantics enforces frame axioms such as minimality of models, and since many combinatorial problems are easily and succinctly described using constraints and such implicit frame axioms, we are motivated in our search for efficient implementation techniques. For example, the stable model semantics can be used to solve as diverse problems as finding Hamiltonian circuits and n-colorings in graphs, diagnosing electronic circuits, and constructing plans; all in a declarative logic programming framework.

The intuition behind the stable model semantics is derived from the autoepistemic logic of Moore [Moo85]. We can think of a stable model of a logic program as a set of beliefs of a rational agent whose inference rules are those of the program and who is able to perform negative introspection: if the agent does not believe a fact, then it concludes the negation of the fact. The stable model semantics is therefore nonmonotonic as adding rules to the program may cause the agent to retract some of its beliefs.

There are two simple examples that demonstrate the basic behavior of the semantics. The first one is given by the program

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

which has two stable models: $\{a\}$ and $\{b\}$. The property this program exhibits is that we may freely make negative assumptions as long as we do not bump into any contradictions. For instance, we may assume $\mathrm{not}\,(b)$ in order to deduce the stable model $\{a\}$. The second example,

$$a \leftarrow a,$$

which has the empty set as its only stable model, exposes the fact that we can not use positive assumptions to deduce what is included in a model. A stable model must be grounded, i.e., it must be inferable from purely negative assumptions. Thus, even if a implies a, the set $\{a\}$ is not stable.

As a simple example of how one can encode problems as logic programs, we take a look at the graph-coloring problem. We want to color the nodes of a graph with three colors such that any two nodes connected by an edge have different colors. If we assume that the predicate $edge(a, b)$ expresses that there is an edge between the nodes a and b and that the predicate $color(a, red)$ expresses that the node a is red, then the program

$$\begin{aligned}
false &\leftarrow edge(N_1, N_2), color(N_1, C), color(N_2, C) \\
color(N, red) &\leftarrow \text{not}\big(color(N, green)\big), \text{not}\big(color(N, blue)\big) \\
color(N, green) &\leftarrow \text{not}\big(color(N, blue)\big), \text{not}\big(color(N, red)\big) \\
color(N, blue) &\leftarrow \text{not}\big(color(N, red)\big), \text{not}\big(color(N, green)\big)
\end{aligned}$$

together with the necessary $edge(\cdot, \cdot)$ facts constitute our encoding. The first rule restricts the coloring of the nodes in the way stated earlier, and the rest of the rules picks a color for every node. The stable models not containing the atom $false$ are now precisely the three-colorings of the graph.

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]. Since the stable model semantics has become a standard method for supplying semantics to nonmonotonic logic programs, there is considerable interest in automating its computation. 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]. Lately, more specialized algorithms have been developed [BNNS94, SNV95, CW96, CMT96, Dim96]. However, they all suffer from exponential space complexity or very weak pruning techniques. In terms of the computational complexity, 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 an algorithm for computing the stable models of finite ground logic programs. The algorithm is based on a decision method for default logic [Nie95] that has subsequently been simplified and improved [Sim95, NS96a, NS96b]. The contribution of this work lies in the incorporation of backward propagation of truth values [CW96] and in the addition of lookahead, a new pruning technique, and backjumping, an improvement over chronological backtracking. Furthermore, a strong heuristic is derived, and a method for handling variables by restricting their use [NS96b] is examined.

1.1 Outline of the Work

In Section 2 we recall some definitions and propositions that concern the stable model semantics and introduce a decision procedure. In Section 3 we present two functions that together with the decision procedure implement a strong pruning technique. We further improve this pruning technique by adding lookahead and backjumping in Section 4. In Section 5 we continue by examining how one can efficiently compute the deductive closures that are used in the functions, and in Section 6 we consider heuristics. After this we perform a complexity analysis in Section 7 and investigate logic programs with variables in Section 8. In Section 9 we evaluate the algorithm, in Section 10 we compare it with other stable model decision procedures, and finally, in Section 11 we present our conclusions.

2 Preliminaries

We examine the terminology and introduce a decision procedure for the stable model semantics. Various versions of this decision procedure have been discussed at great lengths in [Nie95, NS95, Sim95, NS96a, NS96b].

We consider logic programs that consist of rules of the form

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

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

$$\mathrm{not}\,(A) = \{\mathrm{not}\,(a) \mid a \in A\}.$$

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

The stable model semantics for a general logic program P is defined as follows [GL88]. The reduct P^A of P with respect to the set of atoms A is obtained by deleting each rule in P that has a not-atom $\mathrm{not}\,(x)$ in its body such that $x \in A$ and by deleting all not-atoms in the remaining rules. A set of atoms A is a stable model of P if and only if A is the deductive closure of P^A when the rules in P^A 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 A, then the reduct of P with respect to A is $P^A = \{a \leftarrow;\ b \leftarrow b\}$. The closure of this reduct is $\{a\}$. If, on the other hand, b belongs to the set A then the reduct $P^A = \{b \leftarrow b\}$. But, as the closure of $\{b \leftarrow b\}$ is empty, the set A is not a stable model of P.

The deductive closure of a set of rules P and a set of literals B is denoted by $\mathrm{Dcl}\,(P, B)$, where $\mathrm{Dcl}\,(P, B)$ is the smallest set of atoms that contains B^+ and is closed under $\mathrm{R}\,(P, B)$ when

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

is seen as a set of inference rules. Notice that if $B \subseteq B'$ then $\mathrm{Dcl}\,(P, B) \subseteq \mathrm{Dcl}\,(P, B')$, 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 B be the set of literals $\{\mathrm{not}\,(b), c\}$. Then, $\mathrm{R}\,(P, B) = \{a \leftarrow\}$ and the deductive closure $\mathrm{Dcl}\,(P, B) = \{a, c\}$.

The deductive closure provides us with a fixpoint characterization of the stable models.

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

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

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

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

holds. Moreover, we denote the set of atoms that appear in P, negated or not, by $\mathrm{Atoms}\,(P)$. For instance, $\mathrm{Atoms}\,(\{a \leftarrow \mathrm{not}\,(b);\; d \leftarrow c\}) = \{a, b, c, d\}$. Similarly, for a set of literals B we let

$$\mathrm{Atoms}\,(B) = \{x \mid x \in B,\ \text{or } \mathrm{not}\,(x) \in B\}.$$

We will also need some definitions expressing relations between sets of atoms and sets of literals.

Definition 2.2. Let A be a set of atoms and let B be a set of literals. We say that the set of atoms A *agrees* with B if

$$B^{+} \subseteq A \quad \text{and} \quad B^{-} \cap \mathrm{not}\,(A) = \emptyset,$$

and we say that B *covers* A if

$$A \subseteq \mathrm{Atoms}\,(B)\,.$$

Definition 2.3. A set of not-atoms C is P-full if for all $x \in \mathrm{NAnt}\,(P)$

$$\mathrm{not}\,(x) \in C \quad \text{if and only if} \quad x \notin \mathrm{Dcl}\,(P, C).$$

There is a natural one-to-one correspondence between full sets and stable models.

Theorem 2.4. *If* C *is* P*-full, then* $\Delta = \mathrm{Dcl}\,(P, C)$ *is a stable model of* P *and*

$$C = \mathrm{not}\big(\mathrm{NAnt}\,(P) - \Delta\big).$$

Conversely, if Δ *is a stable model of* P*, then* $C = \mathrm{not}\big(\mathrm{NAnt}\,(P) - \Delta\big)$ *is a* P*-full set and*

$$\Delta = \mathrm{Dcl}\,(P, C).$$

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

Let Δ be a stable model of P and let $C = \mathrm{not}\big(\mathrm{NAnt}\,(P) - \Delta\big)$. Then, $\Delta = \mathrm{Dcl}\,(P, C)$ as Δ is a stable model. Hence, for all $x \in \mathrm{NAnt}\,(P)$, $\mathrm{not}\,(x) \in C$ if and only if $x \notin \Delta$ if and only if $x \notin \mathrm{Dcl}\,(P, C)$. ☐

2.1 The Decision Procedure

The decision procedure for the stable model semantics is presented in Figure 1. It computes a stable model Δ of a program P agreeing with a set of literals B such that the function **test** (Δ) returns true, or it returns false if no such stable model exists.

The procedure assumes the existence of two functions, **expand** and **conflict**, which satisfy some basic conditions. Let $B' = \textbf{expand}\,(P, B)$. We assume that

E1 $B \subseteq B'$, and that

E2 every stable model Δ of P that agrees with B agrees with B'.

In addition, we assume that the function **conflict** returns true or false and satisfies the two conditions;

C1 if B covers $\mathrm{NAnt}\,(P)$ and the deductive closure $\mathrm{Dcl}\,(P, B^-)$ does not agree with B, then **conflict** (P, B) returns true, and

C2 if **conflict** (P, B) returns true, then there is no stable model Δ of P such that Δ agrees with B.

It should be apparent that the only function of **expand** is to prune the search space. The function **conflict**, on the other hand, is given the task of enforcing the stable model semantics.

We are now able to prove the correctness of the decision procedure.

Theorem 2.5 (Soundness). *Let P be a set of rules and let B be a set of literals. If* **smodels** (P, B) *returns true, then there is a stable model Δ of P agreeing with B such that* **test** (Δ) *returns true.*

Proof. Assume that **smodels** (P, B) returns true, and let

$$\mathrm{nc}(P, B) = \mathrm{NAnt}\,(P) - \mathrm{Atoms}\,(B)$$

be the atoms not covered by B. We prove the theorem by induction on the size of $\mathrm{nc}(P, B)$.

Assume that $nc(P, B) = \emptyset$. Then, $B \subseteq B'$ by (E1). Since **conflict** (P, B') returns false, (C1) implies that $\mathrm{Dcl}\,(P, B'^-)$ agrees with B'. Thus, for $x \in \mathrm{NAnt}\,(P)$, $\mathrm{not}\,(x) \in B'$ implies $x \notin \mathrm{Dcl}\,(P, B'^-)$, and $\mathrm{not}\,(x) \notin B'$ implies $x \in B'$, which in turn implies $x \in \mathrm{Dcl}\,(P, B'^-)$. Hence, B'^- is P-full and by

function smodels (P, B)
$B' :=$ **expand** (P, B)
if conflict (P, B') returns true **then**
  return false
else if NAnt (P) is covered by B' **then**
  return test$\big(\mathrm{Dcl}\,(P, B')\big)$
else
  Take some $x \in$ NAnt (P) not covered by B'
  if smodels$\big(P, B' \cup \{x\}\big)$ returns true **then**
    return true
  else
    return smodels $(P, B' \cup \{\mathrm{not}\,(x)\})$
  endif
endif

Figure 1: The decision procedure for the stable model semantics

Theorem 2.4, $\Delta = \mathrm{Dcl}\,(P, B'^{-})$ is a stable model of P that agrees with B' and therefore with B. In addition, $\Delta = \mathrm{Dcl}\,(P, B')$ and **test** (Δ) returns true as **smodels** (P, B) does.

Assume that $nc(P, B) \neq \emptyset$. Since **smodels** (P, B) returns true, the function **conflict** (P, B') returns false. If B' covers NAnt (P), then, by the above, we are done. Otherwise, $B \subseteq B'$ by (E1) and **smodels**$\big(P, B' \cup \{\mathrm{not}\,(x)\}\big)$ or **smodels** $(P, B' \cup \{x\})$ returns true for an atom x. Hence, induction shows that there is a stable model Δ of P agreeing with B such that **test** (Δ) returns true. □

Theorem 2.6 (Completeness). *Let P be a set of rules and let B be a set of literals. If there is a stable model Δ of P agreeing with B such that* **test** (Δ) *returns true, then* **smodels** (P, B) *returns true.*

Proof. The proof is by induction on the size of $\mathrm{nc}(P, B) = \mathrm{NAnt}\,(P) - \mathrm{Atoms}\,(B)$. Assume that there is a stable model Δ of P agreeing with B such that the function **test** (Δ) returns true.

Assume that $nc(P, B) = \emptyset$. Then, $B \subseteq B'$ by (E1) and by (C2) and (E2), **conflict** (P, B') returns false. Thus, (C1) implies that that $\mathrm{Dcl}\,(P, B'^{-})$ agrees with B'. Consequently, for $x \in \mathrm{NAnt}\,(P)$, $\mathrm{not}\,(x) \in B'$ implies that $x \notin \mathrm{Dcl}\,(P, B'^{-})$, and $\mathrm{not}\,(x) \notin B'$ implies that $x \in B'$, which in turn implies that $x \in \mathrm{Dcl}\,(P, B'^{-})$. Hence, B'^{-} is P-full and **test**$\big(\mathrm{Dcl}\,(P, B')\big)$ returns true.

Assume that $nc(P, B) \neq \emptyset$. Since (E2) implies that Δ agrees with B', (C2) implies that **conflict** (P, B') returns false. If B' covers NAnt (P), then, by the above, we are done. If not, $B \subseteq B'$ by (E1) and induction shows that if $x \notin \Delta$, then Δ agrees with $B' \cup \{\text{not}\,(x)\}$ and **smodels**$\big(P, B' \cup \{\text{not}\,(x)\}\big)$ returns true. Otherwise, Δ agrees with $B' \cup \{x\}$ and **smodels**$\,(P, B' \cup \{x\})$ returns true. $\square$

3 The Functions Expand and Conflict

The decision procedure **smodels** finds stable models by nondeterministically exploring the search space consisting of all subsets of $\mathrm{NAnt}\,(P)$. This search space is furthermore pruned by the functions **expand** and **conflict**. Therefore, in order to prune the search space as much as possible, we want the **expand** function to return as large a set as possible and the **conflict** function to return true on as many sets as possible. In this section we find good candidates for the two functions.

But first, we establish the idea of a proof in the stable model semantics. A proof of the atom x from the set of literals B in the program P is any subset P' of P such that

$$\begin{aligned} P' = \{r \in P' \mid\ & r = h \leftarrow a_1, \ldots, a_n, \mathrm{not}\,(b_1), \ldots, \mathrm{not}\,(b_m), \\ & \mathrm{not}\,(b_1), \ldots, \mathrm{not}\,(b_m) \in B \text{ and } a_1, \ldots, a_n, x \in \mathrm{Dcl}\,(P', B)\}. \end{aligned}$$

We note that since $\mathrm{Dcl}\,(\cdot, B)$ is monotone, the union of any two proofs is again a proof. We may therefore speak of the maximal proof

$$\mathrm{Proof}\,(P, B) = \bigcup \{P' \subseteq P \mid P' \text{ is a proof from } B \text{ in } P\}.$$

Of particular interest is the proof of Δ from $\mathrm{not}\,(\overline{\Delta})$, as it contains the rules actually used to deduce the stable model.

If the stable model Δ of P agrees with the set of literals B, we can construct an approximation of the proof of Δ by introducing another reduct of P. Let P_B, the reduct of P with respect to B, be the set of rules

$$\{h \leftarrow l_1, \ldots, l_n \in P \mid \mathrm{not}\,(l_i) \notin B\},$$

where $l_1, \ldots, l_n$ are literals and we assume that $\mathrm{not}\big(\mathrm{not}\,(x)\big) = x$. Then, it readily follows that

$$\mathrm{Proof}\big(P, \mathrm{not}\,(\overline{\Delta})\big) \subseteq P_B. \tag{1}$$

We now turn to the function **expand**. Since **expand** (P, B) is going to enlarge the set B, we do not have to worry about (E1). The question is: what can we easily deduce about the stable models of P agreeing with B. We make the following observations.

1. If the body $l_1, \ldots, l_n$ of the rule $h \leftarrow l_1, \ldots, l_n$ is a subset of B, then the head h belongs to every stable model agreeing with B.

2. If P_B contains no rule with head h, then h is not an element of any stable model agreeing with B [Fit85].

3. If $h \in B$ is the head of only one rule $h \leftarrow l_1, \ldots, l_n$ in P_B, then $l_1, \ldots, l_n \in \Delta \cup \mathrm{not}\,(\overline{\Delta})$ for every stable model Δ agreeing with B [CW96].

4. If $\mathrm{not}\,(h) \in B$ and h is the head of the rule $h \leftarrow l_1, \ldots, l_n$ in P_B for which $l_1, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n \in B$, then every stable model that agrees with B agrees with $B \cup \{\mathrm{not}\,(l_i)\}$ [CW96].

The first observation follows immediately from Proposition 2.1 and the other observations follow from (1). In light of this, it makes sense to define the deductive closure $\mathrm{Cl}\,(P, B)$ as a minimal set closed under 1–4, taken as inference rules, that contains B. We have arrived at the result.

Proposition 3.1. *If the stable model Δ of P agrees with B, then Δ agrees also with* $\mathrm{Cl}\,(P, B)$.

We can deduce a bit more about the stable models that agree with B. From (1) we see that

$$\begin{aligned}\Delta &= \mathrm{Dcl}\big(P, \mathrm{not}\,(\overline{\Delta})\big)\\ &= \mathrm{Dcl}\big(\mathrm{Proof}\,(P, \mathrm{not}\,(\overline{\Delta})), \mathrm{not}\,(\overline{\Delta})\big)\\ &\subseteq \mathrm{Dcl}\big(P_B, \mathrm{not}\,(\mathrm{NAnt}\,(P))\big),\end{aligned}$$

which proves the following proposition.

Proposition 3.2. *If the stable model Δ of P agrees with B, then Δ agrees with*

$$B \cup \big\{\mathrm{not}\,(x) \,\big|\, x \in \mathrm{Atoms}\,(P) \;\; \textit{and}\; x \notin \mathrm{Dcl}\big(P_B, \mathrm{not}\,(\mathrm{NAnt}\,(P))\big)\big\}\,.$$

We now have motivation for the implementation of **expand** given in Figure 2. Moreover, the two propositions guarantee that the function satisfies condition (E2).

We let **conflict** (P, B) return true when

$$\mathrm{not}\,(B^+) \cap B^- \neq \emptyset.$$

It is clear that (C2) is satisfied. Since **conflict** (P, B) is only called directly after **expand**, we can make use of the fact that

$$\mathrm{Dcl}\,(P, B^-) \subseteq \mathrm{Cl}\,(P, B) = B$$

function expand (P, B)
repeat
  $B' := B$
  $B := \mathrm{Cl}\,(P, B)$
  $B := B \cup \bigl\{\mathrm{not}\,(x) \bigm| x \in \mathrm{Atoms}\,(P) \text{ and } x \notin \mathrm{Dcl}\bigl(P_B, \mathrm{not}\,(\mathrm{NAnt}\,(P))\bigr)\bigr\}$
until $B' = B$
return B

function conflict (P, B)
  Precondition: $B =$ **expand** (P, B)
if $\mathrm{not}\,(B^+) \cap B^- \neq \emptyset$ **then**
  return true
else
  return false
endif

Figure 2: The functions **expand** and **conflict**

and that

$$x \notin \mathrm{Dcl}\bigl(P_B, \mathrm{not}\,(\mathrm{NAnt}\,(P))\bigr) \quad \text{implies} \quad \mathrm{not}\,(x) \in B.$$

Assume that B covers $\mathrm{NAnt}\,(P)$ and that $\mathrm{Dcl}\,(P, B^-)$ does not agree with B. If there is a literal $\mathrm{not}\,(x) \in B$ such that $x \in \mathrm{Dcl}\,(P, B^-)$, then $x \in B^+$ and $\mathrm{not}\,(B^+) \cap B^- \neq \emptyset$. Otherwise, there is an atom $x \in B$ such that $x \notin \mathrm{Dcl}\,(P, B^-)$. We observe that since B covers $\mathrm{NAnt}\,(P)$ and since $\mathrm{not}\,(x) \in \mathrm{NAnt}\,(P_B)$ implies $x \notin B$,

$$\begin{aligned}\mathrm{Dcl}\bigl(P_B, \mathrm{not}\,(\mathrm{NAnt}\,(P))\bigr) &= \mathrm{Dcl}\bigl(P_B, \mathrm{not}\,(\mathrm{NAnt}\,(P) - B)\bigr)\\ &= \mathrm{Dcl}\,(P_B, B^-)\\ &\subseteq \mathrm{Dcl}\,(P, B^-).\end{aligned}$$

Therefore, $x \notin \mathrm{Dcl}\,(P, B^-)$ implies $x \notin \mathrm{Dcl}\bigl(P_B, \mathrm{not}\,(\mathrm{NAnt}\,(P))\bigr)$, which in turn implies $\mathrm{not}\,(x) \in B$. In other words, $\mathrm{not}\,(B^+) \cap B^- \neq \emptyset$ and we have proved that **conflict** satisfies (C1).

4 Backtracking Improvements

The decision procedure **smodels** employs chronological backtracking search. This simple search method blindly explores all dead ends of the search space without exploiting any properties of the problem domain to lessen the amount of work. Ideally, we want a search method that backtracks directly to the cause of a conflict, skipping the parts not responsible for the lack of a stable model. However, finding the cause of a conflict is problematic. Instead, we investigate the use of lookahead as a way to try to avoid the issue of backjumping. We consider the remaining cases that in all circumstances require backjumping in Subsection 4.2.

4.1 Adding Lookahead

Lookahead, as its name indicates, is based on the idea of testing every possible choice before committing to one. In the case of **smodels**, this means that we can avoid choosing literals that directly leads to a conflict. Namely, if the stable model Δ agrees with the set of literals B, but Δ does not agree with $B \cup \{l\}$, we know that Δ agrees with $B \cup \{\text{not}\,(l)\}$. Thus, lookahead immediately gives us a stronger pruning method: if **conflict** (P, B') returns true for $B' = $ **expand** $(P, B \cup \{l\})$, add $\text{not}\,(l)$ to B.

Apart from the better pruning method, lookahead also guarantees that conflicts are caused by the chronologically next to last choice. Assume that the literal l is the next to last choice and that **conflict** (P, B') returns false for $B' = $ **expand** $(P, B \cup \{l\})$, but that the choice of l does not affect the conflict that arises when we choose to add the literal l_2 to B'. That is to say, **conflict** (P, B'') returns true for $B'' = $ **expand** $(P, B \cup \{l_2\})$. Since by using lookahead we examine all possible ways of adding a literal to B, we notice this conflict prior to choosing l. Hence, the literal l is never chosen, and by the same argument, neither is any other literal not pertinent to the conflict.

We add lookahead to **smodels** by replacing the arbitrary choice of an atom not covered by B' with the return value of the function **lookahead**, see Figure 3. As testing every literal not covered by B can be rather expensive, we implement a scheme that evades most of the useless tests. The scheme is based on the fact that the implementation of **expand** is monotonic and computes a fixed point and that the implementation of **conflict** respects this monotonicity in the sense that $B_1 \subseteq B_2$ implies that **conflict** (P, B_1) returns false whenever **conflict** (P, B_2) does. Hence, if testing against a literal l does not reveal a conflict, then neither does testing against any other literal in **expand** $(P, B \cup \{l\})$.

function lookahead (P, B)
$A := \mathrm{NAnt}\,(P) - \mathrm{Atoms}\,(B)$
$A := A \cup \mathrm{not}\,(A)$
while $A \neq \emptyset$ **do**
  Take any literal $\ell \in A$
  $B' :=$ **expand** $(P, B \cup \{\ell\})$
  if conflict (P, B') returns true **then**
    return ℓ
  else
    $A := A - B'$
  endif
end while
return any atom in $\mathrm{NAnt}\,(P)$ not covered by B

Figure 3: The function **lookahead**

4.2 Adding Backjumping

Although many conflicts are discovered using lookahead, there are still situations when the decision procedure exhaustively searches through assignments that are not relevant to a set of conflicts. A simple example is furnished by the union of two programs that do not share atoms. We may assume that the first program has several stable models and that the second one has none. If we always begin by choosing atoms from the first program, then we will search through all stable models of the first program before we discover that the joint program has no stable models.

To alleviate this problem we introduce the notion of a path between two atoms in a program. A logic program generates an undirected graph whose nodes are the atoms appearing in the program and whose edges connect every pair of atoms appearing in the same rule. In effect, if

$$a_1 \leftarrow a_2, \ldots, a_m, \mathrm{not}\,(a_{m+1}), \ldots, \mathrm{not}\,(a_n)$$

is a rule in the program, then every pair (a_i, a_j) for $1 \leq i, j \leq n$ is an edge in the graph. A path between two atoms in the program is then a path in the corresponding graph. Moreover, we accept empty paths and define a path between two literals as the path between the atoms the literals cover.

There is a useful relation between proofs and paths.

Proposition 4.1. *If P is a minimal proof of x from the set of literals B, then there is a path from x to any atom in* $\mathrm{Atoms}\,(P)$.

Proof. Let P be a minimal proof of x from B. Let

$$P' = \{r \in P \mid \text{there is a path from } x \text{ to the head of } r\}.$$

If $P = P'$ we are done. Otherwise, the minimality of P implies that P' is not a proof of x, i.e., there is a rule in P' that has an atom y in its body such that $y \notin \mathrm{Dcl}\,(P', B)$. But this means that there is a rule $r \in P - P'$ whose head is y. Thus, there is a path from x to the head of r, and this contradicts the definition of P'. Hence, there is a path from x to any atom in $\mathrm{Atoms}\,(P)$. □

We are interested in the absence of paths. In particular, we will prove that if a literal $l \in B$ is not connected to the literal chosen immediately before we encounter a conflict, then we will still encounter the conflict if we remove l from B. For this we need three lemmas.

Lemma 4.2. *Let P be a program, let B be a set of literals and let l be a literal. If $x \notin \mathrm{Dcl}\big(P_{B\cup\{l\}}, \mathrm{not}\,(\mathrm{NAnt}\,(P))\big)$ and there is no path between x and l in P_B, then $x \notin \mathrm{Dcl}\big(P_B, \mathrm{not}\,(\mathrm{NAnt}\,(P))\big)$.*

Proof. Assume that there is no path between x and l in P_B. If the atom $x \in \mathrm{Dcl}\big(P_B, \mathrm{not}\,(\mathrm{NAnt}\,(P))\big)$, then by Proposition 4.1 there is a proof $P' \subseteq P_B$ of x such that $\mathrm{not}\,(l)$ does not appear in P'. Hence, $P' \subseteq P_{B\cup\{l\}}$ and $x \in \mathrm{Dcl}\big(P_{B\cup\{l\}}, \mathrm{not}\,(\mathrm{NAnt}\,(P))\big)$. □

Lemma 4.3. *Let P be a program, let B be a set of literals and let l_1 and l_2 be literals. If $l_2 \in \mathrm{Cl}\,(P, B \cup \{l_1\})$ and there is no path between l_1 and l_2 in P_B, then $l_2 \in \mathrm{Cl}\,(P, B)$.*

Proof. Assume that $l_2 \notin \mathrm{Cl}\,(P, B)$. If $l_2 \in \mathrm{Cl}\,(P, B \cup \{l_1\})$, then by the definition of $\mathrm{Cl}\,(\cdot,\cdot)$ there is a path from l_1 to l_2. □

Lemma 4.4. *Let P be a program, let B be a set of literals and let l_1 and l_2 be literals. If $l_2 \in$ **expand** $(P, B \cup \{l_1\})$ and there is no path between l_1 and l_2 in P_B, then $l_2 \in$ **expand** (P, B).*

Proof. Let B be any set of literals. If the literal $l_3 \notin \mathrm{Cl}\,(P, B \cup \{l_1\})$ but $l_3 \in \mathrm{Cl}\,(P, B \cup \{l_1, l_2\})$, then, by Lemma 4.3, there is a path between l_3 and l_2. Similarly, if the literal $l_3 \in \mathrm{Dcl}\big(P_{B\cup\{l_1\}}, \mathrm{not}\,(\mathrm{NAnt}\,(P))\big)$ but $l_3 \notin \mathrm{Dcl}\big(P_{B\cup\{l_1,l_2\}}, \mathrm{not}\,(\mathrm{NAnt}\,(P))\big)$, then, by Lemma 4.2, there is a path between l_3 and l_2. Thus, if there is a path from l_1 to l$_2$, then there is a path from l_1 to l_3. Hence, if $l_2 \notin$ **expand** (P, B) and $l_2 \in$ **expand** $(P, B \cup \{l_1\})$, then, by induction over the definition of **expand**, there is a path between l_1 and l_2. □

Careful examination of Lemmas 4.2 and 4.3 reveals that we can restrict ourselves to paths that do not go through atoms covered by B^-. Henceforth, we get backtracking improvements by assuming this restriction.

We are now ready to prove the following technical result.

Theorem 4.5. *Let P be a program, let B be a set of literals, and let l_1 and l_2 be two literals such that there is no path from l_1 to l_2 in P_B and such that* **conflict** (P, B') *returns true for* $B' = \mathbf{expand}\,(P, B \cup \{l_1, l_2\})$ *but* $\mathbf{conflict}\big(P, \mathbf{expand}\,(P, B \cup \{l_1\})\big)$ *returns false. Then,* **conflict** (P, B'') *returns true for* $B'' = \mathbf{expand}\,(P, B \cup \{l_2\})$.

Proof. Assume that there is no path from l_1 to l_2 in P_B, and assume that the function **conflict** (P, B') returns true for $B' = \mathbf{expand}\,(P, B \cup \{l_1, l_2\})$ but $\mathbf{conflict}\big(P, \mathbf{expand}\,(P, B \cup \{l_1\})\big)$ returns false. Then, there is a literal l such that $l, \mathrm{not}\,(l) \in \mathbf{expand}\,(P, B \cup \{l_1, l_2\})$. Without loss of generality we assume that $l \notin \mathbf{expand}\,(P, B \cup \{l_1\})$. Now, Lemma 4.4 implies that there is a path from l to l_2. Since there is no path from l_1 to l_2, there is therefore no path from l_1 to l. Thus, Lemma 4.4 implies that $l \in \mathbf{expand}\,(P, B \cup \{l_2\})$.

If the literal $\mathrm{not}\,(l)$ is not in $\mathbf{expand}\,(P, B \cup \{l_1\})$, we similarly get $\mathrm{not}\,(l) \in \mathbf{expand}\,(P, B \cup \{l_2\})$. Assume $\mathrm{not}\,(l) \in \mathbf{expand}\,(P, B \cup \{l_1\})$. If $\mathrm{not}\,(l) \in \mathbf{expand}\,(P, B)$, then by (E1) $\mathrm{not}\,(l) \in \mathbf{expand}\,(P, B \cup \{l_2\})$. If $\mathrm{not}\,(l) \notin \mathbf{expand}\,(P, B)$, then by Lemma 4.4 there is a path from $\mathrm{not}\,(l)$ to l_1. But this is not possible since, as shown above, there is no path between l and l_1.

In conclusion, $l, \mathrm{not}\,(l) \in \mathbf{expand}\,(P, B \cup \{l_2\}) = B''$ and it follows that **conflict** (P, B'') returns true. □

Theorem 4.5 tells us what literals we can skip during backtracking from a conflict. Since we also backtrack from stable models for which the function **test** returns false and since we have to guarantee completeness of the decision procedure, we have to distinguish between these two modes of backtracking. Specifically, backtracking from stable models must be chronological in order to avoid the risk of omitting some stable models.

The decision procedure incorporating backjumping from conflicts and backtracking from stable models is presented in Figure 4. The algorithm assumes the existence of two global variables: $conflict$ and *top*. The variable $conflict$ holds the last choice leading up to a conflict, while the variable *top* keeps the level, or depth of recursion, above which backtracking is chronological.

The soundness of **smodels** is obviously not affected by backjumping since backjumping only prevents the algorithm from exploring certain parts of the search space. Completeness, on the other hand, must be proved.

function smodels $(P, B, level)$
$B' :=$ **expand** (P, B)
if conflict (P, B') returns true **then**
    return false
else if $\mathrm{NAnt}(P)$ is covered by B' **then**
    $top := level$
    return test$\big(\mathrm{Dcl}(P, B')\big)$
else
    $conflict := \ell :=$ **lookahead** (P, B')
    if smodels $(P, B' \cup \{\ell\}, level + 1)$ returns true **then**
        return true
    else if $level < top$ or there is a path from ℓ to $conflict$ in $P_{B'}$ **then**
        if $level < top$ **then**
            $top := level$
        endif
        return smodels$\big(P, B' \cup \{\mathrm{not}(\ell)\}, level + 1\big)$
    else
        return false
    endif
endif

Figure 4: Incorporating backjumping into **smodels**

Theorem 4.6 (Completeness). *Let P be a set of rules and let B be a set of literals. If there is a stable model Δ of P agreeing with B such that* **test** (Δ) *returns true, then* **smodels** (P, B, top) *returns true.*

Proof. We show that completeness is not lost from introducing backjumping. Since backtracking is chronological, and therefore exhaustive, for a depth of recursion smaller than top, we only have to consider the case $level \geq top$.

Assume that there is no stable model agreeing with $B \cup \{l_0\}$ and that when **smodels** $(P, B \cup \{l_0\}, level + 1)$ returns false, there is no path between the literal $conflict$ and l_0. Furthermore, assume that Δ is a stable model agreeing with the set $B \cup \{\mathrm{not}(l_0), l_1, \ldots, l_n\}$ and that the set covers $\mathrm{NAnt}(P)$. Then, as was seen in Theorem 2.5, **conflict** (P, B') returns true for $B' =$ **expand** $(P, B \cup \{l_0, l_1, \ldots, l_n\})$, since there is no stable model agreeing with B'. Take i such that $conflict = l_i$ or $conflict = \mathrm{not}(l_i)$, and let j be the smallest number for which **conflict**$\big(P,$ **expand** $(P, B \cup \{l_i, l_0, l_1, \ldots, l_j\})\big)$ returns true and **conflict**$\big(P,$ **expand** $(P, B \cup \{l_0, l_1, \ldots, l_j\})\big)$ returns false. Then, **conflict**$\big(P,$ **expand** $(P, B \cup \{l_i, l_1, \ldots, l_j\})\big)$ returns true by Theorem 4.5 and (E1). Hence, **conflict**$\big(P,$ **expand** $(P, B \cup \{\mathrm{not}(l_0), l_1, \ldots, l_n\})\big)$ also returns true. But this contradicts the existence of Δ by (C2). Thus, there is no stable model agreeing with $B \cup \{\text{not}\,(l_0)\}$, and as the decision procedure is sound, $\mathbf{smodels}\big(P, B \cup \{\text{not}\,(l_0)\}, \mathit{level}+1\big)$ returns false.

We complete the proof by induction on the size of

$$nc(P, B \cup \{l\}) = \text{NAnt}\,(P) - \text{Atoms}\,(B \cup \{l\})\,.$$

If $B \cup \{l\}$ covers $\text{NAnt}\,(P)$, if the function $\mathbf{smodels}\,(P, B \cup \{l\}, \mathit{level}+1)$ returns false and if $\mathit{level} \geq \mathit{top}$, then there is no stable model agreeing with $B \cup \{l\}$. Let $nc(P, B \cup \{l\}) \neq \emptyset$, let $\mathbf{smodels}\,(P, B \cup \{l\}, \mathit{level}+1)$ return false and let $\mathit{level} \geq \mathit{top}$. Then, $\mathbf{smodels}\,(P, B \cup \{l, l_2\})$ returns false for $l_2 \in nc(P, B \cup \{l\})$, and by induction there is no stable model agreeing with $B \cup \{l, l_2\}$. If there is no path from l_2 to $\mathit{conflict}$ in $P_{B \cup \{l\}}$, then there is no stable model agreeing with $B \cup \{l, \text{not}\,(l_2)\}$ either. If, on the other hand, there is a path from l_2 to $\mathit{conflict}$, then $\mathbf{smodels}\big(P, B \cup \{l, \text{not}\,(l_2)\}\big)$ returns false and there is again no stable model that agrees with $B \cup \{l, \text{not}\,(l_2)\}$. Therefore, there is no stable model that agrees with $B \cup \{l\}$. $\square$

5 Computing Deductive Closures

Computing deductive closures is a simple matter — you just apply the inference rules until nothing more can be deduced. Moreover, the key to an efficient implementation lies in the right data structures. In this section we discuss implementations of the deductive closures $\mathrm{Cl}\,(P, B)$ and $\mathrm{Dcl}\big(P_B, \mathrm{not}\,(\mathrm{NAnt}\,(P))\big)$ that are based on a linear time algorithm of Dowling and Gallier [DG84], and show how they tie in with an efficient implementation of backtracking.

The data structures should support fast deductions. To this end we associate to every rule three counters *pos*, *neg*, and *inactive*, two lists *poslist* and *neglist*, and an atom *head*. The idea is that *poslist* and *neglist* hold the atoms that appear as atoms respectively not-atoms in the body of the rule, and that *head* holds the head of the rule. The counters *pos* and *neg*, in turn, hold the number of atoms in *poslist* not covered by B^+ and the number of atoms in *neglist* not covered by B^-. Finally, the counter *inactive* holds the sum of the atoms in *poslist* covered by B^- and the atoms in *neglist* covered by B^+. For example, the rule r

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

has $r.head = a$, $r.poslist = \{b, c\}$, and $r.neglist = \{d, e\}$. If the set $B = \{b, d\}$, then $r.pos = 1$, $r.neg = 2$, and $r.inactive = 1$. In summary,

$$\begin{aligned}
r &= h \leftarrow a_1, \ldots, a_n, \mathrm{not}\,(b_1), \ldots, \mathrm{not}\,(b_m), \\
r.head &= h, \\
r.poslist &= \{a_1, \ldots, a_n\}, \\
r.neglist &= \{b_1, \ldots, b_m\}, \\
r.pos &= \left| r.poslist - B^+ \right|, \\
r.neg &= \left| r.neglist - \mathrm{Atoms}\left(B^-\right) \right|, \text{ and} \\
r.inactive &= \left| \mathrm{Atoms}\left(B^-\right) - r.poslist \right| + \left| B^+ - r.neglist \right|.
\end{aligned}$$

To each atom we join the three lists *poslist*, *neglist*, and *headlist* containing respectively the rules in which the atom appears as an atom in the body, as a not-atom in the body, and as a head. In addition, every atom has a counter *headof* containing the number of rules in *headlist* whose *inactive*-counter is zero, and a flag *inB* which is true if the atom belongs to B^+ and which is false if the atom is covered by B^-. Briefly,

$$\begin{aligned}
a.poslist &= \{r \mid a \in r.poslist\}, \\
a.neglist &= \{r \mid a \in r.neglist\}, \\
a.headlist &= \{r \mid a = r.head\}, \\
a.headof &= \left|\{r \in a.headlist \mid r.inactive = 0\}\right|, \text{ and}
\end{aligned}$$

$$a.inB = \begin{cases} \text{true} & \text{if } a \in B^+, \\ \text{false} & \text{if not}\,(a) \in B^-, \\ \text{not set} & \text{otherwise.} \end{cases}$$

We can now easily recognize when a deductive step is possible. If the rule r has $r.pos = r.neg = 0$, then we infer that $r.head$ should be in the closure. If the counter $a.headof$ is zero, then not (a) must be in the closure. Furthermore, if not $(r.head)$ is in the closure and if one of $r.pos$ and $r.neg$ is zero and the other is one, then there is a literal l in the body of r such that not (l) should be in the closure. Finally, if $a = r.head$ is in the closure and $a.headof$ is one, then every literal of the body of r is put in the closure.

After these preparations we can describe the basis for the algorithm computing $\mathrm{Cl}\,(P, B)$. The computation revolves around two queues, *posqueue* and *negqueue*, and the application of inference rules whose results are stored on the queues. Basically, when an atom is removed from a queue we check if the atom is in B, i.e., we check if the flag inB is set. If it is not, then we add the atom to B by setting the flag to true if the atom was removed from *posqueue* and to false if the atom was removed from *negqueue*. Then, we examine the rules in *poslist*, *neglist*, and *headlist* to see if we can perform any inferences. The results of these are then pushed onto *posqueue* or *negqueue* as appropriate: the atoms are pushed onto *posqueue* and the not-atoms onto *negqueue*. We repeat until both queues are empty. The full description of the algorithm **cl** is found in Figures 5 and 6.

When we want to compute $\mathrm{Cl}\,(P, B)$, we initialize the queue *posqueue* with the heads of the rules whose bodies are empty and with B^+, and we initialize the queue *negqueue* with $\mathrm{Atoms}\,(B^-)$. Observe that the algorithm can also be used incrementally. If $\mathrm{Cl}\,(P, B)$ has been computed and we are going to compute $\mathrm{Cl}\,(P, B \cup \{x\})$, then we just push x onto *posqueue* and call **cl**. Similarly, we compute $\mathrm{Cl}\big(P, B \cup \{\text{not}\,(x)\}\big)$ by first pushing x onto *negqueue* and then calling **cl**.

The function **cl** computes the deductive closure $\mathrm{Cl}\,(P, B)$ linearly in the size of the program P. This is seen by noting that the list *headlist* is traversed at most twice for every atom and that the lists *poslist* and *neglist* are traversed at most once for every atom and at most twice for every rule. To be precise, the list *a.headlist* is traversed in the function **cl** only when the flag $a.inB$ is being set and this can happen only once. The same holds for the lists *a.poslist* and *a.neglist*. Moreover, the only other place where the list *a.headlist* is examined is in the function **inactivate** and there too only once, when $a.headof$ is decremented to one.

The lists *r.poslist* and *r.neglist* are only traversed in the function **backchain**.

By the same argument as above, this function is called from **cl** at most once for every rule. The function **backchain** is also called from **fire**, but then $r.head.inB$ is false and a list is gone through if $r.pos + r.neg = 1$, which only happens once. Finally, **backchain** is called from **inactive**, but only once when $headof = 1$.

5.1 Diminishing Deductive Closures

The deductive closure $\mathrm{Dcl}\big(P_B, \mathrm{not}\,(\mathrm{NAnt}\,(P))\big)$ can be computed in a manner similar to that of $\mathrm{Cl}\,(P, B)$. However, since P_B diminishes as the set B grows, the data structures would then have to be reinitialized every time the set B changes. To avoid this we directly compute a smaller deductive closure from a larger one. Hence, we exchange a guaranteed linear computation for a potentially sub-linear one.

For every atom a we need a boolean flag $a.closure$ that expresses whether the atom is in the closure or not, and for every rule r we need a new counter $r.pos_2$ holding the number of atoms in $r.poslist$ that are in the closure. The counter $r.pos_2$ is therefore zero if the rule r takes part in the deduction. The computation proceeds as follows. If the closure $\mathrm{Dcl}\big(P_B, \mathrm{not}\,(\mathrm{NAnt}\,(P))\big)$ has been computed and we want to compute a closure $\mathrm{Dcl}\big(P_{B'}, \mathrm{not}\,(\mathrm{NAnt}\,(P))\big)$ for $B \subseteq B'$, then we initialize the queue $queue$ to contain the heads of the rules in $P_B - P_{B'}$ that are in the larger closure. The queue now holds some atoms that are not necessarily in the smaller deductive closure. To find all such atoms we repeat the following until the queue is empty. Remove an atom a from the queue. If $a.closure$ is true, then set $a.closure$ to false and increment $r.pos_2$ for every rule r in $a.poslist$. If $r.pos_2$ reaches one and r is in $P_{B'}$, which we assume is the case if $r.inactive$ is zero, then add the head of r to the queue.

We have now found all atoms in the larger closure that potentially are not in the smaller one. To find the final closure we put all atoms that went onto the queue in the first stage and whose $headlist$ contains a rule in $P_{B'}$ with pos_2 equal to zero onto the queue. Then, we again propagate truth values until the queue is empty: we remove an atom a from the queue, if $a.closure$ is false, we set it to true and decrement $r.pos_2$ for all rules r in $a.poslist$. If $r.pos_2$ becomes zero, then $r.head$ is added to the queue.

The linear-time algorithm is presented in Figure 7.

```
function cl ()
while posqueue or negqueue is not empty do
    while posqueue is not empty do
        a := pop (posqueue)
        if a.inB is not set then
            a.inB := true
            for each rule r ∈ a.poslist do
                r.pos := r.pos − 1
                fire (r)
            end for
            for each rule r ∈ a.neglist do
                inactivate (r)
            end for
            if a.headof = 1 then
                Let r be the rule in a.headlist for which r.inactive = 0
                backchain (r)
            endif
        endif
    end while
    while negqueue is not empty do
        a := pop (negqueue)
        if a.inB is not set then
            a.inB := false
            for each rule r ∈ a.neglist do
                r.neg := r.neg − 1
                fire (r)
            end for
            for each rule r ∈ a.poslist do
                inactivate (r)
            end for
            if a.headof > 0 then
                for every rule r ∈ a.headlist do
                    backchain (r)
                end for
            endif
        endif
    end while
end while
```

Figure 5: Computing $\mathrm{Cl}\,(P, B)$

```
function fire (r)
if r.pos = 0 and r.neg = 0 then
    push (r.head, posqueue)
else if r.head.inB = false then
    backchain (r)
endif

function inactivate (r)
r.inactive := r.inactive + 1
if r.inactive = 1 then
    b := r.head
    b.headof := b.headof − 1
    if b.headof = 0 then
        push (b, negqueue)
    else if b.inB = true and b.headof = 1 then
        Let r' be the rule in b.headlist for which r'.inactive = 0
        backchain (r')
    endif
endif

function backchain (r)
if r.head.inB = true then
    for every a ∈ r.poslist do
        push (a, posqueue)
    end for
    for every a ∈ r.neglist do
        push (a, negqueue)
    end for
else
    if r.pos = 0 and r.neg = 1 then
        Let a be the atom in r.neglist for which a.inB is not set
        push (a, posqueue)
    else if r.pos = 1 and r.neg = 0 then
        Let a be the atom in r.poslist for which a.inB is not set
        push (a, negqueue)
    endif
endif
```

Figure 6: The functions **fire**, **inactivate** and **backchain**

```
function dcl ()
N := ∅
while queue is not empty do
    a := pop (queue)
    if a.closure = true then
        a.closure := false
        N := N ∪ {a}
        for each rule r ∈ a.poslist do
            r.pos_2 := r.pos_2 + 1
            if r.pos_2 = 1 and r.inactive = 0 then
                push (r.head, queue)
            endif
        end for
    endif
end while
for each atom a ∈ N do
    if there is a rule r ∈ a.headlist such that r.pos_2 = 0
        and r.inactive = 0 then
        push (r.head, queue)
    endif
end for
while queue is not empty do
    a := pop (queue)
    if a.closure = false then
        a.closure := true
        for each rule r ∈ a.poslist do
            r.pos_2 := r.pos_2 − 1
            if r.pos_2 = 0 and r.inactive = 0 then
                push (r.head, queue)
            endif
        end for
    endif
end while
```

Figure 7: Computing $\mathrm{Dcl}\big(P_B, \mathrm{not}\,(\mathrm{NAnt}\,(P))\big)$

5.2 Backtracking

Backtracking is the act of restoring the state of the data structures to what they were earlier in the computation. Since we want a linear-space implementation of **smodels**, we must not store snapshots of the states of the data structures but instead store the changes that take place.

In fact, it is enough to keep track of the changes to B. For example, if the atom a is added to B, i.e., the flag $a.inB$ is set to true, then for every rule r in *a.poslist* the counter *r.pos* is decremented by one and for every rule r in *a.neglist* the counter *r.inactive* is incremented by one. In addition, if the counter *r.inactive* ends up being equal to one, then $a.headof$ is decremented by one, and nothing else is modified. Thus, the values of the counters are completely determined from their previous values, and the previous values are similarly completely determined by their new values. Hence, if we during backtracking undo the changes to B in the opposite order in which they took place, then we can compute the old values of the counters from this information alone.

6 Heuristics

The philosophy behind the algorithm for computing stable models comprises the principle that the search space should be pruned as much as possible. Naturally, this principle should also be manifest in the heuristic that chooses the order in which the search space is explored.

We first consider the basic algorithm that do not include lookahead. Since we want to prune the search space, we have to find a literal whose inclusion in the set of literals B causes $\mathbf{expand}\,(P, B)$ to return a larger set B'. A simple and fast way of achieving this is to examine the rules in P_B for a rule whose body, save one not-atom, is in B and whose head is not in B. If the not-atom is included in B, then it is guaranteed that the head will be included in the set B'. When no such rule exists, we can try to create one by choosing a not-atom from the rule that has the least not-atoms not in B. This heuristic is easily computed in linear time by looking for a rule whose counter *inactive* is zero, whose counter *pos* is zero and whose counter *neg* is positive.

Notice that if $B = \mathbf{expand}\,(P, B)$ and all heads are covered by B, then B covers $\mathrm{NAnt}\,(P)$ and the heuristic is not applied. Moreover, notice that if every rule in P_B whose head is not covered by B has an atom a in its body such that $a \notin B$, then $a \notin \mathrm{Dcl}\big(P_B, \mathrm{not}\,(\mathrm{NAnt}\,(P))\big)$ and consequently $\mathrm{not}\,(a) \in B$. But this contradicts the existence of such rules in P_B. Hence, we can always find a rule on which we can evaluate the heuristic.

If we use lookahead, we can exploit the many invocations of **expand** in the function **lookahead** to obtain a much stronger heuristic. The pruning of the search space that the choice of a literal l is responsible for is reflected in the sets

$$B_1 = \mathbf{expand}\,(P, B \cup \{l\}) \quad \text{and} \quad B_2 = \mathbf{expand}\big(P, B \cup \{\mathrm{not}\,(l)\}\big).$$

By assuming that the search space is a complete binary tree of height N and by letting $p = |(B_1 - B) \cap \mathrm{NAnt}\,(P)|$ and $n = |(B_2 - B) \cap \mathrm{NAnt}\,(P)|$, we can approximate the size of the remaining search space with the number

$$\begin{aligned} 2^{N-p} + 2^{N-n} &= 2^N \left(2^{-p} + 2^{-n}\right) \\ &= 2^N \frac{2^n + 2^p}{2^{p+n}}. \end{aligned}$$

Minimizing this number is equal to minimizing

$$\log \frac{2^n + 2^p}{2^{p+n}} = \log\,(2^n + 2^p) - (p + n).$$

We note that

$$2^{\max\{n,p\}} < 2^n + 2^p < 2^{\max\{n,p\}+1},$$
$$\max\{n,p\} < \log\left(2^n + 2^p\right) < \max\{n,p\} + 1,$$

and

$$-\min\{n,p\} < \log\left(2^n + 2^p\right) - (p+n) < 1 - \min\{n,p\}.$$

Hence, it suffices to maximize $\min\{n,p\}$. Moreover, if two different literals have equal minimum, $\min\{n,p\}$, then we should choose the one with the greater maximum, $\max\{n,p\}$.

Since **lookahead** does not compute **expand** for all literals, there are situations in which one or both of n and p are not computed for a particular literal. When we arrive at such a case we compute one of n and p, if necessary, and compare it to the highest minimum seen so far. If the computed value of n or p is smaller than the highest minimum, then we can not get a new higher minimum and we do not have to compute the other value. Thus, we can avoid some **expand** calls and still retain a strong heuristic.

We can further refine the computation of the heuristic by noticing that the size of $(B' - B) \cap \mathrm{NAnt}\,(P)$ is an upper bound on the minimum of n and p for any literal in $B' = \textbf{expand}\,(P, B \cup \{l\})$. Hence, when this bound is lower than the highest minimum we do not have to compute either of n and p.

The worst-case complexity for computing **lookahead** and the heuristic is $O(n^3)$ for n proportional to the size of the program: the function **lookahead** might call **expand** a linear number of times, the function **expand** might compute a linear number of deductive closures, and a deductive closure might take linear time to compute.

7 Complexity Analysis

It is of interest to establish upper bounds on the complexity of computing stable models. Since the problem of finding a stable model of a logic program is NP-complete [MT91], the bounds will be of an exponential nature.

We are mainly interested in bounding the computation for finding one stable model or for determining that there are no stable models. However, we will need results on how many stable models a program may have.

The complexity of the algorithm for computing stable models will be measured in the number of times the decision procedure **smodels** is called. Since **smodels** calls itself recursively at most two times, we can represent the search space as a binary tree whose nodes are instances of the set of literals that is passed as an argument to the procedure. By noting that the height of the tree is bounded by the number of atoms in $\mathrm{NAnt}\,(P)$, we arrive at the trivial upper bound

$$2^{n+1} - 1$$

given by the number of nodes in a complete tree of height $n = |\mathrm{NAnt}\,(P)|$.

With the help of extremal logic programs, i.e., programs with a maximal number of stable models, we can determine a lower bound on the upper bounds. For $n = 3k$, where k is a positive integer, there is an extremal program P_n with $3^{n/3}$ stable models [CT96]. We let $n = 3k + 1$ and construct a program

$$P = P_{n-1} \cup \{a \leftarrow \mathrm{not}\,(a)\}$$

such that the atom a does not appear in P_{n-1}. By assuming that the atom a is always chosen as the last atom from $\mathrm{NAnt}\,(P)$, we see that all stable models of P_{n-1} will be constructed before the decision procedure notices that there are no stable models of P. Thus, the search tree contains at least

$$3^{(n-1)/3} > 2^{0.52(n-1)}$$

nodes, and any upper bound on the complexity of the algorithm will asymptotically lie above $\Omega(3^{n/3})$. A more complicated program,

$$\begin{array}{lll}
a \leftarrow \mathrm{not}\,(b), \mathrm{not}\,(c), g & d \leftarrow \mathrm{not}\,(e), a & d \leftarrow \mathrm{not}\,(e), b \\
b \leftarrow \mathrm{not}\,(a), \mathrm{not}\,(c), g & e \leftarrow \mathrm{not}\,(f), a & e \leftarrow \mathrm{not}\,(f), b \\
c \leftarrow \mathrm{not}\,(a), \mathrm{not}\,(b), g & f \leftarrow \mathrm{not}\,(d), a & f \leftarrow \mathrm{not}\,(d), b
\end{array}$$

$$
\begin{array}{lll}
p_i \leftarrow \text{not}\,(q_i), \text{not}\,(r_i), g & \qquad d \leftarrow \text{not}\,(e), c & \qquad g \leftarrow \\
q_i \leftarrow \text{not}\,(p_i), \text{not}\,(r_i), g & \qquad e \leftarrow \text{not}\,(f), c & \\
r_i \leftarrow \text{not}\,(p_i), \text{not}\,(q_i), g & \qquad f \leftarrow \text{not}\,(d), c &
\end{array}
$$

$$
i = 1, \ldots, k,
$$

shows that the addition of lookahead and backjumping do not improve on this bound. The program consists of an extremal half

$$
\begin{aligned}
p_i &\leftarrow \text{not}\,(q_i), \text{not}\,(r_i), g \\
q_i &\leftarrow \text{not}\,(p_i), \text{not}\,(r_i), g \\
r_i &\leftarrow \text{not}\,(p_i), \text{not}\,(q_i), g
\end{aligned}
$$

with $3^{k/3}$ stable models that is connected to another half with no stable models. The trick is to prevent lookahead from noticing that there are no stable models while making the heuristic choose the literals in the extremal part of the program first.

8 Handling Variables

Some logic programs are more easily written with the help of variables than without. One can compute the stable models of these programs by first converting them to ordinary propositional programs, i.e., by grounding them. However, if this is done naively, it will produce a program containing a lot of rules that are not relevant to the stable model computation. In this section we examine how a class of logic programs with variables, the class of function-free range restricted logic programs, can be grounded without generating many of the rules that do not influence the set of stable models.

We begin by defining what a logic program with variables is. A term is a variable, a constant, or a function whose arguments are terms. An atom $p(t_1, \ldots, t_n)$ consists of an n-ary predicate symbol p together with its arguments, the terms $t_1, \ldots, t_n$. A ground term, atom, or program is one that is variable free. The Herbrand universe of a program consists of all ground terms that can be constructed from the constants and function symbols in the program. If there are no constants in the program, then an arbitrary constant is introduced.

The ground instantiation of a program P is the set of rules obtained from the rules in P by substituting the variables for the terms in the Herbrand universe in every possible way. For example, the ground version of the logic program

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

where the variables are denoted by uppercase letters, 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 stable models of a logic program with variables is defined to be the stable models of the corresponding ground instantiation.

A function-free rule is range restricted if every variable that appears in an atom in the head or in a not-atom in the body of the rule also appears in an atom in the body of the rule. For instance, the rules

$$a(X) \leftarrow b(X), \text{not}\,(c) \quad \text{and} \quad p \leftarrow q(X), \text{not}\big(r(X)\big)$$

are range restricted, but the rules

$$a(X) \leftarrow b(Y) \quad \text{and} \quad p \leftarrow \text{not}\big(q(X)\big)$$

are not. We say that a program is range restricted if every rule in the program is.

Our aim is to find an algorithm that directly computes a small subset of the ground instantiation of a range restricted logic program such that no stable models are lost. We achieve this by identifying a set of literals that all stable models agree with. Let P_r be a range restricted logic program, let P be the corresponding ground program, and let B be a set of literals. Furthermore, let

$$B_n = \{\text{not}\,(x) \mid x \in \text{NAnt}\,(P) - B \text{ and there is no rule in } P_r \text{ whose head can be transformed to } x \text{ by substituting the variables for constants}\}$$

and let

$$B_p = \{x \mid \text{there is a ground rule in } P_r \text{ whose head is } x, \text{ and the literals in the body of the rule are in } B \cup B_n\}.$$

If the stable model Δ of P agrees with B, then Δ agrees with B_n as $x \in \text{Atoms}\,(B_n)$ implies $x \notin \text{Dcl}\big(P, \text{not}\,(\overline{\Delta})\big) = \Delta$ by definition, and this, in turn, implies that Δ also agrees with B_p. Thus, Δ agrees with the set of literals

$$A = B \cup B_n \cup B_p.$$

Finally, we let

$$A' = \text{Dcl}\big(P, A \cup \text{not}\,(\text{NAnt}\,(P) - A)\big)$$

and

$$P' = \{r \in P \mid r = h \leftarrow a_1, \ldots, a_n, \text{not}\,(b_1), \ldots, \text{not}\,(b_m), a_1, \ldots, a_n \in A' \text{ and } b_1, \ldots, b_m \notin A\}.$$

Note that $\Delta \subseteq A'$ as $\overline{\Delta} \cap \text{NAnt}\,(P) \subseteq \text{NAnt}\,(P) - A$.

The programs P and P' have the same stable models.

Proposition 8.1. *The set Δ is a stable model of P agreeing with B if and only if Δ is a stable model of P' agreeing with B.*

Proof. Let Δ be a stable model of P agreeing with B. Since $P' \subseteq P$, $\text{Dcl}\big(P', \text{not}\,(\overline{\Delta})\big) \subseteq \text{Dcl}\big(P, \text{not}\,(\overline{\Delta})\big)$. Consider a rule $r \in \text{Proof}\big(P, \text{not}\,(\overline{\Delta})\big)$ of the form

$$h \leftarrow a_1, \ldots, a_n, \text{not}\,(b_1), \ldots, \text{not}\,(b_m).$$

Then, $a_i \in \Delta$ and $b_j \notin \Delta$. Since $\Delta \subseteq A'$, $a_i \in A'$, and since Δ agrees with A, $b_j \notin \Delta$ implies $b_j \notin A$. Hence, $r \in P'$ and

$$\text{Dcl}\big(P, \text{not}\,(\overline{\Delta})\big) = \text{Dcl}\big(\text{Proof}\,(P, \text{not}\,(\overline{\Delta})), \text{not}\,(\overline{\Delta})\big) \subseteq \text{Dcl}\big(P', \text{not}\,(\overline{\Delta})\big),$$

which proves that Δ is a stable model of P'.

Let Δ be a stable model of P' agreeing with B. Then, Δ agrees with B_n and therefore also with B_p. It follows that $\Delta \subseteq A'$. Let $r \in P - P'$ for

$$r = h \leftarrow a_1, \dots, a_n, \text{not}\,(b_1), \dots, \text{not}\,(b_m).$$

Then, there is an i such that $a_i \notin A'$ or $b_i \in A$. If $a_i \notin A'$, then $a_i \notin \Delta$, and if $b_i \in A$, then $b_i \in \Delta$. Thus, $r \notin \text{Proof}\big(P, \text{not}\,(\overline{\Delta})\big)$ and we conclude that $\text{Proof}\big(P, \text{not}\,(\overline{\Delta})\big) \subseteq P'$. Therefore,

$$\text{Dcl}\big(P, \text{not}\,(\overline{\Delta})\big) = \text{Dcl}\big(\text{Proof}\,(P, \text{not}\,(\overline{\Delta})), \text{not}\,(\overline{\Delta})\big) \subseteq \text{Dcl}\big(P', \text{not}\,(\overline{\Delta})\big)$$

and Δ is a stable model of P. □

From this point on we assume that $\text{not}\,(A^+) \cap A^- = \emptyset$, since otherwise there are no stable models agreeing with A. In order to compute P', we have to compute the sets A and A'. Since we can test membership in B_n without computing the ground program P, we can directly compute B_p. Hence, we have an explicit representation of $A^+ = B^+ \cup B_p$. We find P' by combining the computation of A' with the grounding of P_r as in Figure 8, where the set A' is stored in $N \cup U$. For example, if $B = \{t(b)\}$ and if

$$\begin{aligned} P_r = \big\{&p(a) \leftarrow,\ q(a) \leftarrow,\ r(a) \leftarrow \text{not}\big(s(a)\big), \\ &t(X) \leftarrow p(X), \text{not}\big(r(X)\big),\ p(X) \leftarrow q(X), \text{not}\big(t(X)\big)\big\}, \end{aligned}$$

then $P' = \big\{p(a) \leftarrow,\ q(a) \leftarrow,\ r(a) \leftarrow \text{not}\big(s(a)\big)\big\}$. Observe that, e.g., $t(a) \leftarrow p(a), \text{not}\big(r(a)\big)$ is not included as $r(a) \in A$ and $t(b) \leftarrow p(b), \text{not}\big(r(b)\big)$ is not included as $p(b) \notin A'$.

The problem with the aforementioned algorithm is that even if the ground program P' is small the intermediate calculations may make P_r huge. This is solved by representing the growth of the set P_r implicitly, i.e., by trading computation for memory. In short, instead of adding a new non-ground rule r' to P_r we construct this rule from r and the atoms in U during the following iterations over P_r. Observe that in spite of this one can not avoid super-exponential space complexity, since, for instance, the ground version of the program

$$\{p(X_1, \dots, X_n) \leftarrow q(X_1), \dots, q(X_n),\ q(a_1) \leftarrow,\ \dots, q(a_n) \leftarrow\}$$

contains $n + n^n$ rules.

```
function ground (P_r, A+)
P' := {r ∈ P_r | the positive literals of the body of r are in A+
        and the negative literals are not in not (A+)}
P_r := P_r − P'; U := ∅
N := A+ ∪ {a | there is a rule r ∈ P' whose head is a}
while N is not empty do
    Choose a ∈ N; N := N − {a}; U := U ∪ {a}
    for each rule r in P_r do
        for every rule r' resulting from substituting a minimal number
            of variables in r such that a is in the body of the rule r' do
            if no negative literal in the body of r' is in not (A+) then
                if r' is grounded and all positive literals in the body of r'
                    is in N ∪ U then
                    P' := P' ∪ {r'}
                    if the head of r' is not in U then
                        N := N ∪ {the head of r'}
                    endif
                else
                    P_r := P_r ∪ {r'}
                endif
            endif
        end for
    end for
end while
return P'
```

Figure 8: A procedure for grounding range restricted programs

9 Evaluation

In this section we compare an implementation of the **smodels** algorithm, *smodels*, with a state-of-the-art satisfiability checker, *tableau* [Cra96]. In addition, we investigate what influence the lookahead pruning technique and its related heuristic have on the implementation. Older versions of our algorithm have previously been shown to be superior to other stable model semantics decision procedures [Sim95, NS96a, NS96b]. It has also been shown that *smodels* can solve planning problems as fast as the best special purpose planners [DNK97]. In fact, *smodels* has in some cases even solved otherwise unsolvable problems.

The test domain is random 3-SAT, i.e., randomly generated propositional formulas in conjunctive normal form whose clauses contain exactly three atoms. We restrict ourselves to formulas with a clause to atom ratio of 4.3, since this particular ratio determines a region of hard satisfiability problems [CA96]. The formulas are generated by a program developed by Bart Selman [Sel94].

We translate the 3-SAT problem from a propositional formula into a logic program in the following way. For every atom a in the formula we include the rules

$$a \leftarrow \mathrm{not}\,(\overline{a}) \quad \text{and} \quad \overline{a} \leftarrow \mathrm{not}\,(a),$$

and for every clause c we include $c \leftarrow a$ if a is in the clause, and $c \leftarrow \overline{a}$ if $\neg a$ is in the clause. The satisfying assignments are then exactly given by the stable models that contain all clauses. For example, the formula

$$(a \vee \neg b \vee c) \wedge (\neg a \vee c \vee \neg d)$$

is translated into the program

$$\begin{array}{llll}
a \leftarrow \mathrm{not}\,(\overline{a}) & b \leftarrow \mathrm{not}\,(\overline{b}) & c \leftarrow \mathrm{not}\,(\overline{c}) & d \leftarrow \mathrm{not}\,(\overline{d}) \\
\overline{a} \leftarrow \mathrm{not}\,(a) & \overline{b} \leftarrow \mathrm{not}\,(b) & \overline{c} \leftarrow \mathrm{not}\,(c) & \overline{d} \leftarrow \mathrm{not}\,(d) \\
c_1 \leftarrow a & c_1 \leftarrow \overline{b} & c_1 \leftarrow c & \\
c_2 \leftarrow \overline{a} & c_2 \leftarrow c & c_2 \leftarrow \overline{d}, &
\end{array}$$

and we search for the stable models containing the atoms c_1 and c_2. One such model is $\{a, c, c_1, c_2\}$, which gives the satisfying assignment a and c true and b and d false.

The test results are tabulated in Figures 9–14. The maximum, minimum, and average is for every instance calculated from ten different runs on a pseudo-randomly shuffled logic program or propositional formula. We shuffle the programs and the formulas as a particular ordering might help the algorithms to avoid backtracking, thereby giving a skewed picture of their behavior. The times are in seconds and they represent the time to find one stable model or satisfying assignment, if one exists, or the time to decide that there are no stable models or satisfying assignments. The number of choice points, also known as branch points, describes how many times the algorithms use their heuristics to decide which atom to test next.

All tests were run on a Pentium 75MHz, with 32 MB of memory and the Linux 2.0.30 operating system. Neither program used backjumping as this had no effect on the number of choice points. The C++ program *smodels*, available at http://saturn.hut.fi/pub/smodels/, and the C program *tableau* were compiled using gcc version 2.7.2.1.

The program *smodels* is at worst about 25 times slower than *tableau* even if *smodels* always goes through a fewer number of choice points. The performance differences can be attributed to the overhead associated with the more general data structures of *smodels*, the lookahead pruning technique, and the encoding of the problem, which maps to an internal representation that is not as compact as the one a special purpose program can use. The program *tableau* also performs lookahead, but only on a small subset of all available atoms. This, in turn, accounts for the weaker pruning technique of *tableau*. The similar exponential behaviors of *tableau* and *smodels* is no surprise, since the two implementations effectively use the same propagation rules to prune the search space.

In Figure 13 we can see the difference between using lookahead together with its related heuristic and not using lookahead but using the simpler heuristic. In Figure 14 we have displayed the results one gets when using lookahead without heuristic. This is again contrasted with the timings of lookahead together with its heuristic. The graphs confirm that lookahead is a very strong pruning technique and that the heuristic is worth computing.

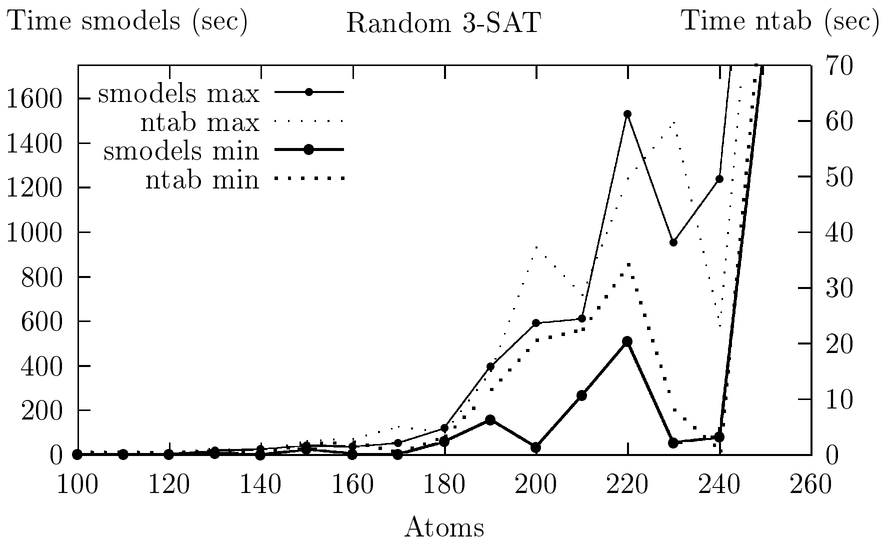

Figure 9: The test results

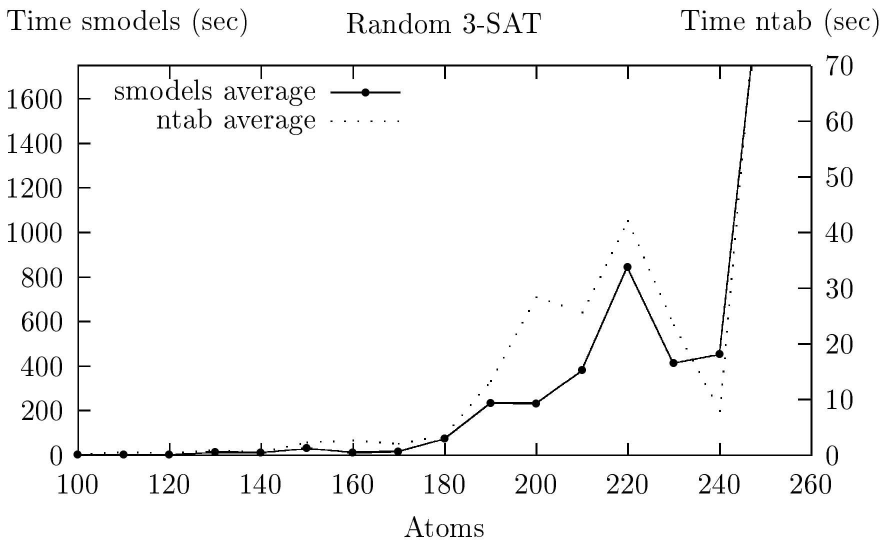

Figure 10: The average of the test results

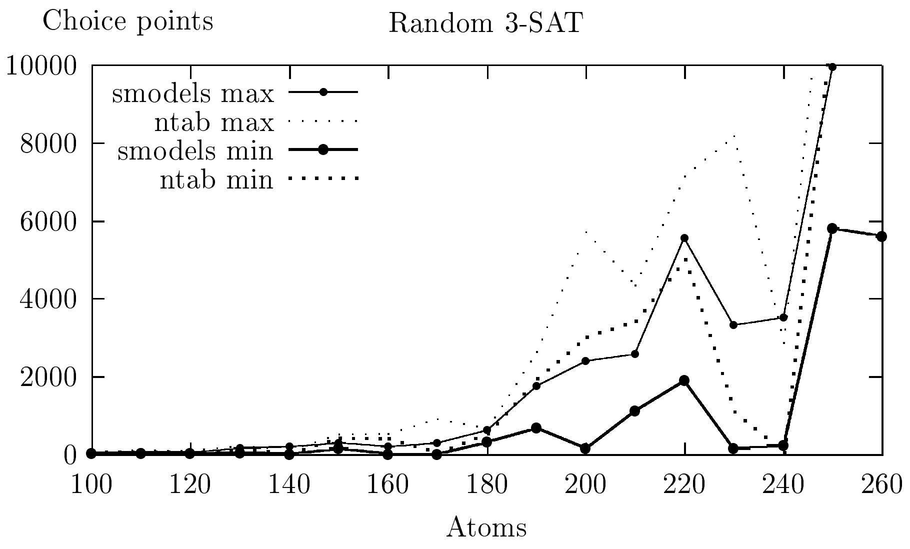

Figure 11: Choice points

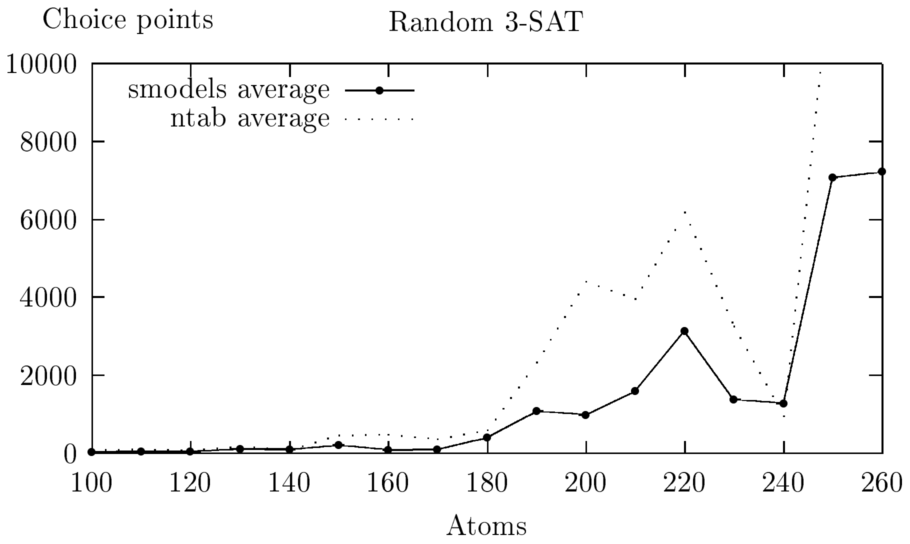

Figure 12: Choice points average

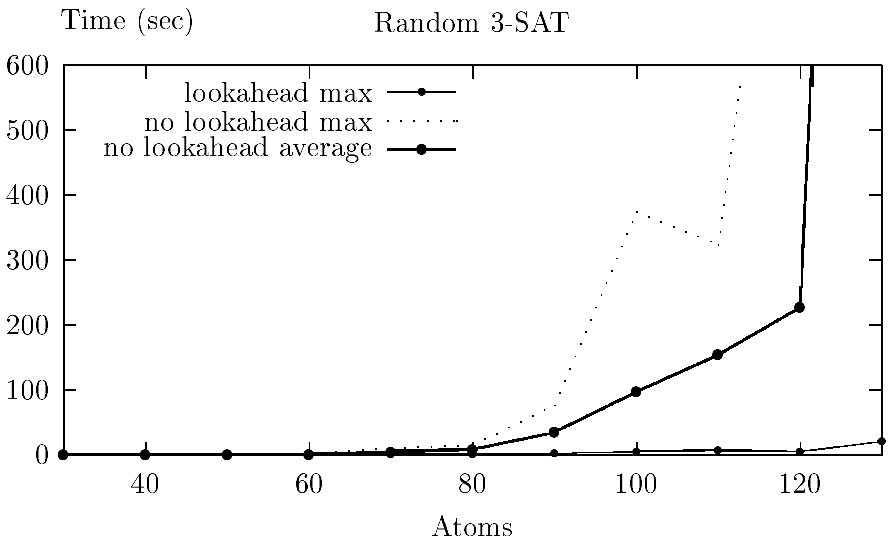

Figure 13: Lookahead versus no lookahead

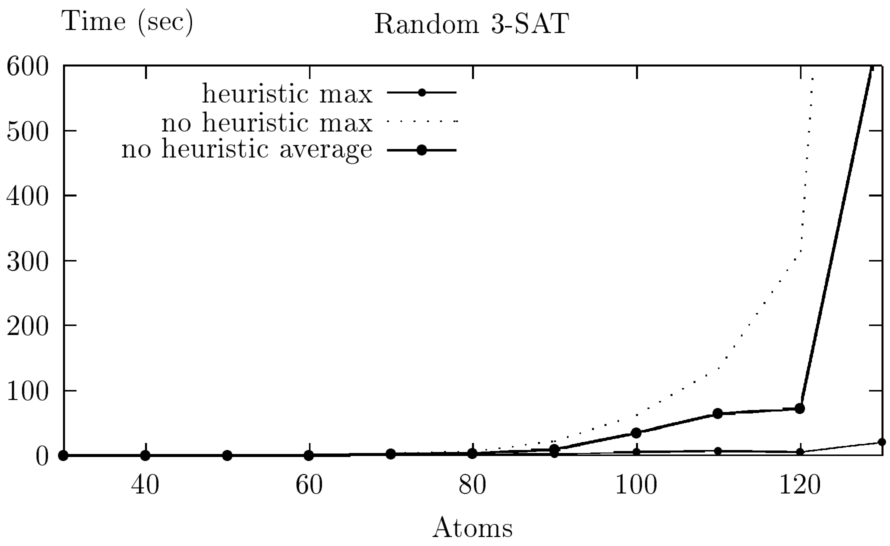

Figure 14: Heuristic versus no heuristic

10 Comparison with other Algorithms

We make a comparison of the **smodels** algorithm and some other algorithms for computing the stable model semantics. In order to facilitate the comparison we must first examine the well-founded semantics [VGRS91], which we do in Subsection 10.1. In Subsection 10.2 we examine the branch and bound algorithm of [SNV95], in Subsection 10.3 the SLG system of [CW96], and in Subsection 10.4 the mixed integer programming methods of [BNNS94]. Finally, we present the comparison in Subsection 10.5.

10.1 The Well-founded Semantics

Since the definition of the well-founded semantics [VGRS91] is somewhat complicated, we make use of an equivalent definition that suits our needs better. Let

$$\Gamma_P\,(A) = \mathrm{Dcl}\big(P, \mathrm{not}\,(\overline{A})\big)$$

for any ground program P and any set of atoms A. Then, the operator $\Gamma_P\,(A)$ is anti-monotonic and we can define the well-founded semantics with the help of the least and greatest fixed points (lfp and gfp) of the monotonic operator

$$\Gamma_P^2\,(A) = \Gamma_P\big(\Gamma_P(A)\big).$$

According to [BS93], the well-founded model B of a program P is defined as follows:

$$\begin{aligned} B^+ &= \mathrm{lfp}\,(\Gamma_P^2) \\ B^- &= \mathrm{not}\big(\overline{\mathrm{gfp}\,(\Gamma_P^2)}\big). \end{aligned}$$

Observe that if $B = \textbf{expand}\,(P, \emptyset)$, then

$$\mathrm{Dcl}\big(P_B, \mathrm{not}\,(\mathrm{NAnt}\,(P))\big) \subseteq \mathrm{Dcl}\big(P, \mathrm{not}\,(\overline{B^+})\big)$$

and

$$\mathrm{not}\big(\overline{\mathrm{Dcl}\,(P_B, \mathrm{not}\,(\mathrm{NAnt}\,(P)))}\big) \subseteq B^-,$$

which implies

$$\mathrm{not}\big(\overline{\mathrm{Dcl}\,(P, \mathrm{not}\,(\overline{B^+}))}\big) \subseteq B^-,$$

which in turn implies

$$\Gamma_P^2\bigl(B^+\bigr) \subseteq \mathrm{Dcl}\,(P, B^-) \subseteq B^+.$$

Thus, B^+ is a fixed point of Γ_P^2. Moreover, since

$$\Gamma_P^2\bigl(\Gamma_P\,(B^+)\bigr) = \Gamma_P\bigl(\Gamma_P^2\,(B^+)\bigr) = \Gamma_P\,(B^+),$$

also $\Gamma_P\,(B^+)$ is a fixed point,

$$\Gamma_P\,(B^+) \subseteq \mathrm{gfp}\,(\Gamma_P^2),$$

and consequently

$$\mathrm{not}\bigl(\overline{\mathrm{gfp}\,(\Gamma_P^2)}\bigr) \subseteq \mathrm{not}\bigl(\overline{\Gamma_P\,(B^+)}\bigr) \subseteq B^-.$$

Hence, the well-founded model of P is a subset of B. Remark that as the function **expand** can return a contradictory set of literals, the function **expand** provides a stronger pruning technique than one gets using the well-founded semantics to reduce the program. For example, consider the program

$$P = \{a \leftarrow \mathrm{not}\,(b),\ b \leftarrow \mathrm{not}\,(a),\ a \leftarrow b\}$$

and make the assumption that we are looking for a stable model containing the atom b. A typical reduction of P would now delete the first two rules from the program and the atom b from the body of the last rule, resulting in the program

$$a \leftarrow .$$

An application the well-founded semantics to this program produces the conclusion a, but not the fact that there are no stable models containing the atom b. The function call **expand** $(P, \{b\})$ on the other hand returns a set containing both b and $\mathrm{not}\,(b)$, thereby explicitly showing that there are no models that include b.

10.2 Branch and Bound

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

Starting with the first simplified program, the algorithm computes the stable models by generating smaller and smaller programs from the programs it has already generated. Two new programs are generated from one program by assuming that an atom, whose truth value is unknown according to the well-founded semantics, belongs or do not belong to the stable model under construction. The search space is pruned 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 quite a bit, 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 or partially constructed models is large.

10.3 The SLG System

The SLG system [CW96], which supports goal-oriented query evaluation of logic programs under the well-founded semantics, simplifies a logic program during the query evaluation producing a residual program. Provided that all negative literals in the residual program are ground, the system can then compute the stable models using an assume-and-reduce algorithm.

The assume-and-reduce algorithm constructs the stable models of a logic program in a similar fashion to the branch and bound algorithm. However, 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 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. But, the algorithm can with the help of backward propagation in some specialized circumstances derive whether an atom belongs to the stable model or not. This slightly improves the pruning technique.

10.4 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, which is then used to compute all minimal models of the logic program. The models are subsequently tested by another integer linear program in order to exclude the models that are not stable. As the number of minimal models can be very large compared to the number of stable models, we conclude that this approach is very inefficient.

10.5 Comparison

We compare the branch and bound algorithm, the SLG system, and the mixed integer programming methods with the **smodels** algorithm. 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 **smodels** algorithm nor with the SLG system. Both the branch and bound algorithm and the SLG system prune the search space less effectively than **smodels**. The main difference between **smodels** without lookahead and the branch and bound algorithm and the SLG system is that **smodels** does not differentiate between assumed and derived literals. Hence, **smodels** automatically avoids exploring the parts of the search space that the branch and bound algorithm avoids by storing all partially constructed stable models and that the SLG system does not avoid.

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

$$\{a_1 \leftarrow \text{not}\,(b_1)\ \ldots\ ,\ a_n \leftarrow \text{not}\,(b_n)\}$$

has one stable model but 2^n minimal models.

We conclude that even without lookahead the **smodels** algorithm is significantly better than the branch and bound algorithm, the mixed integer programming methods, and the SLG system. The addition of lookahead makes, as was seen in the previous section, the difference even more evident.

11 Conclusions

We have developed an algorithm for computing stable models of ground logic programs. A pruning technique that encompasses backward propagation of truth values has been introduced, and then been further improved by the addition of lookahead. This leads to an algorithm that never chooses an atom that is the direct cause of a conflict. Hence, any conflict is caused by the chronologically next to last choice, and any form of backjumping is in many cases neither necessary nor possible. Furthermore, we have introduced backjumping in order to handle the remaining cases.

The nondeterministic choices in the algorithm are handled by a strong heuristic, which we have derived from one basic assumption, namely the assumption that the search space can be approximated by a complete binary tree. The heuristic greedily tries to minimize the size of this tree by making choices that prune the branches of the tree as much as possible.

The pruning technique is based on two kinds of deductive closures. We have demonstrated a linear-time implementation method for computing these closures, and we have shown how the method simplifies backtracking. Moreover, we have presented a simple upper bound on the complexity of the algorithm for finding a stable model or for determining that there are no stable models, and we have with the help of extremal logic programs established a high lower bound on all upper bounds.

We have examined logic programs with variables, and a procedure for grounding function-free range restricted programs that avoids generating the whole ground instantiation has been presented. In addition, an implementation of the algorithm has been compared with a state-of-the-art satisfiability checker on hard 3-SAT problems. The stable model algorithm prunes the search space more than the satisfiability checker, but due to a higher overhead the algorithm is about 25 times slower than the checker on the test problems.

Finally, a comparison between the algorithm and some other procedures for computing the stable model semantics has been made. The conclusion being that our algorithm is unequivocally superior to the other procedures.

References

[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.

[BS93] C.R. Baral and V.S. Subrahmanian. Dualities between alternative semantics for logic programming and nonmonotonic reasoning. *Journal of Automated Reasoning*, 10:399–420, 1993.

[CA96] J.M. Crawford and L.D. Auton. Experimental results on the crossover point in random 3-SAT. *Artificial Intelligence*, 81(1):31–57, 1996.

[CMT96] P. Cholewiński, V.W. Marek, and M. Truszczyński. Default reasoning system DeReS. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 518–528, Cambridge, MA, USA, November 1996. Morgan Kaufmann Publishers.

[Cra96] J.D. Crawford. The tableau algorithm, 1996. Available at http://www.cirl.uoregon.edu/crawford/.

[CT96] P. Cholewiński and M. Truszczyński. Extremal problems in logic programming and stable model computation. In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 408–422, Bonn, Germany, September 1996.

[CW96] W. Chen and D.S. Warren. Computation of stable models and its integration with logical query processing. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):742–757, October 1996.

[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.

[Dim96] Y. Dimopoulos. On computing logic programs. *Journal of Automated Reasoning*, 17:259–289, 1996.

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

[DNK97] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in non-monotonic logic programs. In *Proceedings of the Fourth European Conference on Planning*, Toulouse, France, September 1997. Springer-Verlag. To appear.

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

[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.

[Fit85] M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[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.

[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.

[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.

[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.

[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.

[NS95] I. Niemelä and P. Simons. Evaluating an algorithm for default reasoning. In *Working Notes of the IJCAI'95 Workshop on Applications and Implementations of Nonmonotonic Reasoning Systems*, pages 66–72, Montreal, Canada, August 1995.

[NS96a] I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. Forschungsbericht 7–96, Universität Koblenz-Landau, 1996.

[NS96b] I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 289–303, Bonn, Germany, September 1996. 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.

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

[Sel94] B. Selman. Random k-SAT generator, 1994. Available at ftp://ftp.research.att.com/dist/ai/makewff.sh.Z.

[Sim95] P. Simons. Efficient implementation of the stable model semantics for normal logic programs. Research Report 35, Helsinki University of Technology, Helsinki, Finland, September 1995.

[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.

[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.