Efficient Implementation of the Well-founded and Stable Model Semantics

Ilkka Niemelä
Department of Computer Science
University of Koblenz
Rheinau 1, D-56075 Koblenz, Germany
ini@informatik.uni-koblenz.de

Patrik Simons
Department of Computer Science
Helsinki University of Technology
Otakaari 1, FIN-02150 Espoo, Finland
Patrik.Simons@hut.fi

Abstract

An implementation of the well-founded and stable model semantics for range-restricted function-free normal programs is presented. It includes two modules: an algorithm for implementing the two semantics for ground programs and an algorithm for computing a grounded version of a range-restricted function-free normal program. The latter algorithm does not produce the whole set of ground instances of the program but a subset which is sufficient in the sense that no stable models are lost. The implementation of the stable model semantics for ground programs is based on bottom-up backtracking search. It works in linear space and employs a powerful pruning method based on an approximation technique for stable models which is closely related to the well-founded semantics. The implementation includes an efficient algorithm for computing the well-founded model of a ground program. The implementation has been tested extensively and compared with a state of the art implementation of the stable model semantics, the SLG system. In tests involving ground programs it clearly outperforms SLG.

1 Introduction

In recent years the semantical underpinnings of logic programs have been studied intensively. This work has led to the development of new declarative semantics for logic programs with the well-founded semantics [13] and the stable model semantics [7] as the two most influential representatives. However, there is a limited amount of work where actual implementations of these semantics have been developed.

In this paper we aim to develop an implementation of the stable model semantics for normal logic programs. We consider a decidable fragment consisting of range-restricted function-free programs. Our implementation includes two modules: (i) an algorithm for implementing the stable model semantics for ground programs and (ii) an algorithm for computing a grounded version of a range-restricted function-free normal program. The latter algorithm does not generate the whole set of ground instances of the program but a subset which is sufficient to ensure that no stable models are lost.

The algorithm for implementing the stable model semantics for ground programs is developed using an approach introduced for implementing default logic [9]. This approach provides a framework for developing bottom-up methods for computing extensions of default theories. In this paper we use the framework to devise an efficient implementation for the stable model semantics that possesses some interesting properties.

- The implementation is able to solve a range of decision problems related to the stable model semantics. It can decide whether a program has a stable model, compute all stable models of a program and handle two basic query-answering tasks: decide whether a given formula is satisfied in some or all of the stable models of a program.
- The implementation is based on bottom-up backtracking search and a powerful pruning method employing an approximation technique for stable models which is closely related to the well-founded semantics.
- The implementation includes an efficient (quadratic time) algorithm for computing the well-founded model of a ground program.
- The implementation runs in linear space.

The new implementation opens up the possibility to apply the stable model semantics in areas where resulting programs are highly non-stratified and possess a potentially large number of stable models. Because of the polynomial space complexity our implementation is able to solve these kinds of hard instances provided that adequate amount of running time is allocated. We have tested the implementation extensively. In order to obtain challenging test cases we have used, e.g., combinatorial graph problems as a test domain. This domain has also been used in TheoryBase [5] which is a system for generating test cases for nonmonotonic reasoning.

There exist several approaches to computing stable models (see, e.g., [1]). Our method compares favorably with recent implementations [2, 4, 11]. The methods described in [2, 11] cannot handle programs with a large number of stable models because of exponential worst-case space requirements and when considering programs with a few stable models, our implementation seems to outperform them.

The SLG system [3, 4] appears to be the best currently available implementation of the stable model semantics. SLG also exploits the well-founded semantics: first a residual ground program is produced from a program and a given query using a top-down query evaluation method for the well-founded semantics and then an assume-and-reduce algorithm is employed to compute stable models of the residual program. However, SLG differs in several respects from our approach. The residual program does not preserve stable model semantics, i.e., the stable models of the residual program do not coincide with the stable models of the original program [4]. On the other hand, the grounded version computed by our implementation preserves the stable model semantics. When searching for the stable models of a ground program, our algorithm seems to prune the search space much more efficiently than the assume-and-reduce algorithm proposed in [4]. The experiments we have run clearly confirm this and show that for ground programs the performance of our implementation is superior to that of SLG.

The paper is organized as follows. In the next section we briefly introduce the stable model semantics and the well-founded semantics. In Section 3 the algorithm for implementing the stable model semantics of ground programs is developed. Section 4 presents a method to compute a grounded version of a range-restricted function-free program. The test results and comparisons are presented in Section 5.

2 Preliminaries

We review the stable model and the well-founded semantics and present a characterization of stable models which provides the basis for the novel algorithm. We consider *normal* logic programs consisting of rules of the form

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

where a_i, b_j and c are atoms. For a ground program P, the *stable model semantics* [7] can be defined using an operator Γ_P as follows. The *reduct* P^Δ with respect to a set of atoms Δ is a (Horn clause) program obtained from P by deleting (i) each rule that has $\mathrm{not}(b)$ in its body with $b \in \Delta$ and (ii) all $\mathrm{not}(l)$ in the bodies of the remaining rules. Then $\Gamma_P(\Delta)$ is defined as the deductive closure of P^Δ where the rules in P^Δ are seen as inference rules. Now a set of ground atoms Δ is a *stable model* of P iff $\Delta = \Gamma_P(\Delta)$. For a non-ground program P, the stable models are those of the ground program obtained by taking all ground instances of the rules in P.

For describing our approach to computing stable models we generalize the operator Γ_P used above and define the *deductive closure* $\mathrm{Dcl}(P, L)$ for a ground program P and a set of ground literals L. Literals are atoms and not-atoms (formulae of the form $\mathrm{not}(a)$). For a set of literals L, L^+ (L^-) denotes the set of atoms (not-atoms) in L. Now $\mathrm{Dcl}(P, L)$ is the smallest set of atoms containing L^+ which is closed under the inference rules

$$\begin{aligned} \mathrm{R}(P, L) \quad = \quad \{\, c \leftarrow a_1, \ldots, a_n \mid \quad & c \leftarrow a_1, \ldots, a_n, \mathrm{not}(b_1), \ldots, \mathrm{not}(b_m) \in P, \\ & \{\mathrm{not}(b_1), \ldots, \mathrm{not}(b_m)\} \subseteq L^- \}. \end{aligned}$$

For instance, if $P = \{\ q \leftarrow \mathrm{not}(r);\quad q \leftarrow \mathrm{not}(p);\quad r \leftarrow q\}$, then $\mathrm{R}(P, \{p, \mathrm{not}(p)\}) = \{\ q \leftarrow;\quad r \leftarrow q\}$ and $\mathrm{Dcl}(P, \{p, \mathrm{not}(p)\}) = \{p, q, r\}$. Using the new deductive closure, the operator Γ_P can be expressed easily: $\Gamma_P(\Delta) = \mathrm{Dcl}(P, \mathrm{not}(\overline{\Delta}))$ where $\mathrm{not}(\overline{\Delta}) = \{\mathrm{not}(\phi) \mid \phi \in \overline{\Delta}\}$ and $\overline{\Delta}$ is the complement of Δ, i.e., the atoms not in Δ. Hence, Δ is a *stable model* of a ground program P iff $\Delta = \mathrm{Dcl}(P, \mathrm{not}(\overline{\Delta}))$.

Our approach to computing stable models is based on a novel compact characterization of stable models originating from a characterization of extensions in default logic [9]. The characterizing sets for stable models are called *full sets*. They are sets of not-atoms built from atoms in $\mathrm{NAnt}(P)$, the *negative antecedents* in P, i.e., atoms b such that $\mathrm{not}(b)$ appears in P. A key observation is that for the deductive closure of P, a premise $\mathrm{not}(b)$ plays a role only when $b \in \mathrm{NAnt}(P)$, i.e., $\mathrm{Dcl}(P, \mathrm{not}(\overline{\Delta})) = \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - \Delta))$. Thus, a set of atoms Δ is a stable model of P iff $\Delta = \mathrm{Dcl}(P, \mathrm{not}(\mathrm{NAnt}(P) - \Delta))$. The following characterization theorem provides the basis for our computational treatment of stable models. (The proofs can be found in a longer version of the paper [10]).

Definition 2.1 *A set Λ of* not*-atoms is called P-full iff for all $\phi \in \mathrm{NAnt}(P)$, $\mathrm{not}(\phi) \in \Lambda$ iff $\phi \notin \mathrm{Dcl}(P, \Lambda)$.*

Theorem 2.2 *Let P be a ground program and Λ a set of* not*-atoms.*

(i) If Λ is P-full, then $\mathrm{Dcl}(P, \Lambda)$ is a stable model of P.

(ii) If Δ is a stable model of P, then $\Lambda = \mathrm{not}(\mathrm{NAnt}(P) - \Delta)$ is a P-full set such that $\Delta = \mathrm{Dcl}(P, \Lambda)$.

The well-founded semantics [13] can be defined using the fixed points of the operator Γ_P^2 ($\Gamma_P^2(\Delta) = \Gamma_P(\Gamma_P(\Delta))$) [12]. The operator Γ_P is anti-monotonic and, hence, Γ_P^2 is monotonic. Thus it has a least and a greatest fixed point which we denote by $\mathrm{lfp}(\Gamma_P^2)$ and $\mathrm{gfp}(\Gamma_P^2)$, respectively. The *well-founded model* of a ground program P is the set of ground literals

$$\mathrm{lfp}(\Gamma_P^2) \cup \mathrm{not}(\overline{\mathrm{gfp}(\Gamma_P^2)}).$$

3 An Algorithm for Computing Stable Models

In this section we develop an algorithm for computing the stable model semantics of ground programs using a framework based on an approach introduced for default logic [9]. The framework consists of a basic algorithm where two key functions are not given directly but are specified by stating conditions that these functions have to satisfy in order to guarantee the soundness and completeness of the algorithm. We first introduce the basic algorithm and then develop optimized implementations of the two functions.

The basic algorithm is actually a skeleton for a range of decision procedures for solving reasoning problems concerning the stable model semantics. By changing a certain test function, the same algorithm can be adapted to deciding whether a program has a stable model, to computing all stable models of a program and to handling two basic query-answering tasks: decide whether a given formula is satisfied in some or all of the stable models of a program.

When describing the algorithm we use the following two concepts. We say that a set of atoms S *agrees* with a set of literals Λ (i) if $\phi \in \Lambda^+$ implies $\phi \in S$ and (ii) $\mathrm{not}(\phi) \in \Lambda^-$ implies $\phi \notin S$. For example, the set $\{a, b\}$ agrees with $\{a, \mathrm{not}(c)\}$ but not with $\{a, \mathrm{not}(b)\}$. A set of literals Λ *covers* an atom ϕ if either $\phi \in \Lambda$ or $\mathrm{not}(\phi) \in \Lambda$. For example, the set $\{\mathrm{not}(a), b\}$ covers a. A set of atoms S is covered by Λ if each atom in S is covered by Λ.

The basic algorithm is presented in Figure 1 as a function **sm**. It takes as input a set of rules P, a set B which represents the common part of the stable models to be considered and a formula ϕ which is just passed as an argument to the function **test**. The set B represents a partially constructed full set in the following way: B^- are those not-atoms already included in the full set and an atom $\phi \in B$ indicates that $\mathrm{not}(\phi)$ cannot be included in the full set. The function **sm** performs *focused model search*: it returns true iff there is a stable model Δ of P *agreeing with* B such that $\mathbf{test}(\Delta, \phi)$ returns true. This is accomplished by constructing P-full sets agreeing with B until a full set Λ is found such that for the corresponding stable model $\Delta = \mathrm{Dcl}(P, \Lambda)$, $\mathbf{test}(\Delta, \phi)$ returns true.

The function **sm** employs three functions: **expand** (for expanding B), **conflict** (for detecting conflicts), and **test** (for testing stable models). By changing the function **test** the various decision procedures are obtained. The first two functions provide the key targets for optimizations in the algorithm. In order to facilitate the development of optimizations we present requirements (E1,E2,C1,C2) for the implementations of the two functions which guarantee the soundness and completeness of the decision procedures.

E1: $B \subseteq \mathbf{expand}(P, B)$.

E2: Every stable model Δ of P agreeing with B agrees with $\mathbf{expand}(P, B)$.

C1: If B covers $\mathrm{NAnt}(P)$ and for some $\mathrm{not}(\chi) \in B$, $\chi \in \mathrm{Dcl}(P, B^-)$ or for an atom $\chi \in B$, $\chi \notin \mathrm{Dcl}(P, B^-)$, then $\mathbf{conflict}(P, B)$ returns true.

C2: If $\mathbf{conflict}(P, B)$ returns true, then there exists no stable model Δ of P agreeing with B.

The function **sm** starts by cautiously expanding the set B. Condition (E1) ensures that B is really extended and (E2) guarantees that no stable models are lost. Then a conflict test is performed. Condition (C1) ensures that if all negative antecedents are already covered and there is an apparent conflict, it is detected and (C2) guarantees that if a conflict is reported, there are no stable models agreeing with B' left. If there are no conflicts and B' covers every negative antecedent, B'^- is a full set corresponding to the stable

```
function sm(P, B, φ)
B' := expand(P, B);
if conflict(P, B') returns true then return false
else if NAnt(P) is covered by B' then return test(Dcl(P, B'), φ)
else
    take some χ ∈ NAnt(P) not covered by B';
    if sm(P, B' ∪ {not(χ)}, φ) returns true then return true
    else return sm(P, B' ∪ {χ}, φ)
    end if
end if
```

Figure 1: The Skeleton for the Decision Procedures for Stable Models

model $\mathrm{Dcl}(P, B')$ and **sm** returns what $\mathbf{test}(\mathrm{Dcl}(P, B'), \phi)$ returns. If there is an atom $\chi \in \mathrm{NAnt}(P)$ not covered by B', then either $\mathrm{not}(\chi)$ belongs to the full set to be constructed or not. The two alternatives are handled by backtracking. First B' is extended by $\mathrm{not}(\chi)$ by calling $\mathbf{sm}(P, B' \cup \{\mathrm{not}(\chi)\}, \phi)$. If the function returns false, $\mathrm{not}(\chi)$ cannot belong to the full set and **sm** returns what $\mathbf{sm}(P, B' \cup \{\chi\}, \phi)$ returns. It can be shown that **sm** is sound and complete.

Theorem 3.1 (Soundness and completeness) *Let P be a set of rules and B a set of literals. Then $\mathbf{sm}(P, B, \phi)$ returns true iff there exists a stable model Δ of P agreeing with B such that $\mathbf{test}(\Delta, \phi)$ returns true.*

The theorem implies that decision procedures for the different reasoning tasks can be obtained by (i) taking the empty set as the common part of the stable models to be constructed and (ii) choosing an appropriate **test** function as shown in the following corollary.

Corollary 3.2 *Let P be a set of rules and ϕ a formula.*

1. Let $\mathbf{test}(\Delta, \phi)$ return true for all Δ and ϕ. Then $\mathbf{sm}(P, \emptyset, \phi)$ returns true iff P has a stable model.

2. Let $\mathbf{test}(\Delta, \phi)$ return true iff ϕ is satisfied by Δ. Then $\mathbf{sm}(P, \emptyset, \phi)$ returns true iff there exists a stable model of P satisfying ϕ.

3. Let $\mathbf{test}(\Delta, \phi)$ return true iff ϕ is not satisfied by Δ. Then $\mathbf{sm}(P, \emptyset, \phi)$ returns true iff there exists a stable model of P not satisfying ϕ (and, in other words, $\mathbf{sm}(P, \emptyset, \phi)$ returns false iff ϕ is satisfied by every stable model of P).

It is easy to devise implementations of **expand** and **conflict** satisfying conditions E1–E2, C1–C2, e.g., the following would do: (i) $\mathbf{expand}(P, B)$ returns $B \cup \{\chi \in \mathrm{NAnt}(P) \mid \chi \text{ not covered by } B, \chi \in \mathrm{Dcl}(P, B^-)\}$ and (ii) $\mathbf{conflict}(P, B)$ returns true iff B covers $\mathrm{NAnt}(P)$ and there is an atom $\chi \in B$ with $\chi \notin \mathrm{Dcl}(P, B^-)$ or a literal $\mathrm{not}(\chi) \in B$ with $\chi \in \mathrm{Dcl}(P, B^-)$.

Example 1 The basic algorithm is illustrated by using it to decide whether

$$P = \{\ a \leftarrow c, \mathrm{not}(b);\ b \leftarrow c, \mathrm{not}(a);\ c \leftarrow \mathrm{not}(a)\} \tag{2}$$

has a stable model satisfying $a \wedge c$. Hence, we let $\mathbf{test}(\Delta, \phi)$ return true iff ϕ is satisfied by Δ and consider the execution of $\mathbf{sm}(P, \emptyset, a \wedge c)$.

First $\mathbf{expand}(P, \emptyset)$ returns $\emptyset$ and $\mathbf{conflict}(P, \emptyset)$ returns false. Since $\mathrm{NAnt}(P) = \{a, b\}$ is not covered by $\emptyset$, we must choose either a or b in order to continue. Let us take a. Then $\mathbf{sm}(P, \{\mathrm{not}(a)\}, a \wedge c)$ is executed. Now $\mathbf{expand}(P, \{\mathrm{not}(a)\})$ returns $\{\mathrm{not}(a), b\}$ and $\mathbf{conflict}(P, \{\mathrm{not}(a), b\})$ returns false. Because $\mathrm{NAnt}(P)$ is covered by $\{\mathrm{not}(a), b\}$, we have found a stable model $\mathrm{Dcl}(P, \{\mathrm{not}(a), b\}) = \{b, c\}$ of P but $\mathbf{test}(\{b, c\}, a \wedge c)$ returns false. Hence, $\mathbf{sm}(P, \{a\}, a \wedge c)$ is executed. This leads to a call to $\mathbf{sm}(P, \{a, \mathrm{not}(b)\}, a \wedge c)$ which return false as does $\mathbf{sm}(P, \{a, b\}, a \wedge c)$ and finally also $\mathbf{sm}(P, \emptyset, a \wedge c)$. Thus, no stable model of P satisfies $a \wedge c$. ■

The given implementations already provide us with quite an efficient decision method for the stable model semantics. However, we aim to develop further optimized implementations of the two functions where the search space of the remaining stable models is more effectively pruned. The idea is to strengthen **expand** so that it extends the common part of the models as far as possible without losing any models. Hence, the number of choice points for the backtracking search is reduced. On the other hand, detecting conflicts as early as possible prunes the search space effectively. There is an interesting *trade-off*: the work spent in **expand** and **conflict** saves work in the remaining backtracking search. Our solution to the problem of dividing the work between the two functions and the search is that we spend polynomial time (actually quadratic time) in the two functions for each recursive call and the rest of the work is done by the backtracking search.

We start by optimizing the function **expand**. First, we observe that we can strengthen the deductive closure used in the initial version above without compromising the property (E2) by employing a *Fitting operator* F_P. We define $\mathrm{F}_P(B)$ to be the smallest set S containing B such that (i) if $c \leftarrow a_1, \ldots, a_n, \mathrm{not}(b_1), \ldots, \mathrm{not}(b_m) \in P$ and $\{a_1, \ldots, a_n, \mathrm{not}(b_1), \ldots, \mathrm{not}(b_m)\} \subseteq S$, then $c \in S$ and (ii) if for each rule with c as the head in P, some atom a is in the body but $\mathrm{not}(a) \in S$ or some $\mathrm{not}(b)$ is in the body but $b \in S$, then $\mathrm{not}(c) \in S$. Now F_P is monotonic for any program P and we define $\mathrm{Dcl}^{\mathrm{F}}(P, B)$ to be the least fixed point of the series $S_0 = B$ and for all $i > 0$, $S_i = \mathrm{F}_P(S_{i-1})$. Notice that $\mathrm{Dcl}^{\mathrm{F}}(P, B)$ is a superset of $\mathrm{Dcl}(P, B)$ but often considerably larger. For example, let P be as in (2) and $B = \{a\}$. Then $\mathrm{Dcl}(P, B) = \{\}$ but $\mathrm{Dcl}^{\mathrm{F}}(P, B) = \{a, \mathrm{not}(c), \mathrm{not}(b), \mathrm{not}(a)\}$.

Second, we notice that we can extend B by not-atoms and still satisfy condition (E2) by borrowing an idea from the well-founded computation: we can add $\mathrm{not}(b)$ if $b \notin \Gamma_P(B^+) = \mathrm{Dcl}(P^{B^+}, \emptyset)$. In fact, we can further optimize this by exploiting the not-atoms in B. For this we *generalize the reduct* P^S of a program P to the case where S can contain also not-atoms and use P^B instead of P^{B^+}. Hence, we define P^S, a reduct of P with respect to a set of *literals* S, as a set of rules obtained from P by deleting (i) each rule that has $\mathrm{not}(b)$ in its body but $b \in S$ or a in the body but $\mathrm{not}(a)$ in S and (ii)

function expand(P, B)
repeat
  $B' := B$;
  $B := B \cup \{\phi \in \mathrm{Dcl}^{\mathrm{F}}(P, B) \mid$
    the atom in ϕ is in $\mathrm{NAnt}(P)$
    but not covered by $B\}$;
  $B := B \cup \{\mathrm{not}(\phi) \mid \phi \in \mathrm{NAnt}(P)$,
    ϕ is not covered by B
    and $\phi \notin \mathrm{Dcl}(P^B, \emptyset)\}$;
  Mark all new literals added to B
  (i.e. in $B - B'$)
until $B = B'$;
return B

function conflict(P, B)
if for some unmarked $\mathrm{not}(\phi) \in B$,
    $\phi \in \mathrm{Dcl}^{\mathrm{F}}(P, B)$ or
    for some unmarked atom $\phi \in B$,
    $\phi \notin \mathrm{Dcl}(P^B, \emptyset)$ **then**
  return true
else
  return false
end if

Figure 2: Functions **expand** and **conflict**

all $\mathrm{not}(l)$ in the bodies of the remaining rules. For example, let P be as in (2) and $B = \{\mathrm{not}(c)\}$. Then $P^B = \{\, c \leftarrow \}$ but $P^{B^+} = \{\, a \leftarrow c;\ b \leftarrow c;\ c \leftarrow \}$.

Similar considerations are applicable when implementing **conflict**. The resulting implementations of **expand** and **conflict** are presented in Figure 2. It is straightforward to establish that the implementations satisfy the conditions (E1, E2, C1, C2) [10].

The optimized implementation of **expand** can be seen as a generalization of well-founded model computation: when starting from an empty model, **expand** computes the well-founded model as shown in the next theorem. When the initial model is not empty, **expand** computes a well-founded type approximation for all the stable models agreeing with the initial model.

Theorem 3.3 *Let P be a ground program. If $B = \mathbf{expand}(P, \emptyset)$, then $I = \mathrm{Dcl}(P, B) \cup \mathrm{not}(\overline{\mathrm{Dcl}(P^B, \emptyset)})$ is the well-founded model of P.*

The algorithm in Figure 1 together with the implementations of **expand** and **conflict** in Figure 2 form the basis of our implementation. In order to obtain an efficient implementation, two important issues must be addressed:

- the implementations of $\mathrm{Dcl}^{\mathrm{F}}(P, B)$ and $\mathrm{Dcl}(P^B, \emptyset)$ employed in the functions **expand** and **conflict** in Figure 2 and
- the search heuristics.

The deductive closures can be computed in linear time for a ground program using a modified version of the linear time algorithm of Dowling and Gallier [6] for computing the deductive closure of a set of Horn clauses (for details, see [10]). This implies that **expand** can be implemented in quadratic time yielding a quadratic time implementation of the well-founded semantics for ground programs. Our implementation of **expand** frequently seems to have a linear time performance. This is because the Fitting operator often yields (for programs without positive loops) the well-founded model in linear time and then the other closure is not needed at all. This positive effect of the Fitting operator has also been pointed out, e.g., in [11].

By using an appropriate *search heuristic* the performance of the algorithm **sm** in Figure 1 can sometimes be substantially improved. The heuristic can be incorporated in the phase where there are atoms in $\mathrm{NAnt}(P)$ not yet covered by B' and one of them has to be chosen in order to extend the full set under construction. Here our idea is to devise an *application independent* basic heuristic which would lead to good performance across a range of applications and which would offer a starting point for developing more refined heuristics for particular application areas.

We employ a heuristic which 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. Our heuristics selects a not-atom from the body of an applicable rule with the least number of not-atoms not in B^- in its body. A rule is applicable if it might change the set B, i.e., the head of an applicable rule is not in $\mathrm{Dcl}^{\mathrm{F}}(P, B)$, all of the positive literals in the body of the rule are in the closure, and none of the negative literals cover any atom in B^+. If there are no applicable rules left, the atoms in $\mathrm{NAnt}(P)$ not covered by B cannot affect the deductive closures. Therefore, they are immediately added as not-atoms to B^-. Computing this heuristic is linear in the number of rules in the program (for details, see [10]).

4 Handling Rules with Variables

In this section we develop a method to compute a grounded version P' of a range-restricted function-free program P such that P and P' have exactly the same stable models agreeing with a given set of literals B, but P' is often considerably smaller than the whole set of ground instances of P. We first define the set P' and then present an algorithm for computing it directly from P without considering all of its ground instances. To define P' we let

$$A' = \mathrm{Dcl}(P_{\mathrm{g}}, A \cup \mathrm{not}(\mathrm{NAnt}(P_{\mathrm{g}}) - A))$$

where P_{g} is the set of all ground instances of the rules in P, $A = B \cup B_n \cup B_p$, $B_n = \{\mathrm{not}(c) \mid c \in \mathrm{NAnt}(P_{\mathrm{g}}) - B$ and there is no rule in P having the same predicate as c in the head $\}$, and $B_p = \{c \mid$ there is a ground rule in P with c as its head and all literals in the body in $B_n \cup B\}$. Then let

$$\begin{aligned} P' \quad = \quad & \{\, c \leftarrow a_1, \ldots, a_n, \mathrm{not}(b_1), \ldots, \mathrm{not}(b_m) \in P_{\mathrm{g}} \mid \\ & \{a_1, \ldots, a_n\} \subseteq A', \{b_1, \ldots, b_m\} \cap A = \emptyset\}. \end{aligned} \tag{3}$$

Figure 3 presents an algorithm that computes P' directly from P and A without grounding P first when P is a range-restricted function-free program. A program is *range-restricted* if for each rule it holds that if a variable appears

```
function ground(P, A)
P₂ := {r ∈ P | the positive literals of the body of r is in
    A and the negative literals are in A⁻ or not in not(A⁺)}
P := P − P₂; U := ∅;
N := A⁺ ∪ {a | there exists a rule r ∈ P₂ 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 do
        for every rule r′ resulting from applying a substitution
            to 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
                    P₂ := P₂ ∪ {r′}
                    if the head of r′ is not in U then
                        N := N ∪ {the head of r′}
                    end if
                else
                    P := P ∪ {r′}
                end if
            end if
        end for
    end for
end while
return P₂
```

Figure 3: A Procedure for Grounding Range Restricted Programs

in the head or in a not-atom in the body, it also appears in a positive literal in the body. When using P' instead of the whole P_{g} no stable models are lost. Notice that when invoking the algorithm to compute the stable models for the grounded version P' we can give as the initial part of the stable models not only B but $A \cup \mathrm{not}(\mathrm{NAnt}(P') - A')$.

Proposition 4.1 *Δ is a stable model of P_{g} agreeing with B iff Δ is a stable model of P' agreeing with $A \cup \mathrm{not}(\mathrm{NAnt}(P') - A')$.*

Example 2 Consider a range-restricted program

$$\begin{aligned} P \;=\; & \{\, q(a) \leftarrow;\; s(b) \leftarrow;\; p(a) \leftarrow \mathrm{not}(r(a)); \\ & t(X) \leftarrow q(X), \mathrm{not}(p(X));\; q(X) \leftarrow s(X), \mathrm{not}(t(X))\}. \end{aligned}$$

Let $B = \{t(b)\}$. Now the grounding is $P' = \{\, q(a) \leftarrow;\; s(b) \leftarrow;\; p(a) \leftarrow \mathrm{not}(r(a))\}$. Observe that, e.g., $t(a) \leftarrow q(a), \mathrm{not}(p(a))$ is not included as $p(a) \in A$ and $t(b) \leftarrow q(b), \mathrm{not}(p(b))$ is not included since $q(b) \notin A'$. ■

5 Experimental Results

We have implemented the **sm** algorithm in C++ and the resulting implementation of the stable model and well-founded semantics can be obtained from `ftp://saturn.hut.fi/pub/smodels/`. Details of the implementation specific choices we made to ensure an efficient implementation of the **sm** algorithm can be found in a longer version of the paper [10].

We have tested our implementation extensively. All the test cases can be obtained from `ftp://saturn.hut.fi/pub/smodels/tests`. In order to get challenging test cases we have employed combinatorial graph problems as a test domain. Test cases from the literature and from the domain of diagnostics have also been used. In this section we report some of the test results and compare the performance of our implementation with the SLG system [4, 3] which is a state of the art implementation of the stable model semantics. Further test results are reported in a longer version of the paper [10]. The two implementations are compared by using them to search for 3-colorings and Hamiltonian circuits in different graphs. In addition, two benchmark programs in [4] and the win-move example [11] are used. Three types of graphs are used in the tests. A special type of planar graph, created by the *plane* function found in the Stanford GraphBase [8], a random graph created by the *random_graph* function, and a 'ladder'-graph, see Figure 4. 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 ground programs such that the stable models of the programs provide the solutions to the graph problems.

The 3-coloring problem is translated from a graph into a logic program in the following way. For each vertex a with neighbors $p_1, \ldots, p_j$ and each color $i \in \{0, 1, 2\}$, include the rule

$$\begin{aligned} color(a,i) \quad \leftarrow \quad & \mathrm{not}(color(p_1,i)), \ldots, \mathrm{not}(color(p_j,i)), \\ & \mathrm{not}(color(a,(i+1) \bmod 3)), \mathrm{not}(color(a,(i+2) \bmod 3)). \end{aligned}$$

Then include the rule $h \leftarrow \mathrm{not}(color(a,0)), \mathrm{not}(color(a,1)), \mathrm{not}(color(a,2))$ for each vertex a. Finally add the rules $h_2 \leftarrow \mathrm{not}(h)$ and $h_2 \leftarrow \mathrm{not}(h_2)$. The number of rules is thus $4V + 2$ if V is the number of vertices.

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, where $b_1, \ldots, b_i$ are the neighbors of a excluding b and $a_1, \ldots, a_j$ are the neighbors of b excluding a, include the rule

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

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 $a \leftarrow \text{not}(a)$, and then add the rules $d \leftarrow$ and $d' \leftarrow \text{not}(d')$. If the number of vertices is denoted by V and the number of edges by E, the number of rules in this problem is $4E + V + 1$.

In the win-move examples the rule $win(X) \leftarrow move(X,Y), \text{not}(win(Y))$ is combined with rules of the form $move(\cdot,\cdot) \leftarrow$. The *move* rules represent the moves in a game and are chosen from random graphs.

Since SLG is goal-directed we must add some rules to the translations of the graph problems to make the system compute the whole stable model. We add the rule $s \leftarrow d'$ to the Hamiltonian circuit problem and for each vertex a and each color i we add the rule $s \leftarrow color(a,i)$ to the 3-coloring problem. Finally, for every vertex a in the random graphs we add the rule $s \leftarrow win(a)$ to the win-move examples. The resulting set of rules is then used when testing the SLG system, using the goal s, and the algorithm **sm**.

We have also used the two benchmark programs found in [4] to generate test cases, see Figure 4. We make larger benchmark programs by adding together a number of copies of the original program where we have renamed the atoms. If the number of copies is n, the number of stable models of the first benchmark is 2^{2n}. The second benchmark has no stable models.

The results are tabulated in Figure 4. 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 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 or could not be completed.

All tests were run on a SPARCstation IPX workstation with 16 MB memory and the SunOS 4.1.3 operating system. SLG was compiled in SICStus 2.1 #5 with the fastcode compilation-flag set. The C++ program **sm** was compiled using gcc version 2.5.8. The timing was handled by the SICStus command 'statistics' in SLG, and by the C function 'gettimeofday' in **sm**.

Both systems display quite regular behavior in the tests with a surprisingly monotonic correspondence between the problem size and the execution time even for randomly generated combinatorial problems. In the tests **sm** had actually only one "overlong" run for one particular random shuffle of a program representing Hamiltonian circuits in a planar graph with 20 nodes.

The test results show clearly that **sm** computes stable models significantly faster than SLG. We can explain the performance dissimilarities in the simple examples by the different implementation techniques: **sm** is implemented in C++ while SLG is implemented as a meta-interpreter on top of Prolog. However, in the more complicated test cases involving combinatorial graph problems the difference in performance grows surprisingly fast with the size of the examples and, in fact, SLG can only handle test cases up

		SLG			**sm**		
Ladder graphs, 3-coloring							
Vertices	Rules	min	max	ave	min	max	ave
10	73	0.48	1.24	0.72	0.02	0.05	0.03
14	101	0.74	18.14	3.72	0.02	0.06	0.03
18	129	1.08	508.85	88.82	0.03	0.04	0.03
22	157	1.30	659.86	278.38	0.04	0.08	0.05
100	703				0.22	0.30	0.24
300	2103				1.41	1.50	1.45
500	3503				3.60	3.75	3.65
Planar graphs, 3-coloring							
Vertices	Rules	min	max	ave	min	max	ave
10	73	7.57	34.45	23.81	0.03	0.11	0.04
12	87	2.68	140.01	82.53	0.03	0.04	0.03
14	101	35.09	657.48	390.56	0.03	0.06	0.04
16	115	3.75	2444.90	1421.19	0.04	0.06	0.05
100	703				0.19	0.23	0.21
300	2103				0.63	0.86	0.71
500	3503				1.03	1.35	1.12
Ladder graphs, Hamiltonian circuit							
Vertices	Rules	min	max	ave	min	max	ave
10	64	1.03	1.39	1.21	0.02	0.05	0.03
14	92	2.87	6.02	4.70	0.03	0.05	0.04
18	120	5.62	25.66	19.03	0.05	0.13	0.07
22	148	23.98	89.19	67.78	0.06	0.17	0.10
26	176	18.13	419.64	240.77	0.08	0.34	0.20
30	204	66.70	1649.53	899.46	0.11	0.52	0.25
60	414				0.30	6.31	2.61
80	554				0.50	23.08	12.82
100	694				0.73	48.67	21.55
Planar graphs, Hamiltonian circuit							
Vertices	Rules	min	max	ave	min	max	ave
10	104	33.90	60.61	47.62	0.04	0.10	0.06
12	130	139.40	522.98	343.52	0.06	0.16	0.08
14	156	1140.33	3746.17	2132.48	0.08	0.15	0.10
16	174				0.09	0.25	0.12
18	200				0.12	4.68	0.59
20	226				0.13	97.65	10.10
22	252				0.15	1.73	0.34
24	274				0.17	4.62	0.84
SLG benchmark 1, all stable models							
Copies	Rules	min	max	ave	min	max	ave
3	48	0.94	0.99	0.96	0.13	0.19	0.15
4	64	3.56	3.66	3.60	0.53	0.64	0.59
5	80	14.86	15.35	15.12	2.42	2.62	2.53
6	96	66.10	67.16	66.64	10.53	10.78	10.63
7	112	285.03	289.57	287.21	44.84	46.61	45.40
SLG benchmark 2							
Copies	Rules	min	max	ave	min	max	ave
100	600	2.82	2.92	2.88	0.09	0.11	0.09
150	900	4.38	4.50	4.44	0.13	0.20	0.15
200	1200	5.96	6.07	6.01	0.18	0.24	0.19
250	1500	7.49	7.62	7.57	0.22	0.26	0.23
300	1800	9.09	9.23	9.19	0.27	0.32	0.28
Win-move example, 200 edges							
Vertices	Rules	min	max	ave	min	max	ave
60	460	2.35	4.30	3.34	0.09	0.86	0.44
70	470	1.91	1.99	1.94	0.07	0.29	0.16
80	480	1.14	1.35	1.23	0.07	0.10	0.08
90	490	1.15	1.28	1.21	0.08	0.13	0.09
100	500	1.03	1.28	1.16	0.08	0.12	0.09

Benchmark 1

$z1 \leftarrow v1, w1$
$z3 \leftarrow v2, w1$
$v1 \leftarrow s$
$w1 \leftarrow p$
$t \leftarrow \mathrm{not}(s)$
$p \leftarrow \mathrm{not}(q)$
$m \leftarrow z1$
$m \leftarrow z3$
$z2 \leftarrow v1, w2$
$z4 \leftarrow v2, w2$
$v2 \leftarrow t$
$w2 \leftarrow q$
$s \leftarrow \mathrm{not}(t)$
$q \leftarrow \mathrm{not}(p)$
$m \leftarrow z2$
$m \leftarrow z4$

Benchmark 2

$s \leftarrow p, q$
$s \leftarrow q, r$
$q \leftarrow \mathrm{not}(r)$
$s \leftarrow p, r$
$p \leftarrow \mathrm{not}(q)$
$r \leftarrow \mathrm{not}(p)$

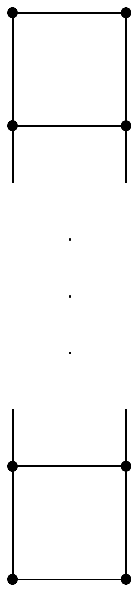

Figure 4: The Test Results, the Benchmark Programs, and the Ladder Graph

to 150–200 rules, whereas our implementation copes with even thousands of rules. The key to our success appears to lie in the new approximation technique for stable models which is closely related to the well-founded semantics also exploited in SLG. The crucial difference is that in SLG the well-founded semantics is employed only in the beginning to produce a residual program whereas we use our well-founded type approximation recursively during the whole search. This keeps the search space smaller in our implementation. The importance of the new approximation technique is supported by the observation that even when turning off our search heuristic we are able to handle considerably larger examples than SLG.

6 Conclusions

We have developed an implementation of the well-founded and stable model semantics for range-restricted function-free normal programs. It includes two modules: an algorithm for implementing the two semantics for ground programs and an algorithm for computing a grounded version of a range-restricted function-free normal program. The emphasis has been on the implementation of the stable model semantics for ground programs. We have tested the implementation with a wide variety of test cases and compared it with a state of the art implementation of the stable model semantics, the SLG system. The tests show that for ground programs our implementation clearly outperforms SLG. The tests indicate that the key to our success is in the new approximation technique for stable models, which is a generalization of the well-founded semantics. Our implementation provides an efficient method for computing the well-founded model of a ground program and the well-founded model is often computed in linear time. It seems that our implementation advances the state of the art in computing the stable model semantics for ground programs to a level where highly non-stratified ground programs with several hundreds of rules can be handled quite efficiently.

Our implementation is based on a bottom-up method and is most suitable when complete stable models should be computed. A certain degree of goal-directedness can be introduced by exploiting the capability of the implementation to perform *focused model search*, i.e., to concentrate the search on models with given characteristics. Further work is needed in handling non-ground programs. In our current approach for a non-ground program, a (sometimes rather large) grounded version is computed and then the efficient algorithm for the ground case is exploited. For computing complete stable models this method can be quite competitive. But for query-answering, especially with respect to the well-founded semantics, goal-directed methods such as that in SLG could be more suitable. An interesting topic for further research would be to couple the goal-directed query-answering method in SLG, which computes a ground residual program, and our implementation of computing the stable model semantics for the ground program.

References

[1] C. Baral and M. Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 19&20(73–148), 1994.

[2] 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.

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

[4] 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.

[5] P. Cholewiński et al. Experimenting with nonmonotonic reasoning. In *Proc. of the 12th International Conference on Logic Programming*, pp. 267–281, Tokyo, June 1995.

[6] 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.

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

[8] D.E. Knuth. Stanford GraphBase, 1993. Available by ftp from labrea.stanford.edu.

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

[10] I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. Fachbericht Informatik 7–96, Universität Koblenz-Landau, 1996. Available at http://www.uni-koblenz.de/universitaet/fb4/publications/GelbeReihe/.

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

[12] A. Van Gelder. The alternating fixpoints of logic programs with negation. In *ACM Symp. on Principles of Database Systems*, pp. 1–10, 1989.

[13] 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, 1991.