Computing stable models: worst-case performance estimates

Zbigniew Lonc[1] and Mirosław Truszczyński[2]

[1] Faculty of Mathematics and Information Science, Warsaw University of Technology, 00-661 Warsaw, Poland

[2] Department of Computer Science, University of Kentucky, Lexington, KY 40506-0046, USA

Abstract. We study algorithms for computing stable models of propositional logic programs and derive estimates on their worst-case performance that are asymptotically better than the trivial bound of $O(m2^n)$, where m is the size of an input program and n is the number of its atoms. For instance, for programs, whose clauses consist of at most two literals (counting the head) we design an algorithm to compute stable models that works in time $O(m \times 1.44225^n)$. We present similar results for several broader classes of programs, as well.

1 Introduction

The stable-model semantics was introduced by Gelfond and Lifschitz [GL88] to provide an interpretation for the negation operator in logic programming. In this paper, we study algorithms to compute stable models of propositional logic programs. Our goal is to design algorithms for which one can derive non-trivial worst-case performance bounds.

Computing stable models is important. It allows us to use logic programming with the stable-model semantics as a *computational* knowledge representation tool and as a declarative programming system. In most cases, when designing algorithms for computing stable models we restrict the syntax to that of DATALOG *with negation* (DATALOG$^\neg$), by eliminating function symbols from the language. When function symbols are allowed, models can be infinite and highly complex, and the general problem of existence of a stable model of a finite logic program is not even semi-decidable [MNR94]. However, when function symbols are not used, stable models are guaranteed to be finite and can be computed.

To compute stable models of *finite* DATALOG$^\neg$ programs we usually proceed in two steps. In the first step, we *ground* an input program P and produce a *finite* propositional program with the same stable models as P (finiteness of the resulting *ground* program is ensured by finiteness of P and absence of function symbols). In the second step, we compute stable models of the ground program by applying search. This general approach is used in *smodels* [NS00] and *dlv* [EFLP00], two most advanced systems to process DATALOG$^\neg$ programs.

It is this second step, computing stable models of propositional logic programs (in particular, programs obtained by grounding DATALOG$^\neg$ programs), that is of interest to us in the present paper. Stable models of a propositional logic program P can be computed by a trivial brute-force algorithm that generates all subsets of the set of atoms of P and, for each of these subsets, checks the stability condition. This algorithm can be implemented to run in time $O(m2^n)$, where m is the size of P and n is the number of atoms in P (we will use m and n in this meaning throughout the paper). The algorithms used in *smodels* and *dlv* refine this brute-force algorithm by employing effective search-space pruning techniques. Experiments show that their performance is much better than that of the brute-force algorithm. However, at present, no non-trivial upper bound on their worst-case running time is known. In fact, *no* algorithms for computing stable models are known whose worst-case performance is *provably* better than that of the brute-force algorithm. Our main goal is to design such algorithms.

To this end, we propose a general template for an algorithm to compute stable models of propositional programs. The template involves an auxiliary procedure whose particular instantiation determines the specific algorithm and its running time. We propose concrete implementations of this procedure and show that the resulting algorithms for computing stable models are asymptotically better than the straightforward algorithm described above. The performance analysis of our algorithms is closely related to the question of how many stable models logic programs may have. We derive bounds on the maximum number of stable models in a program with n atoms and use them to establish lower and upper estimates on the performance of algorithms for computing all stable models.

Our main results concern propositional logic programs, called t-programs, in which the number of literals in rules, including the head, is bounded by a constant t. Despite their restricted syntax t-programs are of interest. Many logic programs that were proposed as encodings of problems in planning, model checking and combinatorics become propositional 2- or 3-programs after grounding. In general, programs obtained by grounding finite DATALOG$^{\neg}$ programs are t-programs, for some fixed, and usually small, t.

In the paper, for every $t \geq 2$, we construct an algorithm that computes all stable models of a t-program P in time $O(m\alpha_t^n)$, where α_t is a constant such that $\alpha_t < 2 - 1/2^t$. For 2-programs we obtain stronger results. We construct an algorithm that computes all stable models of a 2-program in time $O(m3^{n/3}) = O(m \times 1.44225^n)$. We note that $1.44225 < \alpha_2 \approx 1.61803$. Thus, this algorithm is indeed a significant improvement over the algorithm following from general considerations discussed above. We obtain similar results for a subclass of 2-programs consisting of programs that are purely negative and do not contain *dual* clauses. We also get significant improvements in the case when $t = 3$. Namely, we describe an algorithm that computes all stable models of a 3-program P in time $O(m \times 1.70711^n)$. In contrast, since $\alpha_3 \approx 1.83931$, the algorithm implied by the general considerations runs in time $O(m \times 1.83931^n)$.

In the paper we also consider a general case where no bounds on the length of a clause are imposed. We describe an algorithm to compute all stable models of such programs. Its worst-case complexity is slightly lower than that of the brute-force algorithm.

It is well known that, by introducing new atoms, every logic program P can be transformed in polynomial time into a 3-program P' that is, essentially, equivalent to P: every stable model of P is of the form $M' \cap At$, for some stable model M' of P' and, for every stable model M' of P', the set $M' \cap At$ is a stable model of P. This observation might suggest that in order to design fast algorithms to compute stable models, it is enough to focus on the class of 3-programs. It is not the case. In the worst case, the number of new atoms that need to be introduced is of the order of the size of the original program P. Consequently, an algorithm to compute stable models that can be obtained by combining the reduction described above with an algorithm to compute stable models of 3-programs runs in time $O(m2^m)$ and is asymptotically slower than the brute-force approach outlined earlier. Thus, it is necessary to study algorithms for computing stable models designed explicitly for particular classes of programs.

2 Preliminaries

For a detailed account of logic programming and stable model semantics we refer the reader to [GL88,Apt90,MT93]. In the paper, we consider only the propositional case. For a logic program P, by $At(P)$ we denote the set of all atoms appearing in P. We define $Lit(P) = At(P) \cup \{\mathbf{not}(a) \colon a \in At(P)\}$ and call elements of this set *literals*. Literals b and $\mathbf{not}(b)$, where b is an atom, are *dual* to each other. For a literal β, we denote its dual by $\mathbf{not}(\beta)$.

A *clause* is an expression c of the form $p \leftarrow B$ or $\leftarrow B$, where p is an atom and B is a set of literals (no literals in B are repeated). The clause of the first type is called *definite*. The clause of the second type is called a *constraint*. The atom p is the *head* of c and is denoted by $h(c)$. The set of atoms appearing in literals of B is called the *body* of c. The set of all positive literals (atoms) in B is the *positive body* of c, $b^+(c)$, in symbols. The set of atoms appearing in negated literals of B is the *negative body* of c, $b^-(c)$, in symbols.

A *logic program* is a collection of clauses. If every clause of P is definite, P is a *definite logic program*. If every clause in P has an empty positive body, that is, is *purely negative*, P is a *purely negative* program. Finally, a logic program P is a *t-program* if every clause in P has no more than t literals (counting the head).

A clause c is a *tautology* if it is definite and $h(c) \in b^+(c)$, or if $b^+(c) \cap b^-(c) \neq \emptyset$. A clause c is a *virtual constraint* if it is definite and $h(c) \in b^-(c)$. We have the following result [Dix95].

Proposition 1. *Let P be a logic program and let P' be the subprogram of P obtained by removing from P all tautologies, constraints and virtual constraints. If M is a stable model of P then it is a stable model of P'.*

Thanks to this proposition, when designing algorithms for computing stable models we may restrict attention to definite programs without tautologies and virtual constraints.

For a set of literals $L \subseteq Lit(P)$, we define:

$$L^+ = \{a \in At(P) \colon a \in L\} \ \text{ and } \ L^- = \{a \in At(P) \colon \mathbf{not}(a) \in L\}.$$

We also define $L^0 = L^+ \cup L^-$. A set of literals L is *consistent* if $L^+ \cap L^- = \emptyset$. A set of atoms $M \subseteq At(P)$ is *consistent* with a set of literals $L \subseteq Lit(P)$, if $L^+ \subseteq M$ and $L^- \cap M = \emptyset$.

To characterize stable models of a program P that are consistent with a set of literals $L \subseteq Lit(P)$, we introduce a *simplification* of P with respect to L. By $[P]_L$ we denote the program obtained by removing from P

1. every clause c such that $b^+(c) \cap L^- \neq \emptyset$
2. every clause c such that $b^-(c) \cap L^+ \neq \emptyset$
3. every clause c such that $h(c) \in L^0$
4. every occurrence of a literal in L from the bodies of the remaining clauses.

The simplified program $[P]_L$ contains all information necessary to reconstruct stable models of P that are consistent with L. The following result was obtained in [Dix95] (we refer also to [SNV95,CT99]).

Proposition 2. *Let P be a logic program and L be a set of literals of P. If M is a stable model of P consistent with L, then $M \setminus L^+$ is a stable model of $[P]_L$.*

Thus, to compute all stable models of P that are consistent with L, one can first check if L is consistent. If not, there are no stable models consistent with L. Otherwise, one can compute all stable models of $[P]_L$, for each such model M' check whether $M = M' \cup L^+$ is a stable model of P and, if so, output M. This approach is the basis of the algorithm to compute stable models that we present in the following section.

3 A high-level view of stable model computation

We will now describe an algorithm $stable(P, L)$ that, given a *definite* program P and a set of literals L, outputs all stable models of P that are consistent with L. The key concept we need is that of a complete collection. Let P be a logic program. A nonempty collection $\mathcal{A}$ of nonempty subsets of $Lit(P)$ is *complete* for P if every stable model of P is consistent with at least one set $A \in \mathcal{A}$. Clearly, the collection $\mathcal{A} = \{\{a\}, \{\mathbf{not}(a)\}\}$, where a is an atom of P, is an example of a complete collection for P. In the description given below, we assume that *complete*(P) is a procedure that, for a program P, computes a collection of sets of literals that is complete for P.

$stable(P, L)$
if L is consistent **then**
    if $[P]_L = \emptyset$ **then**
        check whether L^+ is a stable model of P and, if so, output it
    else
      $\mathcal{A} := complete([P]_L)$;
      for every $A \in \mathcal{A}$ **do**
        $stable(P, L \cup A)$
    end of *stable*.

Proposition 3. *Let P be a definite finite propositional logic program. For every $L \subseteq Lit(P)$, $stable(P, L)$ returns all stable models of P consistent with L.*

Proof: We proceed by induction on $|At([P]_L)|$. To start, let us consider a call to $stable(P, L)$ in the case when $|At([P]_L)| = 0$ and let M be a set returned by $stable(P, L)$. It follows that L is consistent and that M is a stable model of P. Moreover, since $M = L^+$, M is consistent with L. Conversely, let M be a stable model of P that is consistent with L. By Proposition 2, $M \setminus L^+$ is a stable model of $[P]_L$. Since L is consistent (as M is consistent with L) and $[P]_L = \emptyset$, $M \setminus L^+ = \emptyset$. Since M is consistent with L, $M = L^+$. Thus, M is returned by $stable(P, L)$.

For the inductive step, let us consider a call to $stable(P, L)$, where $|At([P]_L)| > 0$. Let M be a set returned by this call. Then M is returned by a call to $stable(P, L \cup A)$, for some $A \in \mathcal{A}$, where $\mathcal{A}$ is a complete family for $[P]_L$. Since elements of a complete family are nonempty and consist of literals actually occurring in $[P]_L$, $|At([P]_{L \cup A})| < |At([P]_L)|$. By the induction hypothesis it follows that M is a stable model of P consistent with $L \cup A$ and, consequently, with L.

Let us now assume that M is a stable model of P consistent with L. Then, by Proposition 2, $M \setminus L^+$ is a stable model of $[P]_L$. Since $\mathcal{A}$ (computed in line (4)) is a complete collection for $[P]_L$, there is $A \in \mathcal{A}$ such that $M \setminus L^+$ is consistent with A. Since $A \cap L = \emptyset$ (as $A \subseteq At([P]_L)$), M is a stable model of P consistent with $L \cup A$. Since $|At([P]_{L \cup A})| < |At([P]_L)|$, by the induction hypothesis it follows that M is output during the recursive call to $stable(P, L \cup A)$. □

We will now study the performance of the algorithm *stable*. In our discussion we follow the notation used to describe it. Let P be a definite logic program and let $L \subseteq Lit(P)$. Let us consider the following recurrence relation:

$$s(P, L) = \begin{cases} 1 & \text{if } [P]_L = \emptyset \text{ or } L \text{ is not consistent} \\ \sum_{A \in \mathcal{A}} s(P, L \cup A) & \text{otherwise.} \end{cases}$$

As a corollary to Proposition 3 we obtain the following result.

Corollary 1. *Let P be a finite definite logic program and let $L \subseteq Lit(P)$. Then, P has at most $s(P, L)$ stable models consistent with L. In particular, P has at most $s(P, \emptyset)$ stable models.*

We will use the function $s(P, L)$ to estimate not only the number of stable models in definite logic programs but also the running time of the algorithm *stable*. Indeed, let us observe that the total number of times we make a call to the algorithm *stable* when executing $stable(P, L)$ (including the "top-level" call to $stable(P, L)$) is given by $s(P, L)$. We associate each execution of the instruction (i), where $0 \leq i \leq 5$, with the call in which the instruction is executed. Consequently, each of these instructions is executed no more than $s(P, L)$ times during the execution of $stable(P, L)$.

Let m be the size of a program P. There are linear-time algorithms to check whether a set of atoms is a stable model of a program P. Thus, we obtain the following result concerned with the performance of the algorithm *stable*.

Theorem 1. *If the procedure complete runs in time $O(t(m))$, where m is the size of an input program P, then executing the call stable(P, L), where $L \subseteq Lit(P)$, requires $O(s(P, L)(t(m) + m))$ steps in the worst case.*

The specific bound depends on the procedure *complete*, as it determines the recurrence for $s(P, L)$. It also depends on the implementation of the procedure *complete*, as the implementation determines the second factor in the running-time formula derived above.

Throughout the paper (except for Section 7, where a different approach is used), we specify algorithms to compute stable models by describing particular versions of the procedure *complete*. We obtain estimates on the running time of these algorithms by analyzing the recurrence for $s(P, L)$ implied by the procedure *complete*. As a byproduct to these considerations, we obtain bounds on the maximum number of stable models of a logic program with n atoms.

4 t-programs

In this section we will instantiate the general algorithm to compute stable models to the case of t-programs, for $t \geq 2$. To this end, we will describe a procedure that, given a definite t-program P, returns a complete collection for P.

Let P be a definite t-program and let $x \leftarrow \beta_1, \ldots, \beta_k$, where β_i are literals and $k \leq t - 1$, be a clause in P. For every $i = 1, \ldots, k$, let us define

$$A_i = \{\mathbf{not}(x), \beta_1, \ldots, \beta_{i-1}, \mathbf{not}(\beta_i)\}$$

It is easy to see that the family $\mathcal{A} = \{\{x\}, A_1, \ldots, A_k\}$ is complete for P. We will assume that this complete collection is computed and returned by the procedure *complete*. Clearly, computing $\mathcal{A}$ can be implemented to run in time $O(m)$.

To analyze the resulting algorithm *stable*, we use our general results from the previous section. Let us define

$$c_n = \begin{cases} K_t & \text{if } 0 \leq n < t \\ c_{n-1} + \ldots + c_{n-t} & \text{otherwise,} \end{cases}$$

where K_t is the maximum possible value of $s(P, L)$ for a t-program P and a set of literals $L \subseteq Lit(P)$ such that $|At(P)| - |L| \leq t$. We will prove that if P is a t-program, $L \subseteq Lit(P)$, and $|At(P)| - |L| \leq n$, then $s(P, L) \leq c_n$. We proceed by induction on n. If $n < t$, then the assertion follows by the definition of K_t. So, let us assume that $n \geq t$. If L is not consistent or $[P]_L = \emptyset$, $s(P, L) = 1 \leq c_n$. Otherwise,

$$s(P, L) = \sum_{A \in \mathcal{A}} s(P, L \cup A) \leq c_{n-1} + c_{n-2} + \ldots + c_{n-t} = c_n.$$

The inequality follows by the induction hypothesis, the definition of $\mathcal{A}$, and the monotonicity of c_n. The last equality follows by the definition of c_n. Thus, the induction step is complete.

The characteristic equation of the recurrence c_n is $x^t = x^{t-1} + \ldots + x + 1$. Let α_t be the largest real root of this equation. One can show that for $t \geq 2$, $1 < \alpha_t < 2 - 1/2^t$. In particular, $\alpha_2 \approx 1.61803$, $\alpha_3 \approx 1.83931$, $\alpha_4 \approx 1.92757$ and $\alpha_5 \approx 1.96595$. The discussion in Section 3 implies the following two theorems.

Theorem 2. *Let t be an integer, $t \geq 2$. There is an algorithm to compute stable models of t-programs that runs in time $O(m\alpha_t^n)$, where n is the number of atoms and m is the size of the input program.*

Theorem 3. *Let t be an integer, $t \geq 2$. There is a constant C_t such that every t-program P has at most $C_t\alpha_t^n$ stable models, where $n = |At(P)|$.*

Since for every t, $\alpha_t < 2$, we indeed obtain an improvement over the straightforward approach. However, the scale of the improvement diminishes as t grows.

To establish lower bounds on the number of stable models and on the worst-case performance of algorithms to compute them, we define $P(n,t)$ to be a logic program such that $|At(P)| = n$ and P consists of all clauses of the form

$$x \leftarrow \mathbf{not}(b_1), \ldots, \mathbf{not}(b_t),$$

where $x \in At(P)$ and $\{b_1, \ldots, b_t\} \subseteq At(P) \setminus \{x\}$ are different atoms. It is easy to see that $P(n,t)$ is a $(t+1)$-program with n atoms and that stable models of $P(n,t)$ are precisely those subsets of $At(P)$ that have $n-t$ elements. Thus, $P(n,t)$ has exactly $\binom{n}{t}$ stable models.

Clearly, the program $P(2t-1, t-1)$ is a t-program over the set of $2t-1$ atoms. Moreover, it has $\binom{2t-1}{t}$ stable models. Let $kP(2t-1, t-1)$ be the logic program formed by the disjoint union of k copies of $P(2t-1, t-1)$ (sets of atoms of different copies of $P(2t-1, t-1)$ are disjoint). It is easy to see that $kP(2t-1, t-1)$ has $\binom{2t-1}{t}^k$ stable models. As an easy corollary of this observation we obtain the following result.

Theorem 4. *Let t be an integer, $t \geq 2$. There is a constant D_t such that for every n there is a t-program P with at least $D_t \times \binom{2t-1}{t}^{n/2t-1}$ stable models.*

This result implies that every algorithm for computing all stable models of a t-program in the worst-case requires $\Omega(\binom{2t-1}{t}^{n/2t-1})$ steps, as there are programs for which at least that many stable models need to be output. These lower bounds specialize to approximately $\Omega(1.44224^n)$, $\Omega(1.58489^n)$, $\Omega(1.6618^n)$ and $\Omega(1.71149^n)$, for $t = 2, 3, 4, 5$, respectively.

5 2-programs

Stronger results can be derived for more restricted classes of programs. We will now study the case of 2-programs and prove the following two theorems.

Theorem 5. *There is an algorithm to compute stable models of 2-programs that runs in time $O(m3^{n/3}) = O(m \times 1.44225^n)$, where n is the number of atoms in P and m is the size of P.*

Theorem 6. *There is a constant C such that every 2-program P with n atoms, has at most $C \times 3^{n/3}$ ($\approx C \times 1.44225^n$) stable models.*

By Proposition 1, to prove these theorems it suffices to limit attention to the case of definite programs not containing tautologies and virtual constraints. We will adopt this assumption and derive both theorems from general results presented in Section 3.

Let P be a definite 2-program. We say that an atom $b \in At(P)$ is a *neighbor* of an atom $a \in At(P)$ if P contains a clause containing both a and b (one of them as the head, the other one appearing positively or negatively in the body). By $n(a)$ we will denote the number of neighbors of an atom a. Since we assume that our programs contain neither tautologies nor virtual constraints, no atom a is its own neighbor.

We will now describe the procedure *complete*. The complete family returned by the call to *complete*(P) depends on the program P. We list below several cases that cover all definite 2-programs without tautologies and virtual constraints. In each of these cases, we specify a complete collection to be returned by the procedure *complete*.

Case 1. There is an atom, say x, such that P contains a clause with the head x and with the empty body (in other words, x is a fact of P). We define $\mathcal{A} = \{\{x\}\}$. Clearly, every stable model of P contains x. Thus, $\mathcal{A}$ is complete.

Case 2. There is an atom, say x, that does not appear in the head of any clause in P. We define $\mathcal{A} = \{\{\mathbf{not}(x)\}\}$. It is well known that x does not belong to *any* stable model of P. Thus, $\mathcal{A}$ is complete for P.

Case 3. There are atoms x and y, $x \neq y$, such that $x \leftarrow y$ and at least one of $x \leftarrow \mathbf{not}(y)$ and $y \leftarrow \mathbf{not}(x)$ are in P. In this case, we set $\mathcal{A} = \{\{x\}\}$. Let M be a stable model of P. If $y \in M$, then $x \in M$ (due to the fact that the clause $x \leftarrow y$ is in P). Otherwise, $y \notin M$. Since M satisfies $x \leftarrow \mathbf{not}(y)$ or $y \leftarrow \mathbf{not}(x)$, it again follows that $x \in M$. Thus, $\mathcal{A}$ is complete.

Case 4. There are atoms x and y such that $x \leftarrow y$ and $y \leftarrow x$ are both in P. We define

$$\mathcal{A} = \{\{x, y\}, \{\mathbf{not}(x), \mathbf{not}(y)\}\}.$$

If M is a stable model of P then, clearly, $x \in M$ if and only if $y \in M$. It follows that either $\{x, y\} \subseteq M$ or $\{x, y\} \cap M = \emptyset$. Thus, $\mathcal{A}$ is complete for P. Moreover, since $x \neq y$ (P does not contain clauses of the form $w \leftarrow w$), each set in $\mathcal{A}$ has at least two elements.

Case 5. None of the Cases 1-4 holds and there is an atom, say x, with exactly one neighbor, y. Since P does not contain clauses of the form $w \leftarrow w$ and $w \leftarrow \mathbf{not}(w)$, we have $x \neq y$. Moreover, x must be the head of at least one clause (since we assume here that Case 2 does not hold).

Subcase 5a. P contains the clause $x \leftarrow y$. We define

$$\mathcal{A} = \{\{x, y\}, \{\mathbf{not}(x), \mathbf{not}(y)\}\}.$$

Let M be a stable model of P. If $y \in M$ then, clearly, $x \in M$. Since we assume that Case 3 does not hold, the clause $x \leftarrow y$ is the only clause in P with x as

the head. Thus, if $y \notin M$, then we also have that $x \notin M$. Hence, $\mathcal{A}$ is complete.
Subcase 5b. P does not contain the clause $x \leftarrow y$. We define

$$\mathcal{A} = \{\{x, \mathbf{not}(y)\}, \{\mathbf{not}(x), y\}\}.$$

Let M be a stable model of P. Since x is the head of at least one clause in P, it follows that the clause $x \leftarrow \mathbf{not}(y)$ belongs to P. Thus, if $y \notin M$ then $x \in M$. If $y \in M$ then, since $x \leftarrow \mathbf{not}(y)$ is the only clause in P with x as the head, $x \notin M$. Hence, $\mathcal{A}$ is complete.
Case 6. None of the Cases 1-5 holds. Let $w \in At(P)$ be an atom. By $x_1, \ldots, x_p$ we denote all atoms x in P such that $w \leftarrow \mathbf{not}(x)$ or $x \leftarrow \mathbf{not}(w)$ is a clause in P. Similarly, by $y_1, \ldots, y_q$ we denote all atoms y in P such that $y \leftarrow w$ is a clause of P. Finally, by $z_1, \ldots, z_r$ we denote all atoms z of P such that $w \leftarrow z$ is a clause of P. By our earlier discussion it follows that the sets $\{x_1, \ldots, x_p\}$, $\{y_1, \ldots, y_q\}$ and $\{z_1, \ldots, z_r\}$, are pairwise disjoint and cover all neighbors of w. That is, the number of neighbors of w is given by $p + q + r$. Since we exclude Case 5 here, $p+q+r \geq 2$. Further, since w is the head of at least one edge (Case 2 does not hold), it follows that $p + r \geq 1$
Subcase 6a. For some atom w, $q \geq 1$ or $p + q + r \geq 3$. Then, we define

$$\mathcal{A} = \{\{w, y_1, \ldots, y_q\}, \{\mathbf{not}(w), x_1, \ldots, x_p, \mathbf{not}(z_1), \ldots, \mathbf{not}(z_r)\}\}.$$

It is easy to see that $\mathcal{A}$ is complete for P. Moreover, if $q \geq 1$ then, since $p+r \geq 1$, each of the two sets in $\mathcal{A}$ has at least two elements. If $p + q + r \geq 3$, then either each set in $\mathcal{A}$ has at least two elements, or one of them has one element and the other one at least four elements.
Subcase 6b. Every atom w has exactly two neighbors, and does not appear in the body of any Horn clause of P. It follows that all clauses in P are purely negative. Let w be an arbitrary atom in P. Let u and v be the two neighbors of w. The atoms u and v also have two neighbors each, one of them being w. Let u' and v' be the neighbors of u and v, respectively, that are different from w. We define

$$\mathcal{A} = \{\{\mathbf{not}(w), u, v\}, \{\mathbf{not}(u), w, u'\}, \{\mathbf{not}(v), w, v'\}\}.$$

Let M be a stable model of P. Let us assume that $w \notin M$. Since w and u are neighbors, there is a clause in P built of w and u. This clause is purely negative and it is satisfied by M. It follows that $u \in M$. A similar argument shows that $v \in M$, as well. If $w \in M$ then, since M is a stable model of P, there is a 2-clause C in P with the head w and with the body satisfied by M. Since P consists of purely negative clauses, and since u and v are the only neighbors of w, $C = w \leftarrow \mathbf{not}(u)$ or $C = w \leftarrow \mathbf{not}(v)$. Let us assume the former. It is clear that $u \notin M$ (since M satisfies the body of C). Let us recall that u' is a neighbor of u. Consequently, u and u' form a purely negative clause of P. This clause is satisfied by M. Thus, $u' \in M$ and M is consistent with $\{\mathbf{not}(u), w, u'\}$. In the other case, when $C = w \leftarrow \mathbf{not}(v)$, a similar argument shows that M is consistent with $\{\mathbf{not}(v), w, v'\}$. Thus, every stable model of P is consistent with one of the three sets in $\mathcal{A}$. In other words, $\mathcal{A}$ is complete.

Clearly, given a 2-program P, deciding which of the cases described above holds for P can be implemented to run in linear time. Once that is done, the output collection can be constructed and returned in linear time, too.

This specification of the procedure *complete* yields a particular algorithm to compute stable models of definite 2-programs without tautologies and virtual constraints. To estimate its performance and obtain the bound on the number of stable models, we define

$$c_n = \begin{cases} K & \text{if } 0 \leq n < 4 \\ \max\{c_{n-1}, 2c_{n-2}, c_{n-1} + c_{n-4}, 3c_{n-3}\} & \text{otherwise,} \end{cases}$$

where K is the maximum possible value of $s(P, L)$, when P is a definite finite propositional logic program, $L \subseteq Lit(P)$ and $|At(P)| - |L| \leq 3$. It is easy to see that K is a constant that depends neither on P nor on L. We will prove that $s(P, L) \leq c_n$, where $n = |At(P)| - |L|$. If $n \leq 3$, then the assertion follows by the definition of K. So, let us assume that $n \geq 4$. If L is not consistent or $[P]_L = \emptyset$, $s(P, L) = 1 \leq c_n$. Otherwise,

$$s(P, L) = \sum_{A \in \mathcal{A}} s(P, L \cup A) \leq \max\{c_{n-1}, 2c_{n-2}, c_{n-1} + c_{n-4}, 3c_{n-3}\} = c_n.$$

The inequality follows by the induction hypothesis, the properties of the complete families returned by *complete* (the cardinalities of sets forming these complete families) and the monotonicity of c_n.

Using well-known properties of linear recurrence relations, it is easy to see that $c_n = O(3^{n/3}) = O(1.44225^n)$. Thus, Theorems 5 and 6 follow.

As concerns bounds on the number of stable models of a 2-program, a stronger (exact) result can be derived. Let

$$g_n = \begin{cases} 3^{n/3} & \text{if } n = 0 \pmod 3 \\ 4 \times 3^{(n-4)/3} & \text{if } n = 1 \pmod 3\text{, and } n > 1 \\ 2 \times 3^{(n-2)/3} & \text{if } n = 2 \pmod 3 \\ 1 & \text{if } n = 1 \end{cases}$$

Exploiting connections between stable models of purely negative definite 2-programs and maximal independent sets in graphs, and using some classic results from graph theory [MM65] one can prove the following result.

Corollary 2. *Let P be a 2-program with n atoms. Then P has no more than g_n stable models.*

The bound of Corollary 2 cannot be improved as there are logic programs that achieve it. Let $P(p_1, \ldots, p_k)$, where for every i, $p_i \geq 2$, be a disjoint union of programs $P(p_1, 1), \ldots, P(p_k, 1)$ (we discussed these programs in Section 2). Each program $P(p_i, 1)$ has p_i stable models. Thus, the number of stable models of $P(p_1, \ldots, p_k)$ is $p_1 p_2 \ldots p_k$. Let P be a logic program with $n \geq 2$ atoms and of the form $P(3, \ldots, 3)$, $P(2, 3, \ldots, 3)$ or $P(4, 3, \ldots, 3)$, depending on $n (\text{mod } 3)$. It is easy to see that P has g_n stable models. In particular, it follows that our algorithm to compute all stable models of 2-programs is must execute at least $\Omega(3^{n/3})$ steps in the worst case.

Narrowing the class of programs leads to still better bounds and faster algorithms. We will discuss one specific subclass of the class of 2-programs here. Namely, we will consider definite purely negative 2-programs with no *dual* clauses (two clauses are called *dual* if they are of the form $a \leftarrow \mathbf{not}(b)$ and $b \leftarrow \mathbf{not}(a)$). We denote the class of these programs by $\mathcal{P}_2^n$. Using the same approach as in the case of arbitrary 2-programs, we can prove the following two theorems.

Theorem 7. *There is an algorithm to compute stable models of 2-programs in the class $\mathcal{P}_2^n$ that runs in time $O(m \times 1.23651^n)$, where n is the number of atoms and m is the size of an input program.*

Theorem 8. *There is a constant C such that every 2-program $P \in \mathcal{P}_2^n$ has at most $C \times 1.23651^n$ stable models.*

Theorem 8 gives an upper bound on the number of stable models of a program in the class $\mathcal{P}_2^n$. To establish a lower bound, we define S_6 to be a program over the set of atoms $a_1, \ldots, a_6$ and containing the rules (the arithmetic of indices is performed modulo 6): $a_{i+1} \leftarrow \mathbf{not}(a_i)$ and $a_{i+2} \leftarrow \mathbf{not}(a_i)$, $i = 0, 1, 2, 3, 4, 5$. The program S_6 has three stable models: $\{a_0, a_1, a_3, a_4\}$, $\{a_1, a_2, a_4, a_5\}$ and $\{a_2, a_3, a_5, a_0\}$.

Let P be the program consisting of k copies of S_6, with mutually disjoint sets of atoms. Clearly, P has 3^k stable models. Thus, there is a constant D such that for every $n \geq 1$ there is a program P with n atoms and with at least $D \times 3^{n/6}$ ($\approx D \times 1.20094^n$) stable models.

6 3-programs

We will now present our results for the class of 3-programs. Using similar techniques as those presented in the previous section, we prove the following two theorems.

Theorem 9. *There is an algorithm to compute stable models of 3-programs that runs in time $O(m \times 1.70711^n)$, where m is the size of the input.*

Theorem 10. *There is a constant C such that every 3-program P has at most $C \times 1.70711^n$ stable models.*

The algorithm whose existence is claimed in Theorem 9 is obtained from the general template described in Section 3 by a proper instantiation of the procedure *complete* (in a similar way to that presented in detail in the previous section for the case of 2-programs).

The lower bound in this case follows from an observation made in Section 4 that there is a constant D_3 such that for every n there is a 3-program P such that P has at least $D_3 \times 1.58489^n$) stable models (cf. Theorem 4). Thus, every algorithm for computing all stable models of 3-programs must take at least $\Omega(1.58489^n)$ steps in the worst case.

7 The general case

In this section we present an algorithm that computes all stable models of arbitrary propositional logic programs. It runs in time $O(m2^n/\sqrt{n})$ and so, provides an improvement over the trivial bound $O(m2^n)$. However, our approach is quite different from that used in the preceding sections. The key component of the algorithm is an auxiliary procedure *stable_aux*(P,π). Let P be a logic program and let $At(P) = \{x_1, x_2, \ldots, x_n\}$. Given P and a permutation π of $\{1, 2, \ldots, n\}$, the procedure *stable_aux*(P,π) looks for an index j, $1 \leq j \leq n$, such that the set $\{x_{\pi(j)}, \ldots, x_{\pi(n)}\}$ is a stable model of P. Since no stable model of P is a proper subset of another stable model of P, for any permutation π there is at most one such index j. If such j exists, the procedure outputs the set $\{x_{\pi(j)}, \ldots, x_{\pi(n)}\}$.

In the description of the algorithm *stable_aux*, we use the following notation. For every atom a, by $pos(a)$ we denote the list of all clauses which contain a (as a non-negated atom) in their bodies, and by $neg(a)$ a list of all clauses that contain $\mathbf{not}(a)$ in their bodies. Given a standard linked-list representation of logic programs, all these lists can be computed in time linear in m.

Further, for each clause C, we introduce counters $p(C)$ and $n(C)$. We initialize $p(C)$ to be the number of positive literals (atoms) in the body of C. Similarly, we initialize $n(C)$ to be the number of negative literals in the body of C. These counters are used to decide whether a clause belongs to the reduct of the program and whether it "fires" when computing the least model of the reduct.

stable_aux(P,π)
(1) $M = At(P)$;
(2) $Q :=$ set of clauses C such that $p(C) = n(C) = 0$;
(3) $lm := \emptyset$;
(4) **for** $j = 1$ **to** n **do**
(5)   **while** $Q \neq \emptyset$ **do**
(6) $C_0 :=$ any clause in Q;
(7)   mark C_0 as used and remove it from Q;
(8)   **if** $h(C_0) \notin lm$ **then**
(9) $lm := lm \cup \{h(C_0)\}$;
(10)   **for** $C \in pos(h(C_0))$ **do**
(11) $p(C) := p(C) - 1$;
(12)   **if** $p(C) = 0$ & $n(C) = 0$ & C not used **then** add C to Q;
(13)   **if** $lm = M$ **then** output M and stop;
(14) $M := M \setminus \{x_{\pi(j)}\}$;
(15)   **for** $C \in neg(x_{\pi(j)})$ **do**
(16) $n(C) := n(C) - 1$;
(17)   **if** $n(C) = 0$ & $p(C) = 0$ & C not used **then** add C to Q.

Let us define $M_j = \{x_{\pi(j)}, \ldots, x_{\pi(n)}\}$. Intuitively, the algorithm *stable_aux* works as follows. In the iteration j of the **for** loop it computes the least model of the reduct P^{M_j} (lines (5)-(12)). Then it tests whether $M_j = lm(P^{M_j})$ (line (13)). If so, it outputs M_j (it is a stable model of P) and terminates. Otherwise, it computes the reduct $P^{M_{j+1}}$. In fact the reduct is not explicitly computed. Rather, the number of negated literals in the body of each rule is updated to reflect the fact that we shift attention from the set M_j to the set M_{j+1} (lines (14)-(17)). The key to the algorithm is the fact that it computes reducts P^{M_j} and least models $lm(P^{M_j})$ in an incremental way and, so, tests n candidates M_j for stability in time $O(m)$ (where m is the size of the program).

Proposition 4. *Let P be a logic program and let $At(P) = \{x_1, \ldots, x_n\}$. For every permutation π of $\{1, \ldots, n\}$, if $M = \{x_{\pi(j)}, \ldots, x_{\pi(n)}\}$ then the procedure $stable\_aux(P, \pi)$ outputs M if and only if M is a stable model of P. Moreover, the procedure stable_aux runs in $O(m)$ steps, where m is the size of P.*

We will now describe how to use the procedure *stable_aux* in an algorithm to compute stable models of a logic program. A collection $\mathcal{S}$ of permutations of $\{1, 2, \ldots, n\}$ is *full* if every subset S of $\{1, 2, \ldots, n\}$ is a final segment (suffix) of a permutation in $\mathcal{S}$ or, more precisely, if for every subset S of $\{1, 2, \ldots, n\}$ there is a permutation $\pi \in \mathcal{S}$ such that $S = \{\pi(n - |S| + 1), \ldots, \pi(n)\}$.

If S_1 and S_2 are of the same cardinality then they cannot occur as suffixes of the same permutation. Since there are $\binom{n}{\lfloor n/2 \rfloor}$ subsets of $\{1, 2, \ldots, n\}$ of cardinality $\lfloor n/2 \rfloor$, every full family of permutations must contain at least $\binom{n}{\lfloor n/2 \rfloor}$ elements. An important property is that for every $n \geq 0$ there is a full family of permutations of cardinality $\binom{n}{\lfloor n/2 \rfloor}$. An algorithm to compute such a minimal full set of permutations, say $\mathcal{S}_{min}$, is described in [Knu98] (Vol. 3, pages 579 and 743-744). We refer to this algorithm as $perm(n)$. The algorithm $perm(n)$ enumerates all permutations in $\mathcal{S}_{min}$ by generating each next permutation entirely on the basis of the previous one. The algorithm $perm(n)$ takes $O(n)$ steps to generate a permutation and each permutation is generated only once.

We modify the algorithm $perm(n)$ to obtain an algorithm to compute all stable models of a logic program P. Namely, each time a new permutation, say π, is generated, we make a call to $stable\_aux(P, \pi)$. We call this algorithm $stable^p$. Since $\binom{n}{\lfloor n/2 \rfloor} = \Theta(2^n/\sqrt{n})$ we have the following result.

Proposition 5. *The algorithm $stable^p$ is correct and runs in time $O(m2^n/\sqrt{n})$.*

Since the program $P(n, \lfloor n/2 \rfloor)$ has exactly $\binom{n}{\lfloor n/2 \rfloor}$ stable models, every algorithm to compute all stable models of a logic program must take at least $\Omega(2^n/\sqrt{n})$ steps.

8 Discussion and conclusions

We presented algorithms for computing stable models of logic programs with worst-case performance bounds asymptotically better than the trivial bound of $O(m2^n)$. These are first results of that type in the literature. In the general case, we proposed an algorithm that runs in time $O(m2^n/\sqrt{n})$ improving the performance over the brute-force approach by the factor of $\sqrt{n}$. Most of our work, however, was concerned with algorithms for computing stable models of t-programs. We proposed an algorithm that computes stable models of t-programs in time $O(m\alpha_t^n)$, where $\alpha_t < 2 - 1/2^t$. We strengthened these results in the case of 2- and 3-programs. In the first case, we presented an algorithm that runs in time $O(m3^{n/3})$ ($\approx O(m \times 1.44225^n)$). For the case of 3-programs, we presented an algorithm running in the worst case in time $O(m \times 1.70711^n)$.

In addition to these contributions, our work leads to several interesting questions. A foremost among them is whether our results can be further improved. First, we observe that in the case when the task is to compute *all* stable models, we already have proved optimality (up to a polynomial factor) of the algorithms developed for the class of all programs and the class of all 2-programs. However, in all other cases there is still room for improvement — our lower and upper bounds do not coincide.

The situation gets even more interesting when we want to compute one stable model (if stable models exist) rather than all of them. Algorithms we presented here can, of course, be adapted to this case (by terminating them as soon as the first model is found). Thus, the upper bounds derived in this paper remain valid. But the lower bounds, which we derive on the basis of the number of stable models input programs may have, do not. In particular, it is no longer clear whether the algorithm we developed for the case of 2-programs remains optimal. One cannot exclude existence of pruning techniques that, in the case when the input program has stable models, would on occasion eliminate from considerations parts of the search space possibly containing some stable models, recognizing that the remaining portion of the search space still contains some.

Such search space pruning techniques are possible in the case of satisfiability testing. For instance, the pure literal rule, sometimes used by implementations of the Davis-Putnam procedure, eliminates from considerations parts of search space that may contain stable models [MS85,Kul99]. However, the part that remains is guaranteed to contain a model as long as the input theory has one. No examples of analogous search space pruning methods are known in the case of stable model computation. We feel that nonmonotonicity of the stable model semantics is the reason for that but a formal account of this issue remains an open problem.

Finally, we note that many algorithms to compute stable models can be cast as instantiations of the general template introduced in Section 3. For instance, it is the case with the algorithm used in *smodels*. To view *smodels* in this way, we define the procedure complete as (1) picking (based on full lookahead) an atom x on which the search will split; (2) computing the set of literals $A(x)$ by assuming that x holds and by applying the unit propagation procedure of *smodels* (based, we recall on the ideas behind the well-founded semantics); (3) computing in the same way the set $A(\mathbf{not}(x))$ by assuming that $\mathbf{not}(x)$ holds; and (4) returning the family $\mathcal{A} = \{A(x), A(\mathbf{not}(x))\}$. This family is clearly complete.

While different in some implementation details, the algorithm obtained from our general template by using this particular version of the procedure complete is essentially equivalent to that of *smodels*. By modifying our analysis in Section

5, one can show that on 2-programs *smodels* runs in time $O(m \times 1.46558^n)$ and on purely negative programs without dual clauses in time $O(m \times 1.32472^n)$. To the best of our knowledge these are first non-trivial estimates of the worst-case performance of *smodels*. These bounds are worse from those obtained from the algorithms we proposed here, as the techniques we developed were not designed with the analysis of *smodels* in mind. However, they demonstrate that the worst-case analysis of algorithms such as *smodels*, which is an important open problem, may be possible.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0097278.

References

[Apt90] K. Apt. Logic programming. In J. van Leeuven, editor, *Handbook of theoretical computer science*, pages 493–574. Elsevier, Amsterdam, 1990.

[BE96] P.A. Bonatti and T. Eiter. Querying Disjunctive Databases Through Nonmonotonic Logics. *Theoretical Computer Science*, 160:321–363, 1996.

[CT99] P. Cholewiński and M. Truszczyński. Extremal problems in logic programming and stable model computation. *Journal of Logic Programming*, 38:219–242, 1999.

[Dix95] J. Dix. A classification theory of semantics of normal logic programs: Ii. weak properties. *Fundamenta Informaticae*, 22(3):257 – 288, 1995.

[EFLP00] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative problem-solving in DLV. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, Dordrecht, 2000.

[GL88] M. Gelfond and V. Lifschitz. The stable semantics for logic programs. In R. Kowalski and K. Bowen, editors, *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[Knu98] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison Wesley, 1998. Second edition.

[Kul99] O. Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, pages 1–72, 1999.

[MM65] J.W. Moon and L. Moser. On cliques in graphs. *Israel J. Math*, pages 23–28, 1965.

[MNR94] W. Marek, A. Nerode, and J. B. Remmel. The stable models of predicate logic programs. *Journal of Logic Programming*, 21(3):129–154, 1994.

[MS85] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, pages 287–295, 1985.

[MT93] W. Marek and M. Truszczyński. *Nonmonotonic logics; context-dependent reasoning*. Springer-Verlag, Berlin, 1993.

[NS00] I. Niemelä and P. Simons. Extending the smodels system with cardinality and weight constraints. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer Academic Publishers, 2000.

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