Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Report 58
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 58
Espoo 2000 HUT-TCS-A58

Extending and Implementing the Stable Model Semantics

Patrik Simons

Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Report 58
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 58
Espoo 2000 HUT-TCS-A58

Extending and Implementing the Stable Model Semantics

Patrik Simons

Dissertation for the degree of Doctor of Technology to be presented with due permission for public examination and debate in Auditorium T2 at Helsinki University of Technology (Espoo, Finland) on the 28th of April, 2000, at 12 o'clock noon.

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Abstract

An algorithm for computing the stable model semantics of logic programs is developed. It is shown that one can extend the semantics and the algorithm to handle new and more expressive types of rules. Emphasis is placed on the use of efficient implementation techniques. In particular, an implementation of lookahead that safely avoids testing every literal for failure and that makes the use of lookahead feasible is presented. In addition, a good heuristic is derived from the principle that the search space should be minimized.

Due to the lack of competitive algorithms and implementations for the computation of stable models, the system is compared with three satisfiability solvers. This shows that the heuristic can be improved by breaking ties, but leaves open the question of how to break them. It also demonstrates that the more expressive rules of the stable model semantics make the semantics clearly preferable over propositional logic when a problem has a more compact logic program representation. Conjunctive normal form representations are never more compact than logic program ones.

Preface

I began working at the Laboratory for Theoretical Computer Science in 1995. This thesis is the result. I would like to thank Professor Leo Ojala and Docent Ilkka Niemelä for giving me this opportunity, and I would also like to thank my colleagues in the laboratory for creating a pleasant working atmosphere. I am especially grateful for the advice and comments that I have got from Ilkka during these years.

The research has been founded by the Academy of Finland (project 43963) and the Helsinki Graduate School in Computer Science and Engineering. The financial support from the Jenny and Antti Wihuri Foundation is acknowledged with gratitude.

I dedicate this work to my love Eeva, to my parents, and to my sister.

Otaniemi, January 2000

Patrik Simons

Contents

List of Figures

List of Algorithms

1 Introduction

Logic programming with the stable model semantics has emerged as a viable constraint programming paradigm [34, 42]. In this paradigm problems are expressed as logic programs, and the stable models of the programs give the solutions to the problems. Since it is in general hard to compute stable models, the typical algorithm employs an exhaustive search when it tries to find a stable model. Different algorithms vary in how much of the structure of a program they exploit when they limit their search.

We extend the stable model semantics to include three new types of rules:

- choice rules for encoding subsets of a set,
- cardinality rules for enforcing cardinality limits on the subsets, and
- weight rules for writing inequalities over weighted linear sums.

In addition, we define optimize statements that can be used to find the largest or smallest stable models. The new rules can be translated into normal rules, but not without introducing extra atoms and rules. The motivation for including the new rules comes from reflecting on how to solve problems by encoding them as logic programs. More expressive rules lead to smaller programs that are easier to solve. The reason for these three particular types is found in their usefulness and in the ease and efficiency with which they can be implemented.

We only consider variable-free programs. Ground programs can be produced from non ground ones by, for example, the tool *lparse* [64]. Accordingly, we see the propositional rules as primitives that can be employed as building blocks for a high level language.

A procedure for computing the stable models of logic programs containing the new rules has been implemented. The procedure and its implementation bear the name *smodels*. The purpose of this work is to explain the workings of *smodels*.

1.1 Related Work

The stable model semantics is a form of nonmonotonic reasoning that is closely related to the default logic of Reiter [52, 35], circumscription of McCarthy [37, 38, 32], and the autoepistemic logic of Moore [39, 21]. Since the stable model semantics has become a standard method for supplying semantics to nonmonotonic logic programs, there is a considerable interest in automating its computation. The earliest methods for finding stable models were based on the truth maintenance system of Doyle [15] and the assumption-based TMS of deKleer [11], see [16, 17, 50]. One of the first algorithms that took advantage of the well-founded semantics [65] was the one presented in [29] and generalized in [30]. Lately, more specialized algorithms have been developed [2, 63, 7, 12, 8, 18]. However, they all suffer from exponential space complexity or weak pruning techniques.

If we look in a broader context, then finding a stable model is a combinatorial search problem. Other forms of combinatorial search problems are propositional satisfiability, constraint satisfaction, constraint logic programming and integer linear programming problems, and some other logic programming problems such as those expressible in NP-SPEC [6]. The difference between these problem formalisms and the stable model semantics is that they do not include default negation. In addition, all but the last one are not nonmonotonic.

From an algorithmic standpoint the progenitor of the *smodels* algorithm is the Davis-Putnam (-Logemann-Loveland) procedure [10] for determining the satisfiability of propositional formulas. This procedure can be seen as a backtracking search procedure that makes assumptions about the truth values of the propositional atoms in a formula and that then derives new truth values from these assumptions in order to prune the search space.

While the extended rules of this work are novel, there are some analogous constructions in the literature. The choice rule can be seen as a generalization of the disjunctive rule of the possible model semantics [54]. The disjunctive rule of disjunctive logic programs [51] also resembles the choice rule, but the semantics is in this case different. The stable models of a disjunctive program are subset minimal while the stable models of a logic program are grounded, i.e., atoms can not justify their own inclusion. If a program contains choice rules, then a grounded model is not necessarily subset minimal.

Since the optimize statements lexicographically order the stable models, they can be used for prioritized reasoning. Priorities have previously been used to lexicographically order rules [53, 5] and to order atoms [55].

1.2 Applications

The program *smodels* has been used in several contexts. In the field of verification of distributed systems, it has successfully been applied to deadlock and reachability problems [23, 25, 24]. In addition, it has been used to lessen the impact of the state space explosion inherent in the reachability analysis of place/transition nets [66]. It has also been used for model checking in a system for computing alternating fixed points [33]. In the area of product configuration it has provided a base for a rule-based language with favorable computational properties [62]. An application of *smodels* in the planning domain has resulted in performance comparable to and sometimes better than that of other efficient general purpose planners [13]. Lastly, it has served as an implementation base for dynamic constraint satisfaction problems [61] and for logic programs with weight constraint rules [47].

1.3 A Brief History

My work on *smodels* began in February 1995. For my Master's Thesis [57] I implemented a decision procedure for the stable model semantics. The procedure had arisen during the work of my instructor, Ilkka Niemelä, on autoepistemic logic [40], and it was directly applicable to both default logic [41] and the stable model semantics [43]. The algorithm was needlessly complex and I was able to simplify it while at the same time making it prune the search space more. In response to this, Ilkka devised a way to strengthen the algorithm [44, 45] by employing the Fitting semantics [19]. I further improved the procedure [58] by making systematic use of backward chaining [7] and lookahead. Different types of rules were then introduced [59] and more optimizations were done.

1.4 Contributions

The key contributions of this work are: the new rule types and the extension of the stable model semantics, an algorithm for computing the stable models of sets of extended rules, and its generalization to compute specific stable models. An important contribution is the derivation of a heuristic. The heuristic was not found by experimentation, instead it was derived using the principle that the search space should be minimized.

There are also some contributions that concern the efficient implementation of the *smodels* algorithm. Naturally, one must be familiar with the algorithm to understand the concepts involved. The first contribution is an implementation of lookahead that safely avoids testing every literal for failure. The implementation makes the use of lookahead feasible. The second contribution decreases the amount of work needed when pruning the search space and it consists of the use of source pointers and strongly connected components in the computation of the upper closure. The last contribution is an improvement of the way *smodels* backtracks and is a type of backjumping.

We also note that the central pruning function, the *expand* function, is an efficient implementation of the well-founded semantics [65, 57].

1.5 Outline of the Work

The stable model semantics and its extension to choice, cardinality, and weight rules are presented in Section 2. In Section 3 we present an implementation of the semantics, the *smodels* procedure, that computes stable models of logic programs. We also show how the procedure can be extended to find specific stable models such as the lexicographically smallest one. The implementation is described in greater detail in Section 4, and the complexity of the reasoning tasks are discussed in Section 5. In Section 6 we compare *smodels* with the Davis-Putnam procedure for testing satisfiability of propositional formulas and with some algorithms for computing stable models, and in Section 7 we compare it with some propositional satisfiability checkers by running experiments on random satisfiability problems, on pigeon-hole problems, and on Hamiltonian cycle problems. The conclusions follow in Section 8. Finally, we state some useful properties of monotone functions in Appendix A.

2 The Stable Model Semantics

In this section we introduce the stable model semantics for normal logic programs. We extend the semantics to cover three new types of rules: choice rules that encode subsets of a set, cardinality rules that enforce cardinality limits on the subsets, and weight rules that express inequalities over weighted linear sums. In addition, we present the compute statement, which is used when one searches for models that contain certain atoms, and the optimize statements, which are used when one searches for models of optimal weight. We begin by defining the stable model semantics.

2.1 The Stable Model Semantics

Let *Atoms* be a set of primitive propositions, or atoms. A logic program is a set of rules of the form

$$h \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m,$$

where $h, a_1, \ldots, a_n, b_1, \ldots, b_m$ are members of *Atoms*. The atom h is the head of the rule and the other atoms in the rule make up the body. We call the expression *not* b a not-atom — atoms and not-atoms are referred to as literals.

Example 2.1. Think of a stable model as the set of atoms that are true, any other atoms are false. The program

$$\begin{aligned} a &\leftarrow b \\ b &\leftarrow c, not\ d \\ d &\leftarrow not\ b \\ c &\leftarrow a \end{aligned}$$

has, perhaps surprisingly, only one stable model: the set $\{d\}$.

The stable model semantics for a logic program P is defined as follows [22]. The reduct P^A of P with respect to the set of atoms A is obtained by

1. deleting each rule in P that has a not-atom *not* x in its body such that $x \in A$, and by

2. deleting all not-atoms in the remaining rules.

The deductive closure of P^A is the smallest set of atoms that is closed under P^A when the rules in P^A are seen as inference rules.

Definition 2.2. A set of atoms S is a stable model of P if and only if S is the deductive closure of P^S.

Example 2.1 (continued). The set $\{a, b, c\}$ is not a stable model of the program P

$$\begin{aligned} &a \leftarrow b \\ &b \leftarrow c, not\ d \\ &d \leftarrow not\ b \\ &c \leftarrow a \end{aligned}$$

since the deductive closure of

$$P^{\{a,b,c\}} = \{a \leftarrow b,\ b \leftarrow c,\ c \leftarrow a\}$$

is the empty set.

In order to facilitate the definition of more general forms of rules, we introduce an equivalent characterization of the stable model semantics.

Proposition 2.3. *We say that $f_P : 2^{Atoms} \rightarrow 2^{Atoms}$ is a closure of the program P if*

$$\begin{aligned} f_P(S) = \{h \mid h \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m \in P,& \\ a_1, \ldots, a_n \in f_P(S),\ b_1, \ldots, b_m \notin S\}&. \end{aligned}$$

Let

$$g_P(S) = \bigcap \{f_P(S) \mid f_P : 2^{Atoms} \rightarrow 2^{Atoms} \textit{ is a closure}\}.$$

Then, S is a stable model of P if and only if

$$S = g_P(S).$$

Proof. Note that the deductive closure of the reduct P^S is a closure, and note that for every f_P that is a closure, the deductive closure of P^S is a subset of $f_P(S)$. □

A stable model is therefore a model that follows from the complement of itself by means of the smallest possible closure. One speaks of stable models as grounded models, as atoms in the models do not imply themselves. Atoms are not true without grounds.

Remark. The alternative definition of the stable model semantics is basically a variation of the definition of the semantics of default logic given by Reiter [52]. The function g_P just computes the least fixed point of the monotonic operator

$$\begin{aligned} f_P^S(A) = \{h \mid h \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m \in P,& \\ a_1, \ldots, a_n \in A,\ b_1, \ldots, b_m \notin S\}&. \end{aligned}$$

Example 2.4. The problem of deciding whether a program has a stable model is NP-complete [36]. Hence, one can encode satisfiability problems as logic programs. The satisfying assignments of the formula

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

correspond to the stable models of the program

$$\begin{array}{rlrl}
a & \leftarrow \mathit{not}\ a' & a' & \leftarrow \mathit{not}\ a \\
b & \leftarrow \mathit{not}\ b' & b' & \leftarrow \mathit{not}\ b \\
c & \leftarrow \mathit{not}\ c' & c' & \leftarrow \mathit{not}\ c \\
d & \leftarrow \mathit{not}\ d' & d' & \leftarrow \mathit{not}\ d \\
\mathit{false} & \leftarrow \mathit{not}\ a, \mathit{not}\ b, c & \mathit{false} & \leftarrow a, \mathit{not}\ b, d \\
\mathit{false} & \leftarrow b, \mathit{not}\ c, \mathit{not}\ d & \mathit{contradiction} & \leftarrow \mathit{not}\ \mathit{contradiction}, \mathit{false}
\end{array}$$

In this case, there are ten satisfying assignments. The encoding works by rejecting models that contain *false* and by stating that an unsatisfied clause implies *false*. The intersection of a stable model and the atoms in the formula is a satisfying assignment.

2.2 More Expressive Rules

Consider the problem of ensuring that at most k of the atoms $a_1, \ldots, a_n$ are included in every stable model of a program. A naive programmatic solution would consist of the rules

$$\{\mathit{false} \leftarrow a_{i_1}, \ldots, a_{i_{k+1}} \mid 1 \le i_1 < \cdots < i_{k+1} \le n\}$$

of which there are $\binom{n}{k+1}$ and of the stable models that do not include *false*. There is also a quadratic solution, or to be more precise, a solution that needs on the order of nk rules.

Let the atom $l(a_i, j)$ represent the fact that at least j of the atoms in $\{a_i, \ldots, a_n\}$ are in a particular stable model. Then, the demand that at most k of $a_1, \ldots, a_n$ are in a model can be handled by the constraint $\mathit{false} \leftarrow l(a_1, k+1)$. The definition of $l(a_i, j)$ is given by the program

$$\begin{aligned}
l(a_i, j) &\leftarrow l(a_{i+1}, j) \\
l(a_i, j+1) &\leftarrow a_i, l(a_{i+1}, j) \\
l(a_i, 1) &\leftarrow a_i
\end{aligned}$$

and we need about nk instances of it to cover all values of i and j.

There is a need for a representation that is more compact. We propose to introduce three types of rules that have sufficient expressiveness to compactly describe problems such as the one above. The first type is the cardinality rule, which is of the form

$$h \leftarrow k\,\{a_1, \ldots, a_n, \mathit{not}\ b_1, \ldots, \mathit{not}\ b_m\}.$$

It is interpreted as follows: if at least k literals in the set

$$\{a_1, \ldots, a_n, \mathit{not}\ b_1, \ldots, \mathit{not}\ b_m\}$$

are satisfied by a stable model, then the atom h must be in the stable model. An atom a is satisfied by a stable model S if $a \in S$ and a not-atom $\mathit{not}\ b$ is satisfied by S if $b \notin S$.

Example 2.5. Once is a mistake, twice is a habit. Or,

$$\mathit{habit} \leftarrow 2\,\{\mathit{mistake}_1, \ldots, \mathit{mistake}_n\}.$$

The second type is the choice rule,

$$\{h_1, \ldots, h_k\} \leftarrow a_1, \ldots, a_n, \mathit{not}\ b_1, \ldots, \mathit{not}\ b_m,$$

which implements a nondeterministic choice over the atoms in $\{h_1, \ldots, h_k\}$ whenever the literals in the body are satisfied by a stable model. That is, the rule gives ground for the inclusion of any number of atoms in its head. For example, the rule

$$\{h_1, \ldots, h_k\} \leftarrow$$

can be encoded by the program

$$\begin{aligned} h_i &\leftarrow \mathit{not}\ h_i' \\ h_i' &\leftarrow \mathit{not}\ h_i, \qquad i = 1, \ldots, k. \end{aligned}$$

Example 2.6. Each stable model of the program

$$\begin{aligned} \{a_1, a_2, a_3, a_4\} &\leftarrow \\ \mathit{false} &\leftarrow \mathit{not}\ a_1, \mathit{not}\ a_2, \mathit{not}\ a_3, \mathit{not}\ a_4 \end{aligned}$$

that does not contain *false* includes at least one atom.

Finally, the third type of rule, the weight rule, is of the form

$$h \leftarrow \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \mathit{not}\ b_1 = w_{b_1}, \ldots, \mathit{not}\ b_m = w_{b_m}\} \geq w,$$

and its head h will be in a stable model S if

$$\sum_{a_i \in S} w_{a_i} + \sum_{b_i \notin S} w_{b_i} \geq w.$$

Here, the weights are real numbers.

We can restrict ourselves to positive weights, since a rule with negative weights can be translated into a rule with only positive weights. Namely,

$$h \leftarrow \{a = w_a, b = -w_b, not\ c = w_c, not\ d = -w_d\} \geq w, \quad w_a, w_b, w_c, w_d > 0$$

is equivalent to

$$\begin{aligned} h \leftarrow \{&a = w_a, b = -w_b, b = w_b, not\ b = w_b, \\ &not\ c = w_c, not\ d = -w_d, not\ d = w_d, d = w_d\} \geq w + w_b + w_d \end{aligned}$$

which is equivalent to

$$h \leftarrow \{a = w_a, not\ b = w_b, not\ c = w_c, d = w_d\} \geq w + w_b + w_d.$$

Furthermore, the rule

$$h \leftarrow \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, not\ b_1 = w_{b_1}, \ldots, not\ b_m = w_{b_m}\} \leq w$$

can be transformed into

$$\begin{aligned} h \leftarrow \{&a_1 = -w_{a_1}, \ldots, a_n = -w_{a_n}, \\ &not\ b_1 = -w_{b_1}, \ldots, not\ b_m = -w_{b_m}\} \geq -w. \end{aligned}$$

Example 2.7. The stable models of the program

$$\begin{aligned} \{a_1, \ldots, a_n\} &\leftarrow \\ \mathit{false} &\leftarrow \{a_1 = w_1, \ldots, a_n = w_n\} \geq w \\ \mathit{true} &\leftarrow \{a_1 = v_1, \ldots, a_n = v_n\} \geq v \end{aligned}$$

containing the atom *true* but not the atom *false* correspond to the ways one can pack a subset of $a_1, \ldots, a_n$ in a bin such that the total weight is less than w and the total value is at least v. The weights and values of the items are given by respectively $w_1, \ldots, w_n$ and $v_1, \ldots, v_n$.

Often only stable models including or excluding certain atoms are of interest. We can guarantee that every stable model of a program includes a specific atom *true* by adding a rule

$$\mathit{true} \leftarrow not\ \mathit{true}$$

to the program. Similarly, we can exclude an atom *false* from all stable models of a program with the help of the rule

$$\mathit{contradiction} \leftarrow not\ \mathit{contradiction}, \mathit{false}$$

provided that the atom *contradiction* does not appear anywhere else in the program. Despite the simplicity of the two previous rules, we will use compute statements of the form

$$\mathit{compute}\ \{a_1, \ldots, a_n, \mathit{not}\ b_1, \ldots, \mathit{not}\ b_m\}$$

to state that we only accept stable models that contain $a_1, \ldots, a_n$ but not $b_1, \ldots, b_m$.

Example 2.8. The satisfying assignments of the formula

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

correspond to the stable models of the program

$$\begin{aligned} &\{a, b, c, d\} \leftarrow \\ &\quad \mathit{false} \leftarrow \mathit{not}\ a, \mathit{not}\ b, c \\ &\quad \mathit{false} \leftarrow a, \mathit{not}\ b, d \\ &\quad \mathit{false} \leftarrow b, \mathit{not}\ c, \mathit{not}\ d \\ &\mathit{compute}\ \{\mathit{not}\ \mathit{false}\} \end{aligned}$$

Sometimes one wants to find the stable model with the least number of atoms. Other times the atoms have been given priorities and one wants to find the stable model with the highest priority. In order to be able to express such preferences, we introduce two optimize statements: the minimize statement and its dual the maximize statement. The minimize statement

$$\mathit{minimize}\ \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \mathit{not}\ b_1 = w_{b_1}, \ldots, \mathit{not}\ b_m = w_{b_m}\}$$

declares that we want to find a stable model S with the smallest weight

$$\sum_{a_i \in S} w_{a_i} + \sum_{b_i \notin S} w_{b_i}.$$

If there are several minimize statements, then we order the stable models lexicographically according to the weights of the statements. In this case, the first statement is the most significant. The maximize statement

$$\mathit{maximize}\ \{a = w_a, \mathit{not}\ b = w_b\}$$

is just another way to write

$$\mathit{minimize}\ \{a = -w_a, \mathit{not}\ b = -w_b\}$$

or

$$\mathit{minimize}\ \{a = k - w_a, \mathit{not}\ b = k - w_b\},$$

if one wants to avoid negative weights.

Example 2.9. The lexicographically first stable model of the program

$$
\begin{aligned}
a &\leftarrow \textit{not } b \\
b &\leftarrow \textit{not } a \\
\textit{minimize}\, &\{a = 1\} \\
\textit{minimize}\, &\{b = 1\}
\end{aligned}
$$

is $\{b\}$.

Example 2.10. Multiple minimize statements can be written as one:

$$
\begin{aligned}
&\textit{minimize}\,\{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}\} \\
&\textit{minimize}\,\{b_1 = w_{b_1}, \ldots, b_m = w_{b_m}\}
\end{aligned}
$$

is equivalent to

$$\textit{minimize}\,\{a_1 = 2^k w_{a_1}, \ldots, a_n = 2^k w_{a_n}, b_1 = w_{b_1}, \ldots, b_m = w_{b_m}\}$$

if $2^k > \sum w_{b_i}$.

We now turn to the formal definition of the stable model semantics for the extended syntax.

Definition 2.11. A basic rule r is of the form

$$h \leftarrow a_1, \ldots, a_n, \textit{not } b_1, \ldots, \textit{not } b_m$$

and is interpreted by the function $f_r : 2^{\textit{Atoms}} \times 2^{\textit{Atoms}} \to 2^{\textit{Atoms}}$ as follows.

$$f_r(S, C) = \{h \mid a_1, \ldots, a_n \in C,\ b_1, \ldots, b_m \notin S\}.$$

The intuition behind the function f_r is that it produces the result of a deductive step when applied to a candidate stable model S and its consequences C. In other words, if C is a deductive closure, then $f_r(S, C) \subseteq C$. By this reasoning, the definition of f_r for the new rules follows straightforwardly.

Definition 2.12. A cardinality rule r is of the form

$$h \leftarrow k\,\{a_1, \ldots, a_n, \textit{not } b_1, \ldots, \textit{not } b_m\}$$

and is interpreted by

$$f_r(S, C) = \big\{h \,\big|\, |\{a_1, \ldots, a_n\} \cap C| + |\{b_1, \ldots, b_m\} - S| \geq k\big\}.$$

A choice rule r is of the form

$$\{h_1, \ldots, h_k\} \leftarrow a_1, \ldots, a_n, \textit{not } b_1, \ldots, \textit{not } b_m$$

and is interpreted by

$$f_r(S,C) = \big\{h \,\big|\, h \in \{h_1,\ldots,h_k\} \cap S, a_1,\ldots,a_n \in C,\ b_1,\ldots,b_m \notin S\big\}.$$

Finally, a weight rule r is of the form

$$h \leftarrow \{a_1 = w_{a_1},\ldots,a_n = w_{a_n}, \mathit{not}\ b_1 = w_{b_1},\ldots,\mathit{not}\ b_m = w_{b_m}\} \geq w,$$

for $w_{a_i}, w_{b_i} \geq 0$, and is interpreted by

$$f_r(S,C) = \{h \mid \sum_{a_i \in C} w_{a_i} + \sum_{b_i \notin S} w_{b_i} \geq w\}.$$

Example 2.13. Let $S = \{a\}$ be a model of a logic program, one of whose rules is r:

$$h \leftarrow \{a = 1, b = 2, \mathit{not}\ c = 3\} \geq 4.$$

Since we expect a stable model to be a deductive closure, we set $C = \{a\}$ and compute $f_r(S,C) = \{h\}$. As $h \notin C$, we note that S can not be a stable model.

Definition 2.14. Let P be a set of rules. As before we say that $f_P : 2^{Atoms} \to 2^{Atoms}$ is a closure if

$$f_P(S) = \bigcup_{r \in P} f_r\big(S, f_P(S)\big),$$

and we define

$$g_P(S) = \bigcap \{f_P(S) \mid f_P : 2^{Atoms} \to 2^{Atoms} \text{ is a closure}\}.$$

Then, S is a stable model of the program P if and only if

$$S = g_P(S).$$

Lemma 2.15. *Let P be a logic program and let S be a set of atoms. Define*

$$f_P^S(A) = \bigcup_{r \in P} f_r(S,A).$$

Then, the least fixed point $\mathit{lfp}(f_P^S)$ of f_P^S is equal to $g_P(S)$.

Proof. Note that the operator $f_P^S(A)$ is monotonic and that it has a fixed point $f_P(S)$ for any closure f_P. Hence, $\mathit{lfp}(f_P^S) \subseteq f_P(S)$ for all f_P and therefore $\mathit{lfp}(f_P^S) \subseteq g_P(S)$. Since $\mathit{lfp}(f_P^S)$, taken as a function of S, defines a closure, $g_P(S) \subseteq \mathit{lfp}(f_P^S)$. Thus, $g_P(S) = \mathit{lfp}(f_P^S)$ follows. □

Example 2.16. Let P be the logic program

$$\begin{aligned} \{a, b, c\} &\leftarrow \\ \mathit{true} &\leftarrow 2\,\{a, b, c\} \\ \mathit{compute}\ &\{\mathit{true}\} \\ \mathit{minimize}\ &\{a = 1, b = 2\}. \end{aligned}$$

The stable models of the two rules in P are $\emptyset$, $\{a\}$, $\{b\}$, $\{c\}$, $\{a, b, \mathit{true}\}$, $\{a, c, \mathit{true}\}$, $\{b, c, \mathit{true}\}$, and $\{a, b, c, \mathit{true}\}$, as we can easily check using Lemma 2.15. For instance, $f_P^{\{a\}}(\emptyset) = \{a\} = f_P^{\{a\}}(\{a\})$. Because of the compute statement we only want stable models that contain the atom *true*, namely the models $\{a, b, \mathit{true}\}$, $\{a, c, \mathit{true}\}$, $\{b, c, \mathit{true}\}$, and $\{a, b, c, \mathit{true}\}$. But since there is also a minimize statement we are really only interested in the smallest of these: $\{a, c, \mathit{true}\}$.

The stable models of a logic program that do not contain choice rules are subset minimal, whereas the stable models of a program with choice rules can be subsets of each other. It is therefore not possible to translate a program containing choice rules into one with no choice rules without introducing new atoms. In this sense, the introduction of choice rules makes the stable model semantics more expressive.

Proposition 2.17. *Let P be a logic program that does not contain any choice rules. If S and S' are stable models of P, then $S \subseteq S'$ implies $S = S'$.*

Proof. Let P be a logic program that does not contain any choice rules and let $S \subseteq S'$ be two stable models of P. As

$$f_P^{S'}(S) = \bigcup_{r \in P} f_r(S', S) \subseteq \bigcup_{r \in P} f_r(S, S) = f_P^S(S) = S$$

since $f_r(S', S)$ is anti-monotonic in its first argument, $S' = \mathit{lfp}(f_P^{S'}) \subseteq S$. $\square$

Corollary 2.18. *Let P be the set of all logic programs and let P' be the set of all logic programs that do not contain choice rules. Then, there is no mapping from P to P' that preserves stable models.*

Example 2.19. We can translate the disjunctive rules of the possible model semantics [54] into choice rules and basic rules such that possible models correspond to stable models. Namely, change every disjunctive rule

$$a_1 \vee \cdots \vee a_k \leftarrow b_1 \wedge \cdots \wedge b_n \wedge \mathit{not}\ c_1 \wedge \cdots \wedge \mathit{not}\ c_m$$

into a choice rule

$$\{a_1, \ldots, a_k\} \leftarrow b_1, \ldots, b_n, \mathit{not}\ c_1, \ldots, \mathit{not}\ c_m$$

and a basic rule

$$false \leftarrow b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_m, not\ a_1, \ldots, not\ a_k,$$

and add the compute statement

$$compute\ \{not\ false\}.$$

3 The Algorithm

Having defined the stable model semantics, we want to find a method that can be used to compute stable models. We take the obvious approach and enumerate all subsets of the atoms in a program and test each subset for stability. During the exhaustive search we make use of the properties of the stable model semantics to prune away large numbers of subsets. The procedure is put forth as a backtracking search algorithm. Since we want to be able to handle large programs, we avoid constructs that require more than a linear amount of space.

At the heart of the algorithm is a set of literals, which we name A, that represents a set of stable models. The atoms in the set A are members of these models and the not-atoms in the set are atoms that are not in the stable models. It follows that if there is an atom in A that also appears as a not-atom in A, then the set of models that A represents is empty. Hence, our algorithm begins with A as the empty set. It adds atoms and not-atoms to A and checks whether the resulting set corresponds to at least one stable model. If it does not, then it backtracks by removing atoms and not-atoms from A and by changing atoms into not-atoms and vice versa. The relationship between the partial model A and one of its stable models is exemplified in Figure 1.

The search space consisting of all possible configurations of A is pruned by deducing additions to A from the program using the properties of the stable model semantics. For example, if the rule

$$a \leftarrow b, not\ c$$

is in a program and $b, not\ c \in A$, then we deduce that every stable model of the program that contains b but not c must contain a. Consequently, we add a to A. Expanding A can lead to situations in which an atom in A is also a not-atom in A. If such a conflict takes place, then the algorithm backtracks.

We can prune the search space some more by wisely choosing what literals we add to A. A good heuristic helps, but choosing literals that immediately give rise to conflicts avoids a lot of backtracking. We find these literals by looking ahead: for each literal not in A we temporarily add the literal to A, expand it, and check for conflicts.

We explain the algorithm in greater detail in the rest of the section.

3.1 The Decision Procedure

For an atom a, let $not\,(a) = not\ a$, and for a not-atom $not\ a$, let

$$not\,(not\ a) = a.$$

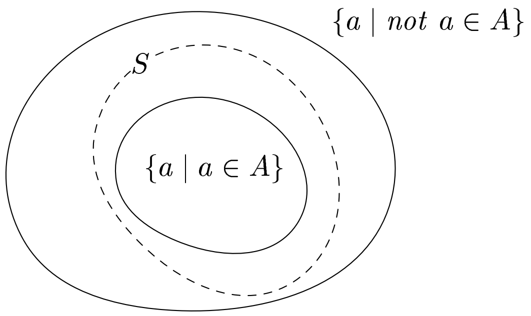


Figure 1: The relation between the set A and one of its stable models S

For a set of literals A, define

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

Let $A^+ = \{a \in Atoms \mid a \in A\}$ and let $A^- = \{a \in Atoms \mid not\ a \in A\}$. Define $Atoms(A) = A^+ \cup A^-$, and for a program P, define $Atoms(P) = Atoms(L)$, where L is the set of literals that appear in the program.

A set of literals A is said to cover a set of atoms B if $B \subseteq Atoms(A)$, and B is said to agree with A if

$$A^+ \subseteq B \qquad \text{and} \qquad A^- \subseteq Atoms - B.$$

Algorithm 1 displays a decision procedure for the stable model semantics. The function *smodels*(P, A) returns true whenever there is a stable model of P agreeing with the set of literals A. In fact, the function computes a stable model and, as will become apparent, it can be modified to compute all stable models of P that agree with A.

The decision procedure calls four functions: *expand*(P, A), *conflict*(P, A), *lookahead*(P, A), and *heuristic*(P, A). The function *expand*(P, A) expands the set A using the functions *Atleast*(P, A) and *Atmost*(P, A), the function *conflict*(P, A) discovers conflicts, and the function *heuristic*(P, A) computes heuristically good literals that can be included in A. For now we let the function *lookahead*(P, A) return A.

Let $A' = expand(P, A)$. We assume that

E1 $A \subseteq A'$ and that

E2 every stable model of P that agrees with A also agrees with A'.

Moreover, we assume that the function *conflict*(P, A) satisfies the two conditions

C1 if A covers $Atoms(P)$ and there is no stable model that agrees with A, then *conflict*(P, A) returns true, and

```
function smodels(P, A)
  A := expand(P, A)
  A := lookahead(P, A)
  if conflict(P, A) then
    return false
  else if A covers Atoms(P) then
    return true {A⁺ is a stable model}
  else
    x := heuristic(P, A)
    if smodels(P, A ∪ {x}) then
      return true
    else
      return smodels(P, A ∪ {not(x)})
    end if
  end if.

function expand(P, A)
  repeat
    A' := A
    A := Atleast(P, A)
    A := A ∪ {not x | x ∈ Atoms(P) and x ∉ Atmost(P, A)}
  until A = A'
  return A.

function conflict(P, A)
  {Precondition: A = expand(P, A)}
  if A⁺ ∩ A⁻ ≠ ∅ then
    return true
  else
    return false
  end if.

function lookahead(P, A)
  return A.
```

Algorithm 1: A decision procedure for the stable model semantics

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

In addition, we expect *heuristic*(P, A) to return a literal not covered by A.

Theorem 3.1. *Let P be a set of rules and let A be a set of literals. Then, there is a stable model of P agreeing with A if and only if smodels(P, A) returns true.*

Proof. Let $nc(P, A) = Atoms(P) - Atoms(A)$ be the atoms not covered by A. We prove the claim by induction on the size of $nc(P, A)$.

Assume that the set $nc(P, A) = \emptyset$. Then, $A' = expand(P, A)$ covers $Atoms(P)$ by E1 and $smodels(P, A)$ returns true if and only if $conflict(P, A')$ returns false. By E2, C1, and C2, this happens precisely when there is a stable model of P agreeing with A.

Assume $nc(P, A) \neq \emptyset$. If $conflict(P, A')$ returns true, then $smodels(P, A)$ returns false and by E2 and C2 there is no stable model agreeing with A. On the other hand, if $conflict(P, A')$ returns false and A' covers $Atoms(P)$, then $smodels(P, A)$ returns true and by E2 and C1 there is a stable model that agrees with A. Otherwise, induction together with E1 and E2 show that $smodels(P, A' \cup \{x\})$ or $smodels\big(P, A' \cup \{not\ (x)\}\big)$ returns true if and only if there is a stable model agreeing with A. $\square$

Let S be a stable model of P agreeing with the set of literals A. Then, $f_r(S, S) \subseteq S$ for $r \in P$, and we make the following observations. Let

$$min_r(A) = \bigcap_{\substack{A^+ \subseteq C \\ A^- \cap C = \emptyset}} f_r(C, C)$$

be the inevitable consequences of A, and let

$$max_r(A) = \bigcup_{\substack{A^+ \subseteq C \\ A^- \cap C = \emptyset}} f_r(C, C)$$

be the possible consequences of A. Then,

1. if $r \in P$, then S agrees with $min_r(A)$,

2. if there is an atom a such that for all $r \in P$, $a \notin max_r(A)$, then S agrees with $\{not\ a\}$,

3. if the atom $a \in A$, if there is only one $r \in P$ for which $a \in max_r(A)$, and if there exists a literal x such that $a \notin max_r(A \cup \{x\})$, then S agrees with $\{not\ (x)\}$, and

4. if $not\ a \in A$ and if there exists a literal x such that for some $r \in P$, $a \in min_r(A \cup \{x\})$, then S agrees with $\{not\ (x)\}$.

Note that if $a, not\ a \in A$, then $min_r(A) = Atoms$ and $max_r(A) = \emptyset$.

Proposition 3.2. *The claims 1–4 hold.*

Proof. Recall from Lemma 2.15 that

$$f_P^S(A) = \bigcup_{r \in P} f_r(S, A)$$

is monotonic and that its least fixed point is equal to $g_p(S)$. Hence, if S is a stable model of P, then for any $r \in P$, $f_r(S, S) \subseteq S$.

Let the stable model S agree with the set A. As $min_r(A) \subseteq f_r(S, S)$, the first claim follows. Notice that

$$S = \bigcup_{r \in P} f_r(S, S).$$

If for all $r \in P$, $a \notin max_r(A)$, then for all $r \in P$, $a \notin f_r(S, S)$ and consequently $a \notin S$. This proves the second claim.

If $a \in A$ and there is only one $r \in P$ for which $a \in max_r(A)$, then $a \in f_r(S, S) \subseteq S$. If $a \notin max_r(A \cup \{x\})$ for some literal x and if S agrees with $\{x\}$, then $a \notin f_r(S, S)$ which is a contradiction. Thus, the third claim holds.

Finally, if $not\ a \in A$ and there is a literal x such that for some $r \in P$, $a \in min_r(A \cup \{x\})$, then by the first claim $a \in S$ which is a contradiction. Therefore, also the fourth claim is true. $\square$

The four statements help us deduce additional literals that are in agreement with S. Define $Atleast(P, A)$ as the smallest set of literals containing A that can not be enlarged using 1–4 above, i.e., let $Atleast(P, A)$ be the least fixed point of the operator

$$\begin{aligned} f(B) = A \cup B & \\ & \cup \{a \in min_r(B) \mid a \in Atoms(P) \text{ and } r \in P\} \\ & \cup \{not\ a \mid a \in Atoms(P) \text{ and for all } r \in P,\ a \notin max_r(B)\} \\ & \cup \big\{not\ (x) \,\big|\, \text{there exists } a \in B \text{ such that } a \in max_r(B) \\ & \qquad\qquad \text{for only one } r \in P \text{ and } a \notin max_r(B \cup \{x\})\big\} \\ & \cup \big\{not\ (x) \,\big|\, \text{there exists } not\ a \in B \text{ and } r \in P \text{ such that} \\ & \qquad\qquad a \in min_r(B \cup \{x\})\big\}. \end{aligned}$$

Example 3.3. Let P be the program

$$\begin{aligned} a &\leftarrow b, not\ c \\ d &\leftarrow not\ a \\ e &\leftarrow not\ b \end{aligned}$$

We will compute $A = Atleast(P, \{d\})$. Since c does not appear in the head of any rule in P, $not\ c \in A$ by claim 2. As $d \in A$, $not\ a \in A$ by claim 3. It follows that $not\ b \in A$ by 4. Finally, $e \in A$ by 1. Hence, $Atleast(P, \{d\}) = \{not\ a, not\ b, not\ c, d, e\}$.

Lemma 3.4. *The function $Atleast(P, A)$ is monotonic in its second argument.*

Proof. Observe that the function $min_r(B)$ is monotonic and that the function $max_r(B)$ is anti-monotonic. Hence,

$$\{a \in min_r(B) \mid r \in P\},$$
$$\{not\ a \mid a \in Atoms(P) \text{ and for all } r \in P,\ a \notin max_r(B)\},$$

and

$$\big\{not\ (x) \,\big|\, \text{there exists } not\ a \in B \text{ and } r \in P \text{ such that } a \in min_r(B \cup \{x\})\big\}$$

are monotonic with respect to B. Assume that there exists $a \in B$ such that $a \in max_r(B)$ for only one $r \in P$ and $a \notin max_r(B \cup \{x\})$. If $B \subseteq B'$ and $a \notin max_r(B')$, then

$$not\ a \in \{not\ a' \mid a' \in Atoms(P) \text{ and for all } r \in P,\ a' \notin max_r(B')\} \subseteq f(B').$$

Consequently, both $a, not\ a \in f(B')$ and therefore

$$min_r\big(f(B')\big) = Atoms,$$

and

$$max_r\big(f(B')\big) = \emptyset.$$

It follows that $f\big(f(B')\big) = Atoms(P) \cup not\big(Atoms(P)\big)$. Thus, f^2 is monotonic and has a least fixed point. Finally, notice that f has the same fixed points as f^2. By the definition of f, $B \subseteq f(B)$. Thus, $f^2(B) = B$ implies

$$B \subseteq f(B) \subseteq f\big(f(B)\big) = B.$$

□

We conclude,

Proposition 3.5. *If the stable model S of P agrees with A, then S agrees with $Atleast(P, A)$.*

Furthermore, we can bound the stable models from above.

Definition 3.6. For a choice rule r of the form

$$\{h_1, \ldots, h_k\} \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m,$$

let

$$f'_r(S, C) = \bigl\{h \in \{h_1, \ldots, h_k\} \bigm| a_1, \ldots, a_n \in C,\ b_1, \ldots, b_m \notin S\bigr\},$$

and for any other type of rule, let $f'_r(S, C) = f_r(S, C)$. Let P be a set of rules and let A be a set of literals. Define $Atmost(P, A)$ as the least fixed point of

$$f'(B) = \bigcup_{r \in P} f'_r(A^+, B - A^-) - A^-.$$

Proposition 3.7. *Let S be a stable model of P that agrees with A. Then, $S \subseteq Atmost(P, A)$.*

Proof. Note that $f'_r(S, C)$ is anti-monotonic in its first argument, i.e., $S \subseteq S'$ implies $f'_r(S', C) \subseteq f'_r(S, C)$, and monotonic in its second argument. Fix a program P, a stable model S of P, and a set of literals A such that S agrees with A. Define

$$f^S_P(B) = \bigcup_{r \in P} f_r(S, B)$$

and

$$f'(B) = \bigcup_{r \in P} f'_r(A^+, B - A^-) - A^-.$$

Let L be the least fixed point of f'. Since S agrees with A,

$$f_r(S, S \cap L) \subseteq f'_r(A^+, S \cap L - A^-) - A^-,$$

and $f^S_P(S \cap L) \subseteq f'(S \cap L) \subseteq L$. Hence, for varying B, the least fixed point of $f^S_P(S \cap B)$, which is equal to the least fixed point of f^S_P, is a subset of L by Lemma A.1. In other words, $S \subseteq L$. □

It follows that *expand*(P, A) satisfies the conditions E1 and E2. The function *conflict*(P, A) obviously fulfills C2, and the next proposition shows that also C1 holds.

Proposition 3.8. *If $A =$ expand(P, A) covers the set Atoms(P) and $A^+ \cap A^- = \emptyset$, then A^+ is a stable model of P.*

Proof. Assume that $A = expand(P, A)$ covers $Atoms(P)$ and that $A^+ \cap A^- = \emptyset$. Then, $A^+ = Atmost(P, A)$ and

$$f_r(A^+, B) = f'_r(A^+, B) - A^- = f'_r(A^+, B - A^-) - A^-$$

for every $B \subseteq A^+$, since $f_r(A^+, B) \subseteq f_r(A^+, A^+) = min_r(A) \subseteq A^+$. Thus, A^+ is the least fixed point of

$$f_P^{A^+}(B) = \bigcup_{r \in P} f_r(A^+, B),$$

from which we infer, by Lemma 2.15, that A^+ is a stable model of P. $\square$

Example 3.9. Let P be the program

$$a \leftarrow not\ b$$
$$c \leftarrow a$$

Then, $Atmost(P, \emptyset) = \{a, c\}$ as $f'_{a \leftarrow not\ b}(\emptyset, \emptyset) = \{a\}$ and $f'_{c \leftarrow a}(\emptyset, \{a\}) = \{c\}$. But, $Atmost(P, \{not\ a\}) = \emptyset$.

Example 3.10. Let P be the program

$$a \leftarrow not\ b$$
$$b \leftarrow not\ a, c$$
$$c \leftarrow c$$

By computing $A = expand(P, \emptyset)$ we see that this program has only one stable model. Namely, $Atleast(P, \emptyset) = \emptyset$ and $Atmost(P, \emptyset) = \{a\}$. Hence, $not\ b, not\ c \in A$. Since $Atleast(P, \{not\ b, not\ c\}) = \{a, not\ b, not\ c\}$ and since $Atmost(P, \{a, not\ b, not\ c\}) = \{a\}$, $A = \{a, not\ b, not\ c\}$. Therefore, A covers $Atoms(P)$ and $\{a\}$ is the only stable model of P.

Example 3.11. The *smodels* algorithm traverses a search space consisting of sets of literals. One can visualize the path the algorithm takes as a tree whose nodes are the sets and whose edges are labeled with the heuristic choices that have been made during the computation. By convention we assume that the algorithm goes down the left branch first. Hence, the whole computation corresponds to an in-order traversal of the tree. For example, in Figure 2 the algorithm first tries the atom a and experiences a conflict. It then changes to $not\ a$ and tries $not\ b$ from which it continues through some unspecified choices that all lead to a conflict. After this b is asserted and the choice of the atom c ends in a stable model.

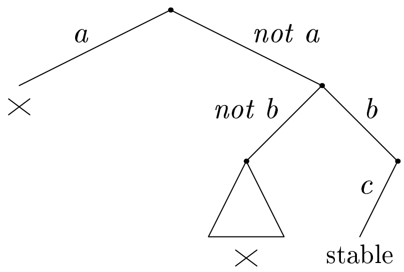


Figure 2: A visualization of the search process

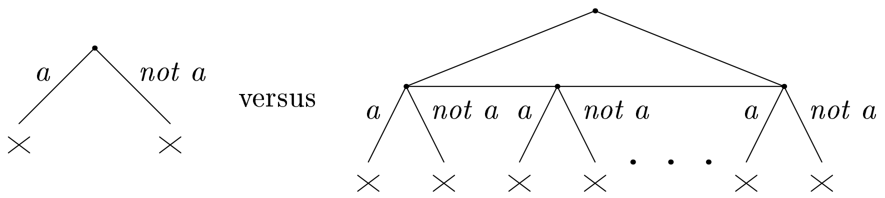


Figure 3: Looking ahead yields a potentially smaller search space

3.2 Looking Ahead

The *expand* function does a good job of reducing the search space. It makes use of some simple properties of the stable model semantics to refine a partially computed model. Even if the function only has to satisfy the two general conditions E1 and E2, it is in practice severely constrained by the small amount of time it can take to do its work. Since the function is called so often, it must be fast. Otherwise, the algorithm will not achieve acceptable performance on programs with many stable models.

By E2 the *expand* function must not loose any stable models. It can therefore not enlarge the partial model A by much if there are many stable models agreeing with A. A slower but slightly better *expand* function does not help. On the other hand, if a partial model does not agree with any stable models, then *expand* should return as large a set as possible. A slower function is not such an objection then. This dichotomy is addressed here.

The *expand* function as presented is a compromise that sacrifices optimality for performance. We want to strengthen it to better handle partial models that can not be enlarged to stable models. Consider a program P, a partial model A, and an atom a such that both $expand(P, A \cup \{a\})$ and $expand(P, A \cup \{not\ a\})$ contain conflicts, i.e., $conflict(P, A')$ returns true for $A' = expand(P, A \cup \{x\})$, where $x = a, not\ a$. If $smodels(P, A)$ chooses a or $not\ a$ immediately, then it will return after only two *expand* calls. If it does not, then the number of *expand* calls can be potentially very large, as is illustrated in Figure 3.

The literals that instantly give rise to conflicts can be found by testing. We call the testing procedure lookahead, as it corresponds to looking ahead

```
function lookahead(P, A)
  repeat
    A' := A
    A := lookahead_once(P, A)
  until A = A'
  return A.

function lookahead_once(P, A)
  B := Atoms(P) − Atoms(A)
  B := B ∪ not (B)
  while B ≠ ∅ do
    Take any literal x ∈ B
    A' := expand(P, A ∪ {x})
    B := B − A'
    if conflict(P, A') then
      return expand(P, A ∪ {not (x)})
    end if
  end while
  return A.
```

Algorithm 2: Looking ahead

and seeing how *smodels* behaves when it has chosen a literal. Observe that if the stable model S agrees with the partial model A but not with $A \cup \{x\}$ for some literal x, then S agrees with $A \cup \{not\,(x)\}$. Hence, we make progress as soon as we find a literal x that causes a conflict. That is to say, we can then enlarge A by $not\,(x)$. In addition, since $x' \in expand(P, A \cup \{x\})$ implies

$$\begin{aligned} expand(P, A \cup \{x'\}) &\subseteq expand\big(P, expand(P, A \cup \{x\})\big) \\ &= expand(P, A \cup \{x\}) \end{aligned}$$

due to the monotonicity of *expand*, it is not even necessary to examine all literals in $Atoms(P)$ not covered by A. As we test a literal x we can directly rule out all atoms in $expand(P, A \cup \{x\})$. We implement lookahead according to these observations by rewriting the function *lookahead* as in Algorithm 2. Possibly calling *expand* on the order of $|Atoms(P) - Atoms(A)|$ times for each new literal might not seem like a good idea, but in practice it has proven amazingly effective.

The idea that one can use the detection of conflicts to prune the search space has been presented in the context of propositional satisfiability checkers by Zabih and McAllester [67]. It is interesting to note that they concluded that the pruning method seems promising but causes too much overhead. Modern satisfiability checkers avoid the overhead by only employing lookahead on a small heuristically chosen subset of all atoms.

3.3 Heuristics

The heuristic choices that are made in a backtracking algorithm can drastically affect the time the algorithm has to spend searching for a solution. Since a correct choice brings the algorithm closer to a solution while a wrong choice leads the algorithm astray, great effort is often expended on creating heuristics that find the correct choices. This seems to be a bad approach. A heuristic invariably fails at some point, otherwise it would not be a heuristic, and then it tries to find nonexistent solutions when it should be minimizing the duration of the search in that part of the search space. We will therefore optimize our heuristic for the case of no stable models. That is, we will try to minimize the size of the remaining search space.

For a literal x, let

$$A_p = \mathit{expand}(P, A \cup \{x\})$$

and

$$A_n = \mathit{expand}\big(P, A \cup \{\mathit{not}\ (x)\}\big).$$

Assume that the search space is a full binary tree of height H. A full binary tree is a binary tree whose paths from the root to the leaves are all of equal length. Let $p = |A_p - A|$ and $n = |A_n - A|$. Then,

$$2^{H-p} + 2^{H-n} = 2^H \frac{2^n + 2^p}{2^{p+n}}$$

is an upper bound on the size of the remaining search space. Minimizing this number is equal to minimizing

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

Since

$$2^{\max(n,p)} < 2^n + 2^p \leq 2^{\max(n,p)+1}$$

is equivalent to

$$\max(n,p) < \log(2^n + 2^p) \leq \max(n,p) + 1$$

and

$$-\min(n,p) < \log(2^n + 2^p) - (p+n) \leq 1 - \min(n,p),$$

it suffices to maximize $\min(n,p)$. If two different literals have equal minimums, then one chooses the one with the greater maximum, as this minimizes $2^{-\max(n,p)}$.

When the best literal x has been found, we have to return one of x and $not\,(x)$. If there are no stable models agreeing with A, then it does not matter which one. But if there are stable models agreeing with both $A \cup \{x\}$ and $A \cup \{not\,(x)\}$, then we should again try to minimize the remaining search space. Hence, we return the one that shrinks the search space the most.

The function *heuristic* is shown in Algorithm 3. In an implementation of *smodels* one naturally integrates *lookahead* and *heuristic* to avoid unnecessary work. At the same time one can take advantage of the fact that $x' \in expand(P, A \cup \{x\})$ implies

$$|expand(P, A \cup \{x'\}) - A| \leq |expand(P, A \cup \{x\}) - A|$$

to evade some of the *expand* computations.

Freeman noted in [20] that a good satisfiability heuristic should choose the literal that minimizes the quantity $2^{H-p} + 2^{H-n}$, where n and p are computed using unit propagation instead of *expand*. It is again interesting to note that he then concluded that such a heuristic is too slow in practice.

function $heuristic(P, A)$
  $B := Atoms(P) - Atoms(A)$
  $min := 0$
  $max := 0$
  while $B \neq \emptyset$ **do**
    Take any atom $a \in B$
    $B := B - \{a\}$
    $p := |expand(P, A \cup \{a\}) - A|$
    if $p \geq min$ **then**
      $n := |expand\big(P, A \cup \{not\ a\}\big) - A|$
      if $\min(n,p) > min$ or $\big(\min(n,p) = min$ and $\max(n,p) > max\big)$ **then**
        $min := \min(n,p)$
        $max := \max(n,p)$
        if $p = \max(n,p)$ **then**
          $x := a$
        else
          $x := not\ a$
        end if
      end if
    end if
  end while
  return x.

Algorithm 3: The heuristic

Lookahead order

The *lookahead* function does not test the literals in any particular order. Since the function returns as soon as it finds a conflict, it is desirable that it should find them as soon as possible. We notice that in a large program P and for a small set of literals A, $expand(P, A \cup \{x, y\})$ probably does not contain a conflict if $expand(P, A \cup \{x\})$ does not. Therefore, if we first test a literal x and do not find a conflict and then test another literal y and find a conflict, then expanding $A \cup \{x, not(y)\}$ will quite likely not lead to a conflict. Hence, keeping the literals in a least recently used order is reasonable. Indeed, it has been verified that it often removes many needless *expand* calls.

3.4 Searching for Specific Stable Models

Besides searching for one stable model, one may also search for many, all, or just certain stable models. All of these variants can easily be incorporated into the *smodels* algorithm. In particular, we will deal with the case of optimize statements, as we must at least implicitly examine all stable models to find the smallest or largest one.

The simplest way to handle the search for specific stable models is to modify the *conflict*(P, A) function such that it returns true whenever one can be sure that we are not searching for any stable models agreeing with A. This can be thought of as changing what it means for a stable model to agree with a set of literals. Only the stable models that we accept can agree with a set of literals. Hence, we need not redo the proof of soundness and completeness for the modified algorithm. The new *conflict* function is presented in Algorithm 4. The function *unacceptable*(P, A) implements the test for the acceptable stable models. If it returns true, then there must not be any acceptable stable models agreeing with A. Moreover, if A^+ is a stable model that is not acceptable, then *unacceptable*(P, A) must return true.

function *conflict*(P, A)
  {Precondition: $A = expand(P, A)$}
  if $A^+ \cap A^- \neq \emptyset$ **then**
    return true
  else
    return *unacceptable*(P, A)
  end if.

Algorithm 4: Removing unacceptable stable models

What about computing all stable models of a program? We can control how many stable models are computed by calling a function *stable*(P, A)

function *smodels*(P, A)
  $A := \mathit{expand}(P, A)$
  $A := \mathit{lookahead}(P, A)$
  if $\mathit{conflict}(P, A)$ **then**
    return false
  else if A covers $\mathit{Atoms}(P)$ **then**
    return $\mathit{stable}(P, A)$ $\{A^+$ is a stable model$\}$
  else
    $x := \mathit{heuristic}(P, A)$
    if $\mathit{smodels}(P, A \cup \{x\})$ **then**
      return true
    else
      return $\mathit{smodels}\big(P, A \cup \{\mathit{not}\ (x)\}\big)$
    end if
  end if.

Algorithm 5: A generalization of the decision procedure

instead of returning true when a stable model has been found, cf Algorithm 5. If $\mathit{stable}(P, A)$ returns false, then the algorithm will search for the next model. Otherwise, it stops. Since returning false from *stable* is indistinguishable from returning false from *conflict*, it is guaranteed that *stable* is only called for acceptable stable models. If *stable* always returns false, then it will be called once for every stable model of P. Therefore, the function can be used to, e.g., display or count the models.

We now turn to the optimize statements. Without loss of generality we consider only programs containing one minimize statement and only positive weights:

$$\mathit{minimize}\ \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \mathit{not}\ b_1 = w_{b_1}, \ldots, \mathit{not}\ b_m = w_{b_m}\}.$$

We let the literals not in the minimize statement have zero weight. Let B be a global variable that is initially empty. We define the acceptable models as models whose weight is smaller than the weight of B if B is not empty. When an acceptable stable model A^+ is found we assign A to B. In addition, we let *stable* always return false. Thus, when *smodels* returns, B contains a stable model of minimal weight if one exists. Otherwise, it is empty. The corresponding *unacceptable* and *stable* functions are portrayed in Algorithm 6.

function *unacceptable*(P, A)
  if $B \neq \emptyset$ and $\sum_{a \in A} w_a + \sum_{not\ b \in A} w_b \geq \sum_{a \in B} w_a + \sum_{not\ b \in B} w_b$ **then**
    return true
  else
    return false
  end if.

function *stable*(P, A)
  $B := A$
  return false.

Algorithm 6: Finding a minimal stable model

4 Implementation

In this section we will present an efficient implementation of the functions $Atleast(P, A)$ and $Atmost(P, A)$. Both implementations are variations of a linear time algorithm of Dowling and Gallier [14]. We will also examine various optimizations that can be used to improve the implementation and the *smodels* algorithm.

The basic Dowling-Gallier algorithm computes the deductive closure of a set of basic rules in time linear in the size of the set of rules. The rules must not contain any not-atoms, i.e., they are Horn clauses. A variant of the basic algorithm is shown in Algorithm 7. It follows naturally from one observation and one implementation trick. We observe that a deductive step is monotone. Namely, if the body of a rule is in a set of atoms, then it is also in any superset of the same set. Hence, the order in which the rules are applied does not matter. The trick is to use a counter for each rule to find out when a rule can be used in the deduction. The counters should initially hold the number of atoms in the bodies of the rules. Every time an atom is added to the closure, the counters of the rules in whose bodies the atom appears are decremented. If any counter reaches zero, then the body of the corresponding rule is in the closure and the head of the rule is put in the closure. The basic algorithm is obviously correct. An atom is in the closure if and only if the atom is the head of a rule whose body is in the closure.

procedure Dowling-Gallier (P)
  {Invariant: $|\text{body} - \text{closure}| = \text{counter}$ and $\text{counter} = 0$ is equivalent to head $\in$ closure $\cup$ queue}
  Initialize the counter of every rule to the number of atoms in its body
  let the queue contain the heads of the rules in P whose bodies are empty
  while the queue is not empty **do**
    remove the first element from the queue and call it a
    if a is not in the closure **then**
      add a to the closure
      for each rule $r \in P$ in whose body a appears **do**
        decrement the counter of r by one
        if the counter of r is equal to zero **then**
          append the head of r to the end of the queue
        end if
      end for
    end if
  end while.

Algorithm 7: The basic Dowling-Gallier algorithm

4.1 At Least

We begin with basic rules of the form

$$h \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m.$$

For every rule $r \in P$ we create a literal counter $r.literal$ that holds the number of literals in the body of r that are not members of the partially computed closure $Atleast(P, A)$. In addition, an inactivity counter $r.inactive$ is also created. If the set A is a partially computed closure, then the inactivity counter records the number of literals in the body of r that are in $not\,(A)$. The counter $r.inactive$ is therefore positive, and the rule r is inactive, if the set $max_r(A)$ is empty and one can not then use r to deduce its head. For every atom a we create a head counter $a.headof$ that holds the number of active rules with head a.

Recall that a literal can be brought into $Atleast(P, A)$ in four different ways. We handle the four cases with the help of the three counters.

1. If $r.literal$ reaches zero, then the head of r is added to the closure.

2. If $a.headof$ reaches zero, then $not\ a$ is added to the closure.

3. If $a.headof$ is equal to one and a is in the closure, then every literal in the body of the only active rule with head a is added to the closure.

4. Finally, if a is the head of r, if $not\ a$ is in the closure, and if $r.literal = 1$ and $r.inactive = 0$, then there is precisely one literal x in the body of r that is not in the closure, and $not\,(x)$ is added to the closure.

Cardinality rules and choice rules are easily incorporated into the same framework. Specifically, one does neither use the first nor the fourth case together with choice rules, and one does not compare the literal and inactivity counters of a cardinality rule

$$h \leftarrow k\,\{a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m\}$$

with zero but with $m + n - k$. A weight rule

$$h \leftarrow \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, not\ b_1 = w_{b_1}, \ldots, not\ b_m = w_{b_m}\} \geq w,$$

is managed using the upper and lower bound of the sum of the weights in its body. Given a set of literals A, the lower bound is

$$\sum_{a_i \in A^+} w_{a_i} + \sum_{b_i \in A^-} w_{b_i}$$

and the upper bound is

$$\sum_{a_i \notin A^-} w_{a_i} + \sum_{b_i \notin A^+} w_{b_i}.$$

If the upper bound is less than w, then the rule is inactive, and if the lower bound is at least w, then the head is in the closure.

Note that by Lemma 3.4 the order in which the four ways of bringing atoms into the closure are used is of no importance for the end result. The actual implementation will not compute all of $Atleast(P, A)$ when it equals $Atoms(P) \cup not\big(Atoms(P)\big)$, but a subset B that satisfies $B^+ \cap B^- \neq \emptyset$. This is enough since no conflicts are lost.

In preparation for the procedure that computes $Atleast(P, A)$, we construct two Boolean flags for each atom a: $a.inA^+$ is set to true when a is included in the closure $Atleast(P, A)$ and $a.inA^-$ is set to true when $not\ a$ is included in the closure. All atoms have three lists as well. The list *a.plist* contains pointers to every rule r for which $a \in r.body$, the list *a.nlist* contains pointers to every rule r for which $not\ a \in r.body$, and the list *a.hlist* contains pointers to every rule r for which $a \in r.head$. To summarize, we have the following variables:

a.headof	The number of active rules whose head contains a.
$a.inA^+$	A flag that is true if a is in the current closure.
$a.inA^-$	A flag that is true if $not\ a$ is in the current closure.
a.plist	The rules in whose bodies a appears.
a.nlist	The rules in whose bodies $not\ a$ appears.
a.hlist	The rules in whose heads a appears.
r.literal	The number of literals in the body that are in the current closure.
r.inactive	The number of literals in the body that are in the negation of the current closure, i.e., in $not\,(A)$ if A is the current closure.
r.body	The body of the rule r.
r.head	The head or heads of the rule r.

The procedure *atleast*() that computes $Atleast(P, A)$ is described in Algorithm 8. It follows the basic Dowling-Gallier algorithm, but extends it to handle the four ways a literal can be included in $Atleast(P, A)$. The procedure is written such that the main work happens in the four functions *fire*(), *inactivate*(), *backchaintrue*(), and *backchainfalse*(), corresponding to the cases 1, 2, 3, and 4, respectively. We need specific instances of these functions for every type of rule. Therefore, we write *r.fire*() to denote the *fire*() function that corresponds to the rule r, and we use the same notation for the other functions. The instances used together with basic rules are given in Algorithms 9–10. The rest are in Appendix B. Since the procedure

```
procedure atleast()
  while posq or negq are not empty do
    if posq is not empty then
      a := posq.pop()
      a.inA+ := true
      for each rule r ∈ a.plist do
        r.fire()
      end for
      for each rule r ∈ a.nlist do
        r.inactivate()
      end for
      if a.headof = 1 then
        Let r be the only active rule in a.hlist
        r.backchaintrue()
      end if
    end if
    if negq is not empty then
      a := negq.pop()
      a.inA- := true
      for each rule r ∈ a.nlist do
        r.fire()
      end for
      for each rule r ∈ a.plist do
        r.inactivate()
      end for
      if a.headof > 0 then
        for each rule r in a.hlist do
          r.backchainfalse()
        end for
      end if
    end if
  end while.
```

Algorithm 8: The implementation of $Atleast(P, A)$

```
function r.fire()
  r.literal := r.literal − 1
  if r.literal = 0 then
    posq.push(r.head)
  else if r.head.inA⁻ then
    r.backchainfalse()
  end if.

function r.inactivate()
  r.inactive := r.inactive + 1
  if r.inactive = 1 then
    a := r.head
    a.headof := a.headof − 1
    if a.headof = 0 then
      negq.push(a)
    else if a.inA⁺ and a.headof = 1 then
      Let r′ be the only active rule in a.hlist
      r′.backchaintrue()
    end if
  end if.
```

Algorithm 9: Auxiliary functions for the basic rules

atleast() and the functions must be efficient they do not take any arguments but work on global data.

Two queues are used in *atleast*(): *posq* and *negq*. We assume that they are implemented such that pushing an atom onto the end of a queue does nothing if the atom is already on the queue. In addition, we assume that pushing an atom a whose $a.inA^+$ flag is true onto the queue *posq* or pushing an atom a whose $a.inA^-$ flag is true onto the queue *negq* also have no effect. These assumptions have a simple Boolean flag implementation and they make the descriptions a bit shorter.

One can use stacks instead of queues, but tests have shown that queues are faster. The reason seems to be that if there is a conflict, then it is found quicker if one derives literals breadth-first using a queue than if one derives them depth-first using a stack. A stack implementation might do a lot of work before it notices a conflict that is only a few rules from the start of the derivation.

Before computing $Atleast(P, A)$ we have to initialize the queue *posq* with A^+ and the heads of the rules whose bodies are empty. The queue *negq* must be initialized with the atoms in A^-. Observe that the procedure can also be used incrementally. If $Atleast(P, A)$ has been computed and we are going to compute $Atleast(P, A \cup \{a\})$, then we just initialize *posq* with a and call *atleast*().

function $r.backchaintrue()$
  for every $a \in r.body^+$ **do**
    $posq.push(a)$
  end for
  for every $a \in r.body^-$ **do**
    $negq.push(a)$
  end for.

function $r.backchainfalse()$
  if $r.literal = 1$ and $r.inactive = 0$ **then**
    for every $a \in r.body^+$ **do**
      if $a.inA^+ =$ false **then**
        $negq.push(a)$
        return
      end if
    end for
    for every $a \in r.body^-$ **do**
      if $a.inA^- =$ false **then**
        $posq.push(a)$
        return
      end if
    end for
  end if.

Algorithm 10: More auxiliary functions for the basic rules

Proposition 4.1. *The procedure atleast() computes Atleast(P, A) in time linear in the size of the program P.*

Proof. Note that for every atom a, the list $a.hlist$ is traversed at most twice and the lists $a.plist$ and $a.nlist$ are traversed at most once. For every rule r, the list $r.body$ is traversed at most twice. To be precise, the list $a.hlist$ is only traversed in the procedure *atleast*() when $a.inA^+$ or $a.inA^-$ is being set and this can only happen once. The same holds for the lists $a.plist$ and $a.nlist$. Moreover, the only other place where the list $a.hlist$ is examined is in the function *inactivate*() and there too only once, when $a.headof$ is decremented to one.

The list $r.body$ is only traversed in *backchaintrue*() and *backchainfalse*(). These functions are, for each rule, called at most once from the procedure *atleast*(). The function *backchainfalse*() is also called from *fire*(), but then $r.head.inA^-$ is true and the list is gone through if $r.literal = 1$, which only happens once. Finally, *backchaintrue*() is called from *inactivate*(), but only once when $r.head.headof = 1$. □

In conclusion, notice how the amount of work performed by the procedure *atleast*() corresponds to the number of literals in $Atleast(P, A) - A$. Specifically, observe that if *atleast*() is called in succession with an increasingly larger set A, then the total amount of work done is still linear in the size of P.

4.2 At Most

The deductive closure $Atmost(P, A)$ can be computed in a style similar to that of $Atleast(P, A)$. However, since $Atmost(P, A)$ diminishes as A grows, one would then have to compute it from scratch each time A changes. If $Atmost(P, A)$ is large and if it changes only a little, then we would be doing a lot of extra work if we computed it anew.

We will try to localize the computation by using the basic Dowling-Gallier algorithm in two stages. Assume that we have computed the deductive closure $Atmost(P, A)$ and that we want to compute $Atmost(P, A')$ for a set $A' \supset A$. We begin by calculating a set $B \subseteq Atmost(P, A')$ with the help of a version of the basic Dowling-Gallier algorithm. Instead of deriving new atoms, this version removes them. After this stage we apply the basic algorithm to incrementally compute $Atmost(P, A')$ from B.

The first stage removes atoms from the closure $Atmost(P, A)$ in the following way. If it notices that a rule can no longer be used to derive the atom or atoms in the head of the rule, then it removes the atom or atoms from the closure. The removal may lead to the inactivation of more rules and subsequent removals of more atoms. Since there can be rules that still imply the inclusion of an atom in $Atmost(P, A')$ after the atom has been removed, the first stage might remove too many atoms. We therefore need the second stage to add them back. Sufficiently many atoms are removed by the first stage, as an atom that is not removed is the head of a rule that is not inactive. The atoms in the body of this rule are in the same way heads of other rules that are not inactive, and this succession continues until one reaches rules that imply the inclusion of their heads even if the closure were empty.

The two stages need a new counter for every rule. We call this counter *upper*, since it is used to compute the upper closure that approximates stable models from above. In addition, each atom a has a Boolean flag $a.inUpper$ that is true if a is a member of the upper closure. The procedure that computes $Atmost(P, A)$ is described in Algorithm 11. The main work happens in the three functions *propagateFalse*(), *propagateTrue*(), and *isUpperActive*(), corresponding to decrementing the counter, incrementing the counter, and testing if the rule can be used to derive its head or heads, respectively. The functions for the basic rules are given in Algorithm 12. The functions for the other types are in Appendix C.

We assume that the queue *queue* is implemented such that pushing an atom onto the end of it does nothing if the atom is already on the queue. The *inactive* counters are used by the auxiliary functions and the inA^- flag is used by the main procedure. For a rule r, $r.inactive > 0$ if

$$max'_r(A) = \bigcup_{\substack{A^+ \subseteq C \\ A^- \cap C = \emptyset}} f'_r(C, C)$$

is the empty set, and for an atom a, $a.inA^-$ is true if $a \in A^-$. In addition, we assume that the flag $a.inUpper$ is set to false whenever $a.inA^-$ is true.

The first and largest upper closure, $Atmost(P, \emptyset)$, can be computed by the second stage if the queue is initialized with the atoms that follow immediately from some rules. These are the atoms in

$$\bigcup_{r \in P} f'_r(\emptyset, \emptyset).$$

The problem that remains is initializing the queue such that the first stage removes sufficiently many atoms from $Atmost(P, A)$. We must initialize the queue with the atoms in the heads of the rules that potentially become inactive when A changes to A'. Recall that $Atmost(P, A)$ is the least fixed point of the operator

$$f'(B) = \bigcup_{r \in P} f'_r(A^+, B - A^-) - A^-.$$

For $A \subseteq A'$, $C = Atmost(P, A)$ and for every basic or choice rule r we initialize the queue with the atoms in

$$f'_r(A^+, C - A^-) - f'_r(A'^+, C - A'^-) - A'^-.$$

```
procedure atmost()
  F := ∅
  while queue is not empty do
    a := queue.pop()
    if a.inUpper = true then
      for each rule r ∈ a.plist do
        r.propagateFalse()
      end for
      a.inUpper := false
      F := F ∪ {a}
    end if
  end while
  for each atom a ∈ F do
    for each rule r ∈ a.hlist do
      if r.isUpperActive() then
        queue.push(a)
      end if
    end for
  end for
  while queue is not empty do
    a := queue.pop()
    if a.inUpper = false and a.inA⁻ = false then
      for each rule r ∈ a.plist do
        r.propagateTrue()
      end for
      a.inUpper := true
    end if
  end while.
```

Algorithm 11: The implementation of $Atmost(P, A)$

For every cardinality and weight rule r we initialize the queue with the atoms in

$$f'_r(A^+, C - A^-) - f'_r(A'^+, \emptyset) - A'^-.$$

The reason behind the more lax condition for cardinality and weight rules is best illustrated by an example.

Example 4.2. Let P be the program consisting of the rule r

$$a \leftarrow 1\,\{a, not\ b\}.$$

Then, $C = Atmost(P, \emptyset) = \{a\}$ and $Atmost(P, \{b\}) = \emptyset$. However,

$$f'_r(\{b\}^+, C - \{b\}^-) - \{b\}^- = f'_r(\{b\}, \{a\}) = \{a\}.$$

function $r.propagateFalse()$
  $r.upper := r.upper + 1$
  if $r.upper = 1$ and $r.inactive = 0$ **then**
    $queue.push(r.head)$
  end if.

function $r.propagateTrue()$
  $r.upper := r.upper - 1$
  if $r.upper = 0$ and $r.inactive = 0$ **then**
    $queue.push(r.head)$
  end if.

function $r.isUpperActive()$
  if $r.upper = 0$ and $r.inactive = 0$ **then**
    return true
  else
    return false
  end if.

Algorithm 12: Auxiliary functions for the basic rules

Hence, for $A = \emptyset$ and for $A' = \{b\}$,

$$f'_r(A^+, C - A^-) - f'_r(A'^+, C - A'^-) - A'^- = \emptyset$$

and we miss an inactive rule if we assume that the atoms in C remain in the closure.

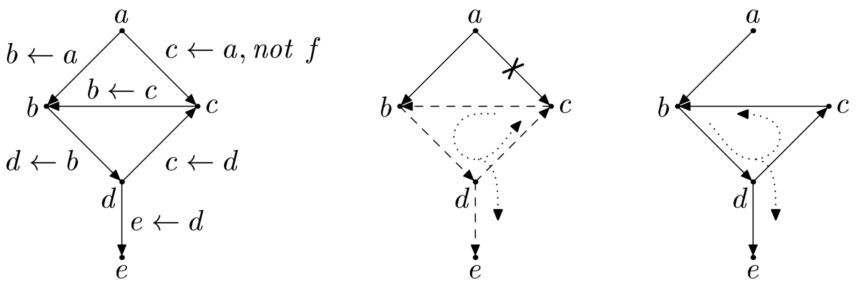
Example 4.3. An illustration of the two stages of *atmost*() is shown in Figure 4. The program P

$$\begin{array}{ll} b \leftarrow a & c \leftarrow a, not\ f \\ b \leftarrow c & c \leftarrow d \\ d \leftarrow b & e \leftarrow d \\ a \leftarrow & \end{array}$$

is displayed as a graph whose nodes are the atoms and whose edges are the rules that partake in the computation. The initial state is shown in 4(a) and corresponds to $Atmost(P, \emptyset) = \{a, b, c, d, e\}$. In 4(b) the atom f is added to $A = \emptyset$ and the rule $c \leftarrow a, not\ f$ becomes inactive. The first stage then removes the atoms c, b, d, and e. The second stage notices that b still follows from $b \leftarrow a$ and adds the atoms b, d, e, and c back into the closure. The end result $Atmost(P, \{f\}) = \{a, b, c, d, e\}$ is shown in 4(c).

Proposition 4.4. *The procedure atmost() computes $Atmost(P, A)$ in time linear in the size of the program P.*

Proof. Note that for every atom a, the list $a.plist$ is traversed at most twice and the lists $a.hlist$ is traversed at most once. □



(a) $A = \emptyset$, all atoms are in the closure

(b) $A = \{f\}$, the first stage removes all atoms but a

(c) The second stage adds the atoms back

Figure 4: A visualization of the computation of $Atmost(P, A)$

4.3 Various Optimizations

It is possible to optimize the *smodels* algorithm in various ways. For example, if the algorithm has deduced that an atom can not be part of any model, then one can remove the atom from the bodies of the rules in which it appears. The bodies shorten and any following traversal of them is faster. Similarly, an inactive rule can safely be removed from the lists of the atoms. A reduction of this type leads only to a small performance improvement. Hence, reducing a program by removing rules and atoms is best done just before the heuristic is computed for the first time, as one then need not undo the changes later.

The heuristic can in principle return any atom in $Atoms(P)$ for a program P. Since $smodels(P, A)$ calls itself twice for every choice point that the heuristic finds, the algorithm explores at worst a search space of size $2^{|Atoms(P)|}$. Restricting the choice points to a smaller set could substantially improve the algorithm.

The procedure *atmost*() will in the worst case remove all atoms from the upper closure before adding them back again. If we could localize the computation of *atmost*() to small parts of the program, then we could avoid the worst case.

Below, we identify the choice points that can be ignored and present two methods for localizing the computation of $Atmost(P, A)$.

Reducing the Search Space

Fix a program P. Let B be the set of atoms that appear as not-atoms in the bodies of the rules in P or in the heads of the choice rules in P. Then, for any rule $r \in P$ and for any sets of atoms $S, C \subseteq Atoms(P)$,

$$f_r(S \cap B, C) = f_r(S, C)$$

by the definition of f_r. Hence, there is a one-to-one correspondence between the stable models and their intersections with B. In other words, we do not have to consider choice points that are not in B.

The program P defines a directed graph $G = (V, E)$ where the vertices is the set $V = \mathit{Atoms}(P) \cup P$ and where the edges is the set

$$E = \{\langle r, a\rangle \mid a \in r.\mathit{head}\} \cup \{\langle a, r\rangle \mid a \in r.\mathit{body}^+ \text{ or } a \in r.\mathit{body}^-\}.$$

We say that the atom a appears as a not-atom in a cycle in the graph if the edge $\langle a, r\rangle$ is part of the cycle and $a \in r.\mathit{body}^-$.

Let

$$P_a = \{r \mid \text{there is a directed path from } r \text{ to } a \text{ in } G\}.$$

If a does not appear as a not-atom in a cycle and if a does not appear in the head of a choice rule, then $g_{P_a}(S) = g_{P_a}(S - \{a\})$ for all S. Since

$$g_{P_a}(S) \subseteq g_P(S)$$

and since $a \in g_P(S)$ implies $a \in g_{P_a}(S)$ for any S, $a \in S = g_P(S)$ for a stable model S if and only if $a \in g_{P_a}(S)$. Thus, the choice points can be restricted to the set $\mathit{Atoms}(P) - \{a\}$. Furthermore, we can combine all restrictions as $P_a \subseteq P_b$ if there is a path from a to b. Therefore, the set of choice points B can be taken to be the atoms that appear as not-atoms in some cycles or that appear in the heads of some choice rules.

Proposition 4.5. *Let P be a program, let B be the set of atoms that appear as not-atoms in some cycles or that appear in the heads of some choice rules. Let S and S' be stable models of P. If $S \cap B = S' \cap B$, then $S = S'$.*

Proof. Let B be the set of atoms that appear as not-atoms in some cycles or that appear in the heads of some choice rules of P. Define $B_0 = B$ and

$$\begin{aligned} B_{i+1} = B_i \cup \{a \in \mathit{Atoms}(P) - B_i \mid {} & \text{for every } b \in \mathit{Atoms}(P) - B_i, \\ & \text{if there is a path from } b \text{ to } a, \text{ then there is a path from } a \text{ to } b\}. \end{aligned}$$

If $a \in B_{i+1} - B_i$, then a does not appear in the head of a choice rule nor does it appear as a not-atom in

$$P_{i+1} = \bigcup_{b \in B_{i+1} - B_i} P_b.$$

Hence,

$$g_{P_{i+1}}(S \cap B_i) = g_{P_{i+1}}(S \cap B_{i+1})$$

for any S. Now, for any $a \in B_{i+1} - B_i$ and for any S, $a \in g_P(S)$ if and only if $a \in g_{P_{i+1}}(S)$.

Let S and S' be two stable models of P for which $S \cap B_i = S' \cap B_i$ holds. If $a \in B_{i+1} - B_i$, then $a \in S = g_P(S)$ implies

$$\begin{aligned} a \in g_{P_{i+1}}(S) = g_{P_{i+1}}(S \cap B_{i+1}) = g_{P_{i+1}}(S \cap B_i) = g_{P_{i+1}}(S' \cap B_i) \\ = g_{P_{i+1}}(S' \cap B_{i+1}) = g_{P_{i+1}}(S') \subseteq g_P(S') = S'. \end{aligned}$$

Thus, $S \cap B_{i+1} = S' \cap B_{i+1}$ by symmetry. Consequently, $S \cap B = S' \cap B$ implies $S = S'$. □

We need a stronger result than that the stable model semantics makes the atoms in the set B define the stable models. We want *smodels* to stop searching as soon as B is covered. This holds if we use both $Atmost(P, A)$ and $Atleast(P, A)$ in *expand*.

Proposition 4.6. *Let P be a program, let B be the set of atoms that appear as not-atoms in some cycles or that appear in the heads of some choice rules of P. Let A be a set of literals such that $Atoms(A) = B$. Then, $expand(P, A)$ covers $Atoms(P)$.*

Proof. Let $C = expand(P, A)$. Assume that $Atoms(C) \subset Atoms(P)$. Then, there exists an atom $a \in Atmost(P, C) - C^+$ and a rule $r \in P$ that is not a choice rule such that

$$a \in f'_r(C^+, C^+ - C^-) - C^- = f_r(C^+, C^+).$$

Since otherwise $f'_r(C^+, C^+ - C^-) - C^- \subseteq C^+$, $f'(C^+) \subseteq C^+$, and consequently $Atmost(P, C) \subseteq C^+$ by Lemma A.1.

As the atoms in $Atoms(P) - Atoms(C)$ can not appear as not-atoms in the body of r,

$$f_r(C^+, C^+) = f_r(C', C^+)$$

for $C^+ \subseteq C'$ and $C^- \cap C' = \emptyset$. Hence, $f_r(C^+, C^+) = min_r(C)$ since f_r is monotonic in its second argument. But then,

$$a \in Atleast(P, C) \subseteq C$$

which is a contradiction. Thus, C covers $Atoms(P)$. □

It is easy to make the *smodels* algorithm ignore choice points. We just change the cost function of the heuristic. Recall that the heuristic searches for the literal x that minimizes $2^{-p} + 2^{-n}$ for $p = |A_p - A|$ and $n = |A_n - A|$, where

$$A_p = expand(P, A \cup \{x\}) \quad \text{and} \quad A_n = expand\big(P, A \cup \{not\,(x)\}\big).$$

To limit the relevant choice points to B we simply redefine p and n as

$$p = |(A_p - A) \cap B| \quad \text{and} \quad n = |(A_n - A) \cap B|.$$

Notice that we do not really restrict the choice points to B. Instead, we guarantee that if *smodels* branches on a choice point, then at least one literal in B will follow when we prune the search space. Thus, we retain the greatest possible freedom in choosing choice points and we can still be certain that the size of the search space stays below $2^{|B|}$.

Example 4.7. Let P be the program

$$a \leftarrow \mathit{not}\ b \qquad\qquad b \leftarrow c$$
$$c \leftarrow \mathit{not}\ a \qquad\qquad d \leftarrow \mathit{not}\ c.$$

Since d does not appear as a not-atom in the program, it is not a choice point. Since c does not appear as a not-atom in a cycle, it is not a choice point either. Thus, the set of choice points of P are $B = \{a, b\}$.

Strongly Connected Components

Consider the computation of $Atmost(P, A')$ that starts from $Atmost(P, A)$. The function $atmost()$ operates on the graph $G = (V, E)$ given by $V = Atoms(P)$ and

$$E = \{\langle a, b\rangle \mid \text{there is a rule } r \text{ such that } a \in r.body^+ \text{ and } b \in r.head\}.$$

Take an atom $a \in Atmost(P, A)$. Observe that the only atoms that determine whether a is removed from the upper closure during the first stage of $atmost()$ are the atoms that have a directed path to a in G. Hence, we can localize $atmost()$ by computing both stages inside a strongly connected component of G before moving on to the rest of the graph. A strongly connected component of a directed graph is a set of vertices such that there is a directed path from any vertex in the component to any other.

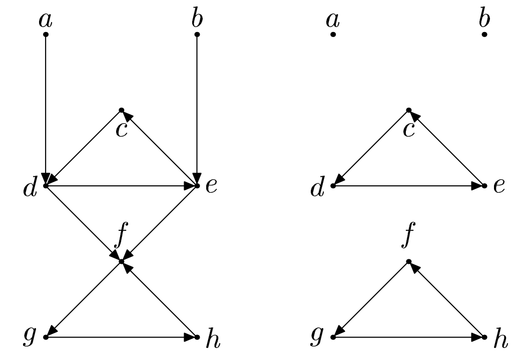
In practice, this is done by removing rules from the list $a.plist$ for every atom a. Specifically, if $r \in a.plist$ and if $r.head$ is not in the same strongly connected component as a, then we remove r from $a.plist$. Here we assume that $atmost()$ and $atleast()$ use their own versions of $a.plist$.

One $atmost()$ invocation will only compute a superset of the new upper closure after this change. If the atom a is removed, then we know that $a.inA^-$ can be set to true. Setting $a.inA^-$ to true in turn, influences the *inactive* counters of some rules. Therefore, we can possibly initialize the queue with some atoms. If we call $atmost()$ every time we initialize the queue with new atoms, we eventually remove everything not in $Atmost(P, A')$.

Notice that we do not explicitly keep track of the strongly connected components nor the acyclic graph they induce. Hence, the components are not processed in any fixed order even if the order given by the acyclic graph would be favorable. Changing the queue in $atmost()$ into a priority queue would solve this problem. Also notice that if there are no strongly connected components, then we need not compute $atmost()$.

$$
\begin{aligned}
c &\leftarrow e\\
d &\leftarrow a\\
d &\leftarrow c\\
e &\leftarrow b\\
e &\leftarrow d\\
f &\leftarrow d\\
f &\leftarrow e\\
f &\leftarrow h\\
g &\leftarrow f\\
h &\leftarrow g
\end{aligned}
$$

(a) Original program



(b) Original program as a graph

(c) The strongly connected components

Figure 5: Localizing $Atmost(P, A)$ with strongly connected components

Example 4.8. A partial program is shown in Figure 5(a). The corresponding graph is shown in Figure 5(b). If we assume that a and b are in the upper closure then the upper closure is the set $\{a, b, c, d, e, f, g, h\}$. If we remove a, then the first stage of *atmost*() removes all atoms except b and the second stage adds them back.

Removing the rules that leave a strongly connected component results in the graph in Figure 5(c). The first stage of *atmost*() now removes only the atoms c, d, and e.

The Source Pointer

Partitioning a program into strongly connected components localizes the upper closure computation to a component if the component remains in the upper closure. If the component is removed, then the computation spreads to other components, perhaps in vain.

We can partly overcome this problem with the help of the following construct. For every atom a, we create a source pointer $a.source$ whose mission is to point to the first rule that causes a to be included in the upper closure. During the first stage of *atmost*(), it suffices to only remove atoms which are to be removed due to a rule in a source pointer. For if the rule in a source pointer does not justify the removal of an atom, then the atom is reentered into the closure in the second stage of the computation. The source pointer is implemented by modifying the auxiliary functions

function $r.propagateFalse()$
  $r.upper := r.upper + 1$
  if $r.upper = 1$ and $r.inactive = 0$ and
    $(r.head.source = 0$ or $r.head.source = r)$ **then**
    $r.head.source := 0$
    $queue.push(r.head)$
  end if.

function $r.propagateTrue()$
  $r.upper := r.upper - 1$
  if $r.upper = 0$ and $r.inactive = 0$ **then**
    if $r.head.source = 0$ **then**
      $r.head.source := r$
    end if
    $queue.push(r.head)$
  end if.

Algorithm 13: Auxiliary functions and the source pointer

propagateFalse() and *propagateTrue*() as can be seen in Algorithm 13.

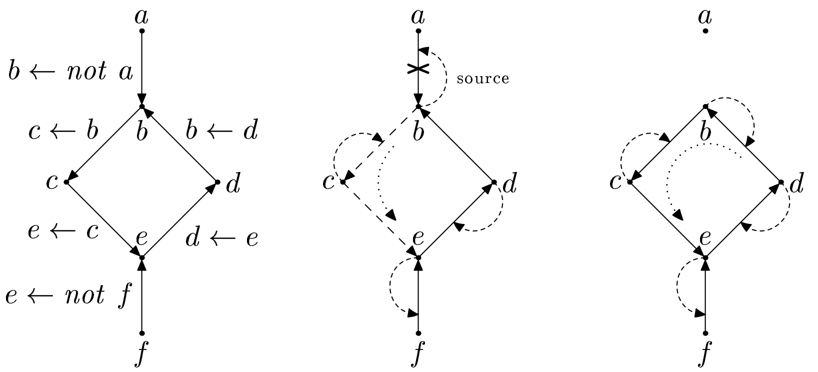
Example 4.9. Let P be the program

$$\begin{array}{ll} b \leftarrow not\ a & b \leftarrow d \\ c \leftarrow b & d \leftarrow e \\ e \leftarrow c & e \leftarrow not\ f. \end{array}$$

Three pictures illustrating the source pointers and the two stages of *atmost*() are shown in Figure 6. The initial state is shown in 6(a) and corresponds to $Atmost(P, \emptyset) = \{b, c, d, e\}$. In 6(b) the atom a is added to $A = \emptyset$ and the rule $b \leftarrow not\ a$ becomes inactive. The first stage then removes the atom b, as $b.source$ points at the rule $b \leftarrow not\ a$, and the atom c, as $c.source$ points at $c \leftarrow b$. The second stage notices that b still follows from $b \leftarrow d$ and adds the atoms b and c back into the closure. The source pointer $b.source$ is at the same time updated to point at $b \leftarrow d$. The end result $Atmost(P, \{a\}) = \{b, c, d, e\}$ is shown in 6(c).

4.4 Backtracking

The $smodels(P, A)$ procedure employs chronological backtracking. The last literal that is included in a partial stable model is the first that is removed when a conflict is detected. Since we want a linear-space implementation of $smodels(P, A)$, we can not store snapshots of the states of the data structures. We must instead store the changes that take place.



(a) $A = \emptyset$, all atoms are in the closure

(b) $A = \{a\}$, the first stage removes b and c

(c) The second stage adds the atoms back

Figure 6: Localizing $Atmost(P, A)$ with source pointers

It is enough to keep track of the changes to the set A. For example, if the atom a is added to A, i.e., if the flag $a.inA^+$ is set to true, then for every basic rule r in *a.plist* the counter *r.literal* is decremented by one. Hence, if a is removed from A, then *r.literal* should be incremented. Generally, the value of every counter is completely determined from its previous value, and the previous value is completely determined by its new value. Hence, if we during backtracking undo the changes to A in the opposite order to which they took place, then we can directly compute the old values of the counters.

Consequently, backtracking can be implemented with the help of a stack of size $Atoms(P)$. Every time a literal is added to A it is also pushed onto the stack. We backtrack by popping literals off the stack and by computing the correct values for the relevant counters.

Backjumping

Lookahead guarantees that conflicts are caused by the chronologically next to last choice. Assume that the literal x_1 is the next to last choice, as given by the heuristic. Furthermore, assume that $conflict(P, A')$ returns false for $A' = expand(P, A \cup \{x_1\})$, but that the choice of x_1 does not affect the conflict that arises when the literal x_2 is added to A'. That is to say, $conflict(P, A'')$ returns true for $A'' = expand(P, A \cup \{x_2\})$. Since *lookahead* examines all possible ways of adding a literal to A, it notices this conflict before x_1 is chosen. Hence, the literal x_1 is not the next to last choice, and by the same argument, neither is any other literal not related to the conflict.

Despite the fact that many conflicts are discovered by the *lookahead* function, there are still situations where the decision procedure exhaustively searches through assignments that are not relevant to a set of conflicts. A simple example is given by the union of two programs that do not share atoms. We assume that the first program has several stable models and that the second one has none. If we always begin by trying 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.

The excessive backtracking in the preceding example is caused by the fact that *smodels*(P, A) does not notice that the conflicts in one subprogram are independent from the truth-assignments in the other. Fortunately, it is easy to improve *smodels*.

Let *independent*(P, A, x_1, x_2) be a function whose arguments are a program P, a set of literals A, and two literals x_1 and x_2. Let

$$A_1 = \mathit{expand}(P, A \cup \{x_1\}),$$
$$A_2 = \mathit{expand}(P, A \cup \{x_2\}),$$

and

$$A_3 = \mathit{expand}(P, A \cup \{x_1, x_2\}).$$

We assume that *independent* fulfills the two conditions

I1 if *independent*(P, A, x_1, x_2) returns true, if *conflict*(P, A_2) returns false, and if *conflict*(P, A_3) returns true, then *conflict*(P, A_1) returns true, and

I2 if *independent*(P, A, x_1, x_2) returns true and if A is a subset of A', then *independent*(P, A', x_1, x_2) returns true.

We say that x_1 is independent from x_2 if *independent*(P, A, x_1, x_2) returns true.

If x_1 is independent from x_2, if x_2 is just before x_1 on the stack, and if x_1 gives rise to a conflict, then one can remove both x_1 and x_2 from the stack before pushing *not* (x_1) onto it without loosing any stable models. Thus, we can skip some literals while backtracking. We call this backjumping.

Consider the generalized *smodels* procedure that only accepts a stable model for which *stable*(P, A) returns true. Since we can backtrack from a stable model and since we have to guarantee the completeness of the decision procedure, we have to distinguish between two modes of backtracking. Specifically, backtracking from a stable model must be chronological while backtracking from a conflict may take advantage of backjumping.

A decision procedure that incorporates backjumping is presented in Algorithm 14. 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.

```
function smodels(P, A, level)
  A := expand(P, A)
  A := lookahead(P, A)
  if conflict(P, A) then
    return false
  else if A covers Atoms(P) then
    top := level
    return stable(P, A) {A⁺ is a stable model}
  else
    x := heuristic(P, A)
    conflict := x
    if smodels(P, A ∪ {x}, level + 1) then
      return true
    else if level ≥ top and independent(P, A, conflict, x) then
      return false
    else
      if level < top then
        top := level
      end if
      return smodels(P, A ∪ {not (x)}, level + 1)
    end if
  end if.
```

Algorithm 14: Incorporating backjumping

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.

Theorem 4.10. *Let P be a set of rules and let A be a set of literals. If there is a stable model S of P agreeing with A such that $stable(P, S')$ returns true for $S' = S \cup not\big(Atoms(P) - S\big)$, then $smodels(P, B, top)$ returns true.*

Proof. We show that completeness is not lost when backjumping is introduced. 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 $A \cup \{x_0\}$ and that when $smodels(P, A \cup \{x_0\}, level + 1)$ returns false,

$$independent(P, A, conflict, x_0)$$

returns true. Furthermore, assume that S is a stable model agreeing with the set $A \cup \{not\,(x_0), x_1, \ldots, x_n\}$ and that the set covers $Atoms(P)$. Then, $conflict(P, A')$ returns true for

$$A' = expand(P, A \cup \{x_0, x_1, \ldots, x_n\}),$$

since there is no stable model agreeing with A'. Take i such that $conflict = x_i$ or $conflict = not\,(x_i)$, and let j be the smallest number for which

$$conflict\big(P, expand(P, A \cup \{x_i, x_0, x_1, \ldots, x_j\})\big)$$

returns true and $conflict\big(P, expand(P, A \cup \{x_0, x_1, \ldots, x_j\})\big)$ returns false. Then, $conflict\big(P, expand(P, A \cup \{x_i, x_1, \ldots, x_j\})\big)$ returns true by I1, I2, and E1. Hence, by C2 there is no stable model that agrees with $A \cup \{not\,(x_0), x_1, \ldots, x_n\}$ and this contradicts the existence of S. Thus, there is no stable model agreeing with $A \cup \{not\,(x_0)\}$.

We complete the proof by induction on the size of

$$nc(P, A \cup \{x\}) = Atoms(P) - Atoms(A \cup \{x\}).$$

If $A \cup \{x\}$ covers $Atoms(P)$, if the function $smodels(P, A \cup \{x\}, level + 1)$ returns false and if $level \geq top$, then there is no stable model agreeing with $A \cup \{x\}$. Let $nc(P, A \cup \{x\}) \neq \emptyset$, let $smodels(P, A \cup \{x\}, level + 1)$ return false and let $level \geq top$. Then, $smodels(P, A' \cup \{x'\}, level + 2)$ returns false for $A' = expand(P, A \cup \{x\})$ and $x' \in nc(P, A')$, and by induction there is no stable model agreeing with $A' \cup \{x'\}$. If $independent(P, A', conflict, x')$ returns true, then by the above there is no stable model agreeing with $A' \cup \{not\,(x')\}$ either. If, on the other hand, $independent(P, A', conflict, x')$ returns false, then $smodels\big(P, A' \cup \{not\,(x')\}, level + 2\big)$ returns false and there is again by induction no stable model that agrees with $A' \cup \{not\,(x')\}$. Therefore, there is no stable model that agrees with $A \cup \{x\}$. $\square$

Let the undirected graph $G = (V, E)$ of a program P be defined by $V = Atoms(P)$ and

$$E = \{\langle a, b\rangle \mid a, b \in r.head \cup r.body^+ \cup r.body^- \text{ for some } r \in P\}.$$

A simple but easy to compute instance of $independent(P, A, x_1, x_2)$ returns true exactly when there is no path in G between the atoms that x_1 and x_2 cover. Since *expand* can not propagate any truth values via any inactive rules, we can safely remove these rules from G. Similarly, any atom in A^- can also be removed. However, we must not remove any atoms in A^+, as *atmost*() examines them during its computation. An example of what can go wrong if the atoms in A^+ are removed from G is given in the next example.

Example 4.11. Let P be the program

$$\begin{array}{ll} a \leftarrow b & \qquad b \leftarrow a \\ a \leftarrow c & \qquad b \leftarrow d, not\ d \\ \{c\} \leftarrow & \qquad \{d\} \leftarrow \end{array}$$

and let $A = \{a, b\}$. If we remove the literals in A^+ from the graph, then d and c are not connected. Hence, by first choosing *not* c and then *not* d we get a conflict, and when we try d we get another conflict. Since d is not connected to c, we backjump past c thereby missing the stable models $\{a, b, c\}$ and $\{a, b, c, d\}$.

5 Complexity

The *smodels* procedure solves an NP-complete problem. It returns true if and only if a program has a stable model. By Proposition 3.8, we can test whether a set of atoms is a stable model in polynomial time.

Theorem 5.1. *Deciding whether a set of basic, choice, cardinality, and weight rules has a stable model is NP-complete.*

Unsurprisingly, the *smodels* procedure has a worst-case time complexity that is exponential in the number of atoms of a program. The interesting thing about exponential computations is that one can make them faster asymptotically by trading an arbitrary amount of polynomial work for a tiny reduction of the exponent. For instance, for any k

$$n^k 2^{\frac{(k-1)n}{k}} < 2^n$$

for large n.

We get exponential behavior from *smodels* if we try to solve the pigeon-hole problem. This is the problem of trying to stuff n pigeons into k holes such that there is at most one pigeon per hole. Clearly, there is no solution if $n > k$. Let the atom $p_{i,j}$ be true when pigeon number i is in hole number j. Then, the pigeon-hole problem is encoded by the program P

$$\begin{aligned}
\{p_{i,1}, \ldots, p_{i,k}\} &\leftarrow \\
\mathit{false} &\leftarrow 2\,\{p_{i,1}, \ldots, p_{i,k}\} \\
\mathit{false} &\leftarrow \mathit{not}\ p_{i,1}, \ldots, \mathit{not}\ p_{i,k} \\
\mathit{false} &\leftarrow 2\,\{p_{1,j}, \ldots, p_{n,j}\} \\
\mathit{compute}\ &\{\mathit{not}\ \mathit{false}\}
\end{aligned}$$

for $1 \le i \le n$ and $1 \le j \le k$.

Let A be a set of literals having *not false* as a member. Since the program does not contain any positive loops, $\mathit{expand}(P, A) = \mathit{Atleast}(P, A)$. We will examine the behavior of $\mathit{Atleast}(P, A)$. Notice that if $p_{i,j} \in A$, then

$$\mathit{not}\ p_{1,j}, \ldots, \mathit{not}\ p_{i-1,j}, \mathit{not}\ p_{i+1,j}, \ldots, \mathit{not}\ p_{n,j} \in \mathit{Atleast}(P, A)$$

and

$$\mathit{not}\ p_{i,1}, \ldots, \mathit{not}\ p_{i,j-1}, \mathit{not}\ p_{i,j+1}, \ldots, \mathit{not}\ p_{i,k} \in \mathit{Atleast}(P, A).$$

Furthermore, if

$$\mathit{not}\ p_{i,1}, \ldots, \mathit{not}\ p_{i,j-1}, \mathit{not}\ p_{i,j+1}, \ldots, \mathit{not}\ p_{i,k} \in A,$$

then $p_{i,j} \in \mathit{Atleast}(P, A)$, and there is no other way of deducing additional literals in $\mathit{Atleast}(P, A)$.

Denote by P_k the encoding of the pigeon-hole problem when $n = k + 1 > 3$. Then, *smodels*$(P_k, \emptyset)$ will call itself recursively at least 2^{k-2} times. By inspection, we see that *smodels*$(P_k, \{p_{i,j}\})$ and *smodels*$(P_k, \{not\ p_{i,j}\})$ are called for some i and j when $k = 3$. For the inductive case notice that if $A \cap \{p_{1,k}, \ldots, p_{n,k}\} = \emptyset$, then $Atleast(P_k, A \cup \{p_{n,k}\})$ is equal to $Atleast(P_{k-1}, A)$ modulo all atoms that handle pigeon n and hole k. Furthermore, under the same assumption *lookahead*$(P_k, A \cup \{not\ p_{n,k}\})$ does not find any conflicts if *lookahead*(P_{k-1}, A) does not. In addition, one can take any atom $p_{i,j}$ instead of $p_{n,k}$ as we can freely rename atoms. Since $Atleast(P_k, A) \subseteq Atleast(P_{k-1}, A)$ for any A for which $A = Atleast(P_{k-1}, A)$, it immediately follows that if *smodels*$(P_{k-1}, \emptyset)$ calls itself recursively at least $2^{(k-1)-2}$ times, then *smodels*$(P_k, \emptyset)$ calls itself recursively at least 2^{k-2} times.

Since *smodels*(P, A) solves an NP-complete problem it can also solve coNP-complete problems. In fact, since it can go through all stable models of a logic program, it can solve Δ_2^p-complete problems such as deciding the lexicographically largest stable model.

Proposition 5.2. *Let* $L = \{P \mid P$ *has a stable model and the lexicographically largest one contains the least significant atom*$\}$. *Then,* L *is* Δ_2^p-*complete.*

Proof. Given P we can check whether $P \in L$ using the oracle

$$O = \{\langle P, A\rangle \mid \text{there exist a stable model of } P \text{ that agrees with } A\}$$

a linear number of time. Namely, the language L is decided by the function

function *isinL*(P)
  $A = \emptyset$
  for each atom a in $Atoms(P)$ in order, most significant first **do**
    if there exists a stable model of P that agrees with $A \cup \{a\}$ **then**
      $A := A \cup \{a\}$
    else
      $A := A \cup \{not\ a\}$
    end if
  end for
  if the least significant atom is in A **then**
    return true
  else
    return false
  end if.

Hence, $L \in \Delta_2^p$. Deciding the least significant atom of the lexicographically largest satisfying assignment of a Boolean formula is Δ_2^p-complete [27]. Thus, L is also Δ_2^p-complete. □

5.1 Function Problem Complexity

As noted in [49] any optimization problem, whose decision version is in NP, is in $\mathrm{FP}^{\mathrm{NP}}$. Hence, finding an optimal stable model of a logic program is in $\mathrm{FP}^{\mathrm{NP}}$. In fact, finding an optimal stable model is $\mathrm{FP}^{\mathrm{NP}}$-complete.

Theorem 5.3. *Let P be a program containing at least one minimize statement. Let $\leq_P$ order the stable models of P according to the minimize statements in P. Then, the problem* Opt*: given a logic program P, find a stable model S of P such that for any other stable model S' of P, $S \leq_P S'$, is $\mathrm{FP}^{\mathrm{NP}}$-complete.*

Proof. We begin by showing that the problem is in $\mathrm{FP}^{\mathrm{NP}}$. Assume without loss of generality that all weights are positive. If m is the minimize statement

$$\textit{minimize}\,\{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \textit{not }\, b_1 = w_{b_1}, \ldots, \textit{not }\, b_m = w_{b_m}\},$$

then define

$$w(m) = \sum_{i=1}^{n} w_{a_i} + \sum_{i=1}^{m} w_{b_i}.$$

Furthermore, if S is a set of atoms, then define

$$w(S,m) = \sum_{a_i \in S} w_{a_i} + \sum_{b_i \notin S} w_{b_i}.$$

Since the oracle

$$\begin{aligned} O = \{\langle P, S, k\rangle \mid\ & \text{there exists a stable model } S \text{ of } P \\ & \qquad\qquad \text{that agrees with } A \text{ such that } w(S,m) \leq k\} \end{aligned}$$

is in NP, the procedure *findOptimal*(P) in Algorithm 15 shows that the problem of finding an optimal stable model of a logic program is in $\mathrm{FP}^{\mathrm{NP}}$. The procedure begins with a binary search for the weight of the optimal model. It then constructs the model using a linear number of oracle calls.

We prove that the problem is $\mathrm{FP}^{\mathrm{NP}}$-hard by noticing that the problem Max-weight Sat can be reduced to Opt. If we are given a set of clauses, each with an integer weight, then Max-weight Sat is the problem of finding the truth assignment that satisfies a set of clauses with the greatest total weight. The problem Max-weight Sat is $\mathrm{FP}^{\mathrm{NP}}$-complete [49].

For each clause $c = a_1 \vee \cdots \vee a_n \vee \neg b_1 \vee \cdots \vee \neg b_m$ with weight w_c, create the rule

$$c \leftarrow \textit{not }\, a_1, \ldots, \textit{not }\, a_n, b_1, \ldots, b_m.$$

For each atom a that appears in a clause create the rule $\{a\} \leftarrow$. Finally, add the maximize statement

$$maximize\ \{not\ c = w_c, \ldots\}.$$

The maximal stable model of this program is the truth assignment of greatest total weight. $\square$

We note that if a logic program contains one minimize statement, then we can determine the weight of the lightest stable model using a logarithmic number of oracle calls.

function *findOptimal*(P)
  $A = \emptyset$
  if there is no stable model of P **then**
    return false
  end if
  for each minimize statement m in P in order, most significant first **do**
    $u := w(m)$
    $l := 0$
    while $l < u$ **do**
      if there exists a stable model S of P that agrees with A such that $w(S, m) \leq \lfloor (u + l)/2 \rfloor$ **then**
        $u := \lfloor (u + l)/2 \rfloor$
      else
        $l := \lfloor (u + l)/2 \rfloor + 1$
      end if
    end while
    for each literal x in m **do**
      if there exists a stable model S of P that agrees with $A \cup \{x\}$ such that $w(S, m) = l$ **then**
        $A := A \cup \{x\}$
      else
        $A := A \cup \{not\ (x)\}$
      end if
    end for
  end for
  for each atom a in $Atoms(P) - Atoms(A)$ **do**
    if there exists a stable model of P that agrees with $A \cup \{a\}$ **then**
      $A := A \cup \{a\}$
    else
      $A := A \cup \{not\ a\}$
    end if
  end for
  return A^+.

Algorithm 15: Proof that finding an optimal stable model is in $\mathrm{FP}^{\mathrm{NP}}$

6 Comparison with other Algorithms

We compare the *smodels* algorithm with the Davis-Putnam procedure and with some algorithms for computing stable models of logic programs. We begin with the Davis-Putnam procedure.

6.1 The Davis-Putnam Procedure

The Davis-Putnam (-Logemann-Loveland) procedure [10] for determining the satisfiability of propositional formulas in conjunctive normal form has several similarities to the *smodels* algorithm. We compare the two methods and highlight their differences.

The main difference between the methods comes from the different semantics of the underlying problem. The stable model semantics requires that a model is grounded. Hence, the program

$$a \leftarrow a$$

has only one stable model, the empty set, while the corresponding propositional formula

$$\neg a \vee a$$

has two models: $\{a\}$ and the empty set. All other dissimilarities are the result of this fundamental disparity.

The Davis-Putnam procedure can prune its search space in three ways: by the subsumption of clauses, by using the pure literal rule, and by unit propagation.

If all truth-assignments that satisfy a clause c also satisfy another clause c', then c subsumes c'. For instance, $a \vee b$ subsumes $a \vee b \vee \neg c$. Subsumed clauses can therefore be removed from a formula without changing the set of models of the formula. The *smodels* algorithm has presently no corresponding way of pruning the search space. However, testing for subsumed clauses is expensive and state-of-the-art satisfiability checkers only perform subsumption in a preprocessing step.

The pure literal rule removes any clauses containing a literal whose complement do not appear in any clause. This corresponds to setting the literal to true in a truth-assignment. For example, as the literal $\neg a$ does not appear in the formula

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

the pure literal rule makes a true and removes both clauses yielding the model $\{a\}$. Hence, the pure literal rule can discard models, and in this case it discards the models $\{b\}$, $\{a, b\}$, and $\{a, b, c\}$. Since *smodels* does not discard models, it does not make use of any similar rule. Apart from this, it is not clear what form a rule that does not preserve all models would have under the stable model semantics. The presence of maximize and minimize statements further complicates matters.

Unit propagation consists of unit resolution and unit subsumption. Unit resolution removes all literals that are false from every clause and unit subsumption removes all clauses that contain a literal that is true. Since a unit is a clause of length one, it defines the truth value of a literal and this is the reason for the name. In *smodels* unit propagation coincides with forward propagation and with backward propagation of rules with false heads. Making heads without active rules false and propagating rules with true heads backwards have no correspondence in the Davis-Putnam procedure. Similarly, the upper closure is unique to *smodels*.

Lookahead is in part used by modern satisfiability checkers. They typically only employ lookahead on a small subset of all possible atoms and they do not avoid testing literals that follow from lookahead tests of other literals. Their heuristics take advantage of the lookahead that they do perform, but not in the search space minimizing form of *smodels*.

6.2 Stable Model 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 [65]. We then examine the branch and bound algorithm of [63], the SLG system of [7], the mixed integer programming methods of [2], the modified Davis-Putnam method of [12], and the `dlv` system of [18]. Finally, we end with a comparison with *smodels*.

The Well-founded Semantics

Since the definition of the well-founded semantics [65] is somewhat complicated, we make use of an equivalent definition that suits our needs better. Let

$$\Gamma_P(A) = \mathit{lfp}\left(\bigcup_{r\in P} f_r(A,\cdot)\right)$$

for any normal logic 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 ($\mathit{lfp}\,(\cdot)$ and $\mathit{gfp}\,(\cdot)$) of the monotonic operator

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

Notice that

$$\Gamma_P(A) = \mathit{Atmost}(P, A).$$

According to [1], the well-founded model W of a program P is defined as follows:

$$\begin{aligned} W^+ &= \mathit{lfp}(\Gamma_P^2) \\ W^- &= \mathit{Atoms}(P) - \mathit{gfp}(\Gamma_P^2). \end{aligned}$$

Proposition 6.1. *If W is the well-founded model of the normal logic program P, then $W \subseteq \mathit{expand}(P, \emptyset)$.*

Proof. Let $B = \mathit{expand}(P, \emptyset)$. Then,

$$\mathit{Atoms}(P) - B^- = \mathit{Atmost}(P, B) \subseteq \mathit{Atmost}(P, B^+).$$

Consequently,

$$\begin{aligned} \Gamma_P^2(B^+) &= \Gamma_P\big(\mathit{Atmost}(P, B^+)\big) \\ &\subseteq \Gamma_P\big(\mathit{Atmost}(P, B)\big) \\ &= \Gamma_P\big(\mathit{Atoms}(P) - B^-\big). \end{aligned}$$

In addition, $\mathit{Atleast}(P, B) = B$ implies

$$\mathit{min}_r(B) = \bigcap_{\substack{B^+ \subseteq C \\ B^- \cap C = \emptyset}} f_r(C, C) \subseteq B^+.$$

As $B^- \cap C = \emptyset$ implies $C \subseteq \mathit{Atoms}(P) - B^-$ (we can assume $C \subseteq \mathit{Atoms}(P)$),

$$f_r\big(\mathit{Atoms}(P) - B^-, C\big) \subseteq f_r(C, C),$$

and as $B^+ \subseteq C$,

$$f_r\big(\mathit{Atoms}(P) - B^-, B^+\big) \subseteq f_r(C, C).$$

It follows that

$$f_r\big(\mathit{Atoms}(P) - B^-, B^+\big) \subseteq \mathit{min}_r(B) \subseteq B^+.$$

Hence,

$$\bigcup_{r \in P} f_r\big(\mathit{Atoms}(P) - B^-, B^+\big) \subseteq B^+$$

and therefore

$$\mathit{lfp}\left(\bigcup_{r \in P} f_r\big(\mathit{Atoms}(P) - B^-, \cdot\big)\right) \subseteq B^+.$$

Thus,

$$\Gamma_P\big(Atoms(P) - B^-\big) = Atmost(P, Atoms(P) - B^-) \subseteq B^+,$$
$$\Gamma_P^2(B^+) \subseteq B^+,$$

and

$$lfp(\Gamma_P^2) \subseteq B^+.$$

Let $A = lfp(\Gamma_P^2)$. Then,

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

Hence, $\Gamma_P(A)$ is a fixed point of Γ_P^2 and

$$\Gamma_P(A) \subseteq gfp(\Gamma_P^2).$$

Now,

$$Atoms(P) - gfp(\Gamma_P^2) \subseteq Atoms(P) - \Gamma_P(A) \subseteq Atoms(P) - \Gamma_P(B^+) \subseteq B^-$$

as $Atoms(P) - B^- \subseteq \Gamma_P(B^+)$ and we have proved that the well-founded model of P is a subset of B. $\square$

In fact, the well-founded model $W = expand(P, \emptyset)$ [57]. Notice that since $expand(P, A)$ can return a conflicting set of literals if $A \neq \emptyset$, it provides a stronger pruning technique than the well-founded semantics. For example, consider the program

$$P = \{a \leftarrow b,\ b \leftarrow not\ c,\ c \leftarrow not\ d,\ d \leftarrow not\ b\}$$

and make the assumption that we are looking for a stable model containing the atom a. A typical reduction of P with respect to $\{a\}$ would not change P, and an application of the well-founded semantics produces the empty set, but not the fact that there are no stable models containing a. The function call $expand(P, \{a\})$ on the other hand returns a set containing both a and $not\ a$, thereby explicitly showing that there are no models that include a.

Branch and Bound

The branch and bound algorithm of [63] computes the stable models of a ground logic program in two stages. In the first stage the logic program is simplified while the algorithm computes the well-founded semantics of the program. In the second stage, the branch and bound stage, all stable models of the logic program are constructed.

Starting with the simplified program the algorithm computes all 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 does not belong to the stable model that is being constructed. 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 that has already been 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.

The SLG System

The SLG system [7] supports goal-oriented query evaluation of logic programs under the well-founded semantics. It simplifies a logic program during the query evaluation and produces a residual program. If all negated literals in the residual program are ground, then the system can compute stable models using an assume-and-reduce algorithm.

The assume-and-reduce algorithm constructs the stable models of a logic program in a fashion similar 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 a partially constructed stable model using the well-founded semantics. Instead it derives truth values by repeatedly reducing the program. An atom that appears in the head of a rule with an empty body is assumed to be in the stable model and an atom that does not appear in any head is assumed to not be in the stable model. In addition, 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.

The Mixed Integer Programming Approach

The mixed integer programming methods of [2] computes the stable models of a logic program by translating the program into an integer linear program, which is then used to compute all subset minimal models of the logic program. The models are subsequently tested by another integer linear program and models that are not stable are removed. As the number of minimal models can be very large compared to the number of stable models and as the minimal models must be stored, we conclude that this approach is very inefficient.

Since a logic program can be encoded as a satisfiability problem [3] and since a satisfiability problem can easily be encoded as an integer linear program, it is possible to compute the stable models of a logic program using only one integer linear program. However, it would hardly be reasonable to actually utilize such a complex encoding.

The Modified Davis-Putnam Method

In the modified Davis-Putnam method of [12], a logic program is translated into a set of clauses in such a way that the propagation rules of the Davis-Putnam procedure can deduce as much as the $Atleast(P, A)$ function of *smodels*. In particular, the translation needs a literal for every atom and rule in the logic program.

In addition, the Davis-Putnam procedure is modified such that it only branches on literals that correspond to rules of the original program. Furthermore, the branch points are chosen such that any model that the procedure finds is a stable model.

The DeReS system [8], which implements default logic, is another system that branches on rules instead of on atoms. It does not prune its search space much.

The `dlv` system

The `dlv` system of [18] is a knowledge representation system that uses disjunctive logic. When it searches for stable models of normal logic programs, it prunes the search space using the propagation rules of the $Atleast(P, A)$ function. It is therefore the system that is most similar to *smodels*.

Comparison

We compare the branch and bound algorithm, the SLG system, the mixed integer programming approach, the modified Davis-Putnam method, and the `dlv` system 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. 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

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

has one stable model but 2^n minimal models.

The modified Davis-Putnam method and the `dlv` system do not prune the search space using the upper closure. Hence, they also prune less than *smodels*. It would be quite easy to integrate the upper closure computation into the `dlv` system as it is similar to *smodels*. Including the upper closure computation into the modified Davis-Putnam method would probably be a lot harder.

We conclude that even without lookahead the *smodels* algorithm prunes the search space significantly more than the branch and bound algorithm, the mixed integer programming approach, and the SLG system. Moreover, the *smodels* algorithm also prunes the search space more than the modified Davis-Putnam method and the `dlv` system. As the pruning in the *smodels* algorithm can be efficiently implemented, *smodels* will compute stable models faster than the other systems if it is given a program that requires some search.

7 Experiments

In order to demonstrate the performance of *smodels*, we test an implementation, *smodels* version 2.23 [60], on some combinatorially hard problems. Since there are no competitive systems for computing stable models of logic programs, we compare *smodels* with some propositional satisfiability checkers. The one stable model system that can approach the same magnitude of performance as *smodels* is `dlv` [28], and we also compare against it. The intention of the tests is to assess *smodels* in relation to other general purpose systems, not to compare it with a different special purpose algorithm for each problem.

Each test instance is randomly shuffled and run ten times. We shuffle the test instances since a particular ordering might help the algorithms to avoid backtracking, thereby giving a skewed picture of their behavior. That is, we try to lessen the impact of lucky choices. The durations of the tests are given in seconds and they represent the time to find a solution or the time to decide that there are no solutions. They include the time it takes to read the input and write the result. 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 under Linux 2.2.12 on 450 MHz Pentium III computers with 256 MB of memory.

7.1 3-SAT

We compare *smodels* with three propositional satisfiability checkers: tableau or *ntab* [9], *SATO* 3.2 [68], and *satz* [31], and with `dlv`. The test domain is random 3-SAT, i.e., randomly generated propositional formulas in conjunctive normal form whose clauses contain exactly three literals. The problems are chosen such that the clause to atom ratio is $4.258 + 58.26a^{-5/3}$, where a is the number of atoms, since this particular ratio determines a region of hard satisfiability problems [9].

The three satisfiability checkers are all variants of the Davis-Putnam procedure. *SATO* strengthens the procedure by adding clauses to the problem during the search. Every time *SATO* arrives at a contradiction it stores the negation of the choices that led to the contradiction in a new clause. If the length of the clause is less than 20, then it is added to the set of clauses. This approach runs into problems if the number of added clauses grows too big.

Tableau or *ntab* and *satz* both perform lookahead on a subset of all available atoms, but *satz* uses a more sophisticated heuristic, which is also used to measure the hardness of the problem during the search. If the problem is hard, then *satz* performs lookahead on all atoms. The program *satz* also does some preprocessing on the satisfiability problems during which it adds clauses to the problem.

We translate a 3-SAT problem into a logic program as follows. The truth-assignments of the atoms $a_1, \ldots, a_n$ of the problem are encoded by a choice rule

$$\{a_1, \ldots, a_n\} \leftarrow$$

and each clause

$$a_1 \vee \cdots \vee a_n \vee \neg b_1 \vee \cdots \vee \neg b_m$$

is translated into a rule

$$\mathit{false} \leftarrow \mathit{not}\ a_1, \ldots, \mathit{not}\ a_n, b_1, \ldots, b_m.$$

Since the clauses have to be satisfied, we deny the inclusion of the atom *false* in the stable models using the statement

$$\mathit{compute}\ \{\mathit{not}\ \mathit{false}\}.$$

When we test `dlv`, we use the rules $a_i \leftarrow \mathit{not}\ \bar{a}_i$ and $\bar{a}_i \leftarrow \mathit{not}\ a_i$, for $i = 1, \ldots, n$, instead of the choice rule.

We test the implementations on problems having from 150 to 400 atoms. For each problem size we generate ten satisfiability problems using a program developed by Bart Selman [56]. Every problem is randomly shuffled and tested ten times. The test results are shown in Figures 7–8. The number of choice points is not available for `dlv`.

The implementation of *smodels* prunes the search space more than *SATO* and *ntab*, but less than *satz*. In addition, *SATO* prunes the search space least of all. This indicates, as one would expect, that doing lookahead substantially reduces the search space. Since *smodels* is not as good at pruning the search space as *satz*, it seems that the heuristic of *satz* is better than that of *smodels*.

There is, however, a difference between how *smodels* and *satz* propagate truth values. The *satz* procedure makes use of the pure literal rule. A literal is pure if its complement does not appear in any clauses. Hence, if a set of clauses is satisfiable, then one can remove all clauses containing pure literals and the remaining set of clauses will still stay satisfiable. Since *smodels* is designed such that it can compute all stable models of a program, it does not take advantage of this reduction.

The heuristic of *satz* is similar to that of *smodels*. The heuristic of *smodels* measures how the set of undefined literals changes when we fix the truth value of an atom, while the heuristic of *satz* measures how the set of clauses changes when we fix the truth value of an atom. Let p and n be a measure of the change when the truth value of an atom is set to respectively

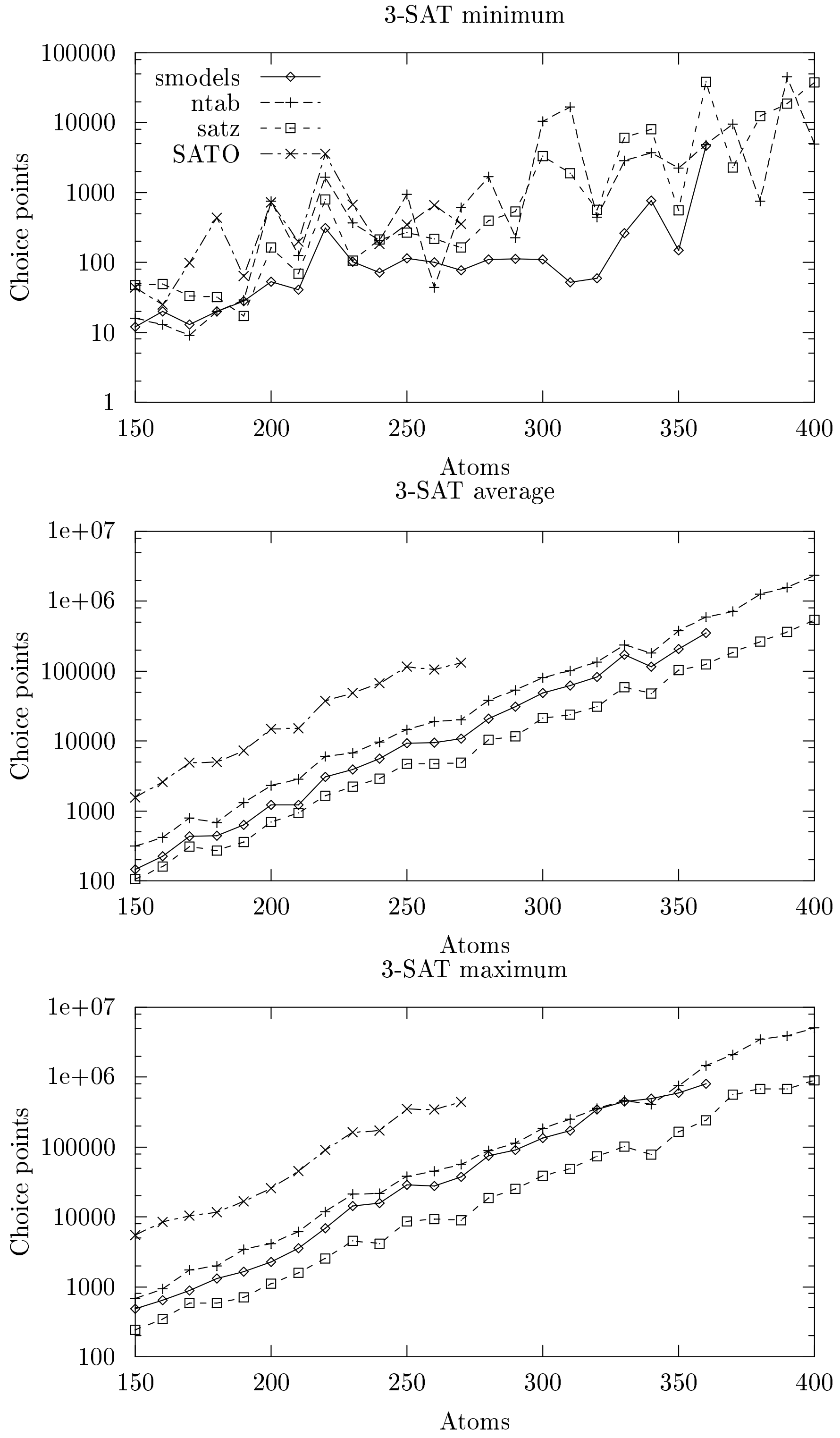


Figure 7: 3-SAT, number of choice points

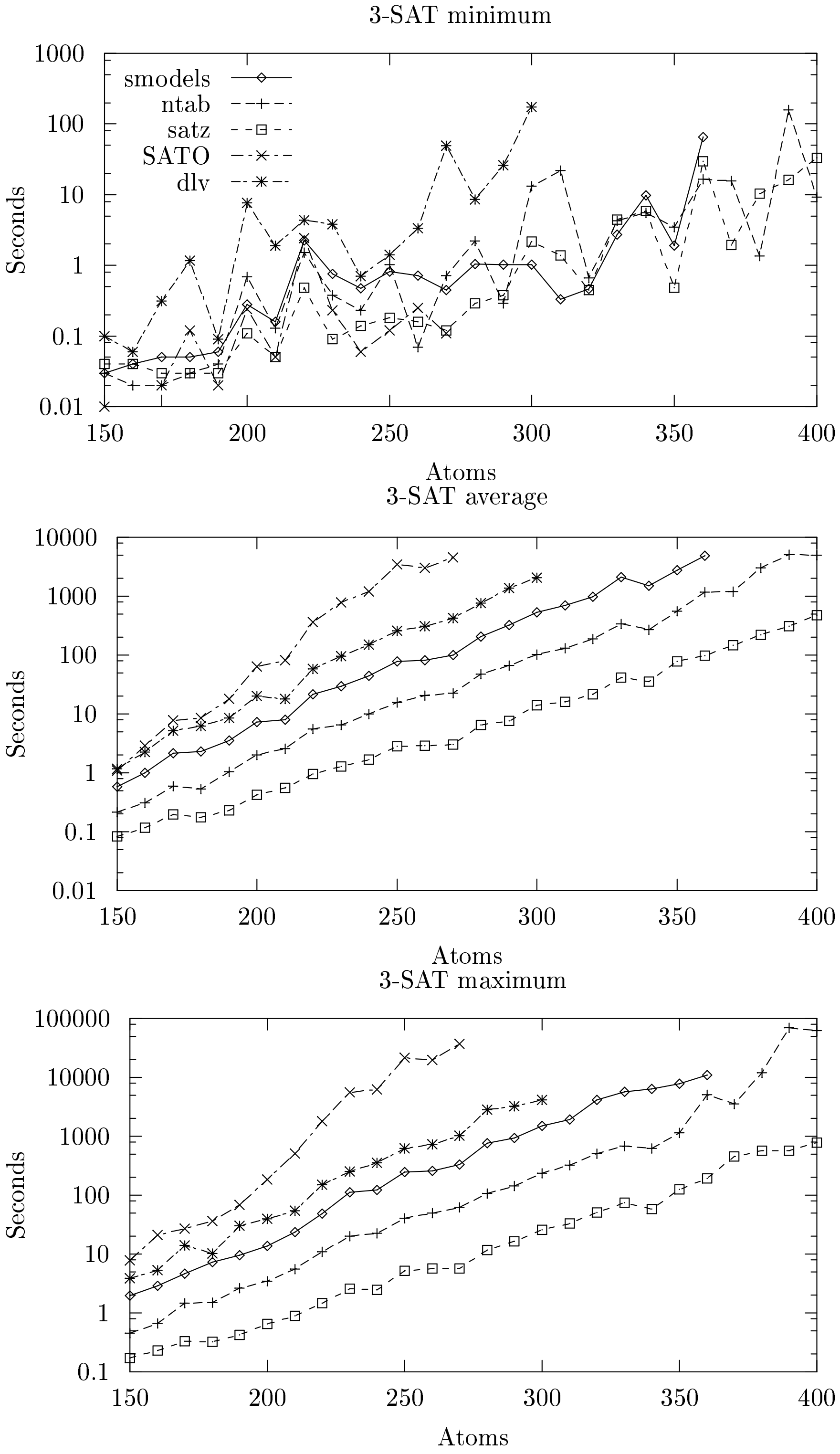


Figure 8: 3-SAT, duration in seconds

true or false. Then, *smodels* maximizes $\min(n,p)$ and $\max(n,p)$, where $\min(n,p)$ is more significant, and *satz* maximizes

$$1024np + n + p.$$

We refer to these formulas as the cost functions of *smodels* and *satz*. In order to decide whether it is the heuristic of *satz* that is the reason for the better pruning, we have tested the following variants of *smodels* and *satz* on the 3-SAT problems: *satz* without the preprocessing step, *satz* without the pure literal rule, *satz* with the cost function of *smodels* measuring literals and measuring clauses, *smodels* with the pure literal rule, *smodels* with the cost function of *satz*, and *smodels* with both the pure literal rule and the cost function of *satz*. The result is that the variants prune the search space as much or slightly less than the original versions. We therefore conclude that the heuristic of *satz* is better than that of *smodels* on hard 3-SAT problems.

We will use the fact that we shuffled and tested each satisfiability problem ten times, to measure how tightly the heuristics define which literal to try next. If a heuristic is lax, then it is possible to refine it such that it breaks the ties between the literals in a better way. For each satisfiability problem we compute the ratio between the largest and smallest number of choice points needed to solve a randomly shuffled instance of the problem. The results are shown in Figure 9.

Clearly, *satz* has a very tight and *smodels* a very lax heuristic.

7.2 Pigeon-hole Problems

We test *smodels*, *satz*, and *SATO* on some pigeon-hole problems, where the number of pigeons is one more than the number of holes. The *SATO* program generates the satisfiability problems for itself and *satz*. Since the generated problems are not 3-SAT problems and since *ntab* does not handle clauses of length greater than three, we do not test *ntab*. We encode the pigeon-hole problems for *smodels* as in Section 5. When we test `dlv`, we replace a cardinality rule of the form

$$\mathit{false} \leftarrow 2\,\{p_1, \ldots, p_n\}$$

with the basic rules

$$\mathit{false} \leftarrow p_{i_1}, p_{i_2}, p_{i_3}, \qquad\qquad 1 \le i_1 < i_2 < i_3 \le n.$$

Similar translations are used in the following tests and will not be mentioned. The results are shown in Figures 10–11.

We notice that *smodels* and *satz* explore very similar sized search spaces. If we examine the durations of the tests, we see that the overhead of *smodels* is smaller than it was on the 3-SAT problems. This is explained by the compact encoding of the pigeon-hole problem. The program *SATO* experiences a strange improvement when there are ten pigeons.

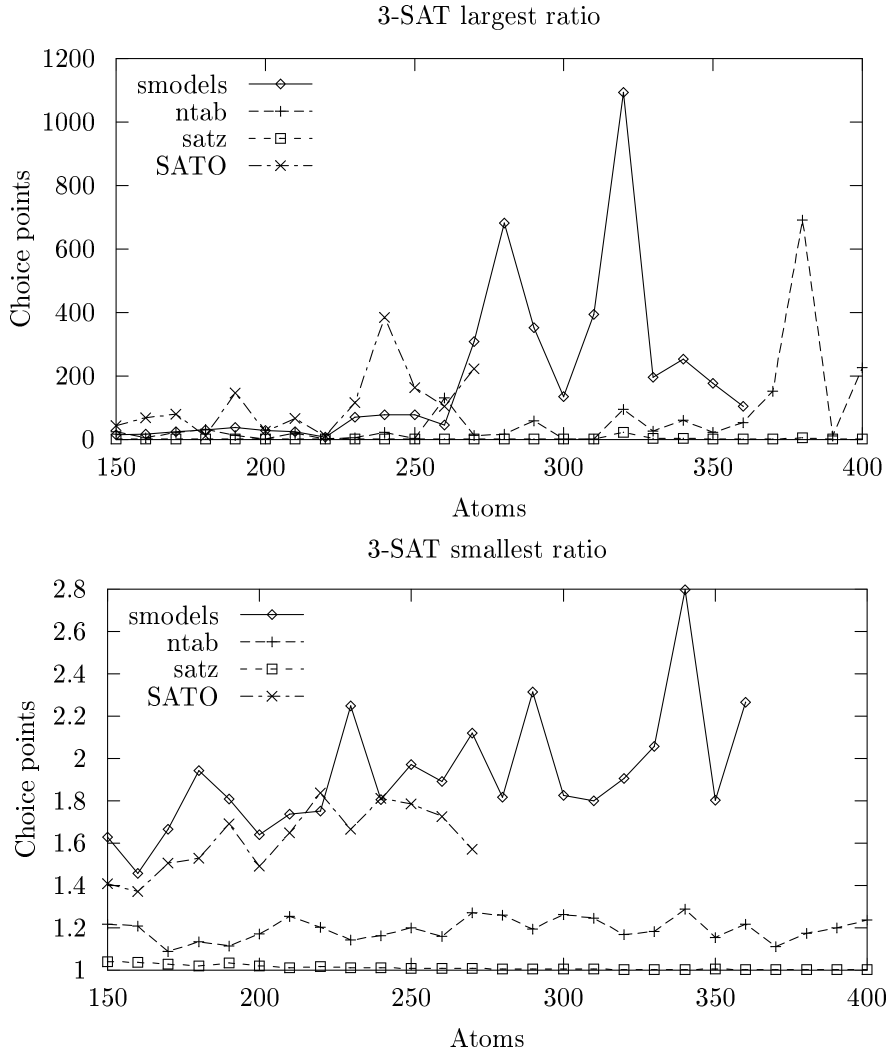


Figure 9: 3-SAT, choice points max/min ratio

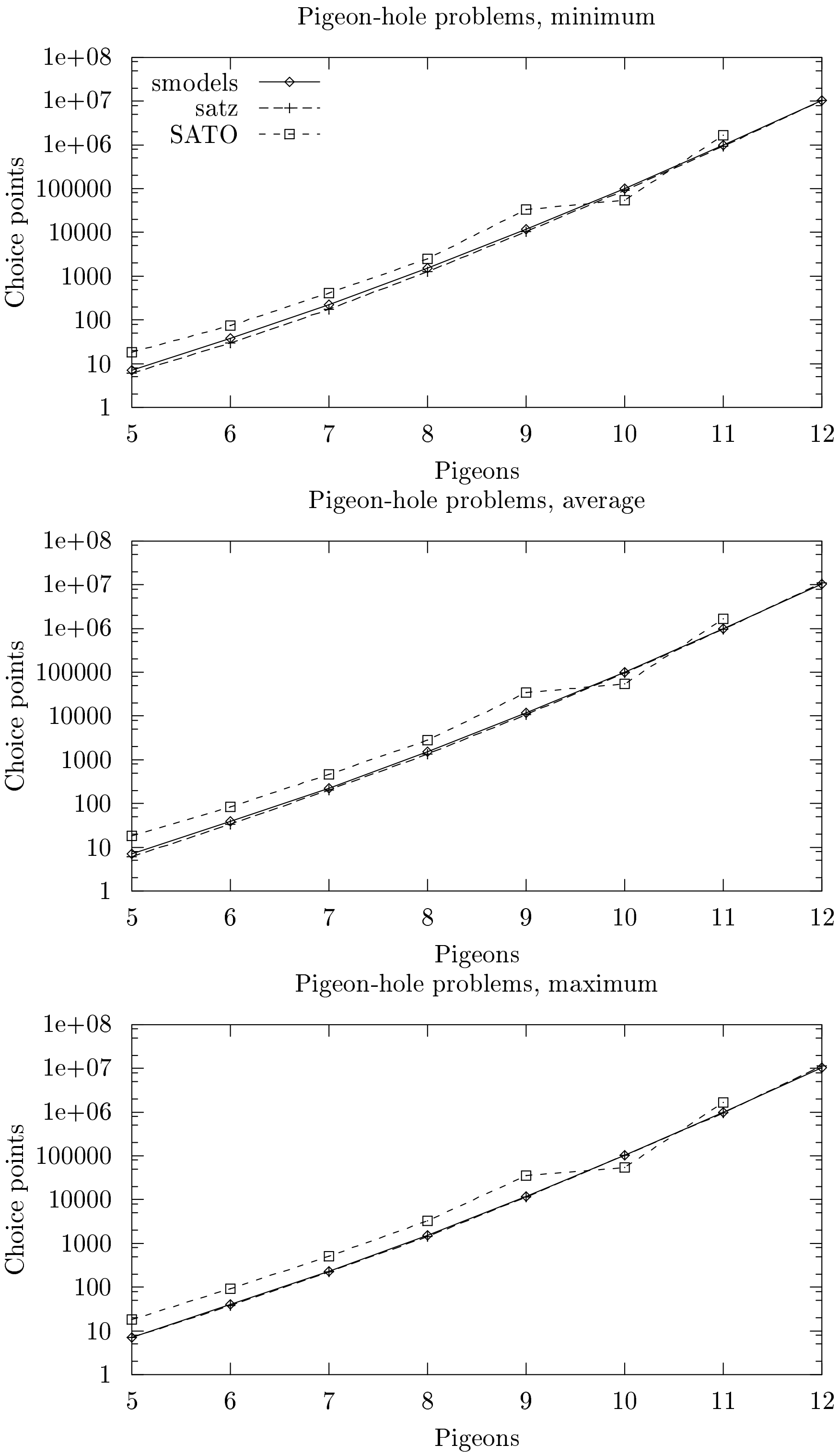


Figure 10: Pigeon-hole problem, number of choice points

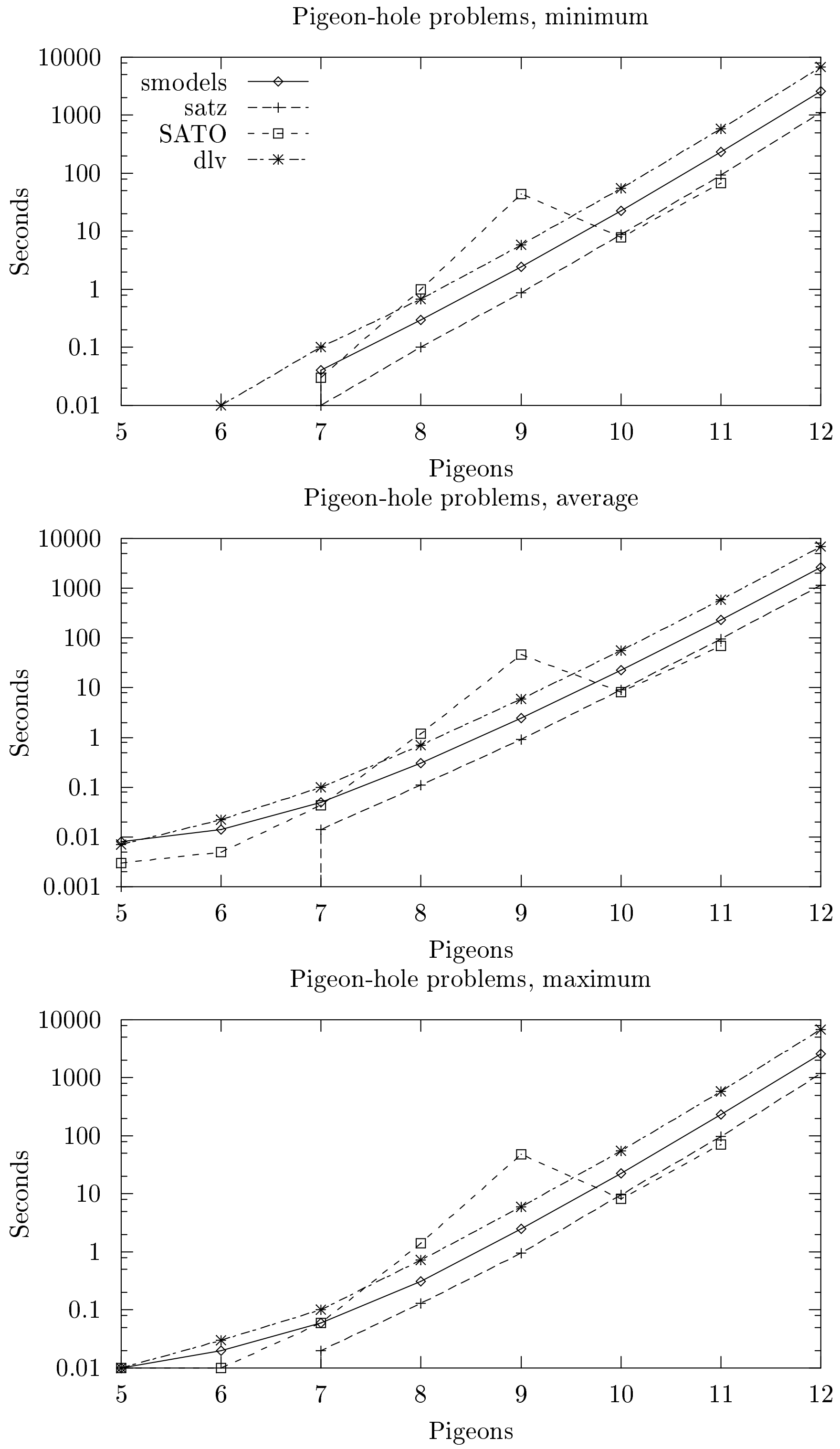


Figure 11: Pigeon-hole problem, duration in seconds

7.3 Hamiltonian Cycles

A Hamiltonian cycle is a cycle in a graph such that the cycle visits each node precisely once. If $G = (V, E)$ is an undirected graph, then we encode the Hamiltonian cycles of G in a program as follows. For each edge $\{v, w\} \in E$ we make an atom $e_{v,w}$. Since the graph is undirected, we do not distinguish between $e_{v,w}$ and $e_{w,v}$. The idea of the encoding is that if $e_{v,w}$ is in a stable model, then $\{v, w\}$ participates in a Hamiltonian cycle.

We notice that if v is a vertex of a cycle, then there are exactly two edges incident to v that are in the cycle. Hence, we create the three rules

$$
\begin{array}{rll}
\{e_{v,w_1}, \ldots, e_{v,w_n}\} \leftarrow & & \{v, w_i\} \in E \\
& \mathit{false} \leftarrow 3\,\{e_{v,w_1}, \ldots, e_{v,w_n}\} & \{v, w_i\} \in E \\
& \mathit{false} \leftarrow n-1\,\{\mathit{not}\ e_{v,w_1}, \ldots, \mathit{not}\ e_{v,w_n}\} & \{v, w_i\} \in E
\end{array}
$$

for every vertex v of G. We must now avoid stable models that contain more than one cycle. First we pick an arbitrary vertex $v_1 \in V$ and then we create the rule

$$v_1 \leftarrow$$

and the rules

$$w \leftarrow v, e_{v,w} \qquad \{v, w\} \in E \text{ and } w \neq v_1.$$

It follows that a vertex v is in a stable model if and only if v and v_1 are in the same cycle. Hence, we can force the stable models to contain exactly one cycle by including the compute statements

$$\mathit{compute}\ \{v_1, \ldots, v_n\} \qquad v_1, \ldots, v_n \in V$$

and

$$\mathit{compute}\ \{\mathit{not}\ \mathit{false}\}.$$

We translate the Hamiltonian cycle problem into clauses following Papadimitriou [49]. The translation encodes a total order on the vertices of the graph such that there is an edge between any pair of vertices that are adjacent in the order. Given the vertices $v_1, \ldots, v_n$, we use the atom $v_{i,j}$ to encode that vertex v_i is in position j using the clauses:

$v_{i,1} \vee \cdots \vee v_{i,n}$		v_i is in some position,
$\neg v_{i,j} \vee \neg v_{i,k}$	$j \neq k$	v_i is in at most one position,
$v_{1,j} \vee \cdots \vee v_{n,j}$		some vertex is in position j, and
$\neg v_{i,k} \vee \neg v_{j,k}$	$i \neq j$	v_i and v_j are not in the same position.

Then we deny orders that do not correspond to Hamiltonian cycles:

$$\neg v_{i,k} \vee \neg v_{j,k+1 \bmod n} \quad \{v_i, v_j\} \notin E.$$

Hence, if the graph has n vertices, then we need n^2 atoms and on the order of n^3 clauses.

One can also create a more complex Hamiltonian cycle translation that mirrors the logic programming translation. For a graph G and for each edge $\{v, w\} \in E$, we make an atom $e_{v,w}$. As before, we do not distinguish between the atoms $e_{v,w}$ and $e_{w,v}$. For a vertex $v \in V$, let the incident edges be $\{v, w_1\}, \ldots, \{v, w_n\}$. We force the inclusion of at least two incident edges by creating the clauses

$$e_{v,w_{i_1}} \vee \cdots \vee e_{v,w_{i_k}} \qquad 1 \leq i_1 < i_2 < \cdots < i_k \leq n,\ k = n-1$$

and we deny the inclusion of more than two edges by creating the clauses

$$\neg e_{v,w_{i_1}} \vee \neg e_{v,w_{i_2}} \vee \neg e_{v,w_{i_3}} \qquad 1 \leq i_1 < i_2 < i_3 \leq n$$

We avoid multiple cycles by picking a vertex and demanding that there is a path to every other vertex. For every vertex v, we create an atom v^k, for $k = 0, \ldots, n$, that denotes that v is reachable through a path of length k. We keep the final conjunctive normal form encoding small by letting an auxiliary atom $t^k_{v,w}$ denote that w is reachable in k steps and that v is one step away from w. Thus, the formulas

$$\begin{array}{ll} v^{k+1} \leftrightarrow t^k_{v,w_1} \vee \cdots \vee t^k_{v,w_n} & k = 0, 1, \ldots, n-1, \\ t^k_{v,w_i} \leftrightarrow e_{v,w_i} \wedge w^k_i & i = 1, \ldots, n, \text{ and} \\ \quad v^1 \vee \cdots \vee v^n & \end{array}$$

ensure that there is a path from some vertex to v. Finally, we pick a vertex v that begins all paths by creating the clauses

$$v^0$$

and

$$\neg w^0 \qquad w \neq v \text{ and } w \in V.$$

We solve the Hamiltonian cycle problem on a special type of planar graphs created by the *plane* function found in the Stanford GraphBase [26]. We generate ten graphs for each problem size and all these graphs have Hamiltonian cycles. The results are displayed in Figures 12–13. We observe that the second satisfiability encoding is better for *SATO* and mostly better

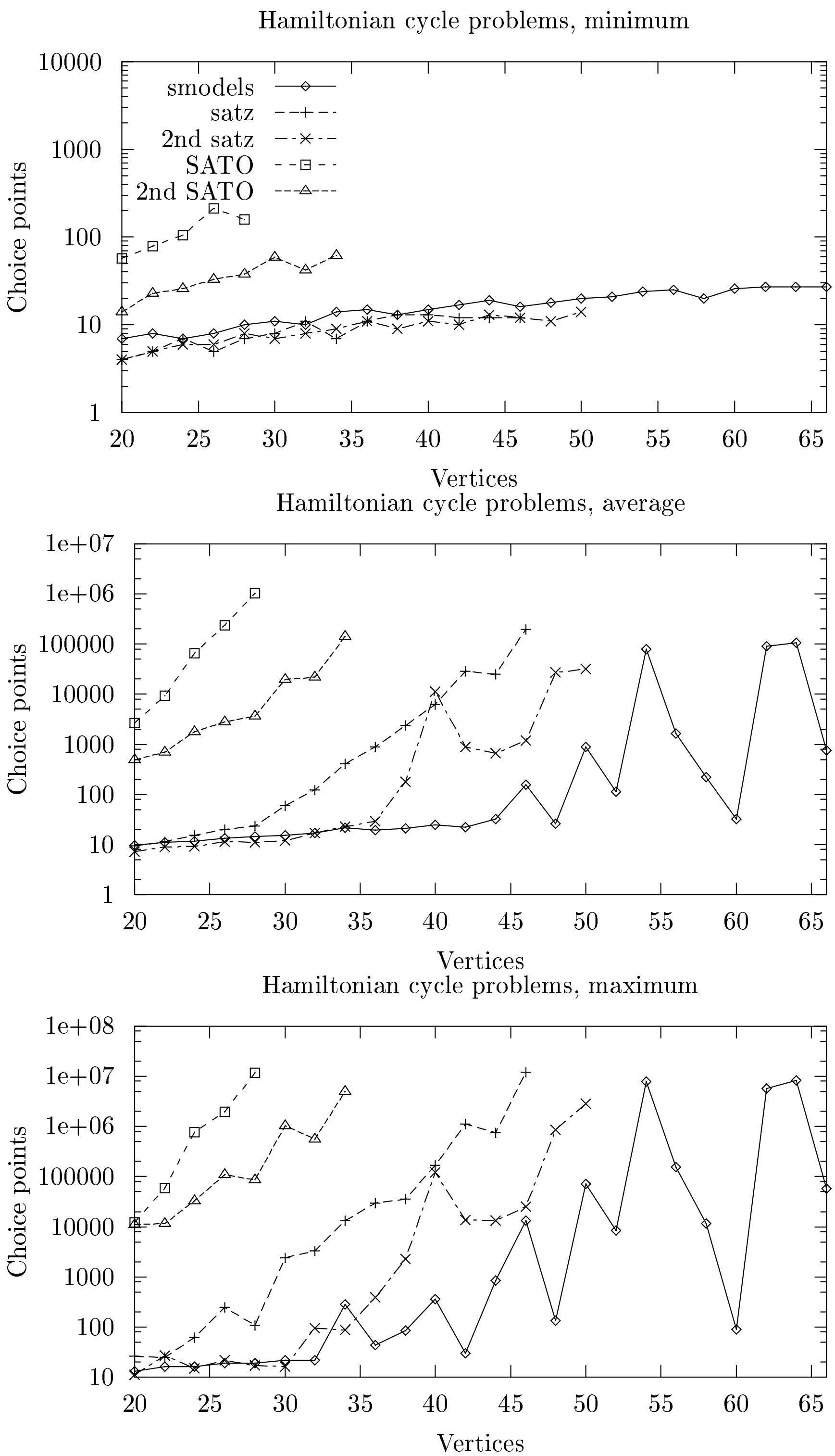


Figure 12: Hamiltonian cycle problem, number of choice points

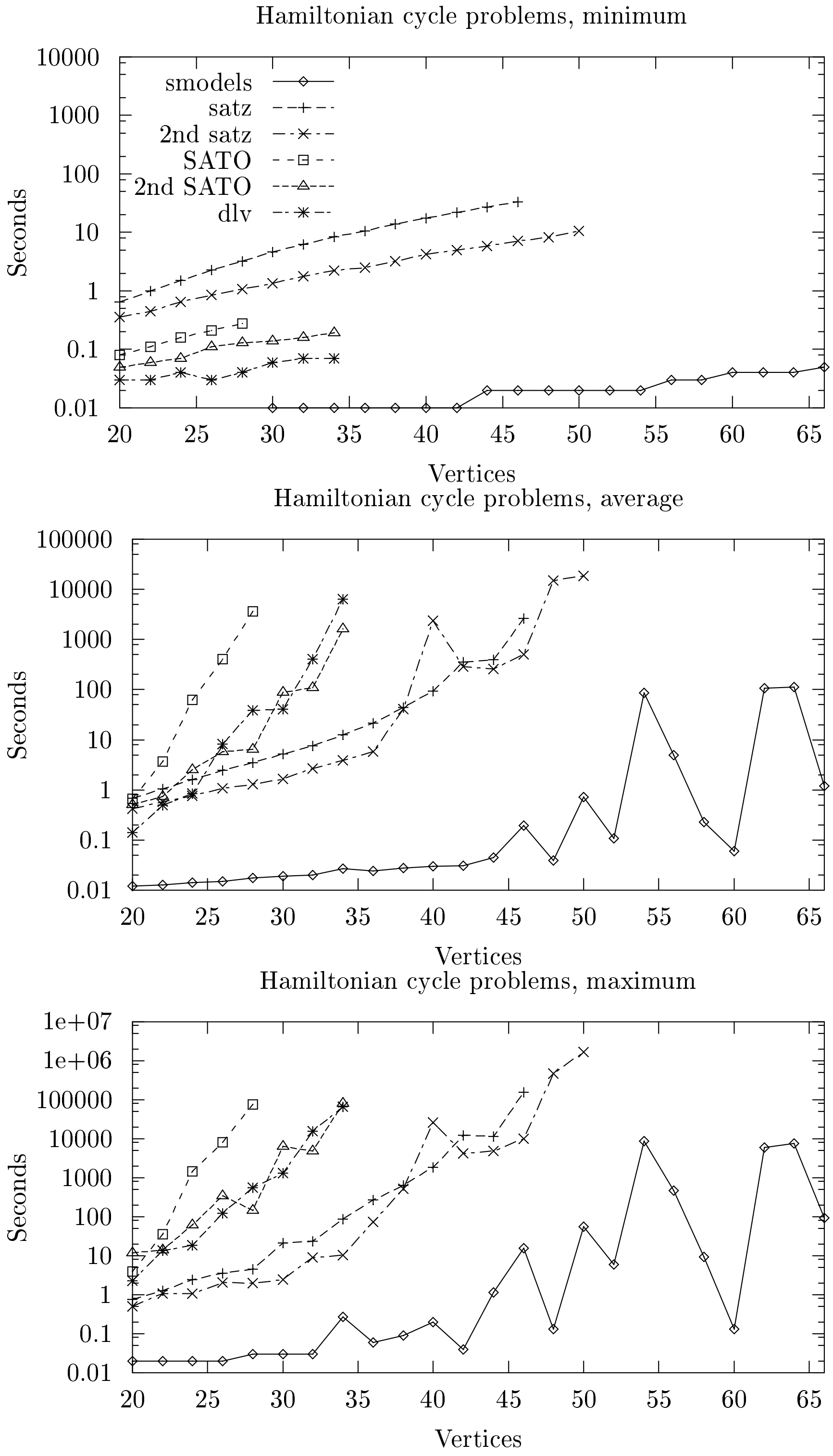


Figure 13: Hamiltonian cycle problem, duration in seconds

for *satz*. Apparently, the heuristic of *satz* does not work well on these problems.

It is precisely problems of this type that provides us with an incentive to use a stable model semantics solver instead of a satisfiability checker. Any problem that one can easily encode as a satisfiability problem we can just as easily encode as a logic program. The converse does not hold.

Consider a mapping T from logic programs to sets of propositional clauses. We say that T is modular if for every program partitioned into two disjoint parts P_1 and P_2, the program $P_1 \cup P_2$ has a stable model if and only if $T(P_1) \cup T(P_2)$ is satisfiable.

Proposition 7.1 (Niemelä [42]). *There is no modular mapping from the class of logic programs to the sets of clauses.*

Proof. Consider the program $P = \{p \leftarrow \mathit{not}\ p\}$ and assume that T is a modular mapping. Then, $T(P)$ is unsatisfiable as P has no stable models. It follows that also $T(P) \cup T(\{p \leftarrow\})$ is unsatisfiable. But this implies that $P \cup \{p \leftarrow\}$ has no stable models, which is clearly not the case. Hence, T is not modular. $\square$

Even if there is no local translation of logic programs into satisfiability problems, there are more complex ones [3].

Example 7.2. The translation from a logic program to a satisfiability problem has two parts. The first part ensures that the rules of the logic program are satisfied and the second part ensures that an atom can not justify its own inclusion in a model. It follows that a model of the translation is grounded and that it corresponds to a stable model of the original logic program.

Each atom in the logic program has a corresponding atom in the satisfiability problem. In addition, there are new atoms that assign indices to these atoms. By requiring that an atom can only be justified by atoms of lower index, we avoid circular justifications.

For example, let P be the program

$$\begin{aligned} p &\leftarrow q \\ q &\leftarrow p. \end{aligned}$$

We translate P into a set of propositional clauses as follows. First, we include the clauses

$$\neg q \vee p \quad \text{and} \quad \neg p \vee q,$$

so that if q is true then also p is true and if p is true then also q is true.

Then, we introduce the new atoms p_1, p_2, q_1, and q_2 whose truth-values decide what indices p and q have. The formulas $p_1 \vee p_2$, $\neg p_1 \vee \neg p_2$, $q_1 \vee q_2$, and $\neg q_1 \vee \neg q_2$ assign precisely one index to p and one to q. Finally, the two formulas

$$\neg p \vee (q \wedge q_1 \wedge p_2) \quad \text{and} \quad \neg q \vee (p \wedge p_1 \wedge q_2)$$

guarantee that p follows from q only if the index of q is lower than that of p and that q follows from p only if the index of p is lower than that of q.

One problem remains. If p is false, then either p_1 or p_2 is true, and we get two models instead of one. Therefore, we set the index of p to 1 when it is false. The same must be done for q. Hence, we include the formulas

$$p \vee p_1 \quad \text{and} \quad q \vee q_1.$$

This satisfiability problem has only one model, $\{p_1, q_1\}$, which corresponds to the empty stable model of P.

Take the program $P = \{p \leftarrow \textit{not } p\}$ from the proof of Proposition 7.1. We can translate P into a satisfiability problem without using any indices as the atom p can not justify its own inclusion in a stable model. The translation consists of two clauses. The clause $p \vee p$ guarantees that the only rule in P is satisfied and the clause $\neg p \vee \neg p$ guarantees that if p is true, then it is justified by the only rule. The two clauses have no models and the program has no stable models.

If we add the rule $p \leftarrow$ to P, then we have to change the translation. The clause $p \vee p$ remains unchanged and we add the clause p to keep $p \leftarrow$ satisfied. The clause $\neg p \vee \neg p$ is replaced by $\neg p \vee \neg p \vee \top$, where $\top$ is a tautology. The new translation has $\{p\}$ as its only model.

It is now clear that the more involved stable model semantics incurs a computational overhead. But in exchange for the overhead we gain a more powerful language. Hence, problems that can be more compactly represented by logic programs can still be more quickly solved with *smodels* than with a satisfiability checker.

7.4 Error-correcting Codes

We will search for sets of binary words of length n such that the Hamming distance between any two words is at least d. The size of the largest of these sets is denoted by $A(n, d)$. For example, $A(5, 3) = 4$ and any 5-bit one-error-correcting code contains at most 4 words. One such code is $\{00000, 00111, 11001, 11110\} = \{0, 7, 25, 30\}$. Finding codes becomes very quickly very hard. For instance, it was only recently proved that $A(10, 3) = 72$ [48].

We construct a program whose stable models are the maximal codes with Hamming distance d. If $j_1, \ldots, j_k$ are the words whose distance to i is positive and less than d, then we create a rule

$$w_i \leftarrow \textit{not } w_{j_1}, \ldots, \textit{not } w_{j_k}$$

for every $i = 0, \ldots, 2^n$. Since a code remains a code even if we swap the zeroes for ones or permute the positions of the bits in every word in the same way, we can restrict ourselves to codes that include the zero word and a word whose d lowest bits are set. Therefore, we create the rules

$$\begin{aligned} w_0 &\leftarrow \\ \mathit{false} &\leftarrow \mathit{not}\ w_{j_1}, \ldots, \mathit{not}\ w_{j_{2^{n-d}}}, \end{aligned}$$

where $j_k = 2^d(k+1) - 1$, and the compute statement

$$\mathit{compute}\ \{\mathit{not}\ \mathit{false}\}.$$

Since we want to search for the largest stable model, we include the maximize statement

$$\mathit{maximize}\ \{w_0, \ldots, w_{2^n}\}.$$

The results are shown in Figure 14.

7.5 Bin-packing

In order to test the weight rules we try *smodels* on some bin-packing problems. The object is to pack a number of items of varying sizes into a number of bins of equal size. We represent the fact that item i is in bin j by the atom $b_{i,j}$, and we assume that the size of item i is given by the positive integer w_i. If we have n items and m bins, then we can distribute the items among the bins using the rules

$$\begin{aligned} \{b_{i,1}, \ldots, b_{i,m}\} &\leftarrow \\ \mathit{false} &\leftarrow 2\,\{b_{i,1}, \ldots, b_{i,m}\} \\ \mathit{false} &\leftarrow \mathit{not}\ b_{i,1}, \ldots, \mathit{not}\ b_{i,m} \qquad i = 1, \ldots, n. \end{aligned}$$

Let the bins have size b. We prevent the bins from containing too many items with the help of the rules

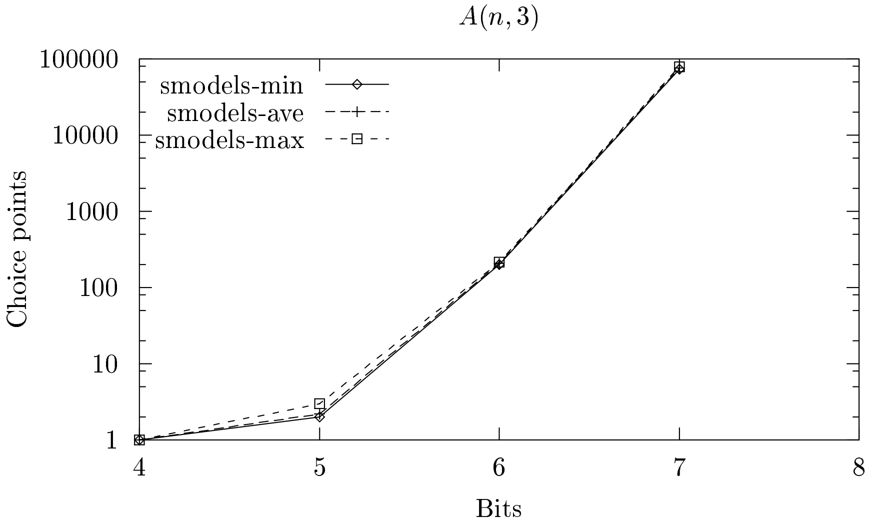
$$\mathit{false} \leftarrow \{b_{1,j} = w_1, \ldots, b_{n,j} = w_n\} \geq b + 1 \qquad j = 1, \ldots, m.$$

Finally, we include the rule

$$\mathit{compute}\ \{\mathit{not}\ \mathit{false}\}$$

as we do not want *false* to be in any stable model.

For our tests we choose the sizes of the items uniformly from the integer interval $1, \ldots, \mathit{max}$ for some number max. A bit of experimentation shows that most bin-packing problems are easy. However, *smodels* have problems when there are many items and the expected total size of all the items



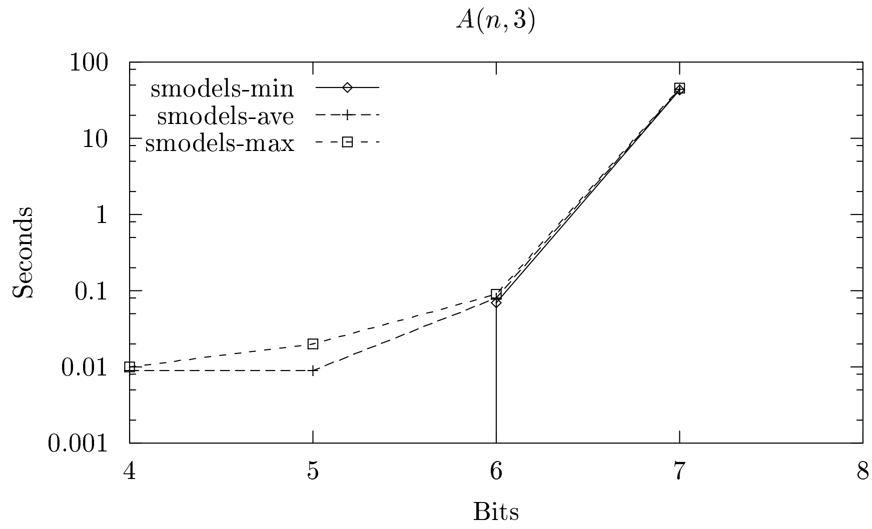


Figure 14: Maximal error-correcting codes

equals the available space in the bins. In hindsight, this is not surprising as it includes the case when the items almost fit.

We solve bin-packing problems with 16 items, bins of size 16, and with the maximum size of the items set to twice the number of bins. The results are shown in Figure 15. As a comparison, a mixed integer linear programming system such as *lp_solve* [4] solves each of these problems in about 0.01 seconds.

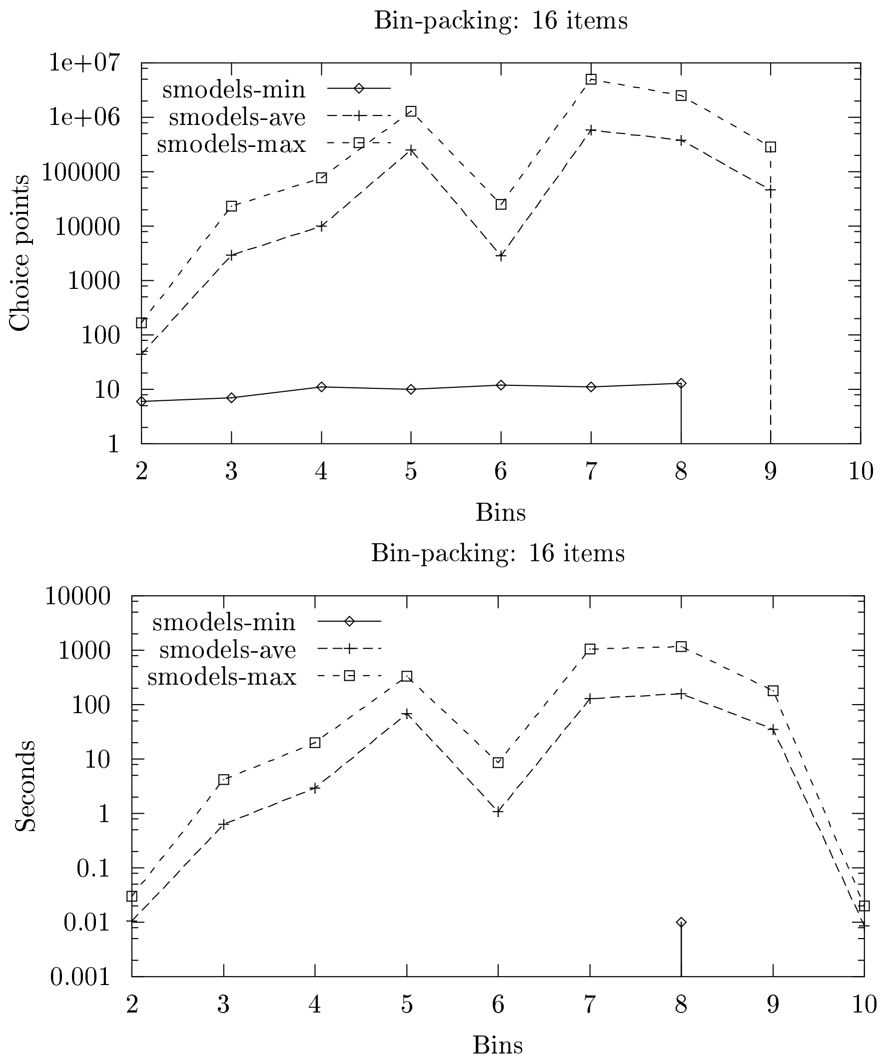


Figure 15: Bin-packing problems

8 Conclusions

We have explored an algorithm that solves the stable model semantics for logic programs. We have shown that it is comparatively easy to extend the semantics and the algorithm to handle new types of rules. We have also shown that one can easily change the algorithm to search for specific stable models such as the lexicographically smallest or largest one.

We have compared the algorithm with three good satisfiability solvers on random satisfiability problems, pigeon-hole problems, and Hamiltonian cycle problems. The best satisfiability solver goes through a smaller search space than our algorithm when testing the random SAT problems. We have attributed the difference to the different heuristics of the procedures, and we have found indications that our heuristic can be refined such that it works better. The algorithm and the solvers behave similarly on the pigeon-hole problems, while our algorithm is significantly better than the solvers when it comes to the Hamiltonian cycle problems. Since the Hamiltonian cycle problem contains more structure, it requires both a good heuristic and full lookahead before it can be solved satisfactorily.

To conclude, the stable model semantics has a computational overhead. But the overhead provides us with a more powerful language. Consequently, problems that can be more compactly represented by logic programs can be more quickly solved with *smodels* than with a satisfiability checker.

8.1 Future Work

Since testing every literal before resorting to a heuristic improves the decision procedure, one can ask if testing every two literals or every set of n literals would improve the algorithm even further. The problem is, of course, the overhead. Ordinary *lookahead* calls *expand* in the worst case a linear number of times. If we test two literals, then we call *expand* a quadratic number of times, and if we test sets of n literals, then the number of *expand* calls is already on the order of the number of atoms to the power of n.

Luckily we can approximate n-*lookahead* if we are prepared to use a quadratic amount of memory. Assume that $a, b \notin \mathit{expand}(P, A)$ and that $b \in \mathit{lookahead}(P, A \cup \{a\})$. Then, any stable model S that agrees with A but do not contain b can not contain a. Hence, if we store the fact that *not* b implies *not* a, then we can use this information to strengthen *expand*. Later calls to *lookahead* will then take advantage of all the relations between the literals that previous calls have found.

The problem is that even a quadratic amount of memory seems to create too much overhead. Two questions are therefore left to future research. Can one avoid the memory overhead by making use of the structure of the programs? Does there exist another approximation that evades this problem?

A Monotone Functions

Let X be a set and let $f : 2^X \to 2^X$ be a function. If $A \subseteq B$ implies $f(A) \subseteq f(B)$, then f is monotonic. We have the following version of the Knaster-Tarski fixpoint theorem.

Lemma A.1. *Let $f : 2^X \to 2^X$ be a monotonic function, and let $A \subseteq X$. If $f(A) \subseteq A$, then $\mathit{lfp}\,(f) \subseteq A$, where $\mathit{lfp}\,(f)$ denotes the least fixed point of f.*

Proof. Define

$$S = \bigcap_{f(A) \subseteq A} A \qquad\qquad \big(f(X) \subseteq X\big).$$

Then, $f(A) \subseteq A$ implies $S \subseteq A$, which in turn implies $f(S) \subseteq f(A)$ by the monotonicity of f. Hence, $f(S) \subseteq A$, and consequently

$$f(S) = \bigcap_{f(A) \subseteq A} f(S) \subseteq \bigcap_{f(A) \subseteq A} A = S.$$

Now, $f(S) \subseteq S$ implies $f\big(f(S)\big) \subseteq f(S)$, which by the definition of S implies $S \subseteq f(S)$. Thus, $S = f(S)$. Moreover, for any fixed point A,

$$f(A) \subseteq A \quad \text{implies} \quad S \subseteq A,$$

and hence $\mathit{lfp}\,(f) = S$ by definition. □

Similarly, $A \subseteq f(A)$ implies $A \subseteq \mathit{gfp}\,(f)$ for the greatest fixed point of f. Notice that if X is finite, then $\mathit{lfp}\,(f) = f^n(\emptyset)$ for some $n \leq |X|$ since $f(\emptyset) \subseteq \mathit{lfp}\,(f)$. Furthermore, observe that if we are given k monotonic functions $f_1, \ldots, f_k$, then the least fixed point of

$$g(A) = \bigcup_{i=1}^{k} f_i(A)$$

is the limit of any nest

$$A_{n+1} = A_n \cup f_{i(n)}(A_n), \quad A_0 = \emptyset \text{ and } f_{i(n)}(A_n) \subseteq A_n \Rightarrow \forall j f_j(A_n) \subseteq A_n.$$

In other words, the least fixed point of g can be computed by repeated applications of $f_1, \ldots, f_k$. Finally, note that $a \in \mathit{lfp}\,(g)$ if and only if there is a sequence $i(0), \ldots, i(n)$ such that $a \in A_{n+1}$.

B Auxiliary Functions for $Atleast(P, A)$

We present the auxiliary functions used by $Atleast(P, A)$ for cardinality, choice, and weight rules. We begin with the choice rule.

function $r.fire()$
  $r.literal := r.literal - 1$.

function $r.inactivate()$
  $r.inactive := r.inactive + 1$
  if $r.inactive = 1$ **then**
    for each atom a in $r.head$ **do**
      $a.headof := a.headof - 1$
      if $a.headof = 0$ **then**
        $negq.push(a)$
      else if $a.inA^+$ and $a.headof = 1$ **then**
        Let r' be the only active rule in $a.hlist$
        $r'.backchaintrue()$
      end if
    end for
  end if.

function $r.backchaintrue()$
  if $r.literal > 0$ **then**
    for every $a \in r.body^+$ **do**
      $posq.push(a)$
    end for
    for every $a \in r.body^-$ **do**
      $negq.push(a)$
    end for
  end if.

function $r.backchainfalse()$
  return.

A cardinality rule r

$$h \leftarrow k\,\{a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m\}$$

does not need an additional variable that keeps track of k if $r.literal$ is initialized to k and $r.inactive$ is initialized to $k - (n + m)$.

```
function r.fire()
  r.literal := r.literal − 1
  if r.literal = 0 then
    posq.push(r.head)
  else if r.head.inA⁻ then
    r.backchainfalse()
  end if.

function r.inactivate()
  r.inactive := r.inactive + 1
  if r.inactive = 1 then
    a := r.head
    a.headof := a.headof − 1
    if a.headof = 0 then
      negq.push(a)
    else if a.inA⁺ and a.headof = 1 then
      Let r' be the only active rule in a.hlist
      r'.backchaintrue()
    end if
  end if.

function r.backchaintrue()
  if r.literal > 0 and r.inactive = 0 then
    for every a ∈ r.body⁺ do
      posq.push(a)
    end for
    for every a ∈ r.body⁻ do
      negq.push(a)
    end for
  end if.

function r.backchainfalse()
  if r.literal = 1 and r.inactive ≤ 0 then
    for every a ∈ r.body⁺ do
      if a.inA⁺ = false then
        negq.push(a)
      end if
    end for
    for every a ∈ r.body⁻ do
      if a.inA⁻ = false then
        posq.push(a)
      end if
    end for
  end if.
```

A weight rule r

$$h \leftarrow \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, not\ b_1 = w_{b_1}, \ldots, not\ b_m = w_{b_m}\} \geq w$$

will need some auxiliary variables. If a is an atom in the body of r, then let $a.weight$ be the weight of the atom in r. For a partial model A, let $r.atleast = w$, let

$$r.max = \sum_{a \in r.body^+ - A^-} a.weight + \sum_{b \in r.body^- - A^+} b.weight$$

and let

$$r.min = \sum_{a \in r.body^+ \cap A^+} a.weight + \sum_{b \in r.body^- \cap A^-} b.weight.$$

function $r.fire()$
  Let a be the atom that is being given a truth value
  $r.min = r.min + a.weight$
  if $r.max \geq r.atleast$ and $r.min - a.weight < r.atleast$ **then**
    if $r.min \geq r.atleast$ **then**
      $posq.push(r.head)$
    else if $r.head.inA^- =$ true **then**
      $r.backchainfalse()$
    end if
  end if.

function $r.inactivate()$
  Let a be the atom that is being given a truth value
  $r.max = r.max - a.weight$
  if $r.max + a.weight \geq r.atleast$ and $r.min < r.atleast$ **then**
    if $r.\mathrm{max} < r.atleast$ **then**
      $a := r.head$
      $a.headof := a.headof - 1$
      if $a.headof = 0$ **then**
        $negq.push(a)$
      else if $a.inA^+$ and $a.headof = 1$ **then**
        Let r' be the only active rule in $a.hlist$
        $r'.backchaintrue()$
      end if
    else if $r.head.inA^+ =$ true and $r.head.headof = 1$ **then**
      $r.backchaintrue()$
    end if
  end if.

Consider the case of $backchaintrue()$ and $a \in r.body$. If

$$r.max - a.weight < r.atleast,$$

then a must be pushed onto the queue $posq$. On the other hand, if

$$r.max - a.weight \geq r.atleast,$$

then we can immediately skip all atoms whose weight is less than $a.weight$.

We take advantage of this observation in the following way. We introduce two new variables: $r.last^{+}$ and $r.last^{-}$, and we go through the body of r in decreasing weight order. When we do backward chaining in $r.backchaintrue()$ we start with the atom given by $r.last^{+}$ and check every atom until we arrive at an atom a for which $r.max - a.weight \geq r.atleast$. Then, we stop and update $r.last^{+}$ to a. When we backtrack we must restore $r.last^{+}$ to its previous value, and this can easily be done without needing any extra memory. If we remove the truth value of an atom a or if we remove a from a queue, and if $a.weight \geq r.last^{+}.weight$, then we set $r.last^{+} = a$.

function $r.backchaintrue()$
  if $r.min < r.atleast$ **then**
    for $a \in Atoms(r.body)$ in order, largest weight first, starting with $r.last^{+}$ **do**
      $r.last^{+} = a$
      if $a.inA^{+} =$ false and $a.inA^{-} =$ false **then**
        if $r.max - a.weight < r.atleast$ **then**
          if $a \in r.body^{+}$ **then**
            $posq.push(a)$
          else
            $negq.push(a)$
          end if
        else
          return
        end if
      end if
    end for
  end if.

function $r.backchainfalse()$
  if $r.max \geq r.atleast$ and $r.min < r.atleast$ **then**
    for $a \in Atoms(r.body)$ in order, largest weight first, starting with $r.last^{-}$ **do**
      $r.last^{-} = a$
      if $a.inA^{+} =$ false and $a.inA^{-} =$ false **then**
        if $r.min + a.weight \geq r.atleast$ **then**

```
            if a ∈ r.body⁺ then
                negq.push(a)
            else
                posq.push(a)
            end if
        else
            return
        end if
    end if
  end for
end if.
```

C Auxiliary Functions for $Atmost(P, A)$

We present the auxiliary functions used by $Atmost(P, A)$ for cardinality, choice, and weight rules. We begin with the choice rule.

```
function r.propagateFalse()
  r.upper := r.upper + 1
  if r.upper = 1 and r.inactive = 0 then
    for every a ∈ r.head do
      if a.source = 0 or a.source = r then
        a.source := 0
        queue.push(a)
      end if
    end for
  end if.
```

```
function r.propagateTrue()
  r.upper := r.upper − 1
  if r.upper = 0 and r.inactive = 0 then
    for every a ∈ r.head do
      if a.source = 0 then
        a.source := r
      end if
      queue.push(a)
    end for
  end if.
```

```
function r.isUpperActive()
  if r.upper = 0 and r.inactive = 0 then
    return true
  else
    return false
  end if.
```

For a cardinality rule r

$$h \leftarrow k\,\{a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m\}$$

we initialize $r.upper$ to $k - m$. We also need a counter $r.lower$ that holds the value $k - |r.body^- - A^+|$.

```
function r.propagateFalse()
  r.upper := r.upper + 1
  if r.lower ≥ 1 and r.inactive ≤ 0 and
    (r.head.source = 0 or r.head.source = r) then
    r.head.source := 0
    queue.push(r.head)
  end if.

function r.propagateTrue()
  r.upper := r.upper − 1
  if r.upper = 0 and r.inactive ≤ 0 then
    if r.head.source = 0 then
      r.head.source := r
    end if
    queue.push(r.head)
  end if.

function r.isUpperActive()
  if r.upper ≤ 0 and r.inactive ≤ 0 then
    return true
  else
    return false
  end if.
```

We can optimize the auxiliary functions a bit by changing $r.lower$ to keep track of the value

$$k - |r.body^{-} - A^{+}| - |B - A^{-}|,$$

where B is the set of atoms in $r.body^{+}$ that are not part of a positive loop that goes through r.

A weight rule r

$$h \leftarrow \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, not\ b_1 = w_{b_1}, \ldots, not\ b_m = w_{b_m}\} \geq w$$

also needs the two variables $r.upper$ and $r.lower$. The variable $r.upper$ is initialized to

$$\sum_{b \in r.body^{-}} b.weight$$

and $r.lower$ holds the value

$$\sum_{a \in B - A^{-}} a.weight + \sum_{b \in r.body^{-} - A^{+}} b.weight,$$

where B is defined as above.

function $r.propagateFalse()$
Let a be the atom that is being removed from the upper closure
$r.upper := r.upper - a.weight$
if $r.max \geq r.atleast$ and $r.upper + a.weight \geq r.atleast$ and $r.lower < r.atleast$ and $(r.head.source = 0$ or $r.head.source = r)$ **then**
$r.head.source := 0$
$queue.push(r.head)$
end if.

function $r.propagateTrue()$
Let a be the atom that is being added to the upper closure
$r.upper := r.upper + a.weight$
if $r.max \geq r.atleast$ and $r.upper - a.weight < r.atleast$ and $r.upper \geq r.atleast$ **then**
if $r.head.source = 0$ **then**
$r.head.source := r$
end if
$queue.push(r.head)$
end if.

function $r.isUpperActive()$
if $r.upper \geq r.atleast$ **then**
return true
else
return false
end if.

D Time-Line

A time-line for the releases of *smodels* that introduced new features.

Version	Date	Description
1.0	28.5.1995	First public release
1.1	7.3.1996	Fitting semantics
1.2	28.3.1996	Backward chaining if head is true, optional lookahead
1.3	27.9.1996	Backward chaining if head is false
1.6	30.6.1997	Lookahead is on by default, the heuristic is introduced
	31.8.1997	Work on smodels 2.0 has begun
1.8	4.9.1997	Strongly connected components optimization of the upper closure
pre-2.0-4	25.3.1998	Source pointer
2.0	22.10.1998	New rule types
2.6	27.1.1999	Reduction of the search space

References

[1] 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.

[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] R. Ben-Eliyahu and R. Dechter. Default reasoning using classical logic. *Artificial Intelligence*, 84:113–150, 1996.

[4] M. Berkelaar. Lp_solve 3.0. ftp://ftp.ics.ele.tue.nl/pub/lp_solve/, 1999. A mixed integer linear program solver.

[5] F. Buccafurri, N. Leone, and P. Rullo. Strong and weak constraints in disjunctive datalog. In *Logic Programming and Nonmonotonic Reasoning*, volume 1265 of *Lecture Notes in Artificial Intelligence*, pages 2–17, Dagstuhl, Germany, July 1997. Springer-Verlag. Proceedings of the 4th International Conference, LPNMR'97.

[6] M. Cadoli, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC: An executable specification language for solving all problems in NP. In *Practical Aspects of Declarative Languages*, volume 1551 of *Lecture Notes in Computer Science*, pages 16–30, San Antonio, Texas, USA, January 1999. Springer-Verlag. Proceedings of the First International Workshop, PADL'99.

[7] 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.

[8] P. Cholewiński, V.W. Marek, A. Mikitiuk, and M. Truszczyński. Computing with default logic. *Artificial Intelligence*, 112:105–146, 1999.

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

[10] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

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

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

[13] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in non-monotonic logic programs. In *Proceedings of the Fourth European Conference on Planning*, pages 169–181, Toulouse, France, September 1997. Springer-Verlag.

[14] 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.

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

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

[17] 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.

[18] W. Faber, N. Leone, and G. Pfeifer. Pushing goal derivation in DLP computations. In *Logic Programming and Nonmonotonic Reasoning*, volume 1730 of *Lecture Notes in Artificial Intelligence*, pages 177–191, El Paso, Texas, USA, December 1999. Springer-Verlag. Proceedings of the 5th International Conference, LPNMR'99.

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

[20] J.W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms.* PhD thesis, University of Pennsylvania, USA, 1995.

[21] 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.

[22] 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.

[23] K. Heljanko. Deadlock checking for complete finite prefixes using logic programs with stable model semantics (extended abstract). In *Proceedings of the 1998 Workshop on Concurrency, Specification & Programming*, pages 106–115. Humboldt-Universität, Berlin, September 1998. Informatik-Bericht Nr. 110.

[24] K. Heljanko. Minimizing finite complete prefixes. In *Proceedings of the 1999 Workshop on Concurrency, Specification & Programming*, pages 83–95, Warsaw, Poland, September 1999.

[25] K. Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe petri nets. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 240–254, Amsterdam, The Netherlands, March 1999. Springer-Verlag. Proceedings of the 5th International Conference, TACAS'99.

[26] D.E. Knuth. The Stanford GraphBase, 1993. Available at ftp://labrea.stanford.edu/.

[27] M. Krentel. The complexity of optimization problems. *Journal of Computer and System Sciences*, 36:490–509, 1988.

[28] N. Leone et al. Dlv, release 1999-11-24. http://www.dbai.tuwien.ac.at/proj/dlv/, 1999. A Disjunctive Datalog System.

[29] N. Leone, M. Romeo, P. Rullo, and D. Saccà. Effective implementation of negation in database logic query languages. In *LOGIDATA+: Deductive Databases with Complex Objects*, volume 701 of *Lecture Notes in Computer Science*, pages 159–175. Springer-Verlag, 1993.

[30] N. Leone, P. Rullo, and M. Romeo. Disjunctive stable models: Unfounded sets, fixpoint semantics and computation. *Information and Computation*, 135:69–112, 1997.

[31] C.M. Li and Anbulagan. Looka-ahead versus look-back for satisfiability problems. In *Principles and Practice of Constraint Programming – CP97*, volume 1330 of *Lecture Notes in Computer Science*, pages 341–355, Linz, Austria, October/November 1997. Springer-Verlag. Proceedings of the third International Conference.

[32] 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.

[33] X. Liu, C.R. Ramakrishnan, and S. Smolka. Fully local and efficient evaluation of alternating fixed points (extended abstract). In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 5–19, Lisbon, Portugal, March 1998. Springer-Verlag. Proceedings of the 4th International Conference, TACAS'98.

[34] V.W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K.R. Apt, W. Marek, M. Truszczyński, and D.S. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer Verlag, 1999. cs.LO/9809032.

[35] 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.

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

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

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

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

[40] I. Niemelä. Autoepistemic logic as a unified basis for nonmonotonic reasoning. Doctoral dissertation. Research report A24, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, August 1993.

[41] 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.

[42] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. In *Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, pages 72–79. Research Report A52, Helsinki University of Technology, May 1998.

[43] 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.

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

[45] 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.

[46] I. Niemelä and P. Simons. Smodels — an implementation of the stable model and well-founded semantics for normal logic programs. In

Logic Programming and Nonmonotonic Reasoning, volume 1265 of *Lecture Notes in Artificial Intelligence*, pages 420–429, Dagstuhl, Germany, July 1997. Springer-Verlag. Proceedings of the 4th International Conference, LPNMR'97.

[47] I. Niemelä, P. Simons, and T. Soininen. Stable model semantics for weight constraint rules. In *Logic Programming and Nonmonotonic Reasoning*, volume 1730 of *Lecture Notes in Artificial Intelligence*, pages 317–331, El Paso, Texas, USA, December 1999. Springer-Verlag. Proceedings of the 5th International Conference, LPNMR'99.

[48] P. Östergård, T. Baicheva, and E. Kolev. Optimal binary one-error-correcting codes of length 10 have 72 codewords. *IEEE Transactions on Information Theory*, 45(4):1229–1231, May 1999.

[49] C.H. Papadimitriou. *Computational Complexity.* Addison-Wesley Publishing Company, 1995.

[50] 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.

[51] T. Przymusinski. Stable semantics for disjunctive programs. *New Generation Computing*, 9(3/4):401–424, 1991.

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

[53] J. Rintanen. Lexicographic ordering as a basis of priorities in default reasoning. Technical Report A41, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, December 1996.

[54] C. Sakama and K. Inoue. An alternative approach to the semantics of disjunctive logic programs and deductive databases. *Journal of Automated Reasoning*, 13:145–172, 1994.

[55] C. Sakama and K. Inoue. Representing priorities in logic programs. In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 82–96, Bonn, Germany, September 1996.

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

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

[58] P. Simons. Towards constraint satisfaction through logic programs and the stable model semantics. Research Report A47, Helsinki University of Technology, August 1997.

[59] P. Simons. Extending the stable model semantics with more expressive rules. In *Logic Programming and Nonmonotonic Reasoning*, volume 1730 of *Lecture Notes in Artificial Intelligence*, pages 305–316, El Paso, Texas, USA, December 1999. Springer-Verlag. Proceedings of the 5th International Conference, LPNMR'99, cs.LO/9908004.

[60] P. Simons. Smodels 2.23. http://www.tcs.hut.fi/Software/smodels/, 1999. A system for computing the stable models of logic programs.

[61] T. Soininen, E. Gelle, and I. Niemelä. A fixpoint definition of dynamic constraint satisfaction. In *Principles and Practice of Constraint Programming – CP'99*, volume 1713 of *Lecture Notes in Computer Science*, pages 419–433, Alexandria, Virginia, USA, October 1999. Springer-Verlag. Proceedings of the 5th International Conference.

[62] T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Practical Aspects of Declarative Languages*, volume 1551 of *Lecture Notes in Computer Science*, pages 305–319, San Antonio, Texas, USA, January 1999. Springer-Verlag. Proceedings of the First International Workshop, PADL'99.

[63] 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.

[64] T. Syrjänen. Implementation of local grounding for logic programs with stable model semantics. Technical Report 18, Helsinki University of Technology, Finland, October 1998.

[65] 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.

[66] K. Varpaaniemi. Stable models for stubborn sets. In *Proceedings of the 1999 Workshop on Concurrency, Specification & Programming*, pages 263–274, Warsaw, Poland, September 1999.

[67] R. Zabih and D. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the Seventh National Conference on Artificial Intelligence, AAAI-88*, pages 155–160, St. Paul, Minnesota, August 1988. Morgan Kaufmann.

[68] H. Zhang. SATO: An efficient propositional prover. In *Automated Deduction – CADE-14*, volume 1249 of *Lecture Notes in Computer Science*, pages 272–275, Townsville, North Queensland, Australia, July 1997. Springer-Verlag. Proceedings of the 14th International Conference on Automated Deduction.

Index