Extending and Implementing the Stable Model Semantics*

Patrik Simons[1], Ilkka Niemelä[2], and Timo Soininen[3]

[1] Neotide Oy, Wolffintie 36, FIN-65200 Vaasa, Finland
Patrik.Simons@neotide.fi

[2] Helsinki University of Technology, Dept. of Computer Science and Eng.,
Lab. for Theoretical Computer Science, P.O.Box 5400, FIN-02015 HUT, Finland
Ilkka.Niemela@hut.fi

[3] Helsinki University of Technology, Dept. of Computer Science and Eng.,
Software Business and Engineering Institute, P.O.Box 9600, FIN-02015 HUT, Finland
Timo.Soininen@hut.fi

Abstract

A novel logic program like language, weight constraint rules, is developed for answer set programming purposes. It generalizes normal logic programs by allowing weight constraints in place of literals to represent, e.g., cardinality and resource constraints and by providing optimization capabilities. A declarative semantics is developed which extends the stable model semantics of normal programs. The computational complexity of the language is shown to be similar to that of normal programs under the stable model semantics. A simple embedding of general weight constraint rules to a small subclass of the language called basic constraint rules is devised. An implementation of the language, the SMODELS system, is developed based on this embedding. It uses a two level architecture consisting of a front-end and a kernel language implementation. The front-end allows restricted use of variables and functions and compiles general weight constraint rules to basic constraint rules. A major part of the work is the development of an efficient search procedure for computing stable models for this kernel language. The procedure is compared with and empirically tested against satisfiability checkers and an implementation of the stable model semantics. It offers a competitive implementation of the stable model semantics for normal programs and attractive performance for problems where the new types of rules provide a compact representation.

1 Introduction

Recently, logic programs with the stable model semantics have been put forward as a novel paradigm for applying declarative logic programming techniques [40, 44]. In this approach, called *answer set programming* (a term coined by Vladimir Lifschitz), a problem is solved by devising a logic program such that the stable models of the program provide the answers to the problem. Then an implementation of the stable model semantics can be used to compute the answers, i.e., the stable models of the program. Answer set programming

*This is an extended and revised version of two conference papers [46, 55]. The work was done while the first author was with the Lab. for Theoretical Computer Science at Helsinki University of Technology.

has been proposed as a methodology for solving, e.g., combinatorial, graph, and planning problems [5, 18, 36, 40, 44].

We illustrate answer set programming by showing how to solve the 3-coloring problem using normal logic programs. This is the problem of finding an assignment of one of three colors to each vertex of a graph such that vertices connected with an edge do not have the same color. Given a graph, we introduce three atoms $v(1), v(2), v(3)$ for each vertex v in the graph, intuitively meaning that the vertex v is colored with the color $1, 2$, or 3, respectively. Using these we build a program where for each vertex v we include the three rules on the left and for each edge (v, u) the three rules on the right

$$\begin{array}{ll} v(1) \leftarrow \mathit{not}\ v(2), \mathit{not}\ v(3) & \qquad \leftarrow v(1), u(1) \\ v(2) \leftarrow \mathit{not}\ v(1), \mathit{not}\ v(3) & \qquad \leftarrow v(2), u(2) \\ v(3) \leftarrow \mathit{not}\ v(1), \mathit{not}\ v(2) & \qquad \leftarrow v(3), u(3) \end{array} \tag{1}$$

Stable models are sets of ground atoms and, for instance, the first rule on the left says that if neither $v(2)$ nor $v(3)$ is in a stable model, then $v(1)$ is. Thus the rules on the left force each stable model to contain one of $v(1), v(2), v(3)$ for each vertex v in the graph. The first rule on the right excludes stable models containing both $v(1)$ and $u(1)$ and similarly for the other two rules. Hence, the rules on the right exclude any stable model having atoms $v(n)$ and $u(n)$ for some edge (v, u) in the graph where $n \in \{1, 2, 3\}$. This implies that a stable model of the program gives a legal coloring of the graph where a node v is colored with the color n iff $v(n)$ is included in the stable model.

This paper describes an implementation of the stable model semantics called the SMODELS system which has been developed since 1995. The system is available at `http://www.tcs.hut.fi/Software/smodels/`. SMODELS is directly usable for answer set programming. For instance, the 3-coloring problem discussed above can be solved using SMODELS by giving the logic program encoding of an instance of the 3-coloring problem as input and by letting SMODELS compute a stable model of the program which in turn gives directly a legal coloring of the graph. In fact, SMODELS has already been employed in a number of areas including planning [13, 44, 36], model checking [38], reachability analysis [26], product configuration [59, 62, 60], dynamic constraint satisfaction [58], logical cryptanalysis [1, 27], and network security [2].

Initially SMODELS supported only normal logic programs. When working towards applications we have observed that normal rules are inadequate for many interesting domains. They lack expressivity to represent, e.g., choices over subsets as well as cardinality and weight constraints in a compact way. As an example consider product configuration which can be roughly defined as the task of producing a correct specification, i.e., *configuration*, of a product individual. A correct configuration is a combination of components taken from a set of predefined component types while respecting a set of restrictions on combining the component types. The set of component types and restrictions are defined in a *configuration model* of the product. The following example demonstrates some typical forms of knowledge in a configuration model.

Example 1.1. A simplified configuration model of a PC could include the following. There is a set of different types of IDE hard disks, software packages, and other components that can be chosen to be parts of a PC. A PC must have from one to four IDE hard disks. In addition, the software packages use and hard disks produce disk space. Different types of hard disks provide and different software packages use varied amounts of disk space. The amount of disk space provided in a configuration must be larger than its use.

Normal logic program rules or even disjunctive logic programs do not provide a compact representation for such a configuration model. They lack constructs for representing choices with lower and upper cardinality bounds among a set of alternatives, such as for the IDE devices. In addition, resource constraints on a configuration exemplified by disk space cannot be captured conveniently.

In order to represent this type of knowledge we have devised a new rule language called *weight constraint rules*. It extends normal logic programs by allowing *weight constraints* in place of literals in a rule. A weight constraint can express a resource constraint as a linear inequality written as a set of literals with associated weights. A useful special case is a *cardinality constraint* that can be used to represent a choice from a set of literals with enforced cardinality limits. Furthermore, the language includes *optimize statements* for finding the (lexicographically) largest or smallest stable models with respect to a sequence of cost functions expressed as sets of literals with weights.

We generalize the stable model semantics of normal logic programs to weight constraint rules. The idea is to preserve the important properties of the semantics, in particular, the fact that stable models are grounded, i.e., atoms are not included in a model without grounds. However, we aim at a semantics with no substantial increase in computational complexity when compared to normal programs. In fact, we show that for variable free programs deciding whether a set of weight constraint rules (without optimization) has a stable model is NP-complete as is the case for normal programs. Despite similar computational complexity, weight constraint rules provide a generalization of normal programs with extra modeling power which seems to be challenging to capture using normal rules. Thus, weight constraint rules can provide more concise and maintainable programs.

We have identified an interesting simple subclass of ground (variable free) weight constraint rules called *basic constraint rules* which provides a sort of "normal form" of general rules. This means that ground weight constraint rules can be compiled to basic constraint rules in a modular and faithful fashion without a significant increase in the size of the program. This subclass plays an important role in our implementation of the weight constraint rules, the SMODELS system, which is based on a *two level architecture* consisting of a *front-end* and a *kernel language implementation*.

The front-end, the *lparse* program [63], compiles the weight constraint rules into the kernel language, basic constraint rules, so that the stable models correspond. To increase the usefulness of the weight constraint rules, we also allow variables and built-in functions in the front-end. For making the compilation to basic constraint rules effective we require that all rules are domain-restricted, a condition guaranteeing finite domains for all variables in the program.

A major part of the work has been the development of the kernel language implementation, the *smodels* program. It is an efficient search procedure for computing stable models of basic constraint rules. The kernel language has been developed as a compromise between two objectives. On one hand, enough expressivity should be provided in order to represent in a compact way, e.g., cardinality and resource constraints. On the other hand, the number of primitives should be kept small and the language simple for efficiency of the implementation. The ability to capture general weight constraints indicates adequate expressivity. The use of the kernel language seems to simplify the overall implementation and to keep the data structures and algorithms relatively uncomplicated. Moreover, the two-level architecture facilitates developing front-ends for other, possibly more application specific, languages besides weight constraint rules while taking advantage of the efficiency of *smodels*.

1.1 Contributions

This work has a number of key contributions. We have devised a new rule language and a corresponding extension of the stable model semantics. We have identified a "normal form", basic constraint rules, for the general language and analyzed the complexity of relevant computational problems. A major contribution is the *smodels* algorithm for computing the stable models for basic constraint rules. Here an important element is a novel heuristic which was not found by experimentation but derived using the principle that the search space should be minimized.

There are also novelties that concern the efficient implementation of the *smodels* algorithm. The first contribution is an implementation of lookahead that safely avoids testing every literal for failure. This implementation makes the use of lookahead feasible. The second contribution decreases the amount of work needed when pruning the search space. It consists of the use of source pointers and strongly connected components in a key part of the central pruning function, *expand*, which also provides an efficient implementation of the well-founded semantics [66, 56] for ground programs.

1.2 Related Work

The stable model semantics of normal logic programs is a form of nonmonotonic reasoning that is closely related to default logic of Reiter [50, 39], circumscription of McCarthy [41, 42, 35], and autoepistemic logic of Moore [43, 24]. While extended rules in this work are novel, there are some analogous constructions in the literature. Weight constraint rules can be seen as a generalization of disjunctive programs under the possible model semantics [52]. There is some similarity to the stable model semantics of disjunctive logic programs [49] but the semantics are fundamentally different (see Section 4). 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 [51, 5] and to order atoms [53].

If we look in a broader context, then finding a stable model is a combinatorial search problem. Other forms of combinatorial search problems include propositional satisfiability, constraint satisfaction, constraint logic programming and integer linear programming problems, and some other logic programming approaches such as NP-SPEC [6] and `dcs` [16]. The difference between these formalisms and the stable model semantics is that they do not include default negation. In addition, all but the last two are monotonic. In [25] several types of aggregates are integrated to Datalog in a framework based on stable models in order to express dynamic programming optimization problems using logic programs with choice constructs. However, only stratified negation is allowed.

Since the stable model semantics has become a standard method for supplying semantics to nonmonotonic logic programs, there is considerable interest in automating its computation. The earliest methods for finding stable models were based on the truth maintenance system of Doyle [15] and the assumption-based TMS of deKleer [11], see [19, 20, 48]. One of the first algorithms that took advantage of the well-founded semantics [66] was the one presented in [31] and generalized in [32]. Lately, more specialized algorithms have been developed [3, 61, 7, 12, 8, 21]. 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. Section 9 includes detailed comparisons of the more advanced previous algorithms and *smodels*.

There are a number of implementations of declarative semantics of logic programs already available. For example, the XSB system [67] is a WAM-based full logic programming system supporting the well-founded semantics. $\texttt{LDL}^{++}$ [69] is a deductive database system based on the stable model semantics supporting choice constructs and user defined aggregates but only for stratified programs. There are also systems supporting full stable model semantics. The DeReS system [65] was originally developed as a system for computing extensions of Reiter's default logic but now includes tools tuned for computing stable models of normal logic programs.

The dlv [30] and SMODELS systems have been developed specially for computing stable models of logic programs. They both use a two-level architecture where rules with variables are compiled to a set of ground rules preserving stable models and then an implementation for the ground rules is employed. The main difference is that dlv handles disjunctive logic programs. In dlv considerable emphasis has been placed on integrating deductive databases techniques to the grounding phase, e.g., for evaluating recursive stratified programs efficiently. Moreover, dlv supports weak constraints [5] and contains front-ends, e.g., for planning, abductive reasoning, and diagnosis. The SMODELS system does not handle disjunctive rules and has not been optimized for recursive database applications. However, it supports cardinality and weight constraints for rules with variables as well as constructs for solving optimization problems. In Section 9 the algorithms for the ground rules used in *smodels* and dlv are compared.

1.3 Outline of the Paper

The weight constraint rules and their stable model semantics are presented in Section 2. In Section 3 the complexity of the main decision and function problems related to weight constraint rules is analyzed. The relationship of weight constraint rules to other formalisms is discussed in Section 4, concentrating on other forms of logic programs and classical propositional logic. Section 5 introduces basic constraint rules and shows how to translate general ground weight constraint rules to this subclass. Section 6 gives an overview of how the SMODELS system implements general weight constraint rules including variables and, in particular, how rules with variables are mapped to basic constraint rules. In Section 7 a key element of the implementation, the *smodels* procedure, that computes stable models for basic constraint rules is developed. The implementation of *smodels* is described in greater detail in Section 8. In Section 9 we contrast *smodels* with the Davis-Putnam procedure for testing satisfiability of propositional formulae and with other advanced algorithms for computing stable models. In Section 10 we compare *smodels* with some propositional satisfiability checkers and the dlv system by running experiments on random satisfiability problems and on Hamiltonian cycle problems. Finally, conclusions and topics for further research are presented in Section 11.

2 Weight Constraint Rules

In this section we introduce weight constraint rules, a generalization of normal logic programs. The idea is to use weight constraints (instead of atoms) as basic building blocks for rules. First we treat ground rules. We present their syntax and semantics. Constructs for expressing optimality criteria, a natural extension of the weight constraints, are then incorporated into the language. In the last part of this section, we briefly present our approach to extending the weight constraint rules with variables and built-in functions.

2.1 Syntax of Weight Constraint Rules

In order to construct weight constraint rules we start from a set *Atoms* of ground atomic formulae, or atoms. However, atoms are not employed directly to build rules but they are used in weight constraints of which rules are constructed. A *weight constraint* is of the form

$$l \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \mathit{not}\ b_1 = w_{b_1}, \ldots, \mathit{not}\ b_m = w_{b_m}\} \leq u \tag{2}$$

where each a_i, b_j is an atom. We call a literal an atom b or a not-atom $\mathit{not}\ b$. Each literal in a constraint has an associated *weight*, e.g., in (2) w_{b_1} is the weight of the literal $\mathit{not}\ b_1$. The numbers l and u give the *lower* and *upper bounds* of the constraint, respectively. The weights and bounds can be real numbers, i.e., also negative weights are allowed.[1] We denote by $\mathrm{lit}(C)$ the set of literals in a weight constraint C. Intuitively, a weight constraint is satisfied by a set of atoms S if the sum of weights of those literals in $\{a_1, \ldots, a_n, \mathit{not}\ b_1, \ldots, \mathit{not}\ b_m\}$ that are satisfied by S is between the bounds l and u. Either of the bounds can also be omitted in which case the missing lower bound is taken to be $-\infty$ and the missing upper bound ∞.

A *weight constraint rule* is an expression of the form

$$C_0 \leftarrow C_1, \ldots, C_n \tag{3}$$

where each C_i is a weight constraint. A weight constraint rule program is a set of such rules.

We introduce useful short hands for special cases of weight constraints. We call a *cardinality constraint* an expression of the form

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

which is a short hand for the weight constraint with all weights equal to one:

$$l \leq \{a_1 = 1, \ldots, a_n = 1, \mathit{not}\ b_1 = 1, \ldots, \mathit{not}\ b_m = 1\} \leq u\ .$$

We denote by a literal l the simple weight constraint $1 \leq \{l = 1\}$. Hence, we can write rules such as

$$2 \leq \{a = 8, b = 2, \mathit{not}\ c = 4\} \leq 11 \leftarrow b, 2\ \{\mathit{not}\ d_1, \mathit{not}\ d_2, \mathit{not}\ d_3\} \tag{4}$$

where different kinds of short hands are used.

We let an *integrity constraint*

$$\leftarrow C_1, \ldots, C_n \tag{5}$$

be a short hand for a rule with an unsatisfiable head constraint C_0, say $1 \leq \{\}$.

Finally, we introduce some concepts that are used throughout the work. For an atom a, let $\mathit{not}\,(a) = \mathit{not}\ a$, and for a not-atom $\mathit{not}\ a$, let $\mathit{not}\,(\mathit{not}\ a) = a$. For a set of literals A, define $\mathit{not}\,(A) = \{\mathit{not}\,(a) \mid a \in A\}$, $A^+ = \{a \in \mathit{Atoms} \mid a \in A\}$, $A^- = \{a \in \mathit{Atoms} \mid \mathit{not}\ a \in A\}$, and $\mathit{Atoms}(A) = A^+ \cup A^-$. For a program P, define $\mathit{Atoms}(P) = \mathit{Atoms}(L)$, 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 $B \subseteq \mathit{Atoms}(A)$, and B is said to *agree* with A if $A^+ \subseteq B$ and $A^- \subseteq \mathit{Atoms} - B$.

[1]However, in the current implementation only integer weights are supported.

2.2 Stable Model Semantics

Models of weight constraint rules are sets of ground atoms similar to models of normal programs. A model S is said to satisfy an atom a if $a \in S$ and *not* a if $a \notin S$. Next we define when a model satisfies a weight constraint program.

Definition 2.1. A set of atoms S *satisfies* a weight constraint C of the form (2), denoted by $S \models C$, iff $l \leq \mathrm{w}(C, S) \leq u$, where

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

is the sum of the weights of the literals in C satisfied by S.

A rule $C_0 \leftarrow C_1, \ldots, C_n$ is satisfied by S ($S \models C_0 \leftarrow C_1, \ldots, C_n$) iff S satisfies C_0 whenever it satisfies each of $C_1, \ldots, C_n$. A program P is satisfied by S ($S \models P$) if each rule in P is satisfied by S.

Example 2.2. Consider the weight constraint C in the head of the rule (4) for which $\mathrm{w}(C, \{a, b, c\}) = 8 + 2 = 10$ and $\mathrm{w}(C, \{a\}) = 8 + 4 = 12$. So $\{a, b, c\} \models C$ but $\{a\} \not\models C$.

Now, the idea is to define a stable model of a weight constraint rule program as a set of atoms that satisfies the program and which is *grounded* (or *justified*) by it. By this we mean that every atom in a stable model is grounded by the rules in the program, i.e., has a non-circular justification. For example, consider a rule $2 \leq \{a = 2\} \leq 3 \leftarrow 1 \leq \{a = 1\} \leq 2$ which is satisfied by a model $\{a\}$. However, in this model the atom a has only a circular justification.

We formalize the justifiability property by suitably generalizing the concepts of a *reduct* for a normal program [24] and its *deductive closure*. For weight constraint rules, the reduct turns out to be a set of rules in a special form

$$h \leftarrow C_1, \ldots, C_n \tag{6}$$

where h is an atom and each constraint C_i contains only positive literals and has only a lower bound condition. We call such rules *Horn constraint rules*. The deductive closure $\mathrm{cl}(P)$ of a set of Horn constraint rules P can be defined as the unique smallest set of atoms closed under P in the same way as for definite rules. The uniqueness is implied by the fact that Horn constraint rules are monotonic, i.e., if the body of a rule is satisfied by a model S, then it is satisfied by any superset of S.

Example 2.3. Consider a set of Horn constraint rules P

$$\begin{aligned}
&a \leftarrow 1 \leq \{a = 1\} \\
&b \leftarrow 0 \leq \{b = 100\} \\
&c \leftarrow 6 \leq \{b = 5, d = 1\}, 2 \leq \{b = 2, a = 2\}
\end{aligned}$$

The deductive closure of P is $\{b\}$. If a rule

$$d \leftarrow 1 \leq \{a = 1, b = 1, c = 1\}$$

is added, then the closure is $\{b, d, c\}$.

We point out that negative weights and negative literals are closely related. They can replace each other and that one is inessential when the other is available. In order to keep the definition of stable models as simple as possible we define stable models only for rules with non-negative weights. A constraint with negative weights is treated as a short hand for an equivalent constraint with non-negative weights obtained as follows. Given a constraint C

$$l \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, not\ b_1 = w_{b_1}, \ldots, not\ b_m = w_{b_m}\} \leq u$$

with negative weights we can transform it using simple linear algebraic manipulation to an equivalent form C' with non-negative weights

$$\begin{aligned} l + \sum_{w_{a_i}<0} |w_{a_i}| + \sum_{w_{b_i}<0} |w_{b_i}| \leq \\ \{\underbrace{a_i = w_{a_i}}_{w_{a_i}\geq 0}, \ldots, \underbrace{b_j = |w_{b_j}|}_{w_{b_j}<0}, \ldots, \underbrace{not\ b_k = w_{b_k}}_{w_{b_k}\geq 0}, \ldots, \underbrace{not\ a_l = |w_{a_l}|}_{w_{a_l}<0}, \ldots\} \\ \leq u + \sum_{w_{a_i}<0} |w_{a_i}| + \sum_{w_{b_i}<0} |w_{b_i}| \end{aligned}$$

where each negative weight and the associated literal is complemented and the sum of absolute values of all negative weights is added to the bounds. The equivalence of C and C' can be seen as follows. In order to eliminate a negative weight w_{a_l}, we can transform C to an equivalent constraint by adding a pair

$$a_l = |w_{a_l}|, not\,(a_l) = |w_{a_l}|$$

to C and increasing its bounds by $|w_{a_l}|$. The constraint then contains

$$a_l = w_{a_l}, a_l = |w_{a_l}|, not\,(a_l) = |w_{a_l}|$$

which is equivalent to $not\,(a_l) = |w_{a_l}|$. Similarly, all negative weights w_{b_i} can be eliminated.

Example 2.4. Consider the rule

$$1 \leq \{a = 2\} \leq 2 \leftarrow -1 \leq \{a = -4, not\ b = -1\} \leq 0$$

where we can eliminate the negative weights in the body using the method above. The resulting rule is

$$1 \leq \{a = 2\} \leq 2 \leftarrow 4 \leq \{not\ a = 4, b = 1\} \leq 5\ .$$

We introduce the reduct for rules with non-negative weights in two steps. First we define the reduct of a weight constraint and then generalize this to rules.

Definition 2.5. The reduct C^S of a weight constraint C of the form (2) w.r.t. a set of atoms S is the constraint

$$l' \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}\} \tag{7}$$

with the lower bound

$$l' = l - \sum_{b_i \notin S} w_{b_i}\ .$$

Hence, in the reduct all negative literals and the upper bound are removed. In addition, the lower bound is decreased by the sum of the weights of the negative literals satisfied by S to account for the contribution of the negative literals towards satisfying the lower bound.

Example 2.6. For a constraint C

$$3 \leq \{a_1 = 1, not\ b_1 = 2, not\ b_2 = 3\} \leq 5$$

its reducts w.r.t. $\{b_1\}$ and $\{b_2\}$ are as follows

$$C^{\{b_1\}}: \ 0 \leq \{a_1 = 1\} \qquad C^{\{b_2\}}: \ 1 \leq \{a_1 = 1\}$$

The reduct P^S for a program P w.r.t. a set of atoms S is a set of Horn constraint rules which contains a rule r' with an atom p as the head if $p \in S$ and there is a rule $r \in P$ such that p appears in the head of r and the upper bounds of the constraints in the body of r are satisfied by S. The body of r' is obtained by taking the reduct of the constraints in the body of r. Formally the reduct is defined as follows.

Definition 2.7. Let P be a weight constraint rule program and S a set of atoms. The reduct P^S of P w.r.t. S is defined by

$$\begin{aligned} P^S \ = \ \{p \leftarrow C_1^S, \ldots, C_n^S \mid\ & C_0 \leftarrow C_1, \ldots, C_n \in P,\ p \in \mathrm{lit}(C_0) \cap S \\ & \text{and } \mathrm{w}(C_i, S) \leq u \text{ for all } C_i \text{ of the form (2)} \\ & \text{where } i = 1, \ldots, n\} \end{aligned}$$

Now a stable model of a program P is defined as an atom set S that *satisfies* all rules of P and that is the *deductive closure* of the *reduct* of P w.r.t. S.

Definition 2.8. A set of atoms S is a *stable model* of a program P with non-negative weights iff the following two conditions hold:
(i) $S \models P$,
(ii) $S = \mathrm{cl}(P^S)$.

Example 2.9. Consider the program P

$$2 \leq \{b = 2, c = 3\} \leq 4 \leftarrow 2 \leq \{not\ a = 2, b = 4\} \leq 5$$

The groundedness requirement (ii) guarantees that a stable model of a weight program is a subset of the atoms appearing in the heads of the rules in the program, i.e., a subset of $\{b, c\}$ in this case. The empty set is not a stable model as $\emptyset \not\models P$. The set $\{b\}$ satisfies P but it is not a stable model of P: b is not grounded because the reduct $P^{\{b\}}$ is empty, as the upper bound in the body of the rule is exceeded. However, $S = \{c\}$ is a stable model as $S \models P$ and $\mathrm{cl}(P^S) = \{c\}$ where $P^S = \{c \leftarrow 0 \leq \{b = 4\}\}$. In fact, $\{c\}$ is the only stable model of P.

2.3 Optimize Statements

In many applications optimization capabilities are needed: each feasible solution can be assigned a cost and we are looking for an optimal solution, i.e., a feasible solution with minimum or maximum cost. In order to support such applications we extend the weight constraint rule language by introducing two optimize statements, the minimize statement and its dual the maximize statement, in which the cost functions for stable models are given as linear sums of weights of literals. A minimize statement M of the form

$$minimize \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, not\ b_1 = w_{b_1}, \ldots, not\ b_m = w_{b_m}\} \tag{8}$$

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

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

There can be several minimize statements which order the stable models lexicographically according to the weights of the statements with the first statement being the most significant. More precisely, let $M_1, \ldots, M_l$ be the sequence of statements of the form (8) in a program P. The ordering $\leq_P$ for stable models is obtained by defining that $S \leq_P S'$ holds iff $\mathrm{w}(M_i, S) = \mathrm{w}(M_i, S')$ for all $i = 1, \ldots, l$ or there is some $j \leq l$ such that $\mathrm{w}(M_j, S) < \mathrm{w}(M_j, S')$ and $\mathrm{w}(M_i, S) = \mathrm{w}(M_i, S')$ for all $i = 1, \ldots, j-1$. We call a stable model S of a program P *optimal* if for any other stable model S' of P, $S \leq_P S'$.

Example 2.10. The optimal stable model of the program below is $\{b\}$.

$$a \leftarrow not\ b$$
$$b \leftarrow not\ a$$
$$minimize \{a = 1\}$$
$$minimize \{b = 1\}$$

The maximize statement

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

is just another way to write $minimize \{a = -w_a, not\ b = -w_b\}$ or equivalently $minimize \{a = k - w_a, not\ b = k - w_b\}$ where $k \geq w_a + w_b$ if one wants to avoid negative weights. Notice that negative weights in minimize statements can be eliminated using the technique presented in Section 2.2.

Finally, we point out that multiple minimize statements with non-negative weights can be written as one by suitably scaling the weights. The idea is that for a sequence of minimize statements $M_1, \ldots, M_l$ one minimize statement

$$c_1 \times M_1 \cup \cdots \cup c_l \times M_l$$

is formed where the weights of the first (most significant) statement are multiplied by c_1, those of the second by c_2 and so on. In order to preserve the lexicographic ordering it is enough to set $c_l = 1$ and require for $i = 1, \ldots, l-1$,

$$c_{l-i} > \sum_{j=l-(i-1)}^{l} c_j \mathrm{w}(M_j) \tag{9}$$

where

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

is the sum of the weights in a minimize statement M of the form (8).

Example 2.11. Given a sequence of minimize statements M_1, M_2

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

it is enough to set $c_2 = 1$ and $c_1 = 2c_2\mathrm{w}(M_2) = 2\sum_{i=1}^{m} w_{b_i}$ and write

$$\mathit{minimize}\ \{a_1 = c_1 w_{a_1}, \ldots, a_n = c_1 w_{a_n}, b_1 = c_2 w_{b_1}, \ldots, b_m = c_2 w_{b_m}\}\ .$$

2.4 Weight Constraint Rules with Variables and Functions

Until now we have considered only ground weight constraint rules, i.e., rules where all literals are variable free. In many applications it would be impractical and error-prone to manually write the ground rules that capture the solutions. Consider, e.g., rules (1) for the 3-coloring problem where there are three rules for each vertex and three rules for each edge of the graph. Usually it is possible to produce such a set of ground rules automatically but this requires programming and makes maintaining and revising programs harder.

Hence, it is useful to allow weight constraint rules with variables. The stable model semantics can be extended to handle such a program by treating a rule with variables as a short hand for all its ground instantiations with respect to the Herbrand universe of the program. However, even normal logic programs are highly undecidable if function symbols are allowed. Here we present a practical approach to extending weight constraint rules with variables and function symbols in a limited way. It guarantees decidability and seems to be adequate for many applications. This is achieved by allowing only *domain-restricted rules*, which we introduce next. In addition, we discuss two useful extensions for rules with variables, conditional literals and built-in functions.

The basic idea behind domain-restrictedness is as follows. A domain-restricted program P can be thought of as being divided into two parts: P_{Do} that defines *domain predicates* and P_{Ot} that contains all other rules. The form of the domain rules P_{Do} is restricted in such a way that they have a unique finite stable model D which should be relatively efficiently computable. All the other rules in P_{Ot} are then domain-restricted in the sense that every variable in a rule must appear in a domain predicate which occurs positively in the body of the rule. Then P has the same stable models as a subset P_D of the ground instantiation of P where for each ground rule the instances of the domain predicate are satisfied by D. Notice that now it is also possible to use function symbols without losing decidability because the program P_D remains finite.

The idea is to employ a subclass of normal logic programs as rules for defining domain predicates. There are several natural candidates for subclasses of rules having a unique stable model, e.g., non-recursive rules or stratified rules could be used. An interesting class of domain rules is obtained by considering stratified rules which are domain-restricted in such a way that also function symbols can be used without losing decidability [64]. Here the restriction is that all variables that occur in a rule must also occur in a positive body literal that belongs to a strictly lower stratum than the head in a stratification of the domain rules. This is the class of domain rules supported by the current SMODELS system, further details can be found in [64].

Example 2.12. We illustrate recursive domain rules with function symbols using the following example. Consider a relation r given as a set R of ground facts of the form $r(x, y) \leftarrow$ and a 3-coloring problem for a graph for which the edge relation is the transitive closure of r with each vertex v renamed as a term $t(v)$ using a function symbol t. Hence, an edge $(t(x), t(y))$ is in the graph iff (x, y) is in the transitive closure of r. The problem can be formalized as a domain-restricted program P which includes the facts R and the rules

$$d(Y) \leftarrow r(X,Y) \tag{10}$$
$$tc_r(X,Y) \leftarrow r(X,Y) \tag{11}$$
$$tc_r(X,Y) \leftarrow r(X,Z), tc_r(Z,Y), d(Y) \tag{12}$$
$$edge(t(X), t(Y)) \leftarrow tc_r(X,Y) \tag{13}$$
$$vertex(X) \leftarrow edge(X,Y);\;\; vertex(Y) \leftarrow edge(X,Y) \tag{14}$$
$$col(c_1) \leftarrow;\;\; col(c_2) \leftarrow;\;\; col(c_3) \leftarrow \tag{15}$$
$$1\ \{color(X,c_1), color(X,c_2), color(X,c_3)\}\ 1 \leftarrow vertex(X) \tag{16}$$
$$\leftarrow edge(X,Y), color(X,C), color(Y,C), col(C) \tag{17}$$

where a term beginning with an upper-case letter, like X, Y, is a variable. For P, P_{Do} includes the facts R and the rules (10–15). The domain predicates are $r, d, tc_r, col, edge$, and $vertex$. Notice that to guarantee the domain-restrictedness of the recursive rule (12) the domain predicate $d(Y)$ is needed. Similarly, the rule (17) would not be domain-restricted if the last body literal $col(C)$ were removed because then the variable C would only appear in predicate *color* which is not a domain predicate.

We now turn to two useful extensions. In order to be able to compactly write down sets of literals for weight constraints, we introduce the notion of a *conditional literal* of the form $l : d$ where l is a literal and the conditional part d is a domain predicate. Such a conditional literal corresponds to the sequence of all the instances l' of the literal l obtained by making a substitution to $l : d$ such that for the resulting $l' : d'$, d' is in the unique stable model of P_{Do}. For example, the rule (16) could be expressed using a conditional literal as

$$1\ \{color(X,C) : col(C)\}\ 1 \leftarrow vertex(X)\ .$$

Finally, we note that it is straightforward to extend domain-restricted rules with built-in functions. Domain-restrictedness guarantees that the domain over which each variable in a built-in function ranges is restricted by domain predicates. Thus, floundering problems are avoided. The set of built-in functions supported by the current SMODELS system includes functions for integer arithmetic as well as mechanisms for providing user-defined built-in functions. This allows rules such as

$$area(C, W \times L) \leftarrow width(C,W), length(C,L)$$
$$0 \leq \{component(C) : area(C,A) = A\} \leq 90 \leftarrow$$

where *area* is a domain predicate defined using domain predicates *width* and *length* (giving the width and length of a component) and a built-in function '$\times$' for integer multiplication. The second rule specifies a subset of the components with the sum of areas at most 90. Note that weights of literals can be expressed using domain predicates, such as $area(C, A)$ above.

3 Complexity

We study the complexity of two computational problems related to ground weight constraint rules: the *decision problem* of determining whether a program has a stable model and the *function problem* of computing a stable model for a program. First we consider the case where no optimize statements are allowed and then extend the analysis to include optimization. Complexity analysis of weight constraint rules with variables is beyond the scope of this paper; however, we point to [64] for further information. For the complexity analysis we assume that all weights and bounds are integers. For the definitions of the complexity classes used in the analysis, see e.g. [47].

3.1 Rules Without Optimize Statements

First we establish that checking a stable model can be done in polynomial time for a ground program without optimize statements.

Proposition 3.1. *Let P be a finite set of ground weight constraint rules without optimize statements and S a set of ground atoms. Then it can be decided in polynomial time whether S is a stable model of P.*

Proof. The set S is a stable model of P iff (i) $S \models P$ and (ii) $S = \mathrm{cl}(P^S)$. Clearly, (i) can be checked and the reduct P^S can be constructed in polynomial time given P and S. Hence, it remains to show how to compute closure $\mathrm{cl}(P^S)$ in polynomial time. This is achieved by an algorithm where a set S' (for the closure) is initialized to the empty set and then the statement

$$\textbf{if} \text{ for some rule } h \leftarrow C_1, \ldots, C_n \in P^S,\ S' \models C_1, \ldots, C_n,\ \textbf{then}\ \ S' := S' \cup \{h\}$$

is repeated until there is no change is S'. As the rules in P^S are monotonic, the set S can only increase until no new rule has its body satisfied. Hence, the statement can be executed at most r times where r is the number of rules in P^S and each execution can be accomplished in polynomial time. □

Without optimize statements deciding the existence of a stable model turns out to be NP-complete. Inclusion in NP is implied by Proposition 3.1. As propositional satisfiability is NP-complete, for NP-hardness it is sufficient to show that propositional models can be captured using stable models through a polynomial time embedding. For the embedding we use a small subset of weight constraint rules including only integrity constraints with literals and simple choice rules with empty bodies.

Consider propositional models of a set of clauses Σ. They can be captured as stable models of a program P_Σ which contains for each clause $a_1 \vee \cdots \vee a_n \vee \neg b_1 \vee \cdots \vee \neg b_m \in \Sigma$, the rule

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

and for each atom a that appears in the clause, the simple choice rule $\{a\} \leftarrow$. Now a stable model of P_Σ is a propositional model of Σ (the atoms assigned true) and vice versa. We point out that the embedding can be extended to a general propositional formula ϕ, i.e., the models of ϕ can be captured as stable models of a set P_ϕ of simple choice rules and normal rules [45].

Proposition 3.1 and the embedding above establish the following.

Theorem 3.2. *Deciding whether a set of ground weight constraint rules without optimize statements has a stable model is NP-complete.*

Moreover, the function problem of computing a stable model is FNP-complete.

Theorem 3.3. *Computing a stable model for a set of ground weight constraint rules without optimize statements is FNP-complete.*

Proof. Proposition 3.1 implies that computing a stable model is in FNP. For establishing FNP-hardness we consider the FNP-complete problem FSat [47] of computing a satisfying truth assignment for a propositional formula ϕ. This can be reduced to the problem of computing a stable model of the program P_ϕ capturing the models of ϕ. □

Notice that propositional logic can be embedded into normal logic program rules if new atoms can be introduced. This is done by replacing in the translations above each rule $\{a\} \leftarrow$ with the pair of rules $a \leftarrow not\ \bar{a}$ and $\bar{a} \leftarrow not\ a$ where $\bar{a}$ is a new atom. Hence, it can be shown that computing a stable model for a normal logic program is FNP-complete, too.

3.2 Optimize Statements

When the program contains optimize statements, we are looking for *optimal* stable models, i.e., stable models S of P such that for any other stable model S' of P, $S \leq_P S'$ where $\leq_P$ is the (lexicographic) order of stable models according to the optimize statements in P. Hence, such an optimal stable model exists iff there is a stable model of the program obtained by removing all optimize statements. If we are looking for an optimal stable model containing a given atom, a Δ_2^p-hard problem is obtained even for a very simple form of rules.

Proposition 3.4. *Deciding whether a set of weight constraint rules has an optimal model containing a given atom is Δ_2^p-hard.*

Proof. Deciding the truth value of the least significant atom p of the lexicographically largest satisfying truth assignment of a propositional formula ϕ is Δ_2^p-complete [29]. This problem can be reduced to a problem of deciding whether a program has an optimal model containing p for a program P which consists of a set of simple choice rules and normal rules P_ϕ (capturing the models of ϕ as its stable models) extended with the maximize statement

$$maximize\ \{a = 1\}$$

for each atom a in the lexicographic order. □

In fact, the problem is Δ_2^p-complete.

Theorem 3.5. *Deciding if a set of ground weight constraint rules with optimize statements has an optimal model containing a given atom is Δ_2^p-complete.*

Proof. Given Proposition 3.4 it remains to be shown that the problem is in Δ_2^p for general weight constraint rules, i.e., that deciding whether for a set of general rules there is an optimal stable model containing a given atom p can be done in polynomial time using a polynomial number of NP oracle calls. We establish this by employing an oracle which given a program P (without optimize statements), a minimize statement M, a set of literals A, and an integer k decides whether there exists a stable model S of P that agrees with A such that $\mathrm{w}(M, S) \leq k$. By Proposition 3.1 this is an NP oracle.

Assume without loss of generality that all weights are positive and only minimize statements are used. We first rewrite the program with optimize statements to a program P without optimize statements and one minimize statement M as described in Section 2.3. Then we use Algorithm 1 which given P and M determines with a binary search the weight of the optimal models using a logarithmic number of oracle calls w.r.t. $\mathrm{w}(M)$. Notice that the number of oracle calls can be kept polynomial w.r.t. the original program containing a sequence of minimize statements $M_1, \ldots, M_k$. This is because when rewriting $M_1, \ldots, M_k$ into M, it is enough to multiply each M_i with c_i where $c_k = 1$ and for $i = 1, \ldots, k-1$

$$c_{k-i} = 2\mathrm{w}(M_k) \cdots 2\mathrm{w}(M_{k-(i-1)})$$

in order to satisfy the scaling condition (9). Then it can be shown that $\mathrm{w}(M) = \sum_{i=1}^{k} c_i \mathrm{w}(M_i) \leq k2^k \mathrm{w}(M_1) \cdots \mathrm{w}(M_k)$.

function *findOptimal*(P, M)
  if there is no stable model of P **then**
    return false
  end if
  $u := \mathrm{w}(M)$
  $l := 0$
  while $l < u$ **do**
    if there exists a stable model S of P such that $\mathrm{w}(M, S) \leq \lfloor (u+l)/2 \rfloor$ **then**
      $u := \lfloor (u+l)/2 \rfloor$
    else
      $l := \lfloor (u+l)/2 \rfloor + 1$
    end if
  end while
  return l.

Algorithm 1: A procedure for determining the weight of the optimal models.

Now the existence of an optimal model containing p can be decided with one additional oracle call testing whether there is a stable model S of P that agrees with $\{p\}$ such that $\mathrm{w}(M, S) \leq l$. □

We finish the section by showing that the function problem of computing an optimal stable model is $\mathrm{FP}^{\mathrm{NP}}$-complete.

Theorem 3.6. *Let P be a set of ground weight rules with optimize statements. Then computing an optimal stable model of P is $\mathrm{FP}^{\mathrm{NP}}$-complete.*

Proof. We begin by showing that the problem is in $\mathrm{FP}^{\mathrm{NP}}$. We use the approach of the previous proof with the same oracle. We rewrite the program such that it contains only one minimize statement and then use Algorithm 1 to find the weight l of the optimal models. Then an optimal stable model can be constructed using Algorithm 2 with a linear number of additional oracle calls.

We prove that the problem is $\mathrm{FP}^{\mathrm{NP}}$-hard by showing that the problem MAX-WEIGHT SAT can be reduced to it. If we are given a set of clauses, each with an integer weight, then MAX-WEIGHT SAT is the problem of finding a truth assignment that satisfies a subset of the clauses with the greatest total weight. The problem MAX-WEIGHT SAT is $\mathrm{FP}^{\mathrm{NP}}$-complete [47].

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

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

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

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

A stable model of this program with maximal weight provides a truth assignment of greatest total weight. Note that we again use a very restricted form of rules only. □

function *optimalModel*(P, M, l)
  $A = \emptyset$
  for each atom a in $\mathit{Atoms}(P)$ **do**
    if there exists a stable model S of P that agrees with $A \cup \{a\}$ such that $\mathrm{w}(M, S) \leq l$
    then
      $A := A \cup \{a\}$
    else
      $A := A \cup \{\mathit{not}\ a\}$
    end if
  end for
  return A^+.

Algorithm 2: A procedure for constructing a stable model S of P such that $\mathrm{w}(M, S) \leq l$ provided that such a model exists

4 Relationships to Other Formalisms

In this section we discuss the relationship of weight constraint rules and other formalisms. As our approach uses ideas from the stable model semantics of logic programs, we focus on the relationship between logic programs and weight constraint rules. In the end of the section we also discuss connections to propositional logic.

4.1 Normal Logic Programs and Weight Constraint Rules

First we show that our definition of stable models is a generalization of the original definition for normal programs by Gelfond and Lifschitz [24]. They define a set S of atoms to be a stable model of a set P_N of normal rules

$$h \leftarrow a_1, \ldots, a_n, \mathit{not}\ b_1, \ldots, \mathit{not}\ b_m \tag{18}$$

if $S = \mathrm{cl}(P_N^S)$, i.e., S is the closure of the reduct

$$P_N^S = \{h \leftarrow a_1, \ldots, a_n \mid \begin{array}{l} h \leftarrow a_1, \ldots, a_n, \mathit{not}\ b_1, \ldots, \mathit{not}\ b_m \in P_N, \\ \{b_1, \ldots, b_m\} \cap S = \emptyset\} \ . \end{array}$$

According to our short hand conventions a normal rule can be seen as a weight constraint rule where each literal p corresponds to a simple weight constraint $1 \leq \{p = 1\}$ (written as a cardinality constraint $1\ \{p\}$ below). First, we note that (18) is satisfied by a model S (in the sense of normal programs) iff it is satisfied by S (seen as a weight constraint rule). Second, we compare the original reduct P_N^S of a normal program P_N to the reduct P_W^S of the corresponding weight constraint program P_W. Note that P_W^S consists of rules

$$h \leftarrow 1\ \{a_1\}\ , \ldots, 1\ \{a_n\}\ , l_1\ \{\}\ , \ldots, l_m\ \{\} \tag{19}$$

where $h \in S$ and $l_i = 0$ if $b_i \notin S$ and $l_i = 1$ if $b_i \in S$. Hence, rules for which $b_i \in S$ ($l_i = 1$) for some i cannot contribute to the closure of the reduct and, the only essential difference between P_W^S and P_N^S is that the latter can contain also rules where $h \notin S$. Thus, it is straightforward to establish

$$\mathrm{cl}(\{h \leftarrow a_1, \ldots, a_n \in P_N^S \mid h \in S\}) = \mathrm{cl}(P_W^S)\ . \tag{20}$$

Now we are ready to show that our definition of stable models coincides with the original definition.

Proposition 4.1. *Given a normal program P_N, a set of atoms S is a stable model of the corresponding weight constraint rule program P_W iff $S = \mathrm{cl}(P_N^S)$.*

Proof. ($\Rightarrow$) Let S be a stable model of P_W. Then $S \models P_W$ and, hence, $S \models P_N$. As $\mathrm{cl}(P_N^S)$ is the least model of P_N, $\mathrm{cl}(P_N^S) \subseteq S$. Thus, $\mathrm{cl}(P_N^S) = \mathrm{cl}(\{h \leftarrow a_1, \ldots, a_n \in P_N^S \mid h \in S\})$ which implies $S = \mathrm{cl}(P_W^S) = \mathrm{cl}(P_N^S)$ by (20).

($\Leftarrow$) Let $S = \mathrm{cl}(P_N^S)$ hold. Then $S \models P_N$ and, consequently, $S \models P_W$. Similarly, $\mathrm{cl}(P_N^S) = \mathrm{cl}(\{h \leftarrow a_1, \ldots, a_n \in P_N^S \mid h \in S\})$ which again implies that S is a stable model of P_W by (20). $\square$

As the FNP-completeness results for weight constraint rules and normal logic programs in Section 3 indicate, there is a polynomial time translation from a set P of weight constraint rules to a normal program P' such that given a stable model of P', a stable model of P can be computed in polynomial time. However, it seems to be non-trivial to find a concise mapping from weight constraint rules to normal programs which is *modular*, i.e., in which rules are translated independently of each other. In particular, translating general weight constraints seems challenging but for cardinality constraints and choice constructs modular translations are easier to develop. We illustrate possible translations and related difficulties by considering two simple types of rules.

A naive normal logic program representation of a cardinality constraint rule

$$h \leftarrow k\,\{a_1, \ldots, a_n\} \tag{21}$$

consists of the rules

$$\{h \leftarrow a_{i_1}, \ldots, a_{i_{k+1}} \mid 1 \leq i_1 < \cdots < i_{k+1} \leq n\}$$

but the number of such rules is $\binom{n}{k+1}$. There is a $\mathcal{O}(n^2)$ solution, or more precisely, a solution that needs $\mathcal{O}(nk)$ rules using the following idea. Let the atom $l(a_i, j)$ represent the fact that at least j of the atoms in $\{a_i, \ldots, a_n\}$, i.e. of the atoms that have an equal or higher index than i, are in a particular stable model. Then, the cardinality rule (21) can be handled by a rule $h \leftarrow l(a_1, k)$. The definition of $l(a_i, j)$ is given by the rules

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

Both of the translations are modular and seem feasible for small values of k. The problem is that if the set $\{a_1, \ldots, a_n\}$ is big, then even for the quadratic translation the resulting set of rules is rather large, requiring $\mathcal{O}(nk)$ new atoms to be introduced. Moreover, the size of the translation grows towards $\mathcal{O}(n^2)$ with the *value* of k.

As another example we consider simple choice rules $\{h\} \leftarrow$ employed in Section 3.1 to capture propositional models. As indicated there, we can translate such a choice construct to normal rules by introducing new atoms. It turns out that new atoms are necessary because such simple choice rules allow stable models which are not subset minimal, whereas for normal logic programs the stable models are subset minimal. It is therefore not possible to translate a program containing simple choice rules into a normal program without introducing new atoms.

Proposition 4.2. *Let Π be the class of weight constraint rule programs and Π' the class of normal programs. Then, there is no mapping from Π to Π' that preserves stable models and does not introduce new atoms.*

To summarize, the complexity results imply that there is a polynomial time mapping from weight constraint rules to normal programs such that their stable models correspond. However, mapping general weight constraints to normal rules seems challenging and it is an open question to devise a compact and modular translation.

4.2 Disjunctive Logic Programs

Extending normal programs by allowing disjunctions in the heads of the rules does not seem to make it easier to capture general weight constraints. The main semantics of such disjunctive programs are built on the paradigm of minimal models [18]. Hence, even simple choice rules cannot be translated to disjunctive rules without introducing new atoms. However, answer sets of general extended disjunctive programs, introduced in [37] as a subclass of the logic of minimal belief and negation as failure, allow non-minimal models [53]. In this class of programs negative literals can be used in the heads of the rules as in weight constraint rules and choice rules can be encoded without new atoms. Negative head literals are essential here because answer sets of extended programs follow the minimal model paradigm for programs without such negative head literals.

On one hand, it is thus open how to embed general weight constraint rules into disjunctive programs using a concise modular mapping. On the other hand, it is unlikely that disjunctive programs can be embedded in polynomial time into weight constraint rules. This is because for minimal model based semantics of disjunctive programs the relevant decision problems are complete for the second level of the polynomial hierarchy [17] but for weight constraint rules they are NP-complete. Hence, a polynomial time mapping from disjunctive programs to weight constraint rules preserving stable models cannot exist unless the polynomial time hierarchy collapses.

An exception of the minimal model paradigm for disjunctive programs is the possible model semantics [52] where non-minimal models are allowed. Under this semantics, disjunctive programs can be embedded into weight constraint rules in a straightforward manner and special cases of weight constraints such as the simple choice rules can be captured using disjunctive rules [59, 56].

4.3 Propositional Logic

The FNP-completeness result (Theorem 3.3) in Section 3 indicates that there are polynomial time embeddings between propositional logic and weight constraint rules. However, there appears to be fundamental differences in expressivity, i.e., in the ability of one formalism to capture the other. Propositional logic can be embedded into weight constraint rules in a straightforward way as shown in Section 3. Notice that the mapping is computationally easy and modular, e.g., each clause can be translated independently of other clauses.

However, propositional logic does not seem to be able to capture weight constraint rules through a concise, modular translation. In particular, we note that the default negation '*not*' used in the rules cannot be embedded modularly into propositional logic. Consider a mapping T from logic programs to sets of propositional formulae. 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 4.3 (Niemelä [44]). *There is no modular mapping from the class of normal logic programs to propositional formulae.*

Although there is no modular translation of normal logic programs into propositional logic, there are more complex ones [4]. However, these translations are rather complicated and hide much of the structure of the original problem, which makes them cumbersome to use when modeling. Capturing general weight constraints rules seems to be even more non-trivial.

5 Basic Constraint Rules

In this section we introduce an interesting subclass of ground weight constraint rules which we call *basic constraint rules*. The idea is that they can serve as a "normal form" for general weight constraint rules, i.e, we show that any weight constraint rule can be captured by a set of basic rules. Basic constraint rules play a major role in implementing general weight constraint rules. The idea is to develop an efficient implementation for this *kernel language* and handle more general rules by translating them to basic constraint rules.

Basic constraint rules include two types of rules:

- A *weight rule* is of the form

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

  where all weights w, w_{a_l}, w_{b_k} are non-negative and it is the short hand for

$$\begin{aligned} &1 \leq \{h = 1\} \leftarrow \\ &\qquad w \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, not\ b_1 = w_{b_1}, \ldots, not\ b_m = w_{b_m}\}\,. \end{aligned}$$

- A *choice rule* is of the form

$$\{h_1, \ldots, h_k\} \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m \tag{22}$$

  and it is the short hand for

$$\begin{aligned} &0 \leq \{h_1 = 1, \ldots, h_k = 1\} \leftarrow \\ &\qquad n + m \leq \{a_1 = 1, \ldots, a_n = 1, not\ b_1 = 1, \ldots, not\ b_m = 1\}\,. \end{aligned}$$

A weight rule captures the lower bound condition for a single weight constraint. A choice rule encodes a conditional choice stating that if the body of the rule is satisfied, then any subset (including the empty one) can be selected from the set in the head but if the body is not satisfied, no subset can be chosen.

We introduce the following short hand notations for frequently used special forms of basic constraint rules.

- A *cardinality rule* $h \leftarrow k\,\{a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m\}$ corresponds to
$$h \leftarrow \{a_1 = 1, \ldots, a_n = 1, not\ b_1 = 1, \ldots, not\ b_m = 1\} \geq k\ .$$
- A *normal rule* $h \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m$ corresponds to
$$h \leftarrow \{a_1 = 1, \ldots, a_n = 1, not\ b_1 = 1, \ldots, not\ b_m = 1\ \} \geq n + m\ .$$
- An *integrity constraint* $\leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m$ corresponds to
$$\begin{aligned} f \leftarrow \{\quad & a_1 = 1, \ldots, a_n = 1, \\ & not\ b_1 = 1, \ldots, not\ b_m = 1, not\ f = 1\} \geq n + m + 1 \end{aligned}$$
where f is a new atom used only in integrity constraints. Notice that if a program contains such an integrity constraint, then it cannot have a stable model S such that $\{a_1, \ldots, a_n\} \subseteq S$ *and* $\{b_1, \ldots, b_m\} \cap S = \emptyset$.

Next we show that a weight constraint rule program can be translated to basic constraint rules. This can be accomplished by a modular mapping where each weight constraint rule is translated independently of the other rules in the program, such that the stable models of weight constraint rules and basic constraint rules correspond. Moreover, the mapping increases the size of the program only linearly.

For a set of weight constraint rules P, its translation $\mathrm{tr}(P)$ is obtained by mapping each weight constraint rule $C_0 \leftarrow C_1, \ldots, C_r \in P$ to a set of basic constraint rules as follows. For each $i = 0, \ldots, r$, a constraint C_i of the form
$$l \leq \{a_1 = w_1, \ldots, a_n = w_n, not\ b_1 = w_{n+1}, \ldots, not\ b_m = w_m\} \leq u$$
is mapped into two rules encoding whether the lower bound is satisfied (c_i^l) and the upper bound is not (c_i^u).
$$c_i^l \leftarrow \{a_1 = w_1, \ldots, not\ b_m = w_m\} \geq l \tag{23}$$
$$c_i^u \leftarrow \{a_1 = w_1, \ldots, not\ b_m = w_m\} > u \tag{24}$$
where c_i^l, c_i^u are new atoms. If only integer weights are allowed, the latter rule can be expressed using '$\geq$' instead of '$>$', by incrementing u by one[2]. Then we add a choice rule and integrity constraints
$$\begin{aligned} \{a_1, \ldots, a_n\} &\leftarrow c_1^l, not\ c_1^u, \ldots, c_r^l, not\ c_r^u \\ &\leftarrow not\ c_0^l, c_1^l, not\ c_1^u, \ldots, c_r^l, not\ c_r^u \\ &\leftarrow c_0^u, c_1^l, not\ c_1^u, \ldots, c_r^l, not\ c_r^u \end{aligned}$$

[2]For handling real valued weights it is straightforward to extend the approach to allow weight rules with $>$.

where the first one selects a subset of the positive literals $\{a_1,\ldots,a_n\}$ in the head constraint C_0 of the original rule and the last two enforce that the lower and upper bounds of the head of the rule hold if the body of the rule is satisfied. Finally negative weights are eliminated as described in Section 2.2.

Example 5.1. The weight constraint rule

$$2 \leq \{a=2, b=3\} \leq 4 \leftarrow 3 \leq \{a=1, b=2, \mathit{not}\ c=3\} \leq 5$$

is translated into the basic constraint rules:

$$\begin{array}{llll}
c_0^l & \leftarrow \{a=2, b=3\} \geq 2 & \{a,b\} & \leftarrow c_1^l, \mathit{not}\ c_1^u \\
c_0^u & \leftarrow \{a=2, b=3\} \geq 5 & & \leftarrow \mathit{not}\ c_0^l, c_1^l, \mathit{not}\ c_1^u \\
c_1^l & \leftarrow \{a=1, b=2, \mathit{not}\ c=3\} \geq 3 & & \leftarrow c_0^u, c_1^l, \mathit{not}\ c_1^u \\
c_1^u & \leftarrow \{a=1, b=2, \mathit{not}\ c=3\} \geq 6 & &
\end{array}$$

Given a set of ground weight constraint rules, the translation captures its stable models in the following sense.

Theorem 5.2. *Let P be a set of weight constraint rules and* $\mathrm{tr}(P)$ *its translation to basic constraint rules described above. Then for each stable model S of P, there is a stable model S' of* $\mathrm{tr}(P)$ *such that $S = S' \cap \mathit{Atoms}(P)$ and for each stable model S' of* $\mathrm{tr}(P)$, *$S = S' \cap \mathit{Atoms}(P)$ is a stable model of P.*

Proof. ($\Rightarrow$) Suppose S is a stable model of P. Let $C_P(S)$ be the set of c_i^l and c_i^u atoms for which the body of the corresponding rule (23)/(24) is satisfied by S and let $S' = S \cup C_P(S)$. Now it is straightforward to show that $S' \models \mathrm{tr}(P)$. We show that S' is a stable model of $\mathrm{tr}(P)$ by establishing that S' is the smallest set closed under $\mathrm{tr}(P)^{S'}$, i.e., $S' = \mathrm{cl}(\mathrm{tr}(P)^{S'})$.

First we observe that S' is closed under $\mathrm{tr}(P)^{S'}$ (which we denote by $S' \models \mathrm{tr}(P)^{S'}$). Moreover, it can be shown that for any $S'' \subseteq S'$, if $S'' \models \mathrm{tr}(P)^{S'}$, then $(S'' \cap \mathit{Atoms}(P)) \models P^S$. But then $S \subseteq S'' \cap \mathit{Atoms}(P)$ as S is a stable model of P. Consider $a \in S' - \mathit{Atoms}(P)$, then $a \in C_P(S)$ and $a \in S''$ because $S'' \models \mathrm{tr}(P)^{S'}$. This is means that $S' \subseteq S''$ and that S' is the smallest set closed under $\mathrm{tr}(P)^{S'}$, i.e., a stable model of $\mathrm{tr}(P)$.

($\Leftarrow$) Suppose S' is a stable model of $\mathrm{tr}(P)$. Let $S = S' \cap \mathit{Atoms}(P)$. Then $S \models P$. Moreover, $S \models P^S$ holds. To show that S is a stable model of P it is sufficient to establish that S is the smallest such set. Consider a set $S'' \subseteq S$ with $S'' \models P^S$. It can be shown that $S^* \models \mathrm{tr}(P)^{S'}$ where $S^* = S'' \cup C_P(S'')$. Now $S' \subseteq S^*$ as S' is a stable model of $\mathrm{tr}(P)$. Hence, $S = S' \cap \mathit{Atoms}(P) \subseteq S^* \cap \mathit{Atoms}(P) = S''$ implying that S is a stable model of P. □

6 From Weight Constraint Rules with Variables to Basic Constraint Rules

Our implementation of general weight constraint programs is based a *two-level architecture* where in the first phase a *front-end* compiles a program with variables into a relatively simple *kernel language*. The computation tasks related to the program are then handled in the second phase by an *implementation of the kernel language* taking full advantage of its special features.

Our implementation of weight constraint rules, the SMODELS system, employs such an architecture and uses as the kernel language the basic constraint rules introduced in

Section 5. The main task of SMODELS is to compute a desired number of stable models for a weight constraint program. Given such a program as input the *front-end* of SMODELS, *lparse* [63], compiles the program into the kernel language such that the stable models are preserved. Then in the second phase an *implementation of the kernel language*, *smodels* [57], is used for completing the task, i.e., for computing the desired stable models. The algorithm for the kernel language and its implementation are developed in Sections 7–8. Below we discuss the front-end, i.e., mapping rules with variables to basic constraint rules.

The current SMODELS system allows stratified domain predicates with function symbols and takes as input domain-restricted weight constraint rules with variables, conditional literals, and built-in functions (see Section 2.4 and [64]). However, it supports only integer weights to avoid problems caused by the finite precision of real number arithmetic in standard programming languages.

The *lparse* [63] front-end instantiates and compiles weight constraint rules with variables into basic constraint rules through four steps as follows.

- During the first step the program is parsed and a maximal set of domain predicates is automatically detected.[3]

- In the second step the domain predicates are evaluated using database techniques by building a stratification of the rules defining domain predicates and evaluating them one stratum at a time starting from the lowest strata. This gives the unique stable model of the domain rules.

- In the third step the ground instances of the rest of the rules are computed one rule at a time by essentially evaluating relational joins of the domain predicates in the bodies. During this process conditional literals are also extended to sequences of literals and built-in functions are evaluated.

- The result of the three first steps (which can be interleaved) is a set of ground weight constraint rules. In the fourth step these rules are compiled into basic constraint rules using the translation described in Section 5.

7 The Algorithm for the Basic Constraint Rules

We now address the problem of implementing our kernel language, the basic constraint rules. First, we develop a procedure for computing stable models for a program without optimize statements. In the end of the section we show how the procedure can be extended to handle optimization. Throughout the section we use the term rule to mean a basic constraint rule. [4]

We take the obvious approach to computing stable models 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

[3] The front-end *lparse* supports directly the different kinds of short hand notations introduced in the paper. For the exact input syntax, the user's manual of *lparse* should be consulted.

[4] Our development of algorithms for computing stable models of basic constraint rules in Sections 7–8 is based on [56] where more details and proofs of correctness can be found. The treatment in [56] employs an interesting alternative way of defining stable models of basic constraint rules.

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 search space consists of all possible subsets A of the literals built from the atoms in the program. It 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 which 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.

7.1 The Decision Procedure

Algorithm 3 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 and to handle optimize statements.

The decision procedure calls the four functions $extend(P, A)$, $conflict(P, A)$, $stop(P, A)$ and $heuristic(P, A)$. The function $extend(P, A)$ is used for pruning the search space. The basic pruning procedure of our approach is the $expand(P, A)$ function expanding the set A using the functions $Atleast(P, A)$ and $Atmost(P, A)$. These two functions are developed in more detail below. In Algorithm 3, $extend(P, A)$ is implemented by directly calling $expand(P, A)$. However, it will later be improved by using a lookahead technique. The function $conflict(P, A)$ discovers conflicts. It calls the function $unacceptable(P, A)$ which will be modified when treating optimize statements. For the decision procedure it is sufficient to return false. Similarly, the function $stop(P, A)$ will be used for computing several stable models and for the decision procedure it is enough for it to return true. After pruning the search space, conflicts are checked for and if none are detected and the search space is exhausted (A covers $Atoms(P)$), then a stable model has been found. Otherwise the function $heuristic(P, A)$ is used for computing a heuristically good literal x that can be included in A and the search continues recursively covering the two cases, models agreeing with x and with $not\ x$, respectively.

First, we show that the decision procedure works correctly for any implementation of $extend(P, A)$, $conflict(P, A)$, and $heuristic(P, A)$ satisfying the following assumptions. Let $A' = extend(P, A)$. We assume that

```
function smodels(P, A)
  A := extend(P, A)
  if conflict(P, A) then
    return false
  else if A covers Atoms(P) then
    return stop(P, A) {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 extend(P, A)
  return expand(P, A).

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 unacceptable(P, A)
  end if.

function stop(P, A)
  return true.

function unacceptable(P, A)
  return false.
```

Algorithm 3: A decision procedure for the stable model semantics

E1 $A \subseteq A'$ and that

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

The function $\mathit{conflict}(P, A)$ is assumed to satisfy the two conditions

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

C2 if $\mathit{conflict}(P, A)$ returns true, then there is no stable model of P that agrees with A.

In addition, we expect $\mathit{heuristic}(P, A)$ to return a literal not covered by A.

Theorem 7.1. *Let P be a set of rules and A a set of literals. Then, there is a stable model of P agreeing with A if and only if $\mathit{smodels}(P, A)$ returns true.*

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

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

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

The key procedure for pruning the search space in *smodels* is the $\mathit{expand}(P, A)$ function which employs the functions $\mathit{Atleast}(P, A)$ and $\mathit{Atmost}(P, A)$. The idea is that $\mathit{Atleast}(P, A)$ bounds the stable models of P agreeing with A from below and $\mathit{Atmost}(P, A)$ from above so that *expand* satisfies the conditions E1 and E2. The function $\mathit{Atleast}(P, A)$ is built on the concepts of inevitable and possible consequences. Based on these we devise four ways of deriving literals to bound a stable model from below. Then $\mathit{Atleast}(P, A)$ is defined to be the least fixed point which extends A and is closed under derived literals.

Definition 7.2. Let P be a set of rules and let A be a set of literals. For a weight rule $r \in P$ 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 a set of literals B, let

$$\mathit{min}_r(B) = \begin{cases} \{h\} & \text{if } \sum_{a_i \in B} w_{a_i} + \sum_{\mathit{not}\ b_i \in B} w_{b_i} \geq w \\ \emptyset & \text{otherwise} \end{cases}$$

be the *inevitable consequences* of B. Similarly, for a choice rule $r \in P$ of the form $\{h_1, \ldots, h_k\} \leftarrow a_1, \ldots, a_n, \mathit{not}\ b_1, \ldots, \mathit{not}\ b_m$, let

$$\mathit{min}_r(B) = \begin{cases} \{h_1, \ldots, h_k\} \cap B & \text{if } a_1, \ldots, a_n, \mathit{not}\ b_1, \ldots, \mathit{not}\ b_m \in B \\ \emptyset & \text{otherwise.} \end{cases}$$

In addition, for a weight rule $r \in P$ let

$$max_r(B) = \begin{cases} \{h\} & \text{if } \sum_{a_i \notin B^-} w_{a_i} + \sum_{b_i \notin B^+} w_{b_i} \geq w \\ \emptyset & \text{otherwise} \end{cases}$$

be the *possible consequences* of B. For a choice rule $r \in P$ let

$$max_r(B) = \begin{cases} \{h_1, \ldots, h_k\} - B^- & \text{if } a_1, \ldots, a_n \notin B^- \text{ and } b_1, \ldots, b_m \notin B^+ \\ \emptyset & \text{otherwise.} \end{cases}$$

Set

$$\begin{aligned} f_P^1(B) &= \{a \in min_r(B) \mid a \in Atoms(P) \text{ and } r \in P\} \\ f_P^2(B) &= \{not\ a \mid a \in Atoms(P) \text{ and for all } r \in P,\ a \notin max_r(B)\} \\ f_P^3(B) &= \{not\ (x) \mid \text{there exists } a \in B \text{ such that } a \in max_r(B) \\ &\qquad \text{for only one } r \in P \text{ and } a \notin max_r(B \cup \{x\})\} \\ f_P^4(B) &= \{not\ (x) \mid \text{there exists } not\ a \in B \text{ and } r \in P \text{ such that} \\ &\qquad a \in min_r(B \cup \{x\})\}. \end{aligned}$$

and

$$f_P^5(B) = \begin{cases} Atoms(P) & \text{if } B^+ \cap B^- \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

Define $Atleast(P, A)$ as the least fixed point of

$$f(B) = A \cup B \cup f_P^1(B) \cup f_P^2(B) \cup f_P^3(B) \cup f_P^4(B) \cup f_P^5(B).$$

Example 7.3. 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 f_P^2. As $d \in A$, $not\ a \in A$ by f_P^3. It follows that $not\ b \in A$ by f_P^4. Finally, $e \in A$ by f_P^1. Hence, $Atleast(P, \{d\}) = \{not\ a, not\ b, not\ c, d, e\}$.

Lemma 7.4. *The function $Atleast(P, A)$ is monotonic in its second argument.*[5]

The functions $Atleast(P, A)$ extends A without loosing stable models.

Proposition 7.5. *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.

[5]For the proofs of the technical lemmata and propositions in this section, see [56].

Definition 7.6. Let P be a set of rules and let A be a set of literals. For a weight rule $r \in P$ of the form

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

and a set of atoms B, let

$$f_r(B) = \begin{cases} \{h\} & \text{if } \sum_{a_i \in B - A^-} w_{a_i} + \sum_{b_i \notin A} w_{b_i} \geq w \\ \emptyset & \text{otherwise.} \end{cases}$$

For a choice rule $r \in P$ of the form

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

let

$$f_r(B) = \begin{cases} \{h_1, \ldots, h_k\} & \text{if } a_1, \ldots, a_n \in B - A^- \text{ and } b_1, \ldots, b_m \notin A^+ \\ \emptyset & \text{otherwise.} \end{cases}$$

Define $Atmost(P, A)$ as the least fixed point of $f(B) = \bigcup_{r \in P} f_r(B) - A^-$.

Proposition 7.7. *Let S be a stable model of P that agrees with A. Then, $S \subseteq Atmost(P, A)$.*

Example 7.8. Let P be the program

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

Then, $Atmost(P, \emptyset) = \{a, c\}$ but $Atmost(P, \{not\ a\}) = \emptyset$.

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

Proposition 7.9. *If $A = expand(P, A)$ covers the set $Atoms(P)$ and $A^+ \cap A^- = \emptyset$, then A^+ is a stable model of P.*

Example 7.10. 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 7.11. 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 1 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.

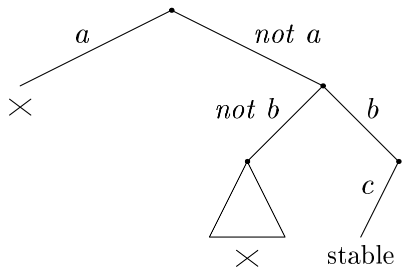


Figure 1: A visualization of the search process

7.2 Looking Ahead

The *extend* function implemented by *expand* 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 *extend* function must not lose 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 *extend* function does not help. On the other hand, if a partial model does not agree with any stable models, then *extend* should return as large a set as possible. A slower function is not such an objection then. This dichotomy is addressed here.

The *extend* function implemented by *expand* 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 $\mathit{expand}(P, A \cup \{a\})$ and $\mathit{expand}(P, A \cup \{\mathit{not}\ a\})$ contain conflicts, i.e., $\mathit{conflict}(P, A')$ returns true for $A' = \mathit{expand}(P, A \cup \{x\})$, where $x = a, \mathit{not}\ a$. If $\mathit{smodels}(P, A)$ 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 because in the worst case there are no other conflicts and a is chosen as the last alternative in an exponential number of leaf nodes of a search tree.

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 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 $A \cup \{\mathit{not}\,(x)\}$. 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 $\mathit{not}\,(x)$. In addition, since $x' \in \mathit{expand}(P, A \cup \{x\})$ implies

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

due to the monotonicity of *expand*, it is not even necessary to examine all literals in $\mathit{Atoms}(P)$ not covered by A. As we test a literal x, we can directly rule out all atoms in $\mathit{expand}(P, A \cup \{x\})$. We implement lookahead according to these observations by rewriting the function *lookahead* and *extend* as in Algorithm 4. Possibly calling *expand* on the order of $|\mathit{Atoms}(P) - \mathit{Atoms}(A)|$ times for each new literal might not seem like a good idea, but in practice it has proven amazingly effective.

function *extend*(P, A)
  $B := expand(P, A)$
  return $lookahead(P, B)$.

function *lookahead*(P, A)
  repeat
    $A' := A$
    $A := lookahead\_once(P, A)$
  until $A = A'$
  return A.

function *lookahead_once*(P, A)
  $B := Atoms(P) - Atoms(A)$
  $B := B \cup not\,(B)$
  while $B \neq \emptyset$ **do**
    Take any literal $x \in B$
    $A' := expand(P, A \cup \{x\})$
    $B := B - A'$
    if $conflict(P, A')$ **then**
      return $expand(P, A \cup \{not\,(x)\})$
    end if
  end while
  return A.

Algorithm 4: Looking ahead

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 [68]. 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.

7.3 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

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

and

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

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 literals have equal minimums, then the one with the greater maximum is chosen 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 5. 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 [23] 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.

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.

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 5: The heuristic

7.4 Searching for Optimal 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.

We control how many stable models are computed by calling a function $stop(P, A)$ when a stable model has been found. If $stop(P, A)$ returns false, then the algorithm will search for the next model. Otherwise, it stops. Since returning false from *stop* is indistinguishable from returning false from *conflict*, it is guaranteed that *stop* is only called for acceptable stable models. If *stop* 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 positive weights:

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

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 *stop* 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 *stop* functions are portrayed in Algorithm 6. If we want to compute all optimal models, then we could, e.g., remember all models A in the *stop* function that are as small as the current optimal one, discarding them if a smaller model is found.[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 *stop*(P, A)
  $B := A$
  return false.

Algorithm 6: Finding a minimal stable model

8 Implementation of the Algorithm

In the *smodels* algorithm most of the work, i.e. search space pruning and computation of the heuristic, is actually done by the *expand* function which is based on the functions $Atleast(P, A)$ and $Atmost(P, A)$. Hence, developing effective implementations for these functions is crucial for the efficiency of the *smodels* algorithm. We start by presenting the basis for an efficient implementation of the two functions. More details can be found in [56]. Then we show how to optimize the computation of $Atmost(P, A)$ using strongly connected components and source pointers. We show how to restrict the set of choice points and discuss how backtracking can be done space efficiently in *smodels*.

8.1 Implementing $Atleast(P, A)$ and $Atmost(P, A)$

Functions $Atleast(P, A)$ and $Atmost(P, A)$ can be implemented as 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 definite rules in time linear in the size of the set of rules. Definite rules are normal rules not containing 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.

The implementation of $Atleast(P, A)$ follows the basic Dowling-Gallier algorithm, but extends it to handle the four ways a literal can be included in $Atleast(P, A)$. We need

[6]The current implementation of *smodels* does not include such an extension but is able to find only one optimal model.

procedure Dowling-Gallier (P)
{Invariant: $|\text{body} - \text{closure}| = $ counter and 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.

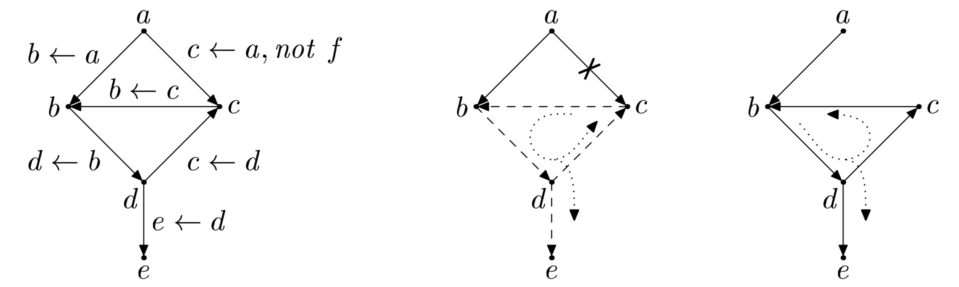
Algorithm 7: The basic Dowling-Gallier algorithm

three counters instead of one: one to keep track of how many literals in A appear in the body of a rule, one to keep track of how many literals in A appear negated in the body, and one to keep track of how many heads of still usable rules an atom appears in. For weight rules, the counters must keep track of the sum of the weights instead of the number of literals. We note that for efficiency the short hands for normal rules and cardinality rules are treated separately instead of translating them to proper basic constraint rules. More details about this can be found in [56].

The deductive closure $Atmost(P, A)$ can be computed in a similar style. 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 changes only a little, then a lot of extra work would be done if it were computed 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.



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

Example 8.1. An illustration of the two stages of $Atmost(P, A)$ is shown in Figure 2. The program P

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

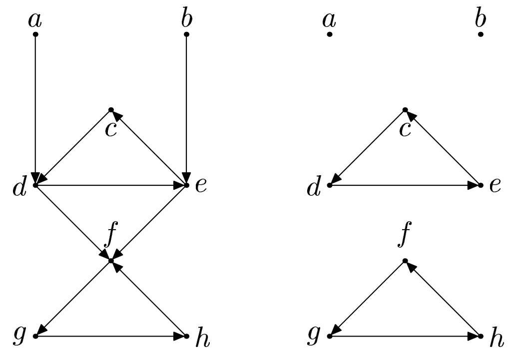
is displayed as a graph whose nodes are the atoms and whose edges are the rules that partake in the computation. The initial state is shown in 2(a) and corresponds to $Atmost(P, \emptyset) = \{a, b, c, d, e\}$. In 2(b) the atom f is added to $A = \emptyset$ and the rule $c \leftarrow a, not\ f$ becomes 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 2(c).

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 implementation of $Atmost(P, A)$ will in the worst case remove all atoms from the upper closure before adding them back again. If we could localize the computation of $Atmost(P, A)$ to small parts of the program, then we could avoid the worst case. Next we present two methods for localizing the computation of $Atmost(P, A)$.

$c \leftarrow e$
$d \leftarrow a$
$d \leftarrow c$
$e \leftarrow b$
$e \leftarrow d$
$f \leftarrow d$
$f \leftarrow e$
$f \leftarrow h$
$g \leftarrow f$
$h \leftarrow g$

(a) Original program



(b) Original program as a graph

(c) The strongly connected components

Figure 3: Localizing $Atmost(P, A)$ with strongly connected components

8.2 Strongly Connected Components

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

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

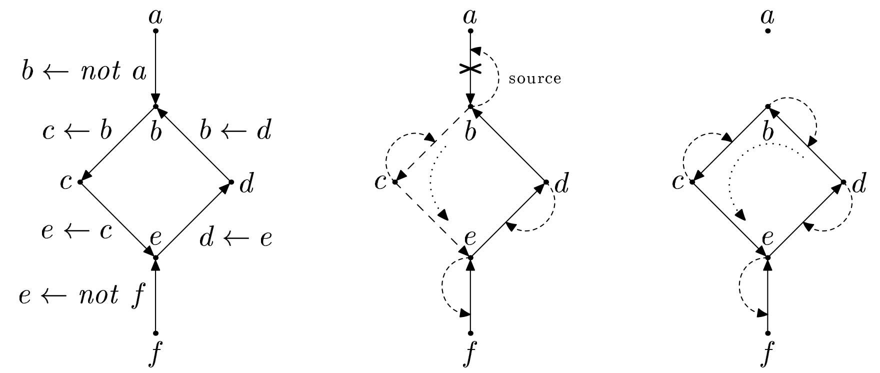
Take an atom $a \in Atmost(P, A)$. Observe that the only atoms that determine whether a is removed from the upper closure during the first stage of the computation are the atoms that have a directed path to a in G. Hence, we can localize $Atmost(P, A)$ by computing both stages inside a strongly connected 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.

Example 8.2. A partial program is shown in Figure 3(a). The corresponding graph is shown in Figure 3(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 the computation of $Atmost(P, A)$ 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 3(c). The first stage now removes only the atoms c, d, and e.

8.3 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.



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

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 the computation of $Atmost(P, A)$, 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.

Example 8.3. 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 the computation of $Atmost(P, A)$ are shown in Figure 4. The initial state is shown in 4(a) and corresponds to $Atmost(P, \emptyset) = \{b, c, d, e\}$. In 4(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 4(c).

8.4 Reducing the Search Space

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.

First we point out that a stable model of a program P is in effect determined by a set of atoms that appear in the heads of the choice rules or as not-atoms in a cycle in a directed graph $G = (V, E)$ where $V = Atoms(P) \cup P$ and

$$E = \{\langle r, a\rangle \mid a \text{ appears in the head of } r\} \cup \{\langle a, r\rangle \mid a \text{ or } not\ a \text{ appear in the body of } r\}.$$

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

Proposition 8.4. *Let P be a program and B the set of atoms that appear as not-atoms in some cycles or in the heads of some choice rules. Let S and S' be stable models of P. If $S \cap B = S' \cap B$, then $S = S'$.*

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 8.5. *Let P be a program and B the set of atoms that appear as not-atoms in some cycles or 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 $Atoms(P)$.*

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 8.6. Let P be the program

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

The set of relevant choice points of P is $B = \{a, b\}$. The atom d is not included as it does not appear as a not-atom in the program and c is not included since it does not appear as a not-atom in a cycle.

8.5 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.

It is enough to keep track of the changes to the set A. Generally, the value of every counter of a rule 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.

9 Comparison with other Algorithms

We compare the *smodels* algorithm with some advanced algorithms for computing stable models of logic programs and with the Davis-Putnam procedure [10] for determining the satisfiability of propositional formulas.

9.1 Stable Model Algorithms

We begin by describing the stable model algorithms and examine the branch and bound algorithm of [61], the SLG system of [7], the mixed integer programming method of [3], the modified Davis-Putnam method of [12], and the `dlv` algorithm [21, 22].

Branch and Bound. The branch and bound algorithm of [61] 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 [66] 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 method of [3] 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 [4] 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 $\mathit{Atleast}(P, A)$ function. 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` Algorithm. The `dlv` algorithm of [21] computes stable models for disjunctive programs. It is based on backtracking search with pruning similar to *smodels*. When it searches for stable models of normal programs, it prunes the search space as the $\mathit{Atleast}(P, A)$ function. The heuristic is based on using lookaheads but does not combine information from the lookaheads of an atom and its complement as *smodels* does. This approach is adopted in a more recent heuristic [22] but there the heuristic value given by a lookahead is not based on the number of undefined atoms as in *smodels* but on the number of unsupported true atoms and unsatisfied rules.

9.2 Comparison

We compare the branch and bound algorithm, the SLG system, the mixed integer programming approach, the modified Davis-Putnam method, the `dlv` algorithm, and the Davis-Putnam procedure [10] for determining the satisfiability of propositional formulas with the *smodels* algorithm.

The Stable Model Algorithms. 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 two approaches 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` algorithm [21] do not prune the search space using the upper closure. Hence, they prune less than *smodels*. It would be quite easy to integrate the upper closure computation into the `dlv` algorithm as it is similar to *smodels*. Including the upper closure computation into the modified Davis-Putnam method seems to be more challenging.

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 prunes the search space more than the modified Davis-Putnam method and the `dlv` algorithm. As the pruning in *smodels* can be efficiently implemented, *smodels* computes stable models faster than the other systems for a program that requires substantial search in the other systems.

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.

The main difference between the methods comes from the different underlying semantics. 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', e.g., $a \vee b$ subsumes $a \vee b \vee \neg c$. Subsumed clauses can be removed from a formula without changing its set of models. 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 does 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

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

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 pruning rule. Moreover, it is open what kind of pruning rules not preserving models could be used in the presence of optimize statements.

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. 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*.

10 Experiments

In order to study the performance of the kernel language implementation, the *smodels* algorithm, we test an implementation of it, *smodels* version 2.26 [57], on two combinatorially hard problems: determining propositional satisfiability and Hamiltonian cycles. We compare *smodels* against `dlv` [30], a state-of-the-art stable model implementation capable of handling also disjunctive programs. Since there are few competitive systems for computing stable models, we compare *smodels* with three propositional satisfiability checkers: tableau or *ntab* [9], *SATO* 3.2.1 [70], and *satz*215 [34, 33]. 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.

All tests were run under Linux 2.2.17 on a 1 GHz AMD Athlon computer with 512 MB of memory and are available at `http://www.tcs.hut.fi/Software/smodels/tests/sns01-tests.tar.gz`. The durations of the tests are given in seconds and they represent the time to find a solution or 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 (branch points) describes how many times the algorithms use their heuristics to decide which atom to test next. Testing was done by choosing a size parameter and generating 50 instances for each selected value of the parameter. For measuring performance, we used the average running time and average number of choice points of the set of instances of the same size. A cut-off limit of 20 CPU hours was imposed, i.e., a test run on an instance was interrupted when the limit was exceeded, leaving the average running time and number of choice points of that size undefined.

10.1 3-SAT

The first 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 so 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

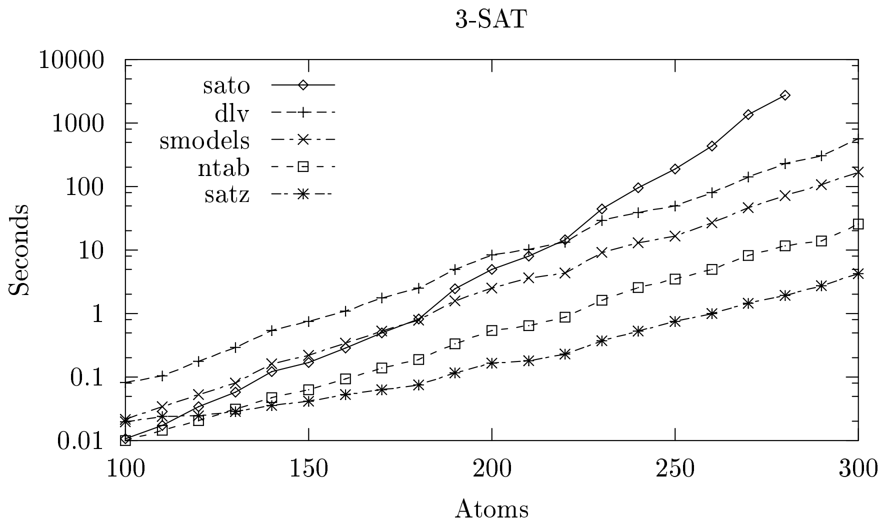


Figure 5: 3-SAT, average duration in seconds

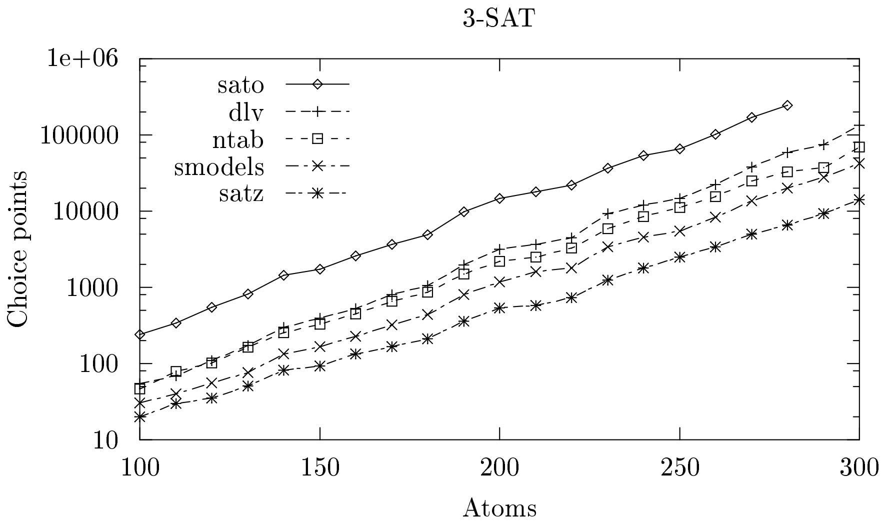


Figure 6: 3-SAT, average number of choice points

atoms, but *satz* uses a more sophisticated heuristic. The program *satz* also does some preprocessing during which it adds clauses to the problem.

We translate clauses in a 3-SAT problem into a set of basic constraint rules as described in Section 3.1. When we test `dlv`, we use the rules $a_i \leftarrow not\ \bar{a}_i$ and $\bar{a}_i \leftarrow not\ a_i$ instead of a choice rule $\{a_i\} \leftarrow$.

We test the systems on problems having from 100 to 300 atoms in increments of 10. For each problem size we generate 50 satisfiability problems using a program developed by Selman [54]. The test results are shown in Figures 5–6.

The implementation of *smodels* prunes the search space more than *SATO*, `dlv`, 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

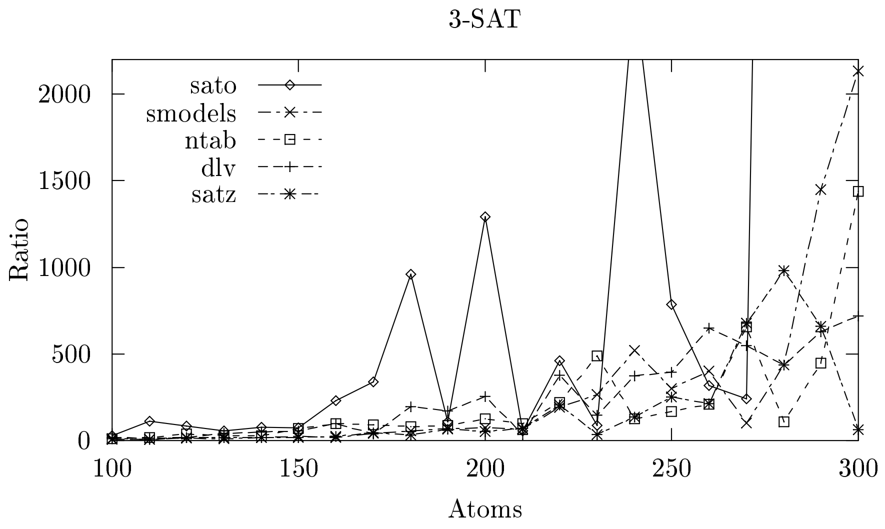


Figure 7: 3-SAT, choice points max/min ratio

how *smodels* and *satz* propagate truth values. The *satz* procedure makes use of the pure literal rule. 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 the changes in the set of clauses. Let p and n be a measure of the change when the truth value of an atom is set to respectively true or false. Then, *satz* maximizes

$$1024np + n + p$$

but *smodels* maximizes $\min(n, p)$ and $\max(n, p)$, where $\min(n, p)$ is more significant, i.e. $\min(n, p)$ is maximized first and if two atoms have equal minimums, then the one with the greater maximum is chosen (see Section 7.3). We refer to these formulas as the cost functions of *smodels* and *satz*. In order to determine whether it is the heuristic of *satz* that is the reason for the better pruning, we have experimented with 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 results indicate that the variants prune the search space as much or slightly less than the original versions suggesting that the heuristic of *satz* is better than that of *smodels* on hard 3-SAT problems.

For studying the tightness of the heuristics we compute the ratio between the largest and smallest number of choice points needed to solve a random 3-SAT instance of a given size. The results are shown in Figure 7. *SATO* has the loosest heuristic allowing the most variation on the number of choice points among instances of the same size. Also the heuristics of *smodels* and *ntab* seem to lose their tightness as the size approaches 300 atoms whereas `dlv` and, in particular, *satz* remain tight.

10.2 Hamiltonian Cycles

We note that 3-SAT problems correspond to a very limited subclass of rules. In order to study the performance of *smodels* with more complicated programs involving, e.g., recursive definitions, we consider the Hamiltonian cycle problem. First, we develop encodings of the problem using logic programs and propositional logic and then discuss the experimental results.

A Hamiltonian cycle is a cycle in a graph visiting 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

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

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 fact $v_1 \leftarrow$ and the rules

$$w \leftarrow v, e_{v,w} \qquad \{v, w\} \in E \text{ and } w \neq v_1. \tag{25}$$

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 rules

$$\leftarrow \mathit{not}\ v \qquad v \in V.$$

When testing `dlv`, we use the rules $e_{v,w_i} \leftarrow \mathit{not}\ \bar{e}_{v,w_i}$ and $\bar{e}_{v,w_i} \leftarrow \mathit{not}\ e_{v,w_i}$, for $\{v, w_i\} \in E$, instead of the choice rule. We replace each cardinality rule

$$\leftarrow k\,\{p_1, \ldots, p_n\}$$

with the rules

$$\leftarrow p_{i_1}, \ldots, p_{i_k}, \qquad 1 \leq i_1 < \cdots < i_k \leq n.$$

In order to get an idea how different encodings behave we study an alternative encoding with normal programs which employs a more recursive definition[7]. The idea is to see an undirected graph as a directed one where the edge relation is symmetric and irreflexive. Given the graph as a set of facts $arc(v, u) \leftarrow$ enumerating such an edge relation, it is sufficient to pick a vertex $v_1 \in V$, add a fact $start(v_1) \leftarrow$ and rules with variables

$$\begin{array}{rl} \{h(X,Y)\} & \leftarrow start(X), arc(X,Y) \\ \{h(X,Y)\} & \leftarrow reached(X), arc(X,Y) \\ reached(Y) & \leftarrow h(X,Y) \\ & \leftarrow h(X,Y), h(X,Z), Y \neq Z \\ & \leftarrow h(X,Y), h(Z,Y), X \neq Z \\ & \leftarrow arc(X,Y), \mathit{not}\ reached(X) \end{array} \tag{26}$$

[7] This encoding was suggested by a reviewer of the paper.

where predicates $h(X, Y)$ and $reached(Y)$ have mutually recursive definitions. This should be compared to the first encoding where such mutual recursion is avoided: the recursive reachability condition (25) depends on the selected edges but not vice versa. Now the resulting program has a stable model exactly when the graph has a Hamiltonian cycle. A stable model provides a Hamiltonian cycle as facts $h(v, u)$ true in the model giving the edges of the cycle. For `dlv` we replace the choice construct $\{h(X, Y)\}$ in the head by a disjunction $h(X, Y) \vee out(X, Y)$. In order to get a ground (variable free) encoding, we employ `dlv` using the '`-instantiate`' option to produce a ground program from the set of facts giving the edge relation and rules above. The corresponding ground program for *smodels* is obtained by simply replacing the disjunctions in the heads by corresponding choice constructs.

We translate the Hamiltonian cycle problem into a propositional formula in conjunctive normal form following Papadimitriou [47]. 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:

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

Hence, if the graph has n vertices, then we need n^2 atoms and $\mathcal{O}(n^3)$ clauses.

One can also create a more complex Hamiltonian cycle translation that mirrors the logic program 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

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

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

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

As benchmarks we use the Hamiltonian cycle problem on a special type of random planar graphs created by the *plane* function in the Stanford GraphBase [28]. We test *smodels* using three encodings: the first logic program encoding with cardinality constraints, its translation into normal rules, and the recursive encoding (26). For `dlv` we use two encodings: the normal rule translation of the first encoding and the recursive encoding. The two propositional encodings are tested using *satz* and *SATO*. We generate 50 graphs for each problem size (measured by the number of vertices). For each system and encoding we start from 10 vertices and increase the size in increments of two until the cut-off limit (20 CPU hours) is exceeded by an instance. All the graphs used in the tests have Hamiltonian cycles.

The results are displayed in Figures 8–9. We observe that the first satisfiability encoding scales better for both satisfiability checkers but *satz* performs substantially better than *SATO*. However, *satz* scales worse than *smodels* and `dlv` when using the first logic program encoding or its translation into normal rules. Apparently, the heuristic of *satz* does not work well on these problems and the rapidly growing size of the propositional encoding creates overhead. For *smodels* there is little difference between the first encoding using cardinality constraints and its translation into normal rules. This seems to be due to the small size of the sets in the cardinality constraints (number of adjacent vertices to a vertex). In fact, for normal rules the heuristic of *smodels* appears to scale slightly better compensating the overhead caused by the additional rules needed to encode the cardinality constraints. Using the normal program version of the first encoding, `dlv` scales worse than *smodels* and the cut-off limit is met at 50 vertices whereas for *smodels* it is reached at 68 vertices. For the encoding employing mutual recursion both systems scale worse but the heuristic of `dlv` works better.

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 using a modular embedding. The converse does not hold as discussed in Section 4.3. It is 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.

11 Conclusions

We have introduced a novel answer set programming language, weight constraint rules, that extends normal logic programs. The generalization is motivated by the need to represent weight and cardinality constraints in many applications. The language supports the use of variables and includes optimization capabilities. The weight constraint rules are given a declarative semantics generalizing the stable model semantics of normal programs. However, the complexity of computing stable models for the novel rules is found to be similar to that of normal programs in the ground case without optimization.

Weight constraint rules can be embedded into a subclass of the language called basic constraint rules in a relatively simple manner. We have developed an implementation of

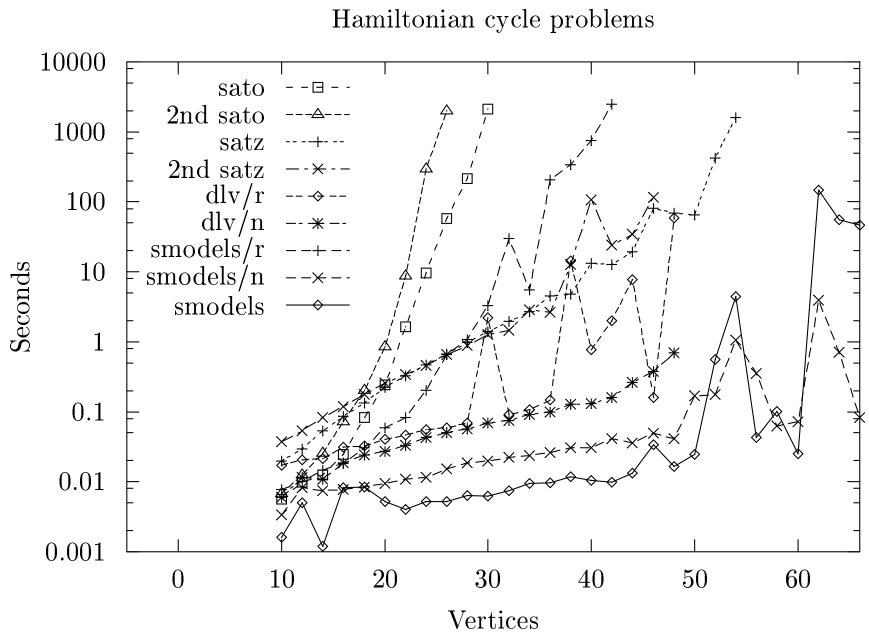


Figure 8: Hamiltonian cycle problem, average duration in seconds

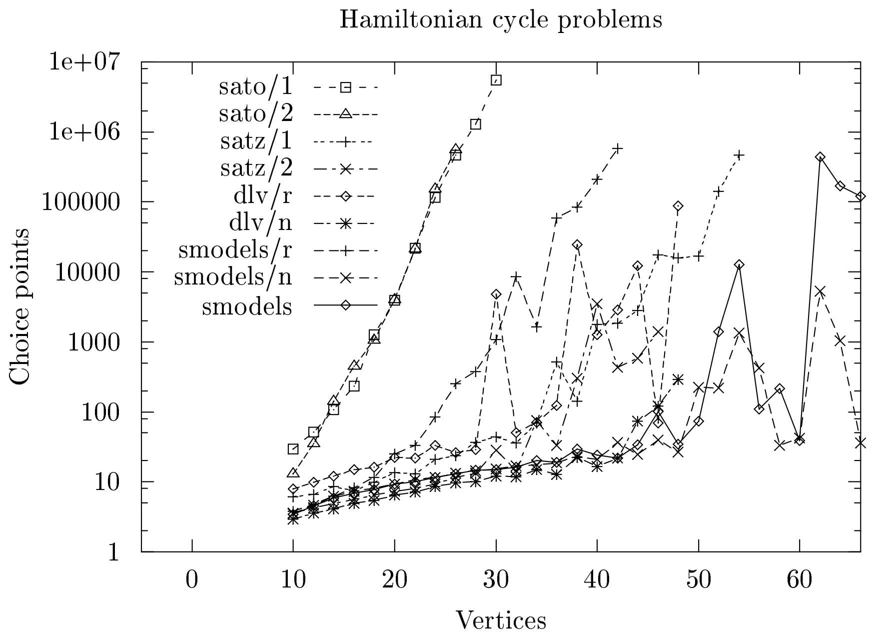


Figure 9: Hamiltonian cycle problem, average number of choice points

the language based on this embedding. It computes stable models of a weight constraint rule program by compiling the program to basic constraint rules and by then using an efficient search procedure, *smodels*, to compute stable models of the compiled rule set. A major part of this work is the development of the *smodels* algorithm and its efficient implementation.

We have compared *smodels* with three good satisfiability solvers and an implementation of the stable model semantics for disjunctive logic programs on random satisfiability problems and Hamiltonian cycle problems. The best satisfiability solver goes through a smaller search space than our algorithm when testing on hard random 3-SAT problems. We have attributed the difference to the different heuristics of the procedures, and have found indications that our heuristic can be refined to work better. Our algorithm scales better than the other systems when it comes to the Hamiltonian cycle problem. 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 of weight constraint rules has a computational overhead. But the overhead provides us with a more powerful language. Consequently, problems that can be more compactly represented by the novel rules can be more efficiently solved with *smodels* than with a satisfiability checker.

In order to develop the weight constraint rule language to better meet application requirements, the implementation could be extended by including real-valued arithmetic and real-valued weights for literals. Further work is also needed for developing more efficient implementation techniques for the optimization task. A key to the efficiency of the *smodels* algorithm is the use of the one literal lookahead technique. Hence, 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.

Acknowledgements

We thank the anonymous referees for their useful comments and suggestions. The work of the first author has been funded by the Academy of Finland (Project 43963) and the Helsinki Graduate School in Computer Science and Engineering (HeCSE), the second author by the Academy of Finland (Project 43963) and the third by HeCSE and the Technology Development Centre of Finland. We thank Tommi Syrjänen for implementing the *lparse* front-end for the SMODELS system.

References

[1] L.C. Aiello and F. Massacci. Verifying security protocols as planning in logic programming. *ACM Transactions on Computational Logic*, 2(4):542–580, 2001.

[2] T. Aura, M. Bishop, and D. Sniegowski. Analyzing single-server network inhibition. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 108–117, Cambridge, UK, July 2000. IEEE Computer Society Press.

[3] 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.

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

[5] F. Buccafurri, N. Leone, and P. Rullo. Strong and weak constraints in disjunctive datalog. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning*, pages 2–17. Springer-Verlag, 1997.

[6] Marco Cadoli, Luigi Palopoli, Andrea Schaerf, and Domenico Vasile. NP-SPEC: An executable specification language for solving all problems in NP. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 16–30, San Antonio, Texas, January 1999. Springer-Verlag.

[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] D. East and M. Truszczyński. DATALOG with constraints – an answer-set programming system. In *Proceedings of the 17th National Conference on Artificial Intelligence*, pages 163–168, Austin, Texas, USA, July/August 2000. The MIT Press.

[17] T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15:289–323, 1995.

[18] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive datalog. *ACM Transactions on Database Systems*, 22(3):364–418, 1997.

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

[20] 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.

[21] W. Faber, N. Leone, and G. Pfeifer. Pushing goal derivation in DLP computations. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 177–191, El Paso, Texas, USA, December 1999. Springer-Verlag.

[22] W. Faber, N. Leone, and G. Pfeifer. Optimizing the computation of heuristics for answer set programming systems. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 295–308, Vienna, Austria, September 2001. Springer-Verlag.

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

[24] 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.

[25] S. Greco. Dynamic programming in Datalog with aggregates. *IEEE Transactions on Knowledge and Data Engineering*, 11(2):265–283, 1999.

[26] K. Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets. *Fundamenta Informaticae*, 37(3):247–268, 1999.

[27] M. Hietalahti, F. Massacci, and I. Niemelä. DES: a challenge problem for nonmonotonic reasoning systems. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (cs.AI/0003073)*, Breckenridge, Colorado, USA, April 2000. cs.AI/0003039.

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

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

[30] N. Leone et al. Dlv, release 2001-06-11. http://www.dbai.tuwien.ac.at/proj/dlv/, 2001. A Disjunctive Datalog System.

[31] 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.

[32] N. Leone, L. Rullo, and F. Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Information and Computation*, 135(2):69–112, 1997.

[33] C.M. Li. A constraint-based approach to narrow search trees for satisfiability. *Information Processing Letters*, 71:75–80, 1999.

[34] C.M. Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, pages 341–355, Linz, Austria, October/November 1997. Springer-Verlag.

[35] 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.

[36] V. Lifschitz. Answer set planning. In *Proceedings of the 16th International Conference on Logic Programming*, pages 25–37, Las Cruces, New Mexico, December 1999. The MIT Press.

[37] V. Lifschitz and T.Y.C. Woo. Answer sets in general nonmonotonic reasoning (preliminary report). In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning*, pages 603–614, Cambridge, MA, USA, October 1992. Morgan Kaufmann Publishers.

[38] X. Liu, C.R. Ramakrishnan, and S.A. Smolka. Fully local and efficient evaluation of alternating fixed points. In *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 5–19, Lisbon, Portugal, March/April 1998. Springer-Verlag.

[39] V.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.

[40] V.W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K.R. Apt, V.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.

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

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

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

[44] I. Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.

[45] I. Niemelä and P. Simons. Extending the Smodels system with cardinality and weight constraints. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, chapter 21, pages 491–521. Kluwer Academic Publishers, 2000.

[46] I. Niemelä, P. Simons, and T. Soininen. Stable model semantics of weight constraint rules. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 317–331, El Paso, Texas, USA, December 1999. Springer-Verlag.

[47] C.H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1995.

[48] 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.

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

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

[51] J. Rintanen. Lexicographic priorities in default logic. *Artificial Intelligence*, 106:221–265, 1998.

[52] 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.

[53] 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.

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

[55] P. Simons. Extending the stable model semantics with more expressive rules. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 305–316, El Paso, Texas, USA, December 1999. Springer-Verlag.

[56] P. Simons. Extending and implementing the stable model semantics. Doctoral Dissertation. Research report A58, Helsinki University of Technology, Helsinki, Finland, April 2000.

[57] P. Simons. Smodels 2.26. http://www.tcs.hut.fi/Software/smodels/, 2000. A system for computing the stable models of logic programs.

[58] T. Soininen, E. Gelle, and I. Niemelä. A fixpoint definition of dynamic constraint satisfaction. In Joxan Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming*, pages 419–433, Alexandria, Virginia, USA, October 1999. Springer-Verlag.

[59] T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 305–319, San Antonio, Texas, January 1999. Springer-Verlag.

[60] T. Soininen, I. Niemelä, J. Tiihonen, and R. Sulonen. Representing configuration knowledge with weight constraint rules. In *Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming*, pages 195–201, Stanford, USA, March 2001. AAAI Press.

[61] 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.

[62] T. Syrjänen. Including diagnostic information in configuration models. In *Proceedings of the First International Conference on Computational Logic, Automated Deduction: Putting Theory into Practice*, pages 837–851, London, U.K., July 2000. Springer-Verlag.

[63] T. Syrjänen. lparse, a procedure for grounding domain-restricted logic programs. http://www.tcs.hut.fi/Software/smodels/lparse/, 2001.

[64] T. Syrjänen. Omega-restricted programs. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 267–279, Vienna, Austria, September 2001. Springer-Verlag.

[65] M. Truszczynski et al. DeReS, a default reasoning system. http://www.cs.engr.uky.edu/ai/deres.html, 1999.

[66] 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.

[67] D.S. Warren et al. The XSB programming system. http://www.cs.sunysb.edu/~sbprolog/xsb-page.html, 2000.

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

[69] C. Zaniolo et al. LDL^{++}, a second-generation deductive database system. http://www.cs.ucla.edu/ldl/, 1999.

[70] H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the 14th International Conference on Automated Deduction*, pages 272–275, Townsville, North Queensland, Australia, July 1997. Springer-Verlag.