Extending the Smodels System with Cardinality and Weight Constraints

Ilkka Niemelä and Patrik Simons
Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science
P.O.Box 5400, FIN-02015 HUT, Finland
email: {Ilkka.Niemela,Patrik.Simons}@hut.fi

Abstract

The `Smodels` system is one of the state-of-the-art implementations of stable model computation for normal logic programs. In order to enable more realistic applications, the basic modeling language of normal programs has been extended with new constructs including cardinality and weight constraints and corresponding implementation techniques have been developed. This paper summarizes the extensions that have been included in the system, demonstrates their use, provides basic application methodology, illustrates the current level of performance of the system, and compares it to state-of-the-art satisfiability checkers.

1 Introduction

Recently logic programs with the stable model semantics have emerged as a novel paradigm for applying declarative logic programming techniques. We call the paradigm *answer set programming*, a term that was coined by Vladimir Lifschitz. It nicely captures the main idea behind the approach: solutions or answers are represented as sets of objects. Furthermore, logic programs with the answer set semantics [GL90], a generalization of the stable model semantics to handle programs with classical negation, can be seen as an instance of the approach. Answer set programming has been proposed as a methodology for solving, e.g., combinatorial, graph and planning problems [Lif99, MT99b, Nie99].

We have been developing an implementation for the stable model semantics of normal logic programs called the `Smodels` system (available at `http://www.tcs.hut.fi/Software/smodels/`). When working towards applications we have observed that (i) the applicability of the system can be widened by extending the underlying language of normal rules with more expressive constructs for representing choices and restrictions on them and (ii) that such extensions can be supported with efficient implementation techniques.

The purpose of this paper is to summarize the extensions that we have included in our system, demonstrate their use, provide basic application methodology, and illustrate the current level of performance of our system.

We start by explaining the basic ideas how the answer set programming approach can be used for applying logic programs with the stable model semantics. More details can be found in [MT99b, Nie99]. In answer set programming a problem is solved by devising a logic program such that the stable models of the program provide the answers to the problem and by using an implementation of the stable model semantics to compute an answer, i.e., a stable model of the resulting program. As an example consider the 3-coloring problem, i.e., 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 build a program where for each vertex v in the graph we take the three rules on the left and for each edge (v, u) the three rules on the right:

$$\begin{array}{lr} v(1) \leftarrow \text{not } v(2), \text{not } v(3) & \leftarrow v(1), u(1) \\ v(2) \leftarrow \text{not } v(1), \text{not } v(3) & \leftarrow v(2), u(2) \\ v(3) \leftarrow \text{not } v(1), \text{not } v(2) & \leftarrow v(3), u(3) \end{array} \tag{1}$$

Stable models are sets of ground atoms and, for instance, the first rule on the left says that if neither of $v(2), v(3)$ 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 atoms $v(1)$ and $u(1)$ and similarly for the other two rules. Hence, the rules on the right exclude any stable model having atoms $v(n), u(n)$ 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.

Normal logic programs provide a framework where a variety of combinatorial, constraint satisfaction, and planning problems can be handled [Nie99]. However, there are conditions which are hard to capture using normal programs. These are typically related to choices with cardinality, costs and resources. Consider as an example the vertex cover problem where the task is to find a subset of vertices of size less than k such that for each edge (v, u) at least one of the vertices v, u is included in the cover. Given a suitable notion of *cardinality constraints* vertex covers can be captured for a given graph as follows. For each edge (v, u) in the graph a rule

$$1\ \{v, u\}\ \leftarrow \tag{2}$$

is included and then an integrity constraint type of a rule

$$\leftarrow k\ \{v_1, \ldots, v_n\} \tag{3}$$

is added where $\{v_1, \ldots, v_n\}$ is the set of vertices in the graph. The first rule expresses a choice with a cardinality constraint saying that at least one end point for each edge should be selected and the second rule states a cardinality restriction saying that the cover must have size less than k (if the cover contains k or more vertices, it is not acceptable). Such a choice rule (2) is not expressible by normal rules without introducing new atoms in the program because for normal rules stable models are subset minimal, i.e., a stable model cannot be a proper subset of another stable model. However, for rules with cardinality constraints this is possible. Furthermore, there seems to be no compact encoding of the cardinality restriction (3) using normal rules.

Hence, cardinality constraints appear to be a useful extension to normal logic programs. However, sometimes even more general constraints are needed, e.g., when representing conditions about resources and costs. As an example consider the knapsack problem where we have a set of items with given values and weights and we need to choose a subset such that a given weight limit W is not exceeded but a value limit V is achieved. Such constraints can be captured by generalizing cardinality constraints. The idea is that in a cardinality constraint every atom is treated as having a weight of one and a natural generalization is to allow arbitrary weights. Now the knapsack problem can be captured using the following rules:

$$0 \le \{i_1 = w_1, \ldots, i_n = w_n\} \le W \leftarrow \tag{4}$$

$$\leftarrow \{i_1 = v_1, \ldots, i_n = v_n\} \le V \tag{5}$$

The first rule expresses a selection of a subset of the items $\{i_1, \ldots, i_n\}$ such that the sum of the weights of the selected items is at most W and the second rule states a constraint eliminating all subsets of the items whose sum of values is not more than V.

Such weight constraints provide an expressive and uniform framework for handling large classes of combinatorial problems. The `Smodels` system takes weight constraints as the basic building blocks of the rules and considers *weight rules* of the form

$$C_0 \leftarrow C_1, \ldots, C_n \tag{6}$$

where each C_i is a weight constraint of the form

$$L \leq \{a_1 = w_1, \ldots, a_n = w_n, \text{not } a_{n+1} = w_{n+1}, \ldots, \text{not } a_m = w_m\} \leq U \tag{7}$$

and every a_i is an atomic formula. This is a generalization of normal logic program rules where each C_i is a literal, i.e., an atomic formula a and its negation not a.

Weight rules have been given a declarative nonmonotonic semantics [NSS99] that extends the stable model semantics of normal logic programs [GL88] and generalizes the propositional choice rules presented in [SN99]. There are a number of approaches where priorities, preferences, costs, probabilities or certainty factors are associated to rules, see e.g. [BE98, MT99a, NS94, LNS96] and the references there. This is different from weight rules where the aim is to provide a relatively simple way of associating weights or costs to atoms and representing constraints using the weights. Approaches such as NP-SPEC [CPSV99], constraint logic programs (CLP) [JL87] and constraint satisfaction problems [Tsa93] are not based on stable model semantics and thus do not include, e.g., default negation. In addition, the semantics of weight rules treats the constraints, rules and choices uniformly unlike the CLP and NP-SPEC approaches. There is also some related work based on stable models. For example, in [BLR97] priorities are added to integrity constraints. However, this is done to express *weak constraints*, as many of which as possible should be satisfied, and not weight constraints which must all be satisfied. In [Gre99] 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 stratified negation and choice constructs.

Disjunctive logic programs allow rules which have disjunctions in the heads of the rules and seem to be close to rules with cardinality constraints. However, weight rules are more general than disjunctive rules, since they allow cardinality and weight constraints in both the head and the body of a rule. Moreover, there is a fundamental difference in the semantical basis because the main semantics of disjunctive programs are built on the paradigm of minimal models [EGM97]. This means that a model of the program cannot be a proper subset of another model. An exception is the possible model semantics [SI94] where non-minimal models are allowed. Weight rules may also have non-minimal models and, indeed, rules under the possible model semantics can be seen as a simple subclass of rules with cardinality constraints [SN99].

There are a number of implementations of declarative semantics of logic programs already available. For example, the `XSB` system [W$^+$99] is a WAM-based full logic programming system supporting the well-founded semantics and `LDL`$^{++}$ [Z$^+$99] is a deductive database system based on the stable model semantics supporting stratified programs with choice constructs and user defined aggregates. There are also systems supporting full stable model semantics. The `DeReS` system [T$^+$99] 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` and `Smodels` systems have been developed specially for computing stable models of logic programs. The main differences are that `dlv` handles disjunctive logic programs and that considerable emphasis has been given to integrating deductive database techniques in `dlv`, e.g., for evaluating efficiently recursive stratified programs. The `Smodels` system does not handle disjunctive rules but supports normal programs and recursive database query evaluation techniques have not been included in `Smodels`. However, `Smodels` has been extended to support both cardinality and weight constraints for rules with variables and built-in functions. Also constructs for solving optimization problems have been added.

The rest of the paper is organized as follows. We start by discussing the underlying stable model semantics for weight rules and illustrate the basic constructs with a number of examples. Then we present a simple generate and test method that can be used as a starting point for developing applications. The method is demonstrated with examples on propositional satisfiability and Hamiltonian cycles. In Section 4 we briefly explain the techniques that have been developed for implementing weight rules. Then we show the results of some experiments on propositional satisfiability and Hamiltonian cycle problems where the `Smodels` system is compared with `dlv` and with state-of-the-art propositional satisfiability checkers.

2 Weight Rules and the Stable Model Semantics

In this section we explain the stable model semantics for weight rules

$$C_0 \leftarrow C_1, \ldots, C_n$$

originally introduced in [NSS99] through several steps. First we discuss ground rules, i.e., rules where no variables are allowed. We start with the original stable model semantics for normal rules where each constraint C_i is a literal. Then we move to rules where all constraints are cardinality constraints. This provides a suitable setting for explaining the basic intuition behind extending the stable model semantics. Then we show how to generalize the semantics to weight constraints. We discuss the constructs that have been provided for solving optimization problems and finally present the method for integrating variables and built-in functions used in the `Smodels` system.

2.1 Normal Rules

The stable model semantics [GL88] generalizes the minimal model semantics of definite programs to normal logic program rules

$$A \leftarrow B_1, \ldots, B_m, \text{not } C_1, \ldots, \text{not } C_n \tag{8}$$

where negative body literals (not C_i) are allowed. For a variable-free (ground) program P, the stable models are defined as follows. The *reduct* P^S of a program P with respect to a set of atoms S is the program obtained from P by deleting

- each rule that has a negative literal not C in its body with $C \in S$ and then
- all negative literals in the remaining rules.

The reduct P^S can be seen as the set of potentially applicable rules given the stable model S, i.e., as the rules where the negative body literals are satisfied by the model. Note that in the reduct the negative body literals of the potentially applicable rules are removed and, hence, the rules are definite. The idea is that a stable model should be *justified* in the sense that every atom in the model is a consequence of the potentially applicable rules and every consequence of the potentially applicable rules is included in the model. The atomic consequences of a set of definite rules can be captured as the *deductive closure* of such rules defined in the following way. A set of atoms is *closed* under a set of rules if each rule is satisfied by the atom set, i.e., if the body atoms are in the set, then the head atom is as well. A set of definite rules P has a unique smallest set of atoms closed under P. We call it the *deductive closure* and denote it by $\text{cl}(P)$. The uniqueness is implied by the fact that definite 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. Note that the closure can be constructed iteratively by starting from the empty set of atoms and iterating over the set of rules and updating the set of atoms with the head of a rule not yet satisfied until no unsatisfied rules are left.

Example 1 *Consider a definite program P*

$$\begin{array}{ll} p \leftarrow & t \leftarrow r, s \\ q \leftarrow p & s \leftarrow s \\ r \leftarrow p, q & \end{array}$$

Now $\text{cl}(P) = \{p, q, r\}$ which is the least set of atoms closed under the rules. The closure can be constructed in a forward chaining manner by starting from the empty set and including the head of a rule to the set if the body is already contained in the set. In this way $\text{cl}(P)$ is constructed by including first p and then q followed by r.

Definition 2.1 *Let P be a ground program. Then a set of ground atoms S is a* stable model *of P iff $S = \text{cl}(P^S)$.*

Example 2 *Program P*

$$
\begin{aligned}
p &\leftarrow \text{not } q, r \\
q &\leftarrow \text{not } p \\
r &\leftarrow \text{not } s \\
s &\leftarrow \text{not } p
\end{aligned}
$$

has a stable model $S = \{r, p\}$ *because* $S = \mathrm{cl}(P^S)$ *where the reduct* P^S *has two rules*

$$
\begin{aligned}
p &\leftarrow r \\
r &\leftarrow
\end{aligned}
$$

In addition to this model, P has another stable model $\{s, q\}$ *which can be verified similarly by constructing the reduct and its closure.*

2.2 Cardinality Constraints

Now we consider rules where all constraints C_i are *cardinality constraints* of the form

$$L \ \{a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m\} \ U \tag{9}$$

where L and U are two integers giving the *lower* and *upper bound* of the constraint, respectively. For a cardinality constraint C, we denote by $\mathrm{lit}(C)$ the corresponding set of literals $\{a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m\}$. The idea is that such a constraint is satisfied by a model for which the cardinality of the subset of the literals satisfied by the model is between the integers L and U. It is allowed to omit either of the bounds in which case a missing lower bound is to be taken as $-\infty$ and an upper bound as ∞. Hence, we can write a rule such as

$$\{a_1, a_2, a_3\} \ \leftarrow 1 \ \{\text{not } b_1, \text{not } b_2\} \ , 2 \ \{c_1, c_2, c_3, c_4\} \ 3$$

which says that if at least one of b_1, b_2 is missing from a stable model and at least 2 but at most 3 are included from $\{c_1, c_2, c_3, c_4\}$, then some subset of $\{a_1, a_2, a_3\}$ is contained in the model. Note that the empty set is also a possible choice for the subset.

Notice also that cardinality constraint rules can be seen as a generalization of normal rules which are a special case with cardinality constraints of the form $1\{l\}1$ where l is a literal. For example, a normal rule $a \leftarrow \text{not } b$ can be seen as a cardinality constraint rule $1\{a\}1 \leftarrow 1\{\text{not } b\}1$.

Our semantics for cardinality constraint rules is a generalization of the stable model semantics for normal logic programs and is given in terms of models that are sets of atoms. First we define when a model satisfies a rule and then using this concept the notion of stable models. Given a model S (a set of atoms) we use the notation $S \models a$ iff $a \in S$ and $S \models \text{not } a$ iff $a \notin S$.

Definition 2.2 *A set of atoms S* satisfies *a cardinality constraint C of the form (9) ($S \models C$) iff* $L \leq \mathrm{W}(C, S) \leq U$ *where*

$$\mathrm{W}(C, S) = |\{l \in \mathrm{lit}(C) : S \models l\}|$$

is the number of 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* $C_1, \ldots, C_n$.

We also allow *integrity constraints*, i.e., rules without the *head* constraint C_0, which are satisfied if at least one of the *body* constraints $C_1, \ldots, C_n$ is not.

Example 3 *Consider a rule*

$$1 \ \{a, b\} \ 1 \leftarrow 1 \ \{a, \text{not } b, \text{not } c\} \ 2$$

The constraint in the body is satisfied by the model $\{a, b\}$ *because*

$$\mathrm{W}(1\ \{a, \mathrm{not}\ b, \mathrm{not}\ c\}\ 2, \{a, b\}) = |\{a, \mathrm{not}\ c\}| = 2.$$

However, it is not satisfied by the model $\{a\}$. *Hence, the rule is satisfied by* $\{a\}$ *but it is not satisfied by* $\{a, b\}$ *because the constraint in the head is not satisfied by* $\{a, b\}$. *An integrity constraint*

$$\leftarrow 1\ \{a, b\}\ 1, 1\ \{a, \mathrm{not}\ b, \mathrm{not}\ c\}\ 2$$

is satisfied by models $\{a, b\}$ *and* $\{a\}$ *but not by a model* $\{a, c\}$. *The latter is the case as both constraints in the body are satisfied by* $\{a, c\}$.

A stable model of normal rules is a set of atoms that satisfies the rules and which is justified by the rules in the sense that it coincides with the closure of the reduct for the rules. This justification property is essential in many applications where solutions should not contain any unnecessary (unjustified) elements. When devising semantics for rules with cardinality (and weight) constraints the aim is to preserve this justifiability property by suitably generalizing the concepts of a *reduct* and its *deductive closure* introduced in the previous section for normal programs. For cardinality (and weight) rules, the reduct turns out to be a set of rules in a special form

$$H \leftarrow C_1, \ldots, C_n \tag{10}$$

where H is a ground 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. Recall that the closure can be constructed iteratively in a forward chaining manner by starting from the empty set and including the head of a rule to the set if the body is already contained in the set.

Example 4 *Consider a set of Horn rules* P

$$\begin{aligned}
&a \leftarrow 1\ \{a\}\\
&b \leftarrow\\
&c \leftarrow 2\ \{b, d\}\ , 1\ \{b, a\}\ \ .
\end{aligned}$$

The deductive closure of P *is the set of atoms* $\{b\}$ *which can be constructed iteratively by starting from the empty set and realizing that the (empty) body of the second rule is satisfied by the empty set and, hence,* b *should be added to the closure. This set is already closed under the rules. If a rule*

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

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

Next we introduce the reduct in two steps. First we define the reduct of a constraint and then generalize this to rules. The reduct C^S of a constraint C of the form (9) w.r.t. a set of atoms S is the constraint

$$L'\ \{a_1, \ldots, a_n\} \tag{11}$$

where $L' = L - |\{\mathrm{not}\ b \in \mathrm{lit}(C) : S \models \mathrm{not}\ b\}|$. Hence, in the reduct all negative literals and the upper bound are removed and the lower bound is decreased by the number of negative literals satisfied by S to account for the contribution of the negative literals towards satisfying the lower bound. For example, for a set $S = \{q\}$ and a constraint C

$$3\ \{\mathrm{not}\ q, \mathrm{not}\ r, p\}\ 4$$

the reduct C^S is

$$2 \{p\}$$

as only one of the negative literals in the constraint is satisfied by S.

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 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 2.3 *Let P be a ground program and S a set of ground atoms. The reduct P^S of P w.r.t. S is defined by*

$$\begin{aligned} P^S \;=\; & \{p \leftarrow C_1^S, \ldots, C_n^S : C_0 \leftarrow C_1, \ldots, C_n \in P \textit{ with } p \in \mathrm{lit}(C_0) \cap S \\ & \textit{and for all } i = 1, \ldots, n, \textit{ for the constraint } C_i \textit{ of the form} \\ & L \{a_1, \ldots, \mathrm{not}\; b_m\}\; U \;, \mathrm{W}(C_i, S) \leq U\} \end{aligned}$$

Now the idea is to follow the case of normal programs and define a stable model of a program P as an atom set S that *satisfies* all rules of P and that is the *deductive closure* of a *reduct* of P w.r.t. S. The role of the reduct is to provide the possible justifications for the atoms in S. Each atom in a stable model is justified by the program P in the sense that it is derivable from the reduct.

Definition 2.4 *A set of ground atoms S is a* stable model *of a program P iff the following two conditions hold:*
(i) $S \models P$,
(ii) $S = \mathrm{cl}(P^S)$.

Example 5 *Consider a program P*

$$1 \{a_1, a_2, a_3\} \; 1 \leftarrow$$

Observe that a stable model of a program P must be a subset of the atoms appearing in the heads of the rules in the program. This is because for any other atom the reduct P^S would not contain a Horn rule with this atom as its head and, hence, the atom could not be in the deductive closure of the reduct and thus not in a stable model. Hence, a stable model of P must be a subset of $\{a_1, a_2, a_3\}$.

The empty set is not a stable model because $\emptyset \not\models P$ and similarly for every subset having more than one of the atoms. However, $\{a_1\}$ is a stable model of P because it satisfies the rule and the reduct of P, $P^{\{a_1\}} = \{a_1 \leftarrow\}$ has $\{a_1\}$ as its closure. In fact, P has three stable models $\{a_1\}$, $\{a_2\}$, and $\{a_3\}$ as one would expect.

Notice that for

$$1 \{a_1, a_2, a_3\} \; 1 \leftarrow 1 \{a_1, b\} \; 2$$

the only stable model is $\emptyset$. For instance, $\{a_1\}$ is not a stable model because the reduct is $\{a_1 \leftarrow 1 \{a_1, b\} \}$ and its deductive closure is $\emptyset$.

A program

$$\{a_1, a_2, a_3\} \; \leftarrow$$

has all eight subsets of $\{a_1, a_2, a_3\}$ as its stable models. This demonstrates that stable models are not necessarily subset minimal.

2.3 Weight Constraints

Now we turn to extending the stable model semantics to handle weight constraints. The generalization can be explained by considering the following connection between cardinality constraints and linear inequalities. A cardinality constraint

$$L\ \{a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m\}\ U$$

can be seen as a linear inequality

$$L \leq a_1 + \cdots + a_n + \overline{b_1} + \cdots + \overline{b_m} \leq U$$

where $a_i, \overline{b_j}$ are variables with values 0 or 1 such that $x + \overline{x} = 1$ for every variable x. This can be generalized by allowing an arbitrary coefficient w_v for each variable v leading to linear inequalities of the form

$$\begin{array}{lll} L & \leq & w_{a_1} \times a_1 + \cdots + w_{a_n} \times a_n + \\ & & w_{b_1} \times \overline{b_1} + \cdots + w_{b_m} \times \overline{b_m} \quad \leq \quad U \end{array} \tag{12}$$

where a coefficient for a variable can be seen as the weight for the corresponding atom. Hence, we are considering constraints of the form

$$\begin{array}{lll} L & \leq & \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \\ & & \text{not } b_1 = w_{b_1}, \ldots, \text{not } b_m = w_{b_m}\} \quad \leq \quad U \end{array} \tag{13}$$

where, e.g., w_{a_1} is a weight for the atom a_1. As before we denote by $\text{lit}(C)$ the set of literals in a weight constraint C. The numbers L and U give the *lower* and *upper bounds* of the constraint, respectively. By $\text{w}(C)$ we denote the *local weight function* which gives the weights of the literals in C, i.e., $\text{w}(C)(a_i) = w_{a_i}$ and $\text{w}(C)(\text{not } b_j) = w_{b_j}$. The weights and bounds can be arbitrary real numbers. However, our current implementation allows only integers. This restriction has been adopted in order to avoid complications arising from the finite precision of real number arithmetic in standard programming languages.

The stable model semantics presented for cardinality constraints can be generalized to weight constraints in a straightforward way. The basic idea is to generalize the notion of the weight $\text{W}(C, S)$ of a constraint C in a model S. For a cardinality constraint C the weight $\text{W}(C, S)$ is defined as the number of literals satisfied in S. This corresponds to the case where every literal has weight equal to one. The obvious generalization for the case where arbitrary weights are allowed is to take the sum of the weights of the literals satisfied in S, i.e., for a weight constraint C the weight $\text{W}(C, S)$ of C in S is given by

$$\text{W}(C, S) = \sum_{l \in \text{lit}(C) \text{ and } S \models l} \text{w}(C)(l)$$

Now satisfiability of a weight constraint and a weight rule is defined as for the cardinality constraints in Definition 2.2.

Example 6 *Consider a weight constraint C*

$$1.02 \leq \{a = 1.0, b = 0.02, \text{not } c = 0.04\} \leq 1.03$$

for which $\text{W}(C, \{a, b, c\}) = 1.0 + 0.02 = 1.02$ *and* $\text{W}(C, \{a\}) = 1.0 + 0.04 = 1.04$*. So* $\{a, b, c\} \models C$ *but* $\{a\} \not\models C$*.*

Given the notion of satisfiability of a rule it is clear what the deductive closure of Horn constraint rules of the form (10) is where H is a ground atom and each weight constraint C_i contains only positive literals and has only a lower bound condition.

Example 7 *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 $\{b, d, c\}$.

When the rules have no negative weights, stable models can be defined using a straightforward generalization of the reduct for cardinality constraints. However, when negative weights are allowed, it is more tricky to guarantee the justifiability of models. This is because negative weights and negative literals are closely related. It turns out that they can replace each other and that one is inessential when the other is available. In order to avoid unnecessary complications we assume that all weights are non-negative and that negative literals are allowed. First we present the semantics for weight constraint rules in this case. We then show how to transform a rule with negative weights to an equivalent rule with positive weights only.

Suppose weight rules have no negative weights. The notion of a reduct for cardinality constraints is now easy to extend to handle weight constraints: the reduct C^S of a weight constraint C is obtained by removing all negative literals and the upper bound as in (11) but with the lower bound defined as

$$L' = L - \sum_{\text{not } p \in \text{lit}(C) \text{ and } S \models \text{not } p} \text{w}(C)(\text{not } p)$$

to reflect the fact that arbitrary weights are allowed. A stable model of a set of weight rules with non-negative weights is defined as in Definition 2.4 using the notion of a reduct given in Definition 2.3.

Example 8 *Consider the program P*

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

As for cardinality constraints, the justifiability of models 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 same holds if $S = \{b\}$ because the reduct P^S is empty since the upper bound in the body is exceeded. However, $S = \{c\}$ is a stable model as $S \models P$ and $\text{cl}(P^S) = \{c\}$ where $P^S = \{c \leftarrow 0 \leq \{b = 4\}\}$. In fact, $\{c\}$ is the only stable model of P.

Weight constraints with negative weights can be transformed to constraints with non-negative weights by a simple linear algebraic manipulation which translates a constraint C

$$L \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \text{not } b_1 = w_{b_1}, \ldots, \text{not } b_m = w_{b_m}\} \leq U$$

to an equivalent form C' with non-negative weights

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

where each negative weight is complemented together with the corresponding literal and the sum of absolute values of all negative weights is added to the bounds.

The equivalence of C and C' can be shown using the linear inequality (12) for C. We can eliminate any negative weight w_{a_l} by adding

$$|w_{a_l}| \times (a_l + \overline{a_l}) = |w_{a_l}| \qquad (a_l + \overline{a_l} = 1)$$

to the inequality. This leaves the term $|w_{a_l}| \times \overline{a_l}$ in the middle corresponding to the weighted literal not $a_l = |w_{a_l}|$. Similarly, all negative weights w_{b_i} can be eliminated.

Example 9 *Consider the rule*

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

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

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

Let $S = \{a\}$. Then the reduct for the resulting rule is $\{a \leftarrow 4 \leq \{b = 1\}\}$ and its deductive closure is $\emptyset$. Hence, S is not a stable model of the rule. In fact, the rule has no stable models.

Notice that although weight programs extend, e.g., normal logic programs considerably, the computational complexity remains unaffected, i.e., stays in **NP**. This is due to the fact that given a candidate model S it can be checked in polynomial time whether S is a stable model of a given program P. This holds because (i) the reduct P^S can be constructed in polynomial time and (ii) its deductive closure $\text{cl}(P^S)$ can be computed iteratively in polynomial time.

Theorem 2.5 ([NSS99]) *The problem of deciding whether a ground weight program has a stable model is* **NP***-complete.*

2.4 Optimization Capabilities

For solving optimization problems the `Smodels` system provides constructs for minimizing or maximizing a function given as a linear sum of weights of literals. For example, a minimization statement

$$\textit{minimize } \{a_1 = w_1, \ldots, a_n = w_n, \text{not } a_{n+1} = w_{n+1}, \ldots, \text{not } a_m = w_m\}$$

specifies that the model given as a solution should be a stable model S of the program with the smallest value for the sum

$$\sum_{l \in L \text{ and } S \models l} \text{w}(C)(l)$$

where $L = \{a_1, \ldots, a_n, \text{not } a_{n+1}, \ldots, \text{not } a_m\}$.

Example 10 *Consider the vertex cover problem but with an optimization task, i.e., we would like to find a vertex cover of minimum size. In order to do this we can take the rules (2) but replace the rule (3) by an optimization statement.*

$$\textit{minimize } \{v_1 = 1, \ldots, v_n = 1\}$$

where $\{v_1, \ldots, v_n\}$ is the set of vertices in the graph.

Even multiple minimization/maximization statements are allowed. Here the semantics is that the statements order models lexicographically according to the weights of the statements such that the first statement is the most significant.

Example 11 *Consider statements*

$$\begin{array}{l} minimize\ \{a_1 = 1\} \\ \vdots \\ minimize\ \{a_n = 1\} \end{array}$$

which order the models such that the solution is a model not containing a_1 if such a stable model exists and among those a model not containing a_2 if such a model exists and so on.

Notice that the semantics of multiple minimization statements and that of weak constraints in [BLR97] are closely related. On one hand, weak constraints can express lexicographic minimization principles such as that used in the previous example and, on the other hand, weight rules and minimization statements can be used for implementing weak constraints.

2.5 Rules with Variables

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

Hence, it would often be useful to employ rules with variables and functions. The semantics for rules with variables can be obtained by considering the ground instantiation of the program with respect to its Herbrand universe [NSS99]. However, even normal logic programs with the stable model semantics are highly undecidable when function symbols are allowed. Here we present a practical approach employed in the `Smodels` system to extending weight rules to allow variables and built-in functions in a limited way which guarantees decidability but which seems to be adequate for most practical applications. We start by introducing the basic subclass of rules that is supported by the `Smodels` system called *domain-restricted rules.* Then we discuss two useful extensions, conditional literals and built-in functions.

In a domain-restricted program P no proper function symbols are allowed. The program can be thought of as being divided into two parts: P_{Defs} that defines *domain predicates* and P_{Cs} that contains all other rules. The rules P_{Defs} for the domain predicates are non-recursive normal logic program rules and all the rules in P are domain-restricted in the sense that every variable in a rule must appear in a domain predicate which appears positively in the body of the rule. Notice that in domain-restricted programs it is possible to define new domain predicates from previous ones using unions, intersections, complements, joins, and projections of relations. This is demonstrated by the following example.

Example 12 *Consider a setting where a set R of ground facts of the form $r(x,y)$ is given. A predicate that appears in the head of a rule only for ground facts qualifies immediately as a domain predicate. Suppose we are to solve a 3-coloring problem for a graph which is defined by the relation r such that there is an edge (x,y) in the graph iff $r(x,y)$ holds in R but $r(y,x)$ does not. The problem can be formalized as a domain-restricted program P which includes the facts R and the rules*

$$col(c_1) \leftarrow;\ \ col(c_2) \leftarrow;\ \ col(c_3) \leftarrow \tag{14}$$
$$edge(X,Y) \leftarrow r(X,Y), \text{not}\ r(Y,X) \tag{15}$$
$$vertex(X) \leftarrow edge(X,Y) \tag{16}$$
$$vertex(Y) \leftarrow edge(X,Y) \tag{17}$$
$$1\ \{color(X,c_1), color(X,c_2), color(X,c_3)\}\ 1 \leftarrow vertex(X) \tag{18}$$
$$\leftarrow edge(X,Y), color(X,C), color(Y,C), col(C) \tag{19}$$

where a term beginning with an upper-case letter, like X,Y, is a variable and other terms, such as c_1, c_2, and c_3, are constants. For P, P_{Defs} includes the facts R and the rules (14–17) and the domain predicates are $r, col, edge$, and $vertex$. Observe, e.g., that the last rule 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.

For the `Smodels` system the subclass of domain-restricted programs has been chosen as a compromise. On one hand, it enables a simple and efficient two phase implementation technique where in the first (grounding) phase rules with variables are compiled to ground (variable-free) rules which are processed in the second phase by an efficient method for computing stable models for ground programs. On the other hand, non-recursive domain predicates allow a fair amount of modeling power for defining domains of variables.

Next we briefly explain why domain-restricted programs with variables can be compiled efficiently to a set of ground rules which is typically considerably smaller than the Herbrand instantiation of the program but still has exactly the same stable models. Then we introduce two useful extensions of the language, conditional literals and built-in functions. More details on implementation techniques can be found in Section 4. Efficient compilation is possible because of the following properties of domain-restricted programs: (i) The domain rules P_{Defs} have a unique stable model D which can be computed using database techniques as rules for defining domain predicates can be seen as view definitions. (ii) For the model D, it holds that P has the same stable models as P_D where P_D contains those ground instances of rules in P where each ground instance of a domain predicate is in D. (iii) Given D, P_D can be computed efficiently by processing one rule in P at a time.

An obvious extension would be to allow recursive domain predicates, i.e., programs where the part P_{Defs} is stratified. This means that, e.g., the transitive closure tc_r of a domain predicate r could be defined as a domain predicate by the rules

$$\begin{aligned} tc_r(X,Y) &\leftarrow r(X,Y) \\ tc_r(X,Y) &\leftarrow r(X,Z), tc_r(Z,Y) \end{aligned}$$

Notice that tc_r given using these rules could not serve as a further domain predicate for domain-restricted programs because the second rule is recursive. In order to transform the rule to a domain-restricted one, a new domain predicate for the variable Y needs to be added. A straightforward way to define such a domain predicate d is to use a rule $d(Y) \leftarrow r(X,Y)$. Applications where recursive stratified rules are used extensively could benefit if recursive domain predicates where allowed and corresponding techniques for evaluating recursive rules were incorporated in the grounding phase. This has been done, e.g., in the `dlv` system [L+99]. In the `Smodels` system the focus has not been on such deductive database applications and techniques for handling recursive rules in the grounding phase have not been included.

Conditional Literals

In order to be able to compactly write down sets of literals needed in constraint expressions we have introduced a notion of *conditional literals* of the form

$$l : d \tag{20}$$

where l is a literal and the conditional part d is a domain predicate. We also require that rules with conditional literals are *domain-restricted* in the sense that each variable in the rule appears in a domain predicate which is a positive body predicate or the conditional part of some conditional literal in the rule.

When using conditional literals we need to distinguish between *local* and *global* variables in a rule. The idea is that global variables quantify over the whole rule but the scope of a local variable is a single conditional literal. We do not introduce any notation to make the distinction explicit but use the following convention: a variable is local to a conditional literal if it appears only in this literal in the rule and all other variables are global to the rule. For example, in the rule

$$1\ \{color(X, C) : col(C)\}\ 1 \leftarrow vertex(X) \tag{21}$$

the variable C is local to the conditional literal in the head but X is a global variable.

The semantics of domain-restricted rules with conditional literals can be defined by regarding a rule with conditional literals as a shorthand for a set of ordinary ground rules in the following way. Consider a domain-restricted program P with conditional literals and its two parts: a set of non-recursive normal rules P_{Defs} defining the domain predicates and P_{Cs} containing all other rules. We denote by D the unique stable model of P_{Defs}. For a rule r with conditional literals, the corresponding set of ground rules contains every ground rule that can be obtained by the following two steps. First, all global variables in r are eliminated. This means that every global variable in the rule is substituted by some ground term such that the following condition holds: for each proper domain predicate in the body of the rule (which is not in a conditional literal), the resulting ground instance is included in D. For example, consider the rule (21). If $vertex(v) \in D$, then the rule

$$1\ \{color(v, C) : col(C)\}\ 1 \leftarrow vertex(v) \tag{22}$$

is a possible resulting rule after eliminating the global variable X. In fact, there is a possible resulting rule of the form (22) for every ground term v such that $vertex(v) \in D$.

In the second step we eliminate conditional literals as follows. Because the original rule is domain-restricted, elimination of the global variables leads to rules containing only local variables such that for each conditional literal $l : d$, every variable in it appears also in d. Such a conditional literal is taken as a sequence of ground instances l' of the literal l such that $l' : d'$ is a ground instance of $l : d$ with $d' \in D$. For example, the conditional literal

$$color(v, C) : col(C)$$

corresponds to the sequence of literals

$$color(v, c_1), color(v, c_2), color(v, c_3)$$

if D contains the facts $col(c_1), col(c_2), col(c_3)$.

The stable models of a program P with conditional literals are defined to be the stable models of the ground program which consists of the union of the sets of corresponding ground rules for each rule in P.

Example 13 *Using conditional literals we can rewrite the rule (18) as (21) such that the available colors can be changed by giving the corresponding set of facts $col(c)$. For example, if the facts*

$$col(c_1) \leftarrow; \ldots ; col(c_5) \leftarrow$$

for five colors are given, then in the ground instantiation of the coloring program P there is a ground rule

$$1\ \{color(v, c_1), \ldots , color(v, c_5)\}\ 1 \leftarrow vertex(v)$$

for each term v for which $vertex(v)$ is in the stable model D of P_{Defs}.

Built-in Functions

It is straightforward to extend domain predicates with built-in functions, e.g., for arithmetic. Domain-restrictedness of the rules guarantees that built-in functions can be evaluated together with the domain predicates because for each variable in a built-in function, the domain over which the variable ranges is restricted by domain predicates and thus floundering problems are avoided.

The `Smodels` system includes such an extension where integer arithmetic is available as well as mechanisms for providing user-defined built-in functions. This allows rules such as

$$\begin{array}{c} \mathit{area}(C, W \times L) \leftarrow \mathit{width}(C, W), \mathit{length}(C, L) \\ 0 \leq \{\mathit{circuit}(C) : \mathit{area}(C, A) = A\} \leq 90 \leftarrow \end{array}$$

where *area* is a domain predicate defined using domain predicates *width* and *length* (giving the width and length of a circuit) and a built-in function '$\times$' for integer multiplication. The second rule specifies a choice of a subset of the circuits with the sum of areas at most 90. Our implementation also allows expressing weights by rules involving domain predicates as in the example.

3 Application Methodology

In this section we present a straightforward application methodology for rules with the stable model semantics which also works surprising well in practice. This approach is based on the following idea of dividing the program into two parts: a *generator* part and a *tester* part where the stable models of the generator corresponds to all possible candidate solutions for the problem and the tester then eliminates from the candidates all non-valid ones. Examples of a similar technique for applying disjunctive logic programs can be found, e.g., in [EGM97, ELM+98].

Using the choice constructs expressible by cardinality and weight constraints it is straightforward to devise generators. For the tester, integrity constraints provide a powerful and simple-to-use technique to prune unwanted stable models because integrity constraints cannot introduce new stable models but can only eliminate them.

Proposition 3.1 *Let P be a program and IC a set of integrity constraints. Then if S is a stable model of $P \cup IC$, then S is a stable model of P.*

Next we demonstrate this basic application methodology using two examples.

3.1 Propositional Satisfiability

Consider first the satisfiability problem for a set S of clauses, i.e., a set of formulae of the form

$$a_1 \vee \cdots \vee a_n \vee \neg b_1 \vee \cdots \vee \neg b_m \tag{23}$$

The idea is to construct a program P such that models of S and stable models of P coincide. The generator is now easy to write because potential solutions are truth assignments to the set of atoms $\{a_1, \ldots, a_l\}$ appearing in S which correspond to subsets of $\{a_1, \ldots, a_l\}$ (giving the atoms assigned true). A program having every subset of $\{a_1, \ldots, a_l\}$ as its stable models is easy to write as a single rule

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

or as a rule $\{a_j\} \leftarrow$ for each atom a_j. The tester is also straightforward as it only needs to specify for each clause (23) the condition when the clause is not satisfied by a truth assignment. This can be done by an integrity constraint

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

Example 14 *Consider the set of clauses S and the corresponding program P:*

$$\begin{array}{rlrl} S: & a \vee \neg b & P: & \leftarrow \text{not } a, b \\ & \neg a \vee b & & \leftarrow a, \text{not } b \\ & & & \{a, b\} \leftarrow \end{array}$$

P has two stable models $\{\}$ and $\{a, b\}$ which are also the models of S.

As a more involved example of constructing a tester part of a problem we consider satisfiability of general propositional formulae constructed from connectives $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$. Given a propositional formula ϕ, we construct a program $\mathrm{LP}(\phi)$ such that the models of ϕ and the stable models of the program correspond. A program $\mathrm{LP}(\phi)$ models the satisfiability of each subformula of ϕ and it is defined recursively as follows:

- $\mathrm{LP}(\neg\phi) = \{s_{\neg\phi} \leftarrow \mathrm{not}\ s_\phi\} \cup \mathrm{LP}(\phi)$
- $\mathrm{LP}(\phi \wedge \psi) = \{s_{\phi\wedge\psi} \leftarrow s_\phi, s_\psi\} \cup \mathrm{LP}(\phi) \cup \mathrm{LP}(\psi)$
- $\mathrm{LP}(\phi \vee \psi) = \{s_{\phi\vee\psi} \leftarrow 1\ \{s_\phi, s_\psi\}\ \} \cup \mathrm{LP}(\phi) \cup \mathrm{LP}(\psi)$
- $\mathrm{LP}(\phi \rightarrow \psi) = \{s_{\phi\rightarrow\psi} \leftarrow 1\ \{\mathrm{not}\ s_\phi, s_\psi\}\ \} \cup \mathrm{LP}(\phi) \cup \mathrm{LP}(\psi)$
- $\mathrm{LP}(\phi \leftrightarrow \psi) = \left\{ \begin{array}{l} s_{\phi\leftrightarrow\psi} \leftarrow s_\phi, s_\psi \\ s_{\phi\leftrightarrow\psi} \leftarrow \mathrm{not}\ s_\phi, \mathrm{not}\ s_\psi \end{array} \right\} \cup \mathrm{LP}(\phi) \cup \mathrm{LP}(\psi)$
- $\mathrm{LP}(\phi) = \{\{s_\phi\} \leftarrow\}$ if ϕ is an atom.

Here all atoms s_χ are new atoms representing the fact that the subformula χ is satisfiable. Now each stable model represents a possible truth assignment (true atoms χ are given by atoms s_χ in the stable model where χ is an atom in the propositional formula ϕ). In addition, each stable model contains information about the truth values of the subformulae in the truth assignment specified by the stable model. Given this extra information, it is easy to write a tester to exclude models where ϕ is false: a propositional formula ϕ is satisfiable iff $\{\leftarrow \mathrm{not}\ s_\phi\} \cup \mathrm{LP}(\phi)$ has a stable model.

Example 15 *Consider a formula $\phi = (a \vee b) \leftrightarrow \neg(\neg a \wedge b)$. Now*

$$\begin{array}{ll} \mathrm{LP}(\phi): & s_\phi \leftarrow s_{a\vee b}, s_{\neg(\neg a\wedge b)} \\ & s_\phi \leftarrow \mathrm{not}\ s_{a\vee b}, \mathrm{not}\ s_{\neg(\neg a\wedge b)} \\ & s_{a\vee b} \leftarrow 1\{s_a, s_b\} \\ & s_{\neg(\neg a\wedge b)} \leftarrow \mathrm{not}\ s_{\neg a\wedge b} \\ & s_{\neg a\wedge b} \leftarrow s_{\neg a}, s_b \\ & s_{\neg a} \leftarrow \mathrm{not}\ s_a \\ & \{s_a\} \leftarrow \\ & \{s_b\} \leftarrow \end{array}$$

3.2 Hamiltonian Cycles

As demonstrated above, rules can capture propositional satisfiability problems. However, there are efficient standard techniques for solving satisfiability problems at least when they are represented as clauses (in conjunctive normal form). Furthermore, a rule representation of a set of clauses does not seem to offer any essential advantages as far as the compactness of the representation is concerned. In this subsection we demonstrate that rules enable succinct representations by discussing another problem which is challenging for propositional logic.

The example that we consider is the Hamiltonian cycle problem for undirected graphs, i.e., the problem of finding a path in a graph that visits each vertex of the graph exactly once and returns to the starting vertex. Cardinality constraint rules enable a straightforward and direct encoding of this problem as shown below. Logic program encodings of the Hamiltonian cycle problem for directed graphs can be found, e.g., in [Lif99, MT99b, Nie99]. This problem seems to be challenging for propositional logic both conceptually and computationally. We illustrate this in Section 5 where an encoding of Hamiltonian cycles in propositional logic is given and compared experimentally to logic program encodings.

For our logic program encoding, we assume that the graph (V, E) is represented as a set of facts including $vertex(v)$ for each vertex $v \in V$ of the graph, and $edge(v, u)$ for every edge $\{u, v\} \in E$ in the graph. For simplicity suppose that only one of the facts $edge(v, u)$ and $edge(u, v)$ is included for an edge $\{u, v\} \in E$.

A program for Hamiltonian cycles can be constructed using the following ideas. A Hamiltonian cycle is represented by atoms of the form $hc(v, u)$ so that stable models and Hamiltonian cycles correspond. A fact $hc(v, u)$ in a stable model of the program says that the edge $\{u, v\}$ belongs to the corresponding Hamiltonian cycle. The generator part of the program selects edges in such a way that for each vertex v exactly two edges $\{v, u\}$ and $\{w, v\}$ (two facts $hc(v, u)$ and $hc(w, v)$) are present in any stable model. Hence, models represent cycles but there could be more than one separate cycle. To eliminate such models we devise a tester using two additional concepts: we take some arbitrary vertex w as the initial vertex (the fact $initialvertex(w)$ is added) and define a predicate $r(v)$ to capture the fact that vertex v is reachable through the chosen edges from the initial vertex. Now the tester says simply that all vertices should be reachable from the initial vertex. These considerations lead to the following program

$$
\begin{aligned}
&2\ \{hc(V,U) : edge(V,U),\ hc(W,V) : edge(W,V)\}\ 2 \leftarrow vertex(V)\\
&r(V) \leftarrow initialvertex(V)\\
&r(V) \leftarrow hc(V,U), edge(V,U), r(U)\\
&r(V) \leftarrow hc(U,V), edge(U,V), r(U)\\
&\leftarrow vertex(V), \mathrm{not}\ r(V)
\end{aligned}
$$

where the first rule provides the generator and the rest of the rules define the tester. Notice that conditional literals are used as a compact method to represent the set of literals needed in the generator rule and that the domain predicate *edge* is used to guarantee domain-restrictedness in the rules defining r.

4 Implementation Techniques

The `Smodels` system implements the stable model semantics for domain-restricted rules where cardinality and weight constraints as well as variables and built-in functions can be used. The basic function of `Smodels` is to compute stable models for a given program. This is done in two phases where the first phase, implemented in the program `lparse` [Syr99], compiles the logic program into a set of primitive ground rules for which in the second phase stable models are computed using a Davis-Putnam like procedure [Sim99a], implemented in the program `smodels` [Sim99b]. Hence, `lparse` functions as a front-end for `smodels` but any program that creates primitive rules and wants to compute stable models can also use `smodels` as a library. Below we describe the two phases briefly. Further details can be found in [Sim99a, NSS99].

First Phase

The first phase is implemented in two steps. During the first step the program is parsed and domain predicates are automatically detected. Then domain predicates are evaluate using databases techniques by building a stratification of the rules defining domain predicates and evaluating them one stratum at a time starting from the lowest strata. For a program P this gives the unique stable model D of the part P_{Defs} for domain predicate rules. After this the ground instances P_D of the rules can be computed one rule at a time by essentially evaluating relational joins of domain predicates because the program is domain-restricted. During this process also conditional literals are extended to corresponding sequences of ordinary literals and built-in functions are evaluated. The result of the first step is a set of ground rules without built-in functions or conditional literals.

In the second step of the first phase the ground rules are compiled into four types of ground primitive rule forms: basic, choice, constraint, and weight rules. These particular primitive forms were chosen as they can be easily and efficiently implemented, and they are the rules that are the input to the second phase, i.e., to the `smodels` procedure.

The basic rule is a normal logic program rule of the form

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

The choice rule

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

is a limited form of a cardinality constraint rule and the primitive constraint rule

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

is another. Finally, there is the primitive weight rule

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

We restrict ourselves to positive integer weights. A rule with negative weights can be translated into a rule with only positive weights as described in Section 2.3.

We will illustrate the translation from constraint and weight rules into primitive rules by way of an example. A complete description of the translation can be found in [NSS99].

Example 16 *We want to translate the rule*

$$2\,\{h_1, \ldots, h_k\}\,4 \leftarrow 1\,\{a_1, \ldots, a_n\}\,3, 2 \leq \{b_1 = w_1, \ldots, b_m = w_m\} \leq 4$$

into primitive rules. We begin by creating two rules per constraint encoding whether the lower bound is satisfied and whether the upper bound is not, respectively. The constraint $2\,\{h_1, \ldots, h_k\}\,4$ *is translated into the rules*

$$\begin{aligned} t_1 &\leftarrow 2\,\{h_1, \ldots, h_k\} \\ t_2 &\leftarrow 5\,\{h_1, \ldots, h_k\} \end{aligned}$$

where t_1 *and* t_2 *are some new atoms. In the same way the second constraint* $1\,\{a_1, \ldots, a_n\}\,3$ *is translated into the rules*

$$\begin{aligned} t_3 &\leftarrow 1\,\{a_1, \ldots, a_n\} \\ t_4 &\leftarrow 4\,\{a_1, \ldots, a_n\} \end{aligned}$$

and the weight constraint $2 \leq \{b_1 = w_1, \ldots, b_m = w_m\} \leq 4$ *is translated into*

$$\begin{aligned} t_5 &\leftarrow \{b_1 = w_1, \ldots, b_m = w_m\} \geq 2 \\ t_6 &\leftarrow \{b_1 = w_1, \ldots, b_m = w_m\} \geq 5\ . \end{aligned}$$

This lets us capture the truth value of the body of the weight rule with the rule

$$t_7 \leftarrow t_3, \text{not } t_4, t_5, \text{not } t_6\ .$$

Hence, we can encode the original rule as

$$\begin{aligned} \{h_1, \ldots, h_k\} &\leftarrow t_7 \\ &\leftarrow t_7, \text{not } t_1 \\ &\leftarrow t_7, t_2\ . \end{aligned}$$

It is of course important that the atoms $t_1, \ldots, t_7$ *are not used anywhere else in the program.*

Second Phase

In the second phase a Davis-Putnam like procedure computes stable models of a set of primitive rules. This procedure is a backtracking search procedure that finds the stable models of a program by assigning truth values to the atoms of the program. Moreover, it uses the properties of the stable model semantics to infer and propagate additional truth values. The propagation is based on the well-founded semantics of normal programs but extends it, e.g., by handling the new primitive

$$
\begin{array}{l}
inCycle(V,U) \lor outCycle(V,U) \leftarrow edge(V,U) \\
r(V) \leftarrow initialvertex(V) \\
r(V) \leftarrow edge(U,V), r(U), inCycle(U,V) \\
r(V) \leftarrow edge(V,U), r(U), inCycle(V,U) \\
\leftarrow vertex(V), \mathrm{not}\ r(V) \\
n(V,U) \leftarrow edge(V,U), inCycle(V,U) \\
n(V,U) \leftarrow edge(U,V), inCycle(U,V) \\
\leftarrow \quad vertex(V), arc(V,U_1), arc(V,U_2), arc(V,U_3), \\
\qquad n(V,U_1), n(V,U_2), n(V,U_3), U_1 \neq U_2, U_1 \neq U_3, U_2 \neq U_3 \\
twoNs(V) \leftarrow \quad vertex(V), arc(V,U_1), arc(V,U_2), \\
\qquad\qquad\qquad n(V,U_1), n(V,U_2), U_1 \neq U_2 \\
\leftarrow vertex(V), \mathrm{not}\ twoNs(V)
\end{array}
$$

Figure 1: Encoding of the Hamiltonian cycle problem for `dlv`

rules, i.e., primitive choice, constraint, and weight rules, and by back-propagation and lookahead techniques. The procedure employs a dynamic application-independent search heuristic where the idea is to assign truth values to the atoms in an order attempting to minimize the remaining search space. More details on the procedure can be found in [Sim99a].

Since the procedure is in effect traversing a binary search tree, the number of nodes in the search space is in the worst case on the order of 2^n, where n can be taken to be the number of atoms that appear in a constraint in a head of a rule or that appear as a negative literal in a recursive loop of the program. This means, for example, that if we want to find a satisfying assignment of a general propositional formula, then the search space of the encoding given in Section 3.1 is restricted to the atoms of the original formula.

5 Experiments

In this section we demonstrate the current level of performance of the `Smodels` system. We compare it with the `dlv` system, which is another advanced implementation of the stable model semantics, and with state-of-the-art propositional satisfiability checkers: two complete checkers `sato` [Zha97] and `satz` [LA97] together with `walksat` [SKC94] based on local search. These satisfiability checkers provide interesting reference systems because they have been employed as basic inference engines in advanced reasoning systems, e.g., `sato` has been used in `CCALC` [McC99] and the other two checkers in the planning system `Blackbox` [KS99].

For the tests we use random 3-SAT problems and Hamiltonian cycle problems. The 3-SAT problems are chosen such that the clause to atom ratio is $4.258+58.26a^{-5/3}$, where a is the number of atoms, as this particular ratio determines a region of hard satisfiability problems [CA96]. For the Hamiltonian cycle problem we use planar graphs created by the *plane* function found in the Stanford GraphBase [Knu93] which builds a planar graph by inserting a given number of random points on a plane and by constructing the Delaunay triangulation of the points.

The 3-SAT problems are encoded as logic programs using the method in Section 3.1. For `dlv` the generator part includes for each atom a in the set of clauses a rule $a \lor a' \leftarrow$ where a' is a new atom not used anywhere else in the program.

The Hamiltonian cycle problems are encoded for `Smodels` as described in Section 3.2 using cardinality constraints. Since `dlv` does not support cardinality constraints, we have modified the encoding and used the translation given in Figure 1. Here the idea is that a Hamiltonian cycle is represented by facts $inCycle(V,U)$. The first five rules guarantee that in a stable model every vertex is reachable from the initial vertex through the chosen edges $inCycle(V,U)$ in the model. The next three rules exclude stable models where a vertex has three neighbors in the cycle. Here we use the predicate $arc(V,U)$, which is the symmetric closure of $edge(V,U)$, in order to achieve a compact encoding. The last two rules exclude models where there is a vertex which does not have two neighbors.

It is challenging to devise a compact encoding of Hamiltonian cycles in propositional logic. We translate a Hamiltonian cycle problem into a set of clauses following [Pap95] using a quadratic number of atoms w.r.t. the number of vertices in the graph. The translation encodes a total order on the vertices of the graph in such a way 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:

$$\neg v_{i,k} \vee \neg v_{j,k+1 \bmod n} \qquad \{v_i, v_j\} \text{ is not an edge in the graph.}$$

For each problem size we generate ten sets of rules/clauses. Each set is randomly shuffled and run ten times. We shuffle the input since a particular ordering might help the algorithms to avoid backtracking, thereby giving a skewed picture of their behavior. That is, we try to lessen the impact of lucky choices. The durations of the tests are given in seconds and they represent the time to find a solution or the time to decide that there are no solutions. All tests were run under Linux 2.2.12 on 450 MHz Pentium III computers with 256 MB of memory.

We tested each system using standard settings, i.e., without any tuning, using the following versions: `Smodels` with `smodels` 2.25 and `lparse` 0.99.45, `dlv` release 1999-11-24, `sato` 3.2, `satz`41, and `walksat` 35. For `walksat` which is based on local search, some parameters must be chosen and we have used the settings suggested in [KS99]: '-best -noise 50 100 -cutoff 100000000'.

The results of the 3-SAT tests are shown in Figure 2 giving the maximum, average and minimum running times. We did not test `walksat` on 3-SAT problems because `walksat` is an incomplete procedure and not all 3-SAT test cases had a solution. However, all Hamiltonian cycle test cases had a solution and in this case `walksat` could be used. The results of the Hamiltonian cycles tests are given in Figure 3. Here the running times do not include the preprocessing time to translate a graph to a set of propositional clauses or ground rules. The clausal translation was done using a special purpose program but for the rule translation the `lparse` program was used. This means that the execution time for both `dlv` and `Smodels` was measured when given a ground program (produced by `lparse`) as input. In order to study the efficiency of the cardinality constraints, the `Smodels` system was run also using an encoding without cardinality constraints which is obtained from the `dlv` encoding (Figure 1) by replacing the first rule by

$$\{inCycle(V, U)\} \leftarrow edge(V, U) \ .$$

The results for this encoding are denoted by 'smodels-n' in Figure 3.

The `Smodels` system scales in the same way as the satisfiability checkers on the random 3-SAT problems, but it is on average forty to fifty times slower than the fastest checker. When it comes to the Hamiltonian cycle problems the situation is reversed. The `Smodels` system scales better and is much faster than the satisfiability checkers. The difference can be attributed to the more compact logic program encoding of the Hamiltonian cycle problem and to the search heuristic used in the `smodels` procedure.

6 Conclusions

The `Smodels` system has been extended with cardinality and weight constraints in order to enable more realistic applications. Such constructs are important, e.g., in product configuration, resource-bounded planning, and constraint satisfaction to represent different kinds of cardinality, cost and resource constraints. They often enable compact, natural and easy to maintain representations of

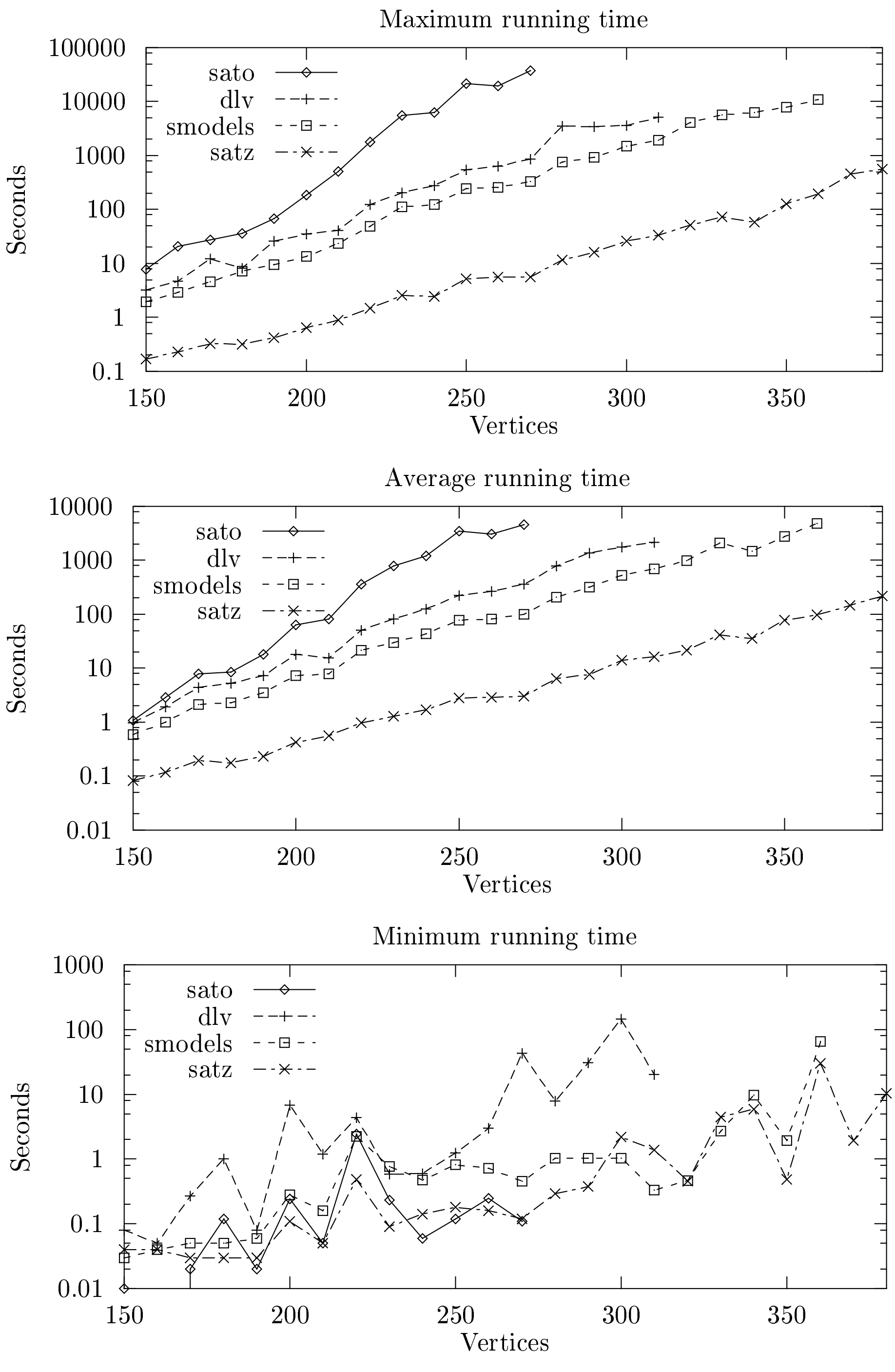

Figure 2: Experimental results on 3-SAT problems

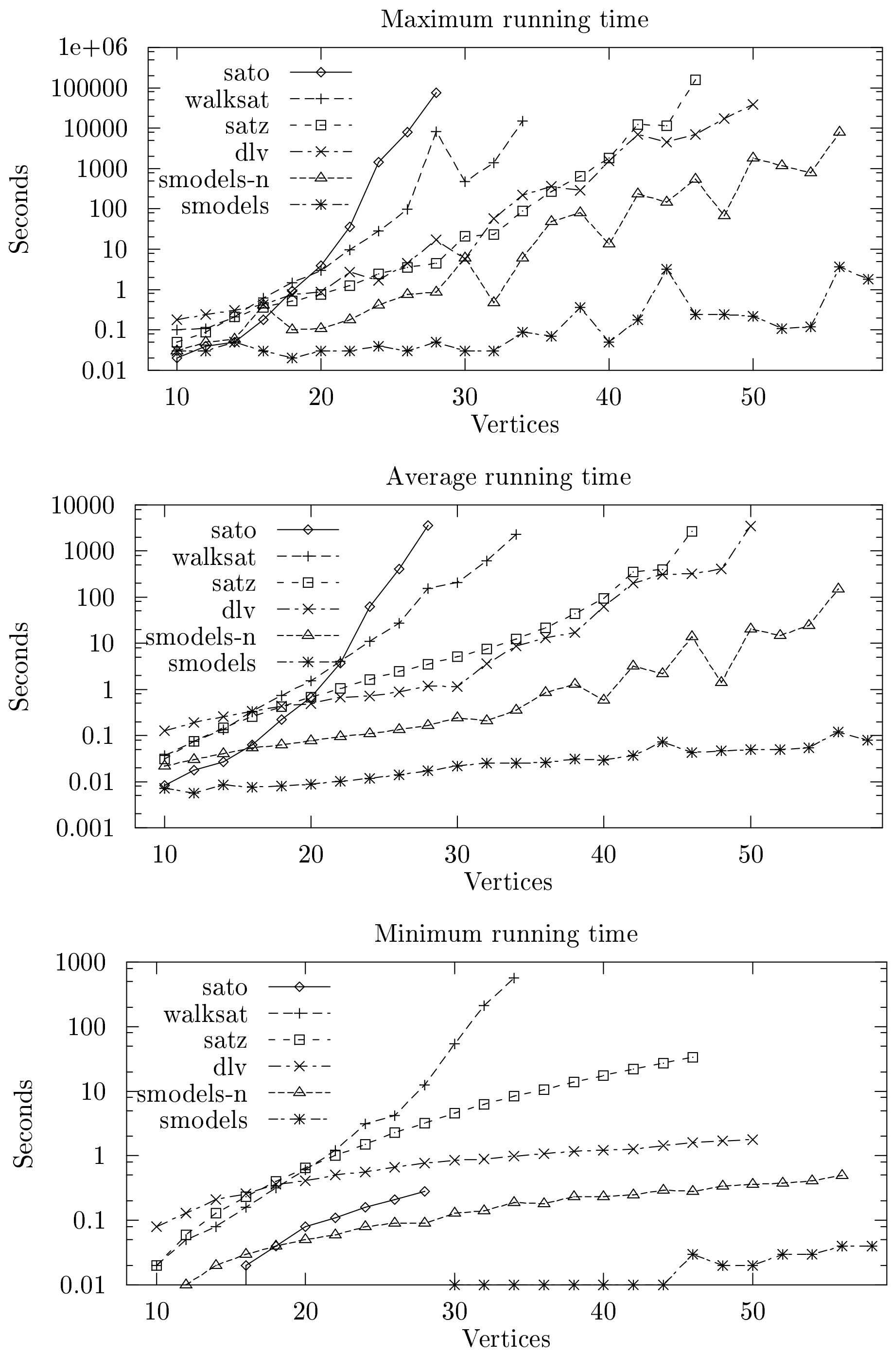

Figure 3: Experimental results on Hamiltonian cycle problems

involved problems. The extension has been done in such a way that basic computational complexity has not been affected, i.e., deciding whether a ground weight program has a stable model remains **NP**-complete as for normal programs. However, corresponding reasoning tasks for closely related semantics of disjunctive programs are typically complete problems on the second level of the polynomial hierarchy [EG95].

We have also developed a two phase implementation technique for the new constraints. In the first phase, rules with variables and built-in functions are compiled to ground weight rules and then to a set of primitive rules and in the second phase the answers, the stable models, are computed as the stable models of the primitive rules. Our implementation handles a subclass of rules, domain-restricted ones, which enable an efficient compilation method based on database techniques. The second phase is based on a Davis-Putnam like backtracking search procedure for which efficient pruning techniques and search heuristics have been developed. The comparisons with state-of-the-art satisfiability checkers show that our implementation techniques are quite competitive.

Acknowledgments

The support of the Academy of Finland (Project 43963) is gratefully acknowledged. The work of the second author has also been supported by the Helsinki Graduate School in Computer Science and Engineering. The authors would like to thank Tommi Syrjänen for developing and implementing the `lparse` procedure.

References

[BE98] Gerhard Brewka and Thomas Eiter. Preferred answer sets for extended logic programs. In A.G. Cohn, L. Schubert, and S.C. Shapiro, editors, *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning*, pages 86–97, Trento, Italy, June 1998. Morgan Kaufmann Publishers.

[BLR97] F. Buccafurri, N. Leone, and P. Rullo. Strong and weak constraints in disjunctive Datalog. In J. Dix, U. Furbach, and A. Nerode, editors, *Proceedings of the Fourth International Conference on Logic Programming and Non-Monotonic Reasoning*, pages 2–17, Dagstuhl Castle, Germany, July 1997. Springer-Verlag.

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

[CPSV99] Marco Cadoli, Luigi Palopoli, Andrea Schaerf, and Domenico Vasile. NP-SPEC: An executable specification language for solving all problems in NP. In G. Gupta, editor, *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 16–30, San Antonio, Texas, January 1999. Springer-Verlag.

[EG95] 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.

[EGM97] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, 1997.

[ELM+98] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarnello. The KR system dlv: Progress report, comparisons and benchmarks. In A.G. Cohn, L. Schubert, and S.C. Shapiro, editors, *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning*, pages 406–417, Trento, Italy, June 1998. Morgan Kaufmann Publishers.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Seattle, USA, August 1988. The MIT Press.

[GL90] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 579–597, Jerusalem, Israel, June 1990. The MIT Press.

[Gre99] S. Greco. Dynamic programming in Datalog with aggregates. *IEEE Transactions on Knowledge and Data Engineering*, 11(2):265–283, 1999.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In Michael J. O'Donnell, editor, *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, FRG, January 1987. ACM Press.

[Knu93] D.E. Knuth. The Stanford GraphBase, 1993. http://labrea.stanford.edu/pub/sgb/.

[KS99] H. Kautz and B. Selman. Unifying sat-based and graph-based planning. In T. Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 318–325, Stockholm, Sweden, July/August 1999. Morgan Kaufmann Publishers.

[L+99] N. Leone et al. Dlv, a disjunctive Datalog system. http://www.dbai.tuwien.ac.at/proj/dlv/, 1999.

[LA97] C.M. Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In G. Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, pages 341–355, Linz, Austria, October/November 1997. Springer-Verlag.

[Lif99] V. Lifschitz. Answer set planning. In D. De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming*, pages 25–37, Las Cruces, New Mexico, December 1999. The MIT Press.

[LNS96] James Lu, Anil Nerode, and V.S. Subrahmanian. Hybrid knowledge bases. *IEEE Trans. on Knowledge and Data Engineering*, 8(5):773–785, 1996.

[McC99] N. McCain. The causal calculator, 1999. http://www.cs.utexas.edu/users/mccain/cc/.

[MT99a] W. Marek and M. Truszczyński. Logic programming with costs. Manuscript which is available at http://www.cs.engr.uky.edu/~mirek/papers.html, 1999.

[MT99b] W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K.R. Apt, V.W. Marek, M. Truszczynski, and D.S. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999.

[Nie99] 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.

[NS94] Raymond Ng and V. Subrahmanian. Stable semantics for probabilistic deductive databases. *Information and Computation*, 110:42–83, 1994.

[NSS99] I. Niemelä, P. Simons, and T. Soininen. Stable model semantics of weight constraint rules. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Proceedings of the Fifth International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 317–331, El Paso, Texas, USA, December 1999. Springer-Verlag.

[Pap95] C.H. Papadimitriou. *Computational Complexity.* Addison-Wesley Publishing Company, 1995.

[SI94] 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.

[Sim99a] P. Simons. Extending the stable model semantics with more expressive rules. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Proceedings of the Fifth International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 305–316, El Paso, Texas, USA, December 1999. Springer-Verlag.

[Sim99b] P. Simons. smodels, a procedure for computing stable models of ground programs. http://www.tcs.hut.fi/Software/smodels/, 1999.

[SKC94] B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 337–343, Seattle, WA, July 1994. AAAI Press.

[SN99] T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In Gopal Gupta, editor, *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 305–319, San Antonio, Texas, January 1999. Springer-Verlag.

[Syr99] T. Syrjänen. lparse, a procedure for grounding domain-restricted logic programs. http://www.tcs.hut.fi/Software/smodels/lparse/, 1999.

[T$^+$99] M. Truszczynski et al. DeReS, a default reasoning system. http://www.cs.engr.uky.edu/ai/deres.html, 1999.

[Tsa93] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, 1993.

[W$^+$99] D.S. Warren et al. The XSB programming system. http://www.cs.sunysb.edu/~sbprolog/xsb-page.html, 1999.

[Z$^+$99] C. Zaniolo et al. LDL^{++}, a second-generation deductive database system. http://www.cs.ucla.edu/ldl/, 1999.

[Zha97] H. Zhang. SATO: An efficient propositional prover. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction*, pages 272–275, Townsville, North Queensland, Australia, July 1997. Springer-Verlag.

Index