Journal of Artificial Intelligence Research 27 (2006) 299-334 Submitted 01/06; published 11/06

Properties and Applications of Programs with Monotone and Convex Constraints

Lengning Liu LLIU1@CS.UKY.EDU
Mirosław Truszczyński MIREK@CS.UKY.EDU
Department of Computer Science, University of Kentucky,
Lexington, KY 40506-0046, USA

Abstract

We study properties of programs with *monotone* and *convex* constraints. We extend to these formalisms concepts and results from normal logic programming. They include the notions of strong and uniform equivalence with their characterizations, tight programs and Fages Lemma, program completion and loop formulas. Our results provide an abstract account of properties of some recent extensions of logic programming with aggregates, especially the formalism of *lparse* programs. They imply a method to compute stable models of *lparse* programs by means of off-the-shelf solvers of pseudo-boolean constraints, which is often much faster than the *smodels* system.

1. Introduction

We study programs with *monotone* constraints (Marek & Truszczyński, 2004; Marek, Niemelä, & Truszczyński, 2004, 2006) and introduce a related class of programs with *convex* constraints. These formalisms allow constraints to appear in the heads of program rules, which sets them apart from other recent proposals for integrating constraints into logic programs (Pelov, 2004; Pelov, Denecker, & Bruynooghe, 2004, 2006; Dell'Armi, Faber, Ielpa, Leone, & Pfeifer, 2003; Faber, Leone, & Pfeifer, 2004), and makes them suitable as an abstract basis for formalisms such as *lparse* programs (Simons, Niemelä, & Soininen, 2002).

We show that several results from normal logic programming generalize to programs with monotone constraints. We also discuss how these techniques and results can be extended further to the setting of programs with convex constraints. We then apply some of our general results to design and implement a method to compute stable models of *lparse* programs and show that it is often much more effective than *smodels* (Simons et al., 2002).

Normal logic programming with the semantics of stable models is an effective knowledge representation formalism, mostly due to its ability to express default assumptions (Baral, 2003; Gelfond & Leone, 2002). However, modeling numeric constraints on sets in normal logic programming is cumbersome, requires auxiliary atoms and leads to large programs hard to process efficiently. Since such constraints, often called *aggregates*, are ubiquitous, researchers proposed extensions of normal logic programming with explicit means to express aggregates, and generalized the stable-model semantics to the extended settings.

Aggregates imposing bounds on weights of sets of atoms and literals, called *weight* constraints, are especially common in practical applications and are included in all recent extensions of logic programs with aggregates. Typically, these extensions do not allow aggregates to appear in the heads of rules. A notable exception is the formalism of *programs with weight constraints* (Niemelä, Simons, & Soininen, 1999; Simons et al., 2002), which we refer to as *lparse* programs[1].

Lparse programs are logic programs whose rules have weight constraints in their heads and whose bodies are conjunctions of weight constraints. Normal logic programs can be viewed as a subclass of *lparse* programs and the semantics of *lparse* programs generalizes the stable-model semantics of normal logic programs (Gelfond & Lifschitz, 1988). *Lparse* programs are one of the most commonly used extensions of logic programming with weight constraints.

Since rules in *lparse* programs may have weight constraints as their heads, the concept of one-step provability is nondeterministic, which hides direct parallels between *lparse* and normal logic programs. An explicit connection emerged when Marek and Truszczyński (2004) and Marek et al. (2004, 2006) introduced *logic programs with monotone constraints*. These programs allow aggregates in the heads of rules and support nondeterministic computations. Marek and Truszczyński (2004) and Marek et al. (2004, 2006) proposed a generalization of the van Emden-Kowalski one-step provability operator to account for that nondeterminism, defined supported and stable models for programs with monotone constraints that mirror their normal logic programming counterparts, and showed encodings of *smodels* programs as programs with monotone constraints.

In this paper, we continue investigations of programs with monotone constraints. We show that the notions of uniform and strong equivalence of programs (Lifschitz, Pearce, & Valverde, 2001; Lin, 2002; Turner, 2003; Eiter & Fink, 2003) extend to programs with monotone constraints, and that their characterizations (Turner, 2003; Eiter & Fink, 2003) generalize, too.

We adapt to programs with monotone constraints the notion of a *tight* program (Erdem & Lifschitz, 2003) and generalize Fages Lemma (Fages, 1994).

We introduce extensions of propositional logic with monotone constraints. We define the completion of a monotone-constraint program with respect to this logic, and generalize the notion of a loop formula. We then prove the loop-formula characterization of stable models of programs with monotone constraints, extending to the setting of monotone-constraint programs results obtained for normal logic programs by Clark (1978) and Lin and Zhao (2002).

Programs with monotone constraints make explicit references to the default negation operator. We show that by allowing a more general class of constraints, called *convex*, default negation can be eliminated from the language. We argue that all results in our paper extend to programs with convex constraints.

Our paper shows that programs with monotone and convex constraints have a rich theory that closely follows that of normal logic programming. It implies that programs with monotone and convex constraints form an abstract generalization of extensions of normal logic programs. In particular, all results we obtain in the abstract setting of programs with monotone and convex constraints specialize to *lparse* programs and, in most cases, yield results that are new.

These results have practical implications. The properties of the program completion and loop formulas, when specialized to the class of *lparse* programs, yield a method to compute stable models of *lparse* programs by means of solvers of *pseudo-boolean* constraints, developed by the propositional satisfiability and integer programming communities (Eén & Sörensson, 2003; Aloul, Ramani, Markov, & Sakallah, 2002; Walser, 1997; Manquinho & Roussel, 2005; Liu & Truszczyński, 2003). We describe this method in detail and present experimental results on its performance. The results show that our method on problems we used for testing typically outperforms *smodels*.

1. Aggregates in the heads of rules have also been studied recently by Son and Pontelli (2006) and Son, Pontelli, and Tu (2006).

2. Preliminaries

We consider the propositional case only and assume a fixed set At of propositional atoms. It does not lead to loss of generality, as it is common to interpret programs with variables in terms of their propositional groundings.

The definitions and results we present in this section come from papers by Marek and Truszczyński (2004) and Marek et al. (2006). Some of them are more general as we allow constraints with infinite domains and programs with inconsistent constraints in the heads.

Constraints. A *constraint* is an expression $A = (X, C)$, where $X \subseteq At$ and $C \subseteq \mathcal{P}(X)$ ($\mathcal{P}(X)$ denotes the powerset of X). We call the set X the *domain* of the constraint $A = (X, C)$ and denote it by $Dom(A)$. Informally speaking, a constraint (X, C) describes a property of subsets of its domain, with C consisting precisely of these subsets of X that *satisfy* the constraint (have property) C.

In the paper, we identify truth assignments (interpretations) with the sets of atoms they assign the truth value *true*. That is, given an interpretation $M \subseteq At$, we have $M \models a$ if and only if $a \in M$. We say that an interpretation $M \subseteq At$ *satisfies* a constraint $A = (X, C)$ ($M \models A$), if $M \cap X \in C$. Otherwise, M does not satisfy A, ($M \not\models A$).

A constraint $A = (X, C)$ is *consistent* if there is M such that $M \models A$. Clearly, a constraint $A = (X, C)$ is consistent if and only if $C \neq \emptyset$.

We note that propositional atoms can be regarded as constraints. Let $a \in At$ and $M \subseteq At$. We define $C(a) = (\{a\}, \{\{a\}\})$. It is evident that $M \models C(a)$ if and only if $M \models a$. Therefore, in the paper we often write a as a shorthand for the constraint $C(a)$.

Constraint programs. Constraints are building blocks of rules and programs. Marek and Truszczyński (2004) defined *constraint programs* as sets of *constraint* rules

$$A \leftarrow A_1, \ldots, A_k, \mathbf{not}(A_{k+1}), \ldots, \mathbf{not}(A_m) \tag{1}$$

where A, $A_1, \ldots, A_n$ are constraints and $\mathbf{not}$ is the *default negation* operator.

In the context of constraint programs, we refer to constraints and negated constraints as *literals*. Given a rule r of the form (1), the constraint (literal) A is the *head* of r and the set $\{A_1, \ldots, A_k, \ldots, \mathbf{not}(A_{k+1}), \ldots, \mathbf{not}(A_m)\}$ of literals is the *body* of r[2]. We denote the head and the body of r by $hd(r)$ and $bd(r)$, respectively. We define the the *headset* of r, written $hset(r)$, as the domain of the head of r. That is, $hset(r) = Dom(hd(r))$.

For a constraint program P, we denote by $At(P)$ the set of atoms that appear in the domains of constraints in P. We define the *headset* of P, written $hset(P)$, as the union of the headsets of all rules in P.

Models. The concept of satisfiability extends in a standard way to literals $\mathbf{not}(A)$ ($M \models \mathbf{not}(A)$ if $M \not\models A$), to sets (conjunctions) of literals and, finally, to constraint programs.

***M*-applicable rules.** Let $M \subseteq At$ be an interpretation. A rule (1) is *M-applicable* if M satisfies every literal in $bd(r)$. We denote by $P(M)$ the set of all M-applicable rules in P.

Supported models. Supportedness is a property of models. Intuitively, every atom a in a supported model must have "reasons" for being "in". Such reasons are M-applicable rules whose heads contain a in their domains. Formally, let P be a constraint program and M a subset of $At(P)$. A model M of P is *supported* if $M \subseteq hset(P(M))$.

Examples. We illustrate the concept with examples. Let P be the constraint program that consists of the following two rules:

2. Sometimes we view the body of a rule as the *conjunction* of its literals.

$(\{c,d,e\}, \{\{c\}, \{d\}, \{e\}, \{c,d,e\}\}) \leftarrow$
$(\{a,b\}, \{\{a\}, \{b\}\}) \leftarrow (\{c,d\}, \{\{c\}, \{c,d\}\}), \mathbf{not}((\{e\}, \{\{e\}\}))$

A set $M = \{a, c\}$ is a model of P as M satisfies the heads of the two rules. Both rules in P are M-applicable. The first of them provides the support for c, the second one — for a. Thus, M is a supported model.

A set $M' = \{a, c, d, e\}$ is also a model of P. However, a has no support in P. Indeed, a only appears in the headset of the second rule. This rule is not M'-applicable and so, it does not support a. Therefore, M' is not a supported model of P. △

Nondeterministic one-step provability. Let P be a constraint program and M a set of atoms. A set M' is *nondeterministically one-step provable* from M by means of P, if $M' \subseteq hset(P(M))$ and $M' \models hd(r)$, for every rule r in $P(M)$.

The *nondeterministic one-step provability operator* T_P^{nd} for a program P is an operator on $\mathcal{P}(At)$ such that for every $M \subseteq At$, $T_P^{nd}(M)$ consists of all sets that are nondeterministically one-step provable from M by means of P.

The operator T_P^{nd} is *nondeterministic* as it assigns to each $M \subseteq At$ a *family* of subsets of At, each being a possible outcome of applying P to M. In general, T_P^{nd} is *partial*, since there may be sets M such that $T_P^{nd}(M) = \emptyset$ (no set can be derived from M by means of P). For instance, if $P(M)$ contains a rule r such that $hd(r)$ is inconsistent, then $T_P^{nd}(M) = \emptyset$.

Monotone constraints. A constraint (X, C) is *monotone* if C is closed under superset, that is, for every $W, Y \subseteq X$, if $W \in C$ and $W \subseteq Y$ then $Y \in C$.

Cardinality and weight constraints provide examples of monotone constraints. Let X be a *finite* set and let $C_k(X) = \{Y : Y \subseteq X,\ k \leq |Y|\}$, where k is a non-negative integer. Then $(X, C_k(X))$ is a constraint expressing the property that a subset of X has at least k elements. We call it a *lower-bound cardinality constraint* on X and denote it by kX.

A more general class of constraints are *weight constraints*. Let X be a finite set, say $X = \{x_1, \ldots, x_n\}$, and let $w, w_1, \ldots, w_n$ be non-negative reals. We interpret each w_i as the *weight* assigned to x_i. A *lower-bound weight constraint* is a constraint of the form (X, C_w), where C_w consists of those subsets of X whose total weight (the sum of weights of elements in the subset) is at least w. We write it as

$$w[x_1 = w_1, \ldots, x_n = w_n].$$

If all weights are equal to 1 and w is an integer, weight constraints become cardinality constraints. We also note that the constraint $C(a)$ is a cardinality constraint $1\{a\}$ and also a weight constraint $1[a = 1]$. Finally, we observe that lower-bound cardinality and weight constraints are monotone.

Cardinality and weight constraints (in a somewhat more general form) appear in the language of *lparse* programs (Simons et al., 2002), which we discuss later in the paper. The notation we adopted for these constraints in this paper follows the one proposed by Simons et al. (2002).

We use cardinality and weight constraints in some of our examples. They are also the focus of the last part of the paper, where we use our abstract results to design a new algorithm to compute models of *lparse* programs.

Monotone-constraint programs. We call constraint programs built of monotone constraints — *monotone-constraint programs* or *programs with monotone constraints*. That is, monotone-constraint programs consist of rules of rules of the form (1), where A, $A_1, \ldots, A_m$ are *monotone* constraints.

From now on, unless explicitly stated otherwise, programs we consider are monotone-constraint programs.

2.1 Horn Programs and Bottom-up Computations

Since we allow constraints with infinite domains and inconsistent constraints in heads of rules, the results given in this subsection are more general than their counterparts by Marek and Truszczyński (2004) and Marek et al. (2004, 2006). Thus, for the sake of completeness, we present them with proofs.

A rule (1) is *Horn* if $k = m$ (no occurrences of the negation operator in the body or, equivalently, only monotone constraints). A constraint program is *Horn* if every rule in the program is Horn.

With a Horn constraint program we associate *bottom-up* computations, generalizing the corresponding notion of a bottom-up computation for a normal Horn program.

Definition 1. *Let P be a Horn program. A P-*computation *is a (transfinite) sequence $\langle X_\alpha \rangle$ such that*

1. $X_0 = \emptyset$,
2. *for every ordinal number α, $X_\alpha \subseteq X_{\alpha+1}$ and $X_{\alpha+1} \in T_P^{nd}(X_\alpha)$,*
3. *for every* limit *ordinal α, $X_\alpha = \bigcup_{\beta<\alpha} X_\beta$.*

Let $t = \langle X_\alpha \rangle$ be a P-computation. Since for every $\beta < \beta'$, $X_\beta \subseteq X_{\beta'} \subseteq At$, there is a least ordinal number α_t such that $X_{\alpha_t+1} = X_{\alpha_t}$, in other words, a least ordinal when the P-computation stabilizes. We refer to α_t as the *length* of the P-computation t.

Examples. Here is a simple example showing that some programs have computations of length exceeding ω and so, the transfinite induction in the definition cannot be avoided. Let P be the program consisting of the following rules:

$(\{a_0\}, \{\{a_0\}\}) \leftarrow .$
$(\{a_i\}, \{\{a_i\}\}) \leftarrow (X_{i-1}, \{X_{i-1}\})$, for $i = 1, 2, \ldots$
$(\{a\}, \{\{a\}\}) \leftarrow (X_\infty, \{X_\infty\})$,

where $X_i = \{a_0, \ldots a_i\}$, $0 \leq i$, and $X_\infty = \{a_0, a_1, \ldots\}$. Since the body of the last rule contains a constraint with an infinite domain X_∞, it does not become applicable in any finite step of computation. However, it does become applicable in the step ω and so, $a \in X_{\omega+1}$. Consequently, $X_{\omega+1} \neq X_\omega$. △

For a P-computation $t = \langle X_\alpha \rangle$, we call $\bigcup_\alpha X_\alpha$ the *result* of the computation and denote it by R_t. Directly from the definitions, it follows that $R_t = X_{\alpha_t}$.

Proposition 1. *Let P be a Horn constraint program and t a P-computation. Then R_t is a supported model of P.*

Proof. Let $M = R_t$ be the result of a P-computation $t = \langle X_\alpha \rangle$. We need to show that: (1) M is a model of P; and (2) $M \subseteq hset(P(M))$.
(1) Let us consider a rule $r \in P$ such that $M \models bd(r)$. Since $M = R_t = X_{\alpha_t}$ (where α_t is the length of t), $X_{\alpha_t} \models bd(r)$. Thus, $X_{\alpha_t+1} \models hd(r)$. Since $M = X_{\alpha_t+1}$, M is a model of r and, consequently, of P, as well.

(2) We will prove by induction that, for every set X_α in the computation t, $X_\alpha \subseteq hset(P(M))$. The base case holds since $X_0 = \emptyset \subseteq hset(P(M))$.

If $\alpha = \beta + 1$, then $X_\alpha \in T_P^{nd}(X_\beta)$. It follows that $X_\alpha \subseteq hset(P(X_\beta))$. Since P is a Horn program and $X_\beta \subseteq M$, $hset(P(X_\beta)) \subseteq hset(P(M))$. Therefore, $X_\alpha \subseteq hset(P(M))$.

If α is a limit ordinal, then $X_\alpha = \bigcup_{\beta<\alpha} X_\beta$. By the induction hypothesis, for every $\beta < \alpha$, $X_\beta \subseteq hset(P(M))$. Thus, $X_\alpha \subseteq hset(P(M))$. By induction, $M \subseteq hset(P(M))$. □

Derivable models. We use computations to define *derivable* models of Horn constraint programs. A set M of atoms is a *derivable model* of a Horn constraint program P if for some P-computation t, we have $M = R_t$. By Proposition 1, derivable models of P are supported models of P and so, also models of P.

Derivable models are similar to the least model of a normal Horn program in that both can be derived from a program by means of a bottom-up computation. However, due to the nondeterminism of bottom-up computations of Horn constraint programs, derivable models are not in general unique nor minimal.

Examples. For example, let P be the following Horn constraint program:

$$P = \{1\{a, b\} \leftarrow\}$$

Then $\{a\}$, $\{b\}$ and $\{a, b\}$ are its derivable models. The derivable models $\{a\}$ and $\{b\}$ are minimal models of P. The third derivable model, $\{a, b\}$, is not a minimal model of P. △

Since inconsistent monotone constraints may appear in the heads of Horn rules, there are Horn programs P and sets $X \subseteq At$, such that $T_P^{nd}(X) = \emptyset$. Thus, some Horn constraint programs have no computations and no derivable models. However, if a Horn constraint program has models, the existence of computations and derivable models is guaranteed.

To see this, let M be a model of a Horn constraint program P. We define a *canonical computation* $t^{P,M} = \langle X_\alpha^{P,M} \rangle$ by specifying the choice of the next set in the computation in part (2) of Definition 1. Namely, for every ordinal β, we set

$$X_{\beta+1}^{P,M} = hset(P(X_\beta^{P,M})) \cap M.$$

That is, we include in $X_\alpha^{P,M}$ *all* those atoms occurring in the heads of $X_\beta^{P,M}$-applicable rules that belong to M. We denote the result of $t^{P,M}$ by $Can(P, M)$. Canonical computations are indeed P-computations.

Proposition 2. *Let P be a Horn constraint program. If $M \subseteq At$ is a model of P, the sequence $t^{P,M}$ is a P-computation.*

Proof. As P and M are fixed, to simplify the notation in the proof we will write X_α instead of $X_\alpha^{P,M}$.

To prove the assertion, it suffices to show that (1) $hset(P(X_\alpha)) \cap M \in T_P^{nd}(X_\alpha)$, and (2) $X_\alpha \subseteq hset(P(X_\alpha)) \cap M$, for every ordinal α.

(1) Let $X \subseteq M$ and $r \in P(X)$. Since all constraints in $bd(r)$ are monotone, and $X \models bd(r)$, $M \models bd(r)$, as well. From the fact that M is a model of P it follows now that $M \models hd(r)$. Consequently, $M \cap hset(P(X)) \models hd(r)$ for every $r \in P(X)$. Since $M \cap hset(P(X)) \subseteq hset(P(X))$,

$$M \cap hset(P(X)) \in T_P^{nd}(X).$$

Directly from the definition of the canonical computation for P and M we obtain that for every ordinal α, $X_\alpha \subseteq M$. Thus, (1), follows.
(2) We proceed by induction. The basis is evident as $X_0 = \emptyset$. Let us consider an ordinal $\alpha > 0$ and let us assume that (2) holds for every ordinal $\beta < \alpha$. If $\alpha = \beta + 1$, then $X_\alpha = X_{\beta+1} = hset(P(X_\beta)) \cap M$. Thus, by the induction hypothesis, $X_\beta \subseteq X_\alpha$. Since P is a Horn constraint program, it follows that $P(X_\beta) \subseteq P(X_\alpha)$. Thus

$$X_\alpha = X_{\beta+1} = hset(P(X_\beta)) \cap M \subseteq hset(P(X_\alpha)) \cap M.$$

If α is a limit ordinal then for every $\beta < \alpha$, $X_\beta \subseteq X_\alpha$ and, as before, also $P(X_\beta) \subseteq P(X_\alpha)$. Thus, by the induction hypothesis for every $\beta < \alpha$,

$$X_\beta \subseteq hset(P(X_\beta)) \cap M \subseteq hset(P(X_\alpha)) \cap M,$$

which implies that

$$X_\alpha = \bigcup_{\beta<\alpha} X_\beta \subseteq hset(P(X_\alpha)) \cap M.$$

□

Canonical computations have the following *fixpoint* property.

Proposition 3. *Let P be a Horn constraint program. For every model M of P, we have*

$$hset(P(Can(P, M))) \cap M = Can(P, M).$$

Proof. Let α be the length of the canonical computation $t^{P,M}$. Then, $X_{\alpha+1}^{P,M} = X_\alpha^{P,M} = Can(P, M)$. Since $X_{\alpha+1} = hset(X_\alpha) \cap M$, the assertion follows. □

We now gather properties of derivable models that extend properties of the least model of normal Horn logic programs.

Proposition 4. *Let P be a Horn constraint program. Then:*

1. *For every model M of P, $Can(P, M)$ is a greatest derivable model of P contained in M*

2. *A model M of P is a derivable model if and only if $M = Can(P, M)$*

3. *If M is a minimal model of P then M is a derivable model of P.*

Proof. (1) Let M' be a derivable model of P such that $M' \subseteq M$. Let $T = \langle X_\alpha \rangle$ be a P-derivation such that $M' = R_t$. We will prove that for every ordinal α, $X_\alpha \subseteq X_\alpha^{P,M}$. We proceed by transfinite induction. Since $X_0 = X_0^{P,M} = \emptyset$, the basis for the induction is evident. Let us consider an ordinal $\alpha > 0$ and assume that for every ordinal $\beta < \alpha$, $X_\beta \subseteq X_\beta^{P,M}$.

If $\alpha = \beta + 1$, then $X_\alpha \in T_P^{nd}(X_\beta)$ and so, $X_\alpha \subseteq hset(P(X_\beta))$. By the induction hypothesis and by the monotonicity of the constraints in the bodies of rules in P, $X_\alpha \subseteq hset(P(X_\beta^{P,M}))$. Thus, since $X_\alpha \subseteq R_t = M' \subseteq M$,

$$X_\alpha \subseteq hset(P(X_\beta^{P,M})) \cap M = X_{\beta+1}^{P,M} = X_\alpha^{P,M}.$$

The case when α is a limit ordinal is straightforward as $X_\alpha = \bigcup_{\beta<\alpha} X_\beta$ and $X_\alpha^{P,M} = \bigcup_{\beta<\alpha} X_\beta^{P,M}$.
(2) ($\Leftarrow$) If $M = Can(P, M)$, then M is the result of the canonical P-derivation for P and M. In particular, M is a derivable model of P.
($\Rightarrow$) if M is a derivable model of P, then M is also a model of P. From (1) it follows that $Can(P, M)$ is the greatest derivable model of P contained in M. Since M itself is derivable, $M = Can(P, M)$.
(3) From (1) it follows that $Can(P, M)$ is a derivable model of P and that $Can(P, M) \subseteq M$. Since M is a minimal model, $Can(P, M) = M$ and, by (2), M is a derivable model of P. □

2.2 Stable Models

In this section, we will recall and adapt to our setting the definition of stable models proposed and studied by Marek and Truszczyński (2004) and Marek et al. (2004, 2006) Let P be a monotone-constraint program and M a subset of $At(P)$. The *reduct* of P, denoted by P^M, is a program obtained from P by:

1. removing from P all rules whose body contains a literal $\mathbf{not}(B)$ such that $M \models B$;

2. removing literals $\mathbf{not}(B)$ for the bodies of the remaining rules.

The reduct of a monotone-constraint program is Horn since it contains no occurrences of default negation. Therefore, the following definition is sound.

Definition 2. *Let P be a monotone-constraint program. A set of atoms M is a* stable *model of P if M is a derivable model of P^M. We denote the set of stable models of P by $St(P)$.*

The definitions of the reduct and stable models follow and generalize those proposed for normal logic programs, since in the setting of Horn constraint programs, derivable models play the role of a least model.

As in normal logic programming and its standard extensions, stable models of monotone-constraint programs are supported models and, consequently, models.

Proposition 5. *Let P be a monotone-constraint program. If $M \subseteq At(P)$ is a stable model of P, then M is a supported model of P.*

Proof. Let M be a stable model of P. Then, M is a derivable model of P^M and, by Proposition 1, M is a supported model of P^M. It follows that M is a model of P^M. Directly from the definition of the reduct it follows that M is a model of P.

It also follows that $M \subseteq hset(P^M(M))$. For every rule r in $P^M(M)$, there is a rule r' in $P(M)$, which has the same head and the same non-negated literals in the body as r. Thus, $hset(P^M(M)) \subseteq hset(P(M))$ and, consequently, $M \subseteq hset(P(M))$. It follows that M is a supported model of P. □

Examples. Here is an example of stable models of a monotone-constraint program. Let P be a monotone-constraint program that contains the following rules:

$$2\{a, b, c\} \leftarrow 1\{a, d\}, \mathbf{not}(1\{c\})$$
$$1\{b, c, d\} \leftarrow 1\{a\}, \mathbf{not}(3\{a, b, d\}))$$
$$1\{a\} \leftarrow$$

Let $M = \{a, b\}$. Therefore, $M \not\models 1\{c\}$ and $M \not\models 3\{a, b, d\}$. Hence the reduct P^M contains the following three Horn rules:

$$2\{a, b, c\} \leftarrow 1\{a, d\}$$
$$1\{b, c, d\} \leftarrow 1\{a\}$$
$$1\{a\} \leftarrow$$

Since $M = \{a, b\}$ is a derivable model of P^M, M is a stable model of P.

Let $M' = \{a, b, c\}$. Then $M' \models 1\{c\}$ and $M \not\models 3\{a, b, d\}$. Therefore, the reduct $P^{M'}$ contains two Horn rules:

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

Since $M' = \{a, b, c\}$ is a derivable models of $P^{M'}$, M' is also a stable model of P. We note that stable models of a monotone-constraint program, in general, do not form an anti-chain. $\triangle$

If a normal logic program is Horn then its least model is its (only) stable model. Here we have an analogous situation.

Proposition 6. *Let P be a Horn monotone-constraint program. Then $M \subseteq At(P)$ is a derivable model of P if and only if M is a stable model of P.*

Proof. For every set M of atoms $P = P^M$. Thus, M is a derivable model of P if and only if it is a derivable model of P^M or, equivalently, a stable model of P. $\square$

In the next four sections of the paper we show that several fundamental results concerning normal logic programs extend to the class of monotone-constraint programs.

3. Strong and Uniform Equivalence of Monotone-constraint Programs

Strong equivalence and uniform equivalence concern the problem of replacing some rules in a logic program with others without changing the overall semantics of the program. More specifically, the strong equivalence concerns replacement of rules within *arbitrary* programs, and the uniform equivalence concerns replacements of all *non-fact* rules. In each case, the stipulation is that the resulting program must have the same stable models as the original one. Strong (and uniform) equivalence is an important concept due to its potential uses in program rewriting and optimization.

Strong and uniform equivalence have been studied in the literature mostly for normal logic programs (Lifschitz et al., 2001; Lin, 2002; Turner, 2003; Eiter & Fink, 2003).

Turner (2003) presented an elegant characterization of strong equivalence of *smodels* programs, and Eiter and Fink (2003) described a similar characterization of uniform equivalence of normal and disjunctive logic programs. We show that both characterizations can be adapted to the case of monotone-constraint programs. In fact, one can show that under the representations of normal logic programs as monotone-constraint programs (Marek et al., 2004, 2006) our definitions and characterizations of strong and uniform equivalence reduce to those introduced and developed originally for normal logic programs.

3.1 *M*-maximal Models

A key role in our approach is played by models of Horn constraint programs satisfying a certain maximality condition.

Definition 3. *Let P be a Horn constraint program and let M be a model of P. A set $N \subseteq M$ such that N is a model of P and $M \cap hset(P(N)) \subseteq N$ is an M-*maximal *model of P, written $N \models_M P$.*

Intuitively, N is an M-maximal model of P if N satisfies each rule $r \in P(N)$ "maximally" with respect to M. That is, for every $r \in P(N)$, N contains all atoms in M that belong to $hset(r)$ — the domain of the head of r.

To illustrate this notion, let us consider a Horn constraint program P consisting of a single rule:

$$1\{p, q, r\} \leftarrow 1\{s, t\}.$$

Let $M = \{p, q, s, t\}$ and $N = \{p, q, s\}$. One can verify that both M and N are models of P. Moreover, since the only rule in P is N-applicable, and $M \cap \{p, q, r\} \subseteq N$, N is an M-maximal model of P. On the other hand, $N' = \{p, s\}$ is not M-maximal even though N' is a model of P and it is contained in M.

There are several similarities between properties of models of normal Horn programs and M-maximal models of Horn constraint programs. We state and prove here one of them that turns out to be especially relevant to our study of strong and uniform equivalence.

Proposition 7. *Let P be a Horn constraint program and let M be a model of P. Then M is an M-maximal model of P and $Can(P, M)$ is the least M-maximal model of P.*

Proof. The first claim follows directly from the definition. To prove the second one, we simplify the notation: we will write N for $Can(P, M)$ and X_α for $X_\alpha^{P,M}$.

We first show that N is an M-maximal model of P. Clearly, $N \subseteq M$. Moreover, by Proposition 3, $hset(P(N)) \cap M = N$. Thus, N is indeed an M-maximal model of P.

We now show N is the least M-maximal model of P.

Let N' be any M-maximal model of P. We will show by transfinite induction that $N \subseteq N'$. Since $X_0 = \emptyset$, the basis for the induction holds. Let us consider an ordinal $\alpha > 0$ and let us assume that $X_\beta \subseteq N'$, for every $\beta < \alpha$. To show $N \subseteq N'$, it is sufficient to show that $X_\alpha \subseteq N'$.

Let us assume that $\alpha = \beta + 1$ for some $\beta < \alpha$. Then, since $X_\beta \subseteq N'$ and P is a Horn constraint program, we have $P(X_\beta) \subseteq P(N')$. Consequently,

$$X_\alpha = X_{\beta+1} = hset(P(X_\beta)) \cap M \subseteq hset(P(N')) \cap M \subseteq N',$$

the last inclusion follows from the fact thatN' is an M-maximal model of P.

If α is a limit ordinal, then $X_\alpha = \bigcup_{\beta<\alpha} X_\beta$ and the inclusion $X_\alpha \subseteq N'$ follows directly from the induction hypothesis. □

3.2 Strong Equivalence and SE-models

Monotone-constraint programs P and Q are *strongly equivalent*, denoted by $P \equiv_s Q$, if for every monotone-constraint program R, $P \cup R$ and $Q \cup R$ have the same set of stable models.

To study the strong equivalence of monotone-constraint programs, we generalize the concept of an *SE-model* due to Turner (2003).

There are close connections between strong equivalence of normal logic programs and the logic here-and-there. The semantics of the logic here-and-there is given in terms of Kripke models with two words which, when rephrased in terms of pairs of interpretations (pairs of sets of propositional atoms), give rise to SE-models.

Definition 4. *Let P be a monotone-constraint program and let X, Y be sets of atoms. We say that (X, Y) is an* SE-model *of P if the following conditions hold: (1) $X \subseteq Y$; (2) $Y \models P$; and (3) $X \models_Y P^Y$. We denote by $SE(P)$ the set of all SE-models of P.*

Examples. To illustrate the notion of an SE-model of a monotone-constraint program, let P consist of the following two rules:

$$2\{p, q, r\} \leftarrow 1\{q, r\}, \mathbf{not}(3\{p, q, r\})\}$$
$$1\{p, s\} \leftarrow 1\{p, r\}, \mathbf{not}(2\{p, r\})$$

We observe that $M = \{p, q\}$ is a model of P. Let $N = \emptyset$. Then $N \subseteq M$ and $P^M(N)$ is empty. It follows that $M \cap hset(P^M(N)) = \emptyset \subseteq N$ and so, $N \models_M P^M$. Hence, (N, M) is an SE-models of P.

Next, let $N' = \{p\}$. It is clear that $N' \subseteq M$. Moreover, $P^M(N') = \{1\{p, s\} \leftarrow 1\{p, r\}\}$. Hence $M \cap hset(P^M(N')) = \{p\} \subseteq N'$ and so, $N' \models_M P^M$. That is, (N', M) is another SE-model of P. △

SE-models yield a simple characterization of strong equivalence of monotone-constraint programs. To state and prove it, we need several auxiliary results.

Lemma 1. *Let P be a monotone-constraint program and let M be a model of P. Then (M, M) and $(Can(P^M, M), M)$ are both SE-models of P.*

Proof. The requirements (1) and (2) of an SE-model hold for (M, M). Furthermore, since M is a model of P, $M \models P^M$. Finally, we also have $hset(P(M)) \cap M \subseteq M$. Thus, $M \models_M P^M$.

Similarly, the definition of a canonical computation and Proposition 1, imply the first two requirements of the definition of SE-models for $(Can(P^M, M), M)$. The third requirement follows from Proposition 7. □

Lemma 2. *Let P and Q be two monotone-constraint programs such that $SE(P) = SE(Q)$. Then $St(P) = St(Q)$.*

Proof. If $M \in St(P)$, then M is a model of P and, by Lemma 1, $(M, M) \in SE(P)$. Hence, $(M, M) \in SE(Q)$ and, in particular, $M \models Q$. By Lemma 1 again,

$$(Can(Q^M, M), M) \in SE(Q).$$

By the assumption,

$$(Can(Q^M, M), M) \in SE(P)$$

and so, $Can(Q^M, M) \models_M P^M$ or, in other terms, $Can(Q^M, M)$ is an M-maximal model of P^M. Since $M \in St(P)$, $M = Can(P^M, M)$. By Proposition 7, M is the least M-maximal model of P^M. Thus, $M \subseteq Can(Q^M, M)$. On the other hand, we have $Can(Q^M, M) \subseteq M$ and so, $M = Can(Q^M, M)$. It follows that M is a stable model of Q. The other inclusion can be proved in the same way. □

Lemma 3. *Let P and R be two monotone-constraint programs. Then $SE(P \cup R) = SE(P) \cap SE(R)$.*

Proof. The assertion follows from the following two simple observations. First, for every set Y of atoms, $Y \models (P \cup R)$ if and only if $Y \models P$ and $Y \models R$. Second, for every two sets X and Y of atoms, $X \models_Y (P \cup R)^Y$ if and only if $X \models_Y P^Y$ and $X \models_Y R^Y$. □

Lemma 4. *Let P, Q be two monotone-constraint programs. If $P \equiv_s Q$, then P and Q have the same models.*

Proof. Let M be a model of P. By r we denote a constraint rule $(M, \{M\}) \leftarrow$. Then, $M \in St(P \cup \{r\})$. Since P and Q are strongly equivalent, $M \in St(Q \cup \{r\})$. It follows that M is a model of $Q \cup \{r\}$ and so, also a model of Q. The converse inclusion can be proved in the same way. □

Theorem 1. *Let P and Q be monotone-constraint programs. Then $P \equiv_s Q$ if and only if $SE(P) = SE(Q)$.*

Proof. ($\Leftarrow$) Let R be an arbitrary monotone-constraint program. Lemma 3 implies that $SE(P \cup R) = SE(P) \cap SE(R)$ and $SE(Q \cup R) = SE(Q) \cap SE(R)$. Since $SE(P) = SE(Q)$, we have that $SE(P \cup R) = SE(Q \cup R)$. By Lemma 2, $P \cup R$ and $Q \cup R$ have the same stable models. Hence, $P \equiv_s Q$ holds.
($\Rightarrow$) Let us assume $SE(P) \setminus SE(Q) \neq \emptyset$ and let us consider $(X, Y) \in SE(P) \setminus SE(Q)$. It follows that $X \subseteq Y$ and $Y \models P$. By Lemma 4, $Y \models Q$. Since $(X, Y) \notin SE(Q)$, $X \not\models_Y Q^Y$. It follows that $X \not\models Q^Y$ or $hset(Q^Y(X)) \cap Y \not\subseteq X$. In the first case, there is a rule $r \in Q^Y(X)$ such that $X \not\models hd(r)$. Since $X \subseteq Y$ and Q^Y is a Horn constraint program, $r \in Q^Y(Y)$. Let us recall that $Y \models Q$ and so, we also have $Y \models Q^Y$. It follows that $Y \models hd(r)$. Since $hset(r) \subseteq hset(Q^Y(X))$, $Y \cap hset(Q^Y(X)) \models hd(r)$. Thus, $hset(Q^Y(X)) \cap Y \not\subseteq X$ (otherwise, by the monotonicity of $hd(r)$, we would have $X \models hd(r)$).

The same property holds in the second case. Thus, it follows that

$$(hset(Q^Y(X)) \cap Y) \setminus X \neq \emptyset.$$

We define

$$X' = (hset(Q^Y(X)) \cap Y) \setminus X.$$

Let R be a constraint program consisting of the following two rules:

$$(X, \{X\}) \leftarrow$$
$$(Y, \{Y\}) \leftarrow (X', \{X'\}).$$

Let us consider a program $Q_0 = Q \cup R$. Since $Y \models Q$ and $X \subseteq Y$, $Y \models Q_0$. Thus, $Y \models Q_0^Y$ and, in particular, $Can(Q_0^Y, Y)$ is well defined. Since $R \subseteq Q_0^Y$, $X \subseteq Can(Q_0^Y, Y)$. Thus, we have

$$hset(Q_0^Y(X)) \cap Y \subseteq hset(Q_0^Y(Can(Q_0^Y, Y))) \cap Y = Can(Q_0^Y, Y)$$

(the last equality follows from Proposition 3). We also have $Q \subseteq Q_0$ and so,

$$X' \subseteq hset(Q^Y(X)) \cap Y \subseteq hset(Q_0^Y(X)) \cap Y.$$

Thus, $X' \subseteq Can(Q_0^Y, Y)$. Consequently, by Proposition 3, $Y \subseteq Can(Q_0^Y, Y)$. Since $Can(Q_0^Y, Y) \subseteq Y$, $Y = Can(Q_0^Y, Y)$ and so, $Y \in St(Q_0)$.

Since P and Q are strongly equivalent, $Y \in St(P_0)$, where $P_0 = P \cup R$. Let us recall that $(X, Y) \in SE(P)$. By Proposition 7, $Can(P^Y, Y)$ is a least Y-maximal model of P^Y. Since X is a Y-maximal model of P (as $X \models_Y P^Y$), it follows that $Can(P^Y, Y) \subseteq X$. Since $X' \not\subseteq X$, $Can(P_0^Y, Y) \subseteq X$. Finally, since $X' \subseteq Y$, $Y \not\subseteq X$. Thus, $Y \neq Can(P_0^Y, Y)$, a contradiction.

It follows that $SE(P) \setminus SE(Q) = \emptyset$. By symmetry, $SE(Q) \setminus SE(P) = \emptyset$, too. Thus, $SE(P) = SE(Q)$. □

3.3 Uniform Equivalence and UE-models

Let D be a set of atoms. By r_D we denote a monotone-constraint rule

$$r_D = \ (D, \{D\}) \leftarrow .$$

Adding a rule r_D to a program forces all atoms in D to be true (independently of the program).

Monotone-constraint programs P and Q are *uniformly equivalent*, denoted by $P \equiv_u Q$, if for every set of *atoms* D, $P \cup \{r_D\}$ and $Q \cup \{r_D\}$ have the same stable models.

An SE-model (X, Y) of a monotone-constraint program P is a *UE-model* of P if for every SE-model (X', Y) of P with $X \subseteq X'$, either $X = X'$ or $X' = Y$ holds. We write $UE(P)$ to denote the set of all UE-models of P. Our notion of a UE-model is a generalization of the notion of a UE-model due to Eiter and Fink (2003) to the setting of monotone-constraint programs.

Examples. Let us look again at the program we used to illustrate the concept of an SE-model. We showed there that $(\emptyset, \{p, q\})$ and $(\{p\}, \{p, q\})$ are SE-models of P. Directly from the definition of UE-models it follows that $(\{p\}, \{p, q\})$ is a UE-model of P. △

We will now present a characterization of uniform equivalence of monotone-constraint programs under the assumption that their sets of atoms are finite. One can prove a characterization of uniform equivalence of arbitrary monotone-constraint programs, generalizing one of the results by Eiter and Fink (2003). However, both the characterization and its proof are more complex and, for brevity, we restrict our attention to the finite case only.

We start with an auxiliary result, which allows us to focus only on atoms in $At(P)$ when deciding whether a pair (X, Y) of sets of atoms is an SE-model of a monotone-constraint program P.

Lemma 5. *Let P be a monotone-constraint program, $X \subseteq Y$ two sets of atoms. Then $(X, Y) \in SE(P)$ if and only if $(X \cap At(P), Y \cap At(P)) \in SE(P)$.*

Proof. Since $X \subseteq Y$ is given, and $X \subseteq Y$ implies $X \cap At(P) \subseteq Y \cap At(P)$, the first condition of the definition of an SE-model holds on both sides of the equivalence.

Next, we note that for every constraint C, $Y \models C$ if and only if $Y \cap Dom(C) \models C$. Therefore, $Y \models P$ if and only if $Y \cap At(P) \models P$. That is, the second condition of the definition of an SE-model holds for (X, Y) if and only if it holds for $(X \cap At(P), Y \cap At(P))$.

Finally, we observe that $P^Y = P^{Y \cap At(P)}$ and $P(X) = P(X \cap At(P))$. Therefore,

$$Y \cap hset(P^Y(X)) = Y \cap hset(P^{Y \cap At(P)}(X \cap At(P))).$$

Since $hset(P^{Y \cap At(P)}(X \cap At(P))) \subseteq At(P)$, it follows that

$$Y \cap hset(P^Y(X)) \subseteq X$$

if and only if

$$Y \cap At(P) \cap hset(P^{Y \cap At(P)}(X \cap At(P))) \subseteq X \cap At(P).$$

Thus, $X \models_Y P^Y$ if and only if $X \cap At(P) \models_{Y \cap At(P)} P^{Y \cap At(P)}$. That is, the third condition of the definition of an SE-model holds for (X, Y) if and only if it holds for $(X \cap At(P), Y \cap At(P))$. ☐

Lemma 6. *Let P be a monotone-constraint program such that $At(P)$ is finite. Then for every $(X, Y) \in SE(P)$ such that $X \neq Y$, the set*

$$\{X' \colon X \subseteq X' \subseteq Y,\ X' \neq Y,\ (X', Y) \in SE(P)\} \tag{2}$$

has a maximal element.

Proof. If $At(P) \cap X = At(P) \cap Y$, then for every element $y \in Y \setminus X$, $Y \setminus \{y\}$ is a maximal element of the set (2). Indeed, since $(X, Y) \in SE(P)$, by Lemma 5, $(X \cap At(P), Y \cap At(P)) \in SE(P)$. Since $X \cap At(P) = Y \cap At(P)$ and $y \notin At(P)$, $X \cap At(P) = (Y \setminus \{y\}) \cap At(P)$. Therefore, $((Y \setminus \{y\}) \cap At(P), Y \cap At(P)) \in SE(P)$. Then from Lemma 5 and the fact $Y \setminus \{y\} \subseteq Y$, we have $(Y \setminus \{y\}, Y) \in SE(P)$. Therefore, $Y \setminus \{y\}$ belongs to the set (2) and so, it is a maximal element of this set.

Thus, let us assume that $At(P) \cap X \neq At(P) \cap Y$. Let us define $X' = X \cup (Y \setminus At(P))$. Then $X \subseteq X' \subseteq Y$ and $X' \neq Y$. Moreover, no element in $X' \setminus X$ belongs to $At(P)$. That is, $X' \cap At(P) = X \cap At(P)$. Thus, by Lemma 5, $(X', Y) \in SE(P)$ and so, X' belongs to the set (2). Since $Y \setminus X' \subseteq At(P)$, by the finiteness of $At(P)$ it follows that the set (2) contains a maximal element containing X'. In particular, it contains a maximal element. ☐

Theorem 2. *Let P and Q be two monotone-constraint programs such that $At(P) \cup At(Q)$ is finite. Then $P \equiv_u Q$ if and only if $UE(P) = UE(Q)$.*

Proof. ($\Leftarrow$) Let D be an arbitrary set of atoms and Y be a stable model of $P \cup \{r_D\}$. Then Y is a model of $P \cup \{r_D\}$. In particular, Y is a model of P and so, $(Y, Y) \in UE(P)$. It follows that $(Y, Y) \in UE(Q)$, too. Thus, Y is a model of Q. Since Y is a model of r_D, $D \subseteq Y$. Consequently, Y is a model of $Q \cup \{r_D\}$ and thus, also of $(Q \cup \{r_D\})^Y$.

Let $X = Can((Q \cup \{r_D\})^Y, Y)$. Then $D \subseteq X \subseteq Y$ and, by Proposition 7, X is a Y-maximal model of $(Q \cup \{r_D\})^Y$. Consequently, X is a Y-maximal model of Q^Y. Since $X \subseteq Y$ and $Y \models Q$, $(X, Y) \in SE(Q)$.

Let us assume that $X \neq Y$. Then, by Lemma 6, there is a maximal set X' such that $X \subseteq X' \subseteq Y$, $X' \neq Y$ and $(X', Y) \in SE(Q)$. It follows that $(X', Y) \in UE(Q)$. Thus, $(X', Y) \in UE(P)$ and so, $X' \models_Y P^Y$. Since $D \subseteq X'$, $X' \models_Y (P \cup \{r_D\})^Y$. We recall that Y is a stable model of $P \cup \{r_D\}$. Thus, $Y = Can((P \cup \{r_D\})^Y, Y)$. By Proposition 7, $Y \subseteq X'$ and so we get $X' = Y$, a contradiction. It follows that $X = Y$ and, consequently, Y is a stable model of $Q \cup \{r_D\}$.

By symmetry, every stable model of $Q \cup \{r_D\}$ is also a stable model of $P \cup \{r_D\}$.

($\Rightarrow$) First, we note that $(Y, Y) \in UE(P)$ if and only if Y is a model of P. Next, we note that P and Q have the same models. Indeed, the argument used in the proof of Lemma 4 works also under the assumption that $P \equiv_u Q$. Thus, $(Y, Y) \in UE(P)$ if and only if $(Y, Y) \in UE(Q)$.

Now let us assume that $UE(P) \neq UE(Q)$. Let (X, Y) be an element of $(UE(P) \setminus UE(Q)) \cup (UE(Q) \setminus UE(P))$. Without loss of generality, we can assume that $(X, Y) \in UE(P) \setminus UE(Q)$. Since $(X, Y) \in UE(P)$, it follows that

1. $X \subseteq Y$
2. $Y \models P$ and, consequently, $Y \models Q$
3. $X \neq Y$ (otherwise, by our earlier observations, (X, Y) would belong to $UE(Q)$).

Let $R = (Q \cup \{r_X\})^Y$. Clearly, R is a Horn constraint program. Moreover, since $Y \models Q$ and $X \subseteq Y$, $Y \models R$. Thus, $Can(R, Y)$ is defined. We have $X \subseteq Can(R, Y) \subseteq Y$. We claim that $Can(R, Y) \neq Y$. Let us assume to the contrary that $Can(R, Y) = Y$. Then $Y \in St(Q \cup \{r_X\})$. Hence, $Y \in St(P \cup \{r_X\})$, that is, $Y = Can((P \cup \{r_X\})^Y, Y)$. By Proposition 7, Y is the least Y-maximal model of $(P \cup \{r_X\})^Y$ and X is a Y-maximal model of $(P \cup \{r_X\})^Y$ (since $(X, Y) \in SE(P)$, $X \models_Y P^Y$ and so, $X \models_Y (P \cup \{r_X\})^Y$, too). Consequently, $Y \subseteq X$ and, as $X \subseteq Y$, $X = Y$, a contradiction.

Thus, $Can(R, Y) \neq Y$. By Proposition 7, $Can(R, Y)$ is a Y-maximal model of R. Since $Q^Y \subseteq R$, it follows that $Can(R, Y)$ is a Y-maximal model of Q^Y and so, $(Can(R, Y), Y) \in SE(Q)$. Since $Can(R, Y) \neq Y$, from Lemma 6 it follows that there is a maximal set X' such that $Can(R, Y) \subseteq X' \subseteq Y$, $X' \neq Y$ and $(X', Y) \in SE(Q)$. By the definition, $(X', Y) \in UE(Q)$. Since $(X, Y) \notin UE(Q)$. $X \neq X'$. Consequently, since $X \subseteq X'$, $X' \neq Y$ and $(X, Y) \in UE(P)$, $(X', Y) \notin UE(P)$.

Thus, $(X', Y) \in UE(Q) \setminus UE(P)$. By applying now the same argument as above to (X', Y) we show the existence of X'' such that $X' \subseteq X'' \subseteq Y$, $X' \neq X''$, $X'' \neq Y$ and $(X'', Y) \in SE(P)$. Consequently, we have $X \subseteq X''$, $X \neq X''$ and $Y \neq X''$, which contradicts the fact that $(X, Y) \in UE(P)$. It follows then that $UE(P) = UE(Q)$. $\square$

Examples. Let $P = \{1\{p, q\} \leftarrow \mathbf{not}(2\{p, q\})\}$, and $Q = \{p \leftarrow \mathbf{not}(q),\ q \leftarrow \mathbf{not}(p)\}$. Then P and Q are strongly equivalent. We note that both programs have $\{p\}$, $\{q\}$, and $\{p, q\}$ as models. Furthermore, $(\{p\}, \{p\})$, $(\{q\}, \{q\})$, $(\{p\}, \{p, q\})$, $(\{q\}, \{p, q\})$, $(\{p, q\}, \{p, q\})$ and $(\emptyset, \{p, q\})$ are "all" SE-models of the two programs [3].

Thus, by Theorem 1, P and Q are strongly equivalent.

We also observe that the first five SE-models are precisely UE-models of P and Q. Therefore, by Theorem 2, P and Q are also uniformly equivalent.

It is possible for two monotone-constraint programs to be uniformly but not strongly equivalent. If we add rule $p \leftarrow$ to P, and rule $p \leftarrow q$ to Q, then the two resulting programs, say P' and Q', are uniformly equivalent. However, they are not strongly equivalent. The programs $P' \cup \{q \leftarrow p\}$ and $Q' \cup \{q \leftarrow p\}$ have different stable models. Another way to show it is by observing that $(\emptyset, \{p, q\})$ is an SE-model of Q' but not an SE-model of P'. $\triangle$

4. Fages Lemma

In general, supported models and stable models of a logic program (both in the normal case and the monotone-constraint case) do not coincide. Fages Lemma (Fages, 1994), later extended by Erdem and Lifschitz (2003), establishes a sufficient condition under which a supported model of a normal logic program is stable. In this section, we show that Fages Lemma extends to programs with monotone constraints.

3. From Lemma 5 and Theorem 1, it follows that only those SE-models that contain atoms only from $At(P) \cup At(Q)$ are the essential ones.

Definition 5. *A monotone-constraint program P is called* tight *on a set $M \subseteq At(P)$ of atoms, if there exists a mapping λ from M to ordinals such that for every rule $A \leftarrow A_1, \ldots, A_k, \mathbf{not}(A_{k+1}), \ldots, \mathbf{not}(A_m)$ in $P(M)$, if X is the domain of A and X_i the domain of A_i, $1 \leq i \leq k$, then for every $x \in M \cap X$ and for every $a \in M \cap \bigcup_{i=1}^{k} X_i$, $\lambda(a) < \lambda(x)$.*

We will now show that tightness provides a sufficient condition for a supported model to be stable. In order to prove a general result, we first establish it in the Horn case.

Lemma 7. *Let P be a Horn monotone-constraint program and let M be a supported model of P. If P is tight on M, then M is a stable model of P.*

Proof. Let M be an arbitrary supported model of P such that P is tight on M. Let λ be a mapping showing the tightness of P on M. We will show that for every ordinal α and for every atom $x \in M$ such that $\lambda(x) \leq \alpha$, $x \in Can(P, M)$. We will proceed by induction.

For the basis of the induction, let us consider an atom $x \in M$ such that $\lambda(x) = 0$. Since M is a supported model for P and $x \in M$, there exists a rule $r \in P(M)$ such that $x \in hset(r)$. Moreover, since P is tight on M, for every $A \in bd(r)$ and for every $y \in Dom(A) \cap M$, $\lambda(y) < \lambda(x) = 0$. Thus, for every $A \in bd(r)$, $Dom(A) \cap M = \emptyset$. Since $M \models bd(r)$ and since P is a Horn monotone-constraint program, it follows that $\emptyset \models bd(r)$. Consequently, $hset(r) \cap M \subseteq Can(P, M)$ and so, $x \in Can(P, M)$.

Let us assume that the assertion holds for every ordinal $\beta < \alpha$ and let us consider $x \in M$ such that $\lambda(x) = \alpha$. As before, since M is a supported model of P, there exists a rule $r \in P(M)$ such that $x \in hset(r)$. By the assumption, P is tight on M and, consequently, for every $A \in bd(r)$ and for every $y \in Dom(A) \cap M$, $\lambda(y) < \lambda(x) = \alpha$. By the induction hypothesis, for every $A \in bd(r)$, $Dom(A) \cap M \subseteq Can(P, M)$. Since P is a Horn monotone-constraint program, $Can(P, M) \models bd(r)$. By Proposition 3, $hset(r) \cap M \subseteq Can(P, M)$ and so, $x \in Can(P, M)$.

It follows that $M \subseteq Can(P, M)$. By the definition of a canonical computation, we have $Can(P, M) \subseteq M$. Thus, $M = Can(P, M)$. By Proposition 6, M is a stable model of P. ☐

Given this lemma, the general result follows easily.

Theorem 3. *Let P be a monotone-constraint program and let M be a supported model of P. If P is tight on M, then M is a stable model of P.*

Proof. One can check that if M is a supported model of P, then it is a supported model of the reduct P^M. Since P is tight on M, the reduct P^M is tight on M, too. Thus, M is a stable model of P^M (by Lemma 7) and, consequently, a derivable model of P^M (by Proposition 6). It follows that M is a stable model of P. ☐

5. Logic PL^{mc} and the Completion of a Monotone-constraint Program

The *completion* of a normal logic program (Clark, 1978) is a propositional theory whose models are precisely supported models of the program. Thus, supported models of normal logic programs can be computed by means of SAT solvers. Under some conditions, for instance, when the assumptions of Fages Lemma hold, supported models are stable. Thus, computing models of the completion can yield stable models, an idea implemented in the first version of *cmodels* software (Babovich & Lifschitz, 2002).

Our goal is to extend the concept of the completion to programs with monotone constraints. The completion, as we define it, retains much of the structure of monotone-constraint rules and allow us, in the restricted setting of *lparse* programs, to use pseudo-boolean constraint solvers to compute supported models of such programs. In this section we define the completion and prove a result relating supported models of programs to models of the completion. We discuss extensions of this result in the next section and their practical computational applications in Section 8.

To define the completion, we first introduce an extension of propositional logic with monotone constraints, a formalism we denote by PL^{mc}. A *formula* in the logic PL^{mc} is an expression built from monotone constraints by means of boolean connectives $\wedge$, $\vee$ (and their *infinitary* counterparts), $\rightarrow$ and $\neg$. The notion of a model of a constraint, which we discussed earlier, extends in a standard way to the class of formulas in the logic PL^{mc}.

For a set $L = \{A_1, \ldots, A_k, \mathbf{not}(A_{k+1}), \ldots, \mathbf{not}(A_m)\}$ of literals, we define

$$L^{\wedge} = A_1 \wedge \ldots \wedge A_k \wedge \neg A_{k+1} \wedge \ldots \wedge \neg A_m.$$

Let P be a monotone-constraint program. We form the *completion* of P, denoted $Comp(P)$, as follows:

1. For every rule $r \in P$ we include in $Comp(P)$ a PL^{mc} formula

$$[bd(r)]^{\wedge} \rightarrow hd(r)$$

2. For every atom $x \in At(P)$, we include in $Comp(P)$ a PL^{mc} formula

$$x \rightarrow \bigvee\{[bd(r)]^{\wedge} \colon r \in P, x \in hset(r)\}$$

(we note that when the set of rules in P is infinite, the disjunction may be infinitary).

The following theorem generalizes a fundamental result on the program completion from normal logic programming (Clark, 1978) to the case of programs with monotone constraints.

Theorem 4. *Let P be a monotone-constraint program. A set $M \subseteq At(P)$ is a supported model of P if and only if M is a model of $Comp(P)$.*

Proof. ($\Rightarrow$) Let us suppose that M is a supported model of P. Then M is a model of P, that is, for each rule $r \in P$, if $M \models bd(r)$ then $M \models hd(r)$. Since $M \models bd(r)$ if and only if $M \models [bd(r)]^{\wedge}$, it follows that all formulas in $Comp(P)$ of the first type are satisfied by M.

Moreover, since M is a supported model of P, $M \subseteq hset(P(M))$. That is, for every atom $x \in M$, there exists at least one rule r in P such that $x \in hset(r)$ and $M \models bd(r)$. Therefore, all formulas in $Comp(P)$ of the second type are satisfied by M, too.
($\Leftarrow$) Let us now suppose that M is a model of $Comp(P)$. Since $M \models bd(r)$ if and only if $M \models [bd(r)]^{\wedge}$, and since M satisfies formulas of the first type in $Comp(P)$, M is a model of P.

Let $x \in M$. Since M satisfies the formula $x \rightarrow \bigvee\{[bd(r)]^{\wedge} \colon r \in P, x \in hset(r)\}$, it follows that M satisfies $\bigvee\{[bd(r)]^{\wedge} \colon r \in P, x \in hset(r)\}$. That is, there is $r \in P$ such that M satisfies $[bd(r)]^{\wedge}$ (and so, $bd(r)$, too) and $x \in hset(r)$. Thus, $x \in hset(P(M))$. Hence, M is a supported model of P. $\square$

Theorems 3 and 4 have the following corollary.

Corollary 5. *Let P be a monotone-constraint program. A set $M \subseteq At(P)$ is a stable model of P if P is tight on M and M is a model of $Comp(P)$.*

We observe that for the material in this section it is not necessary to require that constraints appearing in the bodies of program rules be monotone. However, since we are only interested in this case, we adopted the monotonicity assumption here, as well.

6. Loops and Loop Formulas in Monotone-constraint Programs

The completion alone is not quite satisfactory as it relates *supported* not *stable* models of monotone-constraint programs with models of PL^{mc} theories. Loop formulas, proposed by Lin and Zhao (2002), provide a way to eliminate those supported models of normal logic programs, which are not stable. Thus, they allow us to use SAT solvers to compute stable models of *arbitrary* normal logic programs and not only those, for which supported and stable models coincide.

We will now extend this idea to monotone-constraint programs. In this section, we will restrict our considerations to programs P that are *finitary*, that is, $At(P)$ is finite. This restriction implies that monotone constraints that appear in finitary programs have finite domains.

Let P be a finitary monotone-constraint program. The *positive dependency graph* of P is the directed graph $G_P = (V, E)$, where $V = At(P)$ and $\langle u, v\rangle$ is an edge in E if there exists a rule $r \in P$ such that $u \in hset(r)$ and $v \in Dom(A)$ for some monotone constraint $A \in bd(r)$ (that is, A appears non-negated in $bd(r)$). We note that positive dependency graphs of finitary programs are finite.

Let $G = (V, E)$ be a directed graph. A set $L \subseteq V$ is a *loop* in G if the subgraph of G induced by L is strongly connected. A loop is *maximal* if it is not a proper subset of any other loop in G. Thus, maximal loops are vertex sets of strongly connected components of G. A maximal loop is *terminating* if there is no edge in G from L to any other maximal loop.

These concepts can be extended to the case of programs. By a *loop* (*maximal loop*, *terminating loop*) of a monotone-constraint program P, we mean the loop (maximal loop, terminating loop) of the positive dependency graph G_P of P. We observe that every finitary monotone-constraint program P has a terminating loop, since G_P is finite.

Let $X \subseteq At(P)$. By $G_P[X]$ we denote the subgraph of G_P *induced* by X. We observe that if $X \neq \emptyset$ then every loop of $G_P[X]$ is a loop of G_P.

Let P be a monotone-constraint program P. For every model M of P (in particular, for every model M of $Comp(P)$), we define $M^- = M \setminus Can(P^M, M)$. Since M is a model of P, M is a model of P^M. Thus, $Can(P^M, M)$ is well defined and so is M^-.

For every loop in the graph G_P we will now define the corresponding loop formula. First, for a constraint $A = (X, C)$ and a set $L \subseteq At$, we set $A_{|L} = (X, \{Y \in C \colon Y \cap L = \emptyset\})$ and call $A_{|L}$ the *restriction* of A to L. Next, let r be a monotone-constraint rule, say

$$r = \quad A \leftarrow A_1, \ldots, A_k, \mathbf{not}(A_{k+1}), \ldots, \mathbf{not}(A_m).$$

If $L \subseteq At$, then define a PL^{mc} formula $\beta_L(r)$ by setting

$$\beta_L(r) = A_{1|L} \wedge \ldots \wedge A_{k|L} \wedge \neg A_{k+1} \wedge \ldots \wedge \neg A_m.$$

Let L be a loop of a monotone-constraint program P. Then, the *loop formula* for L, denoted by $LP(L)$, is the PL^{mc} formula

$$LP(L) = \bigvee L \rightarrow \bigvee \{\beta_L(r) \colon r \in P \text{ and } L \cap hset(r) \neq \emptyset\}$$

(we recall that we use the convention to write a for the constraint $C(a) = (\{a\}, \{\{a\}\})$. A *loop completion* of a finitary monotone-constraint program P is the PL^{mc} theory

$$LComp(P) = Comp(P) \cup \{LP(L)\colon L \text{ is a loop in } G_P\}.$$

The following theorem exploits the concept of a loop formula to provide a necessary and sufficient condition for a model being a stable model. transfinite one.

Theorem 6. *Let P be a finitary monotone-constraint program. A set $M \subseteq At(P)$ is a stable model of P if and only if M is a model of $LComp(P)$.*

Proof. ($\Rightarrow$) Let M be a stable model of P. Then M is a supported model of P and, by Theorem 4, $M \models Comp(P)$.

Let L be a loop in P. If $M \cap L = \emptyset$ then $M \models LP(L)$. Thus, let us assume that $M \cap L \neq \emptyset$. Since M is a stable model of P, M is a derivable model of P^M, that is, $M = Can(P^M, M)$. Let $(X_n)_{n=0,1,\ldots}$ be the canonical P^M-derivation with respect to M (since we assume that P is finite and each constraint in P has a finite domain, P-derivations reach their results in finitely many steps). Since $Can(P^M, M) \cap L = M \cap L \neq \emptyset$, there is a smallest index n such that $X_n \cap L \neq \emptyset$. In particular, it follows that $n > 0$ (as $X_0 = \emptyset$) and $L \cap X_{n-1} = \emptyset$.

Since $X_n = hset(P^M(X_{n-1}) \cap M$ and $X_n \cap L \neq \emptyset$, there is a rule $r \in P^M(X_{n-1})$ such that $hset(r) \cap L \neq \emptyset$, that is, such that $L \cap hset(r)) \neq \emptyset$. Let r' be a rule in P, which contributes r to P^M. Then, for every literal $\mathbf{not}(A) \in bd(r')$, $M \models \mathbf{not}(A)$. Let $A \in bd(r')$. Then $A \in bd(r)$ and so, $X_{n-1} \models A$. Since $X_{n-1} \cap L = \emptyset$, $X_{n-1} \models A_{|L}$, too, By the monotonicity of $A_{|L}$, $M \models A_{|L}$. Thus, $M \models \beta_L(r')$. Since $hset(r') \cap L \neq \emptyset$, $L \cap hset(r)) \neq \emptyset$ and so, $M \models LP(L)$. Thus, $M \models LComp(P)$.

($\Leftarrow$) Let us consider a set $M \subseteq At(P)$ such that M is not a stable model of P. If M is not a supported model of P that $M \not\models Comp(P)$ and so M is not a model of $LComp(P)$. Thus, let us assume that M is a supported model of P. It follows that $M^- \neq \emptyset$. Let $L \subseteq M^-$ be a terminating loop for $G_P[M^-]$.

Let r' be an arbitrary rule in P such that $L \cap hset(r')) \neq \emptyset$, and let r be the rule obtained from r' by removing negated constraints from its body. Now, let us assume that $M \models \beta_{r'}(L)$. It follows that for every literal $\mathbf{not}(A) \in bd(r')$, $M \models \mathbf{not}(A)$. Thus, $r \in P^M$. Moreover, since L is a terminating loop for $G_P[M^-]$, for every constraint $A \in bd(r')$, $Dom(A) \cap M^- \subseteq L$. Since $M \models A_{|L}$, it follows that $Can(P^M, M) \models A$. Consequently, $hset(r') \cap L \subseteq hset(r') \cap M \subseteq Can(P^M, M)$ and so, $L \cap Can(P^M, M) \neq \emptyset$, a contradiction. Thus, $M \not\models \bigvee\{\beta_{r'}(L)\colon r' \in P \text{ and } L \cap hset(r')) \neq \emptyset\}$. Since $M \models \bigvee L$, it follows that $M \not\models LP(L)$ and so, $M \not\models LComp(P)$. □

The following result follows directly from the proof of Theorem 6 and provides us with a way to filter out specific non-stable supported models from $Comp(P)$.

Theorem 7. *Let P be a finitary monotone-constraint program and M a model of $Comp(P)$. If M^- is not empty, then M violates the loop formula of every terminating loop of $G_P[M^-]$.*

Finally, we point out that, Theorem 6 does not hold when a program P contains infinitely many rules. Here is a counterexample:

Examples. Let P be the set of following rules:

$1\{a_0\} \leftarrow 1\{a_1\}$
$1\{a_1\} \leftarrow 1\{a_2\}$
$\dots$
$1\{a_n\} \leftarrow 1\{a_{n+1}\}$
$\dots$

Let $M = \{a_0, \dots, a_n, \dots\}$. Then M is a supported model of P. The only stable model of P is $\emptyset$. However, $M^- = M \setminus \emptyset$ does not contain any terminating loop. The problem arises because there is an infinite simple path in $G_P[M^-]$. Therefore, $G_P[M^-]$ does not have a sink, yet it does not have a terminating loop either. $\triangle$

The results of this section, concerning the program completion and loop formulas — most importantly, the loop-completion theorem — form the basis of a new software system to compute stable models of *lparse* programs. We discuss this matter in Section 8.

7. Programs with Convex Constraints

We will now discuss programs with convex constraints, which are closely related to programs with monotone constraints. Programs with convex constraints are of interest as they do not involve explicit occurrences of the default negation operator $\mathbf{not}$, yet are as expressive as programs with monotone-constraints. Moreover, they directly subsume an essential fragment of the class of *lparse* programs (Simons et al., 2002).

A constraint (X, C) is *convex*, if for every $W, Y, Z \subseteq X$ such that $W \subseteq Y \subseteq Z$ and $W, Z \in C$, we have $Y \in C$. A constraint rule of the form (1) is a *convex-constraint rule* if A, $A_1, \dots, A_n$ are convex constraints and $m = k$. Similarly, a constraint program built of convex-constraint rules is a *convex-constraint program*.

The concept of a model discussed in Section 2 applies to convex-constraint programs. To define supported and stable models of convex-constraint programs, we view them as special programs with monotone-constraints.

To this end, we define the *upward* and *downward closures* of a constraint $A = (X, C)$ to be constraints $A^+ = (X, C^+)$ and $A^- = (X, C^-)$, respectively, where

$C^+ = \{Y \subseteq X\colon \text{ for some } W \in C, W \subseteq Y\}$, and
$C^- = \{Y \subseteq X\colon \text{ for some } W \in C, Y \subseteq W\}$.

We note that the constraint A^+ is monotone. We call a constraint (X, C) *antimonotone* if C is closed under subset, that is, for every $W, Y \subseteq X$, if $Y \in C$ and $W \subseteq Y$ then $W \in C$. It is clear that the constraint A^- is antimonotone.

The upward and downward closures allow us to represent any convex constraint as the "conjunction" of a monotone constraint and an antimonotone constraint.Namely, we have the following property of convex constraints.

Proposition 8. *A constraint (X, C) is convex if and only if $C = C^+ \cap C^-$.*

Proof. ($\Leftarrow$) Let us assume that $C = C^+ \cap C^-$and let us consider a set M such that $M' \subseteq M \subseteq M''$, where $M', M'' \in C$. it follows that $M' \in C^+$ and $M'' \in C^-$. Thus, $M \in C^+$ and $M \in C^-$. Consequently, $M \in C$, which implies that (X, C) is convex.

($\Rightarrow$) The definitions directly imply that $C \subseteq C^+$ and $C \subseteq C^-$. Thus, $C \subseteq C^+ \cap C^-$. Let us consider $M \in C^+ \cap C^-$. Then there are sets $M', M'' \in C$ such that $M' \subseteq M$ and $M \subseteq M''$. Since C is convex, $M \in C$. Thus, $C^+ \cap C^- \subseteq C$ and so, $C = C^+ \cap C^-$. $\square$

Proposition 8 suggests an encoding of convex-constraint programs as monotone-constraint programs. To present it, we need more notation. For a constraint $A = (X, C)$, we call the constraint $(X, \overline{C})$, where $\overline{C} = \mathcal{P}(X) \setminus C$, the *dual constraint* for A. We denote it by $\overline{A}$. It is a direct consequence of the definitions that a constraint A is monotone if and only if its dual $\overline{A}$ is antimonotone.

Let C be a convex constraint. We set $mc(C) = \{C\}$ if C is monotone. We set $mc(C) = \{\mathbf{not}(\overline{C})\}$, if C is antimonotone. We define $mc(C) = \{C^+, \mathbf{not}(\overline{C^-})\}$, if C is neither monotone nor antimonotone. Clearly, C and $mc(C)$ have the same models.

Let P be a convex-constraint program. By $mc(P)$ we denote the program with monotone constraints obtained by replacing every rule r in P with a rule r' such that

$$hd(r') = hd(r)^+ \text{ and } bd(r') = \bigcup\{mc(A)\colon A \in bd(r)\}$$

and, if $hd(r)$ is *not* monotone, also with an additional rule r'' such that

$$hd(r'') = (\emptyset, \emptyset) \text{ and } bd(r'') = \{\overline{hd(r)^-}\} \cup bd(r').$$

By our observation above, all constraints appearing in rules of $mc(P)$ are indeed monotone, that is, $mc(P)$ is a program with monotone constraints.

It follows from Proposition 8 that M is a model of P if and only if M is a model of $mc(P)$. We extend this correspondence to supported and stable models of a convex constraint program P and the monotone-constraint program $mc(P)$.

Definition 6. *Let P be a convex constraint program. Then a set of atoms M is a supported (or stable) model of P if M is a supported (or stable) model of $mc(P)$.*

With these definitions, monotone-constraint programs can be viewed (almost) directly as convex-constraint programs. Namely, we note that monotone and antimonotone constraints are convex. Next, we observe that if A is a monotone constraint, the expression $\mathbf{not}(A)$ has the same meaning as the antimonotone constraint $\overline{A}$ in the sense that for every interpretation M, $M \models \mathbf{not}(A)$ if and only if $M \models \overline{A}$.

Let P be a monotone-constraint program. By $cc(P)$ we denote the program obtained from P by replacing every rule r of the form (1) in P with r' such that

$$hd(r') = hd(r) \text{ and } bd(r') = \bigcup\{A_i\colon i = 1, \ldots, k\} \cup \bigcup\{\overline{A_j}\colon j = k+1, \ldots, m\}$$

One can show that programs P and $cc(P)$ have the same models, supported models and stable models. In fact, for every monotone-constraint program P we have $P = mc(cc(P))$.

Remark. Another consequence of our discussion is that the default negation operator can be eliminated from the syntax at the price of allowing antimonotone constraints and using antimonotone constraints as negated literals. $\square$

Due to the correspondences we have established above, one can extend to convex-constraint programs all concepts and results we discussed earlier in the context of monotone-constraint programs. In many cases, they can also be stated *directly* in the language of convex-constraints. The most important for us are the notions of the completion and loop formulas, as they lead to new algorithms for computing stable models of *lparse* programs. Therefore, we will now discuss them in some detail.

As we just mentioned, we could use $Comp(mc(P))$ as a definition of the completion $Comp(P)$ for a convex-constraint logic program P. Under this definition Theorems 9 extends to the case of convex-constraint programs. However, $Comp(mc(P))$ involves monotone constraints and their negations and *not* convex constraints that appear in P. Therefore, we will now propose another approach, which preserves convex constraints of P.

To this end, we first extend the logic PL^{mc} with convex constraints. In this extension, which we denote by PL^{cc} and refer to as the *propositional logic with convex-constraints*, formulas are boolean combinations of convex constraints. The semantics of such formulas is given by the notion of a model obtained by extending over boolean connectives the concept of a model of a convex constraint.

Thus, the only difference between the logic PL^{mc}, which we used to define the completion and loop completion for monotone-convex programs and the logic PL^{cc} is that the former uses monotone constraints as building blocks of formulas, whereas the latter is based on convex constraints. In fact, since monotone constraints are special convex constraints, the logic PL^{mc} is a fragment of the logic PL^{cc}.

Let P be a convex-constraint program. The completion of P, denoted by $Comp(P)$, is the following set of PL^{cc} formulas:

1. For every rule $r \in P$ we include in $Comp(P)$ a PL^{cc} formula

$$[bd(r)]^{\wedge} \rightarrow hd(r)$$

(as before, for a set of convex constraints L, $L^{\wedge}$ denotes the conjunction of the constraints in L)

2. For every atom $x \in At(P)$, we include in $Comp(P)$ a PL^{cc} formula

$$x \rightarrow \bigvee\{[bd(r)]^{\wedge}\colon r \in P,\ x \in hset(r)\}$$

(again, we note that when the set of rules in P is infinite, the disjunction may be infinitary).

One can now show the following theorem.

Theorem 8. *Let P be a convex-constraint program and let $M \subseteq At(P)$. Then M is a supported model of P if and only if M is a model of $Comp(P)$.*

Proof. (Sketch) By the definition, M is a supported model of P if and only if M is a supported model of $mc(P)$. It is a matter of routine checking that $Comp(mc(P))$ and $Comp(P)$ have the same models. Thus the assertion follows from Theorem 4. □

Next, we restrict attention to *finitary* convex-constraint programs, that is, programs with finite set of atoms, and extend to this class of programs the notions of the positive dependency graph and loops. To this end, we exploit its representation as a monotone-constraint program $mc(P)$. That is, we define the positive dependency graph, loops and loop formulas for P as the positive dependency graph, loops and loop formulas of $mc(P)$, respectively. In particular, L is a loop of P if and only if

L is a loop of $mc(P)$ and the loop formula for L, with respect to a convex-constraint program P, is defined as the loop formula $LP(L)$ with respect to the program $mc(P)$[4]. We note that since loop formulas for monotone-constraint programs only modify non-negated literals in the bodies of rules and leave negated literals intact, there seems to be no simple way to extend the notion of a loop formula to the case of a convex-constraint program P without making references to $mc(P)$.

We now define a *loop completion* of a finitary convex-constraint program P as the PL^{cc} theory

$$LComp(P) = Comp(P) \cup \{LP(L)\colon L \text{ is a loop of } P\}.$$

We have the following theorem that provides a necessary and sufficient condition for a set of atoms to be a stable model of a convex-constraint program.

Theorem 9. *Let P be a finitary convex-constraint program. A set $M \subseteq At(P)$ is a stable model of P if and only if M is a model of $LComp(P)$.*

Proof. (Sketch) Since M is a stable model of P if and only of M is a stable model of $mc(P)$, Theorem 6 implies that M is a stable model of P if and only if M is a stable model of $LComp(mc(P))$. It is a matter of routine checking that $LComp(mc(P))$ and $LComp(P)$ have the same models. Thus, the result follows. $\square$

In a similar way, Theorem 7 implies the following result for convex-constraint programs.

Theorem 10. *Let P be a finitary convex-constraint program and M a model of $Comp(P)$. If M^- is not empty, then M violates the loop formula of every terminating loop of $G_P[M^-]$.*

We emphasize that one could simply use $LComp(mc(P))$ as a definition of the loop completion for a convex-constraint logic program. However, our definition of the completion component of the loop completion retains the structure of constraints in a program P, which is important when using loop completion for computation of stable models, the topic we address in the next section of the paper.

8. Applications

In this section, we will use theoretical results on the program completion, loop formulas and loop completion of programs with convex constraints to design and implement a new method for computing stable models of *lparse* programs (Simons et al., 2002).

8.1 *Lparse* Programs

Simons et al. (2002) introduced and studied an extension of normal logic programming with weight atoms. Formally, a *weight atom* is an expression

$$A = l[a_1 = w_1, \ldots, a_k = w_k]u,$$

where a_i, $1 \le i \le k$ are propositional atoms, and l, u and w_i, $1 \le i \le k$ are non-negative integers. If all weights w_i are equal to 1, A is a *cardinality atom*, written as $l\{a_1, \ldots, a_k\}u$.

4. There is one minor simplification one might employ. For a monotone constraint A, $\neg A$ and $\overline{A}$ are equivalent and $\overline{A}$ is antimonotone and so, convex. Thus, we can eliminate the operator $\neg$ from loop formulas of convex-constraint programs by writing $\overline{A}$ instead of $\neg A$.

An *lparse rule* is an expression of the form

$$A \leftarrow A_1, \ldots, A_n$$

where A, $A_1, \ldots, A_n$ are weight atoms. We refer to sets of *lparse* rules as *lparse programs*. Simons et al. (2002) defined for *lparse* programs the semantics of stable models.

A set M of atoms is a *model* of (or *satisfies*) a weight atom $l[a_1 = w_1, \ldots, a_k = w_k]u$ if

$$l \leq \sum_{i=1}^{k} \{w_i \colon a_i \in M\} \leq u.$$

With this semantics a weight atom $l[a_1 = w_1, \ldots, a_k = w_k]u$ can be identified with a constraint (X, C), where $X = \{a_1, \ldots, a_k\}$ and

$$C = \{Y \subseteq X \colon l \leq \sum_{i=1}^{k} \{w_i \colon a_i \in Y\} \leq u\}.$$

We notice that all weights in a weight atom W are non-negative. Therefore, if $M \subseteq M' \subseteq M''$ and both M and M'' are models of W, then M' is also a model of W. It follows that the constraint (X, C) we define above is convex.

Since (X, C) is convex, weight atoms represent a class of convex constraints and *lparse* programs syntactically are a class of programs with convex constraints. This relationship extends to the stable-model semantics. Namely, Marek and Truszczyński (2004) and Marek et al. (2004, 2006) showed that *lparse* programs can be encoded as programs with monotone constraints so that the concept of a stable model is preserved. The transformation used there coincides with the encoding mc described in the previous section, when we restrict the latter to *lparse* programs. Thus, we have the following theorem.

Theorem 11. *Let P be an lparse program. A set $M \subseteq At$ is a stable model of P according to the definition by Simons et al. (2002) if and only if M is a stable model of P according to the definition given in the previous section (when P is viewed as a convex-constraint program).*

It follows that to compute stable models of *lparse* programs we can use the results obtained earlier in the paper, specifically the results on program completion and loop formulas for convex-constraint programs.

Remark. To be precise, the syntax of *lparse* programs is more general. It allows both atoms and negated atoms to appear within weight atoms. It also allows weights to be negative. However, negative weights in *lparse* programs are treated just as a notational convenience. Specifically, an expression of the form $a = w$ within a weight atom (where $w < 0$) represents the expression $\mathbf{not}(a) = -w$ (eliminating negative weights in this way from a weight atom requires modifications of the bounds associated with this weight atom). Moreover, by introducing new propositional variables one can remove occurrences of negative literals from programs. These transformations preserve stable models (modulo new atoms). Marek and Truszczyński (2004) and Marek et al. (2004, 2006) provide a detailed discussion of this transformation.

In addition to weight atoms, the bodies of *lparse* rules may contain propositional literals (atoms and negated atoms) as conjuncts. We can replace these propositional literals with weight atoms as follows: an atom a can be replaced with the cardinality atom $1\{a\}$, and a literal $\mathbf{not}(a)$ — with the cardinality atom $\{a\}0$. This transformation preserves stable models, too. Moreover, the size of the resulting program does not increase more than by a constant factor. Thus, through the transformations discussed here, monotone- and convex-constraint programs capture arbitrary *lparse* programs. $\square$

8.2 Computing Stable Models of *Lparse* Programs

In this section we present an algorithm for computing stable models of *lparse* programs. Our method uses the results we obtained in Section 7 to reduce the problem to that of computing models of the loop completion of an *lparse* program. The loop completion is a formula in the logic PL^{cc}, in which the class of convex constraints is restricted to weight constraints, as defined in the previous subsection. We will denote the fragment of the logic PL^{cc} consisting of such formulas by PL^{wa}.

To make the method practical, we need programs to compute models of theories in the logic PL^{wa}. We will now show a general way to adapt to this task off-the-shelf *pseudo-boolean constraint solvers* (Eén & Sörensson, 2003; Aloul et al., 2002; Walser, 1997; Manquinho & Roussel, 2005; Liu & Truszczyński, 2003).

Pseudo-boolean constraints (PB for short) are integer programming constraints in which variables have 0-1 domains. We will write them as inequalities

$$w_1 \times x_1 + \ldots + w_k \times x_k \; comp \; w, \tag{3}$$

where $comp$ stands for one of the relations $\leq$, $\geq$, $<$ and $>$, w_i's and w are integer coefficients (not necessarily non-negative), and x_i's are integers taking value 0 or 1. A set of pseudo-boolean constraints is a *pseudo-boolean theory*.

Pseudo-boolean constraints can be viewed as constraints. The basic idea is to treat each 0-1 variable x as a propositional atom (which we will denote by the same letter). Under this correspondence, a pseudo-boolean constraint (3) is equivalent to the constraint (X, C), where $X = \{x_1, \ldots, x_k\}$ and

$$C = \{Y \subseteq X \colon \sum_{i=1}^{k} \{w_i \colon x_i \in Y\} \; comp \; w\}$$

in the sense that solutions to (3) correspond to models of (X, C) ($x_i = 1$ in a solution if and only if x_i is true in the corresponding model). In particular, if all coefficients w_i and the bound w in (3) are non-negative, and if $comp = $ '$\geq$', then the constraint (3) is equivalent to a monotone lower-bound weight atom $w[x_1 = w_1, \ldots, x_n = w_n]$.

It follows that an arbitrary weight atom can be represented by one or two pseudo-boolean constraints. More generally, an arbitrary PL^{wa} formula F can be encoded as a set of PB constraints. We will describe the translation as a two-step process.

The first step consists of converting F to a *clausal* form $\tau_{cl}(F)$[5]. To control the size of the translation, we introduce auxiliary propositional atoms. Below, we describe the translation $F \mapsto \tau_{cl}(F)$ under the assumption that F is a formula of the loop completion of an *lparse* program P. Our main motivation is to compute stable models of logic programs and to this end algorithms for computing models of loop completions are sufficient.

5. A PL^{wa} *clause* is any formula $B_1 \wedge \ldots \wedge B_m \rightarrow H_1 \vee \ldots \vee H_n$, where B_i and H_j are weight atoms.

Let F be a formula in the loop completion of an *lparse*-program P. We define $\tau_{cl}(F)$ as follows (in the transformation, we use a propositional atom x as a shorthand for the cardinality atom $C(x) = 1\{x\}$).

1. If F is of the form $A_1 \wedge \ldots \wedge A_n \rightarrow A$, then $\tau_{cl}(F) = F$
2. If F is of the form $x \rightarrow ([bd(r_1)]^{\wedge}) \vee \ldots \vee ([bd(r_l)]^{\wedge})$, then we introduce new propositional atoms $b_{r,1}, \ldots, b_{r,l}$ and set $\tau_{cl}(F)$ to consist of the following PL^{wa} clauses:

$$x \rightarrow b_{r,1} \vee \ldots \vee b_{r,l}$$

$$[bd(r_i)]^{\wedge} \rightarrow b_{r,i}, \text{ for every } bd(r_i)$$

$$b_{r,i} \rightarrow A_j, \text{ for every } bd(r_i) \text{ and } A_j \in bd(r_i)$$

3. If F is of the form $\bigvee L \rightarrow \bigvee_r \{\beta_L(r)\}$, where L is a set of atoms, and every $\beta_L(r)$ is a conjunction of weight atoms, then we introduce new propositional atoms $bdf_{L,r}$ for every $\beta_L(r)$ in F and represent $\bigvee L$ as the weight atom $W_L = 1[l_i = 1 : l_i \in L]$. We then define $\tau_{cl}(F)$ to consist of the following clauses:

$$W_L \rightarrow \bigvee bdf_{L,r}$$

$$\beta_L(r) \rightarrow bdf_{L,r}, \text{ for every } \beta_L(r) \in F$$

$$bdf_{L,r} \rightarrow A_j, \text{ for every } \beta_L(r) \in F \text{ and } A_j \in \beta_L(r).$$

It is clear that the size $\tau_{cl}(F)$ is linear in the size of F.

The second step of the translation, converts a PL^{wa} formula C in a clausal form into a set of PB constraints, $\tau_{pb}(C)$. To define the translation $C \rightarrow \tau_{pb}(C)$, let us consider a PL^{wa} clause C of the form

$$B_1 \wedge \ldots \wedge B_m \rightarrow H_1 \vee \ldots \vee H_n, \tag{4}$$

where B_i's and H_i's are weight atoms.

We introduce new propositional atoms $b_1, \ldots, b_m$ and $h_1, \ldots, h_n$ to represent each weight atom in the clause. As noted earlier in the paper, we simply write x for a weight atoms of the form $1[x = 1]$. With the new atoms, the clause (4) becomes a propositional clause $b_1 \wedge \ldots \wedge b_m \rightarrow h_1 \vee \ldots \vee h_n$. We represent it by the following PB constraint:

$$-b_1 - \ldots - b_m + h_1 + \ldots + h_n \geq 1 - m. \tag{5}$$

Here and later in the paper, we use the same symbols to denote propositional variables and the corresponding 0-1 integer variables. The context will always imply the correct meaning of the symbols. Under this convention, it is easy to see that a propositional clause $b_1 \wedge \ldots \wedge b_m \rightarrow h_1 \vee \ldots \vee h_n$ and its PB constraint (5) have the same models.

We introduce next PB constraints that enforce the equivalence of the newly introduced atoms b_i (or h_i) and the corresponding weight atoms B_i (or H_i).

Let $B = l[a_1 = w_1, \ldots, a_k = w_k]u$ be a weight atom and b a propositional atom. We split B to B^+ and B^- and introduce two more atoms b^+ and b^-. To model $B \equiv b$, we model with pseudo-boolean constraints the following three equivalences: $b \equiv b^+ \wedge b^-$, $b^+ \equiv B^+$, and $b^- \equiv B^-$.

1. The first equivalence can be captured with three propositional clauses. Hence the following three PB constraints model that equivalence:

$$-b + b^+ \geq 0 \tag{6}$$

$$-b + b^{-} \geq 0 \tag{7}$$

$$-b^{+} - b^{-} + b \geq -1 \tag{8}$$

2. The second equivalence, $b^{+} \equiv B^{+}$, can be modeled by the following two PB constraints

$$(-l) \times b^{+} + \sum_{i=1}^{k} (a_i \times w_i) \geq 0 \tag{9}$$

$$-(\sum_{i=1}^{k} w_i - l + 1) \times b^{+} + \sum_{i=1}^{k} (a_i \times w_i) \leq l - 1 \tag{10}$$

3. Similarly, the third equivalence, $b^{-} \equiv B^{-}$, can be modeled by the following two PB constraints

$$(\sum_{i=1}^{k} w_i - u) \times b^{-} + \sum_{i=1}^{k} (a_i \times w_i) \leq \sum_{i=1}^{k} w_i \tag{11}$$

$$(u + 1) \times b^{-} + \sum_{i=1}^{k} (a_i \times w_i) \geq u + 1 \tag{12}$$

We define now $\tau_{pb}(C)$, for a PL^{wa} clause C, as the set of all pseudo-boolean constraints (5) and (6), (7), (8), (11), (12), (9), (10) constructed for every weight atom occurring in C. One can verify that the size of $\tau_{pb}(C)$ is linear in the size of C. Therefore, $\tau_{pb}(\tau_{cl}(F))$ has size linear in the size of F.

In the special case where all B_i's and H_j's are weight atoms of the form $1[b_i = 1]$ and $1[h_j = 1]$, we do not need to introduce any new atoms and PB constraints (6), (7), (8), (11), (12), (9), (10). Then $\tau_{pb}(C)$ consists of a single PB constraint (5).

We have the following theorem establishing the correctness of the transformation τ. The proof of the theorem is straightforward.

Theorem 12. *Let F be a loop completion formula in logic PL^{wa}, and M a set of atoms, $M \subseteq At(F)$. Then M is a model of F in PL^{wa} logic if and only if M has a unique extension M' by some of the new atoms in $At(\tau_{pb}(\tau_{cl}(F)))$ such that M' is a model of the pseudo-boolean theory $\tau_{pb}(\tau_{cl}(F))$.*

We note that when we use solvers designed for PL^{wa} theories, then translation τ_{pb} is no longer needed. The benefit of using such solvers is that we do not need to split weight atoms in the PL^{wa} theories and do not need the auxiliary atoms introduced in τ_{pb}.

8.2.1 THE ALGORITHM

We follow the approach proposed by Lin and Zhao (2002). As in that paper, we first compute the completion of a *lparse* program. Then, we iteratively compute models of the completion using a PB solver. Whenever a model is found, we test it for stability. If the model is not a stable model of the program, we extend the completion by loop formulas identified in Theorem 10. Often, adding a single loop formula filters out several models of $Comp(P)$ that are not stable models of P.

The results given in the previous section ensure that our algorithm is correct. We present it in Figure 1. We note that it may happen that in the worst case exponentially many loop formulas are

```
Input: P — a lparse program;
       A — a pseudo-boolean solver
BEGIN
  compute the completion Comp(P) of P;
  T := τ_pb(τ_cl(Comp(P)));
  do
    if (solver A finds no models of T)
        output "no stable models found" and terminate;
    M := a model of T found by A;
    if (M is stable) output M and terminate;
    compute the reduct P^M of P with respect to M;
    compute the greatest stable model M', contained in M, of P^M;
    M^- := M \ M';
    find all terminating loops in M^-;
    compute loop formulas and convert them into PB constraints using
       τ_pb and τ_cl;
    add all PB constraints computed in the previous step to T;
  while (true);
END
```

Figure 1: Algorithm of *pbmodels*

needed before the first stable model is found or we determine that no stable models exist (Lin & Zhao, 2002). However, that problem arises only rarely in practical situations[6].

The implementation of *pbmodels* supports several PB solvers such as *satzoo* (Eén & Sörensson, 2003), *pbs* (Aloul et al., 2002), *wsatoip* (Walser, 1997). It also supports a program *wsatcc* (Liu & Truszczyński, 2003) for computing models of PL^{wa} theories. When this last program is used, the transformation, from "clausal" PL^{wa} theories to pseudo-boolean theories is not needed. The first two of these four programs are complete PB solvers. The latter two are local-search solvers based on *wsat* (Selman, Kautz, & Cohen, 1994).

We output the message "no stable model found" in the first line of the loop and not simply "no stable models exist" since in the case when A is a local-search algorithm, failure to find a model of the completion (extended with loop formulas in iteration two and the subsequent ones) does not imply that no models exist.

8.3 Performance

In this section, we present experimental results concerning the performance of *pbmodels*. The experiments compared *pbmodels*, combined with several PB solvers, to *smodels* (Simons et al., 2002) and *cmodels* (Babovich & Lifschitz, 2002). We focused our experiments on problems whose state-

6. In fact, in many cases programs turn out to be tight with respect to their supported models. Therefore, supported models are stable and no loop formulas are necessary at all.

ments explicitly involve pseudo-boolean constraints, as we designed *pbmodels* with such problems in mind.

For most benchmark problems we tried *cmodels* did not perform well. Only in one case (vertex-cover benchmark) the performance of *cmodels* was competitive, although even in this case it was not the best performer. Therefore, we do not report here results we compiled for *cmodels*. For a complete set of results we obtained in the experiments we refer to `http://www.cs.uky.edu/ai/pbmodels`.

In the experiments we used instances of the following problems: *traveling salesperson*, *weighted n-queens*, *weighted Latin square*, *magic square*, *vertex cover*, and *Towers of Hanoi*. The *lparse* programs we used for the first four problems involve general pseudo-boolean constraints. Programs modeling the last two problems contain cardinality constraints only.

Traveling salesperson problem (TSP). An instance consists of a weighted complete graph with n vertices, and a bound w. All edge weights and w are non-negative integers. A solution to an instance is a Hamiltonian cycle whose total weight (the sum of the weights of all its edges) is less than or equal to w.

We randomly generated 50 weighted complete graphs with 20 vertices, To this end, in each case we assign to every edge of a complete undirected graph an integer weight selected uniformly at random from the range $[1..19]$. By setting w to 100 we obtained a set of "easy" instances, denoted by *TSP-e* (the bound is high enough for every instance in the set to have a solution). From the same collection of graphs, we also created a set of "hard" instances, denoted by *TSP-h*, by setting w to 62. Since the requirement on the total weight is stronger, the instances in this set in general take more time.

Weighted n-queens problem (WNQ). An instance to the problem consists of a weighted $n \times n$ chess board and a bound w. All weights and the bound are non-negative integers. A solution to an instance is a placement of n queens on the chess board so that no two queens attack each other and the weight of the placement (the sum of the weights of the squares with queens) is not greater than w.

We randomly generated 50 weighted chess boards of the size 20×20, where each chess board is represented by a set of $n \times n$ integer weights $w_{i,j}$, $1 \leq i, j \leq n$, all selected uniformly at random from the range $[1..19]$. We then created two sets of instances, easy (denoted by *wnq-e*) and hard (denoted by *wnq-h*), by setting the bound w to 70 and 50, respectively.

Weighted Latin square problem (WLSQ). An instance consists of an $n \times n$ array of weights $w_{i,j}$, and a bound w. All weights $w_{i,j}$ and w are non-negative integers. A solution to an instance is an $n \times n$ array L with all entries from $\{1, \ldots, n\}$ and such that each element in $\{1, \ldots, n\}$ occurs exactly once in each row and in each column of L, and $\sum_{i=1}^{n} \sum_{j=1}^{n} L[i, j] \times w_{i,j} \leq w$.

We set $n = 10$ and we randomly generated 50 sets of integer weights, selecting them uniformly at random from the range $[1..9]$. Again we created two families of instances, easy (*wlsq-e*) and hard (*wlsq-h*), by setting w to 280 and 225, respectively.

Magic square problem. An instance consists of a positive integer n. The goal is to construct an $n \times n$ array using each integer $1, \ldots n^2$ as an entry in the array exactly once in such a way that entries in each row, each column and in both main diagonals sum up to $n(n^2 + 1)/2$. For the experiments we used the magic square problem for $n = 4, 5$ and 6.

Vertex cover problem. An instance consists of graph with n vertices and m edges, and a non-negative integer k — a bound. A solution to the instance is a subset of vertices of the graph with no more than k vertices and such that at least one end vertex of every edge in the graph is in the subset.

We randomly generated 50 graphs, each with 80 vertices and 400 edges. For each graph, we set k to be a smallest integer such that a vertex cover with that many elements still exists.

Towers of Hanoi problem. This is a slight generalization of the original problem. We considered the case with six disks and three pegs. An instance consists of an initial configuration of disks that satisfies the constraint of the problem (larger disk must not be on top of a smaller one) but does not necessarily requires that all disks are on one peg. These initial configurations were selected so that they were 31, 36, 41 and 63 steps away from the goal configuration (all disks from the largest to the smallest on the third peg), respectively. We also considered a standard version of the problem with seven disks, in which the initial configuration is 127 steps away from the goal.

We encoded each of these problems as a program in the general syntax of *lparse*, which allows the use of relation symbols and variables (Syrjänen, 1999). The programs are available at `http://www.cs.uky.edu/ai/pbmodels`. We then used these programs in combination with appropriate instances as inputs to *lparse* (Syrjänen, 1999). In this way, for each problem and each set of instances we generated a family of ground (propositional) *lparse* programs so that stable models of each of these programs represent solutions to the corresponding instances of the problem (if there are no stable models, there are no solutions). We used these families of *lparse* programs as inputs to solvers we were testing. All these ground programs are also available at `http://www.cs.uky.edu/ai/pbmodels`.

In the tests, we used $pbmodels$ with the following four PB solvers: $satzoo$ (Eén & Sörensson, 2003), pbs (Aloul et al., 2002), $wsatcc$ (Liu & Truszczyński, 2003), and $wsatoip$ (Walser, 1997). In particular, $wsatcc$ deals with PL^{wa} theories directly.

All experiments were run on machines with 3.2GHz Pentium 4 CPU, 1GB memory, running Linux with kernel version 2.6.11, gcc version 3.3.4. In all cases, we used 1000 seconds as the timeout limit.

We first show the results for the *magic square* and *towers of Hanoi* problems. In Table 1, for each solver and each instance, we report the corresponding running time in seconds. Local-search solvers were unable to solve any of the instances in the two problems and so are not included in the table.

Benchmark	*smodels*	*pbmodels-satzoo*	*pbmodels-pbs*
magic square (4×4)	1.36	1.70	2.41
magic square (5×5)	> 1000	28.13	0.31
magic square (6×6)	> 1000	75.58	> 1000
towers of Hanoi $(d = 6, t = 31)$	16.19	18.47	1.44
towers of Hanoi $(d = 6, t = 36)$	32.21	31.72	1.54
towers of Hanoi $(d = 6, t = 41)$	296.32	49.90	3.12
towers of Hanoi $(d = 6, t = 63)$	> 1000	> 1000	3.67
towers of Hanoi $(d = 7, t = 127)$	> 1000	> 1000	22.83

Table 1: Magic square and towers of Hanoi problems

Both $pbmodels\text{-}satzoo$ and $pbmodels\text{-}pbs$ perform better than $smodels$ on programs obtained from the instances of both problems. We observe that $pbmodels\text{-}pbs$ performs exceptionally well in the tower of Hanoi problem. It is the only solver that can compute a plan for 7 disks, which requires 127 steps. Magic square and Towers of Hanoi problems are highly regular. Such problems

	# of SAT instances	*# of UNSAT instances*	*# of UNKNOWN instances*
TSP-e	50	0	0
TSP-h	31	1	18
wnq-e	49	0	1
wnq-h	29	0	21
wlsq-e	45	4	1
wlsq-h	8	41	1
vtxcov	50	0	0

Table 2: Summary of Instances

	smodels	*pbmodels-satzoo*	*pbmodels-pbs*
TSP-e	$45/17$	$50/30$	$18/3$
TSP-h	$7/3$	$16/14$	$0/0$
wnq-e	$11/5$	$26/23$	$0/0$
wnq-h	$2/2$	$0/0$	$0/0$
wlsq-e	$21/1$	$49/29$	$46/19$
wlsq-h	$0/0$	$47/26$	$47/23$
vtxcov	$50/40$	$50/1$	$47/3$
sum over all	$136/68$	$238/123$	$158/48$

Table 3: Summary on all instances

are often a challenge for local-search problems, which may explain a poor performance we observed for $pbmodels$-$wsatcc$ and $pbmodels$-$wsatoip$ on these two benchmarks.

For the remaining four problems, we used 50-element families of instances, which we generated randomly in the way discussed above. We studied the performance of complete solvers ($smodels$, $pbmodels$-$satzoo$ and $pbmodels$-pbs) on all instances. We then included local-search solvers ($pbmodels$-$wsatcc$ and $pbmodelswsatoip$) in the comparisons but restricted attention only to instances that were determined to be satisfiable (as local-search solvers are, by their design, unable to decide unsatisfiability). In Table 2, for each family we list how many of its instances are satisfiable, unsatisfiable, and for how many of the instances none of the solvers we tried was able to decide satisfiability.

In Table 3, for each of the seven families of instances and for each *complete* solver, we report two values s/w, where s is the number of instances solved by the solver and w is the number of times it was the fastest among the three.

The results in Table 3 show that overall $pbmodels$-$satzoo$ solved more instances than $pbmodels$-pbs, followed by $smodels$. When we look at the number of times a solver was the fastest one, $pbmodels$-$satzoo$ was a clear winner overall, followed by $smodels$ and then by $pbmodels$-pbs. Looking at the seven families of tests individually, we see that $pbmodels$-$satzoo$ performed better than the other two solvers on five of the families. On the other two $smodels$ was the best performer (although, it is a clear winner only on the vertex-cover benchmark; all solvers were essentially ineffective on the *wnq-h*).

We also studied the performance of $pbmodels$ combined with local-search solvers $wsatcc$ (Liu & Truszczyński, 2003) and $wsatoip$ (Walser, 1997). For this study, we considered only those instances in the seven families that we knew were satisfiable. Table 4 presents results for all solvers

	smodels	*pbmd-satzoo*	*pbmd-pbs*	*pbmd-wsatcc*	*pbmd-wsatoip*
TSP-e	$45/3$	$50/5$	$18/2$	$32/7$	$47/34$
TSP-h	$7/0$	$16/2$	$0/0$	$19/6$	$28/22$
wnq-e	$11/0$	$26/0$	$0/0$	$49/45$	$49/4$
wnq-h	$2/0$	$0/0$	$0/0$	$29/15$	$29/14$
wlsq-e	$21/0$	$45/0$	$44/0$	$45/33$	$45/14$
wlsq-h	$0/0$	$7/0$	$8/0$	$7/1$	$8/7$
vtxcov	$50/0$	$50/0$	$47/0$	$50/36$	$50/15$
sum over all	$136/3$	$194/7$	$117/2$	$231/143$	$256/110$

Table 4: Summary on SAT instances

we studied (including the complete ones). As before, each entry provides a pair of numbers s/w, where s is the number of solved instances and w is the number of times the solver performed better than its competitors.

The results show superior performance of $pbmodels$ combined with local-search solvers. They solve more instances than complete solvers (including $smodels$). In addition, they are significantly faster, winning much more frequently than complete solvers do (complete solvers were faster only on 12 instances, while local-search solvers were faster on 253 instances).

Our results demonstrate that $pbmodels$ with solvers of pseudo-boolean constraints outperforms $smodels$ on several types of search problems involving pseudo-boolean (weight) constraints).

We note that we also analyzed the run-time distributions for each of these families of instances. A run-time distribution is regarded as a more accurate and detailed measure of the performance of algorithms on randomly generated instances[7]. The results are consistent with the summary results presented above and confirm our conclusions. As the discussion of run-time distributions requires much space, we do not include this analysis here. They are available at the website `http://www.cs.uky.edu/ai/pbmodels`.

9. Related work

Extensions of logic programming with means to model properties of *sets* (typically consisting of ground terms) have been extensively studied. Usually, these extensions are referred to by the common term of *logic programming with aggregates*. The term comes from the fact that most properties of sets of practical interest are defined through "aggregate" operations such as sum, count, maximum, minimum and average. We chose the term *constraint* to stress that we speak about abstract properties that define constraints on truth assignments (which we view as sets of atoms).

Mumick, Pirahesh, and Ramakrishnan (1990), and Kemp and Stuckey (1991) were among the first to study logic programs with aggregates. Recently, Niemelä et al. (1999) and Simons et al. (2002) introduced the class of *lparse* programs. We discussed this formalism in detail earlier in this paper.

Pelov (2004) and Pelov et al. (2006) studied a more general class of aggregates and developed a systematic theory of aggregates in logic programming based on the approximation theory (Denecker, Marek, & Truszczyński, 2000). The resulting theory covers not only the stable models semantics but also the supported-model semantics and extensions of 3-valued Kripke-Kleene and

7. Hoos and Stützle (2005) provide a detailed discussion of this matter in the context of local-search methods.

well-founded semantics. The formalism introduced and studied by Pelov (2004) and Pelov et al. (2006) allows for arbitrary aggregates (not only monotone ones) to appear in the bodies of rules. However, it does not allow for aggregates to appear in the heads of program clauses. Due to differences in the syntax and the scope of semantics studied there is no simple way to relate Pelov's (2004) and Pelov et al.'s (2006) formalism to programs with monotone (convex) constraints. We note though that programs with abstract monotone constraints with the heads of rules of the form $C(a)$ can be viewed almost literally as programs in the formalism by Pelov (2004) and Pelov et al. (2006) and that they have the same stable models according to the definitions we used in this paper and those by Pelov (2004) and Pelov et al. (2006).

Faber et al. (2004) developed the theory of *disjunctive* logic programs with aggregates. Similarly as Pelov (2004) and Pelov et al. (2006), Faber et al. (2004) do not allow for aggregates to appear in the heads of program clauses. This is one of the differences between that approach and programs with monotone (convex) constraints we studied here. The other major difference is related to the postulate of the minimality of stable models (called *answer sets* in the context of the formalism considered by Faber et al., 2004). In keeping with the spirit of the original answer-set semantics (Gelfond & Lifschitz, 1991), answer sets of disjunctive programs with aggregates, as defined by Faber et al. (2004), are minimal models. Stable models of programs with abstract constraints do not have this property. However, for the class of programs with abstract monotone constraints with the heads of rules of the form $C(a)$ the semantics of answer sets defined by Faber et al. (2004) coincides with the semantics of stable models by Marek and Truszczyński (2004) and Marek et al. (2004, 2006).

Yet another approach to aggregates in logic programming was presented by Son and Pontelli (2006). That approach considered programs of the syntax similar to programs with monotone abstract constraints. It allowed arbitrary constraints (not only monotone ones) but not under the scope of $\mathbf{not}$ operator. A general principle behind the definition of the stable-model semantics by Son and Pontelli (2006) is to view a program with constraints as a concise representation of a set of its "instances", each being a normal logic program. Stable models of the program with constraints are defined as stable models of its instances and is quite different from the operator-based definition by Marek and Truszczyński (2004) and Marek et al. (2004, 2006). However, for programs with *monotone* constraint atoms which fall in the scope of the formalism of Son and Pontelli (2006) both approaches coincide.

We also note that recently Son et al. (2006) presented a *conservative* extension of the syntax proposed by Marek and Truszczyński (2004) Marek et al. (2006), in which clauses are built of arbitrary constraint atoms.

Finally, we point out the work by Ferraris and Lifschitz (2004) and Ferraris (2005) which treats aggregates as *nested expressions*. In particular, Ferraris (2005) introduces a propositional logic with a certain nonclassical semantics, and shows that it extends several approaches to programs with aggregates, including those by Simons et al. (2002) (restricted to core lparse programs) and Faber et al. (2004). The nature of the relationship of the formalism by Ferraris (2005) and programs with abstract constraints remains an open problem.

10. Conclusions

Our work shows that concepts, techniques and results from normal logic programming, concerning strong and uniform equivalence, tightness and Fages lemma, program completion and loop formulas, generalize to the abstract setting of programs with monotone and convex constraints. These general properties specialize to *new* results about *lparse* programs (with the exception of the characterization strong equivalence of *lparse* programs, which was first obtained by Turner, 2003).

Given these results we implemented a new software *pbmodels* for computing stable models of *lparse* programs. The approach reduces the problem to that of computing models of theories consisting of pseudo-boolean constraints, for which several fast solvers exist (Manquinho & Roussel, 2005). Our experimental results show that *pbmodels* with PB solvers, especially local search PB solvers, performs better than $smodels$ on several types of search problems we tested. Moreover, as new and more efficient solvers of pseudo-boolean constraints become available (the problem is receiving much attention in the satisfiability and integer programming communities), the performance of *pbmodels* will improve accordingly.

Acknowledgments

We acknowledge the support of NSF grants IIS-0097278 and IIS-0325063. We are grateful to the reviewers for their useful comments and suggestions.

This paper combines and extends results included in conference papers (Liu & Truszczyński, 2005b, 2005a).

References

Aloul, F., Ramani, A., Markov, I., & Sakallah, K. (2002). PBS: a backtrack-search pseudo-boolean solver and optimizer. In *Proceedings of the 5th International Symposium on Theory and Applications of Satisfiability*, pp. 346 – 353.

Babovich, Y., & Lifschitz, V. (2002). *Cmodels package.*. `http://www.cs.utexas.edu/users/tag/cmodels.html`.

Baral, C. (2003). *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press.

Clark, K. (1978). Negation as failure. In Gallaire, H., & Minker, J. (Eds.), *Logic and data bases*, pp. 293–322. Plenum Press, New York-London.

Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., & Pfeifer, G. (2003). Aggregate functions in disjunctive logic programming: semantics, complexity, and implementation in DLV. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-2003)*, pp. 847–852. Morgan Kaufmann.

Denecker, M., Marek, V., & Truszczyński, M. (2000). Approximations, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning. In Minker, J. (Ed.), *Logic-Based Artificial Intelligence*, pp. 127–144. Kluwer Academic Publishers.

Eén, N., & Sörensson, N. (2003). An extensible SAT solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT-2003*, Vol. 2919 of *LNCS*, pp. 502–518. Springer.

Eiter, T., & Fink, M. (2003). Uniform equivalence of logic programs under the stable model semantics. In *Proceedings of the 2003 International Conference on Logic Programming*, Vol. 2916 of *Lecture Notes in Computer Science*, pp. 224–238, Berlin. Springer.

Erdem, E., & Lifschitz, V. (2003). Tight logic programs. *Theory and Practice of Logic Programming*, *3*(4-5), 499–518.

Faber, W., Leone, N., & Pfeifer, G. (2004). Recursive aggregates in disjunctive logic programs: Semantics and complexity.. In *Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004)*, Vol. 3229 of *LNAI*, pp. 200 – 212. Springer.

Fages, F. (1994). Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, *1*, 51–60.

Ferraris, P. (2005). Answer sets for propositional theories. In *Logic Programming and Nonmonotonic Reasoning, 8th International Conference, LPNMR 2005*, Vol. 3662 of *LNAI*, pp. 119–131. Springer.

Ferraris, P., & Lifschitz, V. (2004). Weight constraints ans nested expressions. *Theory and Practice of Logic Programming*, *5*, 45–74.

Gelfond, M., & Leone, N. (2002). Logic programming and knowledge representation – the A-prolog perspective. *Artificial Intelligence*, *138*, 3–38.

Gelfond, M., & Lifschitz, V. (1988). The stable semantics for logic programs. In *Proceedings of the 5th International Conference on Logic Programming*, pp. 1070–1080. MIT Press.

Gelfond, M., & Lifschitz, V. (1991). Classical negation in logic programs and disjunctive databases. *New Generation Computing*, *9*, 365–385.

Hoos, H., & Stützle, T. (2005). *Stochastic Local Search Algorithms — Foundations and Applications*. Morgan-Kaufmann.

Kemp, D., & Stuckey, P. (1991). Semantics of logic programs with aggregates. In *Logic Programming, Proceedings of the 1991 International Symposium*, pp. 387–401. MIT Press.

Lifschitz, V., Pearce, D., & Valverde, A. (2001). Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, *2(4)*, 526–541.

Lin, F. (2002). Reducing strong equivalence of logic programs to entailment in classical propositional logic. In *Principles of Knowledge Representation and Reasoning, Proceedings of the 8th International Conference (KR2002)*. Morgan Kaufmann Publishers.

Lin, F., & Zhao, Y. (2002). ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-2002)*, pp. 112–117. AAAI Press.

Liu, L., & Truszczyński, M. (2003). Local-search techniques in propositional logic extended with cardinality atoms. In Rossi, F. (Ed.), *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming, CP-2003*, Vol. 2833 of *LNCS*, pp. 495–509. Springer.

Liu, L., & Truszczyński, M. (2005a). Pbmodels - software to compute stable models by pseudoboolean solvers. In *Logic Programming and Nonmonotonic Reasoning, Proceedings of the 8th International Conference (LPNMR-05)*, LNAI 3662, pp. 410–415. Springer.

Liu, L., & Truszczyński, M. (2005b). Properties of programs with monotone and convex constraints. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05)*, pp. 701–706. AAAI Press.

Manquinho, V., & Roussel, O. (2005). Pseudo boolean evaluation 2005.. `http://www.cril.univ-artois.fr/PB05/`.

Marek, V., Niemelä, I., & Truszczyński, M. (2004). Characterizing stable models of logic programs with cardinality constraints. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning*, Vol. 2923 of *Lecture Notes in Artificial Intelligence*, pp. 154–166. Springer.

Marek, V., Niemelä, I., & Truszczyński, M. (2006). Logic programs with monotone abstract constraint atoms. *Theory and Practice of Logic Programming*. Submitted.

Marek, V., & Truszczyński, M. (2004). Logic programs with abstract constraint atoms. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04)*, pp. 86–91. AAAI Press.

Mumick, I., Pirahesh, H., & Ramakrishnan, R. (1990). The magic of duplicates and aggregates. In *Proceedings of the 16th International Conference on Very Large Data Bases, VLDB 1990*, pp. 264–277. Morgan Kaufmann.

Niemelä, I., Simons, P., & Soininen, T. (1999). Stable model semantics of weight constraint rules. In *Proceedings of LPNMR-1999*, Vol. 1730 of *LNAI*, pp. 317–331. Springer.

Pelov, N. (2004). Semantics of logic programs with aggregates. *PhD Thesis. Department of Computer Science, K.U.Leuven, Leuven, Belgium.*

Pelov, N., Denecker, M., & Bruynooghe, M. (2004). Partial stable models for logic programs with aggregates. In Lifschitz, V., & Niemelä, I. (Eds.), *Logic programming and Nonmonotonic Reasoning, Proceedings of the* 7^{th} *International Conference*, Vol. 2923, pp. 207–219. Springer.

Pelov, N., Denecker, M., & Bruynooghe, M. (2006). Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming*. Accepted (available at `http://www.cs.kuleuven.ac.be/~dtai/projects/ALP/TPLP/`).

Selman, B., Kautz, H., & Cohen, B. (1994). Noise strategies for improving local search. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-1994)*, pp. 337–343, Seattle, USA. AAAI Press.

Simons, P., Niemelä, I., & Soininen, T. (2002). Extending and implementing the stable model semantics. *Artificial Intelligence*, *138*, 181–234.

Son, T., & Pontelli, E. (2006). A constructive semantic characterization of aggregates in anser set programming. *Theory and Practice of Logic Programming*. Accepted (available at `http://arxiv.org/abs/cs.AI/0601051`).

Son, T., Pontelli, E., & Tu, P. (2006). Answer sets for logic programs with arbitrary abstract constraint atoms. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI-06)*. AAAI Press.

Syrjänen, T. (1999). *lparse*, a procedure for grounding domain restricted logic programs. `http://www.tcs.hut.fi/Software/smodels/lparse/`.

Turner, H. (2003). Strong equivalence made easy: Nested expressions and weight constraints. *Theory and Practice of Logic Programming*, *3, (4&5)*, 609–622.

Walser, J. (1997). Solving linear pseudo-boolean constraints with local search. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pp. 269–274. AAAI Press.