Conflict-Driven Disjunctive Answer Set Solving

Christian Drescher and **Martin Gebser** and **Torsten Grote** and **Benjamin Kaufmann** and **Arne König** and **Max Ostrowski** and **Torsten Schaub**
Universität Potsdam, Institut für Informatik, August-Bebel-Str. 89, D-14482 Potsdam, Germany

Abstract

We elaborate a uniform approach to computing answer sets of disjunctive logic programs based on state-of-the-art Boolean constraint solving techniques. Starting from a constraint-based characterization of answer sets, we develop advanced solving algorithms, featuring backjumping and conflict-driven learning using the First-UIP scheme as well as sophisticated unfounded set checking. As a final result, we obtain a competitive solver for Σ_2^P-complete problems, taking advantage of Boolean constraint solving technology without using any legacy solvers as black boxes.

Introduction

Answer Set Programming (ASP; (Baral 2003)) has become an attractive tool for Knowledge Representation and Reasoning (KRR). In fact, many important problems in KRR have an elevated degree of complexity, calling for expressive solving paradigms being able to capture problems at the second level of the polynomial hierarchy (cf. (Schaefer & Umans 2002) for a survey). One possibility to deal with such a problem consists in expressing it as a Quantified Boolean Formula (QBF) and then to use some QBF solver to compute its solutions.[1] Another approach is furnished by ASP solvers dealing with disjunctive logic programs, that is, logic programs allowing for disjunction in the heads and (default) negation in the bodies of rules.

As regards knowledge representation, the semantics underlying ASP allows for specifying problems in a uniform way through pairs of an instance-independent encoding (containing schematic rules with first-order variables) and a set of facts (cf. (Schlipf 1995; Marek & Truszczyński 1999; Niemelä 1999)). The computation of answer sets, corresponding to problem solutions, is then divided into two phases: first, the grounding of the encoding relative to the given instance, which is done by grounders like (the grounding component of) *dlv* (Ricca, Faber, & Leone 2006), *gringo* (Gebser, Schaub, & Thiele 2007), or *lparse* (Syrjänen); second, the computation of answer sets of the ground logic program resulting from the first phase by some ASP solver. In this paper, we deal with the second phase: the solving of ground programs in disjunctive ASP.

The first efficient disjunctive ASP solver, *dlv* (Leone *et al.* 2006), was developed just about a decade ago. Since then, only two further disjunctive ASP solvers have emerged, namely, *gnt* (Janhunen *et al.* 2006) and *cmodels* (Lierler 2005). The rareness of available solvers is mainly due to the elevated complexity of the underlying solving process along with the resulting implementation difficulties.

We address this gap by proposing a uniform approach to disjunctive ASP solving based on advanced Boolean constraint solving techniques (Mitchell 2005), as applied for instance in the area of Boolean Satisfiability (SAT). Our approach builds upon and extends the one developed in (Gebser *et al.* 2007b) for normal logic programs. We start from a logical characterization of answer sets based on loop formulas due to (Lee 2005). In turn, we develop a corresponding characterization in terms of Boolean constraints, specified via the generic concept of nogoods (Dechter 2003), that is, sets of literals not contained in any solution. Based on this, we devise constraint-based algorithms for disjunctive ASP solving, featuring backjumping and conflict-driven learning using the First-UIP scheme as well as elaborate unfounded set checking. As a final result, we obtain a competitive ASP solver for Σ_2^P-complete problems, taking advantage of Boolean constraint solving technology without using any legacy solvers (for either SAT or ASP) as black boxes.

Background

A (*disjunctive*) *logic program* over an alphabet $\mathcal{A}$ is a finite set of *rules* r of the form

$$a_1; \ldots; a_l \leftarrow b_{l+1}, \ldots, b_m, \sim c_{m+1}, \ldots, \sim c_n \ ,$$

where $a_i, b_j, c_k \in \mathcal{A}$ are *atoms* for $1 \leq i \leq l < j \leq m < k \leq n$. Let $H(r) = \{a_1, \ldots, a_l\}$ be the *head* of r and $B(r) = \{b_{l+1}, \ldots, b_m, \sim c_{m+1}, \ldots, \sim c_n\}$ the *body* of r. The set of atoms occurring in a logic program Π is denoted by $A(\Pi)$, and $B(\Pi) = \{B(r) \mid r \in \Pi\}$ is the set of bodies in Π.

Following (Lee 2005), we characterize the answer sets of a logic program by its (classical) models satisfying all loop formulas. A program Π is then represented by the set RF_Π

[1] The satisfiability problem for general QBFs is $PSPACE$-complete (see, e.g., (Papadimitriou 1994)), while it is Σ_2^P-complete for 2QBFs. General QBF solving methods and ones specialized to 2QBFs have been investigated in (Ranjan, Tang, & Malik 2004).

of formulas defined as follows:

$$RF_\Pi = \Big\{ \Big(\bigwedge_{b \in B(r) \cap \mathcal{A}} b \ \wedge \bigwedge_{\sim c \in B(r)} \neg c \Big) \rightarrow \bigvee_{a \in H(r)} a \ \mid r \in \Pi \Big\} .$$

Furthermore, for a set Y of atoms, we let

$$sup_\Pi(Y) = \{ r \in \Pi \mid H(r) \cap Y \neq \emptyset, B(r) \cap Y = \emptyset \}$$

be the set of rules from Π that can externally support Y. The (*disjunctive*) *loop formula* (Lee 2005) of Y, $LF_\Pi(Y)$, is:

$$\bigvee_{a \in Y} a \ \rightarrow \bigvee_{r \in sup_\Pi(Y)} \Big(\bigwedge_{b \in B(r) \cap \mathcal{A}} b \ \wedge \bigwedge_{\sim c \in B(r)} \neg c \ \wedge \bigwedge_{a \in H(r) \setminus Y} \neg a \Big) . \quad (1)$$

According to (Lee 2005), a set $X \subseteq \mathcal{A}$ is an *answer set* of a program Π, if $X \models RF_\Pi \cup \{ LF_\Pi(Y) \mid Y \subseteq \mathcal{A} \}$. However, the set of loop formulas can be further restricted: We call a nonempty set $L \subseteq \mathcal{A}$ a *loop* of Π, if for all nonempty $K \subset L$, there is some $r \in \Pi$ such that $H(r) \cap K \neq \emptyset$ and $B(r) \cap (L \setminus K) \neq \emptyset$ (cf. (Gebser, Lee, & Lierler 2006)). Note that every singleton contained in $\mathcal{A}$ is a loop of Π, and if all loops of Π are singletons, then Π is called *tight* (Erdem & Lifschitz 2003). Finally, let $loop(\Pi)$ denote the set of all loops of Π and $LF_\Pi = \{ LF_\Pi(L) \mid L \in loop(\Pi) \}$. Then, X is an answer set of Π iff $X \models RF_\Pi \cup LF_\Pi$ (Lee 2005).

A Boolean *assignment* $\mathbf{A}$ over a *domain*, $dom(\mathbf{A})$, is a sequence $(\sigma_1, \ldots, \sigma_n)$ of (*signed*) *literals* σ_i of form $\mathbf{T}p$ or $\mathbf{F}p$ for $p \in dom(\mathbf{A})$ and $1 \leq i \leq n$; $\mathbf{T}p$ expresses that p is *true* and $\mathbf{F}p$ that it is *false*. We denote the complement of a literal σ by $\overline{\sigma}$, that is, $\overline{\mathbf{T}p} = \mathbf{F}p$ and $\overline{\mathbf{F}p} = \mathbf{T}p$. Furthermore, $\mathbf{A} \circ \mathbf{B}$ denotes the sequence obtained by concatenating assignments $\mathbf{A}$ and $\mathbf{B}$. We sometimes abuse notation and identify an assignment with the set of its contained literals. Given this, we access the true and false propositions in $\mathbf{A}$ via $\mathbf{A}^\mathbf{T} = \{ p \in dom(\mathbf{A}) \mid \mathbf{T}p \in \mathbf{A} \}$ and $\mathbf{A}^\mathbf{F} = \{ p \in dom(\mathbf{A}) \mid \mathbf{F}p \in \mathbf{A} \}$.

For a canonical representation of Boolean constraints, we use the CSP concept of a nogood (Dechter 2003). In our setting, a *nogood* is a set $\{\sigma_1, \ldots, \sigma_m\}$ of literals, expressing a constraint violated by any assignment $\mathbf{A}$ containing $\sigma_1, \ldots, \sigma_m$. Given a set Δ of nogoods, we adopt the convention that $dom(\mathbf{A}) = \bigcup_{\delta \in \Delta} (\{ p \mid \mathbf{T}p \in \delta \} \cup \{ p \mid \mathbf{F}p \in \delta \})$. Finally, an assignment $\mathbf{A}$ such that $\mathbf{A}^\mathbf{T} \cup \mathbf{A}^\mathbf{F} = dom(\mathbf{A})$ and $\mathbf{A}^\mathbf{T} \cap \mathbf{A}^\mathbf{F} = \emptyset$ is a *solution* for Δ, if $\delta \not\subseteq \mathbf{A}$ for all $\delta \in \Delta$.

Nogoods

Our approach to disjunctive ASP solving is centered around lookback techniques relying on conflict analysis, primarily tracking the reasons for unit propagation. In order to identify such reasons, we specify the constraints underlying unit propagation in terms of nogoods (Dechter 2003). This provides us with a uniform framework for describing propagation via a program, its completion, and loop formulas.

We start by considering programs as sets of implications. To abstract from default negation, for a default literal ℓ, let

$$\boldsymbol{t}\ell = \begin{cases} \mathbf{T}\ell & \text{if } \ell \in \mathcal{A} \\ \mathbf{F}a & \text{if } \ell = \sim a \end{cases} \quad \text{and} \quad \boldsymbol{f}\ell = \begin{cases} \mathbf{F}\ell & \text{if } \ell \in \mathcal{A} \\ \mathbf{T}a & \text{if } \ell = \sim a . \end{cases}$$

The following set $\gamma(C)$ of nogoods then defines whether a set C of default literals must be assigned $\mathbf{T}$ or $\mathbf{F}$ in terms of the conjunction of its elements:

$$\gamma(C) = \big\{ \{\mathbf{F}C\} \cup \{ \boldsymbol{t}\ell \mid \ell \in C \} \big\} \cup \big\{ \{\mathbf{T}C, \boldsymbol{f}\ell\} \mid \ell \in C \big\} .$$

This allows us to characterize the implications expressed by a program Π via the following nogoods:

$$\Delta_\Pi = \textstyle\bigcup_{r \in \Pi} \big(\gamma(B(r)) \cup \big\{ \{\mathbf{T}B(r)\} \cup \{ \mathbf{F}a \mid a \in H(r) \} \big\} \big) .$$

For a program Π containing rule $a; b \leftarrow c, \sim d$, the nogoods in $\gamma(\{c, \sim d\}) = \big\{ \{\mathbf{F}\{c, \sim d\}, \mathbf{T}c, \mathbf{F}d\}, \{\mathbf{T}\{c, \sim d\}, \mathbf{F}c\}, \{\mathbf{T}\{c, \sim d\}, \mathbf{T}d\} \big\}$ and $\{\mathbf{T}\{c, \sim d\}, \mathbf{F}a, \mathbf{F}b\}$ belong to Δ_Π.

The solutions for Δ_Π, projected to $A(\Pi)$, correspond to the models of Π.

Proposition 1 *Let Π be a logic program and $X \subseteq A(\Pi)$.*
Then, $X \models RF_\Pi$ iff $X = \mathbf{A}^\mathbf{T} \cap A(\Pi)$ for a (unique) solution $\mathbf{A}$ for Δ_Π.

In order to describe the completion of a program Π via nogoods, we make use of "shifting" (Gelfond *et al.* 1991):

$$\vec{\Pi} = \big\{ a_i \leftarrow B(r), \sim a_1, \ldots, \sim a_{i-1}, \sim a_{i+1}, \ldots, \sim a_l \mid r \in \Pi, H(r) = \{a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_l\} \big\} .$$

Note that every answer set of $\vec{\Pi}$ is also an answer set of Π, but not vice versa.

However, shifting retains the loop formulas of singletons.

Proposition 2 *Let Π be a logic program.*
For every $a \in \mathcal{A}$, we have $LF_\Pi(\{a\}) \equiv LF_{\vec{\Pi}}(\{a\})$.

This property allows us to check the support of singletons on the shifted version of a program.

To this end, the following nogood $\delta(a, D)$ excludes that $a \in \mathcal{A}$ is assigned $\mathbf{T}$ while all elements of D are false:

$$\delta(a, D) = \{\mathbf{T}a\} \cup \{ \mathbf{F}d \mid d \in D \} .$$

For singletons, the nogoods in $\Theta_{\vec{\Pi}}$ then regulate support:

$$\Theta_{\vec{\Pi}} = \textstyle\bigcup_{\vec{r} \in \vec{\Pi}} \gamma(B(\vec{r})) \cup \{ \delta(a, B(sup_{\vec{\Pi}}(\{a\}))) \mid a \in A(\vec{\Pi}) \} .$$

For instance, if $sup_\Pi(\{a\}) = \{a; b \leftarrow c, \sim d, \ a \leftarrow e, \sim f\}$, we have $sup_{\vec{\Pi}}(\{a\}) = \{a \leftarrow c, \sim d, \sim b, \ a \leftarrow e, \sim f\}$, so that $\Theta_{\vec{\Pi}}$ contains the nogoods in $\gamma(\{c, \sim d, \sim b\})$, $\gamma(\{e, \sim f\})$, and $\delta(a, \{\{c, \sim d, \sim b\}, \{e, \sim f\}\}) = \{\mathbf{T}a, \mathbf{F}\{c, \sim d, \sim b\}, \mathbf{F}\{e, \sim f\}\}$.

Similar to Proposition 1, we obtain the following result.

Proposition 3 *Let Π be a logic program and $X \subseteq \mathcal{A}$.*
Then, $X \models \{ LF_\Pi(\{a\}) \mid a \in \mathcal{A} \}$ iff $X = \mathbf{A}^\mathbf{T} \cap A(\Pi)$ for a (unique) solution $\mathbf{A}$ for $\Theta_{\vec{\Pi}}$.

Given that, for a tight program Π, every loop of Π is a singleton, the answer sets of Π (or models of $RF_\Pi \cup LF_\Pi$, respectively) coincide with the solutions for $\Delta_\Pi \cup \Theta_{\vec{\Pi}}$.

Theorem 4 *Let Π be a tight logic program and $X \subseteq \mathcal{A}$.*
Then, X is an answer set of Π iff $X = \mathbf{A}^\mathbf{T} \cap A(\Pi)$ for a (unique) solution $\mathbf{A}$ for $\Delta_\Pi \cup \Theta_{\vec{\Pi}}$.

The last result still holds after replacing Δ_Π by $\Delta_{\vec{\Pi}}$. We further discuss this alternative in the system section below.

Note that $\Delta_\Pi \cup \Theta_{\vec{\Pi}}$ amounts to the completion (Ben-Eliyahu & Dechter 1994; Lee & Lifschitz 2003) of Π, provided that Π does not contain any tautological rules r where

$H(r) \cap B(r) \neq \emptyset$. Notably, conjunctions expressed by the bodies in $B(\Pi)$ or $B(\vec{\Pi})$, respectively, are represented by propositions in $\Delta_\Pi \cup \Theta_{\vec{\Pi}}$. For normal programs (being primitive disjunctive programs), this has been shown to exponentially reduce proof complexity (Gebser & Schaub 2006).

We now consider non-tight programs where loops are not necessarily singletons. In contrast to tight programs, in the worst case, exponentially many loop formulas are required to single out the answer sets among the models of a program's completion (Lifschitz & Razborov 2006). The nogoods associated to such loop formulas are thus not meant to be determined a priori. Rather, we below use them to explain why assignments do not correspond to answer sets.

In order to identify the nogoods arising from loop formulas, reconsider $LF_\Pi(Y)$ given in (1). For a particular $r \in sup_\Pi(Y)$, observe that r is satisfied and, hence, does not support Y wrt an interpretation if either $B(r)$ is false or some $a \in H(r) \setminus Y$ is true. Accordingly, $sat_r(Y)$ contains all literals that would satisfy r independently from Y:

$$sat_r(Y) = \{\mathbf{F}B(r)\} \cup \{\mathbf{T}a \mid a \in H(r) \setminus Y\} .$$

We call a set Y of atoms *unfounded* by Π wrt an assignment $\mathbf{A}$, if for each $r \in sup_\Pi(Y)$, $\mathbf{A}$ contains some literal from $sat_r(Y)$. In this case, all atoms in Y must be false, which is expressed by the following set of nogoods:

$$\lambda_\Pi(Y) = \big\{\{\sigma_1, \ldots, \sigma_m\} \mid (\sigma_1, \ldots, \sigma_m) \in \{\mathbf{T}a \mid a \in Y\} \times \textstyle\prod_{r \in sup_\Pi(Y)} sat_r(Y)\big\} .$$

Note that the number of nogoods in $\lambda_\Pi(Y)$ is exponential in $|sup_\Pi(Y)|$. However, as mentioned above, we do not intend to construct $\lambda_\Pi(Y)$ a priori. Rather, in the next section, we detail how violations of $\lambda_\Pi(Y)$ can be checked on Π itself.

As an example, consider $sup_\Pi(\{a, e\}) = \{a; b \leftarrow c, {\sim}d,\ e; f \leftarrow d\}$. We have $sat_{a;b \leftarrow c,{\sim}d}(\{a, e\}) = \{\mathbf{F}\{c, {\sim}d\}, \mathbf{T}b\}$ and $sat_{e;f \leftarrow d}(\{a, e\}) = \{\mathbf{F}\{d\}, \mathbf{T}f\}$. Thus, we get $\lambda_\Pi(\{a, e\}) = \big\{\{\sigma, \mathbf{F}\{c, {\sim}d\}, \mathbf{F}\{d\}\}, \{\sigma, \mathbf{F}\{c, {\sim}d\}, \mathbf{T}f\}, \{\sigma, \mathbf{T}b, \mathbf{F}\{d\}\}, \{\sigma, \mathbf{T}b, \mathbf{T}f\} \mid \sigma \in \{\mathbf{T}a, \mathbf{T}e\}\big\}$.

Given that singletons are already dealt with via the nogoods in $\Theta_{\vec{\Pi}}$, additional nogoods, mainly aiming at the external support of loops, can concentrate on non-singletons:

$$\Lambda_\Pi = \textstyle\bigcup_{Y \subseteq A(\Pi), |Y| > 1} \lambda_\Pi(Y) .$$

We thus obtain the following counterpart of Proposition 3.

Proposition 5 *Let Π be a logic program and $X \subseteq A(\Pi)$.*

Then, $X \models \{LF_\Pi(Y) \mid Y \subseteq A(\Pi), |Y| > 1\}$ iff $X = \mathbf{A}^\mathbf{T} \cap A(\Pi)$ for a (unique) solution $\mathbf{A}$ for $\Lambda_\Pi \cup \bigcup_{r \in \Pi} \gamma(B(r))$.

Combining Proposition 3 and 5 yields the next result.

Proposition 6 *Let Π be a logic program and $X \subseteq \mathcal{A}$.*

Then, $X \models LF_\Pi$ iff $X = \mathbf{A}^\mathbf{T} \cap A(\Pi)$ for a (unique) solution $\mathbf{A}$ for $\Theta_{\vec{\Pi}} \cup \Lambda_\Pi \cup \bigcup_{r \in \Pi} \gamma(B(r))$.

Finally, the nogoods in Λ_Π allow us to extend Theorem 4 to non-tight programs.

Theorem 7 *Let Π be a logic program and $X \subseteq \mathcal{A}$.*

Then, X is an answer set of Π iff $X = \mathbf{A}^\mathbf{T} \cap A(\Pi)$ for a (unique) solution $\mathbf{A}$ for $\Delta_\Pi \cup \Theta_{\vec{\Pi}} \cup \Lambda_\Pi$.

Algorithms

Our decision procedure for disjunctive programs is based on the one for normal programs presented in (Gebser *et al.* 2007b). But in contrast to normal programs, the problem of deciding whether a disjunctive program has an answer set is Σ_2^P-complete (Eiter & Gottlob 1995). The source of this complexity increase is recognizing (the absence of) unfounded sets, which is $coNP$-complete in general (Leone, Rullo, & Scarcello 1997). Regarding our framework in the previous section, this means that detecting violations of Λ_Π, using its compact representation by Π itself, may be intractable for certain programs Π. However, for so-called head-cycle-free programs, unfounded set checking is tractable, so that deciding the existence of answer sets drops into NP (Ben-Eliyahu & Dechter 1994).

Our algorithm exploits head-cycle-freeness as well as the fact that the consideration of unfounded sets can safely be restricted to loops (cf. (Lee 2005)). To this end, we partition the atoms of a given program Π into *components* via $C_\Pi = \{C_1, \ldots, C_j\}$, where C_i is a $\subseteq$-maximal element of $loop(\Pi)$ that is not a singleton for $1 \leq i \leq j$. It is not hard to check that $(K \cup L) \in loop(\Pi)$ if $K, L \in loop(\Pi)$ and $K \cap L \neq \emptyset$. Hence, the components in C_Π are mutually disjoint. Furthermore, by restricting attention to non-singletons, C_Π focuses on atoms belonging to loops that are not already handled by $\Theta_{\vec{\Pi}}$. In order to access the component of some atom a, let $C_\Pi(a) = C$, if $C \in C_\Pi$ such that $a \in C$. We say that a component $C \in C_\Pi$ is *head-cycle-free* (HCF), if for every $r \in \Pi$, we have $|H(r) \cap C| \leq 1$. Finally, we denote the set of HCF components in C_Π by $C_\Pi^\oplus$, and let $C_\Pi^\vee = C_\Pi \setminus C_\Pi^\oplus$ be the set of non-HCF components in C_Π.

Before we provide the details of our algorithms, let us briefly outline the context. Our algorithmic approach is based on Conflict-Driven Clause Learning (CDCL) for SAT (Mitchell 2005), having conflict analysis at its core. Here, the First-UIP scheme (Marques-Silva & Sakallah 1999; Zhang *et al.* 2001), enabling backjumping and conflict-driven learning, has turned into a quasi-standard. These techniques are adopted by our algorithms, but in the more abstract setting of nogoods. In fact, every clause describes a nogood consisting of the complements of literals in the clause. Conversely, every nogood can be syntactically represented by a clause, but other representations are also possible. A prominent example in ASP are cardinality and weight constraints (Simons, Niemelä, & Soininen 2002), compactly representing a number of nogoods that can be exponential. In order to also capture such extended constructs, we express the semantics of Boolean constraints via nogoods and switch to the term Conflict-Driven Nogood Learning (CDNL).

Main Algorithm

Algorithm 1 shows our procedure for deciding whether a disjunctive program Π has some answer set. It is inspired by Conflict-Driven Clause Learning (CDCL) for SAT (Mitchell 2005) and our previous algorithm for normal programs (Gebser *et al.* 2007b). In fact, our procedure is centered around conflict-driven learning. This is reflected by the dynamic nogoods in ∇, initialized in Line 2 of Algo-

Algorithm 1: CDNL-ASP-D

Input : A program Π.
Output: An answer set of Π.

```
1  A ← ∅                                   // assigment over A(Π) ∪ B(Π) ∪ B(Π⃗)
2  ∇ ← ∅                                   // set of dynamic nogoods
3  dl ← 0                                  // decision level
4  loop
5      (A, ∇) ← PROPAGATION(Π, ∇, A)
6      if δ ⊆ A for some δ ∈ Δ_Π ∪ Θ_Π⃗ ∪ ∇ then
7          if dl = 0 then return no answer set
8          (ε, k) ← ANALYSIS(δ, Π, ∇, A)
9          ∇ ← ∇ ∪ {ε}
10         A ← A \ {σ ∈ A | k < dl(σ)}
11         dl ← k
12     else if A^T ∪ A^F = A(Π) ∪ B(Π) ∪ B(Π⃗) then
13         U ← ∅                           // unfounded set
14         foreach C ∈ C^∨_Π do
15             if U = ∅ then U ← CDNL(Γ^A_Π(C))
16         if U ≠ ∅ then
17             let δ ∈ λ_Π(U) such that δ ⊆ A in
18                 if {σ_δ ∈ δ | 0 < dl(σ_δ)} = ∅ then
19                     return no answer set
20                 (ε, k) ← ANALYSIS(δ, Π, ∇, A)
21                 ∇ ← ∇ ∪ {ε}
22                 A ← A \ {σ ∈ A | k < dl(σ)}
23                 dl ← k
24         else return A^T ∩ A(Π)
25     else
26         σ_d ← SELECT(Π, ∇, A)
27         dl ← dl + 1
28         A ← A ∘ (σ_d)
```

rithm 1 and filled with learned nogoods in Line 5, 9, and 21. Note that the dynamic nogoods added to ∇ in Line 5 are elements of Λ_Π, while those added in Line 9 and 21 result from conflict analysis (see below) invoked in Line 8 or 20, respectively. In addition to conflict-driven learning, our procedure performs backjumping (Lines 10–11 and 22–23), guided by a decision level k that is determined by conflict analysis.

The purpose of decision levels is to count (Line 27) the number of decision literals, i.e., literals heuristically selected in Line 26[2] and added to assignment $\mathbf{A}$ in Line 28, being present in $\mathbf{A}$. Initially, assignment $\mathbf{A}$ is empty (Line 1), and thus the decision level dl is zero (Line 3). For any literal σ present in $\mathbf{A}$, we write $dl(\sigma)$ to refer to the decision level at which σ has been added to $\mathbf{A}$ (the value dl had at that point). A conflict, detected in Line 6 or 17 via some nogood $\delta \subseteq \mathbf{A}$, at decision level zero indicates that Π has no answer set, in which case our procedure terminates (Line 7 or 18–19). Note that, before selecting any decision literal, propagation (see below) is performed in Line 5. For the result, one of the following three major cases applies: some known nogood $\delta \in \Delta_\Pi \cup \Theta_{\vec{\Pi}} \cup \nabla$ is violated (Lines 6–11); assignment $\mathbf{A}$ is total, that is, $\mathbf{A}$ assigns either $\mathbf{T}$ or $\mathbf{F}$ to each element of $A(\Pi) \cup B(\Pi) \cup B(\vec{\Pi})$ (Lines 12–24); or the obtained assignment $\mathbf{A}$ is partial (Lines 25–28).

Let us focus on the case of a total assignment $\mathbf{A}$, which is specific to disjunctive programs. In fact, propagation includes a polynomial check for unfounded sets, which in the case of normal (or HCF) programs allows us to simply return $\mathbf{A}^\mathbf{T} \cap A(\Pi)$ as an answer set of Π. The same is done in Line 24, but only after performing exponential (in the worst case) unfounded set checks on the non-HCF components $C \in C^\vee_\Pi$, iterated over in Lines 14–15. For each such C, a nonempty unfounded set contained in $C \cap \mathbf{A}^\mathbf{T}$ is a solution to a separate search problem given by the following nogoods:

$$\Gamma^\mathbf{A}_\Pi(C) = \big\{\{\mathbf{T}a \mid a \in H(r) \cap \mathbf{A}^\mathbf{T}\} \cup \{\mathbf{F}a \mid a \in B(r) \cap C\} \mid r \in \Pi, sat_r(C) \cap \mathbf{A} = \emptyset\big\} \cup \big\{\{\mathbf{F}a \mid a \in C \cap \mathbf{A}^\mathbf{T}\}\big\}\,.$$

For a solution U for $\Gamma^\mathbf{A}_\Pi(C)$, represented by the atoms assigned $\mathbf{T}$, and each $r \in \Pi$ such that $H(r) \cap \mathbf{A}^\mathbf{T} \subseteq C$ and $B(r) \notin \mathbf{A}^\mathbf{F}$, the nogoods in $\Gamma^\mathbf{A}_\Pi(C)$ stipulate that either r is satisfied independently from U, i.e., $(H(r) \cap \mathbf{A}^\mathbf{T}) \setminus U \neq \emptyset$, or r depends on U, i.e., $B(r) \cap U \neq \emptyset$. Also note that all rules satisfied independently from C, i.e., rules r where $sat_r(C) \cap \mathbf{A} \neq \emptyset$, do not contribute any nogoods to $\Gamma^\mathbf{A}_\Pi(C)$.

For illustration, consider the following program Π:

$$\begin{Bmatrix} a; b \leftarrow & a; c; e \leftarrow b & c \leftarrow d, {\sim}b & d \leftarrow e, {\sim}a \\ c; d \leftarrow & b; d \leftarrow c & c \leftarrow e & e \leftarrow c, d \end{Bmatrix}. \quad (2)$$

Observe that the only component of Π is $C = \{b, c, d, e\}$, which is non-HCF. Taking a total assignment $\mathbf{A}$ such that $\mathbf{A}^\mathbf{T} \cap A(\Pi) = \{a, c, d, e\}$ and $\mathbf{A}^\mathbf{F} \cap A(\Pi) = \{b\}$, we get:

$$\Gamma^\mathbf{A}_\Pi(C) = \big\{\{\mathbf{T}c, \mathbf{F}d\}, \{\mathbf{T}c, \mathbf{T}d\}, \{\mathbf{T}d, \mathbf{F}c\}, \{\mathbf{T}c, \mathbf{F}e\}, \{\mathbf{T}e, \mathbf{F}c, \mathbf{F}d\}\big\} \cup \big\{\{\mathbf{F}c, \mathbf{F}d, \mathbf{F}e\}\big\}\,.$$

Note that, from the first line in (2), only rule $c \leftarrow d, {\sim}b$ contributes nogood $\{\mathbf{T}c, \mathbf{F}d\}$ to $\Gamma^\mathbf{A}_\Pi(C)$, while the other three rules r are already satisfied because either $a \in H(r) \cap \mathbf{A}^\mathbf{T}$ or ${\sim}a \in B(r)$ implying $B(r) \in \mathbf{A}^\mathbf{F}$. As one can check, there is no solution for $\Gamma^\mathbf{A}_\Pi(C)$, meaning that $C \cap \mathbf{A}^\mathbf{T} = \{c, d, e\}$ does not contain any nonempty unfounded set. Indeed, we have that $\mathbf{A}^\mathbf{T} \cap A(\Pi) = \{a, c, d, e\}$ is an answer set of Π.

The orthogonal search problem specified via $\Gamma^\mathbf{A}_\Pi(C)$ can be solved externally to our main algorithm. In Line 15, we assume that the true atoms of a solution are returned, if there is some solution, or the empty set, if $\Gamma^\mathbf{A}_\Pi(C)$ is unsatisfiable. In the former case, due to the construction of $\Gamma^\mathbf{A}_\Pi(C)$, we know that $\mathbf{A}$ contains some nogood $\delta \in \lambda_\Pi(U)$, which is determined in Line 17 and used for conflict analysis in Line 20. In the latter case, if $U = \emptyset$, we proceed with the next component in $C^\vee_\Pi$. Only after all non-HCF components have been processed and no nonempty unfounded set has been found, the true atoms in $\mathbf{A}$ are returned as an answer set of Π (Line 24). Finally, note that restricting unfounded set checks to non-HCF components is justified by the fact that the loop formula of some loop of Π is violated if $\mathbf{A}$ does not correspond to an answer set of Π. In addition, such a loop cannot belong to a HCF component, which unlike non-HCF components are thoroughly dealt with by the polynomial unfounded set check within propagation.

[2] We assume $\{\sigma_d, \overline{\sigma_d}\} \cap \mathbf{A} = \emptyset$ for literals σ_d selected in Line 26.

Algorithm 2: PROPAGATION

Input : A program Π, a set ∇ of nogoods, and an assignment $\mathbf{A}$.
Output: An extended assignment and set of nogoods.

```
1  U ← ∅                                                  // unfounded set
2  loop
3      repeat
4          if δ ⊆ A for some δ ∈ Δ_Π ∪ Θ_Π⃗ ∪ ∇ then
5              return (A, ∇)
6          Σ ← {δ ∈ Δ_Π ∪ Θ_Π⃗ ∪ ∇ | δ \ A = {σ}, σ̄ ∉ A}
7          if Σ ≠ ∅ then let σ ∈ δ \ A for some δ ∈ Σ in
8              A ← A ∘ (σ̄)
9      until Σ = ∅
10     if C_Π = ∅ then return (A, ∇)
11     U ← U \ A^F
12     if U = ∅ then U ← UnfoundedSet(Π, A)
13     if U = ∅ then return (A, ∇)
14     let a ∈ U and δ ∈ λ_Π(U) such that δ \ {Ta} ⊆ A
       in
15         ∇ ← ∇ ∪ {δ}
```

Propagation

We now explain our propagation procedure shown in Algorithm 2. Lines 3–9 amount to unit propagation (cf. (Mitchell 2005)) on the nogoods in $\Delta_\Pi \cup \Theta_{\vec{\Pi}} \cup \nabla$, either resulting in a conflict (Lines 4–5) or a fixpoint $\mathbf{A}$. In the latter case, if $C_\Pi = \emptyset$, we have that Π is tight, and according to Theorem 4, sophisticated unfounded set checks are unnecessary (Line 10). Otherwise, we proceed with the consideration of unfounded sets. As argued in (Gebser *et al.* 2007b), falsifying a single atom in an unfounded set might enable unit propagation to falsify further atoms of the set. Once a nonempty unfounded set U has been identified, we thus falsify its elements one by one, doing unit propagation inbetween. On this account, we remove false atoms from U in Line 11. If the resulting set is empty, in Line 12, we look for another nonempty unfounded set (see below). Provided that such a U has been determined, for each $a \in U$, there is some nogood $\delta \in \lambda_\Pi(U)$ such that either $\delta \setminus \mathbf{A} = \{\mathbf{T}a\}$, i.e., δ implies $\mathbf{F}a$, or $\delta \subseteq \mathbf{A}$, i.e., δ is violated by $\mathbf{A}$. Such a δ is determined in Line 14 and recorded in Line 15, triggering either unit propagation or conflict analysis.

(Polynomial) Unfounded Set Detection

Our unfounded set detection procedure in Algorithm 3 uses *source pointers* (Simons, Niemelä, & Soininen 2002), indicating for each non-false atom a justifying rule. We denote the source pointer of an atom a by $source(a)$. For a program Π, we require as invariants that $a \in H(source(a))$, for each $a \in (\bigcup_{C \in C_\Pi} C)$, and that the graph $(\bigcup_{C \in C_\Pi} C, \{(a,b) \mid a \in (\bigcup_{C \in C_\Pi} C), source(a) = r, b \in B(r) \cap C_\Pi(a)\})$ is acyclic. After an appropriate initialization, these invariants are maintained by Algorithm 3. Furthermore, we use the following notation to access the prefix

Algorithm 3: UNFOUNDEDSET

Input : A program Π and an assignment $\mathbf{A}$.
Output: An unfounded set of Π wrt $\mathbf{A}$.

```
   S ← {a ∈ (⋃_{C∈C_Π} C) \ A^F | B(source(a)) ∈ A^F or
1        H(source(a)) ∩ ((A^T \ C_Π(a)) ∪ A[Ta]^T) ≠ ∅}
2  repeat
       T ← {a ∈ (⋃_{C∈C_Π} C) \ (A^F ∪ S) |
3            B(source(a)) ∩ C_Π(a) ∩ S ≠ ∅}
4      S ← S ∪ T
5  until T = ∅
6  while S ≠ ∅ do let a ∈ S in
7      U ← {a}
8      repeat
9          R ← {r ∈ sup_Π(U) | sat_r(U) ∩ A = ∅}
10         if R = ∅ then return U
11         let r ∈ R in
12             if B(r) ∩ C_Π(a) ∩ S = ∅ then
                   T ← {a' ∈ (H(r) ∩ U) |
13                          H(r) ∩ A[Ta']^T = ∅}
14                 foreach a' ∈ T do source(a') ← r
15                 S ← S \ T
16                 U ← U \ T
17             else U ← U ∪ (B(r) ∩ C_Π(a) ∩ S)
18     until U = ∅
19 return ∅
```

of an assignment $\mathbf{A}$ up to a literal σ:

$$\mathbf{A}[\sigma] = \begin{cases} (\sigma_1, \ldots, \sigma_m) & \text{if } \mathbf{A} = (\sigma_1, \ldots, \sigma_m, \sigma, \ldots) \\ \mathbf{A} & \text{if } \sigma \notin \mathbf{A}\,. \end{cases}$$

If a rule head contains several atoms of the same component that are true wrt $\mathbf{A}$, the prefix allows us to identify the atom that became true first. Only this atom can potentially use the rule as its source pointer, while the other atoms may not. Note that it would be incorrect to completely exclude disjunctive rules with several true head atoms as source pointers, as it would lead to speciously reckon sets as unfounded.

The general purpose of source pointers is to designate the scope of unfounded set checks in reaction to changes of $\mathbf{A}$. In fact, in Line 1 of Algorithm 3, we determine all non-false atoms a, belonging to some component in C_Π, whose source pointer has recently been invalidated, either because $B(source(a))$ has become false or because some atom in $H(source(a)) \setminus \{a\}$ has become true. As mentioned above, if all true head atoms belong to the same component, then the first atom that became true may still use the respective rule as its source pointer. After determining the atoms with invalidated source pointers, in Lines 2–5, we iteratively collect further atoms whose source pointers depend on them. The resulting set S provides the scope for the second part of Algorithm 3 in Lines 6–18, trying to reestablish the source pointers of the atoms in S. To this end, we investigate a (nonempty) subset U of S, starting from some $a \in S$, together with the rules that externally support U and are not satisfied independently from U (Line 9). If no such rule ex-

ists, then U is unfounded, and we immediately return it in Line 10. Otherwise, we check for some such rule r whether $B(r)$ contains atoms belonging both to $C_\Pi(a)$ and to the scope S (Line 12). If so, these atoms are added to U in Line 17, achieving that r does not externally support the resulting set U. In contrast, if no atom from $B(r)$ can be added to U, then r is eligible as new source pointer for some atoms in U, where we again distinguish the first true atom from U in $H(r)$ in case that $H(r)$ contains true atoms (Line 13). In Lines 14–16, we eventually reestablish source pointers and remove the corresponding atoms both from the scope S as well as the unfounded set U to be constructed.

Reconsider the program Π in (2), non-HCF component $C = \{b, c, d, e\}$, and an assignment $\mathbf{A}$ such that $\mathbf{A}^\mathbf{T} \cap A(\Pi) = \{a, c, d, e\}$ and $\mathbf{A}^\mathbf{F} \cap A(\Pi) = \{b\}$. Observe that, for every rule $r \in \Pi$ such that $H(r) \cap (C \setminus \mathbf{A}^\mathbf{F}) \neq \emptyset$, where $C \setminus \mathbf{A}^\mathbf{F} = \{c, d, e\}$, we have either $B(r) \cap (C \setminus \mathbf{A}^\mathbf{F}) \neq \emptyset$ or $|H(r) \cap \mathbf{A}^\mathbf{T}| > 1$. In particular, the latter applies to $c; d \leftarrow$. If we did not permit $c; d \leftarrow$ as the source pointer of either c or d, then the acyclic character of source pointers would imply that the atoms in $\{c, d, e\}$ get into the scope of Algorithm 3 at some point and would then wrongly be returned as an unfounded set, making us discard answer set $\{a, c, d, e\}$ of Π. For another example, consider an assignment $\mathbf{A}$ such that $\mathbf{A}^\mathbf{T} \cap A(\Pi) = \{b, c, d, e\} = C$. Then, if $\mathbf{T}b$ and $\mathbf{T}c$ precede $\mathbf{T}d$ and $\mathbf{T}e$ in $\mathbf{A}$, only rules $d \leftarrow e, \sim a$ and $e \leftarrow c, d$ can potentially be used as source pointers of d or e, respectively. However, since these rules are cyclic, Algorithm 3 cannot reset the source pointers of d and e, thus identifying unfounded set $\{d, e\}$. In contrast, if $\mathbf{T}e$ is first in $\mathbf{A}$, then the source pointer of e can be set to $a; c; e \leftarrow b$ (as $a; b \leftarrow$ can be used for b), which afterwards allows Algorithm 3 to set the source pointers of c and d to $c \leftarrow e$ and $d \leftarrow e, \sim a$, respectively. Hence, unfounded set $\{d, e\}$ is not detected.

Note that Algorithm 3 is complete for HCF components, C, in the sense that it detects a nonempty unfounded set $U \subseteq C \setminus \mathbf{A}^\mathbf{F}$ if there is one. For non-HCF components, Algorithm 3 can only approximate unfounded sets. In fact, any determined set U is unfounded, but we are not guaranteed to detect every unfounded set that may exist. Allowing the first true atom in a disjunctive head to use the corresponding rule as its source pointer is not an exact criterion, and other choices may lead to different outcomes. However, the lack of exactness is compensated by the computational cost: While Algorithm 3 has linear complexity,[3] deciding whether a nonempty unfounded set exists is NP-complete for non-HCF components (Leone, Rullo, & Scarcello 1997).

Conflict Analysis

Finally, our conflict analysis procedure in Algorithm 4 resolves a violated nogood δ against other nogoods implying some of its literals until encountering a Unique Implication Point (UIP), σ, which has the property that $dl(\sigma_\delta) < dl(\sigma)$ for all $\sigma_\delta \in \delta \setminus \{\sigma\}$. To this end, we pick in Line 2 the literal $\sigma \in \delta$ that has been added last to $\mathbf{A}$. If σ is not a UIP, we resolve δ against a nogood ε implying σ (Line 6). Note

[3]Within *claspD*, described below, we even retain the scope S for repeated calls to UNFOUNDEDSET at the same decision level.

Algorithm 4: ANALYSIS

Input : A violated nogood δ, a program Π, a set ∇ of nogoods, and an assignment $\mathbf{A}$.
Output: A derived nogood and a decision level.

```
1 loop
2   let σ ∈ δ such that δ \ A[σ] = {σ} in
3     k ← max({dl(σ_δ) | σ_δ ∈ δ \ {σ}} ∪ {0})
4     if k = dl(σ) then
5       let ε∈Δ_Π∪Θ_Π⃗∪∇ such that ε\A[σ]={σ̄} in
6         δ ← (δ \ {σ}) ∪ (ε \ {σ̄})
7     else return (δ, k)
```

that such a nogood ε always exists because σ is necessarily different from the decision literal of $dl(\sigma)$. Otherwise, if σ is a UIP, we are done with conflict analysis (Line 7). The particularity of a UIP σ is that the corresponding nogood δ implies $\overline{\sigma}$ at the maximum decision level k of literals in $\delta \setminus \{\sigma\}$. Hence, backjumping can return to decision level k in order to afterwards assert $\overline{\sigma}$. By stopping conflict analysis at the UIP σ encountered first, which is not necessarily the decision literal of $dl(\sigma)$, Algorithm 4 is similar to the First-UIP scheme of CDCL (Mitchell 2005).

System

We implemented our approach to disjunctive ASP solving as an extension of the conflict-driven ASP solver *clasp* (Gebser *et al.* 2007b) and call the resulting system *claspD* (claspD). In fact, *claspD* inherits many features from *clasp*, such as conflict-driven learning, lookback-based decision heuristics, restart policies, watched literals, etc. The input language of *claspD* consists of logic programs in *lparse*'s output format (Syrjänen). Like *clasp*, also *claspD* supports answer set enumeration (Gebser *et al.* 2007a) and optimization. It also handles cardinality and weight constraints (Simons, Niemelä, & Soininen 2002), currently through compilation.

The global architecture of *claspD* is shown in Figure 1. Given a logic program Π, the PREPROCESSOR takes care of creating an internal representation, comprising the static nogoods in $\Delta_\Pi \cup \Theta_{\vec{\Pi}}$ as well as the components in C_Π. Notably, preprocessing also includes program simplifications (Eiter *et al.* 2004) and equivalence reasoning (Gebser *et al.* 2008), both adapted to disjunctive programs. The actual search for answer sets can be further distinguished into a generating part, providing answer set candidates, and a testing part, verifying the provided candidates. Since both of these tasks can be computationally complex, they are performed by associated inference engines (indicated by CDNL in Figure 1), implemented in *claspD* by feeding the core search module from *clasp* with particular Boolean constraints. While the generator traverses the search space for answer sets of Π, communicating its current state through an assignment to the tester, the latter checks for unfounded sets and reports them back via nogoods of Λ_Π. As shown in Algorithm 2, procedure UNFOUNDEDSET is integrated into propagation and thus continuously applied during the

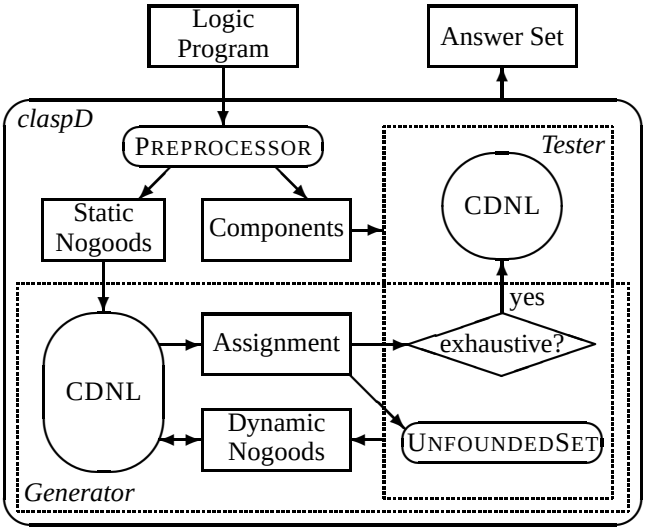

Figure 1: The system architecture of *claspD*.

generation of answer set candidates. In contrast, the checks encoded by $\Gamma^{\mathbf{A}}_{\Pi}(C)$, for non-HCF components $C \in C^{\vee}_{\Pi}$, are performed only selectively, e.g., if assignment $\mathbf{A}$ is total, due to their high computational cost. Having sketched the overall architecture, the remainder of this section outlines particular features of *claspD* related to degrees of freedom in the algorithms specified above.

As mentioned below Theorem 4, the models of a program Π (or RF_{Π}, respectively) can also be captured by $\Delta_{\vec{\Pi}}$ as substitute for Δ_{Π}. On the one hand, shifting potentially increases the number of nogoods used to express models because $|\Pi| \leq |\vec{\Pi}|$. On the other hand, the propositional variables occurring in $\Delta_{\vec{\Pi}}$ are exactly the same as those in $\Theta_{\vec{\Pi}}$, while Δ_{Π} introduces extra variables for bodies in $B(\Pi) \setminus B(\vec{\Pi})$. With *claspD*, the set of nogoods to use can be selected via command line options. By default, *claspD* encodes models via $\Delta_{\vec{\Pi}}$ in order to maximize the reuse of variables, particularly, in learned nogoods. Another representation-related issue is checking whether $sat_r(U) \cap \mathbf{A} = \emptyset$ holds in Line 9 of Algorithm 3, as it requires the investigation of $B(r)$ and the atoms in $H(r) \setminus U$. For reducing the number of the latter, *claspD* performs a "component-wise shifting." That is, for each $C \in C_{\Pi}$ and $r \in \Pi$ such that $H(r) \cap C \neq \emptyset$, we introduce a variable $B = B(r) \cup \{\sim a \mid a \in H(r) \setminus C\}$ along with the nogoods in $\gamma(B)$. This allows us to check $sat_r(U) \cap \mathbf{A} = \emptyset$ by investigating just B and the atoms in $(H(r) \cap C) \setminus U$. Note that, if C is HCF, we have $|H(r) \cap C| = 1$, which implies that B is already present in $\Theta_{\vec{\Pi}}$ and can thus be reused. During preprocessing, *claspD* introduces a variable standing for a conjunction (being the body of either an original or a shifted rule) only once and reuses it if the same conjunction recurs.

Even with component-wise shifting, for non-HCF components $C \in C^{\vee}_{\Pi}$, *claspD* may detect some nonempty unfounded set $U \subset C$ such that there is a rule $r \in sup_{\Pi}(U)$ for which $|\Sigma_r = \big(\big(\{\mathbf{F}(B(r) \cup \{\sim a \mid a \in H(r) \setminus C\})\} \cup \{\mathbf{T}a \mid a \in (H(r) \cap C) \setminus U\}\big) \cap \mathbf{A}\big)| > 1$, where $\mathbf{A}$ is the current assignment. Since in this case we can choose among the literals in Σ_r, several nogoods can be extracted and used for falsifying an atom in U by unit propagation. However, *claspD* does not aim at the exhaustive recording of such nogoods, as it would be memory-consuming and might even slow down the solving process as a whole. On this account, *claspD* selects the literal $\sigma \in \Sigma_r$ added first to $\mathbf{A}$, i.e., $\Sigma_r \cap \mathbf{A}[\sigma] = \emptyset$, in order to construct a single nogood δ to be recorded. The underlying idea is that, if δ is resolved within conflict analysis, then literals of smaller decision levels are likely to permit longer backjumps. The described heuristics is applicable (and applied by *claspD*) to unfounded sets $U \subset C$, for some $C \in C^{\vee}_{\Pi}$, determined either via the polynomial unfounded set check in Algorithm 3 or as a solution for $\Gamma^{\mathbf{A}}_{\Pi}(C)$. Beyond that, once a nonempty unfounded set $U \subset C$ has been determined, where $C \in C^{\vee}_{\Pi}$, there possibly are atoms $a \in C \setminus (U \cup \mathbf{A}^{\mathbf{F}})$ such that $U \cup \{a\}$ is also an unfounded set. Furthermore, we might have $a \in B(r)$ for some rule $r \in sup_{\Pi}(U)$, so that $r \notin sup_{\Pi}(U \cup \{a\})$. Following this observation, it can happen that $|sup_{\Pi}(U \cup \{a\})| < |sup_{\Pi}(U)|$ and that the nogood to be recorded for $U \cup \{a\}$ turns out to be smaller (in terms of the number of literals) than the one for U. For unfounded sets U determined as solutions for $\Gamma^{\mathbf{A}}_{\Pi}(C)$, we exploit this idea and greedily add atoms $a \in C \setminus (U \cup \mathbf{A}^{\mathbf{F}})$ to U, provided that $U \cup \{a\}$ stays unfounded and that the nogood already constructed for U becomes smaller due to the addition of a. This strategy is guided by the assumption that smaller nogoods constrain the search space stronger and by the consideration that the effort made to identify U justifies the overhead of a posteriori reducing its associated nogood.

A related feature of *claspD* goes back to ideas from (Janhunen *et al.* 2006; Pfeifer 2004), where exhaustive unfounded set checks are repeated during backtracking and are thus not limited to total assignments, as it is the case in Algorithm 1. On the one hand, such strategies make sure that backtracking proceeds to a state such that no nogood in Λ_{Π} is violated, which is not guaranteed in Algorithm 1 because it computes an arbitrary nonempty unfounded set in Line 15. On the other hand, by integrating exhaustive unfounded set checks into backtracking, they are applied on demand and in a controlled way, while it would be harder to predict their usefulness during the generation of an answer set candidate. In the implementation of *claspD*, the loop in Lines 14–15 of Algorithm 1 is repeated after backjumping because of an invalid answer set candidate. Since assignments $\mathbf{A}$ can be partial in such situations, for a non-HCF component $C \in C^{\vee}_{\Pi}$, encoding $\Gamma^{\mathbf{A}}_{\Pi}(C)$ is modified as follows:

$$\begin{aligned}\Upsilon^{\mathbf{A}}_{\Pi}(C) = \big\{\{\mathbf{T}a \mid a \in H(r) \cap \mathbf{A}^{\mathbf{T}}\} \cup \{\mathbf{F}a \mid a \in B(r) \cap C\} \mid \\ r \in \Pi, sat_r(C) \cap \mathbf{A} = \emptyset, H(r) \cap \mathbf{A}^{\mathbf{T}} \neq \emptyset\big\} \\ \cup \big\{\{\mathbf{T}a\} \cup \{\mathbf{F}a \mid a \in B(r) \cap C\} \mid \\ r \in \Pi, sat_r(C \setminus \mathbf{A}^{\mathbf{T}}) \cap \mathbf{A} = \emptyset, a \in (H(r) \cap C) \setminus \mathbf{A}^{\mathbf{F}}\big\} \\ \cup \big\{\{\mathbf{F}a \mid a \in C \cap \mathbf{A}^{\mathbf{T}}\}\big\}\,.\end{aligned}$$

Note that the first set of nogoods in $\Upsilon^{\mathbf{A}}_{\Pi}(C)$ is almost similar to the corresponding set in $\Gamma^{\mathbf{A}}_{\Pi}(C)$, but since $\mathbf{A}$ might be partial, we explicitly require that $H(r) \cap \mathbf{A}^{\mathbf{T}} \neq \emptyset$. The second set of nogoods is used to encode rules r not having

No.	Class	n	*claspD*	*claspD*$_{ns}$	*claspD*$_{na}$	*claspD*$_{nr}$	*claspD*$_{np}$	*claspD*$_{nl}$	*cmodels*	*dlv*	*gnt*
1	SCore-Tight	39	9.57 (17)	9.13 (16)	10.03 (16)	8.80 (17)	9.83 (16)	32.49 (12)	34.58 (20)	75.45 (48)	40.03 (22)
			95.36	89.93	90.71	94.70	90.54	90.70	131.23	290.06	145.32
2	SCore-NonTight	56	10.74 (0)	11.26 (0)	11.16 (0)	10.95 (0)	10.78 (0)	29.32 (13)	17.62 (12)	78.62 (23)	42.18 (23)
			10.74	11.26	11.16	10.95	10.78	73.48	59.22	150.00	118.55
3$^{\mathbf{r}}$	DLV-HCF-Hamilton	100	2.36 (2)	1.94 (0)	2.36 (2)	2.36 (2)	1.53 (3)	24.59 (108)	10.64 (2)	0.04 (0)	16.69 (67)
			6.34	1.94	6.34	6.34	7.51	231.74	14.57	0.04	149.29
4$^{\mathbf{s}}$	DLV-HCF-Sokoban	118	1.23 (0)	1.02 (0)	1.24 (0)	1.23 (0)	1.24 (0)	4.64 (9)	0.20 (6)	2.40 (9)	5.13 (17)
			1.23	1.02	1.24	1.23	1.24	19.78	10.37	17.59	33.70
5$^{\mathbf{r}}$	DLV-QBF.cgs	100	0.07 (0)	0.21 (0)	0.08 (0)	0.07 (0)	0.05 (0)	0.00 (6)	0.26 (0)	3.23 (15)	51.76 (269)
			0.07	0.21	0.08	0.07	0.05	12.00	0.26	33.07	543.35
6$^{\mathbf{r}}$	DLV-QBF.gw	100	0.00 (0)	0.00 (0)	0.00 (0)	0.00 (0)	0.01 (0)	3.52 (0)	0.01 (0)	0.01 (0)	50.10 (85)
			0.00	0.00	0.00	0.00	0.01	3.52	0.01	0.01	205.91
7	SCore-Mutex	7	53.42 (0)	53.38 (0)	52.86 (0)	55.26 (0)	17.79 (15)	46.91 (0)	0.42 (18)	14.15 (0)	1.64 (18)
			53.42	53.38	52.86	55.26	433.65	46.91	514.35	14.15	514.52
8$^{\mathbf{r}}$	SCore-RandomQBF	15	9.38 (0)	9.39 (0)	9.33 (0)	8.54 (0)	304.27 (1)	9.31 (0)	101.73 (29)	15.55 (0)	— (45)
			9.38	9.39	9.33	8.54	310.84	9.31	422.84	15.55	600.00
9$^{\mathbf{r}}$	SCore-StratComp	15	53.16 (2)	70.04 (2)	47.44 (0)	53.16 (2)	52.56 (2)	136.16 (21)	93.67 (1)	43.32 (2)	84.27 (6)
			77.46	93.59	47.44	77.46	76.89	352.62	104.92	68.06	153.03
Average Time (Sum Timeouts)			15.55 (21)	17.37 (18)	14.94 (18)	15.60 (21)	44.23 (37)	31.88 (169)	28.79 (88)	25.86 (97)	36.48 (552)
Average Penalized Time			28.22	28.97	24.35	28.28	103.50	93.34	139.75	65.39	273.74

Table 1: Experiments computing one answer set on a 3.4GHz PC under Linux; each run limited to 600s time and 1GB RAM.

a true head atom, where one nogood containing literal $\mathbf{T}a$ is included per atom $a \in (H(r) \cap C) \setminus \mathbf{A^F}$. By also taking unassigned atoms into account, we allow for solutions $U \not\subseteq \mathbf{A^T}$, in which case $U \cap \mathbf{A^T} \neq \emptyset$ is implicitly granted since *claspD* applies UNFOUNDEDSET within Algorithm 2 before solving $\Upsilon^{\mathbf{A}}_{\Pi}(C)$. Similar to $\Gamma^{\mathbf{A}}_{\Pi}(C)$, the third nogood thus excludes the empty (unfounded) set as a solution. Another strategy, called "Partial Checks Forwards" in (Pfeifer 2004), consists of incorporating an exhaustive unfounded set check into the generation of a new answer set candidate after the previous one has turned out to be invalid. Based on $\Upsilon^{\mathbf{A}}_{\Pi}(C)$, we also adopted this technique in *claspD*.

Experiments

We conducted experiments on a variety of benchmarks, stemming from the *dlv* team[4] and from the normal (*SCore*) and disjunctive (*SCore*$^{\vee}$) solver categories of the ASP system competition[5]. Instances from the normal *SCore* category are divided into classes No. 1 and 2 in Table 1, depending on tightness. Both subclasses contain randomly generated as well as structured instances. For the other classes (No. 3–9), we indicate by a superscript $^{\mathbf{r}}$ or $^{\mathbf{s}}$ whether they consist of randomly generated or structured instances, respectively. Class No. 7 contains (unsatisfiable) instances, also used in (Faber *et al.* 2007), that have been obtained from 2QBFs whose particular nature is unknown to us. Note that class No. 1 consists of tight, classes No. 2–4 of non-tight HCF, and classes No. 5–9 of (non-tight) non-HCF programs.

Our study considers *claspD* in six settings: default configuration (*claspD*); no shifting for nogoods encoding model conditions (*claspD*$_{ns}$); no polynomial unfounded set approximation within non-HCF components (*claspD*$_{na}$); no reduction of loop nogoods violated by total assignments (*claspD*$_{nr}$); no exhaustive unfounded set checks on partial assignments (*claspD*$_{np}$); neither learning nor backjumping, instead using lookahead (*claspD*$_{nl}$). For comparison, we also incorporate the disjunctive ASP solvers *cmodels* (3.68), *dlv* (Oct 11 2007), and *gnt* (2.1). Table 1 summarizes runtime results in seconds, excluding times spent by *lparse* (Syrjänen) and subtracting times of *dlv* with option "instantiate" from run-times of *dlv*. Each line averages over $3n$ runs on n benchmark instances, each shuffled three times using ASP tools from TU Helsinki[6]. For every benchmark class, the first line gives the average time for completed runs and the number of timeouts in parentheses, whereas the second line provides the average time penalizing timeouts with maximum time, viz., 600 seconds. Similarly, we summarize at the bottom of Table 1 the average run-times over all benchmark classes (weighted equally). So the last but one line gives the average time per solver along with the sum of all timeouts in parentheses, while the last line provides the average time including the aforementioned penalty.

The summary in Table 1 shows that *claspD*, *claspD*$_{ns}$, *claspD*$_{na}$, and *claspD*$_{nr}$ are close to each other, and they outperform the other solvers regarding timeouts. The fact that most of the available instances, in particular, in the non-HCF classes (No. 5–9), are randomly generated could be a reason for the observed indifference; more differentiated benchmark classes and instances are needed for a meaningful comparison. However, on non-HCF classes No. 7 and 8, we observe degrading performance of *claspD*$_{np}$, showing the positive impact of exhaustive unfounded set checks on partial assignments. We note that differing run-times and timeouts among the first five *claspD* variants on HCF classes (No. 1–4) are noise effects, caused by implementation de-

[4] http://www.dlvsystem.com/examples/tocl-dlv.zip

[5] http://asparagus.cs.uni-potsdam.de/contest/

[6] http://www.tcs.hut.fi/Software/asptools/

tails influencing the variable ordering in the data structures of *claspD*. The non-learning variant *claspD*$_{nl}$ overall performs worse than the five learning ones, but it shows surprisingly good performance in terms of timeouts on class No. 1. Here, we verified that all of the observed timeouts occurred on randomly generated instances of the "BlockedNQueens" problem. Among the other solvers, the approach of *cmodels*, using conflict-driven learning SAT solvers, is closest to *claspD*. The fact that *cmodels* does currently not exploit exhaustive unfounded set checks on partial assignments is likely to be the reason for the limited performance on classes No. 7 and 8, where also *claspD*$_{np}$ shows declines. In contrast, *dlv* uses such checks and is successful on classes No. 7 and 8. Also, we observe distinguished performance of *dlv* on class No. 3, consisting of particularly tailored planar graphs (Leone *et al.* 2006), possibly suiting the heuristics of *dlv*. The most problematic classes for *dlv* are No. 1 and 2, used in *SCore*, as well as non-HCF class No. 5. Finally, we observe that *gnt* shows weakest performance overall, which might be somewhat explained by the fact that it deploys *smodels* (Simons, Niemelä, & Soininen 2002).

Related Work

Our approach to disjunctive ASP solving builds upon previous work on normal programs (Gebser *et al.* 2007b). The common idea is to exploit advanced lookback-based techniques from Boolean satisfiability and constraint solving (Mitchell 2005; Dechter 2003) in the context of ASP. Many of these general techniques, e.g., backjumping, conflict-driven learning, decision heuristics, restart policies, and watched literals, are implemented in *clasp* and extended in *claspD* to the disjunctive case. We note that *smodels*$_{cc}$ (Ward & Schlipf 2004) augments *smodels* with similar techniques, and a prototypical extension of *dlv* (Faber *et al.* 2007) implements backjumping along with lookback-based heuristics.

In accord with the computational complexity of disjunctive ASP solving (Eiter & Gottlob 1995), *claspD* deploys a generate and test approach, realizing both tasks through *clasp*'s core technology. Notably, the generating part applies the enumeration technique described in (Gebser *et al.* 2007a) for the repetition-less computation of multiple answer sets, without falling back to solution recording. Furthermore, the basic data structure of *clasp* is that of a Boolean constraint, permitting the native (that is, compilation-less) support of cardinality and weight rules (Simons, Niemelä, & Soininen 2002). Such extended constructs are currently handled in *claspD* through compilation, and their native support is a subject to the future.

Unfounded set checking in *clasp* makes use of source pointers, an implementation technique first applied in *smodels* (Simons, Niemelä, & Soininen 2002). But different from *smodels*, *clasp* does not aim at determining greatest unfounded sets (Van Gelder, Ross, & Schlipf 1991), which might even be non-existent for non-HCF components (Leone, Rullo, & Scarcello 1997). As it does not rely on greatest unfounded sets, the extension of *clasp*'s source pointer technique realized in *claspD* is applicable to both HCF and non-HCF components, though complexity obstructs exactness for the latter. In *dlv* (Leone *et al.* 2006), unfounded set checking is also integrated into propagation but limited to computing greatest unfounded sets within HCF components (Calimeri *et al.* 2006). Instead of source pointers, *dlv* uses a "must-be-true" value to indicate true atoms whose support might be circular. Interestingly, *dlv* may assign true, rather than must-be-true, to the first atom satisfying the head of a rule (Faber 2008). Though in *dlv* it serves a different purpose, this strategy is similar to the one in Algorithm 3, possibly permitting the first true atom of a disjunctive head to use the corresponding rule as its source pointer. In contrast to *claspD* and *dlv*, on non-tight programs, *cmodels* (Lierler 2005) performs sophisticated unfounded set checks only after an answer set candidate has been generated, but not during propagation.

Like *claspD*, the solvers *cmodels*, *dlv*, and *gnt* (Janhunen *et al.* 2006) rely on a generate and test approach. For both tasks, *cmodels* makes use of SAT solvers, typically performing conflict-driven learning according to the First-UIP scheme (Marques-Silva & Sakallah 1999; Zhang *et al.* 2001; Mitchell 2005). To this end, *cmodels* encodes both the generation and the testing problem by CNF formulas, abbreviating conjunctions by propositions in the generating part. In particular, the encoding of program completion (Ben-Eliyahu & Dechter 1994; Lee & Lifschitz 2003) used in *cmodels*, which in parts is based on shifting (Gelfond *et al.* 1991), reuses propositions standing for bodies of rules in the original program (Lierler 2008). For example, a rule $a; b \leftarrow c, {\sim}d$ gives rise to nogoods $\{\mathbf{T}\{c,{\sim}d\}, \mathbf{F}a, \mathbf{F}b\}$ and $\gamma(\{c,{\sim}d\})$ for describing the rule as such, along with $\gamma'(\{c,{\sim}d,{\sim}b\}) = \big\{\{\mathbf{F}\{c,{\sim}d,{\sim}b\}, \mathbf{T}\{c,{\sim}d\}, \mathbf{F}b\}, \{\mathbf{T}\{c,{\sim}d,{\sim}b\}, \mathbf{F}\{c,{\sim}d\}\}, \{\mathbf{T}\{c,{\sim}d,{\sim}b\}, \mathbf{T}b\}\big\}$ used for the completion of a. Observe that the proposition standing for conjunction $\{c,{\sim}d\}$ is reused in the nogoods of $\gamma'(\{c,{\sim}d,{\sim}b\})$, encoding the extended conjunction $\{c,{\sim}d,{\sim}b\}$ obtained by shifting. In this respect, *cmodels* offers an interesting alternative representation of completion, aiming at conciseness.[7]

In the generating part of *dlv*, the nogoods resulting from program completion are not made explicit, rather, *dlv* represents programs directly. However, the nogoods implicitly underlie propagation conditions (Faber 2002), whose technical description is much more involved at the logic program level. While the generating part of standard *dlv* applies systematic backtracking without learning, an experimental version (Ricca, Faber, & Leone 2006) supports backjumping (but not learning), pursuing a strategy that boils down to the Decision scheme (Zhang *et al.* 2001). As with *cmodels*, the testing part of *dlv* is realized via a reduction to SAT (Koch, Leone, & Pfeifer 2003), which is similar to $\Gamma^{\mathbf{A}}_{\Pi}(C)$ used in Algorithm 1. In *gnt*, the generating and testing part are both implemented on top of *smodels*, thus using systematic backtracking without learning, where normal programs express program completion and minimality conditions, respectively. Notably, the completion representations used in *claspD* and *cmodels*, introducing propositions for rule bodies, permit exponential improvements in terms of proof complexity (Beame & Pitassi 1998) in comparison to

[7] For using SAT solvers, *cmodels* represents nogoods by clauses.

purely atom-based approaches, as pursued in *dlv* and *gnt*, already for normal programs (Gebser & Schaub 2006).

In the following, we focus on the test of answer set candidates. In *claspD*, each non-HCF component C is investigated, encoding a nonempty unfounded set contained in C via $\Gamma^{\mathbf{A}}_{\Pi}(C)$, similar to the SAT reduction used in *dlv*. Different from *claspD*, *dlv* first recomputes components relative to an answer set candidate (Koch, Leone, & Pfeifer 2003), in the hope that a non-HCF component collapses into HCF components. Regarding non-HCF components, the strategy of *cmodels* is similar to those of *claspD* and *dlv*, but in contrast to them, *cmodels* also needs to investigate HCF components in order to perform polynomial unfounded set checks (Lierler 2008). The encoding used in *gnt* is rather different from $\Gamma^{\mathbf{A}}_{\Pi}(C)$, as it aims at a model smaller than an answer set candidate, but not explicitly at an unfounded set. If such a smaller model is found, the test is reapplied during backtracking to (possibly) partial answer set candidates until a candidate passes (Janhunen *et al.* 2006). A similar technique is applied in *dlv*, but before the test for a nonempty unfounded set is redone, *dlv* checks whether the previously determined set is still unfounded after backtracking (Pfeifer 2004), in which case backtracking can proceed. In *claspD* and (SAT solvers deployed by) *cmodels*, the loop formula of an unfounded set invalidating an answer set candidate gives rise to conflict-driven learning and backjumping, so that repeatedly checking the unfoundedness of the set is unnecessary. However, after having disposed a nonempty unfounded set, it may be possible to find another one to invalidate a partial answer set candidate. A respective strategy is applied in *dlv* (Pfeifer 2004) and adopted by *claspD*. Finally, we note that our encoding $\Upsilon^{\mathbf{A}}_{\Pi}(C)$ allows us to identify both unfounded sets $U \subseteq \mathbf{A^T}$ and $U \not\subseteq \mathbf{A^T}$, while the approach in (Pfeifer 2004) only admits unfounded sets $U \subseteq \mathbf{A^T}$.

Interestingly, the first among the algorithms for 2QBF solving described in (Ranjan, Tang, & Malik 2004), reported as the most robust of the compared algorithms (some of which applicable to general QBFs and others specialized to 2QBFs), also relies on two core solvers feeding each other with particular problems and assignments resulting from the computed solutions, respectively. For problems located at the second level of the polynomial hierarchy, this indicates that the entangling of two NP-oracles is a promising approach, where details of the entangling are a major subject to optimizations, like the ones suggested in (Pfeifer 2004).

Discussion

We introduced a uniform constraint-based approach to disjunctive ASP solving. This provides us with flexibility in problem representation offering several degrees of freedom, like optional shifting in the encoding of model conditions or component-wise shifting for facilitating unfounded set checking. Moreover, our approach allows us to take advantage of Boolean constraint solving technology without using any legacy ASP or SAT solvers as black boxes. To this end, we developed advanced solving algorithms, featuring conflict-driven learning and backjumping based on the First-UIP scheme as well as an elaborate component-oriented unfounded set checking strategy. As dictated by complexity, the latter is exponential on non-HCF components only, while it stays polynomial on HCF and in an approximative way also on non-HCF components. Notably, our polynomial unfounded set checking technique generalizes source pointers to the disjunctive case. As a result, we implemented a new disjunctive ASP solver, *claspD*, being competitive with current state-of-the-art solvers. Further experiments using realistic, i.e., not randomly generated, instances of Σ^P_2-complete problems are needed to fine-tune *claspD*.

Acknowledgments We would like to thank Wolfgang Faber, Tomi Janhunen, and Yuliya Lierler for helpful comments on an earlier draft of this paper. This work was partially funded by the Federal Ministry of Education and Research within project GoFORSYS.

References

Baral, C.; Brewka, G.; and Schlipf, J., eds. 2007. *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*. Springer-Verlag.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Beame, P., and Pitassi, T. 1998. Propositional proof complexity: Past, present, and future. *Bulletin of the European Association for Theoretical Computer Science* 65:66–89.

Ben-Eliyahu, R., and Dechter, R. 1994. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence* 12(1-2):53–87.

Calimeri, F.; Faber, W.; Pfeifer, G.; and Leone, N. 2006. Pruning operators for disjunctive logic programming systems. *Fundamenta Informaticae* 71(2-3):183–214.

claspD. http://www.cs.uni-potsdam.de/claspD.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann Publishers.

Eiter, T., and Gottlob, G. 1995. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence* 15(3-4):289–323.

Eiter, T.; Fink, M.; Tompits, H.; and Woltran, S. 2004. Simplifying logic programs under uniform and strong equivalence. In Lifschitz and Niemelä (2004), 87–99.

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

Faber, W.; Leone, N.; Maratea, M.; and Ricca, F. 2007. Experimenting with look-back heuristics for hard ASP programs. In Baral et al. (2007), 110–122.

Faber, W. 2002. *Enhancing Efficiency and Expressiveness in Answer Set Programming Systems*. Dissertation.

Faber, W. 2008. Personal communication.

Gebser, M., and Schaub, T. 2006. Tableau calculi for answer set programming. In Etalle, S., and Truszczyński, M., eds., *Proceedings of the Twenty-second International*

Conference on Logic Programming (ICLP'06), 11–25. Springer-Verlag.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007a. Conflict-driven answer set enumeration. In Baral et al. (2007), 136–148.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007b. Conflict-driven answer set solving. In Veloso, M., ed., *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, 386–392. AAAI Press/MIT Press.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2008. Advanced preprocessing for answer set solving. In *Proceedings of the Eighteenth European Conference on Artificial Intelligence (ECAI'08)*. IOS Press.

Gebser, M.; Lee, J.; and Lierler, Y. 2006. Elementary sets for logic programs. In Gil, Y., and Mooney, R., eds., *Proceedings of the Twenty-first National Conference on Artificial Intelligence (AAAI'06)*. AAAI Press.

Gebser, M.; Schaub, T.; and Thiele, S. 2007. GrinGo: A new grounder for answer set programming. In Baral et al. (2007), 266–271.

Gelfond, M.; Lifschitz, V.; Przymusinska, H.; and Truszczyński, M. 1991. Disjunctive defaults. In Allen, J.; Fikes, R.; and Sandewall, E., eds., *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, 230–237. Morgan Kaufmann Publishers.

Janhunen, T.; Niemelä, I.; Seipel, D.; Simons, P.; and You, J. 2006. Unfolding partiality and disjunctions in stable model semantics. *ACM Transactions on Computational Logic* 7(1):1–37.

Koch, C.; Leone, N.; and Pfeifer, G. 2003. Enhancing disjunctive logic programming systems by SAT checkers. *Artificial Intelligence* 151(1-2):177–212.

Lee, J., and Lifschitz, V. 2003. Loop formulas for disjunctive logic programs. In Palamidessi, C., ed., *Proceedings of the Nineteenth International Conference on Logic Programming (ICLP'03)*, 451–465. Springer-Verlag.

Lee, J. 2005. A model-theoretic counterpart of loop formulas. In Kaelbling, L., and Saffiotti, A., eds., *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, 503–508. Professional Book Center.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic* 7(3):499–562.

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

Lierler, Y. 2005. cmodels – SAT-based disjunctive answer set solver. In Baral, C.; Greco, G.; Leone, N.; and Terracina, G., eds., *Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, 447–451. Springer-Verlag.

Lierler, Y. 2008. Personal communication.

Lifschitz, V., and Niemelä, I., eds. 2004. *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04)*. Springer-Verlag.

Lifschitz, V., and Razborov, A. 2006. Why are there so many loop formulas? *ACM Transactions on Computational Logic* 7(2):261–268.

Marek, V., and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In Apt, K.; Marek, W.; Truszczyński, M.; and Warren, D., eds., *The Logic Programming Paradigm: a 25-Year Perspective*, 375–398. Springer-Verlag.

Marques-Silva, J., and Sakallah, K. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers* 48(5):506–521.

Mitchell, D. 2005. A SAT solver primer. *Bulletin of the European Association for Theoretical Computer Science* 85:112–133.

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

Papadimitriou, C. 1994. *Computational Complexity*. Addison-Wesley.

Pfeifer, G. 2004. Improving the model generation/checking interplay to enhance the evaluation of disjunctive programs. In Lifschitz and Niemelä (2004), 220–233.

Ranjan, D.; Tang, D.; and Malik, S. 2004. A comparative study of 2QBF algorithms. In *Electronic Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*.

Ricca, F.; Faber, W.; and Leone, N. 2006. A backjumping technique for disjunctive logic programming. *AI Communications* 19(2):155–172.

Schaefer, M., and Umans, C. 2002. Completeness in the polynomial-time hierarchy: A compendium. *ACM SIGACT News* 33(3):32–49.

Schlipf, J. 1995. The expressive powers of the logic programming semantics. *Journal of Computer and Systems Sciences* 51:64–86.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1-2):181–234.

Syrjänen, T. Lparse 1.0 user's manual. Available at http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz.

Van Gelder, A.; Ross, K.; and Schlipf, J. 1991. The well-founded semantics for general logic programs. *Journal of the ACM* 38(3):620–650.

Ward, J., and Schlipf, J. 2004. Answer set programming with clause learning. In Lifschitz and Niemelä (2004), 302–313.

Zhang, L.; Madigan, C.; Moskewicz, M.; and Malik, S. 2001. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'01)*, 279–285.