Evaluation of disjunctive programs in WASP

Mario Alviano[1][0000−0002−2052−2063], Giovanni Amendola[1][0000−0002−2111−9671], Carmine Dodaro[2][0000−0002−5617−5286], Nicola Leone[1][0000−0002−9742−1252], Marco Maratea[2][0000−0002−9034−2527], and Francesco Ricca[1][0000−0001−8218−3178]

[1] DEMACS, University of Calabria, Rende, Italy,
{alviano,amendola,leone,ricca}@mat.unical.it,
[2] DIBRIS, University of Genoa, Genoa, Italy
{dodaro,marco}@dibris.unige.it

Abstract. Answer Set Programming (ASP) is a well-established declarative programming language based on logic. The success of ASP is mainly due to the availability of efficient ASP solvers, therefore their development is still an important research topic. In this paper we report the recent improvements of the well-known ASP solver WASP. The new version of WASP includes several improvements of the main solving strategies and advanced reasoning techniques for computing paracoherent answer sets. Indeed, WASP is the first ASP solver handling paracoherent reasoning under two mainstream semantics, namely semi-stable and semi-equilibrium. However, semi-equilibrium semantics may require the introduction of several disjunctive rules, which are usually considered as a source of inefficiency for modern solvers. Such a drawback is addressed in WASP by implementing ad-hoc techniques to efficiently handle disjunctive logic programs. These techniques are presented and evaluated in this paper.

Keywords: answer set programming, answer set computation, disjunctive logic programs

1 Introduction

Answer set programming (ASP) [17] is a declarative formalism for knowledge representation and reasoning based on the stable model semantics [27]. The success of ASP is witnessed by the increasing number of academic and industrial applications [1, 11, 14, 20, 28], and it is mainly due to the combination of its high knowledge-modeling power with robust solving technology [5, 23]. For this reason, the development of new efficient solvers and solving techniques is still an important research topic.

In this paper we present the progress in the development of the ASP solver WASP [4, 5]. Among the features recently included in WASP, advanced reasoning techniques for computing *paracoherent answer sets* [9] are of particular interest, as in fact WASP is the first solver that is able to compute paracoherent answer sets according to two mainstream semantics, namely *semi-stable* and *semi-equilibrium* [10, 12, 13]. In this context, it is important to emphasize that the evaluation of ASP programs under the semi-equilibrium semantics may lead to a deterioration of the performance of the solver due to a significant amount of disjunctive rules introduced by the implemented algorithm [8].

Disjunctive rules are a common source of inefficiency for many ASP solvers based on the Clark's completion [18], such as CMODELS [29], LP2SAT [30], and CLASP [23]. In particular, in the disjunctive case, such solvers apply a rewriting technique, called *shift* [16], that causes a quadratic blow-up of the input program. This drawback is addressed in WASP by applying a linear rewriting technique that extends Clark's completion to the disjunctive case [3] (see Section 3.1). Moreover, disjunctive rules might increase the computational complexity of several reasoning tasks because the evaluation of disjunctive logic programs may require to perform an additional co-NP-complete task, usually referred to as *answer set checking* (or *stability checking*).

Answer set checking is usually carried out by checking the unsatisfiability of a propositional formula, which can be constructed according to different strategies. The first of such strategies was implemented in the ASP solver DLV and is based on the reduct of the input program with respect to the answer set candidate to be checked [32]. Albeit the construction of such a formula can be done in polynomial time, in practice its creation is often more expensive than the unsatisfiability check. Moreover, the traditional reduct-based approach cannot reuse any information from previous checks and requires to build a new formula each time the stability check is required. An alternative strategy was implemented in the ASP solver CLASP, where a characterization of answer sets based on *unfounded sets* is used to obtain a formula that can be reused for all stability checks [24]. However, the formula built using this strategy is quadratic in the size of the program, while the reduct-based approach produces linear formulas.

The main contribution of this paper is to show how to improve the efficiency of the mainstream strategies for handling disjunctive logic programs under stable model semantics. In particular, we describe how the reduct-based approach can be modified in order to use the same formula in all answer set checks (Section 3.2), and we propose a slight, yet effective, modification of the unfounded-based approach so to make it linear (Section 3.3); the new algorithms are integrated in the solver WASP. After that, we empirically assess the impact of the new features on several benchmarks, showing that WASP can efficiently handle disjunctive ASP programs (Section 4).

2 Preliminaries

2.1 Propositional logic

Syntax. Let $\mathcal{A}$ be a fixed, countable set of (propositional) *atoms*. A *literal* ℓ is either an atom p, or its negation $\neg p$. For a negative literal $\neg p$, $\neg\neg p := p$. A *clause* is a set of literals representing a disjunction, and a propositional formula φ is a set of clauses representing a conjunction, i.e., only formulas in *conjunctive normal form* (CNF) are considered here. For a formula φ, $size(\varphi) := \sum_{c\in\varphi} |c|$, and $At(\varphi)$ is the set of atoms appearing in φ. For $n \geq 0$, and $\ell_0, \ldots, \ell_n$ being literals, formula $\ell_0 \leftrightarrow \ell_1 \wedge \cdots \wedge \ell_n$ is a compact representation of the following clauses: $\{\ell_0\} \cup \{\neg\ell_i \mid i \in [1..n]\}$; $\{\neg\ell_0, \ell_i\}$, for all $i \in [1..n]$. Similarly, $\ell_0 \leftrightarrow \ell_1 \vee \cdots \vee \ell_n$ is a compact representation of the following clauses: $\{\ell_0, \neg\ell_i\}$, for all $i \in [1..n]$; $\{\neg\ell_0\} \cup \{\ell_i \mid i \in [1..n]\}$.

Semantics. An interpretation I is a set of atoms in $\mathcal{A}$. Intuitively, atoms in I are true, and those in $\mathcal{A} \setminus I$ are false. Relation $\models$ is defined as follows: for $p \in \mathcal{A}$, $I \models p$ if $p \in I$,

and $I \models \neg p$ if $p \notin I$; for a clause c, $I \models c$ if $I \models \ell$ for some $\ell \in c$; for a formula φ, $I \models \varphi$ if $I \models c$ for all $c \in \varphi$. If $I \models \varphi$ then I is a *model* of φ, I *satisfies* φ, and φ is true w.r.t. I. If $I \not\models \varphi$ then I is not a model of φ, I *violates* φ, and φ is false w.r.t. I. Similarly for literals, and clauses. A formula φ is *satisfiable* if there is an interpretation I such that $I \models \varphi$; otherwise, φ is *unsatisfiable*.

2.2 Answer set programming

A *literal* ℓ is either an atom p, or its negation $\sim p$, where $\sim$ denotes *negation as failure*. Let $\overline{\ell}$ denote the complement of ℓ, i.e., $\overline{p} := \sim p$, and $\overline{\sim p} := p$, for all $p \in \mathcal{A}$. This notation is extended to sets of literals, i.e., for a set S of literals, $\overline{S} := \{\overline{\ell} \mid \ell \in S\}$.

A disjunctive logic program Π is a finite set of rules of the following form:

$$a_1 \mid \cdots \mid a_n \leftarrow b_1, \cdots, b_k, \sim b_{k+1}, \cdots, \sim b_m \tag{1}$$

where $n \geq 1$, $m \geq k \geq 0$, and $a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms in $\mathcal{A}$. For a rule r of the form (1), set $\{a_1, \ldots, a_n\}$ is called *head* of r, and denoted $H(r)$; while $\{b_1, \ldots, b_k, \sim b_{k+1}, \ldots, \sim b_m\}$ is named *body* of r, and denoted $B(r)$; sets $\{b_1, \ldots, b_k\}$ and $\{b_{k+1}, \ldots, b_m\}$ of positive and negative literals in $B(r)$ are denoted $B^+(r)$ and $B^-(r)$, respectively. Given an atom p, $heads(\Pi, p) := \{r \mid r \in \Pi, p \in H(r)\}$. For a rule r of the form (1), $size(r) := n + m$. For a program Π, $size(\Pi) := \sum_{r \in \Pi} size(r)$ and $At(\Pi)$ denotes the set of atoms appearing in Π.

Semantics. An interpretation I is a set of atoms in $\mathcal{A}$. Relation $\models$ is extended as follows: for a negative literal $\sim a$, $I \models \sim a$ if $I \not\models a$; for a rule r, $I \models B(r)$ if $I \models \ell$ for all literals $\ell \in B(r)$, $I \not\models B(r)$ if $I \not\models \ell$ for a literal $\ell \in B(r)$, $I \models r$ if $H(r) \cap I \neq \emptyset$ whenever $I \models B(r)$; for a program Π, $I \models \Pi$ if $I \models r$ for all $r \in \Pi$. An interpretation I is a *model* of Π if $I \models \Pi$. An interpretation I is *supported* in Π if for all $p \in I$ there is a rule $r \in \Pi$ such that $I \models B(r)$ and $H(r) \cap I = \{p\}$. The definition of answer set is based on a notion of program reduct [27]: Let Π be a disjunctive logic program, and I an interpretation. The reduct of Π with respect to I, denoted Π^I, is obtained from Π by deleting each rule r such that $I \not\models B(r)$, and removing negative literals and false head atoms in the remaining rules. A supported model I of Π is an *answer set* if there is no $J \subset I$ such that $J \models \Pi^I$. Let $AS(\Pi)$ denote the set of answer sets of Π. Program Π is *coherent* if $AS(\Pi) \neq \emptyset$; otherwise, it is *incoherent*.

3 Answer set computation

In this section, we review the main techniques employed by WASP for the computation of an answer set. In particular, WASP first encodes the input program Π as a propositional formula by applying the (Clark's) completion (see Section 3.1), whose models are all supported models of Π [18, 33]. After that, WASP searches for an answer set by implementing a variant of the CDCL backtracking algorithm on the completion of Π as described in [5].

The backtracking algorithm is based on the pattern *choose-propagate-learn*. In a nutshell, the algorithm builds an answer set step-by-step starting from an empty set

of literals A. At each step, a literal, called *branching literal*, is added to A (*choice*), and the deterministic consequences of this choice are *propagated*, that is, other literals are added to A. Propagation is carried out by applying several inference rules, called *propagators*. In case the propagation leads to a *conflict*, i.e., an atom and its negation are both in A, the algorithm *learns* a new clause, undoes the choices leading to the conflict, and restores the consistency of A. This process is repeated until the incoherence of Π is proven or $I := A \cap atoms(\Pi)$ is a (supported) model of Π. In the latter case, a stability check on I is possibly performed (more specifically, if Π is *non head-cycle-free* [24]); if the stability check is successful, I is an answer set and the algorithm terminates, otherwise a conflict is raised and a new clause is learned. The stability check amounts to checking the satisfiability of a formula φ, built starting from Π and I. Actually, in WASP the formula φ can be created according to two strategies, referred to as reduct-based (Section 3.2) and unfounded-based (Section 3.3).

3.1 Completion

In the following, we briefly recall the *Clark's completion* [18] and we describe the completion implemented by WASP. First, consider programs without disjunction, i.e., where for each rule of the form (1), n is equal to 1. In particular, given a program Π without disjunction, the completion of Π, denoted $Comp(\Pi)$, is the set of clauses:

$$a_1^r \leftrightarrow b_1 \wedge \cdots \wedge b_k \wedge \neg b_{k+1} \wedge \cdots \wedge \neg b_m \tag{2}$$

for all $r \in \Pi$ of the form (1) with $n = 1$, where a_1^r is a fresh atom (true if and only if r is a support of a_1), together with

$$a \leftrightarrow \bigvee_{r \in heads(\Pi, a)} a^r \tag{3}$$

for all $a \in At(\Pi)$. Note that the construction of the completion is linear in size.

In order to apply completion to programs in general, a transformation known as *shift* [16] is first applied to the input program Π, so to obtain a program $Shift(\Pi)$ with the same supported models. Formally, for a program Π, $Shift(\Pi)$ is defined as follows. For all rules $r \in \Pi$ of the form (1) and for all $a_i \in H(r)$, $Shift(\Pi)$ contains a rule r', such that $H(r') := \{a_i\}$ and $B(r') := B(r) \cup (\overline{H(r) \setminus \{a_i\}})$. The strength of the shift is to preserve supported models, however the construction is not linear, but quadratic in size. This weakness is circumvented in WASP by directly extending completion to the disjunctive case [3]. In particular, auxiliary atoms a_i^r will be used with the same meaning of the disjunction-free case, i.e., rule r of the form (1) supports atom a_i, for $i \in [1..n]$. However, since n may be greater than 1, other atoms occurring in the head of r have to be taken into account. Additional auxiliary atoms will be thus used, and in particular: s_i^r, true if and only if rule r may support a_i, for $i \in [1..n]$; d_i^r, true if and only if the disjunction $a_i \vee \cdots \vee a_n$ is true, for $i \in [2..n]$.

The completion of a program Π, denoted $Comp^{\vee}(\Pi)$, is the set of clauses:

$$d_i^r \leftrightarrow a_i \vee d_{i+1}^r \quad \forall i \in [2..n-1] \tag{4}$$
$$d_n^r \leftrightarrow a_n \quad \text{if } n \geq 2 \tag{5}$$
$$s_1^r \leftrightarrow b_1 \wedge \cdots \wedge b_k \wedge \neg b_{k+1} \wedge \cdots \wedge \neg b_m \tag{6}$$
$$s_i^r \leftrightarrow s_{i-1}^r \wedge \neg a_{i-1} \quad \forall i \in [2..n] \tag{7}$$
$$a_i^r \leftrightarrow s_i^r \wedge \neg d_{i+1}^r \quad \forall i \in [1..n-1] \tag{8}$$
$$a_n^r \leftrightarrow s_n^r \tag{9}$$

for all $r \in \Pi$ of the form (1), together with (3) for all $a \in At(\Pi)$. Note that (5) defines d_n^r as an alias of a_n. Similarly, (9) defines s_n^r as an alias of a_n^r. It turns out that d_n^r and s_n^r could be simplified in the above construction, but they are left to ease the reading. Note that for $n = 1$ the above equations essentially give (2): only (6) and (9) are used in this case, and (6) is precisely (2) if s_1^r is replaced by its alias a_1^r.

Finally, we mention that WASP supports a *disjunctive propagator*, which is used to compactly represent clauses from (4) to (9), as detailed in [3]. The disjunctive propagator usually reduces the memory footprint and the solving time of WASP.

3.2 Reduct-based stability check

Let Π be a program, and I be an interpretation. Let $\mathcal{C}(\Pi, I)$ be the propositional formula $\{\mathcal{C}(r, I) \mid r \in \Pi^I\}$, where for each rule r, $\mathcal{C}(r, I)$ is the following clause:

$$(H(r) \cap I) \cup \{\neg b \mid b \in B^+(r)\}.$$

Intuitively, the clauses $\mathcal{C}(\Pi, I)$ encode the program reduct Π^I. Let $c_{\subset}(I)$ denote the clause $\{\neg a \mid a \in I\}$, enforcing at least one atom in I to be assigned false. Formula $red_{bas}(\Pi, I)$ is thus $\mathcal{C}(\Pi, I) \cup \{c_{\subset}(I)\}$.

The following mapping between stability and satisfiability checks is established.

Proposition 1 (Theorem 4.2 of [32]). *Let Π be a program, and I be an interpretation. $I \in AS(\Pi)$ if and only if $red_{bas}(\Pi, I)$ is unsatisfiable.*

Note that both $\mathcal{C}(\Pi, I)$, and $c_{\subset}(I)$ depend on I. Therefore, sensibly different propositional formulas have to be built for each stability check, and in general exponentially many checks may be performed while searching for an answer set of the input program. The following example should better clarify this aspect.

Example 1. Consider the following program Π_1:

$$a \mid b \leftarrow c \qquad a \leftarrow b, {\sim}e \qquad b \leftarrow a, {\sim}e \qquad c \mid d \leftarrow \qquad e \mid f \leftarrow \qquad a \leftarrow {\sim}b$$

and the answer set candidate to check is $I_1 := \{a, b, c, f\}$. Then, $\mathcal{C}(\Pi_1, I_1)$ is composed by $\{a, b, \neg c\}$, $\{a, \neg b\}$, $\{b, \neg a\}$, $\{c\}$, and $\{f\}$; while $c_{\subset}(I_1) := \{\neg a, \neg b, \neg c, \neg f\}$. Formula $red_{bas}(\Pi_1, I_1)$ is unsatisfiable, thus I_1 is an answer set. Consider again the program Π_1 and suppose that the answer set candidate to check is $I_2 = \{a, b, d, f\}$. In this case, $\mathcal{C}(\Pi_1, I_2)$ is composed by $\{a, \neg b\}$, $\{b, \neg a\}$, $\{d\}$, and $\{f\}$; while $c_{\subset}(I_2) =$

$\{\neg a, \neg b, \neg d, \neg f\}$. Formula $red_{bas}(\Pi_1, I_2)$ is satisfiable, thus I_2 is not an answer set. Note that the two formulas $red_{bas}(\Pi_1, I_1)$, and $red_{bas}(\Pi_1, I_2)$ have in common several clauses, i.e., $\{a, \neg b\}, \{b, \neg a\}, \{f\}$. However, at each check the formula is rebuilt without taking into account this information. $\lhd$

In order to overcome the main weakness of the basic stability check, the propositional formula $red_{bas}(\Pi, I)$ is replaced by a refined formula $red_{adv}(\Pi, I)$ such that each of its clauses depends on either Π, or I, but not both. Actually, many clauses of the new formula will only depend on Π, which will allow to reuse them in subsequent stability checks. As will be clarified soon, these clauses compactly encode all possible reducts for the input program Π, so that the specific reduct Π^I for the interpretation I to be checked can be selected by properly adding to $red_{adv}(\Pi, I)$ a set of unit clauses, i.e., clauses consisting of a single literal.

Formally, for a rule r, let $\mathcal{C}(r)$ denote the following clause:

$$H(r) \cup \{\neg b \mid b \in B^+(r)\} \cup \{b' \mid b \in B^-(r)\}$$

where each b' is a fresh atom, i.e., an atom not occurring in Π. These fresh atoms are required because the interpretation of negative literals in program reducts is fixed by definition: their falsity implies the deletion of r, and their truth imply their own elimination. For a program Π, let $\mathcal{C}(\Pi)$ be the propositional formula $\{\mathcal{C}(r) \mid r \in \Pi\}$.

For an interpretation I, define $\mathit{fix}(I)$ to be the following set of clauses:

$$\{\{\neg a\} \mid a \in \mathcal{A} \setminus I\} \cup \{\{b'\} \mid b \in I\} \cup \{\{\neg b'\} \mid b \in \mathcal{A} \setminus I\}.$$

Intuitively, $\mathit{fix}(I)$ fixes the interpretation of false as well as fresh atoms. Finally, formula $red_{adv}(\Pi, I)$ is defined as $\mathcal{C}(\Pi) \cup \mathit{fix}(I) \cup \{c_{\subset}(I)\}$.

It is important to observe that simplifying $\mathcal{C}(\Pi)$ by means of the unary clauses in $\mathit{fix}(I)$ would result in the formula $\mathcal{C}(\Pi, I)$. The analogous of Proposition 1 can thus be established for the advanced stability check.

Theorem 1. *Let Π be a program, and I be an interpretation. $I \in AS(\Pi)$ if and only if $red_{adv}(\Pi, I)$ is unsatisfiable.*

Example 2. Consider the program Π_1 of Example 1. Suppose that the answer set candidate to check is $I_1 := \{a, b, c, f\}$. Then, $\mathcal{C}(\Pi_1)$ comprises the clauses $\{a, b, \neg c\}$, $\{a, \neg b, e'\}$, $\{b, \neg a, e'\}$, $\{c, d\}$, $\{f, e\}$, and $\{a, b'\}$. The set of clauses $\mathit{fix}(I_1)$ comprises $\{\neg d\}$, $\{\neg e\}$, $\{\neg e'\}$ and $\{b'\}$; while $c_{\subset}(I_1) := \{\neg a, \neg b, \neg c, \neg f\}$. Note that $\mathit{fix}(I_1)$ should also contain the unit clauses $\{a'\}$, $\{c'\}$, $\{\neg d'\}$, $\{f'\}$ which however are not necessary since such atoms do not appear in any other clause. The formula $red_{adv}(\Pi_1, I_1)$ is then unsatisfiable, thus I_1 is an answer set. Consider again the program Π_1. Suppose that the answer set candidate to check is $I_2 = \{a, b, d, f\}$. Note that, $\mathcal{C}(\Pi_1)$ is not dependent on the interpretation thus it can be reused also in this check. Then, $\mathit{fix}(I_2)$ is composed by $\{\neg c\}$, $\{\neg e\}$, $\{\neg e'\}$, and $\{b'\}$; while $c_{\subset}(I_2) = \{\neg a, \neg b, \neg d, \neg f\}$. The formula $red_{adv}(\Pi_1, I_2)$ is satisfiable, thus I_2 is not an answer set. $\lhd$

3.3 Unfounded-based stability check

Let Π be a program, and I be an interpretation. A set X of atoms is an unfounded set for Π with respect to I if for each $r \in \Pi$ with $H(r) \cap X \neq \emptyset$ then $I \not\models B(r)$, or $B^+(r) \cap X \neq \emptyset$, or $(H(r) \setminus X) \cap I \neq \emptyset$. Note that, I is an answer set of Π iff $I \models \Pi$ and no unfounded set X is such that $X \cap I \neq \emptyset$. In the following, for a rule r we define $\neg B(r) := \{\neg q \mid q \in B^+(r)\} \cup \{q \mid q \in B^-(r)\}$.

Given a program Π and an interpretation I, the stability of I can be checked by encoding the unfounded conditions. In more detail, for each atom $p \in \mathcal{A}$ two auxiliary atoms are used, namely u_p and h_p, where atom u_p is true iff p is unfounded and atom h_p is true iff p is true and founded. Then, $\mathcal{U}(r, p)$ denotes the following clause:

$$\{\neg u_p\} \cup \neg B(r) \cup \{u_q \mid q \in B^+(r)\} \cup \{h_q \mid q \in H(r) \setminus \{p\}\}$$

and $\mathcal{U}(\Pi)$ is the formula $\{\mathcal{U}(r, p) \mid r \in \Pi, p \in H(r)\}$. Moreover, let $\mathcal{H}(p)$ be the clauses $h_p \leftrightarrow p \wedge \neg u_p$, $\mathcal{H}(\Pi)$ be the formula $\{\mathcal{H}(p) \mid p \in At(\Pi)\}$ and $c(\Pi)$ be the clause $\{u_p \mid p \in At(\Pi)\}$. For an interpretation I, define $\mathit{fix}'(I)$ to be the formula:

$$\{\neg u_p \mid p \notin I\} \cup \{\neg p \mid p \notin I\} \cup \{p \mid p \in I\}$$

Finally, formula $\mathit{unf}_{\mathit{qdt}}(\Pi, I)$ is defined as $\mathcal{U}(\Pi) \cup \mathcal{H}(\Pi) \cup \{c(\Pi)\} \cup \mathit{fix}'(I)$.

Proposition 2 (Theorem 3 of [24]). *Let Π be a program, and I be an interpretation. $I \in AS(\Pi)$ if and only if $\mathit{unf}_{\mathit{qdt}}(\Pi, I)$ is unsatisfiable.*

Example 3. Consider the program Π_1 in Example 1. Suppose that the answer set candidate to check is $I_1 := \{a, b, c, f\}$. Then, $\mathcal{U}(\Pi_1)$ is $\{\neg u_a, \neg c, u_c, h_b\}$, $\{\neg u_b, \neg c, u_c, h_a\}$, $\{\neg u_a, \neg b, u_b, e\}$, $\{\neg u_b, \neg a, u_a, e\}$, $\{\neg u_c, h_d\}$, $\{\neg u_d, h_c\}$, $\{\neg u_e, h_f\}$, $\{\neg u_f, h_e\}$, and $\{\neg u_a, b\}$. Moreover, for $p \in \{a, b, c, d, e, f\}$, $\mathcal{H}(p)$ comprises $\{\neg h_p, p\}$, $\{\neg h_p, \neg u_p\}$, $\{h_p, \neg p, u_p\}$; while $c(\Pi_1)$ is the clause $\{u_a, u_b, u_c, u_d, u_e, u_f\}$. The clauses in $\mathit{fix}'(I_1)$ are $\{\neg u_d\}$, $\{\neg u_e\}$, $\{\neg d\}$, $\{\neg e\}$, $\{a\}$, $\{b\}$, $\{c\}$, and $\{f\}$. The formula $\mathit{unf}_{\mathit{qdt}}(\Pi_1, I_1)$ is then unsatisfiable, thus I_1 is an answer set. Consider again the program Π_1. Suppose that the answer set candidate to check is $I_2 = \{a, b, d, f\}$. Interestingly, $\mathcal{U}(\Pi_1)$, $\mathcal{H}(\Pi_1)$, and $c(\Pi_1)$ are not dependent on the interpretation thus they can be reused also in this check. Then, $\mathit{fix}'(I_2)$ is composed by $\{\neg u_c\}$, $\{\neg u_e\}$, $\{\neg c\}$, $\{\neg e\}$, $\{a\}$, $\{b\}$, $\{d\}$, and $\{f\}$. The formula $\mathit{unf}_{\mathit{qdt}}(\Pi_1, I_2)$ is satisfiable, thus I_2 is not an answer set. $\lhd$

A weakness of $\mathit{unf}_{\mathit{qdt}}(\Pi, I)$ is that its size is not always linear with respect to the size of Π, as formalized next.

Proposition 3. *In the worst case, for a rule r of a program Π, $\mathit{size}(\{\mathcal{U}(r, p) \mid p \in H(r)\})$ is quadratic with respect to $\mathit{size}(r)$.*

Proof. Let r be of the form (1). Hence, $|\{\mathcal{U}(r, p) \mid p \in H(r)\}| = n$, and each $\mathcal{U}(r, p)$ has size $n + m + k$. Hence, $\mathit{size}(\{\mathcal{U}(r, p) \mid p \in H(r)\}) = n \cdot (n + m + k)$. $\square$

In order to circumvent this weakness, in the following we propose a modification of the formula $\mathcal{U}(\Pi)$. In particular, $\mathcal{U}'(r, p)$ denotes the clause $\{\neg u_p, \mathit{aux}_r\}$ and $\mathcal{U}'(r)$ denotes the clause $\{\neg \mathit{aux}_r\} \cup \neg B(r) \cup \{u_q \mid q \in B^+(r)\} \cup \{h_q \mid q \in H(r)\}$, where aux_r is a fresh atom not appearing elsewhere in the formula, and $\mathcal{U}'(\Pi)$ is the formula $\{\mathcal{U}'(r, p) \mid r \in \Pi, p \in H(r)\} \cup \{\mathcal{U}'(r) \mid r \in \Pi\}$. Finally, formula $\mathit{unf}_{\mathit{lin}}(\Pi, I)$ is defined as $\mathcal{U}'(\Pi) \cup \mathcal{H}(\Pi) \cup \{c(\Pi)\} \cup \mathit{fix}'(I)$.

Theorem 2. *Let Π be a program, and I be an interpretation. $I \in AS(\Pi)$ if and only if $unf_{lin}(\Pi, I)$ is unsatisfiable.*

Proof. We know that $I \in AS(\Pi)$ iff $unf_{qdt}(\Pi, I)$ is unsatisfiable (Theorem 3 of [24]). Hence, to prove our claim we can show that $unf_{lin}(\Pi, I)$ is satisfiable iff $unf_{qdt}(\Pi, I)$ is satisfiable. For an interpretation M, let $ext(M)$ be $(M \cap At(unf_{qdt}(\Pi, I))) \cup \{aux_r \mid r \in \Pi, M \models \mathcal{U}'(r) \setminus \{\neg aux_r\}\}$. We shall show the following properties:

(i) if $M \models unf_{qdt}(\Pi, I)$, then $ext(M) \models unf_{lin}(\Pi, I)$;
(ii) if $M \models unf_{lin}(\Pi, I)$, then $M \cap At(unf_{qdt}(\Pi, I)) \models unf_{qdt}(\Pi, I)$.

Proof of (i). Let $M \models unf_{qdt}(\Pi)$. We have to show that $ext(M) \models \mathcal{U}'(\Pi)$ (since the other clauses in $unf_{lin}(\Pi, I)$ also belong to $unf_{qdt}(\Pi, I)$). Recall that $\mathcal{U}'(\Pi)$ contains clause $\mathcal{U}'(r)$ for each $r \in \Pi$, and clause $\mathcal{U}'(r, p)$ for each atom $p \in H(r)$. Let us first consider a clause $\mathcal{U}'(r)$. If $ext(M) \models \neg aux_r$, then $ext(M) \models \mathcal{U}'(r)$ trivially. Otherwise, if $ext(M) \models aux_r$, then $M \models \mathcal{U}'(r) \setminus \{\neg aux_r\}$ by construction of $ext(M)$; hence, $ext(M) \models \mathcal{U}'(r)$. Let us now consider a clause $\mathcal{U}'(r, p)$. If $ext(M) \models aux_r$, then $ext(M) \models \mathcal{U}'(r, p)$ trivially. Otherwise, if $ext(M) \models \neg aux_r$ then $M \not\models \mathcal{U}'(r) \setminus \{\neg aux_r\}$ by construction of $ext(M)$; hence, $M \not\models \mathcal{U}(r, p) \setminus \{\neg u_p\}$ (because $\mathcal{U}(r, p) \setminus \{\neg u_p\} \subset \mathcal{U}'(r) \setminus \{\neg aux_r\}$), and therefore $M \models \neg u_p$ (because $M \models \mathcal{U}(r, p)$ by assumption). We can thus conclude that $ext(M) \models \mathcal{U}'(r, p)$.

Proof of (ii). Let $M \models unf_{lin}(\Pi)$ and M' be $M \cap At(unf_{qdt}(\Pi, I))$. We have to show that $M' \models \mathcal{U}(\Pi)$ (since the other clauses in $unf_{qdt}(\Pi, I)$ also belong to $unf_{lin}(\Pi, I)$). Recall that $\mathcal{U}(\Pi)$ contains clause $\mathcal{U}(r, p)$ for each rule $r \in \Pi$ and for each atom $p \in H(r)$. If $M \models \neg u_p$ then $M' \models \mathcal{U}(r, p)$ trivially. Otherwise, if $M \models u_p$, then $M \models aux_r$ (because $M \models \mathcal{U}'(r, p)$) and $M \models \neg h_p$ (because $M \models \mathcal{H}(p)$). Hence, $M' \models \mathcal{U}'(r) \setminus \{\neg aux_r, h_p\}$, and since $\mathcal{U}'(r) \setminus \{\neg aux_r, h_p\} = \mathcal{U}(r, p) \setminus \{\neg u_p\}$ we can conclude that $M' \models \mathcal{U}(r, p)$. □

Example 4. Let r be the rule $a \mid b \leftarrow c$ of program Π_1 in Example 3. Then, $\mathcal{U}'(r)$ is $\{\neg aux_r, \neg c, u_c, h_a, h_b\}$, while $\mathcal{U}'(r, a)$ is $\{\neg u_a, aux_r\}$ and $\mathcal{U}'(r, b)$ is $\{\neg u_b, aux_r\}$. ◁

Proposition 4. *In the worst case, for a rule r of a program Π, $size(\{\mathcal{U}'(r, p) \mid p \in H(r)\} \cup \{\mathcal{U}'(r)\})$ is linear with respect to $size(r)$.*

Proof. Let r be of the form (1). Then, $size(\{\mathcal{U}'(r, p) \mid p \in H(r)\}) = 2 \cdot n$, while $size(\{\mathcal{U}'(r) \mid r \in \Pi\}) = 1 + m + k + n$. Thus, $size(\{\mathcal{U}'(r, p) \mid p \in H(r)\} \cup \{\mathcal{U}'(r) \mid r \in \Pi\}) = 3 \cdot n + m + k + 1$. □

4 Experiments

The impact of the techniques described in this paper on the performance of WASP was assessed empirically on three benchmarks: (i) instances from the latest ASP Competition [25] containing cyclic disjunctive rules; (ii) a synthetic benchmark containing disjunctive rules with increasing size of heads; and (iii) computation of paracoherent answer sets of programs from [8]. For (i) and (ii), WASP was executed with the reduct-based and unfounded-based strategies for answer set checking, referred to as WASP_{RED}

and WASP$_{\text{UNF}}$, respectively; and compared with CLASP [22, 23] version 3.3.3. For (iii), WASP was also compared with CLASP version 3.3.3. Since the latter does not support paracoherent reasoning, we used the preprocessor of WASP for the computation of the *externally extended supported program* as described in [10]. Then, this program is extended by adding weak constraints as in the algorithm WEAK, described in [9], in such a way that each optimal answer set of the new program is guaranteed to be a paracoherent answer set. For both solvers, we used a similar algorithm based on *unsatisfiable cores* [6, 22] for computing an optimal answer set. In all cases, the completion was enabled using the strategy *auto* that applies the syntactic rewriting $Comp^{\vee}$ for rules whose head size is at most 4, and the propagator for rules with larger heads. The experiment was run on an Intel CPU 2.4 GHz with 16 GB of RAM. Time and memory were limited to 1200 seconds and 15 GB, respectively. All instances were grounded by GRINGO 4.5.4 [21], whose execution time and memory consumption are accounted in our analysis. Benchmarks can be found at https://doi.org/10.5281/zenodo.2605076.

Concerning benchmark (i), the tested encodings are *Complex Optimization Of Answer Sets*, *Minimal Diagnosis*, and *Random Disjunctive Programs* [15]. Results are provided in Table 1, where the number of solved instances is reported for each solver. WASP$_{\text{RED}}$ solves 47 instances, whereas the performance achieved by WASP$_{\text{UNF}}$ is slightly worse, with 43 solved instances. CLASP is the best performing solver on this benchmark: it solves 54 instances in the allotted time; its advantage is due to the good performance on *Random Disjunctive Programs*, where it solves 6 instances more than WASP.

The advantage of the linear rewriting techniques for handling disjunctive rules does not emerge on the instances of benchmark (i), whose head sizes are at most 2. Hence, in order to assess the scalability of the proposed techniques, we considered the synthetic benchmark (ii). The idea is to have a prototypical family of programs that allows to link the efficiency of a solver with the size of disjunctive heads. Specifically, we generated programs of the following form varying the constant n: $\{a_1 \mid \cdots \mid a_n \leftarrow\} \cup \{a_{i+1} \leftarrow a_i \mid i \in [1..n-1]\} \cup \{a_1 \leftarrow a_n\}$. The results of our experiment are reported on Figure 1. We observe that CLASP scales worse than both WASP$_{\text{RED}}$ and WASP$_{\text{UNF}}$, and cannot solve the instance with $n = 10000$ in the allotted time. On the contrary, we verified that WASP scales linearly also for larger values of n: WASP$_{\text{RED}}$ and WASP$_{\text{UNF}}$ solve the instance with $n = 100000$ in 60 and 90 seconds, respectively. We report that the advantage of the linear rewriting techniques is also visible in terms of memory usage. Indeed, the memory footprint of CLASP is 2450 MB for $n = 10000$, while WASP$_{\text{RED}}$ and WASP$_{\text{UNF}}$ use 40 and 49 MB, respectively. Actually, CLASP exceeds the allotted memory with $n \geq 30000$, while WASP$_{\text{RED}}$ and WASP$_{\text{UNF}}$ use 318 and 402 MB when $n = 100000$, respectively.

Table 1. Results of benchmark (i) executed on the instances from the latest ASP competition.

Problem	#	CLASP	WASP$_{\text{RED}}$	WASP$_{\text{UNF}}$
Complex Optimization Of Answer Sets	20	20	19	16
Minimal Diagnosis	20	20	20	19
Random Disjunctive Programs	20	14	8	8

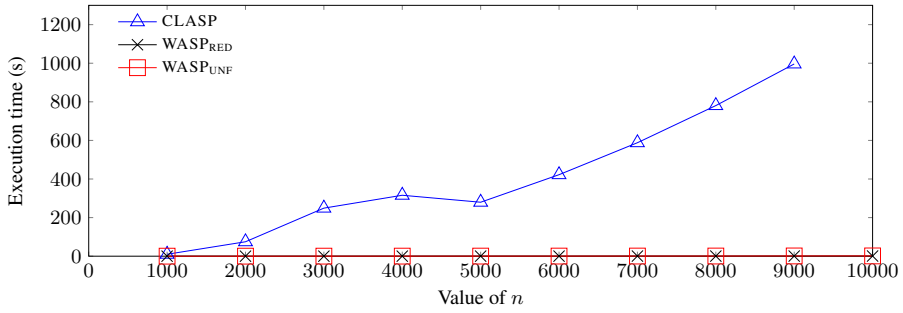

Fig. 1. Scalability analysis on the benchmark (ii).

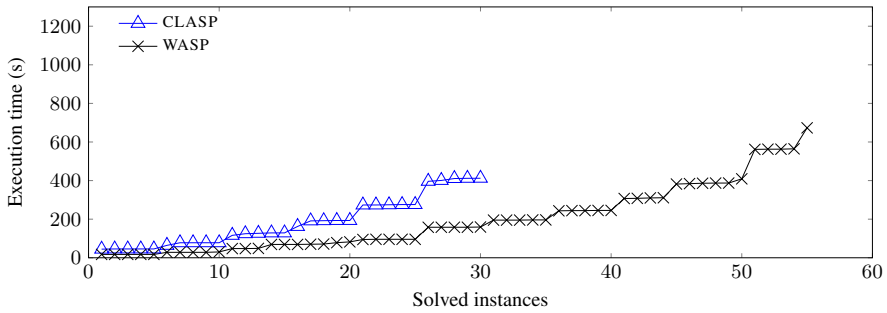

Fig. 2. Comparison of CLASP and WASP on benchmark (iii).

As for the benchmark (iii), we considered the *Stable Roommates Problem* as presented in [8], which is interesting since the semi-equilibrium transformation may produce long disjunctive rules. In our experiment, we considered different numbers of persons (from 500 to 1500), and for each of them we randomly generated 5 instances. Table 2 reports, for the different number of persons, the cumulative number, the mini-

Table 2. Sizes of disjunctive rules after the semi-equilibrium transformation on benchmark (iii).

#Persons	#Disjunctive rules	Min size	Max size	Avg size
500	5246	2	499	238
600	6252	2	599	288
700	7240	2	699	338
800	8244	2	799	388
900	9282	2	899	436
1000	10304	2	999	485
1100	11260	2	1099	537
1200	12438	2	1199	579
1300	13246	2	1299	638
1400	14244	2	1399	688
1500	15304	2	1499	735

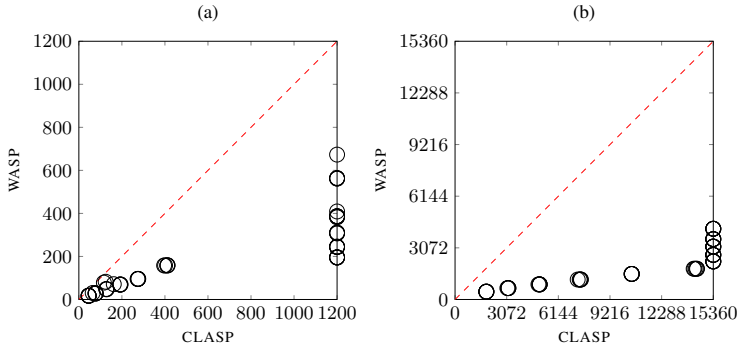

Fig. 3. Instance-wise comparison of solving time in seconds (a) and memory usage in MB (b) of CLASP and WASP on benchmark (iii).

mum size, the maximum size, and the average size of disjunctions, respectively. Results show that WASP solves all 55 instances, while CLASP solves 30 instances, and in general WASP scales better than CLASP as shown in Figure 2. Actually, WASP is faster than CLASP in all the tested instances as shown in the instance-wise comparison of the solving time reported in Figure 3 (a). Moreover, we observe that WASP uses less memory than CLASP as illustrated in Figure 3 (b). Indeed, the latter exceeds the allotted memory in all the instances with a number of persons greater than or equal to 1100, whereas WASP uses on average 4205 MB on the instances with 1500 persons.

5 Related work

Answer set computation is performed by WASP applying the CDCL algorithm on the (Clark's) completion of the input program. Clark's completion was introduced in the solver ASSAT [33] and later on also adopted by CMODELS [29], LP2SAT [30] and CLASP [22, 23], as well as by WASP 2 [5]. In case of disjunctive programs, such solvers apply a technique called *shift*, which is quadratic in size. The completion employed by the new version of WASP is instead linear and it can be applied as it is also by the aforementioned solvers. Interestingly, the quadratic blow-up of the shift does not affect DLV [7], GNT [31] and WASP 1 [4], which are not based on the Clark's completion but they employ custom data structures and algorithms to handle disjunction. However, the (Clark's) completion can lead to an exponential performance gain [26], and custom data structures are in general harder to maintain and require complex optimizations to achieve efficiency [5].

Concerning the stability checks, the reduct-based approach based on red_{bas} was introduced by DLV [32]. A major drawback of this approach consists of building a new propositional formula at each stability check. This limitation is overcome in WASP by using the formula red_{adv}, which is built once and then reused in all stability checks. The unfounded-based stability check based on the formula unf_{qdt} was instead introduced in [24] and implemented in CLASP. As observed in Section 3.3, the formula unf_{qdt} is in general quadratic in size. Such a drawback is addressed by WASP by applying the

formula unf_{lin}, which is instead linear in size. Interestingly, the formula unf_{qdt} is more compact than unf_{lin} up to rules with disjunctive heads of size 3, since for a rule r with $|H(r)| \leq 3$, $size(\{\mathcal{U}(r,p) \mid p \in H(r)\}) < size(\mathcal{U}'(r) \cup \{\mathcal{U}'(r,p) \mid p \in H(r)\})$. Thus, one can also combine $\mathcal{U}$ and $\mathcal{U}'$ by selecting the best one according to the rule size.

6 Conclusion and future work

In this paper we presented the techniques employed by WASP for evaluating disjunctive logic programs. Results of our empirical analysis show that WASP can efficiently handle disjunctive programs, even with long disjunctive rules. As future work, we plan to revise strategy red_{adv} because in principle it may introduce exponentially many clauses of the form $c_{\subset}(I)$, even if we never observed such a drawback in our tests. Our idea is to replace $c_{\subset}(I)$ with alternative implementations, among them a compact representation via *pseudo-Boolean constraints*, introducing the notion of *or-assumptions literals*, or driving the heuristic choices as in the algorithm *opt* [19]. Finally, we mention that WASP is part of the system DLV2 [2] and is available at https://www.mat.unical.it/DLV2/wasp.

References

1. Adrian, W.T., Manna, M., Leone, N., Amendola, G., Adrian, M.: Entity set expansion from the web via ASP. In: Technical Communications of ICLP. OASICS, vol. 58, pp. 1:1–1:5. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017). https://doi.org/10.4230/OASIcs.ICLP.2017.1
2. Alviano, M., Calimeri, F., Dodaro, C., Fuscà, D., Leone, N., Perri, S., Ricca, F., Veltri, P., Zangari, J.: The ASP system DLV2. In: LPNMR. LNCS, vol. 10377, pp. 215–221. Springer (2017). https://doi.org/10.1007/978-3-319-61660-5_19
3. Alviano, M., Dodaro, C.: Completion of disjunctive logic programs. In: IJCAI. pp. 886–892. IJCAI/AAAI Press (2016)
4. Alviano, M., Dodaro, C., Faber, W., Leone, N., Ricca, F.: WASP: A native ASP solver based on constraint learning. In: LPNMR. LNCS, vol. 8148, pp. 54–66. Springer (2013). https://doi.org/10.1007/978-3-642-40564-8_6
5. Alviano, M., Dodaro, C., Leone, N., Ricca, F.: Advances in WASP. In: LPNMR. LNCS, vol. 9345, pp. 40–54. Springer (2015). https://doi.org/10.1007/978-3-319-23264-5_5
6. Alviano, M., Dodaro, C., Marques-Silva, J., Ricca, F.: Optimum stable model search: algorithms and implementation. Journal of Logic and Computation. In press (2015). https://doi.org/10.1093/logcom/exv061
7. Alviano, M., Faber, W., Leone, N., Perri, S., Pfeifer, G., Terracina, G.: The disjunctive datalog system DLV. In: Datalog. LNCS, vol. 6702, pp. 282–301. Springer (2010). https://doi.org/10.1007/978-3-642-24206-9_17
8. Amendola, G.: Solving the stable roommates problem using incoherent answer set programs. In: RiCeRcA Workshop. CEUR Workshop Proceedings, vol. 2272. CEUR-WS.org (2018)
9. Amendola, G., Dodaro, C., Faber, W., Leone, N., Ricca, F.: On the computation of paracoherent answer sets. In: AAAI. pp. 1034–1040. AAAI Press (2017)
10. Amendola, G., Dodaro, C., Faber, W., Ricca, F.: Externally supported models for efficient computation of paracoherent answer sets. In: AAAI. pp. 1720–1727. AAAI Press (2018)
11. Amendola, G., Dodaro, C., Leone, N., Ricca, F.: On the application of answer set programming to the conference paper assignment problem. In: AI*IA. LNCS, vol. 10037, pp. 164–178. Springer (2016). https://doi.org/10.1007/978-3-319-49130-1_13

12. Amendola, G., Eiter, T., Fink, M., Leone, N., Moura, J.: Semi-equilibrium models for paracoherent answer set programs. Artif. Intell. **234**, 219–271 (2016). https://doi.org/10.1016/j.artint.2016.01.011
13. Amendola, G., Eiter, T., Leone, N.: Modular paracoherent answer sets. In: JELIA. LNCS, vol. 8761, pp. 457–471. Springer (2014). https://doi.org/10.1007/978-3-319-11558-0_32
14. Amendola, G., Greco, G., Leone, N., Veltri, P.: Modeling and reasoning about NTU games via answer set programming. In: IJCAI. pp. 38–45. IJCAI/AAAI Press (2016)
15. Amendola, G., Ricca, F., Truszczynski, M.: Generating hard random boolean formulas and disjunctive logic programs. In: IJCAI. pp. 532–538. ijcai.org (2017). https://doi.org/10.24963/ijcai.2017/75
16. Ben-Eliyahu, R., Dechter, R.: Propositional semantics for disjunctive logic programs. Ann. Math. Artif. Intell. **12**(1-2), 53–87 (1994). https://doi.org/10.1007/BF01530761
17. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM **54**(12), 92–103 (2011). https://doi.org/10.1145/2043174.2043195
18. Clark, K.L.: Negation as failure. In: Symposium on Logic and Data Bases. pp. 293–322. Advances in Data Base Theory, Plemum Press (1977)
19. Di Rosa, E., Giunchiglia, E., Maratea, M.: Solving satisfiability problems with preferences. Constraints **15**(4), 485–515 (2010). https://doi.org/10.1007/s10601-010-9095-y
20. Erdem, E., Gelfond, M., Leone, N.: Applications of answer set programming. AI Magazine **37**(3), 53–68 (2016)
21. Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., Schaub, T.: Abstract gringo. TPLP **15**(4-5), 449–463 (2015). https://doi.org/10.1017/S1471068415000150
22. Gebser, M., Kaminski, R., Kaufmann, B., Romero, J., Schaub, T.: Progress in clasp series 3. In: LPNMR. LNCS, vol. 9345, pp. 368–383. Springer (2015). https://doi.org/10.1007/978-3-319-23264-5_31
23. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artif. Intell. **187**, 52–89 (2012). https://doi.org/10.1016/j.artint.2012.04.001
24. Gebser, M., Kaufmann, B., Schaub, T.: Advanced conflict-driven disjunctive answer set solving. In: IJCAI. pp. 912–918. IJCAI/AAAI (2013)
25. Gebser, M., Maratea, M., Ricca, F.: The design of the seventh answer set programming competition. In: LPNMR. LNCS, vol. 10377, pp. 3–9. Springer (2017). https://doi.org/10.1007/978-3-319-61660-5_1
26. Gebser, M., Schaub, T.: Tableau calculi for logic programs under answer set semantics. ACM Trans. Comput. Log. **14**(2), 15:1–15:40 (2013). https://doi.org/10.1145/2480759.2480767
27. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Comput. **9**(3/4), 365–386 (1991). https://doi.org/10.1007/BF03037169
28. Gençay, E., Schüller, P., Erdem, E.: Applications of non-monotonic reasoning to automotive product configuration using answer set programming. J. Intelligent Manufacturing **30**(3), 1407–1422 (2019). https://doi.org/10.1007/s10845-017-1333-3
29. Giunchiglia, E., Lierler, Y., Maratea, M.: Sat-based answer set programming. In: AAAI. pp. 61–66. AAAI Press / The MIT Press (2004)
30. Janhunen, T.: Cross-translating answer set programs using the ASPTOOLS collection. KI **32**(2-3), 183–184 (2018). https://doi.org/10.1007/s13218-018-0529-9
31. Janhunen, T., Niemelä, I.: GNT - A solver for disjunctive logic programs. In: LPNMR. LNCS, vol. 2923, pp. 331–335. Springer (2004). https://doi.org/10.1007/978-3-540-24609-1_29
32. Koch, C., Leone, N., Pfeifer, G.: Enhancing disjunctive logic programming systems by SAT checkers. Artif. Intell. **151**(1-2), 177–212 (2003). https://doi.org/10.1016/S0004-3702(03)00078-X
33. Lin, F., Zhao, Y.: ASSAT: computing answer sets of a logic program by SAT solvers. Artif. Intell. **157**(1-2), 115–137 (2004). https://doi.org/10.1016/j.artint.2004.04.004