Shared aggregate sets in answer set programming

MARIO ALVIANO

DEMACS, University of Calabria, Italy
(*e-mail:* `alviano@mat.unical.it`)

CARMINE DODARO, MARCO MARATEA

DIBRIS, University of Genova, Italy
(*e-mail:* `dodaro@dibris.unige.it, marco@dibris.unige.it`)

Abstract

Aggregates are among the most frequently used linguistic extensions of answer set programming. The result of an aggregation may introduce new constants during the instantiation of the input program, a feature known as value invention. When the aggregation involves literals whose truth value is undefined at instantiation time, modern grounders introduce several instances of the aggregate, one for each possible interpretation of the undefined literals. This paper introduces new data structures and techniques to handle such cases, and more in general aggregations on the same aggregate set identified in the ground program in input. The proposed solution reduces the memory footprint of the solver without sacrificing efficiency. On the contrary, the performance of the solver may improve thanks to the addition of some simple entailed clauses which are not easily discovered otherwise, and since redundant computation is avoided during propagation. Empirical evidence of the potential impact of the proposed solution is given.

KEYWORDS: answer set programming; aggregations in logic programming; efficient computation.

1 Introduction

Answer set programming (ASP) extends logic programming with linguistic constructs conceived to ease the representation of common knowledge (Gelfond and Lifschitz 1988; Brewka et al. 2011; Janhunen and Niemelä 2016; Lifschitz 2016), and with constructs specifically designed for industrial applications (Gebser et al. 2013; Dodaro et al. 2016). Aggregates are among the most frequently used linguistic extensions of ASP (Simons et al. 2002; Liu et al. 2010; Bartholomew et al. 2011; Ferraris 2011; Faber et al. 2011; Alviano et al. 2011; Alviano 2011; Alviano et al. 2016), providing a natural representation to reason on properties involving sets of literals. Simple forms of aggregations combine common aggregation functions, aggregate sets, and comparison operators; common aggregation functions include `#sum`, `#count` (special case of `#sum` with uniform weights), `#min` and `#max` (replaced by normal rules by modern grounders). However, an aggregation may also result into *value invention*, as new constants are possibly introduced for assignments over aggregate sets involving literals whose truth value is undefined at instantiation time. In such cases, several instances of the aggregation are produced, one for each possible interpretation of the undefined literals in the aggregate set. Actually, such instances share the same ground aggregate set, and only the compared terms differ.

An example of the scenario described above is given by the following program (in ASP-Core-2

syntax; see Calimeri et al. 2013):

$$\{\mathtt{p(2); p(5)}\}. \qquad \mathtt{q(S)} :- \mathtt{S} = \#\mathtt{sum}\{\mathtt{X:p(X)}\}.$$

whose instantiation contains $\mathtt{q}(s) :- s = \#\mathtt{sum}\{2:\mathtt{p(2)}; 5:\mathtt{p(5)}\}$ for $s \in \{0,2,5,7\}$. Therefore, the ground program contains 4 aggregates sharing the same aggregate set.

Grounders have to apply some additional transformations, as the input language of modern solvers requires a normal form that essentially restricts rules with aggregates to the form $a :- \#\mathtt{sum}\{w_1 : \ell_1; \cdots; w_n : \ell_n\} \geq b$, where a is an atom not occurring in any other rule head, $n \geq 1$, $\ell_1, \ldots, \ell_n$ are literals, and $w_1, \ldots, w_n, b$ are positive integers. After that, solvers such as CLASP (Gebser et al. 2012; Gebser et al. 2009; Gebser et al. 2015; Gebser et al. 2013) and WASP (Dodaro et al. 2011; Alviano et al. 2013; Alviano et al. 2015) associate every normalized aggregate with a *propagator*, that is, a specific data structure receiving notifications for truth assignments to literals $a, \ell_1, \ldots, \ell_n$, and possibly deriving new assignments for these literals.

Back to the above example, the program processed by the solver would be the following:

$$\{\mathtt{p(2); p(5)}\}. \qquad a_x :- \#\mathtt{sum}\{2:\mathtt{p(2)}; 5:\mathtt{p(5)}\} \geq x. \quad (\forall x \in \{1,2,3,5,6,7\})$$
$$\mathtt{q(0)} :- \mathtt{not}\, a_1. \quad \mathtt{q(2)} :- a_2, \mathtt{not}\, a_3. \quad \mathtt{q(5)} :- a_5, \mathtt{not}\, a_6. \quad \mathtt{q(7)} :- a_7.$$

where intuitively each aggregate $s = \#\mathtt{sum}\{2:\mathtt{p(2)}; 5:\mathtt{p(5)}\}$ was replaced by the conjunction $\#\mathtt{sum}\{2:\mathtt{p(2)}; 5:\mathtt{p(5)}\} \geq s$, $\mathtt{not}\, \#\mathtt{sum}\{2:\mathtt{p(2)}; 5:\mathtt{p(5)}\} \geq s+1$. Actually, some of the aggregates in the above program differ only at a syntactic level, while they are indistinguishable at a semantic level, as they encode the same boolean function. For example, $\#\mathtt{sum}\{2:\mathtt{p(2)}; 5:\mathtt{p(5)}\} \geq 1$ and $\#\mathtt{sum}\{2:\mathtt{p(2)}; 5:\mathtt{p(5)}\} \geq 2$ are both true whenever at least one literal in their aggregate set is true. Moreover, there are some simple entailed formulas which are not easily derivable by clause learning; specifically, $a_x \rightarrow a_y$ for any $x < y$. Finally, notifications are always triggered to all propagators, whose computation is therefore redundant.

Aggregates with the same aggregate set are not necessarily the outcome of the instantiation of assignments, and may result by the instantiation of different rules. Actually, solvers cannot speculate on the origin of normalized aggregates in the instantiated program, and therefore any optimization technique must rely only on the information encoded in the aggregates themselves. The definition of such techniques is the focus of this paper.

First of all, the bound of any aggregate is replaced by the smallest possible sum being greater or equal than the bound itself, so that rule $a_1 :- \#\mathtt{sum}\{2:\mathtt{p(2)}; 5:\mathtt{p(5)}\} \geq 1$ becomes rule $a_1 :- \#\mathtt{sum}\{2:\mathtt{p(2)}; 5:\mathtt{p(5)}\} \geq 2$. After that, duplicated aggregates are removed, and aggregates with the same aggregate sets are identified, so that they can be associated with a new propagator, referred to as *shared aggregate set propagator* (Section 3). The new propagator reduces the memory footprint of the solver, preserves the behavior of the propagators currently used by CLASP and WASP, and its compact representation eases the introduction of rules encoding the entailed formulas described above.

The proposed techniques are implemented in WASP (Section 4.1), and tested empirically on a synthetic domain obtained by simplifying an encoding for a real world application in medical informatics (the authors were asked to analyze the inefficiency of ASP solvers on this encoding, but constrained to not disclose any sensitive information; see Section 4.2). The potential impact of the new techniques is clearly evident on the synthetic domain, where the performance of WASP goes over the state-of-the-art solver CLASP (Section 4.3). An additional benchmark is obtained from instances of the ASP Competitions (Alviano et al. 2013; Calimeri et al. 2016; Gebser et al. 2017), with the aim to verify the absence of overhead when shared aggregate set propagators

are applied to encodings that are already handled efficiently by CLASP and WASP. Within this respect, instances with assignments over count aggregates are considered.

2 Preliminaries

This section introduces minimal background knowledge required to present the results of this paper. Specifically, syntax and semantics of ASP programs are given, where the syntax is properly simplified to ease the presentation. After that, the stable model search procedure implemented by modern ASP solvers is sketched, focusing on the notion of propagator.

2.1 Syntax and semantics

Let $\mathscr{A}$ be a set of *atoms*. An *atomic formula* is either an atom, or the connective $\perp$. A *literal* is an atomic formula possibly preceded by the *default negation* symbol $\sim$. For a literal ℓ, let $\overline{\ell}$ denote the *complement* of ℓ, that is, $\overline{p} = \sim p$ and $\overline{\sim p} = p$ for all $p \in \mathscr{A} \cup \{\perp\}$; for a set L of literals, let $\overline{L}$ be $\{\overline{\ell} \mid \ell \in L\}$. Let $\top$ be a compact representation of $\sim\perp$.

A *rule* is of one of the following forms:

$$p_1 \vee \cdots \vee p_m \leftarrow \ell_1, \ldots, \ell_n \tag{1}$$

$$p \leftarrow \text{SUM}\{w_0 : \ell_0, \ldots, w_n : \ell_n\} \geq b \tag{2}$$

where $m \geq 1$, $n \geq 0$, $p_1, \ldots, p_m$ are atomic formulas, p is an atom, $\ell_0, \ldots, \ell_n$ are distinct literals, and $b, w_0, \ldots, w_n$ are positive integers. For a rule r of the form (1), let $H(r)$ denote the set $\{p_1, \ldots, p_m\} \cap \mathscr{A}$ of *head atoms*, and $B(r)$ denote the set $\{\ell_1, \ldots, \ell_n\}$ of *body literals* (Note that $\perp$ may occur in $B(r)$, but it is excluded from $H(r)$). For a rule r of the form (2), define $id(r) := p$, $sum(r) := \text{SUM}\{w_0 : \ell_0, \ldots, w_n : \ell_n\} \geq b$, $elem(r) := \{(w_i, \ell_i) | i \in [0..n]\}$, and $bound(r) := b$.

A *program* Π is a finite set of rules. Let $atoms(\Pi)$, $rules^\vee(\Pi)$, and $rules^\Sigma(\Pi)$ denote respectively the set of atoms occurring in Π, the set of rules of the form (1) in Π, and the set of rules of the form (2) in Π. In the following, every program Π is assumed to satisfy the following property: for each $r \in rules^\Sigma(\Pi)$, there is no $r' \in rules^\vee(\Pi)$ with $id(r) \in H(r')$, and there is no $r' \in rules^\Sigma(\Pi) \setminus \{r\}$ with $id(r) = id(r')$; stated differently, $id(r)$ is an identifier for the aggregation $sum(r)$.

Example 2.1 (*Running example*)
Let Π_{run} be the following program (similar to the example in the introduction):

$$g_2 : p_2 \vee n_2 \leftarrow \quad g_5 : p_5 \vee n_5 \leftarrow \quad r_x : a_x \leftarrow \text{SUM}\{2 : p_2, 5 : p_5\} \geq x \quad \forall x \in \{1, 2, 3, 5, 6, 7\}.$$

Note that, for all $x \in \{1, 2, 3, 5, 6, 7\}$, $id(r_x) = a_x$ does not occur in any other rule head, and therefore it is an identifier for $sum(r_x) = \text{SUM}\{2 : p_2, 5 : p_5\} \geq x$, whose elements are $elem(r_x) = \{(2, p_2), (5, p_5)\}$, and whose bound is $bound(r_x) = x$. ◁

The *dependency graph* G_Π of Π has nodes $atoms(\Pi)$, and an arc xy (where x and y are atoms) for each rule $r \in rules^\vee(\Pi)$ such that $x \in H(r)$ and $y \in B(r)$, and for each rule $r \in rules^\Sigma(\Pi)$ such that $x = id(r)$ and $(w, y) \in elem(r)$ for some integer w. An atom is *recursive* in Π if it is involved in a cycle of G_Π. In the following every program Π is assumed to satisfy the following property: if $r \in \Pi$ is of the form (2), $id(r)$ is not recursive in Π. Note that Π_{run} has such a property.

An *assignment I* is a set of literals (different from $\perp$ and $\top$) such that $I \cap \overline{I} = \emptyset$; literals in I are true, literals in $\overline{I}$ are false, and all other literals are undefined. An *interpretation I* for a program

Π is an assignment such that $I \cup \bar{I} = atoms(\Pi) \cup \overline{atoms(\Pi)}$. Relation $\models$ is inductively defined as follows: $I \not\models \bot$ (hence, $I \models \top$); for $p \in \mathscr{A}$, $I \models p$ if $p \in I$; $I \models {\sim}p$ if $I \not\models p$; for a rule r of the form (1), $I \models B(r)$ if $I \models \ell$ for all $\ell \in B(r)$, $I \models H(r)$ if $I \models p$ for some $p \in H(r)$, and $I \models r$ if $I \models H(r)$ whenever $I \models B(r)$; for a rule r of the form (2), $I \models sum(r)$ if $\sum_{(w,\ell) \in elem(r),\, I \models \ell} w \geq bound(r)$, and $I \models r$ if $I \models id(r)$ whenever $I \models sum(r)$; for a program Π, $I \models \Pi$ if $I \models r$ for all $r \in \Pi$. For any expression π, if $I \models \pi$, we say that I is a *model* of π.

The *reduct* Π^I of a program Π with respect to an interpretation I comprises the following rules: for each rule $r \in rules^\vee(\Pi)$ such that $I \models B(r)$, Π^I contains a rule r^I of the form (1) with $H(r^I) = H(r)$, and $B(r^I) = B(r) \cap \mathscr{A}$; for each rule $r \in rules^\Sigma(\Pi)$ such that $I \models sum(r)$, Π^I contains a rule r^I of the form (2) with $id(r^I) = id(r)$, $bound(r^I) = bound(r)$, and $elem(r) = \{(w,\ell) \in elem(r) \mid I \models \ell,\ \ell \in \mathscr{A}\} \cup \{(\sum_{(w,\ell) \in elem(r),\, I \models \ell,\ \ell \notin \mathscr{A}} w, \top)\}$. An interpretation I is a *stable model* of a program Π if $I \models \Pi$ and there is no $J \subset I$ such that $J \models \Pi^I$. Let $SM(\Pi)$ denote the set of stable models of Π. For instance, for Π_{run} from Example 2.1, $SM(\Pi_{run})$ comprises the following models: $\{n_2, n_5\}$, $\{n_2, p_5, a_1, a_2, a_3, a_5\}$, $\{p_2, n_5, a_1, a_2\}$, and $\{p_2, p_5, a_1, a_2, a_3, a_5, a_6, a_7\}$.

2.2 *Stable model search and propagators*

Stable model search is implemented in modern ASP solvers as a *conflict-driven clause learning* (CDCL) algorithm (Gebser et al. 2012), which is based on the pattern *choose-propagate-learn*. Intuitively, the idea is to build a stable model step-by-step starting from an empty assignment I. At each step of computation, a branching literal is heuristically *chosen* to be added in I, and *propagated* so to add deterministic consequences to I, if possible. Otherwise, if the complement of a deterministic consequence already belongs to I, a *conflict* is identified. Each deterministic consequence ℓ added to I is also associated with a set of *reasons*, essentially the true literals causing the addition of ℓ to I. Conflicts are analyzed to *learn* new clauses by applying backward resolution on the reasons of the conflictual literals; while performing backward resolution, literals previously added to I are removed, until the learned clause causes the addition of a new deterministic consequence that drives the search into a different branch. This process is repeated until either I is a stable model, or the empty clause is learned, meaning that the input program has no stable models.

A *propagator* is a module for computing deterministic consequences of an assignment. The simplest of such modules is *unit propagation*: unit propagation adds a literal ℓ to the current assignment I if the input program contains a rule r of the form (1) such that r can be satisfied only by $I \cup \{\ell\}$. More formally, let $C(r) := \{p_1, \ldots, p_m, \overline{\ell_1}, \ldots, \overline{\ell_n}\}$ be the clause representation of r. A literal $\ell \in C(r) \setminus I$ is unit propagated with respect to I and r if $\overline{C(r) \setminus \{\ell\}} \subseteq I$; let $reasons(\ell)$ be $\overline{C(r) \setminus \{\ell\}}$ in this case. For instance, atom p_2 is unit propagated with respect to the assignment $\{a_1, a_2, {\sim}n_2\}$ and rule g_2 from Example 2.1, and $reasons(p_2)$ is $\{{\sim}n_2\}$.

Concerning rules of the form (2), specific propagators have been proposed in the literature (Gebser et al. 2009; Faber et al. 2011), here referred to as *aggregate propagators*. The idea is that four types of inferences are associated with rules of the form (2). In particular, given a rule r of the form (2), and an assignment I, the following literals are inferred (if not already in I):

(A1) $id(r)$, whenever $\sum_{(w,\ell) \in elem(r),\, \ell \in I} w \geq bound(r)$, with $reasons(id(r))$ comprising true literals in $elem(r)$, that is, $\{\ell \mid (w,\ell) \in elem(r),\ \ell \in I\}$;

(A2) $\overline{id(r)}$, whenever $\sum_{(w,\ell) \in elem(r),\, \overline{\ell} \notin I} w < bound(r)$, with $reasons(\overline{id(r)})$ comprising the complements of false literals in $elem(r)$, that is, $\{\overline{\ell} \mid (w,\ell) \in elem(r),\ \overline{\ell} \in I\}$;

(A3) any ℓ such that $(w,\ell) \in elem(r)$, whenever both $id(r) \in I$ and $\sum_{(w',\ell')\in elem(r)\setminus\{(w,\ell)\},\ \overline{\ell'}\notin I} w'$
   $< bound(r)$, with $reasons(\ell)$ comprising $id(r)$ and the complements of false literals in
   $elem(r)$, that is, $\{id(r)\} \cup \{\overline{\ell'} \mid (w',\ell') \in elem(r),\ \overline{\ell'} \in I\}$;

(A4) any ℓ such that $(w,\overline{\ell}) \in elem(r)$, whenever both $\overline{id(r)} \in I$ and $\sum_{(w',\ell')\in elem(r)\setminus\{(w,\overline{\ell})\},\ \ell'\in I} w'$
   $\geq bound(r) - w$, with $reasons(\ell)$ comprising $\overline{id(r)}$ and true literals in $elem(r)$, that is,
   $\{\overline{id(r)}\} \cup \{\ell' \mid (w',\ell') \in elem(r),\ \ell' \in I\}$.

For instance, for the rules from Example 2.1, if I is $\{a_1,a_2,\sim p_2,p_5\}$, then a_3,a_5 are inferred by
(A1), and $\sim a_6, \sim a_7$ are inferred by (A2); in this case, $reasons(a_3) = reasons(a_5) = \{p_5\}$, and
$reasons(\sim a_6) = reasons(\sim a_7) = \{\sim p_2\}$. Similarly, if I is $\{a_5,\sim a_6\}$, then p_5 is inferred by (A3),
and $\sim p_2$ is inferred by (A4); in this case, $reasons(p_5) = \{a_5\}$, and $reasons(\sim p_2) = \{\sim a_6\}$.

3 Handling ASP programs with shared aggregate sets

As discussed in the introduction, ASP programs may contain several rules of the form (2) with
the same elements but different bounds. In such cases, aggregate propagators introduced in the
previous section are subject to some intrinsic inefficiency. First of all, very similar data structures
have to be stored. Second, and more important, in order to check the applicability of the four
inference rules described in the previous section, redundant computation is performed because
the same sum is computed several times. The objective of this section is to introduce a new
propagator for circumventing such inefficiencies. The new propagator actually comes with a few
simplifications that are applied to the input program in order to remove trivially unsatisfiable
sums, to normalize bounds, to identify some equivalent sums, and to add a few entailed integrity
constraints that are not easy to discover by CDCL.

Additional notation is introduced here to simplify the presentation of the simplifications. For
a program Π, and a rule $r \in rules^{\Sigma}(\Pi)$, define $sums(r) := \{\sum_{(\ell,w)\in S} w \mid \emptyset \subset S \subseteq elem(r)\}$, the set
of possible nonzero sums for the elements of r, and define $next(r,\Pi)$ to be

$$\underset{id(r'):r'\in rules^{\Sigma}(\Pi),\ elem(r')=elem(r),\ bound(r')>bound(r)}{\arg\min} bound(r')$$

if it exists, and $\perp$ otherwise; that is, $next(r,\Pi)$ is the id of the rule with the next possible sum (if it
exists; if there are several, one is selected according to some order, as for example lexicographical
order). The following rewritings, from the first to the last, are applied to every rule $r \in rules^{\Sigma}(\Pi)$:

 (i) if $bound(r) > \sum_{(w,\ell)\in elem(r)} w$ then $\Pi := (\Pi \setminus \{r\}) \cup \{\perp \leftarrow id(r)\}$;
 (ii) $bound(r) := \min(\{b \in sums(r) \mid b \geq bound(r)\})$;
 (iii) if there is $r' \in rules^{\Sigma}(\Pi)$ such that $sum(r') = sum(r)$, $\Pi := (\Pi \setminus \{r\}) \cup \{id(r) \leftarrow id(r')\}$;
 (iv) $\Pi := \Pi \cup \{\perp \leftarrow \sim id(r), next(r,\Pi)\}$.

Intuitively, rewriting (i) eliminates from the input program all trivially unsatisfiable sums, that
is, those which are unsatisfied even when all their elements are true. Rewriting (ii) normalizes
bounds of sums, so that the bound of each rule can be obtained by summing some weights
occurring in the rule. After that, rewriting (iii) identifies rules with the same sum, and leave a
single copy of the aggregation, while identifiers of removed sums are forced to preserve their
semantics by a new, simple rule. Finally, rewriting (iv) introduces entailed integrity constraints
that are difficult to identify otherwise. Note that rewritings (i), (ii) and (iii) can be applied, in this
order, one rule at time, while rewriting (iv) has to be delayed until the other rewritings have been
applied to all rules of the program.

Theorem 3.1
Let Π be a program, and Π' be obtained by applying (i)–(iv) to Π. $SM(\Pi) = SM(\Pi')$ holds.

Proof
Let $\Pi_0, \ldots, \Pi_n$ ($n \geq 0$) be the sequence of programs obtained by applying (i)–(iv), that is, $\Pi_0 = \Pi$, $\Pi_n = \Pi'$, and for all $j \geq 0$ Π_{j+1} is obtained from Π_j by a single application of a rewriting among (i)–(iv). We shall show that $SM(\Pi_j) = SM(\Pi_{j+1})$, for any $j \geq 0$.

(i) Let Π_{j+1} be obtained by applying (i) to Π_j.

($\subseteq$) Let $I \in SM(\Pi_j)$. Since $bound(r) > \sum_{(w,\ell) \in elem(r)} w$, $I \not\models sum(r)$ holds, and since $id(r)$ does not occur in any other rule head, $I \not\models id(r)$ holds as well (otherwise, I could not be a stable model). Therefore, $I \models \bot \leftarrow id(r)$, and $I \models \Pi_{j+1}$. Moreover, $\Pi_j^I = \Pi_{j+1}^I$, and therefore $I \in SM(\Pi_{j+1})$.

($\supseteq$) Let $I \in SM(\Pi_{j+1})$. Since $\bot \leftarrow id(r)$ belongs to Π_{j+1}, $I \not\models id(r)$, and therefore $I \models \Pi_j$. Since $bound(r) > \sum_{(w,\ell) \in elem(r)} w$, $I \not\models sum(r)$ holds, and therefore $\Pi_j^I = \Pi_{j+1}^I$. Hence, $I \in SM(\Pi_j)$.

(ii) Let Π_{j+1} be obtained by applying (ii) to Π_j. For any interpretation I, $I \models sum(r)$ if and only if $I \models \text{SUM}(elem(r)) \geq \min(\{b \in sums(r) \mid b \geq bound(r)\})$, that is, the two aggregates define the same boolean function.

(iii) Let Π_{j+1} be obtained by applying (iii) to Π_j.

($\subseteq$) Let $I \in SM(\Pi_j)$. If $I \not\models sum(r)$, then $I \not\models sum(r')$, $I \not\models id(r)$, and $I \not\models id(r')$; hence, $I \models \Pi_{j+1}$, and $\Pi_j^I = \Pi_{j+1}^I$, which imply $I \in SM(\Pi_{j+1})$.
Otherwise, if $I \models sum(r)$, then $I \models sum(r')$, $I \models id(r)$, and $I \models id(r')$. Hence, $I \models \Pi_{j+1}$ holds. It remains to show that $J \subset I$ implies $J \not\models \Pi_{j+1}^I$. Since $I \in SM(\Pi_j)$ by assumption, $J \not\models \Pi_j^I$ holds. Let J be such that $J \models \Pi_j^I \cap \Pi_{j+1}^I$ (otherwise it is trivial); hence, there is $r'' \in \Pi_j^I \setminus \Pi_{j+1}^I$ such that $J \not\models r''$, and such a rule has been obtained from r. Thus, $J \models sum(r)$, and $J \not\models id(r)$. In this case, $J \not\models id(r) \leftarrow id(r')$ because $I \models id(r')$ follows from $J \models \Pi_j^I \cap \Pi_{j+1}^I$ and $J \models sum(r)$. It turns out that $I \in SM(\Pi_{j+1})$ holds.

($\supseteq$) Let $I \in SM(\Pi_{j+1})$. If $I \not\models sum(r)$, then $I \not\models sum(r')$, $I \not\models id(r)$, and $I \not\models id(r')$; hence, $I \models \Pi_j$, and $\Pi_j^I = \Pi_{j+1}^I$, which imply $I \in SM(\Pi_{j+1})$.
Otherwise, if $I \models sum(r)$, then $I \models sum(r')$, $I \models id(r)$, and $I \models id(r')$. Hence, $I \models \Pi_j$ holds. It remains to show that $J \subset I$ implies $J \not\models \Pi_j^I$. Since $I \in SM(\Pi_{j+1})$ by assumption, $J \not\models \Pi_{j+1}^I$ holds. Let J be such that $J \models \Pi_j^I \cap \Pi_{j+1}^I$ (otherwise it is trivial); hence, there is $r'' \in \Pi_{j+1}^I \setminus \Pi_j^I$ such that $J \not\models r''$, and such a rule is $id(r) \leftarrow id(r')$, that is, $J \not\models id(r)$ and $J \models id(r')$. If $J \models sum(r)$, then $J \not\models r$, and therefore $J \not\models \Pi_j^I$. Otherwise, if $J \not\models sum(r)$, then $J \setminus \{id(r')\} \models \Pi_{j+1}^I$, which would contradict the assumption that $I \in SM(\Pi_{j+1})$. It turns out that $I \in SM(\Pi_j)$ holds.

(iv) Let Π_{j+1} be obtained by applying (iv) to Π_j.

($\subseteq$) Let $I \in SM(\Pi_j)$. If $I \models id(r)$, then $I \models \Pi_{j+1}$ and $\Pi_j^I = \Pi_{j+1}^I$, so $I \in SM(\Pi_{j+1})$. Otherwise, let $I \not\models id(r)$. Therefore, $I \not\models \text{SUM}(elem(r)) \geq b$ for all $b \geq bound(r)$. Hence, $I \not\models next(r, \Pi)$, and therefore $I \models \bot \leftarrow \sim id(r), next(r, \Pi)$. It turns out that $I \models \Pi_{j+1}$ and $\Pi_j^I = \Pi_{j+1}^I$, so $I \in SM(\Pi_{j+1})$.

($\supseteq$) Let $I \in SM(\Pi_{j+1})$. Since $\Pi_j \subset \Pi_{j+1}$, $I \models \Pi_j$. Moreover, $\Pi_j^I = \Pi_{j+1}^I$, and therefore $I \in SM(\Pi_j)$.

The proof is complete. $\qquad\qquad\square$

Example 3.1 (Continuing Example 2.1)
Note that, for all $x \in \{1,2,3,5,6,7\}$, $sums(r_x) = \{2,5,7\}$. The application of (i)–(iv) produces the following program Π'_{run}:

$$
\begin{array}{llllll}
g_2: & p_2 \vee n_2 \leftarrow & g_5: & p_5 \vee n_5 \leftarrow & & \\
r'_1: & a_1 \leftarrow a_2 & r_2: & a_2 \leftarrow \mathrm{SUM}\{2:p_2, 5:p_5\} \geq 2 & s_2: & \bot \leftarrow \sim a_2, a_5 \\
r'_3: & a_3 \leftarrow a_5 & r_5: & a_5 \leftarrow \mathrm{SUM}\{2:p_2, 5:p_5\} \geq 5 & s_5: & \bot \leftarrow \sim a_5, a_7 \\
r'_6: & a_6 \leftarrow a_7 & r_7: & a_7 \leftarrow \mathrm{SUM}\{2:p_2, 5:p_5\} \geq 7 & s_7: & \bot \leftarrow \sim a_7, \bot
\end{array}
$$

For instance, the application of (ii) modifies the bound of r_1 from 1 to 2, so that it now occurs in $sums(r_1)$; the resulting rule is then processed by (iii), and replaced by r'_1. Rules s_2, s_5, and s_7 are added by the application of (iv). ◁

Shared aggregate sets are identified during the simplification process described above. For each of these shared aggregate sets, a specific propagator is instantiated. The new propagator, referred to as *shared aggregate set propagator*, has data structures specifically conceived to implement the inference rules of several aggregate propagators avoiding redundant computation. More formally, for a program Π, let X be a maximal subset of $rules^{\Sigma}(\Pi)$ such that all $r, r' \in X$ satisfy $elem(r) = elem(r')$. A shared aggregate set propagator for X is associated with the following sets: $elem(X)$, storing the shared set of elements, that is, $elem(r)$, for any $r \in X$; and $bounds(X)$, storing identifiers and bounds, that is, $\{(bound(r), id(r)) \mid r \in X\}$. Moreover, the following functions are used by the propagator when applied to an assignment I:

- $\mathrm{min\_sum}(X,I) := \sum_{(w,\ell) \in elem(X),\ \ell \in I} w$, the smallest value of the sum among those assigned by interpretations extending the current assignment;
- $\mathrm{max\_sum}(X,I) := \sum_{(w,\ell) \in elem(X),\ \overline{\ell} \notin I} w$, the greatest value of the sum among those assigned by interpretations extending of the current assignment;
- $\mathrm{lower\_bound}(X,I) := \max(\{b \mid (b,\ell) \in bounds(X),\ \ell \in I\} \cup \{0\})$, a value that must be reached by the sum in any extension of the current assignment;
- $\mathrm{upper\_bound}(X,I) := \min(\{b \mid (b,\ell) \in bounds(X),\ \overline{\ell} \in I\} \cup \{+\infty\})$, a value that must not be reached by the sum in any extension of the current assignment.

Example 3.2 (Continuing Example 3.1)
Let X be $rules^{\Sigma}(\Pi'_{run})$. The shared aggregate set propagator associated with X has $elem(X) = \{(2,p_2),(5,p_5)\}$, and $bounds(X) = \{(2,a_2),(5,a_5),(7,a_7)\}$. For the empty assignment, $\mathrm{min\_sum}(X,\emptyset) = 0$, $\mathrm{max\_sum}(X,\emptyset) = 7$, $\mathrm{lower\_bound}(X,\emptyset) = 0$, and $\mathrm{upper\_bound}(X,\emptyset) = +\infty$; for $I = \{p_2, \sim p_5, a_5, \sim a_7\}$, $\mathrm{min\_sum}(X,I) = 2$, $\mathrm{max\_sum}(X,I) = 2$, $\mathrm{lower\_bound}(X,I) = 5$, and $\mathrm{upper\_bound}(X,I) = 7$. ◁

All inferences of shared aggregate set propagators are shown as function OnLiteralTrue, whose input is the set X of rules handled by the propagator, an assignment I, and a literal ℓ, where ℓ is the last literal added to I. The output of OnLiteralTrue(X,I,ℓ) is a set of pairs of the form (ℓ', R), where ℓ' is an inferred literal, and R a set of literals; the solver has to add ℓ' to I, and to assign R to $reasons(\ell')$. Specifically, OnLiteralTrue(X,I,ℓ) may infer something in the following cases:

Case 1. If ℓ, the last literal added to I, occurs in an element (w,ℓ) of the aggregate set (line 1), then the minimum value that can be assigned to the sum is augmented by w, and therefore identifiers associated with bounds being less or equal than such a value can be inferred; the reasons of the inferred literals are the true literals occurring in the aggregate set (line 2); such an inference is

---

Function OnLiteralTrue(X, I, ℓ)

---

1 **if** $(w, \ell) \in elem(X)$ **then**
2    $\quad R := \{\ell' \mid (w', \ell') \in elem(X),\ \ell' \in I\}$;
3    $\quad R' := R \cup \{\overline{\ell'} \mid (\text{upper\_bound}(X, I), \ell') \in bounds(X)\}$;
4    $\quad$ **return** $\{(\ell', R) \mid (b, \ell') \in bounds(X),\ \ell' \notin I,\ b \leq \text{min\_sum}(X, I)\} \cup$
      $\quad\quad \{(\overline{\ell'}, R') \mid (w', \ell') \in elem(X),\ \overline{\ell'} \notin I,\ w' \geq \text{upper\_bound}(X, I) - \text{min\_sum}(X, I)\}$;

5 **if** $(w, \overline{\ell}) \in elem(X)$ **then**
6    $\quad R := \{\overline{\ell'} \mid (w', \ell') \in elem(X),\ \overline{\ell'} \in I\}$;
7    $\quad R' := R \cup \{\ell' \mid (\text{lower\_bound}(X, I), \ell') \in bounds(X)\}$;
8    $\quad$ **return** $\{(\overline{\ell'}, R) \mid (b, \ell') \in bounds(X),\ \overline{\ell'} \notin I,\ b > \text{max\_sum}(X, I)\} \cup$
      $\quad\quad \{(\ell', R') \mid (w', \ell') \in elem(X),\ \ell' \notin I,\ w' > \text{max\_sum}(X, I) - \text{lower\_bound}(X, I)\}$;

9 **if** $(b, \ell) \in bounds(X)$ **and then**
10   $\quad R := \{\overline{\ell'} \mid (w', \ell') \in elem(X),\ \overline{\ell'} \in I\} \cup \{\ell\}$;
11   $\quad$ **return** $\{(\ell', R) \mid (w', \ell') \in elem(X),\ \ell' \notin I,\ w' > \text{max\_sum}(X, I) - \text{lower\_bound}(X, I)\}$;

12 **if** $(b, \overline{\ell}) \in bounds(X)$ **and then**
13   $\quad R := \{\ell' \mid (w', \ell') \in elem(X),\ \ell' \in I\} \cup \{\ell\}$;
14   $\quad$ **return** $\{(\overline{\ell'}, R) \mid (w', \ell') \in elem(X),\ \overline{\ell'} \notin I,\ w' \geq \text{upper\_bound}(X, I) - \text{min\_sum}(X, I)\}$;

15 **return** $\emptyset$;

---

analogous to (A1) for aggregate propagators. Moreover, every literal in the aggregate set whose truth would lead to a conflict are inferred false, where the conflict would arise if the addition of the weight of the inferred literal exceeds the currently known upper bound; the reasons of the inferred literals are the true literals in the aggregate set and the complement of the literal associated to the currently known upper bound (line 3); such an inference is analogous to (A4) for aggregate propagators.

Case 2. If the complement of ℓ occurs in an element $(w, \overline{\ell})$ of the aggregate set (line 5), then the maximum value that can be assigned to the sum is decremented by w, and therefore identifiers associated with bounds being greater than such a value can be inferred false; the reasons of the inferred literals are the complements of the false literals in the aggregate set (line 6); such an inference is analogous to (A2) for aggregate propagators. Moreover, in this case every literal in the aggregate set whose falsity would lead to a conflict are inferred true, where the conflict would arise if the subtraction of the weight of the inferred literal is under the currently known lower bound; the reasons of the inferred literals are the complements of the false literals in the aggregate set and the literal associated to the currently known upper bound (line 7); such an inference is analogous to (A3) for aggregate propagators.

Case 3. If ℓ occurs in an element (b, ℓ) of $bounds(X)$ (line 9), then the lower bound is possibly increased, and therefore every literal in the aggregate set whose falsity would lead to a conflict are inferred true, where the conflict would arise if the subtraction of the weight of the inferred literal does not reach the currently known lower bound; the reasons of the inferred literals are the complements of the false literals in the aggregate set and the literal ℓ (line 10); such an inference is analogous to (A3) for aggregate propagators.

Case 4. If the complement of ℓ occurs in an element $(b, \overline{\ell})$ of $bounds(X)$ (line 12), then the upper bound is possibly decreased, and therefore every literal in the aggregate set whose truth would

lead to a conflict are inferred false, where the conflict would arise if the addition of the weight of the inferred literal exceeds the currently known upper bound; the reasons of the inferred literals are the true literals in the aggregate set and the literal ℓ (line 13); such an inference is analogous to (A4) for aggregate propagators.

Theorem 3.2
Let Π be a program obtained by applying rewritings (i)–(iv), X be a maximal subset of $rules^{\Sigma}(\Pi)$ such that $r, r' \in X$ satisfy $elem(r) = elem(r')$, I be an assignment, and ℓ be the last literal added to I. A literal ℓ' is inferred by OnLiteralTrue(X, I, ℓ) with reasons R if and only if ℓ' is inferred with reasons R by applying (A1)–(A4) to some $r \in X$ and assignment I.

Proof
Let ℓ' be inferred by OnLiteralTrue(X, I, ℓ). Hence, $\ell' \notin I$ holds. Four cases are possible.

1. If there is $(w, \ell) \in elem(X)$, we further distinguish two cases:

   - There is $(b, \ell') \in bounds(X)$ such that $b \leq$ min_sum(X, I). Hence, there is $r \in X$ such that $bound(r) = b$, and $id(r) = \ell'$. Since $b \leq$ min_sum(X, I), (A1) derives $id(r)$. Moreover, both propagators set $reasons(\ell') = \{\ell'' \mid (w'', \ell'') \in elem(X), \ell'' \in I\}$.
   - There is $(w', \overline{\ell'}) \in elem(X)$ such that $w' \geq$ upper_bound$(X, I) -$ min_sum(X, I). Hence, there is $r \in X$ such that $bound(r) = \min(\{b \mid (b, \ell'') \in bounds(X), \overline{\ell''} \in I\})$. Thus, $\overline{id(r)} \in I$, and (A4) derives ℓ'. Moreover, both propagators set $reasons(\ell') = \{\overline{id(r)}\} \cup \{\ell'' \mid (w'', \ell'') \in elem(X), \ell'' \in I\}$.

2. If there is $(w, \overline{\ell}) \in elem(X)$, we further distinguish two cases:

   - There is $(b, \overline{\ell'}) \in bounds(X)$ such that $b >$ max_sum(X, I). Hence, there is $r \in X$ such that $bound(r) = b$, and $id(r) = \overline{\ell'}$. Since $b >$ max_sum(X, I), (A2) derives $\overline{id(r)} = \ell'$. Moreover, both propagators set $reasons(\ell') = \{\overline{\ell''} \mid (w'', \ell'') \in elem(X), \overline{\ell''} \in I\}$.
   - There is $(w', \ell') \in elem(X)$ such that $w' >$ max_sum$(X, I) -$ lower_bound(X, I). Hence, there is $r \in X$ such that $bound(r) = \max(\{b \mid (b, \ell'') \in bounds(X), \ell'' \in I\})$. Thus, $id(r) \in I$, and (A3) derives ℓ'. Moreover, both propagators set $reasons(\ell') = \{id(r)\} \cup \{\overline{\ell''} \mid (w'', \ell'') \in elem(X), \overline{\ell''} \in I\}$.

3. If there is $(b, \ell) \in bounds(X)$, then there is $(w', \ell') \in elem(X)$ such that $w' >$ max_sum$(X, I) -$ lower_bound(X, I). Hence, there is $r \in X$ such that $bound(r) = \max(\{b \mid (b, \ell'') \in bounds(X), \ell'' \in I\})$. Thus, $id(r) \in I$, and (A3) derives ℓ'. Moreover, both propagators set $reasons(\ell') = \{id(r)\} \cup \{\overline{\ell''} \mid (w'', \ell'') \in elem(X), \overline{\ell''} \in I\}$.

4. If there is $(b, \overline{\ell}) \in bounds(X)$, there is $(w', \overline{\ell}) \in elem(X)$ such that $w' \geq$ upper_bound$(X, I) -$ min_sum(X, I). Hence, there is $r \in X$ such that $bound(r) = \min(\{b \mid (b, \ell'') \in bounds(X), \overline{\ell''} \in I\})$. Thus, $\overline{id(r)} \in I$, and (A4) derives ℓ'. Moreover, both propagators set $reasons(\ell')$ equals to $\{\overline{id(r)}\} \cup \{\ell'' \mid (w'', \ell'') \in elem(X), \ell'' \in I\}$.

   As for the other direction, let ℓ' be inferred by (A1)–(A4) applied on I, and not derivable from $I \setminus \{\ell\}$. We distinguish four cases.

1. Literal ℓ' is inferred by (A1). Hence, there is $r \in X$ such that $id(r) = \ell'$, $\sum_{(w'', \ell'') \in elem(r), \ell'' \in I} w'' \geq bound(r)$, and $\sum_{(w'', \ell'') \in elem(r), \ell'' \in I \setminus \{\ell\}} w'' < bound(r)$. Thus, there is $(w, \ell) \in elem(X)$, and ℓ' is derived by OnLiteralTrue(X, I, ℓ) at line 4 because $(bound(r), \ell') \in bounds(X)$ and $bound(r) \leq$ min_sum(X, I). Moreover, both propagators set $reasons(\ell') = \{\ell'' \mid (w'', \ell'') \in elem(X), \ell'' \in I\}$.

2. Literal ℓ' is inferred by (A2). Hence, there is $r \in X$ such that $\overline{id(r)} = \ell'$, $\sum_{(w'',\ell'')\in elem(r),\overline{\ell''}\notin I} w'' < bound(r)$, and $\sum_{(w'',\ell'')\in elem(r),\overline{\ell''}\notin I\setminus\{\ell\}} w'' \geq bound(r)$. Thus, there is $(w,\overline{\ell}) \in elem(X)$, and ℓ' is derived by OnLiteralTrue(X,I,ℓ) at line 8 because $(bound(r),\overline{\ell'}) \in bounds(X)$ and $bound(r) > max\_sum(X,I)$. Moreover, both propagators set $reasons(\ell')$ equals to $\{id(r)\} \cup \{\overline{\ell''} \mid (w'',\ell'') \in elem(X),\overline{\ell''} \in I\}$.

3. Literal ℓ' is inferred by (A3). Hence, there is $r \in X$ such that $(w',\ell') \in elem(r)$, $id(r) \in I$, $\sum_{(w'',\ell'')\in elem(r)\setminus\{(w',\ell')\},\overline{\ell''}\notin I} w'' < bound(r)$, and either $id(r) \notin I\setminus\{\ell\}$ or the following inequality holds: $\sum_{(w'',\ell'')\in elem(r)\setminus\{(w',\ell')\},\overline{\ell''}\notin I\setminus\{\ell\}} w'' \geq bound(r)$.

   - If $id(r) \notin I\setminus\{\ell\}$, then $id(r) = \ell$, and therefore ℓ' is derived by OnLiteralTrue(X,I,ℓ) at line 11 because $(bound(r),\ell) \in bounds(X)$, $(w',\ell') \in elem(X)$ and w' is greater than $max\_sum(X,I) - lower\_bound(X,I)$. Moreover, both propagators set $reasons(\ell')$ equals to $\{id(r)\} \cup \{\overline{\ell''} \mid (w'',\ell'') \in elem(X),\overline{\ell''} \in I\}$.
   - If $\sum_{(w'',\ell'')\in elem(r)\setminus\{(w',\ell')\},\overline{\ell''}\notin I\setminus\{\ell\}} w'' \geq bound(r)$, there is $(w,\overline{\ell}) \in elem(X)$. Hence, ℓ' is derived by OnLiteralTrue(X,I,ℓ) at line 8 because there is $(w',\ell') \in elem(X)$ such that $w' > max\_sum(X,I) - lower\_bound(X,I)$. Moreover, both propagators set $reasons(\ell')$ equals to $\{id(r)\} \cup \{\overline{\ell''} \mid (w'',\ell'') \in elem(X),\overline{\ell''} \in I\}$.

4. Literal ℓ' is inferred by (A4). Hence, there is $r \in X$ such that $(w',\overline{\ell'}) \in elem(r)$, $\overline{id(r)} \in I$, $\sum_{(w'',\ell'')\in elem(r)\setminus\{(w',\overline{\ell'})\},\ell''\in I} w'' \geq bound(r) - w'$, and either $\overline{id(r)} \notin I\setminus\{\ell\}$ or the inequality $\sum_{(w'',\ell'')\in elem(r)\setminus\{(w',\overline{\ell'})\},\ell''\in I\setminus\{\ell\}} w'' < bound(r) - w'$ holds.

   - If $\overline{id(r)} \notin I\setminus\{\ell\}$, then $\overline{id(r)} = \ell$, and therefore ℓ' is derived by OnLiteralTrue(X,I,ℓ) at line 14 because there is $(w',\overline{\ell'}) \in elem(X)$ such that the following inequality holds: $w' \geq upper\_bound(X,I) - min\_sum(X,I)$. Moreover, both propagators set $reasons(\ell')$ equals to $\{\overline{id(r)}\} \cup \{\ell'' \mid (w'',\ell'') \in elem(X),\ell'' \in I\}$.

The proof is complete. □

Example 3.3 (*Continuing Example 3.2*)
Let X be $rules^\Sigma(\Pi'_{run})$, and recall that $elem(X) = \{(2,p_2),(5,p_5)\}$, and $bounds(X) = \{(2,a_2), (5,a_5),(7,a_7)\}$. OnLiteralTrue$(X,\{\sim a_7,\sim n_2,p_2\},p_2)$ returns $(a_2,\{p_2\})$ and $(\sim p_5,\{p_2,\sim a_7\})$ because of case 1; indeed, literal a_2 is inferred because $min\_sum(\{\sim a_7,\sim n_2,p_2\}) = 2$, and literal $\sim p_5$ is inferred because $upper\_bound(\{\sim a_7,\sim n_2,p_2\}) - min\_sum(\{\sim a_7,\sim n_2,p_2\}) = 5$. Similarly, OnLiteralTrue$(X,\{a_2,n_5,\sim p_5\},\sim p_5)$ returns $(\sim a_5,\{\sim p_5\})$, $(\sim a_7,\{\sim p_5\})$ and $(p_2,\{\sim p_5,a_2\})$ because of case 2. OnLiteralTrue$(X,\{a_5\},a_5)$ returns the pair $(p_5,\{a_5\})$ because of case 3. Finally, OnLiteralTrue$(X,\{\sim a_5\},\sim a_5)$ returns the pair $(\sim p_5,\{\sim a_5\})$ because of case 4. ◁

4 Implementation and experiments

Simplifications and the new propagator presented in Section 3 have been implemented in WASP (Alviano et al. 2015), a modern ASP solver implementing the CDCL algorithm. Some details of the implementation are given in Section 4.1, and an application scenario is presented in Section 4.2. WASP already implemented the aggregate propagator described in Section 2.2, and therefore it is an ideal framework for evaluating empirically how aggregate set propagators impact the performance of stable model search. Such an evaluation is reported in Section 4.3.

4.1 Implementation

The implementation of the shared aggregate set propagator in WASP introduces a few additional optimizations. First of all, function OnLiteralTrue is called only if at least one of the conditions at lines 1, 5, 9, and 12 is true. Actually, only one of these conditions can be true for a literal ℓ, which is trivially the case for lines 9 and 12, and guaranteed for lines 1 and 5 by performing an additional rewriting during simplifications: if (w, ℓ) and $(w', \overline{\ell})$ occur in a rule $r \in rules^{\Sigma}(\Pi)$, and $w \geq w'$, then r is replaced by a rule r' of the form (2) such that $id(r') := id(r)$, $elem(r') := (elem(r) \setminus \{(w, \ell), (w', \overline{\ell})\}) \cup \{(w - w', \ell) \mid w - w' > 0\}$, and $bound(r') := bound(r) - w'$.

The second optimization concerns the implementation of the functions min_sum, max_sum, lower_bound, and upper_bound, which are heavily used by OnLiteralTrue(X,I,ℓ). Instead of computing their values by performing a complete iteration on the sets $elem(X)$ and $bounds(X)$, their values with respect to the current assignment I are stored in local variables, which are updated when new literals are assigned, and during backtracking.

Finally, a third optimization regards the computation of reasons, which are used by the solver only in case some inferred literal is involved in the conflict analysis. Hence, it is natural to compute reasons only when they are required by the solver. In order to obtain such a behavior, a *trail* of literals is used by each shared aggregate set propagator to store assigned literals of interest. Specifically, when OnLiteralTrue(X,I,ℓ) is invoked, literal ℓ is added to the trail of set X. If the conflict analysis requires the reasons of a literal inferred by a shared aggregate set propagator, then an iteration on the trail is sufficient to reconstruct the set of literals shown in function OnLiteralTrue.

4.2 Component Assignment

The input of the *Component Assignment* problem is a tuple $(C, p, U, B, \mathscr{I}, \mathscr{R})$ defined as follows: C is a set of components (of a computer); p is a function mapping each component $c \in C$ to its price; U is a set of users; B is a function mapping each user $u \in U$ to an interval of possible expense; $\mathscr{I}$ is a set of sets of jointly incompatible components; $\mathscr{R}$ is a set of sets of required components. The goal is to compute a partial assignment $f : C \nrightarrow U$ of components to users such that the following conditions are satisfied: for all $u \in U$, $\sum_{c \in C, \ f(c)=u} p(c) \in B(u)$ holds (all users expend a permitted amount); for all sets $\mathscr{I}' \in \mathscr{I}$ there are $c, c' \in \mathscr{I}'$ such that $f(c) \neq f(c')$ (no user can be assigned a set of incompatible components); for all sets $\mathscr{R}' \in \mathscr{R}$, and for all users $u \in U$, there is a component $c \in \mathscr{R}'$ such that $f(c) = u$ (at least one component in each requirement is assigned to every configuration). Moreover, the price of each configuration has to be provided in output.

Figure 1 reports an ASP encoding for Component Assignment. Recall that this is an abstraction of a real world problem, actually originated in the area of medical informatics dealing with the assignment of patients to operation rooms. The formulation given here is an excerpt focusing on the main source of inefficiency discovered by analyzing the encoding for the original problem. Specifically, an aggregate similar to `COST = #sum{P,C:assign(C,U),component(C,P)}` was the culprit of the inefficiency of ASP solvers, and was subsequently subject to complex optimization (essentially, a minimax preference), which is out of the scope of this paper.

```
%guess an assignment of components : each component to at most one user
{assign(C,U) : user(U,_,_)} <= 1 :- component(C,_).
%check budget
cost(U,COST) :- user(U,_,_), COST = #sum{P,C : assign(C,U),component(C,P)}.
:- user(U,MIN,MAX), cost(U,C), C < MIN.
:- user(U,MIN,MAX), cost(U,C), C > MAX.
%check incompatibilities and requirements
:- user(U,_,_), incompatibility(I,_), assign(C,U) : incompatibility(I,C).
:- user(U,_,_), requirement(R,_), not assign(C,U) : requirement(R,C).
```

Fig. 1. Tested encoding for the Component Assignment benchmark (ASP-Core-2 syntax).

4.3 Experiments

The experiment comprises 147 random instances for increasing number of users (from 2 to 8) and components (from 30 to 50). Instances are grounded by GRINGO (Gebser et al. 2011; Gebser et al. 2015). Running time of WASP and CLASP 3.3.3 (Gebser et al. 2012; Gebser et al. 2015) were measured on an Intel Xeon 2.4 GHz with 16 GB of memory. Time and memory were limited to 20 minutes and 15 GB, respectively. WASP was tested with aggregate propagators (WASP-ST, the standard version of WASP) and with shared aggregate set propagators (WASP-SH, the new prototype).

The results are given in Table 1. Even if CLASP is faster than WASP-ST, solving one more instance overall, both solvers cannot terminate on instances with 6 or more users. A better performance is reached by WASP-SH, which can solve some instances up to 8 users, and is in general faster than CLASP and WASP-ST, as confirmed by the cactus plot shown in Figure 2. The figure

Table 1. *Solved instances and average running time (in seconds).*

COMPONENT ASSIGNMENT		CLASP		WASP-ST		WASP-SH	
#users	**#inst**	**sol.**	**avg t**	**sol.**	**avg t**	**sol.**	**avg t**
2	21	19	12.2	19	165.8	21	3.3
3	21	15	11.7	15	183.3	17	14.1
4	21	8	100.1	6	386.7	11	24.6
5	21	3	252.9	4	592.8	8	63.5
6	21	0	-	0	-	3	189.8
7	21	0	-	0	-	4	529.0
8	21	0	-	0	-	3	464.0
Total	147	45	43.7	44	240.7	67	77.1

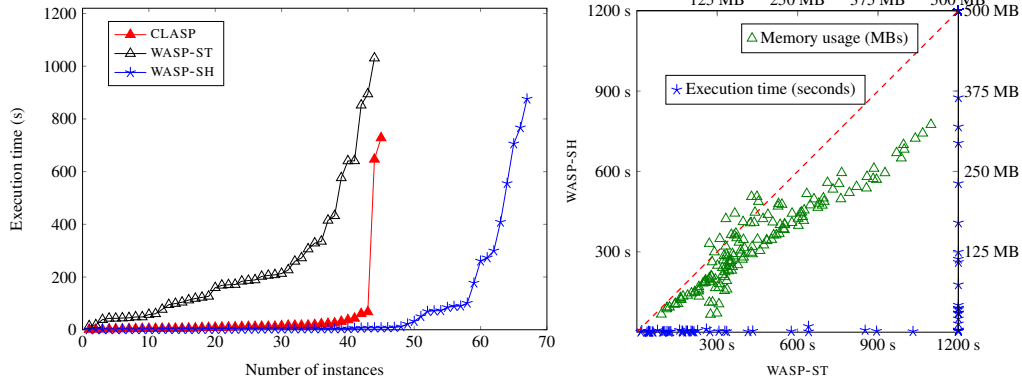| ASP COMPETITION		CLASP		WASP-ST		WASP-SH	
Problem	**#inst**	**sol.**	**avg t**	**sol.**	**avg t**	**sol.**	**avg t**
ADF	200	200	23.3	120	107.3	123	110.2
Bottle Filling	100	100	5.7	100	5.4	100	5.6
Still Life	26	6	81.3	6	112.7	6	31.4
Still Life with Holes	120	70	180.0	90	156.9	88	159.7
Total	446	376	48.7	316	89.3	317	89.5

Fig. 2. Component Assignment: number of solved instances within a given execution time limit (cactus plot on the left), and instance-by-instance comparison of execution time and memory consumption of the two versions of WASP (scatter plot on the right).

also shows an instance-by-instance comparison of the performance of the two tested versions of WASP. Concerning execution time, it can be observed that all the points are below the diagonal, meaning that the propagator always provides a performance gain for the tested instances. Often, shared aggregate set propagators also reduce the memory footprint of WASP.

A few additional testcases are obtained from instances of ASP Competitions. The aim of this benchmark is to verify the absence of overhead when shared aggregate set propagators are applied to encodings that, differently from the program in Figure 1, do not have assignments over aggregate sets with nonuniform weights. In fact, the benchmarks comprise instances that include assignments over count aggregates, that are, Abstract Dialectical Framework (ADF), Bottle Filling, Connected Maximum-density Still Life (Still Life), and Connected Maximum-density Still Life with Holes (Still Life with Holes). Results are given in Table 1, where the fact that CLASP is in general faster than WASP-ST is confirmed by a gap of 60 instances, mainly due to instances of ADF. Comparing WASP-ST and WASP-SH, instead, it can be observed that overall there is no overhead on using the shared aggregate set propagators, and actually there is a slight performance gain. The performance gain depends on the number of aggregates sharing the same aggregate set, as in fact for instances of ADF WASP-ST uses 900 aggregate propagators on average, while WASP-SH only requires 20 shared aggregate set propagators on average. For Still Life, there is no difference in terms of solved instances, but WASP-SH has a clear advantage over WASP-ST in terms of running time, again due to the number of aggregates sharing the same aggregate set. All in all, there is no overhead on using the shared aggregate set propagators also when the input program does not contain assignments over aggregate sets with nonuniform weights.

5 Related work

Rules of the form (2) were introduced in SMODELS (Simons et al. 2002), and called *basic constraint rules*. From a computational point of view, there are two mainstream approaches to evaluate ASP programs with aggregates, here referred to as *translation-based* and *propagator-based*.

The first approach aims at rewriting aggregates in terms of other constructs. For example,

the similarities between aggregates and pseudo-Boolean constraints were used to adapt to the ASP setting some compilations of pseudo-Boolean constraints into clauses (Aavani et al. 2013). Among them, there are adder circuits and binary decision diagrams (Abío et al. 2012; Eén and Sörensson 2006), sorting networks and watchdogs (Bailleux et al. 2009). ASP versions of these translations are implemented in LP2SAT and LP2NORMAL (Bomanson et al. 2014; Bomanson and Janhunen 2013), where the first solver outputs CNF formulas and the second solver outputs rules of the form (1). A translation-based approach is also implemented in CMODELS (Lierler and Maratea 2004; Giunchiglia et al. 2006; Giunchiglia et al. 2008), which maps aggregates to nested logic programs (Ferraris and Lifschitz 2005).

The concept of propagator was introduced in Satisfiability Modulo Theories (Nieuwenhuis et al. 2006; Abío and Stuckey 2012), and used in ASP to handle unfounded sets (Bomanson et al. 2015), support (Alviano and Dodaro 2016b), and also to avoid the instantiation of constraints (Cuteri et al. 2017). Moreover, some extensions of ASP, such as CASP (Baselice et al. 2005), were implemented by adding propagators to CDCL solvers (Banbara et al. 2017). Propagators are also the basis of the implementation of the solver IDP (Bruynooghe et al. 2015).

Concerning aggregates, DLV (Alviano et al. 2010; Alviano et al. 2017) implements ad-hoc techniques to evaluate programs with aggregates, among them a hashmap to compactly represent shared aggregate sets in the input program (Faber et al. 2008). Differently from the technique proposed in this paper, DLV could identify shared aggregate sets only at a symbolic level, before the grounding process. Actually, the aim of DLV was to speedup the grounding phase by not instantiate several times the same aggregation set, and no further advantage was obtained during the solving phase (if not a reduced memory footprint). Specifically, redundant computation during propagation was not eliminated in DLV. Another difference with the propagator introduced in this paper is that the stable model search algorithm implemented in DLV is much simpler, and does not support constraint learning and non-chronological backtracking, key features of modern solvers. The computation of reasons was supported in some version of DLV, but only for backjumping rather than for constraint learning (Faber et al. 2011); moreover, in DLV reasons are computed as soon as a literal is inferred, rather than postponed to when they are used.

The state-of-the-art solver CLASP implements a hybrid approach for handling programs with aggregates (Gebser et al. 2009), where aggregates involving few literals are compiled into normal rules; the threshold on the number of literals can be configured from the command-line. The aggregate propagator introduced in CLASP takes advantage of a trail of literals in order to postpone the computation of reasons. The same trail was used in the aggregate propagator of WASP. Shared aggregate sets generalize this propagator by compactly representing several aggregate propagators differing only on their bounds.

Aggregates in ASP can be interpreted according to several semantics (Ferraris 2011; Faber et al. 2011; Pelov et al. 2007; Son and Pontelli 2007; Gelfond and Zhang 2014), which however agree for programs with non-recursive aggregates. This is the main reason for inhibiting cycles over aggregates in Section 2.1. In fact, recursive aggregates require specific techniques in addition to the propagators presented in Section 2.2, as for example unfounded set detection for the semantics by Ferraris. Such techniques can still be used with the propagator introduced in Section 3, even in the non-convex case (being the complexity boundary; Alviano and Faber 2013) thanks to a rewriting into monotone aggregates (Alviano et al. 2015; Alviano et al. 2016; Alviano 2016). Interestingly, recursive aggregates can be eliminated for the semantics by Gelfond and Zhang by means of *polynomial, faithful and modular* (Janhunen 2006) translation functions (Alviano and Leone 2015; Alviano and Leone 2016). Finally, queries over (super-coherent) ASP

programs (Alviano et al. 2014) with aggregates can be optimized by magic sets (Alviano et al. 2011; Alviano and Faber 2011; Alviano et al. 2012), and the propagator introduced in this paper can be used also in presence of such an optimization.

Aggregates are also used by algorithms for computing optimal answer sets that are based on unsatisfiable core analysis. Among them are OLL (Andres et al. 2012) and ONE (Alviano et al. 2015; Alviano and Dodaro 2016a; Alviano and Dodaro 2017). OLL introduces several aggregates that can be compactly represented by the propagator introduced in this paper, so as to obtain the same behavior of ONE.

6 Conclusion

ASP syntax allows to bind object variables to the result of an aggregation, frequently a sum. Stable models of these programs may have constants not occurring in the input, but being among the possible results of such aggregations. For computing these stable models, mainstream ASP systems have to instantiate such aggregations for all possible interpretations of the undefined literals occurring in aggregate sets involved in an assignment. All these ground aggregates cause redundant computation of the solver. Shared aggregate set propagators are conceived to overcome such an inefficiency, and can be applied directly to the output of a grounder, not relying on any knowledge on the origin of the ground aggregates. The potential performance gain on WASP is of orders of magnitude, and also memory footprint is reduced. Finally, no overhead is introduced when there are few shared aggregate sets in the input program.

7 Acknowledgments

References

AAVANI, A., MITCHELL, D. G., AND TERNOVSKA, E. 2013. New encoding for translating pseudo-boolean constraints into SAT. In *Symposium on Abstraction, Reformulation, and Approximation*. AAAI.

ABÍO, I., NIEUWENHUIS, R., OLIVERAS, A., RODRÍGUEZ-CARBONELL, E., AND MAYER-EICHBERGER, V. 2012. A New Look at BDDs for Pseudo-Boolean Constraints. *Journal of Artificial Intelligence Research 45*, 443–480.

ABÍO, I. AND STUCKEY, P. J. 2012. Conflict directed lazy decomposition. In *International Conference on Principles and Practice of Constraint Programming*. Lecture Notes in Computer Science, vol. 7514. Springer, 70–85.

ALVIANO, M. 2011. Efficient recursive aggregate evaluation in logic programming. *Intelligenza Artificiale 5,* 2, 207–215.

ALVIANO, M. 2016. Evaluating answer set programming with non-convex recursive aggregates. *Fundamenta Informaticae 149,* 1-2, 1–34.

ALVIANO, M., CALIMERI, F., CHARWAT, G., AND ET AL. 2013. The Fourth Answer Set Programming Competition: Preliminary Report. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 8148. Springer, 42–53.

ALVIANO, M., CALIMERI, F., DODARO, C., FUSCÀ, D., LEONE, N., PERRI, S., RICCA, F., VELTRI, P., AND ZANGARI, J. 2017. The ASP system DLV2. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 10377. Springer, 215–221.

ALVIANO, M., CALIMERI, F., FABER, W., LEONE, N., AND PERRI, S. 2011. Unfounded sets and well-founded semantics of answer set programs with aggregates. *Journal of Artificial Intelligence Research 42*, 487–527.

ALVIANO, M. AND DODARO, C. 2016a. Anytime answer set optimization via unsatisfiable core shrinking. *Theory and Practice of Logic Programming 16,* 5-6, 533–551.

ALVIANO, M. AND DODARO, C. 2016b. Completion of disjunctive logic programs. In *International Joint Conference on Artificial Intelligence*. IJCAI/AAAI Press, 886–892.

ALVIANO, M. AND DODARO, C. 2017. Unsatisfiable core shrinking for anytime answer set optimization. In *International Joint Conference on Artificial Intelligence*. ijcai.org, 4781–4785.

ALVIANO, M., DODARO, C., FABER, W., LEONE, N., AND RICCA, F. 2013. WASP: A native ASP solver based on constraint learning. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 8148. Springer, 54–66.

ALVIANO, M., DODARO, C., LEONE, N., AND RICCA, F. 2015. Advances in WASP. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 9345. Springer, 40–54.

ALVIANO, M., DODARO, C., AND RICCA, F. 2015. A MaxSAT Algorithm Using Cardinality Constraints of Bounded Size. In *International Joint Conference on Artificial Intelligence*. AAAI Press, 2677–2683.

ALVIANO, M. AND FABER, W. 2011. Dynamic magic sets and super-coherent answer set programs. *AI Communications 24,* 2, 125–145.

ALVIANO, M. AND FABER, W. 2013. The complexity boundary of answer set programming with generalized atoms under the FLP semantics. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 8148. Springer, 67–72.

ALVIANO, M., FABER, W., AND GEBSER, M. 2015. Rewriting recursive aggregates in answer set programming: back to monotonicity. *Theory and Practice of Logic Programming 15,* 4-5, 559–573.

ALVIANO, M., FABER, W., AND GEBSER, M. 2016. From non-convex aggregates to monotone aggregates in ASP. In *International Joint Conference on Artificial Intelligence*. IJCAI/AAAI Press, 4100–4194.

ALVIANO, M., FABER, W., GRECO, G., AND LEONE, N. 2012. Magic sets for disjunctive datalog programs. *Artificial Intelligence 187*, 156–192.

ALVIANO, M., FABER, W., LEONE, N., PERRI, S., PFEIFER, G., AND TERRACINA, G. 2010. The Disjunctive Datalog System DLV. In *Datalog Reloaded*. Lecture Notes in Computer Science, vol. 6702. Springer, 282–301.

ALVIANO, M., FABER, W., AND STRASS, H. 2016. Boolean functions with ordered domains in answer set programming. In *AAAI Conference on Artificial Intelligence*. AAAI Press, 879–885.

ALVIANO, M., FABER, W., AND WOLTRAN, S. 2014. Complexity of super-coherence problems in ASP. *Theory and Practice of Logic Programming 14,* 3, 339–361.

ALVIANO, M., GRECO, G., AND LEONE, N. 2011. Dynamic magic sets for programs with monotone recursive aggregates. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 6645. Springer, 148–160.

ALVIANO, M. AND LEONE, N. 2015. Complexity and compilation of gz-aggregates in answer set programming. *Theory and Practice of Logic Programming 15,* 4-5, 574–587.

ALVIANO, M. AND LEONE, N. 2016. On the properties of gz-aggregates in answer set programming. In *International Joint Conference on Artificial Intelligence*. IJCAI/AAAI Press, 4105–4109.

ANDRES, B., KAUFMANN, B., MATHEIS, O., AND SCHAUB, T. 2012. Unsatisfiability-based optimization in clasp. In *Technical Communications of the International Conference on Logic Programming*. LIPIcs, vol. 17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 211–221.

BAILLEUX, O., BOUFKHAD, Y., AND ROUSSEL, O. 2009. New encodings of pseudo-boolean constraints into CNF. In *The International Conferences on Theory and Applications of Satisfiability Testing*. Lecture Notes in Computer Science, vol. 5584. Springer, 181–194.

BANBARA, M., KAUFMANN, B., OSTROWSKI, M., AND SCHAUB, T. 2017. Clingcon: The next generation. *Theory and Practice of Logic Programming 17,* 4, 408–461.

BARTHOLOMEW, M., LEE, J., AND MENG, Y. 2011. First-order semantics of aggregates in answer set programming via modified circumscription. In *Logical Formalizations of Commonsense Reasoning AAAI Spring Symposium.* AAAI.

BASELICE, S., BONATTI, P. A., AND GELFOND, M. 2005. Towards an integration of answer set and constraint solving. In *International Conference on Logic Programming.* Lecture Notes in Computer Science, vol. 3668. Springer, 52–66.

BOMANSON, J., GEBSER, M., AND JANHUNEN, T. 2014. Improving the normalization of weight rules in answer set programs. In *European Conference on Logics in Artificial Intelligence.* Lecture Notes in Computer Science, vol. 8761. Springer, 166–180.

BOMANSON, J., GEBSER, M., JANHUNEN, T., KAUFMANN, B., AND SCHAUB, T. 2015. Answer set programming modulo acyclicity. In *International Conference on Logic Programming and Nonmonotonic Reasoning.* Lecture Notes in Computer Science, vol. 9345. Springer, 143–150.

BOMANSON, J. AND JANHUNEN, T. 2013. Normalizing cardinality rules using merging and sorting constructions. In *Logic Programming and Nonmonotonic Reasoning.* Lecture Notes in Computer Science, vol. 8148. Springer, 187–199.

BREWKA, G., EITER, T., AND TRUSZCZYNSKI, M. 2011. Answer set programming at a glance. *Communications of the ACM 54,* 12, 92–103.

BRUYNOOGHE, M., BLOCKEEL, H., BOGAERTS, B., DE CAT, B., POOTER, S. D., JANSEN, J., LABARRE, A., RAMON, J., DENECKER, M., AND VERWER, S. 2015. Predicate logic as a modeling language: modeling and solving some machine learning and data mining problems with *IDP3. Theory and Practice of Logic Programming 15,* 6, 783–817.

CALIMERI, F., FABER, W., GEBSER, M., IANNI, G., KAMINSKI, R., KRENNWALLNER, T., LEONE, N., RICCA, F., AND SCHAUB, T. 2013. ASP-Core-2 Input Language Format.

CALIMERI, F., GEBSER, M., MARATEA, M., AND RICCA, F. 2016. Design and results of the Fifth Answer Set Programming Competition. *Artificial Intelligence 231,* 151–181.

CUTERI, B., DODARO, C., RICCA, F., AND SCHÜLLER, P. 2017. Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis. *Theory and Practice of Logic Programming 17,* 5-6, 780–799.

DODARO, C., ALVIANO, M., FABER, W., LEONE, N., RICCA, F., AND SIRIANNI, M. 2011. The birth of a WASP: preliminary report on a new ASP solver. In *Italian Conference on Computational Logic.* CEUR Workshop Proceedings, vol. 810. CEUR-WS.org, 99–113.

DODARO, C., GASTEIGER, P., LEONE, N., MUSITSCH, B., RICCA, F., AND SCHEKOTIHIN, K. 2016. Combining answer set programming and domain heuristics for solving hard industrial problems (application paper). *Theory and Practice of Logic Programming 16,* 5-6, 653–669.

EÉN, N. AND SÖRENSSON, N. 2006. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation 2,* 1-4, 1–26.

FABER, W., LEONE, N., MARATEA, M., AND RICCA, F. 2011. Look-back techniques for ASP programs with aggregates. *Fundamenta Informaticae 107,* 4, 379–413.

FABER, W., PFEIFER, G., AND LEONE, N. 2011. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence 175,* 1, 278–298.

FABER, W., PFEIFER, G., LEONE, N., DELL'ARMI, T., AND IELPA, G. 2008. Design and implementation of aggregate functions in the DLV system. *Theory and Practice of Logic Programming 8,* 5-6, 545–580.

FERRARIS, P. 2011. Logic programs with propositional connectives and aggregates. *ACM Transactions on Computational Logic 12,* 4, 25.

FERRARIS, P. AND LIFSCHITZ, V. 2005. Weight constraints as nested expressions. *Theory and Practice of Logic Programming 5,* 1-2, 45–74.

GEBSER, M., HARRISON, A., KAMINSKI, R., LIFSCHITZ, V., AND SCHAUB, T. 2015. Abstract gringo. *Theory and Practice of Logic Programming 15,* 4-5, 449–463.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., ROMERO, J., AND SCHAUB, T. 2015. Progress in clasp Series 3. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 9345. Springer, 368–383.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., AND SCHAUB, T. 2009. On the implementation of weight constraint rules in conflict-driven ASP solvers. In *International Conference on Logic Programming*. Lecture Notes in Computer Science, vol. 5649. Springer, 250–264.

GEBSER, M., KAMINSKI, R., KÖNIG, A., AND SCHAUB, T. 2011. Advances in *gringo* series 3. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 6645. Springer, 345–351.

GEBSER, M., KAUFMANN, B., ROMERO, J., OTERO, R., SCHAUB, T., AND WANKO, P. 2013. Domain-specific heuristics in answer set programming. In *AAAI Conference on Artificial Intelligence*. AAAI Press.

GEBSER, M., KAUFMANN, B., AND SCHAUB, T. 2009. The conflict-driven answer set solver clasp: Progress report. In *International Conference Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 5753. Springer, 509–514.

GEBSER, M., KAUFMANN, B., AND SCHAUB, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence 187*, 52–89.

GEBSER, M., KAUFMANN, B., AND SCHAUB, T. 2013. Advanced conflict-driven disjunctive answer set solving. In *International Joint Conference on Artificial Intelligence*. IJCAI/AAAI.

GEBSER, M., MARATEA, M., AND RICCA, F. 2017. The sixth answer set programming competition. *Journal of Artificial Intelligence Research 60*, 41–95.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Logic Programming*. MIT Press, 1070–1080.

GELFOND, M. AND ZHANG, Y. 2014. Vicious circle principle and logic programs with aggregates. *Theory and Practice of Logic Programming 14,* 4-5, 587–601.

GIUNCHIGLIA, E., LEONE, N., AND MARATEA, M. 2008. On the relation among answer set solvers. *Annals of Mathematics and Artificial Intelligence 53,* 1-4, 169–204.

GIUNCHIGLIA, E., LIERLER, Y., AND MARATEA, M. 2006. Answer Set Programming Based on Propositional Satisfiability. *Journal of Automated Reasoning 36,* 4, 345–377.

JANHUNEN, T. 2006. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics 16,* 1-2, 35–86.

JANHUNEN, T. AND NIEMELÄ, I. 2016. The answer set programming paradigm. *AI Magazine 37,* 3, 13–24.

LIERLER, Y. AND MARATEA, M. 2004. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 2923. Springer, 346–350.

LIFSCHITZ, V. 2016. Answer sets and the language of answer set programming. *AI Magazine 37,* 3, 7–12.

LIU, L., PONTELLI, E., SON, T. C., AND TRUSZCZYNSKI, M. 2010. Logic programs with abstract constraint atoms: The role of computations. *Artificial Intelligence 174,* 3-4, 295–315.

NIEUWENHUIS, R., OLIVERAS, A., AND TINELLI, C. 2006. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM 53,* 6, 937–977.

PELOV, N., DENECKER, M., AND BRUYNOOGHE, M. 2007. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming 7,* 3, 301–353.

SIMONS, P., NIEMELÄ, I., AND SOININEN, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138,* 1-2, 181–234.

SON, T. C. AND PONTELLI, E. 2007. A constructive semantic characterization of aggregates in answer set programming. *Theory and Practice of Logic Programming 7,* 3, 355–375.