Shared aggregate sets in answer set programming

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

CARMINE DODARO and MARCO MARATEA
DIBRIS, University of Genova, Italy
(e-mails: dodaro@dibris.unige.it, marco@dibris.unige.it)

submitted 27 April 2018; accepted 11 May 2018

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):

\[
\{ p(2); p(5) \}. \quad q(S) : - \ S = \text{sum}\{ X : p(X) \}.
\]

whose instantiation contains \( q(s) :- s = \text{sum}\{2 : p(2); 5 : p(5)\} \) 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 \( a : - \ \#\text{sum}\{w_1 : \ell_1; \cdots ; w_n : \ell_n\} \geq b \), 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:

\[
\{ p(2); p(5) \}. \quad a_x : - \ \#\text{sum}\{2 : p(2); 5 : p(5)\} \geq x. \quad (\forall x \in \{1, 2, 3, 5, 6, 7\})
\]

\[
q(0) : - \ \text{not} \ a_1. \quad q(2) : - a_2, \ \text{not} \ a_3. \quad q(5) : - a_5, \ \text{not} \ a_6. \quad q(7) : - a_7.
\]

where intuitively each aggregate \( s = \#\text{sum}\{2 : p(2); 5 : p(5)\} \) was replaced by the conjunction \( \#\text{sum}\{2 : p(2); 5 : p(5)\} \geq s, \ \text{not} \ \#\text{sum}\{2 : p(2); 5 : p(5)\} \geq s + 1 \). 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, \( \#\text{sum}\{2 : p(2); 5 : p(5)\} \geq 1 \) and \( \#\text{sum}\{2 : p(2); 5 : p(5)\} \geq 2 \) 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 \( a_1 : - \ \#\text{sum}\{2 : p(2); 5 : p(5)\} \geq 1 \) becomes rule \( a_1 : - \ \#\text{sum}\{2 : p(2); 5 : p(5)\} \geq 2 \). 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 \( \mathcal{A} \) be a set of atoms. An atomic formula is either an atom, or the connective \( \bot \). A literal is an atomic formula possibly preceded by the default negation symbol \( \neg \). For a literal \( \ell \), let \( \overline{\ell} \) denote the complement of \( \ell \), that is, \( \overline{\ell} = \neg p \) and \( \overline{\overline{p}} = p \) for all \( p \in \mathcal{A} \cup \{ \bot \} \); for a set \( \mathcal{L} \) of literals, let \( \overline{\mathcal{L}} = \{ \overline{\ell} \mid \ell \in \mathcal{L} \} \). Let \( \top \) be a compact representation of \( \neg \bot \).

A rule is one of the following forms:

\[
\begin{align*}
    p_1 \lor \cdots \lor p_m & \leftarrow \ell_1, \ldots, \ell_n \quad (1) \\
    p & \leftarrow \sum \{ w_0 : \ell_0, \ldots, w_n : \ell_n \} \geq b \quad (2)
\end{align*}
\]

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 \( \{ p_1, \ldots, p_m \} \cap \mathcal{A} \) of head atoms, and \( B(r) \) denote the set \( \{ \ell_1, \ldots, \ell_n \} \) of body literals (Note that \( \bot \) may occur in \( B(r) \), but it is excluded from \( H(r) \)). For a rule \( r \) of the form (2), define \( id(r) := p, \quad sum(r) := \sum \{ w_0 : \ell_0, \ldots, w_n : \ell_n \} \geq b, \quad elem(r) := \{(w_i, \ell_i) \mid i \in [0..n] \}, \quad and \quad bound(r) := b \).

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

Example 2.1 (Running example)

Let \( \Pi_{run} \) be the following program (similar to the example in the introduction):

\[
g_2 : p_2 \lor n_2 \leftarrow g_5 : p_5 \lor n_5 \leftarrow r_x : a_x \leftarrow \sum \{ 2 : p_2, 5 : p_5 \} \geq x \quad \forall x \in \{1,2,3,5,6,7\}.
\]

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 \( sum(r_x) = \sum \{ 2 : p_2, 5 : p_5 \} \geq x \), whose elements are \( elem(r_x) = \{(2, p_2), (5, p_5)\} \), and whose bound is \( bound(r_x) = x \).

The dependency graph \( G_{\Pi} \) of \( \Pi \) has nodes \( atoms(\Pi) \), and an arc \( xy \) (where \( x \) and \( y \) are atoms) for each rule \( r \in rules^\lor(\Pi) \) 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 \( \Pi \) if it is involved in a cycle of \( G_{\Pi} \). In the following every program \( \Pi \) is assumed to satisfy the following property: if \( r \in \Pi \) is of the form (2), \( id(r) \) is not recursive in \( \Pi \). Note that \( \Pi_{run} \) has such a property.

An assignment \( I \) is a set of literals (different from \( \bot \) 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 \( \Pi \) is an assignment such that \( I \cap \overline{I} = atoms(\Pi) \cup \overline{atoms(\Pi)} \). Relation \( \models \) is inductively defined as follows: \( I \models \bot \) (hence, \( I \models \top \)); for \( p \in \mathcal{A} \), \( I \models p \) if \( p \in I \); \( I \models \neg p \) if \( I \not\models p \); for a rule \( r \) of the form

https://doi.org/10.1017/S1471068418000133 Published online by Cambridge University Press
(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 \( \sum_{(w,\ell)\in \text{elem}(r)} t_\ell w \geq \text{bound}(r) \), and \( I \models r \) if \( I \models \text{id}(r) \) whenever \( I \models \sum(r) \); for a program \( \Pi \), \( I \models \Pi \) if \( I \models r \) for all \( r \in \Pi \). For any expression \( \pi \), if \( I \models \pi \), we say that \( I \) is a model of \( \pi \).

The reduce \( \Pi' \) of a program \( \Pi \) with respect to an interpretation \( I \) comprises the following rules: for each rule \( r \in \text{rules} \setminus \Pi \) such that \( I \models B(r) \), \( \Pi' \) contains a rule \( r' \) of the form (1) with \( H(r') = H(r) \), and \( B(r') = B(r) \setminus \alpha \); for each rule \( r \in \text{rules} \setminus \Pi \) such that \( I \models \sum(r) \), \( \Pi' \) contains a rule \( r' \) of the form (2) with \( \text{id}(r') = \text{id}(r) \), \( \text{bound}(r') = \text{bound}(r) \), and \( \text{elem}(r) = \{(w, \ell) \in \text{elem}(r) : I \models \ell, \ell \in \alpha \} \cup \{(\sum_{(w,\ell)\in \text{elem}(r)} t_\ell w) \mid I \models r, w \in \top \} \}. \) An interpretation \( I \) is a stable model of a program \( \Pi \) if \( I \models \Pi \) and there is no \( J \subset I \) such that \( J \models \Pi' \). Let \( \text{SM}(\Pi) \) denote the set of stable models of \( \Pi \). For instance, for \( \Pi_{\text{run}} \) from Example 2.1, \( \text{SM}(\Pi_{\text{run}}) \) 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 \( \ell \) added to \( I \) is also associated with a set of reasons, essentially the true literals causing the addition of \( \ell \) 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 \( \ell \) 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 \( C(r) := \{ p_{1}, \ldots, p_{m}, \ell_{1}, \ldots, \ell_{n} \} \) be the clause representation of \( r \). A literal \( \ell \in C(r) \setminus I \) is unit propagated with respect to \( I \) and \( r \) if \( C(r) \setminus \{ \ell \} \subseteq I \); let \( \text{reasons}(\ell) \) be \( C(r) \setminus \{ \ell \} \) in this case. For instance, atom \( p_{2} \) is unit propagated with respect to the assignment \{\( a_{1}, a_{2}, \neg n_{2} \} \) and rule \( g_{2} \) from Example 2.1, and \( \text{reasons}(p_{2}) = \{ \neg n_{2} \} \).

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 inferences are inferred (if not already in \( I \)):

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

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

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

https://doi.org/10.1017/S1471068418000133 Published online by Cambridge University Press
(A4) any \( \ell \) such that \((w, \ell) \in \text{elem}(r)\), whenever both \( \text{id}(r) \in I \) and \( \sum_{(w', \ell') \in \text{elem}(r) \setminus \{(w, \ell)\}} w' \geq \text{bound}(r) - w \), with reasons(\( \ell \)) comprising \( \text{id}(r) \) and true literals in \( \text{elem}(r) \), that is, 
\[
\{\text{id}(r)\} \cup \{\ell' \mid (w', \ell') \in \text{elem}(r), \; \ell' \in I\}.
\]

For instance, for the rules from Example 2.1, if \( I = \{a_1, a_2, \neg p_2, p_3\} \), then \( a_3, a_5 \) are inferred by (A1), and \( \neg a_6, \neg a_7 \) are inferred by (A2); in this case, reasons(\( a_3 \)) = reasons(\( a_5 \)) = \{\( p_5 \)\}, and reasons(\( \neg a_6 \)) = reasons(\( \neg a_7 \)) = \{\( \neg p_2 \)\}. Similarly, if \( I = \{a_5, \neg a_6\} \), then \( p_5 \) is inferred by (A3), and \( \neg p_2 \) is inferred by (A4); in this case, reasons(\( p_5 \)) = \{\( a_5 \)\}, and reasons(\( \neg p_2 \)) = \{\( \neg a_6 \)\}.

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 not easy to discover by CDCL.

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

\[
\arg \min_{r' \in \text{rules}(\Pi), \; \text{elem}(r') = \text{elem}(r), \; \text{bound}(r') > \text{bound}(r)} \text{id}(r').
\]

if it exists, and \( \bot \) 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 \( r \in \text{rules}(\Pi)\):

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

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 \( \Pi \) be a program, and \( \Pi' \) be obtained by applying (i)–(iv) to \( \Pi \). 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 $j \geq 0$ $\Pi_{j+1}$ is obtained from $\Pi_j$ 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 $\Pi_{j+1}$ be obtained by applying (i) to $\Pi_j$.

(\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, $\Pi'_j = \Pi'_{j+1}$, and therefore $I \in SM(\Pi_{j+1})$.

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

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

(iii) Let $\Pi_{j+1}$ be obtained by applying (iii) to $\Pi_j$.

(\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 $\Pi'_j = \Pi'_{j+1}$, 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 $J \subseteq I$ implies $J \not\models \Pi'_{j+1}$. Since $I \in SM(\Pi_j)$ by assumption, $J \not\models \Pi'_j$ holds. Let $J$ be such that $J = \Pi'_j \cap \Pi'_{j+1}$ (otherwise it is trivial); hence, there is $r'' \in \Pi'_j \setminus \Pi'_{j+1}$ 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, $J \not\models id(r')$ because $I \models id(r')$ follows from $J = \Pi'_j \cap \Pi'_{j+1}$ 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 $\Pi'_j = \Pi'_{j+1}$, 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 $J \subseteq I$ implies $J \not\models \Pi'_j$. Since $I \in SM(\Pi_{j+1})$ by assumption, $J \not\models \Pi'_{j+1}$ holds. Let $J$ be such that $J \models \Pi'_j \cap \Pi'_{j+1}$ (otherwise it is trivial); hence, there is $r'' \in \Pi'_j \setminus \Pi'_{j+1}$ 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 $J \not\models \Pi'_j$. Otherwise, if $J \not\models sum(r)$, then $J \not\models \{id(r')\} \models \Pi'_{j+1}$, which would contradict the assumption that $I \in SM(\Pi_{j+1})$. It turns out that $I \in SM(\Pi_j)$ holds.

(iv) Let $\Pi_{j+1}$ be obtained by applying (iv) to $\Pi_j$.

(\subseteq) Let $I \in SM(\Pi_j)$. If $I \models id(r)$, then $I \models \Pi_{j+1}$ and $\Pi'_j = \Pi'_{j+1}$, 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 $\Pi'_j = \Pi'_{j+1}$, so $I \in SM(\Pi_{j+1})$.

\supseteq) Let $I \in SM(\Pi_{j+1})$. Since $\Pi_j \subseteq \Pi_{j+1}$, $I \models \Pi_j$. Moreover, $\Pi'_j = \Pi'_{j+1}$, and therefore $I \in SM(\Pi_j)$.

The proof is complete.

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

\[
\begin{align*}
g_2 & : \ p_2 \lor n_2 \leftarrow g_5 : \ p_5 \lor n_5 \leftarrow \\
r'_1 & : \ a_1 \leftarrow a_2 \quad r_2 : \ a_2 \leftarrow \sum \{2 : p_2, 5 : p_5\} \geq 2 \ s_2 : \ \bot \leftarrow \sim a_2, a_5 \\
r'_2 & : \ a_3 \leftarrow a_5 \quad r_3 : \ a_5 \leftarrow \sum \{2 : p_2, 5 : p_5\} \geq 5 \ s_5 : \ \bot \leftarrow \sim a_5, a_7 \\
r'_6 & : \ a_6 \leftarrow a_7 \quad r_7 : \ a_7 \leftarrow \sum \{2 : p_2, 5 : p_5\} \geq 7 \ s_7 : \ \bot \leftarrow \sim a_7, \bot
\end{align*}
\]

For instance, the application of (ii) modifies the bound of \( r_1 \) from 1 to 2, so that it now occurs in \( \text{sums}(r_1) \); 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 \( \Pi \), let \( X \) be a maximal subset of \( \text{rules}^\Sigma(\Pi) \) such that all \( r, r' \in X \) satisfy \( \text{elem}(r) = \text{elem}(r') \). A shared aggregate set propagator for \( X \) is associated with the following sets: \( \text{elem}(X) \), storing the shared set of elements, that is, \( \text{elem}(r) \), for any \( r \in X \); and \( \text{bounds}(X) \), storing identifiers and bounds, that is, \( \{(\text{bound}(r), id(r)) \mid r \in X\} \). Moreover, the following functions are used by the propagator when applied to an assignment \( I \):

- \( \text{min}_\text{sum}(X, I) := \sum_{(w, \ell) \in \text{elem}(X), \ \ell \in I} w \), the smallest value of the sum among those assigned by interpretations extending the current assignment;
- \( \text{max}_\text{sum}(X, I) := \sum_{(w, \ell) \in \text{elem}(X), \ \ell \in I} w \), the greatest value of the sum among those assigned by interpretations extending the current assignment;
- \( \text{lower}_\text{bound}(X, I) := \max(\{b \mid (b, \ell) \in \text{bounds}(X), \ \ell \in I \cup \{0\}\}) \), a value that must be reached by the sum in any extension of the current assignment;
- \( \text{upper}_\text{bound}(X, I) := \min(\{b \mid (b, \ell) \in \text{bounds}(X), \ \ell \in I \cup \{+\infty\}\}) \), 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 \( \text{rules}^\Sigma(\Pi_{\text{run}}') \). The shared aggregate set propagator associated with \( X \) has \( \text{elem}(X) = \{(2, p_2), (5, p_5)\} \), and \( \text{bounds}(X) = \{(2, a_2), (5, a_5), (7, a_7)\} \). For the empty assignment, \( \text{min}_\text{sum}(X, \emptyset) = 0 \), \( \text{max}_\text{sum}(X, \emptyset) = 7 \), \( \text{lower}_\text{bound}(X, \emptyset) = 0 \), and \( \text{upper}_\text{bound}(X, \emptyset) = +\infty \); for \( I = \{p_2, \sim p_5, a_5, \sim a_7\} \), \( \text{min}_\text{sum}(X, I) = 2 \), \( \text{max}_\text{sum}(X, I) = 2 \), \( \text{lower}_\text{bound}(X, I) = 5 \), and \( \text{upper}_\text{bound}(X, I) = 7 \).

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

**Case 1.** If \( \ell \), the last literal added to \( I \), occurs in an element \( (w, \ell) \) 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 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

https://doi.org/10.1017/S1471068418000133 Published online by Cambridge University Press
Function OnLiteralTrue(X, I, ℓ)

1 if (w, ℓ) ∈ elem(X) then
2   R := {ℓ′ | (w′, ℓ′) ∈ elem(X), ℓ′ ∈ I};
3   R′ := R ∪ {ℓ′ | upper_bound(X, I), ℓ′) ∈ bounds(X)};
4   return {{(ℓ′, R) | (b, ℓ′) ∈ bounds(X), ℓ′ /∈ I, b ≤ min_sum(X, I)}} ∪
5      {{(ℓ′, R′) | (w′, ℓ′) ∈ elem(X), ℓ′ /∈ I, w′ ≥ upper_bound(X, I) − min_sum(X, I)}};
6 if (w, ℓ) ∈ elem(X) then
7   R := {ℓ′ | (w′, ℓ′) ∈ elem(X), ℓ′ /∈ I};
8   R′ := R ∪ {ℓ′ | lower_bound(X, I), ℓ′) ∈ bounds(X)};
9   return {{(ℓ′, R) | (b, ℓ′) ∈ bounds(X), ℓ′ /∈ I, b > max_sum(X, I)}} ∪
10  {{(ℓ′, R′) | (w′, ℓ′) ∈ elem(X), ℓ′ /∈ I, w′ > max_sum(X, I) − lower_bound(X, I)}};
11 if (b, ℓ) ∈ bounds(X) and then
12   R := {ℓ′ | (w′, ℓ′) ∈ elem(X), ℓ′ /∈ I} ∪ {ℓ};
13   return {{(ℓ′, R) | (w′, ℓ′) ∈ elem(X), ℓ′ /∈ I, w′ > max_sum(X, I) − lower_bound(X, I)}};
14 if (b, ℓ) ∈ bounds(X) and then
15   R := {ℓ′ | (w′, ℓ′) ∈ elem(X), ℓ′ /∈ I} ∪ {ℓ};
16   return {{(ℓ′, R) | (w′, ℓ′) ∈ elem(X), ℓ′ /∈ I, w′ ≥ upper_bound(X, I) − min_sum(X, I)}};
17 return ∅;

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, ℓ) 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 truth 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, ℓ) 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 $\ell$ (line 13); such an inference is analogous to (A4) for aggregate propagators.

**Theorem 3.2**
Let $\Pi$ 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 $\text{elem}(r) = \text{elem}(r')$, $I$ be an assignment, and $\ell$ be the last literal added to $I$. A literal $\ell'$ is inferred by $\text{OnLiteralTrue}(X, I, \ell)$ with reasons $R$ if and only if $\ell'$ is inferred with reasons $R$ by applying (A1)–(A4) to some $r \in X$ and assignment $I$.

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

1. If there is $(w, \ell) \in \text{elem}(X)$, we further distinguish two cases:
   - There is $(b, \ell') \in \text{bounds}(X)$ such that $b \leq \min\text{sum}(X, I)$. Hence, there is $r \in X$ such that $\text{bound}(r) = b$, and $\text{id}(r) = \ell'$. Since $b \leq \min\text{sum}(X, I)$, (A1) derives $\text{id}(r)$. Moreover, both propagators set $\text{reasons}(\ell') = \{\ell'' \mid (w'', \ell'') \in \text{elem}(X), \ell'' \in I\}$.
   - There is $(w', \ell') \in \text{elem}(X)$ such that $w' \geq \text{upper}\text{bound}(X, I) - \text{min}\text{sum}(X, I)$. Hence, there is $r \in X$ such that $\text{bound}(r) = \min\{b \mid (b, \ell'') \in \text{bounds}(X), \ell'' \in I\}$. Thus, $\text{id}(r) \in I$, and (A4) derives $\ell'$. Moreover, both propagators set $\text{reasons}(\ell') = \{\text{id}(r)\} \cup \{\ell'' \mid (w'', \ell'') \in \text{elem}(X), \ell'' \in I\}$.

2. If there is $(w, \ell) \in \text{elem}(X)$, we further distinguish two cases:
   - There is $(b, \ell') \in \text{bounds}(X)$ such that $b > \max\text{sum}(X, I)$. Hence, there is $r \in X$ such that $\text{bound}(r) = b$, and $\text{id}(r) = \ell'$. Since $b > \max\text{sum}(X, I)$, (A2) derives $\text{id}(r) = \ell'$. Moreover, both propagators set $\text{reasons}(\ell') = \{\text{id}(r)\} \cup \{\ell'' \mid (w'', \ell'') \in \text{elem}(X), \ell'' \in I\}$.
   - There is $(w', \ell') \in \text{elem}(X)$ such that $w' > \max\text{sum}(X, I) - \text{lower}\text{bound}(X, I)$. Hence, there is $r \in X$ such that $\text{bound}(r) = \max\{\{b \mid (b, \ell'') \in \text{bounds}(X), \ell'' \in I\}\}$. Thus, $\text{id}(r) \in I$, and (A3) derives $\ell'$. Moreover, both propagators set $\text{reasons}(\ell') = \{\text{id}(r)\} \cup \{\ell'' \mid (w'', \ell'') \in \text{elem}(X), \ell'' \in I\}$.

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

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

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

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

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

3. Literal \(\ell\) is inferred by (A3). Hence, there is \(r ∈ X\) such that \((w', \ell') ∈ elem(r), id(r) ∈ I, \sum\_{(w'', \ell'') ∈ elem(r) \setminus \{(w', \ell')\}} \overline{\ell''} < bound(r)\), and either \(id(r) ∉ I \setminus \{\ell\}\) or the following inequality holds:\  
\[
\sum\_{(w'', \ell'') ∈ elem(r) \setminus \{(w', \ell')\}, \overline{\ell''} ∈ \{\ell\}} w'' ≥ bound(r).
\]
- If \(id(r) ∉ I \setminus \{\ell\}\), then \(id(r) = \ell\), and therefore \(\ell'\) is derived by OnLiteralTrue(X, I, \ell) at line 11 because \((bound(r), \ell) ∈ bounds(X), (w', \ell') ∈ elem(X)\) and \(w' > \max\_sum(X, I) - lower\_bound(X, I)\). Moreover, both propagators set reasons(\(\ell'\)) equals to \{id(r)\} ∪ \{\overline{\ell''} | (w'', \ell'') ∈ elem(X), \overline{\ell''} ∈ I\}.

- If \(\sum\_{(w'', \ell'') ∈ elem(r) \setminus \{(w', \ell')\}, \overline{\ell''} ∈ \{\ell\}} w'' ≥ bound(r)\), there is \((w, \overline{\ell}) ∈ elem(X)\). Hence, \(\ell'\) is derived by OnLiteralTrue(X, I, \ell) at line 8 because there is \((w', \ell') ∈ elem(X)\) such that \(w' > \max\_sum(X, I) - lower\_bound(X, I)\). Moreover, both propagators set reasons(\(\ell'\)) equals to \{id(r)\} ∪ \{\overline{\ell''} | (w'', \ell'') ∈ elem(X), \overline{\ell''} ∈ I\}.

4. Literal \(\ell\) is inferred by (A4). Hence, there is \(r ∈ X\) such that \((w', \overline{\ell}) ∈ elem(r), \overline{id(r)} ∈ I, \sum\_{(w'', \ell'') ∈ elem(r) \setminus \{(w', \overline{\ell})\}, \overline{\ell''} ∈ \{\ell\}} w'' > bound(r) - w'\), and either \(\overline{id(r)} ∉ I \setminus \{\ell\}\) or the inequality holds:\  
\[
\sum\_{(w'', \ell'') ∈ elem(r) \setminus \{(w', \overline{\ell})\}, \overline{\ell''} ∈ \{\ell\}} w'' < bound(r) - w'\]
\]
- If \(\overline{id(r)} ∉ I \setminus \{\ell\}\), then \(\overline{id(r)} = \ell\), and therefore \(\ell'\) is derived by OnLiteralTrue(X, I, \ell) at line 14 because there is \((w', \overline{\ell}) ∈ elem(X)\) such that the following inequality holds: \(w' ≥ upper\_bound(X, I) - \min\_sum(X, I)\). Moreover, both propagators set reasons(\(\ell'\)) equals to \{\overline{id(r)}\} ∪ \{\overline{\ell''} | (\overline{w''}, \overline{\ell''}) ∈ elem(X), \overline{\ell''} ∈ I\}.

The proof is complete. \(\square\)

Example 3.3 (Continuing Example 3.2)
Let X be rules \(\Sigma(\Pi_{\text{run}})\), and recall that elem(X) = \{(2, p₂), (5, p₅)\}, and bounds(X) = \{(2, a₂), (5, a₅), (7, a₇)\}. OnLiteralTrue(X, \{~a₇, ~n₂, p₂\}, p₂) returns \(\{a₂, \{p₂\}\}\) and \(\{~p₅, \{p₂, ~a₇\}\}\) because of case 1; indeed, literal \(a₂\) is inferred because \(\min\_sum(\{~a₇, ~n₂, p₂\}) = 2\), and literal \(~p₅\) is inferred because \(\min\_bound(\{~a₇, ~n₂, p₂\}) = 2\). Similarly, OnLiteralTrue(X, \{a₂, p₅, ~p₅\}, ~p₅) returns \(\{~a₅, \{p₅\}\}\), \(\{~a₇, \{~p₅\}\}\) and \(\{p₂, \{~p₅, a₇\}\}\) because of case 2. OnLiteralTrue(X, \{a₅, a₇\}) returns the pair \(\{p₅, \{a₅\}\}\) because of case 3. Finally, OnLiteralTrue(X, \{~a₅\}, ~a₅) returns the pair \(\{p₅, \{~a₅\}\}\) because of case 4. \(\triangle\)

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 \(\ell\),
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 \((w, \ell)\) and \((w', \ell')\) occur in a rule \(r \in \text{rules}\Sigma(\Pi)\), and \(w \geq w'\), then \(r\) is replaced by a rule \(r'\) of the form (2) such that \(id(r') := id(r)\), \(elem(r') := (elem(r) \setminus \{(w, \ell), (w', \ell')\}) \cup \{(w - w', \ell) \mid w - w' > 0\}\), and \(\text{bound}(r') := \text{bound}(r) - w'\).

The second optimization concerns the implementation of the functions \(\text{min}\_\text{sum, max}\_\text{sum, lower}\_\text{bound, and upper}\_\text{bound, which are heavily used by OnLiteralTrue}(X,I,\ell)\). Instead of computing their values by performing a complete iteration on the sets \(\text{elem}(X)\) and \(\text{bounds}(X)\), 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 \(\text{OnLiteralTrue}(X,I,\ell)\) is invoked, literal \(\ell\) 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 \(\text{OnLiteralTrue}\).

## 4.2 Component Assignment

The input of the Component Assignment problem is a tuple \((C, p, U, B, \mathcal{I}, \mathcal{R})\) 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; \(\mathcal{I}\) is a set of sets of jointly incompatible components; \(\mathcal{R}\) is a set of sets of required components. The goal is to compute a partial assignment \(f : C \rightarrow U\) 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 \(\mathcal{I}' \in \mathcal{I}\) there are \(c, c' \in \mathcal{I}'\) such that \(f(c) \neq f(c')\) (no user can be assigned a set of incompatible components); for all sets \(\mathcal{R}' \in \mathcal{R}\), and for all users \(u \in U\), there is a component \(c \in \mathcal{R}'\) 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 \(\text{COST} = \#\text{sum}[P, C : \text{assign}(C, U), \text{component}(C, P)]\) 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.

## 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,
% guess an assignment of components: each component to at most one user
\{assign(C,U): user(U,_,_)\} <= 1: − component(C,_,).

% check budget
\text{cost(U,COST)}: − user(U,_,_), COST = \# \text{sum\{P,C: assign(C,U), component(C,P)\}}.
\text{− user(U,MIN,MAX), cost(U,C), C < MIN.}
\text{− user(U,MIN,MAX), cost(U,C), C > MAX.}

% check incompatibilities and requirements
\text{− user(U,_,_), incompatibility(I,_,), assign(C,U): incompatibility(I,C).}
\text{− user(U,_,_), requirement(R,_,), not assign(C,U): requirement(R,C).}

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

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 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

<table>
<thead>
<tr>
<th>#users</th>
<th>#inst</th>
<th>clasp</th>
<th>WASP-St</th>
<th>WASP-SH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>sol.</td>
<td>avg t</td>
<td>sol.</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td>19</td>
<td>12.2</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>21</td>
<td>15</td>
<td>11.7</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>21</td>
<td>8</td>
<td>100.1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>21</td>
<td>3</td>
<td>252.9</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>21</td>
<td>0</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>21</td>
<td>0</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>21</td>
<td>0</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>147</td>
<td>45</td>
<td>43.7</td>
<td>44</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem</th>
<th>#inst</th>
<th>clasp</th>
<th>WASP-St</th>
<th>WASP-SH</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADF</td>
<td>200</td>
<td>200</td>
<td>23.3</td>
<td>120</td>
</tr>
<tr>
<td>Bottle Filling</td>
<td>100</td>
<td>100</td>
<td>5.7</td>
<td>100</td>
</tr>
<tr>
<td>Still Life</td>
<td>26</td>
<td>6</td>
<td>81.3</td>
<td>6</td>
</tr>
<tr>
<td>Still Life with Holes</td>
<td>120</td>
<td>70</td>
<td>180.0</td>
<td>90</td>
</tr>
<tr>
<td>Total</td>
<td>446</td>
<td>376</td>
<td>48.7</td>
<td>316</td>
</tr>
</tbody>
</table>
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; Abio 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

Mario Alviano has been partially supported by the POR CALABRIA FESR 2014-2020 project “DLV Large Scale” (CUP J28C17000220006), by the EU H2020 PON I&C 2014-2020 project “S2BDW” (CUP B28I17000250008), and by GNCS-INDAM. Carmine Dodaro has been partially by the Grant FFABR, “Fondo per il finanziamento delle attività base di ricerca, comma 295 della Legge di Bilancio 2017 (Legge 232/2016)”.

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.


Gebser, M., Kaufmann, B., and Schaub, T. 2009. The conflict-driven answer set solver clasp:


