AMO-aware Aggregates in Answer Set Programming

Mario Alviano, **Carmine Dodaro**, **Salvatore Fiorentino** and **Marco Maratea**
Department of Mathematics and Computer Science, University of Calabria, Italy
{mario.alviano, carmine.dodaro, marco.maratea}@unical.it, fiorentinosalvatore65@gmail.com

Abstract

Aggregates such as sum and count are among the most frequently used linguistic extensions of Answer Set Programming (ASP). At-most-one (AMO) constraints are a specific form of aggregates that excludes the simultaneous truth of multiple elements in a set. This article unleashes a powerful propagation strategy in case groups of elements in an aggregate are also involved in AMO constraints. In fact, the combined knowledge given by aggregates and AMO constraints significantly increases the effectiveness of search space pruning, resulting in sensible performance gains.

1 Introduction

Answer Set Programming (ASP) is a widely adopted formalism for knowledge representation and automated reasoning [Marek and Truszczyński, 1999; Niemelä, 1999]. In ASP, combinatorial problems are expressed in terms of logic rules comprising several linguistic constructs designed to ease the representation of complex knowledge. Solutions of the represented problem are provided in the form of stable models [Gelfond and Lifschitz, 1990], that is, classical models satisfying an additional stability condition: once the interpretation of the negative literals is fixed according to the model, all the knowledge encoded in the model is required to satisfy the reduced program.

In their simplest form, ASP programs comprise normal rules, each normal rule having a head atom and a body being a conjunction of literals. Intuitively, a head atom is required to be true whenever the associated body is true, so that stable models are classical models of the propositional knowledge base obtained by mapping normal rules to implications (body implies head). The stability condition also enforces other properties of stable models, such as being supported models (every true atom is the head of some rule with true body) [Fages, 1994] and unfounded-free models (the support is acyclic) [Dung, 1992]. All such properties are used to achieve efficient computation by combining a *conflict-driven clause learning* (CDCL) algorithm [Gebser *et al.*, 2012] with *propagators*. CDCL is a modern form of non-chronological backtracking based on the pattern *choose-propagate-learn* [Marques-Silva *et al.*, 2021]: a branching literal is heuristically *chosen* and *propagated* to infer deterministic consequences, where possible, and each inferred literal is associated with a reason being a clause (i.e., a set of literals, one of which is required to be true) among those associated with a propagator. Reasons are used to *learn* new clauses in case of conflict, so to effectively prune the search space.

Frequently, normal programs are extended by allowing the use of aggregates [Bartholomew *et al.*, 2011; Faber *et al.*, 2011b; Ferraris, 2011; Gelfond and Zhang, 2014; Liu *et al.*, 2010; Simons *et al.*, 2002], in particular SUMs, in rule bodies. In a SUM, Boolean literals are associated with weights, and the sum of the weights of true literals is required to satisfy a given (in)equality. For example, the inequality

$$1 \cdot x + 2 \cdot y + 2 \cdot z \geq 3 \tag{1}$$

(which is satisfied if $z = 1$ and either $x = 1$ or $y = 1$, or $x = y = 1$ hold) can be enforced by a SUM constraint. There is no general agreement on the stable model semantics of programs extended with arbitrary SUMs [Alviano *et al.*, 2023], but anyhow the results presented in this work apply to the computation of models rather than stable models, and are therefore applicable to any definition of stable model that selects among the classical models of a given program. To simplify the presentation, here we restrict SUMs to be constraints (with nonnegative weights and enforcing the sum of weights to reach a given bound). SUM constraints include at-most-one (AMO) constraints [Surynek, 2020] as special cases, where an AMO essentially inhibits truth of pairs of literals in a given set.

In ASP solvers, SUM (and possibly AMO) constraints are handled by specific propagators [Gebser *et al.*, 2009; Faber *et al.*, 2011a], aiming at delaying the materialization of clauses associated with such constraints. Roughly, the propagator associated with a SUM constraint identifies a literal as necessarily true whenever its weight is required to satisfy the associated inequality. In (1), once z is fixed to 0 (i.e., atom z is assigned false), $x = y = 1$ is inferred (i.e., atoms x, y are necessarily true). In case of conflict, and only in that case, the clauses $\{x, z\}$ and $\{y, z\}$ (i.e., $\overline{z}$ implies x and y) are possibly materialized if involved in the learning process. A conflict would arise if, for example, x and y are also involved in an AMO constraint, and therefore cannot be both true. In this case, the AMO constraint provides the clause

$\{\overline{x}, \overline{y}\}$, which is resolved with $\{y, z\}$ to obtain $\{\overline{x}, z\}$ and in turn with $\{x, z\}$ to obtain the learned clause $\{z\}$ (i.e., z must be true). It is important to understand that in this example the truth of z is actually logically entailed by the SUM and AMO constraints together (as the learned clause contains z alone). On the other hand, the propagation provided by SUM and AMO independently is insufficient to identify such a deterministic consequence, leading to a more expensive computation to learn such an implicit knowledge in the theory.

This work aims at defining a propagator capable of identifying deterministic consequences of combinations of SUM and AMO constraints. The underlying idea is that literals subject to AMO constraints occurring in a SUM can contribute to the sum of weights up to the largest of their weights rather than the sum of their weights. In the example, x and y can contribute up to 2 (rather than $1 + 2 = 3$) to the sum of weights, and therefore truth of z is required to satisfy the inequality. The main research questions addressed by this article are the following:

Q1. *What kind of combinations of SUM and AMO constraints are suitable to be jointly processed by a propagator?* We identify a promising combination of one SUM constraint and several AMO constraints. Each literal involved in the SUM is also involved in one AMO constraint (possibly being a trivially satisfied singleton), so to reduce the overestimate on the sum of weights and unleash the inference potential of the combined knowledge.

Q2. *Is it possible to extend the language with a construct to express such combinations?* We provide a new linguistic construct whose syntax extends that of SUM by allowing for specifying a partition of the literals in the SUM, so that each part imposes an AMO constraint.

Q3. *How much performance gain can the new propagator provide to an ASP solver?* We implemented the proposed propagator and assessed it empirically, reporting performance gain of several orders of magnitude.

2 Preliminaries

This section reports syntax and semantics of normal programs extended with SUM constraints (Section 2.1). Clauses and AMO constraints are given as special cases of SUM (Section 2.2). After that, the stable model search procedure implemented by modern ASP solvers is illustrated with a focus on the concept of propagator (Section 2.3).

2.1 Syntax and Semantics

Let $\mathcal{A}$ be a set of *atoms*. A *literal* is an atom possibly preceded by the *default negation* symbol $\sim$. Given a literal ℓ, $\overline{\ell}$ denotes the *complement* of ℓ, i.e., $\overline{p} = {\sim}p$ and $\overline{{\sim}p} = p$ for all $p \in \mathcal{A}$; for a set L of literals, $\overline{L}$ is $\{\overline{\ell} \mid \ell \in L\}$.

A *(normal) rule* has the form

$$p \leftarrow \ell_1, \ldots, \ell_n \tag{2}$$

where $n \geq 0$, p is an atom, and $\ell_1, \ldots, \ell_n$ are literals. A *SUM constraint* (or simply constraint) has the form

$$\text{SUM}\{w_1 : \ell_1;\ \cdots;\ w_n : \ell_n\} \geq b \tag{3}$$

where $n \geq 0$, $\ell_1, \ldots, \ell_n$ are distinct literals such that $\ell_i \neq \overline{\ell_j}$ (for all $1 \leq i < j \leq n$), and $b, w_0, \ldots, w_n$ are nonnegative integers. (We remark here that in ASP-Core-2 standard [Calimeri *et al.*, 2020] the above constraint is written as the headless rule `:- #sum{`w_1, ℓ_1 `:` ℓ_1`; ...;` w_n, ℓ_n `:` ℓ_n`} <` b`.`) A *program* is a set of rules and constraints.

Example 1 (Running example)**.** *Let Π_{run} be the following:*

$$\begin{array}{rll}
r_\alpha : & \alpha \leftarrow {\sim}\alpha' & \alpha \in \{x, y, z\} \\
r_{\alpha'} : & \alpha' \leftarrow {\sim}\alpha & \alpha \in \{x, y, z\} \\
\sigma_1 : & \text{SUM}\{1 : \overline{x};\ 1 : \overline{y}\} \geq 1 & \\
\sigma_2 : & \text{SUM}\{1 : x;\ 2 : y;\ 2 : z\} \geq 3 &
\end{array}$$

Note that there are six atoms (x, y, z, x', y', z'), six rules and two constraints.

We adopt the following notation. Atoms, rules, and constraints occurring in a program Π are respectively denoted as $atoms(\Pi)$, $rules(\Pi)$, and $constraints(\Pi)$. For a rule r of the form (2), $H(r) := p$ (the head atom of r), $B(r) := \{\ell_1, \ldots, \ell_n\}$ (the body literals of r), $B^+(r) := B(r) \cap \mathcal{A}$ (the atoms occurring in the positive body of r), and $B^-(r) := \overline{B(r) \setminus \mathcal{A}}$ (the atoms occurring in the negative body of r). For a constraint σ of the form (3), let $bnd_\sigma := b$ (the bound of σ is b), $wh_\sigma(\ell_i) := w_i$ (the weight of ℓ_i in σ is w_i; for all $i \in [1..n]$), and $lits_\sigma := \{\ell_1, \ldots, \ell_n\}$ (the literals occurring in σ are $\ell_1, \ldots, \ell_n$). (If σ is clear from the context, it is possibly omitted from the above notation.) Finally, let us define relation $\in$ as $(w_i : \ell_i) \in \sigma$ for $i \in [1..n]$, to be read as $(w_i : \ell_i)$ is an element in (the aggregation set of) σ.

Example 2 (Continuing Example 1)**.** *Rule r_x is such that $H(r_x) = x$, $B^+(r_x) = \emptyset$ and $B^-(r_x) = \{x'\}$. Regarding σ_2, $bnd_{\sigma_2} = 3$, $wh(y) = 2$, $y \in lits$, and $(2 : y) \in \sigma_2$.*

An *assignment* I is a set (or list) of literals such that $I \cap \overline{I} = \emptyset$; literals in I are true, literals in $\overline{I}$ are false, and all other literals are undefined. Abusing of notation, let $I(\ell) := 1$ if $\ell \in I$ and 0 otherwise (true literals assigned to 1); let $I^\uparrow(\ell) := 1$ if $\ell \notin \overline{I}$ and 0 otherwise (true and undefined literals assigned to 1). Moreover, in the subsequent sections, we will iteratively construct an assignment by using a list and occasionally consider its prefixes (i.e., assignments obtained in previous iterations). I is a *(total) interpretation* of a program Π if $I \cup \overline{I} = atoms(\Pi) \cup \overline{atoms(\Pi)}$. For total interpretations, relation $\models$ (*is model of*) is inductively defined as follows: for a literal ℓ, $I \models \ell$ if $\ell \in I$; for a rule r of the form (2), $I \models B(r)$ if $I \models \ell$ for all $\ell \in B(r)$, and $I \models r$ if $I \models H(r)$ whenever $I \models B(r)$; for a constraint σ of the form (3), $I \models \sigma$ if $\sum_{i=1}^{n} w_i \cdot I(\ell_i) \geq bnd_\sigma$; for a program Π, $I \models \Pi$ if $I \models r$ for all $r \in rules(\Pi)$ and $I \models \sigma$ for all $\sigma \in constraints(\Pi)$. The *reduct* Π^I of a program Π with respect to an interpretation I is $\{H(r) \leftarrow B^+(r) \mid r \in rules(\Pi), I \models r\}$. (Note that $constraints(\Pi^I) = \emptyset$.) 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.

Example 3 (Continuing Example 1)**.** *We first observe that, in all stable models of Π_{run}, α' is true whenever α is false, and vice versa ($\forall \alpha \in \{x, y, z\}$). $SM(\Pi_{run})$ contains $\{x, y\}$,*

$\{z\}$, and their supersets (negative literals and prime *atoms are omitted for simplicity).*

2.2 Clauses and AMO as a Special Cases

Given a set $\{\ell_1, \ldots, \ell_n\}$ of $n \geq 0$ literals, an *at-least-one (ALO)* constraint enforces truth of at least one literal in the set. It is also commonly referred to as *clause*, and can be expressed as the following SUM constraint:

$$\text{SUM}\{1 : \ell_1;\ \cdots\ ;\ 1 : \ell_n\} \geq 1 \tag{4}$$

Clause (4) is also written as $\{\ell_1, \ldots, \ell_n\}$ (if not ambiguous).

Given a set $\{\ell_1, \ldots, \ell_n\}$ of $n \geq 1$ literals, an AMO constraint inhibits truth of pairs of literals in the set. It can be expressed as the following SUM constraint:

$$\text{SUM}\{1 : \overline{\ell_1};\ \cdots\ ;\ 1 : \overline{\ell_n}\} \geq n - 1 \tag{5}$$

Essentially, it equivalently asks for falsifying all but possibly one literal in the set. In fact, given an interpretation I, $I \models (5)$ if $\sum_{i=1}^{n} I(\overline{\ell_i}) \geq n-1$, or equivalently $\sum_{i=1}^{n} I(\ell_i) \leq 1$. The AMO constraint (5) is compactly written $\text{AMO}\{\ell_1, \ldots, \ell_n\}$.

Example 4 (Continuing Example 1)**.** *Note that σ_1 is the clause $\{\overline{x}, \overline{y}\}$, or also the AMO constraint $\text{AMO}\{x, y\}$.*

2.3 Stable Model Search and Propagators

Stable model search is implemented in modern ASP solvers using a *conflict-driven clause learning* (CDCL) algorithm [Gebser *et al.*, 2012]. This algorithm relies on the pattern *choose-propagate-learn*. In a nutshell, the approach involves incrementally constructing a stable model, starting with an empty list I (representing the empty assignment). During each computational step, a branching literal is heuristically *chosen* for addition to I. Subsequently, it is *propagated* to introduce **new** deterministic consequences to I, where possible. Each deterministic consequence ℓ incorporated into I is associated with a *reason* being the clause comprising ℓ and the false literals leading to the inclusion of ℓ in I. A *conflict* arises if the complement of a deterministic consequence is already in I, and its analysis leads to *learn* new clauses via *(backward) resolution* starting from the reasons of the conflicting literals. (Recall that resolution combines two clauses $\{p\} \cup L$ and $\{\overline{p}\} \cup L'$ to obtain a new clause $L \cup L'$, where p is an atom, and L, L' are sets of literals.) During backward resolution, previously added literals are removed from I until the learned clause results in the inclusion of a new deterministic consequence, driving the search into a different branch. This iterative process continues until either I represents a stable model or the empty clause is learned. The latter means that the input program does not admit stable models.

A *propagator* is a module designed to compute deterministic consequences of a given assignment. The most basic among these modules is *unit propagation*: a literal ℓ is added to I if there is a clause $\{\ell\} \cup L$ such that $L \subseteq \overline{I}$, that is, if ℓ belongs to a clause that can be satisfied only by extending I with ℓ. In this case, $reason(\ell)$ is defined as $\{\ell\} \cup L$. Modern ASP solvers enrich the input program with clauses enforcing I to be a model of rules of the form (2) (i.e., $\{p, \overline{\ell_1}, \ldots, \overline{\ell_n}\}$), as well as clauses enforcing other required properties of stable models (which are out of the scope of this article).

Example 5. *Let Π have, among others, the rules*

$$x \leftarrow {\sim}z \qquad y \leftarrow {\sim}z \qquad w \leftarrow x, y$$

Hence, a modern ASP solver materializes the clauses

$$\{x, z\} \qquad \{y, z\} \qquad \{w, \overline{x}, \overline{y}\}$$

If the current assignment I is the list $[\overline{w}]$, and $\overline{z}$ is selected as the branching literal, I is updated to $[\overline{w}, \overline{z}]$, and unit propagation infers x, y from the first two clauses, and then $\overline{y}$ (a conflict) from the third clause. Hence, we have $I = [\overline{w}, \overline{z}, x, y, \overline{y}]$, $reason(x) = \{x, z\}$, $reason(y) = \{y, z\}$, and $reason(\overline{y}) = \{w, \overline{x}, \overline{y}\}$. By resolving $reason(y)$ and $reason(\overline{y})$, we obtain $\{w, \overline{x}, z\}$, which still contains more than one atom inferred from the branching point (namely, x and z). By resolving $\{w, \overline{x}, z\}$ and $reason(x)$, we obtain $\{w, z\}$, which contains only one atom inferred from the branching point (namely, z). CDCL therefore continues with $I = [\overline{w}, z]$ and $reason(z) = \{w, z\}$.

For a SUM constraint σ of the form (3), solvers typically employ a specific *aggregate propagator* [Gebser *et al.*, 2009; Faber *et al.*, 2011a], which essentially adds to I the literal ℓ_i ($i \in [1..n]$) if ℓ_i is required to (possibly) reach the bound b, i.e., if

$$\sum_{j \in [1..n], j \neq i} w_j \cdot J^{\uparrow}(\ell_j) < b \tag{6}$$

where J is the first prefix of I meeting the above condition. In this case, $reason(\ell_i)$ is $\{\ell_i\} \cup (lits_\sigma \cap \overline{J})$, that is, the false literals occurring in σ are enforcing truth of ℓ_i. In the special case of (5), that is, if σ is $\text{AMO}\{\ell_1, \ldots, \ell_n\}$, the literal $\overline{\ell_i}$ ($i \in [1..n]$) is added to I if there is ℓ_j, with $j \neq i$, such that $\ell_j \in I$. In this case, $reason(\overline{\ell_i})$ is $\{\overline{\ell_i}, \overline{\ell_j}\}$.

Example 6 (Continuing Example 1)**.** *If I is empty, no literal can be inferred from σ_1 and σ_2. If I is $[\overline{z}]$, then the application of* (6) *to the literals of σ_2 gives*

$$2 \cdot [\overline{z}]^{\uparrow}(y) + 2 \cdot [\overline{z}]^{\uparrow}(z) = 2 \cdot 1 + 2 \cdot 0 = 2 < 3$$
$$1 \cdot [\overline{z}]^{\uparrow}(x) + 2 \cdot [\overline{z}]^{\uparrow}(z) = 1 \cdot 1 + 2 \cdot 0 = 1 < 3$$
$$1 \cdot [\overline{z}]^{\uparrow}(x) + 2 \cdot [\overline{z}]^{\uparrow}(y) = 1 \cdot 1 + 2 \cdot 1 = 3 \not< 3$$

Hence, x and y are inferred with $reason(x) = \{x, z\}$ and $reason(y) = \{y, z\}$. Note that, once $I = [\overline{z}, x, y]$, the application of (6) *to σ_1 gives*

$$1 \cdot [\overline{z}, x, y]^{\uparrow}(\overline{y}) = 1 \cdot 0 = 0 < 1$$
$$1 \cdot [\overline{z}, x, y]^{\uparrow}(\overline{x}) = 1 \cdot 0 = 0 < 1$$

Therefore, a conflict is raised, say because $\overline{y}$ (or similarly $\overline{x}$) is added to I with $reason(\overline{y}) = \{\overline{x}, \overline{y}\}$.

3 AMOSUM Constraints

In this section, SUM constraints of the form (3) are replaced by the more general AMOSUM constraints (Section 3.1). An AMOSUM constraint combines one SUM constraint with a set of AMO constraints, in a convenient syntax that also results into a more effective propagator (Section 3.2), as formally proved in Section 3.3.

3.1 Syntax and Semantics

An *AMOSUM constraint* (or simply constraint from this point) σ has the form

$$\text{AMOSUM}\{w_1 : \ell_1\ [s_1];\ \cdots;\ w_n : \ell_n\ [s_n]\} \geq b \quad (7)$$

where $n \geq 0$, $\ell_1, \ldots, \ell_n$ are distinct literals such that $\ell_i \neq \overline{\ell_j}$ (for all $1 \leq i < j \leq n$), and $b, w_0, \ldots, w_n, s_1, \ldots, s_n$ are nonnegative integers. The notation introduced in Section 2.1 is further extended as follows: $parts_\sigma := \{s_1, \ldots, s_n\}$ (the literals in σ are partitioned into at most n parts $s_1, \ldots, s_n$), $part_\sigma(\ell_i) := s_i$ (the part of ℓ_i in σ is s_i, for all $i \in [1..n]$), and $lits_\sigma|_s := \{\ell \mid part_\sigma(\ell) = s\}$ (the literals in σ belonging to the part s). Moreover, relation $\in$ is now defined as $(w_i : \ell_i\ [s_i]) \in \sigma$ for all $i \in [1..n]$. Regarding relation *is model of*, for σ of the form (7), $I \models \sigma$ if $\sum_{i=1}^{n} w_i \cdot I(\ell_i) \geq bnd_\sigma$, and $\sum_{\ell \in lits_\sigma(s)} I(\ell) \leq 1$ for all $s \in parts_\sigma$. Essentially, the definition for (3) is extended by enforcing an AMO constraint on each part of σ.

Example 7 (Continuing Example 1)**.** *Π_{run} is rewritten by replacing σ_1 and σ_2 with*

$$\sigma_3 : \quad \text{AMOSUM}\{1 : x\ [1];\ 2 : y\ [1];\ 2 : z\ [2]\} \geq 3$$

Note that $parts_{\sigma_3} = \{1, 2\}$, $part_{\sigma_3}(x) = part_{\sigma_3}(y) = 1$, $part_{\sigma_3}(z) = 2$, $lits_{\sigma_3}|_1 = \{x, y\}$, and $lits_{\sigma_3}|_2 = \{z\}$.

3.2 Propagation

A constraint σ of the form (7) is associated with three different inference rules. First of all, the AMO inference given by the parts of σ: the literal $\overline{\ell}$ is added to I if there is $\ell' \in lits_\sigma|_{part_\sigma(\ell)}$ such that $\ell' \in I$. In this case, $reason(\overline{\ell})$ is $\{\overline{\ell}, \overline{\ell'}\}$. The second inference rule is the analogous of the one provided by the aggregate propagator (a literal is inferred true if it is required to reach the bound): the literal ℓ is added to I if J is the first prefix of I such that $lits_\sigma|_{part_\sigma(\ell)} \setminus \overline{J} = \{\ell\}$ (ℓ is the last undefined literal in its part), and

$$\sum_{s \in parts_\sigma \setminus \{part_\sigma(\ell)\}} mwh_\sigma(J, s) < bnd_\sigma \quad (8)$$

where $mwh_\sigma(J, s) := \max\{w \cdot J^\uparrow(\ell) \mid (w : \ell\ [s]) \in \sigma\}$ is the maximum weight that part s can contribute to the overall sum. In this case, $reason(\ell)$ is

$$lits_\sigma|_{part_\sigma(\ell)} \cup \bigcup_{s \in parts_\sigma \setminus \{part_\sigma(\ell)\}} rsn_\sigma(J, \ell, s), \quad (9)$$

where $rsn_\sigma(J, \ell, s) := \{\overline{\ell'}\}$ if $\ell' \in lits_\sigma|_s \cap J$ (i.e., a true literal in the part of ℓ), and $\{\ell' \in lits_\sigma|_s \mid wh_\sigma(\ell') > mwh_\sigma(s)\}$ otherwise (i.e., the false literals in the part of ℓ that could had increased the overall sum). The third inference rule has no counterpart in AMO or SUM constraints, and **enforces falsity of literals in a part** whose weight is guaranteed to be insufficient to reach the bound: the literal $\overline{\ell}$ is added to I if J is the first prefix of I such that

$$wh_\sigma(\ell) + \sum_{s \in parts_\sigma \setminus \{part_\sigma(\ell)\}} mwh_\sigma(J, s) < bnd_\sigma \quad (10)$$

In this case, $reason(\overline{\ell})$ is

$$\{\overline{\ell}\} \cup \bigcup_{s \in parts_\sigma \setminus \{part_\sigma(\ell)\}} rsn_\sigma(J, \ell, s). \quad (11)$$

Example 8 (Continuing Example 7)**.** *Already when I is empty, σ_3 infers z. In fact, z is the last undefined literal in part 2, and* (8) *gives*

$$\max\{1 \cdot [\,]^\uparrow(x), 2 \cdot [\,]^\uparrow(y)\} = \max\{1 \cdot 1, 2 \cdot 1\} = 2 < 3$$

From (9)*, $reason(z) = \{z\}$.*

Example 9. *Let us consider the following constraint:*

$$\sigma_4 : \quad \text{AMOSUM}\{1 : x\ [1]; 2 : y\ [1]; 2 : z\ [2]; 3 : w\ [2]\} \geq 3$$

If $I = [\overline{w}]$, the second inference rule associated with σ_4 infers z with $reason(z) = \{z, w\}$. The same holds if $I = [\overline{x}, \overline{w}]$, with the addition of y with $reason(y) = \{y, x, w\}$; note that w is included in $reason(y)$ because it could increase the overall sum. On the other hand, if $I = [\overline{y}, \overline{w}]$, then x and z are inferred with $reason(x) = \{x, y, w\}$ and $reason(z) = \{z, w, y\}$; note that y is included in $reason(z)$ because it could increase the overall sum (in this specific case y could be ignored, but it is not clear how to efficiently detect such conditions in general). Finally, if $I = [x, \overline{y}, \overline{w}]$, then z is inferred with $reason(z) = \{z, w, \overline{x}\}$; in this case, $\overline{x}$ is included because it enforces a value for part 1 (again, in this specific case it could be ignored).

Example 10. *Let us consider the following constraint:*

$$\sigma_5 : \quad \text{AMOSUM}\{1 : x\ [1]; 2 : y\ [1]; 2 : z\ [2]; 2 : w\ [2]\} \geq 4$$

For $I = [\,]$, the third inference rule of σ_5 infers $\overline{x}$ thanks to the application of (10)*:*

$$1 + \max\{2 \cdot [\,]^\uparrow(z), 2 \cdot [\,]^\uparrow(w)\} = 1 + 2 = 3 < 4$$

In this case, $reason(\overline{x}) = \{\overline{x}\}$. After that, note that the second inference rule is applicable, and y is inferred with $reason(y) = \{x, y\}$.

3.3 Properties

Semantically, it is clear that AMOSUM constraints do not extend the capabilities of the language. In fact, an AMOSUM σ constraint of the form (7) can be expressed in terms of (no more than $n + 1$) SUM constraints (by design):

$$\text{SUM}\{w_1 : \ell_1;\ \cdots;\ w_n : \ell_n\} \geq b$$
$$\text{AMO}\{lits_\sigma|_{s_i}\} \qquad \forall i \in [1..n]$$

On the other hand, it is also true that AMOSUM constraints immediately subsume SUM (and AMO) constraints.

Theorem 1. *Every SUM constraint can be expressed by an AMOSUM constraint.*

Proof. Let σ be a SUM constraint of the form (3). Construct the following AMOSUM constraint σ':

$$\text{AMOSUM}\{w_1 : \ell_1\ [1];\ \cdots;\ w_n : \ell_n\ [n]\} \geq b$$

Note that each part $s \in parts_{\sigma'}$ is a singleton, and therefore trivially satisfies $\sum_{\ell \in lits_{\sigma'}(s)} I(\ell) \leq 1$. Hence, $I \models \sigma$ if and only if $I \models \sigma'$, for every interpretation I. □

The real advantage gained by extending the language with AMOSUM constraints is in the associated inference rules, in particular the second and the third rules (while the first inference rule does not add anything to what can be inferred by AMO constraints). It can be shown that the second inference rule alone is sufficient to capture all deterministic consequences that can be identified by SUM constraints, and actually can result in a larger set of inferred literals.

Theorem 2. *Let σ be the AMOSUM constraint of the form* (7)*, σ' be the SUM constraint of the form* (3)*, and I be an assignment. Let L be the set of literals identified by the second inference rule associated with σ, and L' be the set of literals inferred by σ'. Hence, $L \supseteq L'$ holds, while $L' \supseteq L$ is not guaranteed.*

Proof. Let ℓ_i be inferred by σ' (for some $i \in [1..n]$), and let $S := parts_\sigma \setminus \{s_i\}$. By definition, (6) holds. Hence

$$\sum_{s \in S} mwh_\sigma(J, s) = \sum_{s \in S} \max_{(w:\ell\ [s]) \in \sigma} w \cdot J^\uparrow(\ell) \leq \sum_{j \in [1..n], j \neq i} w_j \cdot J^\uparrow(\ell_j) < b$$

and (8) holds as well, proving $L \supseteq L'$. A witness of $L' \not\supseteq L$ is given in Example 8. □

Moreover, it is relatively easy to observe that the third inference rule identifies deterministic consequences that are not computed by AMO and SUM constraints. In fact, AMO constraints deal with simple binary inferences, and SUM constraints only infer true literals in their aggregation set (while the third inference rule identifies false literals in the AMOSUM). A witness is given in Example 10.

4 Implementation and Experiments

The propagator outlined in Section 2.2 has been implemented in the ASP solver WASP [Alviano *et al.*, 2015] using its Python interface [Dodaro and Ricca, 2020]. This design choice is motivated by the intuitive interface and its seamless integration into the solver through command line options. In our implementation, AMOSUM constraints are represented by facts, which are interpreted by the Python propagator. Specifically, the representation of an AMOSUM constraint σ of the form (7) is the following:

$$group(\ell_i, w_i, s_i, \sigma) \leftarrow \qquad for\ all\ i \in [1..n]$$
$$lb(b, \sigma) \leftarrow$$

where *group* and *lb* are reserved predicates. Continuing Example 9, σ_4 is represented as follows:

$$group(x, 1, 1, \sigma_4) \leftarrow$$
$$group(y, 2, 1, \sigma_4) \leftarrow$$
$$group(z, 2, 2, \sigma_4) \leftarrow$$
$$group(w, 3, 2, \sigma_4) \leftarrow$$
$$lb(3, \sigma_4) \leftarrow$$

Moreover, as WASP already supports efficient propagators for handling AMO constraints, the Python propagator focuses on the other inference rules described in Section 3.2.

The implemented system, referred to as AMOWASP, was assessed empirically against the plain version of WASP [Alviano *et al.*, 2015] v. f3e4c56 and the state-of-the-art system CLINGO v. 5.4.0 [Gebser *et al.*, 2016]. All the tested systems use GRINGO (included in the binary of CLINGO) as grounder. Experiments were executed on an Intel Xeon 2.4 GHz server with 16 GB of memory.

4.1 Benchmarks

Synthetic Benchmark (SB). Designed to empirically assess the properties outlined in Section 3.3, the first benchmark comprises a program with a rule defining the partition and another one incorporating a SUM aggregate. The partition comprises 10 parts of uniform size $part_size \in \{10, 100, 1000\}$, with the i-th literal of each part having weight i. The bound of the SUM is set to $\alpha \cdot C_1$ (achievable) and $C_1 + \alpha \cdot (C_2 - C_1)$ (unachievable), where: $\alpha \in \{0.15, 0.45, 0.6, 0.9\}$; C_1 is the sum of the maximum weight in each part, i.e., $10 \cdot part_size$; C_2 is the sum of all weights, i.e., $5 \cdot part_size \cdot (part_size + 1)$. Hence, the benchmark comprises a total of 24 instances that are trivial for AMOWASP (they are solved without raising any conflict). In contrast, CLINGO and WASP cannot perform the same inferences (see Theorem 2), and we have measured the number of conflicts found by the two systems within the first minute of computation.

Graph Coloring (GC). In the Graph Coloring problem, the input consists of a graph and a set of colors, and the objective is to assign a color to each node so that connected nodes do not share the same color. Here, colors are also associated with weights, and the sum of weights is required to reach a certain threshold value. Instances are generated from those employed in the ASP competition [Calimeri *et al.*, 2016], with the colors red, green, blue, yellow, and cyan associated with the weights 2, 4, 8, 16, and 64 to, respectively. For each instance of n nodes, the threshold is set to $\alpha \cdot 64 \cdot n$, where $\alpha \in \{0.15, 0.45, 0.75\}$. Time and memory were limited to 20 minutes and 15 GB, respectively.

Knapsack (K). A set of *item types* is provided, each with an associated weight and value. Additionally, a knapsack capacity and a threshold are given. The objective is to determine whether it is possible to select a specific number of items from each type in such a way that the total weight remains within the knapsack capacity, while ensuring that the overall value reaches the specified threshold. Instances were randomly generated as follows. The number n of types varies from 10 to 55 with an increasing step of 5. The maximum number of items that can be selected for each type is fixed to $k = 20$. The average value of the items is denoted as v and used to define two critical thresholds: $C_1 := n \cdot v \cdot k$ and $C_2 := n \cdot v \cdot (k \cdot (k + 1))/2$. For each n, 10 instances are generated, categorized as follows: (T1–T3) 3, 3 and 2 instances with a threshold sampled from a uniform distribution within the intervals [0, C_1], [0, C_2], and [$0.1 \cdot C_1$, $1.1 \cdot C_1$], respectively; (T4) 2 instances with a threshold sampled from a normal distribution with a mean of C_1 and a variance of 5000. Time and memory were limited to 20 minutes and 15 GB, respectively.

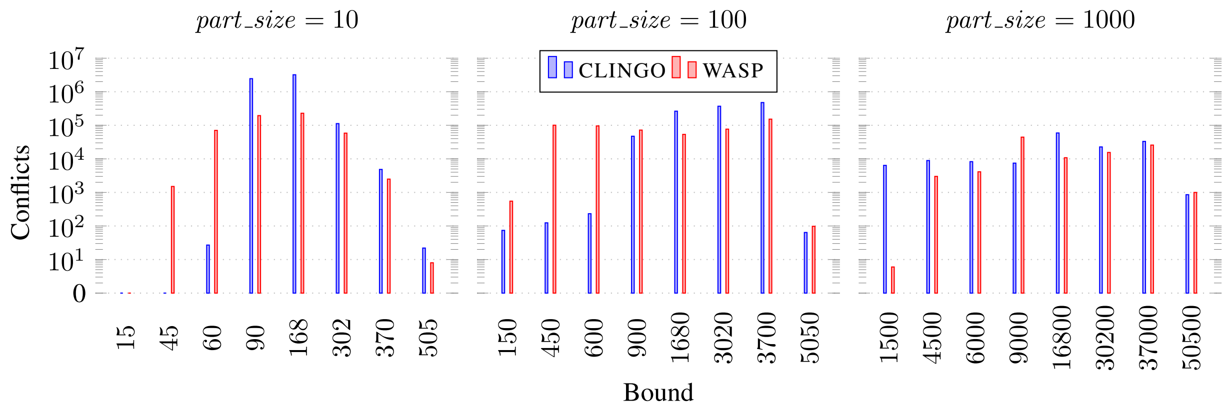

Figure 1: Number of conflicts on synthetic AMOSUM constraints comprising 10 parts of varying size and bound. For each *part_size* there are four satisfiable instances (the first four bounds) and four unsatisfiable instances (the last four bounds). AMOWASP is not reported in the plots because it solves all tested instances in this benchmark without raising any conflict.

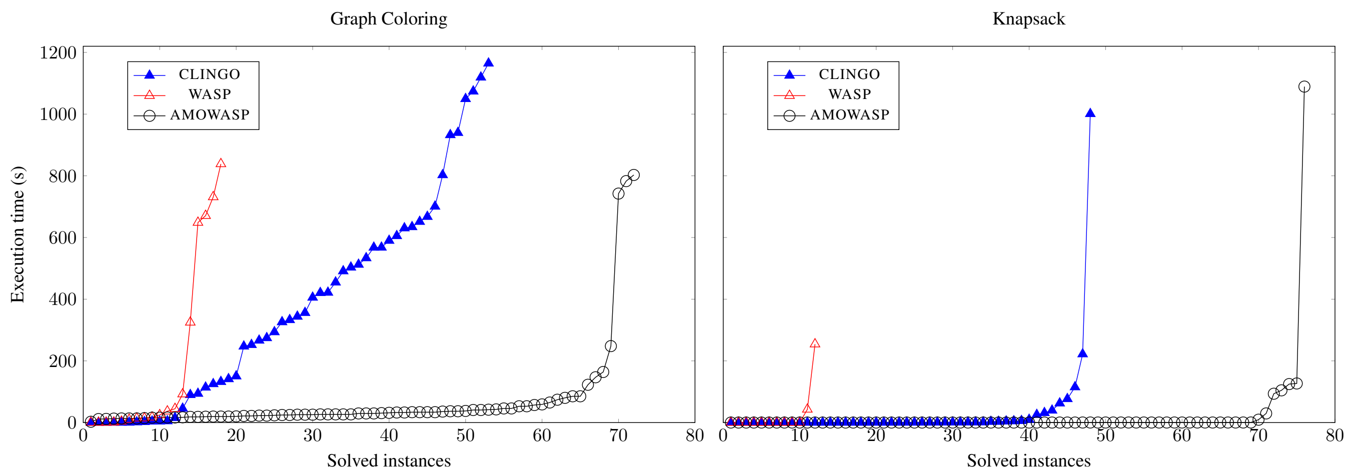

Figure 2: Number of solved instances (x-axis) within a time limit (y-axis) for Graph Coloring (left) and Knapsack (right).

4.2 Results

The experimental results for SB are summarized in Figure 1. We can see that when *part_size* is 10 or 100, CLINGO can handle the first three instances in an efficient way, solving them with only a few conflicts. However, this is not true when *part_size* is 1000. This suggests that the way in which CLINGO makes decisions (branching heuristic) works better for instances with SUM aggregates of small to medium size. Moreover, as we expected, the number of conflicts is related to the bound. Specifically, instances with the smallest and largest bounds have fewer conflicts. This happens because these instances are either not constrained enough or overly constrained.

The results obtained for GC and K are summarized in Figures 2–3. As a first observation, CLINGO proves to be more efficient than WASP in both benchmarks. As highlighted in SB, the branching heuristic of CLINGO is more effective than the one of WASP. However, the inferences made by AMOWASP completely fulfil the gap related to the heuristic, and, in fact, AMOWASP achieves the best performance. Regarding GC, AMOWASP successfully solves 72 instances, surpassing CLINGO and WASP, which solve 53 and 18 instances, respectively. We additionally observe that the advantage of AMOWASP is particularly evident in instances with $\alpha = 0.75$, where it outperforms CLINGO and WASP by solving 48 and 60 more instances, respectively. However, for $\alpha = 0.15$ and $\alpha = 0.45$, the Python implementation introduces overhead, leading to comparatively poorer performance. As for K, AMOWASP successfully solves 76 instances, outperforming both CLINGO and WASP, which solve 48 and 12 instances, respectively. We additionally observe that AMOWASP consistently outperforms WASP across all instance categories, solving 17, 25, 13, and 9 more instances of categories T1, T2, T3, and T4, respectively. In comparison to CLINGO, AMOWASP exhibits slightly poorer performance on T1 instances, where CLINGO solves one more instance. However, it demonstrates better performance on the other instances, solving 16, 4, and 9 more T2, T3 and T4 instances, respectively.

To sum up, AMOWASP outperforms WASP thanks to the technique presented in this paper, as AMOWASP is powered by WASP and thus inherits all of its heuristic parameters.

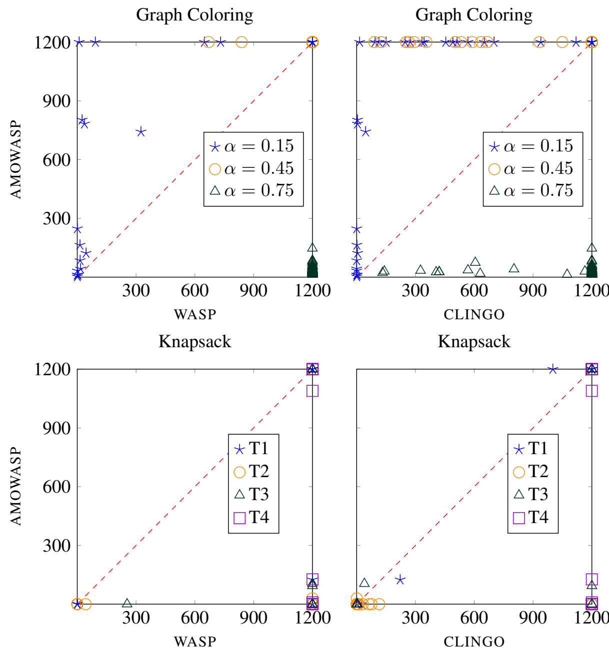

Figure 3: Instance-by-instance comparison on the execution time (in seconds) required to solve Graph Coloring and Knapsack. (Timeouts normalized to 1200 seconds.) Points below the red dashed line are instances in which AMOWASP is faster than the compared system.

5 Related Work

In this work, AMOSUM constraints have been added to the (propositional) language of ASP. Nonetheless, AMOSUMs can be added to any logic-based formalism extending propositional logic, such as SMT [Nieuwenhuis *et al.*, 2006] and CSP [Brailsford *et al.*, 1999]. From a computational point of view, there are two main approaches to extend the capabilities of logic-based languages, namely *propagator-based* and *translation-based*, discussed below.

Solvers adopting the *propagator-based* approach implement ad-hoc algorithms for extending assignments with literals that are required to be true (Section 2.3). The way AMOSUM constraints are handled in this work (Section 3.2) belongs to this category. Regarding the literature, the state-of-the-art system CLINGO [Gebser *et al.*, 2019] implements a hybrid approach for handling programs with aggregates [Gebser *et al.*, 2009]: aggregates comprising a limited number of literals are transformed into regular rules, according to some translation-based approach, and the threshold for the number of literals is adjustable through the command-line interface; other aggregates, including the ones representing AMO constraints, are instead handled by the propagator described in Section 2.3. A similar approach is implemented also in IDP [Denecker and De Cat, 2010; Bogaerts *et al.*, 2016] and WASP [Alviano *et al.*, 2018]. Moreover, CLINGO and WASP offer external interfaces based on Python to define custom propagators [Cabalar *et al.*, 2023; Dodaro and Ricca, 2020]. The propagator defined in Section 3.2 is powered by the Python interface of WASP.

Translation-based approaches consist of compilations of aggregates into alternative constructs. In the context of ASP, the similarities between aggregates and pseudo-Boolean constraints led to the adoption of some compilations of pseudo-Boolean constraints into clauses [Aavani *et al.*, 2013]. These approaches include adder circuits and binary decision diagrams [Abío *et al.*, 2012; Eén and Sörensson, 2006], sorting networks and watchdogs [Bailleux *et al.*, 2009]. Regarding ASP solvers, many of these translations are incorporated into LP2SAT and LP2NORMAL [Bomanson *et al.*, 2014; Bomanson and Janhunen, 2013], where the first solver generates CNF formulas, while the second produces normal rules. Another translation-based approach is 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]. Finally, in the specific case of AMO constraints, translation-based approaches offer various encoding options, such as pairwise (binomial), binary (bitwise), commander, product, sequential counter, and bimander encodings (the reader is referred to [Nguyen *et al.*, 2020] for a recent comparison of these encodings). We remark here that the AMOSUM propagator introduced in this article can be combined with AMO constraint compilers, essentially by replacing the first inference rule defined in Section 3.2 with the compiled clauses.

6 Conclusion

AMOSUM constraints extend the language of ASP with a construct designed to empower search space pruning capabilities of solvers, rather than the expressive power of the language. In fact, the construct compactly represents groups of constraints that are available in ASP with the aim of combining their knowledge to detect more deterministic consequences than those identified individually by the grouped constraints. The empirical assessment of our implementation of AMOSUM constraints confirms that their usage can provide sensible performance gains.

There are several lines of future research. First of all, AMOSUM constraints replace specific kind of constraints, precisely one SUM constraint with target bound and several at-most-one constraints, and therefore it will be interesting to search for other kinds of constraints suitable for a combined propagator. Moreover, SUM constraints are usually normalized to have a target bound, for example by flipping polarities of literals and adjusting numbers in case the constraint imposes to not exceed a given bound. Such a normalization is unnatural for AMOSUM, as flipping polarities of literals essentially breaks the AMO constraints (AMO$\{\ell_1, \dots, \ell_n\}$ does not imply AMO$\{\overline{\ell_1}, \dots, \overline{\ell_n}\}$); essentially, the normalization of a constraint of the form AMOSUM$\{w_1 : \ell_1\ [s_1]; \cdots; w_n : \ell_n\ [s_n]\} \leq b$ would result into a SUM constraint. Another interesting research line is therefore the generalization of the construct introduced in this article to accommodate such transformations. In addition, AMOSUM constraints might improve the performance of existing encodings, especially in the context of scheduling [Dodaro and Maratea, 2017].

Finally, we remark here that all the material required to replicate the experiments are available at https://zenodo.org/records/11115982 [Alviano *et al.*, 2024].

Acknowledgments

This work was partially supported by Italian Ministry of University and Research (MUR) under PRIN project PRODE "Probabilistic declarative process mining", CUP H53D23003420006 under PNRR project FAIR "Future AI Research", CUP H23C22000860006, under PNRR project Tech4You "Technologies for climate change adaptation and quality of life improvement", CUP H23C22000370006, and under PNRR project SERICS "SEcurity and RIghts in the CyberSpace", CUP H73C22000880001; by Italian Ministry of Health (MSAL) under POS projects CAL.HUB.RIA (CUP H53C22000800006) and RADIOAMICA (CUP H53C22000650006); by Italian Ministry of Enterprises and Made in Italy under project STROKE 5.0 (CUP B29J23000430005); and by the LAIA lab (part of the SILA labs). Mario Alviano and Carmine Dodaro are members of the Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

References

[Aavani *et al.*, 2013] Amir Aavani, David G. Mitchell, and Eugenia Ternovska. New encoding for translating pseudo-boolean constraints into SAT. In *SARA*. AAAI, 2013.

[Abío *et al.*, 2012] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Valentin Mayer-Eichberger. A new look at bdds for pseudo-boolean constraints. *J. Artif. Intell. Res.*, 45:443–480, 2012.

[Alviano *et al.*, 2015] Mario Alviano, Carmine Dodaro, Nicola Leone, and Francesco Ricca. Advances in WASP. In *LPNMR*, volume 9345 of *LNCS*, pages 40–54. Springer, 2015.

[Alviano *et al.*, 2018] Mario Alviano, Carmine Dodaro, and Marco Maratea. Shared aggregate sets in answer set programming. *Theory Pract. Log. Program.*, 18(3-4):301–318, 2018.

[Alviano *et al.*, 2023] Mario Alviano, Wolfgang Faber, and Martin Gebser. Aggregate semantics for propositional answer set programs. *Theory Pract. Log. Program.*, 23(1):157–194, 2023.

[Alviano *et al.*, 2024] Mario Alviano, Carmine Dodaro, Salvatore Fiorentino, and Marco Maratea. Dataset: AMO-aware Aggregates in Answer Set Programming. DOI: 10.5281/zenodo.11115982, 2024.

[Bailleux *et al.*, 2009] Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New encodings of pseudo-boolean constraints into CNF. In *SAT*, volume 5584 of *LNCS*, pages 181–194. Springer, 2009.

[Bartholomew *et al.*, 2011] Michael Bartholomew, Joohyung Lee, and Yunsong Meng. First-order semantics of aggregates in answer set programming via modified circumscription. In *Logical Formalizations of Commonsense Reasoning, AAAI Spring Symposium*. AAAI, 2011.

[Bogaerts *et al.*, 2016] Bart Bogaerts, Joachim Jansen, Broes De Cat, Gerda Janssens, Maurice Bruynooghe, and Marc Denecker. Bootstrapping inference in the IDP knowledge base system. *New Gener. Comput.*, 34(3):193–220, 2016.

[Bomanson and Janhunen, 2013] Jori Bomanson and Tomi Janhunen. Normalizing cardinality rules using merging and sorting constructions. In *LPNMR*, volume 8148 of *LNCS*, pages 187–199. Springer, 2013.

[Bomanson *et al.*, 2014] Jori Bomanson, Martin Gebser, and Tomi Janhunen. Improving the normalization of weight rules in answer set programs. In *JELIA*, volume 8761 of *LNCS*, pages 166–180. Springer, 2014.

[Brailsford *et al.*, 1999] Sally C. Brailsford, Chris N. Potts, and Barbara M. Smith. Constraint satisfaction problems: Algorithms and applications. *Eur. J. Oper. Res.*, 119(3):557–581, 1999.

[Cabalar *et al.*, 2023] Pedro Cabalar, Jorge Fandinno, Torsten Schaub, and Philipp Wanko. On the semantics of hybrid ASP systems based on clingo. *Algorithms*, 16(4):185, 2023.

[Calimeri *et al.*, 2016] Francesco Calimeri, Martin Gebser, Marco Maratea, and Francesco Ricca. Design and results of the fifth answer set programming competition. *Artif. Intell.*, 231:151–181, 2016.

[Calimeri *et al.*, 2020] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. Asp-core-2 input language format. *Theory Pract. Log. Program.*, 20(2):294–309, 2020.

[Denecker and De Cat, 2010] Marc Denecker and Broes De Cat. Dpll(agg): An efficient smt module for aggregates. In *Logic and Search, Edinburgh, 15 July 2010*, 2010.

[Dodaro and Maratea, 2017] Carmine Dodaro and Marco Maratea. Nurse scheduling via answer set programming. In *LPNMR*, volume 10377 of *LNCS*, pages 301–307. Springer, 2017.

[Dodaro and Ricca, 2020] Carmine Dodaro and Francesco Ricca. The external interface for extending WASP. *Theory Pract. Log. Program.*, 20(2):225–248, 2020.

[Dung, 1992] Phan Minh Dung. On the relations between stable and well-founded semantics of logic programs. *Theor. Comput. Sci.*, 105(1):7–25, 1992.

[Eén and Sörensson, 2006] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. *J. Satisf. Boolean Model. Comput.*, 2(1-4):1–26, 2006.

[Faber *et al.*, 2011a] Wolfgang Faber, Nicola Leone, Marco Maratea, and Francesco Ricca. Look-back techniques for ASP programs with aggregates. *Fundam. Informaticae*, 107(4):379–413, 2011.

[Faber *et al.*, 2011b] Wolfgang Faber, Gerald Pfeifer, and Nicola Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artif. Intell.*, 175(1):278–298, 2011.

[Fages, 1994] François Fages. Consistency of clark's completion and existence of stable models. *Methods Log. Comput. Sci.*, 1(1):51–60, 1994.

[Ferraris and Lifschitz, 2005] Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. *Theory Pract. Log. Program.*, 5(1-2):45–74, 2005.

[Ferraris, 2011] Paolo Ferraris. Logic programs with propositional connectives and aggregates. *ACM Trans. Comput. Log.*, 12(4):25, 2011.

[Gebser *et al.*, 2009] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. On the implementation of weight constraint rules in conflict-driven ASP solvers. In *ICLP*, volume 5649 of *LNCS*, pages 250–264. Springer, 2009.

[Gebser *et al.*, 2012] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187:52–89, 2012.

[Gebser *et al.*, 2016] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In *Technical Communications of ICLP*, volume 52 of *OASIcs*, pages 2:1–2:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[Gebser *et al.*, 2019] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.*, 19(1):27–82, 2019.

[Gelfond and Lifschitz, 1990] Michael Gelfond and Vladimir Lifschitz. Logic programs with classical negation. In *Logic Programming: Proc. of the Seventh International Conference*, pages 579–597, 1990.

[Gelfond and Zhang, 2014] Michael Gelfond and Yuanlin Zhang. Vicious circle principle and logic programs with aggregates. *Theory and Practice of Logic Programming*, 14(4-5):587–601, 2014.

[Giunchiglia *et al.*, 2006] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. Answer set programming based on propositional satisfiability. *J. Autom. Reason.*, 36(4):345–377, 2006.

[Giunchiglia *et al.*, 2008] Enrico Giunchiglia, Nicola Leone, and Marco Maratea. On the relation among answer set solvers. *Ann. Math. Artif. Intell.*, 53(1-4):169–204, 2008.

[Lierler and Maratea, 2004] Yuliya Lierler and Marco Maratea. Cmodels-2: Sat-based answer set solver enhanced to non-tight programs. In *LPNMR*, volume 2923 of *LNCS*, pages 346–350. Springer, 2004.

[Liu *et al.*, 2010] Lengning Liu, Enrico Pontelli, Tran Cao Son, and Miroslaw Truszczynski. Logic programs with abstract constraint atoms: The role of computations. *Artif. Intell.*, 174(3-4):295–315, 2010.

[Marek and Truszczyński, 1999] Victor Marek and Miroslaw Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-year Perspective*, pages 375–398, 1999.

[Marques-Silva *et al.*, 2021] João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, volume 336 of *FAIA*, pages 133–182. IOS Press, 2021.

[Nguyen *et al.*, 2020] Van-Hau Nguyen, Van-Quyet Nguyen, Kyungbaek Kim, and Pedro Barahona. Empirical study on sat-encodings of the at-most-one constraint. In *SMA*, pages 470–475. ACM, 2020.

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

[Nieuwenhuis *et al.*, 2006] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T). *J. ACM*, 53(6):937–977, 2006.

[Simons *et al.*, 2002] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.

[Surynek, 2020] Pavel Surynek. At-most-one constraints in efficient representations of mutex networks. In *ICTAI*, pages 170–177. IEEE, 2020.