AMO-Aware Optimization in Answer Set Programming*

Mario Alviano[1], Carmine Dodaro[1] and Salvatore Fiorentino[1,*,†]

[1]Department of Mathematics and Computer Science, University of Calabria

Abstract

Answer Set Programming (ASP) is a declarative paradigm for modeling and solving complex combinatorial problems, where solutions are represented as stable models of logic programs. Modern ASP systems support optimization constructs, such as MAXIMIZE, to guide the search toward preferred solutions. Taking advantage of AMOSUM constraints, a recently introduced construct combining sum and *at-most-one* (AMO) reasoning to enable stronger inference during search, we introduce AMOMAXIMIZE, a novel maximization statement that integrates AMO constraints directly into the objective function. Although AMOMAXIMIZE does not extend the expressive power of ASP, it represents additional structural information that can be exploited by the solver to improve propagation and reduce the search space. Our implementation of AMOMAXIMIZE relies on a reformulation of the objective into a sequence of AMOSUM constraints handled under a Linear SAT-UNSAT (LSU) optimization scheme, progressively tightening the bound until optimality is reached. An experimental evaluation on a multifaceted set of benchmarks shows that, in specific scenarios, our approach improves performance compared to CLINGO.

Keywords

Answer Set Programming, Logic Programming, Constraints

1. Introduction

Answer Set Programming (ASP) is a well-established declarative programming paradigm widely used for knowledge representation and reasoning [1, 2]. ASP allows complex problems to be encoded as logic programs, whose solutions are called *stable models or answer sets*. Over the years, ASP has been adopted in several industrial applications [3, 4, 5, 6], thanks to an intuitive syntax and semantics [7, 8, 9, 10] and the availability of efficient solvers [11, 12].

Classical ASP programs are defined by rules of the form $head \leftarrow body$, where the head is a propositional atom derived whenever the body (a conjunction of literals) is satisfied. Rules with an unsatisfiable head are called *constraints*, and their body must be falsified in every stable model. To improve the succinctness and expressiveness of the language, several new linguistic constructs have been proposed over the years, including *sum aggregates*, which allow weighted sums of literals to appear in rule bodies. In practice, sum aggregates are frequently used as *sum constraints* to filter stable models whose weighted sum falls below a given bound.

A well-established technique for handling aggregates in ASP solvers is the use of dedicated procedures called *propagators*, which derive deterministic consequences from partial interpretations to ensure that the semantics of the aggregate is respected. The earlier a propagator can detect such consequences, the more effectively the search space is pruned. A particularly useful special case is the *At-Most-One* (AMO) constraint, which enforces that at most one literal in a given set is true. In many real-world problems, sum constraints and AMO constraints naturally co-occur. Consider, for instance, the Traveling Salesman Problem (TSP), where a tour must visit all cities exactly once while reaching a given revenue: a SUM constraint ensures the revenue exceeds a threshold, while an AMO constraint ensures that each city has at most one successor in the tour.

CILC 2026: The 41st Italian Conference on Computational Logic, June 23–25, 2026, Ferrara, Italy

*Mario Alviano, Carmine Dodaro, Salvatore Fiorentino

†These authors contributed equally.

mario.alviano@unical.it (M. Alviano); carmine.dodaro@unical.it (C. Dodaro); salvatore.fiorentino@unical.it (S. Fiorentino)

https://orcid.org/0000-0002-2052-2063 (M. Alviano); https://orcid.org/0000-0002-5617-5286 (C. Dodaro); 0009-0004-6531-4139 (S. Fiorentino)

published 2026-08-24

Recently, Alviano et al. 2024 proposed AMOSUM, a new ASP construct that combines a SUM constraint with a set of AMO constraints into a single formulation [13]. The authors devised a dedicated AMOSUM propagator and proved that it can detect certain deterministic consequences earlier than treating the two constraints separately, yielding significant improvements in solving time on challenging instances. A similar case arises in the context of optimization: ASP supports the expression of preferences over stable models through *optimization statements*, and in particular the MAXIMIZE statement, which directs the solver to find stable models that maximize a weighted sum of *soft literals*. As with sum constraints, soft literals are frequently subject to AMO constraints in practice, suggesting that the insights behind AMOSUM can be naturally carried over to the optimization setting. Returning to the TSP example: if stable models with higher revenue are preferred, the objective function inherits an AMO constraint, since at most one successor city can be chosen per city in the tour.

In this work, we present the AMOMAXIMIZE statement, a new maximization construct that extends MAXIMIZE by embedding AMO constraints directly into the objective function. Although the expressiveness of the language is preserved, the additional structural information provided by AMOMAXIMIZE enables the solver to perform stronger propagation and more aggressive search-space pruning, with particular benefits when certifying optimality. From an implementation standpoint, AMOMAXIMIZE is realized on top of the AMOSUM propagator via a *Linear SAT-UNSAT (LSU)* strategy: the statement is translated into a sequence of AMOSUM constraints with a progressively tightened bound, each demanding a strictly better solution, until no further improvement can be certified.

The empirical evaluation on a range of benchmarks confirms that in some cases AMOMAXIMIZE improves the performance of the state-of-the-art solver CLINGO [11].

The reminding of this paper is structured as follows. Section 2 introduces the necessary preliminaries. Section 3 recalls the AMOSUM constraint. Section 4 presents the AMOMAXIMIZE statement and its implementation. Section 5 reports the empirical evaluation. Section 6 discusses related work. Section 7 concludes the paper.

2. Preliminaries

2.1. Syntax and Semantics

Let $\mathcal{A}$ be a set of propositional *atoms*. A *literal* is either an atom or its *default negation*, denoted by $\sim$.

For a literal ℓ, we denote by $\overline{\ell}$ its *complement*, defined as follows: $\overline{p} = \sim p$ and $\overline{\sim p} = p$ for every $p \in \mathcal{A}$. For a set L of literals, let $\overline{L} = \overline{\ell} \mid \ell \in L$. Given two sets of literals L and X, we define $L \dot\cup X = (L \setminus \overline{X}) \cup X$.

A *rule* is an expression of the form

$$p \leftarrow \ell_1, \ldots, \ell_n \tag{1}$$

where $n \geq 0$, $p \in \mathcal{A}$, and each ℓ_i is a literal. A *SUM constraint* (or simply *constraint*) has the form

$$\text{SUM}\{w_1 : \ell_1;\ \cdots\ ;\ w_n : \ell_n\} \geq b \tag{2}$$

where $n \geq 0$, the literals $\ell_1, \ldots, \ell_n$ are pairwise distinct and satisfy $\ell_i \neq \overline{\ell_j}$ for all $1 \leq i < j \leq n$, and $w_1, \ldots, w_n, b$ are nonnegative integers.

A *program* is a set of rules and constraints.

Example 2.1 (Running example)**.** Let Π_{run} be the following:

$$\begin{array}{rll}
r_\alpha: & \alpha \leftarrow \sim\alpha' & \alpha \in \{a,b,c,d\} \\
r_{\alpha'}: & \alpha' \leftarrow \sim\alpha & \alpha \in \{a,b,c,d\} \\
\sigma_1: & \text{SUM}\{1:\overline{a};\ 1:\overline{b}\ \} \geq 1 & \\
\sigma_2: & \text{SUM}\{1:\overline{c};\ 1:\overline{d}\ \} \geq 1 & \\
\sigma_3: & \text{SUM}\{1:a;\ 2:b;\ 2:c;\ 2:d\} \geq 4 &
\end{array}$$

Note that there are eight atoms $(a, b, c, d, a', b', c', d')$, eight rules and 3 constraints.

For a program Π, we denote by $atoms(\Pi)$, $rules(\Pi)$, and $constraints(\Pi)$ the sets of atoms, rules, and constraints occurring in Π, respectively. For a rule r of the form (1), we define: $H(r) := p$ (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 positive body atoms of r), and $B^-(r) := \overline{B(r) \setminus \mathcal{A}}$ (the negative body atoms of r). For a constraint σ of the form (2), we define: $bnd(\sigma) := b$ (the bound of σ), $wh_\sigma(\ell_i) := w_i$ (the weight of literal ℓ_i in σ, for each $i \in [1..n]$), and $lits(\sigma) := \{\ell_1, \ldots, \ell_n\}$ (the set of literals occurring in σ). When σ is clear from the context, we may omit it from the above notation. Finally, we write $(w_i : \ell_i) \in \sigma$ to denote that $(w_i : \ell_i)$ is an element of the aggregation set of σ, for $i \in [1..n]$.

Example 2.2 (Continuing Example 2.1)**.** Rule r_a is such that $H(r_a) = a$, $B^+(r_a) = \emptyset$ and $B^-(r_a) = \{a'\}$. Regarding σ_3, $bnd_{\sigma_3} = 4$, $wh(b) = 2$, $b \in lits_{\sigma_3}$, and $(2 : b) \in \sigma_3$.

An *assignment* I is an ordered set 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. By abuse of notation, we define $I(\ell) := 1$ if $\ell \in I$ and 0 otherwise, and $I^\uparrow(\ell) := 1$ if $\ell \notin \overline{I}$ and 0 otherwise; intuitively, $I(\ell)$ assigns 1 to true literals only, while $I^\uparrow(\ell)$ assigns 1 to both true and undefined literals. Moreover, since we will iteratively construct assignments as lists, we occasionally consider their *prefixes*, i.e., the assignments obtained at previous iterations. An assignment I is a *(total) interpretation* of a program Π if $I \cup \overline{I} = atoms(\Pi) \cup \overline{atoms(\Pi)}$. For total interpretations, the satisfaction 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 (1), $I \models B(r)$ 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 (2), $I \models \sigma$ if $\sum_{i=1}^{n} w_i \cdot I(\ell_i) \geq bnd_\sigma$; and
- 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 defined as $\{H(r) \leftarrow B^+(r) \mid r \in rules(\Pi),\ I \models r\}$; note that $constraints(\Pi^I) = \emptyset$. An interpretation I is a *stable model* of Π if $I \models \Pi$ and no $J \subset I$ satisfies $J \models \Pi^I$ [14]. We denote by $SM(\Pi)$ the set of all stable models of Π.

Optimal Stable Models. ASP also supports optimization problems, where some stable models are preferred over others. An optimization problem is defined by a pair (Π, μ), where Π is an ASP program and μ is an optimization statement. An *optimization statement* has the form

$$\textsc{maximize}\{w_1 : \ell_1;\ \cdots\ ;\ w_n : \ell_n\} \tag{3}$$

where $n \geq 0$, $\ell_1, \ldots, \ell_n$ are distinct literals with $\ell_i \neq \overline{\ell_j}$ for all $1 \leq i < j \leq n$, and $w_1, \ldots, w_n$ are nonnegative integers. Intuitively, each true literal ℓ_i contributes its weight w_i to the objective function, which is to be maximized. The objective value of an interpretation I, denoted by $obj(I, \mu)$, corresponds to $\sum_{i=1}^{n} I(\ell_i) \cdot w_i$. A stable model $M \in SM(\Pi)$ is *optimal* if no other stable model achieves a strictly higher objective value. The set of optimal stable models is defined as follows

$$OSM(\Pi, \mu) := \{M \in SM(\Pi) \mid \forall M' \in SM(\Pi),\ obj(M', \mu) \leq obj(M, \mu)\}. \tag{4}$$

Example 2.3 (Continuing Example 2.1)**.** Let Π_{runmax} be $(\Pi_{run} \setminus \{\sigma_3\}) \cup \{\mu_1\}$ where μ is the following maximization statement:

$$\mu_1 :\quad \textsc{maximize}\{1 : a;\ 2 : b;\ 2 : c;\ 2 : d\}$$

Let $I = \{a, c\}$, I is a stable model for Π_{runmax}, i.e., $I \in SM(\Pi_{runmax})$ but is not an optimal model given that $I' = \{b, c\} \in SM(\Pi_{runmax})$ and $obj(I) < obj(I')$, in particular, $obj(I) = 3$ and $obj(I') = 4$. Given that there is no model with higher objective value then $I' \in OSM(\Pi_{runmax})$.

2.2. Clauses and AMO as a Special Instance

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

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

Clause (5) is also written (if not ambiguous) as

$$\{\ell_1, \ldots, \ell_n\} \tag{6}$$

Given two clauses C and D of the form (6) and a literal ℓ such that $\ell \in C$ and $\overline{\ell} \in D$, the *resolution* step [15, 16] of C and D upon ℓ, written $C \otimes_\ell D$, is defined as $(C \setminus \{\ell\}) \cup (D \setminus \{\overline{\ell}\})$. Intuitively, since no interpretation can simultaneously satisfy both ℓ and $\overline{\ell}$, if $C, D \in \Pi$ then either $C \setminus \{\ell\}$ or $D \setminus \{\overline{\ell}\}$ must be satisfied, and hence so must their union $(C \setminus \{\ell\}) \cup (D \setminus \{\overline{\ell}\})$. Note that any interpretation I satisfying both C and D also satisfies $C \otimes_\ell D$. Given a set $\{\ell_1, \ldots, \ell_n\}$ of $n \geq 1$ literals, an *at-most-one* (AMO) constraint requires that at most one literal in the set be true. It can be expressed as the following SUM constraint:

$$\text{SUM}\{1 : \overline{\ell_1};\ \cdots\ ;\ 1 : \overline{\ell_n}\} \geq n - 1 \tag{7}$$

Equivalently, it requires all but at most one literal in the set to be false. Formally, given an interpretation I, we have $I \models (7)$ iff $\sum_{i=1}^{n} I(\ell_i) \leq 1$, or equivalently $\sum_{i=1}^{n} I(\overline{\ell_i}) \geq n - 1$. The AMO constraint (7) is compactly written $\text{AMO}\{\ell_1, \ldots, \ell_n\}$.

Example 2.4 (Continuing Example 2.1)**.** Note that σ_1 is the clause $\{\overline{a}, \overline{b}\}$, or also the AMO constraint $\text{AMO}\{a, b\}$.

2.3. (Optimal) Stable Model Search and Propagators

Stable model search in state-of-the-art ASP solvers is implemented via *conflict-driven clause learning* (CDCL) [17, 18], following the *choose–propagate–learn* pattern. The key idea is to incrementally construct a stable model starting from an empty assignment I. At each step, a literal is *chosen* according to a branching heuristic and added to I. It is then *propagated*: any literal ℓ that becomes a deterministic consequence is added to I together with its *reason*, i.e., the clause comprising ℓ and the false literals that forced its inclusion. 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 $\{\overline{p}\} \cup L'$, written $(\{p\} \cup L) \otimes_p (\{\overline{p}\} \cup L')$, to obtain a new clause $L \cup L'$, where Π 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 process is repeated until either I represents a stable model or the empty clause is derived, the latter indicating that the program admits no stable models.

A *propagator* is a procedure that computes deterministic consequences of a given assignment. The most basic one is *unit propagation*: a literal ℓ is added to I whenever there exists a clause $\{\ell\} \cup L$ with $L \subseteq \overline{I}$, i.e., whenever ℓ belongs to a clause that can only be satisfied by extending I with ℓ. In this case, $\mathit{reason}(\ell) := \{\ell\} \cup L$. Modern ASP solvers augment the input program with clauses that enforce I to be a model of each rule of the form (1) (i.e., $\{p, \overline{\ell_1}, \ldots, \overline{\ell_n}\}$), along with further clauses enforcing additional properties of stable models (whose details are beyond the scope of this article).

Example 2.5. Let Π have, among others, the rules

$$a \leftarrow \sim c \quad b \leftarrow \sim c \quad d \leftarrow a, b$$

Hence, a modern ASP solver materializes the clauses

$$\{a, c\} \quad \{b, c\} \quad \{d, \overline{a}, \overline{b}\}$$

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

For a SUM constraint σ of the form (2), solvers typically employ a dedicated *aggregate propagator* [19, 20], which adds ℓ_i ($i \in [1..n]$) to I whenever ℓ_i is required to potentially reach the bound b, i.e., whenever

$$\sum_{j \in [1..n],\, j \neq i} w_j \cdot I^{\uparrow}(\ell_j) < b \tag{8}$$

where J is the earliest prefix of I satisfying (8). In this case, $\mathit{reason}(\ell_i) := \{\ell_i\} \cup (\mathit{lits}(\sigma) \cap \overline{J})$, i.e., the false literals of σ are those forcing ℓ_i to be true. In the special case of an AMO constraint (7), i.e., $\sigma = \text{AMO}\{\ell_1, \ldots, \ell_n\}$, the literal $\overline{\ell_i}$ ($i \in [1..n]$) is added to I whenever there exists $\ell_j \in I$ with $j \neq i$, with $\mathit{reason}(\overline{\ell_i}) := \{\overline{\ell_i}, \overline{\ell_j}\}$.

Example 2.6 (Continuing Example 2.1)**.** If I is empty, no literal can be inferred from σ_1, σ_2 and σ_3. If I is $[\overline{a}]$, also in this case no literal is inferred from any constraints. If I is extended with $\overline{b}$ then the application of (8) to the literals of σ_3 gives

$$2 \cdot [\overline{a}, \overline{b}]^{\uparrow}(c) = 2 \cdot 1 = 2 < 4$$
$$2 \cdot [\overline{a}, \overline{b}]^{\uparrow}(d) = 2 \cdot 1 = 2 < 4$$

Hence, c and d are inferred with $\mathit{reason}(c) = \{c, a, b\}$ and $\mathit{reason}(d) = \{d, a, b\}$. Note that, once $I = [\overline{a}, \overline{b}, c, d]$, the application of (8) to σ_2 gives

$$1 \cdot [\overline{a}, \overline{b}, c, d]^{\uparrow}(\overline{c}) = 1 \cdot 0 = 0 < 1$$
$$1 \cdot [\overline{a}, \overline{b}, c, d]^{\uparrow}(\overline{d}) = 1 \cdot 0 = 0 < 1$$

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

Optimal Stable Model Search. Two main algorithmic strategies are employed to search for optimal stable models in ASP. *Linear SAT-UNSAT (LSU)* search [21] starts from a stable model and iteratively strengthens the objective bound, adding a constraint that rules out any solution with objective value no greater than the current best, until no further improvement is possible and optimality is certified. *Unsat-core based* search [22], instead, starts without requiring a feasible solution and iteratively relaxes the problem, raising a lower bound on the objective value by analyzing unsatisfiable cores until a satisfying solution is found at the current bound.

In the following, we focus on LSU strategy.

3. Amosum

In this section, we briefly recall how SUM constraints of the form (2) can be replaced by the more general AMOSUM constraints, introduced in [13]. An AMOSUM constraint combines a SUM constraint with a set of AMO constraints into a convenient syntax that also results in a more effective propagator.

3.1. Syntax and Semantics

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

$$\text{AMOSUM}\{w_1 : \ell_1\ [s_1];\ \cdots;\ w_n : \ell_n\ [s_n]\} \geq b \tag{9}$$

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 $b, w_1, \ldots, w_n, s_1, \ldots, s_n$ are nonnegative integers.

The notation introduced in Section 2.1 is extended as follows:

- $parts_\sigma := \{s_1, \ldots, s_n\}$ denotes the set of parts, where the literals of σ are partitioned into at most n parts $s_1, \ldots, s_n$;
- $part_\sigma(\ell_i) := s_i$ denotes the part of ℓ_i in σ, for all $i \in [1..n]$, with the symmetry condition $part_\sigma(\overline{\ell_i}) = part_\sigma(\ell_i)$;
- $lits_\sigma|_s$ denotes the set of literals in σ belonging to part s.

Moreover, the membership relation $\in$ is defined as $(w_i : \ell_i\ [s_i]) \in \sigma$ for all $i \in [1..n]$. Regarding the satisfaction relation, for σ of the form (9), $I \models \sigma$ 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 satisfaction condition of (2) is extended by enforcing an AMO constraint on each part of σ.

Example 3.1 (Continuing Example 2.1)**.** Π_{run} is rewritten by replacing σ_1, σ_2 and σ_3 with

$$\sigma_4 : \quad \text{AMOSUM}\{1 : a\ [1];\ 2 : b\ [1];\ 2 : c[2]\ \ 2 : d\ [2]\} \geq 4$$

Note that $parts_{\sigma_4} = \{1, 2\}$, $part_{\sigma_4}(a) = part_{\sigma_4}(b) = 1$, $part_{\sigma_4}(c) = part_{\sigma_4}(d) = 2$, $lits_{\sigma_4}|_1 = \{a, b\}$, and $lits_{\sigma_4}|_2 = \{c, d\}$.

Given an interpretation I and a constraint σ, its *maximum possible sum* $mps_\sigma(I)$ is the maximum possible value of

$$\sum_{i=1}^{n} w_i \cdot I'(\ell_i)$$

such that

$$\sum_{\ell \in lits_\sigma(s)} I'(\ell) \leq 1 \quad \forall s \in parts_\sigma$$

and I' is a total interpretation such that $I' \supseteq I$. In other words, it is the maximum possible sum reachable starting from I that satisfies all the at most one constraints for σ. Given a literal ℓ, the maximum possible sum reachable assuming ℓ true is

$$mps_\sigma(I, \ell) := \sum_{s \in parts} mwh_\sigma(I \dot\cup \{\ell\}, s) \tag{10}$$

where

$$mwh_\sigma(I, s) := \begin{cases} \ell' & \text{if } lits_\sigma|_s \cap I = \{\ell'\} \\ \max\{w \cdot I^{\uparrow}(\ell') \mid (w : \ell'\ [s]) \in \sigma\} & \text{otherwise.} \end{cases}$$

Above, the intuitive reading of the second case is "the maximum weight that part s can contribute to the overall sum." Note that $mps_\sigma(I)$ can be equivalently defined as follows:

$$mps_\sigma(I) := \sum_{s \in parts} mwh_\sigma(I, s) \tag{11}$$

3.2. Propagation

A constraint σ of the form (9) is associated with 2 different inference rules; for each inference rule we assume to start with an interpretation I that is consistent with the *at most one* constraint, that is, $\sum_{\ell \in lits_\sigma(s)} I(\ell) \leq 1$ for all $s \in parts_\sigma$.

- First of all, the AMO inference given by the parts of σ: (*AMO Rule (AMR)*) 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 inferece rule is based on the sum inferece rule but with more capability, given by the information of the AMO constraint; it is named *Amosum Rule (ASR)*. With ASR a literal ℓ can be inferred to true when assuming it is false would lead the *max possible sum* to be under the bound. Formally, let σ be a constraint of the form (9), and let I be an interpretation such that $\sum_{\ell \in lits_\sigma(s)} I(\ell) \leq 1$ for all $s \in parts_\sigma$. A literal is inferred true if it is required in order to reach the bound by the following inference rule: (ASR) An undefined literal ℓ is added to I if $\ell \in lits_\sigma$ and J is the first prefix of I such that

$$mps_\sigma(J, \overline{\ell}) < bnd_\sigma\,. \tag{12}$$

  In this case, $reason(\ell)$ is

$$\{\ell\} \cup \bigcup_{s \in parts_\sigma} rsn_\sigma(J, \ell, s) \tag{13}$$

  where $rsn_\sigma(J, \ell, s) := \{\overline{\ell'}\}$ if $lits_\sigma|_s \cap J = \{\ell'\}$ (i.e., a true literal in $lits_\sigma|_s$), and $\{\ell' \in lits_\sigma|_s \cap \overline{J} \mid wh(\ell) \geq mwh'(J \dot\cup \{\overline{\ell}\}, s)\}$ otherwise (i.e., the false literals in the part of ℓ that could have increased the overall sum).

Example 3.2 (Continuing Example 3.1)**.** Let us consider the constraint σ_4 and let $I = [\,]$ be the current assignment. The inequality (12) of rule ASR is able to detect that b is needed to reach the bound and $reason(b) = \{b\}$. Furthermore inequality (12) of rule ASR is also able to detect that $\overline{a}$ is needed, given that $mps([\,], a) = 3 < 4$, with reason $reason(\overline{a}) = \{\overline{a}\}$. Please note that after b is added to I also AMR rule is able to infer $\overline{a}$.

4. Amomaximize

In this section we explain the new construct, referred to as AMOMAXIMIZE, and how an optimal stable model w.r.t. a maximization statement is computed.

4.1. Syntax and Semantics

Let's assume to have a program Π with a maximization statement of the form (3) and literals in *lits* are split in partitions, such that literals in the same part are subject to an at most one constraint. We can compactly write this maximization statement using an *amomaximize statement*. An *amomaximize statement* μ has the form

$$\text{AMOMAXIMIZE}\{w_1 : \ell_1\ [s_1];\ \cdots;\ w_n : \ell_n\ [s_n]\}. \tag{14}$$

where $m \geq 0, \ell_1, \ldots, \ell_n$ are distinct literals such that $\ell_i \neq \overline{\ell_j}$ for all $1 \leq i \leq j \leq n$, and $b, w_1, \ldots, w_n, s_1, \ldots, s_n$ are nonnegative integers. Each s_i denotes the *part* of literal ℓ_i, and literals sharing the same part as subject to an at-most-one constraint. The syntax mirrors that of AMOSUM (Section 3), with the only difference being that no explicit bound is specified. Formally, we have:

- $parts_\mu = \{s_1, \ldots, s_n\}$: the set of parts;
- $part_\mu(\ell_i) := s_i$ for all $i \in \{1, \ldots, n\}$, with symmetry $part_\mu(\ell_i) = part_\mu(\overline{\ell_i})$;
- $lits_\mu|_s = \{\ell \mid part_\mu(\ell) = s\}$: the set of literals in part s.

Note that μ is a special case of (3), so the objective function *obj* from Section 2 applies directly to compute the objective value of any interpretation.

Algorithm 1: Amomaximize - Linear SAT-UNSAT Optimization

Input: Program Π, amomaximize statement μ
Output: An optimal stable model, or $\bot$ if Π is unsatisfiable

```
1 M ← Solve(Π);
2 if M = ⊥ then return ⊥ ;
3 while M ≠ ⊥ do
4     O ← M;
5     b ← obj(M, μ) + 1;
6     σ ← const(μ, b);
7     M ← Solve(Π ∪ {σ});
8 return O;
```

From Amomaximize to Amosum. A central operation in our approach is converting an AMOMAXIMIZE statement into an AMOSUM constraint given a target bound. This is defined as follows. Given an amomaximize statement μ of the form (14) and an integer $b \geq 0$, we define:

$$const(\mu, b) \;=\; \text{AMOSUM}\{w_1 : \ell_1\ [s_1];\ \cdots\ ;\ w_n : \ell_n\ [s_n]\} \geq b, \tag{15}$$

where $parts_\mu = parts_\sigma$ and $part_\mu = part_\sigma$. Optimality check of a candidate model M can then be checked by solving $\Pi' = \Pi \cup const(\mu,\ obj(M) + 1)$: if $SM(\Pi') \neq \emptyset$, a strictly better model than M exists and M is not optimal. Otherwise, M is optimal. While feasibility check can be done using a classical sum constraint of the form (2), using $const(\mu, b)$ is strictly more efficient. The presence of the amosum constraint gives the possibility to use the efficient propagator defined in Section 3. This propagator has been showed in [13] to be able to effectively prune the search space and we would like to adopt it to get a better lower bound in a smaller time-slot.

Example 4.1 (Continuing Example 2.3)**.** Π_{runmax} is rewritten by replacing σ_1, σ_2 and μ_1 with

$$\mu_2 :\quad \text{AMOMAXIMIZE}\{1 : a\ [1];\ 2 : b\ [1];\ 2 : c[2]\ \ 2 : d\ [2]\}$$

Note that $parts_{\mu_2 qsa} = \{1, 2\}$, $part_{\mu_2}(a) = part_{\mu_2}(b) = 1$, $part_{\mu_2}(c) = part_{\mu_2}(d) = 2$, $lits_{\mu_2}|_1 = \{a, b\}$, and $lits_{\mu_2}|_2 = \{c, d\}$.

4.2. Implementation of AMOMAXIMIZE statement

To find an optimal stable model $M \in OPT(\Pi, \mu)$ we adopt a *Linear SAT-UNSAT* (LSU) search strategy. The idea is to iteratively search for models with strictly increasing objective values: each iteration raises the lower bound by one until the solver returns UNSAT (there are no stable models for the given program), certifying that the last satisfiable bound is optimal. Algorithm 1 shows this procedure.

In particular, it defines how an optimal stable model is found leveraging an AMOSUM constraint. At line 1, a first call to the solver is made, with no constraint. If Π does not admit any stable model, i.e., $\bot$ is returned at line 2. Otherwise, a stable model M is found, and it represents a valid solution, which is not guaranteed to be optimal. Then, the algorithm iterates for finding better solution. In particular, the algorithm takes advantage of the set O to store the best solution found so far 4. Then, the objective value of O is computed and its value, incremented by 1, is stored in b (line 5). Subsequently, the algorithm converts the maximization statement μ using the *const* function using b as bound, thus creating the constraint σ (line 6). Finally, the solver is invoked on input program Π extended with the constraint σ(line 7), a new stable model M is computed. If the program does not admit any new stable model, the algorithm terminates returning O.

Indeed, since at each iteration the algorithm tightens the lower bound on the minimum achievable quantity, when $M = \bot$ we can conclude that the last computed bound is in fact optimal.

Correctness and Termination. We now discuss that Algorithm 1 is both correct and terminating.

Proposition 4.1. *For a program Π, Algorithm 1 terminates and satisfies the following*

1. *If $SM(\Pi) = \emptyset$, it returns $\bot$.*
2. *Otherwise, it returns $O \in OPT(\Pi, \mu)$.*

Proof. Termination. At each iteration, b strictly increase by 1. Since the objective value is bounded by *obj* is bounded above by $\sum_{i\in[1,\dots,n]} w_i$, the loop can execute at most $\sum_{i\in[1,\dots,n]} w_i$ before returning UNSAT.
Correctness. Let O be the stable model returned by 1. It means that the last call gave as output $M = \bot$ with input $\Pi \cup const(\mu, O + 1)$. Given that $M = \bot$ it means that for all $M' \in SM(\Pi)$ then $obj(M', \mu) \leq obj(O, \mu)$. Hence, $O \in OSM(\Pi, \mu)$. □

Example 4.2 (Continuing Example 4.1)**.** Let $I = \{a, c\}$, then $I \in SM(\Pi_{runmax})$ and $obj(I) = 3$. Define $\Pi_4 = \Pi_{runmax} \setminus \{\mu_2\} \cup \{const(\mu_2, 4)\}$ and consider an initially empty interpretation $I' = []$. As shown in Example 3.1, I' is extended with $[\overline{a}, b]$. Assuming the branching literal is c, the AMR rule infers $\overline{d}$, yielding a final interpretation I' with $obj(I') = 4$, which witnesses the existence of a better stable model.

We then construct $\Pi_5 = \Pi_{runmax} \setminus \{\mu_2\} \cup \{const(\mu_2, 5)\}$. Applying ASR to $const(\mu_2, 5)$ with an empty starting interpretation I'' infers the literals $[a, b, c, d, \overline{a}, \overline{b}, \overline{c}, \overline{d}]$, each with a reason equal to the singleton of the literal itself (i.e., each literal is unconditionally true). Applying resolution to $reason(a)$ and $reason(\overline{a})$ yields an empty clause, proving that no stable model of Π_5 exists. Therefore, $I' \in OSM(\Pi_{runmax})$.

Finally, we note that, from a semantic perspective, the AMOMAXIMIZE statement does not increase the expressive power of the language.

Proposition 4.2. *Let μ be an AMOMAXIMIZE statement of the form*

$$\textsc{amomaximize}\{w_1 : \ell_1[s_1];\ \dots;\ w_n : \ell_n[s_n]\}. \tag{16}$$

Then μ can be equivalently expressed using a MAXIMIZE statement in combination with a standard AMO constraints as follows:

$$\begin{aligned} &\textsc{maximize}\{w_1 : \ell_1;\ \cdots;\ w_n : \ell_n\} \\ &\qquad\qquad \textsc{amo}\{lits_\mu|_{s_i}\} \qquad \forall i \in [1..n] \end{aligned}$$

Therefore, the advantage of extending the language with the AMOMAXIMIZE statement lies in the ability to employ the AMOSUM propagator, which is particularly beneficial in cases where proving optimality is difficult.

5. Experiments

This section presents an empirical evaluation of the proposed propagator against the state-of-the-art ASP solver CLINGO [11] (version 5.8.0) on a set of benchmark instances. We begin by describing the selected benchmarks and the experimental setup, including the systems and configurations used. We then analyze and discuss the results obtained.

The solver introduced in Section 4 is implemented on top of CLINGO (version 5.8.0) via its C interface, and is hereafter referred to as AMOMAXIMIZE.

All experiments were conducted on a machine equipped with an Intel® Xeon® Gold 5118 CPU (2.30 GHz), 512 GB of RAM, running Ubuntu 20.04.2 LTS (Linux kernel 5.4.0-137, x86_64). Each solver run was subject to a time limit of 20 minutes and a memory limit of 8 GB.

	AMOMAXIMIZE			CLINGO		
Benchmark	O	Score	Win	O	Score	Win
GC	0	**1.0**	**289**	0	0.885	11
K	0	0.912	21	0	**0.996**	**79**
TSP	**5**	**0.995**	**82**	2	0.942	20
WOD	**6**	0.703	53	0	**0.969**	**92**
Total	**11**	**0.935**	**445**	2	0.927	202

Table 1
Comparison of AMOMAXIMIZE and CLINGO across all benchmarks in terms of *Optimum Found* (*O*), *Score*, and *Win*. Bold values indicate the best result for each metric and benchmark.

5.1. Benchmarks

The experimental analysis is based on a set of benchmarks partly derived from those used by [13] and partly newly introduced to broaden the scope of the evaluation. The newly introduced benchmarks, namely *Traveling Salesman Problem* and *Weighted Operator Dispatch*, are either directly based on or inspired by real-world problems, allowing us to assess the performance of the proposed propagator in more practical settings.

Weighted Graph Coloring (WGC). This benchmark is based on the graph coloring problem. Instances are generated from those employed in the ASP competition [23], with the colors red, green, blue, yellow, and cyan associated with the weights 2, 4, 8, 16, and 64, respectively. The objective is to maximize the sum of weighted nodes. Here the *At Most One* constraint is derived from the fact that for each node n we must choose at most one color.

Knapsack (K). This benchmark is based on a variant of the classical knapsack problem. A set of item types is given, each associated with a specific weight and value. Given a knapsack with a fixed capacity, the task is to maximize the total value of the selected items such that their combined weight does not exceed the capacity. Instances are randomly generated as follows. The number of item types, denoted by n, ranges from 10 to 505 in increments of 5, and the maximum number of selectable items per type is fixed to $k = 20$. We refer to [13] for a more detailed description of the instances.

Traveling Salesman Problem (TSP). Given a weighted graph representing a map, the task is to find a tour visiting each city (node) exactly once while maximizing the total revenue collected along the traversed edges. The *At Most One* constraint arises from the fact that for each city, at most one successor city can be selected. Instances are randomly generated with edge revenues drawn uniformly from $[1, 100]$ and a map density (i.e., the probability that an edge exists between any two cities) of 0.5.

Weighted Operator Dispatch (WOD). Given a set of workers and tasks, where each worker has a maximum workload and varying efficiency per task, and tasks may depend on one another, the goal is to assign workers to tasks across discrete time slots, respecting capacity and precedence constraints, so as to maximize the total efficiency of the assignment. The *At Most One* constraint arises from the requirement that each worker can be assigned at most one task per time slot. Instances are randomly generated according to the following parameters:

- **Number of units** (nu): the number of available workers, sampled uniformly from $[50, 500]$.
- **Number of tasks** (nt): the number of tasks, sampled uniformly from $[nu \cdot 0.2,\ nu \cdot 1.2]$.
- **Number of slots** (ns): the number of time slots, sampled uniformly from $[5, 100]$.

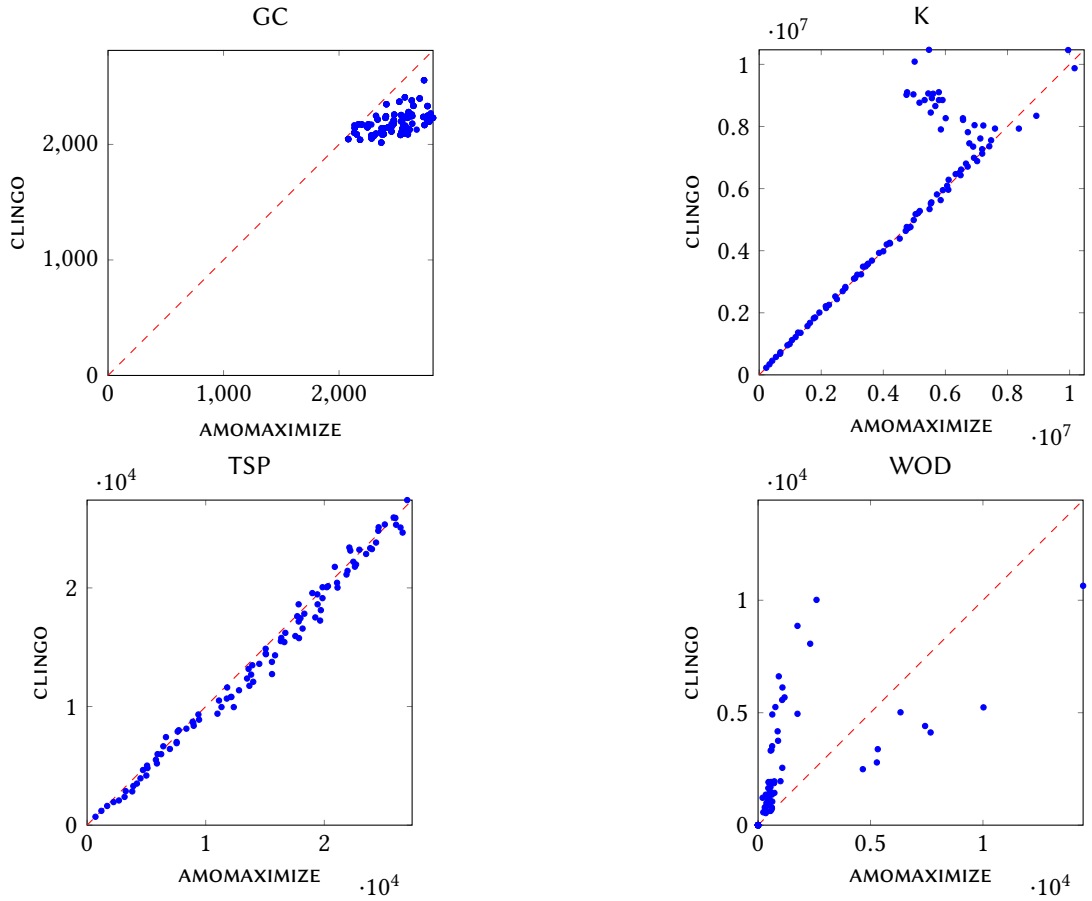

Figure 1: Per-benchmark scatter plots of the *score* metric. Each point represents an instance; the x-axis and y-axis reports the value of the objective of AMOMAXIMIZE and CLINGO, respectively. Points below the diagonal (red dashed line) indicate instances where AMOMAXIMIZE outperforms CLINGO, and vice versa.

5.2. Results

Results are reported in Table 1. For each benchmark, the column O indicates the number of instances for which the optimal solution was found, *Score* denotes the average score over all instances, and *Win* represents the number of instances in which a solver achieved the best solution among all considered solvers. More precisely, for each solver s and instance i, the *Score* metric is defined as $Score_i^s = o_i^s / m_i$, where o_i^s is the objective value returned by solver s on instance i, and m_i is the best objective value obtained for instance i across all solvers. In addition to the metrics reported in Table 1, we also computed the number of memory-outs, the number of timeouts, and the mean PAR1 score. The PAR1 score for a given instance is equal to the actual solving time if the solver terminates within the resource bounds, and is set to the time limit (1200 seconds) otherwise.

Optimum Found. AMOMAXIMIZE found significantly more optimal solutions (11) than CLINGO (2), and this holds consistently across all benchmarks where at least one solver proved optimality. Notably, even in the WOD benchmark, where CLINGO obtains better results in terms of *Win* and *Score*, AMOMAXIMIZE was able to prove optimality for 6 instances, while CLINGO did not find any optimal solution.

Par1, Memory and Time out. For the GC and K benchmarks, both solvers always reach the timeout. In contrast, AMOMAXIMIZE obtains better results on TSP and WOD, with mean PAR1 scores of 1144.92 and 1156.46, compared to 1177.73 and 1200 for CLINGO, respectively. Concerning memory usage on the WOD benchmark, we observe that AMOMAXIMIZE exceeds the memory bound on 85 instances, compared to 83 for CLINGO, suggesting that the AMOSUM propagator introduces memory overhead on

some instances, potentially reducing the number of optimal solutions found.

Win. AMOMAXIMIZE performed particularly well on the GC and TSP benchmarks, achieving 289 and 82 wins, respectively, compared to only 11 and 20 for CLINGO. Conversely, CLINGO outperformed AMOMAXIMIZE on the K and WOD benchmarks, recording 79 and 92 wins, respectively, against 21 and 53 for AMOMAXIMIZE.

Score. AMOMAXIMIZE performs better than CLINGO on the GC and TSP benchmarks, achieving scores of 1.0 vs. 0.88 and 0.99 vs. 0.94, respectively. The results are again reversed on the remaining two benchmarks; notably, AMOMAXIMIZE performs significantly worse on K and WOD. To gain deeper insight into the score distributions, we provide per-benchmark scatter plots in Figure 1. Each point corresponds to a single instance. The x- and y-axes report the objective values obtained by the corresponding solvers on that instance, where the x-axis refers to AMOMAXIMIZE, and the y-axis refers to CLINGO. The red dashed diagonal represents the line of equal performance, points below the line indicate instances where AMOMAXIMIZE obtains a better objective value than CLINGO, whereas points above the line favor CLINGO. In the GC benchmark, the vast majority of points lie below the diagonal, confirming the effectiveness of AMOMAXIMIZE. Interestingly, CLINGO consistently approaches the solution found by AMOMAXIMIZE but rarely matches it, suggesting that improving upon a given solution is inherently difficult in this benchmark, even at relatively low objective values. The TSP benchmark exhibits a similar pattern, with most instances falling below the diagonal. The key difference with respect to GC is that solutions are more spread across the objective value range rather than clustered around a specific value. In the K benchmark, a large portion of instances lie on the diagonal, indicating that both solvers achieve identical scores. However, as the objective value grows, AMOMAXIMIZE progressively underperforms CLINGO, suggesting that the propagator introduces overhead without yielding sufficient pruning benefit on larger instances. Finally, WOD is the most scattered benchmark. Instances cluster near low objective values, and the results are mixed across the range. Notably, given the high magnitude of the objective values, instances where AMOMAXIMIZE achieves a low score contribute disproportionately to the mean, penalizing the overall score despite competitive performance on many instances.

The observed performance gap can be explained by the structural properties of the benchmarks. In GC and TSP, optimality is relatively easy to certify: in GC, no improved coloring may exist beyond a certain bound, and in TSP, no better tour may be reachable from a given solution. As a result, solutions tend to cluster near the optimum, which amplifies the effectiveness of the propagator. In contrast, the K and WOD benchmarks do not share this property. In K, incrementally better knapsack solutions can often be found, which delays the hard unsatisfiability calls and reduces the impact of the propagator.

These observations support the hypothesis that AMOMAXIMIZE is most effective when candidate solutions are tightly clustered near the optimum, as this maximizes the impact of near-optimal propagation. This is further confirmed by the fact that AMOMAXIMIZE consistently proves more optimal solutions than CLINGO, even on benchmarks where its overall score is lower.

6. Related Work

AMOMAXIMIZE statements, as introduced in this work, extend the propositional language of Answer Set Programming (ASP). Their applicability is, however, not limited to ASP: they can in principle be integrated into any logic-based formalism that supports propositional reasoning, including *Maximum Satisfiability (MaxSAT)* [24]. Indeed, several algorithmic techniques have been transferred between MaxSAT and ASP, giving rise to a rich cross-fertilization between the two areas [25, 26]. This generality opens the door to broader adoption of AMOMAXIMIZE statements across a variety of automated reasoning paradigms. Two main strategies have emerged for solving optimization problems in such settings, namely *Linear SAT-UNSAT* and *core-guided* search.

6.1. Linear SAT-UNSAT

Linear SAT-UNSAT (LSU) is one of the most natural and effective strategies for solving optimization problems in ASP. Starting from an initial feasible solution, the approach iteratively checks whether a strictly better solution exists by asserting a constraint that rules out any solution no better than the current one. If such a check yields UNSAT, the current solution is certified as optimal. As discussed in Section 2.3, this corresponds to a sequence of satisfiability checks, each tightening the bound by one unit, hence the term *linear*. LSU is implemented in state-of-the-art ASP solvers such as CLINGO [17, 27] and WASP [28]. The approach is well-suited to settings where feasible solutions are easy to find and improvements are expected to be frequent. Our approach falls within the LSU paradigm. Specifically, we extend the standard LSU strategy by exploiting AMOMAXIMIZE statements to propagate tighter bounds during search, reducing the number of iterations required to reach optimality. We compare directly against the LSU implementation of CLINGO to isolate the contribution of AMOMAXIMIZE-aware propagation over the baseline technique.

6.2. Core-Guided Search

Core-guided approaches take a dual perspective: rather than starting from a feasible solution and improving it, they begin from an infeasible bound and progressively relax it. At each iteration, the solver identifies an *unsatisfiable core* a minimal subset of weighted literals that cannot be simultaneously satisfied at the current bound and uses it to derive a tighter lower bound on the optimal objective value. This process continues until the lower bound meets a feasible solution, at which point optimality is certified. Core-guided methods have proven particularly effective in pseudo-Boolean optimization and Maximum Satisfiability (MaxSAT) [29]. In the ASP setting, many core-guided strategies have been proposed (e.g. [22, 25, 30, 31]). While our work adopts an LSU strategy, the AMOMAXIMIZE construct could in principle be integrated into core-guided frameworks as well.

7. Conclusions and Future Work

In this work, we presented the AMOMAXIMIZE statement, a new maximization construct for ASP that enables efficient solving of optimization problems in which an *at-most-one* constraint is embedded in the objective function. While the construct does not increase the expressive power of the language, it provides the user with additional means to guide *search-space pruning* in ASP solvers. By adopting AMOMAXIMIZE in place of the classic MAXIMIZE statement, the solver gains richer information about the objective function, improving reasoning and the ability to determine whether a better solution exists.

AMOMAXIMIZE is built upon the AMOSUM propagator of Alviano et al. 2024 which ensures that a weighted sum exceeds a given bound by exploiting the combined information from both the SUM and AMO constraints to strengthen propagation. AMOSUM has shown promising results in settings where the maximum achievable sum is close to the bound, i.e., when satisfying the constraint is hard. We reduce the AMOMAXIMIZE statement to an AMOSUM constraint with a dynamic bound, iteratively demanding a strictly better bound until no such bound exists, thus identifying the optimum. The empirical evaluation confirms the potential of AMOMAXIMIZE to deliver performance gains in specific scenarios.

As future work, a possibility is to investigate alternative optimization strategies. Indeed, this work adopts a LSU approach, but a *core-based* approach could be also explored, where unsatisfiable cores explicitly account for AMO constraints, potentially improving search-space pruning further. Another direction is to relax the strict linearity of the LSU scheme by adopting a progressive bound-increase strategy (e.g., exponential steps), which would trade the guarantee that the first UNSAT call proves optimality for the ability to skip useless SAT calls.

Finally, we mention that all code, encodings, and data used in our experiments are publicly available at https://github.com/instafiore/AMOSUM/tree/amomaximize-cilc26.

Acknowledgments

This work was supported by the Italian Ministry of Health (MSAL) under POS projects CAL.HUB.RIA (CUP H53C22000800006) and RADIOAMICA (CUP H53C22000650006); by the Italian Ministry of Enterprises and Made in Italy under project STROKE 5.0 (CUP B29J23000430005); under PN RIC project ASVIN "Assistente Virtuale Intelligente di Negozio" (CUP B29J24000200005); by Regione Calabria under project IOS4OS "Intelligent Optimization System for Optimization Solutions" (CUP J89I24002920005); under project NextGenGuides "Innovazione Tecnologica per le Guide Turistiche del Futuro tramite l'ausilio di tecnologie innovative AI e sistemi di geolocalizzazione avanzata" (CUP J39I24002040005); under project MOZART "Modelli e tecniche di mOdernizzazione di applicaZioni e data silos a supporto di clienti esterni, field force e impiegati di backoffice basati su AI GeneRativa e apprendimento auTomatico" (CUP J49I24001740005); by the LAIA lab (part of the SILA labs). Mario Alviano is a member of Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

Declaration on Generative AI

During the preparation of this work, the author(s) used ChatGPT, X-GPT-4 and Gramby in order to: Grammar and spelling check. After using these tools/services, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] V. Marek, M. Truszczyński, Stable models and an alternative logic programming paradigm, in: The Logic Programming Paradigm: a 25-year Perspective, 1999, pp. 375–398. doi:10.1007/978-3-642-60085-2_17.

[2] I. Niemelä, Logic programming with stable model semantics as a constraint programming paradigm, Annals of Mathematics and Artificial Intelligence 25 (1999) 241–273. doi:10.1023/A:1018930122475.

[3] A. A. Falkner, G. Friedrich, K. Schekotihin, R. Taupe, E. C. Teppan, Industrial applications of answer set programming, Künstliche Intell. 32 (2018) 165–176. doi:10.1007/S13218-018-0548-6.

[4] E. Erdem, V. Patoglu, Applications of ASP in robotics, Künstliche Intell. 32 (2018) 143–149. doi:10.1007/S13218-018-0544-X.

[5] E. Erdem, M. Gelfond, N. Leone, Applications of answer set programming, AI Mag. 37 (2016) 53–68. doi:10.1609/AIMAG.V37I3.2678.

[6] P. Cappanera, S. Caruso, C. Dodaro, G. Galatà, M. Gavanelli, M. Maratea, C. Marte, M. Mochi, M. Nonato, M. Roma, Recent answer set programming applications to scheduling problems in digital health, in: Proc. of AI4CC-IPS-RCRA-SPIRIT Workshops, volume 3883 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2024.

[7] V. Lifschitz, What is answer set programming?, in: Proc. of AAAI, AAAI Press, 2008, pp. 1594–1597.

[8] V. Lifschitz, Answer sets and the language of answer set programming, AI Mag. 37 (2016) 7–12. doi:10.1609/AIMAG.V37I3.2670.

[9] G. Brewka, T. Eiter, M. Truszczynski, Answer set programming at a glance, Commun. ACM 54 (2011) 92–103. doi:10.1145/2043174.2043195.

[10] T. Janhunen, I. Niemelä, The answer set programming paradigm, AI Mag. 37 (2016) 13–24. doi:10.1609/AIMAG.V37I3.2671.

[11] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, P. Wanko, Theory solving made easy with clingo 5, in: Proc. of ICLP TCs, volume 52 of *OASIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:15. doi:10.4230/OASICS.ICLP.2016.2.

[12] M. Alviano, C. Dodaro, N. Leone, F. Ricca, Advances in WASP, in: Proc. of LPNMR, volume 9345 of *LNCS*, Springer, 2015, pp. 40–54. doi:10.1007/978-3-319-23264-5_5.

[13] M. Alviano, C. Dodaro, S. Fiorentino, M. Maratea, Amo-aware aggregates in answer set programming, in: Proc. of IJCAI, ijcai.org, 2024, pp. 3215–3223.
[14] W. Faber, G. Pfeifer, N. Leone, Semantics and complexity of recursive aggregates in answer set programming, Artif. Intell. 175 (2011) 278–298. doi:10.1016/j.artint.2010.04.002.
[15] M. Davis, H. Putnam, A computing procedure for quantification theory, J. ACM 7 (1960) 201–215. doi:10.1145/321033.321034.
[16] J. A. Robinson, A machine-oriented logic based on the resolution principle, J. ACM 12 (1965) 23–41. doi:10.1145/321250.321253.
[17] M. Gebser, B. Kaufmann, T. Schaub, Conflict-driven answer set solving: From theory to practice, Artif. Intell. 187 (2012) 52–89. doi:10.1016/J.ARTINT.2012.04.001.
[18] C. Dodaro, Design and implementation of modern CDCL ASP solvers, Intelligenza Artificiale 18 (2024) 239–259. doi:10.3233/IA-240019.
[19] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, On the implementation of weight constraint rules in conflict-driven ASP solvers, in: Proc. of ICLP, volume 5649 of *LNCS*, Springer, 2009, pp. 250–264.
[20] W. Faber, N. Leone, M. Maratea, F. Ricca, Look-back techniques for ASP programs with aggregates, Fundam. Informaticae 107 (2011) 379–413.
[21] P. Simons, I. Niemelä, T. Soininen, Extending and implementing the stable model semantics, Artif. Intell. 138 (2002) 181–234. doi:10.1016/S0004-3702(02)00187-X.
[22] B. Andres, B. Kaufmann, O. Matheis, T. Schaub, Unsatisfiability-based optimization in clasp, in: Proc. of ICLP TCs, volume 17 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012, pp. 211–221. doi:10.4230/LIPICS.ICLP.2012.211.
[23] F. Calimeri, M. Gebser, M. Maratea, F. Ricca, Design and results of the fifth answer set programming competition, Artif. Intell. 231 (2016) 151–181. doi:10.1016/J.ARTINT.2015.09.008.
[24] C. M. Li, F. Manyà, Maxsat, hard and soft constraints, in: A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, IOS Press, 2021, pp. 903–927. doi:10.3233/FAIA201007.
[25] M. Alviano, C. Dodaro, J. Marques-Silva, F. Ricca, Optimum stable model search: algorithms and implementation, J. Log. Comput. 30 (2020) 863–897. doi:10.1093/LOGCOM/EXV061.
[26] A. Morgado, C. Dodaro, J. Marques-Silva, Core-guided maxsat with soft cardinality constraints, in: Proc. of CP, volume 8656 of *LNCS*, Springer, 2014, pp. 564–573. doi:10.1007/978-3-319-10428-7_41.
[27] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot ASP solving with clingo, Theory Pract. Log. Program. 19 (2019) 27–82.
[28] M. Alviano, C. Dodaro, M. Maratea, Shared aggregate sets in answer set programming, Theory Pract. Log. Program. 18 (2018) 301–318.
[29] L. Zhang, S. Malik, Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications, in: Proc. of DATE, IEEE Computer Society, 2003, pp. 10880–10885. doi:10.1109/DATE.2003.10014.
[30] M. Alviano, C. Dodaro, F. Ricca, A maxsat algorithm using cardinality constraints of bounded size, in: Q. Yang, M. J. Wooldridge (Eds.), Proc. of IJCAI, AAAI Press, 2015, pp. 2677–2683.
[31] M. Alviano, C. Dodaro, Unsatisfiable core analysis and aggregates for optimum stable model search, Fundam. Informaticae 176 (2020) 271–297. doi:10.3233/FI-2020-1974.