Università della Calabria

Dipartimento di Matematica e Informatica

Corso di Laurea Magistrale in

Artificial Intelligence & Computer Science

Tesi di Laurea

Design and Implementation of an AMO-SUM aggregate for ASP

Relatori:

Prof. Carmine Dodaro

Prof. Thomas Eiter

Dr. Tobias Geibinger

Candidato:

Salvatore Fiorentino

Matricola 242706

Anno Accademico 2023/2024

Inserire dedica e ringraziamenti...

Contents

Chapter 1

Introduction

Answer Set Programming (ASP) is a highly used framework for knowledge representation and automated reasoning [1]–[4]. In ASP, combinatorial problems are formulated using logical rules that incorporate various linguistic constructs, which simplify the representation of complex knowledge. In its most basic form, ASP programs consist of normal logic rules, where each rule has a head atom and a body that is a conjunction of literals. Often, normal programs are extended by incorporating aggregates, as discussed by *Bartholomew et al. (2011)* [5], *Faber et al. (2011b)* [6], *Ferraris (2011)* [7], *Gelfond and Zhang (2014)* [8], *Liu et al. (2010)* [9], *and Simons et al. (2002)* [9]. Specifically, *SUM aggregates* are used in rule bodies, where literals are assigned weights, and the sum of the weights of the true literals must satisfy a specified (in)equality. When the head of the rule is false the aggregate becomes a constraint aggregate. Another kind of constraint is the *At Most One* constraint that essentially inhibits truth of pairs of literals in a given set. It is very common that these two constraints (SUM and AMO) are linked together. State of the art solvers treat this case ignoring the correlation between these two constraints. Our work aims to define a new costruct named *AMO-SUM* to efficiently handle this case. This efficiency improvement comes from the fact that we are able to treat the two constraint as a whole constraint.

Nowadays current ASP solver implements a (CDCL) algorithm with propagators, as explained by *Gebser et al. (2012)* [10]. CDCL (Conflict-

Driven Clause Learning) is a contemporary form of non-chronological backtracking that follows the *choose-propagate-learn* pattern, as described by *Marques-Silva et al. (2021)* [11], [12].

This pattern consists of this three phases: **choose** phase, or decision phase, consists in picking a branking literal as true; **propagate** phase derives determinists consequences of the current state; **learn** phase is triggered when a conflict arises and aims at understanding from the conflict to not making the same mistake again. In the propagate phase specific procedures called *propagators* are used to derive such consequences. Propagators are required to explain why a consequence has been derived. This explanation is called reason, it is a set of literals that led to derive that consequence and it is used in the learning phase. When an aggregate is introduced inside the program then a specific propagator is required. Our work provides both a novel propagator for handling the new AMO-SUM construct and an algorithm to minimize the reasons of the aggregate-derived consequences. To provide a more comprehensive understanding of our work the

DA FINIRE

In summary, the contributions of this thesis are the following:

1. We defined a novel propagator for handling AMOSUM constraints (Chapter 3.3)

2. We defined a novel strategy for improving the reasons of the AMOSUM propagator

3. We implemented the novel propagator on the top of the ASP solver WASP

4. We performed an experimental analysis on several benchmarks, showing that ...

Chapter 2

Background

This chapter defines all the background needed to explain our work, trying to guide the reading through a clear and intuitive idea. Initially the syntax and semantics of normal program (section 2.1) is showed, primarly focusing on notions relevant for this thesis, for instance on some extension such as the SUM constraints. Then some specific cases of SUM constraints, the so-called ALO and AMO constraints, is explained in section (2.2). Afterwards, the current State-of-Art ASP solver algorithm to find a stable model is discussed (section 2.3), focusing on the concept of *propagator*.

2.1 Syntax and Semantics

This section proceeds in a bottom-up fashion, introducing the most basic elements to the concept of *program* and *stable model*.

The first element is the set of *atoms*, let $\mathcal{A}$ be such set. $\neg$ is a symbol representing the common negation in logic. A *literal* is an atom with possibly the negation symbol in front of it. For instance $a \in \mathcal{A}$ is a literal (and an atom) and $\neg b$ with $b \in \mathcal{A}$ is a literal (but not an atom); a is said to be a *positive literal* instead b is a negative literal. In a more formal way: given a literal $\ell \in L$, ℓ is positive if $\ell \in \mathcal{A}$ otherwise it is negative. Let L be a set of literals. Given $\ell \in L$ then $\overline{\ell}$ denotes its complement, if ℓ is a positive literal, i.e. $\ell = a \in \mathcal{A}$, then $\overline{\ell} = \neg a$ else when $\ell = \neg a$ (negative literal) with $a \in \mathcal{A}$

then $\overline{\ell} = a$. A set of literal L can be negated, written $\overline{L}$; $\overline{L}$ is equivalent to the set of literals of L where each literal is negated, that is, $\overline{L} = \{\overline{\ell} \mid \ell \in L\}$.

Each atom can be mapped to a truth value (boolean value) by an *intepretation*. An *intepretation* (or *assignment*) I is a set of literals where $I \cap \overline{I} = \emptyset$. If $\mathcal{A} \subseteq (I \cup \overline{I})$ then I is called *total-intepretation*, otherwise it is a *partial-intepretation*. On one hand if $\ell \in I$ then ℓ is *true* under I, on the other hand if $\ell \in \overline{I}$ then it is *false* under I. If either $\ell \notin I$ and $\ell \notin \overline{I}$ then ℓ is said to be *undefined*. Abusing of notation, If $\ell \in I$ let $I^{\top}(\ell) := 1$, 0 otherwise; if $\ell \notin \overline{I}$ let $I^{\neg\perp}(\ell) := 1$, 0 otherwise.

Now the first main brick can be presented: the rule. A rule is a classic implication in propositional logic.

$$r: \quad p \leftarrow \ell_1, \ldots, \ell_n \tag{2.1}$$

where p is an atom and $\ell_1, \ldots, \ell_n$ with $n \geq 0$ are literals. The rule r 2.1 is equivalent to $\ell_1 \wedge \ldots \wedge \ell_n \rightarrow p$. As in propositional logic p and $\ell_1, \ldots, \ell_n$ are named respectively *head* and *body* of the rule. The head of r is defined by the symbol $H(r) := p$ and the body by the set $B(r) := \{\ell_1, \ldots, \ell_n\}$. Each rule has a *positive* and *negative* part, and it is stricly linked with the concept of positive and negative literal. On one hand, the positive body of the rule r, named $B^+(r)$, is the set of positive literals of r, that is, $B^+(r) = \{\ell \mid \{\ell\} \cap \mathcal{A} \neq \emptyset\}$. On the other hand, the negative body of the rule r, named $B^-(r)$, is the set of negative literals of r, namely, $B^-(r) = \{\ell \mid \{\ell\} \cap \mathcal{A} = \emptyset\}$

Now the relation $\models$ (satisfies, or is model of) is inductly defined: let I be an intepretation, if $\ell \in I$ then $I \models \ell$; if $\ell \in \overline{I}$ then $I \models \overline{\ell}$; if $I \models \ell_+$ for every $\ell_+ \in B^+(r)$ and $I \models \ell_-$ for every $\ell_- \in B^-(r)$, then $I \models B(r)$; if whenever $I \models B(r)$ also $I \models H(r)$ then $I \models r$. A (normal) program Π is a set of rules. $M(\Pi) = \{I \mid I \models \Pi\}$ is the set of *models* of Π.

Let's now shift to another concept that allow us to transition from a normal program to an extension of it: the concept of *SUM constraint*. A SUM constraint (or simply constraint) has the following form

$$\textsc{sum}\{w_1 : \ell_1; \ \cdots ; \ w_n : \ell_n\} \geq b \tag{2.2}$$

where $n \geq 0$, $\{\ell_1, \ldots, \ell_n\}$ is a set of literals such that $\ell_i \neq \ell_j$ for all $i, j \in \{1, \ldots, n\}$ such that $i \neq j$ and $\{b, w_1, \ldots, w_n\}$ is a set of naturals numbers. Let σ be a constraint of the form 2.2 then bnd_σ represents the *bound* of the constraint σ; $w_1, \ldots, w_n$ are the weights of each literal in σ; $lits_\sigma$ is the set of literals of σ. Let's specify the relation $\in$ as $(w_i, \ell_i) \in \sigma$ to be read as $(w_i : \ell_i)$ is an element in (the aggregation set of) σ; The function $wh_\sigma(\ell_i) = w_i$, namely, this is a function that maps every literal of σ to its weight. Intuitively, the constraint is satisfied w.r.t. an interpretation I if summing the weight of the true literals under I yields to value greater or equal to the bound. More formally, extending the relation $\models$, σ is satisfied if $\sum_{i=1}^{n} wh_\sigma(l_i) \cdot I^\top(l_i) \geq bnd_\sigma$, written $I \models \sigma$. Note that σ may be omitted from the above notation if its meaning is clear from context.

Just as a side note: in ASP-Core-2 standard [13] the constraint 2.2 is written as the headless rule $| :- \quad \#sum\{w_1, \ell_1 : \ell_1; ...; w_n, \ell_n : \ell_n\} < b.|$ Now a more formal definition of program can be given: a program Π is a set of rules and constraint, referred as $rules(\Pi)$ and $constraints(\Pi)$ respectively. The sets of rules and constraints define the set of atoms of the program and they are named $atoms(\Pi)$. Finally, I satisfies Π, written $I \models \Pi$, if $I \models r$ for all $r \in rules(\Pi)$ and $I \models \sigma$ for all $\sigma \in constraints(\Pi)$.

fare più carino

To make the discussion made so far more understandable, we will introduce the following example:

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

$$
\begin{array}{rlr}
r_\alpha : & \alpha \leftarrow \neg\alpha' & \alpha \in \{x, y, z\} \\
r_{\alpha'} : & \alpha' \leftarrow \neg\alpha & \alpha \in \{x, y, z\} \\
\sigma_1 : & \textsc{sum}\{1 : \overline{x};\ 1 : \overline{y} \qquad \} \geq 1 & \\
\sigma_2 : & \textsc{sum}\{1 : x;\ 2 : y;\ 2 : z\} \geq 3 &
\end{array}
$$

Note that there are six atoms (x, y, z, x', y', z')*, six rules and two constraints.* $wh_{\sigma_1}(\overline{x}) = 1, wh_{\sigma_1}(\overline{y}) = 1, wh_{\sigma_2}(x) = 1, wh_{\sigma_2}(y) = 2, wh_{\sigma_2}(z) = 2, bnd_{\sigma_1} = 1, bnd_{\sigma_2} = 3.$

A possible (total) interpretation I satisfying Π_{run} is: $I_1 = \{x, z, \overline{y}, x', y', z'\}$.

Since, $\sum_{i=1}^{2} wh_{\sigma_1}(l_i) \cdot I_1(l_i) = \sum 1 \cdot I_1(\overline{x}) + 1 \cdot I_1(\overline{y}) = 1 \geq bnd_{\sigma_1}$ and $1 \cdot I_1(x) + 2 \cdot I_1(y) + 2 \cdot I_1(z) = 3 \geq bnd_{\sigma_2}$. Given a program Π and an intepretation I, its *reduct* is defined as follows: $\Pi^I = \{H(r) \leftarrow B^+(r) \mid r \in \mathit{rules}, I \models B(r)\}$, please note that $\mathit{constraints}(\Pi^I) = \emptyset$. One interpretation may differ from another due to its stability. 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 $\mathit{SM}(\Pi)$ denote the set of stable models of Π. Taking in consideration example 1 we can notice that I_1 is not a stable model, that is, $I_1 \notin \mathit{SM}(\Pi_{run})$. This is because taking the partial assignment $J_1 = \{y'\}$ then $J_1 \models \Pi_{run}^{I_1}$ since $\Pi_{run}^{I_1} = \{y' \leftarrow\}$ and $J_1 \subset I_1$. Instead $I_2 = \{x, z, \overline{y}, \overline{x'}, y', \overline{z'}\} \in \mathit{SM}(\Pi_{run})$.

Continuing talking about the above example, a last consideration that let us to move towards the next chapter has to be addressed. The constraint σ_1 is a special one, it says: at least 1 of the 2 literals has to be satisfied. Notice that all the literals are flipped, so referring to the literals without being flipped it is actually saying 'at least 1 of the 2 have to be falsified' , and since $2 = 1 - 1$ it can be reformulated as 'at most 1 of them can be satisfied'. Thus, with an *At least One (ALO)* sum constraint is possible to define an *At Most One (AMO)* constraint.

2.2 ALO (clauses) and AMO as a Special Cases

Given a set $\{\ell_1, \ldots, \ell_n\}$, with $n \geq 1$, an *At Least One (ALO)* constraint over this set will enforce to have at least one literal true. As in the example 1 it can be expressed as:

$$\textsc{sum}\{1 : \ell_1;\ \cdots;\ 1 : \ell_n\} \geq 1 \tag{2.3}$$

This constraint can be written as a *set* of the form

$$\{\ell_1, \ldots, \ell_n\} \tag{2.4}$$

Usually this constraint is named also *clause* (following the *CNF* notation of the propositional logic). Modern ASP solvers enrich the input program with clauses enforcing I to be a model of rules of the form (2.1) (i.e., $\{p, \overline{\ell_1}, \ldots, \overline{\ell_n}\}$). If over this set instead is enforced an *At Most Constraint*, i.e., at most one literal is true, the following constraint is introduced:

$$\textsc{sum}\{1 : \overline{\ell_1};\ \cdots;\ 1 : \overline{\ell_n}\} \geq n - 1 \tag{2.5}$$

To enforce that at most one literal is true of a given set it is enough to enforce that at least $n-1$ literals are falsified. Since, intuitively, if two different literals are satisfied then $n-2$ literals are not falsified. More formally given an interpretation I, I satisfies at most 1 literal if $\sum_{i=1}^{n} I^{\top}(\overline{\ell_i}) \geq n-1$, or equivalently $\sum_{i=1}^{n} I^{\top}(\ell_i) \leq 1$. The AMO constraint (2.5) is compactly written $\textsc{amo}\{\ell_1, \ldots, \ell_n\}$.

Example 2 (Continuing Example 1)**.** *Note that σ_1 is the clause $\{\overline{x}, \overline{y}\}$, or also the AMO constraint* $\textsc{amo}\{x, y\}$.

2.3 Stable Model Search, Propagators and Learning

Currently ASP solvers employ a *conflict-driven clause learning* (CDCL) algorithm [10] to search a stable model. This is an algorithm also used in a lot of SAT solvers and it has been revealed to be a very effective one. The CDCL follows a pattern called *choose-propagate-learn*, where: the *choose* phase consists in deciding a literal to become true, such literal is named *branching literal*; *propagate* phase involves to deterministically derive new consequences from the current state (interpretation); *learn* phase, when a conflict arises, understands some new *clause* (constraint) that was not explicity defined in the program. To dive into each of these phases, understanding the CDCL, some previous concepts have to be mentioned.

A *conflict* occurs in an intepretation I, when two literals $\ell, \overline{\ell}$ are together in I. Given two clauses C, D of the form (2.4) (as described in 2.2) and a literal ℓ such that $\ell \in C$ and $\overline{\ell} \in D$ then the *resolution* ([14], [15]) step of C

and D upon ℓ, written $C \otimes_\ell D$, is equal to $(C \setminus \{\ell\}) \cup (D \setminus \{\bar{\ell}\})$. Intuitively, if $C \in \Pi$ and $D \in \Pi$ then since an intepretation cannot simultaneously satisfying ℓ and $\overline{\ell}$ then $C \setminus \{\ell\}$ or $D \setminus \{\overline{\ell}\}$ have to be satisfied, thus the following clause $(C \setminus \{\ell\}) \cup (D \setminus \overline{\ell})$ has to be satisfied. Please note that if an intepretation $I \models \Pi$ then $I \models C \otimes_\ell D$. So $C \otimes_\ell D$ is implied by the program, written $\Pi \models C \otimes_\ell D$. A clause C is *redundant* in Π if $\Pi \models C$. Given a clause C then an assignment I that *blocks* C is an assignment that falsifies all literals (and no others) of C, i.e., $I = \overline{C}$.

In the *choose* phase a branching (undefined) literal is selected. Every branching (or decision) literal is paired with a *decision level*. To understand the decision level is enough to now that at each choose phase the corresponding literal is added into a 'list' and the decision level is the index into the list (index starting from 1). The term '*backjumping* to a certain level l' refers to the process of 'forgetting' every decision made after decision level l, including the respective propagated literals, and then resuming the decision-making process starting from level l.

In the *propagate*, given an intepretation I, potentially some literal ℓ is derivated (or propagated) using a *Propagate* function according to some *inference rule*. An inference rule is a logical construct used to derive new *literals* (*conclusions*) from a current intepretation I (*premises*). It specifies the conditions under which certain statements (the conclusions) can be inferred from other statements (the premises). After a literal ℓ is inferred, thanks to some inference rule, then a *reason*, written *reason*(ℓ), is specified. This reason defines *why* ℓ has been propagated. More in detail, the reason is a clause of this form

$$R = \{\ell\} \cup \{\overline{\ell_1}, \ldots, \overline{\ell_n}\} \tag{2.6}$$

(2.6) represents the implication rule $\ell_1 \wedge \ldots \wedge \ell_n \rightarrow \ell$ and it specifies that when all the literals $\ell_1 \wedge \ldots \wedge \ell_n$ are true then also ℓ must be true. $B(R) = \overline{\{\overline{\ell_1}, \ldots, \overline{\ell_n}\}}$ is the body of the reason and $H(R) = \{\ell\}$. Given a reason R of a literal z, $J_r = B(R)$; note that J_r is the assignment blocking $R \setminus \{\ell\}$. A clause C

of the form (2.6) is a reason for ℓ, under the assignment I, if $I \supseteq B(C)$ and under intepretation J_c the propagator infers ℓ, thanks to some inference rule. Finally, $reason(\ell)$ has to be a reason of ℓ under I. Can happen that inferring a literal ℓ creates a conflic (i.e., $\overline{\ell} \in I$), in that case ℓ is a *conflict literal*. Moreover ℓ is also paired with a decision literal that it is inherited from the current decision level of the search. *Unit Propagation* is a specific kind of propagation, it is applied when given a clause C and a literal $\ell \in C$, $(C \backslash \{\ell\}) \cap \overline{I} = C \backslash \{\ell\}$. In this case the only way to satisfy C is setting ℓ to true, thus ℓ is propagated to true; the reason is exactly because the other literals are falses, so $reason(\ell) = C$. Please not the $reason(\ell)$ respresents $\overline{\ell_1} \wedge \ldots \wedge \overline{\ell_n} \rightarrow \ell$, where $\{\ell_1 \wedge \ldots \wedge \ell_n\} = C \cap \overline{I}$.The *Propagate* function is implemented using multiple *propagators*, calling them sequentially according to a priority list.

The whole algorithm, initially starts with an empty assignment (all literals undefined) and a the decision level (dl) to 0 , then the *propagate* phases takes place, inferring all the consequences. If a conflict is detected it means that the program is not satisfiable, since without any choice we got a conflict. Else if no conflict is deteceted then the *choose* phase decides the new branching literal. An important note is that just the a branching literal ℓ has a reason of the form $\{\ell\}$, representing $\rightarrow \ell$ (ℓ must be true), since it does not follows from any logic reasoning. Then, again the propagate phase is executed. If a conflict is detected then a process named *conflict analysis* starts. Starting from the reason of the conflict literal a (*backward*) resolution upon a literal of the last decision level is perfomed. This step is iteratively done until the new obtained (redundant) clause contains just one literal of the current decision level, this clause in called *Unique Implication Point (UIP)*. Given the UIP then a *backjump* operation is perfomed to the second highest decision level (*assertion level*); to update the internal state following a backjump, the *Unroll* function is invoked for each propagator. The assertion level is special in the sense that it is the deepest level at which adding the conflict-driven clause (UIP) would allow unit propagation to derive a new implication using that clause. Since the literal with highest dl after backjumping is the only undefined literal inside UIP then the unit propagation will infer that literal, that is, will flip the previous value. When the highest literal in the UIP has 0 as dl then the assertion level is -1 by default. Some **important** final notes: the reason for a literal ℓ must include at least one literal from the most recent decision level, excluding ℓ itself, otherwise, unit propagation cannot be performed; the smaller the cardinality of the conflict literal's reason, the higher the potential jump in the search space. After this, a new choice is made until either all atoms are defined or the assertion level is -1. The above described algorithm is defined below and it is mainly based on the algorithm defined in [16].

Algorithm 1 Typical CDCL algorithm

```
Input: An ASP program Π
begin
    I ← []
    dl ← 0
    if (Propagate(Π, I) == CONFLICT) then
        return Unsat
    while not AllVariablesAssigned(Π, I) do
        ℓ = PickBranchingLiteral(Π, I)
        dl ← dl + 1
        store(I, ℓ, dl)
        if (Propagate(Π, I) == CONFLICT) then
            β = ConflictAnalysis(Π, I)
            if (β < 0) then
                return Unsat
            else
                Backjump(Π, I, β)
                dl ← β
    return I
```

Let's assume for now that the Propagate function is exactly equal to the propagator implementing *Unit propagation*, this is usefull for the next example.

Example 3. *Let* Π *have, among others, the rules*

$$x \leftarrow \neg z \quad y \leftarrow \neg z \quad w \leftarrow x, y$$

Hence, a modern ASP solver materializes the clauses

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

For readability at each literal is associated the relative decision level as superscript. Let's assume that the current interpretation is $I = [\overline{w}^1]$ *and the decision level is* $dl = 1$*. Since no propagation is performed the algorithm goes directly to the propagate phase, let's assume that* $\overline{z}$ *is selected. Then, by unit propagation,* x *and* y *are inferred, thanks to the first two clauses. Unit propagation infers* x, y *from the first two clauses, and then* y *(a conflict literal) from the third clause. The correspective reasons are:* $reason(x) = \{x, z\}$, $reason(y) = \{y, z\}$ *and* $reason(\overline{y}) = \{w, \overline{x}, \overline{y}\}$*. The current intepretation* I *is equal to* $[\overline{w}^1, \overline{z}^2, y^2, x^2]$*. Since there are more than 1 literal of the decision level 2 (the last one) in the conflict clause* ($\{w, \overline{x}, \overline{y}\}$) *a backward resolution step is perfomed. Let's take arbitrarily* $\overline{x}$*:* $\{w, \overline{x}, \overline{y}\} \otimes_{\overline{x}} \{x, z\} = \{w, \overline{y}, z\}$*. Since both* $\overline{y}, z$ *are with decision level 2 then let's take arbitrarily* $\overline{y}$*:* $\{w, \overline{y}, z\} \otimes_{\overline{y}} \{y, z\} = \{w, z, z\} = \{w, z\}$*.*
$\{w, z\}$ *is UIP so let's backjump to the assertion level 1. This will unit propagate* z *with* $reason(z) = \{w, z\}$*. So the intepretation will look like:* $I = [\overline{w}^1, z^1]$

The main difference between a SAT solver and a ASP solver can be summarized focusing on the function *Propagate*(Π, I). SAT solvers employ mainly unit propagation inside the *Propagate* function, instead ASP solvers use, in addition on unit propagation, other two fundamental propagation functions: *Unfounded-free propagation* and *ConstraintPropagation*. The first one (*Unfounded-free propagation*) assesses that the intepretation is a stable model, but its details are out of the scope of this work. The second one (*Constraint-Propagation*) is used to derive new consequences from the constraints present in the program. For a SUM constraint σ of the form (2.2), solvers typically employ a specific *constraint propagator* [17], [18] leveraging the concept of *max possible sum (mps).* Given an intepretation I, a literal ℓ and a constraint σ of the form (2.2), a max possible sum, considering ℓ as false, is $mps_\sigma(I, \ell) = \sum_{j \in [1..n], \ell_j \neq \ell} w_j \cdot I^{\neg\perp}(\ell_j)$; intuitively, if all the undefined literal were true and ℓ was false then the sum would be equal to $mps(I, \ell)_\sigma$. An inference rule of a sum constraint essentially adds to I the literal ℓ if ℓ is required to (possibly) reach the bound b, i.e., if

$$mps_\sigma(I, \ell) < bnd_\sigma \tag{2.7}$$

In this case $reason(\ell) = \{\ell\} \cup lits_\sigma \cap \overline{I}$. A rational behind this is as follows: if in the best case (that is, when the sum is $mps(I, \ell)$) the sum does not reach the bound it means that ℓ cannot be false, that is, it is required to be true. Intuitively, the falses literals are the only justification why the bound bnd_σ could be not reached if ℓ was not added to I. In the special case of (2.5), 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 4 (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* (2.7) *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 (2.7) *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 $reason(\overline{y}) = \{\overline{x}, \overline{y}\}$.

Chapter 3

AMOSUM

In this chapter we propose a new constraint with relative syntax and semantics (section 3.1). This constraint combines a SUM constraint of the form (2.2) with a collection of AMO constraints of the form (2.5). Then the related rules for propagating are described in section 3.2. Finally the *Propagate* and the *Unroll* functions are defined in the last two sections

3.1 Syntex and Semantics

An *AMOSUM* constraint is defined as follows:

$$\textsc{amosum}\{w_1 : \ell_1\ [g_1];\ \cdots\ ;\ w_n : \ell_n\ [g_n]\} \geq b \tag{3.1}$$

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 $b, w_1, \ldots, w_n, g_1, \ldots, g_n$ are naturals numbers. As in (2.2): $\{w_1, \ldots, w_n\}$ is the set of weights and $wh_\sigma(l_i) = w_i$; $bnd_\sigma = b$ is the *bound* of the constraint σ. The new term is g_i, it represents the *group id* of the literal; every literal with the same group id is inside an *AMO* constraint. Relation $\in$ is now defined as $(w_i : \ell_i\ [g_i]) \in \sigma$ for all $i \in [1..n]$. $\mathbb{G}_\sigma$ is the set of possible group id, it means that $g_i \in \mathbb{G}_\sigma$ for all $1 \leq i \leq n$. Given a literal ℓ_i in the aggregate, $group_\sigma$ is function that maps ℓ_i to its group id, that is, $group(\ell_i) = g_i$. Given a group id $g \in \mathbb{G}_\sigma$ then all the literals in the same group are defined $lits_\sigma|_g = \{\ell \mid (w : \ell[g]) \in \sigma, w \in \mathbb{N}\}$. The relation $\models$ for a constraint σ of the form (3.1), defined in 2.1, is extended as follows: given an interpretation I, $I \models \sigma$ if $\sum_{i=1}^{n} w_i \cdot I^{\top}(\ell_i) \geq bnd_\sigma$, and $\sum_{\ell \in lits_\sigma|_g} I^{\top}(\ell) \leq 1$ for all $g \in \mathbb{G}_\sigma$. If the subscript σ is clear from context, it will be omitted.

The corresponding set of constraints defining an AMOSUM (3.1) using just SUM constraints of the form (2.2) is the following:

$$\begin{aligned}
&\textsc{sum}\{w_1 : \ell_1;\ \cdots;\ w_n : \ell_n\} \geq b \\
&\textsc{sum}\{1 : \overline{\ell_1^g};\ \cdots;\ 1 : \overline{\ell_{m_g}^g}\} \geq m_g - 1 \quad g \in G
\end{aligned}$$

where l_i^g is the *i-th* literal in the group g and m_g is the number of literals in the group g.

To provide a more concrete illustration, a concrete example is given.

Example 5 (Continuing Example 1)**.** Π_{run} *is rewritten by replacing* σ_1 *and* σ_2 *with*

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

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

3.2 Inference rules

The propagator for constraint 3.1 has 3 inference rules, the first 2 have a counterpart in the classical setting, instead the last one is a totally new one.

AMO inference rule The first inference rule is the one ensuring the at most one constraint (2.5): given a literal ℓ such that $(w : \ell[g]) \in \sigma$ for some $w \in \mathbb{N}$ and $g \in \mathbb{G}$, then ℓ is inferred as false, i.e., $\overline{\ell} \in I$, if there exists $\ell' \in lits|_g$ such that $\ell' \in I$. In this case $reason(\ell') = \{\overline{\ell}, \overline{\ell'}\}$

SUM inference rule This inference rule has a corresponding couterpart in the SUM constraint, that is, a literal is inferred as true if it is required to reach the bound. As it is done in (2.7) the concept of *max possible sum* is used, a literal ℓ is required to be true if all the literals in $lits|_{group(\ell)} \setminus \{\ell\}$ are falses (i.e., it is the only not false literal in its group) and the *maximum possible sum* (considering ℓ as false) would be less than bnd. In this case the max possible sum is different, since not all the literals are free to contribute to the overall sum; just one literal per group can be true (i.e., contribute to the sum). To get the the maximum possible sum it is enough to pick the maximum not false literal from each group; more formally: $mps\_amo_\sigma(I, \ell) = \sum_{g \in \mathbb{G}_\sigma \setminus \{group_\sigma(\ell)\}} mwh_\sigma(I, g)$ where $mwh_\sigma(I, g) := \max\{w \cdot I^{\neg\perp}(\ell) \mid (w : \ell\ [g]) \in \sigma\} \cup \{0\}$ is the maximum weight that group g can contribute to the overall sum. Finally, the literal ℓ is added to I if the following condition holds

$$mps\_amo_\sigma(I, \ell) < bnd_\sigma \tag{3.2}$$

Furthermore,$mwh_\sigma(g) = \max\{w \mid (w : x[g]) \in \sigma\} \cup \{0\}$
and $ml_\sigma(g) = \arg\max_{e \in S} wh(e)$ where $S = \{x \mid (w : x[g]) \in \sigma, w \in \mathbb{N}\}$

Henceforth, unless otherwise specified, *mps* will be used in place of *mps_amo*. In this case, $reason(\ell)$ is

$$lits_\sigma|_{group_\sigma(\ell)} \cup \bigcup_{g \in \mathbb{G}_\sigma \setminus \{group_\sigma(\ell)\}} just_\sigma(I, g), \tag{3.3}$$

where $just_\sigma(I, g) := \{\overline{\ell'}\}$ if $\ell' \in lits_\sigma|_g \cap I$ (i.e., there exists a true literal in the group g), and $\{\ell' \in lits_\sigma|_g \mid wh_\sigma(\ell') > mwh_\sigma(s)\}$ otherwise (i.e., the false literals in the group g that could had increased the overall sum).

Enforced falsity rule The third inference rule has not counterpart in AMO or SUM constraints. This rule enfore falsity of a literal that it is guaranteed to lead the max possible sum under the bound if that literal was true. Given a literal $\overline{\ell}$ it is added to I if the following condition holds:

$$mps_\sigma(I, \ell) + wh_\sigma(\ell) < bnd_\sigma \tag{3.4}$$

The rational behind this inference rules is the following: if ℓ was true it would contribute to the *mps*, if with this hypothesis the mps would be less than the bound then it means that ℓ must be false. In this case, $reason(\overline{\ell})$ is

$$\{\overline{\ell}\} \cup \bigcup_{g \in \mathbb{G}_\sigma \setminus \{group_\sigma(\ell)\}} just_\sigma(I, g). \tag{3.5}$$

Example 6 (Continuing Example 5)**.** *Already when I is empty, σ_3 infers z. In fact, z is the last undefined literal in part 2, and* (3.2) *gives*

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

From (3.3)*, $reason(z) = \{z\}$.*

Example 7. *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 $r_1 = reason(z)$ because it could increase the overall sum. It is easy to see that if $I = [\overline{w}]$ from (3.2) *z is inferred with $r_2 = reason(z) = \{z, w\}$; note $r_2 \subset r_1$. This case arises from the fact that if y was undefined then z would be inferred anyway, more technically: the increment to the possible sum, given from removing y from the reason, is not enough to increase the mps up to the bound, thus the reason without y is a valid reason. As discussed in section 2.3, a smaller reason in size is preferred. In chapter 4 an algorithm to compute the minimal and cardinality minimal reason is proposed. Finally, if $I = [x, \overline{y}, \overline{w}]$, then z is inferred with $reason(z) = \{z, w, \overline{x}\}$; again, in this specific case x could be ignored.*

3.3 Propagate

In this section on the details about the *Propagate* function implemented in the AMOSUM propagator. The Propagate function is divided in two phases:

Algorithm 2 Propagate

Input : A constraint σ, an interpretation I, a literal $\ell \in I$

Output: A list of propagated literals

begin
    $next\_phase \leftarrow update\_phase(\sigma, I, \ell)$;
    $propagated\_lits \leftarrow \emptyset$;
    if $next\_phase = \top$ **then**
        $propagated\_lits \leftarrow propagate\_phase(\sigma, I)$;
    return $propagated\_lits$;

update_phase and *propagate_phase*; the first one updates the internal states of the propagator and understands if the propagation phase is required; the second one is the 'actual' propagator, it implements the inference rules and correlated reasons, as described in section 3.2. The Propagate function is defined with the algorithm 2.

The update phase, is described in algorithm 3. Abusing of notation the relation $\in$ defined in section 3.1 is extended, defining $\ell \in \sigma$ true if $(w : \ell[g]) \in \sigma$ for some weight $w \in \mathbb{N}$ and group $g \in \mathbb{G}_\sigma$.

Algorithm 3 update_phase

Input : An aggregate σ, an interpretation I, a literal $\ell \in I$.

Output: Boolean to move to the next phase

begin
    if $\ell \in \sigma$ **then**
        $g \leftarrow group(\ell)$;
        if $mwh(g) = wh(\ell)$ **then**
            return $\bot$
        $mps \leftarrow mps - mwh(I, g) + wh(\ell)$;
    else if $\bar{\ell} \in \sigma$ **then**
        $g \leftarrow group(\bar{\ell})$;
        if $wh(\bar{\ell}) = mwh(I \setminus \{\ell\}, g)$ *and* $lits|_{group(\ell)} \cap I = \emptyset$ **then**
            $mps \leftarrow mps + mwh(I, g) - wh(\ell)$;
        else if $|lits|_{group(\ell)} \setminus (I \cup \bar{I})| = 1$ **then**
            return $\top$
        else
            return $\bot$
    else
        return $\bot$
    return $\top$

3.4 Unroll

Actually, in addition to the *Propagate* function, called when a literal has been chosen as a branching literal, also a *Unroll* function has to be defined, to update the internal state of the propagator when a literal becomes undefined (due to some backjump in the search process).

Algorithm 4 propagate_phase

Input : A constraint σ, an interpretation I

Output: A set S of pairs *(literal,reason)*,

begin
    $S \leftarrow \emptyset$
    $\mathit{falses} \leftarrow \emptyset$
    for $g \in \mathbb{G}$ **do**
        `// AMO inference rule`
        if $\mathit{lits}|_g \cap I = \{\ell\}$ **then**
            $S \leftarrow S \cup (\overline{\ell_i}, \{\overline{\ell_i}, \overline{\ell}\}) \quad \forall \ell_i \in \mathit{lits}|_g \setminus (\{\ell\} \cup I \cup \overline{I})$
        `// Enforced falsity rule`
        for $l \in \mathit{lits}|_g$ **do**
            if $\ell \notin I \cup \overline{I}$ **then**
                if $\mathit{mps} - \mathit{mwh}(I, g) + \mathit{wh}(\ell) < \mathit{lb}$ **then**
                    $\mathit{rns}_\ell = \mathit{lits}_\sigma|_g \cup \bigcup_{x \in \mathbb{G}_\sigma \setminus \{g\}} \mathit{just}_\sigma(I, x)$
                    $S \leftarrow (\overline{\ell}, \mathit{reason}(\ell))$
                    $\mathit{falses} \leftarrow \mathit{falses} \cup \{\ell\}$
        `// SUM inference rule`
        if $\mathit{lits}|_g \setminus (I \cup \mathit{falses}) = \{\ell\}$ *and* $\mathit{lits}|_g \cap I = \emptyset$ **then**
            if $\mathit{mps} - \mathit{wh}(\ell) < \mathit{lb}$ **then**
                $S \leftarrow S \cup (\ell, \{\overline{\ell}\} \cup \bigcup_{x \in \mathbb{G}_\sigma \setminus \{g\}} \mathit{just}_\sigma(I, x))$
    return S;

Algorithm 5 Unroll

Input : A constraint σ, an interpretation I, a literal $\ell \notin (I \cup \overline{I})$, previous value v

Output: Updated mps

begin
    if $\ell \in \sigma$ *or* $\overline{\ell} \in \sigma$ **then**
        if $\overline{\ell} \in \sigma$ **then**
            $\ell \leftarrow \overline{\ell}$
            $v \leftarrow \neg v$
        $g \leftarrow group(\ell)$
        if $v = \top$ **then**
            `//` ℓ `was true`
            if $mwh(I, g) > weight[\ell]$ **then**
                $mps \leftarrow mps - weight[\ell] + weight[mw(G)]$;
        else
            `//` ℓ `was false`
            if *G has not true literal* **then**
                $mps \leftarrow mps - max\_w + weight[mw(G)]$;

Chapter 4

Minimizing reason

In this section, we address the problem of minimizing a reason for an AMOSUM constraint of the $\geq$ type. In Section 4.1, we discuss some important properties of the reason. Subsequently, we introduce the concept of a *redundant literal* and explain the notion of *increment*,in section 4.2 and 4.3 respectively. Utilizing these concepts, we define two algorithms in the final sections: one for obtaining the minimal reason and the other for obtaining the cardinality minimal reason.

All the discussion applies on the inference rule (3.2) for a constraint σ of the from (3.1), thus, when it is possible, the subscript σ is omitted .

4.1 Properties of a reason

An important property of a reason R is that $\Pi \models R$, i.e., it is redundant. Let R_1, R_2 be two reason of z of the form:

$$R_1 = \{z\} \cup \{\ell_1, \ldots, \ell_n, y\} \tag{4.1}$$

$$R_2 = \{z\} \cup \{\ell_1, \ldots, \ell_n, \overline{y}\}, \tag{4.2}$$

where $n \geq 0$. In this case $R_1 \setminus \{y\} = R_2 \setminus \{\overline{y}\}$ and $z, y \in R_1$ and $z, \overline{y} \in R_2$. Let Π be a program such that $\Pi \models R_1$ and $\Pi \models R_2$. Let $I \models \Pi$ then $I \models R_1$ and $I \models R_2$, hence $I \models R_1 \wedge R_2$. We have seen in 2.3 that $R_1 \wedge R_2 \rightarrow R_1 \otimes_y R_2$, thus $I \models R_1 \otimes_\ell R_2$, hence $\Pi \models R_1 \otimes_y R_2 = \{z\} \cup \{\ell_1, \ldots, \ell_n\} = R$. Since $z \in R$

and $\Pi \models R$ then R is a reason of z in Π.

Example 8. *continuing example 7*
When $I = [\overline{y}, \overline{w}]$ *then* z *is inferred, thanks to* (3.2),*with* $reason(z) = \{z, y, w\} = R_1$. *If* $I = [y, \overline{w}]$ *then* $mps(I, z) = wh(y) = 2 < 3$ *and* $lits\ |_{group(z)} \setminus \overline{I} = \{z\}$ *so* z *is inferred with* $reason(z) = \{z, \overline{y}, w\} = R_2$. *Since* $R_1 = \{z\} \cup \{w, y\}$, $R_2 = \{z\} \cup \{w, \overline{y}\}$ *are two reason of the form* **??** *then* $\Pi \models R_1 \otimes_y R_2 = \{z, w\} = R$ *and* R *is a reason of* z.

We can see in the above example that y can be removed from the reason, getting R, and R continues to be a reason, so it is *redundant*.

4.2 Redundant literal

Let L be set such that $\ell \in L$, then $L_{\overline{\ell}} = (L \setminus \{\ell\}) \cup \{\overline{\ell}\}$. A literal ℓ is *redundant* in a reason R of the literal z, under intepretation I, if $R_{\overline{\ell}}$ is a reason of z under I_ℓ. If $\ell \in R$ then $R, R_{\overline{\ell}}$ are of the form 4.1 and 4.2 respectively, and $R \otimes_\ell R_{\overline{\ell}} = R \setminus \{\ell\} = R'$ is a reason of z. Note that R' is a reason under $I' = \overline{R' \setminus \{z\}}$; thus, given that $I' \subseteq I$, then R' is a reason under I. It is easy to see that every redundat literal of a reason R can be removed from it.

Example 9. *Continuing example 7*
When $I = [\overline{y}, \overline{w}]$ *then* z *is inferred, thanks to* (3.2),*with* $reason(z) = \{z, y, w\} = R$; R *is a reason of* z *under* I.
Let's check if y *is redundant.* $R_{\overline{y}} = (R \setminus \{y\}) \cup \{\overline{y}\} = \{z, \overline{y}, w\}$ *and* $I_y = [y, \overline{w}]$. *With* $J_{R_{\overline{y}}} = [y, \overline{w}]$ *the propagator, thanks to* (3.2), *would infer* z. *So,* $R_{\overline{y}}$ *is a reason for* z *and* y *is redundant in* R. *Now, let's take the case where* $I = [x, \overline{y}, \overline{w}]$, *and* z *is inferred with* $reason(z) = \{z, w, \overline{x}\} = R$, *that is,* $\overline{w} \wedge x \to z$. *Let's check if* $\overline{x}$ *is redundant.*
$R_x = \{z, w, x\}$ *and it is a reason of* z *under* $I_{\overline{x}}[\overline{w}, \overline{y}, \overline{x}]$, *since* $J_{R_x} = \overline{\{w, x\}} = [\overline{w}, \overline{x}] \subseteq I_{\overline{x}}$ *infers* z. *Hence,* x *is redundant in* R.

Continuing the discussion about the example 9. Let's examine the first case more in detail: $reason(z) = \{z, w, y\}$ is a reason of z under $I = [\overline{y}, \overline{z}]$. This

is true because, given $J_R = \overline{\{z, w, y\} \setminus \{z\}} = \{w, y\}$, $mps(J_R, z) = wh(x) = 1 < 3$ and $lits \mid_{group(z)} \setminus \overline{I} = \{z\}$. Since $R_{\overline{y}} = \{z, w, \overline{y}\}$ then $J_{R_{\overline{y}}} = [\overline{w}, y]$ and $mps(J_{R_{\overline{y}}}, z) = wh(y) = 2 < 3$ and $lits \mid_{group(z)} \setminus \overline{I_y} = \{z\}$, thus z would be added in I_y. More in concrete, y is redundant because $mps(J_R, z) + inc(y) = 1 + 1 < bnd = 3$ and $y \notin group(z)$, where the $inc(y) = mps(J_{R_{\overline{y}}}, z) - mps(J_R, z) = 2 - 1$. Now, let's take the other case, where $I = [x, \overline{y}, \overline{w}]$, and z is inferred with $reason(z) = \{z, w, \overline{x}\}$, that is, $\overline{w} \wedge x \to z$. As already seen $J_{R_x} = [\overline{x}, \overline{w}]$.
$mps(J_{R_x}, z) = wh(y) = 2 < 3$ and $lits \mid_{group(z)} \setminus \overline{I_{\overline{x}}} = \{z\}$, so $reason(z) = \{z, x, w\}$ and $\overline{x}$ is redundant.

4.3 Increment

As mentioned before, increment of ℓ, written $inc(\ell)$, defines how much the mps increases when that literal is flipped in the reason (to possibly be removed).

4.3.1 False literal

Let ℓ be a *false* literal that is, $\overline{\ell} \in I$. A reason R of a literal z under I can contain ℓ, let's assume R to be such reason. Let $J_R = \overline{R \setminus \{z\}}$, $J_{R_{\overline{\ell}}} = \overline{R_{\overline{\ell}} \setminus \{z\}}$, thus $\ell \in J_{R_{\overline{\ell}}}$.
Note that $mps(J_R, z) = \sum_{g \in \mathbb{G} \setminus \{group(z)\}} mwh(J_R, g)$ and
$mps(J_{R_{\overline{\ell}}}, z) = \sum_{g \in \mathbb{G} \setminus \{group(z)\}} mwh(J_{R_{\overline{\ell}}}, g)$. Since $J_R \setminus lits|_{group(\ell)} = J_{R_{\overline{\ell}}} \setminus lits|_{group(\ell)}$, that is, all the literals outside $group(\ell)$ remain with the same value; then just the maximum weight of $lits|_{group(\ell)}$ can change. Hence,

$$inc(\ell) = mps(J_{R_{\overline{\ell}}}, z) - mps(J_r, z) = mwh(J_{R_{\overline{\ell}}}, group(\ell)) - mwh(J_r, group(\ell))$$

Hence, when a literal has to be checked to be redundant and it is false then $inc(\ell)$ has to be considered has the increment of the mps.

4.3.2 True literal

Let ℓ be a *true* literal that is, $\ell \in I$. A reason R of a literal z under I can contain $\overline{\ell}$, let's assume R to be such reason. Then $J_R = \overline{R \setminus \{z\}}$ and $J_{R_\ell} = \overline{R_\ell \setminus \{z\}}$, thus $\overline{\ell} \in J_{R_\ell}$.
Since $\ell \in I$ then R, by construction, does not containt any literal of $group(\ell)$ except ℓ, then R_ℓ will do the same. Since $J_{R_\ell} = \overline{R_\ell \setminus \{z\}}$ then also J_{R_ℓ} has no literal of group of ℓ except $\overline{\ell}$ (since it is false, it does not contribute to the *mps*). That is, $mwh(J_{R_\ell}, group(\ell)) = \max\{mwh(x) \mid group(x) = group(\ell)\} = mwh(group(\ell))$. It is easy to see that

$$inc(\ell) = mps(J_{R_\ell}, z) - mps(J_r, z) = mwh(group(\ell)) - wh(\ell)$$

Hence, when a literal has to be checked to be redundant and it is true then $inc(\ell)$ has to be considered has the increment of the *mps*.

4.3.3 Algorithm

In this section is proposed the algorithm computing the increment and its lower bound 0 is proved. The increment is always non-negative since removing a literal from a reason in the worst case does not affect the mps.

Theorem 1. *Given a reason R and a literal $\ell \in R$ then $inc(\ell) \geq 0$.*

Proof. Case when $I \models \ell \in R$. Then $inc(\ell) = mwh(group(\ell)) - wh(\ell)$. Since $mwh(group(\ell))$ is the maximum weight in $lits\,|_{group(\ell)}$ and $\ell \in lits\,|_{group(\ell)}$ then $mwh(group(\ell)) \geq wh(\ell)$ and $inc(\ell) \geq 0$.

Case when $I \models \overline{\ell} \in R$. For readability let define $g = group(\ell)$.
Then $inc(\ell) = mwh(J_{R_{\overline{\ell}}}, g) - mwh(J_R, g)$.
Since $J_{R_{\overline{\ell}}} = (J_R \setminus \{\overline{\ell}\}) \cup \{\ell\}$ then

$$mwh(J_{R_{\overline{\ell}}}, g) = \max\{w \cdot J_R^{\neg\perp}(\ell) \mid (w : \ell\ [g]) \in \sigma\} \cup \{wh(\ell)\}$$

It is trivial to see that

$$\max\{w \cdot J_R^{\neg\perp}(\ell) \mid (w : \ell\ [g]) \in \sigma\} \cup \{wh(\ell)\} \geq \max\{w \cdot J_R^{\neg\perp}(\ell) \mid (w : \ell\ [g]) \in \sigma\}$$

Since $\max\{w \cdot J_R^{\neg\perp}(\ell) \mid (w : \ell\ [g]) \in \sigma\} = mwh(J_R, g)$ then

$$\max\{w \cdot J_R^{\neg\perp}(\ell) \mid (w : \ell\ [g]) \in \sigma\} \cup \{wh(\ell)\} \geq mwh(J_R, g).$$

Given that $\max\{w \cdot J_R^{\neg\perp}(\ell) \mid (w : \ell\ [g]) \in \sigma\} \cup \{wh(\ell)\} = mwh(J_{R_{\overline{\ell}}}, g)$, finally,

$$mwh(J_{R_{\overline{\ell}}}, g) \geq mwh(J_R, g)$$

In both cases $inc(\ell) \geq 0$.

Hence, $inc(\ell) \geq 0$ for all $\ell \in R$ □

The following algorithm computes the increment as previously described.

Algorithm 6 inc

Input: literal $\ell \in R$, interpretation I, reason R

```
. begin
    if I ⊨ ℓ then
        return mwh(group(ℓ)) − wh(ℓ)
    else
        J_R = \overline{R \ {z}}
        J_{R_ℓ̄} = \overline{R_ℓ̄ \ {z}}
        return mwh(J_{R_ℓ̄}, group(ℓ)) − mwh(J_R, group(ℓ))
```

Given a reason R, intepretation I and a set $S \subseteq S$ the *total increment of* S denoted $inc(S, I, R) = \sum_{\ell \in S} inc(\ell, I, R \setminus S \cup \{\ell\})$

4.4 Minimality

In this section we propose two algorithms to minimize the reason generated by the propagator. The first algorithm (4.4.1) is an algorithm to get the minimal reason and it has linear complexity in the size of the reason. In subsection 4.4.2), instead, a new algorithm to compute the cardinality minimal reason is proposed, with *pseudo-polynomial* complexity. For both algorithms a proof of correcntess is provided and for the second one *NP-Hardness* is proved.

4.4.1 Minimal reason

Given a reason R of a literal ℓ then it is minimal if there not exists a literal $\ell' \in R$ such that $R \setminus \{\ell'\}$ is a reason of ℓ. In other words, R is minimal if it does not containts redundant literals. As seen before with each literal ℓ in a reason R has an *increment*. That is, the increment to the mps when ℓ is removed from the reason.

At this point, an important parallel must be drawn: minimizing a reason R equates to maximizing the number of literals removed from R. This first algorithm identifies the *maximal* set of literals to be removed from R.

Maximal Subset Sum As mentioned before finding the minimal reason is equal to find the maximal set of literal to remove from the reason. Let R be a reason of a literal z under interpretation I and $S \subseteq R'$ be a set of literals where $R' = R \setminus lits|_{group(z)}$. S can be removed from R if

$$\sum_{e \in S} inc(e, I, (R' \setminus S) \cup \{e\}) \leq s$$

where $s = bnd - mps(I, z) - 1$. To comprehend the reasoning behind s, it is crucial to observe that, given $J = B(R' \setminus S)$ then

$$mps(J, z) = mps(I, z) + \sum_{e \in S} inc(e, I, (R' \setminus S) \cup \{e\})$$

For $R' \setminus S$ to remain a reason the condition $mps(R' \setminus S, z) \leq bnd - 1$ must continue to hold. Since $mps(I, z) \leq bnd - 1$ (otherwise R would not be a reason) and $\sum_{e \in S} inc(e, I, (R' \setminus S) \cup \{e\}) \leq bnd - mps(I, z) - 1$ then

$$mps(I, z) + \sum_{e \in S} inc(e, I, (R' \setminus S) \cup \{e\}) \leq mps(I, z) + bnd - mps(I, z) - 1 = bnd - 1$$

. Now, the entire algorithm can be defined. It begins with an empty set S and a current increment ci equal to 0. For each literal ℓ that is not in the group of z it verifies if ℓ is *redundant*. If it is, it is added to the set S and the ci increase of $inc(\ell)$.

Algorithm 7 Maximal Subset sum (mss)

Input : intepretation I, reason R of z, intepretation I, threshold s

Paramenters: *inc* function

Output : subset maximal

begin

  $S \leftarrow [\,]$

  $ci \leftarrow 0$

  for $\ell \in R \setminus lits|_{group(z)}$ **do**

    if $ci + inc(\ell, I, R \setminus S) \leq s$ **then**

      $ci \leftarrow ci + inc(\ell, I, R \setminus S)$

      $S \leftarrow S \cup \{\ell\}$

  return S

Proof Correctness Before proving the theorem about the correcntess of algorithm 7 the following lemma has to be proved.

Lemma 1. *Given a reason R, a set $S \subseteq R$, a literal $\ell \in R \setminus S$ and $ci, s \in \mathbb{N}$. If $ci + inc(\ell, I, R \setminus S) > s$ then every superset $S' \supseteq S \cup \{\ell\}$ yields to an increment greater of s*

Proof. The total increment of $S' = inc(S', I, R) = inc(S, I, R) + inc(\ell, I, R \setminus S) + \sum_{\ell \in S'} inc(x, I, R \setminus S' \cup \{x\})$. Since $inc(S, I, R) + inc(\ell, I, R \setminus S) > s$ by assumption and ,thanks to theorem 1, $inc(x, I, R \setminus S' \cup \{x\}) \geq 0$ then $inc(S', I, R) > s$. Hence, S' is not a subset of R giving as sum a value less than s.

□

Theorem 2. *The Maximal Subset Sum algorithm returns the maximal subset S where its sum of increments of S is less than $bnd - 1$*

Proof. Let S_m the final subset returned by *mss*. Arguing by contradiction, if it is not maximal it means that there is a literal $\ell \in R \setminus S_m$ that could be added to S_m. But since $ci + inc(\ell, I, R \setminus S) \geq s$ for some S. Since $S \subseteq S_m$, then $S \cup \{\ell\} \subseteq S_m \cup \{\ell\}$,; thanks to theorem 1 $S_m \cup \{\ell\}$ is not a subset giving as sum a value less then s, that is, ℓ cannot be added to S_m. Hence, S_m is maximal, $\square$

4.4.2 Cardinality minimal reason

This section, as previously mentioned, we propose a new algorithm to minimize the reason, the main difference is that with algorithm 7 the final reason is not guaranteed to be cardinality-minimal. This new approach can find a cardinality-minimal reason, at the price of having a *pseuso-polynomial* algorithm. Before introducing the algorithm some preliminars have to be done.

Preliminars All the previous notation continue to hold. Let I an interpretation, R be a reason of z generated under I and $S \subseteq R$ a set of redundant literals of R.

Function $ml_{inc} : \mathcal{P}(R) \times G \mapsto R$ returns the *maximum increment literal in S of group g*; that is, $ml_{inc}(S, g) = \arg\max_{k \in S \cap lits|_{group(g)}} inc(k)$.

Given a set , a literal ℓ is active in S if $ml_{inc}(S, group(\ell)) = \ell$; the set of active literal of S is equal to $A_S = \{\ell \in S \mid \ell \text{ is active in S}\}$. Given a literal $l \in R$ then $blw : R \mapsto \mathcal{P}(R)$ is a function that returns all the literals *below* ℓ, that is, $blw(\ell) = \{k \in lits|_{group(\ell)} \mid inc(k, I, R) \leq inc(l, I, R)\}$. Function $sum(S) = \sum_{\ell \in A_S} inc(\ell, I, R)$ is the sum of all increments of active literals of S.

A extension of a set S with a literal $\ell \in R \setminus S$, written as $S' = S \xrightarrow{\text{ext}} \ell$, is equal to $S' = S \cup blw(\ell)$.

A sufficient condition for a literal ℓ to be active in $S' = S \xrightarrow{\text{ext}} \ell$ is that $S \cap group(\ell) = \emptyset$. Let $(lits\_ord_g, \preceq)$ be an ordered set where $lits\_ord_g = lits|_g$ and $\preceq = \{(l, l') \in lits|_g \times lits|_g \,:\, inc(l, I, R) \geq inc(l', I, R)\}$.

Let $L = \{l_1^1, \ldots, l_{m_1}^1, \ldots, l_1^g, \ldots, l_{m_g}^g, \ldots, l_1^k, \ldots, l_{m_k}^k\}$ where $k = |\mathbb{G}|$, m_g is the size of $lits\_ord|_g$, l_j^g is the j-th literal in $lits\_ord_g$. The set L_j represent the first j elements of L and $n = |L|$. Abusing of notation, given a literal $\ell_j \in L$, $\{\ell_j\} = L_j \setminus L_{j-1}$, i.e., ℓ_j is the *j-th* literal in L.

Cardinality-Maximal Set As done in section 4.4.1 to find a minimal reason R' starting from R we compute a maximal subset S of R of redudant literals to remove from R. In the first approach, the cardinal-minimality property is not ensured, instead in this case S is *cardinality-maximal*. Differently from the 7, here the increment is **always** with respect the *'initial'* increment, that is, with respect to the initial reason R; instead in the first algorithm the increment dependes from the *current* reason.

The algorithm 8 computes the cardinality-maximal subset S of L such that the total increment $inc(S, I, R) \leq s$, using a *dynamic* approach. It creates a matrix M where each cell $M_{i,j}$ has domain $\mathcal{P}(L) \cup \{\Box\}$. $M_{i,j}$ represents the cardinality-maximal set with $sum(M_{i,j}) = i$ and $M_{i,j} \subseteq L_j$, if $M_{i,j} = \Box$ it means that such subset does not exists. When $M_{i,j} = \Box$ then, abusing of notation, $|M_{i,j}| = -1$. The matrix is initialized with function *Init* 9. This function will initialize just the first row and column. It is trivial to see that $M_{0,j}$ will contain all the literals of L_j with increment equal to 0, since the sum $i = 0$. Thus, the first row will be $M_{0,j} = \{\ell \mid \ell \in L_j \wedge inc(\ell, I, R) = 0\} \quad \forall j \in \{1, \ldots, n\}$. When the sum $i > 0$ then with $L_0 = \emptyset$ is impossible to reach i, that's why $M_{i,0} = \Box \quad \forall i \in \{1, \ldots, s\}$. After initilization the algorithm 8 constructs each cell $M_{i,j}$, with $1 \geq i \geq s$ and $1 \geq j \geq n$, starting from $M_{i,j-1}$ and $M_{i-w,j-1}$, where $w = inc(\ell_j, I, R)$. Let $S_{\overline{\ell}}$ and S_ℓ represent the cardinality maximum sets, considering literals in L_j and having a total increment equal to i, with ℓ_j non-active and ℓ_j active, respectively. By selecting the larger set between these two, we obtain the maximum subset considering literals in L_j. $S_{\overline{\ell}} = M_{i,j-1}$, since is the largest set giving as sum i and ℓ is not active given that $M_{i,j-1}$ considers just the first $j-1$ literals; note, this does not mean that $M_{i,j-1} \cap \{\ell\} = \emptyset$, since can happen that for some $d < j$ $M_{i,d} = M_{l,k} \xrightarrow{\text{ext}} \ell_d$ and $group(\ell_d) = group(\ell_j)$. Instead finding S_ℓ is not so trivial, we cannot just take $M_{i-w,j-1}$, since $M_{i-w,j-1}$ could containt a literal ℓ_d such that $d < j$, $group(\ell_d) = group(\ell_j)$ and in that case $sum(M_{i-w,j-1}) = sum(M_{i-w,j-1} \xrightarrow{\text{ext}} \ell_j)$; it is due to the fact that ℓ_j would not be active. Since L is ordered, and within each group there is a descending order, and since each cell $M_{i,j}$ is computed from left to right then $M_{i-w,j-1} \cap lits_{group(\ell)} = \emptyset$ is also a *necessary* condition to have ℓ active in $M_{i-w,j-1} \xrightarrow{\text{ext}} \ell$ (so it becomes necessary and sufficient). Since, by construction, $M_{i-w,j-1}$ can contain just literals *'greater'* (in terms of increment) of ℓ_j, and if this happens also ℓ_j is inside $M_{i-w,j-1}$ then it is sufficient to check that $M_{i-w,j-1} \cap \{\ell_j\} = \emptyset$. Taking $M_{i-w,k} \xrightarrow{\text{ext}} \ell_j$ with $k = \arg\max_{k \in \{1,\ldots,j-1\}} \ell_j \notin M_{i-w,k}$ yields the cardinality maximum subset with a sum of i, and ℓ_j active. Hence, $S_\ell = M_{i-w,k}$. Subsequently, after the for loop is completed all cells of the matrix are correct and it is enough to take the largest set in the last column, that is, the largest set giving as sum a value less or equal to s considering all literals in L.

Proof Correcntess To formally prove that algorithm 8 is correct a lemma has to be proved.

Lemma 2. *Given a set of literals L, a literal $\ell \in L$ and a set $S_{\bar{\ell}} \subseteq L$ and $S_\ell \subseteq L$ and a sum s. Let $S_{\bar{\ell}}$ is the maximum cardinality subset giving s as sum with ℓ being an non-active literal and S_ℓ is the maximum cardinality subset with ℓ being an active literal giving s as sum.*
Let $S = \arg\max_{X \in \{S_\ell, S_{\bar{\ell}}\}} |X|$.
Then $S \subseteq L$ is a maximum cardinality subset that gives as sum s.

Proof. Let's assume by contradiction that S is not the maximum cardinality subset that gives as sum s. So there exists a set $S' \subseteq L$ such that $|S'| > |S|$. If $\ell \notin A_{S'}$ then $|S'| > |S_{\bar{\ell}}|$, that is $S_{\bar{\ell}}$ is not the maximum cardinality set with ℓ being a *non-active literal* giving as sum s, but this is a contradiction. So $\ell \in A_{S'}$ then $|S'| > |S_\ell|$, that is S_ℓ is not the maximum cardinality set with ℓ being a *active literal* giving as sum s, but this is a contradiction. So $\ell \in A_{S'}$ and $\ell \notin A_{S'}$ and this is a contradiction. □

Theorem 3. *The Cardinality Maximal Subset Sum algorithm returns the cardinality maximal subset S where its sum of increments of S is less than $bnd - 1$*

Proof. □

Algorithm 8 Cardinality Maximum Subset Sum (cmss)

Input : L, I : interpretation, $s \in \mathbb{N}$

Parameters: $group$: Group Function, inc : Increment Function

Output : S: Set of literals representing the maximum subset

```
begin
    n ← |L|
    M ← Init(I, R, L, s)
    for i ∈ {1, ..., s} do
        for j ∈ {1, ..., s} do
            ℓ ← L[j − 1]
            w ← inc(ℓ)
            M_{i,j} ← M_{i,j−1}
            if i ≥ w then
                if (□ ∉ M_{i,j}) ∨ (□ ∉ M_{i−w,j−1}) then
                    for k ← j − 1 to 0 do
                        if ℓ ∉ M_{i−w,k} ∨ □ ∈ Mi − w, k then
                            break
                    bool ← □ ∉ M_{i−w,k} ∧ ℓ ∉ M_{i−w,k}
                    wls ← M_{i−w,k} --ext--> ℓ   if bool Else □
                    M_{i,j} ← arg max{|wls|, |M_{i,j−1}|}
    S = arg max_{i∈{0,...,s}} |M_{i,n}|
    return S
```

Algorithm 9 Init

Input: interpretation I, reason R, L, threshold $s \in \mathbb{N}$

```
. begin
    M_{0,j} = {ℓ | ℓ ∈ L_j ∧ inc(ℓ, I, R) = 0}   ∀j ∈ {1, ..., |L|}
    M_{i,0} = □   ∀i ∈ {1, ..., s}
```

Bibliography

[1] V. Marek and 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*, vol. 25, no. 3,4, pp. 241–273, 1999. DOI: 10.1023/A:1018930122475.

[3] G. Brewka, T. Eiter, and M. Truszczynski, “Answer set programming at a glance,” *Commun. ACM*, vol. 54, no. 12, pp. 92–103, 2011. DOI: 10.1145/2043174.2043195. [Online]. Available: https://doi.org/10.1145/2043174.2043195.

[4] M. Gelfond and V. Lifschitz, “Logic programs with classical negation,” in *Logic Programming: Proc. of the Seventh International Conference*, 1990, pp. 579–597.

[5] M. Bartholomew, J. Lee, and Y. Meng, “First-order semantics of aggregates in answer set programming via modified circumscription,” in *Logical Formalizations of Commonsense Reasoning, AAAI Spring Symposium*, AAAI, 2011. [Online]. Available: http://www.aaai.org/ocs/index.php/SSS/SSS11/paper/view/2472.

[6] W. Faber, G. Pfeifer, and N. Leone, “Semantics and complexity of recursive aggregates in answer set programming,” *Artif. Intell.*, vol. 175, no. 1, pp. 278–298, 2011. DOI: 10.1016/j.artint.2010.04.002. [Online]. Available: http://dx.doi.org/10.1016/j.artint.2010.04.002.

[7] P. Ferraris, "Logic programs with propositional connectives and aggregates," *ACM Trans. Comput. Log.*, vol. 12, no. 4, p. 25, 2011. DOI: 10.1145/1970398.1970401. [Online]. Available: http://doi.acm.org/10.1145/1970398.1970401.

[8] M. Gelfond and Y. Zhang, "Vicious circle principle and logic programs with aggregates," *Theory and Practice of Logic Programming*, vol. 14, no. 4-5, pp. 587–601, 2014. DOI: 10.1017/S1471068414000222. [Online]. Available: http://dx.doi.org/10.1017/S1471068414000222.

[9] L. Liu, E. Pontelli, T. C. Son, and M. Truszczynski, "Logic programs with abstract constraint atoms: The role of computations," *Artif. Intell.*, vol. 174, no. 3-4, pp. 295–315, 2010. DOI: 10.1016/j.artint.2009.11.016. [Online]. Available: http://dx.doi.org/10.1016/j.artint.2009.11.016.

[10] M. Gebser, B. Kaufmann, and T. Schaub, "Conflict-driven answer set solving: From theory to practice," *Artif. Intell.*, vol. 187, pp. 52–89, 2012. DOI: 10.1016/j.artint.2012.04.001. [Online]. Available: http://dx.doi.org/10.1016/j.artint.2012.04.001.

[11] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability*, ser. FAIA, vol. 336, IOS Press, 2021, pp. 133–182.

[12] D. Carmine, "Design and implementation of modern cdcl asp solvers," 2014.

[13] F. Calimeri, W. Faber, M. Gebser, *et al.*, "Asp-core-2 input language format," *Theory Pract. Log. Program.*, vol. 20, no. 2, pp. 294–309, 2020. DOI: 10.1017/S1471068419000450. [Online]. Available: https://doi.org/10.1017/S1471068419000450.

[14] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM (JACM)*, vol. 7, no. 3, pp. 201–215, 1960.

[15] J. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the ACM (JACM)*, vol. 12, no. 1, pp. 23–41, 1965. DOI: 10.1145/321250.321253.

[16] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., vol. 185, IOS Press, 2009, pp. 131–153. DOI: 10.3233/978-1-58603-929-5-131. [Online]. Available: https://doi.org/10.3233/978-1-58603-929-5-131.

[17] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "On the implementation of weight constraint rules in conflict-driven ASP solvers," in *ICLP*, ser. LNCS, vol. 5649, Springer, 2009, pp. 250–264.

[18] W. Faber, N. Leone, M. Maratea, and F. Ricca, "Look-back techniques for ASP programs with aggregates," *Fundam. Informaticae*, vol. 107, no. 4, pp. 379–413, 2011.