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

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-the-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 $\neg b$ 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 $I^{\neg\bot}(\ell) := 1$, 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

$$\text{SUM}\{w_1 : \ell_1;\ \cdots;\ w_n : \ell_n\} \geq b \tag{2.2}$$

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 $bnd_\sigma = b$ 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*(Π) and *constraints*(Π) respectively. The sets of rules and constraints define the set of atoms of the program and they are named *atoms*(Π). Finally, I satisfies Π, written $I \models \Pi$, if $I \models r$ for all $r \in$ *rules*(Π) and $I \models \sigma$ for all $\sigma \in$ *constraints*(Π).

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:*

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

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, $\sum_{i=1}^{2} wh_{\sigma_1}(\ell_i) \cdot I_1^{\top}(\ell_i) = \sum 1 \cdot I_1^{\top}(\overline{x}) + 1 \cdot I_1^{\top}(\overline{y}) = 1 \geq bnd_{\sigma_1}$ and $1 \cdot I_1(x) + 2 \cdot I_1^{\top}(y) + 2 \cdot I_1^{\top}(z) = 3 \geq bnd_{\sigma_2}$. Given a program Π and an intepretation I, its *reduct* is defined as follows: $\Pi^I = \{H(r) \leftarrow B^+(r) \mid r \in rules, I \models B(r)\}$, please note that $constraints(\Pi^I) = \emptyset$. 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 $SM(\Pi)$ 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, $I_1 \notin SM(\Pi_{run})$. 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 $I_2 = \{x, z, \overline{y}, \overline{x'}, y', \overline{z'}\} \in SM(\Pi_{run})$.

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 Case

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:

$$\text{SUM}\{1 : \ell_1;\ \cdots\ ;\ 1 : \ell_n\} \geq 1 \tag{2.3}$$

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:

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

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 $\text{AMO}\{\ell_1, \ldots, \ell_n\}$.

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

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* occurrs 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 $(C \setminus \{\ell\}) \cup (D \setminus \{\overline{\ell}\})$. 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 $reason(\ell)$, 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 $H(R) = \{\ell\}$. R is said to be a reason of the literal z if $H(R) = \{z\}$. $J_R = B(R)$ is the assigment satisfying the body of R;

note that J_R is blocking $R \setminus H(R)$. 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 $\ell \in C$, $(C \setminus \{\ell\}) \cap \overline{I} = C \setminus \{\ell\}$. 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 detected 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

```
1  begin
2      I ← [ ]
3      dl ← 0
4      if (Propagate(Π, I) == CONFLICT) then
5          return Unsat
6      while not AllVariablesAssigned(Π, I) do
7          ℓ = PickBranchingLiteral(Π, I)
8          dl ← dl + 1
9          store(I, ℓ, dl)
10         if (Propagate(Π, I) == CONFLICT) then
11             β = ConflictAnalysis(Π, I)
12             if (β < 0) then
13                 return Unsat
14             else
15                 Backjump(Π, I, β)
16                 dl ← β
17     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 $Propagate(\Pi, I)$. 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 $mps_\sigma(I,\ell) = \sum_{j\in[1..n],\ell_j\neq\ell} w_j \cdot I^{\neg\bot}(\ell_j)$; intuitively, if all the undefined literal were true and ℓ was false then the sum would be equal to $mps_\sigma(I,\ell)$. 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 Syntax and Semantics

An *AMOSUM* constraint is defined as follows:

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

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:

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

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*

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

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 $mwh_\sigma(I, g) := \max\{w \cdot I^{\neg\bot}(\ell) \mid (w : \ell\,[g]) \in \sigma\} \cup \{0\}$ 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, other functions are defined:
$mwh_\sigma(g) = \max\{w \mid (w : x[g]) \in \sigma\} \cup \{0\}$;$ml_\sigma(g) = \arg\max_{e \in S} wh(e)$ where $S = \{x \mid (w : x[g]) \in \sigma, w \in \mathbb{N}\}$; $ml_\sigma(I, g) = \arg\max_{e \in S} wh(e) \cdot I^{\neg\bot}(e)$

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*

$$\max\{1 \cdot [\,]^{\neg\bot}(x), 2 \cdot [\,]^{\neg\bot}(y)\} = \max\{1 \cdot 1, 2 \cdot 1, 0\} = 2 < 3$$

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, we discuss the details of the *Propagate* function implemented in the AMOSUM propagator. The *Propagate* function consists of two

phases: *update_phase* and *propagate_phase*. The *update_phase* updates the internal states, specifically the global variable *mps* (*Max Possible Sum*), of the propagator and determines if the *propagate_phase* is necessary. The *propagate_phase* is the actual propagator, which implements the inference rules and related logic, as described in Section 3.2. The *Propagate* function is defined in Algorithm 2. Assume the input to the *Propagate* function is the literal ℓ.

Algorithm 2 Propagate

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

Output: A list of propagated literals

```
1 begin
2     next_phase ← update_phase(σ, I, ℓ);
3     propagated_lits ← ∅;
4     if next_phase = ⊤ then
5         propagated_lits ← propagate_phase(σ, I);
6     return propagated_lits;
```

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 $g \in \mathbb{G}_\sigma$. If $\ell \in \sigma$ it means that it means that ℓ is true and it is in the aggregate set of σ. If ℓ becomes true, accordingly to AMO inference rule, it will contribute to the *mps* (since all other literals will be inferred as false). If ℓ was already contributing to the *mps* (it was the maximum undefined), then the *mps* will not change. In this case, no literal will be inferred, and the next phase will not start. However, if ℓ was not previously contributing to the *mps*, the *mps* will change accordingly. To update the *mps* when a literal becomes true, it is necessary to remove the contribution of the previous literal and add the contribution of ℓ , that is, $mps_\sigma \leftarrow mps_\sigma - mwh_\sigma(I, g) + wh_\sigma(\ell)$. In this case *mps* has changed so the next phase is required. If, instead, $\overline{\ell} \in \sigma$, it means that $\overline{\ell}$ is false and may no longer contribute to the *mps*. To check this, it is sufficient to check that it was contributing and there was no true literal in its group; more precisely: $wh_\sigma(\overline{\ell}) = mwh_\sigma(I \setminus \{\ell\}, g)$ and $lits_\sigma|_{group(\ell)} \cap I = \emptyset$. In this case, the contribution of ℓ has to be removed, since it is false, and the new contribution

$(mwh_\sigma(I, g))$ has to be added to *mps*. In this case *mps* has changed so the next phase is required. When $wh_\sigma(\overline{\ell}) = mwh_\sigma(I \setminus \{\ell\}, g)$ is false, the following can occur: there is no true literal in $lits|_g$ and there is only one undefined literal in $lits|_g$. In this case, according to (3.2), some literal could be inferred. Therefore, regardless of the *mps*, the next phase is required. In all the other cases the next phases is not required.

Algorithm 3 update_phase

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

Output: Boolean to move to the next phase

```
1  begin
2      if ℓ ∈ σ then
3          g ← group_σ(ℓ);
4          if mwh_σ(I, g) = wh_σ(ℓ) then
5              return ⊥
6          mps_σ ← mps_σ − mwh_σ(I, g) + wh_σ(ℓ);
7      else if ℓ̄ ∈ σ then
8          g ← group_σ(ℓ̄);
9          if wh_σ(ℓ̄) = mwh_σ(I \ {ℓ}, g) and lits_σ|_group(ℓ) ∩ I = ∅ then
10             mps_σ ← mps_σ + mwh_σ(I, g) − wh_σ(ℓ);
11         else if |lits_σ|_group_σ(ℓ) \ (I ∪ Ī)| = 1 and lits_σ|_group(ℓ) ∩ I = ∅ then
12             return ⊤
13         else
14             return ⊥
15     else
16         return ⊥
17     return ⊤
```

Now, let's proceed to the *propagation* phase. To infer a literal, we only need to consider its group and the current *mps*. Thus, Algorithm 4 iterates through all groups to derive new literals. The first two blocks implement the *AMO inference rule* and the *Enforced falsity rule*. The *SUM inference rule* must start after the first two since some new literals can be inferred as false,

affecting the number of false literals for that group. To keep track of these false literals, a set named *falses* is necessary. After computing this set, the third block can compute the *SUM inference rule*. Note: If

$$lits_\sigma|_g \setminus (I \cup falses) = \{\ell\}$$

and

$$lits_\sigma|_g \cap I = \emptyset,$$

then $\ell = ml(I, g)$. This is because the maximum undefined literal cannot be inferred as false. If it could have been inferred as false, it would have been inferred as such in a previous iteration and would not have remained the maximum undefined literal. That is, in the SUM inference rule $wh(\ell) = mwh(I, g)$, thus, $mps - wh(\ell) = mps(I, \ell)$ (as defined in 3.2).

Algorithm 4 propagate_phase

Input : A constraint σ, an interpretation I

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

```
1  begin
2      S ← ∅
3      falses ← ∅
4      for g ∈ 𝔾 do
           // AMO inference rule
5          if lits_σ|_g ∩ I = {ℓ} then
6              S ← S ∪ (ℓ̄_i, {ℓ̄_i, ℓ̄})   ∀ℓ_i ∈ lits_σ|_g \ ({ℓ} ∪ I ∪ Ī)
7              continue
           // Enforced falsity rule
8          for ℓ ∈ lits_σ|_g do
9              if ℓ ∉ I ∪ Ī then
10                 if mps_σ − mwh_σ(I, g) + wh_σ(ℓ) < bnd_σ then
11                     rns_ℓ = lits_σ|_g ∪ ⋃_{x∈𝔾_σ\{g}} just_σ(I, x)
12                     S ← (ℓ̄, reason(ℓ))
13                     falses ← falses ∪ {ℓ}
           // SUM inference rule
14         if lits_σ|_g \ (I ∪ falses) = {ℓ} and lits_σ|_g ∩ I = ∅ then
15             if mps_σ − wh_σ(ℓ) < bnd_σ then
16                 S ← S ∪ (ℓ, {ℓ̄} ∪ ⋃_{x∈𝔾_σ\{g}} just_σ(I, x))
17     return S;
```

3.4 Unroll

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 (*mps*) of the propagator when a literal becomes undefined (due to some backjump in the search process). Algorithm 5 implements such procedure. Given a literal ℓ that becomes *undefined*, that is $\ell \notin (I \cup \overline{I})$,

to update the *mps* it is necessary to know its previous value. To have such information a *boolean* variable v is provided in input. If $v = \top$ then ℓ was *true*, otherwise it was *false*. If $\overline{\ell} \in \sigma$, to *'simplify'* the algorithm, we only need to flip v and ℓ. On one hand, if ℓ was true can happend that there is an undefined literal with weight greater then ℓ, i.e., $mwh(I, g) > wh(\ell)$; in this case ℓ will not contribute anymore and the *mps* will increase of $mwh(I, g) - wh(\ell)$. On the other hand, if ℓ was false, it may become the new maximum undefined, and there might be no true literal in $lits|_g$. Formally, if $wh(\ell) > mwh(I \cup \{\overline{\ell}\}, g)$ and $lits|_g \cap I = \emptyset$, then ℓ will contribute to the *mps*, increasing it by $wh(\ell) - mwh(I \cup \{\overline{\ell}\}, g)$.

Algorithm 5 Unroll

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

Output: Updated mps

```
1  begin
2      if ℓ ∈ σ or ℓ̄ ∈ σ then
3          if ℓ̄ ∈ σ then
4              ℓ ← ℓ̄
5              v ← ¬v
6          g ← group(ℓ)
7          if v = ⊤ then
               // ℓ was true
8              if mwh(I, g) > wh(ℓ) then
9                  mps ← mps − wh(ℓ) + mwh(I, g);
10         else if wh(ℓ) > mwh(I ∪ {ℓ̄}, g) and lits|_g ∩ I = ∅ then
               // ℓ was false
11             mps ← mps − mwh(I ∪ {ℓ̄}, g) + wh(ℓ);
```

Chapter 4

Minimizing reason

In Section 4.1, we discuss some important properties of a 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 sections 4.4.1, 4.4.2: one for obtaining the minimal reason and the other for obtaining the cardinality minimal reason, respectively. At the end (section 4.5) the complexity of both algorithms is analized, showing that the first algorithm has a *polynomial* complexity and the second one has a *pseudo-polynomial* complexity. In the last part 4.5.2, the problem of finding a Cardinality Minimal Reason is proved to be *NP-Hard*.

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* $lits\mid_{group(z)} \setminus \overline{I} = \{z\}$ *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 4.1, 4.2 (respectively) 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 $I' = B(R')$; 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,* $\overline{w} \wedge x \rightarrow z$. *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: $reason(z) = \{z, w, y\} = R$ is a reason of z under $I = [\overline{y}, \overline{w}]$. This is true because, given $J_R = B(R) = \{\overline{w}, \overline{y}\}$, $mps(J_R, z) = wh(x) = 1 < 3$ and $lits\ |_{group(z)}\ \setminus \overline{I} = \{z\}$. 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 $lits\ |_{group(z)}\ \setminus \overline{I_y} = \{z\}$, 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 $inc(y) = \max\{mps(J_{R_{\overline{y}}}, z) - mps(J_R, z), 0\} = \max\{2 - 1, 0\} = 1$. 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, $\overline{w} \wedge x \rightarrow z$. As already seen $J_{R_x} = [\overline{x}, \overline{w}]$.
$mps(J_{R_x}, z) = wh(y) = 2 < 3$ and $lits\ |_{group(z)}\ \setminus \overline{I_{\overline{x}}} = \{z\}$, so $reason(z) = \{z, x, w\}$ and $\overline{x}$ is redundant.

4.3 Increment

The increment of ℓ, *w.r.t.* a reason R, written $inc(\ell)$, is used to check if ℓ is redundant in R. Given a reason R of a literal z, the increment of a literal $\ell \in R$ is computed as:

$$inc(\ell) = \max\{mps(J_{R_{\overline{\ell}}}, z) - mps(J_R, z), 0\}$$

There is no need to have an increment possibly negative, given that if $inc(\ell) \leq 0$ then by assumption $mps(J_R, z) + inc(\ell) < bnd$. The increment is bounded below by 0 for another important reason: when a literal ℓ is found to be redundant in R, it will be removed from R.

If $inc(\ell) = mps(J_{R_{\overline{\ell}}}, z) - mps(J_R, z)$ were not bounded, then $mps(J_R, z) + inc(\ell)$ would represent the *mps* where ℓ is flipped. However, the new *mps* needs to represent the value when ℓ is removed. To see this, consider that when a literal is removed from R, the new reason R' will have an interpretation where ℓ is undefined. Thus, by construction, the *mps* will be equal to the maximum value reached when ℓ is false or true. If $mps(J_{R_{\overline{\ell}}}, z) < mps(J_R, z)$ then the 'previous' *mps* is the greatest, that is, the increment of it is 0. This property

of the increment is important for the following algorithms (4.4.1, 4.4.2) that try to remove literals from R.

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
$mps(J_{R_{\overline{\ell}}}, z) = \sum_{g \in \mathbb{G} \setminus \{group(z) \cup group(\ell)\}} mwh(J_{R_{\overline{\ell}}}, g) + wh(\ell)$; given that ℓ will be true and thanks to AMO inference rule it will contribute to the mps. Hence:

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

and

$$inc(\ell) = \max\{wh(\ell) - mwh(J_R, group(\ell)), 0\}$$

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 $\overline{\ell}$; note that since ℓ is false in J_{R_ℓ}, 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

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

By construction $mwh(group(\ell)) - wh(\ell) \geq 0$, then:

$$inc(\ell) = \max\{mwh(group(\ell)) - wh(\ell), 0\} = mwh(group(\ell)) - wh(\ell)$$

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, the algorithm computing the increment is proposed. By construction, the function $inc(\ell) \geq 0$ for all $\ell \in R \setminus lits|_{group(z)}$, since removing a literal from a reason in the worst case does not affect the *mps*.

The following algorithm computes the increment as previously described.

Algorithm 6 inc

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

```
1 . begin
2   if I ⊨ ℓ then
3     return mwh(group(ℓ)) − wh(ℓ)
4   else
5     J_R = complement(R \ {z})
6     return max{wh(ℓ) − mwh(J_R, group(ℓ)), 0}
```

Given a reason R, intepretation I and a set $S \subseteq R$ 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 described in (4.4.1) is designed to obtain the minimal reason. In subsection 4.4.2, instead, a new algorithm to compute the cardinality minimal reason is proposed. For both algorithms a proof of correcntess is provided.

4.4.1 Minimal reason

Given a reason R for a literal ℓ, we say that R is minimal if there does not exist a literal $\ell' \in R$ such that $R \setminus \{\ell'\}$ is still a reason for ℓ. In other words, R is minimal if it does not contain redundant literals.

As discussed previously, each literal ℓ in a reason R has an *increment*, which refers to the increment to the *mps* (Maximal Possible Sum) 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 $S \subseteq R$; S can be removed from R if

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

and

$$S \cap lits|_{group(z)} = \emptyset$$

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 $mps(B(R \setminus S), z) \leq bnd - 1$ 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, threshold s

Paramenters: *inc* function

Output : subset maximal

```
1 begin
2     S ← [ ]
3     ci ← 0
4     for ℓ ∈ R \ lits|_group(z) do
5         if ci + inc(ℓ, I, R \ S) ≤ s then
6             ci ← ci + inc(ℓ, I, R \ S)
7             S ← S ∪ {ℓ}
8     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 $S' = inc(S', I, R) = inc(S, I, R) + inc(\ell, I, R \setminus S) + \sum_{\ell \in S'} inc(x, I, R \setminus S' \cup \{x\})$. By assumption $inc(S, I, R) + inc(\ell, I, R \setminus S) > s$ and by construction $inc(x, I, R \setminus S' \cup \{x\}) \geq 0$ then $inc(S', I, R) > s$. Hence, S' is not a subset of R giving as sum a value less than s.

□

Theorem 1. *The Maximal Subset Sum algorithm returns the maximal subset S where $inc(S, I, R) \leq s$*

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 given that ℓ has not been added it means that $ci + inc(\ell, I, R \setminus S) > s$ for some S. Since, by construction, $S \subseteq S_m$, 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. □

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, $ml_{inc}(S, g) = \arg\max_{k \in S \cap lits|_{group(g)}} inc(k, I, (R \setminus S) \cup \{k\})$.

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, $blw(\ell) = \{k \in lits|_{group(\ell)} \cap R \mid inc(k, I, R) \leq inc(\ell, I, R)\}$. 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

$$L = \{l_1^1, \ldots, l_{m_1}^1, \ldots, l_1^g, \ldots, l_{m_g}^g, \ldots, l_1^k, \ldots, l_{m_k}^k\} \tag{4.3}$$

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.

Algorithm 8 Cardinality Maximum Subset Sum (CMSS)

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

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

Output : S: Set of literals representing the maximum subset

```
1  begin
2      n ← |L|
3      M ← Init(I, R, L, s)
4      for i ∈ {1, ..., s} do
5          for j ∈ {1, ..., n} do
6              ℓ ← ℓ_{j−1}
7              w ← inc(ℓ, I, R)
8              if i ≥ w then
9                  k = arg max_{k∈{0,...,j−1}} ℓ_j ∉ M_{i−w,k}
10                 M_{i,j} ← arg max{|M_{i−w,k} ⟶^{ext} ℓ|, |M_{i,j−1}|}
11             else
12                 M_{i,j} ← M_{i,j−1}
13     S = arg max_{i∈{0,...,s}} |M_{i,n}|
14     return S
```

Algorithm 9 Init

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

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

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 $\mathcal{P}(L) \cup \{\square\}$. $M_{i,j}$ represents the cardinality-maximal set with $sum(M_{i,j}) = i$ and $M_{i,j}$ *'considers'* literal in L_j; considers means that **just** literals in L_j can be active, all the literals in $L \setminus L_j$ can appear in $M_{i,j}$ but are **not** active. If a set S is built considering just literals in $D \subseteq L$ then it is written: $S \sqsubseteq D$. If $M_{i,j} = \square$ it means that such set does not exists. When $M_{i,j} = \square$ then, abusing of notation, $|M_{i,j}| = -1$, $M_{i,j} \xrightarrow{\text{ext}} \ell = \square$ and $\ell \notin M_{i,j}$ for all $\ell \in L$. 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 $M_{i,0} = \square \quad \forall i \in \{1, \ldots, s\}$. After initilization the algorithm 8 constructs each cell $M_{i,j}$, with $1 \leq i \leq s$ and $1 \leq j \leq n$, starting from $M_{i,j-1}$ and $M_{i-w,k}$, where $w = inc(\ell_j, I, R)$ and $k = \arg\max_{k \in \{0,\ldots,j-1\}} \ell_j \notin M_{i-w,k}$. Let $S_{\overline{\ell}} \sqsubseteq L_j$ and $S_\ell \sqsubseteq L_j$ and having $sum(S_{\overline{\ell}}) = sum(S_\ell) = i$, with ℓ_j non-active in $S_{\overline{\ell}}$ and ℓ_j active S_ℓ. By selecting the larger set between these two, we obtain the maximum set 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 $M_{i,j-1} \cap \{\ell\} = \emptyset$. 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 $k = \arg\max_{k \in \{0,\ldots,j-1\}} \ell_j \notin M_{i-w,k}$ yields the cardinality maximum set 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.

This informal discussion has outlined the reasoning behind the algorithm. The following section will provide a formal proof of the concepts discussed so far.

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

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

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

To prove that algorithm *CMSS* 8 returns the cardinality maximal subset S where $sum(S) \leq s$ it is necessary to introduce the relation $C_M \subseteq N_1 \times N_2$, where $N_1 = \{-max_{inc}, \ldots, -1, 0, \ldots, s\}$, $N_2 = \{0, \ldots, n\}$ and $max_{inc} = \max\{inc(\ell, I, R) \mid \ell \in L\}$. The relation C_M is defined as follows:

$$C_M = \{(i,j) \in N_1 \times N_2 \mid i \in \{-max_{inc}, \ldots, -1\}\} \quad \cup$$
$$\{(i,j) \in N_1 \times N_2 \mid M_{i,j} \text{ is largest set s.t. } M_{i,j} \sqsubseteq L_j \text{ and } sum(M_{i,j}) = i\}$$

C_M represents the notion of *'correct'*: $(i,j) \in C_M$ *iff* $M_{i,j}$ is actually the maximum set, considering literals in L_j, giving as sum a value equal to i, for all $i \in \{0, \ldots, s\}$ and $j \in \{0, \ldots, n\}$. The first block $\{(i,j) \in N_1 \times N_2 \mid i \in \{-max_{inc}, \ldots, -1\}\}$ has a *'padding'* purpose, i.e., if the sum is less than 0 it cannot be reached (since function *inc* has been proved to be bounded by 0 below), so it is true by default (this is usefull for the following theorem).

Theorem 2. *$C_M(i,j)$ holds for all $i \in \{0, \ldots, s\}$ and $j \in \{0, \ldots, n\}$*

Proof. (Induction proof)
This proof is a two variable induction proof.
Base case(s):

- $C_M(i,j)$ hold $\forall i \in \{-max_{inc}, \ldots, -1\}$ and $\forall j \in \{0, \ldots, n\}$
- $C_M(0,j)$ holds $\forall j \in \{0, \ldots, n\}$
- $C_M(i,0)$ holds $\forall i \in \{1, \ldots, s\}$

Induction hypothesis:

- if $C_M(i, j-1)$ and $C_M(i-w, k)$ hold then $C_M(i,j)$ holds $\forall i \in \{1, \ldots, s\}$ and $\forall j \in \{1, \ldots, n\}$.
  Where $w = inc(\ell_j, I, R)$ and $k = \arg\max_{k \in \{0,\ldots,j-1\}} \ell_j \notin M_{i-w,k}$.

It is easy to see that, if the base cases and the induction are proved then every $C_M(i,j)$ is *'inferred '* from previous cells $(i, j-1)$ and $(i-w, k)$ $\forall i \in \{1, \ldots, s\}$ and $\forall j \in \{1, \ldots, n\}$; it would mean that $C_M(i,j)$ holds $\forall i \in \{0, \ldots, s\}$ and $\forall j \in \{0, \ldots, n\}$.
Proof base cases

- $C_M(i,j)$ hold $\forall i \in \{-max_{inc}, \ldots, -1\}$ and $\forall j \in \{0, \ldots, n\}$
  - It is true since *by construction* $(i,j) \in C_M$ if $i \in \{-max_{inc}, \ldots, -1\}$.
- $C_M(0,j)$ holds $\forall j \in \{0, \ldots, n\}$
  - *By construction*, $(0,j) \in C_M$ *iff* $M_{i,j}$ is the maximum subset on L_j giving as sum 0. Algorithm *CMSS* (8) builds

    $$M_{0,j} = \{\ell \mid \ell \in L_j \wedge inc(\ell, I, R) = 0\} \quad \forall j \in \{1, \ldots, |L|\}$$

    Then $M_{0,j} \subseteq L_j$ and $M_{i,j}$ is the maximum subset s.t. $sum(M_{i,j}) = 0$, given that to get the maximum set it is enough to add all literals with increment 0. Hence, $(0,j) \in C_M \quad \forall j \in \{0, \ldots, n\}$
- $C_M(i,0)$ holds $\forall i \in \{1, \ldots, s\}$
  - Algorithm *CMSS* builds

    $$M_{i,0} = \square \quad \forall i \in \{1, \ldots, s\}$$

    It is trivial to see that, if $i > 0$ and $j = 0$ then it is impossible to reach sum i with 0 elements, so such subset does not exists. Thus, $M_{i,0}$ *'represent'* the maximum subset. Hence, $(i,0) \in C_M \quad \forall i \in \{1, \ldots, s\}$

Now, let's move to the induction hypothesis.

- If $C_M(i, j-1)$ and $C_M(i-w, k)$ hold then $C_M(i,j)$ holds $\forall i \in \{1, \ldots, s\}$ and $\forall j \in \{1, \ldots, n\}$.
  Where $w = inc(\ell_j, I, R)$.
  - If $C_M(i, j-1)$ holds then *by definition* $M_{i,j-1}$ is the largest set such that $M_{i,j-1} \sqsubseteq L_{j-1}$ and $sum(M_{i,j-1}) = i$. Let $S_{\overline{\ell_j}} \sqsubseteq L_j$ be the largest set with sum i and ℓ_j is **non-active**. Now the claim to get is that $M_{i,j-1} = S_{\overline{\ell_j}}$. Since ℓ_j has to be not active, $S_{\overline{\ell_j}}$ has to be built considering $L_j \setminus \{\ell_j\}$, i.e., $S_{\overline{\ell_j}} \sqsubseteq L_{j-1}$. All Possible Set to be considered are $PS = \{S \mid S \sqsubseteq L_{j-1} \text{ and } sum(S) = i\}$. Given

that, *by induction hypothesis*, $C_M(i, j-1)$ holds, then $|M_{i,j-1}| \geq |S| \quad \forall S \in PS$. So, given that $S_{\overline{\ell_j}}$ is the largest set amoung those in PS then $|M_{i,j-1}| = |S_{\overline{\ell_j}}|$. Thus, we can take $M_{i,j-1}$ as $S_{\overline{\ell_j}}$.

If $C_M(i-w, k)$ holds then *by definition* $M_{i-w,k}$ is the largest set such that $M_{i,k} \sqsubseteq L_k$ and $sum(M_{i,k}) = i-w$, where $w = inc(\ell_j, I, R)$ and $k = \arg\max_{k \in \{0,\ldots,j-1\}} \ell_j \notin M_{i-w,k}$. Let $S_{\ell_j} \sqsubseteq L_j$ be the largest set with sum i and ℓ_j is **active**. Let $M_{i,k} \xrightarrow{\text{ext}} \ell_j = E_{\ell_j}$ be the set obtained after *extending* $M_{i,k}$ with ℓ_j. Let's analize what is the value of $sum(E_{\ell_j})$. By *definition* $sum(M_{i-w,k}) = i - w$. By *construction* $M_{i-w,k} \cap lits|_{group(\ell)} = \emptyset$; as seen in the preliminars it is a sufficient condition to deduce that ℓ_j is active in $M_{i,k} \xrightarrow{\text{ext}} \ell_j$, that is, $\ell_j \in A_{E_{\ell_j}}$. Thus, since $sum(M_{i-w,k}) = i - w$ and $\ell_j \notin M_{i,k}$ and $\ell \in A_{E_{\ell_j}}$ then $sum(M_{i,k} \xrightarrow{\text{ext}} \ell_j) = sum(E_{\ell_j}) = sum(M_{i-w,k}) + inc(\ell_j, I, R) = i-w+w = i$. Now the claim to get is that $E_{\ell_l} = S_{\ell_j}$. This claim is reached reasoning by contradition. Let's assume that E_{ℓ_l} is not the largest set, in other words: there exists a set $S' \sqsubseteq L_j$ such that $|S'| > |E_{\ell_j}|$, moreover, $sum(S') = i$ and $\ell_j \in A_{S'}$ (i.e., ℓ_j is active). In this case $\ell_j \in A_{E_{\ell_j}} \cap A_{S'}$, i.e., it is active in both sets; so by construction $blw(\ell_j) \subseteq E_{\ell_j}$ and $blw(\ell_j) \subseteq S'$, furthermore, there not exists a literal $\ell' \in lits|_{group(\ell_j)}$ such that $wh(\ell') > wh(\ell_j)$ and $\ell' \in E_{\ell_j}$ or $\ell' \in S'$, thus, $|S' \setminus blw(\ell_j)| > |E_{\ell_j} \setminus blw(\ell_j)|$. Given that $E_{\ell_j} = M_{i-w,k} \xrightarrow{\text{ext}} \ell_j = M_{i-w,k} \cup blw(\ell_j, j)$ then $E_{\ell_j} \setminus blw(\ell_j) = (M_{i-w,k} \cup blw(\ell_j)) \setminus blw(\ell_j) = M_{i-w,k}$. Given that $\ell_j \in A_{S'}$ it means that $(S' \cap group(\ell_j)) \setminus blw(\ell_j) = \emptyset$, i.e., there is no literal $\ell' \in S'$ such that $wh(\ell') > \ell_j$. So, $S'' = S' \setminus blw(\ell_j) \sqsubseteq L_d$ where $d \in \{0, \ldots, j-1\}$ such that $L_d \cap group(\ell_j) = \emptyset$. Given that ℓ_j is active in S' then $sum(S'') = sum(S') - inc(\ell_j, I, R) = i - w$. Since $|S''| > |M_{i-w,k}|$ and $S'' \sqsubseteq L_d$ and $M_{i-w,k} \sqsubseteq L_k$, it follows that $d > k$. This is because for S'' to be greater than $M_{i-w,k}$, S'' must include more literals than L_k. Otherwise, $M_{i-w,k}$ would not be the largest set

considering L_k. So $d > k = \arg\max_{k \in \{0,\ldots,j-1\}} \ell_j \notin M_{i-w,k}$. So $k < d \leq j-1$, it means that L_d contains some literal of the group of ℓ_j, and it is a contradiction. Thus, E_{ℓ_l} is the largest set such that $E_{\ell_l} \sqsubseteq L_j$ and $\ell_j \in A_{E_{\ell_j}}$. Hence, we can take $S_{\ell_j} = E_{\ell_j}$. By lemma 2, given that $S_{\overline{\ell_j}}$ is the maximum cardinality set giving i as sum with ℓ_j being an *non-active* literal and S_{ℓ_j} is the maximum cardinality set with ℓ_j being an *active literal* giving i as sum; then $S = \arg\max_{X \in \{S_\ell, S_{\overline{\ell}}\}} |X|$ is a maximum cardinality set that gives as sum s. Given that $M_{i,j} = S$ then $C_M(i,j)$ holds, and the prove is completed.

□

Now the theorem proving the correctness is straightforward.

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

Proof. Thanks theorem 2 $C_M(i,j)$ holds for all $i \in \{0,\ldots,s\}$ and $j \in \{0,\ldots,n\}$. Thus, $M_{i,n}$ is the largest set considering $L_n = L$ literals with sum i. Given that $i \in \{0,\ldots,s\}$, taking the largest set in the last column n leads to the maximum set with sum less or equal to s. □

4.5 Complexity

In this section, we discuss the complexity of computing minimal and cardinality-minimal reasons. Let R be a reason with cardinality $n = |R|$, and let S be a set of redundant literals to be removed from R. The time cost for removing S from R is polynomial in n, specifically $O(n^k)$ for some $k \geq 1$. Therefore, after defining the complexity for computing S, the final complexity must include the additional cost of n^k.

4.5.1 Minimal Reason

To compute S, we use the algorithm *MSS* as described in (7). The *for loop* on line 4 iterates up to the size of R. For each literal, the *increment* and the operation $S \cup \{\ell\}$ are computed, each with a complexity of $O(n)$. Therefore, the final complexity of *MSS* is $O(n^2)$. Consequently, computing the minimal reason has a complexity of $O(n^k + n^2)$, which is polynomial to n.

4.5.2 Cardinality Minimal Reason

In this section, we first analyze the time complexity of algorithm 8. In the second part, we prove that the problem of finding the Cardinality Minimal Reason is *NP-Hard*.

Pseudo-Polynomial

Algorithm 8 begins by calling the *Init* function (9). The *Init* function initializes the matrix M of size $s \times n$, setting the first row and first column, which has a complexity of $O(n+s)$. For each cell in M, the increment and the variable k are computed, each with a complexity of $O(n)$. After computing the matrix, the last column is iterated over with a complexity of $O(s)$. Therefore, the overall complexity is $O(2s+n+s \cdot n^3) = O(n^3 \cdot s)$. Since s is not a constant but part of the input, the algorithm depends on the value of s, making it *pseudo-polynomial*.

review done up to here

NP-Hardness

To prove that problem to find a *Cardinality Maximum Subset Sum (CMSS)* is *NP-Hard* we define a reduction from the well-know NP-Hard problem *0/1 Knapsack Problem* to *CMR*.

Problem Definitions

0/1 Knapsack Problem (KN):

Input:

- A set $K = \{(v_1, w_1), \ldots, (v_n, w_n)\}$ of n of items defined by pairs, each item with weight w_i and value v_i, for $i = 1, \ldots, n$.
- A maximum weight capacity W.

Output:

- A subset $Z \subseteq K$ of items that maximizes the total value while keeping the total weight within W.

Cardinality Maximum Subset Sum (CMSS):
Input:

- A reason R of a literal z under interpretation I.
- A threshold s.

Output:

- A set $S \subseteq R$ such that S is the maximum set having $sum(S) \leq s$

Reduction from 0/1 Knapsack Problem to CMSS

To correctly prove that *KN* can be efficiently reduced to *CMSS*, the following steps are required:

1. Define a polynomial function ρ that maps an instance x of *KN* to an instance $\rho(x)$ of *CMSS*.
2. Define a polynomial function ϕ that maps a solution y of *CMSS* to a solution $\phi(y)$ of *KN*.
3. Prove that, given an optimal solution y for *CMSS* with the input $\rho(x)$, the function $\phi(y)$ produces an optimal solution for *KN* with the original input x.

Function ρ Let $< K, W >$ be the input of *KN* where $K = \{(v_1, w_1), \ldots, (v_n, w_n)\}$ is the set of items and W the bound of the knapsack. $\rho(K, W) =< I, R, s >$ where:

$$s = W$$
$$I = \{\neg w\} \bigcup_{\forall g \in G} \overline{lits|_g}$$
$$R = \{z\} \cup \{w\} \bigcup_{\forall g \in G} lits|_g$$

and regarding parameters *group* and *inc*:

$$group(\ell_i^j) = i \quad \forall i \in \{1, \ldots, n\} \forall j \in \{1, \ldots, v_i\}$$
$$inc(\ell_i^j, I, R) = w_i \quad \forall i \in \{1, \ldots, n\} \forall j \in \{1, \ldots, v_i\}$$

Function ϕ Let S be the output of algorithm *CMSS*, $\phi(S) = Z$ is an output for *KN*, where:

$$f(S) = Z = \{(v_g, w_g) \, : \, lits|_g \subseteq S\}$$

Proving correcntess of reduction To prove the correctness of the reduction, some preliminary definitions are required. Let $value : \mathbb{G} \to \mathbb{N}$ be a function that maps each group to its cardinality, i.e., $value(g) = |lits|_g|$. Let $value : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be a function that maps an item (v, w) to its value, i.e., $value(v, w) = v$. Given a set $S \subseteq R$, a group g is said to be active in S if there exists an element $\ell \in lits|_g$ such that $\ell \in S$. Let $\alpha : \mathcal{P}(R) \to \mathcal{P}(\mathbb{G})$ be a function that maps each subset $S \subseteq R$ to the set of its active groups, i.e., $\alpha(S) = \{g : lits|_g \subseteq S\}$.

Theorem 4. *Given an optimal solution S^* for the CMSS problem with the input $\rho(K, W)$, the function $f(S^*) = Z^*$ produces an optimal solution for the KN problem with the original input $< K, W >$.*

Proof. The problem *CMSS* is to find a set that maximizes its size while keeping its *sum* under the bound s. Formally, this can be expressed as:

$$\max_{S \in \mathcal{P}(R)} f(S) = |S| \tag{4.4}$$

$$\text{subject to} \quad \sum_{e \in A_S} inc(e, I, R) \leq s \tag{4.5}$$

By construction, each group g has all its literals with the same increment. Thus, if any literal of g is active in S, then every literal of g is inside S. This is because, for any $\ell \in A_S$, we have $blw(\ell) = lits|_{group(\ell)}$. Therefore, a set S can be defined solely by its *active groups*, and the objective function in 4.4 can be rewritten as $\sum_{g \in \alpha(S)} | \, lits|_g \, |$. Since $value(g) = | \, lits|_g \, |$, it can also be expressed as $\sum_{g \in \alpha(S)} value(g)$. Abusing notation, let $inc(g, I, R)$ denote the increment of group g when $lits|_g \subseteq S$, i.e., $inc(g, I, R) = \max\{inc(\ell, I, R) \mid \ell \in lits|_g\}$. Since $inc(\ell, I, R) = w_g$ for all $\ell \in lits|_g$, it follows that $inc(g, I, R) = w_g$. Using this notation, the constraint in 4.5 can be rewritten as $\sum_{g \in \alpha(S)} inc(g, I, R) \leq s$. Thus, the maximization problem described above can be reformulated as:

$$\max_{S \in \mathcal{P}(R)} \sum_{g \in \alpha(S)} value(g) \tag{4.6}$$

$$\text{subject to} \quad \sum_{g \in \alpha(S)} inc(g, I, R) \leq s \tag{4.7}$$

S^* is the optimal solution, thus, $|S^*| \geq \max_{S \in \mathcal{P}(R)} \sum_{g \in \alpha(S)} value(g)$
or equivalently $\sum_{g^* \in \alpha(S^*)} value(g^*) \geq \max_{S \in \mathcal{P}(R)} \sum_{g \in \alpha(S)} value(g)$.
By construction $value(g) = (v_g, w_g)$ for any weight w_g. Moreover, notice that, $\phi(S) = \{(v_g, w_g) : lits|_g \subseteq S\}$ and $\alpha(S) = \{g : lits|_g \subseteq S\}$ are made from the same groups. Thus, the objective function can be rewritten as: $\sum_{(v_g, w_g) \in \phi(S)} value(v_g, w_g)$. Furthermore, by construction. $inc(g, I, R) = w_g$ and $s = W$, so, the constraint 4.7 can be written as: $\sum_{(v_g, w_g) \in \phi(S)} w_g \leq W$. Hence, the whole formulation can be defined as:

$$\max_{S \in \mathcal{P}(R)} \sum_{(v_g, w_g) \in \phi(S)} value(v_g, w_g) \tag{4.8}$$

$$\text{subject to} \quad \sum_{(v_g, w_g) \in \phi(S)} w_g \leq W \tag{4.9}$$

Finally, $\sum_{(v_g^*, w_g^*) \in \phi(S^*)} value(v_g^*, w_g^*) \geq \max_{S \in \mathcal{P}(R)} \sum_{(v_g, w_g) \in \phi(S)} value(v_g, w_g)$ and $\sum_{(v_g^*, w_g^*) \in \phi(S^*)} w_g^* \leq W$. Hence, $\phi(S^*)$ is optimal solution of KN with input $< K, W >$. □

Chapter 5

Implementation and Experiments

The propagator outlined in Chapter 3 has been implemented in the ASP solver WASP [19] using its Python interface [20]. This design choice is motivated by the intuitive interface and its seamless integration with the solver through command-line options. In our implementation, AMOSUM constraints are represented by facts, which are interpreted by the Python propagator. Specifically, the representation of an AMOSUM constraint σ of the form (3.1) is the following:

$$\begin{aligned} group(\ell_i, w_i, g_i, \sigma) \leftarrow \qquad & for\ all\ i \in [1..n] \\ lb(b, \sigma) \leftarrow \qquad & \end{aligned}$$

where *group* and *lb* are reserved predicates. Continuing Example 7, σ_4 is represented as follows:

$$\begin{aligned} group(x, 1, 1, \sigma_4) \leftarrow \\ group(y, 2, 1, \sigma_4) \leftarrow \\ group(z, 2, 2, \sigma_4) \leftarrow \\ group(w, 3, 2, \sigma_4) \leftarrow \\ lb(3, \sigma_4) \leftarrow \end{aligned}$$

Moreover, since WASP already supports efficient propagators for handling AMO constraints, the Python propagator focuses on other inference rules,

namely (3.2) and (3.4), as described in Section 3.3. Specifically, the *propagate_phase* function in 4 will not include the initial block for implementing the *AMO inference rule*. The *AMO* constraint in WASP can be defined using a *choice-rule* as follows:

$$\{\ell_i : group(\ell_i) = g\} \leq 1 \leftarrow \qquad for\ all\ g \in \mathbb{G}$$

The implemented system, referred to as AMOWASP, was empirically assessed against the plain version of WASP [19] (version f3e4c56) and the state-of-the-art system CLINGO (version 5.4.0) [21]. All the tested systems used GRINGO (included in the binary of CLINGO) as the grounder.

Experiments were conducted on an Intel Xeon 2.4 GHz server with 16 GB of memory. There are three distinct modalities through which an experiment can be launched:

1. No minimization of the reason (*default*).
2. Minimization of the reason (*min*).
3. Cardinality minimization of the reason (*cmin*).

5.1 Benchmarks

Synthetic Benchmark (SB). Designed to empirically assess the intrisic properties of the AMO-SUM propagator. The first benchmark comprises a program with a rule defining the set of groups $\mathbb{G}$ and another one incorporating a SUM aggregate. The set $\mathbb{G}$ comprises 10 groups of uniform size $group\_size \in \{10, 100, 1000\}$, with the i-th literal of each group having weight i. The bound of the SUM is set to $\alpha \cdot C_1$ (achievable) and $C_1 + \alpha \cdot (C_2 - C_1)$ (unachievable), where: $\alpha \in \{0.15, 0.45, 0.6, 0.9\}$; C_1 is the sum of the maximum weight in each group, i.e., $10 \cdot group\_size$; C_2 is the sum of all weights, i.e., $5 \cdot group\_size \cdot (group\_size + 1)$. Hence, the benchmark comprises a total of 24 instances that are trivial for AMOWASP (they are solved without raising any conflict). In contrast, CLINGO and WASP cannot perform the same inferences and we have

measured the number of conflicts found by the two systems within the first minute of computation.

Graph Coloring (GC). The second benchmark is a modified version of the well-known Graph Coloring problem. In the Graph Coloring problem, the input consists of a graph and a set of colors, and the objective is to assign a color to each node so that connected nodes do not share the same color. Here, colors are also associated with weights, and the sum of weights is required to reach a certain threshold value. Instances are generated from those employed in the ASP competition [22], with the colors red, green, blue, yellow, and cyan associated with the weights 2, 4, 8, 16, and 64 to, respectively. For each instance of n nodes, the threshold is set to $\alpha \cdot 64 \cdot n$, where $\alpha \in \{0.15, 0.45, 0.75\}$. This experiment aims to assess the performance of AMOWASP in comparison to two leading solvers, namely CLINGO and WASP, using real-world benchmarks. Time and memory were limited to 20 minutes and 15 GB, respectively.

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

5.2 Results

SB has been tested only with the *default* modality, whereas **KN** and **GC** have been tested with all three modalities: *default*, *min*, and *cmin*. The experimental results for SB are summarized in Figure 5.1. We can see that when *group_size* is 10 or 100, CLINGO can handle the first three instances in an efficient way, solving them with only a few conflicts. However, this is not true when *group_size* is 1000. This suggests that the way in which CLINGO makes decisions (branching heuristic) works better for instances with SUM aggregates of small to medium size. Moreover, as we expected, the number of conflicts is related to the bound. Specifically, instances with the smallest and largest bounds have fewer conflicts. This happens because these instances are either not constrained enough or overly constrained.

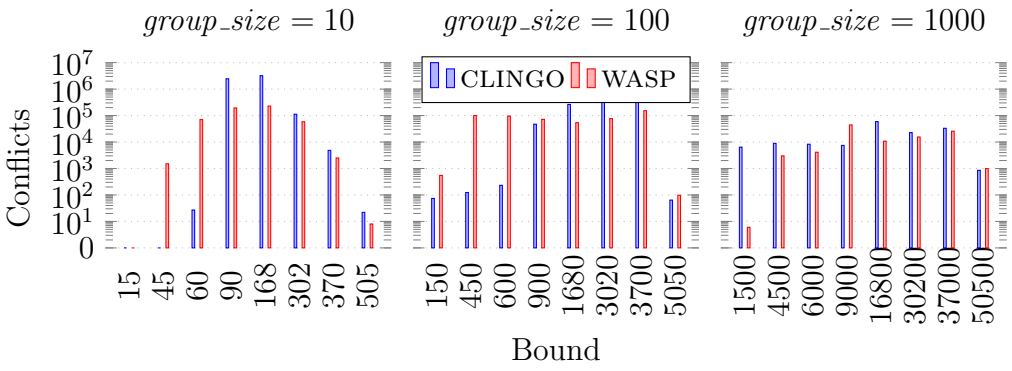

Figure 5.1: Number of conflicts on synthetic AMOSUM constraints comprising 10 groups of varying size and bound. For each *group_size* there are four satisfiable instances (the first four bounds) and four unsatisfiable instances (the last four bounds). AMOWASP is not reported in the plots because it solves all tested instances in this benchmark without raising any conflict.

The results obtained for GC and K are summarized in Figures 5.2- 5.3. As a first observation, CLINGO proves to be more efficient than WASP in both benchmarks. As highlighted in SB, the branching heuristic of CLINGO is more effective than the one of WASP. However, the inferences made by AMOWASP completely fulfil the gap related to the heuristic, and, in fact, AMOWASP achieves the best performance. Regarding GC, AMOWASP successfully solves

72 instances, surpassing CLINGO and WASP, which solve 53 and 18 instances, respectively. We additionally observe that the advantage of AMOWASP is particularly evident in instances with $\alpha = 0.75$, where it outperforms CLINGO and WASP by solving 48 and 60 more instances, respectively. However, for $\alpha = 0.15$ and $\alpha = 0.45$, the Python implementation introduces overhead, leading to comparatively poorer performance. As for K, AMOWASP successfully solves 76 instances, outperforming both CLINGO and WASP, which solve 48 and 12 instances, respectively. We additionally observe that AMOWASP consistently outperforms WASP across all instance categories, solving 17, 25, 13, and 9 more instances of categories T1, T2, T3, and T4, respectively. In comparison to CLINGO, AMOWASP exhibits slightly poorer performance on T1 instances, where CLINGO solves one more instance. However, it demonstrates better performance on the other instances, solving 16, 4, and 9 more T2, T3 and T4 instances, respectively.

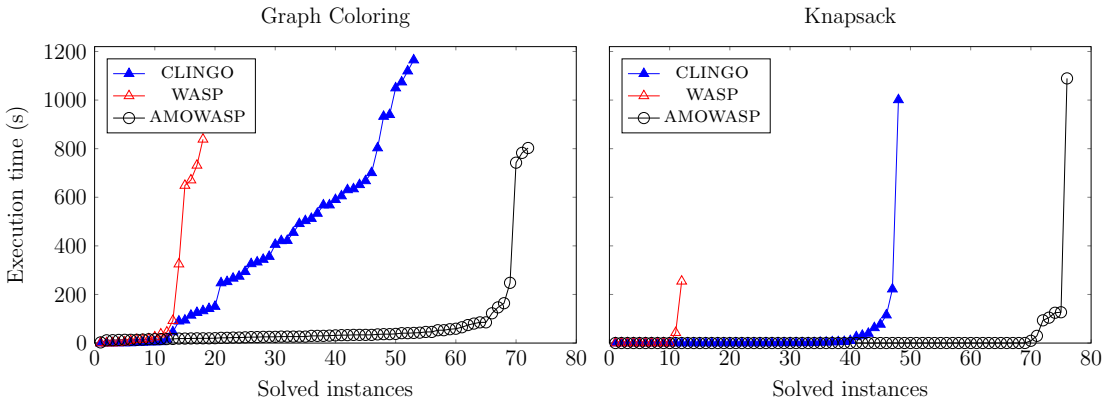

Figure 5.2: Number of solved instances (x-axis) within a time limit (y-axis) for Graph Coloring (left) and Knapsack (right).

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

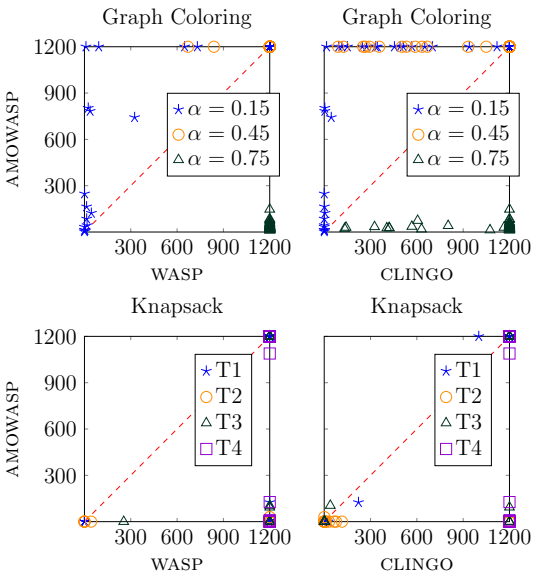

Figure 5.3: Instance-by-instance comparison on the execution time (in seconds) required to solve Graph Coloring and Knapsack. (Timeouts normalized to 1200 seconds.) Points below the red dashed line are instances in which AMOWASP is faster than the compared system.

Chapter 6

Related Work

In this work, AMOSUM constraints have been incorporated into the (propositional) language of ASP. However, AMOSUMs can be integrated into any logic-based formalism that extends propositional logic, such as SMT [23] and CSP [24]. From a computational perspective, there are two primary methods to enhance the capabilities of logic-based languages: the *propagator-based* approach and the *translation-based* approach, as discussed below.

Solvers that utilize the *propagator-based* approach implement specialized algorithms to extend assignments with literals that must be true (Section 2.3). The method used to handle AMOSUM constraints in this work (Section 3.1) falls within this category. In the existing literature, the state-of-the-art system CLINGO [25] employs a hybrid approach for managing programs with aggregates [17]. Aggregates containing a limited number of literals are transformed into regular rules using a translation-based approach, with the threshold for the number of literals being adjustable via the command-line interface. Other aggregates, including those representing AMO constraints, are handled by the propagator described in Section 3.3. A similar approach is also implemented in IDP [26], [27] and WASP [28]. Additionally, both CLINGO and WASP provide external Python-based interfaces for defining custom propagators [20], [29]. The propagator described in Section 3.3 is implemented using the Python interface of WASP.

Translation-based approaches involve compiling aggregates into alterna-

tive constructs. In the context of ASP, the similarities between aggregates and pseudo-Boolean constraints have led to the adoption of certain pseudo-Boolean constraint compilations into clauses [30]. In the context of ASP solvers, many of these translation techniques are integrated into tools like LP2SAT and LP2NORMAL [31], [32]. The former generates CNF formulas, while the latter produces normal rules. Another translation-based method is employed by CMODELS [33]–[35], which translates aggregates into nested logic programs [36]. Finally, for AMO constraints, translation-based approaches provide several encoding options, including pairwise (binomial), binary (bitwise), commander, product, sequential counter, and bimander encodings. A recent comparison of these encodings can be found in [37]. It is worth noting that the AMOSUM propagator introduced in this article can be integrated with AMO constraint compilers by substituting the first inference rule specified in Section 3.2 with the compiled clauses.

Chapter 7

Conclusion

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.

[19] M. Alviano, C. Dodaro, N. Leone, and F. Ricca, "Advances in WASP," in *LPNMR*, ser. LNCS, vol. 9345, Springer, 2015, pp. 40–54. DOI: 10.1007/978-3-319-23264-5_5. [Online]. Available: https://doi.org/10.1007/978-3-319-23264-5%5C_5.

[20] C. Dodaro and F. Ricca, "The external interface for extending WASP," *Theory Pract. Log. Program.*, vol. 20, no. 2, pp. 225–248, 2020. DOI: 10.1017/S1471068418000558. [Online]. Available: https://doi.org/10.1017/S1471068418000558.

[21] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and P. Wanko, "Theory solving made easy with clingo 5," in *Technical Communications of ICLP*, ser. OASIcs, vol. 52, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 2:1–2:15. DOI: 10.4230/OASICS.ICLP.2016.2. [Online]. Available: https://doi.org/10.4230/OASIcs.ICLP.2016.2.

[22] F. Calimeri, M. Gebser, M. Maratea, and F. Ricca, "Design and results of the fifth answer set programming competition," *Artif. Intell.*, vol. 231, pp. 151–181, 2016. DOI: 10.1016/J.ARTINT.2015.09.008. [Online]. Available: https://doi.org/10.1016/j.artint.2015.09.008.

[23] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T)," *J. ACM*, vol. 53, no. 6, pp. 937–977, 2006.

[24] S. C. Brailsford, C. N. Potts, and B. M. Smith, "Constraint satisfaction problems: Algorithms and applications," *Eur. J. Oper. Res.*, vol. 119, no. 3, pp. 557–581, 1999.

[25] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Multi-shot ASP solving with clingo," *Theory Pract. Log. Program.*, vol. 19, no. 1, pp. 27–82, 2019.

[26] M. Denecker and B. De Cat, "Dpll(agg): An efficient smt module for aggregates," in *Logic and Search, Edinburgh, 15 July 2010*, 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:126135475.

[27] B. Bogaerts, J. Jansen, B. D. Cat, G. Janssens, M. Bruynooghe, and M. Denecker, "Bootstrapping inference in the IDP knowledge base system," *New Gener. Comput.*, vol. 34, no. 3, pp. 193–220, 2016.

[28] M. Alviano, C. Dodaro, and M. Maratea, "Shared aggregate sets in answer set programming," *Theory Pract. Log. Program.*, vol. 18, no. 3-4, pp. 301–318, 2018.

[29] P. Cabalar, J. Fandinno, T. Schaub, and P. Wanko, "On the semantics of hybrid ASP systems based on clingo," *Algorithms*, vol. 16, no. 4, p. 185, 2023.

[30] A. Aavani, D. G. Mitchell, and E. Ternovska, "New encoding for translating pseudo-boolean constraints into SAT," in *SARA*, AAAI, 2013.

[31] J. Bomanson, M. Gebser, and T. Janhunen, "Improving the normalization of weight rules in answer set programs," in *JELIA*, ser. LNCS, vol. 8761, Springer, 2014, pp. 166–180. DOI: 10.1007/978-3-319-11558-0_12. [Online]. Available: https://doi.org/10.1007/978-3-319-11558-0%5C_12.

[32] J. Bomanson and T. Janhunen, "Normalizing cardinality rules using merging and sorting constructions," in *LPNMR*, ser. LNCS, vol. 8148, Springer, 2013, pp. 187–199. DOI: 10.1007/978-3-642-40564-8_19. [Online]. Available: https://doi.org/10.1007/978-3-642-40564-8%5C_19.

[33] Y. Lierler and M. Maratea, "Cmodels-2: Sat-based answer set solver enhanced to non-tight programs," in *LPNMR*, ser. LNCS, vol. 2923, Springer, 2004, pp. 346–350. DOI: 10.1007/978-3-540-24609-1_32. [Online]. Available: https://doi.org/10.1007/978-3-540-24609-1%5C_32.

[34] E. Giunchiglia, Y. Lierler, and M. Maratea, "Answer set programming based on propositional satisfiability," *J. Autom. Reason.*, vol. 36, no. 4, pp. 345–377, 2006. DOI: 10.1007/S10817-006-9033-2. [Online]. Available: https://doi.org/10.1007/s10817-006-9033-2.

[35] E. Giunchiglia, N. Leone, and M. Maratea, "On the relation among answer set solvers," *Ann. Math. Artif. Intell.*, vol. 53, no. 1-4, pp. 169–204, 2008. DOI: 10.1007/S10472-009-9113-1. [Online]. Available: https://doi.org/10.1007/s10472-009-9113-1.

[36] P. Ferraris and V. Lifschitz, "Weight constraints as nested expressions," *Theory Pract. Log. Program.*, vol. 5, no. 1-2, pp. 45–74, 2005. DOI: 10.1017/S1471068403001923. [Online]. Available: https://doi.org/10.1017/S1471068403001923.

[37] V. Nguyen, V. Nguyen, K. Kim, and P. Barahona, "Empirical study on sat-encodings of the at-most-one constraint," in *SMA*, ACM, 2020, pp. 470–475. DOI: 10.1145/3426020.3426170. [Online]. Available: https://doi.org/10.1145/3426020.3426170.