Compilation of Aggregates in ASP: Preliminary Results *

Giuseppe Mazzotta, Bernardo Cuteri, Carmine Dodaro, and Francesco Ricca

University of Calabria, Italy

Abstract. Answer Set Programming (ASP) is a well-known problemsolving formalism in computational logic. Among the knowledge modeling constructs that make ASP effective in representing complex problems are aggregates. Aggregates operate on sets of literals and compute a single value (e.g., count, sum, etc.), thus, making the expression of constraints in ASP programs very concise. Traditionally, ASP systems are based on the ground&solve approach that suffers an intrinsic limitation known as grounding bottleneck: the grounding (variable elimination) can fill all the available memory and then the program cannot be evaluated. This happens also in programs that use aggregate functions. Recently, an alternative approach to evaluate ASP programs that avoids the grounding bottleneck has been proposed that is based on ASP program compilation. In this paper we present an extension of ASP program compilation that supports constraints containing the aggregate *count*. Preliminary experimental results demonstrate the viability of the approach.

Keywords: Answer Set Programming \cdot Aggregates \cdot Grounding Bottleneck.

1 Introduction

Answer Set Programming (ASP) [7] is a well-known problem-solving formalism in computational logic that is based on the stable model semantics [24]. ASP systems, such as CLINGO [20] and DLV [1, 3], made possible the development of many real-world applications. In the recent years, ASP has been widely used for solving problems of game theory [5], natural language processing [28], natural language understanding [13], robotics [18], scheduling [16], and more [17].

A key role in the development of applications is played by system performance, and thus, the improvement of ASP systems is an interesting research topic in computational logic. Traditional ASP systems are based on the ground & solve approach [25], in which a *grounder* module transforms the input program (containing variables) in its propositional counterpart, whose stable models are subsequently computed by the *solver* module, which implements an extension of the Conflict Driven Clause Learning (CDCL) algorithm [25]. The traditional

^{*} Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

ASP implementations are effective in many contexts [17] but suffer from an intrinsic limitation: the combinatorial blowup of the grounding of some rules, known as *grounding bottleneck*. The grounding bottleneck is often due to one or few constraints that model the (non) admissibility of problem solutions [27, 9].

Several attempts have been made to solve the grounding bottleneck problem [22], including language extensions (such as constraint programming [27, 6], and difference logic [20, 29]) and *lazy grounding* techniques [14, 26, 30]. Hybrid formalisms are efficiently evaluated by coupling an ASP system with a solver for the other theory, thus circumventing the grounding bottleneck. Lazy grounding implementations instantiate a rule only when its body is satisfied to prevent the grounding of rules which are unnecessary during the search of an answer set. Albeit lazy grounding techniques obtained good preliminary results, their performance is still not competitive with state-of-the-art systems [22].

Recently a different approach was proposed that is based on the *compilation* of problematic constraints as propagators [10, 11]. In this latter approach, problematic constraints are removed from the non-ground input program and the resulting program is provided as input to an extended version of a CDCL-based solver, where the presence of problematic constraints is internally simulated. There are two alternative strategies for implementing such an extension, namely lazy instantiation and propagators. In the lazy instantiation strategy, the solver computes a stable model of the program without problematic constraints. If this stable model satisfies also the omitted constraints, then it is also a stable model of the original program. Otherwise, the violated instances of these constraints are lazily instantiated, and the search continues. The other strategy relies on an extension of the propagation function by adding custom *propagators*, whose role is to perform the inferences of missing constraints during the search. Basically, Cuteri et al. [11] proposed to translate (or compile) some non-ground constraints into a dedicated C++ procedure, which is used by the system to generate propagators in an automatic way. This approach keeps the declarativity of ASP and is effective when the problematic constraints are likely to be satisfied by a candidate model (i.e., whenever lazy instantiation is effective cfr. [10]). Approaches based on compilation revealed to be very promising, outperforming traditional systems in many comparisons [11,12] However, a significant number of problems, especially hard combinatorial problems from ASP competitions [9] exploit an advanced language feature that is not supported: aggregates [19]. Aggregates are among the standardized knowledge modeling constructs that make ASP effective in representing complex problems [8, 23]. Indeed, aggregates operate on sets of literals and compute a single value (e.g., count, sum, etc.), thus, making the expression of constraints in ASP programs very concise.

In this paper, we push forward the idea of [10-12], and we present a novel strategy for translating (compiling) non-ground constraints containing #count aggregates into dedicated C++ procedures that are used as propagators during the search of the CDCL algorithm. We have implemented our extension on top of WASP [4, 2, 15] and conducted an experimental analysis on hard benchmarks from

ASP competitions [9,23]. Results are encouraging, indeed our implementation improves the performance of WASP in all tested scenarios.

2 Preliminaries

2.1 Answer Set Programming

An ASP program π is a set of rules of the form:

$$h_1 | \dots | h_n : -b_1, \dots, b_m$$
 with $n + m > 0$

where $h_1| \ldots |h_n|$ is a disjunction of atoms and is referred to as *head*, instead, b_1, \ldots, b_m is a conjunction of literals and is referred to as *body*. In particular, if n = 0 the rule is called *constraint*, instead if m = 0 the rule is called *fact*.

An atom a is an expression of the form $p(t_1, \ldots, t_k)$ where p is a predicate of arity k and t_1, \ldots, t_k are *terms*. A term is an alphanumeric string that could be either a variable or a constant. According to Prolog, if a term starts with a capital letter is a variable otherwise is constant. If $\forall i \in \{1, \ldots, k\}$, t_i is a constant the atom a is said ground. A literal is an atom a or its negation $\sim a$ where ~ denotes the negation as failure. Given a literal l it is said positive if l = a, *negative* if $l = \sim a$. Given a positive literal l = a, we define the *complement*, $\overline{l} = a$, instead, for a negative literal l = a, $\overline{l} = \overline{a} = a$. However ASP supports also aggregate atoms. An aggregate atom is of the form $f(S) \succ T$, where f(S) is an aggregate function, $\succ \in \{=, <, \leq, >, \geq\}$ is a predefined comparison operator, and T is a term referred to as guard. An aggregate function is of the form f(S), where S is a set term and $f \in \{\#count, \#sum\}$ is an aggregate function symbol. A set term S is a pair that is either a symbolic set or a ground set. A symbolic set is a pair $\{Terms : Conj\}$, where Terms is a list of variables and *Conj* is a conjunction of standard atoms, that is, Conj is of the form a_1, \ldots, a_n and each $a_i (1 \le i \le n)$ is an atom. A ground set, instead, is a set of pairs of the form $(\bar{t}: conj)$, where \bar{t} is a list of constants and conj is a conjunction of ground atoms. Given a program π , we define U_{π} , the Herbrand Universe, as the set of all constants appearing in π and B_{π} , the Herbrand Base, as the set of all possible ground atoms that can be built using predicate in π and constants in U_{π} . \mathcal{B} denotes $B_{\pi} \cup \overline{B_{\pi}}$. Given a rule r and the Herbrand Universe U_{π} , we define ground(r) as the set of all possible instantiations of r that can be built assigning variables in r to constant in U_{π} . Given a program π , instead, $ground(\pi) = \bigcup_{r \in \pi} ground(r)$. An interpretation I is a set of literals. In particular, I is total if $\forall a \in B_{\pi}(a \in I \lor \sim a \in I) \land (a \in I \to \sim a \notin I)$. A literal l is true w.r.t I if $l \in I$, otherwise it is false. A ground conjunction conj of atoms is true w.r.t I if all atoms in conj are true, otherwise, if at least one atom is false then conj is false w.r.t. I. Let I(S) denote the multiset $[t_1|(t_1,\ldots,t_n): conj \in S \land conj$ is true w.r.t. I]. The evaluation I(f(S))of an aggregate function f(S) w.r.t. I is the result of the application of f on I(S).

For example let A be an aggregate atom $A = #count\{(1:p(1,1)), (2:p(2,1)), (3:p(2,1)), (3$

p(3,1) > 1 and let $I = \{p(1,1), p(2,1), \sim p(3,1)\}$. I(S) = [1,2], I(f(S)) = 2 since f = #count so the aggregate atom A is true w.r.t. I.

I is a model for π if $\forall r \in ground(\pi)(\forall l \in body(r), l \in I) \rightarrow (\exists a \in head(r) : a \in I)$. Given a program π and an interpretation I, we define the FLP - reduct of π , denoted by π^I , as the set of rules obtained from π deleting those rules that has body false w.r.t I. Let I be a model for π , I is also a stable model for π if $\not\exists I' \subset I$ such that I' is a model for π^I . Given a program π , π is coherent if it admits at least one stable model otherwise is incoherent.

2.2 Classical CDCL Evaluation

4

The solving approach for ASP is implemented as *conflict-driven clause learning* for SAT but with ad-hoc extensions for ASP that ensure that the model built is also stable. In particular, the idea behind this approach starts from an empty interpretation I and step-by-step will add to I all the deterministic consequences starting from true literals in I. Once all consequences are inferred if the interpretation is not total we will choose heuristically some literals and propagate their deterministic consequences. If we reach the empty clause we come back (*conflict resolution*) to the last non-deterministic choice and will propagate it deterministically and go forward until I is total and then we have a model or we have no choice and then the program is *incoherent*. To better understand, let suppose we have the following ground program π :

a(1). r_0 : r_1 : a(2).a(3)|a(4). r_2 : r_3 : b(1,1)|nB(1,1).b(1,2)|nB(1,2)|. r_4 : r_5 : b(2,2)|nB(2,2)|. b(2,1)|nB(2,1)|. $r_{6}:$ $: -a(1), #count\{1: b(1,1), 2: b(1,2)\} > 1$ $r_7:$ $: -a(2), #count\{1: b(2,2), 2: b(2,1)\} > 1$ r_8 : (-b(1,1), b(2,1), a(3)). r_9 :

The algorithm starts with $I = \emptyset$. Then a(2) is inferred from r_1 , and a(1) is inferred from r_0 . Let suppose that from r_3 is heuristically chosen b(1,1). Let consider its deterministic consequences:

- From r_3 is inferred ~ nB(1,1) since, by property of stable models, this is the only way to support b(1,1)
- From r_7 is inferred ~ b(1,2) since it is the unique way to satisfy r_7
- From r_4 is inferred nB(1,2) because it is the unique way to satisfy r_4 since $\sim b(1,2)$ has been previously added to I

At this point let suppose that heuristically is chosen b(2, 1). As consequences we have:

- From r_6 is inferred ~ nB(2, 1) to ensure that the model will be also a stable model

- From r_8 is inferred ~ b(2,2) since it is the unique way to satisfy r_8
- From r_5 is inferred nB(2,2) because it is the unique way to satisfy r_5 since b(2,2) has been previously added to I

Let suppose that from r_2 is chosen a(3). At this point, the algorithm propagates $\sim a(4)$ and find a conflict in r_9 . Since a conflict is detected last deterministic consequences are unfolded until the last choice and propagate $\sim a(3)$ and its consequences, in this case only a(4) from r_3 . Now, I is total so we have found a stable model. Thanks to the external interface of WASP, we can customize the CDCL evaluation by defining *propagators*, which are procedures intended to compute the deterministic consequences of a true literal. In particular, in our approach, we take as input an ASP constraint, possibly containing an aggregate, and create a propagator that computes the deterministic consequences of a true literal w.r.t. the ASP constraint in input.

2.3Normalization Procedure

In order to simplify the compilation of constraint containing an aggregate literal we normalize the aggregate literals, without losing generality, in such a way that the different comparison operators are unified to one of them that is \geq operator. Let C be a constraint with an aggregate A of the form $count(S) \succ g$ with $\succ \in \{<, \leq, >, \geq, =\}$, the normalization procedure is the following:

- If \succ equal to \geq then A remains as it is;
- If \succ equal to > then A will be mapped into $\#count(S) \ge g+1$;
- If \succ equal to < then A will be mapped into $\sim \#count(S) \ge g$;
- If \succ equal to \leq then A will be mapped into $\sim \#count(S) \geq g+1$;
- If \succ equal to = then A will be mapped into the conjunction of two aggregate literals:
 - $A_1 = \#count(S) \ge g;$
- $A_2 = \stackrel{\sim}{\sim} \#count(S) \ge g + 1.$ If \succ equal to = and A is negated then C will be mapped in two different constraints as follow: • $C_1 =: -body(C), \#count(S) \ge g + 1;$ • $C_2 =: -body(C), \sim \#count(S) \ge g.$

For example given constraint $C =: -\#count\{X : a(X)\} = 3$, it will be normalized as follow:

$$C' =: -\#count\{X : a(X)\} \ge 3, \sim \#count\{X : a(X)\} \ge 4$$

Note that this normalization is often used in real implementations [21].

3 Constraints with aggregates as propagators

In this section, we present our strategy for evaluating constraints with aggregates using propagators that are automatically generated by a compilation-based approach. Hereafter, to simplify the presentation, we assume w.l.o.g. that the bodies of constraints never contain two literals with the same predicate name and constraints contain at most one aggregate literal in the body.

Some notation and code conventions. Let C be a constraint with an aggregate of the form:

$$= -b_1, \dots, b_k, \#count\{V_1, \dots, V_n : l_1, \dots, l_m\} \ge g.$$
where g is a constant.

we define :

- Given an ASP expression (term, literal, body, rule, etc.) e, vars(e) as the set of variable terms appearing in e;
- Given an ASP expression (term, literal, body, rule, etc.) e, pred(e) as the set of all predicate name appearing in e;
- Given a literal l, trm(l) as the list of terms appearing in l;
- -trm(l)[i] with $1 \le i \le |trm(l)|$ as the i th element of the list trm(l) (e.g. let trm(l) = [X, Y, 3], trm(l)[2] is equal to Y);
- $-\sigma: vars(C) \to U_{\pi}$ as a possible variable assignment from the variables of the constraint to the constants in U_{π} ;
- Given an ASP expression $e, \sigma(e)$ as the substitution of the variables appearing in e with the constants which their are mapped to;
- $match(l_1, l_2)$, where l_1 is a literal and l_2 is a ground literal, as a function that checks if $\exists \sigma \mid \sigma(l_1) = l_2$;
- body(C) as the set $\{b_1, ..., b_k\};$
- aggregate(C) as the aggregate literal in the constraint C;
- aggregateVars(C) as the set of variables $\{V_1, ..., V_n\}$;
- aggrLit(C) as the set $\{l_1, ..., l_m\};$
- sharedVars(C) as the set of variables v such that $v \in vars(body(C)) \cap vars(aggrLit(C));$
- Given an interpretation I, *joinTuples* as the set of all tuples of the form (t_1, \ldots, t_m) such that following conditions are true:
 - $(t_i \in I^+ \lor (t_i \in (\mathcal{B} \setminus I)^+ \land \overline{t_i} \notin I)) \forall i \in \{1, \ldots, m\}$ where I^+ denotes the set of positive literals in I.
 - $\exists \sigma \mid \forall i \in \{1, \ldots, m\} match(\sigma(l_i), t_i)$ where σ is a possible variable assignment such that $\forall v \in vars(aggrLit(C)), \sigma(v)$ is defined.
- Given an interpretation I and a join tuple $(t_1, \ldots, t_m) \in joinTuples$ $projOnSharedVar((t_1, \ldots, t_m)) = (trm(t_i)[j] : trm(l_i)[j] \in sharedVars(C)$ with $1 \leq j \leq trm(t_i)$ and $1 \leq i \leq m$;
- Given an interpretation I and a join tuple $(t_1, \ldots, t_m) \in joinTuples$ $projOnAggrVar((t_1, \ldots, t_m)) = (trm(t_i)[j] : trm(l_i)[j] \in aggregateVars(C)$ with $1 \leq j \leq trm(t_i)$ and $1 \leq i \leq m$).

In the algorithms that we present in this section and in the appendix we follow the same pseudo-code convention that is used in [22] to ease readability. In particular, the underlined code is produced by the compiler, instead, the not underlined one (e.g., variables and references) represents the code in the scope of the compiler. If a line contains an underlined part closed between «», it means that the code inside will be first interpreted by the compiler (e.g. variables are substituted by their run-time value) and then is printed in the propagator code. For example, given the constraint : $-a(X), c(X, Z), \#count\{Y : b(X, Y)\} > 2$ then Algorithm 1 at line 6, prints "case a" and "case c".

Algorithm 1 CompilePropagateConstraintWithAggregate

	0 1 1 0 00 0
	Input : A constraint C
	Output : Prints the propagator for C .
1	begin
2	$I_l = \emptyset$
3	$\overline{buildAggregateJoin(C)}$
4	$\mathbf{switch} \ \mathrm{pred}(\mathbf{l})$
5	forall the $c \in body(C)$ do
6	case «pred(c)»:
7	$\overline{\text{CompilePropagateConstraintStartingFromLiteral}(c,C)}$
8	break
9	if $pred(l) \in $ «pred(aggrLit(C))» then
10	$\overline{\text{CompilePropagateConstraintStartingFromAggregate}(C)}$
11	$\underline{\mathbf{return}} \ I_l$

Algorithm 2 CompilePropagateConstraintStartingFromLiteral

```
Input : A constraint C, a literal c \in C
     \mathbf{Output}: Prints an algorithm that is performs the unit propagation of C
                  starting from a ground literal whose predicate is the same of \boldsymbol{c}
 1 \underline{\sigma = \epsilon}
 2 forall the k = 1, \ldots, |trm(c)| do
          if trm(c)/k is variable then
 3
                \sigma = \sigma \cup \{ \operatorname{{\ll}trm}(\mathbf{c})[\mathbf{k}] \mathbb{w} \mapsto trm(l)[\operatorname{{\ll}} \mathbb{w}] \}
 \mathbf{4}
 5 B = printNestedJoinLoop(C, c)
 6 propagateUndefined(B,C)
 7 forall the i = |B|, \ldots, 1 do
 8
              \sigma = \sigma_{\langle\!\langle i \rangle\!\rangle}
 9
              if u = b_{\langle i \rangle} then
                 u = \bot
10
11
          }
```

Compilation procedures. Given a constraint C, Algorithm 1 prints the propagation procedure for C starting from a true literal l. It starts declaring an empty implication list I_l , which will be in charge of accumulating the result of the propagation of a literal l, and then prints the code that builds the set of *join* tuples by executing Algorithm buildAggregateJoin. This algorithm declares different sets that store join tuples, and are used to evaluate the truth value of the aggregate literal (pseudo-code and more detailed description in the appendix, algorithm 6). At this point the propagation procedure, as is shown in Algorithm 1 (lines 4-8), continues with a switch on the predicate name of the literal l. In particular, this switch block has a case for each literal c belonging to body(C) in which is printed, by executing Algorithm 2, the code that evaluates first all body literal starting from c and at the end the aggregate literal. Instead, if the pred(l)belongs to the set of predicate name that appears inside the aggregate literal the evaluation of C starting from the aggregate literal is printed by executing Algorithm 5. Algorithm 2 starts printing the code that builds a variable substitution σ that maps all variables belonging to c to constant in l (Algorithm 2 lines 1-4). Then, executing algorithm 7, different nested join loops that iterate over all possible ground instantiations of body literals are printed (detailed description and pseudo-code algorithm in appendix Algorithm 7). Once all nested blocks are printed, we can evaluate the aggregate literal and make some inferences. In particular, by executing Algorithm 3, the code for unit propagation is printed. It starts declaring a list of values "sharedVarTuple" that contains values to which shared variables are mapped, in order to consider only those join tuples that match the value of shared variables (lines 2-6). Then we declare the reason R in order to accumulate all true literals that are causing propagation (Algorithm 3 lines 7-13). Once the reason is built, if $u \neq \perp$ then we have exactly one undefined body literal that can be propagated if the aggregate is true (Algorithm 3 lines 14-19). Otherwise $u = \bot$ then all body literals are true and so if the propagation condition for the aggregate literal is true (Algorithm 3 lines 20-24) we could infer something on the aggregate literal. Since the aggregate literal in our example is positive, if the size of *truekeys* is exactly g-1 then we can propagate as false those join tuples having exactly one literal l_i with $i \in \{1, \ldots, m\}$ undefined, that is $l_i \in (\mathcal{B} \setminus I)^+ \wedge \overline{l_i} \notin I$ (algorithm 4 lines 6-7). In this way, we ensure that we do not reach the upper bound q and so the constraint is not violated. In the end, we need to roll back the aggregate join structures by executing algorithm Algorithm 5, instead, prints the code that executes a constraint evaluation 8. starting from the aggregate. In particular, the propagator iterates over possible variables assignment for variables belonging to sharedVars(C) line 3. For each variable assignment, first of all, we should update the previously declared aggregate join structures in order to discard those join tuples that do not share the values of the shared variables by executing *algorithm* δ and then check if the aggregate literal is true (lines 9-12). If the aggregate is true the propagator has to build body joins, and so, as is shown in Algorithm 2, nested join loops are printed by executing algorithm 7. Once the last join loop is reached, if there is a body literal undefined (algorithm 5 lines 26-27) it will be propagated and

Algorithm 3 propagateUndefined **Input** : List of body literals B, and a constraint COutput: Prints code for unit propagation. 1 A = aggrLit(C)**2** sharedVarTuple = (3 forall the $v \in sharedVars(C)$ do $\sigma(\langle v \rangle)$ 4 5) 6 updateAggregateWithSharedVars(C, true) 7 $R = \{l\}$ 8 forall the $i = 1, \ldots, |B|$ do 9 $R = R \cup \{b_{\langle\!\langle i \rangle\!\rangle}\}$ 10 $R = R \setminus \{u\}$ 11 for $(l_1, \ldots, l_{|\langle \langle A \rangle \rangle|}) \in trueJoin$ for $j = 1, \ldots, | \ll A \gg |$ $\mathbf{12}$ $R = R \cup \{l_j\}$ $\mathbf{13}$ 14 if $u \neq \perp$ then { 15 if aggregate(C) is positive then if $|trueKey| \ge g$ then $\mathbf{16}$ 17 else if $|trueKey \cup undefKey| < g$ then $\mathbf{18}$ $I_l = I_l \cup (\overline{u}, R)$ $\mathbf{19}$ **20** if aggregate(C) is positive then else if |trueKey| = g - 1 $\mathbf{21}$ 22 else $\mathbf{23}$ $| \} else if |trueKey \cup undefKey| = g$ **24** propAggregate(C)**25** updateAggregateWithSharedVars(C, false)

Algorithm 4 propAggregate

	Input : A constraint C
	Output: Prints code to propagate aggregate atom
1	A = aggrLit(C)
2	for $(l_1, \ldots, l_{ \langle \mathbf{A} \rangle }) \in undefJoin$
3	if $projAggrVars((l_1, \ldots, l_{ \ll \mathbf{A}})) \notin trueKey$ then
4	forall the $i = 1, \dots, A $ do
5	if $aggregate(C)$ is positive then
6	$ \qquad \qquad$
7	$\boxed{I_l = I_l \cup (\overline{l_{\langle\!\langle \mathbf{i} \rangle\!\rangle}}, R)}$
8	else
9	
10	$I_l = I_l \cup (l_{\langle\!\langle \mathbf{i} \rangle\!\rangle}, R)$

then we can close each join loop. Now what we need is an else if block where the propagator enters if the aggregate literal is false and the propagation condition is true (algorithm 5 lines 36-40). In this else-if block, since we want to make inferences on the aggregate, we need to verify that the body without aggregate is true. In order to do this, we need nested join loops, which are printed again by executing algorithm 7, to build possible body join. In the last join loop, if all body literals are true (line 29-30) then the propagator makes inferences on the aggregate atom by executing algorithm 4. In the end, before passing to the next shared variables values, aggregate join structures are restored by executing again algorithm 8. Note that both algorithm 2 and 5 remain as it is for constraints with one aggregate literal but as we describe in section 2.3 there is one special case in which the aggregate literal is transformed into two aggregate literals and their conjunction is equivalent to the original aggregate literal.

The case aggregates with equality guard. The algorithms presented above can be easily updated to handle aggregates of the form $A = #count\{V_1, \ldots, V_n : l_1, \ldots, l_m\} = g$. Indeed, during normalization this aggregate literal is transformed in

$$: -A_1, A_2$$

where

$$A_1 = \#count\{V_1, \dots, V_n : l_1, \dots, l_m\} \ge g,$$

$$A_2 = not \ \#count\{V_1, \dots, V_n : l_1, \dots, l_m\} \ge g + 1$$

Thus, to support this form of aggregate literal, Algorithm 2 changes as follows. The code remains identical for processing A_1 but the following lines have to be modified to compile A_2 :

- At line 18 and 24 we will print an if block to ensure that also A_2 is true.

Algorithm 5 CompilePropagateConstraintStartingFromAggregate

Input : A constraint C**Output**: Prints an algorithm that performs the unit propagation of C starting from a ground literal whose predicate is the same of c1 A = aggrLit(C)2 $\sigma = \epsilon$ **3** for $sharedVarTuple \in sharedVarKey {$ forall the $i \in \{1, \ldots, |sharedVars(C)|\}$ do 4 $\sigma = \sigma \cup (\text{``sharedVars(C)[i]`}, sharedVarTuple[``is`])$ $\mathbf{5}$ $6 \ updateAggregateWithSharedVars(C, true)$ 7 propAqqr = False $\mathbf{s} \ propBodyLit = False$ 9 if aggregate(C) is positive then | if $|trueKey| \ge g$ then{ 10 else 11 if $|trueKey \cup undefKey| < g$ then { $\mathbf{12}$ while $propAggr = False \lor propBodyLit = False$ do $\mathbf{13}$ if propBodyLit = True then 14 propAggr = True15propBodyLit = True16 $\mathbf{17}$ B = printNestedJoinLoop(C, None) $R = \{l\}$ 18 for all the $i = 1, \ldots, |B|$ do 19 $R = R \cup \{b_{\langle i \rangle}\}$ $\mathbf{20}$ $_R = R \setminus \{u\}$ $\mathbf{21}$ for $(l_1, \ldots, l_{|\langle \langle A \rangle \rangle|}) \in trueJoin$ $\mathbf{22}$ for j = 1, ..., | « A » | $\mathbf{23}$ $R = R \cup \{l_i\}$ $\mathbf{24}$ $\mathbf{25}$ if propAggr = False then $\mathbf{26}$ if $u \neq \bot$ then $I_l = I_l \cup (\overline{u}, R)$ $\mathbf{27}$ else $\mathbf{28}$ if $u = \perp$ then $\mathbf{29}$ propAggregate(C)30 forall the $i = |B|, \ldots, 1$ do $\mathbf{31}$ $\mathbf{32}$ if $u = b_{\langle i \rangle}$ then 33 $u = \bot$ $\mathbf{34}$ 35 if propAggr = False then 36 if aggregate(C) is positive then 37 else if |trueKey| = g - 1 $\mathbf{38}$ else 39 else if $|trueKey \cup undefKey| = g$ $\mathbf{40}$ **41** updateAggregateWithSharedVars(C, false)42





- At the end is needed an other else if block in which evaluate if A_2 can be propagated and if it can be propagated, then, the propagator will check that the A_1 is true and finally it makes propagation on A_2 .

at the same time, Algorithm 5 should be changed as follows:

- At line 27 and 40, an if block that check if A_2 is true must be added;
- A copy of the code described so far must be duplicated so to check first A_2 and then A_1 (the entire algorithm 5 contains two twin parts obtained by swapping the role of the two aggregates). This is needed because all possible propagation paths have to be considered.

4 Implementation and Experiments

4.1 Implementation

We started from the baseline system presented in [12] that has been extended to support the compilation of the propagation of aggregates. In particular, the implementation follows the execution presented in pseudo-code in algorithms of Section 3. The resulting compiler has been implemented in C++, and its output is also C++ code compliant to the WASP propagator interface, and is loaded in the ASP solver as a C++ dynamic library. Moreover, even though, in the compiler pseudo-code, we assumed that there is no repetition of predicates names we explicitly handle the case of duplicated predicates in the real implementation. Basically, when we have two literals with the same predicate name, we have to distinguish the data structure that will be declared for both literals in the nested join loops. The latest release is available at https://github.com/WaspWithCompilation/WASP_C.

4.2 Experimental evaluation

We carried our an experimental evaluation to empirically assessed the performance gain of the proposed approach w.r.t. the base solver WASP. Namely, we considered two hard benchmarks of the ASP competitions [9] where there are some constraints containing #count aggregates. The two considered benchmarks are *Combined Configuration* and *Abstract Dialectical Frameworks*.

In Combined Configuration, the problem is to configure an artifact by combining several sub-components in order to achieve some goals; whereas in Abstract Dialectical Frameworks the problem is to find all statements which are necessarily accepted or rejected in a given abstract argumentation framework. In both benchmarks we compile all constraints with aggregates supported by our implementation (i.e. constraints with exaclty one #count aggregate). The experiments were run on an Intel Xeon CPU E7-8880 v4 @ 2.20GHz, time and memory were limited to 10 minutes and 4 GB, respectively.

The results are presented in Figure 1a and Figure 1b as two cactus plots. Overall, our approach is able to boost the performance of WASP, with the result of obtaining smaller execution times, on average, and more solved instances (3 more instances for *Combined Configuration* and 7 more for *Abstract Dialectical Frameworks*). The results are very promising, also considering the fact that the benchmarks in the ASP competitions are more oriented towards the evaluation of solving techniques.

5 Conclusion

In this paper, we have extended the approach for the automatic compilation of constraints into propagators by adding support for the #count aggregate and we implemented it on top of the ASP solver WASP. Our tool has been empirically

validated on hard benchmarks from ASP competitions and demonstrate to be effective on improving the base solver WASP both in terms of number of solved instances and in raw speed. Concerning future work, we plan to extend our implementation for supporting constraints containing more than one aggregate, all the other aggregate functions of the ASP Core 2 standard, and possibly aggregates in rules.

References

- Adrian, W.T., Alviano, M., Calimeri, F., Cuteri, B., Dodaro, C., Faber, W., Fuscà, D., Leone, N., Manna, M., Perri, S., Ricca, F., Veltri, P., Zangari, J.: The ASP system DLV: advancements and applications. Künstliche Intell. **32**(2-3), 177–179 (2018)
- Alviano, M., Amendola, G., Dodaro, C., Leone, N., Maratea, M., Ricca, F.: Evaluation of disjunctive programs in WASP. In: LPNMR. Lecture Notes in Computer Science, vol. 11481, pp. 241-255. Springer (2019)
- Alviano, M., Calimeri, F., Dodaro, C., Fuscà, D., Leone, N., Perri, S., Ricca, F., Veltri, P., Zangari, J.: The ASP system DLV2. In: LPNMR. LNCS, vol. 10377, pp. 215-221. Springer (2017). https://doi.org/10.1007/978-3-319-61660-5 19, https://doi.org/10.1007/978-3-319-61660-5 19
- Alviano, M., Dodaro, C., Leone, N., Ricca, F.: Advances in WASP. In: LPNMR. LNCS, vol. 9345, pp. 40-54. Springer (2015)
- Amendola, G., Greco, G., Leone, N., Veltri, P.: Modeling and reasoning about NTU games via answer set programming. In: IJCAI. pp. 38–45. IJCAI/AAAI Press (2016)
- 6. Balduccini, М., Lierler, Y.: Constraint answer solver set why EZCSP and integration schemas matter. TPLP 17(4),462 - 515(2017).https://doi.org/10.1017/S1471068417000102. https://doi.org/10.1017/S1471068417000102
- Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM 54(12), 92–103 (2011)
- Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Maratea, M., Ricca, F., Schaub, T.: Asp-core-2 input language format. Theory Pract. Log. Program. 20(2), 294-309 (2020). https://doi.org/10.1017/S1471068419000450
- Calimeri, F., Gebser, M., Maratea, M., Ricca, F.: Design and results of the fifth answer set programming competition. Artif. Intell. 231, 151-181 (2016)
- Cuteri, B., Dodaro, C., Ricca, F., Schüller, P.: Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis. TPLP 17(5-6), 780-799 (2017)
- В., 11. Cuteri, Dodaro, С., Ricca, F., Schüller, P.: Partial compilation ofASP programs. TPLP 19(5-6),857-873 (2019).https://doi.org/10.1017/S1471068419000231
- Cuteri, B., Dodaro, C., Ricca, F., Schüller, P.: Overcoming the grounding bottleneck due to constraints in ASP solving: Constraints become propagators. In: IJCAI. pp. 1688-1694. ijcai.org (2020). https://doi.org/10.24963/ijcai.2020/234
- Cuteri, B., Reale, K., Ricca, F.: A logic-based question answering system for cultural heritage. In: JELIA. Lecture Notes in Computer Science, vol. 11468, pp. 526-541. Springer (2019)

- 14. Dal Palù, A., Dovier, A., Pontelli, E., Rossi, G.: GASP: answer set programming with lazy grounding. Fundam. Inform. **96**(3), 297-322 (2009)
- Dodaro, C., Alviano, M., Faber, W., Leone, N., Ricca, F., Sirianni, M.: The birth of a WASP: preliminary report on a new ASP solver. In: CILC. CEUR Workshop Proceedings, vol. 810, pp. 99–113. CEUR-WS.org (2011)
- Dodaro, C., Maratea, M.: Nurse scheduling via answer set programming. In: LP-NMR. LNCS, vol. 10377, pp. 301–307. Springer (2017)
- Erdem, E., Gelfond, M., Leone, N.: Applications of answer set programming. AI Magazine 37(3), 53-68 (2016)
- Erdem, E., Patoglu, V.: Applications of ASP in robotics. KI 32(2-3), 143-149 (2018). https://doi.org/10.1007/s13218-018-0544-x
- Faber, W., Pfeifer, G., Leone, N.: Semantics and complexity of recursive aggregates in answer set programming. Artif. Intell. 175(1), 278-298 (2011). https://doi.org/10.1016/j.artint.2010.04.002
- Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Wanko, P.: Theory solving made easy with clingo 5. In: ICLP TCs. OASICS, vol. 52, pp. 2:1-2:15 (2016)
- Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in gringo series 3. In: LPNMR. LNCS, vol. 6645, pp. 345–351. Springer (2011)
- Gebser, M., Leone, N., Maratea, M., Perri, S., Ricca, F., Schaub, T.: Evaluation techniques and systems for answer set programming: a survey. In: IJCAI. pp. 5450– 5456. ijcai.org (2018)
- Gebser, M., Maratea, M., Ricca, F.: The seventh answer set programming competition: Design and results. Theory Pract. Log. Program. 20(2), 176-204 (2020). https://doi.org/10.1017/S1471068419000061
- Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Comput. 9(3/4), 365-386 (1991)
- Kaufmann, B., Leone, N., Perri, S., Schaub, T.: Grounding and solving in answer set programming. AI Magazine 37(3), 25–32 (2016)
- Lefèvre, C., Nicolas, P.: The first version of a new ASP solver: Asperix. In: LPNMR. LNCS, vol. 5753, pp. 522-527. Springer (2009)
- Ostrowski, M., Schaub, T.: ASP modulo CSP: the clingcon system. TPLP 12(4-5), 485-503 (2012)
- Schüller, P.: Modeling variations of first-order horn abduction in answer set programming. Fundam. Inform. 149(1-2), 159-207 (2016)
- Susman, B., Lierler, Y.: SMT-based constraint answer set solver EZSMT (system description). In: ICLP TCs. OASICS, vol. 52, pp. 1:1–1:15. Schloss Dagstuhl -Leibniz-Zentrum fuer Informatik (2016)
- Weinzierl, A.: Blending Lazy-Grounding and CDNL Search for Answer-Set Solving. In: LPNMR. LNCS, vol. 10377, pp. 191-204 (2017)

A Detailed description of all algorithms

A.1 Algorithm 6

Algorithm 6, declares different sets, that accumulate the aggregate conjunctions (lines 1-5), and an empty variable substitution function σ . In order to build possible ground conjunctions of the form l_1, \ldots, l_m , we need a nested join loop for every l_j . Given the nesting level j, the propagator iterates over ground literals a_j that are true or undefined w.r.t I and update the variable substitution σ mapping variables in A[j] to constant in a_j (lines 17-19). Once the last nested join is reached (line 20), the propagator has to store new values for:

- Shared variables (*lines 20-23*);
- Aggregate variable and join tuple (*lines 25-42*) respectively for true and undefined aggregate conjunctions. Note that an aggregate conjunction is undefined if $\exists a_i$ such that a_i is undefined, otherwise is true.

At the end of each join loop, we roll back the variable substitution σ to its previous state and close nested join loops(*lines 43-46*).

A.2 Algorithm 7

Algorithm 7, prints the code that computes all possible body joins. In order to do this, first reorders body literal with the function computeBodyOrdering that returns a new list B where negative literals are always at the end of B, and c is not in B if c is not None. Once the list of literals B is computed, the algorithm prints a nested join loop for each B[j] such that B[j] is positive and a nested if for each B[j] such that B[j] is negative. Each nested join loop iterates on true literals belonging to I^+ that match $\sigma(B[j])$ and on undefined literals that match $\sigma(B[j])$ (lines 6-10) and update the variable substitution σ with variables in B[j] (lines 13-15). For each nested if, since positive literal was already evaluated and so we have a bound for every variable in negative literals (safeness), checks only if $\sigma(B[j])$ is true or undefined (lines 17-20).

Algorithm 6 buildAggregateJoin(C)

Input : A constraint C**Output:** Prints code that builds join tuples and their projection over aggregate and shared variables 1 $trueJoin = \emptyset$ 2 $undefJoin = \emptyset$ $sharedVarKey = \emptyset$ 3 $trueKey = \emptyset$ $\mathbf{4}$ $undefKey = \emptyset$ $\mathbf{5}$ $\sigma_{aggr} = \epsilon$ 6 7 $u_{aggr} := false$ A = aggrLit(C)8 forall the $j = 1, \dots, |A|$ do 9 $\sigma_{aggr}^{(j)} = \sigma_{aggr}$ 10 $\overline{A_{\langle\!\langle \mathbf{j}\rangle\!\rangle}=\{a\in I^+\mid match(\sigma_{aggr}(\mathsf{*A}[\mathbf{j}]\mathsf{*}),a)\}}$ 11 $\overline{UA_{\langle\!\langle \mathbf{j}\rangle\!\rangle} = \{ p \in (\mathcal{B} \setminus I)^+ \mid match(\sigma_{aggr}(\langle\!\langle \mathbf{A}[\mathbf{j}]\rangle\!\rangle), p) \land \overline{p} \notin I \} }$ $\mathbf{12}$ $for a_{\langle j \rangle} \in (A_{\langle j \rangle} \cup UA_{\langle j \rangle}) \{$ 13 if $a_{\langle ij \rangle} \in UA_{\langle ij \rangle}$ then 14 $u_{aggr} = true$ 15 $u_{aggr}^{(i)} = u_{aggr}$ 16 $\overline{\text{forall the } k} = 1, \dots, |trm(A[j])| \text{ do}$ 17 if trm(A[j])[k] is variable then $\mathbf{18}$ $\sigma_{aggr} = \sigma_{aggr} \cup \{ \operatorname{«trm}(\mathbf{A}[\mathbf{j}])[\mathbf{k}] \gg \mapsto trm(a_{\langle\!\langle \mathbf{j} \rangle\!\rangle})[\langle\!\langle \mathbf{k} \rangle\!\rangle] \}$ $\mathbf{19}$ $sharedVarKey = sharedVarKey \cup \{$ $\mathbf{20}$ 21 forall the $v \in sharedVars(C)$ do $\sigma_{aggr}(\ll v \gg)$ $\mathbf{22}$ $\mathbf{23}$ $\mathbf{24}$ 25 if u_{aggr} is false then $trueKey = trueKey \cup \{$ $\mathbf{26}$ forall the $v \in aggregateVars(C)$ do $\mathbf{27}$ $\sigma_{aggr}($ «v») $\mathbf{28}$ } $\mathbf{29}$ $trueJoin = trueJoin \cup \{($ 30 $\overline{\text{forall the } z = 1, \dots, |A| \text{ do}}$ $\mathbf{31}$ $a_{\langle Z \rangle}$ 32)} 33 34 <u>else</u> $undefKey = undefKey \cup \{$ $\mathbf{35}$ **36 forall the** $v \in aggregateVars(C)$ **do** $\sigma_{aggr}(\text{«v»})$ $\mathbf{37}$ } 38 $undefJoin = undefJoin \cup \{($ 39 forall the $z = 1, \ldots, |A|$ do $\mathbf{40}$ <mark>a</mark>≪z≫ $\mathbf{41}$ 42)} 43 forall the j = |A|, ..., 1 do $\sigma_{aggr} = \sigma_{aggr}^{\langle\!\langle j \rangle\!\rangle}$ 44 $u_{aggr} = u_{agg}^{\langle j \rangle}$ $\mathbf{45}$ } $\mathbf{46}$

Algorithm 7 printNestedJoinLoop			
Input : A constraint C , a literal $c \in C$ that can be also None			
\mathbf{Output} : Return the list of body literals ordered starting from c			
B = computeBodyOrdering(C, c)			
$2 \underline{ u := \bot}$			
$3 \text{forall the } j = 1, \dots, B \text{ do}$			
$4 \qquad \underline{-\sigma_{\langle j \rangle} = \sigma}$			
5if $B[j]$ is positive then			
$6 \left \underline{T}_{\langle \mathbf{j} \rangle} = \{ t \in I^+ \mid match(\sigma(\langle \mathbf{B}[\mathbf{j}] \rangle, t) \} \right $			
7 $U_{\langle j \rangle} = \emptyset$			
8 $if u = \bot then$			
9 $U_{\langle \mathbf{j} \rangle} = \{ p \in (\mathcal{B} \setminus I)^+ \mid match(\sigma(\langle \mathbf{B}[\mathbf{j}] \rangle), p) \land \overline{p} \notin I \}$			
10 $- \underbrace{\text{for } b_{\langle j \rangle} \in (T_{\langle j \rangle} \cup U_{\langle j \rangle}) }_{\text{for } j \rangle} $			
$\begin{array}{c c} 11 & & \\ & \underline{ \mathbf{if} \ b_{\ll j \gg} \in U_{\ll j \gg} \ \mathbf{then}} \end{array}$			
12 $u = b_{\langle j \rangle}$			
13 forall the $k = 1, \dots, trm(B[j]) $ do			
14if $trm(B[j])[k]$ is variable then			
15 $ \qquad $			
16else			
17 $b_{\langle j \rangle} = \sigma(\langle B[j] \rangle)$			
18 $_ if \ b_{\langle j \rangle} \in I \lor (u = \bot \land b_{\langle j \rangle} \in (\mathcal{B} \setminus I)) \{ then \}$			
19 if $u = \bot \land b_{\langle\!\langle j \rangle\!\rangle} \in (\mathcal{B} \setminus I)$ then			
20 $u = b_{\langle j \rangle}$			
21 return B			

A.3 Algorithm 8

Algorithm 8, instead, prints the code that discards those join tuples that do not share the values of the shared variables or rolls back the join tuple structures to their previous state. In particular, *lines 2-20* are printed if we want to discard join tuples. In this case, first, we save the previous state in fresh structures *lines 2-6* and then we modify the existing ones *line 7-20*. Otherwise, if we need to roll back join tuples we assign previous state structures to the current ones (*lines 22-26*).

Algorithm 8 updateAggregateWithSharedVars					
I	Input : A constraint C , a boolean value d				
(Dutput : Prints code to discard those join tuples that don't match a value of				
	shared variables or restore the initial set of join tuples				
1 i	f d is true then				
2	$\underline{trueJoinPrevious = trueJoin}$				
3	undefJoinPrevious = undefJoin				
4	$\underline{sharedVarKeyPrevious = sharedVarKey}$				
5	trueKeyPrevious = trueKey				
6	undefKeyPrevious = undefKey				
7	$trueKey = \emptyset$				
8	for $t \in trueJoin$ {				
9	$if projOnSharedVars(t) \neq sharedVarTuple then$				
10	$trueJoin = trueJoin \setminus \{t\}$				
11	else				
12	$trueKey = trueKey \cup projOnAggrVars(t)$				
13					
14	$undefKey = \emptyset$				
15	for $t \in undefJoin$ {				
16	$if projOnSharedVars(t) \neq sharedVarTuple then$				
17	$undefJoin = undefJoin \setminus \{t\}$				
18	else				
19	$undefKey = undefKey \cup projOnAggrVars(t)$				
20					
21 E	else				
22	$\underline{trueJoin = trueJoinPrevious}$				
23	$_undefJoin = undefJoinPrevious$				
24	sharedVarKey = sharedVarKeyPrevious				
25	trueKey = trueKeyPrevious				
26	undefKey = undefKeyPrevious				