IOS Press

Look-back Techniques for ASP Programs with Aggregates[*]

Wolfgang Faber
Department of Mathematics - University of Calabria

Nicola Leone [C]
Department of Mathematics - University of Calabria

Marco Maratea
DIST - University of Genova, and
Department of Mathematics - University of Calabria

Francesco Ricca
Department of Mathematics - University of Calabria

Abstract. The introduction of aggregates has been one of the most relevant language extensions to Answer Set Programming (ASP). Aggregates are very expressive, they allow to represent many problems in a more succinct and elegant way compared to aggregate-free programs. A significant amount of research work has been devoted to aggregates in the ASP community in the last years, and relevant research results on ASP with aggregates have been published, on both theoretical and practical sides. The high expressiveness of aggregates (eliminating aggregates often causes a quadratic blow-up in program size) requires suitable evaluation methods and optimization techniques for an efficient implementation. Nevertheless, in spite of the above-mentioned research developments, aggregates are treated in a quite straightforward way in most ASP systems.

In this paper, we explore the exploitation of look-back techniques for an efficient implementation of aggregates. We define a reason calculus for backjumping in ASP programs with aggregates. Furthermore, we describe how these reasons can be used in order to guide look-back heuristics for programs with aggregates. We have implemented both the new reason calculus and the proposed heuristics in the DLV system, and have carried out an experimental analysis on publicly available benchmarks which shows significant performance benefits.

Keywords: Knowledge Representation and Reasoning, Nonmonotonic Reasoning, Answer Set Programming, Heuristics, Aggregates.

Address for correspondence: Department of Mathematics, University of Calabria Via P. Bucci, cubo 30b, 87036 Rende (CS), Italy.

[*]Preliminary versions of this work have been published at 15th RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion (RCRA'08) and Workshop on Logic and Search: Computation of structures from declarative descriptions (LASH'08).

[C]Corresponding author

1. Introduction

Answer Set Programming (ASP) [23] has become a popular logic programming framework during the last decade, the reason being mostly its intuitive declarative reading, a mathematically precise expressivity, and last but not least the availability of some efficient ASP systems, which favoured the implementation of many advanced real-world ASP applications (see, e.g., [34, 36, 29]). One of the most important extensions of the language of ASP has been the introduction of aggregates. Aggregates significantly enhance the language of ASP, allowing for natural and concise modelling of many problems. A lot of work has been done both theoretically (mostly for determining the semantics of aggregates that occur in recursion) [32, 39, 11, 13], and practically, for endowing systems with a selection of aggregate functions [38, 7, 14, 20, 1].

However, work on optimizing system performance with respect to aggregates in ASP is still sparse, and current implementations use more or less ad-hoc techniques. Moreover, fine-grained details on their treatment have been rarely presented (with the recent, notable exception of [18], which nonetheless focus on a single aggregate function).

In this work, we explore the exploitation of look-back techniques for an efficient implementation of aggregates. We build upon a technique for backjumping, which was developed in the setting of the solver DLV for aggregate-free ASP programs. As a main contribution, we describe how the *reason calculus* defined in [35] can be extended for keeping track of the reasons for several types of aggregates supported by DLV. The information collected in this way can then be exploited directly for backjumping, using the original method described in [35].

Importantly, reasons for aggregates can also be exploited for look-back heuristics. Indeed, we show how the look-back heuristics presented in [30] can be extended to the aggregate case. For this task, a key issue is the initialization of heuristic values: since look-back heuristics use information of the computation done so far, they would be completely uninformed at the beginning of the computation, as no information can be looked back on. In order to tackle this issue, we consider two alternatives: in the first, simple alternative, the "relevance" of an aggregate literal is determined by the size of its aggregate set. The second, more informed, alternative applies standard techniques on an aggregate-free program equivalent to the given program with aggregates for initializing the heuristic values. In this second case we pay particular attention to avoiding the materialization of this aggregate-free program, but use the knowledge about its structure for computing the initial values. This second method is exact in the aggregate-stratified case, in the sense that the aggregate-free program is equivalent to the original program with aggregates, and it can still be used for the purpose of a heuristic in the aggregate-unstratified case, as it can serve as a reasonable approximation.

We have implemented the proposed techniques for the aggregate-stratified setting, and report on a performance evaluation of the obtained prototype on publicly available benchmarks, in which we observed performance benefits for the enhanced system.

Summing up the main contributions of the work:

- We extend the reason calculus in [35] to include reasons for all the aggregate functions supported by DLV;
- We show how this extension can be used to guide look-back heuristics, and we present two alternatives for their initialization;

- We implement these new features in DLV;
- We perform an experimental analysis on publicly available benchmarks, which shows performance benefits for the enhanced system employing the more elaborate heuristic.

The paper is structured as follows: first we review syntax and semantics of ASP with aggregates in Section 2, and the backjumping method and reason calculus of DLV in Section 3. We then describe the extension of the reason calculus to aggregates in Section 4. Dealing with look-back heuristics in the presence of aggregates is discussed in Section 5. The experimental evaluation of the enhanced system is presented and discussed in Section 6. In Section 7 we discuss related work and conclude the paper in Section 8. The appendix provides further details on the experiments.

2. Answer Set Programming with Aggregates

In this section, we recall syntax, semantics, and some basic properties of logic programs with aggregates under the answer set semantics.

2.1. Syntax

Variables, Constants, and Predicates. We consider finite sets of *variables*, *constants*, and *predicates*. Similar to Prolog notation, we will denote variables as strings starting with uppercase letters and constants as non-negative integers or strings starting with lowercase letters. Predicates are strings starting with lowercase letters or symbols such as $=$, $<$, $>$ (so-called built-in predicates that have a fixed meaning). An *arity* (non-negative integer) is associated with each predicate.

Standard Atoms and Literals. A *term* is either a variable or a constant. A *standard atom* is an expression $p(t_1, \ldots, t_n)$, where p is a *predicate* of arity n and $t_1, \ldots, t_n$ are terms. A *standard literal* L is either a standard atom A (in this case, it is *positive*) or a standard atom A preceded by the default negation symbol not (in this case, it is *negative*). A conjunction of standard literals is of the form $L_1, \ldots, L_k$ where each L_i ($1 \leq i \leq k$) is a standard literal. Two literals are complementary if they are of the form p and $\mathrm{not}\ p$ (where p is an atom). Given a literal L, let $\neg.L$ denote its complementary literal. Accordingly, given a set $\mathcal{L}$ of literals, $\neg.\mathcal{L} = \{\neg.L \mid L \in \mathcal{L}\}$.

Set Terms. A *set term* is either a symbolic set or a ground set. A *symbolic set* is a pair $\{Vars : conj\}$, where $Vars$ is a list of variables and $conj$ is a conjunction of standard atoms.[1] A *ground set* is a set of pairs of the form $\langle \bar{t} : conj \rangle$, where $\bar{t}$ is a list of constants and $conj$ is a ground conjunction of standard atoms.

Aggregate Functions. An *aggregate function* is of the form $f(S)$, where S is a set term, and f is an *aggregate function symbol*. Intuitively, an aggregate function can be thought of as a (possibly partial) function mapping multisets of constants to a constant.

[1] Intuitively, a symbolic set $\{X : a(X,Y), p(Y)\}$ stands for the set of X-values making $a(X,Y), p(Y)$ true, that is, $\{X \mid \exists Y \mathit{such\ that}\ a(X,Y), p(Y)\ \mathit{is\ true}\}$.

Example 2.1. *In the examples, we adopt the syntax of* DLV *to denote aggregates.* Aggregate functions currently supported by the DLV system are: $\#\texttt{count}$ (number of terms), $\#\texttt{sum}$ (sum of non-negative integers), $\#\texttt{min}$ (minimum term), $\#\texttt{max}$ (maximum term).[2] ∎

Aggregate Literals. An *aggregate atom* is $f(S) \prec T$, where $f(S)$ is an aggregate function, $\prec \in \{=, <, \leq, >, \geq\}$ is a predefined comparison operator, and T is a term (variable or constant) referred to as guard.

Example 2.2. In the following aggregate atoms, the latter contains a ground set and could be a ground instance of the former:

$$\#\texttt{max}\{Z : r(Z), a(Z,V)\} > Y$$
$$\#\texttt{max}\{\langle 2 : r(2), a(2,k)\rangle, \langle 2 : r(2), a(2,c)\rangle\} > 1$$

∎

An *atom* is either a standard atom or an aggregate atom. A *literal* L is an atom A or an atom A preceded by the default negation symbol `not`; if A is an aggregate atom, L is an *aggregate literal*.

Programs. A *rule* r is a construct

$$a_1 \texttt{ v } \cdots \texttt{ v } a_n \texttt{ :- } b_1, \ldots, b_k, \text{ not } b_{k+1}, \ldots, \text{ not } b_m.$$

where $a_1, \ldots, a_n$ are standard atoms, $b_1, \ldots, b_m$ are atoms, and $n \geq 1$, $m \geq k \geq 0$. The disjunction $a_1 \texttt{ v } \cdots \texttt{ v } a_n$ is referred to as the *head* of r while the conjunction $b_1, ..., b_k,$ not $b_{k+1}, ...,$ not b_m is the *body* of r. We define $H(r) = \{a_1, \ldots, a_n\}$, $B^+(r) = \{b_1, ..., b_k\}$, $B^-(r) = \{\text{not } b_{k+1}, ..., \text{not } b_m\}$, and $B(r) = B^+(r) \cup B^-(r)$. A *global* variable of a rule r appears in a standard atom of r (possibly also in other atoms); all other variables are *local*. A *program* is a set of rules.

Note that this syntax does not explicitly allow rules without head atoms, also known as integrity constraints, which are usually found in ASP languages. They can, however, be simulated in a standard way by using a new symbol and negation.

Safety. A rule r is *safe* if the following conditions hold: (i) each global variable of r appears in a positive standard literal in the body of r; (ii) each local variable of r appearing in a symbolic set $\{\mathit{Vars} : \mathit{conj}\}$ appears in an atom of conj; (iii) each guard of an aggregate atom of r is a constant or a global variable. A program $\mathcal{P}$ is safe if all $r \in \mathcal{P}$ are safe. In the following we assume that programs are safe. Note that unsafe rules in general are not domain-independent, a condition which gives rise to semantic issues.

Example 2.3. Consider the following rules with aggregates:

$$p(X) \texttt{ :- } q(X,Y,V), \#\texttt{max}\{Z : r(Z), a(Z,V)\} > Y.$$
$$p(X) \texttt{ :- } q(X,Y,V), \#\texttt{sum}\{S : a(Z,Z)\} > Y.$$
$$p(X) \texttt{ :- } q(X,Y,V), \#\texttt{min}\{Z : r(Z), a(Z,V)\} > T.$$

The first rule is safe, while the second is not, since the local variable S violates condition (ii). The third rule is not safe either, since the guard T violates condition (iii). ∎

[2]The first two aggregates roughly correspond, respectively, to the cardinality and weight constraint literals of LPARSE. $\#\texttt{min}$ and $\#\texttt{max}$ are undefined for an empty set.

Stratification. A program $\mathcal{P}$ is *aggregate-stratified* if there exists a function $|| \, ||$, called *level mapping*, from the set of (standard) predicates of $\mathcal{P}$ to ordinals, such that for each pair a and b of standard predicates, occurring in the head and body of a rule $r \in \mathcal{P}$, respectively: (i) if b appears in an aggregate atom, then $||b|| < ||a||$, and (ii) if b occurs in a standard atom, then $||b|| \leq ||a||$.

Example 2.4. Consider the program consisting of the following two rules:

$$\begin{aligned} q(X) &\text{ :- } p(X),\ \#\mathtt{count}\{Y : a(Y,X), b(X)\} \leq 2. \\ p(X) &\text{ :- } q(X),\ b(X). \end{aligned}$$

The program is aggregate-stratified, as the level mapping $||a|| = ||b|| = 1, \quad ||p|| = ||q|| = 2$ satisfies the required conditions. If we add the rule $b(X) \text{ :- } p(X)$, then no such level-mapping exists and the program becomes aggregate-unstratified. ▮

Intuitively, aggregate-stratification forbids recursion through aggregates. While the semantics of aggregate-stratified programs is more or less agreed upon, different and disagreeing semantics for aggregate-unstratified programs have been defined in the past, cf. [32, 39, 11]. In the following, we will consider aggregate-stratified programs. We refer to [26] for an overview of proposed semantics for the unstratified case and how they relate.

2.2. Answer Set Semantics

Universe and Base. Given a program $\mathcal{P}$, let $U_{\mathcal{P}}$ denote the set of constants appearing in $\mathcal{P}$ (its Herbrand universe), and $B_{\mathcal{P}}$ be the set of standard atoms constructible from the (standard) predicates of $\mathcal{P}$ with constants in $U_{\mathcal{P}}$ (the Herbrand base). Given a set X, let $\overline{2}^X$ denote the set of all multisets over elements from X. Without loss of generality, we assume that aggregate functions map to $\mathbb{Z}$ (the set of integers).

Example 2.5. $\#\mathtt{count}$ is defined over $\overline{2}^{U_{\mathcal{P}}}$, $\#\mathtt{sum}$ over $\overline{2}^{\mathbb{N}}$, $\#\mathtt{min}$ and $\#\mathtt{max}$ are defined over $\overline{2}^{\mathbb{N}} \setminus \{\emptyset\}$. ▮

Instantiation. A *substitution* is a mapping from a set of variables to $U_{\mathcal{P}}$. A substitution from the set of global variables of a rule r (to $U_{\mathcal{P}}$) is a *global substitution for r*; a substitution from the set of local variables of a symbolic set S (to $U_{\mathcal{P}}$) is a *local substitution for* S. Given a symbolic set without global variables $S = \{Vars : conj\}$, the *instantiation of* S is the ground set of pairs $inst(S) = \{\langle \gamma(Vars) : \gamma(conj) \rangle \mid \gamma$ *is a local substitution for* $S\}$.[3]
A *ground instance* of a rule r is obtained in two steps: (1) a global substitution σ for r is first applied over r; (2) every symbolic set S in $\sigma(r)$ is replaced by its instantiation $inst(S)$. The instantiation $Ground(\mathcal{P})$ of a program $\mathcal{P}$ is the set of all possible instances of the rules of $\mathcal{P}$.

Example 2.6. Consider the program $\mathcal{P}_1$:

$$q(1) \,\mathtt{v}\, p(2,2). \qquad q(2) \,\mathtt{v}\, p(2,1). \qquad\qquad t(X) \text{ :- } q(X), \#\mathtt{sum}\{Y : p(X,Y)\} > 1.$$

[3]Given a substitution σ and an object Obj of the language (rule, set, etc.), we denote by $\sigma(Obj)$ the object obtained by replacing each variable X in Obj by $\sigma(X)$.

The instantiation $Ground(\mathcal{P}_1)$ is

$$\begin{array}{ll} q(1)\,\mathtt{v}\,p(2,2). & t(1) \text{ :- } q(1), \#\mathtt{sum}\{\langle 1\!:\!p(1,1)\rangle, \langle 2\!:\!p(1,2)\rangle\} > 1. \\ q(2)\,\mathtt{v}\,p(2,1). & t(2) \text{ :- } q(2), \#\mathtt{sum}\{\langle 1\!:\!p(2,1)\rangle, \langle 2\!:\!p(2,2)\rangle\} > 1. \end{array}$$

∎

Interpretations. An *interpretation* for a program $\mathcal{P}$ is a consistent set of standard ground literals, that is $I \subseteq (B_{\mathcal{P}} \cup \neg.B_{\mathcal{P}})$ such that $I \cap \neg.I = \emptyset$. A standard ground literal L is true (resp. false) with respect to I if $L \in I$ (resp. $L \in \neg.I$). If a standard ground literal is neither true nor false with respect to I then it is undefined with respect to I. We denote by I^+ (resp. I^-) the set of all atoms occurring in standard positive (resp. negative) literals in I. We denote by $\bar{I}$ the set of undefined atoms with respect to I (that is, $B_{\mathcal{P}} \setminus I^+ \cup I^-$). An interpretation I is *total* if $\bar{I}$ is empty (that is, $I^+ \cup \neg.I^- = B_{\mathcal{P}}$), otherwise I is *partial*.

An interpretation also provides a meaning for aggregate literals. Their truth value is first defined for total interpretations, and then generalized to partial ones.

Let I be a total interpretation. A standard ground conjunction is true with respect to I if all its literals are true with respect to I; it is false if any of its literals is false with respect to I. Let $f(S)$ be an aggregate function, where S is a ground set. The valuation $I(S)$ of S with respect to I is the multiset of the first constant of the elements in S whose conjunction is true with respect to I. More precisely, let $I(S)$ denote the multiset $[t_1 \mid \langle t_1, ..., t_n : conj\rangle \in S \wedge$ $conj$ *is true with respect to* I $]$. The valuation $I(f(S))$ of an aggregate function $f(S)$ with respect to I is the result of the application of f on $I(S)$. If the multiset $I(S)$ is not in the domain of f, $I(f(S)) = \bot$ (where $\bot$ is a fixed symbol not occurring in $\mathcal{P}$).

An instantiated aggregate atom A of the form $f(S) \prec k$ is *true with respect to* I if: (i) $I(f(S)) \neq \bot$, and, (ii) $I(f(S)) \prec k$ holds; otherwise, A is false. An instantiated aggregate literal of the form $\mathrm{not} f(S) \prec k$ is *true with respect to* I if (i) $I(f(S)) \neq \bot$, and, (ii) $I(f(S)) \prec k$ does not hold; otherwise, it is false.

If I is a *partial* interpretation, an aggregate literal A is true (resp. false) with respect to I if it is true (resp. false) with respect to *each total* interpretation J extending I (that is, for all J such that $I \subseteq J$, A is true (resp. false) with respect to J); otherwise, it is undefined.

Example 2.7. Consider the atom $A = \#\mathtt{sum}\{\langle 1 : p(2,1)\rangle, \langle 2 : p(2,2)\rangle\} > 1$. Let S be the ground set in A. For the interpretation $I = \{p(2,2)\}$, each extending total interpretation contains either $p(2,1)$ or $\mathrm{not}\ p(2,1)$. Therefore, either $I(S) = [2]$ or $I(S) = [1,2]$ and the application of $\#\mathtt{sum}$ yields either $2 > 1$ or $3 > 1$, hence A is true with respect to I. ∎

The above definitions of interpretation and truth values preserve "knowledge monotonicity". If an interpretation J extends I (that is, $I \subseteq J$), then each literal which is true with respect to I is true with respect to J, and each literal which is false with respect to I is false with respect to J as well.

Minimal Models. Given an interpretation I, a rule r is *satisfied with respect to* I if some head atom is true with respect to I whenever all body literals are true with respect to I. A total interpretation M is a *model* of a program $\mathcal{P}$ if all rules $r \in Ground(\mathcal{P})$ are satisfied with respect to M. A model M for $\mathcal{P}$ is (subset) minimal if no model N for $\mathcal{P}$ exists such that $N^+ \subsetneq M^+$. Note that, under these definitions, the word *interpretation* refers to a possibly partial interpretation, while a *model* is always a total interpretation.

Answer Sets. We now recall the generalization of the Gelfond-Lifschitz transformation and answer sets for programs with aggregates from [11]: Given a ground program $\mathcal{P}$ and a total interpretation I, let $\mathcal{P}^I$ denote the transformed program obtained from $\mathcal{P}$ by deleting all rules in which a body literal is false with respect to I. I is an answer set of a program $\mathcal{P}$ if it is a minimal model of $Ground(\mathcal{P})^I$.

Example 2.8. Consider the total interpretations $I_1 = \{p(a), q(a)\}$, $I_2 = \{\text{not } p(a), q(a)\}$, $I_3 = \{p(a), \text{not } q(a)\}$, and $I_4 = \{\text{not } p(a), \text{not } q(a)\}$ and program:

$$P \quad = \quad \{p(a) \text{ :- } \#\texttt{count}\{X : q(X)\} > 0.\}$$

Then we obtain:

$$\begin{array}{lcl} Ground(P) & = & \{p(a) \text{ :- } \#\texttt{count}\{\langle a : q(a)\rangle\} > 0.\} \\ Ground(P)^{I_1} & = & Ground(P) \\ Ground(P)^{I_2} & = & Ground(P) \\ Ground(P)^{I_3} & = & \emptyset \\ Ground(P)^{I_4} & = & \emptyset \end{array}$$

We observe that: I_1 and I_3 are not answer sets of P, indeed both I_1 and I_3 are not minimal models of respectively $Ground(P)^{I_1}$ and $Ground(P)^{I_3}$; I_2 is not a model for P (the only rule in P is not satisfied in I_2); and I_4 is the only answer set of P. ■

Note that any answer set A of $\mathcal{P}$ is also a model of $\mathcal{P}$ because $Ground(\mathcal{P})^A \subseteq Ground(\mathcal{P})$, and rules in $Ground(\mathcal{P}) \setminus Ground(\mathcal{P})^A$ are satisfied with respect to A.

3. Answer Sets Computation with Backjumping and Reason Calculus in DLV

The computation of the answer sets of a disjunctive program $\mathcal{P}$ is usually carried out in two steps. The first, called *instantiation (or grounding)*, has the role of generating a ground program having the same answer sets of $\mathcal{P}$ (usually, much smaller than —but equivalent to— the theoretical ground instantiation of $\mathcal{P}$); the second step of the computation, often called *model generation*, amounts to searching for the answer sets of the ground program produced by the instantiation.

Model generation is the non-deterministic core of an ASP system, and it is usually implemented as a backtracking search similar to the Davis-Putnam-Logemann-Loveland (DPLL) procedure [4] for SAT solving. The Model Generator algorithm employed by DLV is sketched In Figure 1.[4] Basically, starting from the empty (partial) interpretation ($I = \emptyset$), the $ModelGenerator$ procedure repeatedly assumes truth-values for atoms (chosen according to a heuristic), subsequently computing their deterministic consequences (by a call to $PropagateDetCons$). This is done until either an answer set is found or an inconsistency is detected. In particular, if the input program has an answer set, the procedure $ModelGenerator$ returns True (and I contains the computed solution); otherwise, it returns False.

[4]The algorithm presented here is simplified in order to focus on the aspects that are relevant to our contribution. For details, we refer to [9] for the basic DLV algorithm and [35] for backjumping.

Inconsistencies are detected in two cases: (i) *conflicting literals*, that is, propagation determines that an atom a and its negation $\text{not } a$ should both hold (in this case $PropagateDetCons$ returns the set of all literals $\mathcal{L}$); or (ii) *stability check failures*. The latter case occurs if the checked interpretation, which is guaranteed to be a model, is not stable (and the function $IsAnswerSet$ returns false). This is a peculiarity of disjunctive ASP, since the stability check is not needed in non-disjunctive ASP systems [27]. In both cases, since the last choice might not be the only cause of the found inconsistency, the system detects (using the $ComputeNextLevel$ function) the most recent choice ℓ that is *relevant* for the found inconsistency and it goes back to modify ℓ (non-chronological backtracking or backjumping). Note that this is done in order to avoid encountering again the same inconsistency, thus performing a lot of useless computations.

A crucial point is how relevance for an inconsistency can be determined. The necessary information for deciding relevance is recorded by means of a *reason calculus* [35], which collects information about the choices ("reasons") whose truth-values have caused truth-values of other deterministically derived atoms. In practice, once an atom has been assigned a truth-value during the computation, we can associate a reason to it. For instance, given a rule $a \text{ :- } b, c, \text{not } d.$, if b and c are true and d is false in the current partial interpretation, then a will be derived as true during propagation. In this case, a is true because b and c are true and d is false. Therefore, the reasons for a will consist of the reasons for b, c, and d. More generally, the reason of a derived literal consists of the reasons of those literals that entail its truth; on the other hand, *chosen* literals become true unconditionally, and their only reason is their choice. Therefore, each literal l derived during the propagation has an associated set of positive integers $R(l)$ representing the reasons for l, which contains essentially the recursion levels of the choices which entail l. Hence, for any chosen literal c, $|R(c)| = 1$ holds, while for any derived (that is, non-chosen) literal n, $|R(n)| \geq 1$ holds. For instance, if $R(l) = \{1, 3, 4\}$, then the literals chosen at recursion levels 1,3 and 4 entail l.

The reason information is used for detecting the set of chosen literals that are relevant for an inconsistency. It is easy to see that, for avoiding that the same inconsistency occurs again, we have to go back in the search until at least one choice that causes the inconsistency is undone. This set of choices (that entail the inconsistency) is called the reason for the inconsistency. In the case of conflicting literals, it is obtained by the combination of the reasons for a and $\text{not } a$: $R(a) \cup R(\text{not } a)$. In the case of a stability check failure the reason for such an inconsistency is always based on an unfounded set, which has been determined inside $IsAnswerSet$ as a side-effect. Using this unfounded set, the reason for the inconsistency is composed of the reasons of literals which satisfy rules containing unfounded atoms in their head [35].

In the algorithm of Figure 1, the reason for an inconsistency is stored in the variable $IncReason$, and backjumping is performed by computing the next recursion level ($next\_level$) by calling the function $ComputeNextLevel$. $ComputeNextLevel$, basically, selects the maximal recursion level contained in $IncReason$, which is different from $current\_level$.

It is worth noting that $PropagateDetCons$ plays a crucial role in the model generation process. It is similar to *unit propagation* (UP) in DPLL SAT solvers; however, its implementation is quite more complex than UP, because $PropagateDetCons$ implements a set of inference rules. Those rules combine an extension of the Well-founded operator for disjunctive programs with a number of techniques based on disjunctive ASP program properties. We will not report in detail here all the propagation rules for standard ASP programs and the associated reason calculus, as they are not a novelty of this paper, and refer to [10, 3] for their precise definitions and implementation. However, in the following, we will

```
static integer curr_level = 0; //stores the current recursion level
static integer next_level = 0; //used to control recursion

bool ModelGenerator (Interpretation& I) {

    curr_level ++; //update current recursion level
    next_level = current_level;

    I = PropagateDetCons ( I, IncReason );

    if ( I == $\mathcal{L}$ ) //conflicting literals found during propagation
      curr_level --; return false;

    if ( "no atom is undefined in I" )
      if IsAnswerSet( I, IncReason ); return true; //answer set found
      else {     //inconsistency from model checking
        next_level = ComputeNextLevel(IncReason);
        curr_level --; return false; }

    Select an undefined atom $A$ using a heuristic;

    if ( ModelGenerator( $I \cup \{A\}$ ) return true;
    else if (next_level < curr_level) // control recursion (backjumping?)
          return false;

    if ( ModelGenerator ( $I \cup \{\text{not } A\}$) return true;
    else if (next_level < curr_level) // control recursion (backjumping?)
          return false;

    // tried both $A$ and $\text{not } A$, deal with inconsistency
    next_level = ComputeNextLevel(IncReason);
    curr_level --; return false;
};
```

Figure 1. Computation of Answer Sets in DLV.

describe the inference rules needed for correctly implementing aggregates [38, 7], and we present the associated extension of the reason calculus which allows for dealing with aggregates.

4. Propagation Rules and Reason Calculus for Aggregates

We next report the reason calculus for each aggregate supported by DLV. Hereafter, a partial interpretation (denoted by a set of literals) I is assumed to be given. Moreover, without loss of generality, we assume that aggregate literals are in the simplified form $f(A)\Theta k$, where: (i) the aggregate set A only contains pairs of the form $\langle \bar{t} : a \rangle$, where a is an atom; and (ii) only two comparison operators are allowed, namely $\Theta \in \{<, >\}$.

Actually, DLV internally rewrites the input program $\mathcal{P}$ to obtain this simplified form. In particular, each aggregate literal $f(S) \prec T$ (with $\prec \in \{=, <, \leq, >, \geq\}$) occurring in $\mathcal{P}$ is first transformed in such a way that only one of the comparison operators in $\Theta \in \{<, >\}$ is used;[5] then, for each $\langle \bar{t} : conj \rangle \in S$, $conj$ is replaced by a new atom $aux_{f(S)<T}(\overline{v})$, and the rule $aux_{f(S)<T}(\overline{v})$:- $conj$ is added to the program, where $\overline{v}$ are the variables occurring in $conj$. This means that conjunctions in S are replaced by freshly introduced auxiliary atoms, along with a rule defining the auxiliary atom by means of the conjunction. This transformation has several advantages: it simplifies both the description and the implementation of propagation; and (as it will become clear in the following) allows for defining some additional derivation rules. Moreover, let $A = \{\langle \bar{t}_1 : a_1 \rangle, \ldots, \langle \bar{t}_n : a_n \rangle\}$ be a set term; we define $\mathcal{C}_A = \bigcup_{\langle v, \bar{t}:a \rangle \in A \wedge \text{not } a \in I} R(\text{not } a)$ and $\mathcal{S}_A = \bigcup_{\langle v, \bar{t}:a \rangle \in A \wedge a \in I} R(a)$. Intuitively, $\mathcal{C}_A$ represents the reasons for false atoms in A, while $\mathcal{S}_A$ represents the reason for true atoms in A.

In the next sections, each propagation rule and the corresponding reason calculus are described in detail. In particular, we consider two different scenarios depending on whether the propagation proceeds from atoms in A to aggregate literals $f(A)\Theta k$ (forward inference) or the other way around (backward inference). Basically, in the first case we derive the truth/falsity of the aggregate literal $f(A)\Theta k$ from the truth/falsity of some atoms occurring in A; in the second case, given a rule containing an aggregate atom which is already known to be true or false with respect to the current interpretation,[6] we infer some atoms occurring in the conjunctions in A to be true/false.

4.1. Forward Inference

This kind of propagation rules apply when it is possible to derive an aggregate literal $f(A)\Theta k$ to be true or false because some atom in A is true or false with respect to I. As an example consider the program:

$$a(1). \quad a(2). \quad h : -\#\texttt{count}\{\langle 1 : a(1) \rangle, \langle 1 : a(2) \rangle\} < 1.$$

Since both $a(1)$ and $a(2)$ are facts, they are immediately derived to be true; then, since the function value for the aggregate is 2, the aggregate literal is inferred to be false by forward inference.

In the following, we report in a separate paragraph propagation rules and reason calculus for the aggregates supported by DLV. Hereafter, $\langle v, \bar{t} : a \rangle$ is a syntactic shorthand for $\langle v, t_1, \ldots, t_n : a \rangle$, where v is a constant and $\bar{t}$ is the list of constants $t_1, \ldots, t_n$, $n \geq 0$.

$\#\texttt{count}\{A\} > k$. Suppose that there exists a set $A' \subseteq A$ such that for each $\langle \bar{t} : a \rangle \in A'$, a is false in I and $|A'| \geq |A| - k$, then $\#\texttt{count}\{A\} > k$ is inferred to be false and its reasons are set to $\mathcal{C}_{A'}$. Conversely, suppose that there exists a set $A' \subseteq A$ such that for each $\langle \bar{t} : a \rangle \in A'$, a is true in I and $|A'| > k$, then we infer that $\#\texttt{count}\{A\} > k$ is true and we set its reason to $\mathcal{S}_{A'}$.

Let us now consider the symmetric case.

$\#\texttt{count}\{A\} < k$. Suppose that there exists a set $A' \subseteq A$ such that for each $\langle \bar{t} : a \rangle \in A'$, a is true in I and $|A'| \geq k$, then $\#\texttt{count}\{A\} < k$ is inferred to be false and its reasons are set to $\mathcal{S}_{A'}$ Conversely,

[5]Note that, for the aggregates considered in this paper, $f(S) \leq T$ (resp. $f(S) \geq T$) is equivalent to $f(S) < T + 1$ (resp. $f(S) > T - 1$), and $f(S) = T$ can be replaced by the conjunction $f(S) < T + 1$, $f(S) > T - 1$.

[6]This can happen in our setting as a consequence of the application of either *contraposition for true head* or *contraposition for false head* propagation rules, see [35].

suppose that there exists a set $A' \subseteq A$ such that for each $\langle \bar{t} : a\rangle \in A'$, a is false in I and $|A'| > |A| - k$, then we infer that $\#\mathtt{count}\{A\} < k$ is true and we set its reason to $\mathcal{C}_{A'}$.

Example 4.1. Suppose that the input program contains the rule:

$$h \coloneq c, \#\mathtt{count}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 1.$$

and suppose also that the current partial interpretation I contains both $a(1)$ and $a(2)$. Then, we have that there exists a set, namely $A' = \{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle\}$ (which contains true atoms), that ensures that the aggregate function value is at least greater than the guard; thus, $\#\mathtt{count}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 1$ is derived to be false, and its reason is set to $\mathcal{S}_{A'} = R(a(1)) \cup R(a(2))$. ▮

Note that the specifications described above leaves some freedom for an implementation, since there might exist several sets A' that satisfy the respective properties. A possibility with more information value would be to consider all of these sets, which however might be costly to compute. A compromise solution is to create one such A' by iterating over the set A, adding suitable elements to an initially empty A' until the condition is met. The latter option has been implemented in our prototype (cf. Section 6), also for cases described below, where analogous options are available.

$\#\mathtt{min}\{A\} > k$. Let A' be the set of all pairs $\langle v, \bar{t} : a\rangle \in A$ such that $v \leq k$. If for each $\langle v, \bar{t} : a\rangle \in A'$, a is false in I, and if there exists also a pair $\langle v, \bar{t} : a\rangle \in A$ such that a is true in I then $\#\mathtt{min}\{A\} > k$ is derived to be true and we set its reason to $\mathcal{C}_{A'} \cup R(a)$, otherwise (the function is undefined over the empty set) $\#\mathtt{min}\{A\} > k$ is derived to be false with reason $\mathcal{C}_{A \setminus A'}$. Conversely, suppose there exists a pair $\langle v, \bar{t} : a\rangle \in A$ such that a is true in I and $v \leq k$, then we infer that $\#\mathtt{min}\{A\} > k$ is false and set its reason to $R(a)$.

$\#\mathtt{min}\{A\} < k$. Let A' be the set of all pairs $\langle v, \bar{t} : a\rangle \in A$ such that $v < k$. If for each $\langle v, \bar{t} : a\rangle \in A'$, a is false in I, then $\#\mathtt{min}\{A\} < k$ is derived to be false and we set its reason to $\mathcal{C}_{A'}$. Conversely, suppose there exists a pair $\langle v, \bar{t} : a\rangle \in A$ such that a is true in I and $v < k$, then we infer that $\#\mathtt{min}\{A\} < k$ is true and set its reason to $R(a)$.

Example 4.2. Suppose that the input program contains the rule:

$$h \coloneq c, \#\mathtt{min}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 2.$$

and suppose also that the current partial interpretation I contains both $a(1)$ and $a(3)$. Then, we have that there exists a pair, namely $\langle 1 : a(1)\rangle$, that ensures that the minimum is 1 which is smaller than the guard (here $1 < k = 2$); thus, $\#\mathtt{min}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 2$ is derived to be true and its reason is set to $R(a(1))$. ▮

$\#\mathtt{max}\{A\} < k$. Suppose there exists a pair $\langle v, \bar{t} : a\rangle \in A$ such that a is true in I and $v \geq k$, then we infer that $\#\mathtt{max}\{A\} < k$ is false and set its reason to $R(a)$. Conversely, let A' be the set of all pairs $\langle v, \bar{t} : a\rangle \in A$ such that $v \geq k$. If for each $\langle v, \bar{t} : a\rangle \in A'$, a is false in I, and if there exists also a pair $\langle v, \bar{t} : a\rangle \in A \setminus A'$ such that a is true in I then $\#\mathtt{max}\{A\} < k$ is derived to be true and we set its reason to $\mathcal{C}_{A'} \cup R(a)$, otherwise (the function is undefined over the empty set) $\#\mathtt{max}\{A\} < k$ is derived to be false with reason $\mathcal{C}_{A \setminus A'}$.

$\#\texttt{max}\{A\} > k$**.** Suppose there exists a pair $\langle v, \bar{t} : a\rangle \in A$ such that a is true in I and $v > k$, then we infer that $\#\texttt{max}\{A\} > k$ is true and set its reason to $R(a)$. Conversely, let A' be the set of all pairs $\langle v, \bar{t} : a\rangle \in A$ such that $v > k$, and suppose that for each $\langle v, \bar{t} : a\rangle \in A'$, a is false in I, then $\#\texttt{max}\{A\} > k$ is derived to be false and we set its reason to $\mathcal{C}_{A'}$.

Example 4.3. Suppose that the input program contains the rule:

$$h \coloneq c, \#\texttt{max}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} > 2.$$

and suppose also that the current partial interpretation I contains $\text{not }a(1)$, $\text{not }a(2)$, and $\text{not }a(3)$. Then, we have that the entire set $A = \{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\}$ contains only false atoms; thus, the aggregate atom is is derived to be false, and its reason is set to $\mathcal{C}_A = R(a(1)) \cup R(a(2)) \cup R(a(3))$. ∎

$\#\texttt{sum}\{A\} > k$**.** Suppose that there exists a set $A' \subseteq A$ such that for each $\langle v, \bar{t} : a\rangle \in A'$ a is false in I and $\Sigma_{[v|\langle v,\bar{t}:a\rangle \in A]} v - \Sigma_{[v|\langle v,\bar{t}:a\rangle \in A']} v \leq k$,[7] then $\#\texttt{sum}\{A\} > k$ is false and we set its reason to $\mathcal{C}_{A'}$. Conversely, suppose that there exists a set $A' \subseteq A$ such that for each $\langle v, \bar{t} : a\rangle \in A'$, a is true in I and $\Sigma_{[v|\langle v,\bar{t}:a\rangle \in A']} v > k$, then $\#\texttt{sum}\{A\} > k$ is true and its reason is $\mathcal{S}_{A'}$.

$\#\texttt{sum}\{A\} < k$**.** Suppose that there exists a set $A' \subseteq A$ such that for each $\langle v, \bar{t} : a\rangle \in A'$ a is true in I and $\Sigma_{[v|\langle v,\bar{t}:a\rangle \in A']} v \geq k$, then $\#\texttt{sum}\{A\} < k$ is false and we set its reason to $\mathcal{S}_{A'}$. Conversely, suppose that there exists a set $A' \subseteq A$ such that for each $\langle v, \bar{t} : a\rangle \in A'$, a is false in I and $\Sigma_{[v|\langle v,\bar{t}:a\rangle \in A]} v - \Sigma_{[v|\langle v,\bar{t}:a\rangle \in A']} v < k$, then $\#\texttt{sum}\{A\} < k$ is true and its reason is $\mathcal{C}_{A'}$.

Example 4.4. Suppose that the input program contains the rule:

$$h \coloneq c, \#\texttt{sum}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 4.$$

and suppose also that the current partial interpretation I contains $a(1)$, $a(2)$ and $\text{not }a(3)$. Then there exists a set, namely $A' = \{\langle 3 : a(3)\rangle\}$ (which contains a false atom), that ensures that the function value cannot be greater than the guard (here $3 > 6 - 4$); thus, $\#\texttt{sum}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 4$ is derived to be false and its reason is set to $R(\text{not }a(3))$. ∎

4.2. Backward Inference

This kind of propagation rules apply when an aggregate literal $f(A)\Theta k$, $\Theta \in \{<, >\}$ has been derived true (or false), and there is *a unique way*[8] to satisfy it by inferring that some atom belonging to A is true or false. For example, suppose that I is empty and consider the program:

$$\coloneq \text{not }h. \qquad h : -\#\texttt{count}\{\langle 1 : a\rangle, \langle 1 : b\rangle\} > 1.$$

During propagation we first infer h to be true for satisfying the constraint, and then, in order to satisfy the rule, also the aggregate literal is inferred to be true (independently of its aggregate set). At this point, backward propagation can happen, since there is a unique way to satisfy the aggregate literal: infer both a and b to be true.

[7]Recall that by $[\ldots]$ we denote a multiset.

[8]Since the propagation process must be *deterministic*.

Note that, as far as the reason calculus is concerned, literals are inferred to be true or false by backward inference because both the aggregate literal is true/false and a set of atoms in A (whose elements are either true or false) made the process deterministic; thus, the reason for each atom a inferred by backward inference is set to $R(a) = R(f(A)\Theta k) \cup \mathcal{C}_A \cup \mathcal{S}_A$.

The following paragraphs report sufficient conditions for applying backward inference in the case of the aggregates supported by DLV. Since conditions for $f(A) > k$ to be true (resp. false) coincide with the ones of $f(A) < k+1$ to be false (resp. true), only one of the two cases is reported for each aggregate.

Definition 4.1. Given a ground set A and a partial interpretation I, let T_A be the set $\{\langle \bar{t}_i : a_i\rangle \in A$ such that a_i is true with respect to $I\}$, and F_A be the set $\{\langle \bar{t}_i : a_i\rangle \in A$ such that a_i is false with respect to $I\}$.

$\#\mathtt{count}\{A\} < k$. Suppose that both $\#\mathtt{count}\{A\} < k$ is true with respect to I and $|T_A| = k-1$, then all undefined atoms a_i such that $\langle \bar{t}_i : a_i\rangle \in A$ are made false. Conversely, suppose $\#\mathtt{count}\{A\} < k$ is false with respect to I and $|A| - |F_A| = k$, then all undefined atoms a_i such that $\langle \bar{t}_i : a_i\rangle \in A$ are made true.

Example 4.5. Suppose that the input program contains the rule:

$$h \text{ :- } \#\mathtt{count}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 1.$$

and suppose also that the current partial interpretation I contains both h and $\#\mathtt{count}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 1$. Since we have that $|T_A| = 0 = 1-1$, we infer $a(1), a(2)$, and $a(3)$ to be false, and set their reason to $R(\#\mathtt{count}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 1)$. ▮

$\#\mathtt{min}\{A\} < k$. Suppose that $\#\mathtt{min}\{A\} < k$ is true with respect to I and that there is only one $\langle v, \bar{t} : a\rangle \in A$ such that $v < k$ and a is undefined with respect to I; suppose also that all the remaining $\langle v_i, \bar{t}_i : a_i\rangle \in A$ such that $v_i < k$ are such that a_i is false with respect to I. Then, a is inferred to be true. Conversely, suppose that $\#\mathtt{min}\{A\} < k$ is false with respect to I, there is no $\langle v, \bar{t} : a\rangle \in A$ such that $v < k$ with a true with respect to I, and, in addition, suppose that either: (i) there exist $\langle v', \overline{t'} : a'\rangle \in A$ such that $v' > k$ and a' is true with respect to I or (ii) there is only one $\langle v'', \overline{t''} : a''\rangle \in A$ such that $v'' > k$ with a'' undefined with respect to I. Then, all the a_i such that $\langle v_i, \bar{t}_i : a_i\rangle \in A$ and $v_i < k$ are inferred to be false, and, if case (ii) holds, also a'' is made true with respect to I.

Example 4.6. Suppose that the input program contains the rule:

$$h \text{ :- } \#\mathtt{min}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 2.$$

and suppose also that both $a(2)$, and $\#\mathtt{min}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 2.$ are false with respect to the current partial interpretation. It can be easily verified that condition (ii) holds,[9] then $a(1)$ is inferred to be false, $a(3)$ is inferred to be true, and $R(a(1))$ and $R(a(3))$ are both set to $R(\#\mathtt{min}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 2) \cup R(a(2))$. ▮

[9]Note that the aggregate atom is false, $a(1)$ is the only undefined atom that can make it true, and $a(3)$ is the only undefined atom that can make it false.

$\#\mathtt{max}\{A\} < k$. Suppose that $\#\mathtt{max}\{A\} < k$ is false with respect to I, there is only one $\langle v, \bar{t} : a\rangle \in A$ such that $v > k$ with a undefined with respect to I, while all the remaining $\langle v_i, \bar{t}_i : a_i\rangle \in A$ such that $v_i > k$ are such that a_i is false with respect to I, then a is inferred to be true. Conversely, suppose that both $\#\mathtt{max}\{A\} < k$ is true with respect to I and there is no $\langle v, \bar{t} : a\rangle \in A$ such that $v \geq k$ and a is true with respect to I, and, in addition, suppose that one of the following condition holds: (i) there exist $\langle v', \overline{t'} : a'\rangle \in A$ such that $v' < k$ and a' is true with respect to I; or (ii) there is only one $\langle v'', \overline{t''} : a''\rangle \in A$ such that $v'' < k$ with a'' undefined with respect to I. Then, in case (ii) holds a'' is inferred to be true, and all the remaining undefined a_i such that $\langle v_i, \bar{t}_i : a_i\rangle \in A$ ($a_i \neq a''$) and $v_i < k$ are inferred to be false.

Example 4.7. Suppose that the input program contains the rule:

$$h \text{ :- } \#\mathtt{max}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 3.$$

and suppose also that $\#\mathtt{max}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 3$ is false, $a(1)$ is true, while $a(2)$ and $a(3)$ are undefined (with respect to the current partial interpretation). Then, $a(3)$ is inferred to be true, (note that this is the only way for ensuring that the aggregate atom is false) and $R(a(3))$ is set to $R(a(1)) \cup R(\#\mathtt{max}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 3)$. ∎

$\#\mathtt{sum}\{A\} < k$. Let us denote by $S(X)$ the sum $\sum_{\langle v_i, \bar{t}_i : a_i\rangle \in X} v_i$, and suppose that $\#\mathtt{sum}\{A\} < k$ is true with respect to I and $S(T_A) = k - 1$, then all undefined atoms in A are made false. Conversely, suppose that $\#\mathtt{sum}\{A\} < k$ is false in I and $S(A) - S(F_A) = k$, then all undefined atoms in A are made true.

Example 4.8. Suppose that the input program contains the rule:

$$h \text{ :- } \#\mathtt{sum}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 4.$$

and suppose also that both $a(3)$ and $\#\mathtt{sum}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 4$ are true, while $a(1)$ and $a(2)$ are undefined with respect to the current partial interpretation. It can be easily verified that $S(T_A) = 3$, thus both $a(1)$ and $a(2)$ are inferred to be false, and their reason is set to $R(a(1)) = R(a(2)) = R(a(3)) \cup R(\#\mathtt{sum}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 4)$. ∎

5. Heuristics in the Presence of Aggregates

The efficiency of the answers set computation process strongly depends on heuristics used for choosing the branching variables. In the following sections, we describe both the look-ahead [12] and the look-back [30] heuristics employed by the DLV system; and, in particular, we point out how the presence of aggregates can be taken into account in both cases.

5.1. Look-ahead Heuristic

The look-ahead heuristic of DLV [12] was shown to be very effective on many relevant problems, and it is still the default in the standard distribution.

In general, in a look-ahead heuristic each possible choice literal is tentatively assumed, its consequences are computed, and some characteristic values on the result are recorded. Hence, according to a heuristic criterion based on these values, the choice is determined.

The main heuristic criterion employed by DLV exploits a peculiar property of ASP, called *supportedness*. Basically, for each true atom A of an answer set I, there exists a rule r of the program such that the body of r is true with respect to I and A is the only true atom in the head of r. Since answer sets are supported interpretations, an ASP system must eventually converge to a situation in which there are no *Unsupported True (UT)* atoms, i.e, true atoms missing a supporting rule. Following this observation, the idea is to prefer the choice of those literals that minimize the number of *UnsupportedTrue (UT)* atoms. In more detail, the heuristic of DLV "layers" several criteria, and, in particular, for each literal L the following measures (with respect to the interpretation resulting from the propagation of L) are considered: $UT(L)$, $UT_2(L)$, $UT_3(L)$, $Sat(L)$, $DS(L)$; where UT is the number of UT atoms; UT_2 and UT_3 are, respectively the number of UT atoms occurring in the heads of exactly 2 and 3 unsatisfied rules; $Sat(L)$ is the total number of satisfied rules; and DS is the degree of supportedness (namely, the average number of supporting rules for the true non-head-cycle-free atoms). The heuristic of DLV considers $UT(L)$, $UT_2(L)$ and $UT_3(L)$ in a prioritized way, to favor atoms yielding interpretations with fewer $UT/UT_2/UT_3$ atoms (which should more likely lead to a supported model). If all UT counters are equal, then the heuristic considers the total number $Sat(L)$ of rules which are satisfied; finally, literals with higher degree of supportedness are preferred (this last criterion has been added in order to deal with hard problems, see [15]). Moreover, the heuristic is "balanced", that is, the heuristic values of a literal L depend on both the effect of taking L and $\text{not}.L$.

Example 5.1. Consider the following program:

$$a \vee b \vee c. \quad d \vee e \vee f. \quad :\!-\, \text{not}\; w. \quad w :\!-\, a. \quad w :\!-\, d.$$

$$a \vee z :\!-\, w. \quad b \vee z :\!-\, w. \quad :\!-\, d, z. \quad :\!-\, a, z.$$

and let the current interpretation $I = \{w, \text{not}\; x\}$. Notice that w is true but misses a supporting rule, that is, it is UT. Moreover, a and d are the best choices according to the look-ahead heuristics of DLV as only assuming their truth can eliminate the UT w. Indeed, anything apart from a or d would be a poor choice. ∎

Note that this heuristic does not need to be modified in order to take the presence of aggregate literals into account; indeed, the values for all the above-mentioned counters are directly computed during the propagation of aggregate literals.

5.2. Look-back Heuristics

Look-back heuristics, which have been originally exploited in SAT solvers like CHAFF [31] (where the heuristic is called VSIDS), have also been considered for DLV, in conjunction with backjumping, leading to positive results [30].

The intuition behind this kind of heuristics is to periodically update a numeric value $V(l)$, associated to each literal l, indicating the number of occurrences of l in a conflict. Basically, this heuristic favors the choice of literals which are more likely to lead to inconsistent sub-branches, which in general has the effect of more likely exploring a smaller search tree. In detail, after having chosen k literals, $V(l)$

is updated for each l as follows: $V(l) := V(l)/A_g + I(l)$, where $I(l)$ is the number of inconsistencies l has been a reason for (since the most recent heuristic value update), and A_g is the "aging" factor that allows for giving more importance to recent data. Whenever a choice has to be made among undefined literals, the positive literal with the largest $V(l)$ will be chosen. If several literals have the same $V(l)$, then negative literals are preferred over positive ones, but among negative and positive literals having the same $V(l)$, the ordering will be random.

A key factor of this type of heuristic is the initialization of the weights of the literals [30], to be updated by the reason calculus during the search. Indeed, at the beginning of the search, the solver has no information about inconsistencies, and all $V(l)$ initially will be 0, and so a random choice would be taken. A common practice is to initialize $V(l)$ values with the number of occurrences of l in the input (ground) program. However, this strategy, originally devised for aggregate-free programs, does not take properly into account the presence of aggregates in the program, which can be exploited for guiding the search instead. To this end, we propose two different new heuristics: the first and simpler criterion (called *size-based heuristic*) is based on the size of the aggregate sets, while the second (called *equivalent-program heuristic*) tries to estimate more precisely the effect of aggregate literals in a program by exploiting the following idea: aggregates can be simulated by replacing the original program with an equivalent aggregate-free one, so that standard techniques can be used for counting occurrences. However, physically replacing a program by an aggregate-free one is impractical for a number of reasons, such as the additional space requirement or the loss of structure, which would quite clearly outweigh the benefit of having a smarter heuristic. Our approach is therefore to compute (or in some cases estimate) these values without materializing the equivalent program, as described below.

The generic method for computing the values of $V(l)$ is by iterating on the input rules. Whenever a standard literal l is encountered, $V(l)$ is increased by 1, while when an aggregate literal $f(A)\Theta k$ is encountered, the value of $V(f(A)\Theta k)$ is increased by $\mathcal{E}_{occ}(f(A)\Theta k)$, which is the heuristically estimated weight of the aggregate literal, and for each $\langle \bar{t} : a \rangle \in A$, $V(a)$ is incremented by $\mathcal{E}_{occ}(a)$ (again determined by the chosen heuristic).

Size-based Heuristic. This heuristic is based on a simple principle: if an aggregate literal $f(A)\Theta k$ occurs in the input program, its "weight" is given by the size of its aggregate set; moreover, also the occurrences of each atom a such that $\langle \bar{t} : a \rangle \in A$ have to be added to $V(a)$ (to take into account the role played by a in the aggregate). In particular, we have that $\mathcal{E}_{occ}(f(A)\Theta k) = |A|$, and $\mathcal{E}_{occ}(a)$ is set to the number of occurrences of $\langle \bar{t} : a \rangle$ in A.

Example 5.2. Consider the following program:

$$r_1: \ \ h \,\texttt{:-}\, \#\texttt{count}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} > 1.$$

$$r_2: \ \ b \,\texttt{:-}\, c, d, \#\texttt{sum}\{\langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle, \langle 4 : a(4)\rangle, \langle 5 : a(5)\rangle\} > 3.$$

$$r_3: \ \ a(1). \quad\ r_4: \ a(2).$$

according to the size-based heuristic, we have that:

- $V(h) = V(b) = 1$, because h and b occur once in the head of r_1 and r_2, respectively;
- $V(c) = V(d) = 1$, because c and d both occur once in the body of r_2;
- $V(a(1)) = 2$, because $a(1)$ occurs both in the aggregate literal of rule r_1 and in fact r_3;

- $V(a(2)) = 3$, because $a(2)$ occurs in both aggregate literals of the program and in fact r_4;
- $V(a(3)) = 2$, because $a(3)$ occurs in both aggregate literals of the program;
- $V(a(4)) = V(a(5)) = 1$, because $a(4)$ and $a(5)$ occur in the aggregate literals of rule r_2.
- $V(\#\texttt{count}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} > 1) = 3$ and $V(\#\texttt{sum}\{\langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle, \langle 4 : a(4)\rangle, \langle 5 : a(5)\rangle\} > 3) = 4$, because the corresponding aggregate sets have cardinality 3 and 4, respectively.

∎

Equivalent-program Heuristic. This heuristic computes a somewhat more precise estimation of the impact of aggregates by also taking into account their semantics. The idea is to *virtually* replace each occurrence of an aggregate atom of the form $f(A)\Theta k$ with a fresh-new predicate h, and "define" h by means of a standard subprogram which emulates $f(A)\Theta k$.[10] As mentioned earlier, this equivalent program does not have to be materialized in memory; in Table 1 we summarize the formulas that allow for directly computing the additional number of occurrences $\mathcal{E}_{occ}(l)$ that would have been determined by replacing $f(A)\Theta k$ by its equivalent subprogram, for each literal l occurring in a given aggregate literal $f(A)\Theta k$. Since h replaces $f(A)\Theta k$ in the aggregate-free program, we set $\mathcal{E}_{occ}(f(A)\Theta k)$ to $\mathcal{E}_{occ}(h)$. In the following paragraphs, we provide both the description of the considered equivalent programs, and detail on how the results in Table 1 have been determined.

$\#\texttt{min}\{A\} < k$. The equivalent standard program for this aggregate atom contains a rule of the type h :- a_i, for each $\langle v_i, \bar{t}_i : a_i\rangle \in A$ $(1 \leq i \leq n)$ such that $v_i < k$. In this way, h is true if at least one of the a_i with $v_i < k$ is true, that is, if the minimum computed by the aggregate is less than k. Thus, $\mathcal{E}_{occ}(h) = |\{v_i : \langle v_i, \bar{t}_i : a_i\rangle \in A, v_i < k\}| + 1$, and for each a_i such that $\langle v_i, \bar{t}_i : a_i\rangle \in A$, $\mathcal{E}_{occ}(a_i) = 1$ if $v < k$, otherwise $\mathcal{E}_{occ}(a_i) = 0$.

Example 5.3. Consider the following program:

$$f \text{ :- } c, \#\texttt{min}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 3. \quad a(1).\ \ a(2).$$

Its aggregate-free version is:

$$f \text{ :- } c, h. \quad h \text{ :- } a(1). \quad h \text{ :- } a(2). \quad a(1).\ \ a(2).$$

thus, $V(c) = V(f) = 1$, $V(h) = \mathcal{E}_{occ}(h) = 3$, $V(a(1)) = \mathcal{E}_{occ}(a(1)) + 1 = 2$, $V(a(2)) = \mathcal{E}_{occ}(a(2)) + 1 = 2$, and $V(a(3)) = \mathcal{E}_{occ}(a(3)) = 0$.[11] ∎

$\#\texttt{min}\{A\} > k$. The equivalent standard program for this aggregate atom contains: a single rule of the form h :- $\text{not } b_1 \ldots, \text{not } b_m, h_{aux}$., where $[b_1, \ldots, b_m] = [a_i \mid \langle v_i, \bar{t}_i : a_i\rangle \in A$ and $v_i \leq k]$; and, (possibly) several auxiliary rules of the form h_{aux} :- a_j, one for each $\langle v_j, \bar{t}_j : a_j\rangle \in A$ such that $v_j > k$.

[10] Actually, a distinct not-appearing-elsewhere-in-the-program predicate name $h_{f(A)\Theta k}$ should be employed for each aggregate literal $f(A)\Theta k$ occurring in P. With a small abuse of notation, we omit the additional subscript for obtaining a simpler notation. Note also that equivalence with subprograms in general holds only in the stratified setting, but could also serve as an approximation also in non-recursive settings.

[11] In the following examples, we only report $\mathcal{E}_{occ}(.)$.

Note that, in the obtained equivalent program, h is true if all the a_i (associated with a $v_i \leq k$) are false, and at least one a_j (with $v_j > k$) is true, that is if the actual minimum computed by the aggregate is greater than k (note that the auxiliary rules are needed because $I(min(\emptyset)) = \bot$). Thus, in this case: $\mathcal{E}_{occ}(h) = 2$, and for each a_i such that $\langle v_i, \bar{t}_i : a_i \rangle \in A$ we have that $\mathcal{E}_{occ}(a_i) = 1$ if $v_i > k$, otherwise (if $v_i \leq k$) we have that $\mathcal{E}_{occ}(\text{not } a_i) = 1$.

Example 5.4. Consider the following program:

$$f \coloneq c, \#\mathtt{min}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} > 2. \quad a(1).\ a(2).$$

its equivalent version is:

$$f \coloneq c, h. \quad h \coloneq \text{not } a(1), \text{not } a(2), h_{aux}. \quad h_{aux} \coloneq a(3). \quad a(1).\ a(2).$$

thus, $\mathcal{E}_{occ}(h) = 2$, $\mathcal{E}_{occ}(a(3)) = 1$, and $\mathcal{E}_{occ}(\text{not } a(1)) = \mathcal{E}_{occ}(\text{not } a(2)) = 1$. ∎

$\#\mathtt{max}\{A\} < k$. The equivalent standard program for this aggregate contains a single rule of the form $h \coloneq \text{not } b_1, \ldots, \text{not } b_m, h_{aux}.$, where $[b_1, \ldots, b_m] = [a_i \mid \langle v_i, \bar{t}_i : a_i \rangle \in A \text{ and } v_i \geq k]$; and (possibly) several auxiliary rules of the form $h_{aux} \coloneq a_j$ for each $\langle v_j, \bar{t}_j : a_j \rangle \in A$ such that $v_j < k$.

Note that, in the obtained program, h is true if all the a_i (with $v_i \geq k$) are false, and at least one a_j (with $v_j < k$) is true, that is if the maximum computed by the aggregate is less than k in the current interpretation (note that, again, auxiliary rules are needed because $I(max(\emptyset)) = \bot$). Thus, in this case: $\mathcal{E}_{occ}(h) = 2$, and for each a_i such that $\langle v_i, \bar{t}_i : a_i \rangle \in A, v_i \geq k$ we have that $\mathcal{E}_{occ}(\text{not } a_i) = 1$, otherwise (if $v_i < k$) we have that $\mathcal{E}_{occ}(a_i) = 1$.

Example 5.5. Consider the following program:

$$f \coloneq c, \#\mathtt{max}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 2. \quad a(1).\ a(2).$$

its equivalent version is:

$$f \coloneq c, h. \quad h \coloneq \text{not } a(2), \text{not } a(3), h_{aux}. \quad h_{aux} \coloneq a(1). \quad a(1).\ a(2).$$

thus, $\mathcal{E}_{occ}(h) = 2$, $\mathcal{E}_{occ}(a(1)) = 1$, and $\mathcal{E}_{occ}(\text{not } a(2)) = \mathcal{E}_{occ}(\text{not } a(3)) = 1$. ∎

$\#\mathtt{max}\{A\} > k$. The equivalent standard program for this aggregate contains a rule of the type $h \coloneq a_i$, for each $\langle v_i, \bar{t}_i : a_i \rangle \in A$ such that $v_i > k$ ($1 \leq i \leq n$). In this way, h is true if at least one of the a_i with $v_i > k$ is true, that is if the maximum computed by the aggregate is more than k in the current interpretation. Thus, $\mathcal{E}_{occ}(h) = |\{v_i : \langle v_i, \bar{t}_i : a_i \rangle \in A, v_i > k\}| + 1$, and for each a_i such that $\langle v_i, \bar{t}_i : a_i \rangle \in A$ $\mathcal{E}_{occ}(a_i) = 1$ if $v_i > k$, otherwise $\mathcal{E}_{occ}(a_i) = 0$.

Example 5.6. Consider the following program:

$$f \coloneq c, \#\mathtt{max}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} > 1. \quad a(1).\ a(2).$$

Its aggregate-free version is:

$$f \coloneq c, h. \quad h \coloneq a(2). \quad h \coloneq a(3). \quad a(1).\ a(2).$$

thus, $\mathcal{E}_{occ}(h) = 3$, and $\mathcal{E}_{occ}(a(2)) = \mathcal{E}_{occ}(a(3)) = 1$. ∎

$\#\mathtt{count}\{A\} < k$. For this kind of aggregate atoms we have to distinguish two different cases. In particular, if $k > |A|$ the value of the aggregate cannot exceed the guard; thus there is no interpretation in which the aggregate is false. In this case, the equivalent program is made of a single fact h, and: $\mathcal{E}_{occ}(h) = 2$, $\mathcal{E}_{occ}(a_i) = \mathcal{E}_{occ}(\text{not } a_i) = 0$, for each a_i such that $\langle \bar{t}_i : a_i \rangle \in A$.

In the other case (in which $k \leq |A|$), we consider a more involved subprogram denoted by $P^<$ that is obtained as follows. Let $P_i^<$ be the program containing a rule of the form h :- $\text{not } a_1, \text{not } a_2, \ldots, \text{not } a_{i-1}$ for each $(i-1)$-combination of elements in the aggregate set A, where $a_1, \ldots, a_{i-1}$ are the atoms they contain. Intuitively, h will be derived to be true in $P_i^<$ if there are at least $i-1$ false conjunctions (that is if $\#\mathtt{count}\{A\} < i$.).

Then, the subprogram $P^<$ is obtained by taking the union of all $P_i^<$ such that $i \leq k$, that is $P^< = \cup_{i \leq k} P_i^<$. Intuitively, in $P^<$ we consider the contribution given by each of the possible combinations of $1 < i < k$ atoms of the aggregate set $(a_1, \ldots, a_{i-1})$ that, if true (i.e., $a_1, \ldots, a_{i-1} \in I$), can make the count less than the guard (that is original aggregate to be true). It can be shown that in this case we obtain: $\mathcal{E}_{occ}(h) = \sum_{i=0}^{k-1} \binom{|A|}{i} + 1$, $\mathcal{E}_{occ}(a_i) = 0$, and $\mathcal{E}_{occ}(\text{not } a_i) = \sum_{i=0}^{k-1} \binom{|A|-1}{i}$.

Example 5.7. Consider the following program:

$$f \text{ :- } c, \#\mathtt{count}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} < 2. \quad a(1).\ a(2).$$

Its aggregate-free version is:

$$f \text{ :- } c, h. \quad a(1).\ a(2). \quad h \text{ :- } \text{not } a(1), \text{not } a(2), \text{not } a(3).$$

$$h \text{ :- } \text{not } a(1), \text{not } a(2). \quad h \text{ :- } \text{not } a(1), \text{not } a(3). \quad h \text{ :- } \text{not } a(2), \text{not } a(3).$$

thus, $\mathcal{E}_{occ}(h) = \binom{3}{0} + \binom{3}{1} + 1 = 5$, and $\mathcal{E}_{occ}(\text{not } a(1)) = \mathcal{E}_{occ}(\text{not } a(2)) = \mathcal{E}_{occ}(\text{not } a(3)) = \binom{2}{0} + \binom{2}{1} = 3$. ∎

$\#\mathtt{count}\{A\} > k$. For this kind of aggregate atom, we again distinguish two different cases. In particular, if $|A| \leq k$ the value of the aggregate cannot satisfy the guard; thus, there is no interpretation in which the aggregate is true. In this case, the equivalent program would be made of a single constraint :- h, and: $\mathcal{E}_{occ}(h) = 2$, $\mathcal{E}_{occ}(a_i) = \mathcal{E}_{occ}(\text{not } a_i) = 0$, for each a_i such that $\langle \bar{t}_i : a_i \rangle \in A$.

In the other case (in which $|A| > k$), we consider a subprogram denoted by $P^>$ that is obtained as follows. Let $P_i^>$ be the program containing a rule of the form h :- $a_1, a_2, \ldots, a_{i+1}$ for each possible combination of atoms obtained by taking $i+1$ different atoms from the $|A|$ different ones available in the aggregate set A. Intuitively, h will be derived to be true in $P_i^>$ if there are at least $i+1$ true atoms (that is if $\#\mathtt{count}\{A\} > i$.).

Then, the subprogram $P^>$ is obtained by taking the union of all $P_i^>$ such that $i \leq k$, that is $P^> = \cup_{i \leq k} P_i^>$. Intuitively, in $P^>$ we consider the contribution given by each of the possible combinations of true atoms making the original aggregate true. We obtain that: $\mathcal{E}_{occ}(h) = \sum_{i=k}^{|A|-1} \binom{|A|}{i+1} + 1 = \sum_{i=0}^{|A|-k-1} \binom{|A|}{i} + 1$, $\mathcal{E}_{occ}(a_i) = \sum_{i=k+1}^{|A|} \binom{|A|-1}{i-1} = \sum_{i=1}^{|A|-k} \binom{|A|-1}{i-1}$, and $\mathcal{E}_{occ}(\text{not } a_i) = 0$.

Example 5.8. Consider the following program:

$$f \text{ :- } c, \#\mathtt{count}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} > 1. \quad a(1).\ a(2).$$

Its aggregate-free version is:

$$f \text{ :- } c, h. \quad a(1).\ a(2). \quad h \text{ :- } a(1), a(2), a(3).$$

$$h \text{ :- } a(1), a(2). \quad h \text{ :- } a(1), a(3). \quad h \text{ :- } a(2), a(3).$$

thus, $\mathcal{E}_{occ}(h) = \binom{3}{0} + \binom{3}{1} + 1 = 5$, and $\mathcal{E}_{occ}(a(1)) = \mathcal{E}_{occ}(a(2)) = \mathcal{E}_{occ}(a(3)) = \binom{2}{0} + \binom{2}{1} = 3$. ∎

$\#\mathtt{sum}\{A\}\Theta k$. $\Theta \in [<, >]$. Equivalent programs in the case of $\#sum$ are quite involved, rendering the computation of the exact values fairly inefficient (many binomial coefficients have to be calculated). Therefore we decided to approximate the corresponding heuristic value, replacing $\#sum\{A\}$ by $\#count\{A^*\}$ where A^* contains v_i different elements for each $\langle v_i, \bar{t}_i : a_i \rangle \in A$.

Example 5.9. Consider the following aggregate atom:

$$h \text{ :- } \#\mathtt{sum}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} > 5$$

it is approximated by:

$$\#\mathtt{count}\{\langle 1, 1 : a(1)\rangle, \langle 1, 1 : a(2)\rangle, \langle 1, 2 : a(2)\rangle,$$

$$\langle 1, 1 : a(3)\rangle, \langle 1, 2 : a(3)\rangle, \langle 1, 3 : a(3)\rangle\} > 5$$

Thus: $\mathcal{E}_{occ}(h) = \binom{6}{0} = 1$, $\mathcal{E}_{occ}(a(1)) = \mathcal{E}_{occ}(a(2)) = \mathcal{E}_{occ}(a(3)) = \binom{5}{0} = 1$. ∎

6. Experimental Analysis

We have implemented the techniques described in Sections 3-5 as an extension of the system DLV. In this section we report on the experimental evaluation of the various versions thus obtained.

6.1. Compared Methods

For our experiments, we have compared three versions of DLV, which differ on the employed heuristic for dealing with aggregates, namely:

- DLV, the standard DLV system, employing the heuristic based on look-ahead described in Section 5.1.

- DLV.BJA.VS.SIZE, DLV with backjumping on aggregates and (look-back) size-based heuristic.

- DLV.BJA.VS.EQ, DLV with backjumping on aggregates and (look-back) equivalent-program heuristic.

	$\#\texttt{count}\{A\} < k$	$\#\texttt{min}\{A\} < k$	$\#\texttt{max}\{A\} < k$	$\#\texttt{sum}\{A\} < k$
$\mathcal{E}_{occ}(h)$	$\begin{cases} \sum_{i=0}^{k-1} \binom{\lvert A\rvert}{i} + 1 & k \leq \lvert A\rvert \\ 2 & else \end{cases}$	$\lvert\{v_i \mid \langle v_i, \bar{t}_i : a_i\rangle \in A, v_i < k\}\rvert + 1$	2	$\begin{cases} \sum_{i=0}^{k-1} \binom{\lvert A^*\rvert}{i} + 1 & k \leq \lvert A^*\rvert \\ 2 & else \end{cases}$
$\mathcal{E}_{occ}(a_i)$	0	$\begin{cases} 1 & v_i < k \\ 0 & else \end{cases}$	$\begin{cases} 1 & v_i < k \\ 0 & else \end{cases}$	0
$\mathcal{E}_{occ}(\text{not } a_i)$	$\begin{cases} \sum_{i=0}^{k-1} \binom{\lvert A\rvert-1}{i} & k \leq \lvert A\rvert \\ 0 & else \end{cases}$	0	$\begin{cases} 0 & v_i < k \\ 1 & else \end{cases}$	$\begin{cases} \sum_{i=0}^{k-1} \binom{\lvert A^*\rvert-1}{i} & k \leq \lvert A^*\rvert \\ 0 & else \end{cases}$

	$\#\texttt{count}\{A\} > k$	$\#\texttt{min}\{A\} > k$	$\#\texttt{max}\{A\} > k$	$\#\texttt{sum}\{A\} > k$
$\mathcal{E}_{occ}(h)$	$\begin{cases} \sum_{i=0}^{\lvert A\rvert-k-1} \binom{\lvert A\rvert}{i} + 1 & k \leq \lvert A\rvert \\ 2 & else \end{cases}$	2	$\lvert\{v_i \mid \langle v_i, \bar{t}_i : a_i\rangle \in A, v_i > k\}\rvert + 1$	$\begin{cases} \sum_{i=0}^{\lvert A^*\rvert-k-1} \binom{\lvert A^*\rvert}{i} + 1 & k \leq \lvert A^*\rvert \\ 2 & else \end{cases}$
$\mathcal{E}_{occ}(a_i)$	0	$\begin{cases} 1 & v_i > k \\ 0 & else \end{cases}$	$\begin{cases} 1 & v_i > k \\ 0 & else \end{cases}$	0
$\mathcal{E}_{occ}(\text{not } a_i)$	$\begin{cases} \sum_{i=1}^{\lvert A\rvert-k} \binom{\lvert A\rvert-1}{i-1} & k \leq \lvert A\rvert \\ 0 & else \end{cases}$	$\begin{cases} 0 & v_i > k \\ 1 & else \end{cases}$	0	$\begin{cases} \sum_{i=1}^{\lvert A^*\rvert-k} \binom{\lvert A^*\rvert-1}{i-1} & k \leq \lvert A^*\rvert \\ 0 & else \end{cases}$

Table 1. Occurrence formulas for literals involved in aggregates.

	#count	#min	#max	#sum
BoundedSpanningTree	X			
TravelingSalesperson	X			X
WeightedLatinSquares	X			X
WeightedSpanningTree	X			X
Labyrinth		X	X	
KnightTour	X			
TimeTabling	X			
MagicSquare	X			X

Table 2. Occurrence of aggregate functions in the considered domains.

In look-back heuristics, as a matter of fact, there are two parameters affecting the VSIDS behavior. One is the "importance" of literals in reasons (called "reward", that is, how much the related counters for such literals are to be increased, which corresponds to the coefficient of $I(l)$ in the definition of $V(l)$ in Section 5.2) and the other is the constant factor A_g by which counters are periodically divided. For the experiments presented here, these parameters have been set to 1 and 2, respectively, which are the original values used in CHAFF.

In this paper, we focus on the comparison of our new proposals to the basic version of DLV. This is because we are interested in the evaluation of the techniques presented, in comparison to the standard setting of DLV. This comparison is motivated by the fact that these versions work on exactly the same ground logic program, the output of the internal DLV grounder.

As reference, we also include other ASP solvers in the analysis, namely CLASP [19] ver. 1.0.4, CMODELS [24] ver. 3.75 and SMODELS [38] ver. 2.34, using LPARSE[12] [40] as grounder. We point out that the evaluation of our techniques with respect to solvers working with the output of a different grounder could be (at least in part) misleading, because of both the availability of different features in the grounders' input language and the difference in the ground programs. Thus, their performance should be mainly considered as a reference, and complements our experimental evaluation.

6.2. Benchmarks

For the experimental analysis on benchmarks with aggregates, we have considered some domains (see Table 2) already used in literature for the comparative evaluation of ASP systems on logic programs with aggregates [7, 21, 8, 28].[13] In Table 2 we list these domains and specify what types of aggregates they contain. Observe that all domains but one contain #count, some domains contain #sum, while #min and #max are contained in only one single domain.

[12]http://www.tcs.hut.fi/Software/smodels/lparse/.

[13]The encodings for competing solvers have been mainly taken from the ASP Competitions, except for the *TimeTabling* domain (we adapted the DLV encoding to LPARSE) and the *MagicSquare* problem (we downloaded the LPARSE encoding from [28]). For *KnightTour* we used GRINGO [22] since LPARSE was unable to parse the available encoding. For *TravelingSalesperson* we have used the LPARSE "nontight" encoding (which is faster than the "tight" one).

For two of the domains, *BoundedSpanningTree* and *WeightedSpanningTree*, we have noticed that all the versions based on DLV can solve the available instances easily (instances are solved in less than 0.1 seconds on average). Considering that the graphs the instances are constructed of are relatively small, we have generated larger instances for these domains, and made the additional instances available at `http://www.mat.unical.it/~ricca/downloads/fi-benchmarks.zip`. In the following we briefly describe how these additional instances have been generated.

Instances in the *BoundedSpanningTree* problem are defined on graphs with vertices V, edges E and a parameter d. The problem is deciding whether there exists a d-bounded spanning tree. The instances of [21, 8] are made of four sets of (randomly generated) graphs with $|V|$=35 or $|V|$=45, $|E|$=250, d=2 or d=4. The additional instances have been created randomly (as were the original instances), expanding the graph sizes by a factor of four, maintaining the ratio between the number of nodes and the number of edges. In this way, we have obtained four further sets with $|V|$=140 or $|V|$=180, $|E|$=1000, and $d = 2$ or $d = 4$, consisting of 10 graphs each. In Table 3, we have grouped the instances in two sets, those having $|V|$=140 and $|E|$=1000, and those having $|V|$=180 and $|E|$=1000, respectively.

The *WeightedSpanningTree* problem is similar to the *BoundedSpanningTree*, but each edge has an associated weight between 1 and $|V|$. The previously existing instances contain five sets of instances with $|V|$ between 30 and 45, and $|E|$ between 138 and 146. Again, we have created randomly some new sets, where the graph sizes have been enlarged by a factor of eight (again maintaining the ratio between the number of nodes and the number of edges), and the weights still range between 1 and $|V|$. In particular, we have generated 5 sets of 10 new instances for *WeightedSpanningTree*.

6.3. Results

All the experiments were performed on a machine equipped with two Intel Xeon “Woodcrest” (quad core) processors clocked at 3 GHz with 4MB of Level 2 Cache and 4GB of RAM, running Debian GNU Linux 4.0. Time measurements have been done using the `time` command provided by the system, counting total CPU time for the respective process. We report the execution time elapsed for finding one answer set, if any, within 10 minutes. The memory available to the solvers has been limited to 512MB.

Results are summarized in Table 3, and reported in detail for the various DLV versions in the Appendix.[14] In particular, Table 3 reports for each solver (columns 4-9) and for each domain (rows 2-9) the number of instances solved within the time limit, and the average CPU time (for solved instances); the number of available instances for each domain is reported in column 2. Finally, the last row reports the cumulative results of our experiments.

We remark that the data reported in Table 3 is normally used in system evaluations and competitions (for both presenting results and determining the winners) where the winning system is determined by the number of solved instances, and ties are broken by considering the mean CPU time. As an example we refer to the Max-SAT evaluations.[15]

Focusing on the results of the DLV versions analyzed, the cumulative results of our experiments, reported in the last row of Table 3, clearly indicate that DLV.BJA.VS.EQ performs better than the other two DLV versions in the considered domains. Indeed, DLV.BJA.VS.EQ is the system which solves the greatest number of instances, and it is much faster than the other systems on average. In particular, con-

[14]Results for the original instances of *BoundedSpanningTree* and *WeightedSpanningTree* domains are not reported.
[15]See `http://www.maxsat.udl.cat/09/`, and `http://www.maxsat.udl.cat/10/` for the last two evaluations.

sidering the mean time, DLV.BJA.VS.EQ is more than 4 times faster than DLV (8.57 vs 36.71 seconds), and it is nearly 3 times faster than DLV.BJA.VS.SIZE (8.57 vs 22.41 seconds).

Concerning the results of the specific domains, observe that DLV.BJA.VS.EQ is the best performer on the *WeightedLatinSquares* domain, where it solves 4 (resp. 19) instances more than standard DLV (resp. DLV.BJA.VS.SIZE) and in shorter time; it is also the best on *TimeTabling* where, even if standard DLV solves the same number of instances, it gains one order of magnitude considering average CPU time (while DLV.BJA.VS.SIZE solves only 2 instances on this domain), and on *MagicSquare* where it solves one instance more than DLV.BJA.VS.SIZE, and has a better mean CPU time of approximately 30% than DLV. Moreover, in other four domains, that is, *BoundedSpanningTree*, *WeightedSpanningTree*, *Labyrinth* and *KnightTour* it performs similar to DLV.BJA.VS.SIZE, and better (with respect to the number of problems solved and/or mean CPU time) than standard DLV. In the domains that contain instances with and without answer sets, DLV.BJA.VS.EQ is usually particularly effective on instances having solutions. We refer to the appendix for details.

In Section 5.2 we noted that a characteristic feature of the DLV.BJA.VS.EQ method is that atoms "involved" in aggregates receive a higher priority. So far, we have seen that this heuristic leads to positive results on the domains we considered, but, of course, a more "lazy" heuristic could be preferred in some situation. Indeed, this is witnessed by the results of DLV.BJA.VS.SIZE on the *TravelingSalesperson* domain: besides the good results in comparison to DLV cited above, in the *TravelingSalesperson* domain DLV.BJA.VS.SIZE solves 2 instances more than DLV.BJA.VS.EQ and in less time, being much faster than standard DLV (approximately by a factor of 20).

It is worthwhile evidencing also some relationships between the set of instances solved by the various systems (the complete data are reported in the appendix): (i) in the *WeightedLatinSquares* domain both DLV and DLV.BJA.VS.SIZE solve a subset of the instances solved by DLV.BJA.VS.EQ, while the sets of instances solved by DLV and DLV.BJA.VS.SIZE are incomparable; (ii) in the *Labyrinth* domain, DLV.BJA.VS.SIZE and DLV.BJA.VS.EQ solve the same set of instances, which is incomparable to the set of instances solved by DLV; (iii) in the *KnightTour* domain, the same set of instances is solved by the three methods; (iv) in the *TimeTabling* and *MagicSquare* domains, DLV.BJA.VS.SIZE solves a subset of the instances solved by DLV and DLV.BJA.VS.EQ.

Regarding the other ASP systems that we considered as references, from Table 3 we can see that CLASP performs better than CMODELS and SMODELS, and shows good results in several domains, except for the *BoundedSpanningTree* and *WeightedSpanningTree* domains, where is runs out of memory. Even if on the *WeightedSpanningTree* instances LPARSE can not ground the instances in the given available memory[16], in the *BoundedSpanningTree* domain instances are grounded by LPARSE, but CLASP and CMODELS run out of memory and SMODELS runs out of time on each instance. Further, note that the original 30 *BoundedSpanningTree* instances are solved by CLASP and CMODELS with mean CPU time of 7.62 and 7.97 seconds, respectively (SMODELS only solves 19 out of 30 instances). DLV.BJA.VS.EQ solves all instances with a mean CPU time of 0.02 seconds. Similar results hold for the *WeightedSpanningTree* instances: CLASP, CMODELS and DLV.BJA.VS.EQ solve the 30 original instances with mean CPU time of 3.3, 3.66 and 0.03 seconds, respectively, while SMODELS times out on 9 instances.

A further analysis is devoted to the scalability of the various DLV versions. We consider the domains for which there is a parameter that influences the size of the instances: *BoundedSpanningTree* and

[16]This is also the case when GRINGO [22] is used in place of LPARSE.

	#I		DLV	DLV.BJA.VS.SIZE	DLV.BJA.VS.EQ	CLASP	CMODELS	SMODELS
BoundedSpanningTree	20	#Solved	20	20	20	0	0	0
$\lvert V\rvert$=140 $\lvert E\rvert$=1000		Mean	3.08	0.14	0.15	MEM	MEM	TIME
BoundedSpanningTree	20	#Solved	20	20	20	0	0	0
$\lvert V\rvert$=190 $\lvert E\rvert$=1000		Mean	4.8	0.15	0.15	MEM	MEM	TIME
WeightedLatinSquares	35	#Solved	30	15	34	35	35	34
		Mean	144.07	188.58	27.5	0.03	0.23	105.88
WeightedSpanningTree	50	#Solved	50	50	50	0	0	0
		Mean	1.53	0.17	0.17	MEM	MEM	MEM
Labyrinth	29	#Solved	6	8	8	22	3	0
		Mean	114.73	58.48	58.41	108.13	218.51	TIME
TravelingSalesperson	30	#Solved	30	30	28	30	2	20
		Mean	36.49	1.5	3.34	0.13	151.68	1.98
KnightTour	10	#Solved	6	6	6	9	6	7
		Mean	12.4	0.22	0.23	59.82	0.45	4.45
TimeTabling	9	#Solved	9	2	9	9	9	2
		Mean	2.07	25.1	0.22	0.76	0.82	2.7
MagicSquare	5	#Solved	3	2	3	5	4	3
		Mean	3.25	0.01	2.45	0.49	15.46	0.05
Total	**207**	**#Solved**	**174**	**153**	**178**	**110**	**59**	**66**
		Mean	**36.51**	**22.26**	**8.52**	**26.65**	**17.61**	**55.7**

Table 3. Number of solved instances within the time limit and their mean CPU time for the domains we considered. The last row contains cumulative results.

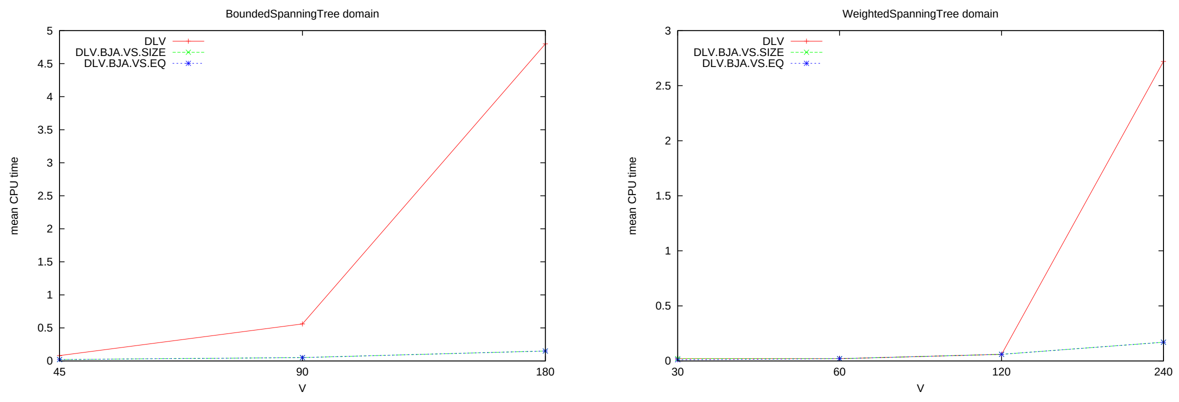


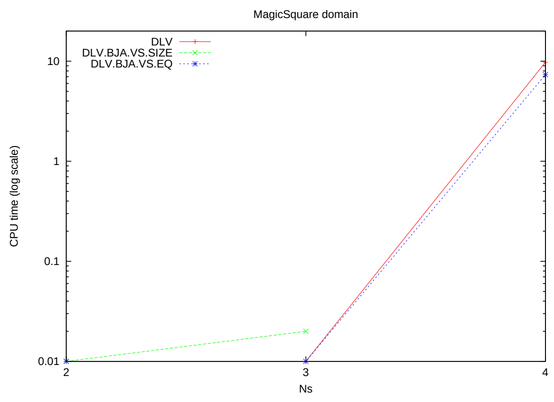
Figure 2. Additional instances of the *BoundedSpanningTree* (Left) and *WeightedSpanningTree* (Right) domain: scalability study with respect to the number of vertices V.

WeightedSpanningTree (i.e., the domains for which we have created new instances), as well as *MagicSquare*, *TimeTabling* and *KnightTour* (for which we have considered the original instances). Figure 2 (Left) contains the results for the *BoundedSpanningTree* instances, with $|V|$=$45, 90, 180$. Similarly for the *WeightedSpanningTree* instances in Figure 2 (Right), where $|V|$=$30, 60, 120, 240$. Figures 3 and 4, instead, show the behavior in terms of scalability of the *MagicSquare* and *TimeTabling* (in Figure 3), and *Knight* domain (in Figure 4). The graphs in Figures 2 and 4 clearly indicate a better scalability for both the new DLV versions in comparison to standard DLV on the *BoundedSpanningTree*, *WeightedSpanningTree* and *KnightTour* domains, while Figure 3 shows better scalability of DLV.BJA.VS.EQ than both DLV.BJA.VS.SIZE and standard DLV on the *MagicSquare* and *TimeTabling* domains.

7. Related Work

Backjumping [17] has been first studied in the area of constraint solving (see, e.g., [5, 33, 6]), and then successfully applied to related research areas such as SAT [2, 37, 31], QBF [25], and ASP [35] solving. We refer to [30] for a detailed comparison of the backjumping strategies employed in these research areas.

In ASP, aggregates are arguably the most important linguistic enhancement in recent years, and most of the available systems are already able to deal with them. In particular, CLASP, CMODELS, SMODELS and PBMODELS support cardinality and weight constraints, which correspond to #`count` and #`sum` aggregates, respectively, while SMODELS-CC supports only cardinality constraints, and both GNT and ASSAT do not support aggregates. Among these, aggregates are considered explicitly for backjumping in SMODELS-CC (where additional arcs are added to its implication graph) and CLASP, while CMODELS (resp. PBMODELS) translates the original program into a propositional (resp. Pseudo-Boolean) formula that is then evaluated by a SAT (resp. PB) solver. In the latter case, backjumping is then (possibly) exploited within the underlying SAT (resp. PB) solver, without thus having the possibility of taking advantage of the original "structure" of the aggregate. Notably, fine-grained details on the



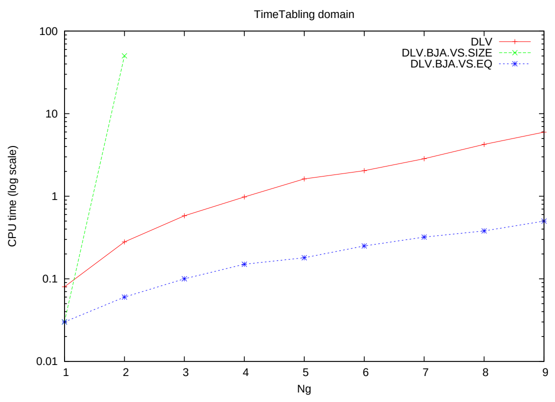


Figure 3. Instances of the *MagicSquare* (Left) and *TimeTabling* (Right) domain. Ns indicates an Ns x Ns square (Left) and Ng is the groups of students (Right).

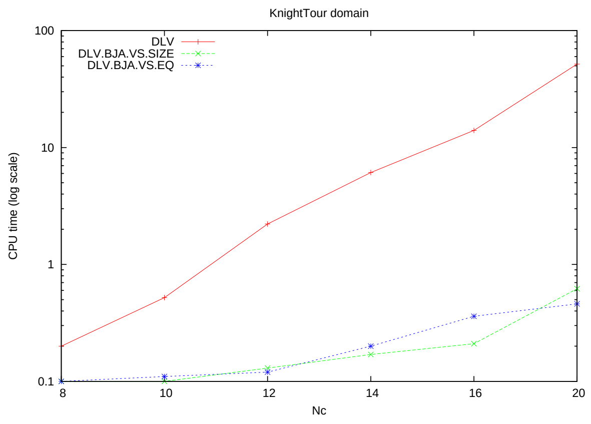


Figure 4. Instances of the Knight domain. Nc indicates that the related chess board size is Nc x Nc.

treatment of aggregates have been rarely presented before, and current implementations use more or less ad-hoc techniques. An exception, which nonetheless involves only #sum (weight constraints), has been recently presented in [18]. In that work, a comparison of different strategies to handle weight constraints in CLASP has been performed. In particular, two strategies have been presented and compared: One where weight constraint rules are incorporated in CLASP's constraint-based characterization in terms of nogoods, and another which (similar to the approach in CMODELS[17]) translates the aggregate into (constraints corresponding to) an aggregate-free program. Experimental analysis on some domains show that each strategy performs well on different domains.

8. Conclusion

In this paper we have described techniques and heuristics for the evaluation of logic programs with aggregates. In particular the main contributions are: (i) an extension of the *reason calculus* defined in [35]; and, (ii) enhanced versions of the heuristic presented in [30] that explicitly take the presence of aggregates into account. Moreover, we have implemented the proposed techniques in a prototype version of the DLV system and performed a set of benchmarks, which indicate performance benefits of the enhanced system employing the equivalent-program heuristic.

Acknowledgements

This work has been partially supported by M.I.U.R. under the PRIN project LoDeN, and under the FIRB project "tocai.it: Tecnologie Orientate alla Conoscenza per Aggregazioni di Imprese in Internet," and by Regione Calabria and EU under POR Calabria FESR 2007-2013 within the PIA project of DLVSYSTEM s.r.l.

We would like to thank the anonymous reviewers for their helpful suggestions. This work is dedicated to "mulo" and two other machines which broke down under the burden of executing the experiments reported in this paper.

References

[1] Alviano, M.: Efficient Recursive Aggregate Evaluation in Logic Programming, *Intelligenza Artificiale.* IOS Press, 2011, To appear.

[2] Bayardo, R., Schrag, R.: Using CSP Look-back Techniques to Solve Real-world SAT Instances, *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-97)*, 1997.

[3] Calimeri, F., Faber, W., Leone, N., Pfeifer, G.: Pruning Operators for Answer Set Programming Systems, *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning (NMR'2002)*, April 2002.

[4] Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem Proving, *Communications of the ACM*, **5**, 1962, 394–397.

[5] Dechter, R.: Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition, *Artificial Intelligence*, **41**(3), 1990, 273–312.

[17] CMODELS implements the transformation described in [16], while CLASP implements a polynomial transformation.

[6] Dechter, R., Frost, D.: Backjump-based backtracking for constraint satisfaction problems., *Artificial Intelligence*, **136**(2), 2002, 147–188.

[7] Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G.: Aggregate Functions in DLV, *Proceedings ASP03 - Answer Set Programming: Advances in Theory and Implementation* (M. de Vos, A. Provetti, Eds.), Messina, Italy, September 2003, Online at `http://CEUR-WS.org/Vol-78/`.

[8] Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczynski, M.: The Second Answer Set Programming Competition, *Proceedings of tre 10th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR'09* (E. Erdem, F. Lin, T. Schaub, Eds.), 5753, Springer, 2009.

[9] Faber, W.: *Enhancing Efficiency and Expressiveness in Answer Set Programming Systems*, Ph.D. Thesis, Institut für Informationssysteme, Technische Universität Wien, 2002.

[10] Faber, W., Leone, N., Pfeifer, G.: Pushing Goal Derivation in DLP Computations, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)* (M. Gelfond, N. Leone, G. Pfeifer, Eds.), 1730, Springer Verlag, El Paso, Texas, USA, December 1999.

[11] Faber, W., Leone, N., Pfeifer, G.: Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity, *Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004)* (J. J. Alferes, J. Leite, Eds.), 3229, Springer Verlag, September 2004.

[12] Faber, W., Leone, N., Pfeifer, G., Ricca, F.: On look-ahead heuristics in disjunctive logic programming, *Annals of Mathematics and Artificial Intelligence*, **51**(2–4), 2007, 229–266.

[13] Faber, W., Pfeifer, G., Leone, N.: Semantics and complexity of recursive aggregates in answer set programming, *Artificial Intelligence*, **175**(1), 2011, 278–298.

[14] Faber, W., Pfeifer, G., Leone, N., Dell'Armi, T., Ielpa, G.: Design and Implementation of Aggregate Functions in the DLV System, *Theory and Practice of Logic Programming*, **8**(5–6), 2008, 545–580.

[15] Faber, W., Ricca, F.: Solving Hard ASP Programs Efficiently, *Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy, September 2005, Proceedings* (C. Baral, G. Greco, N. Leone, G. Terracina, Eds.), 3662, Springer Verlag, September 2005, ISBN 3-540-28538-5.

[16] Ferraris, P., Lifschitz, V.: Weight constraints as nested expressions, *Theory and Practice of Logic Programming*, **5**(1-2), 2005, 45–74.

[17] Gaschnig, J.: A General Backtrack Algorithm That Eliminates Most Redundant Tests, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI) 1977*, 1977.

[18] Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: On the Implementation of Weight Constraint Rules in Conflict-Driven ASP Solvers, *Proceedings of 25th International Conference on Logic Programming (ICLP-09)* (P. M. Hill, D. S. Warren, Eds.), 5649, Springer, 2009.

[19] Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: clasp: A Conflict-Driven Answer Set Solver, *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)* (C. Baral, G. Brewka, J. Schlipf, Eds.), 4483, Springer-Verlag, 2007.

[20] Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-Driven Answer Set Solving, *Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, Morgan Kaufmann Publishers, January 2007.

[21] Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The First Answer Set Programming System Competition, *9th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR'07* (C. Baral, G. Brewka, J. Schlipf, Eds.), 4483, Springer Verlag, Tempe, Arizona, May 2007, ISBN 978-3-540-72199-4.

[22] Gebser, M., Schaub, T., Thiele, S.: GrinGo : A New Grounder for Answer Set Programming, *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings* (C. Baral, G. Brewka, J. S. Schlipf, Eds.), 4483, Springer, 2007.

[23] Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases, *New Generation Computing*, **9**, 1991, 365–385.

[24] Giunchiglia, E., Lierler, Y., Maratea, M.: Answer Set Programming Based on Propositional Satisfiability, *Journal of Automated Reasoning*, **36**(4), 2006, 345–377.

[25] Giunchiglia, E., Narizzano, M., Tacchella, A.: Backjumping for Quantified Boolean Logic Satisfiability, *Artificial Intelligence*, **145**, 2003, 99–120.

[26] Lee, J., Meng, Y.: On Reductive Semantics of Aggregates in Answer Set Programming, *Logic Programming and Nonmonotonic Reasoning — 10th International Conference (LPNMR 2009)* (E. Erdem, F. Lin, T. Schaub, Eds.), 5753, Springer Verlag, September 2009, ISBN 978-3-642-04237-9.

[27] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning, *ACM Transactions on Computational Logic*, **7**(3), July 2006, 499–562.

[28] Liu, L., Truszczyński, M.: The Second Answer Set Programming Competition homepage, Since 2005, `http://www.cs.uky.edu/ai/pbmodels/#Benchmark|region`.

[29] Manna, M., Ruffolo, M., Oro, E., Alviano, M., Leone, N.: The HiLeX System for Semantic Information Extraction, *Transactions on Large-Scale Data and Knowledge-Centered Systems.* Springer Berlin/Heidelberg, 2011, To appear.

[30] Maratea, M., Ricca, F., Faber, W., Leone, N.: Look-Back Techniques and Heuristics in DLV: Implementation, Evaluation and Comparison to QBF Solvers, *Journal of Algorithms in Cognition, Informatics and Logics*, **63**(1–3), 2008, 70–89.

[31] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver, *Proceedings of the 38th Design Automation Conference, DAC 2001*, ACM, Las Vegas, NV, USA, June 2001.

[32] Pelov, N., Denecker, M., Bruynooghe, M.: Well-founded and Stable Semantics of Logic Programs with Aggregates, *Theory and Practice of Logic Programming*, **7**(3), 2007, 301–353.

[33] Prosser, P.: Hybrid Algorithms for the Constraint Satisfaction Problem., *Computational Intelligence*, **9**, 1993, 268–299.

[34] Ricca, F., Alviano, M., Dimasi, A., Grasso, G., Ielpa, S. M., Iiritano, S., Manna, M., Leone, N.: A Logic–Based System for e–Tourism, *Fundamenta Informaticae, IOS Press*, (105), 2010, 35–35.

[35] Ricca, F., Faber, W., Leone, N.: A Backjumping Technique for Disjunctive Logic Programming, *AI Communications – The European Journal on Artificial Intelligence*, **19**(2), 2006, 155–172.

[36] Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S., Leone, N.: Team-building with Answer Set Programming in the Gioia-Tauro Seaport, *Theory and Practice of Logic Programming.* Cambridge University Press, 2011, To appear.

[37] Silva, J. P. M., Sakallah, K. A.: GRASP: A Search Algorithm for Propositional Satisfiability, *IEEE Transaction on Computers*, **48**(5), 1999, 506–521.

[38] Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics, *Artificial Intelligence*, **138**, June 2002, 181–234.

[39] Son, T. C., Pontelli, E.: A Constructive Semantic Characterization of Aggregates in ASP, *Theory and Practice of Logic Programming*, **7**, May 2007, 355–375.

[40] Syrjänen, T.: Lparse 1.0 User's Manual, 2002, `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.

APPENDIX

In Tables 4-11 we show detailed results for the benchmarks. The reported numbers are seconds for runtime (user + system time). We do not report instances where no DLV version can find a solution within the time limit: this is the case for Tables 6, 7, 9 and 10. In Tables 4-11, the first column reports the specific instance name, the second column reports the results for the standard version of DLV, and the third and fourth columns report the results for the new versions: DLV.BJA.VS.SIZE and DLV.BJA.VS.EQ, respectively. Moreover, if the related domain contains both instances that have answer sets and others that do not, a last column (AS?) is added, which indicates whether the related instance has an answer set (Y) or does not (N). If all instances of a domain have answer sets, this column is omitted.

instance	DLV	DLV.BJA.VS.SIZE	DLV.BJA.VS.EQ	AS?
rand-140-1000-b2-0.gph	2.53	0.14	0.14	Y
rand-140-1000-b2-1.gph	2.49	0.14	0.15	Y
rand-140-1000-b2-2.gph	2.57	0.14	0.15	Y
rand-140-1000-b2-3.gph	5.60	0.14	0.15	Y
rand-140-1000-b2-4.gph	3.04	0.14	0.14	Y
rand-140-1000-b2-5.gph	2.69	0.14	0.14	Y
rand-140-1000-b2-6.gph	2.48	0.14	0.15	Y
rand-140-1000-b2-7.gph	2.46	0.14	0.15	Y
rand-140-1000-b2-8.gph	2.50	0.14	0.15	Y
rand-140-1000-b2-9.gph	2.49	0.14	0.14	Y
rand-140-1000-b4-0.gph	2.68	0.14	0.15	Y
rand-140-1000-b4-1.gph	5.70	0.16	0.15	Y
rand-140-1000-b4-2.gph	2.54	0.14	0.16	Y
rand-140-1000-b4-3.gph	2.86	0.15	0.16	Y
rand-140-1000-b4-4.gph	2.71	0.14	0.15	Y
rand-140-1000-b4-5.gph	5.87	0.14	0.16	Y
rand-140-1000-b4-6.gph	2.73	0.15	0.14	Y
rand-140-1000-b4-7.gph	2.44	0.14	0.14	Y
rand-140-1000-b4-8.gph	2.73	0.18	0.16	Y
rand-180-1000-b2-0.gph	6.92	0.15	0.16	Y
rand-180-1000-b2-1.gph	7.27	0.15	0.16	Y
rand-180-1000-b2-2.gph	3.01	0.15	0.16	Y
rand-180-1000-b2-3.gph	3.05	0.16	0.16	Y
rand-180-1000-b2-4.gph	6.44	0.15	0.16	Y
rand-180-1000-b2-5.gph	6.94	0.15	0.16	Y
rand-180-1000-b2-6.gph	7.58	0.15	0.15	Y
rand-180-1000-b2-7.gph	7.28	0.15	0.15	Y
rand-180-1000-b2-8.gph	3.47	0.14	0.16	Y
rand-180-1000-b2-9.gph	3.44	0.15	0.16	Y
rand-180-1000-b4-0.gph	0.13	0.13	0.13	N
rand-180-1000-b4-1.gph	3.55	0.15	0.15	Y
rand-180-1000-b4-2.gph	0.13	0.12	0.13	N
rand-180-1000-b4-3.gph	7.95	0.15	0.16	Y
rand-180-1000-b4-4.gph	7.57	0.15	0.16	Y
rand-180-1000-b4-5.gph	7.26	0.15	0.16	Y
rand-180-1000-b4-6.gph	3.31	0.15	0.16	Y
rand-180-1000-b4-7.gph	7.04	0.16	0.16	Y
rand-180-1000-b4-8.gph	3.56	0.14	0.16	Y
rand-180-1000-b4-9.gph	0.13	0.13	0.12	N

Table 4. Instances of the Bounded Spanning Tree domain. rand-V-E-bW-i.gph indicates the i-th graph with V vertexes, E edges and bound W.

instance	DLV	DLV.BJA.VS.SIZE	DLV.BJA.VS.EQ	AS?
rand-240-1120-b1600-0.gph	9.49	0.19	0.19	Y
rand-240-1120-b1600-1.gph	5.60	0.18	0.19	Y
rand-240-1120-b1600-2.gph	0.15	0.14	0.16	N
rand-240-1120-b1600-3.gph	0.16	0.15	0.16	N
rand-240-1120-b1600-4.gph	5.63	0.19	0.21	Y
rand-240-1120-b1600-5.gph	5.52	0.20	0.20	Y
rand-240-1120-b1600-6.gph	0.15	0.15	0.15	N
rand-240-1120-b1600-7.gph	0.15	0.15	0.15	N
rand-240-1120-b1600-8.gph	0.17	0.16	0.16	N
rand-240-1120-b1600-9.gph	0.15	0.16	0.15	N
rand-256-1120-b1472-0.gph	5.82	0.19	0.19	Y
rand-256-1120-b1472-1.gph	0.16	0.15	0.16	N
rand-256-1120-b1472-2.gph	0.16	0.16	0.16	N
rand-256-1120-b1472-3.gph	0.16	0.15	0.15	N
rand-256-1120-b1472-4.gph	0.16	0.15	0.15	N
rand-256-1120-b1472-5.gph	13.39	0.19	0.21	Y
rand-256-1120-b1472-6.gph	0.15	0.15	0.15	N
rand-256-1120-b1472-7.gph	0.16	0.16	0.16	N
rand-256-1120-b1472-8.gph	5.57	0.20	0.21	Y
rand-256-1120-b1472-9.gph	0.15	0.16	0.16	N
rand-256-1160-b1600-0.gph	0.17	0.18	0.17	N
rand-256-1160-b1600-1.gph	0.18	0.17	0.17	N
rand-256-1160-b1600-2.gph	0.17	0.16	0.16	N
rand-256-1160-b1600-3.gph	6.40	0.21	0.21	Y
rand-256-1160-b1600-4.gph	5.97	0.21	0.21	Y
rand-256-1160-b1600-5.gph	0.16	0.16	0.16	N
rand-256-1160-b1600-6.gph	0.18	0.17	0.18	N
rand-256-1160-b1600-7.gph	0.16	0.16	0.16	N
rand-256-1160-b1600-8.gph	6.76	0.21	0.22	Y
rand-256-1160-b1600-9.gph	0.17	0.17	0.17	N
rand-280-1104-b1984-0.gph	0.16	0.16	0.16	N
rand-280-1104-b1984-1.gph	0.16	0.16	0.17	N
rand-280-1104-b1984-2.gph	0.16	0.16	0.16	N
rand-280-1104-b1984-3.gph	0.15	0.14	0.14	N
rand-280-1104-b1984-4.gph	0.16	0.16	0.16	N
rand-280-1104-b1984-5.gph	0.14	0.13	0.14	N
rand-280-1104-b1984-6.gph	0.16	0.16	0.16	N
rand-280-1104-b1984-7.gph	0.16	0.15	0.14	N
rand-280-1104-b1984-8.gph	0.16	0.16	0.16	N
rand-280-1104-b1984-9.gph	0.15	0.15	0.16	N
rand-360-1104-b2496-0.gph	0.16	0.15	0.15	N
rand-360-1104-b2496-1.gph	0.15	0.16	0.15	N
rand-360-1104-b2496-2.gph	0.16	0.17	0.17	N
rand-360-1104-b2496-3.gph	0.15	0.15	0.16	N
rand-360-1104-b2496-4.gph	0.17	0.16	0.16	N
rand-360-1104-b2496-5.gph	0.17	0.16	0.17	N
rand-360-1104-b2496-6.gph	0.15	0.14	0.14	N
rand-360-1104-b2496-7.gph	0.16	0.17	0.16	N
rand-360-1104-b2496-8.gph	0.16	0.16	0.15	N
rand-360-1104-b2496-9.gph	0.16	0.17	0.16	N

Table 5. Instances of the WeightedSpanningTree domain. rand-V-E-bW-i.gph indicates the i-th graph with V vertexes, E edges and bound W.

instance	DLV	DLV.BJA.VS.SIZE	DLV.BJA.VS.EQ
magic-square-2by2	0.00	0.00	0.00
magic-square-3by3	0.01	0.02	0.01
magic-square-4by4	9.75	TIME	7.34

Table 6. Instances of the MagicSquares domain.

instance	DLV	DLV.BJA.VS.SIZE	DLV.BJA.VS.EQ
knightTour.in1	0.20	0.10	0.10
knightTour.in2	0.52	0.10	0.11
knightTour.in3	2.22	0.13	0.12
knightTour.in4	6.11	0.17	0.20
knightTour.in5	14.00	0.21	0.36
knightTour.in6	51.76	0.62	0.46

Table 7. Instances of the KnightTour domain.

instance	DLV	DLV.BJA.VS.SIZE	DLV.BJA.VS.EQ
time-tabling.dat.1	0.08	0.03	0.03
time-tabling.dat.2	0.28	50.17	0.06
time-tabling.dat.3	0.58	TIME	0.10
time-tabling.dat.4	0.98	TIME	0.15
time-tabling.dat.5	1.62	TIME	0.18
time-tabling.dat.6	2.04	TIME	0.25
time-tabling.dat.7	2.85	TIME	0.32
time-tabling.dat.8	4.25	TIME	0.38
time-tabling.dat.9	5.98	TIME	0.50

Table 8. Instances of the TimeTabling domain.

instance	DLV	DLV.BJA.VS.SIZE	DLV.BJA.VS.EQ
laby-17-17-07	16.83	4.88	4.83
laby-18-18-04	112.46	289.38	289.80
laby-18-18-13	TIME	30.51	40.63
laby-18-18-14	18.16	3.86	3.86
laby-19-19-14	TIME	105.03	105.03
laby-19-19-16	TIME	11.21	11.13
laby-19-19-19	448.95	TIME	TIME
laby-20-20-04	TIME	7.35	7.25
laby-20-20-16	TIME	5.60	5.57
laby-21-21-05	46.96	TIME	TIME
laby-21-21-15	45.04	TIME	TIME

Table 9. Instances of the Labyrinth domain.

instance	DLV	DLV.BJA.VS.SIZE	DLV.BJA.VS.EQ	AS?
505.6.1976043347.dat.data	160.28	TIME	6.33	N
505.6.1976043746.dat.data	313.57	TIME	1.66	N
505.6.1976049584.dat.data	267.15	TIME	34.72	N
505.6.1976055607.dat.data	490.32	TIME	2.79	N
505.6.1976056195.dat.data	426.55	TIME	173.44	N
505.6.1976048051.dat.data	59.31	TIME	0.02	Y
505.6.1976089585.dat.data	3.43	5.86	0.02	Y
505.6.1976090993.dat.data	230.87	TIME	77.41	N
505.6.1976095999.dat.data	476.51	1.07	0.02	Y
505.6.1976097106.dat.data	TIME	0.02	0.04	Y
505.6.1976097683.dat.data	107.60	TIME	488.43	Y
505.6.1976102161.dat.data	378.58	TIME	20.22	N
505.6.1976103815.dat.data	246.21	TIME	0.03	N
505.6.1976128002.dat.data	TIME	645.10	33.53	Y
505.6.1976128122.dat.data	0.15	TIME	0.03	Y
505.6.1976131149.dat.data	31.30	0.99	0.01	Y
505.6.1976135316.dat.data	420.92	TIME	94.67	N
505.6.1976148351.dat.data	TIME	TIME	0.16	Y
505.6.1976153426.dat.data	46.85	42.80	0.02	Y
505.6.1976164056.dat.data	TIME	TIME	1.17	Y
505.6.1976164284.dat.data	0.09	558.48	0.07	Y
511.6.1162362547.dat.data	64.69	3.84	0.02	Y
511.6.1162368434.dat.data	70.96	204.38	0.02	Y
511.6.1162583044.dat.data	0.07	0.44	0.03	Y
511.6.1162586028.dat.data	184.04	TIME	0.02	Y
512.6.1669104030.dat.data	22.26	423.10	0.02	Y
512.6.1669117391.dat.data	0.08	4.23	0.03	Y
512.6.1669132369.dat.data	11.31	TIME	0.02	Y
512.6.1669131750.dat.data	3.37	450.98	0.01	Y
512.6.1669245041.dat.data	24.21	TIME	0.03	Y
512.6.1669208235.dat.data	0.08	TIME	0.02	Y
512.6.1669316059.dat.data	29.62	487.31	0.01	Y
512.6.1669326545.dat.data	10.39	TIME	0.05	Y
513.6.2014058873.dat.data	241.45	0.03	0.02	Y

Table 10. Instances of the WeightedLatinSquares domain.

instance	DLV	DLV.BJA.VS.SIZE	DLV.BJA.VS
dom-rand-70-300-1155482584-3	424.70	1.03	2.09
rand-70-300-1155482584-0	0.10	0.03	0.28
rand-70-300-1155482584-3	0.09	0.06	1.17
rand-70-300-1155482584-4	0.11	0.03	0.31
rand-70-300-1155482584-5	0.11	0.04	0.30
rand-70-300-1155482584-7	0.10	0.07	0.46
rand-70-300-1155482584-8	0.09	0.03	0.29
rand-70-300-1155482584-9	0.09	0.04	0.28
rand-70-300-1155482584-11	0.11	0.03	0.30
rand-70-300-1155482584-12	0.13	0.06	0.30
rand-70-300-1155482584-14	0.11	0.04	1.05
rand-80-340-1159656267-0	0.12	0.06	0.39
rand-80-340-1159656267-4	0.12	0.04	0.63
rand-80-340-1159656267-6	0.13	0.11	0.38
rand-80-340-1159656267-10	0.12	0.05	0.56
rand-80-340-1159656267-11	0.18	0.10	0.70
rand-80-340-1159656267-13	0.12	0.07	1.28
rand-80-340-1159656267-15	0.16	0.44	0.35
rand-80-340-1159656267-16	0.13	0.05	1.00
rand-80-340-1159656267-17	0.14	0.51	0.99
rand-80-340-1159656267-18	0.14	0.04	11.19
tsp-rand-70-300-1155482584-0	0.46	0.04	0.56
tsp-rand-70-300-1155482584-4	85.97	0.88	60.76
tsp-rand-70-300-1155482584-5	0.11	0.24	2.30
tsp-rand-70-300-1155482584-7	1.26	0.08	TIME
tsp-rand-70-300-1155482584-8	0.15	0.03	0.28
tsp-rand-70-300-1155482584-9	2.82	28.30	0.48
tsp-rand-70-300-1155482584-11	540.09	10.75	TIME
tsp-rand-70-300-1155482584-12	0.14	0.16	0.39
tsp-rand-70-300-1155482584-14	0.12	0.04	1.50

Table 11. Instances of the TravelingSalesperson domain.