Look-back Techniques for ASP Programs with Aggregates

Wolfgang Faber[1], Nicola Leone[1], Marco Maratea[1,2], and Francesco Ricca[1]

[1] Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
{faber,leone,maratea,ricca}@mat.unical.it

[2] DIST, University of Genova, 16145 Genova, Italy
marco@dist.unige.it

Abstract

One of the most significant language extensions to Answer Set Programming (ASP) has been the introduction of aggregates. A significant amount of theoretical and practical work on aggregates in ASP has been published in recent years. In spite of these developments, aggregates are treated in a quite straightforward and ad-hoc way in most ASP systems. For the system DLV, several specialized techniques for aggregates have been described in [6], however still leaving a lot of room for improvement.

In this paper, we build upon work on look-back optimization techniques done recently for DLV, and extend its reason calculus for backjumping to include reasons from aggregates. Furthermore, we describe how these reasons can be used in order to tune look-back heuristic counters. We present a preliminary experimental analysis, including also other state-of-the-art ASP systems, showing that our approach is promising.

1 Introduction

Answer Set Programming (ASP) [9] 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 efficient systems. 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) [16, 20, 5] and practically, for endowing systems with a selection of aggregate functions [19, 2, 6, 8].

However, work on optimizing system performance with respect to aggregates is still sparse, and current implementations use more or less ad-hoc techniques. In this work, we report on improvements in this field. In particular, we build upon a technique for backjumping, which had been developed in the setting of the solver DLV in [18]. As a main contribution, we describe how the *reason calculus* defined in [18] can be extended for keeping track of the reasons for several types of aggregates supported in DLV. The information collected in this way can then be exploited directly for backjumping, using the original method described in [18].

Proceedings of the 15th International RCRA workshop (RCRA 2008):
Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion
Udine, Italy, 12–13 December 2008

Importantly, reasons for aggregates can also be exploited for look-back heuristic. Indeed, we show how the look-back heuristics presented in [4] 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, it 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 an aggregate-free program, which corresponds to the given program with aggregates, and use standard techniques for initializing the heuristic values. Importantly, in our technique we make sure to not materialize this aggregate-free program, but use the knowledge about its structure for computing the initialization values. This method is exact in the aggregate-stratified case, in the sense that the aggregate-free program is equivalent to the original program with aggregates. While the program is in general not equivalent to the original one in the aggregate-unstratified case, it can still be used for the purpose of a heuristics, as it will still be 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 selected benchmarks, in which we could observe performance benefits for the system relying on our optimization techniques.

2 Answer Set Programming with Aggregates

2.1 Syntax

We assume that the reader is familiar with standard logic programming; we refer to the respective constructs as *standard atoms*, *standard literals*, *standard rules*, and *standard programs*. Two literals are said to be complementary if they are of the form p and $\mathrm{not}\ p$ for some atom p. Given a literal L, $\neg.L$ denotes its complementary literal. Accordingly, given a set A of literals, $\neg.A$ denotes the set $\{\neg.L \mid\ L \in A\}$. For further background, see [1, 9].

Set Terms. A DLP$^{\mathcal{A}}$ *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, i.e., $\{X \mid \exists Y\, s.t.\ a(X,Y), p(Y)\ is\ true\}$.

Example 1. In the examples, we adopt the syntax of DLV to denote aggregates. *Aggregate functions currently supported by the* DLV *system are:* #count *(number of terms),* #sum *(sum of non-negative integers),* #min *(minimum term),* #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. *The following aggregate atoms are in* DLV *notation, where the latter contains a ground set and could be a ground instance of the former:*

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

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*.

DLP$^{\mathcal{A}}$ **Programs.** A DLP$^{\mathcal{A}}$ *rule* r is a construct

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

where $a_1, \cdots, a_n$ are standard atoms, $b_1, \cdots, b_m$ are atoms, and $n \geq 1$, $m \geq k \geq 0$. The disjunction $a_1 \ \mathtt{v} \ \cdots \ \mathtt{v} \ a_n$ is referred to as the *head* of r while the conjunction $b_1, ..., b_k, \ \mathrm{not}\ b_{k+1}, ..., \mathrm{not}\ b_m$ is the *body* of r. We denote the set of head atoms by $H(r)$, and the set $\{b_1, ..., b_k, \ \mathrm{not}\ b_{k+1}, ..., \mathrm{not}\ b_m\}$ of the body literals by $B(r)$. $B^+(r)$ and $B^-(r)$ denote, respectively, the set of positive and negative literals in $B(r)$. Note that this syntax does not explicitly allow integrity constraints (rules without head atoms). They can, however, be simulated in the usual way by using a new symbol and negation.

A DLP$^{\mathcal{A}}$ *program* is a set of DLP$^{\mathcal{A}}$ rules. In the sequel, we will often drop DLP$^{\mathcal{A}}$, when it is clear from the context. A *global* variable of a rule r appears in a standard atom of r (possibly also in other atoms); all other variables are *local*.

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 $\{ Vars : conj\}$ 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.

[2]The first two aggregates roughly correspond, respectively, to the cardinality and weight constraint literals of Smodels. #min and #max are undefined for empty set.

Stratification. A DLP$^{\mathcal{A}}$ 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 3. *Consider the program consisting of a set of facts for predicates a and b, plus the following two rules:*

$$q(X) \ \texttt{:-}\ p(X),\ \#\texttt{count}\{Y : a(Y, X), b(X)\} \leq 2.$$
$$p(X) \ \texttt{:-}\ q(X),\ b(X).$$

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 $b(X)$:- $p(X)$, 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. [16, 20, 5]. In this work, we will consider only aggregate-stratified programs, but all considerations should apply also to aggregate-unstratified programs under any of the proposed semantics.

2.2 Answer Set Semantics

Universe and Base. Given a DLP$^{\mathcal{A}}$ program $\mathcal{P}$, let $U_{\mathcal{P}}$ denote the set of constants appearing in $\mathcal{P}$, and $B_{\mathcal{P}}$ be the set of standard atoms constructible from the (standard) predicates of $\mathcal{P}$ with constants in $U_{\mathcal{P}}$. 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 $\mathbb{I}$ (the set of integers).

Example 4. $\#\texttt{count}$ *is defined over* $\overline{2}^{U_{\mathcal{P}}}$, $\#\texttt{sum}$ *over* $\overline{2}^{\mathbb{N}}$, $\#\texttt{min}$ *and* $\#\texttt{max}$ *are defined over* $\overline{2}^{\mathbb{N}} - \{\emptyset\}$.

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 following 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 $\mathcal{P}$.

[3]Given a substitution σ and a DLP$^{\mathcal{A}}$ object Obj (rule, set, etc.), we denote by $\sigma(Obj)$ the object obtained by replacing each variable X in Obj by $\sigma(X)$.

Interpretations. An *interpretation* for a DLP$^{\mathcal{A}}$ 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) w.r.t I if $L \in I$ (resp. $L \in \neg.I$). If a standard ground literal is neither true nor false w.r.t I then it is undefined w.r.t 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 w.r.t. I (i.e. $B_{\mathcal{P}} \setminus I^+ \cup I^-$). An interpretation I is *total* if $\bar{I}$ is empty (i.e., $I^+ \cup \neg.I^- = B_{\mathcal{P}}$), 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 (resp. false) w.r.t I if all (resp. any of) its literals are true (resp. false). The meaning of a set, an aggregate function, and an aggregate atom under an interpretation, is a multiset, a value, and a truth-value, respectively. Let $f(S)$ be a an aggregate function. The valuation $I(S)$ of S w.r.t. I is the multiset of the first constant of the elements in S whose conjunction is true w.r.t. 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 w.r.t.* $I\ ]$. The valuation $I(f(S))$ of an aggregate function $f(S)$ w.r.t. 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 w.r.t.* I if: (i) $I(f(S)) \neq \bot$, and, (ii) $I(f(S)) \prec k$ holds; otherwise, A is false. An instantiated aggregate literal $\mathrm{not}A = \mathrm{not}f(S) \prec k$ is *true w.r.t.* I if (i) $I(f(S)) \neq \bot$, and, (ii) $I(f(S)) \prec k$ does not hold; otherwise, A is false.

If I is a *partial* interpretation, an aggregate literal A is true (resp. false) w.r.t. I if it is true (resp. false) w.r.t. *each total* interpretation J extending I (i.e., $\forall\ J$ s.t. $I \subseteq J$, A is true (resp. false) w.r.t. J); otherwise it is undefined.

Example 5. *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 w.r.t.* I.

Remark 1. Our definitions of interpretation and truth values preserve "knowledge monotonicity". If an interpretation J extends I (i.e., $I \subseteq J$), then each literal which is true w.r.t. I is true w.r.t. J, and each literal which is false w.r.t. I is false w.r.t. J as well.

Minimal Models. Given an interpretation I, a rule r is *satisfied w.r.t.* I if some head atom is true w.r.t. I whenever all body literals are true w.r.t. I. A total interpretation M is a *model* of a DLP$^{\mathcal{A}}$ program $\mathcal{P}$ if all $r \in Ground(\mathcal{P})$ are satisfied w.r.t. M. A model M for $\mathcal{P}$ is (subset) minimal if no model N for $\mathcal{P}$ exists such that $N^+ \subset M^+$. 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 DLP$^{\mathcal{A}}$ programs from [5]: Given a ground DLP$^{\mathcal{A}}$ 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 w.r.t. I. I is an answer set of a program $\mathcal{P}$ if it is a minimal model of $Ground(\mathcal{P})^I$.

Example 6. *Consider interpretation $I_1 = \{p(a)\}$, $I_2 = \{\text{not}p(a)\}$ and two programs $P_1 = \{p(a)$:- $\#\texttt{count}\{X : p(X)\} > 0.\}$ and $P_2 = \{p(a)$:- $\#\texttt{count}\{X : p(X)\} < 1.\}$. $Ground(P_1) = \{p(a)$:- $\#\texttt{count}\{\langle a : p(a)\rangle\} > 0.\}$ and $Ground(P_1)^{I_1} = Ground(P_1)$, $Ground(P_1)^{I_2} = \emptyset$. Furthermore, $Ground(P_2) = \{p(a)$:- $\#\texttt{count}\{\langle a : p(a)\rangle\} < 1.\}$, and $Ground(P_2)^{I_1} = \emptyset$, $Ground(P_2)^{I_2} = Ground(P_2)$ hold. I_2 is the only answer set of P_1 (since I_1 is not a minimal model of $Ground(P_1)^{I_1}$), while P_2 admits no answer set (I_1 is not a minimal model of $Ground(P_2)^{I_1}$, and I_2 is not a model of $Ground(P_2) = Ground(P_2)^{I_2}$).*

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 $Ground(\mathcal{P}) - Ground(\mathcal{P})^A$ are satisfied w.r.t. A.

3 Backjumping and Reason Calculus in DLV

DLV is the state-of-the-art *disjunctive* ASP system. DLV relies on backtracking search similar to the DPLL procedure for SAT solving (most other competitive ASP systems exploit similar techniques). Basically, starting from the empty (partial) interpretation, the solver repeatedly assumes truth-values for atoms (chosen according to an heuristic), subsequently computing their deterministic consequences (propagation). This is done until either an answer set is found or an inconsistency is detected. In the latter case, (chronological) backtracking occurs. Since the last choice does not necessarily influence the inconsistency, the procedure may perform a lot of useless computations. In [18], DLV has been enhanced by backjumping [7, 17], which allows for going back to a choice which is relevant for the found inconsistency.[4] A crucial point is how relevance for an inconsistency can be determined. In [18], the necessary information for deciding relevance is recorded by means of a reason calculus, 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 :- $b, c, \text{not } d.$, if b and c are true and d is false in the current partial interpretation, then a will be derived as true. 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. *Chosen* literals are seen as their own reason. So 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. In the

[4] For more details, see [3] for the basic DLV algorithm and [18] for backjumping.

following, we will describe the inference rules needed for correctly implementing aggregates [19, 2], and we present the associated extension of the reason calculus which allows for dealing with aggregates.

4 Reason Calculus for Aggregates

We next report the reason calculus for each aggregate supported by DLV. Hereafter, a partial interpretation (here a set of literals) I is assumed to be given.

Consider a pair $\langle \bar{t} : conj \rangle$ where $\bar{t}$ is a sequence of terms and $conj$ a conjunction of literals. We denote by $\mathcal{C}_{conj}$ (resp. $\mathcal{S}_{conj}$) the reason for $conj$ to be false (resp. true) w.r.t. I. In particular, $\mathcal{C}_{conj}$ is the reason of a false literal in $conj$[5], while $\mathcal{S}_{conj} = \bigcup_{l \in conj} R(l)$, i.e. all reasons for the literals in $conj$. Moreover, let $A = \{\langle \bar{t}_1 : conj_1 \rangle, \ldots, \langle \bar{t}_n : conj_n \rangle\}$ be a set term, define $\mathcal{C}_A = \bigcup_{\langle \bar{t}:conj \rangle \in A \wedge conj \notin I} \mathcal{C}_{conj}$ and $\mathcal{S}_A = \bigcup_{\langle \bar{t}:conj \rangle \in A \wedge conj \in I} \mathcal{S}_{conj}$, where a true (resp. false) conjunction w.r.t. interpretation I is denoted by $conj \in I$ (resp. $conj \notin I$). Intuitively, $\mathcal{C}_A$ represents the reasons for false conjunctions in A, while $\mathcal{S}_A$ represents the reason for true conjunctions in A.

In the following, each propagation rule and the corresponding reason calculus are detailed. Without loss of generality, we focus on rules $h : -f(A)\Theta k$, $\Theta \in \{<, >\}$, since the calculus can easily be extended to the general case. More in detail, we consider two different scenarios depending on whether the propagation proceeds from literals in A to aggregate literals $f(A)\Theta k$ (forward inference) or the other-way round (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 conjunction occurring in A; whereas, in the second case, given a rule containing an aggregate atom which is already known to be true or false w.r.t. the current interpretation,[6] we infer some literals 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 conjunction in A is true or false w.r.t. 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 first assumed to be true; then, since the actual count 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 both

[5] Since a satisfied conjunction can have several "satisfying literals", the literal should be chosen as the reason that allows for the "longest jump," as argued in [18].

[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 [18].

propagation rules and corresponding reason calculus for the aggregates supported by DLV; $\#\texttt{max}\{A\}\Theta k$ is symmetric to $\#\texttt{min}\{A\}\Theta k$ and is not reported.

$\#\texttt{count}\{A\} < k$ **(resp.** $\#\texttt{count}\{A\} > k$**).** Suppose that there exists[7] a set $A' \subseteq A$ s.t. for each $\langle \bar{t} : conj\rangle \in A'$,[8] $conj$ is true (resp. false) in I and $|A'| \geq k$ (resp. $|A'| \geq |A| - k$), then $\#\texttt{count}\{A\} < k$ (resp. $\#\texttt{count}\{A\} > k$) is inferred to be false and its reasons are set to $\mathcal{S}_{A'}$ (resp. $\mathcal{C}_{A'}$). Conversely, suppose that there exists a set $A' \subseteq A$ s.t. for each $\langle \bar{t} : conj\rangle \in A'$, $conj$ is false (resp. true) in I and $|A'| > |A| - k$ (resp. $|A'| > k$), then we infer that $\#\texttt{count}\{A\} < k$ (resp. $\#\texttt{count}\{A\} > k$) is true and we set its reason to $\mathcal{C}_{A'}$ (resp. $\mathcal{S}_{A'}$).

$\#\texttt{min}\{A\} < k$ **(resp.** $\#\texttt{min}\{A\} > k$**).** Let A' be the set of all pairs $\langle v, \bar{t} : conj\rangle \in A$ s.t. $v < k$ (resp. $v \leq k$). If for each $\langle v, \bar{t} : conj\rangle \in A'$, $conj$ is false in I, then $\#\texttt{min}\{A\} < k$ is derived to be false (resp. $\#\texttt{min}\{A\} > k$ derived to be true) and we set its reasons to $\mathcal{S}_{conj_m}$. Conversely, suppose there exists a pair $\langle v, \bar{t} : conj\rangle \in A$ s.t. $conj$ is true in I and $v < k$ (resp. $v \leq k$), then we infer that $\#\texttt{min}\{A\} < k$ is true (resp. $\#\texttt{min}\{A\} > k$ is false) and set its reason to $\mathcal{S}_{conj}$.

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

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*[9] to satisfy it by inferring that some literals belonging to the conjunctions in A is true or false. For example, suppose that I is empty and consider the program:

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

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

[7] As far as the implementation is concerned, in case there are several different sets with this property, a safe choice is to consider their union. Another, less expensive, solution is to build A' by iterating over the elements of A until the condition is met.

[8] Hereafter, $\langle v, \bar{t} : conj\rangle$ is a syntactic shorthand for $\langle v, t_1, \cdots, t_n\rangle$, where v is a constant and $\bar{t}$ is the list of constants $t_1, \cdots, t_n$, $n \geq 0$.

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

true (independently by its aggregate set). At this point, backward propagation can happen, since the unique way to satisfy the aggregate literal is to infer both a and b to be true. Thus, backward propagation happens when an aggregate literal $f(A)\Theta k$ has been derived true (or false) in the current interpretation, and there is *only one way* to satisfy it by *deterministically* setting some $conj_i$ (s.t. $\langle \bar{t}_i : conj_i \rangle \in A$) true (or false) w.r.t I. For doing so, an implementation detail of DLV is exploited, which internally replaces conjunctions in aggregates by freshly introduced auxiliary atoms, along with a rule defining the auxiliary atom by means of the conjunction. So inside DLV, $conj_i$ will always be an atom, which can simply be set to true or false, and its defining rule will then act as a constraint eventually enforcing truth or falsity of the conjunction $conj_i$. As far as the reason calculus is concerned, literals are inferred to be true or false by this operation because both the aggregate literal is true/false and some conjunctions in A (being either true or false) made the process deterministic; thus, the reason for each literal l_i inferred by backward inference is set to $R(l_i) = R(f(A)\Theta k) \cup \mathcal{C}_A \cup \mathcal{S}_A$.

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) basically coincides with the ones of $f(A) < k+1$ to be false (resp. true), only one of the two cases is reported for each aggregate. Moreover, from now on, we assume that, whenever backward inference requires to derive something, this action can be done deterministically (if this is not possible then backward inference is not performed).

$\#\texttt{count}\{A\} < k$. Let T_A be the set T_A = $\{\langle \bar{t}_i : conj_i \rangle \in A$ s.t. $conj_i$ is true w.r.t. $I\}$, and F_A be the set F_A = $\{\langle \bar{t}_i : conj_i \rangle \in A$ s.t. $conj_i$ is false w.r.t. $I\}$, and suppose $\#\texttt{count}\{A\} < k$ is true w.r.t. I and $|T_A| = k - 1$, then all undefined conjunctions in A are made false. Conversely, suppose $\#\texttt{count}\{A\} < k$ is false w.r.t. I and $|A| - |F_A| = k$, then all undefined conjunctions in A are made true.

$\#\texttt{min}\{A\} < k$. Suppose that, $\#\texttt{min}\{A\} < k$ is true w.r.t. I, and there is only one $\langle v, \bar{t} : conj \rangle \in A$ such that $v < k$ and $conj$ is neither true or false w.r.t. I; suppose also that, all the remaining $\langle v_i, \bar{t}_i : conj_i \rangle \in A$ s.t. $v_i < k$ are such that $conj_i$ is false w.r.t. I, then $conj$ is made true. Conversely, suppose that $\#\texttt{min}\{A\} < k$ is false w.r.t. I and, there is no $\langle v, \bar{t} : conj \rangle \in A$ such that $v < k$ and $conj$ is true w.r.t. I. In addition, suppose that either (i) there exist $\langle v', \overline{t'} : conj' \rangle \in A$ s.t. $v' > k$ and $conj'$ is true w.r.t. I or (ii) there is only one $\langle v'', \overline{t''} : conj'' \rangle \in A$ s.t. $v'' > k$ with $conj''$ undefined w.r.t. I. Then all the $conj_i$ such that $\langle v_i, \bar{t}_i : conj_i \rangle \in A$ and $v_i < k$ are made to be false, and, if case (ii) holds, also $conj''$ is made true w.r.t. I.

$\#\texttt{max}\{A\} < k$. Suppose that, $\#\texttt{max}\{A\} < k$ is false w.r.t. I, and there is only one $\langle v, \bar{t} : conj \rangle \in A$ such that $v > k$ and $conj$ is neither true or false w.r.t. I; suppose also that, all the remaining $\langle v_i, \bar{t}_i : conj_i \rangle \in A$ s.t. $v_i > k$

are such that $conj_i$ is false w.r.t. I, then $conj$ is made true. Conversely, suppose that $\#\texttt{max}\{A\} < k$ is true w.r.t. I and, there is no $\langle v, \bar{t} : conj\rangle \in A$ such that $v > k$ and $conj$ is true w.r.t. I. In addition, suppose that either (i) there exist $\langle v', \overline{t'} : conj'\rangle \in A$ s.t. $v' < k$ and $conj'$ is true w.r.t. I or (ii) there is only one $\langle v'', \overline{t''} : conj''\rangle \in A$ s.t. $v'' < k$ with $conj''$ undefined w.r.t. I. Then all the $conj_i$ such that $\langle v_i, \bar{t}_i : conj_i\rangle \in A$ and $v_i < k$ are made to be false, and, in if case (ii) holds, also $conj''$ is made true w.r.t. I.

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

5 Look-back Heuristics in the Presence of Aggregates

Look-back heuristics, which have been originally exploited in SAT solvers like Chaff [13] (where the heuristic is called VSIDS), have also been considered for DLV in [4], in conjunction with backjumping, leading to positive results.

A key factor of this type of heuristic is the initialization of the weights of the literals [4], to be updated with the reasons calculus during the search. A common practice is to initialize those values with the number of occurrences in the input (ground) programs. But, if there are aggregates in the program, we would like to take them into account in order to guide the search. The idea is thus to implicitly consider the equivalent[10] standard program for an aggregate and count also these occurrences for the heuristic. It is worth noting that this equivalent program does not have to be "materialized" in memory. As before, we consider only rules of the form $h : -f(A)\Theta k$ for simplicity. We denote by $l_{i1}, \ldots, l_{im}$ the literals belonging to each $conj_i \in A$, ($m > 0$). Table 1 summarizes the formulas employed for computing literal occurrences. Note that equivalent programs in the case of $\#sum$ are quite involved, rendering the computation of the exact values fairly inefficient (many binomial coefficients). Therefore we decided to approximate the corresponding heuristic value, replacing $\#sum\{A\}$ by $\#count\{A^*\}$ where A^* contains v_i different elements, one for each $\langle v_i, \bar{t}_i : conj_i\rangle \in A$.

As an example, consider a rule of the form $h : -\#\texttt{min}\{A\} < k$. The equivalent standard program contains a rule of the type $h : -conj_i$, for each v_i, $1 \leq i \leq n$ s.t. $v_i < k$. In this way, h becomes true if one of the $conj_i$ having $v_i < k$ becomes true, i.e. if the minimum computed by the aggregate is less than k in current answer set. Thus, the number of occurrences of h in the corresponding standard program are $occ(h) = |\{v_i : \langle v_i, \bar{t}_i : conj_i\rangle \in A, v_i < k\}|$, while for each literal l_{iz}, i.e. the z-th literal of $conj_i$, $occ(l_{iz}) = 1$ if $v_i < k$, otherwise $occ(l_{iz}) = 0$.

[10] Equivalence in general holds only in a stratified setting, which however can serve as an approximation also in non-recursive settings.

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

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

Table 1: Occurrence formulas for literals involved in aggregates.

6 Experimental analysis

We have performed an experimental analysis on benchmarks with aggregates. In particular, we have considered some domains of the last ASP Competition[11] belonging to the MGS class, together with other benchmarks reported in [2]. For the domains of the ASP Competition, we have downloaded the benchmarks at "http://asparagus.cs.uni-potsdam.de/contest/downloads/benchmarks-mgs.tgz" and selected the logic programs with aggregates.

All the experiments were performed on a 3GHz PentiumIV equipped with 1GB of RAM, 2MB of level 2 cache running Debian GNU/Linux. Time measurements have been done using the `time` command shipped with the system, counting total CPU time for the respective process. We report the results in terms of execution time for finding one answer set, if any, within 20 minutes. Results are summarized in Table 2, where the first column reports the domain name, the second column the total number of instances considered (in the given domain), the third and fourth columns report the results for the standard version of DLV ver. of 2007-10-11 in the standard settings and the new system DLV^{BJA} featuring both backjumping and look-back heuristics, and the remaining columns report the results for CLASP [8] ver. 1.0.4, CMODELS [11] ver. 3.75, SMODELS [14] ver. 2.31 and SMODELS-CC [21] ver 1.08, which use LPARSE[12] for grounding. The results for the systems are presented as the mean CPU time of solved instances, along with the number of instances solved within the time limit (in parentheses). Regarding SMODELS-CC, two results are missing (i.e., there is a "no enc." in the Table) because it can not deal with weight constraint rules.

Domain	#I	DLV	DLV^{BJA}	CLASP	CMODELS	SMODELS	SMODELS-CC
BoundedSpanningTree	8	0.13 (8)	0.04 (8)	6.01 (8)	5.69 (8)	101.47 (5)	343.35 (8)
TowerOfHanoi	8	1.16 (8)	1.1 (8)	32.84 (8)	117.32 (7)	259.82 (8)	154.74 (7)
WeightedSpanningTree	8	0.04 (8)	0.02 (8)	2.16 (8)	2.31 (8)	28.51 (6)	no enc.
WeightedLatinSquares	8	542.23 (6)	140.83 (7)	0.03 (8)	0.34 (8)	326.2 (8)	no enc.
TimeTabling	9	4.49 (9)	0.34 (9)	1.15(9)	0.84 (9)	5.12 (3)	96.39 (9)

Table 2: Average execution times (s) (and number of solved instances).

It is useful to know what kinds of aggregates each domain involves: the third and fourth domains involve "$\#count$" and "$\#sum$", the first and last domains involve "$\#count$", while the second domain contains only the "$\#max$" aggregate.

We can see that the first three domains presented are easily solved by both DLV and DLV^{BJA}, slightly better by the enhanced system, while the remaining solvers show higher mean CPU time and/or solve less instances. The last two domains further show the potential of the enhanced system w.r.t. DLV, given that it is able to solve more instances (*WeightedLatinSquares* domain) in considerably shorter time (DLV^{BJA} is on average 15 times faster on *TimeTabling*, where the systems

[11] http://asparagus.cs.uni-potsdam.de/contest/.

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

solve the same instances, and significantly faster on *WeightedLatinSquares*, solving also more instances): interestingly, if compared to the remaining systems, this gain leads DLV^{BJA} to be the best performing solver in 4 domains out of 5 and it performs well in particular in the *TimeTabling* domain. Also in the *WeightedLatinSquares*, DLV^{BJA} has a clear advantage over DLV. However, DLV^{BJA} is still inferior with respect to CLASP, CMODELS and SMODELS.

We have conducted further investigations regarding the differences in performance in the particular domain *WeightedLatinSquares*. One explanation could be the absence of learning in DLV^{BJA}, but also other factors may be important, as discussed next. As a matter of fact, two main parameters affecting VSIDS behavior are the "importance" of literals in reasons (called "reward", i.e., how much the related counters for such literals is to be increased) and the constant factor by which counters are periodically divided (called "aging") in order to possibly focus the search on the last literals involved in reasons (see [4] for details on VSIDS heuristics). In the experiments we have presented so far, these parameters were set to 1, and 2, respectively, i.e., to the original values used by Chaff. But obviously, these might not be the best values for some domains, for example sometimes one would prefer higher values for these parameters in order to let the heuristic value updates take effect earlier in the search. We have informally conducted some experiments with different values for reward and aging. Interestingly, with some of the new setting we were able to solve all *WeightedLatinSquares*, indicating that also these factors may be an important reason for the comparatively poor performance of DLV^{BJA} for this domain.

We have also conducted further benchmarks on selected domains, comparing only DLV^{BJA} and DLV. Of these, we would like to mention as an example the *Seating* benchmarks from [2]. Here, DLV^{BJA} is able to solve more instances than DLV, with a mean CPU time of 1.24 for DLV^{BJA} and 31.46 seconds for DLV.

7 Related Work and Conclusion

Aggregates are an important linguistic enhancement of ASP, and most of the available systems are already able do deal with them. In particular, SMODELS [14, 15], CMODELS [11] and CLASP [8] support cardinality and weight constraints, which correspond to count and sum aggregates, respectively, while SMODELS$_{cc}$ [21] implements only cardinality constraints, and both GNT [10] and ASSAT [12] do not support aggregates. About solvers based on look-back techniques, aggregates are considered explicitly for backjumping in SMODELS$_{cc}$ (where additional arcs are added to the implication graph) and CLASP; conversely, CMODELS translates the original program into a propositional formula that is then evaluated by a SAT solver (possibly exploiting backjumping). Notably, none of the existing systems *directly* exploits aggegates for the computation of heuristics, indeed for all of them the conflict analysis works in a similar way as in the case of "normal" programs (i.e., by exploiting the UIP-based conflict analysis technique borrowed from SAT).

In this paper we have described look-back techniques for the evaluation of aggregates. In particular the main contributions are: (i) an extension of the *reason calculus* defined in [18]; and, (ii) an enhanced version of the heuristic presented in [4] that explicitly takes into account the presence of aggregates. 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.

Encouraged by the results of the performance evaluations, we are currently continuing our work in order to improve the performance of DLV^{BJA} by developing further optimizations both by enhancing the implementation of the reason calculus, by considering different "equivalent programs", and, thus, different VSIDS initializations and by tuning various VSIDS parameters. Additionally, we are also enlarging both the set of domains on which we conduct the performance evaluation, primarly considering other domains from the ASP Competition, and the set of systems, by including PBMODELS[13] in the analysis.

Acknowledgements

Supported by M.I.U.R. within projects "Potenziamento e Applicazioni della Programmazione Logica Disgiuntiva" and "Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione."

References

[1] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[2] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate Functions in DLV. In *Proceedings ASP03*, pages 274–288, CEUR Vol-78, 2003.

[3] W. Faber. *Enhancing Efficiency and Expressiveness in Answer Set Programming Systems*. PhD thesis, Institut für Informationssysteme, TU Wien, 2002.

[4] W. Faber, N. Leone, M. Maratea, and F. Ricca. Experimenting with Look-Back Heuristics for Hard ASP Programs. In *Proceedings of LPNMR 2007, LNAI)* 4483, pages 110–122, 2007. Springer.

[5] W. Faber, N. Leone, and G. Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *Proceedings of JELIA 2004, LNAI* 3229, pages 200–212. Springer, 2004.

[6] W. Faber, G. Pfeifer, N. Leone, T. Dell'Armi, and G. Ielpa. Design and implementation of aggregate functions in the dlv system. *TPLP*. in press.

[13] http://www.cs.uky.edu/ai/pbmodels/.

[7] J. Gaschnig. *Performance measurement and analysis of certain search algorithms*. PhD thesis, C.M. University, Pittsburgh, USA, 1979.

[8] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. Proc. of *IJCAI-07*, pp 386–392. Morgan Kaufmann, 2007.

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

[10] T. Janhunen and I. Niemelä. Gnt - a solver for disjunctive logic programs. In *Proceedings of LPNMR-7, LNAI* 2923, pages 331–335. Springer, 2004.

[11] Y. Lierler. Disjunctive Answer Set Programming via Satisfiability. In *Proceedings of LPNMR'05, LNAI* 3662, pages 447–451. Springer, 2005.

[12] F. Lin and Y. Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1–2):115–137, 2004.

[13] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of DAC 2001*, pages 530–535, Las Vegas, NV, USA, June 2001. ACM.

[14] I. Niemelä and P. Simons. Smodels – An Implementation of the Stable Model and Well-founded Semantics for Normal Logic Programs. In *Proceedings of LPNMR'97, LNAI* 1265, pages 420–429, Dagstuhl, Germany, 1997. Springer.

[15] I. Niemelä, P. Simons, and T. Soininen. Stable Model Semantics of Weight Constraint Rules. In *Proceedings of LPNMR'99), LNAI* 1730, 1999. Springer.

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

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

[18] F. Ricca, W. Faber, and N. Leone. A Backjumping Technique for Disjunctive Logic Programming. *AI Communications*, 19(2):155–172, 2006.

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

[20] T. C. Son and E. Pontelli. A Constructive Semantic Characterization of Aggregates in ASP. *TPLP*, 7:355–375, May 2007.

[21] J. Ward and J. S. Schlipf. Answer Set Programming with Clause Learning. In *Proceedings of LPNMR-7, LNAI* 2923, pages 302–313. Springer, Jan. 2004.