Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV*

Tina Dell'Armi
Dept. of Mathematics,
Univ. of Calabria
87030 Rende (CS), Italy
dellarmi@mat.unical.it

Wolfgang Faber
Inst. f. Informationssysteme 184/3,
TU Wien
A-1040 Wien, Austria
faber@kr.tuwien.ac.at

Giuseppe Ielpa
Dept. of Mathematics,
Univ. of Calabria
87030 Rende (CS), Italy
ielpa@mat.unical.it

Nicola Leone
Dept. of Mathematics, Univ. of Calabria
87030 Rende (CS), Italy
leone@unical.it

Gerald Pfeifer
Inst. f. Informationssysteme 184/2, TU Wien
A-1040 Wien, Austria
pfeifer@dbai.tuwien.ac.at

Abstract

Disjunctive Logic Programming (DLP) is a very expressive formalism: it allows to express every property of finite structures that is decidable in the complexity class Σ_2^P ($\mathrm{NP}^{\mathrm{NP}}$). Despite the high expressiveness of DLP, there are some simple properties, often arising in real-world applications, which cannot be encoded in a simple and natural manner. Among these, properties requiring to apply some arithmetic operators (like sum, times, count) on a set of elements satisfying some conditions, cannot be naturally expressed in DLP. To overcome this deficiency, in this paper we extend DLP by aggregate functions. We formally define the semantics of the new language, named DLP$^{\mathcal{A}}$. We show the usefulness of the new constructs on relevant knowledge-based problems. We analyze the computational complexity of DLP$^{\mathcal{A}}$, showing that the addition of aggregates does not bring a higher cost in that respect. We provide an implementation of the DLP$^{\mathcal{A}}$ language in DLV– the state-of-the-art DLP system – and report on experiments which confirm the usefulness of the proposed extension also for the efficiency of the computation.

1 Introduction

Expressiveness of DLP. Disjunctive Logic Programs (DLP) are logic programs where (nonmonotonic) negation may occur in the bodies, and disjunction may occur in the heads of rules. This language is very expressive in a precise mathematical sense: it allows to express every property of finite structures that is decidable in the complexity class Σ_2^P ($\mathrm{NP}^{\mathrm{NP}}$). Therefore, under widely believed assumptions, DLP is strictly more expressive than *normal* (*disjunction-free*) logic programming, whose expressiveness is limited to properties decidable in NP. DLP can thus express problems which cannot be translated to Satisfiability of CNF formulas in polynomial time. Importantly, besides enlarging the class of applications which can be encoded in the language, disjunction often allows for representing problems of lower complexity in a simpler and more natural fashion (see [Eiter *et al.*, 2000]).

The problem. Despite this high expressiveness, there are some simple properties, often arising in real-world applications, which cannot be encoded in DLP in a simple and natural manner. Among these are properties requiring to apply some arithmetic operator (e.g., sum, times, count) on a set of elements satisfying some conditions. Suppose, for instance, that you want to know if the sum of the salaries of the employees working in a team exceeds a given budget (see **Team Building**, in Section 3). To this end, you should first order the employees defining a successor relation. You should then define a *sum* predicate, in a recursive way, which computes the sum of all salaries, and compare its result with the given budget. This approach has two drawbacks: (1) It is bad from the KR perspective, as the encoding is not natural at all; (2) it is inefficient, as the (instantiation of the) program is quadratic (in the cardinality of the input set of employees). Thus, there is a clear need to enrich DLP with suitable constructs for the natural representation and to provide means for an efficient evaluation of such properties.

Contribution. We overcome the above deficiency of DLP. Instead of inventing new constructs from scratch, we extend the language with a sort of aggregate functions, first studied in the context of deductive databases, and implement them in DLV [Eiter *et al.*, 2000] – the state-of-the-art Disjunctive Logic Programming system. The main contributions of this paper are the following.

- We extend Disjunctive Logic Programming by aggregate functions and formally define the semantics of the resulting

*This work was supported by the European Commission under projects IST-2002-33570 INFOMIX, IST-2001-32429 ICONS, and FET-2001-37004 WASP.

language, named DLP$^{\mathcal{A}}$.

• We address knowledge representation issues, showing the impact of the new constructs on relevant problems.

• We analyze the computational complexity of DLP$^{\mathcal{A}}$. Importantly, it turns out that the addition of aggregates does not increase the computational complexity, which remains the same as for reasoning on DLP programs.

• We provide an implementation of DLP$^{\mathcal{A}}$ in the DLV system, deriving new algorithms and optimization techniques for the efficient evaluation.

• We report on experimentation, evaluating the impact of the proposed language extension on efficiency. The experiments confirm that, besides providing relevant advantages from the knowledge representation point of view, aggregate functions can bring significant computational gains.

• We compare DLP$^{\mathcal{A}}$ with related work.

We present the most relevant aspects of DLP$^{\mathcal{A}}$ and of its implementation here, referring the interested reader to a technical report with all details [Dell'Armi *et al.*, 2003].

2 The DLP$^{\mathcal{A}}$ Language

In this section, we provide a formal definition of the syntax and semantics of the DLP$^{\mathcal{A}}$ language – an extension of DLP by set-oriented functions (also called aggregate functions). We assume that the reader is familiar with standard DLP; we refer to atoms, literals, rules, and programs of DLP, as *standard atoms, standard literals, standard rules,* and *standard programs,* respectively. For further background, see [Gelfond and Lifschitz, 1991; Eiter *et al.*, 2000].

2.1 Syntax

A (DLP$^{\mathcal{A}}$) *set* 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 literals.[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 (variable free) conjunction of standard literals. An *aggregate function* is of the form $f(S)$, where S is a set, and f is a *function name* among $\#\mathtt{count}$, $\#\mathtt{min}$, $\#\mathtt{max}$, $\#\mathtt{sum}$, $\#\mathtt{times}$. An *aggregate atom* is $Lg \prec_1 f(S) \prec_2 Rg$, where $f(S)$ is an aggregate function, $\prec_1, \prec_2 \in \{=, <, \leq, >, \geq\}$, and Lg and Rg (called *left guard*, and *right guard*, respectively) are terms. One of "$Lg \prec_1$" and "$\prec_2 Rg$" can be omitted. An *atom* is either a standard (DLP) 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*.

A *(*DLP$^{\mathcal{A}}$*) rule* r is a construct

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

where $a_1, \cdots, a_n$ are standard atoms, $b_1, \cdots, b_m$ are atoms, and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of r, while the conjunction $b_1, ..., b_k, \mathtt{not}\ b_{k+1}, ..., \mathtt{not}\ b_m$ is the *body* of r. A *(*DLP$^{\mathcal{A}}$*) program* is a set of DLP$^{\mathcal{A}}$ rules.

Syntactic Restrictions and Notation

For simplicity, and without loss of generality, we assume that the body of each rule contains at most one aggregate atom. A *global* variable of a rule r is a variable appearing in some standard atom of r; a *local* variable of r is a variable appearing solely in an aggregate function in r.

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 $\{\,Vars : Conj\}$, also appears in a positive literal in $Conj$; (iii) each guard of an aggregate atom of r is either a constant or a global variable. A program is safe if all of its rules are safe.

Example 1 Consider the following rules:

$p(X){:-}q(X,Y,V), Y < \#\mathtt{max}\{Z : r(Z), \mathtt{not}\ a(Z,V)\}.$
$p(X){:-}q(X,Y,V), Y < \#\mathtt{sum}\{Z : \mathtt{not}\ a(Z,S)\}.$
$p(X){:-}q(X,Y,V), T < \#\mathtt{min}\{Z : r(Z), \mathtt{not}\ a(Z,V)\}.$

The first rule is safe, while the second is not, since both local variables Z and S violate condition (ii). The third rule is not safe either, since the guard T is not a global variable, violating condition (iii).

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 of $\mathcal{P}$, and for each rule $r \in \mathcal{P}$: (i) if a appears in the head of r, and b appears in an aggregate atom in the body of r, then $||b|| < ||a||$, and (ii) if a appears in the head of r, and b occurs in a standard atom in the body of r, then $||b|| \leq ||a||$.

Example 2 Consider the program consisting of a set of facts for predicates a and b, plus the following two rules:

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

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 legal level-mapping exists and the program becomes aggregate-unstratified.

Intuitively, aggregate-stratification forbids recursion through aggregates, which could cause an unclear semantics in some cases. Consider, for instance, the (aggregate-unstratified) program consisting only of rule $p(a){:-}\#\mathtt{count}\{X : p(X)\} = 0$. Neither $p(a)$ nor $\emptyset$ is an intuitive meaning for the program. We should probably assert that the above program does not have any answer set (defining a notion of "stability" for aggregates), but then positive programs would not always have an answer set if there is no integrity constraint. In the following we assume that DLP$^{\mathcal{A}}$ programs are safe and aggregate-stratified.

2.2 Semantics

Given a DLP$^{\mathcal{A}}$ program $\mathcal{P}$, let $U_{\mathcal{P}}$ denote the set of constants appearing in $\mathcal{P}$, $U_{\mathcal{P}}^{\mathbb{N}} \subseteq U_{\mathcal{P}}$ the set of the natural numbers occurring in $U_{\mathcal{P}}$, and $B_{\mathcal{P}}$ the set of standard atoms constructible from the (standard) predicates of $\mathcal{P}$ with constants in $U_{\mathcal{P}}$. Furthermore, given a set X, $\overline{2}^X$ denotes the set of all multisets over elements from X. Let us now describe the domains and the meanings of the aggregate functions we consider:

$\#\mathtt{count}$: defined over $\overline{2}^{U_{\mathcal{P}}}$, the number of elements in the set.

[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., $\{X : \exists Y s.t.\ a(X,Y), p(Y)\ \mathit{is\ true}\}$. Note that also negative literals may occur in the conjunction $Conj$ of a symbolic set.

$\#\mathtt{sum}$: defined over $\overline{2}^{U_{\mathcal{P}}^{\mathcal{N}}}$, the sum of the numbers in the set.
$\#\mathtt{times}$: over $\overline{2}^{U_{\mathcal{P}}^{\mathcal{N}}}$, the product of the numbers in the set.[2]
$\#\mathtt{min}$, $\#\mathtt{max}$: defined over $\overline{2}^{U_{\mathcal{P}}} - \{\emptyset\}$, the minimum/maximum element in the set; if the set contains also strings, the lexicographic ordering is considered.[3]
If the argument of an aggregate function does not belong to its domain, the aggregate evaluates to false, denoted as $\perp$ (and our implementation issues a warning in this case).

A *substitution* is a mapping from a set of variables to the set $U_{\mathcal{P}}$ of the constants appearing in the program $\mathcal{P}$. 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 set S* is the following ground set of pairs $inst(S)$:
$\{\langle\gamma(Vars) : \gamma(Conj)\rangle \mid \gamma$ *is a local substitution for* $S\}$.[4]
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 3 Consider the following program $\mathcal{P}_1$:
$q(1) \vee p(2,2). \qquad q(2) \vee p(2,1).$
$t(X) \text{:-} q(X), \#\mathtt{sum}\{Y : p(X,Y)\} > 1.$
The instantiation $Ground(\mathcal{P}_1)$ is the following:
$q(1) \vee p(2,2). \qquad q(2) \vee p(2,1).$
$t(1) \text{:-} q(1), \#\mathtt{sum}\{\langle 1 : p(1,1)\rangle, \langle 2 : p(1,2)\rangle\} > 1.$
$t(2) \text{:-} q(2), \#\mathtt{sum}\{\langle 1 : p(2,1)\rangle, \langle 2 : p(2,2)\rangle\} > 1.$

An *interpretation* for a DLP$^{\mathcal{A}}$ program $\mathcal{P}$ is a set of standard ground atoms $I \subseteq B_{\mathcal{P}}$. The truth valuation $I(A)$, where A is a standard ground literal or a standard ground conjunction, is defined in the usual way. Besides assigning truth values to the standard ground literals, an interpretation provides the meaning also to (ground) sets, aggregate functions and aggregate literals; 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 set S w.r.t. I is the multiset of the first constant of the first components of the elements in S whose conjunction is true w.r.t. I. More precisely, let $S_I = \{\langle t_1, ..., t_n\rangle \mid \langle t_1, ..., t_n : Conj\rangle \in S \wedge$ *Conj is true w.r.t. I* $\}$, then $I(S)$ is the multiset $[\, t_1 \mid \langle t_1, ..., t_n\rangle \in S_I \,]$. The valuation $I(f(S))$ of an aggregate function $f(S)$ w.r.t. I is the result of the application of the function f on $I(S)$. (If the multiset $I(S)$ is not in the domain of f, $I(f(S)) = \perp$.)

An aggregate atom $A = Lg \prec_1 f(S) \prec_2 Rg$ is *true w.r.t. I* if: (i) $I(f(S)) \neq \perp$, and, (ii) the relationships $Lg \prec_1 I(f(S))$, and $I(f(S)) \prec_2 Ug$ hold whenever they are present; otherwise, A is false.

Using the above notion of truth valuation for aggregate atoms, the truth valuations of aggregate literals and rules, as well as the notion of model, minimal model, and answer set for DLP$^{\mathcal{A}}$ are an immediate extension of the corresponding notions in DLP [Gelfond and Lifschitz, 1991].

Example 4 Consider the aggregate atom $A = \#\mathtt{sum}\{\langle 1 : p(2,1)\rangle, \langle 2 : p(2,2)\rangle\} > 1$ from Example 3. Let S be the ground set appearing in A. For interpretation $I = \{q(2), p(2,2), t(2)\}$, $I(S) = [2]$, the application of $\#\mathtt{sum}$ over [2] yields 2, and A is therefore true w.r.t. I, since $2 > 1$. I is an answer set of the program of Example 3.

3 Knowledge Representation in DLP$^{\mathcal{A}}$

In this section, we show how aggregate functions can be used to encode relevant problems.

Team Building. A project team has to be built from a set of employees according to the following specifications:
(p_1) The team consists of a certain number of employees.
(p_2) At least a given number of different skills must be present in the team.
(p_3) The sum of the salaries of the employees working in the team must not exceed the given budget.
(p_4) The salary of each individual employee is within a specified limit.
(p_5) The number of women working in the team has to reach at least a given number.

Suppose that our employees are provided by a number of facts of the form *emp(EmpId,Sex,Skill,Salary)*; the size of the team, the minimum number of different skills, the budget, the maximum salary, and the minimum number of women are specified by the facts *nEmp(N), nSkill(N), budget(B), maxSal(M)*, and *women(W)*. We then encode each property p_i above by an aggregate atom A_i, and enforce it by an integrity constraint containing not A_i.
$in(I) \vee out(I) \text{:-} emp(I, Sx, Sk, Sa).$
$\text{:-} nEmp(N), \mathtt{not}\ \#\mathtt{count}\{I : in(I)\} = N.$
$\text{:-} nSkill(M), \mathtt{not}\ \#\mathtt{count}\{Sk : emp(I, Sx, Sk, Sa), in(I)\} \geq M.$
$\text{:-} budget(B), \mathtt{not}\ \#\mathtt{sum}\{Sa, I : emp(I, Sx, Sk, Sa), in(I)\} \leq B.$
$\text{:-} maxSal(M), \mathtt{not}\ \#\mathtt{max}\{Sa : emp(I, Sx, Sk, Sa), in(I)\} \leq M.$
$\text{:-} women(W), \mathtt{not}\ \#\mathtt{count}\{I : emp(I, f, Sk, Sa), in(I)\} \geq W.$

Intuitively, the disjunctive rule "guesses" whether an employee is included in the team or not, while the five constraints correspond one-to-one to the five requirements p_1-p_5. Thanks to the aggregates the translation of the specifications is surprisingly straightforward. The example highlights the usefulness of representing both sets and multisets in our language (a multiset can be obtained by specifying more than one variable in $Vars$ of a symbolic set $\{Vars : Conj\}$). For instance, the encoding of p_2 requires a set, as we want to count *different* skills; two employees in the team having the same skill, should count once w.r.t. p_2. On the contrary, p_3 requires to sum the elements of a multiset; if two employees have the same salary, *both* salaries should be summed up for p_3. This is obtained by adding the variable I to $Vars$. The valuation of $\{Sa, I : emp(I, Sx, Sk, Sa), in(I)\}$ yields the set $S = \{\langle Sa, I\rangle : Sa\ is\ the\ salary\ of\ employee\ I\ in\ the\ team\}$. Then, the sum function is applied on the multiset of the first components Sa of the tuples $\langle Sa, I\rangle$ in S (see Section 2.2).

Seating. We have to generate a sitting arrangement for a number of guests, with m tables and n chairs per table.

[2] $\#\mathtt{sum}$ and $\#\mathtt{times}$ applied over an empty set return 0 and 1, respectively.

[3] The latter is not yet supported in our first implementation.

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

Guests who like each other should sit at the same table; guests who dislike each other should not sit at the same table.

Suppose that the number of chairs per table is specified by $nChairs(X)$ and that $person(P)$ and $table(T)$ represent the guests and the available tables, respectively. Then, we can generate a seating arrangement by the following program:

% Guess whether person P sits at table T or not.
$at(P,T) \vee not\_at(P,T) \text{:-} person(P), table(T).$
% The persons sitting at a table cannot exceed the chairs.
$\text{:-} table(T), nChairs(C), \texttt{not}\ \#\texttt{count}\{P : at(P,T)\} \leq C.$
% A person is seated at precisely one table; equivalent
% to $\text{:-} person(P), at(P,T), at(P,U), T <> U.$
$\text{:-} person(P), \texttt{not}\ \#\texttt{count}\{T : at(P,T)\} = 1.$
% People who like each other should sit at the same table...
$\text{:-} like(P1,P2), at(P1,T), \texttt{not}\ at(P2,T).$
% ...while people who dislike each other should not.
$\text{:-} dislike(P1,P2), at(P1,T), at(P2,T).$

4 Computational Complexity of $\text{DLP}^{\mathcal{A}}$

As for the classical nonmonotonic formalisms [Marek and Truszczyński, 1991], two important decision problems, corresponding to two different reasoning tasks, arise in $\text{DLP}^{\mathcal{A}}$:

(Brave Reasoning) Given a $\text{DLP}^{\mathcal{A}}$ program $\mathcal{P}$ and a ground literal L, is L true in some answer set of $\mathcal{P}$?

(Cautious Reasoning) Given a $\text{DLP}^{\mathcal{A}}$ program $\mathcal{P}$ and a ground literal L, is L true in all answer sets of $\mathcal{P}$?

The following theorems report on the complexity of the above reasoning tasks for propositional (i.e., variable-free) $\text{DLP}^{\mathcal{A}}$ programs that respect the syntactic restrictions imposed in Section 2 (safety and aggregate-stratification). Importantly, it turns out that reasoning in $\text{DLP}^{\mathcal{A}}$ does not bring an increase in computational complexity, which remains exactly the same as for standard DLP. (See [Dell'Armi *et al.*, 2003] for the proofs.)

Theorem 5 *Brave Reasoning on ground* $\text{DLP}^{\mathcal{A}}$ *programs is* Σ_2^P*-complete.*

Theorem 6 *Cautious Reasoning on ground* $\text{DLP}^{\mathcal{A}}$ *programs is* Π_2^P*-complete.*

5 Implementation Issues

The implementation of $\text{DLP}^{\mathcal{A}}$ required changes to all modules of DLV. Apart from a preliminary standardization phase, most of the effort concentrated on the Intelligent Grounding and Model Generator modules.

Standardization. After parsing, each aggregate A is transformed such that both guards are present and both $\prec_1$ and $\prec_2$ are set to $\leq$. The conjunction $Conj$ of the symbolic set of A is replaced by a single, new atom Aux and a rule $Aux \text{:-} Conj$ is added to the program (the arguments of Aux being the distinct variables of $Conj$).

Instantiation. The goal of the instantiator is to generate a ground program which has precisely the same answer sets as the theoretical instantiation $Ground(\mathcal{P})$, but is sensibly smaller. The instantiation proceeds bottom-up following the dependencies induced by the rules, and, in particular, respecting the ordering imposed by the aggregate-stratification. Let "$H \text{:-} B, aggr.$" be a rule, where H is the head of the rule, B is the conjunction of the standard body literals, and $aggr$ is an aggregate literal over a symbolic set $\{Vars : Aux\}$. First we compute an instantiation $\overline{B}$ for the literals in B; this binds the global variables appearing in Aux. The (partially bound) atom $\overline{Aux}$ is then matched against its extension (since the bottom-up instantiation respects the stratification, the extension of $\overline{Aux}$ is already available), all matching facts are computed, and a set of pairs $\{\langle\theta_1(Vars) : \theta_1(\overline{Aux})\rangle,\ldots,\langle\theta_n(Vars) : \theta_n(\overline{Aux})\rangle\}$ is generated, where θ_i is a substitution for the local variables in $\overline{Aux}$ such that $\theta_i(\overline{Aux})$ is an admissible instance of $\overline{Aux}$ (recall that DLV's instantiator produces only those instances of a predicate which can potentially become true [Faber *et al.*, 1999a; Leone *et al.*, 2001])[5]. Also, we only store those elements of the symbolic set whose truth value cannot be determined yet and process the others dynamically, (partially) evaluating the aggregate already during instantiation. The same process is then repeated for all further instantiations of the literals in B.

Example 7 Consider the following rule r:

$$p(X) \text{:-} q(X), 1 < \#\texttt{count}\{Y : a(X,Y), \texttt{not}\ b(Y)\}.$$

The standardization rewrites r to:

$$p(X) \text{:-} q(X), 2 \leq \#\texttt{count}\{Y : aux(X,Y)\} \leq \infty.$$
$$aux(X,Y) \text{:-} a(X,Y), \texttt{not}\ b(Y).$$

Suppose that the instantiation of the rule for aux generates 3 potentially true facts for aux: *aux(1,a), aux(1,b),* and *aux(2,c).* If the potentially true facts for q are $q(1)$ and $q(2)$, the following ground instances are generated:[6]

$$p(1) \text{:-} q(1), 2 \leq \#\texttt{count}\{\langle a : aux(1,a)\rangle, \langle b : aux(1,b)\rangle\} \leq \infty.$$
$$p(2) \text{:-} q(2), 2 \leq \#\texttt{count}\{\langle c : aux(2,c)\rangle\} \leq \infty.$$

Duplicate Sets Recognition. To optimize the evaluation, we have designed a hashing technique which recognizes multiple occurrences of the same set in the program, even in different rules, and stores them only once. This saves memory (sets may be very large), and also implies a significant performance gain, especially in the model generation where sets are frequently manipulated during the backtracking process.

Example 8 Consider the following two constraints:

$$c_1: \ \text{:-}\ 10 \leq \#\texttt{max}\{V : d(V,X)\}.$$
$$c_2: \ \text{:-}\ \#\texttt{min}\{Y : d(Y,Z)\} \leq 5.$$

Our technique recognizes that the two sets are equal, and generates only one instance which is shared by c_1 and c_2.

Now assume that both constraints additionally contain a standard literal $p(X)$. In this case, c_1 and c_2 have n instances each, where n is the number of facts for $p(X)$. By means of our technique, each pair of instances of c_1 and c_2 shares a common set, reducing the number of instantiated sets by half.

Model Generation. We have designed an extension of the Deterministic Consequences operator of the DLV system [Faber *et al.*, 1999b] for $\text{DLP}^{\mathcal{A}}$ programs. The new operator makes both forward and backward inferences on aggregate atoms, resulting in an effective pruning of the search space. We have then extended the Dowling and Gallier algorithm [Dowling and Gallier, 1984] to compute a fixpoint of

[5] A ground atom A can potentially become true only if we have generated a ground instance with A in the head.

[6] Note that a ground set contains only those aux atoms which are potentially true.

this operator in linear time using a multi-linked data structure of pointers. Given a ground set T, say, $\{\langle t_1^1, ..., t_n^1 : Aux^1\rangle, ..., \langle t_1^m, ..., t_n^m : Aux^m\rangle$, this structure allows us to access T in $O(1)$ whenever some Aux^i changes its truth value (supporting fast forward propagation); on the other hand, it provides direct access from T to each Aux^i atom (supporting fast backward propagation).

6 Experiments and Benchmarks

To assess the usefulness of the proposed DLP extension and evaluate its implementation, we compare the following two methods for solving a given problem:

• DLV$^\mathcal{A}$. Encode the problem in DLP$^\mathcal{A}$ and solve it by using our extension of DLV with aggregates.

• DLV. Encode the problem in standard DLP and solve it by using standard DLV.

To generate DLP encodings from DLP$^\mathcal{A}$ encodings, suitable logic definitions of the aggregate functions are employed (which are recursive for `#count`, `#sum`, and `#times`).

We compare these methods on two benchmark problems:

Time Tabling is a classical planning problem. In particular, we consider the problem of planning the timetable of lectures which some groups of students have to take. We consider a number of real-world instances at our University, where instance k deals with k groups.

Seating is the problem described in Section 3. We consider 4 (for small instances) or 5 (for larger instances) seats per table, with increasing numbers of tables and persons (with $numPersons = numSeats * numTables$). For each problem size (i.e., seats/tables configuration), we consider classes with different numbers of like resp. dislike constraints, where the percentages are relative to the maximum number of like resp. dislike constraints such that the problem is not overconstrained[7]. In particular, we consider the following classes: -) no like/dislike constraints at all; -) 25% like constraints; -) 25% like and 25% dislike constraints; -) 50% like constraints; -) 50% like and 50% dislike constraints. For each problem size, we have randomly generated 10 instances for each class above.

For Seating we use the DLP$^\mathcal{A}$ encoding reported in Section 3; all encodings and benchmark data are also available on the web at `http://www.dlvsystem.com/examples/ijcai03.zip`.

Number of	Execution Time		Instantiation Size	
Groups	DLV	DLV$^\mathcal{A}$	DLV	DLV$^\mathcal{A}$
1	10.95	0.55	91217	6972
2	36.79	2.05	178533	13986
3	79.84	4.68	264938	20888
4	147.53	7.86	367014	29029
5	224.46	12.30	436544	36043
6	321.85	17.18	518950	42767
7	437.94	25.36	606361	49993
8	618.23	37.78	761429	61916
9	-	57.00	-	74027

Figure 1: Experimental Results for Timetabling

[7]Beyond these maxima there is trivially no solution.

Number of	Execution Time		Instantiation Size	
Persons	DLV	DLV$^\mathcal{A}$	DLV	DLV$^\mathcal{A}$
8	0.010	0.010	320	101
12	0.034	0.010	996	248
16	26.872	0.011	2272	490
25	-	0.024	6643	1346
50	-	0.307	50029	7559
75	-	1.883	165442	22049
100	-	7.082	387886	47946
125	-	64.293	752769	88781
150	-	152.450	1294977	147567

Figure 2: Experimental Results for Seating

We ran the benchmarks on AMD Athlon 1.2 machines with 512MB of memory, using FreeBSD 4.7 and GCC 2.95. We have allowed a maximum running time of 1800 seconds per instance and a maximum memory usage of 256MB. Cumulated results are provided in Figures 1 and 2. In particular, for Timetabling we report the execution time and the size of the residual ground instantiation (the total number of atoms occurring in the instantiation, where multiple occurrences of the same atom are counted).[8] For Seating, the execution time is the average running time over the instances of the same size. A "-" symbol in the tables indicates that the corresponding instance (some of the instances of that size, for Seating) was not solved within the allowed time and memory limits.

On both problems, DLV$^\mathcal{A}$ clearly outperforms DLV. On Timetabling, the execution time of DLV$^\mathcal{A}$ is one order of magnitude lower than that of DLV on all problem instances, and DLV could not solve the last instances within the allowed memory and time limits. On Seating, the difference becomes even more significant. DLV could solve only the instances of small size (up to 16 persons - 4 tables, 4 seats); while DLV$^\mathcal{A}$ could solve significantly larger instances in a reasonable time. The information about the instantiation sizes provides an explanation for such a big difference between the execution times of DLV and DLV$^\mathcal{A}$. Thanks to the aggregates, the DLP$^\mathcal{A}$ encodings of Timetabling and Seating are more succinct than the corresponding encodings in standard DLP; this succinctness is also reflected in the ground instantiations of the programs. Since the evaluation algorithms are then exponential (in the worst case) in the size of the instantiation, the execution times of DLV$^\mathcal{A}$ turn out to be much shorter than the execution times of DLV.

7 Related Works

Aggregate functions in logic programming languages appeared already in the 80s, when their need emerged in deductive databases like LDL [Chimenti *et al.*, 1990] and were studied in detail, cf. [Ross and Sagiv, 1997; Kemp and Ramamohanarao, 1998]. However, the first implementation in Answer Set Programming, based on the Smodels system, is recent [Simons *et al.*, 2002].

Comparing DLP$^\mathcal{A}$ to the language of Smodels, we observe a strong similarity between cardinality constraints there and `#count`. Also `#sum` and weight constraints in Smodels are

[8]Note that also atoms occurring in the sets of the aggregates are counted for the instantiation size.

similar in spirit. Indeed, the DLP$^{\mathcal{A}}$ encodings of both Team Building and Seating can be easily translated to Smodels' language. However, there are some relevant differences. For instance, in DLP$^{\mathcal{A}}$ aggregate atoms can be negated, while cardinality and weight constraint literals in Smodels cannot. Smodels, on the other hand, allows for weight constraints in the heads of rules, while DLP$^{\mathcal{A}}$ aggregates cannot occur in heads. (The presence of weight constraints in heads is a powerful KR feature; however, it causes the loss of some semantic property of nonmonotonic languages [Marek and Remmel, 2002].) Observe also that DLP$^{\mathcal{A}}$ aggregates like #`min`, #`max`, and #`times` do not have a counterpart in Smodels. Moreover, DLP$^{\mathcal{A}}$ provides a general framework where further aggregates can be easily accommodated (e.g., #`any` and #`avg` are already under development). Furthermore, note that symbolic sets of DLP$^{\mathcal{A}}$ directly represent pure (mathematical) sets, and can also represent multisets rather naturally (see the discussion on Team Building in Section 3). Smodels weight constraints, instead, work on multisets, and additional rules are needed to encode pure sets; for instance, Condition p_2 of Team Building cannot be encoded directly in a constraint, but needs the definition of an extra predicate. A positive aspect of Smodels is that, thanks to stricter safety conditions (all variables are to be restricted by *domain predicates*), it is able to deal with recursion through aggregates, which is forbidden in DLP$^{\mathcal{A}}$. Finally, note that DLP$^{\mathcal{A}}$ deals with sets of terms, while Smodels deals with sets of atoms. As far as the implementation is concerned, also Smodels is endowed with advanced pruning operators for weight constraints, which are efficiently implemented; we are not aware, though, of techniques for the automatic recognition of duplicate sets in Smodels.

DLP$^{\mathcal{A}}$ also seems to be very similar to a special case of the semantics for aggregates discussed in [Gelfond, 2002], which we are currently investigating.

Another interesting research line uses 4-valued logics and approximating operators to define the semantics of aggregate functions in logic-based languages [Denecker *et al.*, 2001; 2002; Pelov, 2002]. These approaches are founded on very solid theoretical grounds, and appear very promising, as they could provide a clean formalization of a very general framework for arbitrary aggregates in logic programming and nonmonotonic reasoning, where aggregate atoms can also "produce" new values (currently, in both DLP$^{\mathcal{A}}$ and Smodels the guards of the aggregates need to be bound to some value). However, these approaches sometimes amount to a higher computational complexity [Pelov, 2002], and there is no implementation available so far.

8 Conclusion

We have proposed DLP$^{\mathcal{A}}$, an extension of DLP by aggregate functions, and have implemented DLP$^{\mathcal{A}}$ in the DLV system. On the one hand, we have demonstrated that the aggregate functions increase the knowledge modeling power of DLP, supporting a more natural and concise knowledge representation. On the other hand, we have shown that aggregate functions do not increase the complexity of the main reasoning tasks. Moreover, the experiments have confirmed that the succinctness of the encodings employing aggregates has a strong positive impact on the efficiency of the computation.

Future work will concern the introduction of further aggregate operators, the relaxation of the syntactic restrictions of DLP$^{\mathcal{A}}$, and the design of further optimization techniques and heuristics to improve the efficiency of the computation.

We thank the anonymous reviewers for their thoughtful comments and suggestions for improvements of this paper.

References

[Chimenti *et al.*, 1990] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL System Prototype. *IEEE TKDE*, 2(1), 1990.

[Dell'Armi *et al.*, 2003] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Semantics and Computation of Aggregate Functions in Disjunctive Logic Programming. Tech. Report INFSYS RR-1843-03-07, TU Wien, April 2003.

[Denecker *et al.*, 2001] M. Denecker, N. Pelov, and M. Bruynooghe. Ultimate Well-Founded and Stable Model Semantics for Logic Programs with Aggregates. *ICLP-2001*, pp. 212–226. Springer, 2001.

[Denecker *et al.*, 2002] M. Denecker, V. Marek, and M. Truszczyński. Ultimate Approximations in Monotonic Knowledge Representation Systems. *KR-2002*, pp. 177–188.

[Dowling and Gallier, 1984] W. F. Dowling and J. H. Gallier. Linear-time Algorithms for Testing the Satisfability of Propositional Horn Formulae. *JLP*, 3:267–284, 1984.

[Eiter *et al.*, 2000] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving Using the DLV System. *Logic-Based Artificial Intelligence*, pp. 79–103. Kluwer, 2000.

[Faber *et al.*, 1999a] W. Faber, N. Leone, C. Mateis, and G. Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. *DDLP'99*, pp. 135–139.

[Faber *et al.*, 1999b] W. Faber, N. Leone, and G. Pfeifer. Pushing Goal Derivation in DLP Computations. *LPNMR'99*, pp. 177–191. Springer.

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

[Gelfond, 2002] M. Gelfond. Representing Knowledge in A-Prolog. *Computational Logic. Logic Programming and Beyond*, number 2408 in LNCS, pp. 413–451. Springer, 2002.

[Kemp and Ramamohanarao, 1998] D. B. Kemp and K. Ramamohanarao. Efficient Recursive Aggregation and Negation in Deductive Databases. *IEEE TKDE*, 10:727–745, 1998.

[Leone *et al.*, 2001] N. Leone, S. Perri, and F. Scarcello. Improving ASP Instantiators by Join-Ordering Methods. *LPNMR'01*, LNAI 2173. Springer, September 2001.

[Marek and Remmel, 2002] V.W. Marek and J.B. Remmel. On Logic Programs with Cardinality Constraints. *NMR'2002*, pp. 219–228, April 2002.

[Marek and Truszczyński, 1991] V.W. Marek and M. Truszczyński. Autoepistemic Logic. *JACM*, 38(3):588–619, 1991.

[Pelov, 2002] N. Pelov. Non-monotone Semantics for Logic Programs with Aggregates. `http://www.cs.kuleuven.ac.be/~pelov/papers/nma.ps.gz`., October 2002.

[Ross and Sagiv, 1997] K. A. Ross and Y. Sagiv. Monotonic Aggregation in Deductive Databases. *Journal of Computer and System Sciences*, 54(1):79–97, February 1997.

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