Enhancing the Magic-Set Method for Disjunctive Datalog Programs

Chiara Cumbo[1], Wolfgang Faber[2], Gianluigi Greco[1], and Nicola Leone[1]

[1] Dipartimento di Matematica, Università della Calabria, 87030 Rende, Italy,
{cumbo,greco,leone}@mat.unical.it
[2] Institut für Informationssysteme, TU Wien, 1040 Wien, Austria,
faber@kr.tuwien.ac.at

Abstract. We present a new technique for the optimization of (partially) bound queries over disjunctive datalog programs. The technique exploits the propagation of query bindings, and extends the Magic-Set optimization technique (originally defined for non-disjunctive programs) to the disjunctive case, substantially improving on previously defined approaches.
Magic-Set-transformed disjunctive programs frequently contain redundant rules. We tackle this problem and propose a method for preventing the generation of such superfluous rules during the Magic-Set transformation. In addition, we provide an efficient heuristic method for the identification of redundant rules, which can be applied in general, even if Magic-Sets are not used.
We implement all proposed methods in the DLV system – the state-of-the-art implementation of disjunctive datalog – and perform some experiments. The experimental results confirm the usefulness of Magic-Sets for disjunctive datalog, and they highlight the computational gain obtained by our method, which outperforms significantly the previously proposed Magic-Set method for disjunctive datalog programs.

1 Introduction

Disjunctive datalog (Datalog$^\vee$) programs are logic programs where disjunction may occur in the heads of rules [1, 2]. Disjunctive datalog is very expressive in a precise mathematical sense: it allows to express every property of finite ordered structures that is decidable in the complexity class Σ_2^P [2]. Therefore, under widely believed assumptions, Datalog$^\vee$ is strictly more expressive than *normal* (*disjunction-free*) datalog which can express only problems of lower complexity. 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 [3]).

Recently, disjunctive datalog is employed in several projects, mainly due to the availability of some efficient inference engines, such as the DLV system [4] and the GnT system [5]. E.g., in [6] this formalism has been shown to be very well-suited for database repair, and the European Commission has funded a couple of IST projects focusing on the exploitation of disjunctive datalog in "hot" application areas like information integration and knowledge management.[3]

The increasing application of disjunctive datalog systems stimulates the research on algorithms and optimization techniques, which make these systems more efficient and more widely applicable. Within this framework, we investigate here a promising line of research consisting of the extension of deductive database techniques and, specifically, of binding propagation techniques exploited in the Magic-Set method [7–12], to nonmonotonic logic languages like disjunctive datalog.

Intuitively, the goal of the Magic-Set method (originally defined for non-disjunctive datalog queries only) is to use the constants appearing in the query to reduce the size of the instantiation by eliminating "a priori" a number of ground instances of the rules which cannot contribute to the derivation of the query goal.

The first extension of Magic-Set method to disjunctive programs is due to [13], where the author observes that binding propagation strategies have to be changed for disjunctive rules so that each time a head predicate receives some binding from the query, it eventually propagates this relevant information to all the other head predicates as well as to the body predicates (see Section 3.1). An algorithm implementing the above strategy has been also proposed in [13]. Roughly, it is a rewriting algorithm that bloats the program with some additional predicates (called *collecting* predicates), besides the standard "magic" ones (intrinsic in the Magic-Set method) in order to make the propagation strategy work — in the following we call this algorithm Auxiliary Predicates Method (APM).

In this paper we provide fresh and refined ideas (w.r.t. [13]) for extending the Magic-Set method to disjunctive datalog queries. In particular, we observe that the method in [13] has two major drawbacks. First, the introduction of the new (collecting) predicates enlarges the size of the grounding and consequently reduces the gain that could be potentially achieved by the optimization. Second, several redundant (which are subsumed by the rest of the program) rules are frequently generated by the application of this method. Since the number of rules in a program is a critical performance factor, these redundancies can deteriorate run-time behavior. In extreme cases this overhead alone can outweigh the benefits of the optimization — since the evaluation of a disjunctive datalog program requires exponential time in the size of its instantiation, a polynomial increase in the size of the program instantiation may give an exponential increase in the program evaluation time.

Here, we address both problems above. Specifically, the main contribution is the following:

$\triangleright$ **We define a new Magic-Set method for disjunctive datalog.** The new method, called Disjunctive Magic-Set (DMS), overcomes some drawbacks of the previous magic-set methods for disjunctive datalog. We provide an algorithm for the proposed DMS method, which involves a generalization of sideways information passing to the disjunctive case. Importantly, we formally prove the correctness of the DMS method by

[3] The exploitation of disjunctive datalog for information integration is the main focus of the INFOMIX project (IST-2001-33570); while an application of disjunctive datalog for knowledge management is studied in ICONS (IST-2001-32429).

showing that given a query Q over a program P, the brave and cautious answers of Q over P coincide, respectively, with the brave and cautious answers of Q over P', where P' is the rewriting of P under DMS.
$\triangleright$ **We design effective techniques for avoiding redundant rules.** The head-to-head binding propagation needed for disjunctive programs (see [13] and Section 3.1), very often causes the generation of many redundant rules (both APM and DMS are affected by this problem). We experimentally observe that the presence of redundant rules slows down the computation significantly, and may even counterpoise the advantages of the magic sets optimization. Thus, we design two techniques for redundant-rules prevention and elimination, respectively. The former technique prevents some cases of generation of redundant rules, by storing some extra information on the binding-propagation flow. Since the problem of redundant-rule identification is untractable (like clause subsumption), to eliminate "a posteriori" redundant rules (which could not be avoided by the former technique), we design a new and efficient heuristic for identifying redundant rules. Note that this heuristic is not specific for the disjunctive Magic-Set method, and can be applied for any type of logic program, even in the presence of unstratified negation and constraints. The enhancement of DMS with both our redundancy prevention and elimination techniques, yields an improved method, called Optimized Disjunctive Magic-Set Method (ODMS).
$\triangleright$ **We implement all the proposed methods and techniques.** In particular, we implement the DMS method and its enhancements for redundancy prevention and elimination (yielding ODMS), in the DLV system [4] – the state-of-the-art implementation of disjunctive datalog. Both DMS and ODMS are fully integrated in the DLV system, and their are completely transparent to the end user that can simply enable them by setting the corresponding option. The interested reader can retrieve from `http://www.dlvsystem.com/magic/` a downloadable executable of the DLV system in which an option for using DMS or ODMS is provided — the same url contains some hints for its usage.
$\triangleright$ **We evaluate the efficiency of the implemented method:** We have performed extensive experiments using benchmarks reported in the literature, comparing the performance of the DLV system without optimization, with APM of [13], with DMS, and with ODMS. These experiments show that our methods, especially ODMS, yields speedups in many cases and only rarely produces mild overheads w.r.t. the native DLV system, greatly improving on APM of [13].

2 Preliminaries

2.1 Disjunctive Datalog Queries

A *disjunctive rule* r is of the form $a_1 \texttt{ v } \cdots \texttt{ v } a_n \texttt{ :- } b_1, \cdots, b_k.$, where $a_1, \cdots, a_n, b_1, \cdots, b_k$ are atoms and $n \geq 1$, $k \geq 0$. The disjunction $a_1 \texttt{ v } \cdots \texttt{ v } a_n$ is the *head* of r, while the conjunction $b_1, \ldots, b_k$ is the *body* of r. Moreover, let $H(r) = \{a_1, \ldots, a_n\}$ and $B(r) = \{b_1, \ldots, b_k\}$. A non-disjunctive rule with an empty body (i.e. $n = 1$ and $k = 0$) is called a *fact*. If a predicate is defined only by facts, it is referred to as *EDB predicate*, otherwise as *IDB predicate*. Throughout this paper, we assume that rules are *safe*, that is, each variable of a rule r appears in a positive literal of the body of r. A *disjunctive datalog program* (short. Datalog$^{\vee}$ program) $\mathcal{P}$ is a finite set of rules; if $\mathcal{P}$ is disjunction-free, then it is a *datalog program* (Datalog program). A *query* Q is a non-empty conjunction $b_1, \cdots, b_k$ of atoms.

Given a program $\mathcal{P}$, we denote by $ground(\mathcal{P})$ the set of all the rules obtained by applying to each rule $r \in \mathcal{P}$ all possible substitutions from the variables in r to the set of all the constants in $\mathcal{P}$. The semantics of a program $\mathcal{P}$ is given by the set $\mathcal{MM}(\mathcal{P})$ of the subset-minimal models of $\mathcal{P}$. Note that on Datalog$^{\vee}$ the notion of *answer set* [1] coincides to the notion of minimal model.

Let $\mathcal{P}$ be a Datalog$^{\vee}$ program and let $\mathcal{F}$ be a set of facts. Then, we denote by $\mathcal{P}_{\mathcal{F}}$ the program $\mathcal{P}_{\mathcal{F}} = \mathcal{P} \cup \mathcal{F}$. Given a query $\mathcal{Q}$ and an interpretation M of $\mathcal{P}$, $\vartheta(\mathcal{Q}, M)$ denotes the set containing each substitution ϕ for the variables in $\mathcal{Q}$ such that $\phi(\mathcal{Q})$ is true in M. The answer to a query $\mathcal{Q}$ over $\mathcal{P}_{\mathcal{F}}$, under the *brave* semantics, denoted by $Ans_b(\mathcal{Q}, \mathcal{P}_{\mathcal{F}})$, is the set $\cup_M \vartheta(\mathcal{Q}, M)$, such that $M \in \mathcal{MM}(\mathcal{P} \cup \mathcal{F})$. The answer to a query $\mathcal{Q}$ over the facts in $\mathcal{F}$, under the *cautious* semantics, denoted by $Ans_c(\mathcal{Q}, \mathcal{P}_{\mathcal{F}})$, is the set $\cap_M \vartheta(\mathcal{Q}, M)$, such that $M \in \mathcal{MM}(\mathcal{P} \cup \mathcal{F}) \neq \emptyset$. If $\mathcal{MM}(\mathcal{P} \cup \mathcal{F}) = \emptyset$, then all substitutions over the universe for variables in $\mathcal{Q}$ are in the cautious answer. Finally, we say that programs $\mathcal{P}$ and $\mathcal{P}'$ are *bravely* (resp. *cautiously*) *equivalent* w.r.t. $\mathcal{Q}$, denoted by $\mathcal{P} \equiv_{\mathcal{Q},b} \mathcal{P}'$ (resp. $\mathcal{P} \equiv_{\mathcal{Q},c} \mathcal{P}'$), if for any set $\mathcal{F}$ of facts $Ans_b(\mathcal{Q}, \mathcal{P}_{\mathcal{F}}) = Ans_b(\mathcal{Q}, \mathcal{P}'_{\mathcal{F}})$ (resp. $Ans_c(\mathcal{Q}, \mathcal{P}_{\mathcal{F}}) = Ans_c(\mathcal{Q}, \mathcal{P}'_{\mathcal{F}})$).

2.2 Magic-Set for Non-disjunctive Datalog Queries

We will illustrate how the Magic-Set method simulates the top-down evaluation of a query by considering the program consisting of the rules $\mathtt{path(X,Y)} :\!-\ \mathtt{edge(X,Y)}$. and $\mathtt{path(X,Y)} :\!-\ \mathtt{edge(X,Z)},\ \mathtt{path(Z,Y)}$. together with query $\mathtt{path(1,5)}$?.

Adornment Step: The key idea is to materialize, by suitable *adornments*, binding information for IDB predicates which would be propagated during a top-down computation. These are strings of the letters b and f, denoting bound or free for each argument of an IDB predicate. First, adornments are created for query predicates. The adorned version of the query above is $\mathtt{path^{bb}(1,5)}$.

The query adornments are then used to propagate their information into the body of the rules defining it, simulating a top-down evaluation. Obviously various strategies can be pursued concerning the order of processing the body atoms and the propagation of bindings. These are referred to as Sideways Information Passing Strategies (*SIPS*), cf. [9]. Any SIPS must guarantee an iterative processing of all body atoms in r. Let q be an atom that has not yet been processed, and v be the set of already considered atoms, then a SIPS specifies a propagation $\mathtt{v} \rightarrow_{\chi} \mathtt{q}$, where χ is the set of the variables bound by v, passing their values to q.

In the first rule of the example ($\mathtt{path(X,Y)} :\!-\ \mathtt{edge(X,Y)}$.) a binding is only passed to the EDB predicate edge (which will not be adorned), yielding the adorned rule $\mathtt{path^{bb}(X,Y)} :\!-\ \mathtt{edge(X,Y)}$. In the second rule, $\mathtt{path^{bb}(X,Y)}$ passes its binding information to $\mathtt{edge(X,Z)}$ by $\mathtt{path^{bb}(X,Y)} \rightarrow_{\{\mathtt{X}\}} \mathtt{edge(X,Z)}$. $\mathtt{edge(X,Z)}$ itself is not adorned, but it gives a binding to Z. Then, we consider $\mathtt{path(Z,Y)}$, for which we obtain the propagation $\mathtt{path^{bb}(X,Y)},\ \mathtt{edge(X,Z)} \rightarrow_{\{\mathtt{Y,Z}\}} \mathtt{path(Z,Y)}$. This causes the generation of the adorned atom $\mathtt{path^{bb}(Z,Y)}$, and the resulting adorned rule is $\mathtt{path^{bb}(X,Y)} :\!-\ \mathtt{edge(X,Z)},\ \mathtt{path^{bb}(Z,Y)}$.

In general, adorning a rule may generate new adorned predicates. This step is repeated until all adorned predicates have been processed, yielding the *adorned program*, in our example it consists of the rules $\texttt{path}^{\texttt{bb}}\texttt{(X,Y) :- edge(X,Y).}$ and $\texttt{path}^{\texttt{bb}}\texttt{(X,Y) :- edge(X,Z), path}^{\texttt{bb}}\texttt{(Z,Y).}$

Generation Step: The adorned program is used to generate *magic rules*, which simulate the top-down evaluation scheme. Let the *magic version* ***magic***$(\mathtt{p}^{\alpha})$ for an adorned atom $\mathtt{p}^{\alpha}$ be defined as $\mathtt{magic\_p}^{\alpha}$ in which all arguments labeled f in α are eliminated.

Then, for each adorned atom p in the body of an adorned rule r_a, a magic rule r_m is generated such that (i) the head of r_m consists of ***magic***(p), and (ii) the body of r_m consists of the magic version of the head atom of r_a, followed by all of the predicates of r_a which can propagate the binding on p. In our example we generate $\texttt{magic\_path}^{\texttt{bb}}\texttt{(Z,Y) :- magic\_path}^{\texttt{bb}}\texttt{(X,Y), edge(X,Z).}$

Modification Step: The adorned rules are modified by including magic atoms generated in Step 2 in the rule bodies. The resultant rules are called *modified rules*. For each adorned rule whose head is h, we extend the rule body by inserting ***magic***(h). In our example, $\texttt{path}^{\texttt{bb}}\texttt{(X,Y) :- magic\_path}^{\texttt{bb}}\texttt{(X,Y), edge(X,Y).}$ and $\texttt{path}^{\texttt{bb}}\texttt{(X,Y) :- magic\_path}^{\texttt{bb}}\texttt{(X,Y), edge(X,Z), path}^{\texttt{bb}}\texttt{(Z,Y).}$ are generated.

Processing of the Query: For each adorned atom $\mathtt{g}^{\alpha}$ of the query, (1) the *magic seed* ***magic***$(\mathtt{g}^{\alpha})$. is asserted, and (2) a rule $\mathtt{g} \texttt{ :- } \mathtt{g}^{\alpha}$ is produced. In our example we generate $\texttt{magic\_path}^{\texttt{bb}}\texttt{(1,5).}$ and $\texttt{path(X,Y) :- path}^{\texttt{bb}}\texttt{(X,Y).}$

The complete rewritten program consists of the magic, modified, and query rules. Given a non-disjunctive datalog program $\mathcal{P}$, a query $\mathcal{Q}$, and the rewritten program $\mathcal{P}'$, it is well known (see e.g. [7]) that $\mathcal{P}$ and $\mathcal{P}'$ are equivalent w.r.t. $\mathcal{Q}$, i.e., $\mathcal{P} \equiv_{\mathcal{Q},b} \mathcal{P}'$ and $\mathcal{P} \equiv_{\mathcal{Q},c} \mathcal{P}'$ hold (since brave and cautious semantics coincide for non-disjunctive datalog programs).

3 Magic-Set Method for Disjunctive Datalog Programs

In this section we present the Disjunctive Magic-Set algorithm (short. DMS) for the optimization of disjunctive datalog programs, which has been implemented and integrated into the DLV system [4]. Before discussing the details of the algorithm, we informally present the main ideas that have been exploited for enabling the Magic-Set method to work on disjunctive programs.

3.1 Binding Propagation in Datalog$^{\vee}$ Programs: some Key Issues

As first observed in [13], while in nondisjunctive programs bindings are propagated only head-to-body, any sound rewriting for disjunctive programs has to propagate bindings also head-to-head in order to preserve soundness. Roughly, suppose that a predicate p is relevant for the query, and a disjunctive rule r contains p(X) in the head. Then, besides propagating the binding from p(X) to the body of r (as in the nondisjunctive case), a sound rewriting has to propagate the binding also from p(X) to the other head atoms of r. Consider, for instance, a Datalog$^{\vee}$ program $\mathcal{P}$ containing rule `p(X) v q(Y) :- a(X,Y), r(X).` and the query p(1)?. Even though the query propagates the binding for the predicate p, in order to correctly answer the query, we also need to evaluate the truth value of $\mathtt{q(Y)}$, which indirectly receives the binding through the body predicate $\mathtt{a(X,Y)}$. For instance, suppose that the program contains facts $\mathtt{a(1,2)}$, and $\mathtt{r(1)}$; then atom $\mathtt{q(2)}$ is relevant for query $\mathtt{p(1)}$? (i.e., it should belong to the magic set of the query), since the truth of $\mathtt{q(2)}$ would invalidate the derivation of $\mathtt{p(1)}$ from the above rule, because of the minimality of the semantics. It follows that, while propagating the binding, the head atoms of disjunctive rules must be all adorned as well.

However, the adornment of the head of one disjunctive rule $\mathtt{r}$ may give rise to multiple rules, having different adornments for the head predicates. This process can be somehow seen as "splitting" $\mathtt{r}$ in multiple rules. While this is not a problem in the nondisjunctive case, the semantics of a disjunctive program may be affected. Consider, for instance, the program $\mathtt{p(X,Y)\ v\ q(Y,X)\ {:}{-}\ a(X,Y).}$ in which $\mathtt{p}$ and $\mathtt{q}$ are mutually exclusive (due to minimality), since they do not appear in any other rule head. Assuming the adornments $\mathtt{p^{bf}}$ and $\mathtt{q^{bf}}$ to be propagated, we might obtain rules whose heads have the form $\mathtt{p^{bf}(X,Y)\ v\ q^{fb}(Y,X)}$ (derived while propagating $\mathtt{p^{bf}}$) and $\mathtt{p^{fb}(X,Y)\ v\ q^{bf}(Y,X)}$ (derived while propagating $\mathtt{q^{bf}}$). These rules could support two atoms $\mathtt{p^{bf}(m,n)}$ and $\mathtt{q^{bf}(n,m)}$, while in the original program $\mathtt{p(m,n)}$ and $\mathtt{p(n,m)}$ could not hold simultaneously (due to semantic minimality), thus changing the original semantics.

The method proposed in [13] circumvents this problem by using some auxiliary predicates which collect all facts coming from the different adornments. For instance, in the above example, two rules of the form $\mathtt{collect\_p(X,Y)\ {:}{-}\ p^{fb}(X,Y).}$ and $\mathtt{collect\_p(X,Y)\ {:}{-}\ p^{bf}(X,Y).}$ are added for predicate $\mathtt{p}$. The main drawback of this approach is that collecting predicates, while resolving the semantic problem, bloat the program with additional rules reducing the gain of the optimization.

A relevant advantage of our algorithm (confirmed also by an experimental analysis) is that we do not use collecting predicates; rather, we preserve the correct semantics by stripping off the adornments from non-magic predicates in modified rules. Other computational advantages come from our adornment technique, which is obtained by extending non-disjunctive SIPS to the disjunctive case.

3.2 DMS Algorithm

The salient feature of our algorithm is that we generate modified and magic rules on a rule-by-rule basis. To this end, we exploit a stack S of predicates for storing all the adorned predicates to be used for propagating the binding of the query: At each step, an element is removed from S, and each defining rule is processed at a time. Thus, adorned rules do not have to be stored.

The algorithm DMS (see Figure 1) implements the Magic-Set method for disjunctive programs. Its input is a disjunctive datalog program $\mathcal{P}$ and a query $\mathcal{Q}$. Note that the algorithm can be used for non-disjunctive rules as a special case. If the query contains some non-free IDB predicates, it outputs a (optimized) program $\mathtt{DMS}(\mathcal{Q},\mathcal{P})$ consisting of a set of *modified* and *magic* rules, stored by means of the sets $\mathit{modifiedRules}(\mathcal{Q},\mathcal{P})$ and $\mathit{magicRules}(\mathcal{Q},\mathcal{P})$, respectively. The main steps of the algorithm DMS are illustrated by means of the following running example.

Example 1 (Strategic Companies [14]). We are given a collection C of companies producing some goods in a set G, such that each company $c_i \in C$ is controlled by a set of

```
Input: A Datalog$^\vee$ program $\mathcal{P}$, and a query $\mathcal{Q} = \mathtt{g_1(t_1)}, \ldots, \mathtt{g_n(t_n)}$.
Output: The optimized program $\mathtt{DMS}(\mathcal{Q}, \mathcal{P})$.
var $S$: stack of adorned predicates; $modifiedRules(\mathcal{Q}, \mathcal{P})$,$magicRules(\mathcal{Q}, \mathcal{P})$: set of rules;
begin
 1. if $\mathtt{g_1(t_1)}, \ldots, \mathtt{g_n(t_n)}$ has some IDB predicate then
 2.    $modifiedRules(\mathcal{Q}, \mathcal{P})$:=$\emptyset$; $\langle S$, $magicRules(\mathcal{Q}, \mathcal{P})\rangle$:=BuildQuerySeeds($\mathcal{Q}$);
 3.    while $S \neq \emptyset$ do
 4.       $p^\alpha$:=$S$.pop();
 5.       for each rule $r \in \mathcal{P}$: $\mathtt{p(t)\ v\ p_1(t_1)\ v\ \ldots\ v\ p_n(t_n)\ \text{:-}\ q_1(s_1), \ldots, q_m(s_m)}$ do
 6.          $r_a$:=Adorn($r$,$p^\alpha$,$S$);
 7.          $magicRules(\mathcal{Q}, \mathcal{P})$ := $magicRules(\mathcal{Q}, \mathcal{P})$ $\bigcup$ Generate($r_a$);
 8.          $modifiedRules(\mathcal{Q}, \mathcal{P})$ := $modifiedRules(\mathcal{Q}, \mathcal{P})$ $\bigcup$ {Modify($r_a$)};
 9.       end for
 10.   end while
 11.   $\mathtt{DMS}(\mathcal{Q}, \mathcal{P})$:=$magicRules(\mathcal{Q}, \mathcal{P})$ $\cup$ $modifiedRules(\mathcal{Q}, \mathcal{P})$;
 12. return $\mathtt{DMS}(\mathcal{Q}, \mathcal{P})$;
 13. end if
end.
```

Fig. 1. Disjunctive Magic-Set Method

other companies $O_i \subseteq C$. A subset of the companies $C' \subset C$ is a *strategic set* set if it is a minimal set of companies producing all the goods in G, such that if $O_i \subseteq C'$ for some $i = 1, \ldots, m$ then $c_i \in C'$ must hold. This scenario can be modelled by means of the following program $\mathcal{P}_{sc}$.

r_1 : $\mathtt{sc(C_1)\ v\ sc(C_2)\ \text{:-}\ produced\_by(P, C_1, C_2).}$
r_2 : $\mathtt{sc(C)\ \text{:-}\ controlled\_by(C, C_1, C_2, C_3),\ sc(C_1),\ sc(C_2),\ sc(C_3).}$

Moreover, given a company $\mathtt{c} \in C$, we consider a query $\mathcal{Q}_{sc} = \mathtt{sc(c)}$ asking whether $\mathtt{c}$ belongs to some strategic set of C. □

The computation starts in step *2* by initializing $modifiedRules(\mathcal{Q}, \mathcal{P})$ to the empty set. Then, the function ***BuildQuerySeeds*** is used for storing in $magicRules(\mathcal{Q}, \mathcal{P})$ the magic seeds, and pushing on the stack S the adorned predicates of $\mathcal{Q}$. Note that we do not generate any query rules, because the transformed program will not contain adornments.

Example 2. Given the query $\mathcal{Q}_{sc} = \mathtt{sc(c)}$ and the program $\mathcal{P}_{sc}$, ***BuildQuerySeeds*** creates $\mathtt{magic\_sc^b(c)}$. and pushes $\mathtt{sc^b}$ onto the stack S. □

The core of the technique (steps *4*-*9*) is repeated until the stack S is empty, i.e., until there is no further adorned predicate to be propagated. Specifically, an adorned predicate $\mathtt{p}^\alpha$ is removed from the stack S in step *4*, and its binding is propagated in each (disjunctive) rule r in $\mathcal{P}$ of the form

r : $\mathtt{p(t)\ v\ p_1(t_1)\ v\ \ldots\ v\ p_n(t_n)\ \text{:-}\ q_1(s_1),\ \ldots,\ q_m(s_m).}$

with $\mathtt{n} \geq 0$, having an atom $\mathtt{p(t)}$ in the head (step *5*).
Adorn. Step *6* performs the adornment of the rule. Different from the case of non-disjunctive programs, the binding of the predicate $\mathtt{p}^\alpha$ needs to be also propagated to the atoms $\mathtt{p_1(t_1)}, \ldots, \mathtt{p_n(t_n)}$ in the head. We achieve this by defining an extension of any non-disjunctive SIPS to the disjunctive case. The constraint for such a disjunctive SIPS is that head atoms (different from $\mathtt{p(t)}$) cannot provide variable bindings, they can

only *receive* bindings (similarly to negative literals in standard SIPS). So they should be processed only once all their variables are bound or do not occur in yet unprocessed body atoms.[4] Moreover they cannot make any of their free-variables bound.

The function ***Adorn*** produces an adorned disjunctive rule from an adorned predicate and a suitable unadorned rule by employing the refined SIPS, pushing all newly adorned predicates onto S. Hence, in step *6* the rule r_a is of the form

$$r_a : \mathtt{p}^{\alpha}(\mathtt{t}) \ \mathtt{v} \ \mathtt{p}_1^{\alpha_1}(\mathtt{t}_1) \ \ldots \ \mathtt{p}_n^{\alpha_n}(\mathtt{t}_n) \mathrel{:\!-} \mathtt{q}_1^{\beta_1}(\mathtt{s}_1), \ldots, \mathtt{q}_m^{\beta_m}(\mathtt{s}_m).$$

Example 3. Consider again Example 1. When $\mathtt{sc}^{\mathtt{b}}$ is removed from the stack, we first select rule r_1 and the head predicate $\mathtt{sc}(\mathtt{C}_1)$. Then, the adorned version is

$$r'_{1_a} : \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_1) \ \mathtt{v} \ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_2) \mathrel{:\!-} \mathtt{produced\_by}(\mathtt{P}, \mathtt{C}_1, \mathtt{C}_2).$$

Next r_1 is processed again, this time with head predicate $\mathtt{sc}(\mathtt{C}_2)$, producing

$$r''_{1_a} : \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_2) \ \mathtt{v} \ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_1) \mathrel{:\!-} \mathtt{produced\_by}(\mathtt{P}, \mathtt{C}_1, \mathtt{C}_2).$$

and finally, processing r_2 we obtain

$$r_{2_a} : \mathtt{sc}^{\mathtt{b}}(\mathtt{C}) \mathrel{:\!-} \mathtt{controlled\_by}(\mathtt{C}, \mathtt{C}_1, \mathtt{C}_2, \mathtt{C}_3), \ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_1), \ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_2), \ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_3).$$

□

Generate. The algorithm uses the adorned rule r_a for generating and collecting the magic rules in step *7*. Since r_a is a disjunctive rule, ***Generate*** first produces a non-disjunctive intermediate rule by moving head atoms into the body. Then, the standard technique for Datalog rules, as described in *Generation Step* in Section 2, is applied.

Example 4. In the program of Example 3, from the rule r'_{1_a} first its non-disjunctive intermediate rule

$$\mathtt{sc}^{\mathtt{b}}(\mathtt{C}_1) \mathrel{:\!-} \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_2), \mathtt{produced\_by}(\mathtt{P}, \mathtt{C}_1, \mathtt{C}_2).$$

is produced, from which the magic rule

$$\mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}_2) \mathrel{:\!-} \mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}_1), \ \mathtt{produced\_by}(\mathtt{P}, \mathtt{C}_1, \mathtt{C}_2).$$

is generated. Similarly, from the rule r''_{1_a} we obtain

$$\mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}_1) \mathrel{:\!-} \mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}_2), \ \mathtt{produced\_by}(\mathtt{P}, \mathtt{C}_1, \mathtt{C}_2).$$

and finally r_{2_a} gives rise to the following rules

$$\begin{array}{l}
\mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}_1) \mathrel{:\!-} \mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}), \ \mathtt{controlled\_by}(\mathtt{C}, \mathtt{C}_1, \mathtt{C}_2, \mathtt{C}_3).\\
\mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}_2) \mathrel{:\!-} \mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}), \ \mathtt{controlled\_by}(\mathtt{C}, \mathtt{C}_1, \mathtt{C}_2, \mathtt{C}_3).\\
\mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}_3) \mathrel{:\!-} \mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}), \ \mathtt{controlled\_by}(\mathtt{C}, \mathtt{C}_1, \mathtt{C}_2, \mathtt{C}_3).
\end{array}$$

□

[4] Recall that the safety constraint guarantees that each variable of a head atom also appears in some positive body-atom.

Modify. In step *8* the modified rules are generated and collected. The only difference to the non-disjunctive case is that the adornments are stripped off the original atoms — see Section 3.1. Hence, the function ***Modify*** constructs a rule of the following form

$$\begin{aligned}\mathtt{p(t)\,v\,p_1(t_1)\,v\,\ldots\,v\,p_n(t_n)}\ \mathtt{:-}\ & \boldsymbol{magic}(\mathtt{p}^{\alpha}(\mathtt{t})), \boldsymbol{magic}(\mathtt{p}_1^{\alpha_1}(\mathtt{t_1})), \ldots, \boldsymbol{magic}(\mathtt{p}_n^{\alpha_n}(\mathtt{t_n})),\\ & \mathtt{q_1(s_1)}, \ldots, \mathtt{q_m(s_m)}.\end{aligned}$$

Finally, after all the adorned predicates have been processed the algorithm outputs the program $\mathtt{DMS}(\mathcal{Q}, \mathcal{P})$.

Example 5. In our running example, we derive the following set of modified rules:

r'_{1_m} : $\mathtt{sc(C_1)\ v\ sc(C_2)\ {:-}\ magic\_sc^b(C_1),\ magic\_sc^b(C_2),\ produced\_by(P, C_1, C_2).}$
r''_{1_m} : $\mathtt{sc(C_2)\ v\ sc(C_1)\ {:-}\ magic\_sc^b(C_2),\ magic\_sc^b(C_1),\ produced\_by(P, C_1, C_2).}$
r_{2_m} : $\mathtt{sc(C)\ {:-}\ magic\_sc^b(C),\ controlled\_by(C, C_1, C_2, C_3),\ sc(C_1),\ sc(C_2),\ sc(C_3).}$

where r'_{1_m} (resp. r''_{1_m}, r_{2_m}) is derived by adding magic predicates and stripping off adornments for the rule r'_{1_a} (resp. r''_{1_a}, r_{2_a}). Thus, the optimized program $\mathtt{DMS}(\mathcal{Q}_{sc}, \mathcal{P}_{cs})$ comprises the above modified rules as well as the magic rules in Example 4, and the magic seed $\mathtt{magic\_sc^b(c)}$. □

3.3 Query Equivalence Results

We conclude the presentation of the DMS algorithm by formally proving its soundness. To this aim proofs in [13] cannot be reused, due to the many differences w.r.t. our approach. The result is shown by first establishing a relationship between the minimal models of the program $\mathtt{DMS}(\mathcal{Q}, \mathcal{P})$ and of the program $rel(\mathcal{Q}, \mathcal{P})$ constructed as follows.

Given a set $\mathcal{S}$ of ground rules of $\mathcal{P}$, we denote by $\mathbf{R}(\mathcal{S})$ the set $\{r \in ground(\mathcal{P}) \mid \exists r' \in \mathcal{S}, \exists q \in B(r') \cup H(r')$ s.t. $q \in H(r)\}$. Then, $rel(\mathcal{Q}, \mathcal{P})$ is the least fixed point of the following succession $rel_0(\mathcal{Q}, \mathcal{P}) = \{r \in ground(\mathcal{P}) \mid \exists ground(q) \in Q \cap H(r)\}$, and $rel_{i+1}(\mathcal{Q}, \mathcal{P}) = \mathbf{R}(rel_i(\mathcal{Q}, \mathcal{P}))$, for each $i > 0$.

Notice that the correspondence between the models of $\mathtt{DMS}(\mathcal{Q}, \mathcal{P})$ and of $rel(\mathcal{Q}, \mathcal{P})$ can be established by focusing on non-magic atoms only. Thus, we next exploit the following notation. Given a model M and a predicate symbol g, we denote by $M[g]$ the set of atoms in M whose predicate symbol is g. Then, $M[\mathcal{P}]$ denotes the set of atoms in M whose predicate symbol appears in the head of some rule of $\mathcal{P}$. Finally, given a set of interpretations S, let $S[g] = \{M[g] | M \in S\}$ and $S[\mathcal{P}] = \{M[\mathcal{P}] | M \in S\}$.

Lemma 1. *Given a Datalog$^{\vee}$ program $\mathcal{P}$, and a query $\mathcal{Q}$. Then, it holds that $\forall M' \in \mathcal{MM}(\mathtt{DMS}(\mathcal{Q}, \mathcal{P}))$, and $\exists M \in \mathcal{MM}(rel(\mathcal{Q}, \mathcal{P}))$ s.t. $M = M'[rel(\mathcal{Q}, \mathcal{P})]$.*

Lemma 2. *Given a Datalog$^{\vee}$ program $\mathcal{P}$, and a query $\mathcal{Q}$. Then, it holds that $\forall M \in \mathcal{MM}(rel(\mathcal{Q}, \mathcal{P}))$, and $\exists M' \in \mathcal{MM}(\mathtt{DMS}(\mathcal{Q}, \mathcal{P}))$ s.t. $M = M'[rel(\mathcal{Q}, \mathcal{P})]$.*

Armed with the above results, we can prove the following.

Theorem 1 (Soundness of the DMS Algorithm). *Let $\mathcal{P}$ be a Datalog$^{\vee}$ program, let $\mathcal{Q}$ be a query. Then, $\mathtt{DMS}(\langle \mathcal{Q}, \mathcal{P} \rangle) \equiv_{\mathcal{Q},b} \mathcal{P}$ and $\mathtt{DMS}(\langle \mathcal{Q}, \mathcal{P} \rangle) \equiv_{\mathcal{Q},c} \mathcal{P}$ hold.*

Proof (Sketch). Let $\overline{rel}(\mathcal{Q},\mathcal{P})$ denote the set $ground(\mathcal{P}) - rel(\mathcal{Q},\mathcal{P})$. After lemmas 1 and 2, it suffices to prove that $rel(\mathcal{Q},\mathcal{P}) \equiv_{\mathcal{Q},b} \mathcal{P}$ and $rel(\mathcal{Q},\mathcal{P}) \equiv_{\mathcal{Q},c} \mathcal{P}$. In fact, we can show that $ground(\mathcal{P})$ is partitioned into two modules (see definitions and notations in [2]), i.e., $rel(\mathcal{Q},\mathcal{P}) \triangleright \overline{rel}(\mathcal{Q},\mathcal{P})$, that can be hierarchically evaluated. Thus, the models of $\mathcal{P}$ are such that $\mathcal{MM}(\mathcal{P}) = \bigcup_M \mathcal{MM}(M \cup \overline{rel}(\mathcal{Q},\mathcal{P}))$, for each $M \in \mathcal{MM}(rel(\mathcal{Q},\mathcal{P}))$, where for the sake of simplicity, the model M is also used for denoting the set of the corresponding ground facts in it.

The results follows by observing that for each predicate q in $\mathcal{Q}$, $\mathcal{MM}(\mathcal{P})[q] = (\mathcal{MM}(\mathcal{P})[rel(\mathcal{Q},\mathcal{P})])[q]$. In fact, we can show that $\mathcal{MM}(\mathcal{P})[rel(\mathcal{Q},\mathcal{P})] = \mathcal{MM}(rel(\mathcal{Q},\mathcal{P}))$. Then, it suffices to observe that for each predicate q in $\mathcal{Q}$, the set of ground rules having q in the head is in $rel_0(\mathcal{Q},\mathcal{P}) \subseteq rel(\mathcal{Q},\mathcal{P})$. □

4 Redundant rules: Prevention and Checking

Both the DMS method described above and APM of [13] have a common drawback: Numerous redundant rules may be generated, which can deteriorate the optimization. For instance, in Example 5 the first two modified rules coincide, and this might happen even if the two head predicates differ. We stress that our rewriting algorithm already drastically reduces the impact of such phenomena, as it does not introduce additional predicates and rules (apart from magic rules). Nevertheless, since this aspect is crucial for the optimization process, we next devise some strategies for further reducing the overhead.

Let $\mathcal{P}$ be a disjunctive datalog program, and let r_1 and r_2 be two rules of $\mathcal{P}$. Then, r_1 is *subsumed* by r_2 (denoted by $r_1 \sqsubseteq r_2$) if there exists a substitution ϑ for $H(r_2) \cup B(r_2)$, such that $\vartheta(H(r_2)) \subseteq H(r_1)$ and $\vartheta(B(r_2)) \subseteq B(r_1)$. Finally, a rule r_1 is *redundant* if there exists a rule r_2 such that $r_1 \sqsubseteq r_2$. Unfortunately, deciding whether a rule is subsumed by another rule is a hard task:

Theorem 2. *Let $\mathcal{P}$ be a disjunctive datalog program, and let r_1 and r_2 be two rules of $\mathcal{P}$. Then, the problem of deciding whether $r_1 \sqsubseteq r_2$ is NP-complete in the number of variables of r_1 (program complexity). Hardness holds even for $B(r_1) = B(r_2) = \emptyset$.*

The above result strongly motivates the design of methods for preventing the generation of redundant rules as well as of polynomial time heuristics for their identification. The latter aspect is also of interest outside the context of the Magic-Set method.

4.1 Prevention of Redundant Rules

There are two typical situations in which redundant rules may be generated: **(S1)** when adorning a disjunctive rule with two predicates having the same adornment and arguments, and **(S2)** when adorning a rule with an adorned predicate, which stems solely from a previous adornment of the same rule.

Example 6. **(S1)** Assume that the adorned predicates $\mathtt{p}^{\mathtt{b}}$ and $\mathtt{s}^{\mathtt{b}}$ are used for propagating the binding in the rule `p(X) v s(X) :- a(X)`. Then, both of the modified rules will eventually result in `s(X) v p(X) :- magic_s`$^{\mathtt{b}}$`(X), magic_p`$^{\mathtt{b}}$`(X), a(X)`. □

The source of the redundancy lies in the fact that disjunctive rules may be adorned by two distinct predicates ($\mathtt{s}^{\mathtt{b}}$ and $\mathtt{p}^{\mathtt{b}}$ in the example) sharing the same bound variables.

Example 7. **(S2)** Consider the rule $\mathtt{s(X,Z)\ v\ p(X,Y)\ {:-}\ a(X),\ b(Y),\ c(Z).}$ and the query $\mathtt{s(1,2)?}$. By adorning with $\mathtt{s}^{\mathtt{bb}}$ we obtain the modified rule

$$r_1 :\ \mathtt{s(X,Z)\ v\ p(X,Y)\ {:-}\ magic\_s^{bb}(X,Z),\ magic\_p^{bf}(X),\ a(X),\ b(Y),\ c(Z).}$$

and $\mathtt{p}^{\mathtt{bf}}$ is pushed onto the stack, which gives rise to

$$r_2 :\ \mathtt{s(X,Z)\ v\ p(X,Y)\ {:-}\ magic\_s^{bf}(X),\ magic\_p^{bf}(X),\ a(X),\ b(Y),\ c(Z).}$$

which is not syntactically subsumed by nor subsumes r_1.

Nonetheless, if $\mathtt{p}^{\mathtt{bf}}$ is generated only by the above rule, then r_2 will add no significant information as for the relevance of $\mathtt{s}^{\mathtt{bf}}$, as it would propagate to $\mathtt{s}^{\mathtt{bf}}$ the same binding it had received from predicate $\mathtt{s}$ itself. Conversely, if the predicate $\mathtt{p}^{\mathtt{bf}}$ is eventually generated by some other rules, then it must also be considered for adorning the above rule, since it may provide additional new information. □

Situation **S1** is easy to implement: In the function ***Generate*** we add a check whether the creation of the modified rule is necessary. Let r : $\mathtt{p_1(t_1)\ v \ldots v\ p_n(t_n)}$ $:-$ $\mathtt{q_1(s_1), \ldots, q_m(s_m)}$. be a disjunctive rule, and $\mathtt{p}_{\mathtt{i}}^{\alpha}$ be an adorned predicate that has already been used for generating the modified rule r_m. Then, any other adorned predicate $\mathtt{p}_{\mathtt{j}}^{\alpha'}$ such that (i) $\mathtt{p_j}$ has the same arguments of $\mathtt{p_i}$ and (ii) each argument of $\mathtt{p_i}$ has the same adornment in α and α', will generate for r a modified rule r'_m with $r'_m \sqsubseteq r_m$.

This check can be implemented by storing for each adorned predicate the set of rules it has already adorned, and it can be proven to be sound and complete.

Situation **S2** requires more effort. It implies that an adorned predicate should not always be applied to the whole program. To achieve this, we associate a *target* to each adorned predicate. The first time a predicate $\mathtt{p}^{\alpha}$ is pushed on the stack, it is marked for being used for adorning all the rules but the one that has generated it; this target is termed *allButSource*. Then, if at a certain point $\mathtt{p}^{\alpha}$ is generated again, then two situations may occur:

- if $\mathtt{p}^{\alpha}$ has been marked *allButSource* and already used for adorning the program (hence has been removed from the stack), then the new predicate will be inserted in the stack by marking it for adorning only the rule which was the source of the first generation of $\mathtt{p}^{\alpha}$ (that has not been adorned yet); such a target is called *onlySource*.
- if $\mathtt{p}^{\alpha}$ has been not yet used, then it is simply marked for being used for adorning all the program, giving rise to target *all*.

In the implementation we associate to each adorned predicate also the rule that generated it. Then, we modify step *6* in Fig. 1 as follows. A rule r considered for being adorned with a predicate $\mathtt{p}^{\alpha}$ is actually adorned if and only if (i) the target of $\mathtt{p}^{\alpha}$ is *onlySource* and r has generated the adornment $\mathtt{p}^{\alpha}$, or (ii) the target of $\mathtt{p}^{\alpha}$ is *allButSource* and r has not generated the adornment $\mathtt{p}^{\alpha}$, or (iii) the target of $\mathtt{p}^{\alpha}$ is *all*.

Due to space limits, we omit the correctness proofs of the above solutions.

4.2 Identifying Redundant Rules

Even though the above strategies may significantly reduce the redundancy within the rewritten program, we also exploit a (post-processing) technique for identifying those redundant rules whose generation could not be prevented. Specifically, we implemented a heuristic for rule subsumption checking that has been integrated into the core of the DLV system, and that can be invoked to identify redundancy in any type of program. The heuristic is based on the following observation.

Proposition 1. *Let r_1 and r_2 be two disjunctive rules. Then, r_1 subsumes r_2 if and only if there exists an ordering of all the atoms in $H(r_1) \cup B(r_1)$ of the form $l_1, \ldots, l_m$ and a sequence of substitutions $\vartheta_1, \ldots, \vartheta_m$, such that for each $l_i \in B(r_1)$ (resp. $l_i \in H(r_1)$), there exists $l'_i \in B(r_2)$ (resp. $l'_i \in H(r_2)$) with $(\vartheta_1 \cup \ldots \cup \vartheta_{i-1})(l_i) = \{l'_i\}$.*

Roughly, we try to construct the sequence $l_1, \ldots, l_m$ of the above proposition and the associated substitutions $\vartheta_1, \ldots, \vartheta_m$ in an incremental way. At each step i, we choose an atom l_i in r_1 which has not yet been processed such that there exists a candidate l'_i in r_2 for being subsumed. Moreover, if many atoms in r_i satisfy the above condition, we greedily select the one which subsumes the maximum number of atoms, and among these we prefer those with the maximum number of distinct variables not yet matched.

5 Experimental results

5.1 Compared Methods, Benchmark Problems and Data

In order to evaluate the impact of the proposed methods, we compare DMS and ODMS both with the traditional DLV evaluation without *Magic-Sets* and with the APM method proposed in [13]. For the comparison, we consider the following benchmark problems that have been already used to assess APM in [13] (see therein for more details):

– *Simple Path:* Given a directed graph G and two nodes a and b, does there exist a unique path connecting a to b in G? The graph is the same as the one reported in [13], and the instances are generated by varying the number of nodes.
– *Ancestor:* Given a genealogy graph storing information of relationship (father/brother) among people and given two persons p_1 and p_2, is p_1 an ancestor of p_2? The structure of the "genealogy" graph is the same as the one presented in [13], and the instances are generated by varying the number of nodes, i.e., the number of persons, in the graph.
– *Strategic Companies:* The problem has been formalized in Example 1. The instances are generated according to the ideas presented in [13], by grouping the companies in suitable clusters. Let G be the cluster such that c is in G. Then, the instances are generated with $|G| = 250$, while the number of companies outside G is varied.

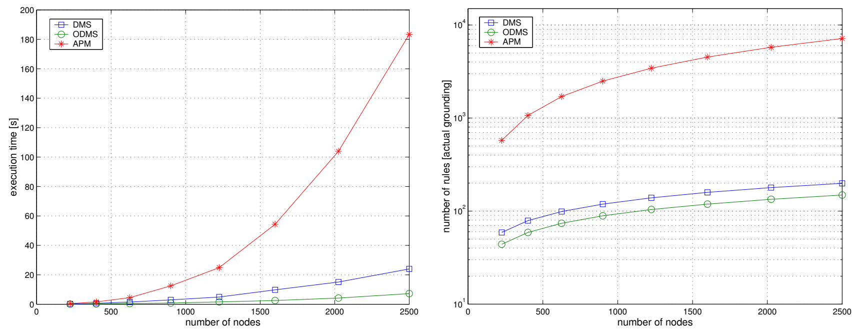

Fig. 2. *Simple Path:* Execution time *(Left)* and Number of rules instances *(Right)*

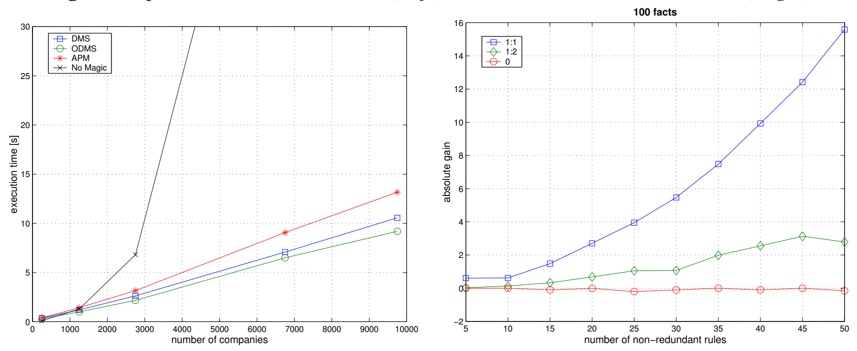

Fig. 3. Timing in *Strategic Companies (Left)* and Impact of subsumption checking *(Right)*

5.2 Results and Discussion

The experiments have been performed on Pentium III machines running GNU/Linux. The DLV prototype used was compiled with GCC 2.95. For every instance, we have allowed a maximum running time of 1800 seconds (half an hour) and a maximum memory usage of 256MB. On all problems, DMS outperforms APM, even without considering the time for the rewriting needed in [13], which is also not reported in the figures.

The results for *Simple Path* are reported in Figure 2. The diagram on the left shows that DMS scales much better than APM on this problem and that ODMS provides additional speed-up. The main reason can be understood by looking at the right diagram, in which the numbers of ground rules are reported: The overhead of the auxiliary rules for APM is evident, it generates about 25 times more rules than DMS. We did not include pure DLV (No Magic) in the diagrams, as it is dramatically slower; e.g. the instance with 255 nodes takes about 195 seconds. Finally, the experimental results for *Ancestor* are very similar to the ones for *Simple Path*.

On the left of Figure 3 we report the results for *Strategic Companies*. The advantages of the Magic-Set method (in both implementations) are evident. Anyhow, we can see that APM performs and scales worse than DMS, while ODMS provides even better performance *and* scaling.

Finally, on the right of Figure 3, we report a more detailed analysis on the impact of subsumption checking. In particular, we want to check whether the application of subsumption checking is computationally heavy (how much performance gets worse in a bad case where no redundant rule is identified). To this end, we test a program with two types of rules, specifically r_i : `pi(X) v qi(X) :- a(X), b(X).` and r'_i : `pi(X) v qi(X) :- a(X).`, where `a` and `b` are *EDB* predicates, and where each rule of the form r_i is subsumed by a rule of the form r'_i.

We fix a database of 100 facts, and we report the gain, calculated as the difference between the execution times of DLV without and with subsumption checking by varying the number of the rules of type r_i.

We report three distinct runs: (0) when no redundancy is added, i.e., when there is no occurrence of rules of type r'_i, (1:1) when one redundant rule is added in correspondence to one non-redundant one, i.e., when a rule of the form r'_i occurs for each rule r_i, and (1:2) when for each pair of rules r_i and r_{i+1} we insert only one occurrence of a redundant rule, namely either r'_i or r'_{i+1}. Importantly, the experiments show that the implementation is lightweight as in case (0) it does not deteriorate the performance of DLV. Moreover, it is effective as it leads to a gain up to 3% for case (1:2) and up to 16% for case (1:1).

6 Related Work and Conclusions

The Magic-Set method is among the most well-known techniques for the optimization of positive recursive Datalog programs due to its efficiency and its generality, even though other focused methods, e.g. the supplementary magic set and other special techniques for linear and chain queries have been proposed as well (see, e.g., [15, 7, 16]).

After seminal papers [8, 9], the viability of the approach was demonstrated e.g. in [17, 18]. Later on, extensions and refinements have been proposed, addressing e.g. query constraints in [10], the well-founded semantics in [11], or integration into cost-based query optimization in [12]. The research on variations of the Magic-Set method is still going on. For instance, in [19] a technique for the class of *soft-stratifiable* programs is given, and in [13] an elaborated technique for disjunctive programs is described.

In this paper, we have elaborated on the issues addressed in [13]. Our approach is similar in spirit to APM, but differs in several respects:

- DMS avoids the use of auxiliary predicates needed for APM, yielding a significant computational benefit.
- DMS is a flexible framework for enhancements and optimizations, as it proceeds in a localized fashion by analyzing one rule at time, while APM processes the whole program at time.
- ODMS extends DMS by employing effective methods for avoiding the generation of and for identifying still left-over redundant rules.
- ODMS has been integrated into the DLV system [4], profitably exploiting the DLV internal datastructures and the ability of controlling the grounding module.
- We could experimentally show that our ODMS implementation outperforms APM on benchmarks taken from the literature.

It has been noted (e.g. in [11]) that in the non-disjunctive case, memoing techniques lead to similar computations as evaluations after Magic-Set transformations. Also in the disjunctive case such techniques have been proposed, e.g. Hyper Tableaux [20], for which a similar relationship might hold. However, we leave this issue for future research, and follow [11] in noting that an advantage of Magic-Sets over such methods is that they can be more easily combined with other database optimization techniques.

Concerning future work, our objective is to extend the Magic-Set method to the case of disjunctive programs with constraints and unstratified negation, such that it can be fruitfully applied on arbitrary DLV programs. We believe that the framework developed in this paper is general enough to be extended to these more involved cases.

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9** (1991) 365–385
2. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS **22** (1997) 364–418
3. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative Problem-Solving Using the DLV System. In Minker, J., ed.: Logic-Based Artificial Intelligence. Kluwer (2000) 79–103
4. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL (2004) To appear. Available via `http://www.arxiv.org/ps/cs.AI/0211004`.
5. Janhunen, T., Niemelä, I., Simons, P., You, J.H.: Partiality and Disjunctions in Stable Model Semantics. In: KR 2000, April 12-15, Morgan Kaufmann (2000) 411–419
6. Arieli, O., Denecker, M., Van Nuffelen, B., Bruynooghe, M.: Database repair by signed formulae. In: FoIKS 2004. LNCS 2942., Springer (2004) 14–30
7. Ullman, J.D.: Principles of Database and Knowledge Base Systems. Computer Science Press (1989)
8. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic Sets and Other Strange Ways to Implement Logic Programs. In: PODS'86. (1986) 1–16
9. Beeri, C., Ramakrishnan, R.: On the power of magic. JLP **10** (1991) 255–259
10. Stuckey, P.J., Sudarshan, S.: Compiling query constraints. In: PODS'94, ACM Press (1994) 56–67
11. Kemp, D.B., Srivastava, D., Stuckey, P.J.: Bottom-up evaluation and query optimization of well-founded models. Theoretical Computer Science **146** (1995) 145–184
12. Seshadri, P., Hellerstein, J.M., Pirahesh, H., Leung, T.Y.C., Ramakrishnan, R., Srivastava, D., Stuckey, P.J., Sudarshan, S.: Cost-based optimization for magic: Algebra and implementation. In: SIGMOD Conference 1996, ACM Press (1996) 435–446
13. Greco, S.: Binding Propagation Techniques for the Optimization of Bound Disjunctive Queries. IEEE TKDE **15** (2003) 368–385
14. Cadoli, M., Eiter, T., Gottlob, G.: Default Logic as a Query Language. IEEE TKDE **9** (1997) 448–463
15. Greco, S., Saccà, D., Zaniolo, C.: The PushDown Method to Optimize Chain Logic Programs (Extended Abstract). In: ICALP'95. (1995) 523–534
16. Ramakrishnan, R., Sagiv, Y., Ullman, J.D., Vardi, M.Y.: Logical Query Optimization by Proof-Tree Transformation. JCSS **47** (1993) 222–248
17. Gupta, A., Mumick, I.S.: Magic-sets Transformation in Nonrecursive Systems. In: PODS'92. (1992) 354–367
18. Mumick, I.S., Finkelstein, S.J., Pirahesh, H., Ramakrishnan, R.: Magic is relevant. In: SIGMOD Conference 1990. (1990) 247–258
19. Behrend, A.: Soft stratification for magic set based query evaluation in deductive databases. In: PODS 2003, ACM Press (2003) 102–110
20. Baumgartner, P., Furbach, U., Niemelä, I.: Hyper Tableaux. In: JELIA'96. LNCS 1126, Springer (1996) 1–17